HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds

Young-kyu Choi and Jason Cong
Computer Science Department, University of California, Los Angeles
{ykchoi,cong}@cs.ucla.edu

ABSTRACT

In order to further increase the productivity of field-programmable gate array (FPGA) programmers, several design space exploration (DSE) frameworks for high-level synthesis (HLS) tools have been recently proposed to automatically determine the FPGA design parameters. However, one of the common limitations found in these tools is that they cannot find a design point with large speedup for applications with variable loop bounds. The reason is that loops with variable loop bounds cannot be efficiently parallelized or pipelined with simple insertion of HLS directives. Also, making highly accurate prediction of cycles and resource consumption on the entire design space becomes a challenging task because of the inaccuracy of the HLS tool cycle prediction and the wide design space. In this paper we present an HLS-based FPGA optimization and DSE framework that produces a high-performance design even in the presence of variable loop bounds. We propose code transformations that increase the utilization of the compute resources for variable loops, including several computation patterns with loop-carried dependency such as floating-point reduction and prefix sum. In order to rapidly perform DSE with high accuracy, we describe a resource and cycle estimation model constructed from the information obtained from the actual HLS synthesis. Experiments on applications with variable loop bounds in Polybench benchmarks with Vivado HLS show that our framework improves the baseline implementation by 75X on average and outperforms current state-of-the-art DSE frameworks.

ACM Reference Format:
Young-kyu Choi and Jason Cong. 2018. HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds. In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3240765.3240815

1 INTRODUCTION

Because of its energy efficiency and reconfigurability, the field-programmable gate array (FPGA) is widely used in computing platforms such as the Amazon Web Service [1] or Microsoft's Azure [19]. However, such an advantage comes at the price of programmability. FPGA programmers have had to design and verify cycle-level logical operations at an RTL level and suffer from the long design cycle. To solve this problem, high-level synthesis (HLS) tools such as the Vivado HLS [9] and Intel HLS [13] have been proposed. These tools automate the transformation from a design written in a high-level language into a low-level FPGA design and increase the productivity of FPGA programmers.

But since HLS tools typically consume several minutes to transform high-level code to FPGA implementation for each set of design parameters, it is difficult to determine the set that will maximize the performance. This calls for automated design space exploration (DSE) for HLS. A considerable amount of literature can be found on this topic (Section 2). However, the previous work has difficulties finding an efficient design for applications with variable loop bounds in the innermost loops. For example, the average speedups for Polybench [21] benchmarks with variable loop bounds [21] are 2.3X and 1.0X using AutoAccel [6] and COMBA [27], while the approach proposed in this paper achieved a 75X speedup.

The main reason for such small speedup in conventional DSE works is that commonly used HLS directives *unroll* and *pipeline* may not be suitable for this type of application. As will be explained in Section 3, these common directives would generate processing elements (PE) with low utilization (<0.2) unless the code is properly transformed. For this reason, existing DSE tools that simply insert optimization directives will not be able to fully optimize the application.

Another reason for the poor optimization is due to the difficulty in performing cycle analysis. Vivado HLS will provide a very large range of cycles for variable loops, and the exact performance after each optimization becomes difficult to predict. It is possible to estimate the performance of applications with variable bounds based on the software simulation flow as proposed in HLScope [3, 4]. However, HLScope requires HLS synthesis of every design point to extract the loop cycle parameters—which is often infeasible. Works such as [23, 27, 28] use their own schedulers without the actual Vivado HLS synthesis; however, the accuracy might not be very satisfactory for loops with variable bound as will be shown in the experimental result.

To solve these problems, we present our initial work on an HLS-based optimization and DSE framework that improves the performance—even in the presence of variable loop bounds. First, we will demonstrate the deficiency in commonly used HLS pragma *unroll* and *pipeline* if used on variable loops. As a solution, we will propose source-to-source HLS code transformations that increase the utilization of the compute resources for variable loops with partial unrolling and pipelining in Section 3.2.

Furthermore, we identify the efficiency challenge that originates from the loop-carried dependency and the variable loop bounds. In particular, we analyze floating-point variable-loop reduction and

```
for (int j = i + 1; j < N; ++j){ // loop 3
  x = A[i][j];
  for (int k = 0; k < i ; ++k){  // loop 4
    x = x - A[j][k]*A[i][k];
  }
  A[j][i] = x * p[i];
}
```

Figure 1: Variable reduction example in Cholesky benchmark

```
for(int k = 0; k < N; k++){        //loop1
  for (int j = k + 1; j < N; j++){ //loop2
    A[k][j] = A[k][j] / A[k][k];
  }
  ...
}
```

Figure 2: Variable loop bound example in LU benchmark

prefix sum patterns that frequently appear in Polybench [21]. We will present code transformations to optimize these computation patterns in Sections 3.3 and 3.4.

Next, we present a cycle and resource estimation model for the variable loops in Section 4. In order to make an accurate resource estimation for large design spaces in a short time, our model interpolates based on a small number of actual HLS synthesis runs and considers the sharing of operands and arrays for various unrolling and array partitioning factors. The cycle estimation is based on the software profiling and the cycle model.

The overall framework and the DSE flow are explained in Section 5. Finally, the experimental result and the comparison with other works on the Polybench benchmark are presented in Section 6.

2 PREVIOUS WORK

The automated DSE framework for HLS is described in several published works. The work in [15] and [22] take high-level parallel patterns such as *map* and *reduce* and generate an FPGA design based on the predefined templates and the statistical performance model. Aladdin [23] omits synthesis and RTL generation and reuses optimization across a large design space for fast exploration among ASIC accelerators. Lin-analyzer [28] takes a similar approach and further considers the FPGA-specific resources (*e.g.,* DSP, BRAM) during its scheduling. Most recently, COMBA [27] and AutoAccel [6] have been proposed. COMBA explores a comprehensive set of HLS optimization directives and finds the best configuration based on their metric-guided search. AutoAccel presents a push-button flow based on their composable, parallel, and pipeline micro-architecture. These works, however, do not guarantee finding an efficient design for applications with variable loop bounds.

Previous published work, such as the work in [8] and ElasticFlow [24] discuss efficient HLS-based methodologies to distribute the dynamic workload among coarse-grain PEs. However, many examples, such as those found in Polybench benchmarks, do not have coarse-grain parallelism in outer loops (*e.g.,* row-wise parallelism in sparse matrix-vector multiplication [8, 24]). Instead, there are several variable loops that are executed in serial, similar to the examples presented in [17]. Thus, our work is more focused on optimizing these innermost loops by exploiting fine-grain parallelism and pipelining, accurately estimating resource sharing among these serial loops, and efficiently allocating non-sharable resource for overall latency minimization. Another difference that we see in these works is that they require modification of HLS scheduling and binding kernels—whereas our work is based on source-to-source transformation to produce HLS codes that can be easily integrated into existing HLS frameworks.

3 HLS CODE TRANSFORMATION FOR VARIABLE-BOUND LOOPS

In this section we will first identify the limitation of applying directives for pipelining and unrolling based on the maximum bound. Next, we will demonstrate the effectiveness of applying source-to-source transformation. In Polybench [21] the applications with variable loop bounds can be classified into three patterns: completely parallel, reduction, and prefix sum. We will discuss the transformation for each pattern in the following subsections.

```
for (int j = 1; j < 512; j++){
#pragma HLS unroll complete
  if(j >= k + 1 && j < N ){
    A[k][j] = A[k][j] / A[k][k];
} }
```

Figure 3: Loop unrolling for loop 2 of baseline LU code (Fig. 2) based on the maximum loop bound found in profiling

3.1 Loop Pipelining and Loop Unrolling Based on the Maximum Loop Bound

Since HLS tools cannot unroll the loops with variable bounds, a common optimization strategy is to pipeline the loop. For illustration, we optimize loop 2 of the LU baseline code (Fig. 2), which has matrix size N=512. After pipelining, the loop can be executed in 130,831 cycles (measured using technique described in [4]). However, pipelining cannot exploit the data-level parallelism that exists in the loop.

Another intuitive optimization strategy is to unroll based on the maximum loop bound measured from testbench profiling. The loop bound is fixed to a constant value as shown in Fig. 3. Condition $(if(j >= k + 1 \&\& j < N))$ is added to invalidate the execution of iterations where the loop index is not between the upper bound and the lower bound of the loop.

However, the resulting unrolled architecture suffers from a severe PE efficiency problem. Even if the dividers were all instantiated, profiling shows that the architecture can process only 130,816 divisions in 4,440,576 cycles—resulting in no speedup. On average, a divider PE is only performing 0.000057 divisions per cycle. There are two reasons for such inefficiency: First, many PEs are left idling when the loop trip count is smaller than the maximum loop bound. Second, the unrolled PEs are not pipelined. Due to such a low PE utilization problem, Vivado HLS only instantiates a single divider and shares it across the loop iterations.

3.2 Partial Unrolling with Pipelining

In order to exploit the loop parallelism while solving the PE inefficiency problem described in the previous subsection, we apply code transformations based on partial unrolling and pipelining. The idea is to place fewer PEs but allow them to proceed to other iterations in a pipelined fashion so that the effective PE utilization ratio would be increased.

Vivado HLS-compatible code transformation steps for the LU benchmark (Fig. 2) are as follows. As a preprocessing step, a common array reference (*e.g.,* $A[k][k]$) that is invariant to the loop is replaced with a temporary scalar variable ($lc1$) and moved out of the loop, as shown in line 3 and line 10 of Fig. 4. The reason is that Vivado HLS will synthesize it to an actual BRAM lookup and unnecessarily consume additional read port per iteration.

Next, we separate the original loop into two loops L2_1 (line 4) and L2_2 (line 7). The inner loop bound L2_2 is fixed to a constant $L2\_UF$ (line 7) so that the HLS tool may fully unroll it after inserting a pipeline directive on the outer loop (line 5). Assuming the original loop has lower bound lb and upper bound ub, the outer loop's lower/upper bound is set to $lb/L2\_UF$ and $(ub-1)/L2\_UF+1$ (line 4), so that the boundary iterations will be included as well. Arrays referenced in the loop are partitioned (line 2) to the unrolling factor ($L2\_UF$) in the array dimension (dim=2) referenced by the unrolled

```
01 #define L2_UF 4
02 #pragma HLS ARRAY_PARTITION variable=A factor=4 dim=2

03 float lc = A[k][k];               // loop-invariant code motion
04 for (int j1 = (k + 1)/L2_UF; j1 < (N-1)/L2_UF + 1; j1++){//L2_1
05 #pragma HLS pipeline
06 #pragma HLS DEPENDENCE variable=A inter false
07   for(int j2 = 0; j2 < L2_UF; j2++){ //fully unrolled    //L2_2
08     int j = j1 * L2_UF + j2;
09     if(j >= k + 1 && j < N ){//from orig loop's upper/lower bnd
10       A[k][j] = A[k][j] / lc;
11 } } }
```

Figure 4: Code after applying the proposed partial unrolling and pipelining techniques to loop 2 of Fig. 2

loop's index (j). If the loop index exists in more than one dimension (*e.g.,* $A[j][j]$), the proposed transformation is not valid. We also place a conditional statement to exclude iterations that were not between lb and ub (line 9).

As a last step, if there was no loop-carried dependency before splitting into two loops, we insert a pragma to declare inter-loop dependency to $false$ (line 6) for better performance [26]. This is legal because j $(= j1 * L2\_UF + j2)$ increases monotonically, and thus $A[k][j]$ will never reference the same array address in previous iterations.

The cycle estimation model of the proposed transformation will be presented in Section 4.2.1. Comparison of the execution cycles and the PE efficiency (average divisions per cycle per divider PE) is shown in Table 1. The proposed transformation exploits the loop parallelism and allows exploring various partial unrolling factors for wider design space exploration.

Table 1: Comparison of the execution cycles and the PE efficiency for loop pipelining (Section 3.1), loop unrolling based on the maximum bound (Section 3.1), and the proposed partial unrolling with pipelining (Section 3.2)

	Pipeline	Max Unr	Proposed Transformation		
			$UF_T = 2$	$UF_T = 4$	$UF_T = 8$
Ex cyc	130,831	4,440,576	81,792	49,088	32,736
PE eff	1.0	0.000057	0.80	0.67	0.50

3.3 Transformation for Variable Reduction

A reduction pattern is detected when a reduction operator [2] (*e.g.,* addition or multiplication) is applied on multiple array elements over a loop, and the result is reduced to a single variable or an array element. Subtraction can also be computed as a reduction pattern after replacing subtraction with addition, and the sign of the final reduction result is flipped. An example can be found in loop 4 of Polybench's Cholesky benchmark (Fig. 1). Note that in a strict sense, a floating-point addition is not a reduction operation (needs to have commutative and associative property), but it can be computed as a reduction pattern if some errors are tolerable [2, 5].

Many FPGA designs utilize a binary tree structure when implementing a reduction circuit [20, 29]. In Vivado HLS, this structure is inferred by specifying directive: "*#pragma HLS unroll factor=xxx.*" However, this implementation style is inefficient for floating-point variable loop reduction. The width of the tree has to be set to half of the maximum of the loop bound (256 in the example), and the depth has to be set to the log_2 of the width (8 in the example). If the loop bound is much smaller than the maximum, many adders will be left idle. To increase the PE efficiency, Vivado HLS will share the adders between different levels in the reduction tree. However, the efficiency is still low, because Vivado HLS does not properly pipeline

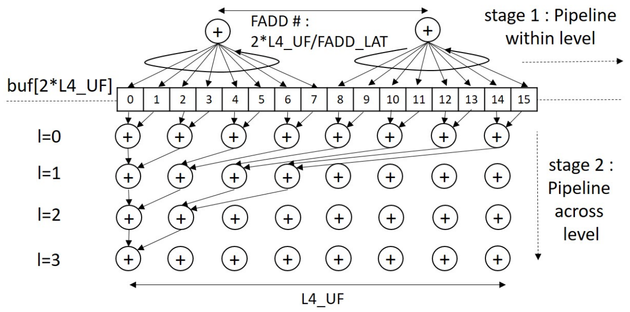

Figure 5: The computation pattern of variable reduction

```
00 #define L4_UF 8                                            // unroll factor
01 #pragma HLS ARRAY_PARTITION variable=A cyclic factor=2 dim=2
02 #pragma HLS RESOURCE variable=A core=RAM_2P_BRAM   //we assume A is in BRAM
03 float buf[2*L4_UF];                      //buf is a completely unrolled register
04 #pragma HLS ARRAY_PARTITION variable=buf complete

05 for( int a = 0 ; a < 2*L4_UF ; a++ ){
06 #pragma HLS unroll complete
07   buf[a] = 0;                                              //initialize to 0
08 }

09 for( int k1 = (0)/(2*L4_UF); k1<(i-1)/(2*L4_UF)+1; k1++ ){// L4_1, stage 1
10 #pragma HLS pipeline II=8
11  for( int k2 = 0 ; k2 < (2*L4_UF) ; k2++ ){                // fully unrolled
12    int k = k1*(2*L4_UF)+k2;
13    buf[k2] += ( k >= 0 && k < i  ) ? A[j][k]*A[i][k] : 0;
14  }          (from baseline loop's   (from baseline loop's comp
15 }            lower / upper bound)          to be reduced)

16 int limit; int len = min(i-0, 2*L4_UF);
17 if( len > 4 ){ limit = 3; }                        //sets reduction loop limit
18 else if( len > 2 ){ limit = 2; }                      // for early termination
19 else if( len == 2 ){ limit = 1; }                       // (uses table look up)
20 else { limit = 0; }

21 for ( int l = 0; l < limit; l++ ){                            // L4_2, stage 2
22 #pragma HLS pipeline II=8
23 #pragma HLS DEPENDENCE variable=buf inter false
24   for( int a = 0 ; a < L4_UF ; a++ ){
25     buf[a] = buf[2*a] + buf[2*a+1];
26 } }

27 x -= buf[0];                          //final result is stored in buf[0]
```

Figure 6: HLS code for loop 4 of Fig. 1 after transformation

the PEs when a partial unrolling factor is specified (Section 3.1). For example, when loop 4 of Cholesky benchmark is unrolled to the factor of 256 times (could not be fully unrolled to 512 due to the resource limitation), the floating-point adder (FADD) efficiency is 0.008 (= 22M adds / 21M cycles / 256 FADDs).

Inserting pipelining directive is also not very efficient for floating-point reduction. The reason is that there is a true loop-carried dependency, and the result of the previous iteration cannot be immediately produced because of the long latency of the floating-point operations (FADD_LAT). In loop 4 of the Cholesky example, FADD_LAT is 8 cycles. Thus, the average PE efficiency for pipelining is only 0.12, and requires 179M cycles to complete. To solve this problem, the work in [12, 30] proposes using shift registers (of length that matches FADD_LAT) to remove the dependency. The average PE efficiency improves to 0.16, but the parallelism is still limited.

We propose a code transformation to address these problems and to enable design space exploration. The reduction operation is divided into two stages (Fig. 5) depending on whether the number of elements to be reduced exceeds $2 * L4\_UF$ or not ($L4\_UF$: addition unrolling factor).

The reduction tree in stage 2 is pipelined across each level. With $L4\_UF$ FADDs, each level can be computed after FADD_LAT (=8) cycles, because the number of elements to be added per level is equal to or less than $2 * L4\_UF$. In order to reduce the tree depth for a small loop bound TC, we support early termination by pipelining each level with a variable loop bound (line 21 of Fig. 6). Stage 2 loop bound ($limit = log_2(min(TC, 2 * L4\_UF))$) is precomputed based on the table lookup (lines 16–20). Note that Vivado HLS instantiates only $L4\_UF/2$ FADDs for stage 2, which increases the pipeline depth of the loop by 1 (FADD_LAT+1 cycles in total).

In the stage 1 of Fig. 5, the computation should be pipelined within each level since the number of elements to be added exceeds $2 * L4\_UF$. In order to increase the efficiency, we adopt the dependence-free pipelining [12, 30] by allowing a single FADD to write intermediate results to $FADD\_LAT$ registers. This is achieved by specifying II to $FADD\_LAT$ (line 10). For higher performance, we increase the parallelism of this technique to $2 * L4\_UF/FADD\_LAT$. In order to reduce the number of iterations for small loop bound, early termination within a reduction level is allowed by setting the loop bound of stage 1 to $(ub - 1)/(2 * L4\_UF) + 1$ to $lb/(2 * L4\_UF)$ (line 9).

The cycle estimation model of the proposed transformation will be presented in Section 4.2.2. The performance comparison of the proposed reduction scheme with conventional pipelining, dependence-free pipelining [12, 30], and conventional unrolling is presented in Table 2. The DSP efficiency of the proposed scheme is higher than the conventional unrolling scheme with the same unrolling factor by 41X (UF=4) to 29x (UF=16). Similar high efficiency can be observed in FF and LUT as well. The high efficiency diminishes with the larger unrolling factor—but nonetheless, the proposed scheme was able to find a final design point that is 14X, 2.2X and 1.7X faster than the conventional pipelining, dependence-free pipelining, and conventional unrolling. Also, due to the early termination functionality for small variable bound across and within reduction levels, the latency for loop bound of 1 (42 cycles) is smaller than dependence-free pipelining (56 cycles) and conventional unrolling of factor 256 (168 cycles).

Table 2: Comparison of the total execution cycles, resource consumption, and latency of various loop bounds (for cases min=1, max=512) for the proposed variable-bound reduction scheme with conventional pipelining, dependence-free pipelining [12, 30], and conventional unrolling, for loop 4 of the Cholesky benchmark

	Unr	Total	Resource			Latency	
	Fac	Cycles	DSP	FF	LUT	LB=1	LB=512
Pipe	-	179M	5	653	680	15	4103
DPip	-	29M	11	3131	2904	56	560
Unrolling	4	537M	16	1922	1674	4104	4104
	8	336M	32	3706	3036	2568	2568
	16	202M	64	7234	5691	1544	1544
	256	22M	1K	114K	86K	168	168
Proposed	4	30M	7	2016	2911	42	570
	8	20M	14	3760	5032	42	322
	16	16M	28	7192	9252	42	202
	128	13M	224	56K	67K	42	114

3.4 Transformation for Variable Prefix Sum

A prefix sum is a computational pattern where the output array y contains a running sum of the input array x ($y_k = \sum_{j=0}^{k} x_j$) [5, 11]. The prefix sum pattern is detected when there exists a loop with an assignment statement written to an array element $y[k]$ with a value that is the sum of the array element assigned in the previous iteration ($y[k-1]$) and an element from the input array ($x[k]$). An example is presented in loop L2_2 of Fig. 7 for rotated integral image

```
L2 : for(int a=0; a<N; a++){ // Compute lower-left triangle
  L2_1 : for(int j=0; j<N-a; j++){ //Aggregate all except intimg[i-1][j-1]
    int i = a+j;
    float arg1 = (j!=N-1 && i!=0) ? intimg[i-1][j+1] : 0;
    float arg2 = (j!=0 && j!=N-1 && i!=0 && i!=1) ? intimg[i-2][j] : 0;
    float arg3 = img[i][j];      float arg4 = (i!=0) ? img[i-1][j] : 0;
    data[j] = arg1 - arg2 + arg3 + arg4;
  }

  psum[0]= data[0]; //init
  L2_2 : for(int k=1; k<N-a; k++){ //compute integral image in diagonal
    psum[k] = psum[k-1] + data[k]; //prefix sum pattern
  }

  L2_3 : for(int k=0; k<N-a; k++){ //copy prefix sum result into intimg
    intimg[a+k][k] = psum[k];
} }
```

Figure 7: Baseline code for rotated integral image computation [18] used in face recognition.

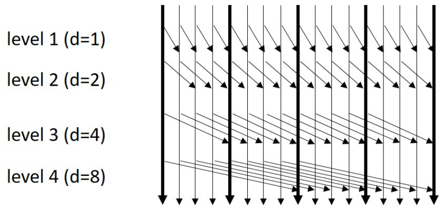

Figure 8: Kogge-Stone prefix sum algorithm [16].

application [18] in face detection. Similar to the reduction pattern (Section 3.3), the true dependency between $psum[k]$ and $psum[k-1]$ prohibits II becoming 1 when the loop is pipelined and $psum$ is a floating-point variable. Applying an unrolling directive as suggested in [14] results in a serialized addition due to the dependency and does not bring any speedup.

In order to increase the performance with parallelization, we use the Kogge–Stone algorithm [16]. Although the algorithm is not work-efficient [5, 11], the consecutive and regular memory access pattern helps simplify the data fetch circuitry between BRAMs and PEs. The algorithm is presented in Fig. 8. In each level l, addition is performed with an array element that is d apart:

$$y_k^l = y_k^{l-1} + ((k \geq d) ? y_{k-d}^{l-1} : 0). \tag{1}$$

Distance d is multiplied by a factor of 2 in each level.

However, direct implementation of Eq. 1 [11] results in a performance improvement of only 0.94X, 1.1X, and 1.2X with unrolling factor 2, 4, and 8, compared to the pipelined version. There are two reasons for such a small speedup.

The first reason is that Vivado HLS will assume that the memory access stride would be an arbitrary number when the stride is a variable (*e.g.*, d in Eq. 1, since d increases in a power of 2). Thus, Vivado HLS will infer wiring from each adder to all memory partitions (M) that results in a large II. However, the actual wiring required for each adder is only $logM$ (d=1, 2, 4, ..., $M/2$).

The second reason is that, as can be seen from Eq. 1, two read (y_k^{l-1}, y_{k-d}^{l-1}) and one write (y_k^l) ports are required per iteration. This requirement holds regardless of whether d is a multiple of the memory partition M. Since Vivado HLS schedules up to two read or write ports per memory partition each cycle, II=1 cannot be achieved. Another related problem is that y_k^{l-1} of Eq. 1 is later accessed by the term y_{k-d}^{l-1} when $k = d$. Thus, overwriting the content y_k^{l-1} with y_k^l will result in an incorrect behavior. Note that [11] solves the latter problem with a double buffering technique, but this requires additional loops to copy the result back from the

```
01 #define N 512                                      // max prefix sum length
02 #define UF 4                                                // # of FADD
03 #define LOGUF 2                                              // LOG2(UF)
04 float dpsum[N];
05 #pragma HLS ARRAY_PARTITION variable=(data,psum,dpsum) cyclic factor=4

06 int l_max = ...;    // compute log2 of the loop bound N-a (code omitted)
07 dpsum[0] = psum[0];                                                //init
                       (from baseline loop's
08 if( l_max >= 1 ){    lower / upper bound)                   //case : d=1
09   for(int k1 = (N-a-1)/UF; k1 >= 1/UF; k1--){   //traverse in decr order
10 #pragma HLS DEPENDENCE variable=psum, dpsum inter false
11 #pragma HLS pipeline II=1
12     for( int k2 = UF-1 ; k2 >= 0 ; k2-- ){
13       int k = k1* UF + k2;
14       int kd = k1* UF + k2 - 1;                                     // d=1
15       if( 0 <= kd && k < N-a ){
16         psum[k] = data[k] + data[kd];                               // add
17         dpsum[k] = psum[k];   //duplicate psum into dpsum for next level
18 } } } }

19 if( l_max >= 2 ){                                           //case : d=2
         ...
20       int kd = k1* UF + k2 - 2;                                     // d=2
           ...
21         psum[k] = psum[k] + dpsum[kd];                              // add
22 } } } }

23 int dUF = 1;                                            // case : d >= UF
24 for( int l = 0 ; l < (l_max-LOGM) ; l++ ){
25 for(int k1 = (N-a-1)/UF; k1 >= 1/UF; k1--){     //traverse in decr order
26 #pragma HLS DEPENDENCE variable=psum, dpsum inter false
27 #pragma HLS pipeline II=1
28     for( int k2 = UF-1 ; k2 >= 0 ; k2-- ){
29       int k = k1* UF + k2;     int kd = (k1-dUF) * UF + k2;
30       if( 0 <= kd && k < N-a ){
31         psum[k] = psum[k] + dpsum[kd];                              // add
32         dpsum[k] = psum[k];   //duplicate psum into dpsum for next level
33   } } }
34   dUF *= 2;                                   //increase dUF (=d*UF) by 2
35 }
```

Figure 9: Proposed transformation of variable prefix sum in loop L2_2 of rotated integral image computation (Fig. 7)

ping-pong buffer when the number of levels is odd. Also, double-buffering does not reduce the number of ports required.

The first problem is solved by explicitly enumerating all possible memory access strides. The code transformation for loop L2_2 of rotated integral image example (Fig. 7) is shown in Fig. 9. We assume the number of FADD (UF) is same as the memory partitioning factor. As can be seen in lines 8-18 (d=1) and lines 19-22 (d=2), all levels with d less than M are explicitly enumerated. When d is a multiple of M, all the operands for Eq. 1 can be obtained from the same memory partition of *psum*, and thus can be packed into a single loop, as shown in lines 23-35.

The second problem is solved by making a duplicate copy of y and traversing the array in a decreasing order. An example is shown in Fig. 9. Starting from the upper bound of the loop iterator (lines 25 and 28), we perform the addition in Eq. 1 (line 31), with y_{k-d}^{l-1} in array *dpsum* (duplicate of y) and y_k^{l-1} in array *psum*. The result y_k^l is updated to both arrays *psum*[k] and *dpsum*[k] (line 32). Although y_k^{l-l} is now overwritten, this does not cause a problem since this value is never accessed again with monotonically decreasing loop iterators $k1$ and $k2$. Also, one read and one write per partition is performed for both *psum* and *dpsum*, which allows the loop to execute with II=1.

Note that the proposed scheme takes advantage of the fact that the computation and memory access schedule can be controlled by FPGA HLS programmer. It is not applicable to CUDA GPU environment [11], because execution in a monotonically decreasing order cannot be guaranteed among multiple CUDA threads.

The cycle estimation model of the proposed scheme will be presented in Section 4.2.3. The comparison of the proposed scheme with the conventional pipelining, unrolling [14], and direct implementation of Kogge-Stone [11] is shown in Table 3. Both pipelining and unrolling infers an architecture that lacks parallelism and has low PE efficiency because of the dependency that exists between $y[k]$ and $y[k-1]$. Direct implementation of Kogge-Stone has a limited speedup with increasing parallelism due to the large II. The performance of the proposed scheme is superior, because we were able to achieve II=1 for all loops. Also, support for early termination (lines 6, 8, 19, 24 of Fig. 9) reduces the cycle for short loop bounds. As a result, the proposed scheme allows DSE to find a design point that is 9.7X faster than the pipelined version.

Table 3: Comparison of the total execution cycles, resource consumption, and latency of various loop bounds (for cases min=1, max=512) for the proposed variable-bound prefix sum scheme with conventional pipelining, unrolling [14], and Kogge-Stone algorithm [11] for loop L2_2 of rotated integral image

	Unr	Total	Resource			Latency	
	Fac	Cycles	DSP	FF	LUT	LB=1	LB=512
Pipe	-	917K	0	575	814	9	3586
Unr	4	2.4M	2	877	1372	4637	4637
[14]	8	2.4M	2	1525	3026	4674	4674
K-S	4	974K	6	2876	3911	35	3781
[11]	8	822K	8	5385	7166	35	3141
Proposed	4	331K	8	2679	3820	19	1253
	8	198K	16	5299	7590	22	691
	16	134K	32	11K	16K	25	417
	64	95K	128	43K	71K	31	229

4 CYCLE / RESOURCE ESTIMATION

Vivado HLS provides cycle and resource estimate in its synthesis report. If the design space is large, generating an HLS report for every possible space is not feasible since it takes several seconds to minutes to generate a single design. Aladdin [23], Lin-analyzer [28], and COMBA [27] solve this problem by estimating cycles with their own scheduler, but there is no guarantee that the HLS tool will follow such a schedule, and may possibly result in a large cycle estimation error, as will be shown in Section 6.

In our framework, we extract basic cycle and resource information from the HLS tool for few designs. Based on this information, we predict the cycle and resource consumption for the entire design space based on our model.

4.1 Resource Estimation

The resource consumption for pipelined loops is obtained from the synthesis report of the Vivado HLS tool. For loop unrolling and array partitioning, however, synthesizing every possible design with the HLS tool becomes difficult due to the large design space. Assuming there are L innermost loops that may each be unrolled up to U times, and A arrays that may each be partitioned up to P times, a naive approach would be to perform $U^L * P^A$ HLS synthesis.

One alternative approach could be to linearly interpolate from a few unrolling factors and array partitions for every loop, assuming all loops' resource consumption is independent from one another. This assumption is not true, however, because of the resource sharing between the loops. As explained by Li, *et al.,* [17], modern HLS tools, including Vivado HLS, will share operators that exceed certain thresholds across loops for high operator utilization. Floating-point operators exceed this threshold and will be shared across loops. Then a new challenge arises to efficiently predict the resource sharing for many possibilities of unrolling factors and array partitions.

We propose a resource prediction method that reduces the number of HLS synthesis and is still based on the actual HLS synthesis report. The high-level strategy is to separate sharable and non-sharable operators from a loop and linearly interpolate the non-sharable resource. The resource for sharable operators is estimated as a maximum of all loops. Finally, we estimate the mux required for sharing the operators and the arrays.

For each loop l=(0, ..., L) that is to be unrolled (after applying transformations described in Sections 3.2, 3.3, and 3.4), we generate and synthesize a new version of code with R+1 different unrolling factor for each loop. Then we compute the resource difference between each version to estimate the increase rate of resource consumption. By referring to the sharable operator (floating-point operators) usage report, the resource increment is separated into sharable resource and non-sharable resource consumption (NS_l). Based on the non-sharable resource consumption for R designs, we use a linear regression technique to find the slope and the intercept of the data points. Then the non-sharable resource of a loop can be estimated for arbitrary unrolling factors.

The estimation error with different R for LU benchmark is shown in Table 4. Since large R increases the number of synthesis runs, we have decided to limit R to 3. Our framework generates loops with unrolling factor 4, 8, and 16 (R=3), since non-linear characteristic is sometimes observed for small (1, 2) unrolling factors.

Table 4: FF/LUT estimation error rate for various R for LU benchmark

	R=2	R=3	R=4
FF/LUT	1.2% / -6.9%	0.9% / -5.5%	0.5% / -7.2%

To estimate the resource for sharable operators, we find the maximum of each type k of operators (*e.g.,* floating-point adder or multiplier) after loop unrolling each loop (S_{lk}). Next, we estimate the resource consumed by mux used for sharing each type of operator among loops. Likewise for the arrays, we estimate the mux needed for different loops to have access to the same a arrays. Then we estimate LUT consumption for the input operands of each operator and the address/data of arrays based on the bitwidth and the number of sharing ports. The LUT consumption model for mux can be found in [7].

The above resource consumption estimation can be summarized as:

$$\sum_l NS_l + \sum_k \max_l(S_{lk}) + \sum_k mux(l,k) + \sum_a mux(l,a) \quad (2)$$

Since only $R+1$ (=4) synthesis are required for each unrolled loop, the number of HLS synthesis runs to estimate the resource for various unrolling factors is $(R+1)*TL$, where TL is the number of innermost loops. This is a large improvement over the naive approach of performing $U^L * P^A$ synthesis. As a result, the design space exploration time is reduced to a few hundreds of seconds, as will be presented in the experimental section.

Our resource estimation method differs from AutoAccel [6] in that AutoAccel does not model resource sharing among loops. Our work has more similarity with [17] in a sense that we separate sharable and non-sharable resource of a loop. The difference is that [17] does not explore loop unrolling or array partitioning and thus does not perform any linear regression or function separation. Instead, they assume that the non-sharable part stays constant over multiple designs. However, assuming a constant non-sharable resource increases the LUT/FF estimation error rate from 0.9%/-5.5% to 32%/-48% in LU because non-sharable resource can increase very rapidly with loop unrolling.

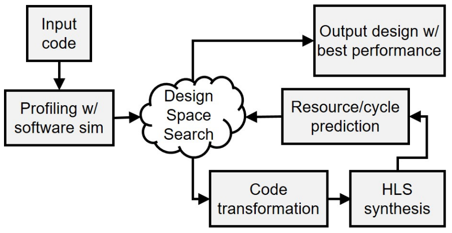
Input code
Output design w/ best performance
Profiling w/ software sim
Design Space Search
Resource/cycle prediction
Code transformation
HLS synthesis

Figure 10: Overall DSE framework

4.2 Cycle Estimation

4.2.1 Model for Partial Unrolling with Pipelining. For the LU benchmark in Fig. 4, if the transformed loop's ($L2\_1$'s) initiation interval, iteration latency, and unroll factor is II_T, IL_T, UF_T, the number of execution cycle is $II_T * \{(ub-1)/UF_T + 1 - (lb/UF_T) - 1\} + IL_T$ [17]. This is approximated as

$$\simeq II_T * (ub - 1 - lb)/UF_T + IL_T = II_T * (TC-1)/UF_T + IL_T \quad (3)$$

where $TC(= ub - lb)$ is the trip count of the original loop (loop 2 in Fig. 2).

4.2.2 Model for Variable Reduction. For the Cholesky benchmark in Fig. 6, the execution cycles of stage 1 can be directly derived from Eq. 3: $II_{L4\_1} * (TC-1)/(2 * UF_{L4}/FADD\_LAT) + IL_{L4\_1}$. The loop in stage 2 will be executed $log_2(min(TC, 2 * L4\_UF))$ times. Thus the total estimated cycle is

$$\begin{aligned} &II_{L4\_1} * (TC-1)/(2 * UF_{L4}/FADD\_LAT) + IL_{L4\_1} \\ &\quad + II_{L4\_2} * (log_2(min(TC, 2 * L4\_UF)) - 1) + IL_{L4\_2} \quad (4) \end{aligned}$$

4.2.3 Model for Variable Prefix Sum. The code in Fig. 9 is a collection of partially unrolled loops modeled in Eq. 3. The number of levels is $l_{max} = logTC$. Thus, the total estimated cycle is

$$\sum_{l=0}^{l_{max}} (II_l * (TC-1)/UF + IL_l). \quad (5)$$

4.2.4 Total Cycles. If the trip counts of all variable loops were available, the total cycle would simply be an accumulated number of the cycles computed in Eqs. 3, 4, 5. However, since storing all trip counts is an expensive process, we simplify the computation by first computing the average of trip counts (AVG_TC) and the number of loop occurrences (OCC) during the software simulation. Then the trip counts TC in Eqs. 3, 4, 5 are replaced with AVG_TC, and the entire equation is multiplied by OCC. Also, we approximate the II, IL of the unrolled loops based on the value already obtained from the selected (R=3) synthesis. As will be shown in the experimental section, these approximations result in a relatively small cycle error rate of 12%.

5 OVERALL FLOW AND THE DESIGN SPACE EXPLORATION

The overall flow is shown in Fig. 10. The input code is first profiled with the Vivado HLS software simulation flow. In this stage, the loop trip counts TC and the number of loop occurrences OCC are recorded. Next, possible transformed codes are generated from the input code discussed in Section 3. Next, R design points for each loop are synthesized with the HLS tool (Section 4.1). Based on the synthesized result, cycle count and resource consumption are estimated as discussed in Section 4. Among possible design points that satisfy the resource constraint, the one with the least latency is chosen and presented as the final output.

Following Lin-analyzer [28], the design parameters evaluated in the framework are shown in Table 5. Loops can be unrolled to their maximum loop bound, and the array can be partitioned to its array size. For simplification, the loop unrolling factor and the array partitioning factor are explored in power of 2s (1, 2, 4 ...). We also explore the loop pipelining. As mentioned in Section 2, the optimization is performed on the innermost loops for fine-grain parallelization and pipelining.

Table 5: Design parameters evaluated in DSE

Parameter	Range
Loop unrolling factor	1 : Max loop bound (pow of 2)
Pipelining	True, False
Array partitioning factor	1 : Array size (pow of 2)

In order to reduce the design space, we prune away design spaces that are not promising. First, we do not consider array partitions that are larger than the number that can be simultaneously consumed or produced by PEs. Second, we do not consider pairs of optimizations that require partial partitioning on multiple dimensions of a local memory, because this complicates routing, and the number of BRAM instances increases rapidly.

6 EXPERIMENTAL RESULT

6.1 Experimental Setup

For evaluation, we use the Polybench benchmark [21] that was also used in Lin-analyzer [28] and COMBA [27]. To demonstrate the effectiveness of our framework, five benchmarks with variable loop bounds have been chosen—Cholesky, LU, Trisolv, Durbin, and Dynprog. We also constructed rotated integral image benchmarks from [18]. The matrix size is set to 512, and the variable types are set to single-precision floating-point for all benchmarks, except Dynprog which was set to have a matrix size of 128 to fit FPGA. The design is synthesized using Vivado HLS 2018.2 [26] software. For the platform, we target the ADM-PCIE-KU3 board [10] with Xilinx's Ultrascale KU060 FPGA [25] at 250MHz. The FPGA resource (DSP/FF/LUT) limit is set to half of the total resource on KU060 to ease the place-and-route process.

6.2 Performance

The performance after applying the proposed transformations to the unmodified baseline code is 8.8X to 317X, as shown in Table 6. LU has a large speedup due to the abundant parallelism; other applications have loop-carried dependencies (reduction or prefix sum patterns) that limit performance improvement. On average, the proposed transformations achieved a 75X speedup.

Table 6: Effect of the proposed code transformations (unit: cycles)

	Baseline	+Part Unr	+Var Rdct	+Var PSum
Cholesky	404M (1.0X)	-	14.2M (28X)	-
LU	942M (1.0X)	2.96M (317X)	-	-
Trisolv	2.50M (1.0X)	-	63.9K (39X)	-
RotIntImg	11.1M (1.0X)	1.61M (6.9X)	-	246K (45X)
Durbin	5.25M (1.0X)	-	-	522K (10X)
Dynprog	872M (1.0X)	-	-	98.8M (8.8X)

For comparison, we obtained access to the source code for two of the latest DSE works, AutoAccel [6] and COMBA [27], and produced the output for the same benchmarks. We adjusted the parameters in AutoAccel and COMBA to match the characteristics of the KU060 FPGA. Since COMBA does not provide code on how to unroll variable bound loops, we applied the conventional unrolling with maximum bound (Section 3.1) with the unrolling factor instructed by the tool. COMBA had a tendency to over-unroll the loops beyond the given resource threshold—probably because its resource estimation is mostly based on the operators only. In this case, we manually reduced the unrolling factor to fit the given threshold. For the LU benchmark, Vivado HLS was unable to unroll the loops (as discussed in Section 3.2), and achieved no speedup when using the configuration suggested by COMBA. Similarly, for prefix sum patterns, the unroll directive infers fixed-length serialized addition (Section 3.4) which does not improve the performance.

The performance comparison is shown in the "Exec Time" column of Table 7. Our framework outperforms COMBA and AutoAccel by 78X and 32X on average. The main reason is that COMBA and AutoAccel do not perform code transformation that can solve the PE efficiency problem of the variable loops (Section 3). Thus, the design space explored by these tools is limited and results in a relatively worse performance. Another reason is that their cycle estimation model does not properly consider variable loop bounds.

6.3 Exploration Speed and Prediction Accuracy

The execution time, DSE result, and the prediction error rates are shown in Table 7. The execution times of the variable loops are measured using the method proposed in [4], and the resource usage is compared to the Vivado HLS synthesized result.

The result shows that the number of HLS synthesis performed is on average only 23% of the entire design spaces explored. The performance and the resource consumption of the rest of the design points are estimated using the model presented in Section 4. Thus, the exploration time is maintained at a few hundreds of seconds.

The table also shows that the prediction error rate of execution time, DSP, FF, LUT, with the proposed model is on average 12%, 0%, 5.1%, 5.7%, respectively. Such a low error rate helps the DSE process find the best design point accurately.

The exploration time and the prediction accuracy for AutoAccel and COMBA is also shown in Table 7. The execution time error of COMBA is probably caused by the mispredicted II and IL compared to the actual Vivado HLS synthesized result. This is due to the fact that COMBA does not reference the HLS report. On the other hand, AutoAccel does refer to the HLS report—however, its execution time is also not very accurate. The reason is that Vivado HLS reports the cycle based on the maximum of the variable loop. The resource estimated for AutoAccel is the result given by the HLS tool itself, and thus has an error rate of 0%.

7 CONCLUSION

Optimization of variable loop bounds with conventional HLS directives for pipelining and unrolling often leads to a low PE utilization problem. We have shown that techniques such as partial unrolling with pipelining or loop early termination will help solve this problem. HLS-based code transformations were devised to demonstrate how these techniques can be applied to common computational patterns. Also, we have proposed a resource estimation method that models operator sharing with a small number of HLS syntheses. The experimental result shows that a 75X speedup was achieved compared to the baseline implementation. As a future work, we

Table 7: Comparison of the performance, design space exploration speed, and the prediction accuracy among proposed, COMBA, and AutoAccel flows (the performance and the prediction error rates are that of the final output design)

Application	Flow	Exec Time	Design Space Exploration			Prediction Error Rates			
		(cycles)	# Design	# HLS Runs	Expl Time	Exec Time	DSP	FF	LUT
Cholesky	Proposed	14.2M	100	11	625s	12%	0%	7.7%	1.3%
	COMBA	21.1M	NA	0	872s	-86%	NA	NA	NA
	AutoAccel	180M	32	NA	252s	500%	0%	0%	0%
LU	Proposed	2.96M	100	11	451s	-3.7%	0%	0.89%	-5.5%
	COMBA	1.01B	NA	0	416s	-99.9%	NA	NA	NA
	AutoAccel	403M	16	NA	145s	201%	0%	0%	0%
Trisolv	Proposed	63.9K	10	5	232s	14%	0%	-1.7%	-0.85%
	COMBA	103K	NA	0	431s	-98%	NA	NA	NA
	AutoAccel	2.10M	32	NA	131s	101%	0%	0%	0%
RotIntImg	Proposed	247K	1000	20	1370s	-29.0%	0%	-18%	-20%
	COMBA	21.5M	NA	0	1,490s	-99.9%	NA	NA	NA
	AutoAccel	2,13M	32	NA	103s	98%	0%	0%	0%
Durbin	Proposed	522K	10	5	245s	-2.5%	0%	1.2%	4.4%
	COMBA	8.12M	NA	0	2,540s	-99%	NA	NA	NA
	AutoAccel	1.08M	32	NA	203s	96%	0%	0%	0%
Dynprog	Proposed	98.8M	64	9	436s	10%	0%	0.54%	2.4%
	COMBA	1.87B	NA	0	2,200s	-99.9%	NA	NA	NA
	AutoAccel	234M	32	NA	121s	454%	0%	0%	0%
Average	Proposed	1.0X	-	-	-	12%	0%	5.1%	5.7%
	COMBA	78X	-	-	-	97%	NA	NA	NA
	AutoAccel	32X	-	-	-	241%	0%	0%	0%

are considering to support more patterns for loops with variable bounds beyond those in the Polybench benchmarks.

ACKNOWLEDGMENTS

This research is partially supported by the NSF/Intel Partnership on Computer Assisted Programming for Heterogeneous Architectures (CAPA) Award #1723773 and NSF/Intel InTrans Award #20134321. We thank Cody Hao Yu (UCLA) and Jieru Zhao (HKUST) for providing their source code and assisting with the experiments. We also thank Professor Miryung Kim, Muhammad Gulzar, Peng Wei, Jie Wang (UCLA), and Dr. Peng Zhang (Falcon Computing) for motivating this research and the helpful discussions. We also thank Janice Wheeler for proofreading this manuscript.

REFERENCES

[1] Amazon. 2018. Amazon EC2 F1 Instance. (2018). https://aws.amazon.com/ec2/instance-types/f1/

[2] R. Chandra, *et al.* 2001. *Parallel Programming in OpenMP.* Morgan Kaufmann, San Francisco, CA.

[3] Y. Choi and J. Cong. 2017. HLScope: High-Level performance debugging for FPGA designs,. In *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM'17).* 125–128.

[4] Y. Choi, P. Zhang, P. Li, and J. Cong. 2017. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *Proc. IEEE/ACM Int. Conf. Computer Aided Design (ICCAD'17).* 691–698.

[5] N. Chong, A. Donaldson, and J. Ketema. 2014. A sound and complete abstraction for reasoning about parallel prefix sums. *ACM SIGPLAN Notices* 49, 1 (2014), 397–409.

[6] J. Cong, P. Wei, C. Yu, and P. Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *Proc. Ann. Design Automat. Conf. (DAC'18).* 154–159.

[7] J. Cong, P. Wei, C. Yu, and P. Zhou. 2017. Bandwidth optimization through on-chip memory restructuring for HLS. In *Proc. Ann. Design Automat. Conf. (DAC'17).*

[8] J. Cong and Y. Zou. 2010. A comparative study on the architecture templates for dynamic nested loops. In *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM'10).* 251–254.

[9] J. Cong, *et al.* 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[10] Alpha Data. 2017. Alpha Data ADM-PCIE-KU3 Datasheet. (2017). http://www.alpha-data.com/pdfs/adm-pcie-ku3.pdf

[11] M. Harris, S. Sengupta, and J. Owens. 2008. Parallel Prefix Sum (Scan) with CUDA. (2008). https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html

[12] Intel. 2018. Intel FPGA SDK for OpenCL Pro Edition. (2018). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf

[13] Intel. 2018. Intel HLS Compiler. (2018). https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html

[14] R. Kastner, M. Matai, and S. Neuendorffer. 2018. Parallel Programming for FPGAs. *ArXiv E-prints* (2018). http://kastner.ucsd.edu/hlsbook/

[15] D. Koeplinger, *et al.* 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proc. Int. Symp. Computer Architecture (ISCA'16).* 115–127.

[16] P. Kogge and H. Stone. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers* 100, 8 (1973), 786–793.

[17] P. Li, P. Zhang, L. Pouchet, and J. Cong. 2015. Resource-aware throughput optimization for high-level synthesis. In *Proc. Int. Symp. Field-Programmable Gate Arrays (FPGA'15).* 200–209.

[18] R. Lienhart, A. Kuranov, and V. Pisarevsky. 2003. Empirical analysis of detection cascades of boosted classifiers for rapid object detection. In *Joint Pattern Recognition Symp.* 297–304.

[19] Microsoft. 2018. Microsoft Azure. (2018). https://azure.microsoft.com/

[20] G. Morris and V. Prasanna. 2005. An FPGA-based floating-point Jacobi iterative solver. In *Proc. Int. Symp. Parallel Architectures, Algorithms and Networks (ISPAN'05).*

[21] L. Pouchet. 2015. PolyBench/C. (2015). http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[22] R. Prabhakar, *et al.* 2016. Generating configurable hardware from parallel patterns. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Vol. 50. 651–665.

[23] Y. Shao, *et al.* 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. Int. Symp. Computer Architecture (ISCA'14).* 97–108.

[24] M. Tan, *et al.* 2015. Elasticflow: A complexity-effective approach for pipelining irregular loop nests. In *Proc. IEEE/ACM Int. Conf. Computer Aided Design (ICCAD'15).* 78–85.

[25] Xilinx. 2018. UltraScale architecture and product data sheet: overview (DS890). (2018). https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[26] Xilinx. 2018. Vivado High-level Synthesis UG902. (2018). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf

[27] J. Zhao, *et al.* 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proc. Int. Conf. Computer Aided Design (ICCAD'17).* 430–437.

[28] G. Zhong, *et al.* 2016. Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *Proc. Ann. Design Automat. Conf. (DAC'16).* 136–141.

[29] L. Zhou and V. Prasanna. 2005. Sparse matrix-Vector multiplication on FPGAs. In *Proc. Int. Symp. Field-Programmable Gate Arrays (FPGA'05).* 63–74.

[30] H. Zohouri, *et al.* 2016. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC'16).* 409–420.