Exploiting Loop-Array Dependencies to Accelerate the Design Space Exploration with High Level Synthesis

Nam Khanh Pham[1,2], Amit Kumar Singh[3], Akash Kumar[2] and Mi Mi Aung Khin[1]
[1] *DCT Division, Data Storage Institute, A*STAR, Singapore*
[2] *ECE Department, National University of Singapore, Singapore*
[3] *Department of Computer Science, University of York, UK*

***Abstract*—Recently, the requirement of shortened design cycles has led to rapid development of High Level Synthesis (HLS) tools that convert system level descriptions in a high level language into efficient hardware designs. Due to the high level of abstraction, HLS tools can easily provide multiple hardware designs from the same behavioral description. Therefore, they allow designers to explore various architectural options for different design objectives. However, such exploration has exponential complexity, making it practically impossible to explore the entire design space. The conventional approaches to reduce the design space exploration (DSE) complexity do not analyze the structure of the design space to limit the number of design points. To fill such a gap, we explore the structure of the design space by analyzing the dependencies between loops and arrays. We represent these dependencies as a graph that is used to reduce the dimensions of the design space. Moreover, we also examine the access pattern of the array and utilize it to find the efficient partition of arrays for each loop optimization parameter set. The experimental results show that our approach provides almost the same quality of result as the exhaustive DSE approach while significantly reducing the exploration time with an average of speed-up of 14x.**

I. Introduction

The increasing complexity of Integrated Circuit designs combined with the shorter time to market requirement has made High Level Synthesis (HLS) a promising direction in EDA area. Recently, HLS tools that can automatically and quickly generate the RTL design from high level abstraction languages have emerged as the next mainstream in the world of electronic system design with the appearance of a lot of commercial products and academic research [4], [5]. With the ability to generate hardware designs in a fast and easy way, HLS tools allow the designers to easily evaluate different architectural implementation alternatives for the same high level behavioral description in order to satisfy different performance requirements and design constraints. To generate a new architectural implementation, the designers need to adjust a set of parameters that control the RTL generation process, such as: number of execution units, amount of resources to share, memories to allocate, data types and sizes, algorithm choice, pipeline stages, unrolling factors, etc. Each combination of different choices for these parameters provides a single option (performance and resource usage) for the generated hardware (HW) engine and forms a specific design point in the design space. The process of traversing the design space to find optimal design points that have an acceptable trade-off between objectives or design goals is called Design Space Exploration (DSE). Number of design points is exponentially proportional to the size of the space, which is governed by the number of parameters to be taken into account. Furthermore, the long runtime of the HLS tool itself is also a bottleneck that makes the DSE for HLS a very time consuming process [16].

To address the time consuming problem of DSE in HLS, different approaches have been proposed. Most of the approaches prune the design space using heuristic algorithms such as: simulated annealing, genetic algorithms, etc. [12], [14]. Some approaches apply machine learning techniques by building predictive model for HLS tools [3], [7]. Although these approaches can reduce the DSE time, they significantly sacrifice the quality of their solutions and can reach only to suboptimal solutions. The main reason is that existing approaches don't try to examine the structure of the design space and the relationship between explorable parameters of the problems. One of the most important relationships that has significant effect on the performance of HLS tools is the dependency between loops and arrays [10], [20].

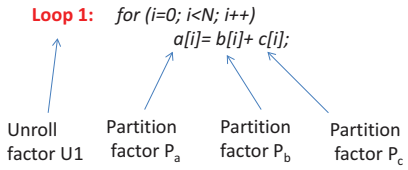


Fig. 1: Motivational example

To illustrate the importance of loop-array dependency on the DSE of HLS, we examine a simple example. Fig. 1 shows the pseudo code for adding two vectors b and c, and storing the results into vector a. If the dimension of the vectors is N then we can have N options for the unroll factor of Loop 1 and for each partition factor of arrays a, b and c as well. This implies that if we do not consider the access pattern of the array and the loop-array dependency, the design space complexity will be $O(N^4)$. However, it is easy to realize that if the partition factors of the array are greater than the unroll factor of the loop, the HW engines cannot utilize all the access to available partitioned memory banks. Hence, there is unnecessary wastage of resources for any array partition factor that is greater than the unroll factor of the loop accessing the array. Moreover, due to the simplicity of the code, we can observe that the optimal partition factor of the arrays should be the same as the unroll factor of the loop. Therefore, it is desirable and sufficient to traverse the design space for the loop unrolling factor only and such consideration will reduce the size of the design space to $O(N)$.

From the example demonstration (Fig.1), it is clear that the unroll factor of the loop and partition factor of the arrays are closely interdependent on each other. Furthermore, when the access pattern of the arrays is complex, it is difficult to extract the dependency between unroll factors of the loops and partition factors of arrays, as well as defining the optimal partition factors that fit the unroll factors. None of previous works have exploited such dependencies to improve the DSE process of HLS tools. Towards exploiting such dependencies, in this work, we propose a DSE framework for HLS tools that can exploit the loop-array dependency to reduce the evaluation time while evaluating optimal or near optimal solutions. Within the framework, following are the main contributions:

- *Loop-array Dependency Graph:* A systematic and formal method to represent the relationship between loops and arrays. We also developed a tool to extract the graph from C code.
- *Array Partition Factor Computation Block:* A module that can generate the Pareto optimal array partition factors according to the related loop optimization techniques.
- *Novel Framework for DSE in HLS:* A multilevel DSE approach that efficiently exploits the loop array-dependency to significantly reduce the DSE time.

Overview: Section II reviews history and trends of DSE approaches for HLS. Section III introduces the overall features of our DSE framework. The loop-array dependency extractor is highlighted in Section IV, while the main components of Array Partition Factors Computation Block are discussed in details in Section V. Experimental results are presented in Section VI to show the efficacy of our method. Finally, Section VII concludes the paper and provides directions for future works.

978-3-9815370-4-8/DATE15/©2015 EDAA

II. Related Work

A. DSE for HLS

The optimization and exploration activities are very common in digital system development and may happen in different levels of the design process. With regards to HLS, the DSE procedure can be roughly divided into two classes as follows:

1) DSE inside HLS: Existing works in this class focus on the DSE procedure for the internal tasks of the HLS tools themselves. As described in [11], the main components in the HLS flow are allocation, scheduling and binding. Each of these steps can be controlled by different factors which have a great impact on the performance and hardware usage of the resulting circuit implementation. Therefore, they are perfect candidates for applying different DSE approaches and a large body of works has been proposed to apply the DSE for different transformations in these steps to find the optimal hardware implementation generated by the HLS tools [12], [17]. Due to its inherence, most of the works in this class require full access to the HLS tool and their result may be applicable for only a specific HLS flow. In contrast, our framework targets the DSE flow in a higher level so that it is more general and can be utilized with various HLS flows. This advantage is also the common characteristic of the works in the second class.

2) DSE with HLS: Studies in this category are orthogonal to the works in the previous class since they are applied in a higher abstraction level and both techniques can be utilized at the same time without any conflict. Works classified in this category consider the HLS tools as a black-box and explore the design space of parameters that are provided to manage the available optimization techniques offered by the HLS tools. These works have appeared quite recently in comparison to the previous class since the HLS tools have only recently sufficiently matured. The earliest works in this direction tried to address the time consuming limitation of DSE by applying a heuristic algorithm called adaptive simulated annealing to prune the suboptimal design points [14]. In [15], [16], Carrion et. al. tried to reduce the complexity of the DSE problem by grouping the components (array, loop, functions) of original source code into smaller clusters and then running the DSE for each cluster. Although the evaluation time is reduced, the quality of the solutions is significantly affected. Trying to mitigate the effect of local optimization in the previous works, the authors applied the genetic algorithm to solve the DSE problem [3]. To further reduce the exploration time, the next generation of the works in this direction tried to apply Machine Learning (ML) techniques to build the predictive model for the HLS tool [3], [7]. In these works, several initial design points are generated by HLS tools to get learning database for the ML tools. Based on these initial data, different learning techniques are applied to get a predictive model that can simulate the behavior of the HLS tool as close as possible. After that, the subsequent evaluations are computed using the predictive model instead of calling the HLS tool.

Although the learning based methods can significantly reduce the evaluation time, the accuracy of the predictive models is usually not comparable to the real execution of the HLS tools. The main limitation of previous approaches is that they traverse the design space without any in-depth analysis of the structure of the space itself or without considering the relationship between parameters of the DSE process. In contrast, our work analyzes the most important dependencies between loop and array, extracts and presents them as a graph. Thereafter, we utilize these correlations to derive the optimal parameters for the array according to the given parameters of the loops. Hence, we limit the dimensions of the DSE process for the loop only and exponentially shorten the evaluation time, without sacrificing the quality of the result.

B. Memory optimization for HLS

With regards to memory optimization techniques for HLS, there are several works that aim to optimize the array partition for loop pipelining in HLS [6], [19], [20]. However, the authors in these works tried to improve the array partitioning process for loop pipelining only, whereas our work proposes a method to obtain the optimal array partition factor for loop unrolling techniques. Furthermore, the main target of earlier works is to develop code transformation tools to provide better input for HLS tools. In contrast, our work focuses on reducing the DSE time when using HLS tools.

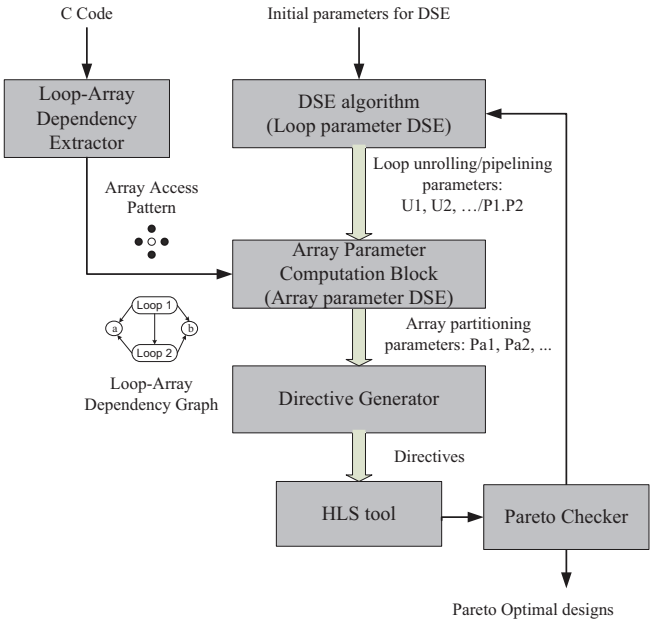


Fig. 2: Overall flow of proposed mapping strategy

III. Proposed DSE framework for HLS

DSE with HLS is usually a multi-objective optimization problem with regards to various conflicting objectives such as performance (throughput, latency), hardware usage or power consumption and reliability. In this work, we address the two main concerns of any digital system design: performance and hardware usage. However, the framework can be applied to include other criterion as well. Our main goal is to produce a Pareto optimal front of the designs so that the designer can choose between different trade-off points.

In HLS, optimization techniques related to loops and arrays have the most significant impact on the performance and resource usage of any hardware implementation [5], [20]. In our framework, we consider two prominent optimization techniques for loop, called *Loop pipelining* and *Loop unrolling*. Similarly, the most important array optimization technique, called *Array partitioning*, is also taken into account. Another reason for the choice of these optimization techniques is that they are commonly available in most of the HLS tools [8]. Therefore, our framework can be integrated with a variety of different HLS tools. The proposed framework is a multi-level DSE solution: first, the normal DSE algorithm runs for the loop parameters optimization. Then, inside each iteration of loop parameter set, another DSE process for array optimization parameters is executed.

The overall framework of our DSE approach is presented in Fig. 2. Three main components of the framework are the DSE Algorithm, the Loop-Array Dependency Extractor (LADE) and the Array Parameters Computation Block (APCB). The DSE algorithm is the main block that controls the whole DSE process by generating the new loop parameter set (*loop unrolling and pipelining parameters*) for each iteration of the framework. Thereafter, the new generated loop parameter set is passed to the second block (APCB) as a reference for the second DSE process to find the Pareto optimal *Array partitioning parameter* for the corresponding input loop parameter set. The DSE for array level runs on a simulator that can guarantee efficient array partition parameters according to each loop parameter set. By using a simulator the second level of DSE takes much less time as compared to traditional approaches that need to call the HLS for evaluating each array parameter set. The second level of DSE for array parameter is executed in the Array Parameter Computation Block (detail in Section 5). To initiate the APCB, we need the *Array Access Pattern* and the information from *Loop-array dependency*

Algorithm 1 Exhaustive DSE algorithm for loop optimization

Input: Loop nest with N loops
Output: All possible loop optimization parameter set
1: int PL // variable indicating pipelining technique in which loop level
2: int $U[N]$ // array storing the unroll factors for each loop level
3: **for** $i = N + 1$ to 1 **do**
4: **if** $i = N + 1$ **then**
5: $PL = 0$
6: TRAVERSE (N, PL, U)
7: **else**
8: $PL = i$
9: **for** $j = i + 1$ to N **do**
10: $U[j] = Max_j$
11: **end for**
12: TRAVERSE $(i - 1, PL, U)$
13: **end if**
14: **end for**
15: **procedure** TRAVERSE(N, PL, U)
16: **if** $N = 1$ **then**
17: **for** $i = 1$ to Max_1 **do**
18: $U[1] = i$
19: Output(PL, U)
20: **end for**
21: **else**
22: **for** $i = 1$ to Max_N **do**
23: $U[N] = i$
24: TRAVERSE$(N - 1, PL, U)$
25: **end for**
26: **end if**
27: **end procedure**

(LAD) graph, which are the output from the LADE. In contrast to the iterative execution behavior of the other blocks in the framework, the LADE is required to run only once at the beginning of the process to extract the LAD graph and the access pattern of the arrays, which are the references for the APCB. The architecture and functionality of LADE block are further discussed in Section IV.

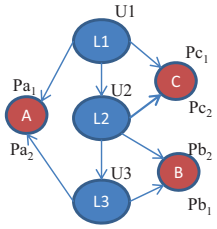
After getting the result from APCB, the tool passes information about the optimization techniques for both the loops and the arrays to the block Directive Generator to form the *Directives* for the HLS tools. Basically, this block is a script that depends on the Directive Library of different HLS tools. Finally, the HLS tools are called to generate the new hardware implementation as well as to evaluate the performance and resource usage of the current parameter set. The Pareto Checker will compare the results of the current parameter set with previous ones to decide whether it belongs to the Pareto front. Then, the control is passed to the DSE algorithm block to begin a new iteration for loop parameter DSE. The framework is terminated when it meets the *stop condition* in the DSE algorithm block.

In this framework, the two blocks LADE and APCB reflect our main contributions that utilize the LAD to mitigate the timing issue of DSE process without affecting the quality of the solutions. These blocks are designed in a modular approach as independent blocks from the DSE algorithm. Therefore, our framework provides users the freedom to utilize different DSE algorithms for the loop parameter DSE process such as exhaustive, heuristics: hill climbing, ant colony or multi-objective algorithm like Genetic Algorithm or Particle Swarm Optimization.

To illustrate the advantages of our approach over the traditional approaches, in this paper we use an exhaustive method for loop parameters DSE as described in Algorithm 1. We have customized the traditional exhaustive algorithm to fit with the problem of loop parameter optimization. As mentioned above, we consider two most important and popular loop optimization techniques: pipelining and unrolling. When pipelining is applied for an outer loop, all the loops nested in this current loop will automatically be unrolled [1]. Therefore, the pipeline directive is more critical and needs to be traversed first using the outer For loop (Line 2). First, the case of no pipelined loop is considered, where all the possible options of unrolling loops are examined using *TRAVERSE()* function (Line 4-7). Then, the loops are pipelined in order from inner-most to outer-most. Whenever, one loop in the loop nest is pipelined (Line 8), the unroll factor of the other loops under it in hierarchy will be set to the maximal value (Line 10), while the unroll factor exploration is considered for the loops outside of the pipelined loop (Line 12). The algorithm terminates when the outer-most loop is pipelined i.e.

```
Loop 1: for (i=0; i<N; i++)
  Loop 2: for (j=0; j<N; j++)
        { C[i][j]=0;
          Loop 3: for (k=0; k<N; k++)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
```

(a) Code listing



(b) LAD graph

Fig. 3: Example of matrix multiplication program

$PL = 1$. *TRAVERSE(N,PL,U)* function is a recursive function that can exploit all the unroll factors of the loops outside loop N and store the current pipeline and unroll factors into variable PL and U.

IV. Loop Array Dependency Graph

This Section discusses about the LAD graph, and its usage in the proposed framework. As shown in Fig. 2, the LADE block takes the C code as input and generates the *Array Access Pattern* and the *LAD graph*. Before going to the discussion of these two forms of representations, we need some preliminary concepts of **Polyhedral Model**, which is the fundamental of these representations.

A. Polyhedral Model

Polyhedral Model is an alternative representation of programs that provides high potential of analysis, expressiveness and flexible transformation for the loop nests. The **Polyhedral Model** is based on three basic concepts: *iteration domain, scattering function* and *access function*. In the scope of this paper, only the *iteration domain* and the *access function* are utilized so only these two concepts are defined. Reader who are interesting in the whole **Polyhedral Model**, can refer to [2] for more complete definitions.

• **Definition 1:** *Iteration Domain*. Given a nest of N loops, the iteration vector I is defined as: $I = (i_1, i_2, ..., i_N)$, where $i_j = 1, .., Max_j$ is the iterator of j^{th} loop. All possible values of the iteration vector forms the *Iteration Domain* of the given loop nest: $D_I = \{(i_1, i_2, ..., i_N) \in Z^N | 0 \leq \ i_j \leq \ Max_j, j = 1, .., N\}$ and each iteration vector represents one instance of loop iteration of the loop nest.

• **Definition 2:** *Array Domain*. Given an array A with M dimensions, the access vector of array A is defined as $R_A = (a_1, a_2, ..., a_M)$. Each particular instance of the access vector gives access to one element of the array: $A[a_1][a_2]...[a_M]$ and all possible values of the access vector form the Array Domain: $D_A = \{(a_1, a_2, ..., a_M) \in Z^M | 0 \leq \ a_j \leq \ Size_j, j = 1, .., M\}$, where $Size_j$ indicates the size of the array in j^{th} dimension.

• **Definition 3:** *Array Access Function*. Given array domain D_A and iteration domain D_I, the *array access function* F_k for k^{th} array reference is defined as: $F_k : D_I \longrightarrow D_A$. The array access function gives us the information about the element of array A that is accessed in a particular iteration of the loop nest. Since we consider only affine access to array, the function F_k has the following form: $F_k(I) = X * I + Y$, where X is an $M \times N$ matrix and Y is a constant vector of size M.

$$\underset{I}{F_k\begin{pmatrix} i_1 \\ i_2 \\ ... \\ i_N \end{pmatrix}} = \underset{X}{\begin{pmatrix} x_{11} & x_{12} & ... & x_{1N} \\ x_{21} & x_{22} & ... & x_{2N} \\ ... & ... & ... & ... \\ x_{M1} & x_{M2} & ... & x_{MN} \end{pmatrix}} \underset{I}{\begin{pmatrix} i_1 \\ i_2 \\ ... \\ i_N \end{pmatrix}} + \underset{Y}{\begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_M \end{pmatrix}} \quad (1)$$

B. Array Access Pattern

• **Definition 4:** *Array Access Pattern*. Given a loop nest I and array A, all existing references to array A in the loop nest forms the *access pattern* of loop nest I for array A. The access pattern for each array is a set of all references to that array and is presented as a set of matrix M and I.

We use an example of matrix multiplication loop nest to further clarify above-defined concepts. Fig. 3 presents the listing of the loop nest and the corresponding parameter values are as follows:

Iteration vector: $I(i, j, k)$

Iteration Domain: $D_I = \{(0, 0, 0), (0, 0, 1), \ldots, (N - 1, N - 1, N - 1)\}$

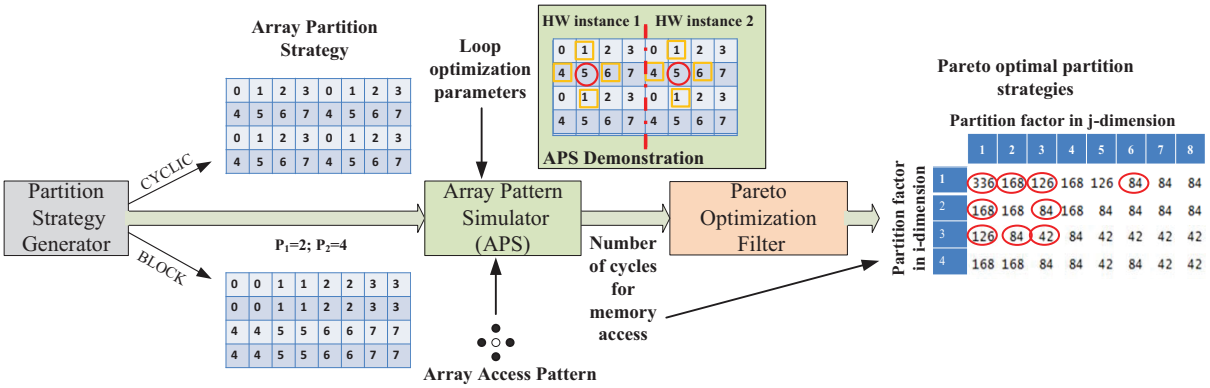


Fig. 4: APCB architecture

Access vector of array A: $R_A = (i, k)$
Array Domain of array A: $D_A = \{(0,0), \ldots, (N-1, N-1)\}$
Access function for array A:

$$F_A \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{X_A} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \underbrace{\begin{pmatrix} 0 \\ 0 \end{pmatrix}}_{Y_A} \quad (2)$$

Access pattern for array A: matrix X_A and vector Y_A

Similarly, access patterns of arrays B and C are given by vector $Y_B = Y_C = Y_A$ and two matrices X_B and X_C as follows:

$$X_B = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (3a) \qquad X_C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad (3b)$$

C. Loop-Array Dependency Graph

The LAD graph captures the overall relationships between different loop levels as well as the dependencies between loops and arrays in the loop nest.

- **Definition 5:** *LAD graph.* Given a loop nest with N loops, $L = \{L_1, L_2, \ldots, L_N\}$ and all the arrays accessed by the loop nest, $A = \{A_1, A_2, \ldots, A_K\}$, then the LAD graph is defined as a directed graph $G(V, E)$, where $V = L \cup A$ and E is defined as follows:
  - If the loops are arranged in order from outer-most to inner-most as: $L_1, L_2, \ldots, L_N$ then edge (L_i, L_{i+1}) belongs to E.
  - If array A_j is accessed by the iterator of loop L_i then edge (L_i, A_j) belongs to E.

Fig. 3b illustrates the LAD corresponding to the loop nest of Matrix Multiplication given in Fig. 3a. According to the above definition, there will be 3 nodes representing the loops in the loop nest (L_1, L_2, L_3) and 3 nodes for arrays representation (A, B, C). Optimization techniques applied to L_1 will affect the optimization of loop L_2, array A, and array C. Similarly, other edges in the LAD graph will indicate different dependencies between loops and arrays. The LAD graph benefits the whole framework by two folds. First, the loop dependencies are passed to the DSE algorithm to define the order of loop pipelining. For example, when we pipeline the loop L_2 then the unroll factor of L_3 should be maximal, and users do not need to consider other options for unrolling loop L_2. Secondly, the dependencies between loop and array are useful for the APCB block to define which array parameters should be considered according to the loop parameters. For example, whenever the optimization techniques are applied to loop L_1 then only partition factors of array A and C need to be examined and there is no need to evaluate the optimization techniques for array B. These representations are customized to the DSE HLS problem and are efficiently used in our framework.

V. Array Parameter Computation Block

The APCB block outputs the Pareto optimal parameters for array optimization techniques corresponding to the current loop parameter set. Fig. 4 illustrates the architecture as well as all the input and output data of this block. Basically, the APCB performs the second level of DSE process for array parameters while fixing the loop parameters. The *Partition Strategy Generator* traverses all over the design space of array parameters and exhaustively produces new *Array partition strategies* for the *Access Pattern Simulator* (APS). Taking all the required input of the access pattern, the current loop optimization parameters and array partition strategy, the APS will simulate the memory access behavior of the given hardware design and output the *number of cycles needed for memory access*. After getting all memory access cycles for every partition strategies, the results are passed through the *Pareto Optimization Filter* (POF) to get the final *Pareto optimal array partition strategies*. The detail implementations of these sub-modules are given in the next sections.

A. Array Partition Strategy

Memory partitioning is a widely used technique to improve memory bandwidth without data duplication. The original array will be placed into N non-overlapping banks, and each bank is implemented with a separate memory block to allow simultaneous accesses to different banks. The two most commonly used data partition schemes are *block partition* and *cyclic partition*, as shown in Fig. 4. These schemes provide regular partitions and thus can be easily implemented. This is desired for hardware synthesis as extra logic required to handle irregular patterns may increase the final design area drastically. Moreover, these schemes are widely supported by different HLS tools such as: Vivado HLS, LegUp, etc. Therefore, the memory partition strategies in our framework cover both cyclic and block partition schemes and are defined as follows:

- **Definition 6:** *Array partition strategy.* Given an array A with M dimension, then an array partition strategy for A is defined as a $(m+1)$ tuples: $PS = (p_0, p_1, \ldots, p_M)$, where:
  - p_i is partition factor for i^{th} dimension $(i \neq 0)$ and
  - $p_0 = 0$ for block partition, or $p_0 = 1$ for cyclic partition.

Mapping function for array partition strategies: Given an array partition strategy, the mapping function defines the index of memory bank for each element in the array. Mapping function is described as a function P that maps array address $A = (a_1, a_2, \ldots, a_M)$ in array domain to partitioned memory banks, that is, $P(A)$ is the memory bank index that A belongs to after partitioning: $P : D_A \longrightarrow Z$. The detail of memory mapping functions for block and cyclic schemes are provided in Algorithm 2.

B. Access Pattern Simulator

The APS utilizes analytical function to approximate the memory cycles instead of calling the HLS tool, and thus achieves much shorter execution time. The accuracy of the results generated by APS guarantees a theoretical bound for the actual results obtained by calling HLS. The performance gains and accuracy are further examined in experimental section.

The implementation details of APS are provided in Algorithm 3. First of all, the memory bank index for each array element is computed according to the partition strategy described in Algorithm 2 (Line 2). Then, the algorithm traverses through all executing iterations of the HW engines (Line 4). In each iteration, it iterates through all the HW instances (Line 5) (defined by the loop optimization parameters) and all memory references (Line 6) (defined by the access pattern of the code). For each memory reference of each HW instance, the simulator will compute the index of the

Algorithm 2 Memory mapping function for partition strategies

Input: Array partition strategies $PS = (p_0, p_1, \ldots, p_M)$; Array access vector $A = (a_1, a_2, \ldots, a_M)$
Output: Memory bank index of element $A[a_1][a_2]\ldots[a_M]$: $bank\_number$
1: **if** $p_0 = 0$ **then**
2: // Block partition
3: **for** $i = 1$ to M **do**
4: $temp_i = \lfloor Size_i \div p_i \rfloor$
5: $id_i = \lfloor a_i \div temp_i \rfloor$
6: **end for**
7: **else**
8: // Cyclic partition
9: **for** $i = 1$ to M **do**
10: $id_i = a_i \mod p_i$
11: **end for**
12: **end if**
13: $bank\_number = \sum_{i=1}^{M} id_i * \prod_{i+1}^{M} p_j$
14: **return** $bank\_number$

Algorithm 3 Access Pattern Simulator implementation

Input: Array partition strategies, Array access pattern, Loop parameters
Output: Memory cycles needed
1: **for** each array element **do**
2: Compute memory bank index using Algorithm 2
3: **end for**
4: **for** each executing iteration **do**
5: **for** each HW instance l^{th} **do**
6: **for** each memory reference k^{th} **do**
7: Compute access vector A using Equation 1
8: $R[l][k] \longleftarrow$ memory bank number of A (from Line 2)
9: **end for**
10: **end for**
11: Count frequency of each element in array R
12: Max = maximum of frequency from Line 11
13: $Sum = Sum + Max$
14: **end for**
15: **return** Sum

accessed element in original array using the access function matrix described in V-A (Line 7) then derive the memory bank index of this element from the result in Line 2. All the bank indexes accessed in current iteration are stored in a 2 dimension array $R[l][k]$, where l indicates the HW instances and k indicates the memory references. Thereafter, the algorithm computes the frequency of different values appeared in array R (Line 11). The values appeared in array R indicate the indexes of memory banks accessed in current iteration and the frequency of each value represents the number of accesses to the corresponding memory bank. The algorithm then identifies the memory bank with maximum number of accesses and stores its frequency into variable Max (Line 12). The final result, which is the total number of required memory cycles, is generated by accumulating the memory cycle of all iterations (Line 13).

An example demonstrating the mechanism of APS is presented in Fig. 4. The demonstration considers Denoise application with input array size of $(8, 4)$. Loop optimization parameters are unrolled 2 times for Loop 2, and no optimization is applied for Loop 1. Hence, there are 2 HW instances working concurrently as shown in the figure. The first HW instance operates on the left part of the original array, while the second instance processes the right part. The current partition strategy is cyclic scheme with partition factor $P_1 = 2$ and $P_2 = 4$. Following this partition strategy, the original array is divided into 8 different memory banks with the index from 0 to 7. The number in each memory cell represents the index of the memory bank that the cell belongs to. The red cycle indicates the current elements that are processed by two HW instances. The surrounding elements marked by the yellow square show all the required memory accesses needed for processing current iteration. In this particular iteration, we can see that each HW instance needs to access 1 element in bank 4, 1 element in bank 6, and 2 elements in bank 1. Since memory access on different memory banks can happen simultaneously, the memory access in bank 1 will be the bottleneck with 4 elements and 8 cycles (each access required 1 cycle for transferring address and 1 cycle for loading value).

C. Pareto Optimization Filter

After getting the memory cycles needed for all partition strategies, this block compares the partition strategies based on their partition factors in all dimensions as well as their memory cycles. Then, it keeps all the non-dominated partition strategies and output the Pareto optimal partition strategies in term of memory cycles and partition factors. According to [1], the resource utilization proportionally increases with the partition factor. Furthermore, since the computation cycle should not change for a fixed loop optimization parameter set, the overall performance will be characterized by the memory cycles. Therefore, the partition strategies generated by this block will be efficient in both latency and resource utilization. As the example demonstration, in Fig. 4, the Pareto-optimal partitions are marked with red cycles.

VI. Experimental Results

The results from our framework are compared with exhaustive search and following existing approaches: an *Adaptive Simulated Annealing (ASA)* approach in [14] and an approach using *multi-objective Genetic Algorithm (NSGA)* in [3]. The *ASA* approach has been implemented in Matlab 2013 using ASAMIN package [13]. Similarly, the evolutionary computing approach is implemented with the *NSGA II algorithm* from NGPM package [18] in Matlab 2013. The LADE block of our framework is implemented as an extension of the CLAN tool [2], while the DSE algorithm and the *APS* are developed in Java 8.0. *Vivado HLS* tool is used as the High Level Synthesis Tool with Zedboard as the synthesized platform. The experiments are performed for five applications from the Polybench [9], a popular benchmark for testing loop and array related problems. These applications are chosen to reflect different array access patterns: image processing function (Denoise), matrix multiplication (Matmul), triangular solver (Trisolve), Floyd-Warshall graph analysis algorithm (Floyd) and 2D Seidel stencil computation (Seidel) [9]. The criterion of the comparison are the quality of the Pareto fronts generated using above approaches and the effort needed by these approaches.

A. Quality of Pareto Front

The comparison on quality of the design space generated by four approaches are illustrated through Fig. 5 and Fig. 6. These figures are plotted in log scale, where the $Latency$ is the number of clock cycles output from Vivado HLS and the hardware utilization (HW) is characterized by following formulas: $HW = \lambda_1 * FF + \lambda_2 * LUT + \lambda_3 * BRAM + \lambda_4 * DSP$; where $FF, LUT, BRAM, DSP$ are the number of FIFO, LUT, BRAM and DSP blocks utilized in the design. The values of λ_i are platform-dependent coefficients and are inversely proportional to available resources for FIFO, LUT, BRAM and DSP on the platform.

Fig. 5 shows the results obtained from the second DSE level on array parameters for Denoise application with unroll factors $u_1 = 2$ and $u_2 = 4$. As shown in the figure, design points generated by our framework efficiently covers all the points on the Pareto front generated by exhaustive approach. The NSGA algorithm also can generate Pareto-optimal points but needs more evaluations while the ASA approach fails to do the same. The nice covering effect of the results from our framework can be explained by the fact that the APS provides a theoretical bound on the latency of the evaluated points. Therefore, the points generated from our approach (which are

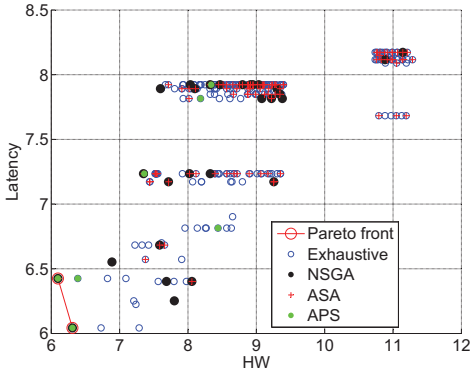
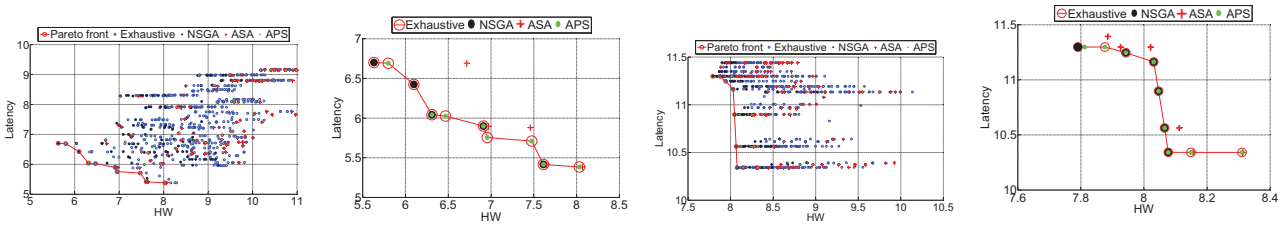


Fig. 5: Results for DSE on array parameters



(a) Design Space for Denoise (b) Pareto Front for Denoise (c) Design Space for Matmul (d) Pareto Front for Matmul

Fig. 6: Overall DSE

obtained from APS) always include all the optimal points in terms of Latency, which are part of the real Pareto front of the design space.

Similar results are observed for the overall DSE on loop level and are presented in Fig. 6. In this level, our framework clearly shows its advantages over the NSGA by covering a large fraction of the Pareto front while the NSGA can do that only with less number of points. The quality of results is described in more details using two concepts proposed in [16]: *Hypervolume* and *Pareto Dominance*. The *Hypervolume* indicates the fraction of design space that is dominated by the points generated from each approaches, while the *Pareto Dominance* counts the number of design points from each approach that belong to the Pareto front. The left part of Table I (Quality) shows that design points from our framework dominate the NSGA and ASA approaches for all the evaluated applications. The advantage of our approach at this level has resulted from the efficient results in the array parameters DSE level. It is also observed that the Pareto points missed by our approach mostly belong to the designs with low HW utilizations (Fig. 6d). It indicates that our approach does not efficiently detect the HW-optimal designs and the reason for this limitation is the simple estimation of HW criteria using array dimension in the Pareto Optimization Filter. This problem can be improved by integrating more efficient HW estimator to the Pareto Filter block, which is one of our directions for future work.

B. Execution time and number of evaluations

One of the most important features that define the efficiency of a DSE strategy is the number of evaluations performed by the strategy. The last two columns of Table I present the timing result and the percentage of number of evaluated points for all approaches. The results are normalized with respect to the exhaustive approach. As can be seen from Table I, the execution time of all approaches are proportional to the number of evaluations. The reason of this effect is that the execution time of the algorithms themselves are not significant (ignorable) when compared with time required for each evaluation, which are performed by executing the HLS tool. In Table I, our framework again demonstrates its advantages over NSGA and ASA approaches on both number of evaluations and execution time. Furthermore, it also brings a huge improvement over the exhaustive approach with an average speed-up of 14×. The efficiency of our approach comes from the fact that in the second level of DSE for array parameters, a significant amount of inefficient design points are eliminated by the APCB module. Hence, only promising candidates for the Pareto front are evaluated by HLS tool in our approach.

VII. Conclusions

This paper presents an efficient framework for utilizing the loop-array dependencies to solve the timing problem of DSE for HLS. The proposed framework includes a systematic and formal representation for the loop-array dependencies, a simulation block that can efficiently compute the Pareto optimal array partition parameters for each loop parameter set. Moreover, a multilevel DSE algorithm is also developed to exploit the framework. The experimental results show that our framework provides better Pareto front in term of resource utilization and latency in comparison to existing approaches while taking less time in execution. In future, to further reduce the exploration time, we plan to develop an analytical approach for evaluating the memory cycles based on the access pattern and loop optimization parameter set. Furthermore, we would like to integrate the framework with more efficient and accurate HW estimator to improve the quality of generated Pareto front.

TABLE I: Quality of the design space

Application	Approaches	Quality		Effort	
		Hypervolume (%)	Pareto Dominance	Evaluated Points (%)	Time (s)
Denoise	Exhaustive	100	10	100	66,484
	NSGA	96.88	5	17.71	18,456
	ASA	95.05	2	15.23	14,164
	APS	98.32	8	13.41	10,766
Matmul	Exhaustive	100	9	100	8,159
	NSGA	90.89	6	17.97	922
	ASA	88.41	2	16.41	679
	APS	96.21	8	7.29	371
Floyd	Exhaustive	100	3	100	29,134
	NSGA	88.80	1	15.49	2,894
	ASA	93.10	0	16.63	3,631
	APS	100	3	6.90	2,247
Trisolv	Exhaustive	100	4	100	193,848
	NSGA	78.26	2	17.32	22,098
	ASA	89.97	3	17.06	17,401
	APS	100	4	16.93	11,234
Seidel	Exhaustive	100	9	100	146,180
	NSGA	97.09	4	16.67	18,063
	ASA	96.35	2	15.63	15,244
	APS	97.66	7	15.49	13,165

References

[1] Vivado design suite user guide: High-level synthesis. Technical report, Xilinx, 01 2014.

[2] C. Bastoul et al. Putting polyhedral loop transformations to work. In *LCPC16*, pages 209–225, october 2003.

[3] B. C. Schafer and K. Wakabayashi. Machine learning predictive modelling high-level synthesis design space exploration. *IET*, page 153, 2012.

[4] A. Canis et al. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *FPGA*, pages 33–36. ACM, 2011.

[5] P. Coussy and A. Morawiec. *High-level synthesis*. Springer, 2010.

[6] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. Memory partitioning and scheduling co-optimization in behavioral synthesis. *ICCAD*, page 488, 2012.

[7] H.-Y. Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. *DAC*, page 1, 2013.

[8] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[9] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/*, 2012.

[10] L.-N. Pouchet et al. Polyhedral-based data reuse optimization for configurable computing. *FPGA*, page 29, 2013.

[11] A. Prost-Boucle, O. Muller, and F. Rousseau. Fast and standalone Design Space Exploration for High-Level Synthesis under resource constraints. *Journal of Systems Architecture*, 60(1):79–93, Jan. 2014.

[12] D. H. Ram, M. Bhuvaneswari, and S. Logesh. A novel evolutionary technique for multi-objective power, area and delay optimization in high level synthesis of datapaths. In *ISVLSI*, pages 290–295. IEEE, 2011.

[13] S. Sakata. Asamin:a matlab gateway routine to lester ingber's adaptive simulated annealing (asa) software, 2014.

[14] B. C. Schafer, T. Takenaka, and K. Wakabayashi. Adaptive simulated annealer for high level synthesis design space exploration. In *VLSI-DAT*, pages 106–109. IEEE, 2009.

[15] B. C. Schafer and K. Wakabayashi. Design space exploration acceleration through operation clustering. *ICCAD*, 29(1):153–157, 2010.

[16] B. C. Schafer and K. Wakabayashi. Divide and conquer high-level synthesis design space exploration. *TODAES*, 17(3):29, 2012.

[17] A. Sengupta and R. Sedaghat. Integrated scheduling, allocation and binding in high level synthesis using multi structure genetic algorithm based design space exploration. In *ISQED*, pages 1–9. IEEE, 2011.

[18] L. Song. Ngpm–a nsga-ii program in matlab. 2011.

[19] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory partitioning for multidimensional arrays in high-level synthesis. *DAC*, page 1, 2013.

[20] W. Zuo et al. Improving high level synthesis optimization opportunity through polyhedral transformations. *FPGA*, page 9, 2013.