Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

Deriving Stencil Hardware Accelerators from a Single Higher-Order Function

Rinse Wester and Jan Kuper

Computer Architecture for Embedded Systems, University of Twente, Enschede, Netherlands.

{R.Wester , J.Kuper} @utwente.nl

Abstract. Stencil computations are array based algorithms that apply a computation to all array elements in a fixed regular pattern and can be found in many scientific and engineering applications. Parallelization of these applications becomes more and more important in order to keep up with the demand for computing power. FPGAs offer a lot of computing power but are considered hard to program. In this paper, a design methodology based on transformations of higher-order functions is introduced to facilitate this parallelization process. Using this methodology, efficient FPGA hardware is derived achieving good performance. Two architectures for heat flow computations are synthesized for an FPGA and evaluated. To show the general applicability of the design methodology, several applications have been implemented.

Keywords. Stencil computations, Space/time tradeoff, Haskell, Higher-order function

Introduction

Stencil computations are array based algorithms where a function is applied to all elements in that array using neighbouring elements as well. These computations are largely independent of each other and are executed in a regular pattern. This allows them to be parallelized for parallel hardware like FPGAs. Stencil computations are used in many applications: FIR filtering, 1D convolution, 1D heat flow and convolution, and heat flow and cellular automata in 2D.

In this paper, we introduce a design methodology for deriving hardware for stencil computations. The methodology is based on rewriting higher-order functions (functions taking functions as argument) specifically for stencil computations. All transformations are therefore performed in the same language. The main contributions of this paper are two transformation steps to implement stencil computations on FPGAs. The first step allows the designer to make a tradeoff between execution time and area usage while the second step covers the implementation of actual hardware. As shown in Figure 1, the methodology starts with a single higher-order function (HOF) for stencil computations. Using the first transformation, computations are distributed over space and time ($\mathbb{S}$ and $\mathbb{T}$). Then, a second transformation is applied to derive hardware which can be simulated cycle accurately and implemented on an FPGA using CλaSH.

As opposed to most of the related work, we propose a more formal approach based on deriving hardware from one higher-order function using transformations. The work presented in this paper applies the transformation-based design methodology of [1] to problems with overlapping data patterns. The description of the hardware architecture, derived using this methodology, is a function representing a Mealy machine. The simulation of this Mealy machine is cycle accurate. The hardware resulting from this methodology is very similar to

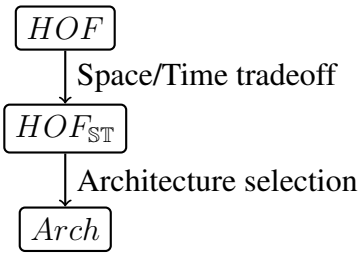

Figure 1. Derivation of hardware.

the related work in both structure and performance. However, the design methodology for deriving these architectures contains no imperative concepts like for-loops.

The rest of this paper is structured as follows. In Section 1, some background information is given on stencil computations and hardware design using a functional language. The design methodology is introduced in Section 2 while simulation and hardware results are given in Section 3. In Section 4, the results from the methodology are put in perspective by considering other implementations of stencil computation algorithms. Finally, in Section 5 conclusions are drawn and possible directions for future work are discussed.

1. Background

1.1. Stencil Computations

Stencil computations are common in signal processing and high-performance computing and use a fixed pattern of computation. Applications include filtering and numerical computations like computational fluid dynamics. A function, the kernel, is applied to every point in the domain with overlap by using surrounding elements. Since values are used for several computations, locality is very important. A lot of communication overhead can be prevented by keeping the data close to the stencil operation. Equation 1 and 3 show a formal definition of stencil computations.

$$Y_i = f([X_{i-M} \dots X_i \dots X_{i+M}]) \tag{1}$$

As shown in Equation 1 the stencil computation function f is given for a range of values from X. Since an additional M elements before and after X_i are supplied to f, the width of the whole window is $2M + 1$ elements. An example of a sliding window function is the moving average filter, often used to filter out noise. This filter determines the average value of a number of subsequent values. Equation 2 shows an average filter for 5 values.

$$Y_i = f([X_{i-2}, X_{i-1}, X_i, X_{i+1}, X_{i+2}]) = \frac{1}{5} \sum_{n=-2}^{2} X_{i+n} \tag{2}$$

Similarly, for 2D stencil computations the window has dimensions $(2M+1) \times (2N+1)$ as can be seen in Equation 3.

$$Y_{i,j} = f\left(\begin{bmatrix} X_{i-M,j+N} & \dots & X_{i+M,j+N} \\ \vdots & X_{ij} & \vdots \\ X_{i-M,j-N} & \dots & X_{i+M,j-N} \end{bmatrix}\right) \tag{3}$$

1.2. Hardware Design using CλaSH

In order to create actual hardware, we use the language CλaSH [2]. CλaSH is based on the functional language Haskell [3] which allows for easy, cycle accurate, simulation of digital hardware. CλaSH is a subset of Haskell, therefore, every CλaSH description is also a valid Haskell program. After simulation, the CλaSH compiler translates the description of the circuit to the hardware description language VHDL. This VHDL code can be synthesized for FPGA using industry standard tooling. A lot of advanced features from Haskell can also be used in the CλaSH description. Examples are polymorphism, higher-order functions, pattern matching and type derivation. Higher-order functions in particular are a powerful abstraction since they allow for reasoning about structure and parallelism of the hardware.

Every circuit in CλaSH is expressed as a Mealy machine, a formalism to describe the behavior of digital circuits. A Mealy machine describes the change of state and output during every clock cycle. Therefore, every output and new state is a function of the input combined with the current state. Listing 1 shows a simple example, a multiply accumulate (MAC). First, the name of the function to be defined is given (mac) followed by the current state (s) and the inputs (a, b). These are arguments of the the function mac and separated by spaces instead of commas. The result consists of the new state s' and the output out. Finally, all calculations are performed without any state changes, in the *where* clause (i.e., combinatorially). Figure 2 shows the hardware corresponding to the code of Listing 1.

Listing 1 Multiply Accumulate example in CλaSH.

$$
\begin{aligned}
&mac\ s\ (a, b) = (s', out)\\
&\quad \textbf{where}\\
&\qquad s' \quad = s + a * b\\
&\qquad out = s'
\end{aligned}
$$

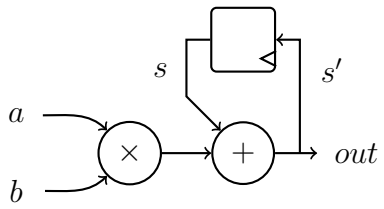

Figure 2. Multiply Accumulate structure.

As mentioned before, CλaSH supports higher-order functions, which are very useful to describe structure and parallelism. There are several ways to look at higher-order functions: a computational perspective and a structural one. The computational view is usually covered in teaching functional programming by considering the evaluation using recursion. A structural perspective, however, is much more applicable to hardware design with regular patterns [4,5]. Higher-order functions express the dependencies between components on the FPGA without time (referentially transparent) and are therefore a proper basis for applying transformations.

Higher-order functions are functions that can accept functions as argument or return a function as a result and are often used when processing lists. CλaSH, however, does not support lists because these may change in size during runtime which is not desirable for hardware. Therefore, CλaSH makes use of vectors (lists with constant length) and the corresponding higher-order functions like $vzipWith$ and $vfoldl$. Listing 2 shows a description of a finite

impulse response (FIR) filter using these higher-order functions. A FIR filter calculates the weighted sum of values in a similar way as a moving average filter.

Compared to the MAC example, the fir example of Listing 2 accepts an additional argument cs, a vector of filter coefficients. The registers of the filter, the input and output are called us, inp and out respectively. The new state of the registers us' is the original state us with the input inp shifted in one position using the $+\!\!\gg$ operator. $vzipWith$ multiplies the coefficients cs with the contents of the registers us in a pair-wise fashion. Finally, the last line shows the use of $vfoldl$ which accepts $+$ as a functional argument and therefore adds all ws together. In other functional languages, $vfoldl$ is sometimes called $reduce$ since a list of values is reduced to a single element by accumulating values using some binary function.

Listing 2 Higher order functions in a FIR filter described using CλaSH.

$$
\begin{aligned}
&fir\ cs\ us\ inp = (us', out)\\
&\quad \textbf{where}\\
&\qquad us' = inp{+\!\!\gg}\ us\\
&\qquad ws = vzipWith\ (*)\ us\ cs\\
&\qquad out = vfoldl\ (+)\ 0\ ws
\end{aligned}
$$

It is important to note that the description in the *where* clause only expresses data dependencies among components and no sequential ordering over time. Therefore, the description is implicitly parallel and there is no need for special notation (i.e., every function in the code becomes a component on the FPGA). Also, $vzipWith$ is implicitly parallel, it applies a function pairwise to the elements of the two lists us and cs. The schematic corresponding with Listing 2 is shown in Figure 3.

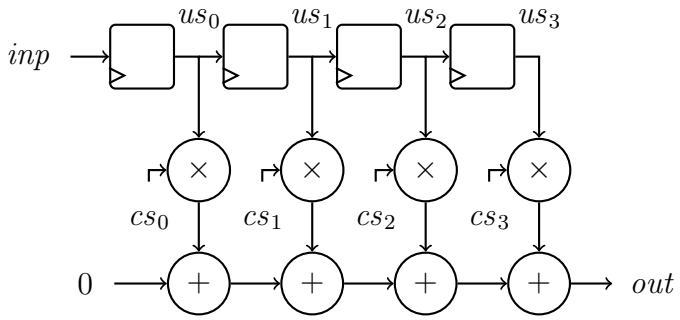

Figure 3. FIR filter structure.

2. Derivation Method

All calculations required for a stencil computation can be expressed using a single higher-order function. This higher-order function expresses all necessary calculations and dependencies among them and can be represented by a graph. The CλaSH compiler performs a one-to-one translation of this graph to components on the FPGA. However, any non-trivial sized stencil computation will result in an enormous resource usage which is why the amount of parallelism has to be limited. Therefore, a tradeoff between parallelism and execution time is required.

In this paper, the derivation of hardware from a higher-order stencil function is performed in two steps. First, a transformation rule distributes the stencil computations over two

domains; a part over space by parallelization and a part over time by sequential execution of partial stencil computations. Adding synchronization after the first transformation results in a description that can easily be translated to hardware. However, this results in a lot of communication overhead which is why a second step is needed to derive efficient hardware. This second step consists of transforming the description to increase data reuse.

2.1. Space/Time Transformation

The design methodology starts with a definition of the stencil computations in Haskell. Stencil computations are performed by the *stencil* function which takes three arguments; a kernel function f, a window width w and a list of inputs xs. The first three lines of Listing 3 show the definition and implementation of a one-dimensional moving average filter being applied to xs. The remainder of Listing 3 gives the Haskell implementation of *stencil*. *stencil* is expressed recursively, where the stencil kernel function f is applied to the beginning of the list. Then, *stencil* is again applied to the list excluding the first element. This process continues until the number of elements left becomes smaller than the window width. In the remainder of this paper, the implementation of *stencil* is not that important anymore since it is considered a native function.

Listing 3 Definition of average filter and stencil higher-order function.

$$
\begin{aligned}
&res = stencil\ avg\ 3\ xs \\
&\quad \textbf{where} \\
&\qquad avg\ xs = \tfrac{1}{3} * sum\ xs \\
&stencil\ f\ w\ xs \\
&\quad \mid length\ xs \geqslant w = \\
&\qquad (f\ (take\ w\ xs)) : (stencil\ f\ w\ (tail\ xs)) \\
&\quad \mid otherwise = [\,]
\end{aligned}
$$

Performing all operations in the stencil computation in a single clock cycle would require a lot of area on an FPGA when synthesized directly. Therefore, a tradeoff between area and execution time is required as previously proposed in [1]. This tradeoff is found by applying a transformation rule to *stencil*. Figure 4 shows the dependency graph of *stencil* where the function f is applied to every sublist of xs with N elements including overlap. Note, that the resulting list is smaller such that corner conditions do not occur (i.e., the result has $2M$ fewer elements: the size of the overlap is M elements).

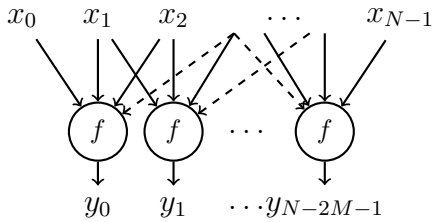

Figure 4. Structure of *stencil*.

In order to save FPGA resources, the kernel functions f are distributed over space $\mathbb{S}$ and time $\mathbb{T}$ by applying the transformation of *stencil* as shown in Listing 4. After this transformation, $stencil_{\mathbb{ST}}$ accepts four parameters: a parallelization factor p, kernel function f, stencil width w and the list of inputs xs. The input data is split into smaller lists using the *split*

function taking into account the overlap between them. These sublists are then processed sequentially (mapped over time), by the $map_{\mathbb{T}}$ function. This function applies the *stencil* function to a complete sublist in one clock cycle using a single (native) combinatorial component $stencil_{\mathbb{S}}$. The amount of parallelism is therefore determined by the size of each sublist which is a parameter for *split* (p). This transformation based method therefore does not need loop unrolling or dependency analysis of *for* loops.

Listing 4 Tradeoff rule for *stencil*.

$$
\begin{aligned}
&stencil_{\mathbb{ST}}\ p\ f\ w\ xs = concat_{\mathbb{T}}\ ress \\
&\quad \textbf{where} \\
&\qquad xss\ = split_{\mathbb{T}}\ w\ p\ xs \\
&\qquad ress = map_{\mathbb{T}}\ (stencil_{\mathbb{S}}\ f\ w)\ xss
\end{aligned}
$$

A similar tradeoff between time and space has been derived for 2D stencil computations. Again, the tradeoff rule ensures that all input data is divided into smaller slices which are processed sequentially. Listing 5 shows the tradeoff rule for 2D higher-order stencil function *stencil2d*. Note the introduction of an additional parallelization parameter vp due to the additional spatial dimension. Similar to $stencil_{\mathbb{ST}}$, $stencil2d_{\mathbb{ST}}$ accepts four arguments as well: a tuple with the horizontal and vertical parallelization factors (hp, vp), the stencil function f, a tuple with stencil width and height (w, h) and the two-dimensional input data img.

Listing 5 Tradeoff rule for *stencil2d*.

$$
\begin{aligned}
&stencil2d_{\mathbb{ST}}\ (hp, vp)\ f\ (w, h)\ img = concat2d_{\mathbb{T}}\ img' \\
&\quad \textbf{where} \\
&\qquad imgss = split2d_{\mathbb{T}}\ (w, h)\ (hp, vp)\ img \\
&\qquad ress\ \ = map2d_{\mathbb{T}}\ (stencil2d_{\mathbb{S}}\ (w, h)\ f)\ imgss
\end{aligned}
$$

$stencil_{\mathbb{ST}}$ could be directly translated to hardware by adding dataflow logic to $stencil_{\mathbb{S}}$ for easy synchronization. However, due to overlap between sublists, additional bandwidth is required to keep the $stencil_{\mathbb{S}}$ component utilized. Therefore, the next step is to rewrite $stencil_{\mathbb{ST}}$ to an architecture that buffers data for reuse, thereby minimizing communication.

2.2. Deriving the Architecture

In order to reduce communication overhead, data must be kept as close to the computation as possible. This is usually implemented by buffering data from previous cycles [6,7,8]. The architecture derived in the second step of the designer methodology should therefore take this buffering into account as well. Deriving the architecture is performed by wrapping $stencil_{\mathbb{S}}$ of Listing 4 into an architecture to handle communication and synchronization.

Figure 5 shows a stencil computation architecture with parallelization $p = 2$. Elements of p samples are accepted each clock cycle and processed by $stencil_{\mathbb{S}}$. These elements are shifted into a shift register such that they can be used in subsequent clock cycles. A shift register is a memory structure for lists where a single element is inserted at one side while removing an element at the other during every clock cycle. The actual stencil computation is implemented combinatorially by $stencil_{\mathbb{S}}$.

Listing 6 shows the code for the one-dimensional stencil computation architecture of Figure 5 in the form of a function named *stencilarch*. *stencilarch* accepts three arguments:

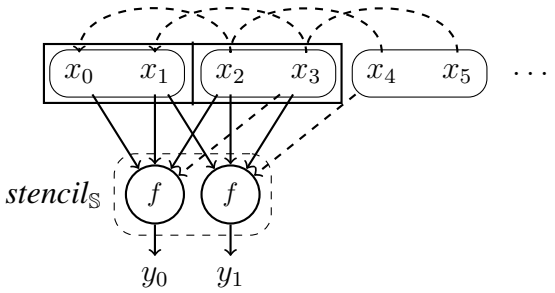

Figure 5. Stencil computation architecture with $p = 2$.

the kernel function f, the current state of the shift register xs and the new input sample(s) inp (shown on line 2 while line 1 gives the type). The result tuple consists of two parts: the new state of the shift register xs' and the computed output $outp$. xs' is found by shifting the input list inp (with p samples) completely into xs such that the last p samples are dropped off. Finally, the output $outp$ is found by applying the kernel function f to all buffered input samples xs. Also, the type of $outp$ is a list containing p elements. *stencilarch* is a higher-order function since it can be parameterized with a specific kernel function f. However, it now represents an actual architecture in the form of a Mealy machine instead of an abstract mathematical function.

Listing 6 One-dimensional stencil architecture.

```
stencilarch :: ([a] → b) → [a] → [a] → ([a], [b])
stencilarch f xs inp = (xs', outp)
  where
    xs'  = inp+⋙ xs
    outp = stencil_S f
```

A similar architecture is derived for 2D stencil computations. The architecture consists of three parts: a window buffer for holding data to which the kernel function f is applied, line buffers for storing complete lines of the input and a part where the actual output is calculated. 2D stencil computations can be parallelized by processing several elements at once. Given a parallelization of $p = 2$, an architecture as shown in Figure 6 is derived.

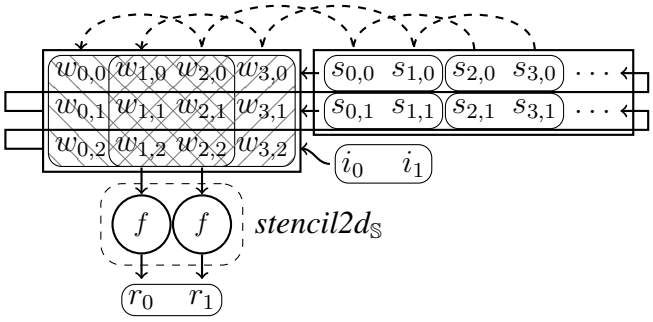

Figure 6. 2D stencil computation architecture with $p = 2$.

As shown in Figure 6 the $stencil2d_{\mathbb{S}}$ is the only part in the architecture where computations are performed. All other parts are used for buffering. Since $p = 2$, two elements (i_0 and i_1) of the input data are sent to the architecture every cycle. This also means that all buffers forward the data in packets of two. Also $stencil2d_{\mathbb{S}}$ processes two stencils at once resulting

in two output samples (r_0 and r_1) being produced at the same time. The code to implement and simulate this architecture is shown in Listing 7.

Listing 7 Two-dimensional stencil computation architecture.

$$
\begin{aligned}
&stencilarch2d :: (Img\ a \to b) \to (Img\ a, Img\ a) \to Img\ a \to ((Img\ a, Img\ a), Img\ b)\\
&stencilarch2d\ f\ (ws, ss)\ inp = ((ws', ss'), outp)\\
&\quad \textbf{where}\\
&\qquad outp = stencil2d_{\mathbb{S}}\ f\ ws\\
&\qquad ws' \ = mergeh\ (droph\ 2\ ws)\ (mergev\ (takeh\ 2\ ss)\ inp)\\
&\qquad ss' \ = mergeh\ (droph\ 2\ ss)\ (takeh\ 2\ (dropv\ 1\ ws))
\end{aligned}
$$

Listing 7 shows the Haskell implementation of the 2D stencil computation architecture. As shown by the type, the first argument f is a function that takes an image of type a and produces an element of type b. The second argument represents the state consisting of two buffers while the third argument is part of the input image with type $Img\ a$. For completion of the Mealy machine structure, the result consists of the new state of the buffers and the actual result. The first line of the where clause implements the parallel processing of the window buffer ws. The new state of the window and line buffers (ws' and ss') are found by shifting in and out pairs of elements using the functions for merging and slicing images (*mergeh*, *mergev*, *droph*, *dropv* and *takev*).

In order to generate actual hardware, the Haskell descriptions of *stencilarch* and *stencilarch2d* have to be altered slightly such that it is accepted by the CλaSH compiler. Lists are not supported by the CλaSH compiler which is why they are replaced by vectors. A small library with higher-order functions specifically for vectors has been developed such that the code can be compiled by the CλaSH compiler more or less unchanged. More details on altering code using lists to vectors can be found in [9].

2.2.1. Implementing Stencil Applications

To implement an actual application, the higher-order function describing the architectures has to be supplied with an argument: the application specific kernel function. In order to show the general applicability of the methodology and architecture, several stencil applications have been implemented. The applications covered in this paper are one and two-dimensional heat flow and a cellular automaton.

1D heat flow. As one of the applications, we have chosen heat flow simulation since it is very computationally intensive. The changes of temperature of an object are governed by a partial differential equation which is generally not algebraically solvable. Therefore, a finite difference method is used where the problem is discretized in sufficiently small steps in both space and time. For every point in the spatial domain (a point on a metal plate for example), the next temperature is a linear combination of current temperature and directly surrounding temperatures. For the 1D heat flow problem, only the point left and right of current position are needed while for 2D heat flow, the temperatures above and below are also needed. Since the computation is the same for every point, the heat flow problem fits the stencil computation paradigm very intuitively. Equation 4 and Equation 5 show how the temperature for a single point evolves over time for the 1D and 2D heat flow problem respectively.

$$T_{k+1,i} = T_{k,i} + c \times (T_{k,i-1} - 2T_{k,i} + T_{k,i+1}) \tag{4}$$

As can be seen in Equation 4, the new temperature $T_{k+1,i}$ depends on the current temperature $T_{k,i}$ and its neighbours $T_{k,i-1}$ and $T_{k,i+1}$. This elementary update functions therefore has

a window width of 3 elements. Translating this to a window function for the aforementioned architecture is now straightforward as can be seen in the Haskell code of Listing 8. This heat flow kernel *hfk* accepts a window containing three elements and returns a linear combination as new temperature.

Listing 8 One-dimensional heat flow kernel.

$$hfk\ [x0, x1, x2] = x1 + c * (x0 - 2 * x1 + x2)$$

2D heat flow. For the 2D heat flow problem, the kernel is very similar. The only difference is that the temperatures above and below the current point are used in the linear combination as well. This is shown in Equation 5 where the new temperature $T_{k+1,i,j}$ for time $k+1$ and position i, j depends on the current and surrounding temperatures.

$$T_{k+1,i,j} = T_{k,i,j} + c \times (T_{k,i-1,j} - 4T_{k,i,j} + T_{k,i+1,j} + T_{k,i,j+1} + T_{k,i,j-1}) \tag{5}$$

The kernel for the 2D heat flow problem accepts a window xs of size 3×3. Again, the next temperature T_{k+1} is found by a linear combination of the current and surrounding temperatures. Distinct elements from the window are selected using the Haskell index operator !!. The Haskell code for the 2D heat flow kernel *hfk2d* is shown in Listing 9.

Listing 9 Two-dimensional heat flow kernel.

```
hfk2d xss = x11 + c * (-4 * x11 + x01 + x12 + x21 + x10)
  where
    (x01, x21)      = (xss !! 0 !! 1, xss !! 2 !! 1)
    (x10, x11, x12) = (xss !! 1 !! 0, xss !! 1 !! 1, xss !! 1 !! 2)
```

Cellular automata. The same approach has been used to implement the cellular automaton rule 110 as described in [10]. Cellular automaton rule 110 is a 1D stencil computation with cells that can be in only two states (0 or 1). The state of a cell of the next row is determined by a very simple pattern based only on state of the current cell and the direct left and right neighbour. Equation 6 shows the mathematical definition of rule 110.

$$C^i_{k+1} = \begin{cases} 1 & \text{if } C_{k,i+\{-1..1\}} \in \{110, 101, 011, 010, 001\} \\ 0 & \text{if } C_{k,i+\{-1..1\}} \in \{111, 100, 000\} \end{cases} \tag{6}$$

For simulation, Equation 6 is trivially translated to Haskell. Using *rule110* as argument for the general 1D stencil computation architecture *stencilarch* (Listing 6), a rule 110 specific Mealy machine is derived. The resulting architecture is now ready for implementation on FPGA using CλaSH.

Other applications. This paper covers only three applications. However, using the higher-order function *stencil*, representing other applications is equally concise. Examples are image processing algorithms like convolution and median filtering and Game of Life.

3. Results

Several stencil computation applications have been implemented using the *stencil* higher-order function: one-dimensional heat flow, elementary cellular automata and two-dimensional

Listing 10 Elementary cellular automaton rule 110.

$$
\begin{aligned}
&\mathit{rule110}\ [x1, x2, x3] = \\
&\quad \textbf{if}\ [x1, x2, x3] \in [[1,1,0],[1,0,1],[0,1,1],[0,1,0],[0,0,1]] \\
&\quad \textbf{then}\ 1 \\
&\quad \textbf{else}\ \ 0
\end{aligned}
$$

heat flow. The rest of this section focuses on simulation results and generation of hardware. First, simulation results are presented for 1D, 2D heat flow and rule 110 after which the corresponding hardware is discussed.

3.1. Simulation

The architecture for 1D heat flow has been simulated resulting in temperature changes as shown in Figure 7. The vertical axis represents the position while the horizontal axis represents time. Over time, the hot spot in the middle evens out over the whole domain. No additional heat is supplied on the edges which is why the whole domain will eventually converge to the single temperature of the edges.

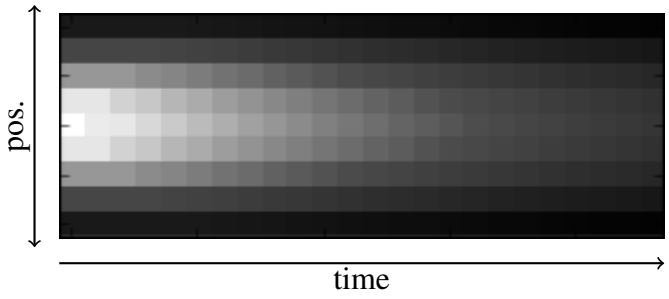

Figure 7. Simulation result of heat flow kernel.

Similarly, the architecture for 2D heat flow has been simulated as well. The architecture is initialized with an image representing the initial temperatures ($t = 0$). The initial image has a hot spot in the top left part. Over time, the heat in this spot evens out ($t = 32$) and eventually disappears ($t = 256$).

Different temperatures are supplied to the borders. The upper and right border are heated while the others kept at a low temperature. Slowly, the heat from the upper and right border propagates deeper into the material ($t = 256$).

Also the architecture for cellular automaton rule 110 has been simulated. The result is shown in Figure 9. The initial row contains only one cell with value 1. Every consecutive row is determined by applying rule 110 to the current row.

3.2. Synthesis

Both the 1D and 2D architectures for heat flow simulation have been translated to VHDL using the CλaSH compiler to derive actual hardware. These have been instantiated with different parallelization factors (PF) to show the scaling on FPGA hardware. All designs have been synthesized for a Xilinx XC5VLX110T FPGA and were all capable of running at 200 MHz. Table 1 shows the amount of resources required for computation (LUT and DSP48E multipliers) and storage (REGs).

As shown in Table 1, both the 1D and 2D architectures scale linearly with parallelization factor PF in terms of LUTs. The amount of registers for the 1D architecture scales linearly as

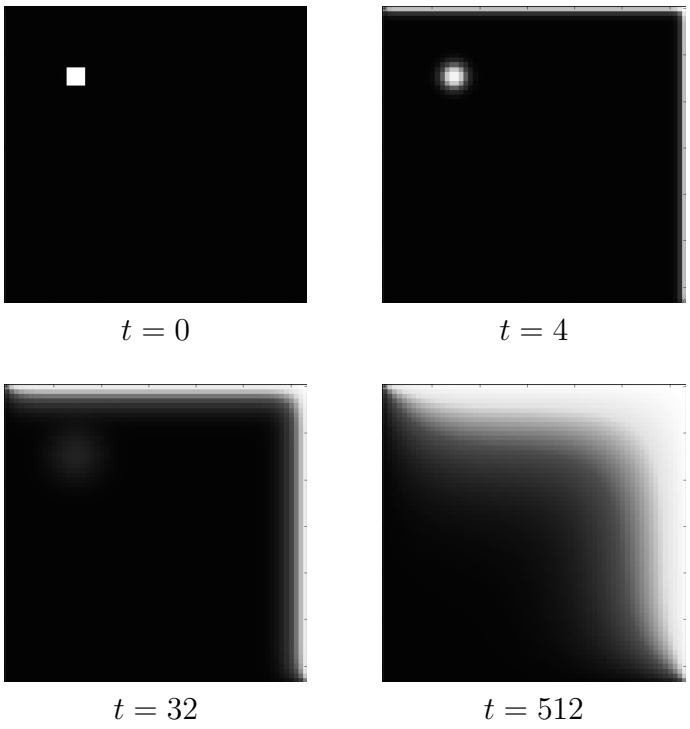

Figure 8. Simulation of 2D heat flow.

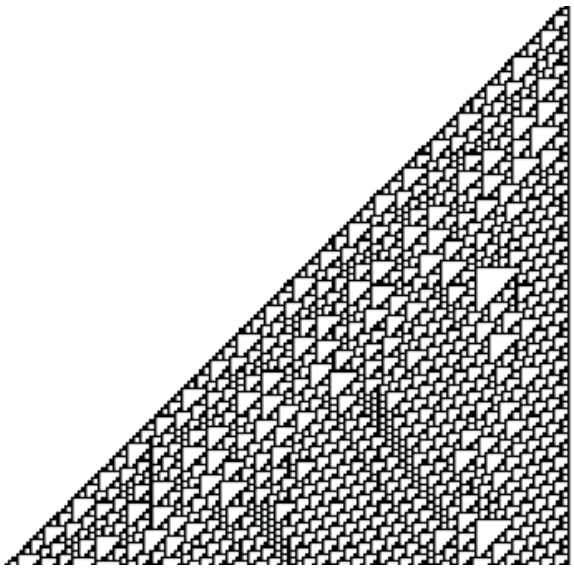

Figure 9. Simulation result of cellular automaton rule 110.

well but for the 2D architecture the amount of registers is practically constant. This is because most registers are used for the line buffers. Compared to related work of [11], a similar 2D architecture is derived with very similar resource consumption even though a different kernel is used. Both the amount of LUTs required and clock frequency is comparable to the design in [11].

4. Related Work

Related work on higher-order functions for hardware design and stencil computations are covered in [5]. In [12] the PASTHA framework is presented, a framework for parallelization of stencil computations on multicore machines. Similar to the VHDL hardware template of

Table 1. Area of stencil computation architectures.

	PF	LUTs	REGs	DSP48E
1D	1	72	72	1
	2	143	96	2
	4	285	144	4
	8	569	240	8
	16	1137	432	16
	32	2273	816	32
2D	1	116	12336	1
	2	231	12360	2
	4	461	12408	4
	8	961	12504	8
	16	1921	12696	16
	32	3841	13080	32

[11], the hardware resulting from the transformations presented in this paper is parameterizable in the amount of parallelism taking into account memory structure and communication like in [6] and [8]. Memory and communication aspects impose constraints on the FPGA implementation as shown in [13], [14] and [15]. Especially in designs with multiple FPGAs, communication patterns become very important to achieve high performance [16].

Most design methods for stencil computation use, at some stage, an imperative description (in C for example) of the operations. Listing 11 shows the pseudocode of such an imperative description. For every point at x, y in a frame v at time t, the new value $v(t+1, x, y)$ is calculated using the stencil function F. Although this description is very similar to the formal definition in Equation 3, the code is inherently sequential. Therefore, a lot of analysis is required to determine the dependencies between loop iterations before parallelization can be performed. Also details of intermediate stages are often hidden or hard to modify. Therefore, the description of the generated hardware looks very different compared to the initial definition. By using the methodology proposed in this paper, the intermediate results following the transformation are more accessible.

Listing 11 Imperative code for stencil computations.

```
for (x = 0; x < WIDTH; x++) {
  for (y = 0; y < HEIGHT; y++) {
    v (t + 1, x, y) = F (w in stencil (v, x, y));
  }
}
```

An other approach to implementing stencil computations on FPGAs is the use of a Domain Specific Language (DSL) [7]. A compiler takes care of parallelization and scheduling. For parallelization of stencil computations the approach taken in this paper is similar to the parallelization of FIR filters in [17] but requires no array index computations. Since the methodology makes use of higher-order functions, it is convenient to use a hardware description language with higher-order function support built in. Therefore, the CλaSH language [2] is used. Other work on translating Haskell to hardware is the famous Lava [18] and the more recent Kansas Lava [19]. The main difference between Lava and CλaSH is that Lava is a language embedded in Haskell while the CλaSH compiler takes plain Haskell code as input and thereby supporting more language features.

5. Conclusions and Future Work

A design methodology for stencil computation applications has been presented based on transformations of a higher-order function. The advantage of this methodology is that the higher-order function *stencil* is inherently parallel. Therefore, sequential transformations like loop-unrolling are not necessary thereby reducing the chance of off-by-one errors.

Several applications have been implemented using this approach including heat flow and cellular automata. These applications have been simulated cycle-accurately to verify their functionality. For the 1 and 2-dimensional heat flow application, real hardware has been designed using CλaSH. This hardware has shown to scale linearly with the parallelization parameter. Also resource consumption is comparable with related work.

Currently, only 1D and 2-Dimensional stencil applications have been considered. In the future, it would be interesting to apply the approach to applications in higher dimensions since this puts more stress on resource usage and buffering. The transformation rule should therefore be altered taking these constraints into account.

Acknowledgements

This research is conducted as part of the Sensor Technology Applied in Reconfigurable systems for sustainable Security (STARS) project www.starsproject.nl and FP7 project Programming Large Scale Heterogeneous Infrastructures (POLCA), grant agreement 610686.

References

[1] R. Wester and J. Kuper. Design space exploration of a particle filter using higher-0rder functions. *Reconfigurable Computing: Architectures, Tools, and Applications*, 8405:219–226, 2014.

[2] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Lille, France*, pages 714–721, USA, September 2010. IEEE Computer Society.

[3] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*, volume 13 of *Journal of Functional Programming*. 2003.

[4] Mary Sheeran. Hardware design and functional programming: a perfect match. 11(7):1135–1158, jul 2005.

[5] M. E. T. Gerards, C. P. R. Baaij, J. Kuper, and M. Kooijman. Higher-Order Abstraction in Hardware Descriptions with CλaSH. In P. Kitsos, editor, *Proceedings of the 14th EUROMICRO Conference on Digital System Design, DSD 2011, Oulu, Finland*, pages 495–502, USA, August 2011. IEEE Computer Society.

[6] Yazhuo Dong, Yong Dou, and Jie Zhou. Optimized Generation of Memory Structure in Compiling Window Operations Onto Reconfigurable Hardware. In *Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'07, pages 110–121, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Wang Luzhou, Kentaro Sano, and Satoru Yamamoto. Domain-Specific Language and Compiler for Stencil Computation on FPGA-Based Systolic Computational-memory Array. In *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'12, pages 26–39, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Haiqian Yu and M. Leeser. Automatic Sliding Window Operation Optimization for FPGA-Based Computing Boards. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 76–88, April 2006.

[9] R. Wester, C. P. R. Baaij, and J. Kuper. A two step hardware design method using CλaSH. In *22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Oslo, Norway*, pages 181–188, USA, August 2012. IEEE Computer Society.

[10] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, January 2002.

[11] M. Schmidt, M. Reichenbach, and D. Fey. A Generic VHDL Template for 2D Stencil Code Applications on FPGAs. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*, pages 180–187, April 2012.

[12] Michael Lesniak. Pastha: Parallelizing stencil calculations in haskell. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[13] Zhi Guo, Betul Buyukkurt, and Walid Najjar. Input Data Reuse in Compiling Window Operations Onto Reconfigurable Hardware. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 249–256, New York, NY, USA, 2004. ACM.

[14] B.A. Draper, J.R. Beveridge, A. P W Bohm, C. Ross, and M. Chawathe. Accelerated image processing on FPGAs. *Image Processing, IEEE Transactions on*, 12(12):1543–1551, Dec 2003.

[15] C. T. Johnston, K. T. Gribbon, and D. G. Bailey. Implementing Image Processing Algorithms on FPGAs. In *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon04, Palmerston North*, pages 118–123, 2004.

[16] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):695–705, March 2014.

[17] Keshab K Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[18] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM.

[19] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing kansas lava. In MarcoT. Morazn and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer Berlin Heidelberg, 2010.