A Dataflow Inspired Programming Paradigm for Coarse-Grained Reconfigurable Arrays

A. Niedermeier, Jan Kuper, and Gerard J.M. Smit

Computer Architecture for Embedded Systems Group
Department of Electrical Engineering, Mathematics and Computer Science
University of Twente, The Netherlands

Abstract. In this paper, we present a new approach towards programming coarse-grained reconfigurable arrays (CGRAs) in an intuitive, dataflow inspired way. Based on the observation that available CGRAs are usually programmed using C, which lacks proper support for instruction-level parallelism, we instead started from a dataflow perspective combined with a language that inherently supports parallel structures. Our programming paradigm decouples the local functionality of a core from the global flow of data, i.e. the kernels from the routing. We will describe the ideas of our programming paradigm, and also the language and compiler itself.

1 Introduction

Many algorithms common in digital signal processing (DSP), like for example audio filtering, contain a high degree of instruction-level parallelism. To accelerate those algorithms, coarse-grained reconfigurable arrays (commonly used in combination with a general purpose control processor) are often used. In such systems, the mostly sequential control operations are executed on the host processor, while the data-intensive algorithms are executed on the reconfigurable array. The reconfigurable array is composed of a network of simple blocks containing a function unit and a small local memory. To run a certain application, the blocks and the interconnects are configured accordingly. Programming CGRAs however remains a challenge, since it is not trivial to extract the instruction level parallelism from a given algorithm.

In the following, we will first give a brief overview about currently available CGRAs. Then, we will shortly discuss the challenges that arise from programming the available CGRAs. Finally, we will motivate our programming paradigm and which is then presented in the remainder of the paper.

1.1 Related Work

A number of articles presenting reconfigurable architecture together with a programming paradigm have been published already. Early publications include PADDY-2 [1] and Matrix [2]. Examples of more recent publications are MorphoSys [3], the eXtreme Processing Platform (XPP) [4], the Reconfigurable Instruction

Cell Array (RICA) [5], and ADRES [6]. In general, they all are composed of an array of small configurable blocks interconnected by a configurable network. Since the exact details are out of scope of this publication, the reader is referred to the respective publications.

The above mentioned architecture all have in common that they are programmed in a C-based approach (or, for the earlier architectures, even in an assembly language). The burden of extracting the algorithm structure (in terms of parallelism and dataflow) lies on the compiler. This implies that the structure of the algorithm is not inherently available in the algorithm specification and might not be completely recognised by the compiler.

1.2 Our contribution

In our opinion, the choice of C for programming CGRAs is not an obvious one. Since C has been designed as a sequential language, it lacks intuitive support to express fine-grained parallelism. Of course, one could argue that C is the standard programming language that is used to implement applications. However, none of the compilers presented for the above mentioned architectures support the complete set of C. Instead, for every architecture, a specific subset of C is chosen.

Motivated by the difficulty of programming such arrays, we developed a dataflow-inspired programming paradigm. In our programming paradigm, a DSP application is represented and implemented as a set of dataflow nodes.

The programming language itself is implemented as an embedded domain specific language (EDSL) in Haskell. This enables a designer to implement DSP applications in a concise and straight-forward manner by using Haskell's higher order functions. As these functions have a notion of structure, all information on parallelism and flow of data is automatically contained in the resulting expressions.

2 Architecture

The architecture for which we present our programming paradigm is a grid of small, independent, reconfigurable cores. It is shown in Figure 1. Each core is connected to its direct neighbours using point-to-point links. Furthermore, the grid is fully connected using a network-on-chip. External data can be provided by using either a broadcast input which is connected to each core, or direct inputs to the cores.

Each core in the architecture follows the rules of dataflow, i.e. as soon as all operands for a certain core are available, the configured operation is performed. Inside each core, a function unit is available, which can execute binary operations, i.e. addition, multiplication and the like. Furthermore, a local register file to store intermediate results and a storage for constants are available. Each core is independent in the sense that it does not have a notion of the global topography of the complete grid. Hence, it only has the notion of internal functionality, the global flow of data is out of the core's scope. For more details on the architecture, the reader is referred to [7].

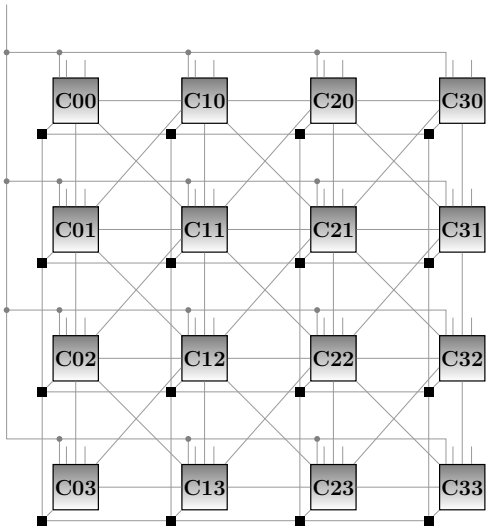


Fig. 1: The architecture

3 Programming Paradigm

In this section, we present our programming paradigm. We adopt ideas from dataflow, such as the firing rule (i.e. a node executes its operation once all required inputs are available and there is sufficient space at the output) and the representation of an algorithm as a graph with nodes representing the operations and edges representing the flow of data. Furthermore, we use finite state machines to extend the possibilities of pure dataflow notation.

On the conceptual level, we consider a DSP application on two different views: The *local view*, i.e. everything that is executed locally on one core, and the *global view*, which is the global flow of data through the array.

3.1 The local view

Figure 2 shows a high-level illustration of both the local view and a core of the target architecture. As mentioned in Section 2, each core includes a function unit, represented by the *OP* block, a register file, represented by the *R* block and a constant storage, represented by the *C* block.

The configuration of the local view is described as a sequence of stages. Each stage is defined in terms of

- source of each **input** (***EX***ternal input $\boldsymbol{i}$, a ***C***onstant stored at $\boldsymbol{x}$, ***R***egister $\boldsymbol{x}$)
- **opcode** defining the current operation (***ADD, MUL, ...***)
- whether to **store** the result (store at ***R***egister $\boldsymbol{x}$, _ (do not store))

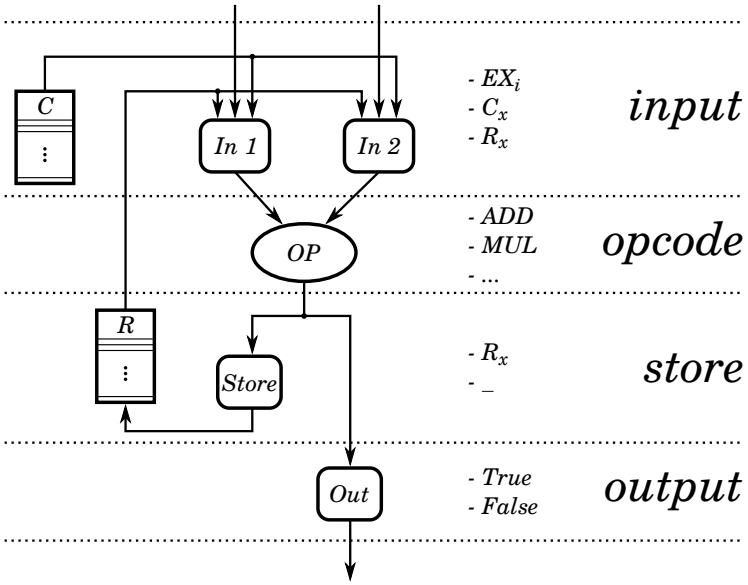


Fig. 2: Configuration principle

– whether the result is visible at the **output** (***True, False***)

For illustration, we use the example of a pipelined multiply-accumulate (MAC) operation on data streams. The MAC operation on the streams x and y is defined as follows:

$$mac = \sum_{i=0}^{N} x_i y_i = x_0 y_0 + x_1 y_1 + x_2 y_2 + \ldots + x_N y_N \tag{1}$$

The *mac* operation is implemented in a pipelined fashion on a single core using separate stages for the multiplication and addition. The implementation of the complete *mac* operation requires three stages. In Figure 3, the configuration is shown.

The first stage is labelled *S0*. Here, the tokens available on the external inputs EX_0 and EX_1 (corresponding x_0 and y_0 from Equation 1) are multiplied and stored in the register file at R_0. The second stage, *S1*, is then executed, which represents a multiplication of x_1 and y_1. The result of this multiplication is stored in R_1. The third stage, *S2*, performs an addition on the values stored in R_0 and R_1 and stores the result in the R_0. From here on, the core alternates between stages *S1* and *S2*.

3.2 The global view

While the local view defines everything that happens inside a core, the global flow of data is out of the core's scope. A core only has the notion that an input

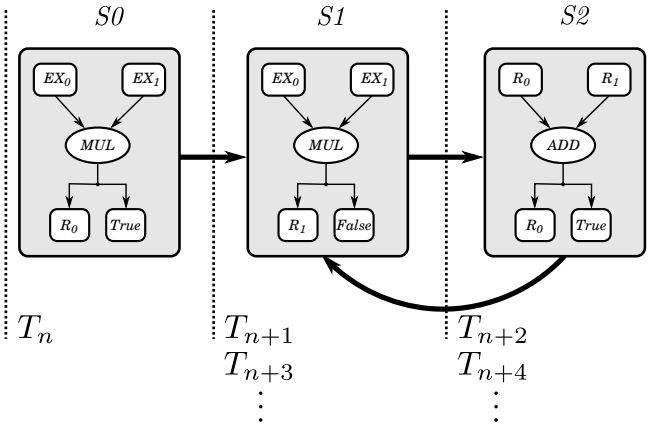


Fig. 3: Local view

can come from an external source, for example another core, but precisely which core is irrelevant. Consequently, for the flow of data a global dataflow scheme is required, i.e. the global view. Tokens are exchanged between different cores in the array by having named inputs and outputs in the local view that correspond to communication channels in the global view. The global view is illustrated in Figure 4. In this example, four cores (*C0* to *C3*) communicate using the core's address and the name of the core's inputs. For example, *C0* is sending a token to input *0* of *C1* by annotating (*1,0*) to the token.

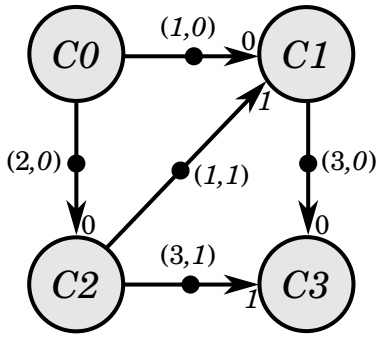


Fig. 4: Global view

The routing of the outgoing tokens is then handled by the interconnection logic in the grid. By decoupling the local specification of a core and the global routing of data, the cores do not require a knowledge of the complete topography and can thus be independently defined.

4 Language and Compiler

In this section we present the specification of the compiler. The starting point is the definition of an Embedded Domain Specific Language (EDSL) targeted at the presented architecture. Then we will show how this EDSL can be used in combination with Haskell's higher order functions to specify algorithms in a convenient way by exploiting the implicit parallelism. Finally, the code generation is presented.

4.1 EDSL

The grammar for the EDSL is based on the operations, which the cores in the architecture can execute, i.e. simple binary operations like multiplication and addition. Furthermore, a notation for delays and feedback loops to the same node is supported. The EDSL is implemented as an algebraic datatype in Haskell. By defining the EDSL as a datatype, one can use Haskell functions to recursively construct a complex expression.

As a consequence, the resulting expression is then the abstract syntax tree (AST) of the expression that was specified. This means that the parser is "for free".

In Listing 1.1, the definition for the EDSL datatype is given. The names of the constructors hereby resemble their functionality. In line 1, the definition how to specify a constant number is given, line 2 specifies how a delay is defined, the definition in line 3 shows how the result from the previous clock cycle can be used (i.e. a feedback loop), line 4 represents an input where the string denotes and input stream and finally in line 5 the operation itself is defined. *Op* is a data constructor of the type *Expr* and indicates a compound operation, and *OpCode* defines the opcode.

```
data Expr = Const Number                                    1
          | DELAYED Expr                                    2
          | PREV_RES                                        3
          | Input String                                    4
          | Op OpCode Expr Expr                             5
data OpCode = ADD | MUL | SQR | AND ...                     6
```

Listing 1.1: recursive EDSL definition for an expression

4.2 A first example

Next we want to show how a simple algorithm can be implemented by using the EDSL in combination with a higher order function. If we want to implement a simple summation of all elements in a vector x, it can be written as follows: $sum\_up\ x = foldl\ (Op\ ADD)\ (Const\ 0)\ x$

As example, the function *sum_up* is applied to an input vector of length four. The expression tree is then automatically unrolled:

```
ghci> sum_up [Input "x0",Input "x1",Input "x2",Input "x3"]
ghci> Op ADD
            (Op ADD
                  (Op ADD
                        (Op ADD (Const 0) (Input "x0"))
                        (Input "x1"))
                  (Input "x2"))
            (Input "x3")
```

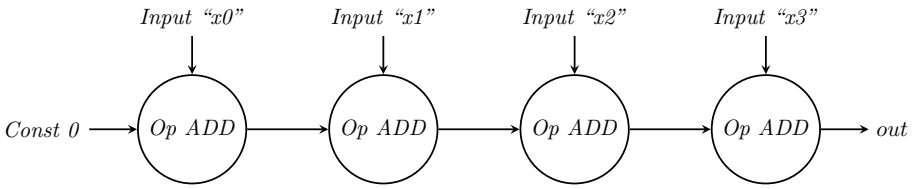


Fig. 5: *sum_up xs*

Note that the input x to *sum_up* is now of type *Expr*, not a list of numbers. The type of *sum_up* x itself is also *Expr*, i.e. it is an expression in our EDSL. The function *foldl* distributes the *Op ADD* over the list of inputs x, the initial value is given by *Const* 0. A graphical representation of this concrete usage of *sum_up* is shown in Figure 5.

4.3 Streaming notation

In many cases, a streaming notation for specifying a chain of operations is desired. Consider the case shown in Figure 6. To the left, a stream x is streamed into the system. In the first stage, *kernel1* performs its computation on x, then *kernel2* executes on the output of *kernel1* and finally *kernel3* is applied to the output of *kernel2*. In our compiler, we implemented a function that supports this streaming notation:

$(\rightarrow)\ a\ f = f\ a$

$\rightarrow$ accepts an argument a and a function f as arguments and applies the function f to the argument a. A usecase example is shown in Listing 1.2. In lines 1 to 3, the kernels are defined, in lines 4 and 5 two different implementation for the streaming pipeline shown in Figure 7 are presented.

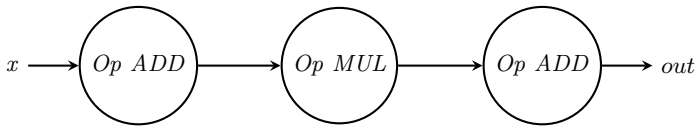


Fig. 7: Graphical representation of the implemented streaming pipeline

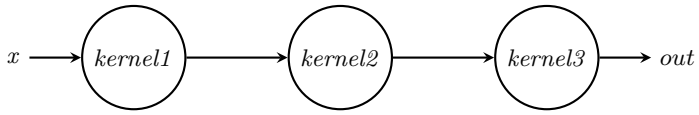


Fig. 6: Implementation of a streaming pipeline

```
kernel1  x = Op ADD x x                                          1
kernel2  x = Op MUL x x                                          2
kernel3  x = Op ADD x x                                          3
stream0 x = foldl (→) x [kernel1, kernel2, kernel3]              4
stream1 x = x → kernel1 → kernel2 → kernel3                      5
```

Listing 1.2: Implementation of a streaming pipeline

4.4 Generation of the code for the cores

In order to generate code for the hardware architecture, an expression in the EDSL is converted into a unique graph as shown in the previous section. The next step of the compiler is to iterate through all nodes in this graph and generate the correct code.

Each node is one of the five different possible cases given in Listing 1.1: A constant, a delayed expression, a pointer to the previous result, an input, or an operation.

Code is only directly generated for nodes that define an operation. All the other cases are used as inputs by the operation nodes and are handled there.

The code generation for an operation node can be split into two cases: Either, the expression is simply an operation on two incoming, non-delayed and non-fedback signals, or one or more of the inputs comprise a delay or a feedback loop.

For a simple expression, code generation is straight forward, for the non-simple case, the delay or the feedback have to be taken into account by providing an initial token for the first iteration and providing information where the data should be stored in the register file. To illustrate the different possibilities, three examples are shown as use cases. In Figure 8, the resulting graph is shown for an addition of one external input and a constant.

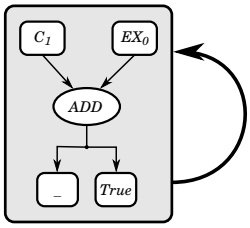


Fig. 8: *out* $y = Op\ ADD\ (CONST\ 1)\ y$

Figures 9 and 10 show two slightly more complex examples where either one input is delayed (*out x y = Op ADD (DELAYED x) y*) or where one of the operands is the previous result, thus forming a feedback loop (*out y = Op ADD PREV_RES y*).

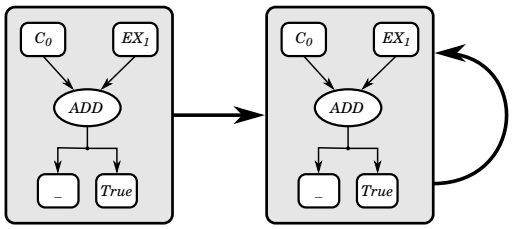


Fig. 9: *out x y = Op ADD (DELAYED x) y*

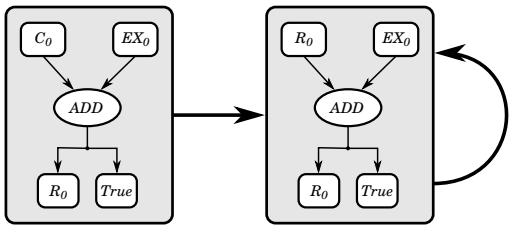


Fig. 10: *out y = Op ADD PREV_RES y*

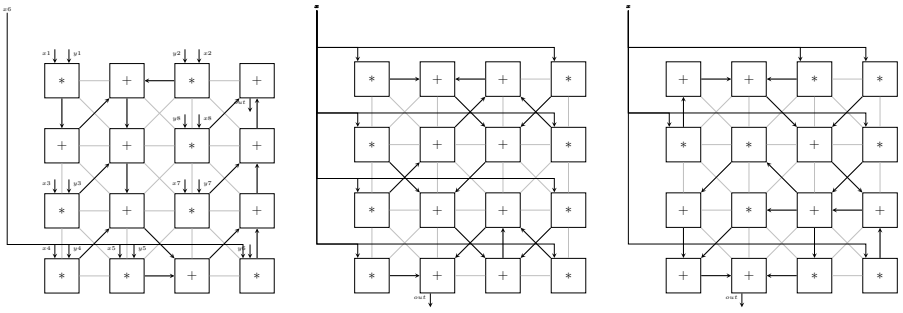
All information except the routing information has now been generated, the routing settings are derived in the next step, the mapping.

4.5 Mapping to the target architecture

After the code for all the cores has been generated, each node in the algorithm graph is mapped to one core in the hardware architecture. This is performed using simulated annealing [8]. The result is a mapping of each node in the algorithm graph to a core in the hardware architecture. The mapping is written to a file which is then processed within the compiler. Using the mapping information, the last missing step in the code generation, namely the global view, i.e. the routing settings, is filled into the programming code of the nodes.

5 Case Studies

To illustrate the usage principle of the complete design flow, three use-case algorithms are presented in this section: a multiplication of two vectors, a finite-



(a) Mapping of *vxv* (b) Mapping of *fir* (c) Mapping of *stream*

Fig. 11: Mapping results

impulse-response (FIR) filter and a chained FIR filter demonstrating the previously introduced streaming notation.

5.1 Multiplication of two vectors

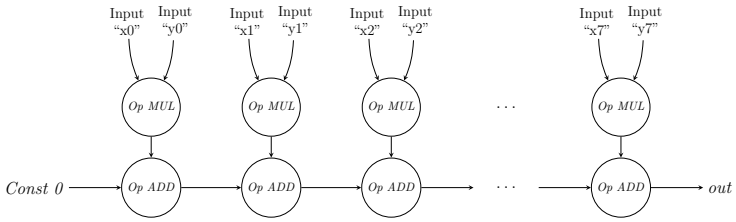


Fig. 12: Structure of *vxv*

The first step in the design flow is shown in Listing 1.3. In this step, the algorithm is specified within our compiler using the EDSL and Haskell's higher order functions.

```
vxv x y = out                                    1
  where                                          2
    ms = zipWith (Op MUL) x y                    3
    out = foldl (Op ADD) (Const 0) ms            4
```

Listing 1.3: Implementation of a multiplication of two vectors

The structure is built using the two higher-order functions *zipWith* and *foldl*. A graphical representation of *vxv* for two vectors of length eight is shown in

Figure 12. In line 1 of the code, the function name *vxv* and its arguments x and y which are the two vectors to be multiplied are defined. *out* is the resulting output. In line 3, the vectors are element wise multiplied which leads to the row of *Op MUL* in Figure 12. Finally, in line 4, the results of the multiplications are added up, which leads to the row of *Op ADD* in Figure 12. Note that the input vectors represent input signals denoted with *Input "name index"*.

The next step in the design flow is the actual code generation and mapping to the target architecture. The mapping for *vxv* is shown in Figure 11a. Both the code generation and mapping were automatically derived using the compiler described in Section 4.

5.2 FIR filter

The next example is a complete example for an 8-tap FIR filter. The filter was implemented in the transposed form, the structure is displayed in Figure 13. The tokens on the arcs represent delay elements between the additions.

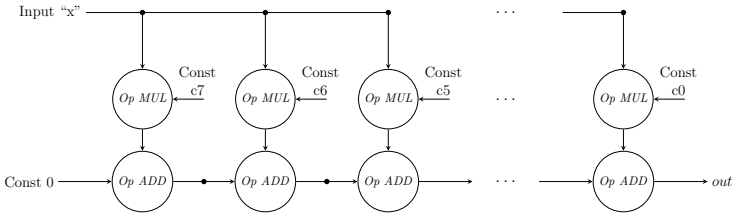


Fig. 13: Structure of *fir*

The implementation using the EDSL is shown in Listing 1.4. In line 1, the function name *fir* and the arguments are defined, in this case c (the filter coefficients) and x (the input sample). In line 3, x is broadcasted to the row of *Op MUL* from Figure 13. Here, the function *map* together with an initial argument x is used . This defines a multiplication of each filter coefficient with the input x. In line 4, the delayed addition represented by the bottom row in Figure 13 is implemented. For this, the higher order function *foldl* is used that starts off with the constant value 0 (shown to the left in Figure 13 and then applies a delayed addition *dadd* (which is defined in line 5) to each result of the previously defined multiplications *ms*. It can be seen that the FIR filter has a comparable communication structure to the multiplication of two vectors, with the difference that it contains delay elements between the additions, indicated by the dots between the *Op ADD* nodes. Also, only one input is used which is broadcasted to all multipliers.

The automatically derived mapping is shown in Figure 11b.

```
fir c x = as                                    1
  where                                         2
    ms      = map (Op MUL x) c                  3
    as      = foldl dadd (Const 0) ms           4
    dadd a b = DELAYED ((Op ADD) a b)           5
```

Listing 1.4: Implementation of a FIR filter

5.3 Streaming pipeline

The final example is a simple streaming pipeline that uses the $\rightarrow$ notation. We use the previously defined FIR filter to stream an input through two chained FIR filters with two sets of filter coefficients *c1* and *c2*. The implementation of the pipeline is as follows:

stream $x = x \rightarrow$ (*fir c1*) $\rightarrow$ (*fir c2*)

The automatically derived mapping is shown in Figure 11c.

6 Conclusion and Outlook

A programming paradigm was developed in order to express instruction-level parallelism for coarse-grained reconfigurable arrays. The functional programming language Haskell was used as a base, as it inherently has a notion of structure and, thus, can easily express parallelism and the flow of data.

For our programming paradigm we adopted principles from dataflow and finite state machine (FSM) notations, which made it possible to express algorithms in the form of dataflow graphs with extended control. Furthermore, we considered a DSP application to be composed of two views: The local view, which represents everything that happens within one core, and the global view, which represents the flow of data through the array.

For the compiler, we implemented an embedded domain specific language (EDSL) as recursive datatype. In combination with higher order functions, this EDSL can be used to construct expressions that directly resemble the structure of a given problem. Consequently, the abstract syntax tree does not explicitly have to be extracted.

References

1. A. Yeung and J. Rabaey, "A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput dsp algorithms," in *System Sciences, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. 1. IEEE, 1993, pp. 169–178.
2. E. Mirsky and A. DeHon, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *FPGAs for Custom Computing Machines, Proceedings.* IEEE, 1996, pp. 157–166.
3. H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 465–481, 2000.
4. V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "Pact xpp—a self-reconfigurable data processing architecture," *the Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.
5. S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 75–85, 2008.
6. B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *Field-Programmable Logic and Applications.* Springer, 2003, pp. 61–70.
7. A. Niedermeier, J. Kuper, and G. Smit, "Dataflow-based reconfigurable architecture for streaming applications," in *System on Chip (SoC), 2012 International Symposium on.* IEEE, 2012, pp. 1–4.
8. E. Aarts and J. Korst, "Simulated annealing and boltzmann machines," 1988.