S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters

Cody Hao Yu[1,2], Peng Wei[1], Max Grossman[4], Peng Zhang[2], Vivek Sarker[3], Jason Cong[1,*]

[1]University of California, Los Angeles [2]Falcon Computing Solutions, Inc.

[3]Georgia Institute of Technology [4]Rice University

{hyu,peng.wei.prc,cong}@cs.ucla.edu,pengzhang@falcon-computing.com,jmg3@rice.com,vsarkar@gatech.edu

ABSTRACT

Big data analytics using the JVM-based MapReduce framework has become a popular approach to address the explosive growth of data sizes. Adopting FPGAs in datacenters as accelerators to improve performance and energy efficiency also attracts increasing attention. However, the integration of FPGAs into such JVM-based frameworks raises the challenge of poor programmability. Programmers must not only rewrite Java/Scala programs to C/C++ or OpenCL, but, to achieve high performance, they must also take into consideration the intricacies of FPGAs. To address this challenge, we present S2FA (**S**park-**to**-**F**PGA-**A**ccelerator), an automation framework that generates FPGA accelerator designs from Apache Spark programs written in Scala. S2FA bridges the semantic gap between object-oriented languages and HLS C while achieving high performance using learning-based design space exploration. Evaluation results show that our generated FPGA designs achieve up to 49.9× performance improvement for several machine learning applications compared to their corresponding implementations on the JVM.

1 INTRODUCTION

Because the volume of datasets has grown dramatically in recent years, there has been a corresponding increase in the demand for scalable high-performance computing. In 2004 Google introduced MapReduce [12], a big data programming framework that enables efficient management of tens of thousands to millions of servers in datacenters with a simple programming model. Inspired by Google MapReduce, open source big data analytic systems such as Apache Hadoop [2] and Spark [28] were developed and have evolved rapidly. However, the breakdown of Dennard's scaling has led to energy efficiency becoming a primary concern in datacenters, and this spawned the adoption of power-efficient accelerators and coprocessors such as GPUs (graphic processing units) and FPGAs (field-programmable gate arrays) in datacenters.

Among various power-efficient devices, FPGAs' low power consumption and reprogrammability allow them to be used as high-performance and energy-efficient accelerators for a variety of applications. Applications with a large fraction of computationally intensive kernels, such as string matching, searching and sorting, are suitable for acceleration using FPGAs. In addition, adopting FPGAs in private datacenters has recently garnered much attention from the community. For example, Microsoft has adopted CPU-FPGA systems in its datacenter to help accelerate the Bing search engine [18]. Amazon also introduced the F1 instance [1], a compute instance equipped with FPGA boards, in its Elastic Compute Cloud (EC2). As a result, datacenters with FPGAs are expected to be widely used in the near future.

To facilitate the ease of use of FPGA for big-data computations, Blaze [14] made efforts to integrate FPGAs into Apache Spark, one of the most widely used big data analytic frameworks, and allow programmers to offload computational kernels to FPGAs easily. However, Blaze leaves to the programmers the responsibility for developing FPGA accelerators for the offloaded kernels. Therefore, a significant amount of programming effort is required for users to manually produce accelerator designs. Worse still, Blaze only supports primitive types as accelerator interfaces, indicating that additional programming effort may be needed to serialize/deserialize composite data types such as structures and classes in Java/Scala.

On the other hand, the user-written computational kernels in Scala for Spark and Blaze contain the semantic information of RDD transformation operators. The semantic information is capable of being used for further optimizing the design and facilitating the design space exploration process. In this paper we present the S2FA (**S**park-**to**-**F**PGA-**A**ccelerator) framework, a compilation framework that automatically transforms Scala computational kernels in Apache Spark applications into optimized accelerator designs. S2FA leverages a parallel learning-based design space exploration approach with several proposed strategies to realize a close-to-optimal microarchitecture configuration efficiently. Also, S2FA generates appropriate application-accelerator interfaces to offload the kernels onto FPGAs. Consequently, S2FA frees software programmers from considering the underlying hardware architecture. To summarize, this paper makes the following contributions:

- An automated framework that compiles computational kernels of Spark applications to FPGA accelerators while generating corresponding interfaces for software-hardware integration.
- Support of object-oriented constructs in the code generation to improve programmability for developers.
- A learning-based design space exploration (DSE) approach to effectively organize a given set of optimization strategies to produce high-performance designs in a few iterations.
- Detailed evaluation of S2FA on a variety of computational kernels on the Amazon EC2 F1 instance, and insights to the impact of DSE optimization strategies on the quality of results.

The evaluation shows that the generated designs achieve up to 49.9× speedup for several machine learning kernels over the JVM.

2 SPARK AND BLAZE RUNTIME

Apache Spark [28] is a fast, general framework for large-scale computations in clusters. It is the successor to Apache Hadoop [2], but with an optimized, lightweight data distribution mechanism called resilient distributed datasets (RDDs). An RDD is a distributed vector which supports in-memory caching to reduce I/O and communication overheads during large-scale data processing. As a result, Spark performs exceedingly well on iterative algorithms in machine learning and graph processing.

To deploy FPGA accelerators with Spark in datacenters easily and efficiently, the Blaze runtime system [14] was developed. Blaze abstracts FPGA accelerators as a service by decoupling the FPGA

[*]J. Cong also serves as the Chief Scientific Advisor of Falcon Computing Solutions Inc., in addition to his primary affiliation at UCLA.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00

https://doi.org/10.1145/3195970.3196109

accelerator and Spark application developments [8]. FPGA accelerators can be registered to the Blaze accelerator manager so that Spark application developers can access FPGA accelerators using provided APIs that hide the details of JVM-to-FPGA data communication.

Code 1: Blaze Application Code Snippet

```
1 val pairs: RDD[(String, String)] = // read input
2 val blaze_pairs = blaze.wrap(pairs)
3 val matching = blaze_pairs.map(new SW)
4
5 class SW() extends Accelerator[(String, String), (String, String)] {
6   val id: String = "SW_kernel"
7   def call(in: (String, String)) = { ... }
8 }
```

Code 1 shows a code snippet of an application to illustrate the Blaze programming model. In this example, we apply the `S-W` method to each string pair, which can be represented as `(String, String)` in the RDD `pairs` to perform the Smith-Waterman algorithm [23] for string matching. Blaze requires that the RDD be wrapped with its API (line 2) to indicate that transformations of the RDD (line 3) should be offloaded to FPGAs. The RDD transformation in line 3 also invokes another Blaze class at the bottom with a string ID to let Blaze assign tasks to the pre-registered accelerator. While Blaze streamlines accessing FPGAs from Spark applications, it still requires users to deal with FPGA kernels and data processing.

In this work we develop a framework for automatically generating 1) an FPGA accelerator design, and 2) data (de)serialization methods for the Blaze runtime system. However, we note that the S2FA framework is able to compile any Java/Scala method that satisfies the constraints listed in Section 3.3 to an FPGA kernel, so we can easily integrate S2FA with other JVM-based runtime systems such as Hadoop and streaming APIs in Java 8.

3 S2FA FRAMEWORK

In this section we begin with an example to discuss the challenges. Then we introduce the S2FA framework and explain how we address these challenges. This is followed by a summary of the limitations.

Code 2: Motivating Example (S-W) in Scala

```
1   def call(in: (String,String)): (String, String) = {
2     var out1 = new Array(String)[256]
3     var out2 = new Array(String)[256]
4     // computation start
5     ... = in._1 ...
6     ... = in._2 ...
7     out1 = ...
8     out2 = ...
9     // computation end
10    (out1, out2)
11  }
```

3.1 Motivating Example and Challenges

We continue to use the S-W example from Section 2 while showing a more detail code snippet in Code 2. Note that methods `_1`, `_2` in lines 5, 6 are used for fetching the first and second element in a `Tuple2`, respectively; line 10 invokes the constructor to create a new `Tuple2` as the output.

To build an automated framework that generates an accelerator to process string matching using Blaze, our goal is to generate a high-performance FPGA accelerator design in HLS C with functionality equivalent to the original Scala method. However, the following impediments make the implementation challenging.

Challenge 1: The semantic gap. Since HLS C is a C-based programming language, it does not support any object-oriented language construct. In this example, the class `Tuple2` with its virtual methods `_1`, `_2` and the constructor are not supported in FPGA kernels. Thus, we must substitute them using FPGA-compatible data representations. For instance, we have to leverage two one-dimensional arrays to store a pair of strings separately instead of using `Tuple2`. Although this might not be a case for expert programmers, it is a serious challenge for a compiler.

Challenge 2: Poor performance of generated FPGA designs. Even we solve the previous challenge and successfully generate an HLS C kernel, the performance on FPGA could be disappointing. Heavy code transformation and pragma insertion with the consideration of FPGA architecture are generally required to improve the performance. Since it is impractical for a Spark-to-FPGA framework to provide HLS-like pragmas or annotations for users to specify, it is more challenging to construct an efficient accelerator design from a large design space.

Challenge 3: System integration. After generating an efficient FPGA kernel, we also need a method to tie the kernel into the host JVM application. In our motivating example, we need a Scala implementation for processing a `Tuple2` object into two separate arrays. Requiring the user to do this task manually would also impose heavy programmer burdens.

These challenges make the implementation of a Spark-to-FPGA framework highly non-trivial. Next, we present our S2FA framework that addresses the these issues.

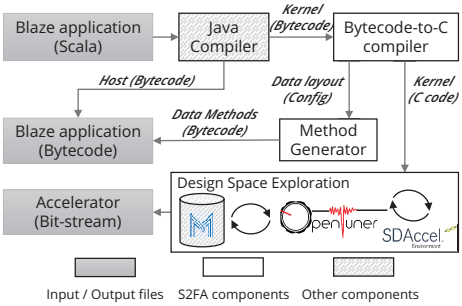

Figure 1: S2FA Framework Overview

3.2 S2FA Framework Overview

Based on the challenges we presented in Section 3.1, we design the S2FA framework shown in Fig. 1. In this section we introduce each key component in the framework, along with the motivating example to illustrate the execution flow.

Bytecode-to-C compiler. The first step of the framework is to leverage the bytecode-to-C compiler to generate a functional equivalent C kernel from the kernel method in Java bytecode that is compiled from the user-written Scala implementation. The reason of compiling bytecode to HLS C instead of hardware description language (HDL) is that 1) HLS C has demonstrated its ability to achieve high performance as RTL for many applications [11], and 2) exploring the design space in the HLS C level is much more efficient than in the HDL level in terms of the difficulty of code reconstruction. Our bytecode-to-C compiler is developed based on the open-source AMD APARAPI [3] framework which performs bytecode-to-OpenCL compilation. The original APARAPI implementation, however, exposes OpenCL-like APIs such as `getGlobalId()` to users, so the APARAPI users still have to implement the kernel with underlying architecture consideration to achieve high performance. In this work we take the APARAPI code generation and modify it heavily to achieve 1) sequential C code generation[1], and 2) commonly used composite types and RDD transformation operator support. Since modifying APARAPI to generate C code from generating OpenCL code is mainly syntax translation, here we explain how to support composite types and transformation operators.

To support commonly used composite types such as `Tuple2`, S2FA flats class fields and inlines class methods. Taking the Scala code snippet in Code 2 along with the generated C kernel in Code 3 as an example: in Code 2, the method invocations `_1`, `_2` in lines 5-6 are replaced by array reference expressions. The accesses of local variables `out1`, `out2` are also replaced by the function arguments

[1]The reason we generate C instead of C++ for HLS is because the FPGA vendor tools either do not allow C++ class as the top-level, or even do not support class types.

for writing outputs so that the class constructor in line 10 can be removed. Note that the class flatten implementation in S2FA is extensible so it is easy to support more customized composite types. We plan to investigate related issues in the future work.

In addition, since users only need to specify a lambda function (also called an anonymous function) inside the RDD transformation operators such as `map` and `reduce`, the bytecode-to-C compiler inserts the corresponding predefined template to achieve the equivalent functionality. For example, the loop in line 10 of Code 3 is inserted to iterate each task to support `map` transformation in Spark.

Code 3: S2FA Generated C Code Snippet for DSE

```
1  void call(char *in_1, char *in_2, char *out_1, char *out_2) {
2    // computation start
3    ... = in_1 ...
4    ... = in_2 ...
5    out_1 = ...
6    out_2 = ...
7    // computation end
8  }
9  void kernel(int N, char *in_1, char *in_2, char *out_1, char *out_2) {
10   for (int i = 0; i < N; i++) {
11     call(&in_1[i*128], &in_2[i*128], &out_1[i*256], &out_2[i*256]);
12   }
13 }
```

Design space creation and exploration. Since our bytecode-to-C compiler only translates the syntax, we have to leverage a fully automated design space exploration (DSE) approach to improve the performance. The challenge at this stage is to find the optimal design from the large design space, including code transformation and HLS pragma insertion that are impractical to be done by Spark users (*Challenge 2*).

In addition to the HLS pragmas, we also adopt a transformation library of the Merlin compiler [9, 10], developed by Falcon Computing Solutions [4] for C/C++ to FPGA compilation, to include code transformation into the design space. The Merlin transformation library provides a set of pragmas for useful code transformations such as loop tiling, tree reduction, coarse-grained parallelism, and so forth. As we will mention in Section 4.1, S2FA analyzes the kernel and inserts pragmas with possible values to construct the design space. The design space will then be explored by our DSE approach[2]. In addition, for the design point selected by the DSE, S2FA executes the Merlin compiler for code transformation and evaluates the transformed kernel by estimating the performance and resource utilization using high-level synthesis (HLS) of the Xilinx SDx [6]. However, performing HLS for each design point is time-consuming, so we present a parallel learning-based design space exploration approach in Section 4 to address this issue.

Data processing method generator. To process the input and output data to fit the accelerator design interface and the host application, our method generator accepts the data layout configuration from the bytecode-to-C compiler and generates corresponding Scala methods by applying the predefined Scala method templates. The generated method uses Java reflection to access object fields and reorganizes them to fit the accelerator interface.

With each of these challenges solved, the final step is to generate the bit-stream using the commercial design flow and broadcast the bit-stream to each worker node in the datacenter so that the generated accelerator designs can be invoked by the Blaze runtime. Due to page limit, we will skip the details of the bytecode-to-c compiler and data processing method, and focus only on our design space exploration approach in Section 4.

3.3 Limitations

The S2FA framework is able to compile the user-written Scala code as long as the function satisfies the following constraints:

Data types: S2FA supports all primitive types and widely used classes that are already defined in the S2FA. For other classes, we currently requires users to implement a S2FA class template. We leave the automation to our future work.

Library calls: Since the bytecode of library methods might not contain the enough information such as type parameter description, S2FA currently does not support library calls.

Dynamic memory allocation: S2FA supports the JVM's `new` operation with a *constant* memory size. All `new` operations will be compiled to static variable declarations in C, and no dynamic memory allocation will be performed on the FPGA.

The above restrictions do not affect design scopes, meaning that users are still able to leverage S2FA to accelerate a kernel with arbitrary functionality. Instead, the restriction only affects the coding style and programming model.

4 DESIGN OPTIMIZATION

As we mentioned in the previous section, the bytecode-to-C compiler only guarantees the functionality. The S2FA framework thus needs a mechanism to optimize the performance of generated designs. In this section we describe the design optimization techniques embedded in the S2FA framework. We first summarize the design space in Section 4.1, and then describe our design space exploration (DSE) approach in Section 4.2. Finally, in Section 4.3, we discuss the strategies we applied for accelerating the DSE process.

4.1 Design Space Identification

Table 1 lists the target design space. Note that the `flatten` option in the loop pipeline factor refers to the Merlin code transformation that tries to apply fine-grained pipelining to a nested loop by fully unrolling all its sub-loops. We identify the design space for each kernel by analyzing the kernel AST using the ROSE compiler infrastructure [5] and polyhedral framework [30] to realize loop trip-counts, available bit-widths, and so on. As can be seen, it is impractical to explore this tremendous design space exhaustively. For example, the design space of the S-W example contains more than a thousand trillion design points. This motivates us to implement an automatic design space exploration (see Section 4.2) to reach a near-optimal solution in a short time.

Table 1: The Target Design Space

Factor	Design Space (Values)
Buffer bit-width	$\{b \mid b = bw(B) \in \mathbb{B}, 8 < b = 2^n < 512\}$
Loop tiling	$\{t \mid t = T(L) \in \mathbb{L}, 1 < t < TC(L)\}$
Loop parallel (coarse-/fine-grained)	$\{u \mid u = UF(L) \in \mathbb{L}, 1 < u < TC(L)\}$
Loop pipeline (coarse-/fine-grained)	$\{p \mid p = P(L) \in \mathbb{L}, p \in \{on, off, flatten\}\}$

4.2 Learning-Based Design Space Exploration

Traditionally, gradient descent like numerical approaches are widely used for performing DSE. Unfortunately, it is inapplicable to S2FA because such approaches require an analytical form to evaluate the design quality. Since S2FA allows programmers to specify any function for RDD transformation, it is hard to develop a unified model to cover all possible designs. As a result, we use a set of reinforcement learning algorithms, including uniform greedy mutation, differential evolution genetic algorithm, particle swarm optimization, and simulated annealing, to perform DSE in the S2FA.

In order to assemble all listed algorithms, the DSE flow of S2FA is built on the top of OpenTuner [7], an open-source framework for building domain-specific program tuners. The OpenTuner runtime allows multiple reinforcement learning algorithms to work simultaneously to cover as many customized tuning problems as possible, and adopts a multi-armed bandit algorithm [13] to judge the effectiveness of each search technique and allocate design points according to the judgment. The algorithm that can efficiently find high-quality design points will be rewarded and allocated more design points, and vice versa.

Although the adopted learning algorithms have full coverage to the design space, it is inefficient to find the best design point. We summarize the main impediments as follows:

[2]Although our DSE approach is used for Spark derived kernels in this paper, this approach can actually be applied to general C kernels.

Impediment 1: Expensive evaluation approach: In order to cover all possible user-written kernels in our framework, we use the Xilinx SDx [6] to perform HLS for resource and cycle estimation instead of building an analytical model. However, HLS takes several minutes to evaluate one design point so only tens of design points can be evaluated in one hour.

Impediment 2: Complex factor dependencies: Many design space factors a have high dependency on each other. For example, enabling fine-grained pipelining to a nested loop (`flatten` in Table 1) causes all sub-loops to be fully unrolled and results in the invalidation of corresponding design space factors. This phenomenon might confuse the learning algorithm, so it has to spend more iterations on realizing such dependencies.

4.3 DSE Optimization

In this section we introduce methodologies for addressing the impediments raised in the previous section to improve the DSE efficiency. The idea of our parallel DSE process is partially inspired by DATuner [26], a parallel auto-tuner for Verilog-to-Routing (VTR) FPGA compilation. DATuner finds the best parameter values of the VTR tool to achieve better resource utilization and frequency in a given, fixed time period by dynamically partitioning the design space and allocating more CPU cores to the partition with higher QoR. In contrast, our flow (shown in Fig. 2) parallelizes the DSE process based on static partition rules to avoid set-up time (Section 4.3.1). Also, unlike DATuner that uses random seeds and a time limit to start and terminate the DSE of a partition, our flow generates effective seeds for each partition to reduce the probability of being trapped in the infeasible design space region (Section 4.3.2), and sets up a stopping criteria to avoid long tails (Section 4.3.3). The effectiveness analysis is presented in Section 5.2.

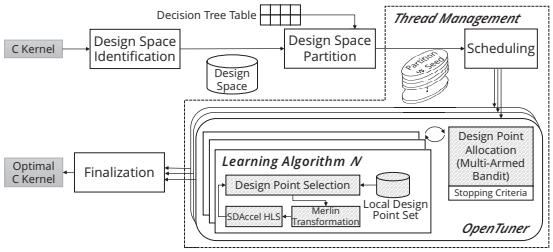

Figure 2: Fast Design Space Exploration Flow in S2FA

4.3.1 Design Space Partition. Since the learning algorithms we adopted are iterative algorithms that have a strong dependency between iterations, we cannot simply increase the DSE efficiency using more CPU cores to address *Impediment 1*. As a result, we statically separate the design space into independent partitions and assign different cores to different partitions to perform the DSE in parallel. As shown in Fig. 2, our flow has a mechanism that adopts the first-come-first-serve approach to schedule partitions to threads, so it perfectly solves *Impediment 1* as long as the partition number is larger than or equal to the number of CPU cores.

Although the authors in [26] claim that the dynamic partition is more case-specific and results in a better convergence rate than the "one-for-all" static partition, it needs several iterations for sampling at the beginning of the DSE process for every partition. Consequently, to take advantage of both, we adopt the "some-for-all" static partition approach. We statically create some sets of rules and choose the set that is most suitable to the design for partitioning only at the beginning of a DSE process.

Our rules are defined based on the following methodologies:

Partition the design space based on loop hierarchy in order to reflect the design factor dependency (*Impediment 2*). According to our observation, the same loop level could have similar impact on performance even in different applications, so we attempt to partition the design space based on the loop level. However, it is impractical to build application-specific loop-level based rules without any training data. As a result, we use a heuristic approach by grouping the applications with similar loop hierarchy geometrically and generate training data to establish the rules.

Partition the design space according to the RDD transformation semantics such as `map` and `reduce` as described in Section 3.2. In detail, we define the rule based on the scheduling of the outermost loop in kernels, since the outermost loop in kernels is always inserted by our bytecode-to-C compiler to achieve the equivalent functionality as the corresponding parallel pattern.

With the rules we obtained from above methodologies, we determine and rank the rules by building a binary decision tree that clusters the design points which potentially have similar resource utilization or latency so that the learning process can be more efficient. Decision tree is a popular method for classification and regression. Each tree node represents a rule that is composed of a parameter and a condition (e.g., `parallel factor < 16`). A path from the tree root to a leaf with all rules on the path are conjugated to form a partition. These nodes are determined by greedily selecting the best rule to maximize the information gain. Formally, we choose nodes from the set $argmax_n IG(n, D)$ where $IG(n, D)$ is the information gain if we apply node n to the dataset D, as it has been defined as follows:

$$IG(n, D) = Imp(D) - \frac{N_l}{N} \times Imp(D_l) - \frac{N_r}{N} \times Imp(D_r) \quad (1)$$

where N_l, N_r, N are the size of the left partition subset D_l, right partition subset D_r and overall dataset D respectively. $Imp(D)$ is an impurity measurement of dataset D. Impurity function is usually selected based on the type of decision tree task (classification or regression). Since the value of each partition in our case is a regressed number (latency), we choose variance as our impurity function.

With partitions from a decision tree, we can efficiently alleviate *Impediment 2* because the learning algorithm is able to learn information without being disturbed by outliers. We note that since all partitions are disjoint and the union of all partitions is the original design space, our design space partition approach preserves the optimality while improving the DSE efficiency.

4.3.2 Seed Generation. Although we have partitioned the design space systematically in the previous section, a partition may still contain millions of design points. However, it is too aggressive to prune the design space using heuristics such as limiting parallel factor or local buffer size, because the boundary of those factors varies from arbitrary user-written kernels and results in a different infeasible region in the design space. For instance, performing coarse-grained parallelism with factor 256 to the outermost loop might be infeasible for most designs due to high routing complexity, but it could be an optimal choice for certain designs that have a very simple computational pattern. As a result, instead of heuristic pruning, we preserve an entire design space but increase the probability of finding the best design point in fewer iterations by providing seeds, the starting point for learning algorithms.

We generate two seeds for each partition with different strategies. The first seed is *performance-driven*. For this seed, we enable pipelining for all loops, set the parallel factor of every loop to 32, and set the buffer bit-width to 512. Although this configuration might fail to be synthesized for some designs, we can significantly reduce the iteration number of the DSE process for others. On the other hand, the second seed is *area-driven*. For this seed, we disable all optimizations so all loops are performed sequentially and all off-chip buffers are set to the minimum bit-width. As a result, this seed has the most conservative configuration in terms of resource utilization and design complexity, so it is less likely to be infeasible from the perspective of the high-level synthesis tool. With the *conservative seed* as a starting point, the learning algorithm is guaranteed to start searching in the feasible region and avoid being trapped in the infeasible region all the time.

4.3.3 Early Stopping Criteria. Since the vanilla OpenTuner does not have a systematic stopping criteria but only adopts the limitation of either execution time or searched point count, the long tail is almost inevitable. In fact, the long tail becomes a serious problem for exploring FPGA accelerator designs because we need minutes to an hour to evaluate a single design point using HLS.

To solve the long tail problem without the knowledge of optimal performance, we add one more criteria in addition to the time limit to stop the DSE process earlier based on the following concept. According to the dataset of explored results D_i after i iterations, and its subset of the uphill performance results between any two consecutive iterations D_i^u, let $P_{D_i}(D_i^u \mid t_j)$ be the experimental conditional probability by mutating design factor t_j, and let $P(t_j)$ be the theoretical probability with equal likelihood to other factors. Our early stopping criteria function should converge when $P_{D_i}(D_i^u \mid t_j)$ is close enough to $P(t_j)$. We use $H(D_i)$—the Shannon entropy [22], a widely used approach in information theory for quantifying uncertainty—to formulate this concept. That means we will terminate the DSE process for a partition at iteration i if we have a low enough uncertainty of finding a better result in that partition at the next iteration. Formally, our early stopping criteria with the Shannon entropy is defined as follows.

$$|H(D_i) - H(D_{i-1})| \leq \theta$$
$$H(D_i) = -\sum_j P_{D_i}(D_i^u \mid t_j) \log P_{D_i}(D_i^b \mid t_j) \quad (2)$$

where θ is the threshold for termination. Note that this metric has also been used in other fields such as image processing [19]. In practice, we terminate the DSE process after the entropy difference is lower than θ for consecutive N iterations to avoid pulses. As we will illustrate in Section 5, this systematic criteria works better than the trivial one that simply stops the process if a better result cannot be found for a number of iterations.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

Our evaluation of S2FA is performed on Amazon EC2 F1 instance [1]. The instance type is `f1.2xlarge`, which includes an 8-core CPU with 122GB of main memory and one Xilinx Virtex UltraScale+™ VU9P FPGA with three separated dies. In addition, we select a set of common Spark applications to evaluate S2FA. We also select two string processing applications in our evaluation since they are classic applications for FPGA acceleration. All applications are built with the software environment that consists of JDK 1.7.0_79, Scala 2.11.4 and Spark 1.5.1.

5.2 Results and Analysis

While the application-level speedup and system-level overhead are transparent to Blaze [14] runtime, this section focuses on the performance evaluation of S2FA-generated accelerators.

Fig. 3 depicts the process of design space exploration. The x-axis is the exploration time in minutes while the y-axis is the normalized execution cycle based on the random seed of the vanilla version of OpenTuner. The dashed line in each sub-figure is performed by the OpenTuner [7], while the solid line is the process of the S2FA DSE. Both processes are performed by eight CPU cores[3]. In summary, the DSE process of S2FA saves 52.5% execution time on average while achieving 35× performance improvement when compared to the OpenTuner. We analyze the effectiveness of our DSE optimization from the figure as follows.

First, the QoR difference of the first explored point illustrates the effectiveness of our *seed generation*. Second, almost all S2FA DSE processes drop faster, meaning that S2FA is capable of finding a better design point in a shorter time than the OpenTuner due to an effective *design space partition*. The exceptions is `KMeans`. We can see in Fig. 3 that OpenTuner also achieves the same performance as S2FA. This is because the design space of `KMeans` is relatively small, so the benefit of design space partition is marginal.

Third, S2FA terminates the DSE process faster (~1.9 hours on average) than the OpenTuner (four hours) due to the early stopping criteria. As a result, even the OpenTuner is able to realize the same design as S2FA for `KMeans`; it still terminates the process at four hours, set up in advance because of the lack of an effective stopping criteria. In addition, we also analyze the effectiveness of one straightforward stopping criteria that stops the DSE process if no better result were found for consecutive 10 iterations[4]. It turns out that compared to our Shannon entropy criteria, the trivial criteria terminates the process one hour later (~2.8 hours) but only improves 4% performance on average.

Table 2: Resource Utilization and Clock Frequency (MHz)

Kernel	Type	BRAM	DSP	FF	LUT	Freq.
PR (PageRank)	graph proc.	25%	2%	16%	18%	250
KMeans (K-Means)	classification	73%	6%	10%	14%	230
KNN (K-Nearest Neighbor)	classification	75%	6%	50%	50%	240
LR (Logistic Regression)	regression	74%	3%	49%	74%	220
SVM (Support Vector Machine)	regression	74%	4%	48%	72%	250
LLS (Least linear square)	regression	74%	3%	45%	21%	230
AES (encryption)	string proc.	36%	0%	3%	6%	250
S-W (Smith-Waterman)	string proc.	33%	30%	54%	75%	100

Based on the best configurations from the DSE, Table 2 lists the resource utilization and clock frequency of each generated design. Since the performance of `AES` and `PR` are bounded by external memory bandwidth, they do not fully utilize hardware resources. On the other hand, other cases fully utilize at least one kind of resource,[5] meaning that those three designs are computationally bounded and their performance can be potentially improved if a larger FPGA is provided. In addition, since we perform place and route using the default setting of Xilinx SDx [6], the frequency fails to achieve the target (250MHz) for satisfying timing constraints for some cases. In the future we plan to model the impact of design factors on frequency during the DSE process.

Fig. 4 shows the speedup of manual and S2FA-generated FPGA designs over the original Spark transformation methods running on a JVM. The x-axis lists all designs while the y-axis illustrates the speedup in logarithm scale. We use a single-threaded Spark executor on the JVM as a baseline because only one thread is necessary for launching FPGA and other threads are able to perform other tasks simultaneously. The manual design for each application is also implemented in HLS C. Both manual and S2FA-generated designs use Xilinx SDx 2017.2 [6] as the design flow. However, as we mentioned in Section 1, S2FA only requires a few hours including bit-stream generation to finish a FPGA design, greatly reducing the development time.

As can be seen in Fig. 4, most S2FA-generated designs achieve competitive speedups to the manual designs (~ 85% on average) and outperform the corresponding Scala implementations on the JVM by 181.5× on average. On the other hand, the core computation of `LR` is the regression model that involves floating point multiplication and exponential calculation so the minimal initial interval is still 13. The `LR` manual design splits the computation statement to multiple stages to form a highly efficient pipeline. Future work would try to solve this problem by analyzing such a performance bottleneck and perform automatic partitioning. In addition, since the computational pattern of `PR` is too simple to hide the communication latency, even the manual HLS implementation cannot achieve a high performance on the FPGA.

6 RELATED WORK

There is some amount of previous work that generates FPGA accelerators from parallel pattern programming models. The author in

[3]Since the OpenTuner does not partition the design space, it uses the eight cores to evaluate top-8 candidates at one iteration. This is not scalable in terms of the efficiency.

[4]We skip this evaluation in Fig. 3 for the sake of visualization.

[5]We set the maximum resource utilization to 75% since the rest of them were used by the vendor-provided control logic.

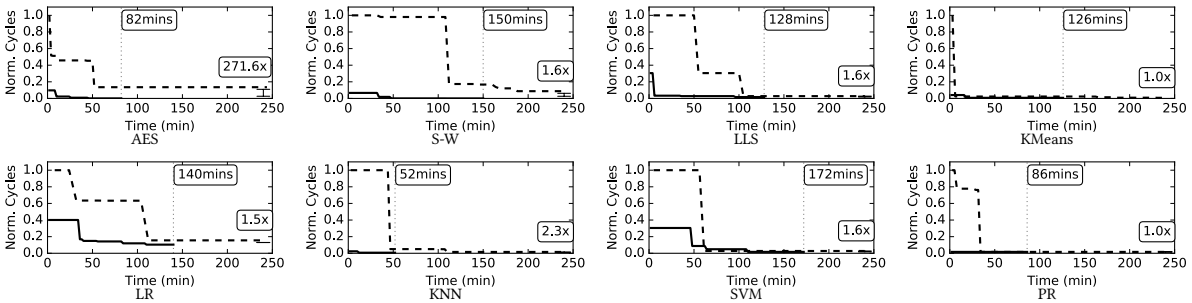

Figure 3: Design Space Exploration Process of the S2FA (solid lines) and OpenTuner [7] (dashed lines)

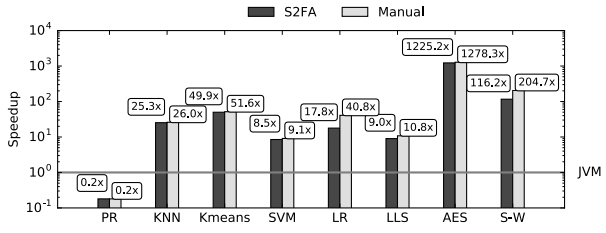

Figure 4: Comparison to Manual Designs over JVMs

[17] lets users write FPGA kernels in parallel patterns and compiles them to their intermediate representation (IR) language, DHDL [15]. Then, a heuristic DSE approach is proposed to improve the QoR. However, the programming model in [17] only supports primitive types, and the design space in DHDL does not include the architecture hierarchy since it is determined and fixed before the exploration. In addition, SparkCL [21] integrates AMD APARAPI [3] into Apache Spark and targets to FPGAs. However, the programming model of SparkCL also limits to primitive types, and requires manual design optimization. Melia [25] is a C-based MapReduce framework that automatically generates OpenCL FPGA kernels, and optimizes them by leveraging an analytical performance model [24]. However, the analytical model in [24] assumes that fully pipelining all loops is achievable, and cannot align to the optimization done by HLS design flow. In addition, Melia is a standalone framework so it is not compatible with any widely used big data analytics frameworks.

In addition, adopting learning-based DSE approaches with HLS tools to deal with the tremendous design space of high-level synthesis is also getting more and more attention [16, 20, 27, 29]. Some of them build predictive models for HLS tools to guide the DSE process[16, 20, 27], but do not consider the dependency of design factors so only suboptimal solutions can be achieved. Although [29] uses a linear model with explored points based on loop hierarchy, it is limited to nested loops and not scalable.

7 CONCLUSIONS

In this paper we present an automated framework that compiles the kernel of Spark applications to FPGA accelerators and integrates the accelerator to Blaze runtime. The S2FA framework supports object-oriented constructs in bytecode-to-C code generation to improve the programmability for programmers. It also adopts an efficient parallel learning-based design space exploration to optimize the accelerator performance. The experimental results show that our generated FPGA kernels reach 1225.2× and 49.9× speedup for string processing and machine learning applications respectively when compared with the equivalent Scala implementations from which they are automatically generated.

There are several opportunities for future work in this area. For example, it is worthwhile to explore the high-performance implementations and DSE of more object-oriented constructs while preserving their semantics and programmability. It is also promising to improve the backend side by 1) introducing more design optimization methodologies, and 2) investigating more approaches for further improving the efficiency of finding the best design point.

ACKNOWLEDGMENT

This research is partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA); Huawei, NEC, and Samsung under the CDSC Industrial Partnership Program. The author would like to thank Tyson Condie for his valuable comments.

REFERENCES

[1] Amazon EC2 F1 Instance. https://aws.amazon.com/ec2/instance-types/f1/.
[2] Apache Hadoop. http://hadoop.apache.org/.
[3] Aparapi in amd developer website. https://github.com/aparapi/aparapi.
[4] Falcon Computing Solutions, Inc. http://falcon-computing.com/.
[5] Rose Compiler Infrastructure. http://rosecompiler.org/.
[6] Xilinx SDx. www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[7] J. Ansel et al. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *PACT*.
[8] Y.-T. Chen et al. 2016. When Spark Meets FPGAs: A Case Study for Next-Generation DNA Sequencing Acceleration. In *HotCloud*.
[9] J. Cong et al. 2016. Source-to-Source Optimization for HLS. In *FPGAs for Software Programmers*. Springer International Publishing.
[10] J. Cong et al. 2016. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper. In *ISLPED*.
[11] J. Cong et al. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD*.
[12] J. Dean et al. 2008. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*.
[13] Á. Fialho et al. 2010. Analyzing bandit-based adaptive operator selection mechanisms. *Ann Math Artif Intell*.
[14] M. Huang et al. 2016. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. In *SoCC*.
[15] D. Koeplinger et al. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *ISCA*.
[16] H.-Y. Liu et al. 2013. On learning-based methods for design-space exploration with high-level synthesis. In *DAC*.
[17] R. Prabhakar et al. 2016. Generating Configurable Hardware from Parallel Patterns. *ASPLOS*.
[18] A. Putnam et al. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA*.
[19] R. Rodríguez et al. 2012. Image segmentation via an iterative algorithm of the mean shift filtering for different values of the stopping threshold. *IJIR* (2012).
[20] B. C. Schafer et al. 2012. Machine learning predictive modelling high-level synthesis design space exploration. *IET CDT* (2012).
[21] O. Segal et al. 2015. SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters. *CoRR*.
[22] C. E. Shannon. 2001. A mathematical theory of communication. *ACM MC2R*.
[23] T. F. Smith et al. 1981. Identification of common molecular subsequences. *JMB*.
[24] Z. Wang et al. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *HPCA*.
[25] Z. Wang et al. 2016. Melia: A MapReduce Framework on OpenCL-based FPGAs. *TPDS*.
[26] C. Xu et al. 2017. A Parallel Bandit-Based Approach for Autotuning FPGA Compilation. In *FPGA*.
[27] S. Xydis et al. 2015. SPIRIT: Spectral-Aware pareto iterative refinement optimization for supervised high-level synthesis. *TCAD* (2015).
[28] M. Zaharia et al. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*.
[29] G. Zhong et al. 2014. Design space exploration of multiple loops on FPGAs using high level synthesis. In *ICCD*.
[30] W. Zuo et al. 2013. Improving Polyhedral Code Generation for High-level Synthesis. In *CODES+ISSS*.