Design Space Exploration of FPGA-based Accelerators with Multi-level Parallelism

Guanwen Zhong*, Alok Prakash†, Siqi Wang*, Yun Liang‡, Tulika Mitra* and Smail Niar§
*School of Computing, National University of Singapore; †SCSE, Nanyang Technological University;
‡School of EECS, Peking University, China; §LAMIH, University of Valenciennes, France
Email: {guanwen,siqi,tulika}@comp.nus.edu.sg, alok@ntu.edu.sg, ericlyun@pku.edu.cn, smail.niar@univ-valenciennes.fr

***Abstract*—Applications containing compute-intensive kernels with nested loops can effectively leverage FPGAs to exploit fine- and coarse-grained parallelism. HLS tools used to translate these kernels from high-level languages (e.g., C/C++), however, are inefficient in exploiting multiple levels of parallelism automatically, thereby producing sub-optimal accelerators. Moreover, the large design space resulting from the various combinations of fine- and coarse-grained parallelism options makes exhaustive design space exploration prohibitively time-consuming with HLS tools. Hence, we propose a rapid estimation framework, MPSeeker, to evaluate performance/area metrics of various accelerator options for an application at an early design phase. Experimental results show that MPSeeker can rapidly (in minutes) explore the complex design space and accurately estimate performance/area of various design points to identify the near-optimal (95.7% performance of the optimal on average) combination of parallelism options.**

I. Introduction

The flexibility offered by Field Programmable Gate Arrays (FPGAs) enables designers to create application-specific systems that achieve high performance within the strict time-to-market and non-recurring engineering constraints. However, the formidable programming effort needed to achieve efficient FPGA designs continues to be a significant hurdle in its mass adoption. High-Level Synthesis (HLS) technology has been developed in the recent years to ease the programming effort. HLS tools can accept application specification in high-level programming languages (such as C, C++, SystemC) and directly target FPGAs without the need for time-consuming manual register-transfer level (RTL) creation. However, achieving an efficient design using HLS tools is still a challenging proposition. The HLS tools provide various optimization pragmas such as loop unrolling, loop pipelining and array partitioning. The designer is responsible for exploring the large number of potential design choices available for an application with these pragmas while optimizing for performance and/or area constraints. Unfortunately, the runtime of the HLS tools is prohibitively large to exclude the possibility of exhaustive design space exploration (DSE), especially for larger and complex applications. The existing research in this area has focused on pruning the design space before invoking the HLS tools for the final synthesis step [12][14].

Given an application kernel to be synthesized on FPGAs, there are two distinct scenarios with respect to the working set of the data. In the first scenario, the working set of the kernel can fit into on-chip storage (BRAM) and the kernel can be synthesized as single processing engine (referred to as PE hereafter) leveraging *fine-grained parallelism* via HLS pragmas. The downside of this approach is reduced performance if the loop kernel does not have enough fine-grained parallelism that can be easily exploited by HLS tools. The second scenario involves kernels where the working set is too large to be accommodated on-chip. A commonly deployed strategy is *Loop Tiling* where the program is transformed (either by programmer or compiler) to partition the iteration space of the loop into smaller blocks (tiles) such that the working set corresponding to a tile can easily fit in BRAM. *Loop Tiling* explicitly exposes *coarse-grained parallelism* as different tiles are independent. The parallelism can be exploited by instantiating multiple PEs within resource budget to process the tiles in parallel, while leveraging fine-grained parallelism inside a PE with pragmas. This multi-level parallelism is crucial to improved accelerator performance [9][12] especially when individual PEs have limited fine-grained parallelism.

Clearly, the design space, already quite large, explodes with multi-grained parallelism. Existing pre-HLS DSE works mentioned earlier, focused on exploring either the fine-grained or the coarse-grained parallelism, in isolation. The works attempting to reap the benefit of multi-level parallelism [9][12] rely on HLS tools to explore the various combinations of fine-grained and coarse-grained parallelism to arrive at the optimal setting. The runtime of HLS tools makes it difficult to navigate the massive design space. Moreover, apart from loop unrolling, other fine-grained parallelism opportunities such as array partitioning and loop unrolling are ignored in the process. We observe that these additional options, if exploited prudently, can improve the performance even further.

We propose a high-level analysis framework, MPSeeker, that considers both fine- and coarse-grained parallelism on FPGAs to estimate accelerator performance and resource requirement from sequential code (C/C++) without invoking HLS. The analysis enables rapid DSE (in minutes) with various parameters such as tile size, number of PEs, loop unrolling, loop pipelining and array partitioning, to identify the optimal parallelism configuration. This chosen configuration is passed on to the HLS tool for the final accelerator synthesis.

Our work builds on our recent performance analysis tool Lin-Analyzer [15] that explores *only* fine-grained parallelism without invoking HLS. As coarse-grained parallelism is neglected, Lin-Analyzer is restricted to single-PE design per loop kernel. The likely resource bottleneck (if any) for single-PE

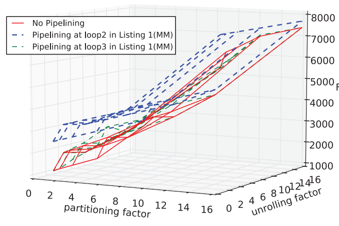

Fig. 1: Nonlinearity in FF usage with different pragmas

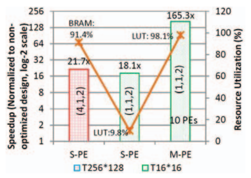

Fig. 2: Single vs. Multi. PEs

HLS-based	MPSeeker
35 hours	2.7 minutes

TABLE I: MM DSE Time

designs is the BRAM that accommodates the entire working set and estimating BRAM demand is straightforward. Thus Lin-Analyzer only needs to focus on performance estimation of a design point as long as its working set can fit into BRAM.

The multi-level parallelism considerations bring in immense additional complexity to the DSE process. The key question is the trade-off between fine-grained and coarse-grained parallelism. Opening up more parallelism per PE restricts the number of PEs that can be synthesized due to resource budget and vice versa. More importantly, as coarse-grained parallelism is exposed through loop tiling, BRAM storage no longer remains the bottleneck. Instead the datapath resources (LUTs, FFs, and DSP blocks) become more precious as the same datapath is replicated many times over with multiple PEs. However, estimating the LUT and FF requirement of a PE without HLS is hugely challenging due to the non-linear relationship between the program features/parallelism settings and resource demands (see Figure 1). **We propose a novel resource estimation model based on a machine learning techniques called Gradient Boosted Machine (GBM) that perfectly captures the resource demands given inherent program features and the parallelism options.** Experiments show that our resource estimation model is accurate achieving 12.7%, 19.8%, 14.7% and 13.2% average error for DSP, BRAM, FF and LUT prediction, respectively. Integrating this new resource model with the existing performance model empowers us to traverse through the complex multi-grained parallelism terrain in the order of minutes to select the optimal configuration under the resource constraints. Experimental results confirm that our DSE approach can make near-optimal recommendation for a range of application kernels.

II. Motivating Example

Listing 1: Tiled Matrix Multiplication Example

```
1  loopTop1: for(t1=0;t1<floor(1024/TS);++t1){
2    loopTop2: for(t2=0;t2<floor(1024/TS);++t2){
3      ... // Initialization
4      loopPE: for(t3=0;t3<floor(1024/TS);++t3){
5        // Step 1: Copy data from DDR to local memory(A, B and C);
6        memcpy(ddr_a,A,offsetA);memcpy(ddr_b,B,offsetB);
7        // Step 2: Kernel Computation
8        loop1: for(t4=0;t4<TS;++t4){
9          loop2: for(t5=0;t5<TS;++t5){
10           loop3: for(t6=0;t6<TS;++t6){
11             C[t4][t5] += A[t4][t6]*B[t6][t5]; }}}}
12     // Step 3: Write data from local memory to DDR
13     memcpy(C,ddr_c,offsetC);
14 }}
```

The complexity of multi-level parallelism extraction can be illustrated with the tiled matrix multiplication (MM) kernel example in Listing 1 with 1024×1024 matrix size and $TS \times TS$ tile size. The yellow highlighted portion is the key computation corresponding to a tile. The data needed for each tile is brought into the local BRAM from off-chip memory (line 6) and written back to off-chip memory (line 13) after the computation for the tile is completed. The tile can be converted into an accelerator or PE. In single-PE design (*S-PE*), the tiles will be executed sequentially whereas in multi-PE design (*M-PE*), multiple PEs are instantiated to be executed in parallel.

Figure 2 shows the comparison between the two. S-PE design explores array partitioning, loop unrolling, pipelining whereas M-PE design additionally includes tile size, number of PEs. The Y-axis denotes performance speedup of the optimal S-PE (red bar 21.7x) and M-PE design (green bar on the right 165.3x) normalized to the unoptimized design (no exploited parallelism). Tuples in Figure 2 are pragma combinations: (*partitioning factor*, *unrolling factor*, *pipelining loop level*).

The optimal S-PE chooses the biggest tile size (256×128) that can fit into on-chip storage (91.4% BRAM usage), while M-PE selects 16×16 tile size instead and instantiates 10 PEs. This design is constrained by LUTs (98.1% LUT usage) rather than BRAM. M-PE achieves 165.3x speedup compared to 21.7x for S-PE by exploiting multi-level parallelism. In order to investigate the rationale behind this result, we look at the speedup of S-PE with 16×16 tile size (middle green bar in Figure 2). Surprisingly, 16×16 S-PE shows comparable speedup to 256×128 S-PE. This is because of dependencies within the inner loop that limit exploitable fine-grained parallelism even as tile-size increases. Hence, it is far more profitable to restrict tile size and instantiate more PEs for this kernel.

Table I shows that it takes 35 hours to perform exhaustive HLS-based DSE considering multi-level parallelism design space (280 points). In contrast, our high-level rapid DSE tool MPSeeker explores the complex design space in under 3 minutes and recommends the same optimal design point.

III. Related Work

To estimate FPGA performance and area, many works [5][9][11][15] start from high-level specifications (C/C++) and focus on exploiting fine-grained parallelism by accelerating the kernel on single PE. Hence, they do not exhaustively exploit the capabilities of FPGAs that can support diverse types of parallelism. Additionally, many of them are based on static analysis[5][9], which suffer from inherently conservative dependence analysis. This typically leads to false dependencies between operations, thereby limiting the available parallelism that can otherwise be exploited by FPGA-based accelerators.

Authors in [9] proposed a framework from C/C++ exploiting multi-level parallelism. They used an analytical method based on HLS results as inputs for area/performance estimation. However, they only considered loop pipelining and ignored resources consumed by registers and multiplexers. Moreover, with only loop pipelining enabled, fine-grained parallelism within a PE is not fully explored. Instead of taking sequential high-level specifications, [12][13] correlate parallel programming languages (OpenCL and CUDA) with FPGA performance. As OpenCL and CUDA explicitly expose massive parallelism of applications, their works can easily exploit multi-level parallelism on FPGAs. [13] proposed a performance

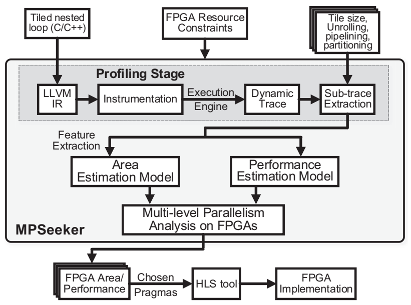

Fig. 3: MPSeeker: An Automatic Design Space Exploration Framework with Multi-level Parallelism

model starting from OpenCL, while [12] starts from CUDA. As [13] does not support area model, their method relies on designers to decide number of PEs and cannot perform automatic DSE. [12] relies on HLS to provide performance of a PE and might suffer from potential long HLS runtime, especially when design space becomes more complex. [7] proposed a template-based approach from domain specific language. Their method uses as large number of memory banks as possible, which incurs extensive BRAM usage. It might reduce number of PEs and end up with sub-optimal designs.

Several works [5][12][14] proposed area estimation using simple regression models with a small design space. As the design space grows more complex, FPGA area shows the non-linear behavior that makes the previous models no longer suitable for prediction. Therefore, Quipu [11] proposed statistical methods to capture the non-linear behavior. However, as the inherent program features collected in their tool are irrelevant to HLS pragmas, Quipu cannot model area consumption according to various pragmas. In contrast, we try to identify program features that are relevant to pragmas and propose an FPGA area model utilizing Gradient Boosted Machine (GBM).

IV. MPSeeker Framework

Overview: The proposed MPSeeker framework, shown in Figure 3, takes a high-level specification (C/C++) of an algorithm in the form of nested loops (with loop tiling enabled), available pragmas (tile size, unrolling, pipelining and array partitioning) and FPGA resource constraints as inputs. In the *Profiling Stage*, MPSeeker converts the input code into an intermediate representation of Low-Level Virtual Machine (LLVM IR) [8] and then instruments the IR for trace collection. With *Execution Engine*, an LLVM Just-in-Time (JIT) compiler, the instrumented IR is executed to obtain a dynamic trace that includes information such as instruction opcodes, basic block frequencies and memory (load/store) addresses. Since a trace contains many instruction instances, MPSeeker extracts an interesting *sub-trace* for analysis according to given pragmas.

With generated *sub-trace*, we exploit techniques in Lin-Analyzer [15] for performance prediction. As [15] only predicts kernel computation, we extend it with data communication cost to predict the accelerator performance (clock cycles) on a single PE with various pragma combinations. [15] also estimates the DSP and BRAM usage of a PE, which we reuse in our work. However, to exploit multi-level parallelism, FF

TABLE II: List of Program Features

No.	Feature	Range	Description
1	ts	$\{8, 16, 32, ..., 256\}$	Tile size. Currently, we only consider square tile.
2	$nLpL$	$[1, \infty)$	Number of loop levels.
3	$instn$	$(1, \infty)$	Total number of instructions
4	$fpRt$; $intRt$; $bwRt$; mRt; $brRt$	$[0, 1)$	$fpRt$: Percentage of floating-point instructions (fadd, fsub, fmul, fdiv, etc.); $intRt$: Percentage of integer instructions (add, sub, mul, div, modulo, etc.); $bwRt$: Percentage of bit-wise instructions (xor, nor, and, shift, etc.); mRt: Percentage of memory load/store instructions; $brRt$: Percentage of branch instructions
5	$avePar$	$[1, \infty)$	Average number of independent operations that can be executed in parallel
6	uf	$\leq$ loop bound	Loop unrolling factor. For area efficiency, we select divisors of loop bounds.
7	pl	$[0, k]$	For a nested loop $L = L_1, L_2, ..., L_k$ (k is the innermost level), pipelining can be applied at any $L_i, i \in [1, k]$. $pl = 0$ means pipelining disabled.
8	upf	$[1, \infty)$	Unified array partitioning factor factor
9	IL	$[1, \infty)$	Latency of a kernel returned by a scheduler
10	II	$[0, \infty)$	Initiation interval (II) of a kernel, where II is defined as the interval between the start of consecutive loop iterations.
11	$mAcc$	$[1, \infty)$	maximum number of accesses per memory bank among arrays in a kernel
12	$mALd$	$(0, \infty)$	We collect information about average number of load accesses per memory bank of all arrays and select the maximum value from them as *mALd*
13	$mASt$	$(0, \infty)$	We collect information about average number of store accesses per memory bank for all arrays and select the maximum value from them as *mASt*
14	$tASize$	$[1, \infty)$	Total array size in bytes. It includes all arrays in a kernel.

and LUT usage become more crucial as mentioned in Section I. A machine learning approach, gradient boosted machine (Section IV-B), is employed to predict FF and LUT usage of a PE. Using the predicted resource usage of single PE, we calculate the maximum number of PEs within resource budget.

The DSE step is performed by estimating both performance and area of a PE for each pragma combination. Finally, MPSeeker recommends the best suited configuration for the kernel considering multi-level parallelism provided by instantiating multiple PEs. Lastly, the HLS tool is invoked with the selected pragmas to generate the final synthesized accelerator.

A. Performance Estimation of a PE:

FPGA performance T_{PE} of a PE is modeled as a sum of kernel computation cost T_{comp} and data communication cost T_{comm}. We use the concepts from [15] to obtain T_{comp} on a single PE with given pragma combinations. T_{comm}, on the other hand, is predicted by a new linear model proposed here.

1) Computation Estimation: Lin-Analyzer is a recently proposed high-level FPGA performance estimation tool based on dynamic analysis. It creates the corresponding dynamic data dependence graph (DDDG) from the generated *sub-trace* and performs essential optimizations. Then, it schedules nodes on the optimized DDDG within resource budget (DSP and BRAM) to obtain an early performance estimate of kernel computation cost. In this work, we leverage Lin-Analyzer to estimate kernel computation cost, T_{comp}.

2) Data Communication Estimation: We assume that each PE on FPGAs serves as a master accelerator that can request and transfer data by itself. Transaction utilizes memory burst mode, which can transfer up to 256 bytes data per memory request. To enable data transfer, designers need to manually insert $memcpy$ function to original code as shown in Listing 2. *copy* and $write$ functions are used to transfer data between main memory (i.e. DDR) and local storage (i.e. BRAM).

Listing 2: Data Communication of MM kernel

```
/* T: TYPE; TS: Tile size */
/* Copy data from main memory (DDR) to local memory */
void copy(...) {
  #pragma HLS pipeline
  memcpy(local_addr1,ddr_addr1,offset1);
  memcpy(local_addr2,ddr_addr2,offset2);}
/* Copy data from local memory to main memory (DDR) */
void write(...){ similar to copy function; }
```

Assume we have n arrays $A = \{A_1, A_2, ..., A_n\}$ to be transferred in function type t where $t \in \{copy, write\}$. b_{A_i} is number of bytes in array A_i. We define α to be the cost for transferring one byte data and $\beta_t, t \in \{copy, write\}$ to be the extra cost to set up communication in function type t. Based on empirical method, we use $\alpha = 0.25$ and $\beta_{copy} = 6, \beta_{write} = 4$. These empirical values are collected via Vivado HLS and we have tested for different Xilinx devices, such as Virtex and ZYNQ7000 series. Then the communication time T'_{comm_t} of function type t is calculated by the following Equation 1,

$$T'_{comm_t} = \max_{i \in [1,n]} (\alpha b_{A_i}) + \beta_t \tag{1}$$

Communication time T_{comm_t} of a PE is related to T'_{comm_t} and number of invocations IV_t of function type t,

$$T_{comm} = \sum_{t \in \{copy, write\}} T'_{comm_t} * IV_t \tag{2}$$

B. Area Estimation of a PE:

As mentioned in Section IV, we use Lin-Analyzer [15] to predict DSP and BRAM usage, while leverage a machine learning method for FF and LUT estimation. Figure 1 plots FF usage of MM kernel while varying unrolling, partitioning factors and pipeline levels. The plot shows non-linear relationship between FF usage and different pragma settings. LUT consumption exhibits a similar behavior. These features make Gradient Boosted Machine (GBM) to be a good solution to capture the correlation between FF/LUT consumption and program features. We also explored several techniques including multivariate polynomial regression, step-wise regression and neural network. We found that GBM consistently outperformed other techniques. Hence we employ GBM to predict FF and LUT usage of a PE.

GBM [6] technique consists of a gradient-based optimization and a boosting step that is typically used in regression or classification problems. Gradient-based optimization tries to minimize a loss function of a training model with gradient computations, while boosting collects an ensemble of tree models, to generate a solid learning system for inferring.

1) **Feature Definition***:* It is challenging to determine a list of program features that can be used in GBM to predict the FPGA area as there is no algorithmic way to find a proper set of features [11]. We started from a set of features (Number 1-4 in Table II) used in [11]. HLS tools can employ unroll and pipelining pragmas to exploit inherent parallelism in a kernel; however, they might incur extensive resource consumption in absence of memory bandwidth constraint. HLS tools typically use BRAM or distributed RAM to store data. As they have limited number of read/write ports, low memory bandwidth usually prohibits HLS tools from instantiating more functional units even if there is plenty of parallelism inside a kernel. By increasing partitioning factor, HLS tools can exploit more parallelism in the kernel and require more resources. Based on these intuitions, we define a set of plausible features that might have a great impact on area consumption as listed in Table II. It should be noted that bit width of operations also has impact on resource consumption. However, in current implementation, we focus on single-precision floating-point kernels; the bit width is set to be 32. Therefore, we do not include bit width in Table II. In future, we will study the effect of bit width.

TABLE III: Key hyperparameters of the GBM models

	FF	LUT	Description
ntrees	125	150	number of trees to grow
max_depth	10	8	maximum depth to grow the tree
nbins	15	15	minimum number of bins in a built histogram
distribution	gamma	gamma	distribution function of the response
learn_rate	0.1	0.16	learning rate

$avePar$: As FPGAs can exploit diverse types of parallelism within applications, we define $avePar$ to capture degree of parallelism in kernels. We reimplement the algorithm in [1] on a generated *sub-trace* to calculate the average number of independent operations as $avePar$ that can be executed in parallel. With larger *avePar*, HLS typically generates higher performance accelerators at the cost of more area.

$mAcc$, $mALd$ and $mASt$: These three features are related to the memory bandwidth requirement of kernels and hence affect partitioning pragma. Without loss of generality, we assume that each memory partition has two read ports and one write port. Designers can leverage partitioning pragma to split arrays into multiple banks so that several memory accesses can be executed simultaneously. We map addresses of load/store instructions in the *sub-trace* to bank IDs according to partitioning factors and collect information, such as number of accesses per bank, to describe memory access features. Larger values of *mAcc, mALd* and *mASt* mean that more data shares the same memory resource and HLS needs to allocate more FPGA resources (i.e., wider multiplexers).

IL and II: Iteration latency (*IL*) and initiation interval (*II*) of a kernel are relevant to optimization pragmas and sub-traces extracted. With proper pragmas, HLS can generate good-quality designs with lower *IL* or *II* and exploit parallelism of the kernel by either increasing memory bandwidth or duplicating functional units at the cost of more area consumption.

upf: Each array A_i in a kernel can be implemented with partitioning factor pf_i, $i \in [1, n]$, where n is number of arrays. In this work, we utilize a unified partition factor upf, which weights partitioning effect on arrays. upf is calculated by $\sum_i^n w_i * pf_i$, where w_i is a weight of A_i. A weight w_i is defined by the ratio of number of memory accesses (load/store) on A_i to the total memory accesses across all arrays.

2) **GBM Implementation Details***:* A GBM is an ensemble of tree-based models and it has a set of configurations that have great impact on model quality. These configurations, for example number of trees to grow, are called *hyperparameters*. We utilized *Cartesian Hyperparameter Grid Search* (GS)[4] to automatically choose *hyperparameter* combinations that leads to the highest quality. Table III lists values of the key parameters after grid search. To validate and improve prediction quality of GBM, we leveraged *K-Fold Cross-Validation* (CV) [6]. The method splits data into K equal-size partitions. It chooses the k-th ($k = 1, 2, ..., K$) partition as a validation set with the remaining K-1 partitions as a training set and calculates the prediction error of the trained model. This process repeats K times and combines the K prediction errors. In this work, we selected *5-Fold Cross Validation*.

For implementations of GBM, GS and CV, we leveraged H_2O package [4] in R environment. The training and testing sets used are mutually exclusive and detailed in Section V.

TABLE IV: Design Options used in Our Work

Options	Range
Tile Size	[16x16, 32x32, 64x64, 128x128]
Loop unrolling factor	$\leq$ loop bound N
Loop pipelining	$[0, k]$, as mentioned in Row 7, Table II
Array partitioning	factor: [1::2::16] (the set of values from 1 to 16 in steps of 2) type: {cyclic, block, complete}

C. Putting It All Together

After obtaining the performance and area estimates for single PE that exploits various fine-grained parallelism, we proceed to calculate the maximum number C of PEs that can be instantiated and run simultaneously on the FPGA under a given resource constraint. This is modeled by Equation 3 as:

$$C = \min_{tp} \left\lfloor \frac{RES^{tp}_{max}}{Res^{tp}_{PE}} \right\rfloor, tp \in \{DSP, BRAM, FF, LUT\} \tag{3}$$

where RES^{tp}_{max} is the maximum available resource of type tp for a given device, Res^{tp}_{PE} is resource of type tp required by a PE.

Total execution time of an FPGA-based accelerator exploiting multi-level parallelism is calculated by Equation 4,

$$T_{FPGA} = \frac{N}{C} * T_{PE} \tag{4}$$

where N is the total number of invocations of a single PE.

V. Experimental Results

We use the Xilinx ZC702 Evaluation Kit [18] along with Vivado HLS version 2015.2 for experiments. Accelerators are set to run at 100MHz. Experiments are conducted on a PC with an Intel Xeon CPU E5-2620 core at 2.10Hz with 64GB RAM, running Ubuntu 14.04 OS. We select mutually exclusive training and testing sets as described below.

Training Set: We explore nested loops in several benchmark suites including Polybench[17], MachSuite[3], Stream[10], Livermore Loops[16] and testing cases in Pluto[2]. 34 kernels are selected from these benchmark suites. We also developed 10 microbenchmarks to examine resource consumption of loop structures with different bounds, communication interfaces, operation units, etc.. Thus, we used 44 kernels in total. These kernels are modified with functions in Listing 2 for data communication. For training, we invoke the proposed tool for feature extraction and Vivado HLS for obtaining performance and area information per design point. The various pragma combinations (loop unrolling, pipelining and array partitioning) lead to a total of 3244 design points in the final data set. We randomly selected 80% data points from this set for training while the rest were used for validation.

Testing Set: As we consider large data size and exploit multi-level parallelism, testing kernels should be tiled. Therefore, we selected five applications: *DCT1D, MM* from Xilinx[18] and *DERICHE1, GEMVER1, MVT* kernels from Polybench[17] as the testing kernels. *DCT1D* and *MM* have three nested loop levels, while *DERICHE1*, *GEMVER1* and *MVT* have two loop levels. The input data size of *MM, DCT1D, DERICHE1* is 1024x1024, while we use 2048x2048 in *GEMVER1* and *MVT*. As large data cannot fit into FPGA devices, we performed loop tiling for kernels with Pluto[2] and inserted functions in Listing 2 for data communication. We consider square tiles. Loop tiling explicitly exposes coarse-grained parallelism to HLS and also avoids exceeding BRAM resource.

TABLE V: Performance/Area Esti. Accuracy and DSE Time

Benchmark	Absolute value of the relative error (%)					DSE Time	
	DSP	BRAM	FF	LUT	Execution Cycle	HLS (hours)	MPSeeker (min.)
DCT1D	13.7	15.3	14.5	13.4	11.1	31.7	3.0
DERICHE1	6.4	18.0	12.2	15.7	7.5	26.7	0.6
GEMVER1	20.0	25.8	19.6	14.5	15.8	23.2	3.3
MM	4.8	16.1	14.9	10.5	16.1	35.9	0.5
MVT	18.8	23.8	12.3	12.1	13.5	22.9	0.6
Average	12.7	19.8	14.7	13.2	12.8	28.1	1.6

The design space of the testing data set considered is shown in Table IV. With the design options, each kernel in the testing set has 280 design points.

A. Performance/Area Estimation Accuracy

To evaluate estimation accuracy of the proposed model, we calculate the relative error between the actual (by Vivado HLS) and predicted results: $RelativeError = |Actual - Predicted|/Actual * 100$.

FPGA performance (execution cycle) prediction: The proposed model can generally provide accurate execution cycle estimation. However, we observed some notable discrepancies in a few design points when applying the pragmas in Table IV to Vivado HLS. The reasons and extra optimizations enabled in Vivado HLS are summarized below.

- Static Analysis in Vivado HLS: As mentioned earlier, this might add false loop-carried dependence between operations when considering partitioning pragma and limit the exploitable parallelism of a kernel [15]. To resolve this, we enabled *set_directive_dependence* pragma in Vivado HLS to explicitly disable false dependences. As our approach is based on dynamic analysis, all the data dependences are known after obtaining the trace.
- Expression balance: In some cases (DERICHE1 and GEMVER1) with only unrolling enabled, Vivado HLS fails to perform expression balance optimization to decrease height of long expression chains, and hence omits potential parallelism [5] for floating-point operations. Thus, we explicitly unroll the original code when unrolling is enabled.

After incorporating these additional optimizations, the relative error, shown in Table V, ranges from 7.5% to 16.1% and average error is 12.8% across all the benchmarks considered, which demonstrates high accuracy of our performance model.

FPGA area prediction: Table V shows the accuracy of our area estimation model for single PE across the testing set kernels. Each kernel has a design space of up to 280 data points considering both fine- and coarse-grained parallelism. As shown in the table, our GBM model is able to predict FF and LUT consumption with high accuracy among testing set kernels over diverse design options and achieves an average relative error of 14.7% and 13.2%, respectively. DSP and BRAM usage predictions are extracted from Lin-Analyzer's scheduling stage and Table V shows an average error of 12.7% and 19.8%, respectively. Among these results, BRAM prediction has the highest average error that primarily comes from the unpredictable behavior of BRAM duplication in Vivado HLS while we calculate BRAM usage based on array size and its partitioning factor. Although the average error is relatively high, BRAM prediction still follows the actual trend.

B. Quality of the GBM Model

Figure 4 shows the training, validation and cross-validation (CV) errors of the GBM model for LUT prediction as a function of training set size. The FF prediction model shows similar behavior. The training error increases slightly with more training samples, while validation and CV errors decrease. With less training samples (<20% of training set), the gap between training and CV errors is large, indicating that the GBM model suffers from high variance. This issue can be resolved by sampling more training data [6] that reduces the original gap to roughly 5% when using the whole training set.

C. Rapid DSE with Multi-level Parallelism

Figure 5 shows the DSE results obtained from three methods, S-PE, Optimal and MPSeeker. S-PE shows the best performance with single PE (exploiting fine-grained parallelism only), while Optimal denotes the best performance with multiple PEs (exploiting both fine- and coarse-grained parallelism). Both S-PE and Optimal are obtained by the exhaustive HLS-based DSE approach and serve as the baseline. The results of MPSeeker are obtained by a single invocation of Vivado HLS with the suggested configurations as predicted by the proposed approach. The design space we considered here is described in Table IV. The left Y-axis in Figure 5 denotes performance speedup normalized to the optimal design of single PE, while the right Y-axis shows number of PEs in each design.

The performance of Optimal for all benchmarks is notably superior to that of S-PE. The best speedup is achieved for *MM*: 7.58x compared to S-PE. This confirms that considering fine-grained parallelism only might lead to sub-optimal designs.

Tuples in Figure 5 are configurations recommended by S-PE, Optimal and MPSeeker approaches that can achieve the best performance within area budget. The configuration format is (*tile width*, *partitioning factor*, *unrolling factor* and *loop pipelining*). As shown in Figure 5, the recommended configurations of Optimal and MPSeeker are identical for MM and MVT benchmarks. For DCT1D, DERICHE1 and GEMVER1 benchmarks, the suggested configurations are different compared to those of Optimal. Differences reside in *tile width* and *partitioning factor* for DCT1D and *unrolling factor* for DERICHE1, while for GEMVER1, it is in *tile width*. Although the suggested configurations from our proposed approach cannot exactly match the best suited configurations by exhaustive-HLS method, the FPGA performance with the suggested configurations can achieve 90.9%, 95.4% and 92.4% of the optimal performance for DCT1D, DERICHE1 and GEMVER1, respectively. On average, the configurations predicted by MPSeeker can achieve 95.7% of the optimal performance from exhaustive DSE for all the applications. This demonstrates that the proposed approach can identify the near-optimal set of parallelism options without exhaustive HLS-based DSE.

Training Time: The total training time to build GBM models is roughly 660 seconds. However, this is only one-time overhead. Additionally, the inferring time is negligibly small.

Table V compares the exploration time between the exhaustive HLS-based and proposed MPSeeker. The exploration time of our method includes the profiling overhead. Vivado HLS failed to synthesize some design points with very complex

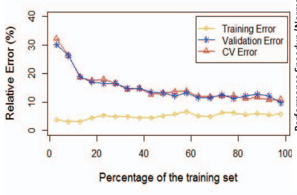

Fig. 4: Learning curves

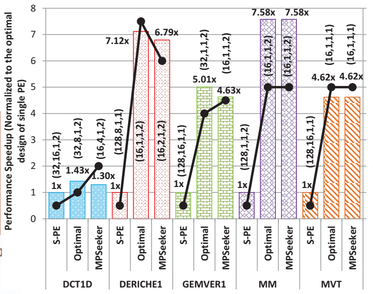

Fig. 5: DSE result.

pragma combinations and we removed their runtime when reporting its DSE time. As shown in Table V, our approach reduces the exploration time from hours or even days to a few minutes, while achieving near-optimal performance.

VI. Conclusion

This paper proposes MPSeeker, a rapid performance/area estimation framework, to evaluate FPGA-based accelerators at an early design stage. MPSeeker explores various fine-grained (loop pipelining, array partitioning and loop unrolling) and coarse-grained (loop tiling) parallelism options in an application without invoking HLS tools. Our results show that MPSeeker recommends same or closely similar combination of pragma settings in minutes instead of hours or sometimes even days taken by exhaustive HLS-based techniques.

VII. Acknowledgments

This work was partially funded by the Singapore Ministry of Education Academic Research Fund Tier 2 MOE2015-T2-2-088.

References

[1] Austin T. et al. Dynamic Dependency Analysis of Ordinary Programs. In *ISCA*, 1992.
[2] Bondhugula U. et al. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI*, 2008.
[3] Brandon R. et al. The MachSuite Benchmark. In *BARC*, 2015.
[4] Click C. et al. Gradient Boosted Models with H2O. 2016.
[5] Gao X. et al. Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis. In *FPGA*, 2016.
[6] Hastie T. et al. *The Elements of Statistical Learning*. Springer Series in Statistics. 2001.
[7] Koeplinger D. et al. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *ISCA*, 2016.
[8] Lattner C. et al. LLVM: a Compilation Framework for Lifelong Program Analysis Transformation. In *CGO*, 2004.
[9] Li P. et al. Resource-aware throughput optimization for high-level synthesis. In *FPGA*, 2015.
[10] McCalpin JD. et al. Memory Bandwidth and Machine Balance in Current High Performance Computers. In *TCCA*, 1995.
[11] Meeuws R. et al. Quipu: A Statistical Model for Predicting Hardware Resources. *TRETS*, 2013.
[12] Papakonstantinou A. et al. Multilevel Granularity Parallelism Synthesis on FPGAs. In *FCCM*, 2011.
[13] Wang Z. et al. A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs. In *HPCA*, 2016.
[14] Zhong G. et al. Design Space Exploration of Multiple Loops on FPGAs Using High Level Synthesis. In *ICCD*, 2014.
[15] Zhong G. et al. Lin-Analyzer: A High-level Performance Analysis Tool for FPGA-based Accelerators. In *DAC*, 2016.
[16] McMahon F.H. The Livermore Fortran Kernels: A Computer Test of The Numerical Performance Range. 1986.
[17] Pouchet L. PolyBench/C 4.1. http://web.cse.ohio-state.edu/~pouchet/software/polybench/.
[18] Xilinx Inc., 2015. www.xilinx.com.