Performance Modeling and Directives Optimization for High Level Synthesis on FPGA

Jieru Zhao, *IEEE Student Member,* Liang Feng, *IEEE Student Member,* Sharad Sinha, *IEEE Member,* Wei Zhang, *IEEE Member,* Yun Liang, *IEEE Member,* and Bingsheng He, *IEEE Member*

***Abstract*—High level synthesis (HLS) relies on the use of synthesis directives to generate digital designs meeting a set of specifications. However, the selection of directives depends largely on designer experience and knowledge of the target architecture and digital design. Existing automated methods of directive selection are very limited in scope and capability to analyze complex design descriptions in high-level languages to be synthesized using HLS. This article proposes a comprehensive model-based analysis framework, COMBA, which is capable of analyzing the effects of a multitude of directives related to functions, loops and arrays in the design description using pluggable analytical models, a recursive data collector and a metric-guided design space exploration algorithm. COMBA reports a small average error in estimating performance when compared with HLS tools like Vivado HLS, and finds a high-performance configuration of synthesis directives within minutes. Given different resource constraints, COMBA finds configurations with higher speed-ups, compared with the state-of-the-art. Moreover, COMBA can be used to guide the performance and area trade-off analysis. Experiments also show that our design space exploration algorithm outperforms the conventional genetic algorithm, and COMBA efficiently finds a near-optimal configuration, which proves the efficiency of our tool for optimizing the practical HLS based designs.**

***Index Terms*—FPGAs, High Level Synthesis, Optimization**

I. Introduction

FPGAs speed up the system performance significantly with low energy consumption, attracting increasing attention in a wide variety of applications [1, 2]. However, the implementation on FPGAs requires deep comprehension of the hardware architecture and great effort to write register transfer level (RTL) codes, which is error-prone and time-consuming. By automatically synthesizing behavioral descriptions into RTL codes, high level synthesis (HLS) has been developed to improve the FPGA programmability. Several FPGA vendors, such as Xilinx and Intel Altera, have released HLS tools [3, 4]. However, the quality of the resulting RTL designs largely depend on the configuration of synthesis directives provided by HLS tools [3]. Significant speed-up can be achieved with properly chosen directives, while improper selection can worsen the design performance. Therefore, an optimal configuration of synthesis directives which maximizes the performance under limited resource constraints is highly beneficial. Finding such a configuration, however, is non-trivial given multiple directives and exponentially increasing design space.

In this article, we propose COMBA, a comprehensive model-based analysis framework, to help designers select the most suitable configuration for C-based description of algorithms to be implemented on FPGAs. COMBA includes a recursive data collector (RDC), a performance and resource estimation model and an improved metric-guided design space exploration (MGDSE-II) algorithm. The RDC computes the required parameters for our models, supporting a rich set of C/C++ code structures and achieving cycle-level accuracy by considering operation chaining. The models consider more directives than previous works, as given in Table I. We further refine the models by considering different BRAM modes, the influence of *array partitioning* on the memory resource estimation and the effects of the channels introduced by *dataflow*. Moreover, we demonstrate how directives interact with each other and analyze how improper array partitioning degrades the application performance, which help direct the search of the optimal configuration. With more complex code structures and more directives, the design space increases exponentially and the brute-force method [5, 6] cannot work. Therefore, we propose MGDSE-II, a metric-guided design space exploration algorithm with three evaluation metrics to prune and explore the design space. By considering the interplay of different arrays, better configurations could be found by MGDSE-II.

Experimental results show that COMBA models the performance closely compared to Vivado HLS, and finds a high-performance configuration within minutes in an exponentially increasing design space. A preliminary version of this paper appears in [7]. In this paper, we refine our analytical models, improve our design space exploration (DSE) algorithm, study the impact of the resource constraints and analyze the trade-off relationship between performance and area.

II. Background

A. Synthesis Directives

HLS tools provide several optimization techniques, known as synthesis directives or pragmas, to create different hardware implementations from the same C/C++ source code [3]. The performance of generated hardware circuits varies with the configurations of directives [7]. This is because each directive has its specific effects on the performance and there is mutual

Manuscript received xxxxxxxxxx; revised xxxxxxxxxx; accepted xxxxxxxxxx. Date of publication xxxxxxxxxx; date of current version xxxxxxxxxx; This work was supported by the Hong Kong Research Grants Council General Research Funds under Grant 16245116. This paper was recommended by Associate Editor Hung William. (*Corresponding author: Jieru Zhao.*)

Jieru Zhao, Liang Feng and Wei Zhang are with Reconfiguration Computing Systems Lab, Department of ECE, Hong Kong University of Science and Technology; email: {jzhaoao, lfengad, wei.zhang}@ust.hk.

Sharad Sinha (sharad_sinha@ieee.org) is an assistant professor with Dept. of Computer Science and Engineering, Indian Institute of Technology (IIT) Goa. This work was done when he was with NTU, Singapore.

Yun Liang (ericlyun@pku.edu.cn) is an associate professor with School of EECS, Peking University, China.

Bingsheng He (hebs@comp.nus.edu.sg) is an associate professor with School of Computing, National University of Singapore.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier xxxxxx

TABLE I: CONFIGURATION OF SYNTHESIS DIRECTIVES

Directives	Target	Configuration
Loop unrolling	Each loop level	Unrolling factors
Loop pipelining	Each loop level	Enabled/Disabled
Array partitioning	Each array dimension	Block/Cyclic/Complete
Function pipelining	Each function	Enabled/Disabled
Dataflow	Top loop/function	Enabled/Disabled
Loop flattening	Inter loop levels	Yes/No
Function inlining	Each function	Yes/No

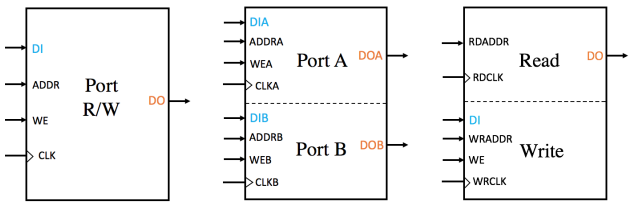
interaction among different directives. To differentiate their behaviors and understand the interaction, multiple methods have been proposed in the HLS research community, but most of these focus on two or three directives [6, 8]–[12]. To cover more complex designs and to target real applications, more directives need to be considered. Table I lists the main characteristics of seven widely-used synthesis directives, provided by Vivado HLS and supported by COMBA. The "target" denotes where the directive can be applied and the "configuration" shows what parameters should be set for each directive.

B. BRAM Modes

On most modern FPGAs, like Xilinx Virtex-7, each block RAM (BRAM) can be configured as either two independent 18Kb RAMs, or one 36Kb RAM, and set to three modes: single-port (SP) mode, true dual-port (TDP) mode and simple dual-port (SDP) mode [13], as shown in Fig. 1 and Table II. The BRAM modes impact the read/write latency and resource usage. The SP mode cannot support simultaneous read and write operations, which may incur a delay and degrade the performance. In the TDP mode, simultaneous memory accesses are supported, but more blocks may be needed to store the same data set due to the smaller maximum data width, thereby increasing the resource usage. Take the 18Kb TDP BRAM as an example, and if the stored data is 32-bit wide, two blocks are required to cover the data width since the maximum width of each block is 18. In the SDP mode, the port width is the double of that of the TDP mode, and the independent read and write ports avoid the conflict problem. However, it cannot support simultaneous multiple reads or multiple writes. Due to the trade-off, a careful choice of the BRAM modes is important. By modeling their effects on the performance and area, COMBA is capable of selecting the suitable BRAM mode for a specific application.

TABLE II: MAXIMUM PORT WIDTH OF DIFFERENT MODES

Storage of block RAM	SP mode	TDP mode	SDP mode
18 K-bit	36	18	36
36 K-bit	72	36	72

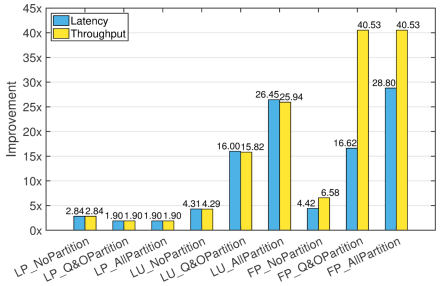


(a) SP mode (b) TDP mode (c) SDP mode

Fig. 1: Three BRAM modes. (a) The SP mode contains one port for both read and write. (b) The TDP mode contains two independent ports A and B, each of which is used for both reads and writes. (c) The SDP mode includes two independent ports, designated as the read port and the write port separately.

```
void decode_block(int QuantBuff[64],
                  int Out_buf[64],
                  int HuffBuff[64])
{
  IZigzagMatrix(HuffBuff, QuantBuff);
  // one loop with bound 64
  IQuantize(QuantBuff);
  // one loop with bound 64
  ChenIDct(QuantBuff, Out_buf);
  // three loops with bounds 8,8,64
  PostshiftIDctMatrix(Out_buf);
  // one loop with bound 64
  BoundIDctMatrix(Out_buf);
  // one loop with bound 64
}
```

(a) Source code



(b) Performance Speed-ups

Fig. 2: Motivation example: *decode_block* from JPEG application. (a) Source code. (b) Speed-ups of latency and throughput for different configurations.

C. Motivation

Real applications contain complex code structures with coupled functions and loops as well as multi-dimension arrays. Figure 2(a) shows the *"decode_block"* kernel in the JPEG application [14], which contains five sub-functions, seven loops and three arrays. For different configurations, the performance improvement, represented by *latency* and *throughput*, varies, as shown in Fig. 2(b). We compare loop pipelining (LP), loop unrolling (LU) and function pipelining (FP), combined with array partitioning for different arrays. Both LU and FP improve the performance greatly if a beneficial array configuration is chosen, while LP plays a minor role. Specifically, FP further improves *throughput* compared with LU. Therefore, function-related directives, like FP, which are not considered in previous works, are of great importance in real applications. Loop-related directives, such as LP and LU, have different effects on the performance, depending on the specific loop structures. Only considering one nested loop, as in [6], does not apply to other loop structures and cannot reveal the relationship between loops. Moreover, the array configuration has a significant influence, resulting in great speed-ups by carefully deciding how to map arrays to memories and how to choose BRAM modes. Therefore, a comprehensive and accurate estimation model is required to differentiate the effects of directives, analyze their mutual interaction and evaluate the performance of different configurations for complex applications.

Considering more directives identifies the design characteristics but leads to a larger design space. Moreover, the different choices of the BRAM modes further expand the design space and increase the difficulty of selecting the best optimization scheme. Given the example in Fig. 2, the functions may or may not be pipelined and *dataflow* may or may not be applied to the top function, resulting in $2^6 * 2$ choices in total. Similarly, configurations of loops and arrays contain $2^7 * (4^2 * 7^5)$ and 12^3 choices, respectively. Considering the three BRAM modes, the design space consists of $2 * 2^6 * 12^3 * 2^7 * 4^2 * 7^5 * 3$ points, which is so large that invoking HLS tools to test each configuration is infeasible. Therefore, it is crucial to develop an efficient DSE algorithm for rapid architectural exploration in an exponentially increasing design space.

III. PROBLEM FORMULATION

In a large design space with numerous configurations of directives, the problem is how to select the optimal configuration, which provides the best delay given certain resource constraints. We formulate the resource-constrained performance

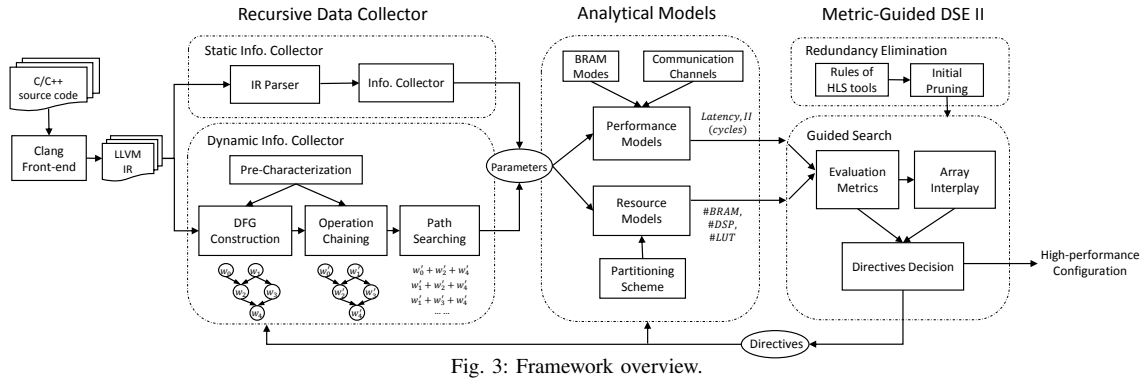


Fig. 3: Framework overview.

optimization problem as:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & r_i(x) \leq c_i,\ i = bram, dsp, lut \end{aligned} \tag{1}$$

where $f(x)$ is the performance estimation function and denotes the execution time (cycle) when applied with configuration x; $r_i(x)$ is the corresponding resource usage, which is bounded by the resource constraint c_i; and i denotes the resource components on FPGAs, i.e., BRAMs, DSPs and LUTs. We do not include flip-flops because they are abundant on FPGAs.

To solve this problem, we develop a comprehensive model-based framework, as shown in Fig. 3. The input is a C/C++ application, and the output is a high-performance configuration (i.e., directive setting) under given resource constraints. First, the C/C++ specification is translated into LLVM IR via the Clang front-end [15]. Next, the IR is sent to the recursive data collector (RDC) to compute the parameters required by our models based on the directive setting and pre-characterization information. With the parameters from RDC, the proposed models estimate the performance and area for the corresponding configuration. Finally, the MGDSE-II evaluates the results and sets the next configuration, and then COMBA iterates from the RDC stage until it finds the high-performance configuration. Note that the proposed models are pluggable and hence can accommodate a vast range of target FPGA architectures, though we use Virtex-7 in this paper.

IV. Recursive Data Collection

To obtain required data statistics, our recursive data collector extracts necessary data information, stores the relationship between instructions and constructs data flow graph (DFG). The input of RDC is the LLVM intermediate representation (IR), which can be transformed into a control and data flow graph [16] and allows efficient analysis through LLVM passes. Our RDC is implemented as an LLVM pass based on *llvm::Module* class, and analyzes the LLVM IR to compute the parameters. The parameters are divided into two categories: the static information, e.g., the memory address of each array element, and the dynamic information, e.g., the iteration latency of loops. Static information is obtained by analyzing the assembly instructions of LLVM IR through IR parser, as shown in Fig. 3, while dynamic information depends on code structures and directives applied, and is computed using DFG.

A. DFG Construction

The DFG constructed by our RDC covers a rich set of code structures like nested loops, function calls and if-else and switch branches. It is constructed by connecting dependent instructions and storing each instruction as a node, with its latency as the node weight. A dynamic programming approach [17, 18] is then employed to trace each path between two dependent instructions, calculate the latency between them and search the longest path. The DFG construction phase also takes into account loop and function hierarchies in the design description and accordingly computes their latencies by adhering to data and control flow dependencies. Sub-functions and loops within the function hierarchy are defined as *"sub-elements"* and viewed as nodes in the DFG of the top function.

```
for(int i = 0; i < 10; i++)
    b[i] = a[i] + i;        //load a[i], store b[i]
for(int j = 0; j < 10; j++)
    c[j] = b[j] * j;        //load b[j], store c[j]
```

Fig. 4: Example of node weight setting: load/store operations.

B. Node Weight Setting

We obtain the node weight by characterizing from micro-benchmarks. Specifically, the latency of load (read) and store (write) instructions depends on two factors. First, it depends on whether or not the memory ports are available. If available, the load latency is two clock cycles (i.e., generating an address in one cycle then reading the data in the next) and the store latency is one clock cycle. If the memory ports are not available, the delay caused by other load/store instructions should be added to get the actual latency. Second, it depends on what directives are set. In the example in Fig. 4, if both loops are unrolled completely, the second loop can access the result of the first loop directly and does not need to load $b[i]$ again. Therefore, the latency of load $b[i]$ is set to zero.

C. Operation Chaining

Our framework also considers *"operation chaining"* to improve scheduling and mimic the scheduling behavior in commercial HLS tools. *Operation chaining* means that more than one operation can be scheduled in one cycle if possible. An example can be seen in [7]. Characterization-based information is used to decide on whether the operations can be chained or not, and this is used to modify the weight of each node when constructing the DFG.

V. Performance Model

In this section, we present our performance model for five frequently-used directives, namely, loop unrolling, loop pipelining, array partitioning, function pipelining and dataflow.

```
for(int i = 0; i < 10; i++){
  for(int j = 0; j < 10; j++){
    d = a[i] + i;
    c[j] = b[j] * d;
  }
}
```

(a) Perfect nested loop

```
for(int i = 0; i < 10; i++){
  d = a[i] + i;
  for(int j = 0; j < 10; j++){
    c[j] = b[j] * d;
  }
}
```

(b) Non-perfect nested loop

```
for(int i = 0; i < 10; i++){
    d += a[i] + i;
}
for(int j = 0; j < 10; j++){
    c[j] = b[j] * d;
}
```

(c) Multiple loops

Fig. 5: Three kinds of loops. (a) Perfect nested loop: only the innermost loop has contents. (b) Non-perfect nested loop: some logic is specified between loop statements. (c) Multiple loops: more than one loop at the same level.

A. Loop Unrolling

Loop unrolling (LU) allows iterations of one loop to execute in parallel. To show the general case, we define a nested loop $\mathcal{L} = \{L_1, \ldots, L_i, \ldots, L_n\}$, where n is the number of loop levels in $\mathcal{L}$ and L_1, L_i and L_n are the outermost loop, sub-loop in level i and the innermost loop, respectively. Let $\mathcal{B} = \{B_1, \ldots, B_i, \ldots, B_n\}$ denote the set of loop bounds and $\mathcal{U} = \{U_1, \ldots, U_i, \ldots, U_n\}$ denote the set of loop unrolling factors respectively. The loop latency is estimated in a recursive way:

$$C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k}, \tag{2}$$

where L_k is the loop in level k; L_{k+1} is the inner loop of L_k; $C_{L_k}^{U_k}$ and $C_{L_{k+1}}^{U_{k+1}}$ are the iteration latencies of L_k and L_{k+1} with unrolling factors U_k and U_{k+1}, respectively; $\frac{B_{k+1}}{U_{k+1}}$ is the trip count after LU is applied; and U_k is a multiplication factor since loops are scheduled in sequence in Vivado HLS, even if they are independent [3]. When L_k is unrolled with U_k, its sub-loop L_{k+1} will be replicated, generating U_k copies to execute in sequence. $C_{L_k \setminus L_{k+1}}^{U_k}$ is the critical-path latency of the logic specified between the loop statements, that is, the codes within L_k and outside L_{k+1}, and is returned by RDC.

The initial state of the recursion is the iteration latency $(C_{L_r}^{U_r})$ of L_r, which is not unrolled completely with inner loops $\{L_{r+1}, \ldots, L_n\}$ unrolled completely. $C_{L_r}^{U_r}$ and the loop bounds are returned by RDC, and the unrolling factors come from the directive setting. Considering each part of one loop, Eq. 2 works for all loop hierarchies, including perfect and non-perfect nested loops, and multiple loops, as shown in Fig. 5. For multiple loops, $C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}}$ becomes $\sum_{j=0}^{m} C_{L_{k+1,j}}^{U_{k+1,j}} \cdot \frac{B_{k+1,j}}{U_{k+1,j}}$ to add the latencies of loops at the same level $k+1$. The latency of the loop in level k (L_k) can then be computed as $Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}$.

B. Loop Pipelining

Loop pipelining allows operations from different iterations to overlap for parallelism. There are three factors: the trip count, the initiation interval and the pipeline depth [19].

1) Trip Count: The trip count is the number of iterations in a pipeline, depending on whether the loop is perfect or not. In a perfect nested loop, when the inner loop is pipelined, the outer loops that are not unrolled can be flattened to feed the inner loop with new data and form a deeper pipeline to improve the overall throughput. Then the trip count is the multiplication of each loop's trip count, as shown in Eq. 3. For non-perfect loops, the codes between loop statements prevent the outer loops from flattening, and the trip count is equal to the inner loop's trip count, $\frac{B_i}{U_i}$ (the trip count of L_i), as shown in Eq. 4. i is the level of the loop which is pipelined.

$$Cycle_{L_k} = D_i + II_i \cdot \left(\frac{B_i}{U_i} \cdot B_{i-1} B_{i-2} \cdots B_k - 1 \right), \tag{3}$$

where $Cycle_{L_k}$ is the latency of L_k, D_i is the pipeline depth of loop L_i, and II_i is the initiation interval. Except for L_i, outer loops are not unrolled and can be flattened; otherwise, the pipeline is maintained up to the loop level which is unrolled.

$$Cycle_{L_i} = D_i + II_i \cdot \left(\frac{B_i}{U_i} - 1 \right), \tag{4}$$

where $Cycle_{L_i}$ is the latency of L_i in an imperfect loop.

2) Initiation Interval: The initiation interval, II_i, is the latency between the initiation of two consecutive iterations. Minimal II_i is constrained by available resources and the loop-carried dependence [20], shown as Eq. 5:

$$II_{i,\min} = \max\left(II_{i,\min}^{res}, II_{i,\min}^{rec}\right), \tag{5}$$

where $II_{i,\min}^{res}$ and $II_{i,\min}^{rec}$ are the resource-constrained and the recurrence-constrained minimum initiation intervals of L_i, respectively. Moreover, sub-functions within L_i are also pipelined, affecting II_i, shown as Eq. 6:

$$II_i = \max\left(II_{i,\min}, II_{sub,\max}\right), \tag{6}$$

where $II_{sub,\max}$ is the maximum initiation interval among all the sub-functions within L_i, i.e., $\max_{sub}(II_{sub})$.

When estimating $II_{i,\min}^{res}$, we assume that computation operators are sufficient [21] and $II_{i,\min}^{res}$ is constrained by memory operations. Besides the LUT-based RAMs, in most cases Vivado HLS automatically maps arrays to single-port, simple dual-port or true dual-port block RAMs, depending on the actual needs. The limited number of BRAM ports in different modes constrains $II_{i,\min}^{res}$ in different ways, shown as Eq. 7:

$$II_{i,\min}^{res} = \begin{cases} \max\limits_{m}\left(\left\lceil \frac{Read_m}{RPort_m} \right\rceil + \left\lceil \frac{Write_m}{WPort_m} \right\rceil\right) & \text{for SP \& TDP} \\ \max\limits_{m}\left(\max\left(\left\lceil \frac{Read_m}{RPort_m} \right\rceil, \left\lceil \frac{Write_m}{WPort_m} \right\rceil\right)\right) & \text{for SDP} \end{cases} \tag{7}$$

where $Read_m$ and $Write_m$ denote the number of read and write operations to array m correspondingly; and $RPort_m$ and $WPort_m$ are the number of read and write ports, respectively, of the memory that stores array m in corresponding memory modes. Specifically, in SP and TDP modes, the II caused by array m is calculated by adding both read and write operations together considering the load and store conflict, while it is the larger value in the SDP mode since the read and write ports are independent, avoiding the conflict problem. Note that if array m is partitioned, $Read_m$, $Write_m$, $RPort_m$ and $WPort_m$ become the number of read/write operations and ports of corresponding partitions, respectively. An example is given in Section IX-B to show the impact of BRAM modes on the II estimation.

$II_{i,\min}^{rec}$ is computed as Eq. 8 [22]:

$$II_{i,\min}^{rec} = \max_{p}\left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil\right), \tag{8}$$

where $Delay_p$ is the latency between a pair (p) of dependent instructions from different iterations and $Distance_p$ is the result of subtracting the corresponding iteration numbers.

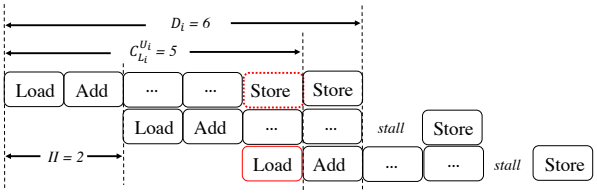


Fig. 6: Pipeline depth.

3) Pipeline Depth: Pipeline depth, D_i, is the latency of one iteration of the pipelined loop, which is related to the original iteration latency $C_{L_i}^{U_i}$. Specifically, if read and write operations access different arrays or the memory is in the SDP mode, D_i equals $C_{L_i}^{U_i}$; otherwise, D_i is computed as $D_i = \lceil C_{L_i}^{U_i}/II_i \rceil \cdot II_i$, considering load/store conflict. Take Fig. 6 as an example, suppose the memory has a single port, an array element is loaded in the first cycle and the result is stored in the same array in the fifth cycle. Therefore $C_{L_i}^{U_i}$ is five cycles. However, when L_i is pipelined with $II = 2$, *store* in the first iteration will conflict with *load* in the third iteration, as shown in the red boxes. Then *store* will be scheduled in the sixth cycle, generating a *"stall"* in each iteration. Therefore, the pipeline depth D_i is six cycles, which can be calculated as $\lceil \frac{5}{2} \rceil \cdot 2 = 6$.

C. Array Partitioning

Memory operations are often the performance bottleneck in real applications. To enable simultaneous accesses, HLS tools like Vivado HLS allow arrays to be partitioned into smaller ones in different dimensions, providing three options: *block*, *cyclic* and *complete* [3]. Our framework supports multi-dimension array partitioning with the same options. By tracking each load/store node and finding the address index of the accessed element, RDC calculates which partition P the element is located in using Eq. 9, and checks whether this partition's ports are available to compute the load/store latency.

$$P_i = \lfloor index_i / \lceil size_i / f_i \rceil \rfloor \tag{9a}$$

$$P_i = (index_i) \bmod (f_i) \tag{9b}$$

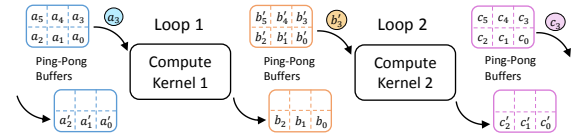
$$P = P_1 + \sum_{i=2}^{n} (P_i \cdot \prod_{k=1}^{i-1} f_k), \tag{9c}$$

where P_i is the partition number in dimension i, $index_i$ is the address index of the array element, $size_i$ is the number of elements and f_i is the array partitioning factor. P in Eq. 9c is the partition number considering n-dimension array partitioning. The partition number in one dimension is calculated in Eq. 9a (*block*) and Eq. 9b (*cyclic*). For the *complete* option, it is calculated by setting $f_i = size_i$ in both equations.

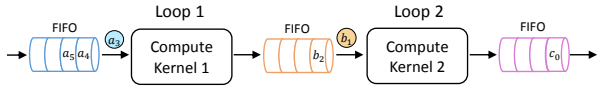
D. Function Pipelining

Real applications often contain multiple functions. As such, *function pipelining* is critical to performance improvement by allowing concurrent execution of operations within a function. We estimate *latency* and *throughput* to evaluate the function performance. Tools like Vivado HLS unroll all the sub-loops completely and pipeline each sub-function inside a pipelined function. Based on this feature, the *latency* is returned by our RDC after building the DFG. The *throughput* measures the number of outputs per cycle and is calculated as $\frac{1}{II}$. II is the initiation interval of a pipelined function as follows:

$$II = \max\left(II_{\min}^{res}, II_{\max}^{sub}\right), \tag{10}$$



(a) Dataflow using ping-pong buffers



(b) Dataflow using FIFOs

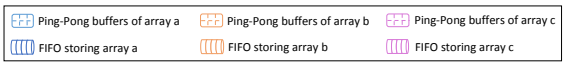


Fig. 7: Illustration of the communication channels for *dataflow*.

where $II_{\min}^{res}$ is the resource-constrained minimum II, and $II_{\max}^{sub}$ is the maximum function II among all sub-functions.

$II_{\max}^{sub}$ is computed by comparing each sub-function's II and choosing the maximal one. $II_{\min}^{res}$ is estimated by counting the number of memory operations in the function, including memory operations in the sub-functions and the logic surrounding sub-functions, then dividing by the number of memory ports.

E. Dataflow

Unlike *function pipelining*, *dataflow* is the "coarse grain" pipelining at the task level, allowing functions and loops to operate concurrently. It does not require sub-functions to be pipelined and sub-loops to be unrolled, but can only be applied to functions at the top level. Also, it aims at applications executing in a producer-consumer model, i.e., the output of the last function/loop is the input of the next function/loop. Many real applications can benefit from this directive, such as applications in the image processing [23] and data transmission [24] domains. In Vivado HLS, functions and loops are extracted as "process functions" (blocks of codes) and *dataflow* places channels between the blocks to keep the data rate, resulting in additional area overhead, which will be discussed in section VI. We also utilize *throughput* and *latency* to evaluate the performance impact of *dataflow*.

1) Throughput: The initiation interval II is used to measure the *throughput*, shown as follows:

$$II = II_{\max}^{sub} = \max_i \left(II_i^{sub}\right), \tag{11}$$

where II equals the maximal initiation interval among the process functions extracted from the sub-elements. Specifically, for a sub-function i, the initiation interval of its corresponding process function, II_i^{sub}, equals the initiation interval of sub-function i. However, for a sub-loop, the II_i^{sub} of its corresponding process function is equal to the latency of the sub-loop regardless of whether or not it is pipelined. This is because the initiation interval of a pipelined loop is the interval between two consecutive iterations, but the II of *dataflow* is larger and measures the interval between two successive loops.

2) Latency: When *dataflow* applied on the top function, the function latency is influenced by the memory configuration of the channels between process functions, shown as follows:

$$Cycle_{Func} = \begin{cases} \max\limits_i \left(Cycle_{L_i}\right) & \text{if FIFOs used,} \\ \sum_i^n \left(Cycle_{L_i}\right) & \text{if ping-pong buffers used.} \end{cases} \tag{12}$$

As introduced before, the channels placed by *dataflow* are to keep the data rate, as shown in Fig. 7. For arrays, the

channels are implemented using ping-pong buffers (i.e., double buffering) or FIFOs [3]. Given a function containing multiple loops, when using FIFOs, data can be transferred to the next loop once processed by the previous loop, improving the efficiency of the data transmission. In this case, the latency of the top function is nearly the latency of the longest sub-loop. Note that they are not exactly equal but are very close since it typically takes one cycle to enter or exit a loop, which is an additional cost. This method is more efficient with smaller area overhead but is constrained to the case that data must be accessed in a sequential order among all the sub-loops. The more frequently used type is the ping-pong buffer. It contains two parts storing the same data and permits one loop to access one part while another loop accesses the other part, improving the efficiency without the sequential constraint. However, the next loop can only obtain the data after the previous loop finishes processing and transferring all the data in one array. Therefore, it is less efficient and the memory resource usage also doubles compared with FIFOs. In this case, the latency of the top function is not reduced and is computed by adding each loop latency as in Eq. 12, but the throughput is still improved.

VI. Resource Model

To obtain the fastest implementation under the fixed resource constraints, a resource model is required to check whether the usage exceeds the available resources.

A. DSP and LUT Estimation

We characterize the resource usage of different operators from micro-benchmarks. For instance, a 32-bit floating point multiplication is mapped to a floating point unit with three DSPs. Then we take into account resource sharing, according to [19, 21]. Specifically, [19] reports an accurate estimation for DSPs and a small estimation error for LUTs (i.e., under 5%), and [21] shows their LUT estimation accurately fits the actual trend. Therefore, we make use of the same method and estimate the resource usage through resource sharing analysis.

For LUT-based and small bitwidth operations, such as integer add/subtract operations, resource sharing incurs large resource usage due to the multiplexers introduced [19, 25]. Therefore, they are not shared, and the number of operations equals the number of allocated instances. Conversely, DSP-based operators, or operators with enough complexity compared to multiplexers, such as double point multiplication operations [19], are sharable with higher efficiency and larger area, so the operator usage is the maximum number of operators executing in parallel. For sharable operators in a pipelined loop, we compute the lower bound $N_{\min}^{op} = \lceil \frac{N_{op}}{II} \rceil$, to calculate the number of instances that must be allocated [21]. N_{op} is the number of the operation *op* used in one iteration, and *II* is the initiation interval of the loop. For example, if *II* is 2 and four floating point adders are used in one iteration, then at least two floating point adders should be allocated. Based on the resource sharing analysis, the usage of DSPs and LUTs for each allocated operator is accumulated and added up.

B. BRAM Estimation

Local reads/writes consume memory resources on FPGAs, and arrays are synthesized into BRAMs, which are provided in blocks, with each block containing 18Kb or 36Kb primitive elements for data storage on most modern Xilinx FPGAs. Each array is synthesized into its own BRAM which contains one or multiple blocks. We model the BRAM usage R_{bram} for each array in Eq. 13 and add them up as the total memory usage:

$$R_{bram} = \left\lceil \frac{\#bits}{width} \right\rceil \cdot \left\lceil \frac{\#element}{depth} \right\rceil \cdot \#partition \cdot d, \tag{13}$$

where R_{bram} is the number of blocks, *#bits* is the width of each array element, *#element* is the number of elements per memory partition, and *width* and *depth* is the width and depth of the selected block configuration, respectively. $\lceil \frac{\#bits}{width} \rceil \cdot \lceil \frac{\#element}{depth} \rceil$ computes the usage of blocks for one memory partition. The selection of the block configuration depends on data types, BRAM modes and devices. For example, for Virtex-7 FPGAs, given an array containing 512 32-bit wide elements, the selected configuration of an 18 Kb block RAM is 512×36 for SP and SDP modes (one block is sufficient), and is $1k \times 18$ for the TDP mode since the maximum width in this mode is 18 (two blocks are needed to cover the width of 32). *#partition* is the number of memory partitions, equal to the product of the partitioning factors in each dimension, i.e., $\prod_{i=1}^{n} f_i$, and d reflects the effect of *dataflow*, which utilizes channels to maintain the data rate between sub-elements. For scalars, the channel is a register. For arrays, the channels are *ping-pong* buffers by default, which means that each BRAM has two copies: one is used for the output buffer of the last function/loop and one is used for the input buffer of the next function/loop. So d is 2 if *dataflow* is applied and the channel is implemented using ping-pong buffers, otherwise d equals 1.

Arrays are assumed to be partitioned evenly in Eq. 13, which means that each partition contains the same number of elements. To estimate BRAM usage for uneven partitioning, we have

$$\begin{aligned} R_{bram} = &\left\lceil \frac{\#bits}{width} \right\rceil \cdot \left\lceil \frac{\#element}{depth} \right\rceil \cdot (\#partition - 1) \cdot d \\ &+ \left\lceil \frac{\#bits}{width} \right\rceil \cdot \left\lceil \frac{\#element_r}{depth} \right\rceil \cdot 1 \cdot d, \end{aligned} \tag{14}$$

where $\#element_r$ is the number of the remaining elements after distributing $\lceil \frac{array\_size}{\#partition} \rceil$ elements to each previous partition; and other parameters keep the same meaning as in Eq. 13.

VII. Design Space Exploration

In this section, we present our two-stage MGDSE-II algorithm, which quickly finds a high-performance configuration in a large design space. We improve our previous algorithm in [7], consider the interplay of different arrays and increase the possibility of finding a better configuration.

A. Two-Stage Exploration

1) Redundancy Elimination: The first stage is to remove the redundant design points based on the rules of HLS tools. For example, in Vivado HLS, the sub-loops in a pipelined loop are unrolled completely and cannot be pipelined. Therefore, no matter what unrolling factor is set and whether or not sub-loops are pipelined, the performance remains the same. These kinds of "redundant points" cannot reduce the latency even if we set the corresponding directives. After removing these points, the design space is reduced significantly without sacrificing the quality of the search space.

Algorithm 1: `MGDSE-II`(F), in *case 0*.

```
Initializes: Dataflow = 0, FuncPipelining = 0, O ← ∅,
  S ← the set of the sub-elements in F;
if S! = ∅ then
    while S! = ∅ do
        s_max ← the longest sub-element in S;
        s_s,max ← the second longest sub-element in S;
        M_diff = PerModel(s_max) − PerModel(s_s,max);
        if s_max is a function then
            OptimizeFunc(s_max, M_diff);
        end
        else if s_max is a loop then
            OptimizeLoop(s_max, M_diff);
        end
        if OptFinish(s_max) then
            UpdateDel(S, s_max);
        end
        O ← the current best configuration;
    end
else
    OptimizeArray(F, arrays);
end
```

Algorithm 2: `OptimizeArray`$(F, arrays)$

```
Initializes: Q ← an array of vectors of configurations;
for each array_i in arrays do initialization
    for each dimension dim_j in array_i do
        Push (Type, Factor) = (block, 1) in Q[array_i];
    end
end
while size(Q)! = 0 do
    for each non-empty vector Q[array_i] do
        for each pair in Q[array_i] do
            for (Type, Factor) → (cyclic, bound) do
                resource = ResourceModel(F);
                latency = PerModel(F);
                if (M_res > 1)&(latency not reduced) then
                    Increase Factor or Type and break;
                end
            end
            O ← the current best configuration;
        end
        if array_i is optimized then Clear Q[array_i];
    end
end
```

2) Guided Search: The second stage is to evaluate the performance of the current design point and determine the next design point to be evaluated through three evaluation metrics, namely M_{diff}, M_{res} and M_{apt}, which identify the performance bottlenecks and indicate a suitable direction to explore.

M_{diff} denotes the difference of the latency between the longest sub-element and the second-longest sub-element in the target function, as shown in Eq.15:

$$M_{diff} = C^{sub}_{\max} - C^{sub}_{\text{s,max}}, \tag{15}$$

where $C^{sub}_{\max}$ and $C^{sub}_{\text{s,max}}$ are the latencies of the longest sub-element and the second-longest sub-element, respectively. According to M_{diff}, MGDSE-II algorithm gives the top optimization priority to the longest sub-element, which is assumed to have the greatest influence on the performance.

M_{res} is utilized to check whether the resource usage exceeds the available resources on FPGAs, as shown in Eq.16:

$$M_{res} = \max\left(\frac{BRAM_{used}}{BRAM_{total}}, \frac{DSP_{used}}{DSP_{total}}, \frac{LUT_{used}}{LUT_{total}}\right), \tag{16}$$

where each fraction denotes the percentage of the resources used. If the resource constraints are not satisfied, the corresponding point will be removed from the design space.

M_{apt} is to decide which partitioning type is more beneficial in dimension i. To compute M_{apt}, we have

$$M_{apt,l} = \frac{\#loads}{\max\limits_{j,k}(index^j_i - index^k_i + 1)}, \tag{17a}$$

$$M_{apt,s} = \frac{\#stores}{\max\limits_{j,k}(index^j_i - index^k_i + 1)}, \tag{17b}$$

where $M_{apt,l}$ and $M_{apt,s}$ represent the value of M_{apt} for load and store operations, respectively, and $index^j_i$ and $index^k_i$ are the indexes in dimension i of the respective array elements j and k. If M_{apt} is smaller than one, the array elements are accessed at intervals and *block* is more beneficial. If M_{apt} is equal to one, the elements are accessed continuously and *cyclic* is better. If the option cannot be determined, e.g., $M_{apt,l}$ and $M_{apt,s}$ of the same array are different, both options will be tested. For the *complete* option, since arrays are partitioned into individual elements and no factor needs to be set, there is only one point and it will always be tested.

B. Overall Algorithm Description

As a whole, the MGDSE-II algorithm works as follows. First, the top function F can be applied with *dataflow* or *function pipelining*. If a function is pipelined, *dataflow* will be ignored. Therefore, there are three choices for the top function: the first is applying *dataflow* (*case 1*); the second is applying *function pipelining* (*case 2*); and the last is applying neither of them (*case 0*). MGDSE-II explores the design space in all three cases one by one. In *case 0*, as presented in Alg. 1, MGDSE-II optimizes the longest sub-element $s_{\max}$ each time, aiming at minimizing M_{diff} until zero. Then it optimizes the new longest sub-element in the optimization set $\mathcal{S}$. If M_{diff} is still larger than zero after optimization, the optimized sub-element will be removed out of the optimization set by `UpdateDel`$(\mathcal{S}, s_{\max})$. Through this iterative way, the latency of all sub-elements will be minimized while honoring the resource constraints. In *case 1*, MGDSE-II first checks if the *producer-consumer* requirement is satisfied. If satisfied, the exploration will be conducted the same way as *case 0* except for that *dataflow* is set to one for initialization; otherwise it will be skipped. In *case 2*, the top function is pipelined, and the configuration of sub-functions (pipelined) and loops (unrolled completely) is fixed. Then MGDSE-II explores the design space by varying the options of *array partitioning* based on M_{apt} and computes M_{res} to remove the points that exceed resource constraints.

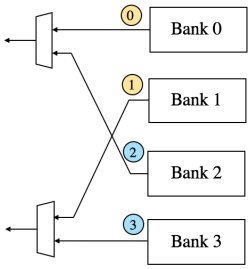
In *cases 0* and *1*, when optimizing the sub-elements, the sub-functions are viewed as new target functions in `OptimizeFunc`$(s_{\max}, M_{diff})$ and are sent to MGDSE-II recursively. For loops, MGDSE-II optimizes them through

```
for(int i=0; i<64; i++)
   #pragma HLS PIPELINE II=1
   a[i] += a[i] * i;
```

(a) Original code

```
for(int i=0; i<64; i+=2)
   #pragma HLS PIPELINE II=1
   a[i] += a[i] * i;
   a[i+1] += a[i+1] * (i+1);
```

(b) Loop unrolling with factor 2



(c) Array partitioning: cyclic, factor 4

Fig. 8: Impacts of improper array partitioning. (a) Original code: no loop-carried dependence. (b) The loop is unrolled with factor 2: still no loop-carried dependence. (c) Diagram of array partitioning for array a in (b): improper partitioning incurs loop-carried dependence. The yellow and blue circles denote the data required by the first and second iterations, respectively, which are distributed to the four memory banks.

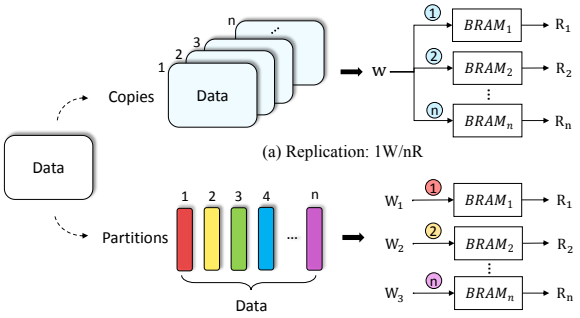


Fig. 9: Two types of multi-port memories: (a) replication; and (b) banking. Each BRAM is a 1W/1R memory and the same color indicates the same data.

$\texttt{OptimizeLoop}(s_{\max}, M_{diff})$, starting from the innermost loop and then the outer loops in each level. Each level's loop is applied with *loop pipelining* and *unrolling*. Then MGDSE-II computes M_{res} and M_{apt} for each loop configuration to vary the setting of *array partitioning* and decide the exploration direction. Based on the evaluation metrics, MGDSE-II ignores the points that worsen the performance and chooses a promising configuration as the next point, exploring the design space and finding a high-performance configuration in minutes.

C. Array Interplay in MGDSE-II

The major difference, between MGDSE-II and our previous MGDSE algorithm [7], is the way they optimize the arrays. The algorithm in [7] optimizes arrays one by one. For each configuration of functions and loops, it varies the type and factor of *array partitioning*, tests each array configuration for different dimensions and chooses the best partitioning scheme for one array. Then it conducts the same exploration for the next array until all the arrays are optimized. In MGDSE-II, we optimize arrays in a different way, as shown in Alg. 2, by considering the interplay of different arrays in the same function or loop. The input is all the arrays of a function F or a loop L. Compared with the previous algorithm, MGDSE-II conducts a finer grain array optimization. Specifically, it first selects one array to optimize from the initial state $(block, 1)$, which means the array partitioning type is *block* with factor one (no partitioning). During the optimization of one array, if the latency is not reduced or the resource constraint is not satisfied, MGDSE-II will suspend the current optimization, save the next design point as the breaking point and turn to the optimization of the next array to check if there is any possibility of improving the performance further. When MGDSE-II comes back to optimize the previous array, optimization will start from the breaking point and perform the same process until the partitioning pair $(Type, Factor)$ becomes $(cyclic, bound)$, which is the end of optimization and the array partitioning type is denoted as *cyclic* with a factor equal to the array size (i.e., the array is partitioned completely). After one array is optimized, the array will be removed from the optimization list. After all the arrays are optimized, $\texttt{OptimizeArray}(F, arrays)$ will return a high-performance array configuration. The search direction in MGDSE-II is to test the array configurations of different arrays alternately to enable a more efficient search of optimal point, while the previous algorithm [7] is to optimize each array continuously.

VIII. Memory Optimization

In this section, we first present two factors that influence the memory performance. By considering these factors, improper array partitioning schemes could be avoided during the DSE stage. Then we introduce the multi-port memories and related techniques, and demonstrate how to improve memory performance through code transformation.

A. Factors Affecting Memory Performance

1) Multiplexers: Improper array partitioning may introduce a great number of multiplexers, leading to additional delays. For instance, if partitioned completely, arrays will be divided into individual elements and mapped to registers. Then each array element is expected to be accessed at any time in one cycle, eliminating the limitation of insufficient memory ports. However, if a loop accesses one element per iteration, a large number of multiplexers will be generated to select the required data from all the elements loaded, resulting in a much longer latency than expected. In this case, there is no need to partition the array, and a single port is enough to satisfy the requirement.

2) Loop-carried dependence: Improper array partitioning may also incur loop-carried dependence, weakening the positive effects of *loop unrolling* and *loop pipelining*. Given the example in Fig. 8, the iterations are independent in Fig. 8(a) and Fig. 8(b). However, if an improper array partitioning scheme is chosen as in Fig. 8(c), the performance may be degraded due to the loop-carried dependence incurred. This is because, for the loop in Fig. 8(b), all the four banks are accessed per iteration even though each time only two of them provide the required data. The next iteration needs to access the same four banks but has to wait until the last iteration finishes reading/writing the data. Therefore, there exists "dependence" between the two iterations, which is the loop-carried dependence and influences the estimation of II. By considering this impact, the estimation accuracy of the *loop unrolling* and *loop pipelining* models is improved further. Note that it also reveals the close interaction among different directives, and this is why we need to model an application considering the overall effects of various directives.

By considering the impact of improper array partitioning, COMBA eliminates the configurations that degrade the memory performance and explores the design space in a more beneficial direction.

```
void func(int array0[64], int array1[64]){
        for (int i=0; i<64; i++)
            array1[i] = array0[i] * array1[i];
}// the invoked function
```

(a) The invoked function

```
void multi(int x[64], int y[64])
{
    int a[64];
    for(int i=0; i<64; i++)
       a[i]=x[i]+y[i];

    func(a, x);
    func(a, y);
    //access the same array a
}
```

(b) The original code

```
void multi(int x[64], int y[64])
{
   int a[64], b[64];
   for(int i=0; i<64; i++)
     a[i]=x[i]+y[i];
     b[i]=a[i];
     //copy elements from a to b
   func(a, x);
   func(b, y); //access b instead
}
```

(c) The modified code

Fig. 10: Example of code transformation.

B. Multi-Port Memories and Code Transformation

BRAMs on FPGAs typically have two ports, which are not sufficient for memory-intensive applications. Multi-port memories, supporting multiple accesses concurrently, increase the overall performance and are extensively used in soft processors and complex systems-on-chip [26]. Although multi-port memories have been studied deeply at the hardware level [26]–[29], they are only considered partially in the current HLS tools [30]. We introduce multi-port memories into HLS through code transformation which adjusts the original code to a more hardware-friendly form.

It is the most efficient way to implement multi-port memories with BRAMs on FPGAs [29]. Figure 9 shows two conventional techniques for the BRAM-based multi-port memories. *Replication* makes multiple copies of the data set, each of which is stored in a BRAM. For *replication*, each additional read port requires an extra BRAM while the single write port remains unchanged. *Replication* supports multiple reads and data sharing across the ports, but has only one write port and costs multifold resource usage. *Banking* divides the data set and stores different partitions in different BRAMs. The number of the read/write ports increases with the number of 1W/1R BRAM, supporting multiple accesses. However, simultaneous accesses to the same partition are not supported, since each read or write port can only access their corresponding memory partition. Previous HLS-related works focused on *banking* to improve memory performance and ignore the advantages of *replication* which is of great benefit for applications requiring data sharing. We consider both methods and give an example in Fig. 10 to show how code transformation generates different memory architectures to satisfy various requirements.

Figure 10(b) shows the original code without modification. It invokes the function in Fig.10(a) twice to process the data from array x and y. Typically, independent functions execute in parallel. However, the invoked functions in Fig.10(b) have to execute in sequence since they share the data from array a. In this case, *banking* cannot eliminate the function dependence caused by data sharing. To improve the performance, *replication* is utilized and code is transformed to the form in Fig. 10(c). In the modified code, array a and b contain the same data, and the invoked functions execute concurrently. In its RTL code generated by Vivado HLS, the write ports of BRAMs that store a and b, are connected to the same register, while the read ports are connected to different registers, which works exactly like a 2R/1W *replication* type multi-port memory. After code transformation, the latency is reduced by half, but the BRAM usage is doubled compared to the original code. Through code transformation, the performance can be improved in advance before utilizing HLS tools. Combined with COMBA, a better hardware implementation can be generated, achieving more efficient architecture optimization.

IX. Experimental Results

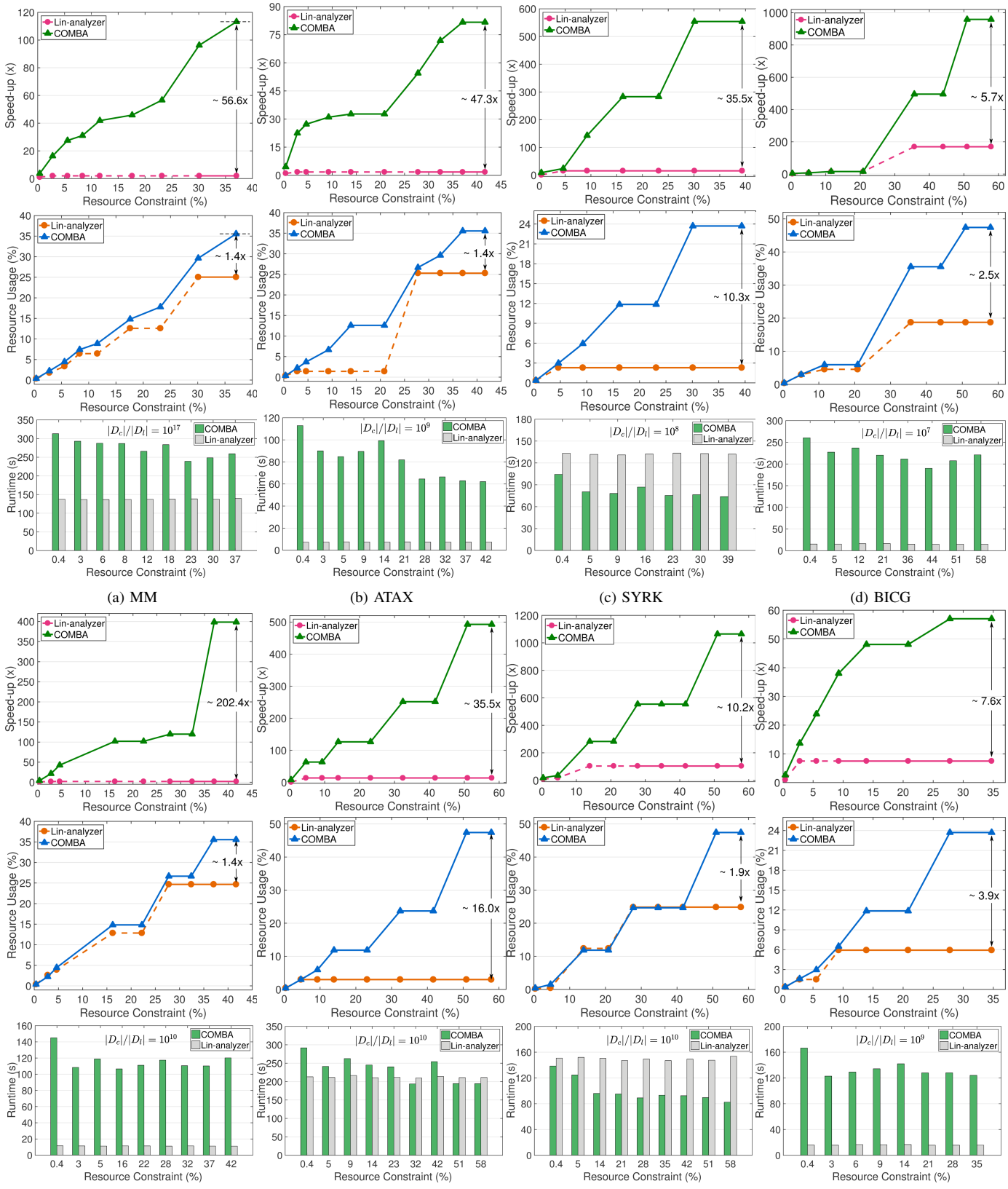
We evaluate COMBA on PolyBench [31], CHStone [14] and the image processing applications used in [12]. COMBA is developed on LLVM 3.4, with Clang 3.4 as the front end, running on CentOS Linux 7.3 with an Intel Core i7-4790 CPU. The target FPGA platform is Virtex-7 XC7V2000T-FLG1925. Vivado HLS v2016.1 is used to evaluate the accuracy of our model and synthesize the configurations found by our tool.

A. Comparison with State-of-the-art

As evaluated in [7], COMBA improves the estimation accuracy compared to the state-of-the-art [6] which generates the execution trace dynamically and estimates the latency based on simple performance models. COMBA conducts analysis at the source code level which covers various code structures. Combined with more comprehensive models, the estimation accuracy has been improved significantly.

In this sub-section, we vary the resource constraints and evaluate COMBA in three different aspects, compared to the state-of-the-art [6], as shown in Fig. 11. By studying the impact of the resource constraints, we can get a better view of the relationship between latency and resource usage. In Fig. 11, we set the resource constraint to different percentages of the total resources, and run our tool, COMBA, and the state-of-the-art, Lin-analyzer [6], given the same resource constraint. The top row of figures for each benchmark shows the latency speed-ups achieved by the configurations returned by the two frameworks, which are computed by comparing the latencies of these configurations with the latency of the baseline (without any directive applied). Since the resource constraint in [6] is used to help schedule the operations and their resource estimation is not very accurate, their tool returns the same configuration given various resource constraints. When the actual resource usage exceeds the resource constraint, the configuration returned will not be feasible. In this case, we test each configuration in their design space in Vivado HLS (around 10^2 points for each application) and select the best configuration with the highest speed-up and effective resource usage, which are represented as the dashed lines in the figures. Compared with [6], COMBA finds configurations with better speed-ups for each application given the same resource constraint. Even when the resource constraints are small, our tool can still find better configurations compared to the dashed lines. By including more directives, considering more diverse code structures and expanding the design space, a broader range of optimization is achieved by COMBA, and configurations with higher performance can be found, improving the application performance significantly. It is notable that when the resource constraints are set to higher values, COMBA will continue to find configurations with higher speed-ups.

Another reason for the performance improvement is that we better utilize the available resources. The middle row of figures for each benchmark in Fig. 11 shows the resource usage of



(a) MM (b) ATAX (c) SYRK (d) BICG

(e) MVT (f) SYR2K (g) GEMM (h) GESUMMV

Fig. 11: Illustration of framework comparison under different resource constraints. For the eight benchmarks, the top row of figures shows the latency speed-ups achieved by the configurations returned by the two frameworks (solid lines). The dashed lines are obtained by testing each configuration of Lin-analyzer through Vivado HLS and selecting the best ones. The middle row of figures presents the resource usage of the corresponding configurations, and the bottom row of figures compares the runtimes of the two frameworks.

the same configurations as in the top row of figures. Note that [6] and our work try to tackle the same resource-constrained optimization problem, which is to optimize the performance given certain resource constraints. What we concern is how to utilize available resources better to obtain a design with higher performance. Compared with [6], we improve the resource utilization rate and the cost is acceptable considering the larger differences between the latency speed-ups achieved by the two frameworks. Taking the MM application as an example, our framework increases the latency speed-up by 56.6 times, while just utilizing 1.4 times more resources.

The bottom row of figures for each benchmark in Fig. 11 compares the runtime, which denotes the time it takes to search the design space and obtain the final configuration. For most applications, the runtime of our framework is longer but acceptable, considering the gap of the design space scale. As discussed in section II-C, our design space is much larger, because we consider more characteristics, such as different kinds of loop structures, multi-dimension arrays, and coupled functions and loops. [6] estimates the latency of a single nested loop without considering other loop structures, like multiple loops. The type of *array partitioning* is also fixed. In contrast, when we consider all the loops within the function, all the dimensions of each array and function-related directives, the design space increases exponentially. The difference in the design space scale is marked on each figure, evaluated by $|D_c|/|D_l|$, where $|D_c|$ and $|D_l|$ are the number of points in the design space of COMBA and Lin-analyzer [6], respectively. For the SYRK and GEMM applications, our runtime is even shorter. This is because we analyze the source code directly and the runtime is the DSE time, while the runtime of [6] includes the time for DSE and profiling. For these applications, the profiling is time-consuming, leading to a longer runtime.

Compared to the state-of-the-art, we conclude that COMBA can efficiently find a better configuration with higher performance within minutes in an exponentially increasing design space, and is more stable given different resource constraints.

B. Evaluation of DSE Algorithms

To evaluate the performance of our DSE algorithm, we compare MGDSE-II and the genetic algorithm (GA) [32] on eleven applications. GA is implemented on DEAP [33] and Python 2.7, and the resource constraints are set to the total resources on FPGA. Based on our accurate estimation models, both algorithms can find high-performance configurations with significant speed-ups as in Fig. 12. Note that GA may generate different results each time, depending on the randomly selected generations, so we run it three times for each application and compute the average. DSE times are compared in Fig. 13.

Compared to GA, the MGDSE-II algorithm achieves 1.2x improvement on speed-ups with around 0.4x time cost. For most applications, the MGDSE-II algorithm outperforms GA and finds better configurations in less time. For some benchmarks, GA finds a higher speed-up, but it is still very close to what MGDSE-II found and its DSE time is much longer. This is because GA randomly generates some points that may be skipped by MGDSE-II, and therefore increases the possibility of achieving better speed-ups for some applications. However,

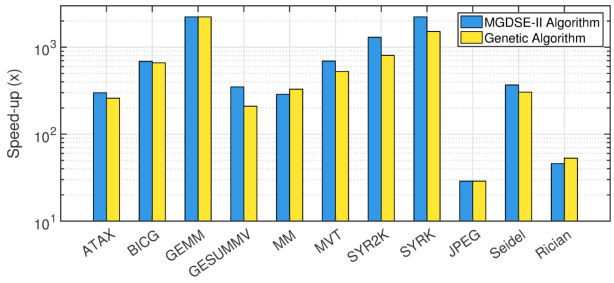


Fig. 12: Comparison of speed-ups as found using different DSE algorithms.

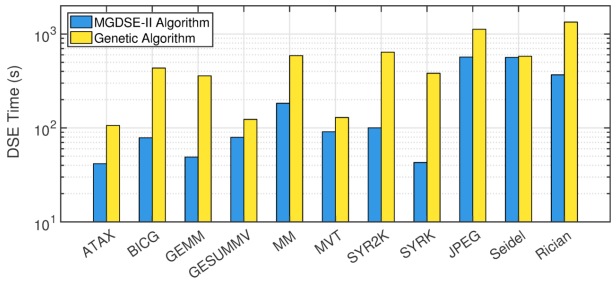


Fig. 13: Comparison of DSE times for different DSE algorithms.

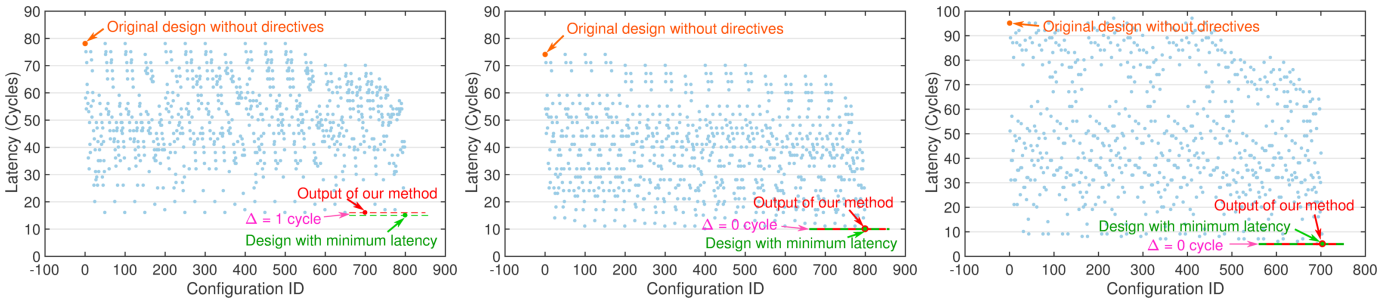
for most applications, the single-objective genetic algorithm only aims at reducing the *latency* without considering the *II*, generating configurations with small *latency* but high *II* and resulting in an incorrect optimization direction. Furthermore, its randomness characteristic brings about uncertainty for the results. We also tried the multi-objective genetic algorithm, but it took an even longer time to find configurations with smaller speed-ups. Although we can set the weights of objectives (*latency* and *II*), it is still difficult to determine the weights for various applications with different code structures. The MGDSE-II algorithm evaluates the performance according to both *latency* and *II*, i.e., reducing *latency* first and then *II* for the same *latency*, and is more stable during the exploration. Also, it does not need to generate many design points like GA and searches the space much more efficiently. The MGDSE-II algorithm also achieves larger speed-ups, compared with our previous MGDSE algorithm in [7]. It is partially because MGDSE-II considers the interplay of arrays, covering more effective design points while eliminating less effective points. Also, we refine our models and enhance the stability of our framework. For example, the BRAM modes impact the *II* estimation. If a pipelined loop contains one read and one write, the smallest *II* will be achieved by mapping the array to a SDP BRAM, which is only one cycle, while in the SP mode, *II* will be two cycles. The refinement of our models, and the more careful consideration such as the impact of improper memory partitioning, help direct MGDSE-II to conduct a more efficient search and find better configurations.

C. Configuration Analysis

The configurations, returned by COMBA in section IX-B, are shown in Table III. The array size denotes the length in each dimension. Column 3 presents whether *dataflow* is applied or not, columns 4 and 5 indicate which function or loop is pipelined, and column 6 shows the unrolling factors of the top loops. The last column presents the configuration of each array with a format of *"array name: (type, factor, dimension)"*. When the top-function is pipelined, the loops cannot

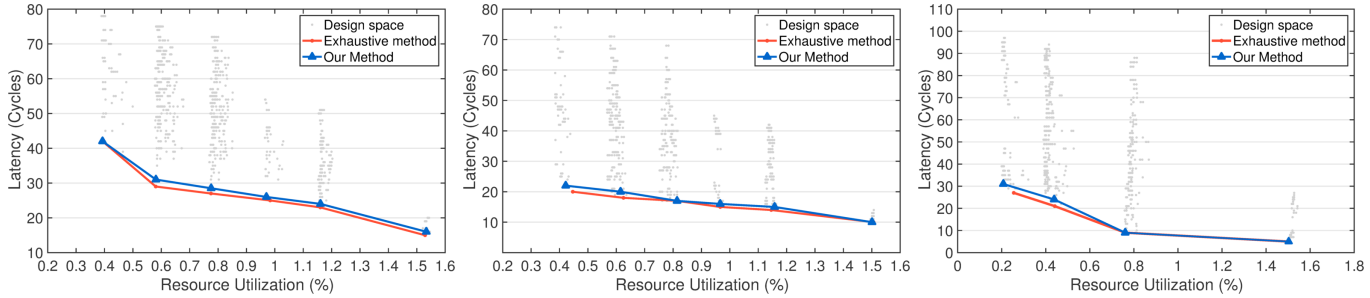
TABLE III: OPTIMIZATIONS OF APPLICATIONS

Benchmark	Array size	Optimizations				
		Dataflow	Function Pipelining	Loop Pipelining	Loop Unrolling	Partition (array name: (type, factor, dimension))
ATAX	16	disabled	top-function	not required	16	A: (complete,16,2) (complete,16,1); x, y, tmp: (complete,16,1)
BICG	32	disabled	disabled	top-loop	8	A: (complete,32,2) (cyclic,4,1); s, p: (complete,32,1); r, q: (cyclic,8,1)
GEMM	16	disabled	disabled	top-loop	2	A, C: (complete,16,2) (cyclic,2,1); B: (block,8,2) (complete,16,1);
GESUMMV	16	disabled	top-function	not required	16	A, B: (complete,16,2) (complete,16,1); x, y, tmp: (complete,16,1)
MM	8	disabled	disabled	top-loops (at the same level)	4, 1	A:(complete,8,2)(cyclic,2,1);B,C:(block,4,2)(complete,8,1);D:(complete,8,2);tmp:(complete,8,2)(cyclic,4,1)
MVT	16	disabled	top-function	not required	16, 16	x1, x2, y_1, y_2: (complete,16,1); A: (complete,16,2) (complete,16,1)
SYR2K	16	disabled	disabled	top-loop	1	A: (cyclic,8,2) (complete,16,1); B: (block,8,2) (complete,16,1); C: (complete,16,2)
SYRK	16	disabled	disabled	top-loop	2	A: (cyclic,8,2) (complete,16,1); C: (complete,16,2) (cyclic,2,1)
JPEG	64	disabled	top-function,sub-functions	not required	8, 8, 64, 64,64, 64, 64	QuantBuff, out_buf, HuffBuff: (complete,64,1)
Seidel	16	disabled	top-function	not required	14, 14	A, B: (complete,16,2) (complete,16,1); C: (complete,16,2) (block,8,1)
Rician	16	enabled	disabled	top-loops (at the same level)	4, 4	v, g: (complete,16,2) (complete,16,1); u: (cyclic,2,2) (complete,16,1)



(a) MM (b) MVT (c) SYRK

Fig. 14: Comparison with the exhaustive search for minimizing the latencies of the modified MM, MVT and SYRK.



(a) MM (b) MVT (c) SYRK

Fig. 15: Comparison with the Pareto-optimal curves of the modified MM, MVT and SYRK.

be pipelined ("not required") and are unrolled completely, such as ATAX. For MM and Rician, there are two top-level loops, and both of them are pipelined. Some applications can be applied with *function pipelining* given available resources, while other applications are improved by optimizing the loops. The array optimization also plays an important role, and how to partition an array in each dimension depends on the memory access pattern and is influenced by other directives. Another interesting thing is that the Seidel and Rician benchmarks have similar code structures with two two-level nested loops and execute in a *producer-consumer* flow. However, the optimization results for both applications are different. The reason is that the second loop in Rician loads elements from and stores results to the same array, leading to loop-carried dependence when it is pipelined. Therefore, the advantage of pipelining weakens and higher performance improvement cannot be achieved. Our framework identifies the bottlenecks of different applications successfully and optimizes the performance as far as possible.

D. Optimality Analysis

In an exponentially increasing design space, it is impossible to obtain the optimal design through the exhaustive search. To analyze the optimality of the results obtained by COMBA, we modify three benchmarks (i.e., MM, MVT and SYRK), reduce the design space by decreasing the number of arrays, loop levels and loop trip counts, and test our framework on the modified applications, compared to the exhaustive method. Figure 14 shows the latencies of all the configurations for the three applications. Each configuration is tested through Vivado HLS, given the same resource constraints. We highlight the configuration returned by COMBA and the optimal configuration with the minimum latency returned by exhaustive search, and compute their latency difference ($\Delta Latency$) as illustrated in Fig. 14 and Table IV. We can see that COMBA finds a configuration with a small latency which is exactly or very close to the optimal configuration obtained by exhaustive search, given the same resource constraints. Moreover, as shown in Table IV, it takes several hours to exhaustively test each configuration through Vivado HLS, while COMBA efficiently finds the high-performance configuration in a few seconds.

TABLE IV: COMPARISON WITH EXHAUSTIVE SEARCH

Benchmark	Design Space	$\Delta Latency$ (*clock cycles*)	DSE Time	
			Exhaustive (*hours*)	COMBA (*seconds*)
MM	800	1	14	6.01
MVT	800	0	20	15.81
SYRK	704	0	13	11.94

The comparison shows that COMBA efficiently solves the resource-constrained performance optimization problem and finds a high-performance configuration which is near-optimal.

By varying the resource constraints, we analyze the trade-off relationship between performance and area and obtain a series of configurations through our framework, as shown in the blue lines in Fig. 15. The red lines are the Pareto-optimal curves of different applications returned by the exhaustive method, and the resource utilization rate is the percentage of resource usage. We can see that our framework successfully predicts the trend of the Pareto-optimal curve and generates a very close approximate curve, showing the trade-off between *latency* and *resource usage*. The reason why the two curves do not overlap exactly is that there are estimation errors introduced by the prediction models. We measure the quality of our approximate curve through the metric *average distance from reference set* (ADRS) [12, 34, 35], which is to evaluate how close the approximate curve is to the Pareto-optimal curve. The lower ADRS, the better the quality of the approximate curve. The values of ADRS are 4.6%, 5.8% and 7.3% for the modified MM, MVT and SYRK, respectively, showing that the approximate set of Pareto-optimal designs obtained by our framework is of high quality.

X. Related Works

OpenCL-based performance models are proposed in [5, 17, 36]. Wang et al. [17] propose a performance analysis framework for OpenCL programs and guides users to optimize the code step by step. Wang et al. [5] and Liang et al. [36] propose more accurate OpenCL-based performance models and their framework supports automatic design space exploration. However, due to the highly parallel OpenCL execution model, their estimation models focus on pipelining and parallelism analysis, making them incapable of modeling the performance of applications specified in sequential languages like C/C++.

Of C-based works, [37] proposes a power-performance simulator for ASICs to represent accelerators and utilizes the dynamic data dependence graph to analyze the performance, while our focus is on enabling HLS tools in generating correct and actual hardware for FPGA-based accelerators through static analysis. Targeting reconfigurable architectures, [38] analyzes the area/delay trade-off through a hierarchical exploration and performance metrics are estimated by the HLS tool. [39] proposes a HLS synthesis flow which enables a rapid design space exploration considering loop unrolling. [12] synthesizes a number of design points and models them in a linear model based on loop-hierarchy-related observations. All of these works only consider loop unrolling in a limited design space and have to invoke HLS tools for estimation. More synthesis directives are considered in [6, 9, 40]–[43]. Specifically, [40] and [41] cluster the operations and explore each cluster separately, investigating the trade-off between the exploration speed and the optimality of results. [42] performs the DSE with a lattice representation of the design space, based on the observation of the low variance among Pareto-optimal configurations. [9] prunes the design space by exploiting loop-array dependencies. However, the invoking of HLS tools in these works increases the DSE time to many hours. [6] proposes analytical models for estimation without running HLS tools, but it restricts itself to *loop unrolling, loop pipelining and array partitioning* only, considers simple loop hierarchies and optimizes one-dimension arrays, which is not enough for complex applications. [43] proposes an accelerator template for cacheable applications to bound the design space and performs DSE to generate the optimal configuration of their template, while our framework optimizes original HLS designs, considers function-related optimization and is not constrained by the application types. Learning-based methods are also proposed in [35, 44, 45]. Specifically, [35] proposes a learning-based model to estimate the latency/area trade-off in a small design space, leading to a long time for training. [44] presents a regression model to estimate the resource usage of LUTs and FFs on FPGAs but considers limited directives and exhaustively explores the design space with a few dozen points. [45] separates the design space into independent partitions and adopts reinforcement learning algorithms in DSE for different partitions on different cores. The DSE time is reduced to one or two hours, but it is still not very efficient due to the invoking of HLS tools.

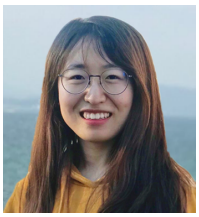
XI. Conclusion

This paper presents a comprehensive model-based analysis framework for high-performance synthesis of complex applications with various code structures. This demonstrates the importance and the significant utility of our framework in making HLS more practical and useful. Experiments show that our framework outperforms the previous work with more accurate models and more powerful DSE algorithm, and optimizes various applications rapidly, within minutes. Our tool is available at http://www.ece.ust.hk/~eeweiz/tools.html.

References

[1] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.
[2] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015, pp. 161–170.
[3] Xilinx, "Vivado design suite user guide high-level synthesis v2016.1."
[4] Intel, "Intel HLS compiler product brief."
[5] S. Wang, Y. Liang, and W. Zhang, "Flexcl: An analytical performance model for opencl workloads on flexible fpgas," in *DAC*, 2017, pp. 1–6.
[6] G. Zhong *et al.*, "Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators," in *DAC*, 2016, p. 136.
[7] 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 *ICCAD*, 2017, pp. 430–437.
[8] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop splitting for efficient pipelining in high-level synthesis," in *FCCM*, 2016.
[9] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin, "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis," in *DATE*, 2015, pp. 157–162.
[10] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *TODAES*, vol. 16, no. 2, p. 15, 2011.
[11] A. Cilardo and L. Gallo, "Interplay of loop unrolling and multidimensional memory partitioning in hls," in *DATE*, 2015, pp. 163–168.
[12] G. Zhong *et al.*, "Design space exploration of multiple loops on fpgas using high level synthesis," in *ICCD*, 2014, pp. 456–463.
[13] Xilinx, "7 series FPGAs memory resources user guide v1.12."
[14] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *JIP*, vol. 17, pp. 242–254, 2009.
[15] Clang, https://clang.llvm.org.
[16] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *DAC*, 2006, pp. 433–438.

[17] Z. Wang *et al.*, "A performance analysis framework for optimizing opencl applications on fpgas," in *HPCA*, 2016, pp. 114–125.
[18] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *TPDS*, vol. 7, no. 5, pp. 506–521, 1996.
[19] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *FPGA*, 2015, pp. 200–209.
[20] T. M. Lattner, "An implementation of swing modulo scheduling with extensions for superblocks," Ph.D. dissertation, Citeseer, 2005.
[21] X. Gao *et al.*, "Automatically optimizing the latency, area, and accuracy of c programs for high-level synthesis," in *FPGA*, 2016, pp. 234–243.
[22] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *MICRO*, 1994, pp. 63–74.
[23] C.-C. Shen *et al.*, "Dataflow-based design and implementation of image processing applications," Tech. Rep., 2011.
[24] J. A. Bingham, "Multicarrier modulation for data transmission: An idea whose time has come," *IEEE Communications magazine*, 1990.
[25] S. Sinha *et al.*, "Dataflow graph partitioning for area-efficient high-level synthesis with systems perspective," *TODAES*, p. 5, 2014.
[26] G. A. Malazgirt *et al.*, "Application specific multi-port memory customization in fpgas," in *FPL*, 2014, pp. 1–4.
[27] E. R. R. Charles Eric LaForest, Ming G. Liu and J. G. Steffan, "Multi-ported memories for FPGAs via XOR," in *FPGA*, 2012, pp. 209–218.
[28] K. R. Townsend, O. G. Attia, P. H. Jones, and J. Zambreno, "A scalable unsegmented multiport memory for fpga-based systems," *International Journal of Reconfigurable Computing*, vol. 2015, p. 11, 2015.
[29] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *FPGA*, 2010, pp. 41–50.
[30] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *FPGA*, 2014.
[31] L. Pouchet, "Polybench/c 4.2."
[32] D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning, 1989," *Reading: Addison-Wesley*, 1989.
[33] F.-A. Fortin *et al.*, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.
[34] C. S. Palermo, Gianluca and V. Zaccaria, "Respir: a response surface-based pareto iterative refinement for application-specific design space exploration," *TCAD*, vol. 28, no. 12, pp. 1816–1829, 2009.
[35] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *DAC*, 2013, p. 50.
[36] Y. Liang *et al.*, "Flexcl: A model of performance and power for opencl workloads on fpgas," *IEEE TC*, vol. 67, no. 12, pp. 1750–1764, 2018.
[37] Y. S. Shao *et al.*, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ACM SIGARCH Computer Architecture News*, 2014.
[38] S. Bilavarn, G. Gogniat, J.-L. Philippe, and L. Bossuet, "Design space pruning through early estimations of area/delay tradeoffs for fpga implementations," *TCAD*, vol. 25, no. 10, pp. 1950–1968, 2006.
[39] A. Prost-Boucle, O. Muller, and F. Rousseau, "A fast and autonomous hls methodology for hardware accelerator generation under resource constraints," in *DSD*, 2013, pp. 201–208.
[40] B. C. Schafer and K. Wakabayashi, "Design space exploration acceleration through operation clustering," *TCAD*, pp. 153–157, 2010.
[41] B. C. Schafer and K. Wakabayashi, "Divide and conquer high-level synthesis design space exploration," *TODAES*, p. 29, 2012.
[42] L. Ferretti, G. Ansaloni, and L. Pozzi, "Lattice-traversing design space exploration for high level synthesis," in *ICCD*, 2018, pp. 210–217.
[43] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *DAC*, 2018, pp. 1–6.
[44] G. Zhong *et al.*, "Design space exploration of fpga-based accelerators with multi-level parallelism," in *DATE*, 2017, pp. 1141–1146.
[45] C. H. Yu, P. Wei, M. Grossman, P. Zhang, V. Sarker, and J. Cong, "S2fa: an accelerator automation framework for heterogeneous computing in datacenters," in *DAC*, 2018, p. 153.



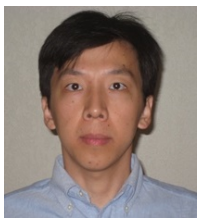
Jieru Zhao received her B.S degree in microelectronics from Nanjing University, Nanjing, China, in 2015. She is currently pursuing the PhD degree with the Department of Electronic and Computer Engineering, the Hong Kong University of Science and Technology, Hong Kong. Her current research interests include high level synthesis, reconfigurable computing, and electronic design automation (EDA).

Liang Feng received his B.S. degree in microelectronics from Nanjing University, Nanjing, China, in 2014. He is currently a PhD student in the Department of Electronic and Computer Engineering, the Hong Kong University of Science and Technology, Hong Kong. Liang's research interests include reconfigurable computing, multi-core system and electronic design automation (EDA).

Sharad Sinha (S03, M14) is an Asst. Prof. with Dept. of Computer Science and Engineering, Indian Institute of Technology (IIT) Goa. Previously, he was a Research Scientist at NTU, Singapore. He has received the Best Paper Award at ICCAD 2017 and the Best Speaker Award from IEEE CASS Society, Singapore Chapter, in 2013 for his PhD work on High Level Synthesis. His research and teaching interests are in computer architecture, embedded systems and reconfigurable computing.

Wei Zhang received the B.S and M.S degrees from Harbin Institute of Technology, Harbin, China, in 1999 and 2001, respectively, and the Ph.D. degree from Princeton University, Princeton, NJ, USA in 2009. She is currently an Associate Professor with the Department of Electronic and Computer Engineering, the Hong Kong University of Science and Technology, Hong Kong, where she established the Reconfigurable System Laboratory. She was an Assistant Professor with the School of Computer Engineering, Nanyang Technological University, Singapore, from 2010 to 2013. She authored over 80 technical papers in referred international journals and conferences, and authored three book chapters. Her current research interests include reconfigurable system, power and thermal management, embedded system security, and emerging technologies.

Dr. Zhang currently serves as an Area Editor of Reconfigurable Computing of the ACM Transactions on Embedded Computing Systems (TECS), Associate Editor of the IEEE Transaction on Very Large Scale Integration (TVLSI) Systems, and Associate Editor of the ACM Journal on Emerging Technologies in Computing Systems (JETC). She also serves on many organization committees and technical program committees, including DAC, ICCAD, ASPDAC, CASES, FPL, etc.



Yun (Eric) Liang is an associate professor in School of EECS, Peking University, China. He received his PhD in Computer Science from National University of Singapore in 2010 and worked as a Research Scientist in UIUC before he joins PKU. His research focuses on heterogeneous computing (GPUs, FPGAs, ASICs) for emerging applications such as AI and big data, computer architecture, compilation techniques, programming model and program analysis, and embedded system design. He has authored over 70 scientific publications in premier international journals and conferences in related domains. His research has been recognized by best paper award at FCCM 2011 and ICCAD 2017 and best paper nominations at DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, CODES+ISSS 2008. Prof Liang serves as Associate Editor for ACM Transactions in Embedded Computing Systems (TECS) and serves in the program committees in the premier conferences in the related domain including (HPCA, PACT, CGO, ICCAD, ICS, CC, DATE, CASES, ASPDAC, ICCD).

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing, National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.