COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications

Jieru Zhao*, Liang Feng*, Sharad Sinha†, Wei Zhang*, Yun Liang§ and Bingsheng He‡
*ECE Department, Hong Kong University of Science and Technology; †SCSE, Nanyang Technological University
§School of EECS, Peking University; ‡Department of CS, National University of Singapore
{jzhaoao,lfengad}@connect.ust.hk, sharad_sinha@ieee.org, wei.zhang@ust.hk,
ericlyun@pku.edu.cn, hebs@comp.nus.edu.sg

***Abstract*—High Level Synthesis (HLS) relies on the use of synthesis pragmas to generate digital designs meeting a set of specifications. However, the selection of a set of pragmas depends largely on designer experience and knowledge of the target architecture and digital design. Existing automated methods of pragma selection are very limited in scope and capability to analyze complex design descriptions in high-level languages to be synthesized using HLS. In this paper, we propose COMBA, a comprehensive model-based analysis framework capable of analyzing the effects of a multitude of pragmas related to functions, loops and arrays in the design description using pluggable analytical models, a recursive data collector (RDC) and a metric-guided design space exploration algorithm (MGDSE). When compared with HLS tools like Vivado HLS, COMBA reports an average error of around 1% in estimating performance, while taking only a few seconds for analysis of Polybench benchmark applications and a few minutes for real-life applications like JPEG, Seidel and Rician. The synthesis pragmas recommended by COMBA result in an average 100x speed-up in performance for the analyzed applications, which establishes COMBA as a superior alternative to current state-of-the-art approaches.**

I. Introduction

FPGAs are attracting increasing attention to accelerate a wide variety of applications, such as data center [10] and deep learning [18]. Due to their reconfigurability and energy efficiency, FPGAs can speed up system performance significantly with low energy consumption. However, 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.

To improve programmability on FPGAs, the High Level Synthesis (HLS) methodology, which automatically transforms the behavioral description specified in high-level languages (e.g., C, C++, SystemC) to RTL-level design, has been developed. Due to its potential to accelerate application development, several FPGA vendors, such as Xilinx and Altera, have released HLS tools. However, the efficiency and quality of the resulting RTL designs largely depend on the configuration of the optimization pragmas provided by HLS tools [17]. Significant speed-up can be achieved with properly chosen pragmas, while improper selection can worsen design performance and resource utilization. Therefore, an optimal configuration of pragmas which maximizes the performance under limited resources is highly beneficial. However, finding such a configuration is non-trivial given the multiple optimization pragmas and exponentially increasing design space.

Several prior works have been presented to analyze the performance of applications under pragmas and explore the design space to find a high-performance configuration. In [14, 15], an analytical model is proposed, but it focuses on OpenCL programs, which is different from direct synthesis of C/C++ applications. C-based works [8, 19, 20] propose performance models under limited optimization pragmas, which is not sufficient for real applications. With coupled functions and loops, as well as multi-dimension arrays in real applications, only optimizing one loop and partitioning one array [20] is insufficient to maximize performance. Also, more factors should be considered to improve the prediction accuracy.

To this end, we propose COMBA, a comprehensive model-based analysis framework, including a pluggable analytical model, a recursive data collector (RDC) and a metric-guided design space exploration (MGDSE) algorithm. The proposed model considers seven optimization pragmas, loop unrolling, loop pipelining, array partitioning, function pipelining, dataflow, loop flattening and function inlining, covering more pragmas than previous works. It also estimates the performance of C/C++ applications more accurately by considering diverse code structures as well as memory access conflict in real applications. The RDC computes the required parameters for the model, which can support a rich set of code structures in C/C++ and achieve cycle-level accuracy by considering operation chaining. With more complex code structures and more optimization pragmas, the design space increases exponentially and the brute-force method in previous papers [14, 20] cannot work. Therefore, we propose a two-stage MGDSE algorithm and three evaluation metrics to prune (first stage) and explore (second stage) the design space.

Our framework can model the performance closely compared to Vivado HLS and find a high-performance pragma configuration with an average 100x speed-up within minutes in an exponentially increasing design space. Users can directly utilize the configuration in an HLS tool like Vivado HLS to obtain the corresponding hardware implementation.

II. Related Works

Existing performance estimation and DSE-related methods have been proposed in [7, 8, 12, 14, 15, 19, 20]. In [14, 15], OpenCL-based performance models are proposed, but not C-based models. Due to the highly parallel OpenCL execution

TABLE I
CONFIGURATION OF PRAGMAS

Pragma	Configuration
Loop unrolling	Unrolling factors
Loop pipelining	Enabled/Disabled
Array partitioning	Block/Cyclic/Complete
Function pipelining	Enabled/Disabled
Dataflow	Enabled/Disabled
Loop flattening	Yes/No
Function inlining*	Yes/No

model, their models focus on pipelining and parallelism analysis, making them incapable of modeling the performance of applications specified in sequential languages. Shao et al., in [12], propose a pre-RTL power-performance model for ASICs, not FPGA-based accelerators. Their resource model is not applicable to the resources on FPGAs (DSPs, BRAMs).

Of C-based works targeting FPGAs, [19, 20] propose analytical performance models but only focus on limited optimization pragmas without considering function related optimization. Specifically, [19] synthesizes a number of design points by invoking HLS tool, and then models them in a linear model based on loop hierarchy related observations, which is only used for nested loops and small benchmarks. [20] restricts itself to *loop unrolling, loop pipelining and array partitioning* only, considers simple loop hierarchies and optimizes only one nested loop. The *loop unrolling* model in [20] estimates the loop latency based on the inner loop, ignoring the logic outside the inner loop but within the outer loop. Its *loop pipelining* model doesn't consider the effects of memory conflict, and the *array partitioning* model only focuses on one-dimension arrays. Furthermore, [20] utilizes the brute-force method to explore the design space with only a few dozen points. [7] proposes a learning-based model to explore the design space, requiring more time than analytical models. In [8] the design space is pruned by exploiting loop-array dependencies, but the invoking of HLS tools increases the DSE time to many hours.

In order to cover more complex designs and target real applications, COMBA supports seven optimization pragmas, as shown in Table I. *Function inlining* is an optional pragma supported in COMBA. Users can decide whether to inline functions or not according to their own needs.

III. MOTIVATION AND OVERVIEW

A. Motivation

Real applications contain complex code structures with coupled functions and loops, and multi-dimension arrays. Figure 1 shows the *"decode_block"* kernel in JPEG application [3], 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. We compare loop pipelining (LP), loop unrolling (LU) and function pipelining (FP), combined with array partitioning for corresponding arrays. Both LU and FP improve the performance significantly if a beneficial array configuration is chosen, while LP plays a minor role. Specifically, FP further improves *throughput* compared with LU. Therefore, function-related pragmas, like FP, which are not considered by previous works, are of great importance in real applications. Loop-related pragmas, such as LP and LU, have different effects on the performance, depending on the specific loop structures. Only considering one nested loop, as in [20], doesn't apply to other loop structures and can't reveal the relationship among loops. Moreover, the configuration of arrays influences the performance a lot. To this end, a comprehensive analytical model is required to consider all the pragmas above, as well as the interactions among them, for complex applications.

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

Fig. 1. Motivation example

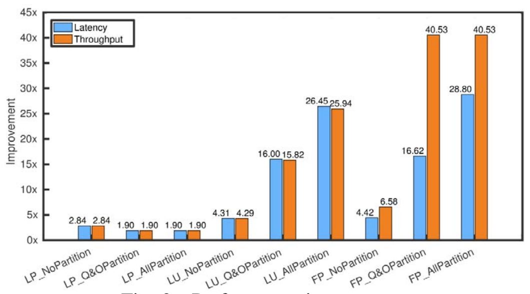


Fig. 2. Performance improvement

Considering more pragmas identifies the characteristics of applications and further improves the performance, but this leads to a large design space. In the same example, the functions may or may not be pipelined and the *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. Therefore the design space contains $2 * 2^6 * 12^3 * 2^7 * 4^2 * 7^5$ points, which is so large that invoking HLS tools to test each configuration is infeasible and the speed of DSE needs to be faster.

B. Framework Overview

The overall framework is shown in Fig.3. The input is a C/C++ application, and the output is the optimal configuration (i.e., the setting of pragmas) with high performance under given resource constraints. First, the C/C++ specification is translated into LLVM IR via the Clang front end. Next, the LLVM IR is sent to the RDC to compute the parameters required by our analytical model according to the pragma setting and profiling library. The profiling library has information about operators for different design frequencies. With the parameters from the RDC, the proposed model estimates the performance and resource usage for the corresponding configuration. Finally, the MGDSE 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 performance and resource 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

With the LLVM IR generated by the Clang front end, the RDC builds the data flow graph (DFG) and computes the data statistics required by our model. The LLVM IR is the

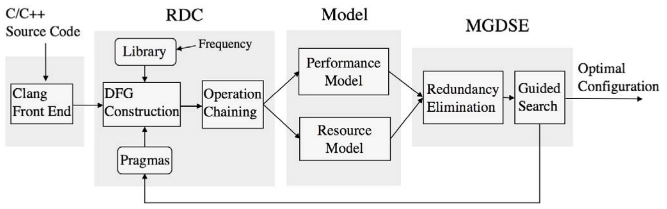
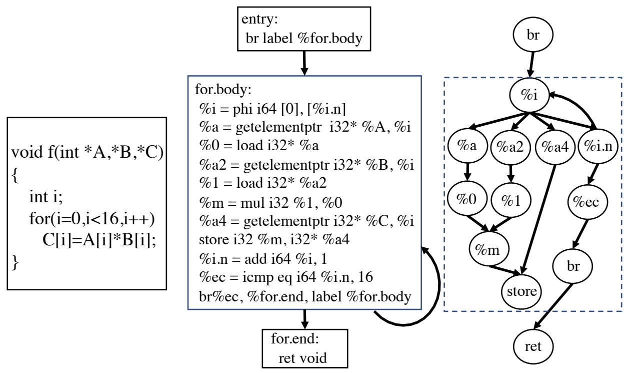


Fig. 3. Framework overview



(a) Source code (b) Control flow graph (c) Data flow graph

Fig. 4. Control flow graph and data flow graph

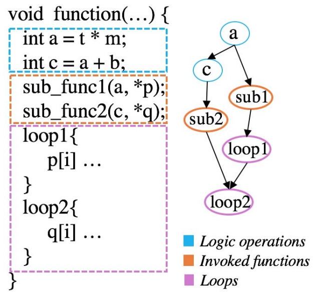
LLVM intermediate representation, which is a Static Single Assignment (SSA) based representation and allows efficient analysis through LLVM passes. It is represented as a Control and Data Flow Graph (CDFG), with separate control flow and data flow graphs combined to represent an application, as shown in Fig.4. There are three basic blocks with multiple assembly level instructions in the CFG shown in Fig.4(b), for which the relationship between instructions is specified in the DFG, as shown in Fig.4(c). The generated DFG covers a rich set of code structures like nested loops, function calls and if-else and switch branches.

The RDC is implemented as an LLVM pass based on *llvm::Module* class, and analyzes the LLVM IR to compute the required parameters. The parameters are divided into two categories: the static information, e.g., the loop bound of each level loop or 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 from the LLVM IR directly, while dynamic information depends on the code structure and optimization pragmas applied, and is computed using the DFG.

1) DFG Construction: The DFG is constructed by connecting dependent instructions and storing each instruction as a node, with its latency as the node weight. A dynamic programming (DP) approach [4, 15] is then employed to trace each path between two dependent instructions and calculate the latency between them. The latency in the critical path can then be computed based on these latencies. 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, as shown in Fig.6.

```
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. 5. Example of load/store latency



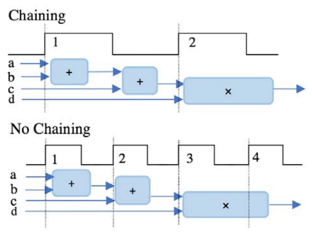


Fig. 6. Function DFG

Fig. 7. Operation chaining

2) Node Weight Setting: We obtain the latency of each instruction by testing microbenchmarks and provide a profiling library for five commonly used frequencies, namely, 100MHz, 125MHz, 150MHz, 200MHz and 250MHz. Specially, the latency of load/store instructions depends on two factors. First, it depends on whether the memory ports are available. If available, the load latency is two cycles (generating an address in one cycle then reading the data in the next) and the store latency is one cycle. If not, the delay caused by other load/store instructions should be added to get the actual latency. Second, it depends on pragmas. In the example in Fig.5, if both loops are unrolled completely, the second loop can access the result of the first loop directly and doesn't need to load $b[i]$ again. Therefore, the latency of load $b[i]$ is set to zero.

3) Operation Chaining: To improve the estimation accuracy, *"operation chaining"* is considered, which means that more than one operation can be scheduled in one cycle if possible, as shown in Fig.7. Profiling-based information is used to decide on the usage of operation chaining under the five design frequencies considered, and this is used to modify the weight of each node when constructing the DFG.

V. Models in COMBA

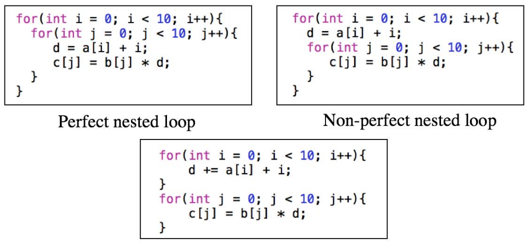
A. Performance Model

In this section, we present the details of the performance model according to five frequently-used optimization pragmas, namely, loop unrolling, loop pipelining, array partitioning, function pipelining and dataflow.

1) 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 correspondingly. The loop latency is estimated in a recursive way using Eq.1:

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

where L_k is the loop in level k; L_{k+1} in level $k+1$ 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



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

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;
   }
}
```

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;
}
```

Multiple loops

Fig. 8. Three kinds of loops

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 [17]. 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. $C^{U_k}_{L_k \setminus L_{k+1}}$ 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 the RDC.

The initial state of the recursion, called as "recursion basis", is the iteration latency ($C^{U_r}_{L_r}$) of L_r, which is not unrolled completely with inner loops $\{L_{r+1}, \ldots, L_n\}$ unrolled completely. $C^{U_r}_{L_r}$ and the loop bounds are returned by the RDC, and the unrolling factors come from the pragma setting. Eq.1 works for all loop hierarchies, including perfect (only innermost loop has contents) and non-perfect (with logic specification between loop levels) nested loops, and multiple loops (more than one loop in the same loop level), as shown in Fig.8. For multiple loops, $C^{U_{k+1}}_{L_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}}$ becomes $\sum_{j=0}^{m} C^{U_{k+1,j}}_{L_{k+1,j}} \cdot \frac{B_{k+1,j}}{U_{k+1,j}}$ to add the latency of loops in the same level $k+1$.

The latency of the loop in level k can then be computed according to Eq.2. $Cycle_{L_k}$ is the latency of L_k:

$$Cycle_{L_k} = C^{U_k}_{L_k} \cdot \frac{B_k}{U_k}. \tag{2}$$

2) Loop Pipelining: Parts of the operations from different iterations can overlap to achieve parallelism using loop pipelining. The three factors of the pipelined model are pipeline depth, initiation interval and trip count [6]. The trip count depends on whether the loop is perfect or not. In a perfect nested loop, when the inner loop is pipelined, outer loops without unrolling can be flattened to feed the inner loop with new data and form a deeper pipeline to improve the overall throughput. The trip count is the multiplication of each loop's trip count, as shown in Eq.3. For non-perfect loops, the trip count is equal to $\frac{B_i}{U_i}$ (the trip count of L_i), as shown in Eq.4. i is the level of the loop that 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 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.

Next we'll demonstrate the estimation of II_i and D_i. II_i is the latency between the initiation of two consecutive iterations.

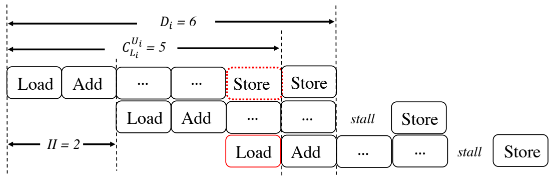


Fig. 9. Pipeline depth

Minimal II_i is constrained by available resources and the loop-carried dependence [5, 16], shown as Eq.5. $II^{res}_{i,\min}$ and $II^{rec}_{i,\min}$ are resource-constrained and 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.

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

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

where $II_{sub,\max}$ is the maximum initiation interval among all the sub-functions within L_i, i.e.,$\max\limits_{sub}(II_{sub})$. When estimating $II^{res}_{i,\min}$, we assume operators are sufficient [2] and $II^{res}_{i,\min}$ is constrained by memory operations and bandwidth. Vivado HLS maps arrays to single-port/dual-port RAMs, depending on the actual needs. However, the limited ports still constrains $II^{res}_{i,\min}$ in a load/store intensive algorithm, shown as Eq.7:

$$II^{res}_{i,\min} = \max_m \left(\left\lceil \frac{Access_m}{Ports_m} \right\rceil \right), \tag{7}$$

where $Ports_m$ is the number of ports and $Access_m$ is the number of accesses to array m, adding both read and write operations together considering load/store conflict. If array m is partitioned, $Ports_m$ and $Access_m$ become the number of ports and accesses to corresponding partitions, respectively.

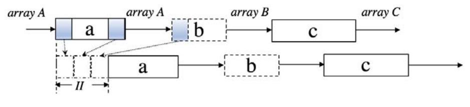
$II^{rec}_{i,\min}$ is computed as Eq.8 [11]:

$$II^{rec}_{i,\min} = \max_p \left(\left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right), \tag{8}$$

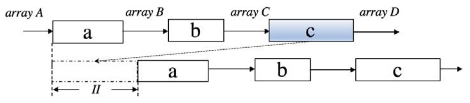
where $Delay_p$ is the latency between a pair (p) of dependent instructions from different iterations and $Distance_p$ is the subtraction of the corresponding iteration numbers.

D_i is related to $C^{U_i}_{L_i}$. If load and store operations access different arrays, D_i equals $C^{U_i}_{L_i}$; otherwise D_i is computed as $D_i = \lceil C^{U_i}_{L_i} / II_i \rceil \cdot II_i$, considering load/store conflict. For example, in Fig.9, an array element is loaded in the first cycle and stored in the same array in the fifth cycle after three logic operations in one iteration, and $C^{U_i}_{L_i}$ is five cycles. 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 $\lceil \frac{5}{2} \rceil \cdot 2 = 6$.

3) Array Partitioning: Memory operations are often the performance bottleneck in real applications. To increase local memory bandwidth, Vivado HLS allows arrays to be partitioned into smaller ones in different dimensions, providing three options: *block*, *cyclic* and *complete* [16]. COMBA supports multi-dimension array partitioning with the same options as Vivado HLS. By tracking each load/store node in the data flow and finding the address index of the accessed element, RDC calculates which partition P it is located in using Eq.9,



(a) Function pipelining



(b) Dataflow

Fig. 10. Difference between function pipelining and dataflow

and then checks whether this partition's ports are available to compute the load/store latency. The partition number in one dimension is calculated in Eq.9a(*block*) and Eq.9b(*cyclic*). For *complete* option, it is calculated by setting $f_i = size_i$ in both equations. We extend it to multiple dimensions in Eq.9c.

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

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

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

where in dimension i, P_i is the partition number, $index_i$ is the address index of the array element, $size_i$ is the number of elements and f_i is the array partitioning factor. P is the partition number considering n-dimension array partitioning.

4) Function Pipelining: Real applications often contain multiple functions. As such, *function pipelining* is critical to improve the performance by allowing concurrent execution of operations within a function. Vivado HLS unrolls all sub-loops completely and pipelines each sub-function inside a pipelined function. Based on this feature, *throughput* is utilized to evaluate the benefit of this pragma. It measures the amount of function outputs per cycle and is calculated as $\frac{1}{II}$. *II* is the initiation interval of a pipelined function as follows:

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

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 sub-functions and logic surrounding sub-functions, then dividing by the number of memory ports. In the example in Fig.10(a), there are two sub-functions, a and c, and one sub-loop, b, within the top-function. Loop b is unrolled completely and becomes logic between a and c. A, B and C are arrays and the numbers of their memory operations are counted. If the counted number for array A is the largest, including reads, writes in sub-function a and reads in logic b, it will be then divided by the number of ports to compute $II_{\min}^{res}$.

5) Dataflow: Dataflow is a "coarse grain" pipelining (task-level) between sub-functions and sub-loops. Unlike function pipelining, it doesn't require sub-functions to be pipelined and sub-loops to be unrolled, but this technique can only be applied to functions or loops at the top level. Also, it aims at applications with *"data flow"* characteristics, i.e., the output of one function/loop is the input of the next function/loop and the whole application works in a flow, like the example in Figure 10(b). Many real applications can benefit from this pragma, such as applications in the image processing and data transmission domains. We also use *II* to evaluate 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 between its sub-elements. If sub-element i is not pipelined, the initiation interval II_i^{sub} is equal to the latency of sub-element i.

B. Resource Model

Resource modeling focuses mainly on the usage of DSP and block-RAM(BRAM), which are the performance bottlenecks in most designs [14, 20].

1) DSP Estimation: We obtain the resource usage of different operators by testing microbenchmarks (e.g., a 32-bit floating point multiplication is mapped to a floating point unit with three DSPs). Then we consider resource sharing according to the kinds of operators. For LUT-based and small bandwidth operations, resource sharing incurs large resource usage due to the multiplexers introduced [13]. Therefore they are not shared, and the number of operations equals the number of instances. Conversely, DSP-based operators are sharable with higher efficiency and larger area, and the number of operations is the maximum number of operators executing in parallel. Specifically, for sharable operators, if a loop is pipelined, we compute the lower bound (i.e., the number of instances that must be allocated) to estimate the resource usage according to $N_{\min}^{op} = \lceil \frac{N_{op}}{II} \rceil$ [2], where N_{op} is the number of operation *op* used in one iteration, and *II* is the initiation interval of the loop. For example, *II* is 2 and four floating point adders are used in one iteration. Therefore, at least two floating point adders are needed for this loop.

2) 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 18K-bit or 36K-bit 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.12 and add them together as the total memory usage:

$$R_{bram} = \left\lceil \frac{\#bits}{width} \right\rceil \cdot \left\lceil \frac{\#element}{depth} \right\rceil \cdot \#partition \cdot d, \tag{12}$$

where R_{bram} is the number of blocks, *#bits* is the width of each array element (e.g., the int data type is 32 bits wide), *#element* is the number of elements per memory partition (i.e., $\lceil \frac{array\_size}{\#partition} \rceil$), and *width* and *depth* are the bandwidth 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 (512×32), the selected configuration of an 18Kb block RAM is 512×36 for single-port and simple dual-port modes (one block is sufficient), and is $1k \times 18$ for the true dual-port 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 places channels between sub-elements to maintain the data rate. 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 is 1 if it is not applied.

VI. Metric-Guided Design Space Exploration

In this section, we present the MGDSE algorithm, which is designed to explore a large design space efficiently. Specifically, the algorithm begins with the default setting (without pragmas) and conducts a two-step exploration, namely, redundancy elimination and guided search.

1) Redundancy Elimination: The first stage is to remove the redundant design points. For example, in Vivado HLS, when a function or loop is pipelined, the sub-loops are unrolled completely and cannot be pipelined. Therefore, no matter what unrolling factor is set and whether sub-loops are pipelined, the performance remains the same. Another example is of perfectly nested loops. When the inner loop is not unrolled completely, unrolling the outer loop cannot improve the performance because Vivado HLS generates copies of the inner loop and schedules them in sequence. After removing these redundant points, which are not promising to improve performance, the design space is reduced significantly without sacrificing the quality of the search space.

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} is the difference in latency between the longest sub-element ($C^{sub}_{\max}$) and the second-longest sub-element ($C^{sub}_{\mathrm{s,max}}$) in the target function, as shown in Eq.13:

$$M_{diff} = C^{sub}_{\max} - C^{sub}_{\mathrm{s,max}}. \tag{13}$$

According to M_{diff}, MGDSE gives the top optimization priority to the longest sub-element, which is assumed to have the greatest influence.

M_{res} is computed to check whether the DSP and BRAM usage exceed the available resources on FPGAs, as shown in Eq.14:

$$M_{res} = \max\left(\frac{DSP_{used}}{DSP_{total}}, \frac{BRAM_{used}}{BRAM_{total}}\right), \tag{14}$$

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

M_{apt} evaluates which array partitioning type (*block* or *cyclic*) is beneficial in dimension i, as shown in Eq.15:

$$M_{apt,l} = \frac{\#loads}{\max\limits_{j,k}(index_i^j - index_i^k + 1)} \tag{15a}$$

$$M_{apt,s} = \frac{\#stores}{\max\limits_{j,k}(index_i^j - index_i^k + 1)}, \tag{15b}$$

where $M_{apt,l}$ and $M_{apt,s}$ represent the value of M_{apt} for loads and stores, respectively, and $index_i^j$ and $index_i^k$ are the indexes in dimension i of array elements j and k. When only one element is accessed, the denominator is zero so we add 1 here. If M_{apt} is smaller than 1, the array elements are accessed at intervals and *block* will be more beneficial. If M_{apt} is equal to 1, the elements are accessed continuously and *cyclic* type will be better. For instance, if a loop accesses $A[0]$ and $A[1]$ in one iteration, $M_{apt} = 1$. Then MGDSE tends to set the type as *cyclic* for array A, leading to two simultaneous loads.

As a whole, the MGDSE algorithm works as follows. First, the top function 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*). In *case 0*, MGDSE selects the longest sub-element to optimize each time, aiming at minimizing M_{diff} until zero. Then MGDSE attempts to optimize the new longest sub-element in the list. If M_{diff} is still larger than zero after optimization, the optimized sub-element will be removed out of the optimization list. Through this iterative way, the latency of all sub-elements will be minimized while honoring the resource constraints. In *case 1*, MGDSE first checks whether the function satisfies the *"producer-consumer"* requirement. If satisfied, the exploration will be conducted the same as *case 0*; 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 varies the array partitioning setting based on M_{apt} and computes M_{res} to remove the points which exceed the resource constraints.

For *cases 0* and *1*, when optimizing the sub-elements, the "function-type" sub-elements are viewed as new target functions and sent to MGDSE recursively. For "loop-type" sub-elements, MGDSE starts from the innermost loop and then the outer loops in each level. Each level's loop can be applied with *loop pipelining* and *loop unrolling* with various factors. MGDSE then computes M_{res} and M_{apt} for each loop configuration to choose the next point to test.

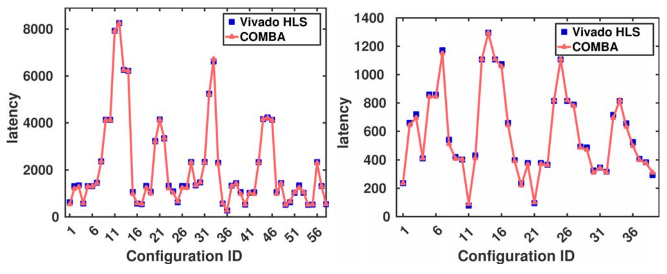
Based on the evaluation metrics, MGDSE ignores the points which make performance worse and chooses a promising configuration as the next point, exploring the design space rapidly and finding a high-performance configuration in minutes.

VII. Experiments and Results

We evaluate COMBA on several benchmarks, Polybench [9], CHStone [3] and the image processing applications used in [1, 19], and compare them with the current state-of-the-art [20]. COMBA is developed on LLVM 3.4 with Clang 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. Xilinx Vivado HLS 2016.1 is used to evaluate the accuracy of our model.

A. Estimation Accuracy

Figure 11 plots the performance of different configurations estimated by COMBA compared with those tested by Vivado



(a) BICG (b) JPEG

Fig. 11. Estimation performance comparison between COMBA and Vivado HLS

HLS at 100 MHz. We choose BICG from Polybench and JPEG from CHStone as examples to show the estimation accuracy for various benchmark suites. The x-axis is the different configurations, and we select representative ones from the original design space to evaluate the accuracy. The representative configurations are those that are promising to improve the performance and useful to guide the design space exploration. The y-axis denotes the latency of the benchmarks with various configurations. The curves in Fig.11 indicate that COMBA models the performance of different configurations precisely compared with the results from Vivado HLS. The average estimation errors of Polybench applications are shown in Table III, compared with [20]. The estimation errors of JPEG and other applications we tested are shown in column 2 in Table V. We can see that our comprehensive model not only improves the estimation accuracy further (around 1% difference) for Polybench, but also estimates the performance of more complicated benchmarks accurately. Note that the estimation error could be larger when array partitioning is applied because of the false dependency analysis of Vivado HLS, which is similar to [20]. Despite this, the tool can still estimate the performance trends and find a high-performance configuration successfully.

B. DSE Results and Comparison

Table II compares COMBA with the framework proposed in [20] for the Polybench benchmark suite. COMBA explores the design space, which is much larger than that in the previous work, as shown in columns 4 and 5. The reason is that we consider more characteristics of the target function, such as different kinds of loop structures, multi-dimension arrays and coupled functions/loops. [20] estimates the iteration latency of the inner-loop, which is not unrolled completely and then computes the total latency of a nested loop with its model, without considering other loop structures, like multiple loops (Fig.8). *Array partitioning* is also applied to one of the arrays in one dimension. Therefore the design space is small, leading to limited optimization results. In contrast, when we consider all the loops within the function, all the dimensions for each array and function-related optimization pragmas, the design space increases exponentially. For example, if each array has 10 configurations in one dimension and the target function contains three two-dimension arrays, the design space for the arrays will be 10^6, considering the combination of the configurations in six dimensions, which is exponentially more than the design space with 10 points for one dimension.

TABLE II
COMPARISON AND EXPERIMENT RESULTS OF POLYBENCH

Benchmark	Design Space		Performance Speed-up		MGDSE Time (s)	
	[20]	Ours	[20]	Ours	[20]	Ours
ATAX	85	1.31e+8	8.35	125.45	8.85	41.01
BICG	95	5.76e+8	15.88	201.38	22.60	89.20
GEMM	85	1.05e+10	8.15	261.63	185.37	65.83
GESUMMV	85	8.39e+8	15.42	83.88	12.05	77.40
MM	85	6.34e+13	15.22	277.03	161.37	292.15
MVT	95	1.05e+10	15.30	189.18	14.88	57.72
SYR2K	85	1.05e+10	7.27	123.96	250.01	223.00
SYRK	85	1.64e+8	8.12	462.65	168.78	86.34

Based on the precise estimation and expanded design space, further optimization can be achieved, as shown in column 7, compared with column 6, in Table II. The performance speed-up for COMBA is computed by comparing the optimal performance found by MGDSE with the baseline performance (without any optimization pragmas). The speed-up of [20] is obtained by testing its optimal configuration in Vivado HLS and comparing its performance with the baseline. We can see that a broader range of optimization improves the performance of applications significantly.

The detailed optimal configurations, returned by COMBA are shown in Table IV. The array size denotes the length in each dimension, which is set to the same value for arrays in the same benchmark. Column 3 presents which loop or function is pipelined, and column 4 shows the loop unrolling factors of top loops. Note that there are two top-level loops in the MM benchmark, and both of them are pipelined. For the MVT benchmark, the top function is pipelined, and therefore the two top loops are unrolled completely. Column 5 presents the configuration of each array in corresponding benchmarks with a format of *"array name: (type, factor, dimension)"*.

The last column in Table II shows the exploration time of our algorithm. Compared with [20], the time cost of COMBA is acceptable, considering the scale of the design space. For some benchmarks, such as GEMM, SYR2K and SYRK, COMBA even reduces the exploration time. We can therefore conclude that the MGDSE algorithm is able to efficiently find the high-performance configuration within minutes in an exponentially increasing design space.

C. Application Case Studies

Real applications often contain various complicated code structures. To evaluate the quality of our framework for analyzing complex applications, we test another three benchmarks, and the experimental results are shown in Table.V.

We choose *"decode_block"* as the target function in the JPEG benchmark, which is used in image decompression. The code structure and corresponding design space have been demonstrated in Section III-A. With five sub-functions and three one-dimension arrays, function pipelining combined with array partitioning is critical for this application. In addition, loop unrolling also improves the performance, as discussed in Section III-A. Figure 12 compares the latency speed-up (blue) and initiation interval (orange) for different configurations. The optimal configuration is *"FP_AllPartition"*, which is used to

TABLE III
ESTIMATION ERROR COMPARISON FOR POLYBENCH

	ATAX	BICG	GEMM	GESUMMV	MM	MVT	SYR2K	SYRK
[20](%)	2.68	1.24	3.25	5.15	2.69	2.05	1.32	2.78
Ours(%)	0.71	0.33	0.46	1.25	0.83	0.48	0.47	0.44

TABLE IV
OPTIMIZATIONS OF POLYBENCH

Benchmark	**Array size**	**Optimizations**		
		Pipeline	Unroll	Partition (array name: (type, factor, dimension))
ATAX	16	top-loop	2	A:(block,8,2); x:(complete,16,1); y:(block,8,1); tmp:(not partitioned)
BICG	32	top-loop	2	A:(block,16,2) (block,2,1); s:(block,16,1); p:(block,8,1); r,q:(not partitioned)
GEMM	16	top-loop	2	B:(block,8,2) (block,2,1); C:(cyclic,2,2); A:(not partitioned)
GESUMMV	16	top-loop	2	A,B:(block,8,2) (block,2,1); x:(complete,16,1); y,tmp:(not partitioned)
MM	8	top-loops(at the same level)	2,2	A,D,tmp:(complete,8,2); B,C:(complete,8,2) (complete,8,1)
MVT	16	top-function	16,16	y1,y2:(complete,16,1); A:(block,8,2) (block,2,1); x1,x2:(not partitioned)
SYR2K	16	top-loop	1	A,B:(complete,16,2); C:(not partitioned)
SYRK	16	top-loop	4	A,C:(complete,16,2)

TABLE V
EXPERIMENT RESULTS FOR CASE STUDY

Benchmark	**Estimation Error(%)**	**Design Space**	**Performance Speed-up**	**MGDSE Time(s)**
JPEG	1.54	7.61e+12	28.8	613
Seidel	0.91	1.05e+10	153.6	407
Rician	1.12	1.05e+10	47.1	343

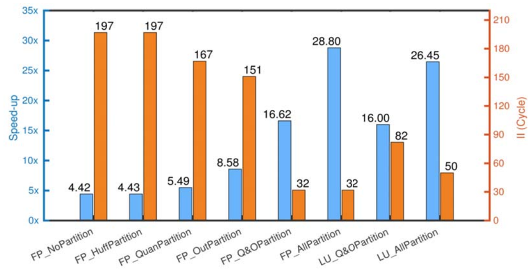


Fig. 12. Case study of JPEG

pipeline *"decode_block"* and partition all the arrays (Huffbuff, Quanbuff and Outbuff) completely. The loop unrolling pragma *"LU_AllPartition"* has close performance but is not as good as *"FP_AllPartition"*. After testing on our framework, COMBA also returns the same optimal configuration, improving the performance of *"decode_block"* significantly, with a 28.8x speed-up and a high throughput ($II = 32 cycles$ at 100 MHz).

The Seidel and Rician benchmarks have similar code structures with two two-level nested loops, and execute in a *producer-consumer* flow for which *dataflow* can be applied. However, the optimization results for both applications are different, as shown in Table V. The performance speed-up of Seidel with the optimal configuration is much larger than that of the Rician application. 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 complex applications successfully and optimizes the performance as far as possible.

VIII. CONCLUSION

This paper presents COMBA, a comprehensive model-based analysis framework to optimize complicated applications with various code structures, which is practical and useful for the optimization of real applications. Experiments demonstrate that COMBA not only performs better than a previous work on simple benchmarks, but also optimizes complex applications efficiently, within minutes. The tool is available at http://www.ece.ust.hk/~eeweiz/tools.html.

ACKNOWLEDGEMENTS

This work was supported by the RGC GRF grant 16245116. We thank the anonymous reviewers for their helpful feedback.

REFERENCES

[1] J. Cong et al. Customizable domain-specific computing. *IEEE Design & Test of Computers*, 28(2):6–15, 2011.
[2] X. Gao et al. Automatically optimizing the latency, area, and accuracy of c programs for high-level synthesis. In *Proc. of FPGA*, pages 234–243, 2016.
[3] Y. Hara et al. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *JIPS*, 17:242–254, 2009.
[4] Y.-K. Kwok et al. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *TPDS*, 7(5):506–521, 1996.
[5] T. M. Lattner. *An implementation of swing modulo scheduling with extensions for superblocks*. PhD thesis, Citeseer, 2005.
[6] P. Li et al. Resource-aware throughput optimization for high-level synthesis. In *Proc. of FPGA*, pages 200–209, 2015.
[7] H.-Y. Liu et al. On learning-based methods for design-space exploration with high-level synthesis. In *Proc. of DAC*, pages 1–7, 2013.
[8] N. K. Pham et al. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *DATE*, pages 157–162, 2015.
[9] L. Pouchet. Polybench/c 4.2.
[10] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. of ISCA*, pages 13–24, 2014.
[11] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of MICRO*, pages 63–74, 1994.
[12] Y. S. Shao et al. Aladdin:a pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. of ISCA*, pages 97–108, 2014.
[13] S. Sinha et al. Dataflow graph partitioning for area-efficient high-level synthesis with systems perspective. *TODAES*, 20(1):5, 2014.
[14] S. Wang et al. Flexcl: An analytical performance model for opencl workloads on flexible fpgas. In *Proc. of DAC*, 2017.
[15] Z. Wang et al. A performance analysis framework for optimizing opencl applications on fpgas. In *Proc. of HPCA*, pages 114–125, 2016.
[16] Xilinx. https://www.xilinx.com/support.html.
[17] Xilinx. Vivado design suite user guide high-level synthesis v2016.1, www.xilinx.com.
[18] C. Zhang et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proc. of FPGA*, pages 161–170, 2015.
[19] G. Zhong et al. Design space exploration of multiple loops on fpgas using high level synthesis. In *Proc. of ICCD*, pages 456–463, 2014.
[20] G. Zhong et al. Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators. In *Proc. of DAC*, page 136, 2016.