TARO: Automatic Optimization for Free-Running Kernels in FPGA High-Level Synthesis

Young-Kyu Choi, *Member, IEEE*, Yuze Chi, Jason Lau, and Jason Cong, *Fellow, IEEE*

***Abstract*—Streaming applications have become one of the key application domains for high-level synthesis (HLS) tools. For a streaming application, there is a potential to simplify the control logic by regulating each task with a stream of input and output data. This is called free-running optimization. But it is difficult to understand when such optimization can be applied without changing the functionality of the original design. Moreover, it takes a large effort to manually apply the optimization across legacy codes. In this article, we present the TARO framework which automatically applies the free-running optimization on HLS-based streaming applications. TARO simplifies the control logic without degrading the clock frequency or the performance. Experiments on Alveo U250 shows that we can obtain an average of 16% LUT and 45% FF reduction for streaming-based systolic array designs.**

***Index Terms*—Field programmable gate arrays (FPGA), free running optimization, high level synthesis (HLS).**

(a)

```
void VecAddTask( int len, stream<int> & a,
             stream<int> & b, stream<int> & c ){
  for(int i = 0; i < len; i++ ){
#pragma HLS pipeline II=1
    c.write(a.read() + b.read());
} }
```

LUT	FF
186	103

(b)

```
void VecAddTask(stream<int> & a,
             stream<int> & b, stream<int> & c ){
#pragma HLS interface ap_ctrl_none port=return
  for(;;){
#pragma HLS pipeline II=1
    c.write(a.read() + b.read());
} }
```

LUT	FF
105	38

Fig. 1. Vector add example: (a) conventional code and (b) free-running optimized.

I. Introduction

High-level synthesis (HLS) [1] tools are becoming very popular in designing field-programmable gate array (FPGA) accelerators. Programmers can write their algorithms in high-level languages, such as C/C++/OpenCL, and the HLS tool automatically decides the low-level architecture. Compared with the traditional register-transfer level (RTL) programming environment, the HLS environment reduces the design effort and shortens the design cycle. Also, the performance of HLS designs has become very competitive with recent advances in HLS optimization techniques [2], [3], [4].

With the increasing demand for utilizing high-level languages for the FPGA design, many efforts have been made to enrich the HLS environment by allowing it to express a wider range of application domains. One such domain includes streaming applications. For example, ST-Accel [5] features high-level abstraction and efficient host-FPGA communication for streaming applications. Spatial [6] is a domain-specific language that abstracts control, memory, and interface (including streams) and boosts the productivity of programmers. Fleet [7] provides a massively parallel streaming model for multiple streams and instances.

For most of the components in streaming applications, we can control their operation with streams of input and output data. For example, in vector add application [Fig. 1(b)], each iteration operates by consuming one token of inputs A and B and producing one token of output C (=A+B). This module can be stopped by either not providing an input or not receiving its output, and we do not need a global control signal to stop its function. This allows us to reduce the related control signals in streaming applications.

But previous versions of two FPGA major vendors' HLS tools, Xilinx Vivado HLS and Intel OpenCL SDK, had limitations in expressing this functionality. Programmers had to specify the exact length of the loops [e.g., Fig. 1(a)] in all modules of the entire datapath—failing to do so would make modules wait forever. A single mistake often leads to deadlock—which is very difficult to debug in the FPGA development environment. Thus, all components in streaming applications required control logic to keep track of the number of input/output data, and they also needed global control signals to indicate that the module had terminated.

This problem was finally addressed by supporting *free-running* kernels [8] in Vitis HLS 2020.1. As the name suggests, a free-running kernel will continuously run, and it cannot be started or stopped by control signals. It performs computation only when all the input data are available and stalls when input FIFOs are empty or output FIFOs are full. Likewise, recent Intel OpenCL SDK now supports *autorun* kernels, which restarts as soon as it finishes execution [9].

An example of a free-running kernel for vector add is shown in Fig. 1. In the conventional Vivado HLS code Fig. 1(a), the length of the vector was provided in a parameter `len`, and the function `VecAddTask` would finish execution when the loop has iterated `len` times. The finite-state machine (FSM) state of `VecAddTask` would proceed to the module's termination state, and the module asserts the `ap_done` [8] signal. In the optimized code Fig. 1(b), on the other hand, `VecAddTask` would still be in the execution state even after the loop iterates `len` times. The module derived from Fig. 1(b) consumes less resources because we can remove the logic that detects whether a module has ended (`ap_done` signal is not needed). The simpler loop control structure is also the main cause for the resource reduction—loop initialization, test, and update are not performed. Moreover, the loop pipeline epilogue is removed.

Although the recent version of Vitis HLS and Intel OpenCL SDK now support the free-running optimization, it is difficult to benefit from this opportunity. HLS developers still use previously developed

Manuscript received 14 July 2022; revised 21 September 2022; accepted 14 October 2022. Date of publication 21 October 2022; date of current version 20 June 2023. This work was supported in part by the Inha University Research Grant; in part by the National Research Foundation (NRF) Grant funded by the Korea Ministry of Science and ICT (MSIT) under Grant 2022R1F1A1074521; and in part by the CRISP Program. Jason Cong has a financial interest in Xilinx/AMD. This article was recommended by Associate Editor H. Zheng. *(Corresponding author: Young-Kyu Choi.)*

Young-Kyu Choi is with the Department of Computer Engineering, Inha University, Incheon 22212, South Korea (e-mail: ykc@inha.ac.kr).

Yuze Chi, Jason Lau, and Jason Cong are with the Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90095 USA.

Digital Object Identifier 10.1109/TCAD.2022.3216544

```
void VecAddTask( int len, tapa::istream<int> & a,
             tapa::istream<int> & b, tapa::ostream<int> & c ){
  [[tapa::pipeline(1)]]      // A fully pipelined loop in TAPA
  for(int i = 0; i < len; i++){
    c.write(a.read() + b.read());
} }
```

Fig. 2. Vector add example written with TAPA attributes.

legacy codes which do not have such optimization, and it would take a large effort to manually apply the optimization across all legacy codes. Also, novice HLS programmers are likely to use the conventional loop coding style in Fig. 1(a), rather than the unfamiliar coding style in Fig. 1(b). Both cases call for automated source modification to incorporate the free-running optimization. But the challenge lies in understanding when such optimization can be applied without changing the functionality of the original design.

This article investigates the condition when the free-running optimization can be safely applied on a streaming application (Section III). Based on the condition learned from the theoretical analysis, our tool, TARO[1], examines a given legacy code and automatically applies the free-running optimization (Section IV). The output code from TARO is then fed into the existing TAPA [10] and Vitis [11] frameworks, and we can generate an optimized FPGA accelerator. To our knowledge, this is the first paper that analyzes the condition for free-running optimization and provides a framework to automatically apply the source transformation.

II. Background: TAPA

TAPA [10] is an open source, customization-friendly extension to the HLS C++ language. It quickly generates large-scale task-parallel dataflow programs through modular, hierarchical approach. Although it is possible to implement TARO for the regular Vitis flow, we found it more convenient to implement TARO on top of TAPA due to a number of benefits—TAPA provides expressive programming interfaces, high quality of results, and fast simulation for streaming applications [10].

TAPA programs are dataflow programs that are composed of hierarchical *tasks*. There is a top-level task that defines the interface between the host and the kernel and instantiates children tasks. Each child task may itself instantiate children tasks or it can do computation. TAPA tasks can communicate with other tasks via streaming interfaces—`tapa::ostream` for producers and `tapa::istream` for consumer. TAPA memory-mapped interfaces (named `tapa::mmap`) provide direct access to the FPGA's external memory (DRAM), which may be read and/or written. A pipelined loop is annotated with `tapa::pipeline` attribute. The vector add example task in Fig. 1 can be rewritten as Fig. 2 in TAPA.

III. Free-Running Optimization

Consider the code structure in Fig. 3(a). It is a conventional for loop structure with the loop initialization statement `Init`, the loop termination condition `Cond`, the loop update `Update`, and the loop body `Body`. The variables modified in `Init` and `Update` will be referred to as *loop index variables*. The statements before the loop will be called `PreLoop`.

We cannot directly apply free-running optimization on this conventional loop structure because the code in Fig. 3(a) cannot process the previous iterations if input data does not exist for the current iteration—for example, if the *i*th iteration of a loop (with a pipeline depth of three) stalls due to the lack of input data, the output of

```
Task(){
  PreLoop
  [[tapa::pipeline(..)]]
  for(Init;Cond;Update){
    Body
  }
}
```

(a)

```
Task(){
  PreLoop
  Init
  [[tapa::pipeline(..)]]
  while(Cond){
    if( !ISs.empty() ){
      Body
      Update
} } }
```

(b)

Fig. 3. Code structure before applying free-running optimization. (a) Conventional loop structure. (b) Flushable loop structure.

```
Task(){
  PreLoop
  [[tapa::pipeline(..)]]
  for(;;){
    if( !ISs.empty() ){
      Body
} } }
```

Fig. 4. Code structure after applying free-running optimization.

the $(i-1)$th and $(i-2)$th iteration will not be written to the output streams (more details in [12] and [13]). This may cause the circuit to fall into a deadlock.

To avoid the deadlock situation, we need to be able to process the existing data in the loop pipeline without a new input data (called *flushable* pipeline). We could use the flushable pipeline provided by Vitis HLS [8], but the tool currently has some limitations in making infinite loops (`for(;;)`) flushable. Instead, we use a workaround of applying a simple source-to-source (S2S) transformation [13] on all pipelined loops in the design. This is illustrated in Fig. 3(b). Compared to Fig. 3(a), the loop `Body` and `Update` are inside the body of *if(!ISs.empty())* clause.[2] This means that `Body` and `Update` for the current iteration are evaluated only if all the input streams in the loop `Body` have input data. If input data does not exist, the loop will try to process the current iteration in the next cycle. The loop will keep processing the existing data in the pipeline convert it into an output without stalling.

The loop `Body` is composed of the following instructions:

$$\begin{aligned} \text{Inst} ::= \text{var} &:= \text{var}' && \text{(variable assignment)} \\ |\text{var} &:= op\left(\overrightarrow{\text{var}'}\right) && \text{(arithmetic operation)} \\ |\text{var} &:= \text{IS.read}() && \text{(blocking stream read)} \\ |&\text{OS.write}\left(\text{var}'\right) && \text{(blocking stream write)}. \end{aligned} \tag{1}$$

Variables var and var′ include registers and local memory, but not external memory, because Vitis HLS does not allow free-running optimization on tasks with external memory access [11]. $\overrightarrow{\text{var}'}$ notates a vector of variables. The input streams will be referred to as *ISs*, and the output streams will be referred to as *OSs*. We expect all stream accesses to be blocking [read() and write()], and we do not consider instructions such as `break` that may terminate the loop (the reason will be explained after Lemma 1). Arithmetic operation *op* includes all operations, such as addition, multiplication, and shift.

Fig. 4 is the code structure after applying the free-running optimization. Compared to the conventional code in Fig. 3(a), the

[1]stands for **T**APA **A**uto**R**un **O**ptimizer

[2]The stream API *empty()* returns `false` if a data *exists* in an input stream (but does not *consume* the data). ISs represents all the input streams referenced in the `Body`. The limitation of this approach is that there can be only one unconditional blocking access to each IS in `Body` (checked by TARO).

```
void VecAddTask( int len, tapa::istream<int> & a,
                 tapa::istream<int> & b, tapa::ostream<int> & c ){
  [[tapa::pipeline(1)]]
  for(;;){
    if( !a.empty() && !b.empty() ){
      c.write(a.read() + b.read());
} } }
```

Fig. 5. Vector add kernel structure after applying free-running optimization.

optimized version will loop continuously with the `for(;;)` loop. It does not have `Init`, `Cond`, and `Update`. For instance, the vector add example in Fig. 2 will be modified to Fig. 5.

Our goal is to show that after applying the free-running optimization on multiple tasks that contain the conventional loop structure, the optimized design will write the same output data to the external memory as the original design. Due to the space limitation, we will only briefly outline the reasoning.

We make the following assumptions.

(A1): The conventional code does not deadlock.

(A2): All tasks have flushable pipeline loops.

(A3): `Body` has at least one access to an IS.

(A4): Loop index variables are not referenced in `Body`.

The programmer should ensure (A1) and provide a working design that does not deadlock. TARO automatically transforms all tasks' loop pipelines to the flushable structure in Fig. 3(b) (A2). TARO also checks if assumptions (A3) and (A4) are met in the target loops. The implementation details will be explained in Section IV.

Let us suppose that the loop in the conventional code iterates N times. Then we claim the following lemma.

Lemma 1: If the ISs in the transformed code receive the same stream of data (ISs.read()) as the conventional code, the OS data [var in OSs.write(var)] in `Body` of the transformed code matches that of the conventional code from first to Nth iteration.

At the beginning of the first iteration of `Body`, the value of all variables *var* in the transformed code, except the loop index variables assigned in `Init`, matches that of the conventional code. In the first iteration of `Body`, the same operation *op* is performed and the loop index variables are not referenced (A4). This means that if the ISs in the transformed code receive the same stream of data as the conventional code, all variables assigned after performing arithmetic operations ($\text{var} \coloneqq \text{op}(\overrightarrow{\text{var}'})$) in the first iteration of `Body` in the transformed code match that of the conventional code. These newly assigned variables can be either written to OSs or referenced in the subsequent iterations. The OS writes will be performed for the first iteration even if input data is not provided for the succeeding iterations because the loop is flushable (A2). The design does not deadlock due to assumption (A1). Thus, the OS data [var in OSs.write(var)] is the same on the first iteration. By a similar reasoning, the same output stream will be written from the second to the Nth iteration. The loop does not terminate before the Nth iteration because `break` is not allowed in (1).

Lemma 2: If the ISs in the transformed code receive the same stream of data as the conventional code, the OSs in the `Body` of the transformed code does not write data after the Nth iteration.

After N iterations, the transformed code will try to evaluate the $(N + 1)$th iteration of `Body`. However, since the ISs receive the same stream of data, the input data for the $(N + 1)$th iteration does not exist. The `Body` of the transformed code has at least one blocking IS access (A3), and the lack of input data will block the loop pipeline (due to the *if(!ISs.empty())* clause). Thus, all `Body` instructions for the $(N + 1)$th iteration will not be evaluated, and the transformed code does not write data into OSs after the Nth iteration.

```
tapa::task()
  .invoke(ReadATask, ...)
  .invoke(ReadBTask, ...)
  .invoke<tapa::detach>(VecAddTask, ...)
  .invoke(WriteCTask, ...);
```

Fig. 6. Detaching the free-running task for vector add.

Theorem 1: If the ISs in the transformed code receive the same stream of data as the conventional code, the OS data in the transformed code matches that of the conventional code.

If `Body`'s OS data in the transformed code does *not* match that of the conventional code, it means that either the output until the Nth iteration is different or there exists output after the Nth iteration. This contradicts Lemma 1 or Lemma 2. Also, there is no difference in `PreLoop` because no code modification has been made in that part.

Theorem 2: The data written to the external memory in the transformed code matches that of the data written in the conventional code.

Starting from the tasks that have only output streams and external memory access (will be referred to as *source tasks*), we assign a level that corresponds to the longest distance from the source tasks. The output stream data from the source tasks do not change because the tasks with external memory access are not free-running optimized (1). Since the level 1 tasks' ISs receive the same stream of data from the source tasks, the level 1 tasks' OS data matches the conventional code by Theorem 1. By the same reasoning, the output streams from levels 2, 3, ..., tasks also match the original design. Thus, a DRAM-accessing task, which receives data from some of these tasks, will write the same data to the external memory.

IV. Implementation

Given a TAPA-compatible C++ input source code, TARO utilizes the APIs in Clang/LLVM 8 [14] to traverse through all tasks and applies *if(!ISs.empty())* on all pipelined loops to make them flushable (A2). Next, TARO finds loops that can be free-running optimized. If a target loop satisfies the assumptions listed in Section III and its `Body` is composed of instructions listed in (1), TARO automatically applies the free-running optimization (Fig. 4) and produces an optimized TAPA code as the output. We use the existing TAPA compilation flow to generate an FPGA bitstream from the output code.

Pipelined loops are found by checking if a loop has an attribute of `tapa::pipeline`. Tasks with external memory access are excluded from the optimization if TARO finds arguments with type `tapa::mmap`. TARO looks for all stream references in the loop `Body` and checks if Assumption (A3) holds. Assumption (A4) is checked by making a list of loop index variables from the loop, and searching for all references of those variables in the loop `Body`.

Next, we modify TAPA so that we can assign a template argument `<tapa::detach>` for the free-running optimized tasks (Fig. 6). This argument is sent to TAPA's global task management scheme. TAPA maintains a global FSM which changes from `idle` to `running` when the user starts the kernel. Without the free-running optimization, the global state becomes `idle` only when all the tasks finish and enter the `idle` state. We modify this scheme so that the `detach`'ed tasks are excluded from the global `idle` state. Even if the free-running optimized tasks never become `idle` state, the global FSM now can become `idle`, and the kernel will terminate properly.

V. Experimental Result

We feed TARO-optimized C++ code into the TAPA framework [10], which utilize Xilinx's Vitis 2020.2 suite [11] to synthesize

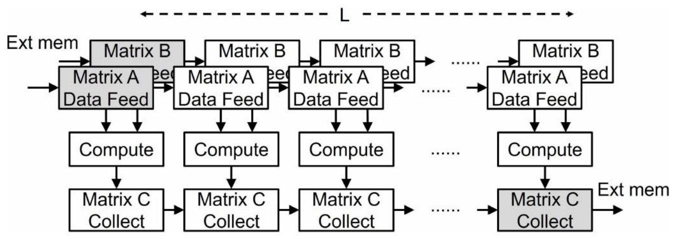

Fig. 7. Systolic array architecture for MM.

TABLE I
RESOURCE/FREQUENCY/CYCLE REDUCTION AFTER APPLYING FREE-RUNNING OPTIMIZATION

App.		LUT	FF	Freq	Cycles
Mat-vec Mult. (MV)	Base Opt	18,064 14,957 (17.2%)	41,221 24,100 (41.5%)	300 300 (0%)	10,585 10,585 (0%)
Mat-mat mult. (MM)	Base Opt	14,785 10,829 (26.8%)	45,277 21,804 (51.8%)	300 300 (0%)	13,114 13,114 (0%)
Needleman -Wunsch (NW)	Base Opt	7,957 6,922 (13.0%)	16,197 9,490 (41.4%)	300 300 (0%)	2,212 2,212 (0%)
Convol. Neu. Net. (CNN)	Base Opt	23,345 21,789 (6.7%)	64,104 33,981 (46.7%)	300 300 (0%)	32,243 32,227 (0%)
Average		(15.9%)	(45.4%)	(0%)	(0%)

the design. Autobridge [15] improves the design's timing. We target Xilinx's Alveo U250 board [16] with the clock frequency of 300 MHz.

Initially, we applied the free-running optimization on the vector add application. But we found that the overall LUT/FF reduction is only about 2%/4%. Even though there was a significant reduction of resource for the compute task in Fig. 1, the proportion of this task in the overall design is very small because we need three DRAM access tasks (for `a`, `b`, and `c`). Thus, the overall LUT and FF reduction ratio is insignificant. Instead, we have applied the proposed optimization on benchmarks with a high proportion of compute and data distribution tasks compared to external memory access tasks—specifically, the designs in systolic array architecture [17] with only few DRAM access points. It includes dense matrix-vector (MV) multiplication, matrix–matrix (MM) multiplication, Needleman–Wunsch (NW), and convolutional neural network (CNN). All benchmarks follow the architecture proposed in PolySA ([17, Fig. 5])—an example for MM is displayed in Fig. 7. The ratio of tasks with DRAM access for these applications are 4.7%–6.9%—we have adjusted the length `L` of the arrays to approximately match the ratio. The data is in short type, and all innermost loops are pipelined. The source transformation for all benchmarks was performed in less than 1 s.

Table I presents the LUT/FF reduction after applying the free-running optimization. We report the Alveo U250 post-PnR resource consumption of the kernel logic and exclude that of the static region. DSP and BRAM consumption is omitted since the difference is small. Note that the baseline implementation refers to the design with the flushable loop structure in Fig. 3(b).

The table shows that, on average, we can obtain 15.9% and 45.4% reduction on LUT and FF consumption, respectively. This is due to the simplification of the loop control and the global task management logic. There is no change in the clock frequency in all benchmarks. The clock cycle for CNN has been slightly changed with the simplification of the loop FSM state, but the difference is minimal.

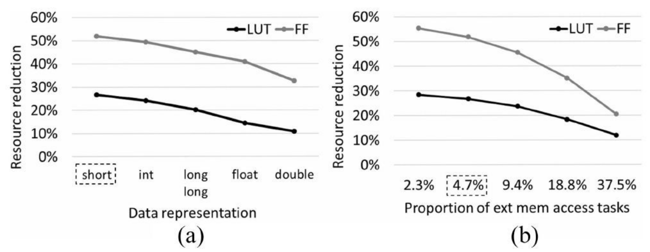

Fig. 8. Resource reduction for MM after varying (a) data representation and (b) proportion of external memory accessing tasks. (The result in Table I was obtained using short type and 4.7% external memory access tasks.)

The amount of resource reduction, however, widely varies on the design parameters. The resource consumption for MM after changing the data representation can be observed in Fig. 8(a). As we utilize more complex data representation, the compute logic becomes larger than the control logic. This reduces the LUT saving from 27% (short type) to 11% (double type). Also, since the free-running optimization cannot be applied on tasks with external memory access, a design with many DRAM-access tasks benefits less from the free-running optimization—if `L` was adjusted so that the ratio of DRAM-access tasks is increased from 4.7% to 37.5%, the LUT saving reduces from 27% to 12% [Fig. 8(b)]. From these observations, we can conclude that free-running optimization is only effective for architectures with: 1) high proportion of control logic compared to compute logic and 2) high proportion of nonexternal memory access tasks.

VI. RELATED WORKS

There are many works that automatically optimizes HLS designs by predicting the performance and inserting the best set of HLS directives on the target designs. COMBA [18] is a comprehensive HLS optimization framework that models the effect of HLS directives and performs metric-guided design space exploration. The work in [19] proposes a metaheuristic approach of combining simulated annealing, genetic algorithm, and ant colony optimization with efficient hyper-parameters. The work in [20] learns the knowledge from previous designs to infer the effect of applying HLS directives on new designs. These works typically optimize HLS designs by finding the best tradeoff point between resource consumption and performance. TARO, on the other hand, reduces the resource consumption of the HLS control logic without degrading the performance of the baseline design.

There are only a few works that optimize the control logic in HLS-generated circuits. Clockwork [21] fuses the FIFO control logic and the buffer of two stages in an image processing pipeline. The work in [4] improves the synchronization and pipelining of HLS broadcast control signals. None of these papers optimizes the loop control and global task management logic as TARO, and their approaches are orthogonal to the proposed free-running optimization.

VII. CONCLUSION

We have proposed the TARO framework which automatically applies the free-running optimization on HLS-based streaming applications. The free-running optimization reduces the LUT and FF needed for the loop control and global task management logic. When a design is composed of flushable loops, the optimization can be applied to a pipelined loop with blocking stream accesses and no loop index variables or breaks in its loop body. A same sequence of data will be read and written in the stream and external memory access after making the source transformation. We have found that the transformation is effective only when the proportion of control logic is high compared to the computation logic and the proportion of the stream access tasks is high compared to the DRAM access tasks. TARO has been implemented on top of TAPA, and it has been open sourced at https://github.com/UCLA-VAST/tapa/tree/taro. We expect that TARO will help reduce the control overhead of streaming applications with minimal code changes from the HLS programmers.

References

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for FPGAs: From prototyping to deployment,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[2] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, “Combining dynamic & static scheduling in high-level synthesis,” in *Proc. FPGA*, 2020, pp. 288–298.

[3] Y. T. Chen, J. H. Kim, K. Li, G. Hoyes, and J. H. Anderson, “High-level synthesis techniques to generate deeply pipelined circuits for FPGAs with registered routing,” in *Proc. FPT*, 2019, pp. 375–378.

[4] L. Guo et al., “Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency,” in *Proc. DAC*, 2020, pp. 1–6.

[5] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong, “ST-Accel: A high-level programming platform for streaming applications on FPGA,” in *Proc. FCCM*, 2018, pp. 9–16.

[6] D. Koeplinger et al., “Spatial: A language and compiler for application accelerators,” in *Proc. PLDI*, 2018, pp. 296–311.

[7] J. Thomas, P. Hanrahan, and M. Zaharia, “Fleet: A framework for massively parallel streaming on FPGAs,” in *Proc. ASPLOS*, 2020, pp. 639–651.

[8] “Vitis high-level synthesis 2020.2 user guide (UG1399).” Xilinx. 2021. [Online]. Available: https://www.xilinx.com/

[9] “Intel FPGA SDK for OpenCL pro edition: Programming guide,” Intel, Santa Clara, CA, USA, 2020.

[10] Y. Chi, L. Guo, J. Lau, Y.-K. Choi, J. Wang, and J. Cong, “Extending high-level synthesis for task-parallel programs,” in *Proc. FCCM*, 2021, pp. 204–213.

[11] “Vitis unified software development platform 2020.2 documentation (UG1416).” Xilinx. 2021. [Online]. Available: https://www.xilinx.com/

[12] S. Dai, M. Tan, K. Hao, and Z. Zhang, “Flushing-enabled loop pipelining for high-level synthesis,” in *Proc. DAC*, 2014, pp. 1–6.

[13] Y.-K. Choi, Y. Chi, J. Wang, and J. Cong, “FLASH: Fast, ParalleL, and accurate simulator for HLS,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4828–4841, Dec. 2020.

[14] “Clang: A C language family frontend for LLVM.” Clang. 2022. [Online]. Available: https://clang.llvm.org

[15] L. Guo et al., “AutoBridge: Coupling coarse-grained Floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs,” in *Proc. FPGA*, 2021, pp. 81–92.

[16] “Alveo data Center accelerator card platforms (UG1120).” Xilinx. 2021. [Online]. Available: https://www.xilinx.com/

[17] J. Cong and J. Wang, “PolySA: Polyhedral-based systolic array auto-compilation,” in *Proc. ICCAD*, 2018, pp. 1–8.

[18] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, “COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications,” in *Proc. ICCAD*, 2017, pp. 430–437.

[19] Z. Wang and B. C. Schafer, “Machine learning to set meta-heuristic specific parameters for high-level synthesis design space exploration,” in *Proc. DAC*, 2020, pp. 1–6.

[20] L. Ferretti, J. Kwon, G. Ansaloni, G. Di Guglielmo, L. P. Carloni, and L. Pozzi, “Leveraging prior knowledge for effective design-space exploration in high-level synthesis,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3736–3747, Nov. 2020.

[21] D. Huff, S. Dai, and P. Hanrahan, “Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs,” in *Proc. FCCM*, 2021, pp. 186–194.