Exploiting Computation Reuse for Stencil Accelerators

Yuze Chi
University of California, *Los Angeles*
chiyuze@cs.ucla.edu

Jason Cong
University of California, *Los Angeles*
cong@cs.ucla.edu

***Abstract*—Stencil kernel is an important type of kernel used extensively in many application domains. Over the years, researchers have been studying the optimizations on parallelization, communication reuse, and computation reuse for various target platforms. However, challenges still exist, especially on the computation reuse problem for accelerators, due to the lack of complete design-space exploration and effective design-space pruning. In this paper, we present solutions to the above challenges for a wide range of stencil kernels (i.e., stencil with reduction operations), where the computation reuse patterns are extremely flexible due to the commutative and associative properties. We formally define the complete design space, based on which we present a provably optimal dynamic programming algorithm and a heuristic beam search algorithm that provides near-optimal solutions under an architecture-aware model. Experimental results show that for synthesizing stencil kernels to FPGAs, compared with state-of-the-art stencil compiler without computation reuse capability, our proposed algorithm can reduce the look-up table (LUT) and digital signal processor (DSP) usage by 58.1% and 54.6% on average respectively, which leads to an average speedup of 2.3× for compute-intensive kernels, outperforming the latest CPU/GPU results.**

I. Introduction

Stencil computation [1] is often intuitively defined as the type of computation that uses a sliding window of the input array to compute the output array. Such computation patterns are widely used in many areas, including image processing (e.g., [2], [3]) and solving partial differential equations (e.g., [4]). Although the concept itself is relatively simple, it is non-trivial to optimize for performance. Researchers have been optimizing stencil kernels in three aspects. The first is *parallelization*. Stencil computation has a large degree of inherent parallelism, but the sliding window access pattern and the dependencies among elements in different stages make it hard to fully utilize the available parallelism [4]–[8]. The second is *communication reuse*. The sliding window pattern makes it possible to reuse input data and reduce external memory communication. On instruction-based processors (CPU, GPU), this translates into improving locality [4], [9] and reducing inter-core communication [10]. On accelerators (FPGA, ASIC) where data paths can be fully customized, the communication reuse problem can be optimally solved [11], [12]. The third is *computation reuse*. Although stencil kernels often consist of multiple stages or iterations and are compute-intensive, almost all such stencil kernels perform commutative and associative reduction operations, thus making it possible to reuse some of the computation [2], [13]–[17]. As a motivating example, for a 17×17 kernel used in calcium image stabilization [2], the number of multiplication operations can be dramatically reduced from 197 to only 30, while yielding the same throughput. However, that design was done with extensive manual optimization. Our goal is to automate such optimization process.

Unlike the parallelization problem and the communication reuse problem which have been optimally solved in [12], there are still major challenges that have not been systematically addressed for the computation reuse problem. One such challenge is that most stencil compilers [13]–[17] are designed for instruction-based processors and do not explore the complete design space for computation reuse, due to the fact that parallelization and communication reuse have more impact on performance and computation reuse is often just a by-product [4], [9]. However, for accelerators, computation reuse can be fully decoupled from parallelization and communication reuse via datapath customization. An ideal stencil compiler for accelerators should be capable of finding the optimal computation reuse if possible. Moreover, since no stencil compiler uses an accelerator-oriented model to evaluate the computation-storage trade-off, it is hard to guide the design-space pruning and find the best solution. This presents another challenge. In this paper, we present solutions to the challenges mentioned above. Our major contributions include:

- **Complete Design-Space Exploration:** We formally define the problem of computation reuse for stencil with reduction operations, under which we present a dynamic programming algorithm that can find the optimal computation reuse pattern.
- **Optimality-Preserving Heuristics:** We present a heuristic beam search algorithm that significantly prunes the evaluated design points while producing near-optimal results, which scales well for large kernels. We also present an architecture-aware metric to enable quantitative analysis of the computation-storage trade-off.
- **Fully Automated Design Flow:** We implement our algorithm based on the open-source SODA compiler [12] and perform design-space exploration and code generation in a fully automated way. Since SODA is designed to be suitable as an intermediate representation for stencil applications, other projects using SODA as a backend (e.g., [18], [19]) can benefit from our work, too.
- **Extensive Experiments:** We evaluate our presented compiler with various artificial and real-world kernels on a state-of-the-art FPGA platform. Post-synthesis results on FPGAs show that on average our proposed algorithm can reduce look-up table (LUT) and digital signal processor (DSP) usage by 58.1% and 54.6%, respectively, compared with the state-of-the-art SODA stencil compiler [12] without computation reuse capability. For compute-intensive stencils, our algorithm achieves an average speedup of 2.3×.

II. Background

Stencil computation kernels can be defined as kernels that compute output data elements using a multidimensional input array according to some fixed, local pattern.

```
void Jacobi(const float X[N][M], float Y[N][M]) {
  for (int j = 1; j < N - 1; j++)
    for (int i = 1; i < M - 1; i++)
      Y[j][i] = (X[j - 1][i] + X[j][i - 1] + X[j][i]+
                 X[j][i + 1] + X[j + 1][i]) * 0.2f;
}
```

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

As an example, Listing 1 shows a 5-point, 2-dimensional Jacobi kernel on an $M \times N$ input X that computes output Y. In general, an n-point, m-dimensional stencil kernel A defines a spatial window $\{\vec{a}_s | s = 0, \dots, n-1\}$ and a function to compute the output at spatial coordinate $\vec{y} = (y^{(0)}, \dots, y^{(m-1)})$ by consuming inputs at

$\{\vec{x}_s \equiv \vec{y} + \vec{a}_s | s = 0, \ldots, n-1\}$. $\vec{a}_s$ denotes the offset between the s-th input and the output. Since all multidimensional indices are eventually converted to 1-dimensional indices [12], when there is no ambiguity, we will omit the vector notation on top of the coordinates.

Reduction operations are operations that are *commutative* and *associative*[1]. Given an n-point stencil kernel with reduction operations

$$Y[y] = g(f_0(X[y+a_0]) \oplus ... \oplus f_{n-1}(X[y+a_{n-1}]))$$

The reduction *expression* we are interested in is

$$f_0[a_0] \oplus ... \oplus f_{n-1}[a_{n-1}] \tag{1}$$

where $f_s[a_s] = f_s(X[y+a_s])$, meaning to apply a pointwise scaling function f_s on the input element in X with an offset of a_s relative to the output element. We will, e.g., use $[-1][0]$ or $X[-1][0]$ to represent $X[j-1][i]$ when it is clear from the context.

Computation reuse is a well-known concept in compiler optimization, more commonly known as common subexpression elimination (CSE). The classical CSE technique is based on expression analysis of the program or value numbering. For example, to evaluate two expressions `x=a*b+c` and `y=a*b+d`, a compiler is expected to find that the two expressions for `x` and `y` share the same subexpression `a*b`, which can be evaluated only once by evaluating a new expression `tmp=a*b` before `x=tmp+c` and `y=tmp+d`.

While the classical CSE is powerful and effective, we notice that it can only achieve *spatial* computation reuse, i.e., common subexpressions exposed independently of the "temporal" loop variables. For example, in Listing 1, there is no common subexpressions in the classical sense, but there actually is computation that can be reused across loop iterations, i.e., *temporal* reuse. This is because when iterating over arrays, different array references from different loop iterations may be referring to the same data element of arrays. For example, in Listing 1, the same computation `X[1][2]+X[2][1]` is done twice, $X[j][i+1] + X[j+1][i]$ for `Y[1][1]` and $X[j-1][i] + X[j][j-1]$ for `Y[2][2]`.

Fig. 1 visualizes the above reuse by showing the overlapping inputs used for producing `Y[1][1]` and `Y[2][2]`. With computation reuse, the new kernel becomes a 2-stage kernel (Formula 2), which requires only 3 additions per output. As a comparison, the original kernel (Listing 1) needs 4 additions per output.

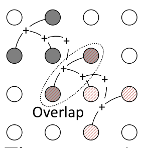

Fig. 1: Overlapping pattern.

$$\begin{aligned} T[j][i] &= X[j-1][i] + X[j][i-1] \\ Y[j][i] &= (T[j][i] + X[j][i] + T[j+1][i+1]) \times 0.2 \end{aligned} \tag{2}$$

Note that there is an implication: when processing such computation reuse, the compiler must recognize the reduction operation and select operands for reuse from a proper computation order, e.g.

$$(([-1][0] + [0][-1]) + [0][0]) + ([0][1] + [1][0]) \tag{3}$$

otherwise the binary $+$ operator will not expose subexpressions like $[0][1]+[1][0]$ due to its default left-to-right associativity. In summary, a compiler must perform both *temporal exploration* among different loop iterations and *spatial exploration* among reduction operands to find the best design point for computation reuse.

III. Related Work

Previous work on computation reuse has limited temporal and/or spatial exploration over the computation reuse design space.

On the temporal exploration side, [16], [17] find reuse among iterations via loop unrolling plus spatial CSE, which is sub-optimal, e.g., for Listing 1 it may only reuse 1 addition operation per 2 outputs, resulting in 3.5 additions per output, as opposed to 3 achieved by Formula 2. [2], [4], [20] only reuse the pointwise scaling operations among iterations, resulting in redundant reduction operations.

On the spatial exploration side, [2], [17], [21] do not consider commutativity and associativity. [13], [21] only consider operands spanning in the horizontal direction (corresponding to the innermost loop variable). [14] only considers operands spanning the horizontal or vertical directions (corresponding to the loop variable of each level of the loop nest). [15] additionally considers diagonal directions, i.e., all loop variables incrementing by the same value. Yet, computation reuse could appear along any spatial direction of the stencil window, which is likely missed by the prior work mentioned above.

Besides, previous work on computation reuse heavily focuses on CPU and/or GPU [4], [13]–[17], where the trade-off between computation and storage relies heavily on register pressure [13] and/or cache [4], [9] analysis, which is generally hard to characterize quantitatively due to the close yet unmanaged interaction between the computation units and the memory system. For accelerators, [12] shows that parallelization and communication can be fully decoupled and presents a microarchitecture that requires the Pareto-optimal on-chip buffer size w.r.t. the degree of parallelism. However, it does not remove any redundant computation. We will show in this paper that computation reuse can be applied independent of parallelization and communication reuse, and how to obtain the Pareto-optimal on-chip buffer size with computation reuse being taken into consideration.

IV. Reuse Discovery Algorithm

A. Problem Formulation

A reduction *expression* defined by Formula 1 in Section II does not define a specific computation order. This means the number of non-redundant operations required to compute the expression may vary. To account for that, we define the reduction *schedule* as a specific computation order of an expression. A schedule has a well-defined computational cost in terms of the number of reduction operations $\oplus$ and the number of scaling operations f, e.g., a naïve left-to-right schedule would require $(n-1)$ $\oplus$ operations and n f operations. Although different schedules produce the same computational result mathematically, the computational cost can be different. Note that even if two schedules have the same computational cost, the storage requirement on accelerators can still be different.

In this paper, we aim to find a schedule of an expression with 1) the least possible number of $\oplus$ reduction operations, and 2) the least possible number of f scaling operations. Notice that f operations can be reused optimally by creating an intermediate array for each scaled operand, we will focus on the optimal reuse of $\oplus$ operations in the following part of this section. Furthermore, if multiple schedules have the same number of operations, we aim to find the schedule with the least storage requirement, which will be discussed in Section V.

B. Optimal Reuse by Dynamic Programming (ORDP)

To discover the optimal reuse, we enumerate over all possible schedules of a reduction expression $opd_0 \oplus opd_1 \oplus ... \oplus opd_{n-1}$ and count the number of unique subexpressions as the number of $\oplus$ operations. Notice that a schedule with n-operands corresponds to a binary reduction tree, whose n leaf nodes correspond to the n operands and $n-1$ non-leaf nodes correspond to the $n-1$ $\oplus$ operations, we can enumerate all schedules via dynamic programming. As an example, let $a+b+c$ be a 3-operand reduction expression. The schedules of $a+b+c$ can be constructed by adding the third operand c to the existing schedules of $a+b$, while $a+b$ only has 1 trivial schedule, which corresponds to the

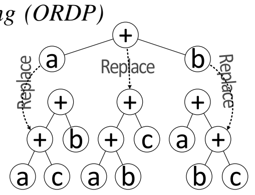

Fig. 2: Reduction trees of $a+b+c$ augmented from $a+b$.

[1] $\oplus$ is commutative iff $a \oplus b \equiv b \oplus a$. $\oplus$ is associative iff $(a \oplus b) \oplus c \equiv a \oplus (b \oplus c)$. We treat floating point additions as if they were associative.

binary tree shown in the upper part of Fig. 2. To obtain schedules of $a+b+c$ from $a+b$, we need to replace one node of $a+b$ with a new node, whose children are the original node and c. Since the reduction tree of $a+b$ has 3 nodes, there are 3 replacement outcomes, too. The lower part of Fig. 2 shows all the 3 reduction trees obtained in this way. The 3 trees correspond to $(a+c)+b$, $(a+b)+c$, and $a+(b+c)$, respectively. In general, assume we have enumerated all schedules of the first k operands, $k = 2, 3, ..., n-1$. To enumerate all schedules of the first $k+1$ operands, all we need to do is to replace one of the nodes in the k-operand reduction tree with a new node, whose children are the newly added operand and the original node. By doing so for all $2k-1$ nodes of all k-operand reduction trees, we can obtain all schedules of the first $k+1$ operands. By induction, we can enumerate all schedules of n operands. Such enumeration achieves spatial exploration of computation reuse.

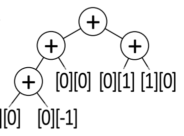

Fig. 3: Reduction tree of Formula 3.

To count the number of operations required for a schedule, we need to count the number of *unique* subexpressions. Since subexpressions can be relatively shifted, we align their access offsets (array references) for comparison. The *aligned access offsets* are obtained by subtracting the least-lexicographical-order [11] offset from all access offsets. As an example, for the schedule given in Formula 3, there are 4 subexpressions (including the whole expression itself), each of which corresponds to a non-leaf node shown in Fig. 3. Among them, two subexpressions, $[-1][0]+[0][-1]$ and $[0][1]+[1][0]$, align to the same $[0][0]+[1][-1]$, which means they can be reused. A hash table is used to count the number of unique subexpressions, where the hash table is keyed by the aligned access offsets and the scaling functions. Subexpressions with the same key indicate reduction operation reuse opportunities. In the previous example, the number of unique subexpressions is 3, which matches the analysis in Section II. Access offset alignment achieves temporal exploration of computation reuse.

The number of all possible schedules of an $(n+1)$-operand expression is $(2n-1)!! = 1 \times 3 \times 5 \times \ldots \times (2n-1)$, which is $(2n-1)\times$ that of an n-operand expression, as discussed in the dynamic programming algorithm presented above. This is asymptotically $O\left(\left(\frac{2n-1}{e}\right)^n\right)$. The optimal solution works well when n is not large ($n \leq 10$) but does not scale. Next, we shall present an efficient heuristic-based solution.

C. Heuristic Search–Based Reuse (HSBR)

In this section, we present a heuristic search–based reuse (HSBR) discovery algorithm that can help us find near-optimal solutions with polynomial time and space complexity. HSBR is a variant of beam search [9] and is composed of three steps, namely 1) *reuse discovery*, 2) *candidate generation*, and 3) *iterative invocation*.

Reuse discovery enumerates all *pairs* of operands to find potential reuse. If a pair of operand appears more than once after alignment, it would be a *reuse pattern* that leads to computation reuse. Note that although we only select pairs of operands, larger patterns are considered since each operand itself can be a subexpression that is composed of multiple operands (e.g., Fig. 4). For the example of Formula 3, after enumerating all pairs of operands, we would find both $[-1][0]+[0][-1]$ and $[0][1]+[1][0]$ align to $[0][0]+[1][-1]$, indicating a reuse opportunity. If no reuse is found in this step, the algorithm terminates.

Candidate generation creates candidate schedules by replacing reuse patterns with new, non-leaf operands. Such non-leaf operands correspond to the intermediate arrays created for reuse, e.g., T in Formula 2. Since there can be many different combinations of reuse patterns, this step may generate a large number of candidates. For example, for Formula 3, in addition to reusing $[-1][0]+[0][-1]$ for $[0][1]+[1][0]$, we would also generate a candidate schedule that reuses $[-1][0]+[0][1]$ for $[0][-1]+[1][0]$. For each candidate, we evaluate how much computation is reused by counting the number of unique subexpressions and how much storage is required as will be discussed in Section V. The best W candidates will be selected for the next step. The constant W is the *beam width* in the beam search algorithm.

Iterative invocation enqueues each selected candidate for the next iteration of HSBR. New reuse patterns are found for each candidate separately, but all next-generation candidates are subject to the same constant bound of beam width W. Since the number of selected candidates in each iteration is $O(W)$ and the number of iterations is $O(n)$ where n is the number of operands in the kernel, the total number of candidates generated and evaluated will be $O(Wn)$. For each candidate, the number of operation required is $O(n^2)$, because we enumerate all pairs of operands. Overall, HSBR is $O(Wn^3)$, which guarantees scalability.

In the remaining part of this subsection, we discuss some optimizations that reduce exploration time and improve quality of result.

1) Operand Selection: We maximize the number of reused operand pairs in each iteration so that the number of iterations is reduced, resulting in faster completion of HSBR, especially for large stencil kernels. This greedy optimization is applied in two places of the candidate generation step. First, for each reuse pattern, we replace as many operand pairs as possible. For example, given $X[0]+2X[1]+X[2]+2X[3]+X[4]+2X[5]$, the reuse discovery step would find that the reuse pattern $T[0] = X[0]+2X[1]$ can be reused for $X[0]+2X[1]$, $X[2]+2X[3]$, and $X[4]+2X[5]$. In the candidate generation step, we greedily replace all operand pairs for reuse (i.e., we replace the aforementioned operand pairs with $T[0]$, $T[2]$, and $T[4]$, respectively). Second, in addition to the operand pairs that reuse the same reuse pattern, we also try to apply other reuse patterns if permissible. For example, given $X[0]+2X[1]+X[2]+2X[3]+3X[4]+4X[5]+3X[6]+4X[7]$, we reuse both $T_1[0] = X[0]+2X[1]$ and $T_2[0] = 3X[0]+4X[1]$ and generate $T_1[0]+T_1[2]+T_2[4]+T_2[6]$ directly in a single iteration.

2) Conflict Resolution: When selecting operand pairs for reuse, sometimes not all valid pairs can be selected at the same time. For example, given $[0]+[1]+[2]+[3]+[4]+[5]$, we'll find that $[0]+[1]$ can be reused for 5 different operations, i.e., $[0]+[1]$, $[1]+[2]$, $[2]+[3]$, $[3]+[4]$, and $[4]+[5]$. However, since these operand pairs overlap, e.g., the first two share the same operand [1], we cannot possibly select all of them for reuse. Although it seems that this problem can be formalized as a graph matching problem, where the nodes are the operands and the edges are the operand pairs, it cannot be solved using the standard minimum matching because the weight (computational cost) of an edge is not static and may vary in different matchings due to the sharing nature of the computational cost. HSBR resolves the conflicts as follows. Notice that we only consider pairs of operands, for the same reuse pattern, each pair can conflict with at most two other pairs, making the conflict graph bipartite, i.e., there are two conflict-free subsets for each group of conflicting operands. In the previous example, the two choices of conflict-free subsets are $\{[0]+[1], [2]+[3], [4]+[5]\}$ and $\{[1]+[2], [3]+[4]\}$. In the candidate generation step, we generate candidates from both choices. To account for the conflicts between different reuse patterns, we generate multiple candidates prioritizing each reuse pattern while greedily selecting other non-conflicting reuse patterns.

3) Regularity Exaction: Reusing operands spanning multiple dimensions may break regularity and lead to sub-optimality. Take a 4×3 uniform-weight kernel as an example. The aforementioned greedy algorithm selects two reuse patterns (labeled ① and ②) in the same iteration, which ends up with a total of 5 $\oplus$ operations, as shown on the left side in Fig. 4. Non-leaf nodes that correspond to the same reuse pattern are labeled with the same number.

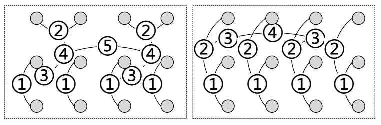
Fig. 4: Different operand selections.

However, if we manually look for reuse, it is not hard to figure out a schedule with only 4 $\oplus$ operations (right side of Fig. 4), which could be generated if we only select patterns along the vertical dimension (①) in the first iteration of the algorithm. To address this, when the number of reuse patterns exceeds a threshold (e.g., number of operands), candidate generation becomes less greedy and only selects reuse patterns along the same direction.

V. Storage Requirement Characterization

Given the number of parallel processing elements (i.e., the parallel factor), prior work [12] generates Pareto-optimal communication reuse buffers and proves that the minimum on-chip storage required by a stencil kernel is determined by the sum of the reuse distance and the parallel factor. Since the parallel factor is an additive term and can be chosen independently, the storage requirement of a stencil kernel can be fully characterized by the reuse distance, independent of the underlying hardware platform or microarchitecture. For a complex multi-stage kernel, which is common after computation reuse is applied (e.g., Formula 2), the conclusion from [12] still holds, and the total storage requirement can be characterized by the total reuse distance[2]. However, the total reuse distance is no longer a constant attribute of the kernel. We will show an example of such case, followed by an algorithm that minimizes it. The minimum total reuse distance obtained will be used to characterize the storage requirement with computation reuse.

A. Total Reuse Distance for a Complex Stencil Kernel

We start with the following example involving two input arrays X_1, X_2, an intermediate array T, and an output array Y:

$$\begin{aligned} T[2] &= X_1[0] + X_1[1] + X_2[0] + X_2[1] \\ Y[0] &= X_1[3] + X_2[3] + T[0] + T[2] \end{aligned} \tag{4}$$

The reuse distances for X_1, X_2, and T are $X_1[0] \cdots X_1[3] = 3$, $X_2[0] \cdots X_2[3] = 3$, and $T[0] \cdots T[2] = 2$, respectively. The total reuse distance is $3 + 3 + 2 = 8$. Notice that Formula 4 implies $T[2]$ and $Y[0]$ are produced at the same time, we can shift the production of T and make $T[4]$ be produced at the same time as $Y[0]$, i.e.

$$\begin{aligned} T[4] &= X_1[2] + X_1[3] + X_2[2] + X_2[3] \\ Y[0] &= X_1[3] + X_2[3] + T[0] + T[2] \end{aligned} \tag{5}$$

The reuse distances become $X_1[2] \cdots X_1[3] = 1$, $X_2[2] \cdots X_2[3] = 1$, and $T[0] \cdots T[4] = 4$, respectively. The total reuse distance becomes $1 + 1 + 4 = 6 < 8$. Obviously, the total reuse distance for a complex stencil kernel may vary as the relative offset between stages change. Section V-B will discuss how to minimize it.

B. Minimizing Total Reuse Distance

Assume we implement our stencil accelerator with a synchronous clock. Given a stencil kernel in which q arrays $\{Y_t | t = 0, ..., q-1\}$ are involved, let $\{Y_s\}$ be the children of Y_t and $Y_s[0]$ consume $\{Y_t[a_u]\}$ from Y_t. For example, in Formula 4, X_1, X_2, T, Y are the

[2] WLOG we assume all element sizes are the same for conciseness.

TABLE I: Stencil benchmarks used in the experiments.

Name	Computation	Size	Name	Computation	Size
s2d5pt	weighted sum of 5	3×3	s2d33pt	weighted sum of 33	17×17
f2d9pt	weighted sum	3×3	f2d81pt	weighted sum	9×9
s3d7pt	weighted sum of 7	3×3×3	s3d25pt	weighted sum of 25	9×9×9
f3d27pt	weighted sum	3×3×3	f3d125pt	weighted sum	5×5×5
contrast[3]	weighted sum of 197	17×17	erosion[3]	minimum	19×19
xcorr[3]	sum except center	19×19	smoother	weighted sum	25×25
bigbiharm	weighted sum of 25	7×7	lilbiharm	weighted sum of 13	5×5

arrays involved. T is a child of both X_1 and X_2. $T[0]$ consumes X_1 and X_2 at $X_1[-2]$, $X_1[-1]$ and $X_2[-2]$, $X_2[-1]$, respectively. Let $\{Y_t[p_t] | t = 0, ..., p-1\}$ be produced at the same cycle. $\{p_t\}$ are the variables to be determined. The reuse distance of each Y_t is

$$D_t = p_t - \min_{s,u} \left(p_s + a_u | s \in \text{children}(t), u \in \text{accesses}(t \to s)\right)$$

Our goal is to minimize the total reuse distance $\sum_t D_t$. For Formula 4,

$$T[p_T] = X_1[p_T - 2] + X_1[p_T - 1] + X_2[p_T - 2] + X_2[p_T - 1]$$
$$Y[p_Y] = X_1[p_Y + 3] + X_2[p_Y + 3] + T[p_Y] + T[p_Y + 2]$$
$$D_{X_1} = p_{X_1} - \min\left(p_T - 2, p_Y + 3\right) \qquad D_T = p_T - p_Y$$
$$D_{X_2} = p_{X_2} - \min\left(p_T - 2, p_Y + 3\right)$$

The constraint is that an array cannot be consumed before produced:

$$p_t \geq p_s + a_u, \forall t, s \in \text{children}(t), u \in \text{accesses}(t \to s)$$

For Formula 4, the constraints are:

$$p_{X_1} \geq p_T - 2 \quad p_{X_1} \geq p_T - 1 \quad p_{X_1} \geq p_Y + 3 \quad p_T \geq p_Y$$
$$p_{X_2} \geq p_T - 2 \quad p_{X_2} \geq p_T - 1 \quad p_{X_2} \geq p_Y + 3 \quad p_T \geq p_Y + 2$$

Notice that each constraint is of the type $x_i - x_j \leq c_{ij}$, this is a systems of difference constraints (SDC) problem and can be solved optimally in polynomial time [22]. For Formula 4, the solution is $p_{X_1} = p_{X_2} = p_Y + 3, p_T = p_Y + 4$, which gives the minimum total reuse distance of 6 and matches Formula 5.

VI. Experimental Results

We extend the open-source SODA compiler [12] to implement the presented algorithms. DSE is written in C++ and runs on a single thread of Intel Xeon E5-2699 v3 CPU. Synthesis is performed by Vivado 2019.1, targeting the Alveo U250 board. The stencil kernels used in the experiments include eight Laplacian kernels used in [4], [12], [20], [23], three image stabilization kernels used in [2], a Gaussian smoother kernel used in pose detection [3], and two biharmonic operator kernels used in [14], [15]. Details about the kernels are listed in Table I. In addition to these real-world benchmarks, we also generate artificial 3×3 kernels to assess the optimality gap between the heuristic algorithm and the optimal one.

A. Number of Operation Reduction

Table II shows the number and type of operations required to produce each output element. The performance of the kernels are fixed to produce 1 output element per clock cycle and all off-chip communication is fully reused. The baseline SODA [12] compiler implements the kernels without computation reuse optimization. Note that SODA outperforms previous papers [11], [23], [24] by up to 9.82× [12]. DCMI [20] is a recent work that synthesizes iterative stencil kernels to FPGAs. DCMI removes redundant multiplication operations, but not the addition (reduction) operations. HSBR shows the result of our heuristic algorithm. Note that for kernels that are small enough (less than 10 points), we are able to verify that *the heuristic algorithm actually produces the optimal result.* On average, our presented algorithm reduces the reduction operations by 58.2%[4].

[3] Marked benchmarks use 8-bit integers; others use 32-bit `float`.
[4] $= 1 - \text{GeoMean}\,\{target/baseline\}$

TABLE II: Operation reduction. **Bold** items are verified to be optimal.

Kernel	Pointwise Operations SODA [12]	Pointwise Operations DCMI [20]/HSBR	Reduction Operations SODA/DCMI	Reduction Operations HSBR
s2d5pt	5	1 *(-80%)*	4	**3** *(-25%)*
s2d33pt	33	9 *(-73%)*	32	24 *(-25%)*
f2d9pt	9	3 *(-67%)*	8	**6** *(-25%)*
f2d81pt	81	15 *(-82%)*	80	48 *(-40%)*
s3d7pt	7	1 *(-86%)*	6	**5** *(-17%)*
s3d25pt	25	5 *(-80%)*	24	20 *(-17%)*
f3d27pt	27	4 *(-85%)*	26	14 *(-46%)*
f3d125pt	125	10 *(-92%)*	124	40 *(-68%)*
contrast	197	30 *(-85%)*	196	113 *(-42%)*
erosion	0	0	360	12 *(-97%)*
xcorr	0	0	359	13 *(-96%)*
smoother	625	91 *(-85%)*	624	336 *(-46%)*
bigbiharm	25	5 *(-80%)*	24	14 *(-42%)*
lilbiharm	9	3 *(-67%)*	12	9 *(-25%)*
average[4]	—	*-81%*	—	*-58%*

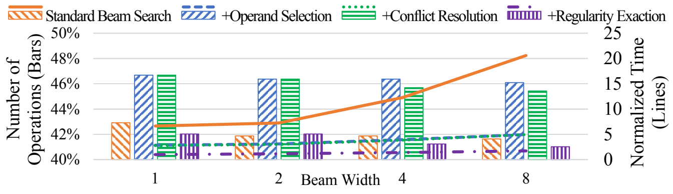

Fig. 5: Impact of heuristics in HSBR. Lower is better.

B. *Impact of Design-Space Pruning Heuristics*

Fig. 5 shows the average operation reduction and the design-space exploration (DSE) time with different beam widths and heuristics used in HSBR. Time is normalized per benchmark to obtain meaningful averages over different benchmarks. In general, larger beam width yields better results, but requires longer DSE time. Operation selection speeds up HSBR by reducing the search depth. Conflict resolution adds some over-pruned points back to the design space and thus compensates some quality of result loss caused by operation selection. Regularity exaction further improves the quality and the runtime by prioritizing regular patterns.

C. *Performance Boost for Compute-Intensive Stencil*

Stencil computation can be compute-intensive if it is iterative [12], [23]–[25], or has a large number of operations per output. For compute-intensive stencil kernels, computation reuse can save resources (Section VI-D) and directly result in a performance boost. We compare the 8 iterative kernels with CPU/GPU results from [4] in Fig. 6. Note that [4] includes all three aspects of stencil optimizations, i.e., parallelization, communication reuse, and computation reuse. All FPGA implementations are scaled up to use the available DSPs and runs at 100 – 125 MHz[5]. On average, DCMI [20] achieves 1.6× speedup over SODA [12], whereas our proposed HSBR algorithm achieves 2.3×. Moreover, thanks to the highly customized datapaths and fully pipelined microarchitecture, HSBR outperforms multi-core Xeon Gold CPU by 10.9×, many-core Xeon Phi processor by 3.17×, and P100 GPU by 1.53× on average, respectively.

D. *Resource Consumption Reduction*

Fig. 7 compares the resource usage of the DCMI optimization and our proposed HSBR algorithm with the baseline SODA implementation. Flip-flop (FF) usage is not reported in the figure because it is tightly coupled with look-up table (LUT) usage and is never used more than LUTs. From the figure, we can see that both optimizations can save computational resources like LUTs and digital-signal processors (DSP) compared with the baseline implementation, possibly at the cost of storage resources (e.g., block random-access memories, BRAMs). On average, DCMI uses 85.1% LUT and 62.6% DSP with 100.0% BRAM usage (compared with SODA baseline). The reduction on LUT and DSP is from the reuse of multiplication operations, and the BRAM usage is the same as SODA since reusing multiplication can be done without additional storage. The HSBR algorithm, on the other hand, only uses 41.0% LUT and 45.4% DSP with 123.7% BRAM usage (compared with SODA baseline). For large kernels (e.g., contrast, erosion, and xcorr), the baseline implementations generate very deep pipelines that lead to a high BRAM usage. With computation reuse, the kernels are decomposed into smaller ones with shallower pipelines, which can significantly reduce the BRAM usage. The geometric mean of BRAM usage is strongly biased by those cases; after excluding them, the average BRAM usage is 231.9% for HSBR. Note that although the storage (BRAM) usage with computation reuse applied can be as high as 7×, we argue that one can scale up the performance at the cost of computational resources (LUTs and DSPs) without significant increase of storage resources, which makes it reasonable to trade-off storage for computation. Actually, when we scale up each benchmark, we find that BRAM usage never bottlenecks the resource usage; the bottleneck is always DSP (for floating-point numbers) or LUT (for fixed-point numbers).

[5] Designs are HLS-based prototypes and are not fine-tuned for high-frequency [26]. We expect instrumentation at RTL level to further improve the frequency and leave that as future work.

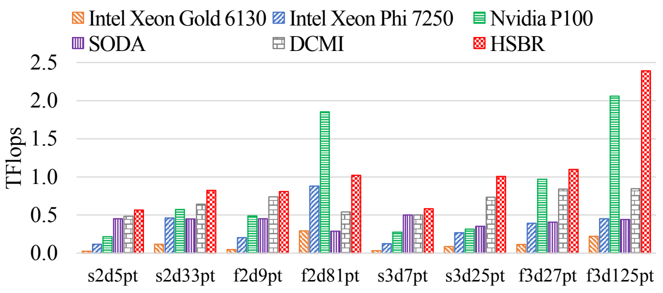

Fig. 6: Performance of iterative kernels. Xeon & P100 are from [4].

E. *Design-Space Exploration Cost*

The optimal algorithm scales up to 10-point stencil kernels and runs in 10 minutes with 6 MiB memory. Although the memory usage remains low, scaling to 11 points requires more than 2 hours on our test machine. Fig. 8 shows the HSBR design-space exploration (DSE) time with various beam widths. Note that since the DSE time is tightly coupled with the kernels, the data points do not align well on a straight line. Since beam search has bounded memory complexity, the memory consumption of HSBR is moderate (< 100 MiB). In general, the cost of the DSE becomes low with the heuristic algorithm.

F. *Optimality Gap*

Although it is impossible to assess the optimality gap for all kernels, we assess the gap between the heuristic algorithm and the optimal algorithm for small kernels. In addition to the real-world benchmarks, we randomly generate artificial 3×3 kernels to examine how well the heuristics perform. Out of the 11528 kernels we generated randomly, there are 5281 kernels with computation reuse opportunity, and our heuristic algorithm can find *all* of them with the least number of required operations with a beam width of 3. Even with the storage overhead (total reuse distance) taken into consideration, HSBR can yield the optimal reuse buffer size with a beam width of 4. This is shown in Fig. 9.

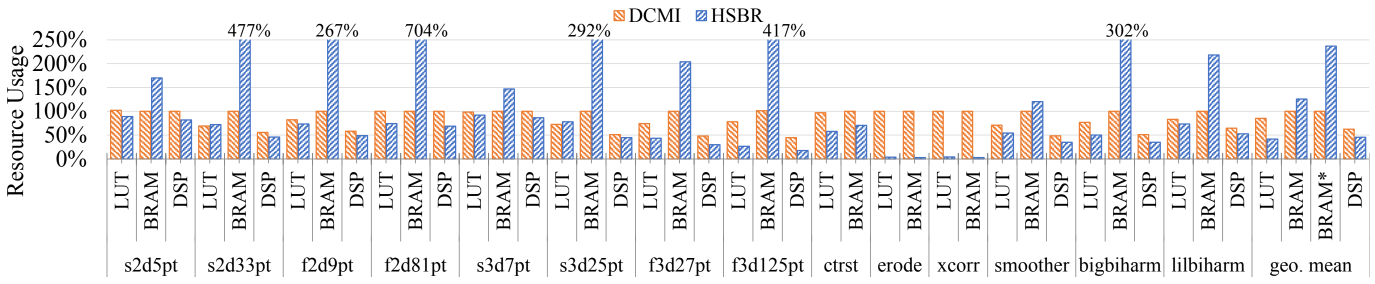

Fig. 7: Resource usage reduction. The normalizaztion baseline is SODA [12]. BRAM* excludes erosion and xcorr to avoid being biased. Truncated bars are marked with values.

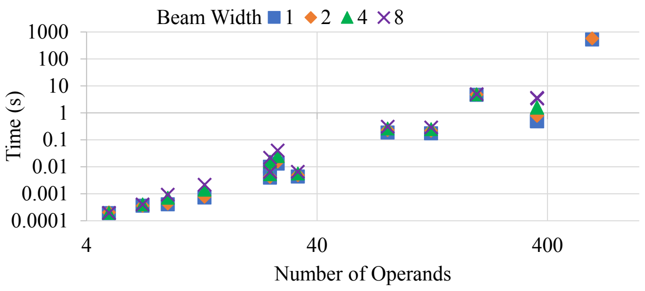

Fig. 8: Polynomial scalability of HSBR. Different points in the same shape correspond to different benchmarks.

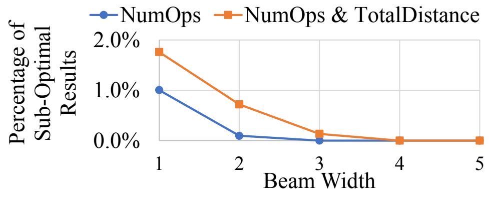

Fig. 9: Optimality gap of HSBR on 5281 artificial 3×3 kernels.

VII. Conclusion

In this paper, we exploit computation reuse optimizations for stencil accelerators. We present an optimal algorithm that can thoroughly explore the complete design space of computation reuse for stencil accelerators with reduction operations. In addition, we present a heuristic beam search algorithm that can effectively prune the vast design space while yielding near-optimal results. Moreover, we fully automate the computation reuse by integrating our algorithms into the SODA compiler. Experimental results show an average of 58.2% reduction on the number reduction operations needed and 58.1% and 54.6% post-synthesis resource reduction on LUT and DSP, respectively, compared with the state-of-the-art SODA compiler. For compute-intensive stencils, our algorithm achieves an average speedup of 2.3× over SODA and outperforms optimized CPU/GPU programs in various benchmarks.

Acknowledgment

The authors would like to thank the anonymous reviewers and our labmates, Zhe Chen and Young-kyu Choi, for their valuable comments and helpful suggestions. This work is partially funded by the NSF/Intel CAPA program (CCF-1723773) and NIH Brain Initiative (U01MH117079), 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] K. Datta *et al.*, “Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures,” in *SC*, 2008.
[2] Z. Chen *et al.*, “LANMC: LSTM-Assisted Non-Rigid Motion Correction on FPGA for Calcium Image Stabilization,” in *FPGA*, 2019.
[3] I. Kim, “tf-pose-estimation,” github.com/ildoonet/tf-pose-estimation.
[4] T. Zhao *et al.*, “Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs,” in *SC*, 2019.
[5] U. Bondhugula and V. Bandishti, “Diamond Tiling : Tiling Techniques to Maximize Parallelism for Stencil Computations,” *TPDS*, vol. 28, no. 5, 2017.
[6] S. Krishnamoorthy *et al.*, “Effective Automatic Parallelization of Stencil Computations,” in *PLDI*, 2007.
[7] J. Holewinski *et al.*, “High-Performance Code Generation for Stencil Computations on GPU Architectures,” in *ICS*, 2012.
[8] N. Maruyama *et al.*, “Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers,” in *SC*, 2011.
[9] A. Adams *et al.*, “Learning to optimize halide with tree search and random programs,” in *SIGGRAPH*, 2019.
[10] M. Wittmann *et al.*, “Multicore-Aware Parallel Temporal Blocking of Stencil Codes for Shared and Distributed Memory,” in *IPDPSW*, 2010.
[11] J. Cong *et al.*, “An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers,” in *DAC*, 2014.
[12] Y. Chi *et al.*, “SODA : Stencil with Optimized Dataflow Architecture,” in *ICCAD*, 2018.
[13] K. Cooper *et al.*, “Redundancy Elimination Revisited,” in *PACT*, 2008.
[14] S. J. Deitz *et al.*, “Eliminating Redundancies in Sum-of-Product Array Computations,” in *ICS*, 2001.
[15] Y. Ding *et al.*, “GLORE: Generalized Loop Redundancy Elimination upon LER-Notation,” in *OOPSLA*, vol. 1, 2017.
[16] M. D. Ernst, “Serializing Parallel Programs by Removing Redundant Computation,” Ph.D. dissertation, MIT, 1994.
[17] S. Kronawitter *et al.*, “Redundancy Elimination in the ExaStencils Code Generator,” in *ICA3PP*, 2016.
[18] Y.-H. Lai *et al.*, “HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing,” in *FPGA*, 2019.
[19] J. Li *et al.*, “HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration,” in *FPGA*, 2020.
[20] M. Koraei *et al.*, “DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs,” *TACO*, vol. 16, no. 4, 2019.
[21] J. Hammes *et al.*, “Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops,” in *IPDPS*, 2001.
[22] J. Cong and Z. Zhang, “An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation,” in *DAC*, 2006.
[23] G. Natale *et al.*, “A Polyhedral Model-Based Framework for Dataflow Implementation on FPGA Devices of Iterative Stencil Loops,” in *ICCAD*, 2016.
[24] H. R. Zohouri *et al.*, “Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL,” in *FPGA*, 2018.
[25] S. Wang and Y. Liang, “A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model,” in *DAC*, 2017.
[26] L. Guo *et al.*, “Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency,” in *DAC*, 2020.