Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators

Citation for published version:
Schlaak, C, Juang, T-H & Dubach, C 2022, 'Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators', *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 2, 16. https://doi.org/10.1145/3501768

Digital Object Identifier (DOI):
10.1145/3501768

Published In:
ACM Transactions on Architecture and Code Optimization

Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators

CHRISTOF SCHLAAK, University of Edinburgh, United Kingdom
TZUNG-HAN JUANG and CHRISTOPHE DUBACH, McGill University, Canada

Specialized accelerators deliver orders of a magnitude of higher performance than general-purpose processors. The ever-changing nature of modern workloads is pushing the adoption of FPGAs (Field Programmable Gate Arrays) as the substrate of choice. However, FPGAs are hard to program directly using HDLs (Hardware Description Languages). Even modern high-level HDLs, e.g., Spatial and Chisel, still require hardware expertise.

This article adopts functional programming concepts to provide a hardware-agnostic higher-level programming abstraction. During synthesis, these abstractions are mechanically lowered into a functional IR (Intermediate Representation) that defines a specific hardware design point. This novel IR expresses different forms of parallelism and standard memory features such as asynchronous off-chip memories or synchronous on-chip buffers. Exposing such features at the IR level is essential for achieving high performance.

The viability of this approach is demonstrated on two stencil computations and by exploring the optimization space of matrix-matrix multiplication. Starting from a high-level representation for these algorithms, our compiler produces low-level VHDL (VHSIC Hardware Description Language) code automatically. Several design points are evaluated on an Intel Arria 10 FPGA, demonstrating the ability of the IR to exploit different hardware features. This article also shows that the designs produced are competitive with highly tuned OpenCL implementations and outperform hardware-agnostic OpenCL code.

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

Additional Key Words and Phrases: High-level synthesis, accelerators, functional IR, compilers

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

New Paper, Not an Extension of a Conference Paper.
This work was supported by Microsoft Research through its PhD Scholarship Programme. This work was also supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.
Authors' addresses: C. Schlaak, University of Edinburgh, United Kingdom; email: christof.schlaak@ed.ac.uk; T.-H. Juang and C. Dubach, McGill University, Canada; emails: tzung-han.juang@mail.mcgill.ca, christophe.dubach@mcgill.ca.

1544-3566/2022/02-ART16
https://doi.org/10.1145/3501768

Corrected Version of Record. V.1.1. Published March 3, 2022.

1 INTRODUCTION

Designing new accelerators is a manual, time consuming, and error-prone process. Current HDLs are not suitable for a rapid-development cycle and there is a lack of high-level languages and abstractions for efficient hardware design. Languages such as Spatial [18], Chisel [2] or OpenCL reduce the amount of boiler plate code required, but remain fairly low-level. Many hardware concepts are still transpiring through the high-level abstractions offered by these languages. To fully automate accelerator design, we need programming abstractions that hide all hardware details.

Recent years have seen a push towards functional approaches for high-performance computing. Delite [36], Lift [34] and Futhark [12] have demonstrated that high-level abstractions and high-performance can go hand in hand. More recently, Lift-hls [19] and Aetherling [10] have demonstrated that the functional approach is viable for producing accelerators.

Lift-hls and Aetherling use a multi-level IR. By rewriting, the IR is lowered into a form suitable for the generation of an efficient accelerator. The highest IR level exposes algorithmic concepts while the lower levels expose hardware paradigms such as pipeline or spatial parallelism. These multiple levels enable a clear separation of concerns between algorithmic and hardware transformations when exploring the design space. Furthermore, this process is transparent from the programmer and leads to an elegant compiler design where transformations are expressed as simple rewrites.

Lift-hls and Aetherling have pioneered the use of functional IRs and rewrite rules for generating hardware. These approaches can express different scheduling strategies and parallelism in the IR. However, they lack explicit support for memory operations which severally restricts their use on real hardware. Aetherling, for instance, has only produced results in simulation with an overly simplified memory model. Maximizing performance requires the use of on-chip and off-chip memories, as well as the use of asynchronous transfers. Given that such concepts are absent from these IRs, their associated compilers are unable to use these capabilities.

This article presents Shir, a multi-level IR inspired by Lift-hls and Aetherling. Uniquely, Shir represents memory concepts such as asynchronous data transfers to off-chip memories explicitly in the IR. Through a formally defined lowering process, the Shir compiler turns the high-level IR into a low-level form where memory operations are explicit. The low-level IR is then turned straightforwardly into VHDL code that is synthesizable on real hardware. The generated designs exploit hardware parallelism, off-chip memories, as well as on-chip block-ram to maximize performance.

Shir is meant to be used as an intermediate language and its code could be generated from any front-end library or framework. TensorFlow would be a good example given that tensor operations can be easily mapped onto the Shir primitives.

The validity of the presented approach is demonstrated using stencil computations and matrix-matrix-multiplication (MxM). Although being easy to understand, MxM exhibits many interesting optimization choices that exercise all the features presented in this article. Starting from a simple hardware-agnostic description, the program representation is automatically lowered, synthesized, and run on an Intel Arria 10 FPGA. Our results shows that the Shir compiler produces correct and efficient hardware that is competitive with OpenCL implementations.

This article makes the following contributions:

- it presents an explicit encoding of memory operations in the IR to enable their efficient use;
- it formalizes the lowering process, turning high-level programs into a form suitable for hardware synthesis;
- it demonstrates this approach on a real FPGA and compares the performance to OpenCL.

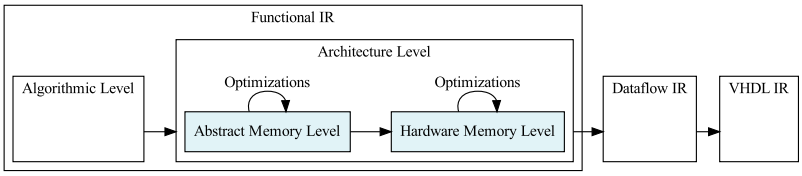


Fig. 1. Compilation flow from Shir algorithmic code to VHDL code through different IRs.

```
Reduce(+, 0, Input(IntT(32), N))
```

(a) Algorithmic Level.

```
1 Let mem = MemAlloc(IntT(32), N) in
2   ReduceStm(+, 0,
3     MapStm(
4       λ addr => Read(mem, addr),
5       Counter(N-1))) // offsets
```

(b) Abstract Memory Level.

```
1 Let rdCtrl = HostRamReadCtrl(...) in
2   ReduceStm(+, 0,
3     JoinStm(MapStm(
4       λ p => VecToStr(SplitVec(p,32)),
5       ReadAsync(
6         rdCtrl,
7         0x00000000, // baseAddr
8         Counter(N/16 -1) // offsets
9   ))))
```

(c) Hardware Memory Level.

Fig. 2. Shir representation at three different levels for the program "sum of an array".

The rest of this article is organized as follows: Section 2 presents an overview. Section 3 introduces the core Shir language and the algorithmic IR level. Section 4 presents the lower levels IRs. Section 5 presents the lowering and compilation process with rewriting steps for optimization. Section 6 evaluates the approach on a real FPGA. Section 7 presents related work while Section 8 concludes the article.

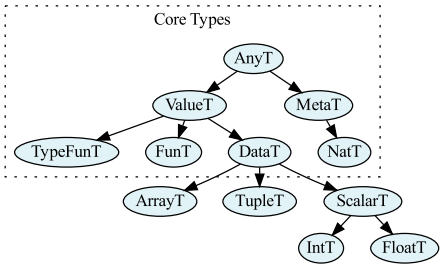
2 OVERVIEW AND EXAMPLE

Figure 1 presents an overview of our approach. The Shir compiler transforms high-level code into multiple levels of IRs. Each level exposes more hardware details and allows optimizations via a rewrite mechanism. The functional IR is first turned into a data-flow IR and then into VHDL code for FPGA synthesis using vendor tools. Before diving into the IR's levels and the lowering process, a brief example illustrates how a simple program "sum of an array" is represented at the three levels.

At the *Algorithmic Level*, programs are represented in a hardware-agnostic way using common built-ins such as `Map`, `Reduce` or `Zip`. The sum of an array of N 32-bit integers is expressed as in Figure 2(a), where `Input` simply represents an input which will be allocated in memory.

The *Abstract Memory Level* exposes memory operations. The compiler turns the high-level algorithmic code into the expression seen in Figure 2(b). Memory for the program's input is allocated in line 1. A counter generates a stream of addresses from 0 to $N-1$ in line 5. `MapStm` calls, for each address, the `Read` memory function. The `MapStm` outputs a stream of data that is reduced to produce the sum. For the sake of simplicity, we omitted details about the program's output, but a similar process takes place where explicit write operations are used. While no specific hardware details are yet exposed in the IR, memory operations are now explicit.

The *Hardware Memory Level* is the last IR level and exposes all the hardware features directly, as shown in Figure 2(c), allowing an easy translation to VHDL code. During lowering, a `HostRamReadCtrl` function is inserted in place of the abstract `MemAlloc` concept. This read controller is directly connected to the DMA (Direct Memory Access) engine on the FPGA. The `ReadAsync` on line 5 takes the read controller in and two other inputs: the base address $0x00000000$



(a) Core types (dashed box) and algorithmic types.

```
Expr   ::= Expr1 | (Expr)
Expr1  ::= Param | Lambda | FunCall
       | LetExpr | Primitive | (Expr1)
       | TypeLambda | TypeFunCall
Param      ::= IDENTIFIER
Lambda     ::= λ Param => Expr
FunCall    ::= Expr (Expr)
LetExpr    ::= Let Param = Expr in Expr
Primitive  ::= IDENTIFIER <Types> (Exprs)
           | IDENTIFIER (Exprs)
           | IDENTIFIER <Types> | LITERAL
Exprs        ::= Expr | Expr,Exprs
TypeLambda   ::= Λ TypeVarT^Type => Expr
TypeFunCall  ::= Expr <Types>
```

(b) Grammar for core expressions.

Fig. 3. Shir core design.

of the input data allocation in memory (determined by the compiler); and a stream of offsets produced by the counter. There are two main differences here compared to the previous level.

First, ReadAsync asynchronously requests data from the hardware DMA controller, which responds at a later time with the desired data. Since the controller accepts multiple pending requests, an asynchronous read takes a stream of offsets as an input, which are requested concurrently.

Second, the transferred data is on a cache line size granularity (512 bits in our hardware). For N 32-bit values we need to read only $N/\frac{512}{32}$ cache lines, which are returned by ReadAsync in a stream. In line 4, this data is reshaped with a SplitVec into pieces of 32 bits before reducing. Finally, a JoinStm is used to merge the two outer dimensions of the stream and produce a single stream of 32-bit integers that are fed into the reduction.

3 CORE SHIR IR AND ALGORITHMIC LEVEL

The core of the Shir language and the high-level algorithmic primitives are inspired by Lift [34], a functional data-parallel language based on typed lambda calculus. The major difference is the support of subtyping, in the style of System F with Subtyping ($F_<$) [7]. This enables an intuitive-type hierarchy and building up new constructions within the language, while the unified type system confirms that the implementation's types are correct.

3.1 Core Types

All types in Shir are implemented as classes in Scala. Their constructors are referred to as *type constructors*. We developed a type checker for Shir, that uses unification and a constraint solver where sub-typing relationship are expressed as constraints. The object oriented manner of these implementations facilitates extensions to new types, without requiring any modification of the core type system or the type checker. When dealing with sub-typing relationship, the type constructors arguments are all *covariant* except for function types, where it is *contravariant* in its input type.

Figure 3(a) shows the Shir core types hierarchy. The root of the hierarchy is AnyT. From there, the types are separated into value types and meta types.

Data and Function Types are both value types; i.e., types that are taken as input to, or returned by, functions. The data types are presented in Section 4 since they are not part of the core language. The function type constructor is $\texttt{FunT}(inT^{ValueT}, outT^{ValueT})$ where inT is the function input type and $outT$ its output type. The superscript denotes a sub-typing relationship, i.e., inT must be a subtype of $ValueT$. In the rest of this article, the function type is represented as: $inT \rightarrow outT$.

Meta Types represent meta-information that is embedded inside other types and treated as types. In the core language, the only *concrete* meta type is `NatT`, the type representing natural numbers. As we will see in Section 4, this type is used to specify the length of arrays (or other collections).

Type-Function Types are used to implement *generic* (or *templated*) functions. Its type constructor takes a type variable and a return type where the type variable can appear. The following notation $TV^T \mapsto U$ declares a type variable TV, subtype of T, that might appear in type U.

3.2 Core IR

Type. Figure 3(b) shows the grammar for the core expressions of the Shir language. As can be seen, each expression has a type. When building expressions, their types do not have to be explicitly provided, a type variable (`TypeVarT`) can be used instead of a concrete type. Again, the superscript notation can be used to indicate that a type variable must be a subtype of another type. During type checking, they will eventually be replaced by a concrete type (if the program is correctly typed).

Param, Lambda, FunCall, Let. These expressions are traditionally found in any implementation of lambda calculus. `Lambda` is an anonymous function that has a `Param` as argument and an `Expr` as body where `Param` can appear. `FunCall` is simply a function call to a lambda. `Let` binds an expression to a `Param` that is shared among all of the `Param`'s appearances in the body.

Primitive. The `Primitive` IR node represents built-in functions call, (e.g., `Add`) or built-in constants (e.g., `2.0f`). Built-in functions can have any number of `Expr` as arguments. For instance, the following code shows a function f used with two arguments, x and y: $\lambda x => \lambda y => f(x, y)$.

`TypeLambda` is used to create *generic* expressions whose type might depend on a type variable. `TypeFunCall` is used when *instantiating* a generic `TypeLambda`. During type-checking, the effect of a `TypeFunCall` is to substitute the type variable by the call argument type.

3.3 Algorithmic Extensions

Types. The high-level algorithmic types are shown in Figure 3(a), outside of the dashed box. The array type stores both the element type T and the array length N, as seen in its type constructor $\texttt{ArrayT}(T^{DataT}, N^{NatT})$. From now on, we will use the following shortcut syntax to represent array types: $[T]_N$. For tuple types (`TupleT`), the short form is $(type1, type2, \ldots)$. Common half, single, and double precision float types are supported in Shir. In contrast to Lift, the integer type stores the number of bits used to represent the integer value. The allowed bit-widths are *not* limited to powers of two, the Shir compiler supports arbitrary precision. The integer-type constructor reflects this flexibility: $\texttt{IntT}(numBits^{NatT})$. This enables efficient area usage especially on FPGAs, because they are re-configurable and not constrained by a specific operator bit-width.

Primitives. The algorithmic IR primitives, listed in Figure 4, are common high-level functional primitives such as `Map`, `Reduce`, `Slide`, `Split`, `Join`, and `Zip`. The `Constant` primitive takes a static `NatT` that specifies its value. The `Input` primitive represents the input data coming from memory. It also supports multidimensional data, which is omitted in Figure 4 for simplicity.

3.4 Summary

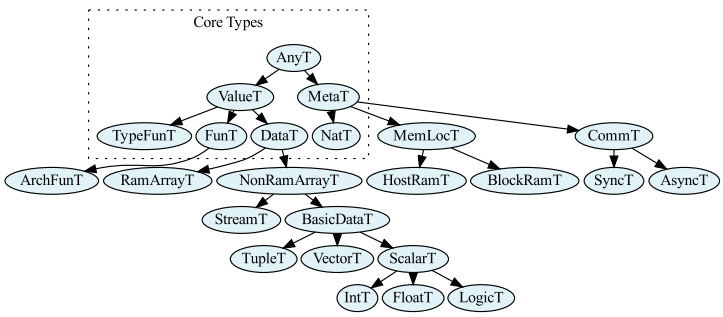
The core Shir language is standard, similar to typed lambda calculus, augmented with support for generics and sub-typing. The high-level algorithmic primitives currently supported by Shir are introduced. In the next section, this language is extended to enable hardware synthesis.

4 ARCHITECTURE LEVEL

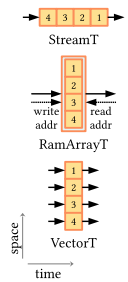
On its way to hardware, the algorithmic representation is lowered through different IR levels. The Architecture Level remains functional, facilitating the application of transformations that

$$Add/Mul : T^{ScalarT} \mapsto (T, T) \to T \tag{1}$$
$$Id : T^{ScalarT} \mapsto T \to T \tag{2}$$
$$Constant : T^{IntT} \mapsto N^{NatT} \mapsto T \tag{3}$$
$$Tuple : T^{ScalarT} \mapsto U^{ScalarT} \mapsto T \to U \to (T, U) \tag{4}$$
$$Map : T^{DataT} \mapsto U^{DataT} \mapsto N^{NatT} \mapsto (T \to U) \to [T]_N \to [U]_N \tag{5}$$
$$Reduce : T^{DataT} \mapsto U^{DataT} \mapsto N^{NatT} \mapsto (U \to (T \to U)) \to U \to [T]_N \to U \tag{6}$$
$$Slide : T^{DataT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto [T]_N \to [[T]_M]_{N-M+1} \tag{7}$$
$$Zip : T^{DataT} \mapsto U^{DataT} \mapsto N^{NatT} \mapsto [T]_N \to [U]_N \to [(T, U)]_N \tag{8}$$
$$Split : T^{DataT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto [T]_N \to [[T]_M]_{N/M} \tag{9}$$
$$Join : T^{DataT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto [[T]_M]_N \to [T]_{N*M} \tag{10}$$
$$Input : T^{DataT} \mapsto N^{NatT} \mapsto [T]_N \tag{11}$$

Fig. 4. Algorithmic primitives and their types. These definitions are read from left to right: First, the type variables (T and U) are specified for each generic primitive. They are concatenated with the $\mapsto$ symbol and have a superscripted super type, e.g., $T^{ScalarT}$ means that T is a subtype of $ScalarT$. After that, the input value types are listed, separated by a $\to$ symbol. The rightmost value type is the return type of the primitive.



(a) Type hierarchy.



(b) Space-time schematic representation of collection types.

Fig. 5. Types at the architecture level.

crucially affect the resulting hardware design and performance without needing to operate on a low HDL-like level. This level introduces a slight notion of timing, as seen in the collection types in Figure 5(a). Figure 5(b) shows the core language types, this time extended with all the architecture types.

4.1 Scalar and Tuple

Tuples and scalar types are similar to the Algorithmic Level, with the addition of `LogicT`, which represents a single bit. The `TupleT` type is slightly different and only accepts elements of type $T^{BasicDataT}$. The `BasicDataT` ensures data is always available in a single clock cycle. This contrasts with stream and ramarray, which require multiple cycles to read their entire content. Primitives that operate on scalars (`Add`, `Mul`, `Id`, `Constant` and `Tuple`) are similar to the ones in Figure 4.

4.2 Stream

Type. The type constructor `StreamT`$(T^{NonRamArrayT}, N^{NatT})$ (short form notation $\overset{STM}{}[T]_N$) represents a sequence of N elements of type T, where T cannot be a ramarray. This data type can be

$$MapStm : T^{NonRamArrayT} \mapsto U^{NonRamArrayT} \mapsto N^{NatT} \mapsto (T \to U) \to \underset{STM}{}[T]_N \to \underset{STM}{}[U]_N \tag{12}$$

$$ReduceStm : T^{BasicDataT} \mapsto U^{DataT} \mapsto N^{NatT} \mapsto (U \to (T \to U)) \to \underset{STM}{}[T]_N \to U \to U \tag{13}$$

$$SlideStm : T^{NonRamArrayT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto \underset{STM}{}[T]_N \to \underset{STM}{}[\underset{VEC}{}[T]_M]_{N-M+1} \tag{14}$$

$$ZipStm : T^{NonRamArrayT} \mapsto U^{NonRamArrayT} \mapsto N^{NatT} \mapsto \underset{STM}{}[T]_N \to \underset{STM}{}[U]_N \to \underset{STM}{}[(T,U)]_N \tag{15}$$

$$SplitStm : T^{NonRamArrayT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto \underset{STM}{}[T]_N \to \underset{STM}{}[\underset{STM}{}[T]_M]_{N/M} \tag{16}$$

$$JoinStm : T^{NonRamArrayT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto \underset{STM}{}[\underset{STM}{}[T]_M]_N \to \underset{STM}{}[T]_{N*M} \tag{17}$$

$$StmToVec : T^{BasicDataT} \mapsto N^{NatT} \mapsto \underset{STM}{}[T]_N \to \underset{VEC}{}[T]_N \tag{18}$$

$$Repeat : T^{NonRamArrayT} \mapsto N^{NatT} \mapsto T \to \underset{STM}{}[T]_N \tag{19}$$

$$Counter : T^{IntT} \mapsto N^{NatT} \mapsto \underset{STM}{}[T]_N \tag{20}$$

Fig. 6. Architecture Level primitives operating on streams.

used to model flow of data in general and pipelining. A new element is produced at each clock cycle unless the stream is stalled. Once a piece of data has been consumed, it cannot be recalled.

Primitives. The high-level algorithmic primitives are refined for streams. `MapStm`, `ReduceStm`, `ZipStm`, `SplitStm`, and `JoinStm` all operate on streams, as listed in Figure 6. These new primitives provide more details about how their functionality is implemented. `MapStm`, for example, instantiates the given function and feeds the individual elements of the stream into it, one at a time. Depending on the function, this generates a pipeline in the hardware. `ReduceStm` will generate an accumulator, which accumulates all incoming elements of the stream using a given function (e.g., `add`). `SlideStm` is implemented in hardware by feeding the input stream into a shift-register and sending out all the register's contents as a vector, whenever a new input value arrives. `Counter` emits a stream of incrementing integers, which is useful for generating memory addresses as seen in the example of Section 2. `StmToVec` converts a stream of data into a vector, using a shift-register in hardware.

4.3 Vector

Type. The type constructor `VectorT`($T^{BasicDataT}, N^{NatT}$) (short-form notation $\underset{VEC}{}[T]_N$) creates a vector type with support for parallel access to all its N elements of type T in a single clock cycle. This type is similar to the tuple type, with the key difference that the vector's elements must all be of the same type. A vector is ideal for parallelization, a common strategy to improve performance.

Primitives. The common high-level primitives also have a counterpart for vectors: `MapVec`, `ZipVec`, `SplitVec`, and `JoinVec`, as listed in Figure 7. Additionally, the `VecToStm` primitive converts a vector into a stream of data, by emitting the vector's elements one after the other. `VecToTuple` converts a vector into a tuple to feed the data into tuple-based operations like `Add`. `MapVec` exploits spatial parallelism by instantiating the given function once for each vector element. `SlideVec` simply generates wires in hardware to rearrange the data to create a vector of vectors.

There is no explicit primitive for `ReduceVec`. Instead, the same functionality is achieved with a smart constructor of the same name, which automatically generates an efficient reduction tree from a combination of `MapVec` and `VecToTuple`. The tree performs $N - 1$ operations in $\log_2 N$ steps.

$$MapVec : T^{BasicDataT} \mapsto U^{BasicDataT} \mapsto N^{NatT} \mapsto (T \to U) \to \overset{VEC}{}[T]_N \to \overset{VEC}{}[U]_N \tag{21}$$

$$SlideVec : T^{BasicDataT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto \overset{VEC}{}[T]_N \to \overset{VEC}{}[\overset{VEC}{}[T]_M]_{N-M+1} \tag{22}$$

$$ZipVec : T^{BasicDataT} \mapsto U^{BasicDataT} \mapsto N^{NatT} \mapsto (\overset{VEC}{}[T]_N, \overset{VEC}{}[U]_N) \to \overset{VEC}{}[(T,U)]_N \tag{23}$$

$$SplitVec : T^{BasicDataT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto \overset{VEC}{}[T]_N \to \overset{VEC}{}[\overset{VEC}{}[T]_M]_{N/M} \tag{24}$$

$$JoinVec : T^{BasicDataT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto \overset{VEC}{}[\overset{VEC}{}[T]_M]_N \to \overset{VEC}{}[T]_{N*M} \tag{25}$$

$$VecToStm : T^{BasicDataT} \mapsto N^{NatT} \mapsto \overset{VEC}{}[T]_N \to \overset{STM}{}[T]_N \tag{26}$$

$$VecToTuple : T^{ScalarT} \mapsto N^{NatT} \mapsto \overset{VEC}{}[T]_N \to (T, ..., T) \tag{27}$$

Fig. 7. Architecture Level primitives operating on vectors.

4.4 Memory

There are two different ways to represent memory in Shir, as explained below. First, the Abstract Memory Level, which provides an intuitive and user-friendly interface to memory. Second, the Hardware Memory Level, which exposes how memory is implemented in hardware.

4.4.1 Abstract Memory Level.

Type. The type constructor `RamArrayT`($T^{BasicDataT}, N^{NatT}, ML^{MemLocT}$) (short-form notation $\overset{RAM}{}[T]_N^{ML}$) represents a ramarray of N elements of type T, which can only be accessed one at a time, however, random-order access is possible. Shir features a flat memory model and forbids nesting of `RamArrayT`. Multi-dimensional data is stored in memory by joining (i.e., flattening) the data before writing to memory and then splitting it after reading. The ramarray's `MemLocT` indicates whether host RAM (Random Access Memory) `HostRamT` or on-chip block RAM `BlockRamT` is used to hold the data. This type also contains an identifier to distinguish between multiple block RAM instances.

Primitives. Figure 8 shows the Abstract Memory Level primitives. `MemAlloc` allocates memory space for N elements of type T and returns a ramarray. A `MemLocT` is required, to identify the memory used. Given a ramarray, the `Read` primitive returns an element at the specified address of type A. The bit-width b of the address depends on the ramarray's length, with $b = \lceil \log_2 N \rceil$.

The `Write` primitive enables the reverse operation: writing a new element at a given position in a ramarray. Similar to the concept of *monads* in functional programming, this primitive returns the updated ramarray. However, the later generated hardware implementation updates the data in-place. Thus, an interesting pattern of code is enabled: By wrapping `Write` into a `ReduceStm`, an entire stream can be *buffered* in a ramarray. In practice, this is beneficial when the ramarray resides on-chip and the stream is read multiple times. The following code illustrates this important use-case, where the stream `inputData` is stored in a ramarray, and then read again:

```
Let allocatedBuffer = MemAlloc(N, dataType, memLocType) in
  Let updatedBuffer =
    ReduceStm(
      λ mem => λ data => Write(mem, data), allocatedBuffer,
      ZipStm(inputData, Counter(N-1))) in
    MapStm(λ addr => Read(updatedBuffer, addr), Counter(N-1))
```

4.4.2 Hardware Memory Level.

Types. At the Hardware Memory Level, the ramarray types disappear, because memory is represented as functions. This corresponds to the hardware paradigm that interaction with memory

$$MemAlloc : N^{NatT} \mapsto T^{BasicDataT} \mapsto ML^{MemLocT} \mapsto \underset{RAM}{}[T]_N^{ML} \tag{28}$$

$$Read : N^{NatT} \mapsto T^{BasicDataT} \mapsto ML^{MemLocT} \mapsto A^{IntT} \mapsto \underset{RAM}{}[T]_N^{ML} \to A \to T \tag{29}$$

$$Write : N^{NatT} \mapsto T^{BasicDataT} \mapsto ML^{MemLocT} \mapsto A^{IntT} \mapsto \underset{RAM}{}[T]_N^{ML} \to (T, A) \to \underset{RAM}{}[T]_N^{ML} \tag{30}$$

Fig. 8. Primitives to express abstract memory. Types named A denote integer based types for addresses, where the bit-width depends on the address space of the memory.

$$BlockRam : T^{BasicDataT} \mapsto A^{IntT} \mapsto (T, A, Bool) \xrightarrow{s} T \tag{31}$$

$$ReadSyncMemCtrl : T^{BasicDataT} \mapsto A^{IntT} \mapsto ((T, A, Bool) \xrightarrow{s} T) \xrightarrow{s} A \to T \tag{32}$$

$$WriteSyncMemCtrl : T^{BasicDataT} \mapsto A^{IntT} \mapsto ((T, A, Bool) \xrightarrow{s} T) \xrightarrow{s} (T, A) \xrightarrow{s} T \tag{33}$$

$$ReadSync : T^{BasicDataT} \mapsto A^{IntT} \mapsto (A \xrightarrow{s} T) \xrightarrow{s} A \xrightarrow{s} A \xrightarrow{s} T \tag{34}$$

$$WriteSync : T^{BasicDataT} \mapsto A^{IntT} \mapsto ((T, A) \xrightarrow{s} T) \xrightarrow{s} A \xrightarrow{s} (T, A) \xrightarrow{s} T \tag{35}$$

$$ReadHostMemCtrl : T^{BasicDataT} \mapsto A^{IntT} \mapsto R^{IntT} \mapsto (A, R) \xrightarrow{a} (T, R) \tag{36}$$

$$WriteHostMemCtrl : T^{BasicDataT} \mapsto A^{IntT} \mapsto R^{IntT} \mapsto (A, T, R) \xrightarrow{a} R \tag{37}$$

$$ReadAsync : T^{BasicDataT} \mapsto N^{NatT} \mapsto A^{IntT} \mapsto R^{IntT} \mapsto ((A, R) \xrightarrow{a} (T, R)) \xrightarrow{s} A \xrightarrow{s} \underset{STM}{}[A]_N \xrightarrow{s} \underset{STM}{}[T]_N \tag{38}$$

$$WriteAsync : T^{BasicDataT} \mapsto N^{NatT} \mapsto A^{IntT} \mapsto R^{IntT} \mapsto ((A, T, R) \xrightarrow{a} R) \xrightarrow{s} A \xrightarrow{s} \underset{STM}{}[(T, A)]_N \xrightarrow{s} A \tag{39}$$

Fig. 9. Memory-related primitives on the Hardware Memory Level. Types named A denote integer based types for addresses. Types R represent integer based types for request ids, which map the responses in asynchronous communication to the previously sent requests.

is performed through functional units. Depending on the type of memory involved, on-chip versus off-chip, the function is either synchronous (data is returned immediately) or asynchronous (requested data is returned later and not in order).

To capture this in the type system, the IR is expanded with a new function type. Its type constructor is `ArchFunT`($inT^{ValueT}, outT^{ValueT}, c^{CommT}$). In short-form notation, we use $inT \xrightarrow{s} outT$ for synchronous communication (where c is `SyncT`) and $inT \xrightarrow{a} outT$ f or asynchronous communication (c is `AsyncT`).

Primitives for Synchronous Block RAM. At this level, there is no abstract `MemAlloc` primitive but a more specialized `BlockRam` primitive, shown in Figure 9, to model synchronous on-chip RAM as functions. When called, this primitive returns a function that accepts a piece of data, an address and a write-enable flag which determines whether we want to read or write. This functional design is in line with its resulting hardware implementation.

Additionally, memory controllers are introduced, which extract only a certain capability of the memory interface. The `ReadSyncMemCtrl` primitive takes a `BlockRam` and provides an interface which only allows us to read data. The write enable flag is not accessible any longer from the outside and is fixed to false internally. The `WriteSyncMemCtrl` extracts a write interface, setting the write enable to true internally. The provided interface from `WriteSyncMemCtrl`, takes a value and an address and returns the same value after writing.

`ReadSync` uses the `ReadSyncMemCtrl` to read a value at a certain address from the block RAM memory, which is behind the memory controller. This primitive calculates the actual requested address by adding the given base address and given offset, both subtypes of integer. The `WriteSync` primitive works in a similar way, but instead takes a tuple of data and offset as input.

Primitives for Asynchronous Host RAM. The notion of host RAM is always present in the system and there is no need to allocate it. Therefore, there is no counterpart primitive to the `BlockRam`. The Hardware Memory Level provides memory controllers for both reading and writing asynchronous memory. During hardware generation, these controllers are connected to the DMA interface of the FPGA which talks to the host RAM via the PCIe (Peripheral Component Interconnect Express) bus.

Each asynchronous memory access must specify a unique request id (R in the type) to link a response to the requested read or write operation. `ReadHostMemCtrl` provides an asynchronous function, which takes a tuple of address and request id and returns a tuple of data and request id. The asynchronous function of the `WriteHostMemCtrl` takes a tuple of address, data, and request id and just returns the same request id, when the data has been written to the address in host RAM.

The host memory controller's signature for reading and writing depends on the FPGA DMA interface specifications. The Intel Arria 10 in our system transfers cache lines of 512 bits via DMA. Smaller pieces of data must be packed and possibly padded with zeros to fill an entire line. For this case, Shir offers a smart converter that creates a combination of split, join and other reshaping primitives to convert cache lines into any desired type of data with arbitrary precision and back. For 7-bit values each cache line contains $\lfloor 512/7 \rfloor = 73$ elements with $(512 \bmod 7) = 1$ padding bit.

The primitives for reading and writing are refined to enable concurrent memory requests. They take the corresponding memory controller and a base address as input. In addition, `ReadAsync` consumes a stream of addresses, while `WriteAsync` consumes a stream of tuples of address and data. Without waiting for the responses of previous requests (non-blocking), these asynchronous primitives send new requests to memory one by one, as soon as a new value from the input stream arrives. Thus, throughput between the host and the FPGA is maximized using multiple in-flight requests. The input stream's length of `ReadAsync` and `WriteAsync` determines the maximum number of possible parallel requests. Once the requested data is loaded, the host sends a response with the payload attached. `WriteAsync` returns the base address, when all the write operations to this memory region are completed and confirmed by the memory controller.

While the IR design looks complicated on the surface, it directly maps to hardware concepts and provides great flexibility in terms of expressible hardware designs.

4.5 Summary

In summary, these collection types are powerful tools to express different points in the hardware design space. A stream, for example, can easily be rewritten as a vector to parallelize computations, using additional logic elements on the FPGA. Furthermore, a stream can be replaced by a ramarray to buffer the stream's data for faster repeated access at the cost of on-chip RAM.

5 COMPILING A SHIR PROGRAM

A Shir program is expressed using high-level primitives of the Algorithmic IR. From this abstract level, the compiler performs several rewriting and lowering steps before generating VHDL code.

5.1 Lowering to the Architecture Level

In order to rewrite an expression from the Algorithmic IR, the Shir compiler traverses it and automatically replaces each occurrence of an algorithmic expression by an equivalent

$$Init[[e]] = \text{Let } mem = MemAlloc\text{<}HostRamT\text{>} \text{ in} \quad (40)$$
$$MapStm^*(\lambda d => Write(mem, d), L_A[[e]])$$
$$L_A[[Add(e)]] = Add(L_A[[e]]) \quad (41)$$
$$L_A[[Mul(e)]] = Mul(L_A[[e]]) \quad (42)$$
$$L_A[[Id(e)]] = Id(L_A[[e]]) \quad (43)$$
$$L_A[[Constant\text{<}T, N\text{>}]] = Constant\text{<}T, N\text{>} \quad (44)$$
$$L_A[[Tuple(e1, e2)]] = Tuple(L_A[[e1]], L_A[[e2]]) \quad (45)$$
$$L_A[[Map(f, e)]] = MapStm(L_A[[f]], L_A[[e]]) \quad (46)$$
$$L_A[[Reduce(f, e1, e2)]] = ReduceStm(L_A[[f]], L_A[[e1]], L_A[[e2]]) \quad (47)$$
$$L_A[[Slide(e)]] = SlideStm(L_A[[e]]) \quad (48)$$
$$L_A[[Zip(e1, e2)]] = ZipStm(L_A[[e1]], L_A[[e2]]) \quad (49)$$
$$L_A[[Split(e)]] = SplitStm(L_A[[e]]) \quad (50)$$
$$L_A[[Join(e)]] = JoinStm(L_A[[e]] \quad (51)$$
$$L_A[[Input\text{<}[T]_N\text{>}]] = \text{Let } mem = MemAlloc\text{<}HostRamT\text{>} \text{ in} \quad (52)$$
$$MapStm^*(\lambda adr => Read(mem, adr), Counter(N^*))$$

Fig. 10. Lowering L_A of Algorithmic Level to Architecture Level primitives with a flat memory representation. Expressions are represented by e, $e1$, $e2$, and f. T is a Value type and N is a Nat type.

expression from the Architecture IR. This procedure is rule-based and does not require any manual input from the user. The lowering instructions L_A are captured in the recursive rules in Figure 10.

The lowering starts in Equation (40). Here, the provided algorithm is wrapped in an expression that allocates memory in host RAM and writes the data returned by expression e back to it. Depending on the type of expression e this might require multiple nested maps, which is represented by the $MapStm^*$ keyword. Data that is to be written back to host RAM must be based on the cache line size. Shir can generate the data type conversion from the given type of e to a cache line-based type automatically, as mentioned in Section 4.4.2. For simplicity, this conversion step is omitted in Figure 10.

In most cases there is a simple one to one translation, as in Equation (41) to Equation (51). Note that, whenever possible the primitives are lowered to their stream-based counterpart, e.g., `Map` becomes `MapStm` in this lowering process. Thus, the very initially generated lower-level design will always be a fully stream-based, minimal area implementation with no parallel computation. However, once the Architecture Level is reached, rewriting can take effect and trade in the FPGA's area for performance, as described in Section 5.2 and demonstrated in Section 6.

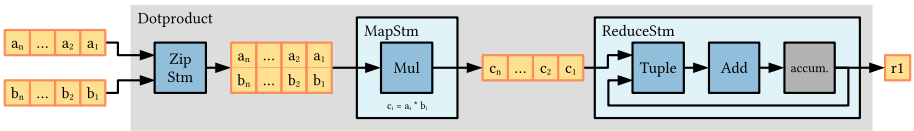
The `Input` primitive in Equation (52) is lowered to an expression, that reads the input data from host RAM. Again, depending on the specified input type, a conversion from a cache-line-based type to this desired data type may become necessary. In this case, the automatic conversion generator is employed again. The number of cache lines required for the input data does not necessarily have to match the number of input elements, due to the conversion. That is why the counter in Equation (52) counts up to N^*, which is derived from N but not always equal. If the desired input has multiple dimensions, nested `MapStm` are required, as indicated by $MapStm^*$.

5.2 Optimizing with Rewrites on the Architecture Level

5.2.1 Parallelize Computation. As described in Section 5.1, the initial expression on the Architecture Level is a stream-based design with minimal area usage, depicted in Figure 11. To exploit more of the available resources on the FPGA, the compiler automatically applies a set of rewrite

```
ReduceStm( λ a => λ b => Add(Tuple(a, b)), 0,
           MapStm(λ m => Mul(m), ZipStm(inputA, inputB)))
```

(a) Shir expression.

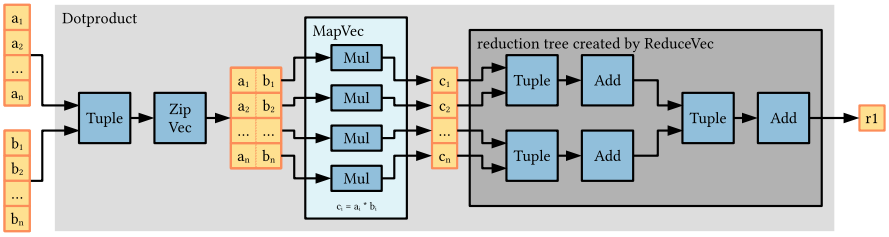


(b) Block diagram with stream notation according to Figure 5(b). The reduction contains an accumulator (box labeled "accum.") and produces a valid result only after at least n cycles. Blue boxes are primitives.

Fig. 11. Stream-based dotproduct implementation.

```
ReduceVec( // reduction tree smart constructor (see section 4.3)
  λ a => λ a => Add(Tuple(a, b)), 0,
  MapVec(λ m => Mul(m), // parallel multiplication
    ZipVec(Tuple(StmToVec(inputA), StmToVec(inputB)))))
```

(a) Shir expression.



(b) Block diagram with vector notation according to Figure 5(b). Multiple parallel multipliers and adders are instantiated. This design can produce a valid output every cycle at best.

Fig. 12. Vector-based dotproduct implementation.

rules to parallelize the computation. This converts the input stream into a vector and replaces the stream-based operators by their vector-based counterparts, as visualized in Figure 12.

The level of parallelism can be controlled by reshaping the dot product's input. In order to process N values in parallel, the input is split into chunks of size N in advance. The overall computation of the dot product then happens in a pipelining (stream) of parallel computations on vectors:

```
ReduceStm( // outer stream-based reduction
  λ a => λ b => Add(Tuple(a, b)), 0,
  MapStm(
    λ p =>
      ReduceVec( // inner parallel dot product with reduction tree
        λ a => λ b => Add(Tuple(a, b)), 0,
        MapVec(λ m => Mul(m), ZipVec(p))), // inner parallel multiplication
    ZipStm(inputA, inputB)))
```

5.2.2 Data Reusage. Whenever data is reread multiple times, buffering the data on-chip avoids repeatedly requesting that data from the slow host RAM. This generally applies, when an expression similar to the following one occurs (where f is any binary operation on two streams):

```
MapStm(λ a => MapStm(λ b => f(a, b), inputB), inputA)
```

Shir detects these specific combinations of expressions automatically and applies rewrite rules to create more efficient solutions, by inserting buffers as follows:

```
MapStm(λ a =>
  MapStm(λ b =>
    f(      // begin memory buffer
      Let bram = MemAlloc(N, ..., BlockRamT) in
        Let buffered = ReduceStm( λ mem => λ data => Write(mem, data),
                                  bram, ZipStm(a, Counter(N-1))) in
          MapStm( λ addr => Read(buffered, addr),
                  Counter(N-1) // gen addrs to read
          ) // end memory buffer
      , b
    ), inputB
  ), inputA)
```

In matrix-matrix-multiplication of two NxN matrices A and B, both matrices are read N times ($2N^2$ rows to read in total). Here, the buffer rewriting rule improves the performance. If the rows of A are buffered, the number of rows to read from host RAM decreases to N^2+N. The same happens, when the entire matrix B is buffered. However, if both the rows of matrix A and the entire matrix B are buffered, the overall number of rows to read from host RAM is drastically reduced to $2N$.

Apart from that, the rewrite rules for buffer insertion offer the ability to scale the input and output width of the memory. That way, more data can be read from the buffer in parallel. This optimization is crucial to enable highly parallel computation, which requires a faster supply with input data. If the computation was more parallel than the provided input data, the compute elements would not be used efficiently, due to waiting times. The sweet spot for matrix-matrix-multiplication is reached when both the output vector of the memory and the input vector to the computation (e.g., dot product) have the same width.

5.2.3 Timing Correction. When a Shir program is lowered and rewritten, a sequence of complex operations may be created that generate long combinational paths in the synthesized FPGA design. Reduction trees, created by `ReduceVec` are predestined to be affected by this issue. These long paths either force a slowdown of the FPGA's clock frequency, which is bad for performance, or worse, they result in faulty hardware operations. Shir contains rewrite rules that prevent this, by automatically inserting a `Registered` expression into the IR to form a pipeline after synthesis:

$$Registered : T^{BasicDataT} \mapsto T \rightarrow T. \tag{53}$$

In hardware, the inserted registers divide the long data path into several shorter sections, allowing the synthesizer to meet the desired target clock frequency.

5.3 Lowering Memory Primitives

Prior to the lowering from Abstract Memory Level to Hardware Memory Level, ramarray data is mapped into memory regions. In case there is more than one allocation in the same memory (e.g., multiple inputs from host RAM), the required memory regions are laid out one after another, by mapping an incremented base address to each of these allocations.

After that, the Shir compiler traverses the given expression in a recursive way and the lowering instructions L_M in Figure 13 are followed. Again, no manual user input is required for this rule-based procedure. Some of these rules contain additional if-conditions, that must hold to allow the substitution. If none of the first lowering instructions are applicable, the very last one, Equation (63), ensures that the recursive descent of L_M is continued.

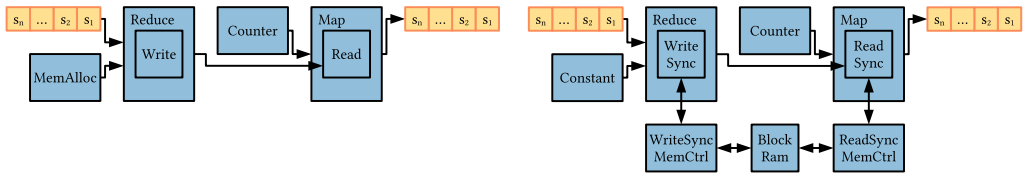
$$
\begin{aligned}
&Init[[root]] && (54)\\
&\quad = \text{Let } rdCtrl = \lambda r => ReadHostMemCtrl(r) \text{ in}\\
&\quad\ \text{Let } wrCtrl = \lambda w => WriteHostMemCtrl(w) \text{ in}\\
&\qquad \text{put}(rdCtrls, HostRamId, rdCtrl);\\
&\qquad \text{put}(wrCtrls, HostRamId, wrCtrl);\\
&\qquad L_M[[root]]\\
&L_M[[Let(p, body, mem]] && (55)\\
&\quad \text{if memId}(mem) \notin rdCtrls\\
&\quad = \text{Let } bram = \lambda p => BlockRam(p) \text{ in}\\
&\quad\ \text{Let } rdCtrl = \lambda r => ReadSyncMemCtrl(bram, r) \text{ in}\\
&\quad\ \text{Let } wrCtrl = \lambda w => WriteSyncMemCtrl(bram, w) \text{ in}\\
&\qquad \text{put}(rdCtrls, id, rdCtrl);\\
&\qquad \text{put}(wrCtrls, id, wrCtrl);\\
&\qquad Let(L_M[[p]], L_M[[body]], L_M[[m]])\\
&L_M[[MemAlloc]] = Constant(baseaddr) && (56)\\
&L_M[[MapStm(\lambda adr => Read(mem, adr), e)]] && (57)\\
&\quad \text{if memloc}(mem) = HostRamT\\
&\quad = ReadAsync(\text{get}(rdCtrls, \text{memId}(mem)),\\
&\qquad L_M[[mem]], L_M[[e]])\\
&L_M[[ReduceStm(\lambda d => Write(mem, d), e)]] && (58)\\
&\quad \text{if memloc}(mem) = HostRamT\\
&\quad = WriteAsync(\text{get}(wrCtrls, \text{memId}(mem)),\\
&\qquad L_M[[mem]], L_M[[e]])\\
&L_M[[Read(mem, e)]] && (59)\\
&\quad \text{if memloc}(mem) = HostRamT\\
&\quad = JoinStm(ReadAsync(\\
&\qquad \text{get}(rdCtrls, \text{memId}(mem)),\\
&\qquad L_M[[mem]], Repeat(L_M[[e]], 1)))\\
&L_M[[Write(mem, e)]] && (60)\\
&\quad \text{if memloc}(mem) = HostRamT\\
&\quad = JoinStm(WriteAsync(\\
&\qquad \text{get}(wrCtrls, \text{memId}(mem)),\\
&\qquad L_M[[mem]], Repeat(L_M[[e]], 1)))\\
&L_M[[Read(mem, e)]] && (61)\\
&\quad \text{if memloc}(mem) = BlockRamT\\
&\quad = ReadSync(\text{get}(rdCtrls, \text{memId}(mem)),\\
&\qquad L_M[[mem]], L_M[[e]])\\
&L_M[[Write(mem, e)]] && (62)\\
&\quad \text{if memloc}(mem) = BlockRamT\\
&\quad = WriteSync(\text{get}(wrCtrls, \text{memId}(mem)),\\
&\qquad L_M[[mem]], L_M[[e]])\\
&L_M[[other]] && (63)\\
&\quad = other \text{ , with } L_M \text{ applied on children of } other
\end{aligned}
$$

Fig. 13. Lowering L_M of Memory primitives. put, get, memId and memloc are helper functions to keep this description short and readable. Additional if-conditions restrict the application of certain rules. The remaining lower case terms are expressions, where *mem* is a *MemAlloc* expression.

The lowering procedure always starts with an initial step, described in Equation (54), that creates the read and write memory controllers for the host RAM. In order to keep track of memory controllers and use these shared functions again at a later step in the lowering process, the read controllers are put into the map container *rdCtrls* and the write controllers are put into *wrCtrls*. The helper function $put(m, id, mc)$ stores the memory controller *mc* in map *m*, while $get(m, id)$ returns the previously stored memory controller with id *id*. Furthermore, to keep the rules short, $memId(e)$ represents the id of a `MemLocT` of the ramarray type of the expression *e*. The shortcut $memloc(e)$ returns the `MemLocT` of the ramarray type of the expression *e*.

Whenever `Let` is encountered with a `MemAlloc` for a memory location, that has not been visited before during traversal, the rule Equation (55) is applied. This creates a block RAM and the corresponding read and write memory controllers, which are put in the *rdCtrls* and *wrCtrls* maps. The block RAM access is shared among the two controllers. In Equation (56), the `MemAlloc` itself is replaced by a `Constant` that returns the allocated base address, defined in the first step of this lowering procedure.

The following rules lower the abstract `Read` and `Write` primitives to the more specific `ReadAsync` and `WriteAsync` for asynchronous memory and `ReadSync` and `WriteSync` for synchronous memory. In Equations (57) and (58), the substitution will result in expressions for reading and writing with concurrent memory requests. This requires the former `Read` and `Write` expressions to be



(a) Abstract Memory Level representation. (b) Hardware Memory Level representation.

Fig. 14. Example lowering from abstract memory usage (a) to a block RAM buffering (b) of stream s. Blue boxes are primitives. `Let` expressions are not shown for simplicity.

nested in a `MapStm`, resp. a `ReduceStm`. If this is not the case, an inefficient implementation of only one read or write request at a time is generated, as in Equation (59) and Equation (60). For this case, the primitive `Repeat` is used to generate an address stream of length $N = 1$ from the given single input address.

Figure 14 depicts how a stream buffer on the Abstract Memory Level is lowered to the Hardware Memory Level. In this example, we assume that `MemAlloc` allocates memory in block RAM. Hence, on the Hardware Memory Level in Figure 14(b) a `BlockRam` is added and shared among the two read and write controllers. Due to the synchronous fashion of block RAM access, the controllers are connected to the `ReadSync` and `WriteSync` primitives. The `MemAlloc` from the Abstract Memory Level is replaced by a `Constant`, that provides the base address of the allocated memory region.

5.4 Optimizing with Rewrites on the Hardware Memory Level

5.4.1 Concurrent DMA Requests. The throughput of host RAM access via DMA depends heavily on the number of concurrently pending read and write requests. The more requests in the queue, the higher the throughput. As mentioned in Section 4.4.2, the length of `ReadAsync`'s input stream determines the maximum number of possible parallel requests. Shir contains a rewrite rule to reshape the input stream with splits and joins and thus change the number of concurrent requests.

5.4.2 Double Buffering Inputs. A common strategy to improve throughput is double buffering, where filling and consuming operations alternate between two buffers. At any time, one of the buffers is busy receiving new data, while the other one is working to send out the available data. That way, Shir can exploit pipeline parallelism to fetch data and run computations simultaneously.

The `Alternate` primitive helps to realize this functionality, by alternately feeding its inputs to one of its functions, while emitting the output of the other function:

$$Alternate : T^{NonRamArrayT} \mapsto U^{NonRamArrayT} \mapsto (T \xrightarrow{s} U) \xrightarrow{s} (T \xrightarrow{s} U) \xrightarrow{s} T \xrightarrow{s} U. \tag{64}$$

Shir contains a rewrite rule, which applies this improvement by replacing a single `ReadAsync` from host RAM with the following expression:

```
Alternate( λ offsets => ReadAsync(rdCtrl, baseAddr, offsets),
           λ offsets => ReadAsync(rdCtrl, baseAddr, offsets))
```

5.5 VHDL Code Generation

Once lowering has taken place, the domain of functional IR is left behind, to generate VHDL code for FPGAs. This is achieved by turning the IR into a dataflow graph before emitting VHDL code.

On the dataflow level, the program is modeled as a directed graph with hierarchical nodes and connections. This representation looks like the block diagrams seen earlier, in Figures 11(b)

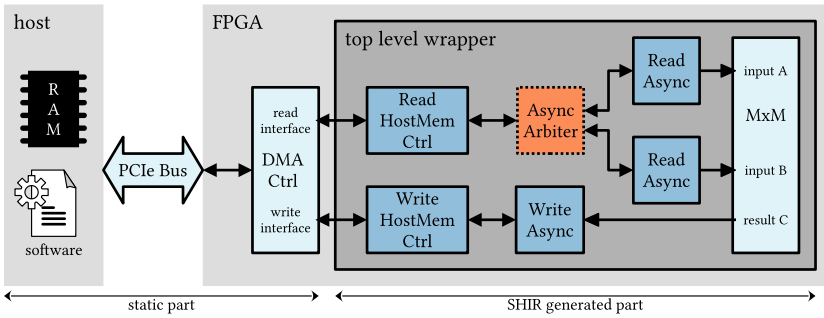


Fig. 15. Direct host RAM access from the FPGA via PCIe. The FPGA offers a DMA controller with separate read and write interfaces that are connected to the corresponding controllers of the design. The computational block, e.g., MxM here, may have more than one input. Therefore, an arbiter (dashed orange box) is inserted to distribute access to the `ReadHostMemCtrl` controller.

and 12(b), where each block is a node in the graph. Communication between nodes is synchronized with a simple handshake protocol, similar to [19], resulting in a dynamic schedule. A dynamic schedule enables the support of asynchronous features such as host RAM access. While this requires some additional control logic, we did not observe any negative impact on the overall throughput or latency, because the control signals are only 1 bit wide and the control logic remains very simple.

Data flows between components when the *valid* and *ready* signals are both active. Sequential combinational operations are combined and processed in one cycle to reduce cycle count.

More complex operations may contain internal state machines (e.g., `StmToVec`) and have registered inputs and outputs (e.g., `Mul`), which add some latency to the overall design.

5.5.1 Arbitration of Shared Resources. Whenever a `Let` expression occurs, where the parameter is used multiple times, a resource, e.g., memory, is shared among its clients. This happens for example, when the high-level program on the Algorithmic Level contains more than one `Input` expression, or multiple `ReadAsync` expressions occur on the Hardware Memory Level after lowering. In this case, the host RAM read controller is shared among all the reading clients. Additonally, for on-chip buffering a `Let` expression is used to share the block RAM instance for both read and write access.

Shared resources can only be accessed by one client at any given time. A compiler pass detects such a sharing and introduces *arbiter* nodes into the dataflow graph, as the dashed orange box in Figure 15 shows. A round robin scheduling strategy is used to fairly distribute access among the clients. Only the currently selected client is connected to the resource's data and handshake signals.

All in all, arbiters are an essential feature to enable the memory usage as modelled in the Shir IR.

5.5.2 VHDL Code Templates. Based on the information from the dataflow graph, the required VHDL templates are loaded from a database. All the graph's edges are translated into VHDL statements and written into a "wrapper" file at the top level of the design hierarchy (see Figure 15).

Shir contains more than 50 fine granular, composable VHDL templates to achieve the user's desired behaviour for the FPGA. There are complex templates like `ReadAsync` and `WriteAsync` with several hundred lines of code, but also simple ones, two of which are presented here:

First, the template for the `Add` expression, which only uses combinational logic to add the two integer values of the incoming tuple.

```
architecture behavioral of add_int is
begin
  port_out_data <= std_logic_vector(
    unsigned(port_in_data.t0) + unsigned(port_in_data.t1));
  port_out_valid <= port_in_valid;
  port_in_ready <= port_out_ready;
end behavioral;
```

Second, the `Counter` template, which is used in particular to generate addresses for memory access. The `cnt` value is increased, only if the following module in the pipeline sends back a ready signal to indicate that the current value has been consumed.

```
architecture behavioral of counter is
  signal cnt: natural range 0 to limit := 0;
begin
  port_out_data <= std_logic_vector(to_unsigned(cnt, port_out_data'length));
  port_out_valid <= '1'; -- counters always produce valid outputs

  process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        cnt <= 0;
      else
        if port_out_ready = '1' then
          if cnt <= limit - 1 then
            cnt <= cnt + 1;
          else
            cnt <= 0;
          end if;
    end if; end if; end if;
  end process;
end behavioral;
```

These examples show the decision for a uniform port design. The port's structure is consistent across all the templates to allow simple connection of the generated VHDL modules. Each port has a valid signal to indicate that the data signal contains valid information; and a ready signal, sent by the consumer, if the data has been processed.

5.5.3 Interfacing to Host. The generated top-level wrapper VHDL file contains the entire hardware design, while only the two outgoing ports of the read and write host memory controllers are exposed, as seen in Figure 15. These ports are connected to the FPGA's DMA engine.

On the software side of the host machine, a small C program loads the initial memory image into a pinned memory page in the host RAM. The base address of this memory region is sent to the FPGA via MMIO (Memory-mapped I/O). Now, all the requested addresses, from the FPGA, are offset by this base address value. The software sends a "go" signal to make the FPGA start the memory transfers and the computation. Once the FPGA has finished, a certain MMIO register is written, which is polled by the software program.

6 EVALUATION

This work focuses on expressing memory features found in real hardware. We opted for evaluating two well-known classes of applications on a real system, rather than going for a variety of benchmarks on a simulator (a much easier task). In real systems, data received from off-chip RAM

using DMA come in random order, and at random time, due to traffic on the bus. In contrast, a simulator will exhibit a deterministic behavior hiding many possible timing issues with the generated hardware, and major issues with the design of the compiler or IR. Furthermore, physical timing issues caused by long signal paths on the FPGA do not occur in simulation.

6.1 Experimental Setup

We use an Intel Arria 10 GX FPGA (at 200 Mhz), connected via PCIe Gen 3 x8, on an Intel Xeon host machine. The bitstream for the FPGA is synthesized with Quartus Prime Version 17.1.1 from the VHDL files generated by the Shir compiler. All the designs produced meet the timing requirements, and in all the experiments, the results are verified against a reference CPU implementation.

6.1.1 Benchmarks. The first set of experiments in Section 6.2 is based on a memcopy benchmark. The following experiments in Section 6.3 perform stencil computations for a 2D Convolution and a 2D Jacobi iteration. Matrix-matrix-multiplications (MxM) are performed in Section 6.4.

The Shir compiler generates the VHDL code for these benchmarks in less than a minute. As a preparation for the experiments, the software side loads the input data (e.g., matrices) into host RAM. In Section 6.5, we compare the performance of Shir and OpenCL generated hardware designs.

6.1.2 Design Space Exploration. The optimization process in Shir is based on simple heuristics. The default strategy is to parallelize as much as possible and to apply buffering whenever sufficient on-chip memory is available. We target various points in the design space, by manually disabling different optimization rewrite rules in the compiler. This helps to show the performance impact of certain optimizations. In usual compiler operation, this manual interaction with the rules is not required and the only input provided by the user is the high-level hardware-agnostic expression.

Shir is able to explore more of the design space than what is covered in this section. We omitted these designs, because they are not in the Pareto frontier with desirable area/throughput trade-offs.

6.2 Memcopy

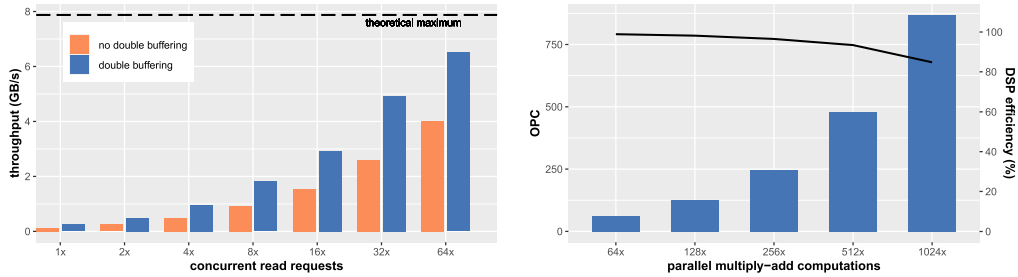
The experiments in this section show how the maximum number of concurrently pending memory read requests via DMA affects memory throughput. They cover the range of 1 to 64 concurrent requests, because the Intel Arria 10 FPGA has a hardware limit of 64. Furthermore, the impact of double buffering the input in on-chip ram as introduced in Section 5.4.2 is also considered.

To demonstrate this, a very simple memcopy-like Shir program is used, which copies data from host RAM to the FPGA and back. This program is expressed on the Algorithmic Level with an `Id` expression and the input specification, which determines the amount of data to copy (512 MB in these experiments). The Shir compiler automatically lowers this expression into the Hardware Memory Level. After that, further rewriting automatically substitutes the single buffered reading with a double buffered implementation. To explore weaker design points with fewer concurrent read requests, we manually modified the rewrite rules in the compiler flow.

The results of this benchmark are presented in Figure 16(a). As expected, the best performance is achieved when the maximum number of concurrent requests is 64 and on-chip double buffering is enabled. Furthermore, the figure allows us to observe two interesting details.

First, the effectiveness of double buffering: Using N concurrent requests with double buffering performs significantly better than using $2N$ concurrent requests without double buffering, although these two designs have the same overall limit of pending concurrent requests.

Second, a tendency towards memory bandwidth saturation: The throughput for memcopy with double buffering increases by $\sim 1.7x$ from 16 concurrent requests to 32 concurrent requests. For the



(a) Throughput in GB/s for different reading strategies in Memcopy with 512 MB of data written to and read from the FPGA. The dashed line shows the PCIe interface's theoretical maximum of 7.875 GB/s.

(b) Operations Per Cycle (OPC) as a function of number of parallel compute elements. The line shows the efficiency of the DSPs (Digital Signal Processors), which increases with fewer idle cycles of the DSPs.

Fig. 16. Memcopy with different concurrent read requests and MxM with varying levels of parallelism.

next doubling of concurrent requests, the throughput is only increases by ~1.3x. With a throughput of 6.5 GB/s, the best memcopy experiment is close to the PCIe interface's theoretical maximum of 7.875 GB/s at the physical layer. The remaining performance gap is due to PCIe protocol overheads and the mixed read and write access in our memcopy experiment. Similar DMA benchmarks [26] show similar speeds of up to 6.25 GB/s for DMA with mixed read and write access.

All the following experiments use 64 concurrent requests to maximize memory throughput and on-chip double buffering to exploit pipeline parallelism for data fetching and computation.

6.3 Stencil Computation

The following benchmarks perform stencil computations on an input matrix of 1024x128 8-bit integers. Since these operations are memory bound, we measure the efficiency by comparing the throughput to the best memcopy experiment of the previous Section 6.2. On the Algorithmic Level in Shir, this kind of computation is expressed as follows:

```
Map(λ rowGroup =>
  Map(λ group =>
    f(group) // application specific operation 'f' on the data group
    ), Slide(rowGroup, windowWidth)
  ), Slide(image, windowHeight))
```

6.3.1 2D Convolution. For 2D Convolution, the inner function `f` in the code above computes a dot product with weights. The experiment shown in Table 1 uses a 3x3 kernel size. The Shir compiler automatically buffers some rows of the input data, so that each row has only to be read once in the entire runtime. The computation of each output element is fully parallelized. With a throughput of 6.4 GB/s the design generated by Shir is as fast as memcopy, saturating the memory bandwidth. Larger input sizes exhibit the same performance.

6.3.2 2D Jacobi. In this benchmark, a single iteration of 2D Jacobi with a 4-point stencil is performed. Here, the inner function `f` in the code above computes the average value of the 4 adjacent points. Again, Shir inserts input row buffers to maximize the performance. The generated hardware does not employ any DSPs, since the algorithm only divides by 4, which is automatically rewritten as a shift operation by the compiler. Nevertheless, the design generated by Shir is very efficient, because it reaches a throughput similar to that of memcopy, as shown in Table 1.

Table 1. Performance and Area Usage of Logic (ALMs (Adaptive Logic Modules)), RAM and DSPs (Digital Signal Processors) for Benchmarks with Stencil Operations on Input Matrices of 1024x128 8-bit Integers

Configuration	**Performance**		**Resource usage**		
Operation	throughput	efficiency*	ALM	on-chip RAM	DSP blocks
2D Convolution	6.4 GB/s	99%	18%	3%	75%
2D Jacobi	6.4 GB/s	99%	8%	4%	0%

*efficiency is the ratio of the experiment's measured throughput to the maximum throughput of memcopy (6.5 GB/s).

Table 2. Experiments for Matrix-Matrix-Multiplication of Two 1024x1024 Matrices with 8-bit Elements

	Configuration			**Performance**				**Resource usage**		
exp. no.	buffers A_{row}	buffers B	par. comp.	host RAM reads	GOPS	OPC	efficiency*	ALM	on-chip RAM	DSP blocks
1			64	33,554,432	1.4	7	11.2	9%	3%	4%
2	✓		64	16,793,600	1.6	7	12.3	9%	3%	4%
3		✓	64	16,793,600	1.7	8	13.0	9%	20%	4%
4	✓	✓	64	32,768	12.7	63	98.9	10%	20%	4%
5	✓	✓	128	32,768	25.1	125	98.2	11%	20%	8%
6	✓	✓	256	32,768	49.4	247	96.5	15%	20%	17%
7	✓	✓	512	32,768	95.7	478	93.5	22%	20%	34%
8	✓	✓	1024	32,768	173.7	868	84.8	36%	20%	67%

Designs differ in buffering strategies and levels of parallel computation, which affect the performance: Host RAM read requests via DMA, GOPS (Giga Operations Per Second) (i.e., 10^9 multiply-add per second), Operations Per Cycle (OPC), and DSP usage efficiency. Furthermore, the area usage of logic (ALMs), RAM and DSPs is listed.
*efficiency is measured as the ratio of active cycles of a DSP to overall cycles for the experiment. With fewer idle cycles, the DSPs work more efficiently. At 100% efficiency they produce a new value each cycle and never have to wait for input data.

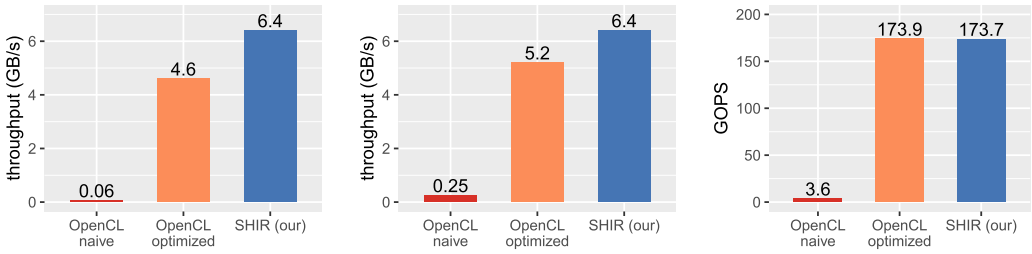
Both of the above stencil computations are memory bound and therefore leave not many design choices for interesting optimizations. The more challenging design space for MxM is explored in the next section.

6.4 Matrix-matrix-multiplication

For MxM, the input matrices consist of 1024x1024 8-bit integers. The experiments are listed in Table 2. We assume here that the matrix B is already transposed on the host to simplify the expression. The MxM experiments are based on the following Algorithmic Level expression, which is the only user input required by the Shir compiler:

```
Map(λ rowA =>
  Map(λ colB =>
    Reduce(λ a => λ b => Add(Tuple(a, b)), 0, Map(λ m => Mul(m), Zip(rowA,
    colB)) // dot product
    ), Input(IntT(8), N, N) // matrix B
  ), Input(IntT(8), N, N)) // matrix A
```

6.4.1 Data Reusage. The experiments 1-4 in Table 2 show how different buffering strategies affect the performance and on-chip RAM usage. In order to avoid artificial slowdown of the



(a) 2D Convolution throughput. (b) 2D Jacobi throughput. (c) MxM performance.

Fig. 17. Performance comparison of naive and optimized OpenCL code to Shir implementations.

designs, one cache line has to be processed in parallel, so that no time-consuming vector to stream conversion occurs. That is why the computation is parallelized by 64, which corresponds to the number of 8-bit elements in a single cache line.

In the first experiment, no buffering is employed. If data is repeated in the computation, it has to be read from host RAM again. This leads to $2N^2$ rows to be read. Each row consists of 1,024 elements and therefore 1024/64 = 16 cache lines. The total number of cache lines read in this experiment is $2 * 1024^2 * 16 = 33{,}554{,}432$.

The second experiment applies the rewrite rule explained in Section 5.2.2. The row of matrix A is buffered, which reduces the number of cache lines read by ~50% to 16,793,600 for $N^2 + N$ rows. There is no significant performance improvement in terms of GOPS, because the matrix B still has to be read from slow host RAM.

For experiment 3, the entire matrix B is buffered with a similar result to the previous experiment. However, the on-chip RAM usage increased to 20%, because the buffer for the matrix requires 1,024 times more memory than a row buffer.

In experiment 4, the buffer for the rows of matrix A and the buffer for matrix B can finally leverage the performance (~9x better than experiment 1). The number of rows to read is decreased to $2N$, which corresponds to 32,768 cache lines.

In these experiments, no tiling is necessary because the entire tile fits in the FPGA's on-chip memory. All the following experiments buffer both the rows of matrix A and the entire matrix B.

6.4.2 Parallelize Computation. In this section, the rewrite rules from Section 5.2.1 and Section 5.2.2 are applied to evaluate the impact of parallelizing the computational part of MxM. Experiments 4-8 in Table 2 cover the range from 64x to 1,024x parallel multiply-add operations. The connection of the on-chip input buffers to the computational part are rewritten to allow more parallel data access, as mentioned in Section 5.2.2.

Figure 16(b) shows that performance scales with the degree of parallelism. The DSPs are used efficiently, because they perform valid multiply-add operations for 85% to 99% of the overall runtime, which includes the data transfer between host RAM and FPGA. In experiment 4, the DSPs receive new input data almost every cycle.

6.5 Comparison to OpenCL

In this section, we compare the performance of the generated hardware implementations from Shir and OpenCL on the same Intel Arria 10 FPGA. The results are shown in Figure 17.

For the OpenCL implementations, we use two versions: A naive parallel OpenCL code, written in a hardware-agnostic way; and an optimized OpenCL code with hardware-specific pragmas as well as explicit local memory usage. Neither Shir's Algorithmic Level code nor the naive OpenCL code

contain any explicit unrolling, local buffers, or other hardware-related optimizations. Still, Shir's generated designs outperform the naive OpenCL versions by up to ~100x, as shown in Figure 17. This is because Shir automatically introduces hardware optimizations by applying rewrite rules.

Matrix-matrix-multiplication. The optimized OpenCL MxM implementation comes from Intel.[1] The generated hardware performs as good as the Shir version. However, with ~30 LOC (Lines of Code), this OpenCL code is more verbose compared to the truly hardware-agnostic Shir input code.

Stencil Computation. For the optimized OpenCL design for 2D Convolution and 2D Jacobi, we follow the techniques mentioned in [13] and [15], such as unrolling and local buffering. To further improve the parallelism and the pipeline efficiency, we choose optimal pragmas and optimization flags. In addition, the memory access is improved by tuning the cache settings in the compiler. We observed some communication overhead in OpenCL for small input sizes of about 128 KB and increased it to 2 GB for better OpenCL results and a fairer comparison.

After exploring all the above-mentioned optimizations, the best OpenCL designs achieve 4.6 GB/s for 2D Convolution and 5.2 GB/s for 2D Jacobi. Again, the Shir code is truly hardware-agnostic and much more compact in comparison to the ~20 LOC of the optimized OpenCL implementation. Moreover, this time even the performance of the Shir generated hardware outperforms the optimized OpenCL version by up to ~1.4x.

6.6 Summary

The goal of Shir is to offer the best of two worlds: high-level hardware-agnostic abstractions for developers and high-performance hardware accelerator implementations. In this evaluation section, we have shown that a multi-level functional IR is well suited to generate efficient FPGA implementations. Although limited, the experiments presented in this article are prime examples demonstrating the potential for such an approach.

7 RELATED WORK

Hardware Design Languages. Historically, hardware is programmed using low level, close to the gates, languages such as Verilog and VHDL. Slightly higher level languages exist such as Bluespec [28], Esterel [11], or more recently Chisel [2], Fleet [37], and Cλash [1], offering higher-level abstractions. Bluespec features functional programming but remains quite verbose. Chisel is embedded in Scala and takes advantage of modern features such as polymorphism and higher order functions. However, these approaches still require programmers to understand hardware concepts, a major obstacle for non-experts.

High-Level Synthesis. High-level synthesis includes approaches that use C-like annotated code, such as Intel's OpenCL SDK, Vivado HLS, SDAccel, LegUp [6] or SOFF [16]. Writing such codes requires many hardware-specific manual optimizations [25].

Another approach is to use functional languages like AnyHSL [29] and [24, 38] or dataflow languages as in [14, 17, 35] and LiquidMetal [3]. Delite [36] uses a functional language and also targets FPGAs in [30]. They employ parameterized hardware templates, which have the disadvantage that they hide memory implementation details. In contrast to Shir, the memory usage in [30] is not exposed in the high-level IR, which prevents optimizations with high-level rewrite rules.

[1]https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html.

Its successor, Spatial [18], raises the programming abstraction, but remains lower level and more verbose compared to Shir. Furthermore, Spatial already includes key performance decisions at their highest abstraction level, where Shir only focuses on the algorithm itself.

HeteroCL [20] features an abstract algorithm specification similar to Shir. It has a general back-end, one for stencil patterns, called SODA [8], and a PolySa [9] back-end for systolic array architectures. All these back-ends generate OpenCL or C-like code, which still lacks predictability [27], portability [29] and suffers from bugs and unsupported features [16]. Furthermore, these approaches do not leverage multi-level IRs with rewrite rules the way Shir does.

DNNWeaver [33] and [23] present parameterized deep learning accelerator architectures. They can handle different CNN configurations with various input matrix and weight sizes, as well as low data precision, by adjusting the parameters of the architecture template accordingly. However, the general structure of their monolithic hardware design is fixed, while Shir uses fine-grained multi-purpose templates, that can be composed to implement the desired behaviour. Shir's approach is beneficial for the expressiveness, maintainability, and modularity, according to [4].

This paper follows the track of functional programming abstractions but uses a high-level hardware-agnostic language as an entry point and lets the compiler explore the design space. The use of rewriting for expressing design choices also guarantees correctness by construction.

Domain-Specific Synthesis. With Spiral [22, 32], linear signal processing transforms can be automatically compiled into RTL for FPGAs and ASICs (Application-Specific Integrated Circuits). It supports a similar notion of stream-based data to enable pipelining and it applies rewrite rules to optimize the design. Halide-HLS [31] is another domain specific approach focused on image processing. In contrast to these works, our approach aims at being generic and domain-agnostic.

Multi-Level IRs. Recently, MLIR [21] has shown the advantages of a compiler infrastructure with multiple IRs on different abstractions levels. Shir's infrastructure shares many ideas of MLIR. Similar to MLIR's tensors, Shir models high-level data in n-dimensional arrays. Both compilers lower these into less abstract types like vectors for SIMD tasks or ramarrays (memrefs in MLIR) for buffering. Shir's memory operators could as well be implemented as an MLIR dialect. However, this is an implementation detail and orthogonal to the concept introduced by Shir.

High-Level Synthesis with Functional IRs. HML [39] and Lava [5] exploit functional-based languages for hardware generation. However, they are not really high-level since hardware concepts are exposed at the highest level, requiring hardware expertise.

Lift-hls [19], Cλash [1] and Aetherling [10] have recently shown the advantages of functional IR. Similar to Shir's vector and stream types, these two feature space- and time-aware types to express parallelism and pipelining on a high level of abstraction.

Lift-hls [19] supports some limited form of memory operations but relies on an ad-hoc approach to move data to/from the FPGA and has also not discussed how this is achieved.

Aetherling [10] has presented results on a larger set of applications. However, they have only demonstrated results in simulation using a simple memory model and have not discussed how memory would be handled on a real hardware. In contrast, we evaluate matrix-matrix-multiplication, which brings some major challenges, for example, the "repetition" of rows from the first matrix. Moreover, all our designs produced have been run on real hardware. To our knowledge, Aetherling is unable to handle blocking communication or express asynchronous communication, which are necessary when targeting a real FPGA.

8 CONCLUSIONS AND FUTURE WORK

This article has presented the Shir multi-level IR which exposes lower-level architectural features explicitly. Uniquely, the Shir IR exposes low-level memory operations which is necessary to

produce hardware that can run on real FPGAs. Such low-level operations include the use of off-chip memories via asynchronous communication, the use of synchronous on-chip memory as well as the ability to exploit a double-buffering mechanism.

Starting from a high-level algorithmic description of a program, the Shir compiler lowers the IR mechanically, step by step. The rewrite operations for lowering have been formalized in this work and always lead to a valid hardware implementation. Another set of rewrites is also used (formalization not shown for space reason) to explore some interesting optimization choices such as exploiting spatial parallelism or use of block RAM memories for data reusage.

The approach has been evaluated on real hardware, using an Intel Arria 10 FPGA, for two stencil computations and a small-scale design space exploration has been conducted using matrix-multiplication. As demonstrated, the hardware designs generated are able to effectively exploit all the hardware features available and are competitive with OpenCL implementations, with a much higher level programming abstraction.

In the future, the VHDL template for reduction will be enhanced to support accumulating matrices. Thus, Shir can apply tiling methods to reduce the off-chip RAM access for programs with large input data. While the presented memory concepts remain applicable and the memory primitives can be reused as they are, this future work still requires some hardware implementation effort.

Another plan is to implement more complex programs with Shir. These will offer larger design spaces to be explored with many more different ways to apply the optimization rewrite rules. For these more complex applications, we can imagine a constraint solving method to identify the best rewrite rule targets in the IR. Since the effect of optimizations on the performance of the generated hardware designs are fairly predictable, no interfacing with machine learning is necessary.

We also aim to target other FPGAs, like devices from Xilinx, in the near future. The development efforts required for this is mainly engineering work, as some of the hardware templates have to be adjusted to the new interfaces of the FPGA.

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] Paul Barham and Michael Isard. 2019. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)* (Bertinoro, Italy). ACM, New York, 177–183. https://doi.org/10.1145/3317550.3321441

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware design in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (Bertinoro, Italy)* (Baltimore, Maryland). ACM, New York, 174–184. https://doi.org/10.1145/289423.289440

[6] 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 (FPGA)* (Monterey, CA). ACM, New York, 33–36. https://doi.org/10.1145/1950413.1950423

[7] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An extension of system F with subtyping. In *Theoretical Aspects of Computer Software*, Takayasu Ito and Albert R. Meyer (Eds.).

[8] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)* (San Diego, CA). ACM, New York, 1–8. https://doi.org/10.1145/3240765.3240850

[9] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. *Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)* (San Diego, CA). ACM, New York, Article 117, 8 pages. https://doi.org/10.1145/3240765.3240838

[10] 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)*. ACM, New York.

[11] Stephen A. Edwards. 2002. High-level synthesis from the synchronous language esterel. In *IWLS*. 401–406.

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[13] Kenneth Hill, Stefan Craciun, Alan George, and Herman Lam. 2015. Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In *Proceedings of the 2015 IEEE 26th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 189–193. https://doi.org/10.1109/ASAP.2015.7245733

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

[15] Qi Jia and Huiyang Zhou. 2016. Tuning stencil codes in OpenCL for FPGAs. In *Proceedings of the 2016 IEEE 34th International Conference on Computer Design (ICCD)*. 249–256. https://doi.org/10.1109/ICCD.2016.7753287

[16] G. Jo, H. Kim, J. Lee, and J. Lee. 2020. SOFF: An OpenCL high-level synthesis framework for FPGAs. In *In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 295–308.

[17] 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.

[18] 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)*.

[19] 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)*. ACM, New York.

[20] 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

[21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. arXiv:2002.11054 [cs.PL]

[22] Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2012. Computer generation of hardware for linear digital signal processing transforms. *ACM Trans. Des. Autom. Electron. Syst.* 17, 2, Article 15 (April 2012), 33 pages. https://doi.org/10.1145/2159542.2159547

[23] 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

[24] Alan Mycroft and Richard Sharp. 2000. A statically allocated parallel functional language. In *Proceedings of ICALP*. 37–48.

[25] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A survey and evaluation of FPGA high-level synthesis tools. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 35, 10 (Oct. 2016), 1591–1604.

[26] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)* (Budapest, Hungary). ACM, New York, 327–341. https://doi.org/10.1145/3230543.3230560

[27] 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 *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. ACM, NewYork.

[28] 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

[29] 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.

[30] 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 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA). 651–665. https://doi.org/10.1145/2872362.2872415

[31] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article 26 (Aug. 2017), 25 pages. https://doi.org/10.1145/3107953

[32] 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

[33] 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. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Oct. 2016). https://doi.org/10.1109/micro.2016.7783720

[34] 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)*. ACM, New York.

[35] 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.

[36] 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

[37] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A framework for massively parallel streaming on FPGAs. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 639–651.

[38] 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 (CC)* (Austin, TX,). 11 pages.

[39] Yanbing Li and M. Leeser. 1995. HML: An innovative hardware description language and its translation to VHDL. In *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*. 691–696.

Received May 2021; revised November 2021; accepted November 2021