Array Partitioning Method for Streaming Dataflow Optimization in High-level Synthesis

Renjing Hou[1], Jianwang Zhai[1], Yajun Wang[2], Zhe Lin[3], Kang Zhao[1†],
[1]School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China
[2]School of Electronic and Information Engineering, Tiangong University, Tianjin, China
[3]School of Integrated Circuits, Sun Yat-sen University, Guangdong, China
{hourj, zhaijw, zhaokang}@bupt.edu.cn, 2231070984@tiangong.edu.cn, linzh235@mail.sysu.edu.cn

2024 2nd International Symposium of Electronics Design Automation (ISEDA) | 979-8-3503-5203-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISEDA62518.2024.10618042

***Abstract*—High-level synthesis (HLS) is a popular method that allows designers to describe the behavior-level functionality and automatically generates efficient register-transfer level (RTL) descriptions. In HLS, dataflow is the key micro-architecture to achieve high parallelism. However, strict conditions such as sequential access on the potential channels often limit the streaming dataflow. To settle this issue, this paper proposes an efficient array partitioning method for the streaming dataflow inference. The key is to explore the potential array partitioning mode that matches the sequential access requirements by streaming channels. An experimental case study is presented on the inference of the convolutional neural networks (CNN). It indicates that the proposed method can achieve about 28.6% performance improvements compared with the default dataflow, with the cost of 7.2% power increasement.**

***Index Terms*—High-level Synthesis, Streaming Dataflow, Array Partitioning, FPGA**

I. Introduction

High-performance computing and data-intensive applications, such as machine learning, are becoming increasingly common. For these designs, the field-programmable gate array (FPGA) is a good choice to bring the benefits of hardware acceleration. However, the traditional flow starting from the register-transfer level (RTL) is challenging for these large and complex designs, because performing RTL development is very time-consuming and error-prone [1]. In comparison, high-level synthesis (HLS) can generate RTL descriptions (e.g., Verilog/VHDL) from behavior-level languages (e.g., C/C++/OpenCL). HLS can reduce the development time, improve the ability of fast testing and simulation, and narrow the huge gap between software engineers and FPGA acceleration. Until now, there are many commercial HLS tools [2]–[4], and Xilinx Vitis HLS [5] is a well-known tool in the FPGA domain.

The main challenge for HLS is how to generate expected good-quality RTL designs. Therefore, HLS tries to convert the serial instruction lists into equivalent logical circuits and achieve good acceleration through parallelism. To achieve this target, commercial HLS tools often support many synthesizable user-control knobs, including pragmas/directives and global options [6]. These pragmas include pipeline, loop unrolling, dataflow, array partitioning, etc. Users can control them to generate different expected microarchitectures based on the same input design. This paper will focus on dataflow and array partitioning optimizations.

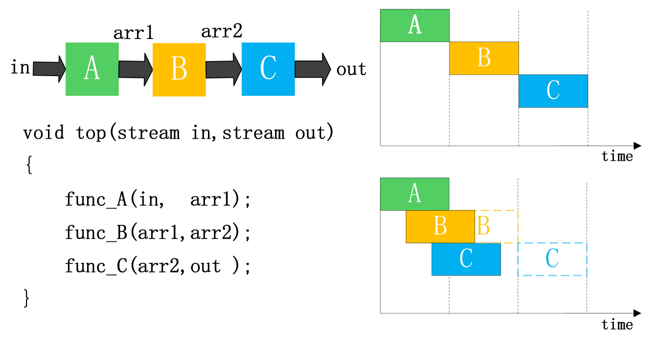
Dataflow is one of the important optimization methods used to achieve task-level parallelism. It has two micro-architectures based on different buffer implementation strategies, i.e., the ping-pong dataflow and the streaming dataflow. Streaming dataflow has both higher parallelism and lower architecture complexity than ping-pong dataflow. Generally, it could bring more performance improvement. However, it has very strict conditions on HLS coding styles. For example, the channels for data transfer require first-in and first-out (FIFO) accessing. Therefore, how to adjust the array accessing orders and try to match the strict FIFO requirements for the dataflow inference is a critical challenge.

Until now, HLS research has a very long history [7]–[10]. And researchers have proposed many methods for array-partition and dataflow optimizations, especially. However, no one has ever proposed solutions to settle this issue. For dataflow optimizations, the academic mainly focuses on the design space exploration (DSE) algorithms. Its key is how to select a proper combination of knobs so as to obtain better performance, including the dataflow knob. For example, Wei Zhang [11] and Xuehai Zhou [12] proposed different methods to identify C-level dataflow structures for DSE, but they did not pay attention to the inference of streaming channels. For the array-partitioning optimization, current researches mainly focus on standalone scenarios. For example, Yuxin Wang [13] proposed block-cyclic optimizations for on-chip memories for high-data throughput; to accelerate the stencil-based kernel, Juan [14] proposed a graph-theoretically optimal memory banking algorithm. In conclusion, unfortunately, there is still no research focusing on the partitioning problem for the streaming dataflow channels.

To address this issue, this paper will propose an efficient array partitioning method. To prove its feasibility, an experimental case study on convolutional neural networks (CNNs) is presented. The motivation for selecting CNN is due to its wide usage in image recognition domains. Its underlying implementation techniques, possess a high

[†] Corresponding Author

This work is partially supported by National Key R&D Program of China (2022YFB2901100) and Beijing Natural Science Foundation under Grant 4244107.



```
void top(stream in, stream out)
{
    func_A(in,  arr1);
    func_B(arr1, arr2);
    func_C(arr2, out );
}
```

Fig. 1 Dataflow can achieve task-level parallelism.

degree of computational parallelism and are well-suited for achieving acceleration through dataflow. However, due to the discontinuous nature of array read and write operations, it frequently results in the premature abandonment of dataflow optimization efforts. Our work shows that by harnessing array partitioning optimization, it becomes feasible to modify the actual storage arrangement of arrays, ensuring compliance with the prerequisites for dataflow applicability and subsequently enhancing design performance.

The rest of the paper is organized as follows. Section II gives the preliminaries and problem formulation. Section III proposes the array partitioning method and describes its strategy details. Section IV shows the experimental results of the case study. Section V draws the conclusion.

II. Preliminaries & Problem Formulation

A. Streaming Dataflow and Constraints

Dataflow is an optimization micro-architecture used to achieve task-level parallelism. As shown in Fig. 1, in the code of the process C, the data is transferred between neighboring processes sequentially, i.e. the process A should finish before B starts. However, if B could start beforehand without any data dependency, A can transfer data to B as early as possible and drive B based on dataflow. This is the key for the dataflow structure, which can improve the parallelism between coarse-grained tasks.

Obviously, it is clear that the key for the dataflow inference is just the data transferring order between neighboring processes. Generally, if the access order is simple and regular, the data could then be transferred downward faster, and much more performance improvements could be achieved. According to different memory channel implementations, the dataflow can be categorized into two modes: the ping-pong dataflow and the streaming dataflow.

The ping-pong dataflow is implemented as shown in Fig. 2, consisting of two buffers known as ping-pong buffers. Task A acts as the data producer, while task B serves as the consumer. Buffer$_1$ and buffer$_2$ rotate for read and write. For example, A writes data to buffer$_1$, simultaneously, B reads data from buffer$_2$. When buffer$_1$ is full and buffer$_2$ is empty, A starts to write data to buffer$_2$, and B reads data from buffer$_1$. The streaming dataflow consists of a FIFO channel, as shown in Fig. 3. The producer process stores the data in the buffer,

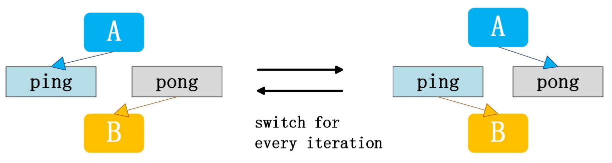


Fig. 2 The illustration of ping-pong buffers.

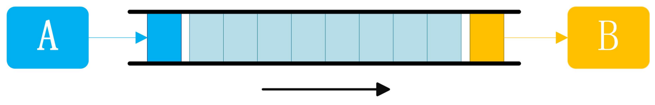


Fig. 3 The illustration of streaming buffers.

and the consumer process retrieves the data sequentially. Compared with the ping-pong dataflow, the streaming buffer only needs to read and write in a continuous space. At the same time, the streaming buffer can be implemented with a capacity that is much smaller than the total data size, while the ping-pong buffer requires twice the entire data space. Thus the streaming buffer is favored for its ease of implementation and higher accessing efficiency.

However, the streaming dataflow has very strict constraints. Since it uses FIFO as the data channel, the producer process can only perform write operations on the candidate channel array, while the consumer process can only perform read. Furthermore, the read and write operations must be consistent and serial in order. This is the motivation of this work, i.e., trying to match the streaming dataflow requirements through array partitioning.

B. Array Partitioning

Array partitioning is a commonly used optimization directive in HLS, trying to explore the possibility of parallel memory access on the array. Fig. 4 illustrates the array partitioning in the HLS tools (Xilinx Vitis HLS [6]), depicting three methods: block, cyclic, and complete modes.

Because array partitioning enables parallelism in array access, it is often used together with other strategies targeting parallel computing. Until now, it is an important part of the HLS design space exploration. Generally, existing works usually try to build up the internal relationship between the array partitioning and final performance. For example, Cilardo et al. [15] analyzed the optimization effects of combining loop unrolling and multidimensional memory partitioning. They mainly consider the impact of memory bank switches brought by array partitioning. Silitonga et al. [16] conducted research combining array partitioning with pipeline in the context of HLS-based optimization for cryptographic modules. Choi et al. [17] explored the design space by combining loop unrolling, pipelining, and array partitioning when dealing with cases of suboptimal HLS automatic optimization.

However, if designers have decided to use streaming dataflow as the coarse-granularity optimization strategy, how to match the strict limitations is the most challenging task. For this special scenario, the array partitioning is usually required to work together with dataflow, targeting transforming the array to stream. This paper will focus on how to analyze and

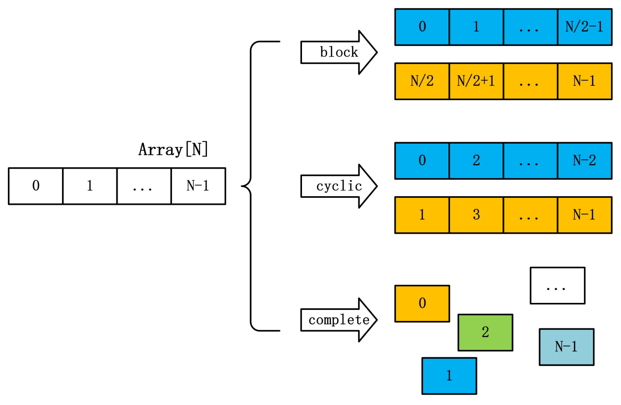


Fig. 4 Three strategies of *array_partition* in HLS

adjust the array access mode so as to match the requirements from dataflow streaming channels. Given the remarkable optimization effects of streaming dataflow, the array partitioning technique is the most promising choice to relax its constraints. To the best of the author's knowledge, there is no prior work targeting such a scenario.

C. Problem Formulation

Due to the strict constraints of the streaming dataflow, if HLS tools cannot ascertain that the candidate channel array is accessed in sequential order, the target array will be implemented with the ping-pong buffer. However, if we can convert non-serial access to continuous access in some way, then we can utilize streaming dataflow to further improve performance. Thus, the problem can be formulated as:

Problem 1 (Array Partitioning for Streaming Dataflow Optimization)**.** *Given an array T with n elements, each element $T[i]$ has its own read/write access $Ac[i]$, where Ac is a permutation from 1 to n. The objective is to partition T into m sub-arrays $\{T_1, T_2, ..., T_m\}$, and guarantee that: for each sub-array T_x ($x \in \{1, ..., m\}$), $Ac[k+1] = Ac[k] + 1$ ($k \in \{1, ..., size(Tx) - 1\}$).*

III. Methodology

To solve the problem above, this section will propose an array partitioning method to match the streaming channel requirements.

A. Overall Flow

To help designers accelerate the software algorithms, we have built up a framework to infer dataflow automatically. This overall flow is shown in Fig. 5, where the green part is the main work of this paper.

The first three steps are used to check if the dataflow pragma can be performed or not. First, focus on the target function region, which only contains multiple sub-function calls. Because Xilinx HLS tool has strict coding style requirements, and cannot extract and generate sub-process easily. Then for each pair of neighboring sub-function calls (process), find the common array parameters as the potential candidate channels. Third, for each candidate channel, judge if it is both write-only in the producer process and read-only in the consumer process. The streaming channel cannot tolerate the scenario of read-write mix access.

After the dataflow region inference, the focus is to judge whether the access order on each candidate channel is sequential. If not, the array will be synthesized to a ping-pong buffer originally, as indicated by the red arrow in Fig. 5. However, this step misses many possible streaming optimizations. For example, the access hops with an order of arithmetic progression. If the accessed index could be recombined under the rule of no data dependency conflict, we could refine the array access order to match the streaming requirements. Therefore, a partition-based method is proposed in this flow which tries to split the target array into multiple small channels to guarantee continuous access. Then these small channels could be synthesized to multiple dataflow FIFOs running in parallel.

If the target array is multi-dimensional, it will be firstly flattened into a one-dimensional array in an iteration order, to ease the mode matching. The key is the access analysis part. It will decide if array partitioning can help or not, which partitioning strategy is the best choice, and how to guarantee both continuous access and no data dependency conflict. After the partitioning plan is decided, the array partitioning will take action next. Finally, the stream depth must be decided and set to avoid dataflow deadlock.

B. Access Analysis & Partitioning

First of all, the access order on the candidate array channel should be consistent, both for the data producer and the consumer processes. The array Ac, accessing the array T, needs to follow the pattern C: (if the array is multi-dimensional, it should be flattened into the one-dimensional array in an iteration order).

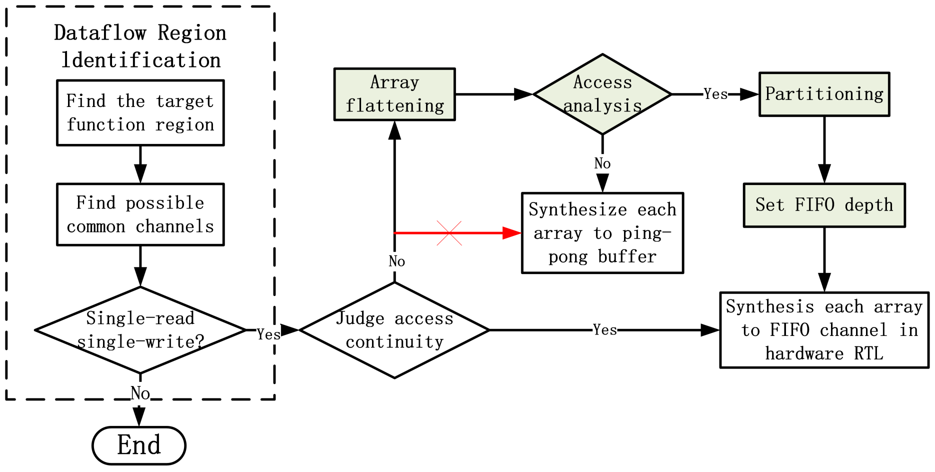
$$\begin{aligned} &C(n,t,m) = B(1,t,m), B(2,t,m), ..., B(n,t,m) \\ &B(i,t,m) = A_1(i,m), A_2(i,m), ..., A_t(i,m), i \in [1,n] \\ &A_j(i,m) = p^i_j, p^i_j + 1, ..., p^i_j + m - 1, j \in [1,t] \\ &p^i_j = (k-1)*n*m + m*(j-1). \end{aligned} \tag{1}$$

The Mode C means that Ac (C) can be cut in n cycles. Each cycle (B) is composed of t serial parts (A_j), and each of them has m blocks. Pick one part (A_j) from each cycle can constitute t blocks, which is shown below:

$$S_j(n,m) = A_j(1,m), A_j(2,m), ..., A_j(n,m), j \in [1,t] \tag{2}$$

The values of S_j are continuous from $(k-1)*n*m$ to $k*n*m-1$, and $k(0 < k \leq t)$ means that the sub-array $S_j(1 \leq j \leq t)$ is the k-th accessing sub-array of the target partition sub-arrays.

To give a more comprehensible explanation, three example cases are shown in Fig. 6. The access analysis and corresponding partitioning choice are also presented (The access order has been marked on the square). In addition, their formula values are displayed in the TABLE I.



Dataflow Region Identification
Find the target function region
Find possible common channels
Single-read single-write?
Yes
No
End
Judge access continuity
No
Yes
Array flattening
Access analysis
Yes
No
Partitioning
Synthesize each array to ping-pong buffer
Set FIFO depth
Synthesis each array to FIFO channel in hardware RTL

Fig. 5 The overall flow of the proposed method.

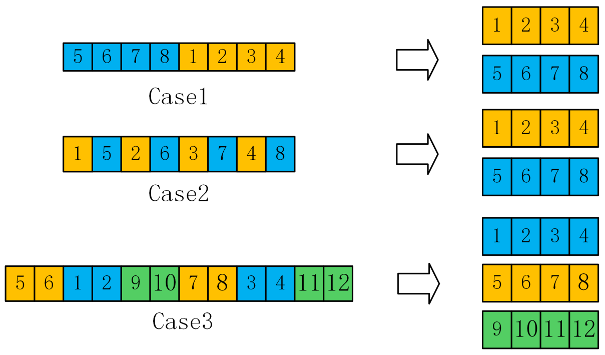


Fig. 6 Three example cases to show the key strategy.

TABLE I Formula values of the cases

Case	C	B	A_j	S_j
Case-1	$(1, 2, 4)$	$(1, 2, 4)$	$(1, 4)$	$(1, 4)$
Case-2	$(4, 2, 1)$	$(i, 2, 1)$	$(i, 1)$	$(4, 1)$
Case-3	$(2, 3, 2)$	$(i, 3, 2)$	$(i, 2)$	$(2, 2)$

Let's take Case-3 as an example. Its value in the mode C is $C(2, 3, 2)$. The corresponding array is 2 cycles. Each cycle could be partitioned into 3 serial blocks, and each serial block has 2 blocks. Corresponding to the formula, in cycle $i(0 < i \leq 2)$, $B(i, 3, 2)$ is composed of $A_1(i, 2)$ and $A_2(i, 2)$. Pick one part($A_j(i, 2), 0 < j \leq 2$) from each cycle can constitute a serial block $S_j(2, 2)$.

The existing *array_partition* pragma can only be performed in block or cyclic modes as shown in Fig. 4. It can only handle the situations when $n = 1$. To achieve the array partitioning for the streaming dataflow, it is necessary to implement the proposed array partitioning method described above.

C. Deadlock and Others

Deadlock is a common error that occurs when using the streaming dataflow. It is usually caused by attempting to read from an empty buffer or write to a full buffer. To avoid dataflow deadlock, it is necessary to ensure that the minimum FIFO size is larger than the data size of a single reading/writing operation. If the deadlock still occurs when the FIFO size is big enough, it is necessary to check whether the code logic meets *single-read-single-write* or not. Within the feasible range, the FIFO size could be much smaller than the total array size, which is good for reducing area.

It should be noted that, since the producer process only writes to the array and the consumer process only reads from it, there are no data dependencies within the candidate array channel. Therefore, the access order can be adjusted to ensure a consistent reading and writing order while complying with the logic of the process.

IV. Experiments

A. Experimental Setup

The experimental environment mainly uses Xilinx Vitis HLS (version 21.2) on a Linux platform with 48 Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz, and a single-layer CNN model [18] is used as the benchmark case. The experiment results are obtained via synthesis and routing with Xilinx Vivado 21.2. The target clock cycle is set as 10ns.

B. Case Study: CNN Acceleration

To demonstrate the effectiveness of the proposed method, CNN is selected for the case study. The convolutional layer and pooling layer are two consecutive parts of CNN. There is an array used to transfer data between them. The convolutional layer only writes on it, while the pooling layer only reads from it. The streaming dataflow can be performed if the accessing order is consistent and continuous. To achieve consistency, the writing order could be modified to be the same as the reading. However, since the pooling layer needs to run on the sub-matrix blocks, the accessing does not meet the continuity requirements. Then, in the next section, our proposed method will be demoed to guarantee serial access.

C. Analysis & Results

The accessing analysis is as follows. As shown in Fig. 7, the pooling operation selects a 2*2 sized submatrix block at a time in the entire two-dimensional array. Due to the non-serial access of the matrix, it cannot meet the condition of streaming

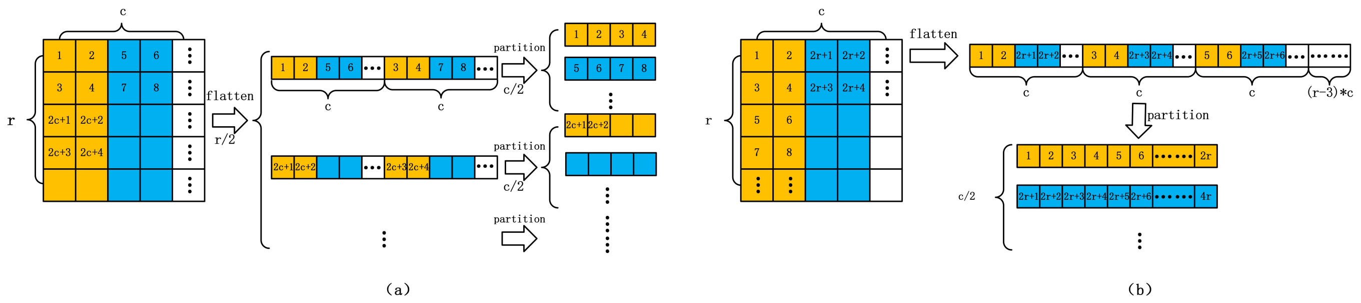


Fig. 7 The illustration of the CNN case study: (a) is the row-major access between blocks, and (b) is the column-major access.

TABLE II QoR of Streaming and Ping-pong Dataflow

Dataflow	Power (W)	Clock Cycle	Timing (ns)	Latency (ns)
Ping-pong	0.290	53293	9.169	488148
Streaming	0.311	32582	10.702	348692

dataflow automatically. While the current array access in each continuous 2 rows conforms to Equation (1), we can partition the matrix in every 2-columns-2-rows. In this way, the access is continuous in each block. Then the streaming dataflow can be used.

According to Equation (1), as shown in Fig. 7 (a), if we flatten the array to one-dimensional matrix in row-major order, the first two elements corresponding to $A_1(1, 2)$, the first row in original two-dimensional matrix corresponding to $B(1, 3, 2)$, the first two rows in original two-dimensional matrix corresponding to $C(1, c/2, 2)$, and the new one-dimensional matrix corresponding to $r/2$ $C(1, c/2, 2)$, thus it can be partitioned in $r/2 * c/2$ blocks which is accessed in the serial order. At the same time, if we change the access order to Fig. 7 (b), the accessing mode of the new one-dimensional matrix could match $C(r, c/2, 2)$, which is means that it can be partitioned in $c/2$ blocks.

After converting the one-dimensional matrix to the original matrix, we can achieve the partitioning in Fig. 7 (b) through *array_partition* pragma with parameter "block" according to Fig. 4. The simulation results of the CNN case are shown in TABLE II. Due to the impact of the memory bank switch and the increasing number of FIFOs caused by array partitioning, the power of the streaming dataflow is 7.2% higher than the power of the ping-pong dataflow. At the same time, the latency (i.e., clock cycle * timing) of streaming dataflow is 28.6% less than the latency of ping-pong dataflow.

V. Conclusion

This paper proposes an efficient array partitioning method to ease the constraints of the streaming dataflow. With this method, the HLS tool can infer the streaming dataflow structure automatically. It is good news because the streaming channel can bring better performance than the default ping-pong structure. The experimental case study on CNN indicates that the proposed method could achieve about 28.6% performance improvements, with the cost of 7.2% power increasement. In the future, how to analyze and adjust the inconsistent access to fit the dataflow requirement is our focus.

References

[1] L. Ma *et al.*, "Acceleration by inline cache for memory-intensive algorithms on FPGA via high-level synthesis," *IEEE Access*, vol. 5, pp. 18 953–18 974, 2017.

[2] H. Zheng *et al.*, "High-level synthesis with behavioral-level multicycle path analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 12, pp. 1832–1845, 2014.

[3] "Synphony model compiler," https://www.microsemi.com/product-directory/dev-tools/4899-synphony.

[4] "C-to-Silicon," https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[5] "Calypto," http://www.calypto.com/catapult_c_synthesis.php.

[6] "Xilinx Vitis HLS," https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/.

[7] B. C. Schafer *et al.*, "Adaptive simulated annealer for high level synthesis design space exploration," in *International Symposium on VLSI Design, Automation and Test*, 2009, pp. 106–109.

[8] M. Alle *et al.*, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *ACM/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[9] N. Wu *et al.*, "High-level synthesis performance prediction using GNNs: Benchmarking, modeling, and advancing," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 49–54.

[10] W. Zuo *et al.*, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2013, pp. 9–18.

[11] T. Liang *et al.*, "Hi-ClockFlow: Multi-clock dataflow automation and throughput optimization in high-level synthesis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–6.

[12] Z. Zou *et al.*, "DataMaster: A GNN-based Data Type Optimizer for Dataflow Design in FPGA," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 247–249.

[13] Y. Wang *et al.*, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2014, pp. 199–208.

[14] J. Escobedo *et al.*, "Graph-theoretically optimal memory banking for stencil-based computing kernels," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 199–208.

[15] A. Cilardo *et al.*, "Interplay of loop unrolling and multidimensional memory partitioning in hls," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2015, pp. 163–168.

[16] A. Silitonga *et al.*, "HLS-Based Performance and Resource Optimization of Cryptographic Modules," in *IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications*, 2018, pp. 1009–1016.

[17] Y.-k. Choi *et al.*, "HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[18] "HLS-CNN Code," https://github.com/FedericoSerafini/HLS-CNN/tree/main/Code.