SASA: A Scalable and Automatic Stencil Acceleration Framework for Optimized Hybrid Spatial and Temporal Parallelism on HBM-based FPGAs

XINGYU TIAN, School of Engineering Science, Simon Fraser University, Canada
ZHIFAN YE*, School of the Gifted Young, University of Science and Technology of China, China
ALEC LU, School of Engineering Science, Simon Fraser University, Canada
LICHENG GUO and YUZE CHI, Computer Science Department, University of California, Los Angeles, United States
ZHENMAN FANG, School of Engineering Science, Simon Fraser University, Canada

Stencil computation is one of the fundamental computing patterns in many application domains such as scientific computing and image processing. While there are promising studies that accelerate stencils on FPGAs, there lacks an automated acceleration framework to systematically explore both spatial and temporal parallelisms for iterative stencils that could be either computation-bound or memory-bound. In this paper, we present SASA, a scalable and automatic stencil acceleration framework on modern HBM-based FPGAs. SASA takes the high-level stencil DSL and FPGA platform as inputs, automatically exploits the best spatial and temporal parallelism configuration based on our accurate analytical model, and generates the optimized FPGA design with the best parallelism configuration in TAPA high-level synthesis C++ as well as its corresponding host code. Compared to state-of-the-art automatic stencil acceleration framework SODA that only exploits temporal parallelism, SASA achieves an average speedup of 3.41× and up to 15.73× speedup on the HBM-based Xilinx Alveo U280 FPGA board for a wide range of stencil kernels.

CCS Concepts: • **Hardware → Hardware accelerators**; **Hardware-software codesign**; • **Computer systems organization → Reconfigurable computing**; **High-level language architectures**.

Additional Key Words and Phrases: Stencil Acceleration, Hybrid Parallelism, HBM-based FPGA, High-Level Synthesis, Automation Framework

ACM Reference Format:
Xingyu Tian, Zhifan Ye, Alec Lu, Licheng Guo, Yuze Chi, and Zhenman Fang. 2022. SASA: A Scalable and Automatic Stencil Acceleration Framework for Optimized Hybrid Spatial and Temporal Parallelism on HBM-based FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2022), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*The work was done when Zhifan was a research intern at Simon Fraser University.

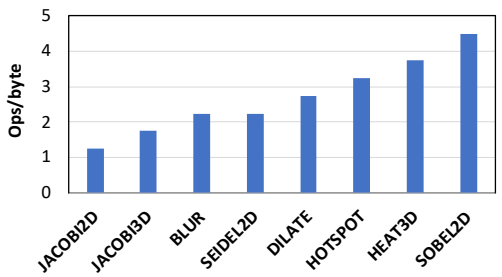
Authors' addresses: Xingyu Tian, xingyu_tian@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6; Zhifan Ye, yezhifan@mail.ustc.edu.cn, School of the Gifted Young, University of Science and Technology of China, No. 96 Jinzhai Road, Hefei, Anhui, China, 230026; Alec Lu, alec_lu@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6; Licheng Guo, lcguo@ucla.edu; Yuze Chi, chiyuze@cs.ucla.edu, Computer Science Department, University of California, Los Angeles, 404 Westwood Plaza, Los Angeles, California, United States, 90095; Zhenman Fang, zhenman@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6.

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 ACM 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.
© 2022 Association for Computing Machinery.
1936-7406/2022/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

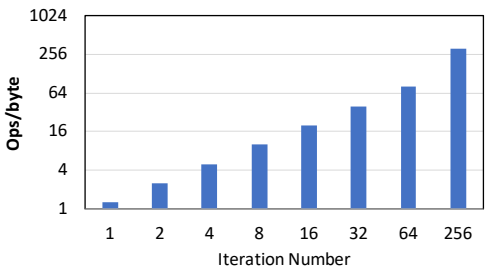
1 INTRODUCTION

Stencil computation is one of the most widely used computing patterns in many important application domains, such as scientific computing, image processing, and cellular automata [1, 11, 17, 26]. Due to its importance, stencil kernels have been well studied and accelerated on multicore CPUs, GPUs, and FPGAs [3, 4, 9, 15, 18, 20, 23–25, 29, 30]. Among these approaches, FPGA acceleration [2–4, 9, 16, 19, 21–23, 25, 29, 30] is getting increasing attention due to its high performance, low power consumption, and high flexibility for customization. For example, in the automatic stencil acceleration framework SODA [4], it designed an optimized dataflow architecture with optimal data reuse and achieved up to 3.28× speedup on an FPGA over a 24-thread CPU.

However, one important factor that is often overlooked in prior studies is that, the stencil computation can be either computation-bound or memory-bound, depending on the stencil operations in the kernel and the number of iterations in the stencil kernel. To demonstrate this, we have measured the computation intensity, defined as the number of algorithmic operations divided by the number of bytes for off-chip memory accesses (OPs/byte), for a wide range of stencil kernels (detailed experimental setup in Section 5.1). The measurement is based on the assumption of the optimal data reuse, i.e., every byte of data only needs to be accessed from off-chip memory once. As shown in Figure 1a, the computation intensity varies between different stencil kernels, ranging from 1.25 to 4.5. Moreover, as shown in Figure 1b, the computation intensity increases linearly with the number of iterations. A high computation intensity indicates that the stencil kernel is computation-bound, while a low one indicates the stencil kernel is memory-bound.



(a) Computation intensity of different stencil kernels with the number of iterations = 1



(b) Computation intensity of the JACOBI2D stencil kernel with different numbers of iterations

Fig. 1. Computation intensity (number of algorithmic operations per byte of off-chip memory access, i.e., OPs/byte) comparison for different stencil kernels and different numbers of iterations.

Such observations suggest that different types of parallelism optimizations are needed for a stencil kernel to achieve the best performance on an FPGA. In general, there is a broad range of iteration numbers for stencil applications. For non-iterative stencil kernels, the iteration number is considered as one. Some iterative stencil kernels could have a large iteration number [4]. Depending on the stencil kernel and its number of iterations, one may need to either 1) parallelize the computation along the iteration dimension (called *temporal parallelism*), or 2) parallelize the memory access along the data dimension (called *spatial parallelism*), or 3) combine both parallelisms together (called *hybrid parallelism*). Moreover, it is nontrivial to program FPGAs to realize the best parallelism for the stencil kernels, especially for domain experts who program in high-level languages. Ideally, domain experts would only need to program in a simple stencil domain-specific language (DSL), and a tool would automatically compile the DSL to a highly efficient stencil accelerator on an FPGA and choose the best parallelism (and the optimal data reuse). Unfortunately, as summarized in Table 1 and Section 2.2, none of the prior studies satisfy all these requirements.

In this paper, we design and implement SASA, a DSL-based, scalable, and automatic stencil acceleration framework on modern HBM-based FPGAs. To support different types of parallelisms for stencil kernels, SASA takes a scalable multi-PE (processing element) approach. For the single PE design, we take a similar design to SODA [4], which explores fine-grained data parallelism that matches the data streaming speed from a single memory bank and achieves the optimal data reuse within a single stencil iteration. Moreover, we further optimize the single PE design by utilizing the *coalesced reuse buffers* (i.e., widened and shortened FIFOs) to reduce its resource utilization and reduce the high fan-out for better timing. For computation-bound stencil kernels, we scale the number of PEs to explore temporal parallelism between stencil iterations and use the same coalesced reuse buffer technique to dataflow between multiple PEs and exploit data reuse between stencil iterations.

For memory-bound stencil kernels, we scale the number of PEs to explore the coarse-grained spatial parallelism to better utilize the available off-chip bandwidth of multiple HBM banks on modern FPGAs. Moreover, we support the combination of temporal and spatial parallelisms to get benefits from both sides.

To bridge the programming gap, we support a simple stencil DSL so that end-users can easily develop their stencil algorithm and get hardware acceleration on FPGAs. Given the stencil DSL and FPGA platform as inputs, SASA can automatically generate a scalable stencil accelerator design in TAPA [5] high-level synthesis (HLS) C++ and its corresponding host code. The generated stencil design automatically chooses the best temporal and spatial parallelism based on our accurate analytical model. Moreover, the open source TAPA framework [5, 13, 14] invokes Vitis HLS to compile our generated TAPA HLS code in parallel, applies coase-grained floorplanning and pipelining to improve the timing closure. Experimental results for a wide range of stencil kernels and iterations confirm the effectiveness of SASA. Compared to state-of-the-art automatic stencil acceleration framework SODA [4] that only explores temporal parallelism, SASA explores the optimized hybrid spatial and temporal parallelism and achieves an average speedup of 3.41× and up to 15.73× speedup on the HBM-based Xilinx Alveo U280 FPGA board.

In summary, this paper makes the following contributions:

- Scalable stencil accelerator design optimizations, including coalesced reuse buffers to further improve the resource usage of the already well-optimized dataflow stencil design [4] that exploits the temporal parallelism, and two design alternatives—redundant computation without communication vs. border streaming for fast border communication—to exploit the spatial parallelism.
- An accurate analytical model, which has less than 5% performance prediction error, to choose the best parallelism configuration for a given iterative stencil kernel, based on whether it is computation-bound or memory-bound.
- An end-to-end automation framework that takes the high-level stencil DSL and FPGA platform as inputs, and automatically generates the optimized FPGA design with the best parallelism configuration on that FPGA. It will be open sourced at https://github.com/SFU-HiAccel/SASA.

The rest of the paper is organized as follows. Section 2 introduces the stencil computation pattern and presents the previous studies and their limitations in accelerating stencil kernels on FPGAs. Section 3 presents the scalable stencil accelerator architecture design of SASA, and its various types parallelism optimizations. Section 4 describes our end-to-end automation framework, including the high-level stencil DSL, the analytical performance models for our accelerator design, the code generator and automation tool flow. Section 5 evaluates the performance of SASA on a comprehensive set of stencil benchmarks with different numbers of iterations, compares the performance of different parallelism optimizations, and demonstrates that SASA achieves an average speedup of 3.41× and up to 15.73× speedup over state-of-the-art automatic stencil acceleration framework SODA [4]. Finally, Section 6 concludes this paper and discusses the future work.

2 BACKGROUND AND RELATED WORK

In this section, we first introduce the stencil computation pattern. Then, we discuss related literature on FPGA acceleration for stencil computations and their limitations. Finally, we describe the goal of our paper.

2.1 Stencil Computation

Stencil computation usually operates on a multidimensional array and updates each data cell using its neighbor cells in a fixed pattern. Listing 1 and Figure 2 show an example of the JACOBI2D stencil kernel, which is a 5-point, 2-dimensional stencil that computes and updates each data cell (i.e., $output[i][j]$) with the values from itself (i.e., $input[i][j]$) and its four neighbor cells (i.e., $input[i][j-1]$, $input[i-1][j]$, $input[i][j+1]$, and $input[i+1][j]$). Its stencil kernel radius size is 1, which is defined as the distance between the center cell and its furthest neighbor cell. In practice, such a stencil kernel will be executed multiple iterations; in the next iteration, the output array from the previous iteration becomes the input, while the input array from the previous iteration becomes the output. As discussed earlier in the introduction, depending on the stencil operations and the number of iterations, the stencil kernel could be either computation-bound or memory-bound, and would require a different parallelism optimization to achieve the best performance.

```
void jacobi2d (float input[R][C], float output[R][C]) {
  for (int i = 1; i < R − 1; ++i)
    for (int j = 1; j < C − 1; ++j)
      output[i][j] = (input[i][j−1] + input[i−1][j] + input[i][j] + input[i][j+1] + input[i+1][j]) / 5;
}
```

Listing 1. A 5-point stencil JACOBI2D kernel

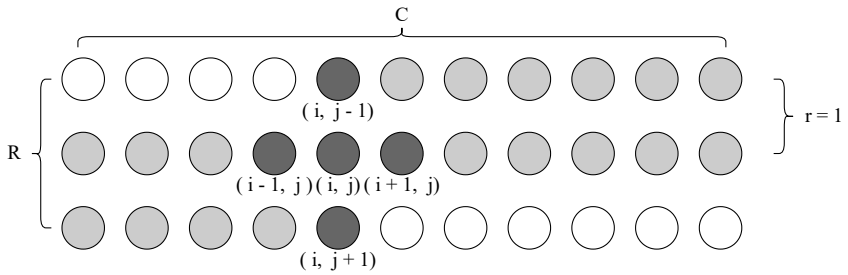


Fig. 2. Stencil access pattern of JACOBI2D: R and C are the number of input rows and columns, r is the stencil radius size.

2.2 FPGA Acceleration for Stencil Computation

Previous research efforts in FPGA-based stencil computing generally target the following aspects: 1) improving the on-chip data reuse and reducing the FPGA on-chip memory usage for the stencil computation, 2) exploring temporal and/or spatial parallelisms in the stencil accelerator designs, including leveraging the HBM bandwidth on modern FPGAs to further extend the spatial parallelism, and 3) facilitating automatic stencil design generation. Some of these studies also need pre-processing on the host CPU to enable their optimizations. Next we discuss the prior studies considering these aspects. Some of the recent studies are also summarized in Table 1 for comparison and illustrating the novelty of our work. We also list their tiling approach (1-dimension, 2-dimension, and not available) and fine-grained parallelism factor.

Table 1. Comparison of stencil acceleration frameworks. #PUs per PE denotes the best possible fine-grained parallelism factor, i.e., the number of processing units inside each processing element, assuming a 512-bit wide off-chip memory interface. NA stands for not available.

	Multi-PE parallelism	Pre-processing free	Automatic optimization	On-chip data reuse	Tiling	#PUs per PE
[2, 19]	temporal	✓	✓	streaming	2D	N/A
[4]	temporal	✓	✓	streaming	2D	16
[21]	temporal	✓	✗	streaming	N/A	16
[23]	temporal	✓	✗	streaming	2D	16
[29]	temporal	✗	✗	streaming	2D	16
[25]	hybrid	✓	✗	buffering	2D	N/A
[22]	hybrid	✓	✗	buffering	2D	16
[9]	hybrid	✗	✗	buffering	N/A	16
[16]	hybrid	✗	✗	streaming	1D	16
Ours	hybrid	✓	✓	streaming	1D	16

In effort to improve on-chip data reuse and reduce the FPGA on-chip memory usage for the stencil computation, Chi et al. presented SODA [4] and proposed the optimal streaming solution to minimize the reuse buffer size and leveraged microarchitectural design optimizations to also minimize the external memory access. Therefore, we also build our baseline design based on SODA. Other methods such as a sliding-window design is used in [7], which requires maintaining only a small on-chip buffer to reduce the BRAM usage, but introduces additional off-chip communication overhead. Further, another graph-theory based implementation proposed in [10] can derive the minimum memory partition factor for the on-chip memory banks, but such a design only supports a limited set of stencil kernels.

Various studies have explored accelerating the iterative stencil computation through different types of parallelisms. Namely, acceleration through **temporal parallelism** has been well studied and explored in most previous iterative stencil kernel accelerator designs on FPGA [2, 4, 9, 12, 16, 19, 21–23, 25, 29, 30]. For example, in [2, 19], Natale and Cattaneo et al. designed a dataflow architecture that executes multiple stencil computing iterations as multiple temporal stages. However, it lacks exploiting the fine-grained spatial parallelism during the processing of a single iteration stage. In SODA [4], Chi et al. also presented a streaming-based accelerator design that exploits the temporal parallelism in the iterative stencil acceleration and proposed microarchitectural design optimizations to minimize the reuse buffer size and external memory access. For these designs [2, 4, 19], there is no data pre-processing requirement; and they also provide an automatic design optimization framework to explore design space and optimize designs based on accurate analytical performance models. Further, Hasitha et al. [23] and Reggiani et al. [21] explored scaling the temporal parallelism of their accelerator design across multiple FPGAs without requiring any redundant computations. However, their design is only efficient for the computation-bound stencil kernel, where it can benefit from the great amount of data reuse. A common limitation these temporally accelerated designs share is that they do not exploit any coarse-grained spatial parallelism, which would lead to under-optimized performance when the stencil kernel has a low number of iterations and is memory-bound.

To leverage both temporal and spatial parallelisms, previous studies have exploited **hybrid parallelism** in their designs [9, 16, 22, 25, 30]. Unlike the streaming-based designs, some of these designs require data buffering and typically have a significant on-chip memory utilization

requirement. For example, the designs in [22, 25] need to load multiple tiles of data on chip for its PEs to execute in parallel; another approach presented in [9] requires a single but rather larger on-chip buffer. To further exploit the spatial parallelism by leveraging the HBM bandwidth on modern FPGAs, previous work also explored utilizing multiple memory banks of an HBM in their multi-PE design [16] and single-PE design [9]. However, their implementations require data pre-processing on the host CPU side to allow the parallel memory access and to exploit the efficient burst access from the FPGA off-chip memory. Another common limitation of these work is the lack of a design automation framework to facilitate the automatic design generation and mitigate the long design exploration process.

2.3 Goal of This Paper

The goal of this paper is to develop an automatic stencil acceleration framework to incorporate the all the features as summarized in Table 1. First, a streaming-based design similar to SODA [4] is used in SASA for achieving the optimal data reuse; we also additionally include coalesced reuse buffers to further reduce resource utilization in our design. Second, SASA leverages a hybrid multi-PE design architecture, exploiting both spatial and temporal parallelisms (presented in Section 3). Furthermore, SASA utilizes the HBM memory on modern FPGAs to achieve higher throughput, yet unlike previous HBM-based stencil designs, our tool does not require any data pre-processing on the host CPU side. And lastly, in terms of design automation, SASA enables end-users to define their stencil operations through a DSL and will automatically compile and optimize the accelerator designs with the best parallelism configuration based an analytical model (detailed in Section 4).

3 SCALABLE STENCIL ACCELERATOR DESIGN WITH HYBRID TEMPORAL AND SPATIAL PARALLELISM

In this section, we explore different types of parallelism optimizations in SASA, as summarized in Figure 4, 5 and 6. It is nontrivial to choose the best parallelism for a stencil kernel, since the most appropriate parallelism varies, depending on the stencil kernel and its number of iterations. First, in Section 3.1, we present our single PE design, which is based on the streaming-based architecture proposed in SODA [4]; and we describe our *coalesced reuse buffer* design optimization for further reducing the resource utilization of the design. Next, in Section 3.2, we briefly describe our temporal parallelism design for computation-bound stencils with high number of iterations, which is similar to SODA but uses our *coalesced reuse buffer* optimization. After that, in Section 3.3, we introduce our spatial parallelism design for memory-bound stencil kernels and discuss two design alternatives to implement it. Finally, in Section 3.4, we explore hybrid parallelism, which exploits benefits of both spatial and temporal parallelisms in our multi-PE designs.

3.1 Single PE Optimization

As mentioned in Section II, our single PE design is based on SODA's design [4], since it achieves the optimal reuse buffer size and off-chip memory access requirement. Figure 3 (a) shows an architecture overview of SODA's single PE design. For the input data that stream from the off-chip memory, SODA exploits the memory coalescing optimization to stream 512-bit wide data every cycle and stores it in an on-chip line buffer using BRAM. Then, it distributes this buffered data into reuse buffer channels composed of FIFOs and FFs, where each has a data width that matches the size of each stencil data cell (e.g., 32-bit for the *float* data type). The data from these reuse buffer channels are then forwarded to the parallel processing units (PUs) for exploiting the fine-grained (spatial) parallelism and generate the output results. Each PU computes and updates for one data cell in the stencil, as shown in the JACOBI2D example in Listing 1. The degree of fine-grained parallelism (i.e., the number of PUs) is set to saturate the off-chip bandwidth of a single memory

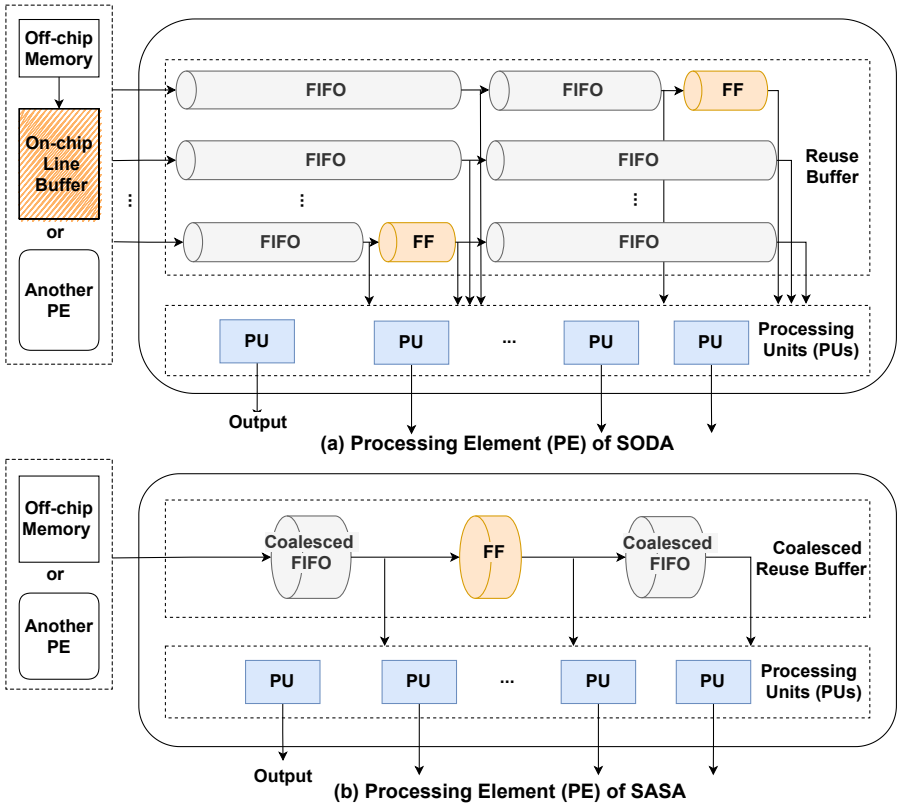


Fig. 3. Single processing element (PE) architecture based on SODA [4], with optimization of coalesced reuse buffers to reduce on-chip BRAM usage.

bank and ensure the design executes in a dataflow fashion. For example, for a single HBM bank that uses 512-bit wide AXI interface and a data cell type of *float*, the number of PUs can be derived as 512 $bits$ / $(8bits/byte)$ / $sizeof(float)$ $bytes$ = 16. For the detailed microarchitecture design of the baseline PE, we refer the audience to the SODA paper [4].

However, based on our experiments, SODA's distributed reuse buffer channel implementation can be further optimized to reduce the on-chip BRAM usage. For such, in SASA, we propose an alternative implementation that removes the on-chip line buffer for storing the input data, and coalesces all the narrow distributed reuse buffers into a single wide coalesced reuse buffer as shown in Figure 3 (b), to further reduce the BRAM usage. With memory coalescing, the input data it reads in from off-chip memory is typically 512-bit wide. Without coalesced FIFOs, it needs an on-chip line buffer to store such 512-bit wide data that is read in a AXI burst mode. And then it distributes such wide data from the on-chip line buffer onto multiple narrow (32-bit wide for floating data type) FIFOs, as shown in Figure 3 (a). With coalesced FIFO, we stream in 512-bit data and write them into the 512-bit wide FIFO (i.e., coalesced FIFO) directly. Thus, we can get rid of the extra on-chip line buffer. Each cycle we also read one 512-bit data from each coalesced FIFO, divide it into multiple 32-bit registers, and feed them to the parallel PUs. Another benefit of our optimization is that it helps reducing the number of fan-outs from SODA's line buffer design, and thus allows the design to achieve a higher operating frequency when further scaling out to multiple PEs.

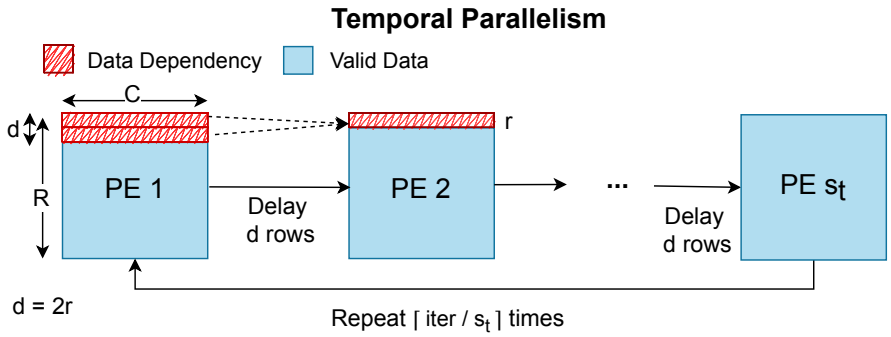


Fig. 4. Temporal parallelism among the stencil iterations. d stands for delay between two temporal stages; r stands for stencil radius size; R and C stand for the row and column size; s_t stands for the number of temporal stages; $iter$ stands for the number of iterations. These are summarized in Table 4 as well.

3.2 Temporal Parallelism Optimization

In order to exploit the temporal parallelism, we instantiate multiple of our single PEs in a cascaded pipeline fashion as shown in Figure 4, which is similar to SODA's temporal parallelism design. The difference is that we use the coalesced reuse buffers to connect multiple PEs. The input data is only read once from the off-chip memory and the output result is also written once back to the off-chip memory after processing N iterations of the stencil computation. Each PE handles one iteration of the stencil processing. For computation-bound stencil kernel designs with high number of iterations, it is more efficient to leverage the temporal parallelism since it allows for a higher level of data reuse across processing multiple consecutive stencil iterations in a pipelined fashion on the FPGA and does not require a huge amount of off-chip memory bandwidth. However, when the number of iterations becomes low, it will be hard to leverage the benefits of this temporal parallelism.

3.3 Spatial Parallelism Optimization

For stencil kernels with low computation intensity and low number of iterations, parallelizing the memory access along the spatial (data) dimension is more efficient, compared to leveraging the temporal parallelism. To fully exploit the spatial parallelism during a single iteration of the stencil computation, first we need to evenly partition the input data and store them onto different HBM banks to allow for more parallel memory access. Note that here we are just simply partitioning the input data vertically by the rows, so there is no data pre-processing overhead. After that, we can then instantiate multiple spatially parallel PEs to distribute the workload for coarse-grained parallel computation and memory access. Due to the dependency of the halo data, which is the boarder data between partitions, synchronization could be required at the end of each stencil iteration to maintain correctness of the output results.

To address the halo synchronization issue when leveraging the spatial parallelism, Figure 5 (a) and 5 (b) present the two approaches in our stencil accelerator design:

1. **Redundant Computation**: In order to reduce the memory transfer overhead during the data synchronization, one way is to avoid data synchronization. As shown in Figure 5 (a), input data is partitioned into multiple tiles and each PE processes one tile. Each PE needs to read additional halo data from neighbouring tiles at the start, then performs the computation of all iterations without synchronization. The halo size is decided by the number of iterations and the stencil algorithm itself.

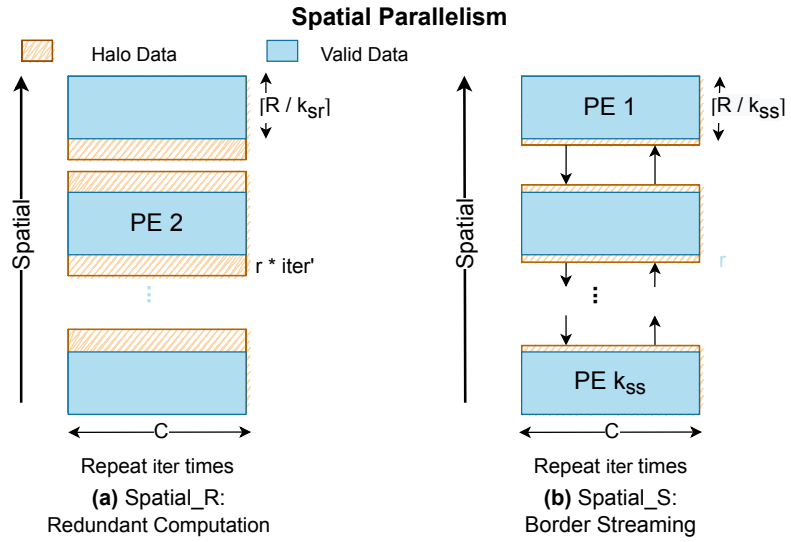


Fig. 5. Spatial parallelism with redundant computation and border streaming. R and C stand for the row and column size; r stands for stencil radius size; k_{sr} stands for the number of PEs in spatial parallelism with redundant computation; k_{ss} stands for for the number of PEs spatial parallelism with border streaming. These are summarized in Table 4 as well.

2. **Border Streaming**: Another way is to exchange the halo data between neighbor PEs via the border streaming technique. As shown in Figure 5 (b), each PE only computes its own input tile without redundant computations for extra halo data. Instead, it exchanges the required halo data with the neighbouring PEs at the end of every iteration. To support efficient halo data exchange, it exchanges such data via on-chip streaming. Compared to redundant computation, this approach uses slightly more on-chip resource (e.g., LUTs and FFs) to implement border streaming interfaces, but can reduce the computation overhead.

3.4 Hybrid Parallelism Optimization

There are limitations for both temporal parallelism and spatial parallelism. As previously discussed in Section 1, the performance bottleneck varies with the algorithmic intensity and number of iterations of a stencil kernel. This is because when the stencil iteration number is high and computation intensity is high (i.e., computation-bound), the major performance improvement comes from the parallel processing across multiple consecutive stencil iterations in a pipelined fashion with high on-chip data reuse on the FPGA (i.e., temporal parallelism). Conversely, for the memory-bound stencil kernels with a low iteration number, the performance gain comes from the parallel memory access within each stencil iteration, and the spatial parallelism can efficiently parallelize the computation across the data dimension.

In our hybrid parallelism approach, both temporal and spatial parallelisms are exploited to better support efficient acceleration of the arbitrary stencil operations. In terms of the design architecture, we integrate the temporal parallelism and explore the two variants of the spatial parallelism optimizations: 1) *Hybrid_R* (temporal with the *Spatial_R* parallelisms) and 2) *Hybrid_S* (temporal with the *Spatial_S* parallelisms), as shown in Figure 6 (a) and (b), respectively.

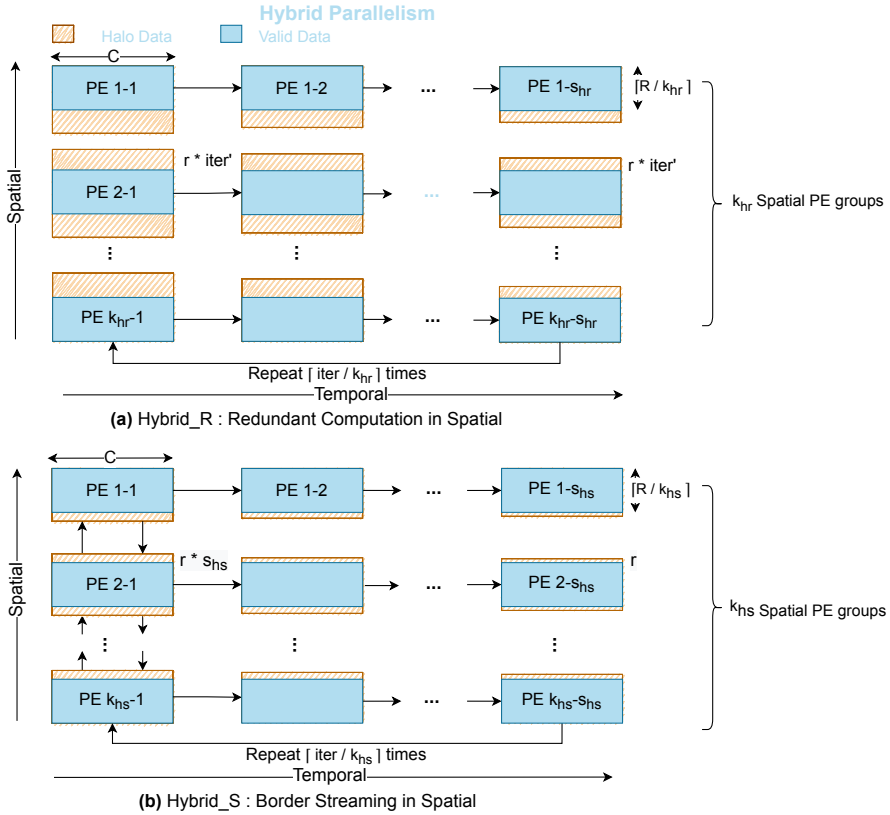


Fig. 6. Hybrid parallelism with redundant computation and border streaming. R and C stand for the row and column size; r stands for stencil radius size; k_{hr} and s_{hr} stand for the degree of spatial parallelism and temporal parallelism respectively in hybrid parallelism with redundant computation; k_{hs} and s_{hs} stand for the degree of spatial parallelism and temporal parallelism respectively in hybrid parallelism with border streaming. These are summarized in Table 4 as well.

1. **Hybrid_R**: To integrate temporal parallelism with *Spatial_R* spatial parallelism, we instantiate multiple (k_{hr}) spatial PE groups to concurrently process different partitions of the input data without any synchronization requirement as described in Section 3.3. Within each spatial PE group, we apply temporal parallelism to concurrently process multiple stencil iterations using multiple (s_{hr}) PEs in a dataflow fashion, as shown in Figure 6 (a). To avoid halo synchronization, PEs in the earlier stages need to compute increasingly more halo data than those PEs in the later stages. In total, there are $k_{hr} \times s_{hr}$ PEs running concurrently, processing s_{hr} stencil iterations at a time. The whole design has to be executed multiple rounds to finish the entire stencil iterations.
2. **Hybrid_S**: To integrate temporal parallelism with *Spatial_S* spatial parallelism, we adopt a similar approach as the *Hybrid_R* design. We denote the number of spatial PE groups as *k_hs* for spatial parallelism, and the number of temporal stages within each spatial PE group as *s_hs*. The main difference lays in the additional synchronization step to update the halo region data after processing each iteration/temporal stage. If we simply replicate the *Spatial_S* design by s_{hs} number of temporal stages, the corresponding number of border streaming connections will also increase, and thus may cause overhead in the placement and routing, as well as design frequency

degradation. As an optimization, in our design, only the spatial PEs in the first temporal stages have the border streaming connections between themselves and perform the halo data exchange. Instead of only exchanging one *halo* rows of data, they exchange all required $halo \times s_hs$ rows of data required by PEs for all following *s_hs* temporal stages. For the remaining temporal stages, no more synchronization is required. The whole design has to be executed multiple rounds to finish the entire stencil iterations, and only at the beginning of each round, there is halo data exchange required.

4 AUTOMATION FRAMEWORK FOR SASA

In this section, we discuss the automation perspectives of SASA and its fundamental components. First, we describe a domain-specific language (DSL), which is similar to the one used in SODA [4], for domain experts to easily define their stencil computation settings. Then, we introduce the analytical model for estimating the design performance under different types of parallelisms according to the design parameters. Finally, we present the entire work flow of our code generator that incorporates our analytical model to automatically determine the best parallelism configuration, compiles the DSL to the corresponding optimized stencil design in TAPA high-level synthesis (HLS) C++ [5], and generates the corresponding TAPA host code in C++ [5].

4.1 Stencil DSL

```
kernel: JACOBI2D
iteration: 4
input float: in_1(9720, 1024)
output float: out_1(0,0) = ( in_1(0,1) + in_1(1,0) + in_1(0,0) + in_1(0,-1) + in_1(-1,0) ) / 5
```

Listing 2. A 5-point stencil JACOBI2D kernel description in SASA DSL

```
kernel: HOTSPOT
iteration: 64
input float: in_1(9720, 1024)
input float: in_2(9720, 1024)
output float: out_1(0,0) = 1.296 * ((in_2(-1,0) + in_2(1,0) - in_2(0,0) + in_2(0,0)) * 0.949219 + in_1(-1,0) +
    (in_2(0,-1) + in_2(0,1) - in_2(0,0) + in_2(0,0)) * 0.010535 + (80 - in_2(0,0)) * 0.00000514403)
```

Listing 3. A 9-point stencil HOTSPOT kernel description in SASA DSL with two inputs

```
kernel: BLUR-JACOBI2D
iteration: 4
input float: in(9720, 1024)
local float: temp(0,0) = (in(-1,0) + in(-1,1) + in(-1,2) + in(0,0) + in(0,1) + in(0,2) + in(1,0) + in(1,1) +
    in(1,2)) / 9
output float: out(0,0) = (temp(0,1) + temp(1,0) + temp(0,0) + temp(0,-1) + temp(-1,0)) / 5
```

Listing 4. A description of two combined stencil kernels in SASA DSL

To allow domain experts to easily define any arbitrary stencil computing workload at a high abstraction level, SASA provides a stencil domain-specific language (DSL) similar to SODA [4]. Here We present a few stencil kernel samples using SASA DSL: Listing 2 shows the description of a 5-point, 2-dimensional JACOBI2D stencil kernel; Listing 3 shows the description of a 9-point, 2 dimensional HOTSPOT stencil kernel handling two input data; and Listing 4 shows the description of two combined stencil loops.

1. The text following the **kernel** keyword specifies the name of the stencil kernel, which is also used as the name of the top-level function in HLS for the FPGA kernel.
2. The number after the **iteration** keyword specifies the number of iterations that the stencil kernel will be executed.
3. For **input** keyword, first, the data type of each stencil cell is specified, followed by the name and dimension of the input data array.
4. Similarly, for the **output** keyword, the data type is first specified. Then, users should specify the name of the output data and the formula to compute and update one output data cell.
5. Multiple inputs and outputs and multiple stencil loops are supported.
6. The **local** keyword is used to define the intermediate data between multiple stencil loops.

4.2 Analytical Performance Model

In this section, to guide the design automation, we build an analytical performance model for our accelerator framework with the temporal, spatial, and hybrid parallelism optimizations presented in Section 3. Such a comprehensive model is not present in previous studies. Table 2 shows the description of the parameters used in our analytical model to determine the latency L of designs with different parallelisms. Parameters such as the number of input rows and columns (R and C), number of stencil iterations ($iter$), and stencil radius size (r) can be extracted from the input stencil DSL. Note that our analytical model only models a two-dimensional stencil. As will be presented in Section 4.3, our code generator will transform a multidimensional array specified in the stencil DSL into a two-dimensional array. Parameters such as the delay between two temporal stages (d) and size of halo region for one iteration ($halo$) can be directly derived from the input r, i.e., $d = halo = 2 \times r$. For other parameters such as the number of PUs inside each PE (U), the degree of spatial parallelism (k with different subscripts), and the degree of temporal parallelism (s with different subscripts), our automation tool flow (Section 4.3) will automatically choose the best configurations.

Table 2. Description of analytical model parameters

Parameter		Definition
Output	L	Overall execution latency
Input	R	Number of input rows
	C	Number of input columns
	$iter$	Number of stencil iterations
	r	Stencil radius size
Derived	d	Delay between two temporal stages ($d = 2 \times r$)
	$halo$	Size of halo region for one iteration ($halo = 2 \times r$)
SASA automatic exploration	U	Unroll factor along column dimension, i.e., number of PUs per PE
	k	Degree of spatial parallelism
	s	Degree of temporal parallelism
Subscript	subscript t	Temporal parallelism
	subscript sr	Spatial parallelism with redundant computation
	subscript ss	Spatial parallelism with border streaming
	subscript hr	Hybrid parallelism with redundant computation
	subscript hs	Hybrid parallelism with border streaming

For each PE, the latency to execute one stencil iteration is determined by the dimension (i.e., number of rows (R) and columns (C)) of the input data and the number of PUs inside each PE (i.e.,

U) of the design, which is $\lceil \frac{R \times C}{U} \rceil$. Next we describe our analytical models to compute the latency for each parallelism configuration for the multi-PE design.

Resource Bound and Memory Bound. The maximum number of PEs that can be implemented is limited by both on-chip hardware resource and available off-chip memory banks (i.e., bandwidth). For the limitation of on-chip resource, we have:

$$\#PE_{res} = \frac{\alpha \times total_FPGA_resource}{resource_per_PE} \tag{1}$$

where α is the FPGA resource utilization constraint ratio and is initially set as 75%, since typical design that uses more than 75% of the FPGA resource becomes very difficult to pass the placement and routing.

For the constraint of off-chip memory banks, the number of spatial PEs is bounded as:

$$\#PE_{bw} = \frac{\#total_off_chip_mem_banks}{\#off_chip_mem_banks_per_spatial_PE} \tag{2}$$

where $\#off_chip_mem_banks_per_spatial_PE$ is defined by the inputs and outputs number of stencil algorithm. Then, we can determine the maximum PE number based on FPGA platform specification and hardware resource constraints:

$$Max\ \#PE = min\ (\#PE_{res},\ \#PE_{bw} \times s) \tag{3}$$

where s is the number of temporal stages (i.e., the degree of temporal parallelism) in each spatial PE group and these temporal stages do not require extra bandwidth.

Temporal Parallelism. As shown in Figure 4, we exploit the temporal parallelism in our design by cascading s_t number of PEs to execute in a dataflow fashion. This also means s_t iterations of the stencil computation is processed concurrently as input data get streamed through our design. To process an iterative stencil computation with $iter$ iterations, our design should be executed $\lceil iter/s_t \rceil$ times. To compute any single output in PE i (except the first PE), data across two stencil radius size (2r) are required from the previous PE $i-1$. Thus, there is a delay $d = 2r$ rows between any two neighbor stages. The last PE s_t has to wait $d \times (s_t - 1) \times C$ cycles to start the execution. Therefore, we determine overall latency of the temporal parallelism design as:

$$L_t = \left\lceil \frac{(R + d \times (s_t - 1)) \times C}{U} \right\rceil \times \left\lceil \frac{iter}{s_t} \right\rceil,\ s_t \leq \#PE_{res} \tag{4}$$

In this case and s_t is limited by $\#PE_{res}$, i.e., the available computing resource.

Spatial Parallelism. In the context of spatial parallelism as shown in Figure 5 (a) and (b), we have two different design implementations: redundant computation ($Spatial_R$) and border streaming ($Spatial_S$). In both of these implementations, the computation of a single stencil iteration is distributed across multiple parallel spatial PEs. Every single PE processes $\lceil R/k_{sr} \rceil$ or $\lceil R/k_{ss} \rceil$ rows of the input data, plus some halo region rows. The design has to be executed $iter$ times. Note in the spatial parallelism, we put one PE inside each FPGA spatial PE group and the number of FPGA spatial PE groups equals to the number of PEs.

For $Spatial_R$, the latency can determined as

$$L_{sr} = \left\lceil \frac{(\lceil \frac{R}{k_{sr}} \rceil + halo \times iter') \times C}{U} \right\rceil \times iter,\ k_{sr} \leq Max\ \#PE \tag{5}$$

where $halo \times iter'$ represents the halo data size gradually decreases over the processing iteration (i.e., $iter'$) as explained in Section 3.3. On average, $iter' = iter/2$.

As for *Spatial_S*, we calculate its latency in Equation 6, since all the PEs synchronize with their neighboring PEs for a fixed number of *halo* rows after each stencil iteration.

$$L_{ss} = \left\lceil \frac{(\lceil \frac{R}{k_{ss}} \rceil + halo) \times C}{U} \right\rceil \times iter,\ k_{ss} \leq Max\ \#PE \tag{6}$$

For both spatial parallelisms, they are limited by both the computing resource and memory bandwidth, i.e., *Max #PE*.

Hybrid Parallelism. As shown in Figure 6 (a) and (b), when combining spatial and temporal parallelisms, there are k_{hr} (or k_{hs}) FPGA spatial PE groups running concurrently, each FPGA spatial PE group processing $\lceil R/k_{hr} \rceil$ (or $\lceil R/k_{hs} \rceil$) rows of input data. Within each FPGA spatial PE group, there are s_{hr} (or s_{hs}) temporal stages running concurrently in a dataflow fashion, processing s_{hr} (or s_{hs}) stencil iterations at a time. Therefore, our design with hybrid parallelism has to execute $\lceil iter/s_{hr} \rceil$ (or $\lceil iter/s_{hs} \rceil$) times. In total, there are $k_{hr} \times s_{hr}$ (or $k_{hs} \times s_{hs}$) PEs running concurrently in the design.

For *Hybrid_R*, which is the combination of temporal parallelism and spatial parallelism with redundant computation, for one round of execution, all PEs within each FPGA spatial PE group i complete exactly at the same time. The reason is that the prior PE $i, j-1$ needs to redundantly compute *halo* more rows of data than the next PE i, j, while the next PE i, j needs to wait d rows of data from the prior PE $i, j-1$, where $halo = d = 2r$. Therefore, we derive Equation 7 for the latency of *Hybrid_R*:

$$L_{hr} = \lceil \frac{(\lceil \frac{R}{k_{hr}} \rceil + halo \times iter') \times C}{U} \rceil \times \lceil \frac{iter}{s_{hr}} \rceil,\ k_{hr} \leq PE_{bw},\ k_{hr} \times s_{hr} \leq\ Max\ \#PE \tag{7}$$

where the first term represents the latency to execute one round of our hybrid parallelism design, and $halo \times iter'$ represents the halo data size gradually decreases over the processing rounds. On average, $iter' = iter/2$. In this case, the degree of spatial parallelism is limited by the memory bandwidth, and the total degree of parallelism is limited by both the computing resource and memory bandwidth.

For *Hybrid_S*, which is the combination of temporal parallelism and spatial parallelism with border computation, similarly, for one round of execution, all PEs within each FPGA spatial PE group i complete exactly at the same time. However, in this design, these PEs only need an extra latency for *halo* more rows within each round. Between different rounds, PEs in the first temporal stage exchange *halo* data with each other using border streaming. Therefore, we derive its latency as below:

$$L_{hs} = \lceil \frac{(\lceil \frac{R}{k_{hs}} \rceil + halo \times s_{hs}) \times C}{U} \rceil \times \lceil \frac{iter}{s_{hs}} \rceil,\ k_{hs} \leq PE_{bw},\ k_{hs} \times s_{hs} \leq\ Max\ \#PE \tag{8}$$

In this case, the degree of spatial parallelism is limited by the memory bandwidth, and the total degree of parallelism is limited by both the computing resource and memory bandwidth.

Automatic Parallelism Optimization. In order to automatically determine the optimal parallelism, the automation tool would need to find the parallelism with the minimum latency as:

$$L_{optimal} = \min(L_t, L_{sr}, L_{ss}, L_{hs}, L_{hs}) \tag{9}$$

Examining our analytical performance model at a high level and assuming maximum PE number is the same across different parallelisms and R, C, U and r are fixed during running time, we summarize the following observations:

1. In spatial parallelism, L_{sr} grows with $iter$ slightly more than linearly, while L_{ss} grows with $iter$ exactly linearly. It shows that both solutions provide proximate performance when the

iteration number is relatively small. But as the iteration number increases, border streaming will outperform redundant computation. For the two hybrid parallelism alternatives, L_{hr} and L_{hs} have the same relation as the one between L_{sr} and L_{ss}.

2. Comparing spatial parallelism with temporal parallelism, when $iter$ is large enough and $iter$ is divisible by s_t, temporal parallelism could achieve a similar performance to spatial parallelism, as the value of s_t would be set to the same as k_{sr} and k_{ss}. In addition, temporal parallelism requires much less amount of off-chip bandwidth. However, when $iter$ is small enough, the largest value of s_t is the same as $iter$, while k_{sr} and k_{ss} can be much larger than $iter$ by exploiting off-chip memory bandwidth (especially on HBM-based FPGAs). This will bring significant performance degradation for temporal parallelism. In this case, hybrid parallelism can further improve the performance with less bandwidth requirement. Finally, when $iter$ is not divisible by s_t (or s_{hr}, or s_{hs}), L_t (or L_{hr}, or L_{hs}) will also suffer some overhead to process the whole input data with some PEs idle in the last round.

4.3 Code Generator and Automation Tool Flow

Figure 7 shows an overview of the automation flow for SASA. It takes a stencil DSL and FPGA platform information as input, and automatically generates optimized FPGA accelerator design with the best parallelism optimization as the output. To address the timing closure issue (and conduct a fairer comparison between different parallelism implementations), we have integrated the open source TAPA/AutoBridge framework [5,14] into our SASA framework to build our generated multi-PE design. TAPA/AutoBridge is a high-performance fast-compiling HLS framework that is fully compatible with the Xilinx Vitis/Vivado workflow. It takes in task-parallel program in Vitis HLS syntax with additional TAPA APIs. It has three major advantages. First, it supports easier programming of task-parallel dataflow programs in C++, without the need of the more complex OpenCL approach (using multiple OpenCL kernels) to support task parallelism. Our SASA code generator automatically converts the stencil DSL to the TAPA HLS and host code. Second, it replaces the resource-inefficient AXI interface with a lightweight streaming interface to access off-chip memory. The standard AXI interface always buffers data in BRAM and consumes a significant amount of resources on the bottom die of the HBM-based U280 FPGA (with multiple AXI interfaces), which often causes place-and-route congestion and timing violation timing congestion on the bottom die. With the lightweight streaming interfaces, it saves resources for actual computations and reduces place-and-route congestion and timing violation timing congestion on the bottom die. Third, it automatically applies coarse-grained floorplanning and pipelining optimizations to improve the timing closure for dataflow programs, and can often greatly improve the design build success rate and the final design frequency. With this integration, we are able to generate high-frequency stencil accelerators.

The detailed steps inside the SASA automation flow are described as below.

1. Our code generator first parses the user programmed stencil DSL for a given stencil application and generates the optimized single PE design in Vitis HLS C++ code. To do this, our code generator uses a Python based meta-language specification, *textX*[8], which uses meta-model to define a DSL. With our pre-defined meta-model grammar as illustrated in Section 4.1, our compiler parses the DSL, generates the abstract syntax tree (AST), and extracts the user-defined stencil configurations. The stencil configurations include the number of input rows (R), the number of input columns (C), the number of stencil iterations ($iter$), and the stencil radius size (r). Note that for a multidimensional array specified in the DSL, our code generator flattens all the dimensions except the first dimension into one dimension. Take 3D stencil input size 256×16×16 as an example, we buffer two rows of 16×16 data in the row buffer like a 2D stencil

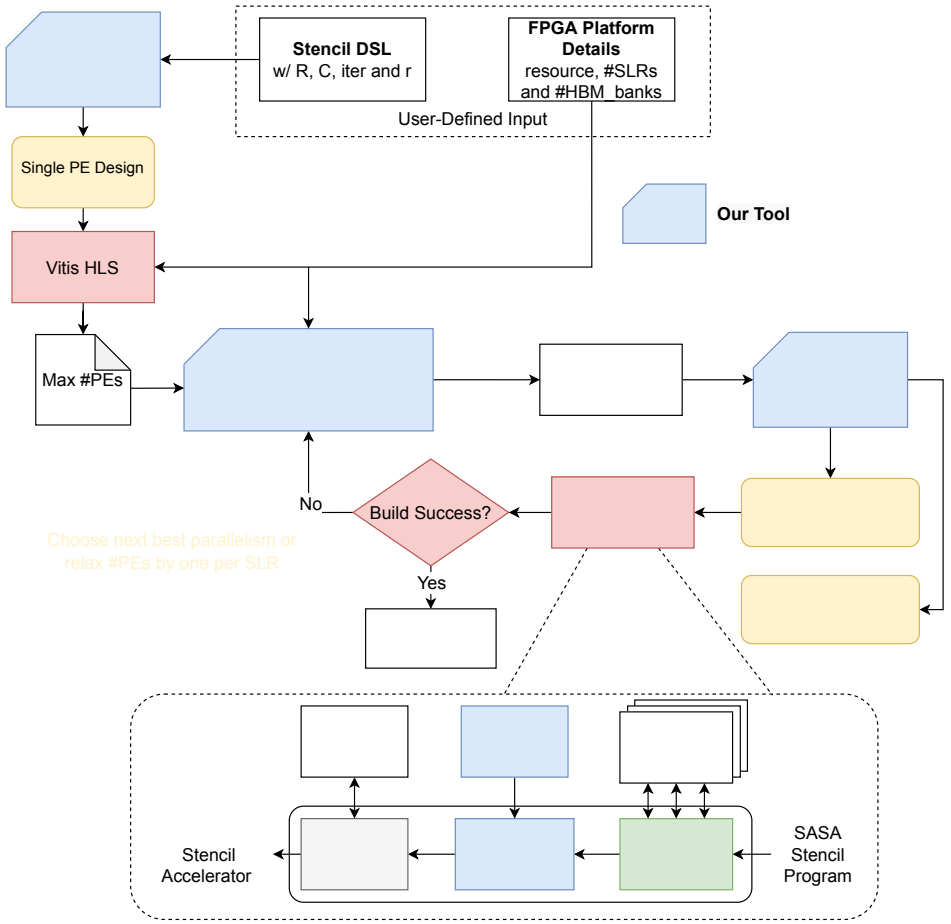


Fig. 7. Overall automation tool flow in SASA

kernel. The difference between 2D and 3D stencil accelerator designs is that they read data from different locations of the row buffer when updating each cell. Then, it coverts the AST into a model consisting of Python objects. The code generator further interprets the model to analyze the data dependency in each statement between the input(s) and output(s). After that, the code generator generates the Vitis HLS C++ code for a single PE design presented in Section 3.1, based on the user-defined configurations (i.e., R, C, $iter$, r) extracted from the DSL. The unroll factor (i.e., the number of PUs inside each PE in Figure 3), U (e.g., 16), is chosen based on the AXI interface width (e.g., 512-bit) of a single memory bank and the size of each stencil data cell (e.g., 32-bit) to saturate the off-chip bandwidth.

2. To determine the maximum number of PEs that can be instantiated on the FPGA platform, we first estimate the resource utilization of the single-PE design generated from the code generator block, by running Vitis HLS [28] synthesis. Then, combined with the FPGA platform specification and hardware utilization constraints, we determine the maximum PE number as described in Equation 1, 2 and 3.

3. Once the $\#PE_{res}$, $\#PE_{bw}$, and $Max\ \#PEs$ are determined, we explore different temporal and spatial parallelism configurations of the multi-PE design based on the analytical performance model presented in Section 4.2, and choose the optimal design choice such that it achieves the least execution latency, based on Equations 4 to 9. For temporal parallelism, we set $s_t = \#PE_{res}$ in Equation 4. For the spatial parallelism alternatives, we set $k_{sr} = k_{ss} = Max\ \#PEs$ in in Equations 5 and 6. For the two hybrid parallelism implementations, we explore all combinations of (k_{hr}, s_{hr}) and (k_{hs}, s_{hs}) that meets $k_{hr} \times s_{hr} = k_{hs} \times s_{hs} = Max\ \#PEs$, $k_{hr} \leq PE_{bw}$, and $k_{hs} \leq PE_{bw}$. in Equations 7 and 8. To simplify the floorplanning, we limit the number of FPGA spatial PE groups k_{hr} and k_{hs} to be a multiple of $\#SLRs$, so that we have a very small number of (k_{hr}, s_{hr}) and (k_{hs}, s_{hs}) pairs to explore. Our analytic model will select the best multi-PE design choice with the best parallelism. When multiple parallelisms achieve a similar performance, we choose the most resource-efficient one. For example, $Spatial_S$ and $Hybrid_S$ are the two best choices among many configurations, then our model will choose $Hybrid_S$ as it uses fewer HBM banks.
4. Once the best multi-PE design choice is selected, our code generator will automatically generate the corresponding multi-PE accelerator design in TAPA HLS C++ [5], based on the multi-PE architecture presented in Section 3 and single-PE design generated in step 1. Moreover, we will also automatically generate the corresponding host code with TAPA API to manage this FPGA kernel, which includes common FPGA device setup, host buffer allocation, data communication between the host and the FPGA, and signal to start the FPGA kernel execution.
5. Finally, we build the optimal design from our code generator using Xilinx Vitis 2020.2 tool to generate the final FPGA bitstream and host executable. If the design is successfully built and meets the frequency requirement, it will be output as the optimal design. Otherwise, our automation tool will first attempt to build the next best parallelism design with the same number of PEs. If none of those designs can pass the requirement, our tool will lower the number of PEs by the number of SLRs (i.e., $Max\ \#PEs = Max\ \#PEs - \#SLRs$) and repeat steps 3 to 5 until the design can be successfully built.

Code generator is one fundamental block of our automation framework, which is utilized at two different stages of the automation flow. First, after the stencil DSL is parsed and interpreted, the code generator needs to automate the generation of a single-PE design. At this point, only the datapath logic is defined based on the stencil operation, and the fine-grained data parallelism is set to match the off-chip memory bandwidth to enable the dataflow computing requirement. The second function of the code generator is to automate the multi-PE binding code generation when the number of PEs and the optimal design parallelism settings have been chosen by our analytical performance model. This time the code generator will return a software driver code to run on the host CPU and an optimized stencil accelerator design to deploy on the chosen FPGA platform.

In summary, with our automation framework SASA, for a given FPGA platform, users can easily define the stencil computing parameters (i.e., input and output data dimension, iteration number, and stencil operation) through a high-level DSL. To automate the design space exploration, we derive analytical models for all five types of parallelisms shown in Figure 4, 5 and 6. As a result, our automation framework supports arbitrary stencil workload and can generate performance portable accelerator designs with the optimized parallelism across different HBM-based FPGAs.

5 EXPERIMENTAL RESULTS

In this section, we conduct a comprehensive evaluation of our proposed framework SASA and compare it to state-of-the-art automatic stencil acceleration framework SODA [4], which only exploits temporal parallelism. First, we introduce the experiment setup of our evaluation. Second,

we present the improvement of our single PE optimization over SODA. Finally, we compare different parallelism optimizations, and discuss the results of the best parallelism configuration.

5.1 Experimental Setup

We evaluate a wide range of stencil benchmarks including:

1. JACOBI2D/3D from from SODA testbench. They are a 2D 5-point stencil kernel and a 3D 7-point stencil kernel, respectively. They are used in linear algebra algorithms to find the solution for linear equations.
2. BLUR from SODA testbench [4]. It is a 2D 9-point stencil kernel. It is commonly used for edge smoothing and noise removing in image processing domains.
3. SEIDEL2D from SODA testbench. It is a 2D 9-point stencil kernel and used in linear algebra to solve a system or linear equations.
4. DILATE from the Rodinia-HLS benchmark suite [6]. It is a 2D 13-point stencil kernel and used to detect and track leukocyte of blood vessel in biomedical research.
5. HOTSPOT from the Rodinia-HLS benchmark suite. It is a 2D 5-point stencil kernel with two inputs and one output. It is used to estimate processor temperature based on power grid and temperature of the corresponding area.
6. HEAT3D from SODA testbench. It is a 3D 7-point stencil kernel and used for heat diffusion simulation.
7. SOBEL2D from SODA testbench. It is a 2D 9-point stencil kernel and used for image processing, particularly for edge detection.

Table 3. Iteration counts scope of different stencil benchmarks

	Iteration counts
JACOBI2D	many, e.g., 256
JACOBI3D	many, e.g., 256
BLUR	few, e.g., 4
SEIDEL2D	many, e.g., 256
DILATE	non-iterative, i.e., 1
HOTSPOT	many, e.g., 256
HEAT3D	many, e.g., 256
SOBEL2D	non-iterative, i.e., 1

We use four different input sizes, 256×256, 720×1024, 9720×1024 and 4096×4096, when evaluating all the 2-dimensional stencil benchmarks; and use $256 \times 16 \times 16$, $720 \times 32 \times 32$, $9720 \times 32 \times 32$ and $4096 \times 64 \times 64$, input sizes for the 3-dimensional stencil benchmarks. Furthermore, we sweep the iteration number from 1 to 256 at a power of 4 increment. We also list the rough scope of iteration counts in Table 3 to demonstrate the common usage of different stencil benchmarks. Some benchmarks, such as DILATE and SOBEL2D, are non-iterative kernels in most applications. We still evaluate the performance of those kernels in different iteration counts to illustrate the performance difference of different parallelism optimizations. We analyze the throughput of input size 720×1024 and $720 \times 32 \times 32$ in Section 5.3, and other input sizes in appendix. Note that when the iteration number is 1, spatial parallelism and hybrid parallelism will be the same and have the same throughput. All these stencil kernels are written in the stencil DSL as illustrated in Section 4.1.

We evaluate SASA on Xilinx Alevo U280 datacenter FPGA board with 32 HBM2 banks [27]. First, SASA compiles the stencil DSL into the optimized FPGA design in Xilinx Vitis HLS C++ with TAPA

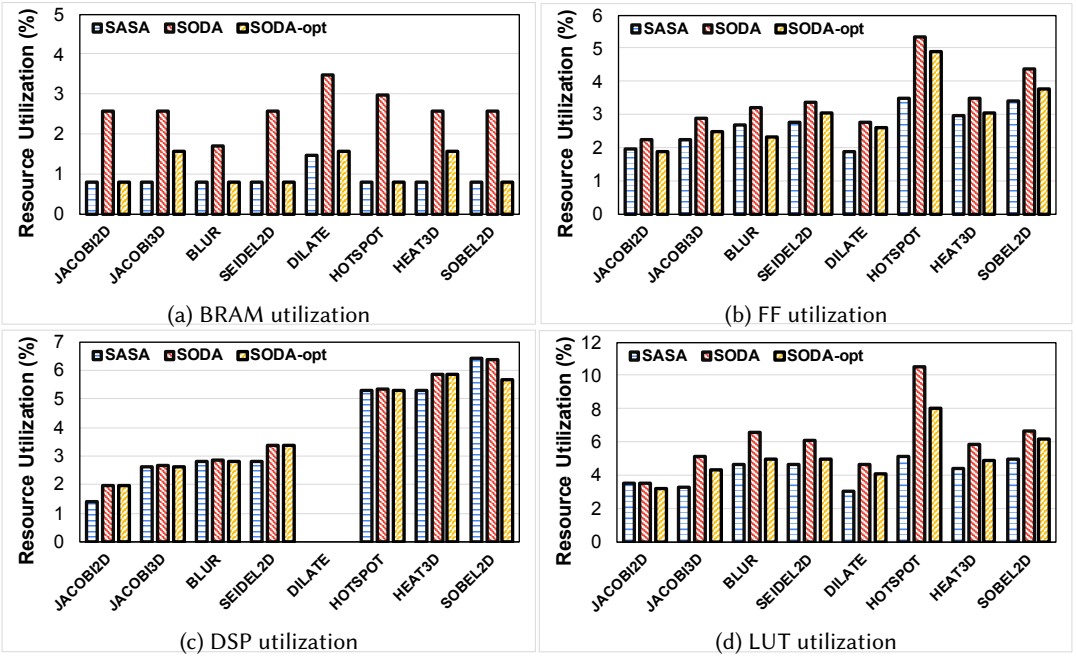


Fig. 8. Resource utilization of a single PE saturating one HBM bank bandwidth on Alveo U280 for the input size of 9720×1024 and 9720×32×32

APIs [5] and the corresponding host code. Then it uses AutoBridge [13] to do the floorplanning and pipelining optimizations for our design and Vitis 2020.2 [28] to build the generated design to run on the U280 FPGA. We set 225 MHz as the target frequency of our designs since all of them use 512-bit wide streaming connections and can already fully utilize the effective bandwidth from each HBM memory bank. This is because on the U280 FPGAs, the two HBM stacks operate at 450 MHz, and are connected to 32 hardened AXI ports in width of 256-bit. Therefore, to achieve the HBM memory bandwidth using a 512-bit AXI port, the kernel frequency needs to be above 450 MHz × 256-bit / 512-bit = 225 MHz. And the theoretical peak bandwidth of a single 512-bit AXI port accessing a single HBM bank is 512 bits/cycle × 225 MHz / 8 bits-per-byte = 14.4 GB/s.

5.2 Results for Single PE Optimization

To demonstrate the quality of our design, we first evaluate our optimized single PE design, which accelerates one stencil iteration using the optimized streaming access from one HBM bank. From the performance perspective, it saturates the bandwidth of a single HBM bank by placing 16 parallel PUs (processing units) inside each PE to execute in a fully streaming fashion. Therefore, it achieves the optimal performance given one HBM bank, which is the same as SODA [4] that uses the optimal data reuse size and memory access requirement.

Next, we focus on the comparison of its resource consumption. Figure 8 shows an overall resource utilization comparison with original SODA and optimized SODA (i.e., SODA-opt) that is integrated with TAPA/AutoBridge [5, 13] for a fair comparison, including BRAM, FF, DSP and LUT consumption. Compared to the original SODA, the major benefit of our design comes from removing the on-chip line buffer by introducing the coalesced reuse buffer design. It brings a 4.3%-69.8% reduction in the BRAM utilization compared to the previous SODA design. Consequently, the

BRAM reduction further reduces the FFs and LUTs consumption of the design by 12.9%-34.8% and 1.8%-51.7%, respectively. Since both of SODA and our design use the same fine-grained parallelism and place 16 PUs inside each PE (i.e., loop unroll factor $U = 16$), we both achieve the same DSP utilization. Note that DILATE only has boolean logic operations and thus does not utilize any DSP resource.

For the majority of benchmarks, both our implementation and the optimized SODA achieve a similar amount of resources.

5.3 Results for Different Multi-PE Parallelisms

In this subsection, we first validate the accuracy our analytical performance model. Then we evaluate the performance trend of temporal parallelism, two spatial parallelisms, and two hybrid parallelisms, respectively, when the number of iterations changes. Finally, we compare the performance between temporal, spatial, and hybrid parallelisms and summarize the best parallelism configurations.

All results for different parallelisms are summarized in Figure 10 (for input size of 720×1024), and Figure 13 to Figure 20 in the Appendix (for all input sizes), which are measured using the common throughput metric GCell/s (i.e., how many billion of stencil data cells it can process per second). We also mark the best parallelism predicted by our automation flow in Figure 10, and Figure 13 to 20 in the Appendix, with a red bar. Note that when iteration count is 1, both spatial parallelism and hybrid parallelism are the same. When two parallelism optimizations have very close predicted performance (within 2% predicted performance difference), our tool chooses the most resource-efficient one: for example, *Spatial_R* is favored over *Spatial_S* since *Spatial_S* costs slightly more wires to implement streaming connections, and *Hybrid_S* is favored over *Spatial_S* since *Hybrid_S* uses fewer HBM banks. Finally, the total number of PEs for different parallelisms are summarized in Figure 11 (for column size of 1024), and Figure 21 (for column size of 256) and Figure 22 (for column size of 4096) in the Appendix, which will be used to explain our results.

5.3.1 Performance Model Accuracy. To evaluate the accuracy of our analytical performance model, we run a wide range of configurations, including different iteration numbers and different parallelism optimizations for each stencil kernel, and compare the model predicted execution time with the actual measured time of on-board execution. Figure 9 shows the average (histogram), maximum (top bar), and minimum (bottom bar) error rates of our performance model for each stencil benchmark with different parallelism optimizations. For each histogram, the error rate is averaged across different numbers of stencil iterations from 1 to 64. For all configurations, our performance model has an error rate within 5% in estimating the performance of our accelerator designs.

5.3.2 Performance Results of Temporal Parallelism Designs. As shown in Figure 10, the performance of the temporal parallelism designs generally increases with the iteration number, as more stencil iterations are concurrently processed on the FPGA in a dataflow fashion. This linear performance improvement trend stops when we could not instantiate more temporal stages (i.e., stencil iterations) on the FPGA. For most benchmarks, their maximum number of PEs in temporal parallelism designs are between 9 to 15 when their iteration number is large enough, as shown in Figure 18 to 20. Therefore, their throughput increases linearly as the iteration number grows from 1 to 4. The two exceptions are JACOBI2D and DILATE. Their linear throughput increase is achieved when iteration ranges from 1 to 16 since their maximum PE numbers are 21 and 18, respectively.

When the iteration number is larger than the maximum number of PEs, this performance does not improve linearly with the iteration number; the performance is mainly decided by the ratio of iteration number and rounds of FPGA kernel execution. For example, in BLUR, when the iteration number is 64 and 16, respectively, the maximum number of PEs is 12 in both cases; therefore, the numbers of FPGA kernel runs are 6 and 2, respectively. While the work to be done is increased by

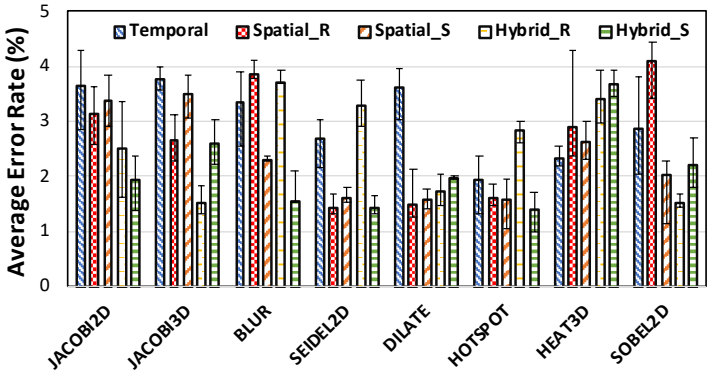


Fig. 9. Accuracy of our analytical performance model

fourfold from iteration number 16 to 64, the execution time is only increased by $6/2 = 3\times$. In this way, the throughput of iteration number 64 is larger than that of iteration number 16 as shown in Figure 10a.

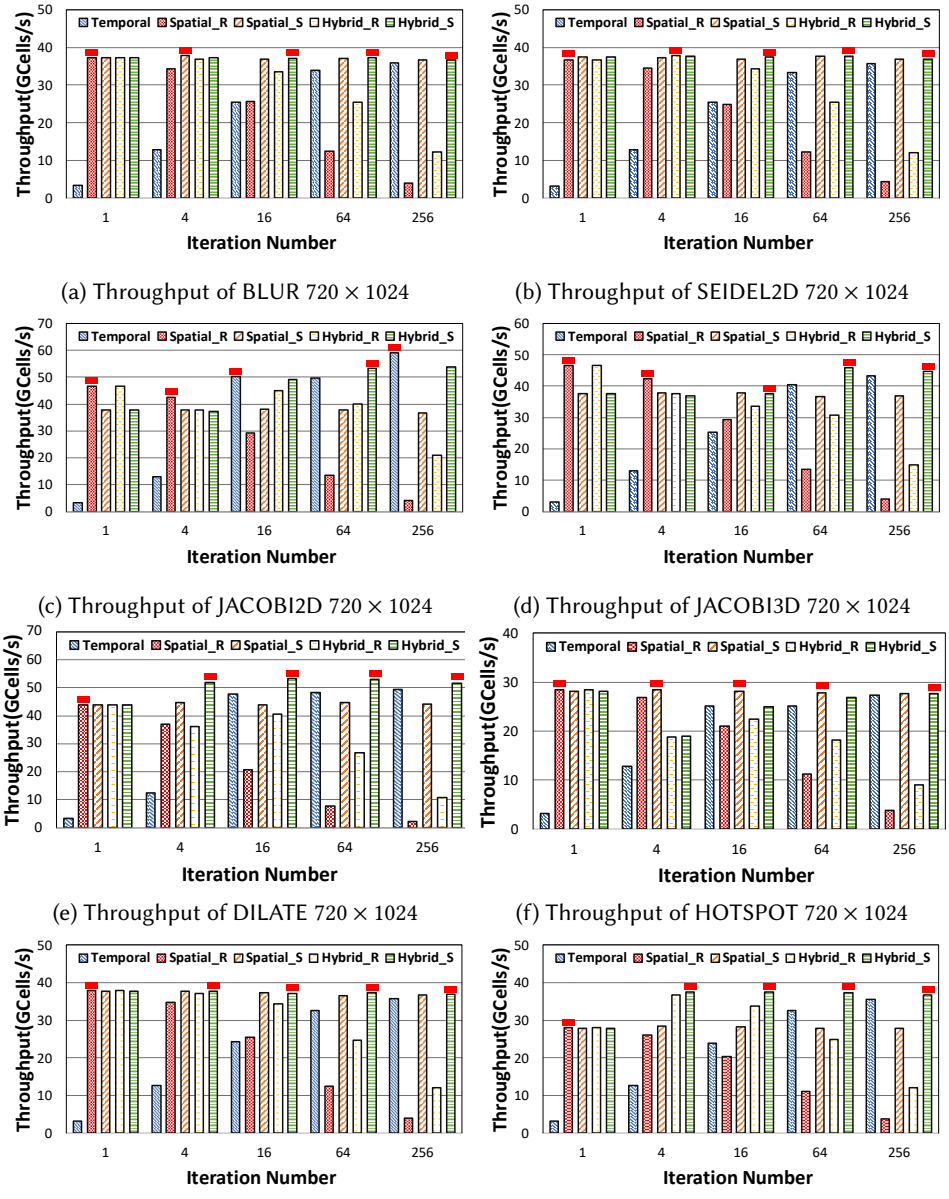
5.3.3 Performance Results of Spatial Parallelism Designs. To better understand the performance difference between the two spatial parallelism design variants presented in Section 3.3, *Spatial_R* and *Spatial_S*, we further analyze the performance trend of these two designs at different iteration numbers, input sizes and stencil kernels. As shown in Figure 10, for the *Spatial_R* design, its performance generally decreases as the iteration number increases. This is mainly due to the increase of the halo data processing as the iteration number increases. The performance decrease is worse on smaller input sizes as halo data increase has more significant impact on smaller input sizes. For example, the throughput of *Spatial_R* drops faster at 256×256 and 720×1024 input sizes in JACOBI2D compared with 9720×1024 and 4096×4096 input sizes. On the other hand, the performance of the *Spatial_S* design does not vary with the iteration number. This is because the amount of halo data exchange remains the same as the iteration number increases. These trends align with our performance model in Equations 5 and 6, respectively.

Comparing between these two design variants, when the iteration number is low (i.e., less than 4) and with the same number of PEs, as shown in Figure 10, *Spatial_R* and *Spatial_S* achieve about the same throughput. And as the iteration number increases, the *Spatial_S* design can maintain its performance and outperforms the *Spatial_R* design especially for smaller input sizes.

As shown in Figure 10c and 10d, there are a few exceptions, JACOBI2D and JACOBI3D, where *Spatial_R* design achieves a better throughput than the *Spatial_S* design as *Spatial_R* can place more PEs. This is because, border streaming based approach consumes slightly more wires due to the streaming connections than redundant computation based approach to implement border streaming, which affects timing closure, especially when the increase of cross-SLR (i.e., cross-die) connections is approaching FPGA board limit.

5.3.4 Performance Results of Hybrid Parallelism Designs. In hybrid parallelism designs, both temporal and spatial parallelisms are exploited. The performance from *Hybrid_R* and *Hybrid_S* parallelism designs reflects a combination of trend from both the temporal parallelism design as described in Section 5.3.2 and the spatial designs discussed in Section 5.3.3.

1. When the iteration number is 1, the hybrid parallelism is the same as spatial parallelism, since each spatial PE group has only one temporal stage. When iteration number is larger than 1,



(a) Throughput of BLUR 720 × 1024

(b) Throughput of SEIDEL2D 720 × 1024

(c) Throughput of JACOBI2D 720 × 1024

(d) Throughput of JACOBI3D 720 × 1024

(e) Throughput of DILATE 720 × 1024

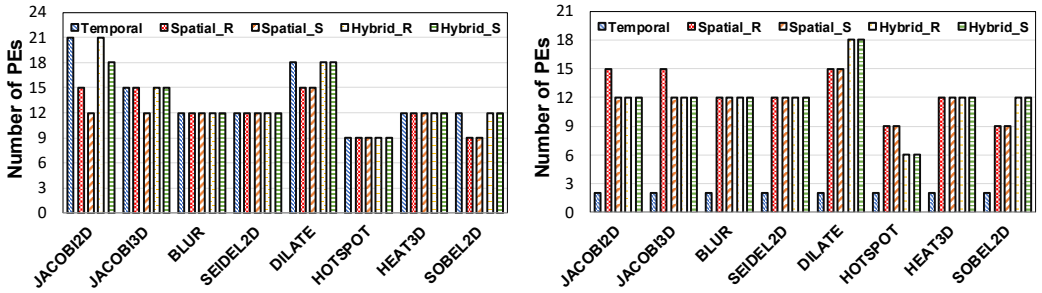
(f) Throughput of HOTSPOT 720 × 1024

(g) Throughput of HEAT3D 720 × 1024

(h) Throughput of SOBEL2D 720 × 1024

Fig. 10. Throughput (GCell/s) comparison of different parallelism optimizations with the number of iterations changing from 1 to 256, with input size of 720 × 1024.

there are multiple combinations of spatial parallelism degree and temporal parallelism degree. For example, in JACOBI3D, maximum number of PEs can be implemented is 15, as shown in Figure 10d. We choose the degree of spatial parallelism based on the number of SLRs, which is 3 on Alevo U280 board. When iteration number is 4, 6 spatial PE groups with 2 temporal stages



(a) Number of PEs with the number of iteration = 64 (b) Number of PEs with the number of iteration = 2

Fig. 11. Total number of PEs for different parallelisms on Alveo U280 with column size = 1024

will outperform 3 spatial PE groups with 5 temporal stages even with less PEs. This is because the former can utilize more off-chip memory bandwidth without idle PEs.

2. Hybrid parallelism has a similar trend as spatial parallelism with the increase of the iteration number, when the ratio of iteration number and rounds of FPGA kernel execution maintains the same, especially when the iteration number is small. The effect of this ratio is illustrated in Section 5.3.2. For example, in BLUR, SEIDEL2D and HEAT3D, the throughput of *Hybrid_R* decreases as the iteration number increases in the range from 4 to 256, since the ratio of iteration number and rounds of FPGA kernel execution does not change. And the throughput of *Hybrid_S* in the same iteration range stays the same as the pattern of *Spatial_S* since they have the same number of PEs.
3. However, when iteration number becomes large enough and this ratio of iteration number and rounds of FPGA kernel execution changes, the throughput of hybrid parallelism will have a noticeable change, as temporal parallelism plays a heavier role. Such pattern is more outstanding at small input size, like 256×256. For example, in JACOBI2D, DILATE, and HOTSPOT, the throughput of *Hybrid_R* decreases when iteration number ranges from 16 to 256, reflecting the characteristics of spatial parallelism. However, there is a big performance boost when the iteration number changes from 4 to 16, since the ratio changes and temporal parallelism play a heavier role.

For most benchmarks, *Hybrid_R* and *Hybrid_S* have the same number of PEs. *Hybrid_S* achieves a similar performance to *Hybrid_R* at a small iteration number. At a large iteration number, *Hybrid_S* outperforms *Hybrid_R*, because the *Hybrid_R* design requires redundant computation for more halo data than *Hybrid_S*. Lastly, there is only one case where the border streaming based approach achieves fewer PEs than the redundant computation based approach. Specifically, for JACOBI2D at 9270×1024 and 4096×4096, *Hybrid_S* has fewer PEs than *Hybrid_R*. As a result, when the iteration number is 32, the performance of *Hybrid_R* is better than *Hybrid_S*. However, such advantage is offset by the redundant halo computation overhead for other iteration numbers.

5.3.5 Performance Impact by Different Input Sizes. For the four different stencil input sizes, 256×256, 720×1024, 9720×1024, and 4096×4096 (for 2D stencils), we have made the following observations.

First, for the majority of our stencil benchmarks under these input sizes (more specifically, different column sizes), the row buffer resource consumption did not become a bottleneck, as each PE roughly needs to buffer only two rows of data on-chip.

Second, the row sizes do have a performance impact, especially for the redundant computation based spatial parallelism (*Spatial_R*) and hybrid parallelism (*Hybrid_R*). With a smaller row size (e.g., 256), when the iteration count becomes larger, the performance of *Spatial_R* decreases

significantly as the redundant computation adds a very significant overhead. Therefore, the border streaming based spatial parallelism (*Spatial_S*) is a better choice. While with a larger row size (e.g., 9720), such overhead is much smaller and the difference between *Spatial_R* and *Spatial_S* is marginal. A similar performance impact is observed for *Hybrid_R*.

Third, in general, the overall throughput for the small 256×256 input size is relatively lower than those with larger input sizes. The reasons are twofold. First, the execution time of extra halo regions for a smaller input size occupies a high execution time percentage, i.e., the overhead is bigger. Second, with the smaller input size, the memory burst size for each HBM bank is relatively small, thus leading to lower off-chip memory bandwidth utilization.

5.3.6 Performance Comparison between Temporal, Spatial, and Hybrid Parallelisms. Overall, temporal parallelism achieves the lowest performance amongst all parallelism variants when the iteration count is low. As shown in Figure 10, when the iteration count is low, e.g., 1 or 4, temporal parallelism cannot efficiently exploit the HBM memory bandwidth. Even when the iteration count is as large as 64, temporal parallelism also may not give the best performance since the iteration count may not be evenly divisible by the temporal stages instantiated on hardware. Take JACOBI2D as an example, there are 21 temporal stages on the hardware as shown in Figure 22a. When its iteration count is 64, it needs to execute the hardware ceil (64 / 21) = 4 rounds. In the last round, there is only one last iteration (64 - 21 $\times$ 3 = 1) that needs to be executed; 20 temporal stages on hardware are under-utilized. When the iteration count becomes 256, the performance difference between temporal parallelism and hybrid parallelism (Hybrid_S) becomes much smaller, sometimes almost the same. The reason is that the performance overhead caused by temporal unrolling factors not evenly divided by the required number of iterations becomes much smaller.

For the remaining parallelism variants, spatial and hybrid, boarder streaming based approach generally achieves better performance than the redundant computation based method as detailed above in Section 5.3.3 and Section 5.3.4. However, depending on stencil kernel, iteration number, and input sizes, the best parallelism may vary.

First, there are cases where *Spatial_S* and *Hybrid_S* achieve a similar performance and are the best among all parallelisms, specifically for BLUR, SEIDEL2D, and HEAT3D kernels. The reason is that both parallelisms have 12 PEs and can fully utilize them under different iteration numbers and input sizes. Specifically for *Hybrid_S*, when the iteration number is 2, the degree of spatial parallelism is 6 and the degree of temporal parallelism is 2; when the iteration number larger than 2, the degree of spatial parallelism is 3 and the degree of temporal parallelism is 4. Therefore, all 12 PEs can be fully utilized with different iteration numbers.

Second, there are cases where *Hybrid_S* outperforms *Spatial_S* and *Hybrid_S* is the best among all parallelisms, specifically for DILATE, SOBEL2D, JACOBI2D with large iteration number, and JACOBI3D with large iteration number. There are two reasons behind this: 1) for DILATE and JACOBI2D, due to the HBM bank (i.e., bandwidth) restriction, *Spatial_S* has fewer PEs than *Hybrid_S*; 2) for SOBEL2D and JACOBI3D, *Spatial_S* has fewer PEs than *Hybrid_S* as it is harder to pass the timing closure.

Third, there is also one case where *Spatial_S* achieves a better performance than *Hybrid_S* and is the best, specifically for HOTSPOT at a small iteration number. In fact, both *Spatial_S* and *Hybrid_S* have 9 PEs in this case. However, in *Hybrid_S*, the degree of spatial parallelism is 3 and the degree of temporal parallelism is 3, which cannot be evenly divided by the iteration number. As a result, some PEs are underutilized in *Hybrid_S*, leading to a lower performance than *Spatial_S*.

Fourth, there are two exceptional cases where *Spatial_R* performs better than *Hybrid_S* and is the best, specifically for JACOBI2D and JACOBI3D when the iteration number is small and the number of input rows is large. This is because the *Hybrid_S* significantly under-utilizes the number

Table 4. Configuration of the best parallelism on Alveo U280, for the input size of 9720×1024

	Iteration = 64					Iteration = 2				
	Parallelism	Frequency	k	s	#HBM banks	Parallelism	Frequency	k	s	#HBM banks
JACOBI2D	Hybrid_S	250 MHz	3	7	6	Spatial_R	233 MHz	15	1	30
JACOBI3D	Hybrid_S	250 MHz	3	5	6	Spatial_R	226 MHz	15	1	30
BLUR	Hybrid_S	249 MHz	3	4	6	Spatial_R	229 MHz	12	1	24
SEIDEL2D	Hybird_S	225 MHz	3	4	6	Spatial_R	225 MHz	12	1	24
DILATE	Hybrid_S	250 MHz	3	6	6	Hybrid_S	250 MHz	6	2	12
HOTSPOT	Hybrid_S	250 MHz	3	3	9	Spatial_S	250 MHz	9	1	27
HEAT3D	Hybrid_S	225 MHz	3	4	6	Spatial_R	230 MHz	12	1	24
SOBEL2D	Hybrid_S	250 MHz	3	4	6	Hybrid_S	250 HHz	3	4	6

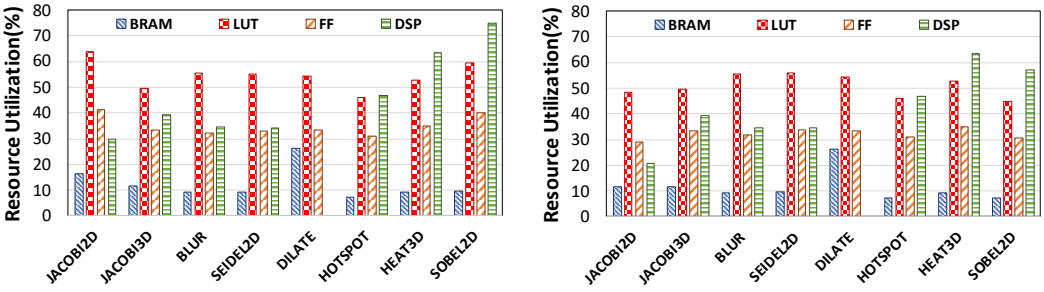
of PEs when the iteration count is small, especially when iteration count is 2 or 4. For example, *Spatial_R* of JACOBI3D can utilize all 15 PEs when the iteration number is 2, while *Hybrid_S* can only utilizes 12 PEs with the best configuration of 6 spatial PE groups and 2 temporal PEs in each group.

Lastly, there are some exceptions where temporal parallelism achieves the best performance when the iteration count is 256. Specifically for JACOBI2D with iteration count of 256 at input size of 720×1024, 9720×1024 and 4096×4096, temporal parallelism achieves the best performance. The reason is that the PE number of temporal parallelism is larger than the one of hybrid parallelism (*Hybrid_S*), shown in Figure 11a and 22a. There are also several cases where temporal parallelism achieves very close performance to *Hybrid_S* due to the large iteration count of 256 and our tool chooses temporal parallelism as the final one since it is more resource-efficient.

5.3.7 The Best Parallelism Configurations and Their Resource Utilization. As discussed above, the best parallelism optimization varies with the stencil benchmark and the number of iterations. Table 4 summarizes the best parallelism configuration for each benchmark for the input size of 9720×1024, when the number of iterations is 64 and 2, respectively. When the number of iterations is 64, *Hybrid_S* achieves the best performance for all benchmarks as it is not affected by the redundant halo computation overhead. Note that one advantage of hybrid parallelism over spatial parallelism is that it requires much less off-chip bandwidth (shown as the number of HBM banks in Table 4). When the number of iterations is 2, spatial parallelism achieves the best performance for most benchmarks for most of the benchmarks; both *Spatial_R* or *Spatial_S* achieve a similar performance. There are some exceptions, DILATE and SOBEL2D, where *Hybrid_S* achieves the best performance. This is because their *Spatial_R* and *Spatial_S* parallelism implements less number of PEs due to the limitation of the available HBM banks and timing closure issues, respectively.

For the best parallelism configurations, the degree of spatial parallelism (k) and the number of temporal stages (s) are also included in Table 4. These number also vary between benchmarks, which again highlights the importance of an automation framework to compile the high-level DSL to the optimized FPGA design. All of our designs achieve a clock frequency of at least 225 MHz to fully utilize bandwidth of each HBM bank.

Finally, we also show the utilization of on-chip resources and off-chip HBM banks for the best parallelism configurations in Figure 12 and Table 4, respectively. The bottleneck resource changes as the computation intensity increases. As shown in Figure 12, for benchmarks with lower computation intensity, such as JACOBI2D, JACOBI3D, BLUR, SEIDEL2D, and DILATE, LUT has the highest resource utilization rate compared with other resources. For benchmarks with higher computation



(a) Resource utilization of the best parallelism with the number of iteration = 64

(b) Resource utilization of the best parallelism with the number of iteration = 2

Fig. 12. Resource utilization of the best parallelism configuration on Alveo U280, for the input size of 9720×1024

intensity, such as HOTSPOT, HEAT3D, and SOBEL2D, DSP is the bottleneck to scale up to more PEs.

5.4 Comparison to Prior Work

As discussed in Section 3.1, SODA [4] is less efficient than our SASA temporal parallelism implementation due to the additional on-chip line buffer usage. To conduct a fair comparison between SODA and SASA , we integrate SODA with TAPA/AutoBridge [5, 13] to address the major resource inefficiency, and rerun all the experiments on the same HBM-based U280 FPGA. The on-chip line buffer to buffer the input data from off-chip memory, shown in Figure 8, is also removed in this integration, since TAPA replaces the resource-inefficient AXI interface with a lightweight streaming interface. As a result, both SODA and SASA temporal parallelism implementation achieve the same performance. However, SODA only supports temporal parallelism, and does not support other types of parallelisms that we have explored in this paper; therefore, its performance is sub-optimal when the iteration count is small or the iteration count cannot be evenly divided by the temporal stages in the hardware design. Compared to SODA, SASA achieves better throughput with an average of at least 3.41× speedup across all configurations. The highest speedup over temporal parallelism is reached in JACOBI3D when iteration number is 1, where redundant computation based spatial parallelism can reach 15.73× speedup.

The stencil accelerator design proposed in [29] only supports temporal parallelism and its throughput is measured when the iteration count is super large. For temporal parallelism, the throughput is determined by the bandwidth of a single memory bank. Since their FPGA uses DDR4, which has a higher bandwidth (19.2GB/s theoretical bandwidth) than that of a single HBM bank (14.4GB/s theoretical bandwidth) in our results, their reported GCell/s is higher than ours. However, for the given HBM bank, our implementation already achieves the best performance that the HBM bank can achieve. More importantly, we have explored different parallelisms and can automatically generate the design with the best parallelism.

6 CONCLUSION

In this paper we propose a scalable and automatic stencil acceleration framework on modern HBM-based FPGAs called SASA. In terms of the accelerator design architecture, SASA employs a multi-PE approach to exploit temporal and spatial parallelisms for better scalability. Each single PE design is optimized for on-chip data reuse, off-chip memory access, and the on-chip buffer usage. For design automation, SASA provides a high-level DSL for domain experts to configure and define

the stencil operation. Then a code generator automatically explores the design space based on our analytical performance model and generates an optimized stencil accelerator design with the best parallelism optimization. Experimental results across a wide range of stencil benchmarks show that our SASA can achieve an average speedup of 3.41 × and up to 15.73× speedup on the HBM-based Xilinx Alveo U280 FPGA, compared to state-of-the-art automatic stencil acceleration framework SODA [4] that only exploits temporal parallelism. Finally, we plan to open source our tool in the near future at this link: https://github.com/SFU-HiAccel/SASA.

ACKNOWLEDGEMENTS

We acknowledge the partial support from Natural Sciences and Engineering Research Council of Canada (NSERC Discovery Grant RGPIN-2019-04613 and DGECR-2019-00120, Alliance Grant ALLRP-552042-2020); Canada Foundation for Innovation John R. Evans Leaders Fund and British Columbia Knowledge Dev. Fund; Simon Fraser University New Faculty Start-up Grant; Mitacs Globalink Research Internship Award; Huawei Canada, Xilinx, and Nvidia.

REFERENCES

[1] Falah Alobaid, Nabil Baraki, and Bernd Epple. 2014. Investigation into improving the efficiency and accuracy of CFD/DEM simulations. *Particuology* 16 (2014), 41–53.

[2] 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, Article 53 (dec 2015), 26 pages.

[3] Yuze Chi and Jason Cong. 2020. Exploiting Computation Reuse for Stencil Accelerators. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. Article 184, 6 pages.

[4] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with Optimized Dataflow Architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[5] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 204–213. https://doi.org/10.1109/FCCM51124.2021.00032

[6] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 93–96. https://doi.org/10.1109/FCCM.2018.00023

[7] Patrick Cooke, Jeremy Fowers, Lee Hunt, and Greg Stitt. 2013. A High-Performance, Low-Energy FPGA Accelerator for Correntropy-Based Feature Tracking (Abstract Only). In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 278.

[8] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. 2017. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems* 115 (2017), 1–4. https://doi.org/10.1016/j.knosys.2016.10.023

[9] Changdao Du and Yoshiki Yamaguchi. 2020. High-Level Synthesis Design for Stencil Computations on FPGA with High Bandwidth Memory. *Electronics* 9, 8 (2020).

[10] Juan Escobedo and Mingjie Lin. 2018. Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 199–208.

[11] Esmaeil Faramarzi, Dinesh Rajan, and Marc P. Christensen. 2013. Unified Blind Method for Multi-Image Super-Resolution and Single/Multi-Image Blur Deconvolution. *IEEE Transactions on Image Processing* 22, 6 (2013), 2101–2114.

[12] Iman Firmansyah, Yusuf Nur Wijayanto, and Yoshiki Yamaguchi. 2018. 2D Stencil Computation on Cyclone V SoC FPGA using OpenCL. In *2018 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*. 121–124.

[13] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, 81–92.

[14] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2022. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–12.

[15] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*. 311–320.

[16] Kamalavasan Kamalakkannan, Gihan R. Mudalige, István Z. Reguly, and Suhaib A. Fahmy. 2021. High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1087–1096.

[17] Nikolaos Kyparissas and Apostolos Dollas. 2020. Large-Scale Cellular Automata on FPGAs: A New Generic Architecture and a Framework. *ACM Trans. Reconfigurable Technol. Syst.* 14, 1, Article 5 (dec 2020), 32 pages.

[18] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. AN5D: Automated Stencil Framework for High-Degree Temporal Blocking on GPUs. 199–211.

[19] Giuseppe Natale, Giulio Stramondo, Pietro Bressana, Riccardo Cattaneo, Donatella Sciuto, and Marco D. Santambrogio. 2016. A polyhedral model-based framework for dataflow implementation on FPGA devices of Iterative Stencil Loops. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[20] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[21] Enrico Reggiani, Emanuele Del Sozzo, Davide Conficconi, Giuseppe Natale, Carlo Moroni, and Marco D. Santambrogio. 2021. Enhancing the Scalability of Multi-FPGA Stencil Computations via Highly Optimized HDL Components. *ACM Trans. Reconfigurable Technol. Syst.* 14, 3, Article 15 (aug 2021), 33 pages.

[22] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2020. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 9–17.

[23] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2019. Multi-FPGA Accelerator Architecture for Stencil Computation Exploiting Spacial and Temporal Scalability. *IEEE Access* 7 (2019), 53188–53201.

[24] Hengjie Wang and Aparna Chandramowlishwaran. 2020. Pencil: A Pipelined Algorithm for Distributed Stencils. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[25] Shuo Wang and Yun Liang. 2017. A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[26] Stephen Wolfram. 2018. *Computation Theory of Cellular Automata.* 159–202.

[27] Xilinx. 2020. Alveo U280 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf Last accessed July 28, 2020.

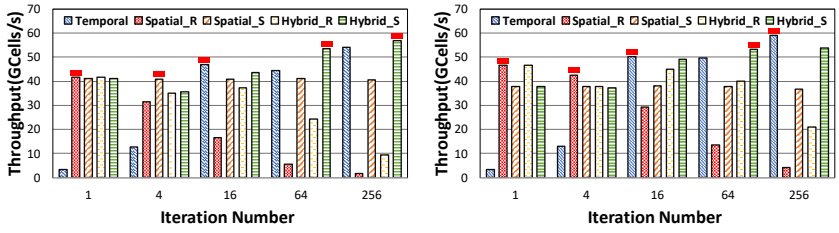
[28] Xilinx. 2020. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html#development Last accessed Nov 26, 2021.

[29] 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 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 153–162.

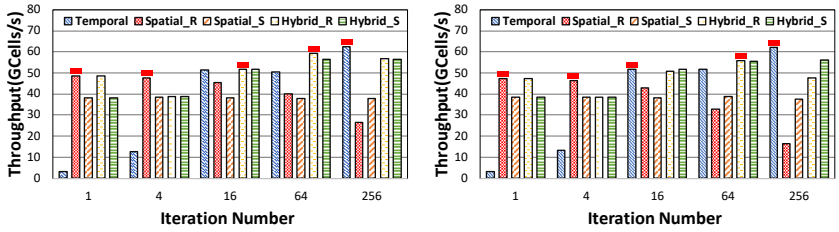
[30] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. High-Performance High-Order Stencil Computation on FPGAs Using OpenCL. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 123–130.

A APPENDIX

Figure 13 to 20 present the throughput of different parallelism optimizations for each benchmark with different input sizes at 256 × 256, 720 × 1024, 9720 × 1024, and 4096 × 4096.

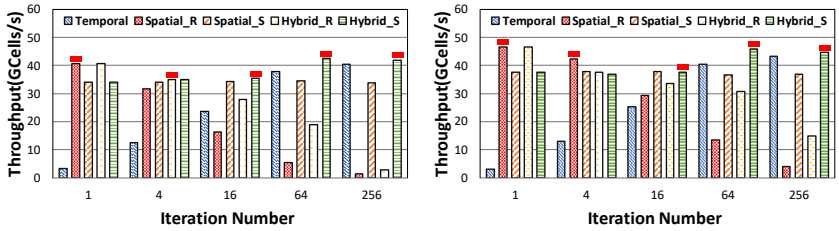


(a) Throughput of JACOBI2D 256 × 256 (b) Throughput of JACOBI2D 720 × 1024

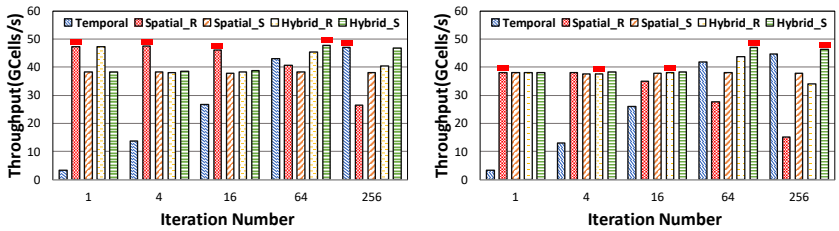


(c) Throughput of JACOBI2D 9720 × 1024 (d) Throughput of JACOBI2D 4096 × 4096

Fig. 13. Throughput (GCell/s) comparison of different parallelism optimizations for JACOBI2D with the number of iterations changing from 1 to 256

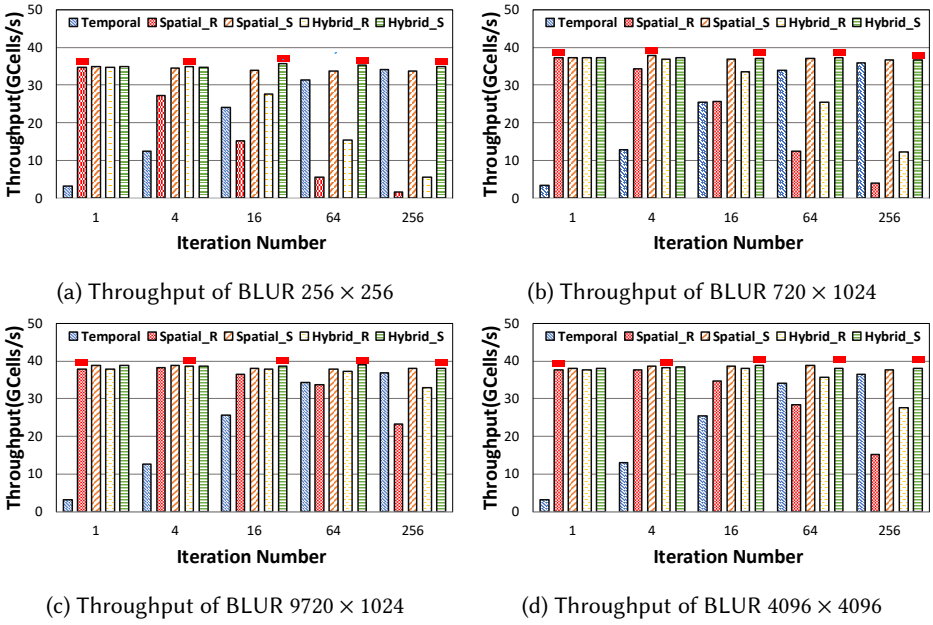


(a) Throughput of JACOBI3D 256 × 16 × 16 (b) Throughput of JACOBI3D 720 × 32 × 32



(c) Throughput of JACOBI3D 9720 × 32 × 32 (d) Throughput of JACOBI3D 4096 × 64 × 64

Fig. 14. Throughput (GCell/s) comparison of different parallelism optimizations for JACOBI3D with the number of iterations changing from 1 to 256



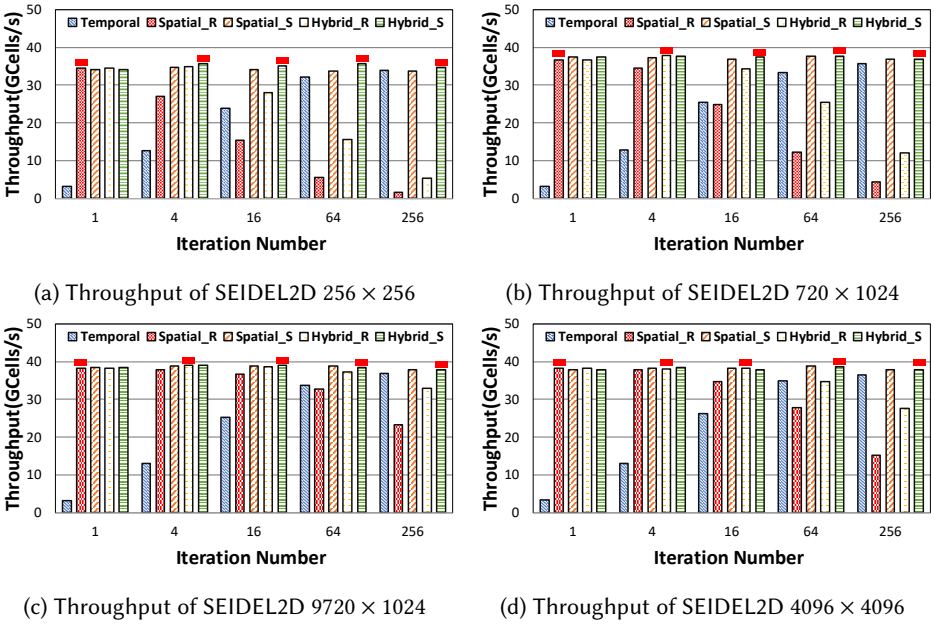
(a) Throughput of BLUR 256 × 256

(b) Throughput of BLUR 720 × 1024

(c) Throughput of BLUR 9720 × 1024

(d) Throughput of BLUR 4096 × 4096

Fig. 15. Throughput (GCell/s) comparison of different parallelism optimizations for BLUR with the number of iterations changing from 1 to 256



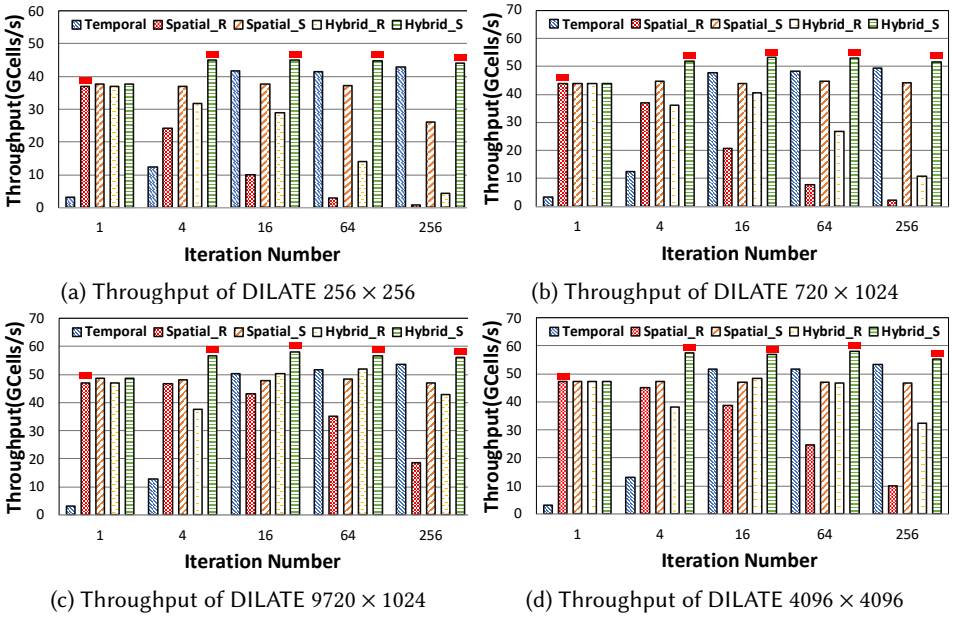
(a) Throughput of SEIDEL2D 256 × 256

(b) Throughput of SEIDEL2D 720 × 1024

(c) Throughput of SEIDEL2D 9720 × 1024

(d) Throughput of SEIDEL2D 4096 × 4096

Fig. 16. Throughput (GCell/s) comparison of different parallelism optimizations for SEIDEL2D with the number of iterations changing from 1 to 256



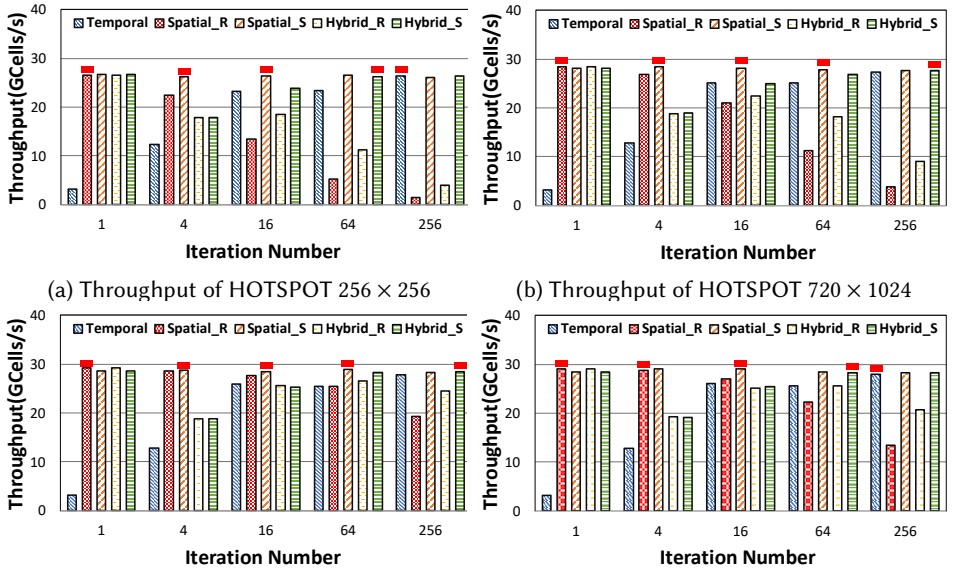
(a) Throughput of DILATE 256 × 256

(b) Throughput of DILATE 720 × 1024

(c) Throughput of DILATE 9720 × 1024

(d) Throughput of DILATE 4096 × 4096

Fig. 17. Throughput (GCell/s) comparison of different parallelism optimizations for DILATE with the number of iterations changing from 1 to 256



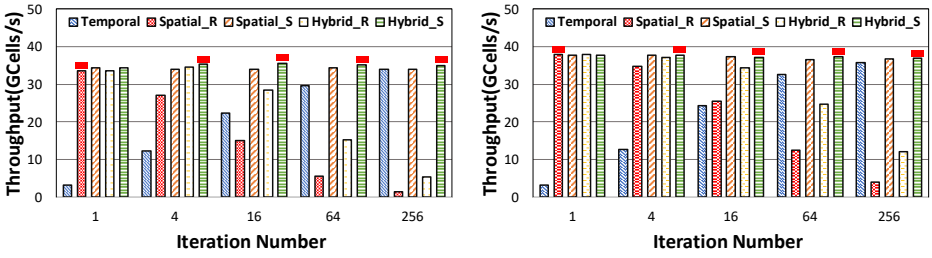
(a) Throughput of HOTSPOT 256 × 256

(b) Throughput of HOTSPOT 720 × 1024

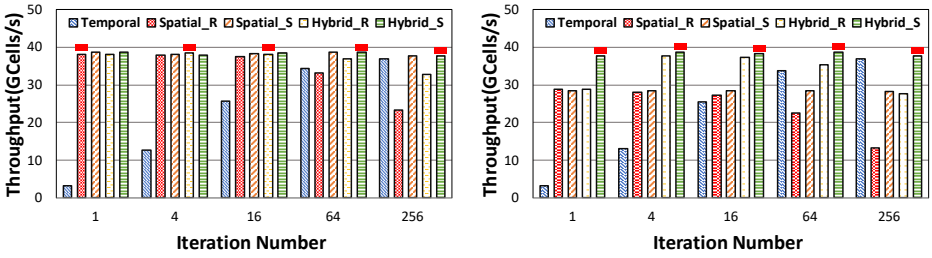
(c) Throughput of HOTSPOT 9720 × 1024

(d) Throughput of HOTSPOT 4096 × 4096

Fig. 18. Throughput (GCell/s) comparison of different parallelism optimizations for HOTSPOT with the number of iterations changing from 1 to 256

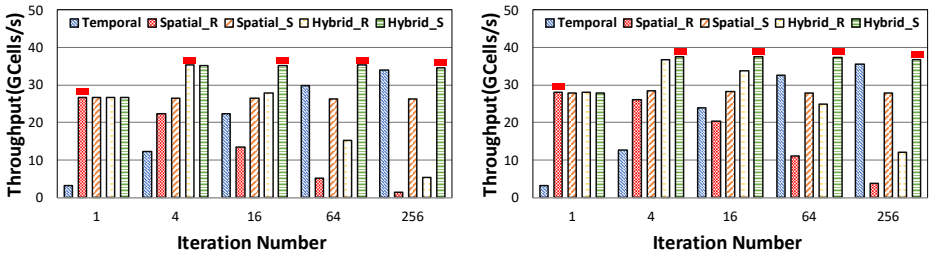


(a) Throughput of HEAT3D $256 \times 16 \times 16$ (b) Throughput of HEAT3D $720 \times 32 \times 32$

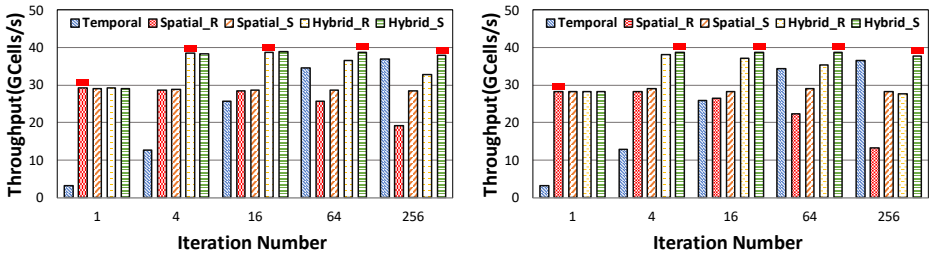


(c) Throughput of HEAT3D $9720 \times 32 \times 32$ (d) Throughput of HEAT3D $4096 \times 64 \times 64$

Fig. 19. Throughput (GCell/s) comparison of different parallelism optimizations for HEAT3D with the number of iterations changing from 1 to 256



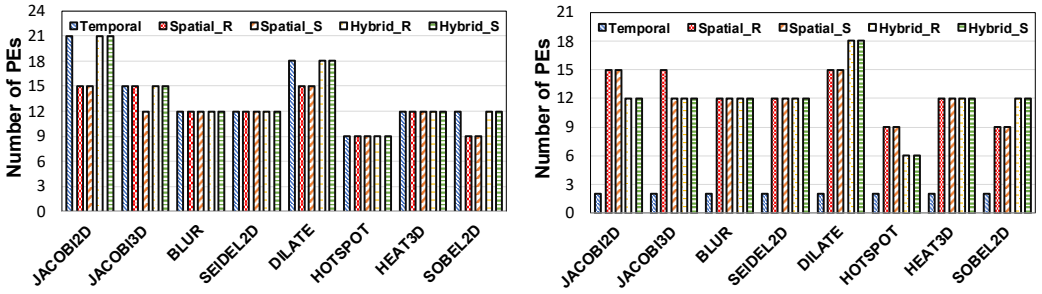
(a) Throughput of SOBEL2D 256×256 (b) Throughput of SOBEL2D 720×1024



(c) Throughput of SOBEL2D 9720×1024 (d) Throughput of SOBEL2D 4096×4096

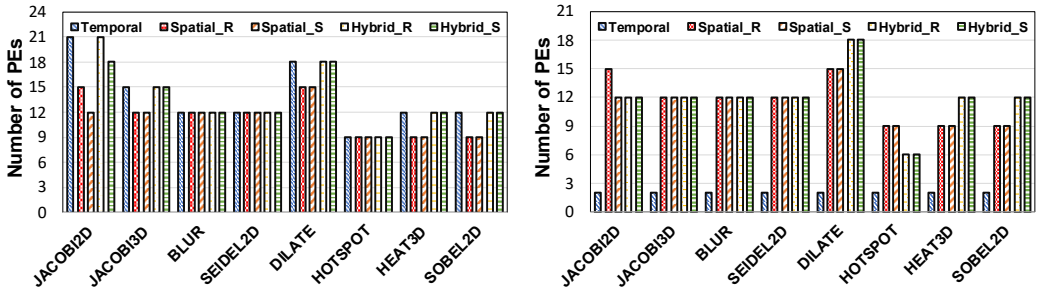
Fig. 20. Throughput (GCell/s) comparison of different parallelism optimizations for SOBEL2D with the number of iterations changing from 1 to 256

Figure 21 and 22 present the total number of PEs for different parallelisms for column size of 256 and 4096, respectively.



(a) Number of PEs with the number of iteration = 64 (b) Number of PEs with the number of iteration = 2

Fig. 21. Total number of PEs for different parallelisms on Alveo U280 with column size = 256



(a) Number of PEs with the number of iteration = 64 (b) Number of PEs with the number of iteration = 2

Fig. 22. Total number of PEs for different parallelisms on Alveo U280 with column size = 4096