DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs

MOSTAFA KORAEI, University of Tehran, Iran and Norwegian University of Science and Technology, Norway

OMID FATEMI, University of Tehran, Iran

MAGNUS JAHRE, Norwegian University of Science and Technology, Norway

Iterative Stencil Loops (ISLs) are the key kernel within a range of compute-intensive applications. To accelerate ISLs with Field Programmable Gate Arrays, it is critical to exploit parallelism (1) among elements within the same iteration and (2) across loop iterations. We propose a novel ISL acceleration scheme called Direct Computation of Multiple Iterations (DCMI) that improves upon prior work by pre-computing the effective stencil coefficients after a number of iterations at design time—resulting in accelerators that use minimal on-chip memory and avoid redundant computation. This enables DCMI to improve throughput by up to 7.7× compared to the state-of-the-art cone-based architecture.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Computing methodologies** → **Parallel computing methodologies**;

Additional Key Words and Phrases: Accelerator, FPGA, iterative stencil loops (ISLs)

ACM Reference format:

Mostafa Koraei, Omid Fatemi, and Magnus Jahre. 2019. DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs. *ACM Trans. Archit. Code Optim.* 16, 4, Article 36 (October 2019), 24 pages. <https://doi.org/10.1145/3352813>

1 INTRODUCTION

General purpose processors can be an inefficient computing infrastructure for many application domains [15]. These inefficiencies are exacerbated by the observation that future computing systems will be limited by energy consumption or power dissipation [3]. These observations motivate for the development of *accelerators*—special-purpose computer architectures that are particularly efficient for a given application or application domain. Broadly, accelerators can be realized with Application Specific Integrated Circuits (ASICs) or by using reconfigurable substrates such as Field Programmable Gate Arrays (FPGAs). Although ASICs are generally more efficient than FPGAs [23], their high non-recurring engineering costs mean that they are only applicable to

This work was partly supported by the European Horizon 2020 project TULIPP (grant agreement number 688403).

Authors' addresses: M. Koraei, University of Tehran, Kargar Street, 1417466191 Tehran, Iran and Norwegian University of Science and Technology, 7491 Trondheim, Norway; email: m.koraei@ut.ac.ir; O. Fatemi, University of Tehran, Kargar Street, 1417466191 Tehran, Iran; email: omid@fatemi.net; M. Jahre, Norwegian University of Science and Technology, 7491 Trondheim, Norway; email: magnus.jahre@ntnu.no.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/10-ART36

<https://doi.org/10.1145/3352813>

high-volume application domains [20]. FPGAs avoid such costs, because they are reconfigurable—i.e., a single FPGA can be used to accelerate different applications. At the same time, FPGAs have the ability to implement arbitrary hardware within the resource constraints of the particular architecture. The flexibility and relative efficiency of FPGAs have caused leading industry players such as Microsoft and Baidu to add FPGAs to their cloud infrastructures [5, 33, 35].

FPGAs are emerging as a platform for accelerating High Performance Computing (HPC) applications. The core kernel of many HPC applications is an *Iterative Stencil Loop (ISL)* [60]. Examples of HPC-applications that contain ISLs are differential equation solvers, weather and climate modeling, as well as seismic and fluid simulations. The core data structure of an ISL application is a (multi-dimensional) array in which all elements are typically updated once for each iteration of the outermost loop; each iteration of the outer loop is commonly referred to as a time-step. The data array is commonly large, which means that it has to be stored in off-chip memory. The inner-loop(s) of an ISL update the array according to a stencil pattern that describes what elements are used as input to the computation that produces an output element. Thus, ISL-applications contain an abundance of arithmetic operations that are mostly independent of each other. However, respecting the dependencies that do exist can lead to significant challenges when parallelizing ISLs [39].

The importance and high potential for parallelism makes ISL-applications an interesting acceleration target. However, developing efficient ISL accelerators is challenging for two reasons [4]. First, ISL-applications operate on a large data array in which all data elements are typically updated in each iteration of the outer-loop. Second, there are commonly true data dependencies across outer-loop iterations and among the elements processed in the inner-loop(s). Developing an efficient ISL accelerator requires optimized memory access behavior and exploiting parallelism in a way that respects data dependencies.

On CPUs and Graphics Processing Units (GPUs), ISL-applications are commonly tiled and time-skewed to expose parallelism to the processor cores and improve locality [16, 55]. For FPGAs, more fine-grained parallelization strategies are possible. First, the accelerator can exploit that inner-loop computations can be performed in parallel as long as an output element is not used as an input element within the same time-step. Intra-time-step dependencies—or spatial dependencies—can commonly be avoided by using a double-buffering strategy [40]. Second, the accelerator can exploit that the same computation is carried out in successive time-steps (i.e., several iterations of the outer loop). In this way, the accelerator can carry out the computations of several outer-loop iterations with a single pass over the data array—i.e., exploit temporal parallelism.

The abundant parallelism of ISLs makes them well suited to FPGA acceleration and prior work has found that FPGAs can perform comparably to high-end GPUs [60] and significantly better than multi-core CPUs [6]. A number of prior works propose FPGA-based ISL accelerators that exploit spatial parallelism (e.g., [11, 12, 17, 21, 26, 27, 32, 46]). However, there are only two main architectural approaches for implementing accelerators that exploit both spatial and temporal parallelism. The first approach is exemplified by the *Streaming Stencil Time-step (SST)* [4] scheme. SST is memory centric and streams the data array through the FPGA while performing the stencil computation. Each SST core exploits spatial parallelism, and temporal parallelism is exploited by instantiating a series of SST-cores on the FPGA. In other words, each SST performs the computation of a single time-step and then passes the intermediate results on to the next SST without accessing off-chip memory. Unfortunately, SST is inefficient, because each SST-core needs to buffer complete dimensions of the input array, which leads to excessive use of On-Chip Memory (OCM).

The second architectural approach is exemplified by the *Cone-based Architecture (CA)* [40]. CA exploits that the input elements of a single output element forms a cone that expands through prior iterations. This cone does not change within a single ISL-application, since it applies the same stencil within and across iterations. CA uses this observation to generate accelerators that compute

multiple time-steps in a single invocation. These accelerators can be replicated to exploit spatial parallelism. This approach is also inefficient, since the input cones of different output elements commonly overlap, which results in redundant computation of intermediate results. The effect of this inefficiency increases with the size of the stencil and the depth of the cone (see Section 2.2).

Our objective is to devise an ISL acceleration strategy that avoids redundant computation and uses minimal OCM. To achieve this, we make two key observations. The first observation is that the core computation within ISLs are often a first degree (or linear) polynomial—i.e., constant coefficients are multiplied with data elements that are spatially close to the output element in the (multi-dimensional) array. Since both multiplication and addition are commutative and associative, it is possible to compute each part of the polynomial independently. This observation simplifies accelerator design as it makes it possible to accumulate partial results in any order.¹ The second observation is that the coefficients of the polynomial across multiple iterations are uniquely determined by the stencil pattern and the depth of the cone. This observation simplifies accelerator design, since it makes it possible to compute the effective coefficients after D time-steps at design time—thereby removing redundant computations at runtime.

We leverage these observations in our *Direct Computation over Multiple Iterations (DCMI)* acceleration strategy for ISLs. We provide an accelerator template that implements the DCMI strategy and thereby exploits both spatial and temporal parallelism while avoiding redundant computation and using minimal OCM. More specifically, DCMI allocates the minimal amount of OCM necessary to fetch each data element exactly once per accelerator invocation (see Section 2.4 for details). At design time, we determine the effect a single input element will have on all its reachable output elements (i.e., compute the effective coefficients). At runtime, we fetch a pre-defined number of input elements from off-chip DRAM and provide these to the DCMI accelerator. The accelerator carries out all possible computations for each data element across a number of time-steps and accumulates partial results in OCM. When an output element is complete (i.e., all partial results have been accumulated), it is written back to DRAM and is ready to be used as input for later time-steps. Determining when output elements are ready is simple, because the stencil computations are regular and complete after a constant latency. In summary, we make the following major contributions:

- We provide a fundamentally new approach for simultaneously exploiting the temporal and spatial parallelism of ISLs in FPGAs. The key idea is to compute the effective coefficients after D time-steps at design time, and use these coefficients to generate all possible partial results for a pre-defined number of input elements at runtime.
- We provide a High-Level Synthesis (HLS) template for configuring and synthesizing DCMI accelerators on FPGAs and a design time tool for computing the effective coefficients after D time-steps. We use these to generate DCMI accelerators for common stencil patterns and find that DCMI improves performance by up to $7.7\times$ compared to a carefully tuned state-of-the-art CA-based accelerator.
- Maximizing the performance of a DCMI accelerator requires tuning the amount of spatial and temporal parallelism to fully utilize the resources of the target FPGA platform. We provide a search heuristic that achieves on average 93% of the performance of an accelerator found by exhaustive search while reducing the number of examined design points from 66 to 11.5 on average—thereby reducing average synthesis time during design space exploration by $5.8\times$.

¹Linear polynomials are common within ISLs. In fact, 9 of 12 SPEC CPU 2017 floating point speed benchmarks [34, 49] have performance-critical procedures that contain linear polynomial ISLs (see Section 5.3 for details).

Fig. 1. Example explaining the stencil computing kernel using a 1D three-point stencil.

2 EXPLAINING THE EFFICIENCY OF DCMI

In this section, we provide an example that illustrates why the DCMI acceleration strategy is superior to current state-of-the-art FPGA-based accelerators [4, 40]. We first explain the operation of the stencil compute kernel in Section 2.1. Then, we provide a simple example that shows that prior work uses OCM inefficiently and performs redundant computation in Section 2.2 before Section 2.3 shows that DCMI overcomes these issues by pre-computing stencil coefficients at design time across multiple time-steps. Finally, Section 2.4 explains in detail why DCMI uses minimal OCM and contrasts DCMI with similar approaches for CPUs and GPUs.

2.1 Stencil Definition

Equation (1) shows a generic stencil computation over a one-dimensional (1D) array of values V :

$$V_i^{t+1} = \sum_{j=-R}^R c_{j+R} \times V_{i+j}^t \quad (1)$$

The inputs to the computation are the values at time-step t and the output is the value at time-step $t + 1$. The computation in Equation (1) is commonly repeated for all elements in V until a desired number of time-steps have been computed or V has converged according to some convergence criterion [28].

The core of the stencil kernel is a multiply-accumulate operation. Here, the elements of the V -array that are within the neighborhood of the current point i are multiplied with a coefficient c and then accumulated to compute the value for the next time-step V_i^{t+1} . Since the neighborhood is different for different stencil kernels, ISL accelerators must be able to handle different neighborhood definitions. Figure 1 exemplifies the stencil kernel computation with a 1D three-point stencil. Here, each input in the neighborhood of index i is multiplied with a coefficient (i.e., c_0 , c_1 , or c_2) before the results of these multiplications are summed to produce the result V_i^{t+1} . To simplify the explanation, we will refer to the application of a single stencil pattern as S in the following figures.

Figure 1 also shows how a naive stencil computation is performed on an input stream. For simplicity, the input stream contains mostly zeros, but two inputs have the values 1 and 2. In each time-step, the stencil pattern is applied to the V -array to compute the values for the next time-step. For example, the V_4^{t+1} is 5, because $(0 \times 1) + (2 \times 2) + (1 \times 1) = 5$. The radius R of the stencil pattern is the maximum distance from a pattern element to the element at the center of the stencil. Under this definition, the stencil is defined as a maximally sized rectangle around the stencil center. The definition covers arbitrary stencil patterns, since the coefficient of a given neighbor can be 0. Different neighbors commonly have different coefficients. We refer to each coefficient and value

Fig. 2. Explaining why SST [4] and CA [40] use excessive OCM and perform redundant computation.

multiplication as a *stencil result component*. For example, the stencil pattern to compute V_4^{t+1} in Figure 1 has result components (0×1) , (2×2) , and (1×1) .

In this section, we focus on 1D stencils, since they are easy to understand and sufficiently complex to explain the difference between DCMI and prior work. However, many ISL-applications use 2D and 3D representations of the value array V (e.g., References [28, 51, 53]). Extending Equation (1) to cover 2D or 3D value arrays is simple. Concretely, there will be one summation across each dimension and the value array V has one index for each dimension. Thus, the core of the stencil computation is still a multiply-accumulate operation over a set of neighbor values. We show that the DCMI strategy performs equally well for 2D and 3D stencils in Section 5.

2.2 State-of-the-art ISL Acceleration Approaches: SST and CA

ISL accelerators for highly parallel computing substrates such as FPGAs need to extract parallelism from all available sources. The two main sources of parallelism in ISLs are spatial and temporal parallelism. Broadly, two main approaches have been proposed for exploiting both forms of parallelism in FPGA-based accelerators, and Streaming Time-Steps (SST) [4] and the Cone-based Architecture (CA) [40] are representative of these two strategies. In this section, we show that both SST and CA are inefficient, because they perform redundant computation and use OCM inefficiently. We refer to the number of time-steps that an acceleration scheme harvests parallelism from as the iteration depth D .

Figure 2 shows how SST [4] and CA [40] would accelerate a 1D three-point stencil. SST exploits temporal parallelism by instantiating an accelerator core for each time-step and connecting the output of one core to the input of the next core. Figure 2 contains two SST accelerator cores and is therefore able to compute two iterations in a single invocation (i.e., $D = 2$). The first accelerator core retrieves the value array from off-chip DRAM and the last accelerator core writes the output back to off-chip DRAM. Each accelerator core can exploit spatial parallelism by instantiating multiple parallel stencil computing units (marked S in Figure 2).

The SST approach has the desirable characteristic that each element of the value array is fetched once from off-chip DRAM within D iterations. Unfortunately, SST achieves this by buffering all stencil pattern input data in each SST core. For a 1D stencil pattern, this results in SST having to buffer $D \times n$ elements in OCM for an n -point stencil. For 2D (3D) stencil patterns, SST have to buffer D copies of complete rows (planes) of the value array. Since OCM is a limited resource in FPGAs, this severely limits the applicability of the SST approach (see Section 5 for a quantitative analysis).

CA takes a different approach to exploiting temporal parallelism. The key observation is that the dependencies of single output element form a cone through preceding time-steps. CA exploits this observation by creating a tree of stencil compute units that cooperate to compute a single

output element across D iterations. Spatial parallelism is exploited by instantiating multiple trees of stencil compute units that work on different input elements in parallel.

The main problem with the CA approach is that it carries out a number of redundant computations. The root cause of this issue is that CA uses independent units to compute complete output values from the current inputs. Figure 2 exemplifies this by showing how a CA-based accelerator with depth 2 computes V_4^{t+2} and V_5^{t+2} from $\{V_2^t, V_3^t, \dots, V_7^t\}$. The key take-away is that the intermediate results 5 and 4 are part of the computation of both V_4^{t+2} and V_5^{t+2} . Alleviating this problem is challenging, because stencil coefficients are applied to different input elements in each invocation of the stencil pattern. SST suffers from the same redundant computation issue as CA, but it is less visible, because the redundant computations are distributed across the different SST cores.

2.3 Explaining the DCMI Strategy

Our DCMI acceleration strategy avoids both the redundant computation and inefficient OCM usage of CA and SST while (1) exploiting both spatial and temporal parallelism, and (2) fetching each input element from off-chip DRAM exactly once per accelerator invocation. Similar to SST and CA, DCMI streams the value array from DRAM, computes the stencil pattern over D iterations, and writes the output back to DRAM. Unlike SST and CA, DCMI leverages the reconfigurability of the FPGA to fully specialize the accelerator to the stencil pattern of the target ISL application.

To specialize for the stencil pattern of the target application's ISL, DCMI carries out a light-weight pre-processing step prior to accelerator synthesis. The key challenge is to identify the combination of coefficients that are applied to each affected output element after an arbitrary number of iterations. We have devised a simple algorithm for computing these coefficients. Our algorithm first identifies the number of result values affected by a single input, which is $2 \times R \times D + 1$ with stencil pattern radius R and iteration depth D . Second, it creates an input array of this length with a single one as the middle element and zeros elsewhere. Then, we apply the stencil pattern S to this input array over D iterations. The outcome of this procedure is the aggregate coefficient for each affected result value.

Figure 3(a) illustrates how DCMI pre-computes the coefficients after two iterations for the example three-point stencil pattern in Figure 1. As expected, the array contains the original coefficients after the first iteration (see ❶), since the naive stencil kernel computation has a depth of 1. After two iterations, the array contains the combined coefficients of each result element after all intermediate computations (see ❷). Figure 3(a) shows the intermediate computations for the middle element at ❸. To explain the coefficient computation, we view the stencil kernel computation as a dependency graph where the nodes are intermediate values and the edges are coefficients. The input value will be multiplied by 1 on the path to node a and then by 1 again on the path to node d . Similarly, the input is multiplied by 2 on the path to node b and by 2 again on the path to node d . The combined coefficient at node d is $(1 \times 1) + (2 \times 2) + (1 \times 1) = 6$.

Figure 3(a) shows that the resulting DCMI accelerator uses the pre-computed coefficients as constant inputs in a multiply-accumulate circuit at ❹. This enables the DCMI accelerator to compute all possible result components of a single input value. Figure 3(b) illustrates how DCMI would carry out the stencil computation for the example input used in Figure 1. All previous input values that affect the current partial results have been 0 when V_4^t is read. Thus, the partial results from previous computations are all 0. Since the input value is 2, the partial outputs from this invocation is the pre-computed coefficients multiplied by 2 (see ❺). These values are added to the previous partial results to produce the new partial results. At this point, all necessary partial results have been added to V_2^{t+2} , which is the left-most partial result (see ❻). Thus, its computation is complete and it can be written back to DRAM.

Fig. 3. Example explaining our DCMI acceleration strategy. DCMI pre-computes coefficients after D iterations at design-time and thereby avoids redundant computation and excessive use of OCM at runtime.

The next value in the input stream is V_5^t and Figure 3(b) shows how DCMI processes it. First, the partial results array is shifted by one element to remove the completed value, and the newly added partial result is initialized to zero (see ⑦). Then, the results of the coefficient multiplications are added to the shifted partial results to compute the new partial results. Again, the left-most partial result (i.e., V_3^{t+2}) is complete and can be written to DRAM. Since all subsequent inputs are zero in the example input, the remaining partial results have their final values. This is an artifact of the example and does not hold in general. However, it enables us to show that the DCMI computation will give the same results as the naive stencil computation method in Figure 1.

2.4 Why Does DCMI Use Minimal OCM?

A key advantage of the DCMI strategy is that it uses the minimal amount of OCM necessary to enable fetching each element in the V -array once from DRAM for each accelerator invocation. Figure 4 explains how DCMI achieves this by considering two processing steps of a DCMI accelerator that computes a 1D three-point stencil for two array elements with a depth of four. For each time-step of the stencil computation, the input elements have an effect on an increasing number of elements in the value array; the growth rate is determined by the diameter of the stencil pattern. For the three-point stencil example, the inputs can have an effect on four array elements in time-step $t + 1$ and 10 elements in time-step $t + 4$. To enable fetching each element of the value array once from DRAM, DCMI buffers exactly the 10 in-flight partial results that can affect the V -array values at time-step $t + 4$.

Fig. 4. Example explaining why DCMI's buffering strategy uses minimal On-Chip Memory (OCM).

At the end of each processing step, the oldest i elements of the value array are complete (i is the size of DCMI's input buffer). DCMI writes these elements to DRAM to make space for buffering new partial results. Processing Step 2 in Figure 4 exemplifies this. Here, the elements completed in Processing Step 1 have been written to DRAM and DCMI uses these locations for the two new partial results generated in Processing Step 2. In general, DCMI's buffer space requirement is a function of its temporal and spatial parallelism parameters (i.e., the input buffer size and iteration depth of the DCMI accelerator instance). In other words, DCMI's buffer requirements are minimal and constant for each accelerator instance.

This example also highlights how the DCMI strategy differs from stencil computing strategies for CPUs and GPUs. One example is ASLI [18], which is similar to DCMI as it creates a new stencil operator that covers multiple time-steps by convolving the operator with itself. Although this approach enables data reuse within a cone, it suffers from the same redundant computation issue as CA, because it does not enable reuse between cones (see Figure 2). The DCMI strategy may seem to be an attractive alternative, since it avoids redundant computation, but applying it to CPUs or GPUs would likely result in significant overheads as the partial results need to be updated explicitly and sequentially (at least in the source code formulation). This issue does not occur in stencil-specific hardware accelerators, since they do not require operations to be represented as instructions—enabling DCMI to compute and update the partial results in parallel.

3 IMPLEMENTING DCMI ACCELERATORS

3.1 DCMI Accelerator Architecture

Figure 5 shows a high-level block diagram of our DCMI-based accelerator template. Since ISL accelerators are memory intensive, we adopt a decoupled access/execute—or streaming—design strategy [48]. With this strategy, memory access is decoupled from computation, and the two steps operate in parallel using FIFO-buffers for synchronization. The main advantage of the decoupled access/execute strategy is that it automatically overlaps the latency of off-chip memory accesses with computation. This hides the memory access latency and thereby improves compute unit utilization and system throughput.

The execute part of the accelerator is handled by the DCMI Compute Engine (DCE). The DCE is responsible for the core stencil computation and will be discussed in detail in Section 3.2. The access part of the accelerator is implemented with two controllers: The OCM-controller and the DCE-controller. The OCM-controller manages the contents of the OCM, which is mapped to FPGA block RAMs. The OCM is used to provide the FIFO buffers that decouples memory accesses and execution. Similarly, the DCE-controller manages the contents of the register file and the operation of the DCE. The register file stores the input data and partial results generated by the DCE. These

Fig. 5. Generic system-level block diagram of our template for implementing DCMI-based accelerators in FPGAs.

Fig. 6. Block diagram showing the internal structure of the DCMI Compute Engine (DCE) and the register file.

values need to be stored in registers, because they are accessed in a highly parallel fashion to exploit spatial parallelism. The OCM-controller notifies the DCE-controller when new data elements have been fetched from DRAM, and the DCE-controller notifies the OCM-controller when data elements are complete and can be written back to DRAM.

Modern DRAM interfaces are implemented using a multi-dimensional structure including banks, rows and columns [41]. The core structure of the DRAM is a two-dimensional data array. To access this array, the memory controller on the FPGA first provides a row address. Then, the internal DRAM controller transfers a complete row of the data array (i.e., a DRAM page) to a row buffer. Finally, the memory controller provides the column address, and the DRAM controller responds with retrieving the data from the internal row buffer and transmitting it to the FPGA over the memory bus. Subsequent accesses to the same row will have low latency, since the requested data are already available in the row buffer. The maximum throughput of the memory bus can be reached by servicing all memory accesses from the row buffers. Thus, high row locality enables high bandwidth utilization, which in turn enables high throughput. By mapping the innermost dimension of the ISL to contiguous memory addresses, we achieve high row locality and thereby high throughput in the memory subsystem.

3.2 The DCMI Compute Engine

Figure 6 shows the internal structure of the DCE and the register file. The core of the DCE is a hardware circuit that implements the core computation of the ISL. This unit exploits both spatial and temporal parallelism (as exemplified in Section 2.3). It computes each result component by multiplying the current input by a coefficient that was automatically generated at design-time. The coefficients depend on both the stencil computation and the number of iterations the accelerator will harvest parallelism from, and we describe our coefficient generation procedure in detail in Section 3.3. The input buffer contains the input to the stencil computation and is filled by the DCE-controller. The DCE uses the Result Buffer (RB) to store partial results. When all result components of a stencil computation have been completed, the resulting value is written to the output buffer. The DCE-controller retrieves these values and transfers them to the OCM, and the OCM-controller then writes these values to off-chip DRAM.

Algorithm 1 illustrates the main operation of the DCE on a 1D stencil. In the first step, the DCE computes all possible result components for each input data element. These computations are independent of each other and can be carried out in parallel. Each new result component is added to the current partial result value in the RB to produce the new partial result. When the computation has reached its steady state, a number of results at the head of the RB will be final. These values are moved to the output buffer in step 2. Finally, the completed elements are set to

ALGORITHM 1: DCE Computation in 1D

```

Input  $R$  – The radius of the stencil
Input  $D$  – Iteration depth
Input  $I$  – Number of inputs to process concurrently
RB– Result Buffer (storage for partial results)

// Computing convenience constants  $A$  and  $E$ 
 $E = (2 \times R \times D) + I$  // Total RB-entries
 $A = (2 \times R \times D) + 1$  // Affected RB-entries

for Each input window do
    // Step 1: Process each input element
    Load  $I$  data elements into buffer Input
    for  $i = 0$  to  $i = I - 1$  do
        for  $j = 0$  to  $j = A - 1$  do
             $RB[i + j] += \text{Coefficient}[j] \times \text{Input}[i]$ 
        end
    end
    // Step 2: Output completed entries
    for  $i = 0$  to  $i = I - 1$  do
         $\text{Output}[i] = RB[i]$ 
    end
    // Step 3: Rotate result buffer
    for  $i = 0$  to  $E - I - 1$  do
         $RB[i] = RB[i + w]$ 
    end
    for  $i = E - w$  to  $E - 1$  do
         $RB[i] = 0$ 
    end
end

```

ALGORITHM 2: Coefficient Generation in 1D

```

Input  $R$  – The radius of the stencil
Input  $D$  – Iteration depth
Coeff– Coefficient array (Size:  $2RD + 1$ )
IC–Original stencil coefficients (Size:  $2R + 1$ )

// Initialize the Coeff-array
Initialize Coeff with all values equal to 0
Set the middle element of Coeff to 1

// Compute the coefficients to use at runtime
for  $d = 1$  to  $D$  do
     $T = \text{Array of same length as Coeff with all zeros}$ 
    for  $i = 0$  to  $2RD + 1$  do
        for  $j = 0$  to  $2R + 1$  do
             $\text{offset} = i + j - R$ 
            if  $\text{offset} > 0$  and  $\text{offset} < 2RD + 1$  then
                 $T[i] += IC[j] \times \text{Coeff}[\text{offset}]$ 
            end
        end
    end
    for  $i = 0$  to  $2RD + 1$  do
         $\text{Coeff}[i] = T[i]$ 
    end
end
return Coeff

```

zero and the buffer is rotated. We implement the rotation with pointers to avoid the overhead of moving data.

3.3 Computing DCE Coefficients

The key observation that leads to the efficiency of the DCMI strategy is that the impact of a single input element on a single output element across multiple iterations can be collapsed into multiplication by a constant. Since these constants do not depend on the input data, they can be determined at design time and provided to the accelerator. Our key insight is that the impact of a single input element can be computed by repeatedly applying the original stencil to an impulse signal.² Concretely, the impulse array has the value one at its middle position and zeros at all other positions. The length of the impulse array depends on the radius of the stencil pattern and the number of iterations to harvest parallelism from. Recall that the radius R of a stencil pattern is the maximum distance from a stencil element to the center of the stencil.

Algorithm 2 illustrates how we compute the coefficients for a 1D stencil. We first create the impulse array and initialize it to the impulse pattern. Then, we apply the input stencil to the impulse array D times where D is the number of iterations to harvest parallelism from. We use a temporary array T to store the output of each iteration, since it otherwise would overwrite impulse

²The ASLI [18] approach can be used to achieve the same result by mathematically convolving the stencil operator with itself D times and then computing the effective coefficients.

Table 1. Experimental Setup

Item	Resource	Value
FPGA (XC7VX690T)	Logic Cells	693 000
	DSP Slices	3 600
	OCM	52Mbit
Off-chip memory	Capacity	8GB
	Bandwidth	12.8GBPS

values that are needed later. Then, we copy the temporary values into the impulse array at the end of each iteration. After D iterations, the impulse array contains the coefficients that will compute the result component for each affected output when multiplied with the input element.

The input stencil will access data elements outside of the impulse array for elements on the array border. Since the impulse value cannot propagate to these elements, their value will always be zero. Thus, they will not influence the coefficients of the affected outputs. In Algorithm 2, we handle this situation by only applying the stencil computation if the offset is within the bounds of the impulse array. Another possibility is to zero-pad the impulse array and then remove the padding when the coefficient computation is complete. The coefficient calculation is carried out at design time using a regular computer so its performance is not a significant concern.

3.4 Multi-dimensional Stencils and Boundary Conditions

The DCMI approach can be straightforwardly extended to 2D and 3D stencils. The key difference is that the OCM consumption of DCMI increases, since we need to buffer complete rows (planes) in 2D (3D) to compute all possible partial results for each input element while only fetching each input from DRAM once. For our platform and stencils, there is sufficient OCM to enable this. For platforms with less OCM available, DCMI's ability to exploit temporal parallelism will be reduced, since OCM consumption increases with iteration depth. Similarly, we assume that the boundaries of the stencils are sufficiently zero-padded to avoid accesses outside of the input array. Other boundary conditions can be implemented by changing the OCM and DCE controllers appropriately.

4 EXPERIMENTAL SETUP

4.1 FPGA Platform and Tools

We implemented our DCMI accelerators as a Xilinx Vivado HLS template [58]. This enables easily scaling the input size and resource usage of DCMI-based accelerators to the constraints of a given FPGA target. We include a script that generates the coefficients (i.e., implements Algorithm 2). This makes it easy for users to instantiate DCMI accelerators for different stencils as well as experiment with different degrees of spatial and temporal parallelism. The HLS template and the coefficient generation script are available on GitHub.³

Table 1 describes our experimental setup. We used the VC709 development board [59] for our experiments. It contains a Xilinx Virtex 7 FPGA with 52Mbit of on-chip block RAM and 8GB of off-chip DRAM. We use the Xilinx MIG memory controller [56, 57] to connect to the DDR3 memory bus. Our memory controller reorders memory requests to maximize memory bandwidth utilization [41]. We synthesize our accelerators with Xilinx Vivado 2017 [58] on a computer with an Intel Core-i7 8550 processor and 32GB RAM. To measure power, we connect a current meter between

³<https://github.com/magnusjahre/dcmi>.

Fig. 7. The 2D stencil patterns used to evaluate the ISL accelerators in this work. The orange cells designate non-zero coefficients. Our 3D stencils follow the same patterns (e.g., the Jacobi stencil is a complete cube).

the power supply and the FPGA platform and use these measurements to calculate average power consumption.

4.2 Performance Metrics

In this work, we are primarily concerned with throughput, and the key metric is the number of output elements that the accelerators provide per second. To measure this, we augmented the accelerators with two counters: A cycle counter and an output element counter. When the cycle counter reaches a certain number (10 cycles in our experiments), we write the counter values to the host computer over an out-of-band debug interface. Thus, gathering performance numbers does not interfere with the operation of the accelerators.

To fairly compare implementations with different amounts of temporal parallelism (i.e., depth), we use Element iterations Per Second (EPS)⁴ as our throughput metric:

$$\text{EPS} = (D \times e)/t$$

To compute EPS, we multiply accelerator iteration depth D with the number of double precision floating point elements e completed before we divide by execution time t . The reason for using EPS is that the amount of computational progress per output element increases linearly with accelerator depth. We explain the importance of capturing this property with a simple example. Consider an accelerator with depth one that produces a single output element each second for a throughput of 1EPS. If an accelerator with depth 5 produces a single element each second, then it has effectively carried out five times as much work. The EPS metric captures this, since the throughput of the five-iteration accelerator is $(5 \times 1)/1 = 5\text{EPS}$.

Prior work (e.g., Reference [4]) used FLOPs as their performance metric. This is not accurate, because FLOPs is not a work-related metric [1]. More specifically, the redundant computations of CA and SST means that the FLOPs can increase even if the elements completed per second does not.

4.3 Stencils

An ISL accelerator needs to provide high throughput across a range of commonly used stencil patterns. To show that DCMI meets this requirement, we focus on three different stencil patterns that are commonly used in Jacobi methods (see Figure 7(a)), Gauss-Seidel-based partial differential equation solvers (see Figure 7(b)), and the Reverse Time Migration (RTM) kernel in seismic imaging (see Figure 7(c)). For ISL accelerators, the stencil pattern is a key performance-determining factor, since it determines the memory address of the inputs to each stencil computation. In addition, the values of the coefficients can differ significantly between applications. For ISL accelerators, the coefficient values do not have a significant performance impact, since the computation

⁴In our experiments, we report Giga-Element iterations Per Second (GEPS); 1GEPS is equal to 10^9EPS .

is determined by the bit-width of the operands (i.e., single or double precision) and the arithmetic operations (which are determined by the stencil pattern). However, the synthesis tool may be able to exploit symmetries within the coefficients to simplify the generated hardware and thereby reduce FPGA resource consumption. To investigate the impact of this effect, we consider stencils with both symmetric and asymmetric (i.e., randomly generated) coefficients for each pattern.

In 1D, the stencil patterns are lines and only differ with respect to the width of the pattern. Thus, we describe the stencil pattern by its width and if the coefficients are symmetric or asymmetric (e.g., 5S or 5A for a 5-wide symmetric or asymmetric pattern, respectively). In 2D (3D), we use the following stencils:

- Jacobi (JS and JA): The pattern is a 3×3 square (cube) where all coefficients are non-zeros (see Figure 7(a)). Further, the coefficients can be symmetric (JS) or asymmetric (JA).
- Gauss-Seidel (GSS and GSA): The coefficients form an x-shaped pattern within a 3×3 square (cube) with symmetric (GSS) or asymmetric (GSA) coefficients (see Figure 7(b)).
- Reverse Time Migration (RTMS and RTMA): The pattern is a cross within a 5×5 element square (cube) with symmetric (RTMS) or non-symmetric (RTMA) coefficients (see Figure 7(c)).

5 RESULTS

In this section, we first evaluate the throughput and power-efficiency of DCMI and compare it to the state-of-the-art SST [4] and CA [40] acceleration strategies. We carefully tuned SST and CA to maximize performance on our platform. Then, we explore how the input buffer and iteration depth of DCMI—i.e., the degree of spatial and temporal parallelism—should be tuned to maximize throughput. We also analyzed the applicability of DCMI to the floating point speed benchmarks of the SPEC 2017 benchmark suite [34, 49]. Finally, we perform a number of sensitivity analyses to establish how sensitive DCMI is to different architectural parameters. Unless otherwise mentioned, we use a problem size of 200MB and perform 50k time-steps.

5.1 DCMI Performance and Power-Efficiency

5.1.1 Performance. In this section, we analyze the performance attained by our best DCMI accelerators and compare it to the best-performing SST and CA configurations. Recall that we use the GEPS metric to fairly compare acceleration schemes that compute different numbers of loop iterations within a single invocation (see Section 4.2). Figure 8 presents the results for our 1D, 2D, and 3D stencil patterns. The SST-strategy was only applicable to the 1D stencils, because it requires excessive OCM for the 2D and 3D stencils. The problem is that entire lines (planes) must be buffered within each SST core in 2D (3D) (see Figure 2). For the same reason, we had to reduce the problem size to 2MB for the 1D case (the 2D and 3D experiments use the default 200MB dataset). In general, CA and DCMI are insensitive to the problem size (see Section 5.4.1).

Figure 8(a) presents the results for the 1D stencils. Recall that the Jacobi, Gauss-Seidel, and RTM patterns collapse into the same structure in the 1D case. Therefore, the only differences between the stencils are their size (i.e., three, five, and seven points) and if their coefficients are symmetric or asymmetric. The key take-away from Figure 8(a) is that DCMI outperforms SST and CA by a significant margin for all stencils. For 3P-A, DCMI performs 2.8× and 2.5× better than CA and SST, respectively. Also, DCMI performs better on symmetric stencils compared to asymmetric stencils, since it exposes the computational symmetries to the synthesis tool and thereby enable low-level optimizations. SST allocates a separate core to each iteration, which means that the synthesis tool cannot optimize across iterations. Similarly, CA allocates a separate cone to each input element.

Fig. 8. Comparing DCMI performance and power-efficiency to SST and CA for 1D, 2D, and 3D stencil patterns with symmetric and asymmetric coefficients. *The key take-away is that DCMI outperforms SST and CA for all dimensions and stencil patterns while maintaining similar power-efficiency.*

Figure 8(c) and (e) shows the results for the 2D and 3D stencils, respectively. DCMI still outperforms CA by a significant margin (up to $7.7\times$ and $3.7\times$ across the 2D and 3D stencils, respectively). Again, DCMI performs better for symmetric stencils (i.e., JS, GSS, and RTMS) than the asymmetric stencils (i.e., JA, GSA, and RTMA) by enabling optimizations during synthesis.

CA performs very similarly for JA, JS, GSA, and GSS in 2D and 3D. The only difference between these stencils is that JA and JS only have non-zero coefficients while some coefficients are zero in GSA and GSS. CA is memory-bound for these stencils, because it uses the FPGA compute resources ineffectively (i.e., it performs redundant computations). In other words, the difference in computational characteristics does not matter, because the FPGA is not the bottleneck resource. DCMI is compute-bound. Thus, its performance is lowest for JA—since it has only non-zero coefficients and no symmetry among them—and highest for GSS—since four of its coefficients are zero and there are symmetries that the synthesis tool can exploit.

Figure 8 shows that absolute performance goes down for both CA and DCMI when the dimensions of the stencil increases. The reason is that the amount of computation and the number of memory accesses increase significantly. For CA, performance goes down, because more memory accesses are needed for each stencil computation while for DCMI the increasing amount of computation is the key issue.

5.1.2 Power-Efficiency. Figure 8 also reports the power-efficiency of DCMI and compares it to that of SST (for the 1D stencils) and CA (for all stencil dimensions). We use GEPS per watt (GEPS/W) as our power-efficiency metric. A key take-away from Figure 8 is that the power-efficiencies of DCMI, SST, and CA are similar. The main reason is that power consumption increases proportionally with the number of FPGA compute elements that are actively used.⁵ Since we optimized for performance when configuring the schemes, the higher resource efficiency of DCMI leads to it consuming more power—and significantly outperforming the other schemes.

Figure 8 also shows that DCMI increases power consumption proportionally to the performance improvement—since power-efficiency is similar to CA and SST. In other words, DCMI is power-scalable. This means that DCMI can be used in contexts where the power budget is fixed by limiting resource consumption to meet the power consumption target (at the cost of lower performance). For the symmetric stencils, DCMI is more power-efficient than CA. The reason is again that DCMI exposes the symmetries of the stencil computation to the synthesis tool. CA is unable to do this, resulting in similar performance and power-efficiency for symmetric and asymmetric stencils.

5.2 Balancing Spatial and Temporal Parallelism

Figure 9 shows the performance effects of varying the input buffer size and iteration depth for our 2D stencils. The unit of the buffer size is the number of double precision floating point elements, and we vary this between 10 and 160. We investigate the impact of varying the iteration depth from 1 to 6. The input buffer size determines the number of elements that DCMI processes in parallel—i.e., the degree of spatial parallelism—and the depth parameter determines the number of iterations to compute in a single invocation—i.e., the degree of temporal parallelism.

Figure 9 shows that the input buffer size and iteration depth parameters have a significant impact on performance and that they must be set to mutually compatible values. The reason is that they collectively determine how well the FPGA resources are utilized. Overall, resource consumption increases when the number of inputs increase and when the iteration depth increases. When the FPGA is underutilized, increasing resource consumption improves performance. Conversely, adding more parallelism when the FPGA is close to full utilization causes the synthesis tool to multiplex different computations onto bottleneck resources, which reduces performance.

The magnitude of these effects differs between stencils (e.g., comparing Figure 9(a) and Figure 9(f)). The key insight is that increasing iteration depth has a coarse-grained impact on parallelism and resource consumption while the input buffer size has a more fine-grained effect. By carefully tuning these parameters, we are able to generate DCMI accelerators that efficiently utilizes the available FPGA resources. Unfortunately, the interactions between the DCMI accelerator specification and the synthesis tool are complex and depend on both the particular stencil pattern and the characteristics of the target FPGA. Thus, we need to perform a parameter sweep to determine an appropriate input buffer size and iteration depth for a given FPGA and stencil.

This requirement is problematic since evaluating each design point requires synthesizing a DCMI accelerator, and synthesizing a single DCMI accelerator takes on average 1.5, 3.5, and 4.5 hours for the 1D, 2D, and 3D stencils, respectively. Fortunately, Figure 9 shows two properties that can be leveraged to efficiently identify a near-optimal configuration point. First, performance is maximized at a single input buffer size, and this size is around 50 elements. Second, performance generally increases monotonically with iteration depth until reaching a maximum value. Then, performance decreases monotonically.

⁵The clock frequency is roughly 200MHz for all schemes and stencils.

Fig. 9. Performance versus input buffer size and iteration depth for the 2D stencils. *Depth and buffer size control the degree to which DCMI exploits temporal and spatial parallelism. Thus, they need to be set to mutually compatible values to best utilize FPGA resources—thereby maximizing performance.*

We leverage these observations to develop a search heuristic for identifying a near-optimal DCMI accelerator configuration. We start with an iteration depth of 1 and a buffer size of 50 elements. Then, we successively increase the iteration depth by one until we see a drop in performance, and choose the evaluated iteration depth that maximizes performance. We then evaluate the performance of this iteration depth for the neighboring buffer sizes. Depending on which configuration performs best, we proceed with increasing (decreasing) the size of the buffer until performance decreases. Finally, we return the depth and buffer size of the highest performing configuration we have evaluated.

We applied our heuristic to the results in Figure 9. On average, the heuristic finds configurations that perform within 93% of the highest performing configuration found using exhaustive exploration. On average, it evaluates 11.5 of 66 design points and thereby reduces synthesis time by 5.8 $\times$ (i.e., from 231 hours to 40 hours). In the worst case, the heuristic evaluates 15 design points, which results in a synthesis time of 53 hours.

Table 2. Applicability of DCMI to the SPEC 2017 Floating Point Speed Benchmarks

Benchmark	Procedure	Perc. of Exec. Time	Polynomial ISL?	Accel. Complexity
bwaves	shell	54%	Yes	Low
	jacobian	36%	Yes	Low
cactBSSN	ML_BSSN_RHS_Body	41%	Yes	High
	ML_BSSN_SelectBoundConds	30%	No	N/A
lbm	main	100%	Yes	Low
wrf	calc_aerosol_goddard_sw	78%	Yes	Medium
pop2	hdiff_t_gm	26%	Yes	High
imagick	RotateKernelInfo	89%	No	N/A
nab	ephi	88%	Yes	Low
fotonik3d	UPML_set_eps_arrays	36%	Yes	Low
	power_interH	27%	No	N/A
roms	step2d_tile	26%	Yes	High
Average		54%		

5.3 ISLs with Linear Polynomial Stencils

We have now established that DCMI generates very efficient accelerators for ISL kernels. We now investigate how common ISLs with linear polynomial stencils are (i.e., the stencils that DCMI is applicable to). For this study, we consider the floating point speed benchmarks from SPEC CPU 2017 [34, 49].⁶ We first profiled the benchmarks with gprof and the reference input sets. Then, we manually inspected the performance-critical functions to identify polynomial ISLs. We consider procedures that account for more than 20% of the total execution time of the benchmark to be performance-critical.

Table 2 summarizes the results of this investigation. The key take-away is that the linear polynomial ISLs targeted by DCMI are common as 9 of 12 performance-critical procedures contain them. Further, the procedures containing polynomial ISLs account for 54% of execution time on average, and the performance impact is significant for bwaves, lbm, wrf, and nab. This reinforces that ISL acceleration schemes such as DCMI have wide applicability.

We also assess the complexity of accelerating the benchmarks. For bwaves, lbm, nab, and fotonik3d, we expect that generating DCMI-based accelerators will be relatively straightforward. The reasons are that the ISLs can be easily identified in the source code and that only a limited amount of CPU-specific optimizations (e.g., manual loop unrolling) have been applied. That said, applying DCMI to complete benchmarks require a non-negligible engineering effort, since we would need to implement an efficient mechanism for transferring input and output data between the host processor and the FPGA platform. Further, the benchmarks would typically require some refactoring to clearly separate the ISL computation from management code. For these reasons, we leave DCMI-based acceleration of complete benchmarks as further work.

5.4 Sensitivity Analyses

We have now established that DCMI significantly outperforms the current state-of-the-art ISL acceleration approaches. In this section, we analyze the sensitivity of DCMI performance to different problem sizes and a range of architectural parameters.

⁶We consider all SPEC17 floating point speed benchmarks except cam4 as it was incompatible with our setup.

Fig. 10. Normalized performance versus problem size for DCMI and the asymmetric 2D stencils.

Fig. 11. Performance versus problem size for SST, CA, and DCMI and the 7PS 1D stencil.

Fig. 12. Off-chip bandwidth utilization for different stencil patterns.

Fig. 13. Performance with FPGA resource consumption constraints for the 2D stencils.

5.4.1 Sensitivity to Problem Size. Figure 10 shows the impact of varying the problem size for DCMI and the asymmetric 2D stencils. Since the absolute performance of each stencil pattern is different, we normalize to the problem size with best performance for each stencil. As aforementioned, our default problem size is 0.2GB. The key take-away is that DCMI is not sensitive to variations in problem size. The reason is that DCMI adopts a streaming decoupled access/execute design strategy and is therefore able to fill the computational pipeline regardless of problem size (see Section 3). Note that GEPS is a throughput metric. Thus, it will take DCMI longer to complete a given number of iterations when the problem size increases—but its computational efficiency remains the same.

Figure 10 shows that DCMI is equally efficient across a large range of problem sizes for 2D stencils. Figure 11 considers the 1D 7PS stencil and compares the performance of DCMI to SST and CA. Both CA and SST are streaming architectures, and CA scales equally well as DCMI albeit performs considerably worse due to redundant computations. SST performs similarly to CA when the problem size is small, but does not scale well. The reason is that the on-chip memories SST uses to buffer intermediate results become the performance bottleneck. Thus, it is unable to saturate the off-chip bandwidth for larger problem sizes than 2.5MB, resulting in overall throughput loss.

5.4.2 Off-Chip Bandwidth Utilization. Figure 12 plots the off-chip bandwidth utilization of our DCMI accelerators relative to the maximum achievable bandwidth of the DDR3 interface on our platform (i.e., 12.8GBPS). It shows that DCMI is close to being memory-bound for the 1D stencils and compute-bound for 2D and 3D stencils. It also shows that our implementation is able to almost fully utilize the available memory bandwidth in the memory-bound case (up to 96% utilization for RTMS). The reason for this high utilization is that ISLs have a DRAM-friendly sequential access pattern—resulting in nearly all memory requests being DRAM page hits. For the 2D and 3D stencils, Figure 12 shows that DCMI uses memory bandwidth efficiently and that there is a potential for improving performance by using a larger FPGA.

Fig. 14. OCM consumption in percentage of OCM capacity and DCMI latency in clock cycles across all stencils. The key take-aways are that DCMI uses significantly less OCM than SST and CA and that its latency is negligible.

5.4.3 Resource Consumption and Latency. We have so far considered DCMI configurations that fully utilize our target FPGA. Thus, it is interesting to evaluate how DCMI performs as a function of resource consumption. We do this by instructing the synthesis tool to not use more than a given percentage of all available resources except I/O pins (e.g., flip-flops, look-up tables, and DSP slices). Figure 13 shows that the performance of DCMI increases nearly linearly with increasing resource usage—illustrating that DCMI is a scalable approach on our platform. This is expected, since DCMI is compute-bound for our 2D stencils. For platforms where the FPGA is large compared to the available memory bandwidth, we expect that performance will saturate when DCMI becomes memory-bound.

Figure 14 reports the OCM consumption for the 1D, 2D, and 3D stencils and DCMI latency for the 2D and 3D stencils. A key take-away is that SST and CA are limited by the amount of OCM available while DCMI is not. Thus, DCMI's more efficient use of OCM enables leveraging more of the other FPGA resources and thereby substantially improve performance. Figure 14(d) shows that the latency for DCMI to provide the first result is negligible. More specifically, the latency is roughly 200 clock cycles (i.e., $1\mu\text{s}$ at 200MHz).

6 RELATED WORK

In this section, we provide an overview of prior approaches for efficiently implementing ISLs and ISL accelerators. Since DCMI targets an FPGA realization, we give most attention to FPGA-based ISL accelerators. The DCMI approach also has some similarities to high-performance ISL implementation approaches for CPUs and GPUs, and we briefly cover these as well.

6.1 FPGA-based ISL Accelerators

The cone-based ISL acceleration approach is the most similar approach to DCMI described in the literature. We have chosen CA [2, 29, 40] as the representative of this class as they share our goal of providing an automatic design flow. Zohouri et al. [60] takes an approach that is architecturally similar to CA but realized through OpenCL. The CA-based approaches are limited by redundant computations, and Wang and Liang [54] propose to reduce the impact of this problem by

enabling data sharing between processing elements (which increases design complexity). DCMI is fundamentally different from these approaches as it leverages reconfigurability to remove redundant computations with an architecturally simple design. In Section 5, we compared to CA and showed that DCMI improves performance by up to $7.7\times$ for our 2D stencils.

Another approach for implementing stencil accelerators is to instantiate a processing element for each iteration and use OCM to buffer intermediate computations. We chose SST [4, 37] as representative for this class, since it is a recent, generic ISL accelerator generation framework. Sano et al. [42, 44] proposes a similar accelerator architecture and explores its application to multi-FPGA systems [43]. Cong et al. [7] showed that the streaming approach has the desirable property that it enables optimal data reuse. Unfortunately, this property comes at the cost of excessive use of OCM for 2D and 3D stencils. SODA [6] reduces the impact of this problem by proposing a mechanism that minimizes OCM usage at the cost of increased design complexity. In contrast to DCMI, SODA still performs redundant computation.

A related line of research explores how to identify good configurations for SST-like accelerators. Nagasu et al. [30] investigated the relative impact of spatial and temporal parallelism in the context of an FPGA-accelerated Tsunami simulator, while Dohi et al. [10] investigates how to configure a heat conduction simulator with Maxeller's MaxCompiler. These approaches are closely tied to the SST acceleration strategy and has limited applicability to DCMI-based accelerators.

A plethora of application-specific ISL accelerators have been proposed. Takei et al. [51] presents an OpenCL-based implementation of a finite differences time domain ISL that exploits temporal parallelism. The initial accelerator had some performance issues that the authors overcame in later works [52, 53]. In addition, a number of stencil-specific ISL accelerators that only exploit spatial parallelism have been proposed [11, 12, 17, 21, 26, 27, 32, 46]. These works do not use architectural approaches that are fundamentally different from CA and SST. In addition, the overhead of implementing stencil-specific accelerators is significant. Thus, a better approach—and the one taken by us, CA, and SST—is to provide ISL accelerator generation approaches.

High-dimension ISLs have significant memory requirements. This has led a number of researchers to propose ISL-specific memory system techniques. Escobedo et al. [13] determine the optimal memory banking structure in the context of HLS systems, while Jin and Bakos [19] propose a custom memory system for 3D stencils that enable efficient reuse and latency-hiding in a multi-FPGA system. Shafiq et al. [47] take a similar approach and develop a custom memory system for 3D stencils that maximise reuse, while Fu and Clapp [14] explores algorithmic and architectural strategies reducing off-chip memory accesses in 3D RTM. In contrast, DCMI improves computational efficiency to the extent that both 2D and 3D stencils become compute-bound—thereby significantly reducing the need for custom memory systems.

6.2 Domain Specific Languages and Code-Generators for ISLs

The difficulty of efficiently implementing ISL algorithms combined with their widespread use has led a number of researchers to propose code-generation approaches and Domain Specific Languages (DSLs) for ISLs. Deest et al. [9] propose an HLS-based code-generator that generates accelerators that are architecturally similar to CA. Similarly, Natale et al. [31] propose an accelerator generation approach for SST. In addition, DSLs such as Halide [38] and Delite [50] are applicable to ISLs. Our work is complementary to these approaches as DCMI can be used to generate highly efficient FPGA-based accelerators in the back-ends of these systems.

6.3 CPUs, GPUs, and ASIC Accelerators

A number of approaches that optimize ISLs for CPUs and GPUs combine computations from different loop levels to reduce the amount of redundant computation. ASLI [18] is an application-level

technique that creates a new stencil operator that covers multiple time-steps by convolving the original stencil operator with itself two or more times. The compiler optimizations loop unrolling (e.g., References [22, 45, 61]) and forward substitution (e.g., Reference [24]) can be used to achieve similar gains. These optimizations differ from DCMI, because they only reduce the amount of redundant computation within a cone (see Section 2.4). In contrast, DCMI removes redundant computation between cones and enables the synthesis tool to reduce redundant computation within cones.

ISL implementations on GPUs and CPUs also exploit both spatial and temporal parallelism. This can be achieved through optimizations such as time-skewing [55] and wave-front scheduling. For example, Rahman et al. [39] performed a detailed analysis of highly optimized stencil codes on multi-cores, while Datta et al. [8] provides an approach that uses autotuning to efficiently implement the 3D heat equation on multi-core CPUs. Examples for GPUs include Meng et al. [25] who presents a model-based approach that appropriately trades off recomputation against synchronization costs. Another example is Holewinski et al. [16] who provide a stencil code generator for GPUs that exploit time-skewing. These approaches cleverly schedule computations at runtime to exploit temporal and spatial parallelism while satisfying all dependencies. In contrast, DCMI generates custom hardware for a specific ISL at design time that respects all dependencies. In other words, DCMI leverages the reconfigurability of FPGAs to avoid complex scheduling at runtime.

Researchers have also proposed ASIC-based accelerators that can be used to implement ISLs. The Convolution Engine [36] is an accelerator for computational photography, image processing, and video processing. Similarly, Hameed et al. [15] implement a custom H.264 accelerator to better understand the sources of inefficiencies in general-purpose processors. DCMI differs from these approaches in that it exploits reconfigurable hardware to generate stencil-specific accelerators.

7 CONCLUSION

In this work, we have presented our DCMI accelerator generation scheme for ISLs with polynomial stencils. DCMI improves upon prior work by exploiting both temporal and spatial parallelism while using minimal on-chip memory and avoiding redundant computation. Our evaluation shows that DCMI accelerators can be effectively configured to fully utilize a target FPGA—and thereby maximize throughput—by adjusting the amount of temporal and spatial parallelism. More specifically, DCMI's iteration depth parameter determines the amount of temporal parallelism and the input buffer size determines the amount of spatial parallelism. We provide an efficient search heuristic that identifies a near-optimal DCMI configuration while reducing design space exploration time by $5.8\times$ compared to an exhaustive parameter sweep. DCMI performs very well. Concretely, it improves throughput by up to $7.7\times$ compared to the state-of-the-art CA ISL acceleration scheme.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions.

REFERENCES

- [1] A. R. Alameldeen and D. A. Wood. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro* 26, 4 (2006), 8–17.
- [2] Ivan Beretta, Vincenzo Rana, Abdulkadir Akin, Alessandro Antonio Nacci, Donatella Sciuto, and David Atienza. 2016. Parallelizing the Chamblolle algorithm for performance-optimized mapping on FPGA devices. *ACM Trans. Embed. Comput. Syst.* 15, 3 (2016), 1–27.
- [3] Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5 (2011), 67.
- [4] Riccardo Cattaneo, Giuseppe Natale, Carlo Sicignano, Donatella Sciuto, and Marco Domenico Santambrogio. 2015. On how to accelerate iterative stencil loops: A scalable streaming-based approach. *ACM Trans. Archit. Code Optim.* 12, 4 (2015), 53:1–53:26.

- [5] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2016. A cloud-scale acceleration architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 1–13.
- [6] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18)*.
- [7] 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 *Proceedings of the Design Automation Conference (DAC'14)*. 77:1–77:6.
- [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 *Proceedings of the Conference on Supercomputing (SC'08)*. 4:1–4:12.
- [9] G. Deest, T. Yuki, S. Rajopadhye, and S. Derrien. 2017. One size does not fit all: Implementation trade-offs for iterative stencil computations on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–8.
- [10] K. Dohi, K. Fukumoto, Y. Shibata, and K. Oguri. 2013. Performance modeling and optimization of 3-D stencil computation on a stream-based FPGA accelerator. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig'13)*. 1–6.
- [11] B. A. Draper, J. R. Beveridge, A. P. W. Bohm, C. Ross, and M. Chawathe. 2002. Implementing image applications on FPGAs. In *Object Recognition Supported by User Interaction for Service Robots*, Vol. 3. 265–268.
- [12] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, M. S. Mirotznik, and D. W. Prather. 2003. Hardware implementation of a three-dimensional finite-difference time-domain algorithm. *IEEE Antennas Wireless Propagat. Lett.* 2 (2003), 54–57.
- [13] Juan Escobedo and Mingjie Lin. 2018. Graph-theoretically optimal memory banking for stencil-based computing kernels. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. 199–208.
- [14] Haohuan Fu and Robert G. Clapp. 2011. Eliminating the memory bottleneck: An FPGA-based solution for 3D reverse time migration. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'11)*. 65–74.
- [15] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*. 37–47.
- [16] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the International Conference on Supercomputing (SC'12)*. 311–320.
- [17] T. Hussain, M. Pericàs, N. Navarro, and E. Ayguadé. 2011. Implementation of a reverse time migration kernel using the HCE high level synthesis tool. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'11)*. 1–8.
- [18] G. C. Januario, B. S. Rosenburg, Y. Park, M. Perrone, J. Moreira, and T. C. M. B. Carvalho. 2016. Speeding up stencil computations with kernel convolution. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'16)*. 76–83.
- [19] Zheming Jin and Jason D. Bakos. 2015. Memory interface design for 3D stencil kernels on a massively parallel memory system. *ACM Trans. Reconfig. Technol. Syst.* 8, 4 (2015), 24:1–24:24.
- [20] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. 2017. Moonwalk: NRE optimization in ASIC clouds. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 511–526.
- [21] R. Kobayashi, S. Takamaeda-Yamazaki, and K. Kise. 2012. Towards a low-power accelerator of many FPGAs for stencil computations. In *Proceedings of the International Conference on Networking and Computing (ICNC'12)*. 343–349.
- [22] A. Koseki, H. Komastu, and Y. Fukazawa. 1997. A method for estimating optimal unrolling times for nested loops. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*. 376–382.
- [23] I. Kuon and J. Rose. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 26, 2 (2007), 203–215.
- [24] Zhiyuan Li and Yonghong Song. 2004. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.* 26, 6 (2004), 975–1028.
- [25] Jiayuan Meng and Kevin Skadron. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the International Conference on Supercomputing (SC'09)*. 256–265.
- [26] E. Motuk, R. Woods, and S. Bilbao. 2005. Implementation of finite difference schemes for the wave equation on FPGA. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'05)*, Vol. 3. 237–240.
- [27] E. Motuk, R. Woods, S. Bilbao, and J. McAllister. 2007. Design methodology for real-time FPGA-based sound synthesis. *IEEE Trans. Sign. Process.* 55, 12 (2007), 5833–5845.
- [28] Jacek Myczkowski and Guy Steele. 1991. Seismic modeling at 14 gigaflops on the connection machine. In *Proceedings of the Conference on Supercomputing (SC'91)*. 316–326.

- [29] Alessandro Antonio Nacci, Vincenzo Rana, Francesco Bruschi, Donatella Sciuto, Ivan Beretta, and David Atienza. 2013. A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices. In *Proceedings of the Design Automation Conference (DAC'13)*. 1–6.
- [30] K. Nagasu, K. Sano, F. Kono, N. Nakasato, A. Vazhenin, and S. Sedukhin. 2016. Parallelism for high-performance tsunami simulation with FPGA: Spatial or temporal?. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. 30–30.
- [31] G. Natale, G. Stramondo, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio. 2016. A polyhedral model-based framework for dataflow implementation on FPGA devices of iterative stencil loops. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'16)*. 1–8.
- [32] Koji Okina, Rie Soejima, Kota Fukumoto, Yuichiro Shibata, and Kiyoshi Oguri. 2016. Power performance profiling of 3-D stencil computation on an FPGA accelerator for efficient pipeline optimization. *SIGARCH Comput. Archit. News* 43, 4 (2016), 9–14.
- [33] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-defined accelerator for large-scale DNN systems. In *Proceedings of the Hot Chips Symposium (HCS'14)*. 1–23.
- [34] R. Panda, S. Song, J. Dean, and L. K. John. 2018. Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon? In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'18)*. 271–282.
- [35] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*. 13–24.
- [36] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. 2013. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*. 24–35.
- [37] M. Rabozzi, G. Natale, B. Festa, A. Miele, and M. D. Santambrogio. 2017. Optimizing streaming stencil time-step designs via FPGA floorplanning. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–4.
- [38] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2017), 106–115.
- [39] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. 2011. Understanding stencil code performance on multicore architectures. In *Proceedings of the International Conference on Computing Frontiers (CF'11)*. 1–10.
- [40] V. Rana, I. Beretta, F. Bruschi, A. A. Nacci, D. Atienza, and D. Sciuto. 2016. Efficient hardware design of iterative stencil loops. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 35, 12 (2016), 2018–2031.
- [41] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of the International Symposium on Computer Architecture (ISCA'00)*. 128–138.
- [42] K. Sano, Y. Hatsuda, and S. Yamamoto. 2011. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*. 234–241.
- [43] K. Sano, Y. Hatsuda, and S. Yamamoto. 2014. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Trans. Parallel Distrib. Syst.* 25, 3 (2014), 695–705.
- [44] Kentaro Sano, Satoru Yamamoto, and Yoshiaki Hatsuda. 2011. Domain-specific programmable design of scalable streaming-array for power-efficient stencil computation. *SIGARCH Comput. Archit. News* 39, 4 (2011), 44–49.
- [45] Vivek Sarkar. 2000. Optimized unrolling of nested loops. In *Proceedings of the International Conference on Supercomputing (ISC'00)*. 153–166.
- [46] M. Schmidt, M. Reichenbach, and D. Fey. 2012. A generic VHDL template for 2D stencil code applications on FPGAs. In *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. 180–187.
- [47] M. Shafiq, M. Pericàs, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadé. 2009. Exploiting memory customization in FPGA for 3D stencil computations. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'09)*. 38–45.
- [48] James E. Smith. 1984. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.* 2, 4 (1984), 289–308.
- [49] SPEC. 2017. SPEC CPU 2017. Retrieved from <https://www.spec.org/cpu2017/>.

- [50] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s (2014), 1–25.
- [51] Yasuhiro Takei, Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Michitaka Kameyama. 2015. FPGA-oriented design of an FDTD accelerator based on overlapped tiling. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'15)*. 72–77.
- [52] H. M. Waidyasooriya and M. Hariyama. 2016. FPGA-based deep-pipelined architecture for FDTD acceleration using OpenCL. In *Proceedings of the International Conference on Computer and Information Science (ICIS'16)*. 1–6.
- [53] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama. 2017. OpenCL-based FPGA-platform for stencil computation and its optimization methodology. *IEEE Trans. Parallel Distrib. Syst.* 28, 5 (2017), 1390–1402.
- [54] S. Wang and Y. Liang. 2017. A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model. In *Proceedings of the Design Automation Conference (DAC'17)*. 1–6.
- [55] David Wonnacott. 2002. Achieving scalable locality with time skewing. *Int. J. Parallel Program.* 30, 3 (2002), 181–221.
- [56] Xilinx. 2010. Memory interface solutions. Retrieved from https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf.
- [57] Xilinx. 2012. 7 series FPGAs memory interface solutions. Retrieved from https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v1_4/ug586_7Series_MIS.pdf.
- [58] Xilinx. 2018. Vivado high-level synthesis. Retrieved from <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>.
- [59] Xilinx. 2019. Retrieved from VC709 evaluation board for the Virtex-7 FPGA – User guide. https://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf.
- [60] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'18)*. 153–162.
- [61] G. Zumbusch. 2012. Tuning a finite difference computation for parallel vector processors. In *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC'12)*. 63–70.

Received April 2019; revised June 2019; accepted July 2019