SODA: Stencil with Optimized Dataflow Architecture

Yuze Chi, Jason Cong, Peng Wei, Peipei Zhou
University of California, Los Angeles
{chiyuze,cong,peng.wei.prc,memoryzpp}@cs.ucla.edu

ABSTRACT

Stencil computation is one of the most important kernels in many application domains such as image processing, solving partial differential equations, and cellular automata. Many of the stencil kernels are complex, usually consist of multiple stages or iterations, and are often computation-bounded. Such kernels are often offloaded to FPGAs to take advantages of the efficiency of dedicated hardware. However, implementing such complex kernels efficiently is not trivial, due to complicated data dependencies, difficulties of programming FPGAs with RTL, as well as large design space.

In this paper we present SODA, an automated framework for implementing Stencil algorithms with Optimized Dataflow Architecture on FPGAs. The SODA microarchitecture minimizes the on-chip reuse buffer size required by full data reuse and provides flexible and scalable fine-grained parallelism. The SODA automation framework takes high-level user input and generates efficient, high-frequency dataflow implementation. This significantly reduces the difficulty of programming FPGAs efficiently for stencil algorithms. The SODA design-space exploration framework models the resource constraints and searches for the performance-optimized configuration with accurate models for post-synthesis resource utilization and on-board execution throughput. Experimental results from on-board execution using a wide range of benchmarks show up to 3.28x speed up over 24-thread CPU and our fully automated framework achieves better performance compared with manually designed state-of-the-art FPGA accelerators.

1 INTRODUCTION

Stencil computation [8] is an important kernel extensively used in many areas such as image processing [1], solving partial differential equations [22], and cellular automata [31]. In these application domains, computation time is often dominated by the stencil kernel, which makes it desirable for acceleration and optimization. For general-purpose processors like CPUs and GPUs, efforts have been made to improve locality [15, 27], increase parallelism [13, 14, 16], and reduce communication [30].

In another dimension, researchers and the industry have been using FPGA accelerators to achieve better energy efficiency on stencil computation kernels [3, 7, 9, 17, 21, 23, 24, 28, 34]. For some stencil kernels, the operational intensity [29], which is defined as the number of operations per input data, is relatively small. Thus, many existing studies focus on data reuse [3, 18, 26], which effectively reduces the external memory access by using line buffers for repeated accesses. For iterative stencil kernels, which are usually computation-intensive, researchers have been implementing multiple iterations to increase the parallelism [17, 34]. Other optimizations—for example, loop pipelining, processing element (PE) replication, and double buffering—are also common techniques to improve accelerator throughput. Furthermore, there are studies trying to distribute the workload to multiple FPGA accelerators [17, 23].

However, there are still two major challenges that have not been addressed thoroughly in state of the art. The first challenge is that existing accelerator designs are suboptimal when multiple PEs are used for a single stage. Existing stencil accelerators [28, 34] replicate the on-chip buffers along with the PEs to enable concurrent accesses. With a buffer size proportional to the number of PEs, both the maximum achievable number of processing elements (PEs) and the maximum achievable input tile size are suboptimal. Suboptimal number of PEs will under-utilize the computation resources and therefore results in suboptimal performance. Suboptimal input size causes performance loss due to the fact that the borders (halos) of stencil kernels need to be retransmitted. The latter problem is especially severe for 3D or even higher dimensional kernels, since their halos take larger portion of the input size [34]. When temporal parallelism (multiple iterations) are implemented at the same time, this becomes even worse since the halo size increases linearly as the number of iteration increases [34].

The other challenge that hasn't been thoroughly addressed is the lack of complete automation and systematic design-space exploration. Due to the difficulty of programming FPGAs, vast design space, and high time-consumption of logic synthesis, having a fully automated design flow and analytical model-based design-space exploration is crucial. Many existing work are either manually designed or template-based, which lacks the flexibility of designing various stencil kernels agilely [28, 34]. Although domain-specific languages (DSLs) has been developed to facilitate stencil accelerator design on FPGAs [11, 19, 21], there still lacks a systematic approach to model the resource and performance, explore the design space, and optimize for performance.

In this paper we present SODA, i.e., Stencil with Optimized Dataflow Architecture, to address both challenges. To address the first challenge, we present an optimal microarchitecture for stencil kernels in Section 3.2, which minimizes the number of external data transfer as well as the reuse buffer size required by a certain number of PEs. We then mathematically prove its optimality in Section 3.3. To implement the SODA microarchitecture efficiently, we apply dataflow optimization to obtain high-frequency, modularized FPGA accelerators, which is discussed in Section 3.4. To address the second challenge, we present a programming model and the corresponding automation framework in Section 4.1, which fully automate and simplify accelerator design for SODA. To enable fast and systematic design-space exploration, we develop a resource model that predicts the post-synthesis resource utilization in Section 4.3 and a performance model that predicts the on-board execution performance in Section 4.4. With these models, we are able to obtain the performance-optimized configuration under the platform resource constraint in just a few minutes.

In summary, our major contributions include:

- **Optimal microarchitecture:** Given a single stage of stencil kernel and the size of the input, the SODA microarchitecture requires the *least number of external data transfer*. This maximizes the external memory utilization. On the basis of this, given the number of PEs, SODA also achieves the *smallest reuse buffer size*. This minimizes on-chip resource consumption under performance requirement constraint.

ICCAD'18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240850

- **Design automation:** We develop the fully automated SODA framework, which automatically generates the SODA microarchitecture for FPGAs from a high-level DSL. The SODA DSL concisely describes the algorithm as well as the design parameters. The SODA automation framework takes the DSL as input, processes the dependency graph, implements the specified kernel with dataflow optimizations, and generates Xilinx OpenCL code for both host and kernel. The user can directly invoke the kernel as a C++ function to use SODA in their own programs.
- **Model-driven exploration:** The modularized dataflow implementation enables accurate resource and performance modeling. We propose accurate models for both resource and performance to enable fast and efficient design-space exploration. To make the models practical, we target post-synthesis resource utilization and on-board execution performance. With these models, the SODA DSE framework can automatically search for the performance-optimized configuration.

2 BACKGROUND AND RELATED WORK

2.1 Stencil Computation

Stencil computations can be intuitively defined as kernels which update data elements over a multidimensional array according to some fixed, local pattern. In practice, the array is often too large to be stored on-chip. Listing 1 shows a 5-point, 2-dimensional Jacobi kernel on an $M \times N$ input as an example. Figure 1 shows one possible iteration pattern and the input data elements it accesses when producing the output for (i, j).

```
void blur(float input[N][M], float output[N][M])
{
    for(int j = 1; j < N-1; ++j)
        for(int i = 1; i < M-1; ++i)
            output[j][i] = (  input[j-1][i]+
                input[j][i-1]+input[j  ][i]+
                input[j][i+1]+input[j+1][i])*0.2f;
}
```

Listing 1: A 5-point 2-dimensional Jacobi kernel.

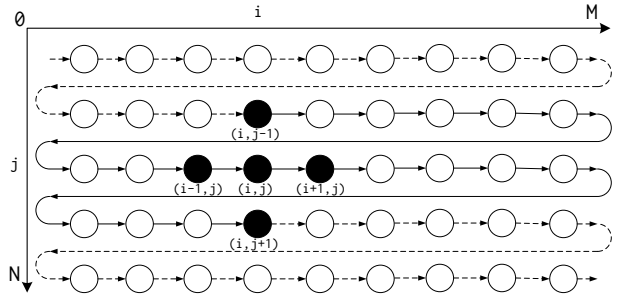

Figure 1: Stencil access pattern of the example in Listing 1.

The *operational intensity*, which is defined as the number of operations per input data, of the example in Listing 1 is relatively low, which makes such application communication-bounded. In practice, stencil kernels are often complex. Some stencil computations consist of multiple stages, where each stage is a simple stencil kernel. Some stencil computations are executed repeatedly over time, where each iteration in time can be treated as a stage connected directly with the previous stage. If the number of stages or iterations is sufficiently large, the operational intensity will be sufficiently high and the application will become computation-bounded (resource-bounded on FPGAs).

2.2 Definitions and Problem Formulation

Stencil Kernel [4]: An n-point, m-dimensional stencil kernel A defines a spatial window $\{\vec{a}^{(s)} | s \in \{0, 1, 2, \ldots, n-1\}\}$ and a function which produces output at spatial coordinate $\vec{y}$ where

$$\vec{y} = (y_0, y_1, y_2, \ldots, y_{m-1})$$

by consuming inputs at spatial coordinates

$$\{\vec{x}^{(s)} | s \in \{0, 1, 2, \ldots, n-1\}\} = \{\vec{y} + \vec{a}^{(s)} | s \in \{0, 1, 2, \ldots, n-1\}\}$$

$\vec{a}^{(s)}$ denotes the offset between the s-th input and the output. For convenience sake, we use the spatial coordinate to represent the data element at that position in this paper.

$\{\vec{a}^{(s)} | s \in \{0, 1, 2, \ldots, n-1\}\}$ is defined as the *stencil window*. The *stencil window size* in dimension d, S_d, is defined as

$$S_d = \max_s \left(a_d^{(s)}\right) - \min_s \left(a_d^{(s)}\right) + 1$$

In our 5-point 2-dimensional example in Listing 1,

$$\{\vec{a}^{(s)}\} = \{(0, -1), (-1, 0), (0, 0), (1, 0), (0, 1)\}, S_0 = S_1 = 3$$

Data Linearization: Computer memory systems use a linear address space. An m-dimensional data must be linearized before it is stored in a memory system. Without loss of generality, a vector coordinate $\vec{x}$ can be linearized to be a scalar offset x:

$$x = x_0 + x_1 T_0 + x_2 T_0 T_1 + \cdots + x_{m-1} \prod_{d=0}^{m-2} T_d \quad (1)$$

where $\vec{T} = (T_0, T_1, T_2, \ldots, T_{m-1})$ is the m-dimensional size of the input data. Similarly, each coordinate vector $\vec{a}^{(s)}$ of A can also be linearized as

$$a^{(s)} = a_0^{(s)} + a_1^{(s)} T_0 + a_2^{(s)} T_0 T_1 + \cdots + a_{m-1}^{(s)} \prod_{d=0}^{m-2} T_d$$

Under the above linearization convention, we will use scalars x and $a^{(s)}$ instead of vectors $\vec{x}$ and $\vec{a}^{(s)}$ in the following parts of this paper. *Reuse distance* D_r can then be defined as

$$D_r = \max_s \left(a^{(s)}\right) - \min_s \left(a^{(s)}\right) + 1$$

which represents the linearized distance between the first and the last access of each input data element. In our example, $\{a^{(s)}\} = \{-M, -1, 0, 1, M\}$, $D_r = 2M + 1$.

Stencil Computation: Given an n-point, m-dimensional stencil kernel A and input set $\{x\}$, find all outputs $\{y\}$ by applying A on all inputs $\{x\}$. Note that due to the border effect, the number of valid output data elements in dimension d is always $S_d - 1$ smaller than the input, where S_d is the stencil window size in dimension d. This disappeared region is often referred to as the *halo*.

Complex Stencil Kernel: Two or more stencil kernels can be connected to compose a complex stencil kernel, where the output of the former is used as the input of the latter. Each stencil kernel component of the complex stencil kernel is defined as a *stage* of the whole kernel. Stages are sometimes regarded as the temporal dimension, in analogy to the spatial dimensions of data elements. In particular, stencil kernels can be computed repeatedly where the output of an iteration is used as the input of the next iteration. Such kernels are defined as *iterative*. For the sake of simplicity, the term stage is also used to refer to an iteration of an iterative stencil kernel in this paper. Note that the halo size in each dimension is the sum of halo sizes among all the stages in that dimension.

Optimization Objective: Given a stencil computation task and the resource constraints on a hardware platform, design an accelerator that achieves the maximum sustained throughput.

2.3 FPGA Accelerators for Stencil Computation

Non-uniform memory partitioning-based line buffers are widely used to enable data reuse and reduce the external memory accesses. Cong et al. [4] proved that it requires the least buffer size for a single PE. Wang and Liang [28] propose to adopt the OpenCL model for iterative stencil algorithms. While the coarse-grained, tile-level parallelism increases the on-chip buffer usage and therefore limits

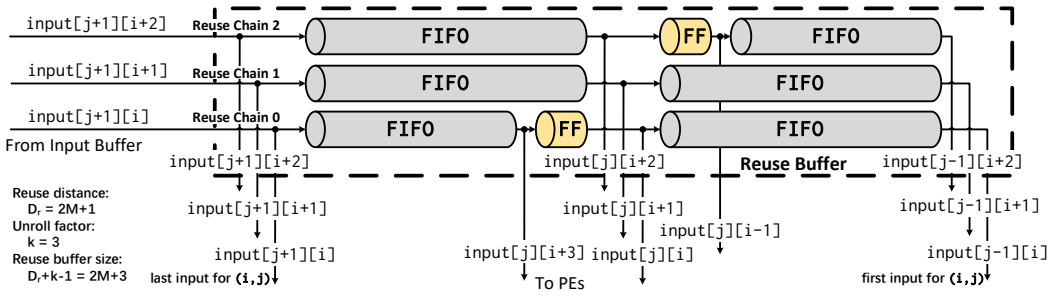

Figure 2: Reuse buffer in SODA microarchitecture: 5-point 2-dimensional Jacobi example in Listing 1 with 3 PEs.

the tile size, OpenCL pipes are used in [28] to alleviate the performance degradation brought by overlapping tile borders. Natale et al. [17] propose to implement multiple temporal iterations as multiple stages and connect them to form a dataflow architecture. This approach scales well as the number of iterations increases, but does not provide parallelism within a single iteration. Zohouri et al. [34] propose to use multiple processing elements (PEs) for each iteration in addition to implementing multiple temporal iterations as multiple stages. However, the reuse buffers are replicated along with the PEs in each stage to provide concurrent accesses in [34]. We shall show later on in Section 3.3 that it is in fact suboptimal.

While the line buffer-based approach is widely used, there are other approaches proposed by researchers. Hegde and Kapre [12] present a soft vector processor for accelerating stencil kernels for OpenCV on FPGAs. Escobedo and Lin [9] use a graph theory-based approach to achieve minimum number of memory banks for a wide range of stencil kernels. However, this approach does not generalize to all stencil kernels. Stitt et al. [25] present a scalable window generator for high bandwidth FPGA systems, which can be used for stencil applications. However, implementing kernels with different window shapes is still non-trivial due to the manual RTL design approach of [25].

In the image processing domain, stencil kernels are ubiquitous. There are several frameworks that can generate efficient FPGA accelerators from high-level image processing DSLs, including Darkroom [11], Halide-HLS [19], and Hipacc [21]. While these DSLs significantly reduce the burden of writing high-quality FPGA codes, analytical model-based efficient design-space exploration and systematic performance optimization for stencil computations, especially for iterative stencil computations, are not present.

In this work, we present the SODA microarchitecture to fully reuse input data while providing scalable fine-grained parallelism for each stage. We can also prove that the SODA microarchitecture requires the smallest reuse buffer size. In addition, to simplify accelerator design, we present a fully automated framework to generate and optimize the SODA microarchitecture. Finally, to perform systematic design-space exploration, we present models for post-synthesis resource utilization and on-board execution throughput. The SODA DSE framework can automatically find the performance-optimized configuration for a given stencil kernel under the platform resource constraints.

3 SODA MICROARCHITECTURE

In this section, the design objectives of the SODA microarchitecture are described first in Section 3.1. The algorithm used to generate the SODA microarchitecture is then presented in Section 3.2. Section 3.3 proves the optimality of the SODA microarchitecture. Section 3.4 shows the modularized dataflow implementation of the SODA microarchitecture.

3.1 Design Objectives

At the microarchitecture level, we aim to optimize the memory resource consumption of one stage as a building block. The input size and throughput constraint is assumed to be given. Choices of tile size and number of PEs will be discussed in Section 4.2. For the proposed microarchitecture, we have four design objectives.

1. *Full pipelining.* Pipelining can increase the throughput with very little resource overhead. Every PE should be fully pipelined and able to consume the input data in one cycle and be ready for the input for the next cycle.

2. *Scalable, fine-grained parallelism.* Compared with coarse-grained parallelism, fine-grained parallelism enables resource sharing and reusing, which make it more efficient and scalable.

3. *Minimum external memory access.* Compared with on-chip memory, external memory access usually has less bandwidth and longer latency. The proposed design fully reuses the input data so that every input data element only needs to be transferred once for a given tile. The streamlined access also enables dataflow optimization between stages.

4. *Minimum reuse buffer size.* We can prove that the proposed microarchitecture achieves the minimum reuse buffer size with given input size and throughput requirement. Compared with other suboptimal architectures, this enables SODA to use more resources to achieve better performance under a given resource constraint.

3.2 Methodology

As mentioned in Section 3.1, the SODA microarchitecture is designed to allow full pipelining, provide scalable parallelism, enable full data reuse, and minimize the required buffer size. This section will discuss the design of the reuse buffer, which plays a crucial role in achieving these objectives. Figure 2 shows an example of the reuse buffer architecture. In the SODA microarchitecture, there are k consecutive output elements generated in each clock cycle, where k is the number of PEs. Suppose the k outputs are $\{y, y+1, y+2, ..., y+k-1\}$. To compute the k outputs, all the needed input data elements are

$$\bigcup_{l=y}^{y+k-1} \left\{ l + a^{(s)} | s \in \{0, 1, 2, \ldots, n-1\} \right\}$$

Let g_k be the number of input data elements needed when producing k outputs. All the needed input data elements can then be represented as

$$\left\{ y + a_{u,k} | u \in 0, 1, 2, \ldots, g_k - 1 \right\}$$

where $a_{u,k}$ denotes the offset of the u-th input data element needed in each clock cycle when producing k outputs.

To generate efficient line buffers, elements in $\{a_{u,k}\}$ are divided into k sets according to their remainder modulo k. Each set will be synthesized as a *reuse chain*. The *reuse buffer* is the collection of all reuse chains. In each cycle, there will be k new inputs fed into the reuse buffer, and each reuse chain will take one input.

Elements in each remainder set cut each reuse chain into several integer intervals. Each interval corresponds to an FF or a FIFO. Since numbers in each set have the same remainder modulo k, the minimum interval length will be k. If the interval length is k, there will be no data elements in-between and the FIFO is actually a

register. If the interval length is larger than k, there will be buffered data in-between, and a FIFO is used. FFs / FIFOs in each set are then connected sequentially to form a complete reuse chain.

The reuse buffer size—i.e., the total number of data elements stored in the reuse buffer—can be calculated as

$$\max_u \left(a_{u,k}\right) - \min_u \left(a_{u,k}\right) + 1 =$$

$$\max_s \left(a^{(s)} + k - 1\right) - \min_s \left(a^{(s)}\right) + 1 = D_r + k - 1$$

Figure 2 shows the proposed microarchitecture with the example of 5-point 2-dimensional stencil. In our $k = 3$ example,

$$\begin{aligned}\left\{a_{u,3}\right\} &= \{-M, -1, 0, 1, M\} \\ &\bigcup \{-M+1, 0, 1, 2, M+1\} \bigcup \{-M+2, 1, 2, 3, M+2\} \\ &= \{-M, -M+1, -M+2, -1, 0, 1, 2, 3, M, M+1, M+2\}\end{aligned}$$

where $g_3 = 11$. In the example of Listing 1 where $M = 9$, there will be $k = 3$ remainder sets / reuse chains:

$$\{-M, 0, 3, M\}, \{-M+1, 1, M+1\}, \{-M+2, -1, 2, M+2\}$$

Chain 0 $\{-M, 0, 3, M\}$ uses two FIFOs for $[-M, 0]$ and $[3, M]$ and an FF for $[0, 3]$. The length of the two FIFOs are $[0-(-M)]/3 = M/3$ and $(M-3)/3 = M/3 - 1$, respectively.

Chain 1 $\{-M+1, 1, M+1\}$ uses two FIFOs for $[-M+1, 1]$ and $[1, M+1]$. The length of the two FIFOs are $[1-(-M+1)]/3 = (M+1-1)/3 = M/3$.

Chain 2 $\{-M+2, -1, 2, M+2\}$ uses two FIFOs for $[-M+2, -1]$ and $[2, M+2]$ and an FF for $[-1, 2]$. The length of the two FIFOs are $[-1-(-M+2)]/3 = M/3-1$ and $(M+2-2)/3 = M/3$, respectively.

The reuse buffer size in this case is $M + 2 - (-M) + 1 = 2M + 3$.

3.3 Optimality

3.3.1 Assumptions. The optimality of the proposed microarchitecture is proven based on the assumptions that the stencil kernel itself, A, and the size of the input, $\vec{T}$, are given. The input can be tiled, and the design choice of tile size will be discussed in Section 4.2. In this section we will discuss the optimality within an input tile. For the proof of optimal reuse buffer size in Section 3.3.3, we further assume that the number of PEs, k, is given.

3.3.2 Optimal Memory Utilization. The minimum requirement on input data is to feed all input data elements at least once; our microarchitecture achieves this by storing the input data on-chip until the last time it is accessed. Therefore, the proposed microarchitecture achieves the optimal memory utilization.

3.3.3 Optimal Reuse Buffer Size. Cong et al. [4] gives a mathematical proof under the polyhedral model that when there is only one PE, the line buffer design has the minimum reuse buffer size equal to the maximum reuse distance, D_r. Based on that, we have the following definitions and theorems:

Lexicographic Order [10]: The lexicographic order relation $\prec$ of two m-dimensional coordinate vectors $\vec{i}$ and $\vec{j}$ is defined as

$$\begin{aligned}\vec{i} \prec \vec{j} \Leftrightarrow\ & (i_{m-1} < j_{m-1}) \vee (i_{m-1} = j_{m-1} \wedge i_{m-2} < j_{m-2}) \\ & \vee (i_{m-1} = j_{m-1} \wedge i_{m-2} = j_{m-2} \wedge i_{m-3} < j_{m-3}) \vee \ldots \\ & \vee (i_{m-1} = j_{m-1} \wedge i_{m-2} = j_{m-2} \wedge \cdots \wedge i_1 = j_1 \wedge i_0 < j_0)\end{aligned}$$

Let A_i represent an n-point stencil window accessing inputs on $\{\vec{i}+\vec{a}^{(0)}, \vec{i}+\vec{a}^{(1)}, \vec{i}+\vec{a}^{(2)}, \ldots, \vec{i}+\vec{a}^{(n-1)}\}$ and producing output on $\vec{i}$. The lexicographic order relation $\prec$ of two stencil windows A_i and A_j is defined as

$$A_i \prec A_j \Leftrightarrow \vec{i} \prec \vec{j}$$

Under the linearization convention in Section 2.2, the lexicographic order of $\vec{i}$ and $\vec{j}$ is the same as the scalar ascending order of linearized i and j. For convenience sake, A_i is also used to denote the input elements of the stencil window A_i and the offset vector $\vec{i}$ is also denoted as i in the following parts.

Theorem 3.1. *The minimum reuse buffer size can only be achieved with PEs producing outputs in* ***consecutive*** *lexicographic order, if the offset i follows the lexicographic order.*

Proof. Suppose an implementation achieving the minimum buffer size is not implemented with PEs producing outputs in consecutive lexicographic order, which means with the k PEs that produce output elements $\{i+p_1, i+p_2, ..., i+p_k\}$ and access input elements $A_{i+p_1} \prec A_{i+p_2} \prec \cdots \prec A_{i+p_k}$ at offset i, $\exists\ p' \notin \{p_1, p_2, ..., p_k\}$ $s.t.\ A_{i+p_1} \prec A_{i+p_2} \prec \cdots \prec A_{i+p'} \prec \cdots \prec A_{i+p_k}$. According to Property 1 in [4], data elements are accessed in lexicographic order as long as the offset vector i follows the lexicographic order. Therefore, if we allocate the k PEs for $p_1, p_2, \ldots, p', \ldots, p_{k-1}$, the buffer size can be reduced by at least one since PE p_k accesses at least one data element lexicographically greater than any of the other PEs. Once PE p_k is replaced by p', the data element accessed only by p_k can be removed from the reuse buffer, which is a contradiction to the assumption of the minimum buffer size for the given implementation. Therefore, we know that Theorem 3.1 is true. □

Theorem 3.2. *The minimum reuse buffer size with $k+1$ PEs is at least the minimum reuse buffer size with k PEs plus* 1.

Proof. Given the an optimal buffer size design with k PEs, if another PE is to be added but no additional input data element is necessary, the additional PE must be lexicographically between the existing PEs. According to Theorem 3.1, we know that the given design must not be an optimal buffer size design since its PE inputs are not in consecutive lexicographic order. Therefore, by contradiction, there must be at least one additional data element added to the buffer. □

Based on Theorem 3.1, Theorem 3.2, and [4], we know by induction that our microarchitecture with the buffer size equal to $D_r + k - 1$ is optimal.

3.4 Dataflow-Based Implementation

The SODA microarchitecture can be efficiently implemented as dataflow modules. The dataflow implementation enables high-frequency synthesis result and accurate resource modeling, due to its localized communication [6] and modularized structure. It also enables the flexibility to connect multiple stages together in a single accelerator. Figure 3 shows the dataflow modules of 1 iteration of the Jacobi kernel shown in Listing 1.

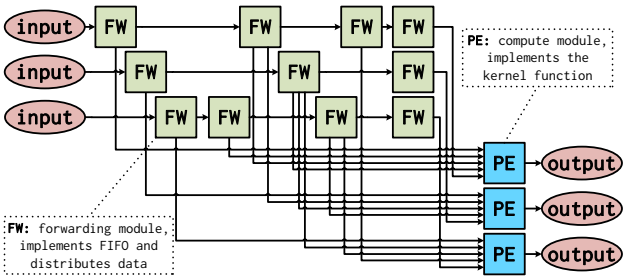

Figure 3: Dataflow modules in a SODA microarchitecture.

As shown in Figure 3, the forwarding modules (`FW`) forward and distribute input data to proper destination modules. Each forwarding module either directly forwards data from the input, or implements a FIFO or FF as part of the reuse buffer. Each FIFO or FF in Figure 2 corresponds to a forwarding module shown in Figure 3. The structure of a forwarding module is only determined by the data type, FIFO depth, and fanout. On FPGAs, FIFOs can be implemented with either shift register lookup tables (SRLs) or block RAMs (BRAMs). On our Xilinx platform, we use `hls::stream` provided in Vivado HLS to implement FIFO. Large FIFO whose capacity is larger than 1024 bits is implemented with BRAM and small FIFO

is implemented with SRL. The compute modules (PE) are the processing elements and implement the kernel function. Each compute module contains 1 PE, which produces 1 output data element per cycle. For a given stencil kernel, all compute modules in the same stage have the same structure. The dataflow architecture enables the flexibility of cascading multiple stages together. The inputs and outputs can be connected to DRAM or to another stage's outputs or inputs. Figure 4 shows the overview of an example of a complete SODA accelerator.

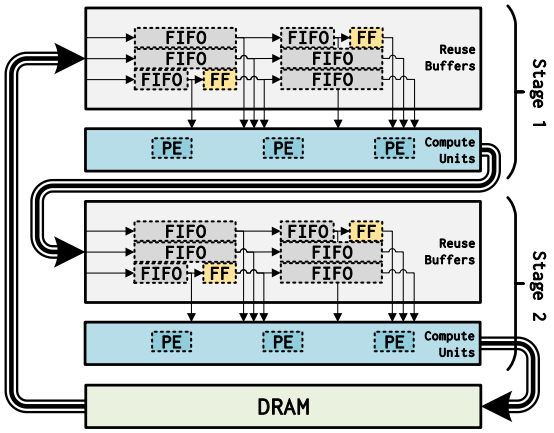

Figure 4: Overview of a complete SODA accelerator.

As a common type of external memory on FPGAs, DRAMs have a burst I/O mode which provides higher bandwidth [32], but it also puts some restrictions on the data. On our Xilinx platform, burst-mode DRAM access is fully pipelined, and in each cycle $W_b = 512$ bits are read/written for the maximum throughput. Therefore, 8-bit, 16-bit, or 32-bit input data must be coalesced before sent / received to / from DRAM to achieve the maximum throughput. The SODA automation framework automatically generates modules that handles the memory coalescing and the corresponding host-side data layout manipulation code, which improves external memory throughput without adding complexity to the programming model.

Also, although burst I/O are fully pipelined, the latency is quite long. Therefore, to hide this latency, burst length—i.e., the number of data elements read/written in each DRAM access—has to be large enough. Thanks to the dataflow implementation, the DRAM access can be automatically performed in burst mode with sufficiently long burst length, without the need of coarse-grained pipelining and double buffering as discussed in [2].

4 AUTOMATION AND EXPLORATION

In this section, the programming model for SODA and the corresponding automation framework are discussed first in Section 4.1. Under the proposed programming model, the configurable parameters are then discussed in Section 4.2. Since these parameters form a large design space and synthesizing an FPGA accelerator is very time-consuming, a resource model and a performance model are proposed in Section 4.3 and Section 4.4 to predict the post-synthesis resource utilization and the on-board execution performance, respectively. With these models, the large design space can be pruned effectively, which is discussed in Section 4.5.

4.1 Programming Model

To simplify accelerator kernel design, SODA defines a domain-specific language (DSL) to specify the design parameters as well as the stencil kernel in a concise and high-level way.

As shown in Listing 2, the `kernel` statement specifies the name of the stencil kernel. The `input` statement specifies the name, type, and tile size of the input data. Note that the last dimension of tile size is * because it is not needed for data linearization and therefore is given at runtime. The `output` statement specifies the name and type of output data as well as the stencil kernel expression to compute it. If the kernel contains more than 1 stage, intermediate stages can be specified with `buffer` statements. The `unroll factor` statement specifies the number of PEs in each stage. The `iterate factor` statement automatically implements the specified number of iterations, which simplifies the expression of iterative kernels. Note that the expressions are not restricted to have only 1 input array; the SODA compiler (sodac) is capable of processing kernels taking multiple arrays as inputs. To connect with user-defined code, the SODA automation framework provides a concise C/C++ binding of the generated accelerator.

```
kernel: jacobi2d
input float: in(3000,*) # specifies the tile size
output float: out(0,0) = (in(0,-1) +
                in(-1,0) + in(0, 0) + in(1,0) +
                           in(0, 1)) * 0.2f
unroll factor: 3
iterate factor: 2
# SODA supports multiple stages:
# buffer float: tmp(0,0) = (in(0,-1) +
#                in(-1,0) + in(0, 0) + in(1,0) +
#                           in(0, 1)) * 0.2f
# output float: out(0,0) = (tmp(0,-1) +
#                tmp(-1,0) + tmp(0, 0) + tmp(1,0) +
#                           tmp(0, 1)) * 0.2f
# SODA supports multiple arrays as input:
# buffer float: t(0,1) = in(0,0) + tmp(0,2)
```

Listing 2: 2-dimensional Jacobi kernel in SODA DSL.

To reduce the burden of programming FPGAs, we develop a fully automated framework to generate efficient hardware accelerators for stencil computations. Currently the SODA automation framework interfaces with Xilinx SDAccel implementation flow. The automation framework takes a high-level DSL as user input, implements tiling automatically, uses the SODA microarchitecture discussed in Section 3 as building blocks for implementing stages, automatically solves the dependencies among the stages, and connects multiple stages or iterations with dataflow optimization.

The complete SODA automation framework is shown in Figure 5. The SODA compiler (sodac) parses the SODA DSL, does a source-to-source transformation, and generates the HLS C++ code as the kernel and the OpenCL API code for the host. Then gcc will be invoked to compile and link the OpenCL API with user-defined application and xocc will be invoked to launch the Xilinx SDAccel flow to do HLS, logic synthesis, placement, and routing. Host program and FPGA bitstream will be the eventual synthesis results, ready for execution on any compatible environment. In addition, there is a standalone design-space exploration (DSE) framework provided by SODA, which is used to automatically tune the kernel configuration parameters for optimal performance.

4.2 Configurable Parameters

Tile Sizes $T_0, T_1, ..., T_{m-2}$. To generate valid accelerators, the linearization convention has to be determined before synthesis. Since size of all but the last dimension appear in Formula 1, the size of the first $m-1$ dimensions of the input must be determined before synthesis. However, the exact input size may not be determined at the time of accelerator design. Moreover, the input may be too large to fit the on-chip storage. Therefore, tiling is a reasonable design choice. Note that our automation framework automatically does tiling with the size specified in the DSL and the user does not have to do tiling manually. In this paper, we argue that with the ever-increasing resolution of sensors, the input is sufficiently large and the tile size is only limited by the on-chip storage size.

Unroll Factor k. To avoid the confusion with the total number of PEs in a complex kernel, the term *unroll factor* is used to represent

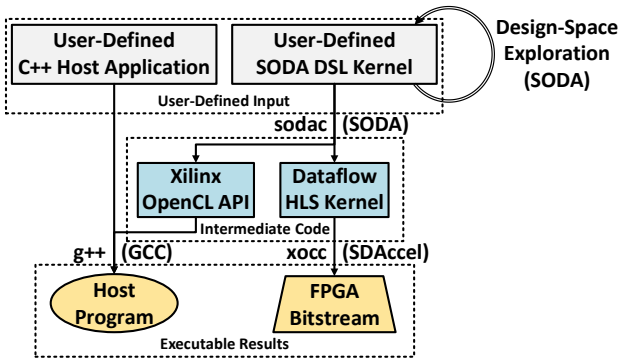

Figure 5: SODA automation framework.

the number of PEs in each stage, in analogy to the case where the number of PEs generated for an unrolled loop is determined by the unroll factor. Since all building-block modules in our dataflow architecture can be fully pipelined (which is optimal for throughput), we argue that it does not make sense to use different unroll factors among different stages, which will cause throughput mismatch among stages and increase the initiation interval (II) of the stage with higher unroll factor. The SODA automation framework implements all stages with the same unroll factor.

Iterate Factor q. For iterative stencil kernels, *iterate factor* is the number of iterations implemented in the accelerator. In practice, the number of iterations required by the application is often much greater than the number of iterations can be implemented in an accelerator. To complete all required iterations, the execution kernel must be invoked multiple times. In this paper, we assume that the number of such invocation is sufficiently large and all iterate factors that can be implemented under the resource constraint are permissible.

4.3 Resource Model

The SODA DSE framework models all four types of resources on FPGAs, i.e. LUT, FF, DSP, and BRAM. To make the model more practical, we target post-synthesis resource utilization instead of the HLS result. Since the HLS report contains modularized details and can be obtained within minutes whereas it is hard to distinguish user-level modules from the post-synthesis report and it takes hours to obtain, as the first step we take the HLS report and model the HLS resource utilization. As the second step, based on the HLS report and post-synthesis report, we then make adjustments to the HLS model accordingly so that the final model can reflect the post-synthesis resource utilization. Thanks to the modularized dataflow implementation, the resource utilization of a SODA accelerator can be accurately modeled at the dataflow module level. Those modules can be divided into three categories.

Compute modules consume the majority of resources for computation. The SODA DSE framework obtains the resource utilization of each compute model by running HLS. Since compute modules in the same stage for all iterations have the same structure, they only need to go through HLS once.

Forwarding modules consume the majority of the remaining resources for communication. For a forwarding module with a ϕ-bit wide data type, LUT and FF consumption grows linearly with fanout κ. For those who implement a FIFO of depth $\eta > 0$, there is an constant LUT and FF overhead for the control logic of the FIFO, in addition to the SRL or BRAM used to implement the FIFO. Since only small FIFOs are implemented as SRLs with LUTs, LUTs used for this purpose are much less than those used to implement logics. Thus, the SRL contribution to the LUT utilization is ignored in the model. The coefficients in the model are kernel-independent and can be obtained from a series of pre-executed HLS results. η, ϕ, and κ are determined for each module by sodac, according to the tile size and unroll factor configuration and the stencil kernel itself. Forwarding modules do not use any DSP.

We observe that it is very hard to develop a closed-form analytical formula to predict the BRAM usage $\Theta(\phi, \eta)$ from data width ϕ and depth η, due to the undocumented optimizations performed by Xilinx tools. For example, we observe that a 16-bit × 8K FIFO is implemented with 8 BRAMs whereas a 16-bit × 16K FIFO is implemented with only 15 BRAMs. To accurately predict BRAM utilization, SODA invokes Xilinx Vivado to synthesize a single FIFO and obtains its BRAM utilization from the synthesis report. Such a simple synthesis only takes 3 to 4 minutes and SODA keeps the results in a database so that the result can be reused. Notice that $\Theta(\phi, \eta)$ is a step function of η and does not depend on module fanout κ, the database can be frequently reused and rarely updated. Among all design-space exploration performed in the experiments done in Section 5, only 115 entries are needed. Since these entries are shared for different kernels and can be generated in parallel, time spent on the $\Theta(\phi, \eta)$ model is negligible compared with the time needed to synthesize a complete accelerator.

Auxiliary modules, including the interconnections with DRAM, control signals, and memory coalescing modules, constitute the remainder of resource consumption. The I/O modules and control modules are independent with the tile size and the iterate factor, whereas the unroll factor has a very weak influence on memory coalescing modules, which is negligible in size compared with the total resource utilization. Consequently, resource utilization of auxiliary modules is considered as a constant and can be obtained from the pre-executed HLS results.

With a given kernel and configuration, the number and parameter of all modules can be determined analytically via the microarchitecture generation algorithm presented in Section 3.2. The total resource consumption is the sum of resource consumption of each module. Note that the BRAM and DSP models obtained above are already reflecting the post-synthesis results. Experimental results in Section 5.2 show 1.84% and 0% average prediction errors for BRAM and DSP, respectively. For the LUT and FF utilization, we observe that the post-synthesis utilization of LUT and FF have a linear relationship with the utilization reported by HLS. Moreover, this linear relationship does not depend on the application kernel or configuration parameters. Therefore, we adjust our model for LUT and FF with a linear adjustment function to get post-synthesis models. Experimental results in Section 5.2 show 6.23% and 7.58% average prediction errors for LUT and FF, respectively.

4.4 Performance Model

In this paper, our optimization objective is the sustained throughput H, which can be measured by the number of output data elements per unit time. For an accelerator running at frequency f and having tile size $\vec{T} = (T_0, T_1, ..., T_{m-1})$, unroll factor k, and iterate factor q, the ideal throughput of the kernel is

$$H_{\text{ideal}}(k, q, \vec{T}) = kqf \prod_{d=0}^{m-1} \frac{T_d - q \cdot (S_d - 1)}{T_d} \tag{2}$$

where m is the number of dimensions and S_d is the stencil window size in each iteration. For non-iterative stencil kernels, $q \equiv 1$.

H_{ideal} may not be achievable since the hardware put constraints on H in two aspects: (1) External bandwidth limits the effective unroll factor (2) Available resource limits the achievable tile size, unroll factor, and iterate factor. The constraints are modeled as

$$H \le H_{\text{ideal}}(k, q, \vec{T}) \tag{3}$$

$$kfW \cdot \frac{H}{H_{\text{ideal}}(k, q, \vec{T})} \le B^{\text{MAX}} \tag{4}$$

$$R^{(\text{AUX}, \chi)} + kqR^{(\text{CP}, \chi)} + q \cdot R^{(\text{FW}, \chi)}(k, \vec{T}) \le R^{(\text{MAX}, \chi)} \tag{5}$$

$\chi \in \{\text{LUT}, \text{FF}, \text{DSP}, \text{BRAM}\}$ represents the type of resource. H is the achieved throughput. B^{MAX} is the maximum available DRAM bandwidth. W is the total width of input and output data types. $R^{(\text{MAX},\chi)}$ is the maximum available χ resource. $R^{(\text{AUX},\chi)}$ is the resource consumption of auxiliary modules. $R^{(\text{CP},\chi)}$ is the resource consumption of compute modules of a single PE in a single iteration. $R^{(\text{FW},\chi)}(k, \vec{T})$ is the resource consumption of forwarding modules in a single iteration, which is a function of k and $\vec{T}$. Note that as a result of the dataflow implementation, the frequency of accelerators achieved is always within 10% of the target, according to our experimental results. Therefore, in the performance model we treat f as a constant. Experimental results in Section 5.2 show 4.22% average prediction error for performance.

4.5 Design-Space Pruning

As shown in Section 4.4, both the objective function and the constraints are non-linear. Therefore, we do not seek closed-form solution. Instead, we prune the design space and use branch-and-bound method to find the optimal configuration.

The design space of unroll factor k is limited by the external interface. On our evaluation platform, the input and output interface are both 512 bits wide. To avoid complex multiplexers, k is restricted to powers of 2. With the minimum data type width being 8 bits, this effectively reduces the design space of k to have at most 6 points. For iterative kernels, the iterate factor q is a positive integer, but it is bounded by the constraint in Formula 5. That is, the total number of PEs is bounded by the available resources. On our evaluation platform, for the simplest PEs, $kq \leq 10^2$. For non-iterative kernels, $q \equiv 1$. The design space of tile sizes $T_0, T_1, ..., T_{m-2}$ is much larger compared with k and q, especially for high-dimensional stencils. Nevertheless, notice that the bound of H is monotonically increasing with respect to T_d in each dimension d for every given k and q, $T_0, T_1, ..., T_{m-2}$ can be efficiently searched via branch-and-bound. Note that T_{m-1} is not part of the design space because it is determined by the input and is a runtime parameter instead of a design parameter. With the on-chip storage size being several megabytes, the total size of design space can then be reduce to less than 10^6 and can be explored within a few minutes.

5 EXPERIMENTAL EVALUATION

5.1 Experiment Setup

We evaluate SODA with a Xilinx Kintex UltraScale FPGA. The AlphaData ADM-PCIE-KU3 board used in our experiments is equipped with XCKU060 FPGA and 2×8GB 1600MT/s DDR3 DRAMs. High-level synthesis is performed by Xilinx Vivado HLS [5, 33]. Xilinx SDAccel 2017.2, the latest version offering support for this platform, is used for system integration. The target clock frequency is 250MHz. The CPU experiments are conducted on a server with two Intel Xeon E5-2620 v3 CPUs and 4×16GB 2133MT/s DDR4 DRAMs.

We use 3 non-iterative benchmarks and 4 iterative benchmarks to evaluate our SODA framework. The benchmarks cover a wide range of application domains and have been used in previous published works [3, 17]. Among the 7 benchmarks, SODEL 2D is used for edge detection in the image processing domain. DENOISE 2D/3D are used in the medical imaging domain [3]. The four iterative benchmarks, namely JACOBI 2D/3D, SEIDEL 2D, and HEAT 3D, are used in the linear algebra domain [17].

5.2 Model Validation

In Section 4.3 and Section 4.4 we proposed a resource model and a performance model. In this section, we run two sets of experiments to validate our model. In the first set of experiments, we fix the tile size and explore the unroll factor. For iterative benchmarks, we also explore different iterate factors. In the second set of experiments, we fix the unroll factor and explore the tile size. In total, 75 different configurations are synthesized. The average of the achieved frequency is 245.66MHz with the lowest being 229.1MHz. To validate the resource model, we compare the model prediction against the *post-synthesis* resource utilization. To validate the performance model, we run *on-board* experiments with 4 different sizes of input and obtain sustained throughput via linear regression of the execution time and the number of pixels processed. The measurement errors of throughput are within 1% for all configurations. The average error rate of the model prediction is listed in Table 2.

Table 1: Benchmarks.

Benchmark	Iterative?	Data Type	Optimal Configuration		
			Tile Size	Unroll Factor	Iterate Factor
SOBEL 2D [3]	No	`uint16_t`	524302	32	–
DENOISE 2D [3]	No	`float`×2 in `float`×1 out	21846	8	–
DENOISE 3D [3]	No	`float`×2 in `float`×1 out	156×157	4	–
JACOBI 2D [17]	Yes	`float`	16392	8	10
JACOBI 3D [17]	Yes	`float`	181×182	8	6
SEIDEL 2D [17]	Yes	`float`	32768	8	9
HEAT 3D [17]	Yes	`float`	256×257	8	5

Table 2: Average error rate of model prediction.

Prediction Item	BRAM	DSP	LUT	FF	Throughput
Average Error	1.84%	0%	6.23%	7.58%	4.22%

5.3 Performance Analysis

Table 3 compares the performance of the optimal configurations found by the SODA DSE framework with four baselines. To make the CPU baseline realistic, all benchmarks are rewritten in Halide [20] DSL and optimized via tiling, parallelization, and vectorization. The resulting CPU code is able to utilize all 24 hyper-threads on our server. We implement the FPGA baselines using the methodologies proposed in [3], [17], and [34]. Note that [17] and [34] do not target non-iterative stencil algorithms and [3] does not target iterative stencil algorithms. The #Op column shows the number of operations per iteration.

Table 3: Performance Comparison.

Benchmark	#Op	Platform	Throughput		Performance (Norm. to 24t-CPU)
			pixel/ns	Op/ns	
SOBEL 2D	16	CPU	6.66	106.59	1
		[3]	0.25	4.00	0.04
		SODA	5.37	85.86	0.81
DENOISE 2D	45	CPU	1.86	83.55	1
		[3]	0.25	11.07	0.13
		SODA	1.05	47.07	0.56
DENOISE 3D	57	CPU	1.91	109.01	1
		[3]	0.25	14.24	0.13
		SODA	0.93	53.16	0.49
JACOBI 2D	5	CPU	5.49	27.44	1
		[17]	16.67	83.34	3.04
		[34]	17.20	86.01	3.13
		SODA	18.01	90.04	3.28
JACOBI 3D	7	CPU	4.24	29.66	1
		[17]	4.72	33.01	1.11
		[34]	9.86	69.04	2.33
		SODA	12.00	83.98	2.83
SEIDEL 2D	6	CPU	5.82	34.90	1
		[17]	15.03	90.18	2.58
		[34]	15.99	95.95	2.75
		SODA	16.22	97.34	2.79
HEAT 3D	15	CPU	4.21	63.18	1
		[17]	4.70	70.57	1.12
		[34]	6.65	99.70	1.58
		SODA	8.99	134.91	2.14

As shown in Table 3, the 24-thread CPU baseline outperforms SODA for non-iterative benchmarks. This is because non-iterative benchmarks are bounded by communication. The CPU platform has 4 DDR4 channels with 68.3GB/s theoretical bandwidth in total whereas the FPGA platform only has 2 DDR3 channels with

25.6GB/s theoretical bandwidth. If they have the same external memory bandwidth, SODA can outperform CPU by 1.65x on average. For iterative benchmarks, SODA (and other FPGA platforms) can compute multiple iterations without extra accesses to the external memory whereas the CPU platform has to make a trade-off between memory access and redundant computation. Consequently, SODA outperforms the CPU baseline by 2.76x on average.

Thanks to scalable, fine-grained parallelism provided by the SODA microarchitecture, SODA shows 9.82x speed up on average compared with [3]. Compared with [17], SODA achieves 1.08x average speedup on 2D benchmarks and 2.23x average speedup on 3D benchmarks. Compared with [34], SODA achieves 1.03x average speedup on 2D benchmarks and 1.28x average speedup on 3D benchmarks. The speedup comes from three aspects: (1) SODA uses less resources for communication and can therefore implement more PEs. (2) SODA can accommodate larger tile sizes and is thus less sensitive to the halo size (which is proportional to the iterate factor). (3) SODA provides scalable, fine-grained spatial parallelism and can reduce the halo size caused by temporal parallelism. The difference of the speedup on 2D and 3D benchmarks is due to (2), where 3D benchmarks have much smaller tile sizes in a single dimension compared with 2D ones (as shown in Table 1) and are thus more sensitive to large halo size. The optimal microarchitecture and systematic DSE brought by SODA give more speedup for 3D benchmarks compared with [34]. In addition, the optimal configuration for SODA can be obtained from fast and automated DSE, which previous accelerator designs do not provide.

6 CONCLUSION

In this paper we present SODA, an automated accelerator design framework for Stencil with Optimized Dataflow Architecture, to address two major challenges for stencil accelerator design on FPGAs, i.e. the suboptimal microarchitecture and the lack of systematic design-space exploration (DSE). The SODA microarchitecture minimizes the on-chip reuse buffer size while providing scalable, fine-grained parallelism, which delivers an optimal microarchitecture as a building block. The SODA automation framework automatically generates modularized dataflow implementation of the SODA microarchitecture. With analytical and accurate resource and performance modeling, the SODA design-space exploration framework is able to perform DSE automatically and find the optimal configuration within just a few minutes. Experimental results show up to 3.28x speedup over 24-thread CPU and our fully automated framework achieves better performance than manually designed FPGA accelerators through its systematic, analytical model-based design-space exploration.

ACKNOWLEDGMENTS

The authors would like to thank Cody Hao Yu and the anonymous reviewers for their valuable comments and helpful suggestions. This work is partially supported by the Intel and NSF joint research program for Computer Assisted Programming for Heterogeneous Architectures (CAPA), and the contributions from Fujitsu Labs, Huawei, and Samsung under the CDSC industrial partnership program. We thank Amazon for providing AWS F1 credits.

REFERENCES

[1] Alan C. Bovik. 2009. *The Essential Guide to Image Processing*. Academic Press.
[2] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *DAC*. 109:1–109:6.
[3] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. 2014. An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers. In *DAC*. 77:1–77:6.
[4] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. 2015. An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers. *TCAD* 35, 3 (2015), 407–418.
[5] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD* (2011).
[6] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality Aware Transformation for High-Level Synthesis. In *FCCM*. 125–128.
[7] Jason Cong, Peng Zhang, and Yi Zou. 2012. Optimizing Memory Hierarchy Allocation with Loop Transformations for High-level Synthesis. In *DAC*. 1233–1238.
[8] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *SC*. 4:1–4:12.
[9] Juan Escobedo and Mingjie Lin. 2018. Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels. In *FPGA*. 199–208.
[10] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time. *IJPP* 21, 5 (1992), 313–347.
[11] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *TOG* 33, 4 (2014), 1–11.
[12] Gopalakrishna Hegde and Nachiket Kapre. 2015. Energy-Efficient Acceleration of OpenCV Saliency Computation Using Soft Vector Processors. In *FCCM*. 76–83.
[13] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ICS*. 311–320.
[14] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *PLDI*. 235–244.
[15] Shih-Wei Liao, Sheng-Jun Tsai, Chieh-Hsun Yang, and Chen-Kang Lo. 2016. Locality-Aware Scheduling for Stencil Code in Halide. In *ICPPW*. 72–77.
[16] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 2011. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *SC*. 11:1–11:12.
[17] Giuseppe Natale, Giulio Stramondo, Pietro Bressana, Riccardo Cattaneo, Donatella Sciuto, and Marco D. Santambrogio. 2016. A Polyhedral Model-Based Framework for Dataflow Implementation on FPGA Devices of Iterative Stencil Loops. In *ICCAD*. 77:1–77:8.
[18] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-Based Data Reuse Optimization for Configurable Computing. In *FPGA*. 29–38.
[19] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2016. Programming Heterogeneous Systems from an Image Processing DSL. (2016), 12 pages.
[20] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. In *SIGGRAPH*, Vol. 31. 1–12.
[21] Oliver Reiche, M. Akif Ozkan, Richard Membarth, Jürgen Teich, and Frank Hannig. 2017. Generating FPGA-based Image Processing Accelerators with Hipacc: (Invited paper). In *ICCAD*. 1026–1033.
[22] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R Gregg Brickner. 1997. Compiling Stencils in High Performance Fortran. In *SC*. 1–20.
[23] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. 2014. Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth. *TPDS* 25, 3 (2014), 695–705.
[24] Muhammad Shafiq, Miquel Pericas, Raul de la Cruz, Mauricio Araya-Polo, Nacho Navarro, and Eduard Ayguadé. 2009. Exploiting Memory Customization in FPGA for 3D Stencil Computations. In *FPT*. 38–45.
[25] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. 2018. Scalable Window Generation for the Intel Broadwell + Arria 10 and High-Bandwidth FPGA Systems. In *FPGA*. 173–182.
[26] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A Framework for Enhancing Data Reuse via Associative Reordering. In *PLDI*. 65–76.
[27] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *SPAA*. 117–128.
[28] Shuo Wang and Yun Liang. 2017. A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model. In *DAC*. 28:1–28:6.
[29] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* (2009).
[30] Markus Wittmann, Georg Hager, and Gerhard Wellein. 2010. Multicore-Aware Parallel Temporal Blocking of Stencil Codes for Shared and Distributed Memory. In *IPDPSW*. 1–7.
[31] Stephen Wolfram. 1984. Computation Theory of Cellular Automata. *Communications in Mathematical Physics* (1984).
[32] Xilinx. 2017. Vivado Design Suite: AXI Reference Guide (UG1037). https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf
[33] Xilinx. 2018. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[34] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *FPGA*. 153–162.