A DATAFLOW-INSPIRED CGRA FOR STREAMING APPLICATIONS

Anja Niedermeier, Jan Kuper, Gerard J.M. Smit

Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architecture Embedded Systems,
University of Twente, The Netherlands

1. MOTIVATION

The herein presented research is motivated by the need for reconfigurable, flexible computing arrays targeted at streaming applications that contain a large degree of instruction-level parallelism. Such arrays are usually referred to as coarse-grained reconfigurable arrays (CGRAs) [1]. CGRAs are composed of small, reconfigurable cores that are interconnected to form a computing grid.

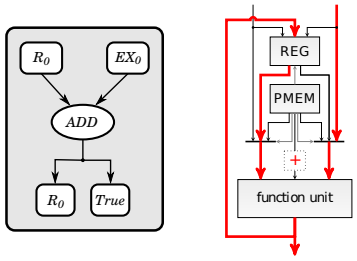
When reviewing related work, it became clear to us that an intuitive, easy-to-use programming paradigm was missing. The vast majority of the available CGRAs use a C-based programming approach. In our opinion however, C is not the most obvious choice to program such systems, since C has originally been designed as a sequential language, and therefore lacks support for structural description of parallelism. As a consequence, the burden of extracting the structure of an implemented algorithm lies on the compiler.

Here, we present a complete CGRA, consisting of an architecture and a programming language. Both the architecture and the programming language are inspired by the principles found in dataflow [2]. In particular, the firing rule, which means that a node fires as soon as all the required inputs are available. Furthermore, instead of developing a programming paradigm based on C, we decided to base our programming paradigm on the base of the functional programming language Haskell, that already has a notion of structure and the ability to describe instruction-level parallelism.

2. THE ARCHITECTURE

The architecture presented herein is composed of small, independent, reconfigurable cores. Each core includes a function unit, a register file and a program memory. The cores are interconnected as a mesh by nearest-neighbour links and a network on chip for full connectivity. The internal structure of one core is shown in Figure 1b, with an example configuration that will be explained in Section 3. For a detailed description of the architecture, the reader is referred to [3].

This research is conducted as part of the Sensor Technology Applied in Reconfigurable systems for sustainable Security (STARS) project. For further information: www.starsproject.nl



(a) Configuration (b) Core

Fig. 1: Programming Paradigm

3. PROGRAMMING PARADIGM

When developing our programming paradigm, we considered the compiler flow on two different views: The *local view*, which defines everything that stays local within a core, and the *global view*, which defines the communication between the cores in the array. Using that principle, the computation and the communication are decoupled from each other.

The *local view* is described best with an example, which is shown in Figure 1a. Figure 1b illustrates the execution of this example on a core. The configuration of a core is defined by the operation, the sources of the inputs, and the output management. The cores in our architecture support simple binary operations such as addition, multiplication, and the like. In the example, the operation is an addition. The inputs can either be an external input, a pointer to the local register file, or a constant value. In the example, the left input is the value stored in register R_0, the right input the external input EX_0. The output handling defines whether an output should be stored in the local register file, in the example R_0, and whether it should be available outside of the core, i.e. sent to the grid. In the example that is the case.

The *global view* represents the communication in the grid. As soon as an algorithm is spread over multiple cores, the cores communicate by exchanging data tokens. To each

978-1-4799-0004-6/13/$31.00 ©2013 IEEE

outgoing data token, an *identifier* of the destination core is attached. Routing hardware in the array then takes care of delivering the data token to the destination core.

4. THE COMPILER

The compiler for the presented architecture, using the proposed programming paradigm, was implemented as an embedded domain specific language (EDSL) in the functional programming language Haskell. It was implemented as a recursive datatype, which can then be used to recursively construct complex expressions.

In Listing 1, a minimalistic implementation of the language is shown. Line 1 defines a *constant*, line 2 a *delay element*, line 3 a *name of an input stream*, and line 4 an *operation*, defined by an *opcode* and two *expressions* as inputs. A selection of *opcodes* is shown in line 5.

$$
\begin{array}{ll}
\textbf{data}\ Expr = Const\ Number & 1 \\
\qquad\qquad \mid\ DELAYED\ Expr & 2 \\
\qquad\qquad \mid\ Input\ String & 3 \\
\qquad\qquad \mid\ Op\ OpCode\ Expr\ Expr & 4 \\
\textbf{data}\ OpCode = ADD \mid MUL \mid AND\ ... & 5
\end{array}
$$

Listing 1: recursive EDSL definition for an expression

5. DESIGN FLOW

In the following, we will demonstrate the design flow for implementing an algorithm on the architecture. As usecase, the example of a multiplication of two vectors x and y is shown.

The first step is the actual implementation of the algorithm using the EDSL presented in Listing 1. The corresponding code is shown in Listing 2.

$$
\begin{array}{ll}
vxv\ x\ y = out & 1 \\
\quad \textbf{where} & 2 \\
\qquad ms\ = zipWith\ (Op\ MUL)\ x \qquad y & 3 \\
\qquad out = foldl \quad (Op\ ADD)\ (Const\ 0)\ ms & 4
\end{array}
$$

Listing 2: Implementation of a multiplication of two vectors

For the implementation, two higher-order functions ($zipWith$ and $foldl$) are chosen. A graphical representation of vxv is shown in Figure 2.

At this moment, the desired algorithm is available as abstract syntax tree, but it cannot be executed yet on the architecture. For that, the compiler has to generate code in the form that was presented in Section 3. To do so, the compiler traverses through the expression tree and generates the corresponding code for each node (see Figure 3).

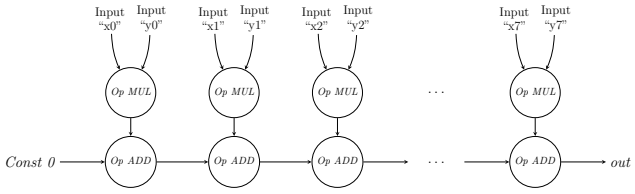


Fig. 2: Structure of vxv

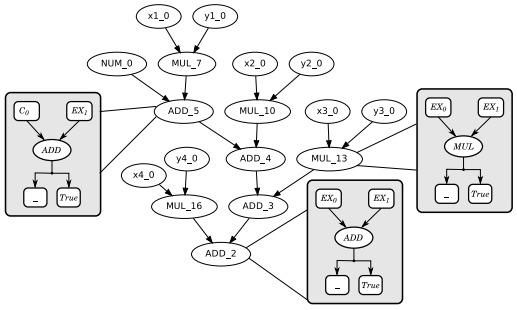


Fig. 3: Generated AST for $vxv\ x\ y$

At this moment, the *local view* has been generated for each of the nodes in the expression tree. As a next step, the *global view* is generated. That means, the compiler maps the expression tree to cores in the architecture. For this, we use a *simulated annealing* algorithm. Each node of the expression tree is mapped to a core in the architecture, and also the routing information is added to the programming of each core. Finally, a set of stimuli is generated and the implemented algorithm is simulated on the target architecture.

6. CONCLUDING REMARKS

In this abstract we presented a CGRA to implement and execute streaming applications that contain a large degree of instruction-level parallelism. As an illustrating example, we showed the implementation of a multiplication of two vectors. For the architecture as well as for the programming paradigm we borrowed principles of dataflow, in particular the firing rule, i.e. execution starts as soon as all the required inputs are available. The programming language was implemented as an EDSL in Haskell, which enables a designer to make use of Haskell's higher order functions.

7. REFERENCES

[1] B. Svensson *et al.*, "Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing," *Microprocessors and microsystems*, vol. 33, no. 3, pp. 161–178, 2009.

[2] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[3] 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.