Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture

ABSTRACT

CPU-FPGA heterogeneous architectures feature flexible acceleration of many workloads to advance computational capabilities and energy efficiency in today's datacenters. This advantage, however, is often overshadowed by the poor programmability of FPGAs. Although recent advances in high-level synthesis (HLS) significantly improve the FPGA programmability, it still leaves programmers facing the challenge of identifying the optimal design configuration in a tremendous design space. In this paper we propose the composable, parallel and pipeline (CPP) microarchitecture as an accelerator design template to substantially reduce the design space. Also, by introducing the CPP analytical model to capture the performance-resource trade-offs, we achieve efficient, analytical-based design space exploration. Furthermore, we develop the AutoAccel framework to automate the entire accelerator generation process. Our experiments show that the AutoAccel-generated accelerators outperform their corresponding software implementations by an average of 72x for a broad class of computation kernels.

1 INTRODUCTION

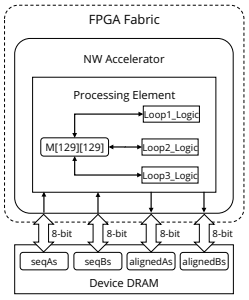
Due to power and energy constraints, conventional general-purpose processors are no longer able to sustain the performance and energy improvement in commercial datacenters. Field programmable gate arrays (FPGAs), which offer the potential of orders-of-magnitude performance/watt gains for a broad class of applications while retaining reconfigurability, are attracting increased attention as a mainstream acceleration technology. Both Microsoft and Baidu have incorporated FPGA accelerators in their datacenters to accelerate large-scale production workloads such as search engines [24] and neural networks [19]. Amazon also introduced the F1 instance [1], a compute instance equipped with FPGA boards, in its Elastic Compute Cloud (EC2). With the $16.7 billion acquisition of Altera, Intel announced the Heterogeneous Architecture Research Platform (HARP) [6] that provides an FPGA and a Xeon processor in a single semiconductor package. Predictions have been made that as much as 30% of datacenter servers will have FPGAs by 2020 [3], indicating that FPGAs could become a major component in future servers.

On the other hand, a serious challenge to FPGA-based acceleration is programmability. FPGA programming is generally considered a RTL (register-transfer level) design practice which requires notable hardware expertise in designing accelerator microarchitectures such as controls, data paths, and finite state machines. This makes the effort of FPGA programming prohibitive to most datacenter application developers. It is even more challenging when the mainstream algorithm in an application domain is constantly evolving; i.e., an algorithm may have already become obsolete during the development process of its accelerator.

Decades of research have focused on improving FPGA programmability. High-level synthesis (HLS) [12] that allows hardware designs to be described in high-level programming languages is recognized as a promising solution. Commercial HLS tools such as Xilinx SDAccel [5] and Intel FPGA SDK for OpenCL [2] now accept an accelerator kernel implementation in C, C++ or OpenCL, and directly transform it down to an FPGA accelerator circuit, without the involvement of RTL design. Nonetheless, such syntactic C languages make it difficult for HLS tools to extract necessary information from the kernel code to optimize memory organization and task scheduling via static analysis. To complement the missing information of static analysis, HLS tools provide a set of language extensions for users to manually exert guidances in order to generate a high-performance accelerator design. As a consequence, proper considerations of underlying FPGA architecture and a significant amount of programming effort are necessary for accelerator designers to make high-performance HLS implementations.

```
void engine(...) {
  int M[LEN+1][LEN+1];
  ...
loop1: for(i=0; i<LEN+1; i++) {M[0][i]=...}
loop2: for(j=0; j<LEN+1; j++) {M[j][0]=...}
loop3: for(i=1; i<LEN+1; i++) {
    for(j=1; j<LEN+1; j++) {...
      M[i][j]=...
  }}
  ...
}
void kernel(char seqAs[], char seqBs[],
     char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*LEN, seqBs+i*LEN,
       alignedAs+i*LEN*2, alignedBs+i*LEN*2);
}}
```

(a) Code Snippet



(b) Microarchitecture

Figure 1: NW Kernel and the Corresponding Architecture

We use the Needleman-Wunsch (NW) algorithm [18], a dynamic programming algorithm with quadratic time complexity, as a motivating example to demonstrate the required programming efforts for producing high-quality design. Fig. 1(a) shows the C implementation of the NW kernel. The `kernel` function processes a series of genome sequence alignment jobs, each with a pair of ASCII-encoded sequences as input and the post-aligned sequences as output. The `engine` function applies the Needleman-Wunsch algorithm to the input sequences and generates the optimal post-aligned sequences based on a predefined scoring system. The corresponding accelerator design generated by Xilinx SDAccel is shown in Fig. 1(b). Since the C implementation is programmed without taking FPGA acceleration into consideration, the generated accelerator design is 92× *slower* than a single modern CPU core. In order to improve the efficiency of the accelerator to finally achieve a speedup over CPUs, accelerator designers typically need to reconstruct the code extensively through the following transformations.

Transformation #1: Improving off-chip DRAM bandwidth utilization. The `kernel` function is the top-level function of the NW kernel and defines the entire accelerator. Its arguments define the input and output buffers that reside in the off-chip DRAM of the FPGA board. The FPGA accelerator connects to these off-chip buffers through AXI channels. The data width of each AXI channel is eight bits, the same as the data type width of the corresponding argument (8-bit `char` type). This results in only one byte/cycle off-chip data transaction throughput for each channel, while state-of-the-art CPU-FPGA platforms typically support 64 byte/cycle. Designers must perform heavy code reconstruction and explore a large design space to improve the off-chip DRAM bandwidth utilization [13].

Transformation #2: Explicit data caching. The microarchitecture in Fig. 1(b) does not present on-chip data caching components, which leads to every memory access physically going off chip. This is because the CPU architectures provide programmer-transparent memory hierarchy, and thus typical software programs do not need

code statements to explicitly manipulate on-chip caches. In contrast, FPGAs' scratchpad BRAM blocks demand explicit manipulation by programmers to use them wisely. As a result, accelerator designers must manually manipulate scratchpad in the behavioral description, and explore various data caching strategies and parameter settings.

Transformation #3: Loop pipelining/parallel. The `kernel` function body is a loop statement that iteratively traverses and aligns every sequence pair via the `engine` module. In Fig. 1(b), the `engine` module processes only one sequence pair at a time, despite the fact that these sequence pairs are independent of each other and thus can be processed in parallel or pipeline. Designers must determine whether to pipeline and/or parallel every loop statement in a program, and explore different combinations of loop parallel factors to optimize the design.

Transformation #4: On-chip memory organization. The major computation of the NW algorithm is to generate a two-dimensional score matrix, `M`. However, the array `M` in Fig. 1(b) is implemented using an on-chip BRAM buffer that has only one write port, so only one request can be processed at each cycle. As a result, even the three loop logic can be paralleled or pipelined based on *Transformation #3*; the performance will still be bounded by the buffer port limitation. Designers must perform partitioning on on-chip buffers, and determine the optimal partition factor for each buffer.

Performing these transformations require heavy code reconstruction[1] and intimate knowledge of hardware intricacies, which is prohibitive to most datacenter application developers. Worse still, these transformations are intervened with each other in terms of performance and resource consumption, which considerably increases the complexity of the performance-resource trade-offs. Even for experienced hardware designers, it still requires a great deal of effort to resolve such complicated trade-offs and identify the optimal design choice. Motivated by these challenges, in this paper we propose the composable, parallel and pipeline (CPP) microarchitecture, an accelerator design template with high flexibility to bound the design space. Then, we perform design space exploration to realize the optimal configuration of the CPP-based accelerator design to maximize the performance under the resource constraints. In particular, we derive an analytical model to analyze and evaluate the design space as well as the performance and resource consumption, and further propose a series of pruning strategies to reduce the design space so that it can be exhaustively searched within one hour. Finally, we develop the AutoAccel framework to automate the entire accelerator generation process and provide end users with a nearly push-button experience. In summary, this paper makes the following contributions:

- **The CPP microarchitecture and the analytical model.** By introducing the accelerator design template, we are able to perform design space exploration to realize the optimal design configuration efficiently using the corresponding performance and resource model.
- **The pruning strategies.** We propose a series of pruning strategies to reduce the design space, so that the optimal design configuration can be found exhaustively in one hour.
- **The AutoAccel framework.** We automate the entire accelerator generation and optimization process by implementing the AutoAccel framework and thus substantially improves the FPGA programmability.

Our experiments show that the AutoAccel-generated accelerators outperform their corresponding software implementations by an average of 72x for the MachSuite [25] computation kernels.

[1] The NW code grows from 80 to over 400 lines of code (5×) after these transformations.

2 ACCELERATOR DESIGN TEMPLATE

In this section we present our approach to automatically transform a user C program to a high-quality accelerator behavioral description. We first formulate the problem, and then introduce the composable, parallel and pipeline (CPP) microarchitecture that serves as an accelerator design template to address the problem.

2.1 Problem Formulation

Formally, this paper aims to solve the following problem: given an input C/C++ computational kernel that satisfies the following constraints, perform automatic code transformation to the kernel under the hardware resource constraints so that the performance of generated accelerator design is maximized.

- ***Synthesizable***. The input kernel must be synthesizable via commercial HLS tools. That is, it should not include recursive function calls or dynamic memory allocation. However, this constraint does not affect the scope of supported kernels since it is always possible for programmers to manually transform such code structures to equivalent, synthesizable structures.
- ***Cacheable***. The memory footprint of any single instance of the top-level loop must be smaller than the FPGA on-chip memory capacity to ensure that the kernel computation and external memory transaction can be fully decoupled.

We develop an algorithm based on the polyhedral analysis from [22] to determine whether an input program meets the constraints.[2] Based on our problem formulation, computational kernels featuring extensive random accesses on a large memory footprint, such as PageRank [20] and the breadth-first search (BFS) algorithm, will probably not meet the *Cacheable* constraint. On the contrary, computational kernels that process input data block by block generally meet these constraints. In fact, almost all streaming and batch processing kernels with regular data-level parallelism fall into this category. These kernels are also well-known to potentially benefit from FPGA acceleration.

For the kernel that satisfies the above constraints, we implement it using our proposed microarchitecture, which we will discuss in the following section, to bound the design space.

2.2 CPP Microarchitecture

The composable, parallel and pipeline (CPP) microarchitecture is proposed as a template of accelerator designs. For an input kernel that meets the above constraints, our approach first fits the kernel into the CPP microarchitecture, then performs design space exploration to identify the optimal parameter configuration, and finally transforms the input kernel code to the CPP microarchitecture description code. The CPP microarchitecture guarantees the quality of the output accelerator design by providing a series of features to realize the transformations we summarized in Section 1. In the remainder of this section, we introduce the key features of the CPP microarchitecture, as shown in Fig. 2, along with the NW motivating example.

Feature #1: Coarse-grained pipeline with data caching. Fig. 2 illustrates the NW accelerator design under the CPP microarchitecture. The overall CPP microarchitecture consists of three stages: `load`, `compute` and `store`. The `kernel` function in the NW source code only corresponds to the `compute` module instead of defining the entire accelerator. The input sequence pairs are processed block by block, i.e., iteratively loading a certain number of sequence pairs into on-chip buffers (Stage `load`), aligning these pairs (Stage `compute`), and storing the post-aligned pairs back to DRAM (Stage

[2] The algorithm is omitted due to page limit.

store). This feature corresponds to *Transformation #2* because off-chip data movement only happens in the load and store stages, leaving data accesses of computation completely on chip. In general, as far as the input kernel meets the *Cacheable* constraint, it is able to fit into this load-compute-store execution process.

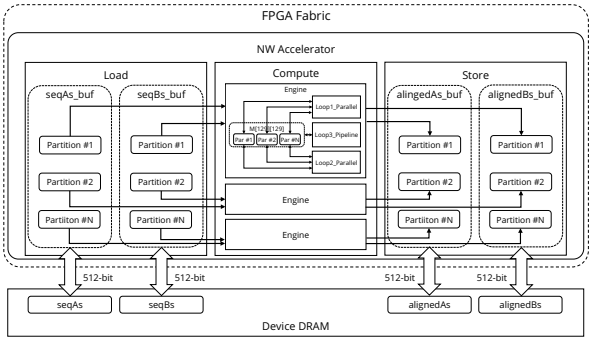


Figure 2: NW Accelerator under CPP Microarchitecture

The load and store modules connect to two input and output DRAM buffers, respectively, through AXI channels. The data widths of the AXI channels are decoupled from the type sizes of the top-level function arguments. Hence, the off-chip bandwidth can potentially approach the physical limit of the CPU-FPGA platform. Furthermore, if no dependency or only forward dependency exists between different blocks of input, the load, compute and store stages of different blocks can be processed in pipeline, and these three stages then form a coarse-grained pipeline that overlaps computation with off-chip data communication. This feature of the CPP microarchitecture could further improve the effective bandwidth of the accelerator. *Transformation #1* is realized as well.

Feature #2: Loop scheduling. The CPP microarchitecture tries to map every loop statement presented in the computational kernel function to either 1) a circuit that processes different loop iterations in parallel, 2) a pipeline where the loop body corresponds to the pipeline stages, or 3) a combination of both. As for the NW example, the loop statement in the kernel function is mapped to a set of engine modules to process the sequence pairs in parallel. The loop statements in the engine function are mapped to parallel and pipeline circuits as well. This realizes *Transformation #3*.

Feature #3: On-chip buffer reorganization. In the CPP microarchitecture, all the on-chip BRAM buffers are partitioned to meet the port requirement of parallel circuits, where the number of partitions of each buffer is determined by the duplication factor of the parallel circuit that connects to the buffer. This feature is used for realizing *Transformation #4*. In the NW example, the on-chip buffers that cache the input and output sequence pairs are partitioned into multiple segments, each segment feeding one engine module. The local buffer M that stores the score matrix is also partitioned to allow parallel read and write transactions.

In summary, the CPP microarchitecture provides these features to realize the aforementioned transformations so as to ensure the quality of output accelerator designs. Moreover, the use of an accelerator design template implies a clear design space: all valid configurations of the CPP microarchitecture. We analyze the design space in the following section.

2.3 Design Space Analysis

The CPP microarchitecture design space is determined by all its loops and external memory buffers, which is formulated as follows:

$$\mathcal{A} = \{\mathcal{L}, \mathcal{B}\} \tag{1}$$

where $\mathcal{A}$ denotes the overall design space, and $\mathcal{L}$ and $\mathcal{B}$ mean the loop set and external memory buffer set, respectively.

We then formulate the possible scheduling of loops as follows:

$$\forall L \in \mathcal{L}, L = \{(\alpha, \beta) \mid 1 < \alpha < L_{tc}, \beta = \{0, 1\}\} \tag{2}$$

where α is the integer unroll factor of loop L with trip count L_{tc} as its maximum, and β is a binary variable to indicate if the pipeline scheduling is enabled or not. As a result, the design space complexity of $\mathcal{L}$ is $O(2^m \times \prod_{L \in \mathcal{L}} L_{tc})$ where m denotes the total number of loops.

Finally, the possible design choices for external memory buffers can be represented as follows:

$$\forall B \in \mathcal{B}, B = \{(\mu, \nu) \mid 8 \le \mu \le 512, 0 \le \nu \le C_{BRAM}\}$$
$$\sum_{B \in \mathcal{B}} B_\nu \le C_{BRAM} \tag{3}$$

where μ and ν are the integer bit-width and the capacity of the on-chip memory buffer that caches a certain external memory buffer B, respectively. C_{BRAM} denotes the total capacity of all BRAM blocks. Thus, the design space complexity of $\mathcal{B}$ is $O((512 \times C_{BRAM})^n)$, where n denotes the total number of buffers.

Consequently, the overall design space complexity is $O((512 \times C_{BRAM})^n \times 2^m \times \prod_{L \in \mathcal{L}} L_{tc})$, which is too large to be explored exhaustively. In fact, even the NW motivating example contains roughly 1.4×10^{17} design points. To rapidly find the optimal design choice among such a tremendous design space, we analytically model performance and resource utilization in Section 3, and introduce our design space exploration flow with a series of pruning strategies in Section 4.

3 ANALYTICAL MODEL

This section presents our analytical model to quantify performance and resource consumption of the CPP microarchitecture. While a number of previous studies have attempted to model FPGA designs [8, 16, 26, 28–31], our model targets at a well-defined accelerator microarchitecture and thus features a highly accurate modeling of the utilization of the FPGA on-chip resources. On the other hand, some of the existing models for general FPGA accelerator designs focus on only the performance estimation [8, 26, 31]. Although others also have the model for different kind of resources [16, 28–30], their LUT models are either based on machine learning [16, 30] or even missing [28, 29]. We discuss more details in Section 6.

3.1 Performance Modeling

The performance model estimates an accelerator's overall execution cycle (C) through Eq. 4:

$$C = max(C_l + C_s, C_c) \tag{4}$$

where C_l, C_c and C_s denote the cycles of the load, compute, and store modules, respectively. Since the load and store modules share the off-chip bandwidth in our experimental platform, we make a maximum operation between the cycles of the load/store modules and that of the compute module.

The execution cycles of the load, compute and store modules, as well as all of their submodules, can be quantified as the total cycles of all the loops (C_{loop}), submodules (C_{mod}) and standalone logic (C_r), as shown in Eq. 5.

$$C_{mod}(M) = \sum_{i \in M.loops} C_{loop}(i) + \sum_{m \in M.mods} C_{mod}(m) + C_r(M) \tag{5}$$

where M denotes an arbitrary hardware module.

Then we model the loop execution. Although a loop statement can be scheduled in pipeline, parallel, or the combination of both, the first two can be treated as special cases of the last one, and can together be modeled as Eq. 6:

$$C_{loop}(L) = C_{iter}(L) + II(L) \times \frac{TC(L)}{UF(L)} \tag{6}$$

where L denotes an arbitrary loop; C_{iter}, II, TC and UF denote the iteration latency, initiation interval, trip count and unroll factor, respectively.

Subsequently, we break down and model the loop iteration in Eq. 7, where the loop iteration latency is composed of the total cycles of all the sub-loops, submodules and standalone logic.

$$C_{iter}(L) = \sum_{i \in L.loops} C_{loop}(i) + \sum_{m \in L.mods} C_{mod}(m) + C_r(L) \tag{7}$$

Eq. 5 and Eq. 7 reflect the architecture hierarchy with nested modules and loops. The proposed model recursively traverses all the loops and modules until a loop or module does not contain any sub-structures. In addition, we can find that Eq. 5 and Eq. 7 are almost identical. This is because the loop iteration can be treated as a special "module" and modeled in the same way for both performance and resource. Hence, we omit the loop iteration breakdowns in the following resource models.

3.2 Resource Modeling

The resource model estimates the consumptions of the four FPGA on-chip resources: BRAMs, LUTs, DSPs and FFs. As the DSP model is fairly straightforward and the FF model is similar to the LUT model, we only demonstrate the BRAM and LUT models due to page limit.

BRAM modeling: The BRAM consumption of a hardware module consists of the BRAM blocks used by all its local buffers (R^{mem}_{buf}) and those used by all its submodules (R^{mem}_{mod}), as shown in Eq. 8:

$$R^{mem}_{mod}(M) = \sum_{b \in M} R^{mem}_{buf}(b) + \sum_{m \in M.mods} R^{mem}_{mod}(m) \times DF(m) \tag{8}$$

where $DF(m)$ is the duplication factor of submodule m which is equivalent to the unroll factor of the loop that includes this submodule. We use "duplication factor" instead of "unroll factor" since the former is a better fit for depicting hardware modules, and the latter is more suitable for describing loop statements.

Then we model the BRAM consumption of on-chip buffers. A buffer's BRAM consumption is determined by three factors: 1) partition factors on all dimensions, $\prod_{d \in dim(B)} PF(d)$; 2) the size of unit partition, $\lceil \frac{S(B)}{\prod_d PF(d)} \rceil$; and 3) the bit-width of the buffer, $bw(B)$, as shown in Eq. 9:

$$R^{mem}_{buf}(B) = \prod_{d \in dim(B)} PF(d) \times V\left(\lceil \frac{S(B)}{\prod_d PF(d)} \rceil, bw(B) \right) \tag{9}$$

Eq. 9 adopts a function $V(s, bw)$ in [13] (Eq. 4) to calculate the BRAM consumption of a single partition. The two parameters are the size (s) and the bit-width (bw) of the partition.

LUT modeling: The LUT consumption of a hardware module (R^{lut}_{mod}) is composed of the number of LUTs used by all loops, submodules, BRAM buffers (for control logic) and the standalone logic:

$$\begin{aligned} R^{lut}_{mod}(M) = &\sum_{l \in M.loops} R^{lut}_{iter}(l) \times UF(l) + \sum_{b \in M.bufs} R^{lut}_{buf}(b) \\ &+ \sum_{m \in M.mods} R^{lut}_{mod}(m) \times DF(m) + R^{lut}_{r}(M) \end{aligned} \tag{10}$$

where R^{lut}_{iter} depicts the LUT consumption of the loop iteration that is, again, treated and modeled as a special "module." R^{lut}_{r} denotes the LUT consumption of the standalone logic.

We then model the LUT consumption of on-chip buffers (R^{lut}_{buf}). It can be decoupled into two parts: 1) the control (R^{lut}_{ctrl}) and data (R^{lut}_{data}) signals of each BRAM partition, and 2) the k-to-1 multiplexer ($R^{lut}_{mux}(k)$) that selects the desired data from all the partitions, as shown in Eq. 11:

$$R^{lut}_{buf}(B) = R^{mem}_{buf}(B) \times (R^{lut}_{ctrl} + R^{lut}_{data}) + R^{lut}_{mux}\left(\prod_{d \in dim(B)} PF(d) \right) \times bw(B) \tag{11}$$

where $R^{lut}_{mux}(k)$ can be calculated using Eq. 7 in [13]. These equations quantify the relationship between a buffer's LUT consumption and its BRAM usage.

Because of the existence of non-linear equations in the proposed model, the problem of identifying the optimal CPP configuration is formulated as an integer non-linear programming (INLP) problem which is not able to be solved in polynomial time. Fortunately, like [13], we can initialize the model by running HLS once or twice to obtain the values of a subset of parameters, since such parameters remain constant once the CPP microarchitecture is constructed: $C_r(M)$, II, TC, $C_r(L)$, S_{unit}, b_{phy}, $R^{lut}_{r}(M)$, R^{lut}_{ctrl} and R^{lut}_{data}. Based on this initialized model, the following section describes our design space exploration approach.

4 DESIGN SPACE EXPLORATION

Fig. 3 illustrates our design space exploration (DSE) flow. The DSE flow first initializes the analytical model by performing the HLS synthesis once or twice, and then parses the reports to generate the values of the design constants. Next, we fetch the design space from the transformed C kernel code with variable parameters and form a design point set.

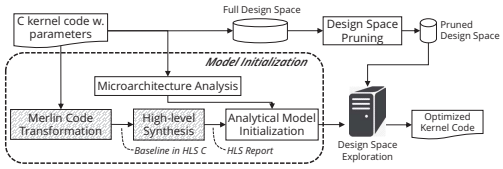


Figure 3: Design Space Exploration Flow

As we pointed out in the previous section, exhaustively searching in such a tremendous design space is impractical. As a result, we propose the following strategies to prune the design space:

Small loop flatten: Empirically, it is better to flatten the innermost loops with fixed, small trip counts. For one thing, it provides more opportunities for HLS to generate a more efficient scheduling. For another, it exerts moderate pressure on the overall resource utilization. As a result, we make an ad hoc strategy to fully unroll innermost loops with trip count less than 16.

Loop unroll factor pruning: Loop unroll factors determine the number of on-chip BRAM partitions. This number is bounded by the total number of BRAM blocks available for user-defined accelerators, which is approximately a few thousand. This pruning strategy is particularly beneficial for programs with deep, complicated loop hierarchy.

Saddleback search for loop unroll factors: The search problem of all loop unroll factors can be formulated as finding a particular value in a N-dimension matrix where the values are sorted in each individual dimension. N denotes the total number of loops. The formulation is based on the following theorem (the proof is omitted due to page limit).

THEOREM 1. *For unroll factor L_α of loop L in the design parameter set, the overall execution cycle C is negatively correlated to L_α; the consumption of any type of resource R is positively correlated to L_α.*

By applying the Saddleback search algorithm [7] to the formulated problem, we can reduce the time complexity of searching all loop unroll factors from $O(\prod_{L \in \mathcal{L}})$ to $O(\prod_{L \in \mathcal{L} \wedge L \notin \{L_p, L_q\}} \times L_p \times \log \frac{L_q}{L_p})$, where L_q and L_p denote the unroll factors of the two loops with the largest trip counts. This strategy works very well for programs with shallow loop hierarchies.

Fine-grained pipeline pruning: In general, loop pipelining achieves higher resource utilization and better performance than parallelism in most cases. Formally, we derive the following theorem to realize the loop that is always benefit pipeline (the proof is omitted due to page limit.)

THEOREM 2. *Given a loop L with trip count L_{tc}, iteration latency C_L and resource consumption R_L^{np} before enabling pipelining, and initiation interval II_L and resource consumption R_L^{p} after enabling pipelining. Enabling pipelining is always better if $\frac{L_\alpha}{L_{tc}} \leq (e-1)$ for unroll factor L_α of L, where $e = \frac{C_L/II_L}{R_L^p/R_L^{np}}$.*

The e in Theorem 2 means the efficiency of enabling pipelining for loop L. Theorem 2 illustrates that when $e \leq 1$, the pipeline implementation is inherently inefficient and should always be disabled. On the other hand, the pipeline implementation is much more efficient than the sequential design and should always be enabled when $e \geq 2$. Finally, when $1 < e < 2$, the unroll factor should not be too large so that the pipelined PE is able to process a sufficient number of loop iterations to ensure the pipeline efficiency.

Power-of-two buffer bit-widths and capacities: We prune the design space by only searching the power-of-two bit-width and capacity values for each buffer. We note that this pruning strategy covers the optimal design point because 1) the BRAM utilization would be the same for all bit-width values that round up to the same power-of-two value, and 2) setting the capacity to be a power-of-two value achieves the highest efficiency for the buffer control logic and is enabled in commercial HLS tools by default.

Taking our motivating example as an instance, the design space is reduced from 1.4×10^{17} to only 3.2×10^{6} by applying the above strategies. The scale of reduced design space is sufficient to be searched within an hour even using a single modern CPU core.

5 EXPERIMENTAL EVALUATION

In this section we first present the AutoAccel framework that automates the entire accelerator generation process. Then we describe our experimental setup, followed by the evaluation of the model accuracy as well as the performance and energy efficiency of the generated accelerators.

5.1 AutoAccel Framework

As shown in Fig. 4, we implement a push-button framework called AutoAccel that takes a nested loop in C as input and performs a series of transformations to produce a high-quality FPGA accelerator under the CPP microarchitecture.

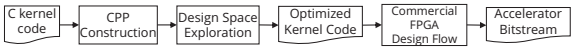


Figure 4: AutoAccel Framework Overview

AutoAccel is implemented on top of the Merlin compiler [4, 9, 10], a source-to-source transformation tool for FPGA acceleration based on the CMOST [27] compilation flow. The Merlin compiler provides a code transformation library which is leveraged by AutoAccel to agilely construct the CPP microarchitecture. Subsequently, we use static analysis to extract the necessary information (e.g., loop trip count) to form the design space. Then the design space exploration flow we introduced in the previous section is adopted to realize the best design specification in minutes. This design can be directly fed into Xilinx SDAccel to produce a high-quality accelerator bitstream.

5.2 Experimental Setup

The evaluation of AutoAccel is performed on the mainstream PCIe-based CPU-FPGA platform with the Xilinx SDAccel design flow. Table 1 lists the detailed hardware and software configuration.

Table 2 lists the benchmarks used in our experiment. To evaluate the AutoAccel framework, we use the MachSuite [25] benchmark suite that contains a broad class of computational kernels programmed as C functions for accelerator study. For each kernel, MachSuite provides at least one implementation that is programmed without the consideration of FPGA acceleration, which makes it a natural fit for demonstrating AutoAccel.

Table 1: Configuration of Hardware and Software

Host CPU Model	Intel Xeon E5-2420 @ 1.9GHz
Host Memory	64GB DDR3-1600
FPGA Fabric	Xilinx Virtex-7
Device Memory	8GB DDR3-1600 (Max Band.: 12.8GB/s)
CPU-FPGA Interface	PCIe Gen3 x8 (Max Band.: 8GB/s)
Transformation Flow	Merlin compiler 2017.1
Synthesis Flow	SDAccel (SDx) 2017.2

Table 2: Benchmark Description

Kernel	Description and Input Information
AES	Advanced encryption standard. Input: 256-bit key; 64MB data.
GEMM	General matrix multiplication. Input: two 1024×1024 double-precision matrices
KMP	Knuth-Morris-Pratt string matching. Input: 128MB string; 16B substring.
NW	Needleman-Wunsch sequence alignment. Input: 64K pairs of 128-nucleotide seq.
SPMV	Sparse matrix-vector multiplication. Input: 4096×512 ELLPACK data and index.
VITERBI	Viterbi algorithm. Input: 1M 128-element chains.
FFT	Fast Fourier transform. Input: 65536 strides each with 1KB size.
STENCIL	Stencil computation. Input: a 4096×4096 image

5.3 Evaluation Results

We first evaluate the error rate between the model-generated result and the HLS report. The average error rate for cycle count, BRAMs, DSPs, LUTs and FFs are only <1%, <1%, <1%, 6.5% and 4.3%, indicating that the proposed model aligns to the HLS report accurately. We then compare this result with the actual on-board result, and list the error rate for each benchmark in Table 3. We can see that the average error rate among all the benchmarks is only 6.2%. We further analyze the benchmarks with over 10% error rate, i.e., `AES` and `KMP`. We find that such a relatively large error rate is mainly because the accelerator designs for these benchmarks have a very small execution time (~10 ms). For these time frames, the start-up and end overhead bias the time significantly. On the contrary, we also observe that the error rate of the model to on-board execution is always less than 5% when a design has an over 100-millisecond execution time. Hence, the proposed model is able to accurately predict the on-board execution time of a design given that its execution time is tens of milliseconds or larger.

Table 3: Error Rates Between Model and On-board Results

Bench.	AES	SPMV	KMP	FFT	VITERBI	NW	STENCIL	GEMM
Avg. err.	13.5%	9.5%	12.2%	0.1%	2.1%	1.1%	7.7%	3.3%

We then evaluate the performance improvement of the generated FPGA accelerator designs. Figure 5 compares the performances between the naive implementation of MachSuite, AutoAccel-generated accelerator designs and manual HLS designs, all of which are normalized to the performances of the corresponding software implementations. We can clearly see that AutoAccel-generated accelerators outperform the naive implementations by 27,000x, indicating that AutoAccel dramatically improves the quality of accelerator designs without manual programming effort. Meanwhile, the AutoAccel-generated accelerators also outperform the software implementations by 72x, indicating that our approach does lead to competitive accelerator designs.

We can also see that the manual designs only outperform the AutoAccel-generated designs by an average 2.5×, even after we spent several days to weeks performing more sophisticated code reconstruction to each kernel. In fact, the AutoAccel-generated designs for the `AES`, `SPMV`, `KMP` and `STENCIL` kernels have already achieved the optimal performance since they have fully utilized the off-chip bandwidth. Although we are able to further improve the

performance of other kernels by manually applying very specialized circuit designs not covered by AutoAccel, e.g., Race Logic [17] for the NW kernel, AutoAccel still preserves a high quality of results while substantially reducing the programming effort.

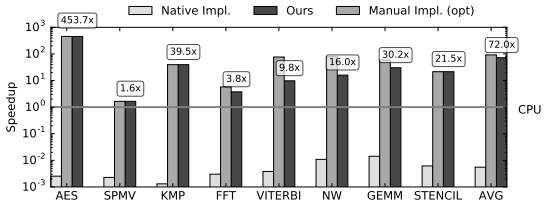


Figure 5: Speedup over an Intel Xeon CPU Core

Finally, we analyze the energy efficiency gain of AutoAccel-generated designs. We estimate the energy efficiency (performance per watt) of our experiments by considering execution time and thermal design power (TDP). The TDP of the Intel Xeon CPU and the Xilinx FPGA used in this comparison is 80W and 25W, respectively. Accordingly, AutoAccel-generated designs can achieve up to a 1677.9× energy efficiency improvement, and 260.4× on average.

6 RELATED WORK

Many previous studies have attempted to propose various solutions to address individual design optimization problems. For example, [14, 21, 22] focus on the problem of on-chip memory partitioning; [11, 15] deal with processing element duplication; [13] handles the improvement of off-chip bandwidth utilization. While these studies model the trade-offs between different design choices and realize the optimal choice via automatic design space exploration, they do not take inter-strategy trade-offs into consideration.

On the other hand, a few recent studies have started paying attention to the interaction of different optimization strategies and global optimization of accelerator designs. [26] and [31] conduct accelerator performance analysis and provide valuable guidances for hardware designers to make good use of various optimization strategies. However, since they do not come up with an automation solution, accelerator developers still have to manually conduct design space exploration. Other representative studies [16, 23] provide a collection of parallel primitives to allow programmers to describe the hardware functionality using parallel patterns. Compared to [16, 23], AutoAccel features an automation solution that transforms input computational kernels programmed without considering FPGA acceleration into high-quality accelerator designs.

Performance and resource modeling for FPGA designs is also gaining interest in recent years [8, 16, 26, 28–31]. [8, 31] present a performance model for HLS-based FPGA designs, but lacks quantitative depiction of resource consumptions. [28, 29] provides a more comprehensive model with the consideration of DSPs and BRAMs, but it does not model the consumption of LUTs which can also be the resource bottleneck in FPGA designs in many cases. In addition, [28, 29] aims to improve the performance by realizing the optimal HLS directives without code transformation. As a result, its qualify of results highly depends on the structure and coding style of the user input kernel code. [16, 30] leverages machine learning to model the LUT consumption. However, the model has to be trained for each specific tool implementation, which means that the model has to be retrained once the tool is changed or updated.

7 CONCLUSION

While the CPU-FPGA heterogeneous architectures are becoming a promising paradigm for providing continued performance and energy improvement in modern datacenters, accelerator programming arises as a serious challenge to application developers. In this paper we propose the AutoAccel framework to provide a nearly push-button experience on mapping C functions into high-quality FPGA accelerator designs. Featuring the CPP microarchitecture, analytical-based design space exploration and automatic code transformation, AutoAccel achieves 72x speedup and 260.4× energy improvement for a broad class of computation kernels.

Furthermore, we believe that the design principles of AutoAccel can be further generalized to stimulate more research on the adoption of FPGAs in datacenters. The CPP microarchitecture serves as a proof-of-concept that using accelerator design templates as specifications of the program-to-behavioral-description transformation fundamentally reduces the design space while preserving the accelerator quality. Hence, more microarchitectures, with their analytical models and code transformation techniques, might be added in AutoAccel to improve the coverage of computation kernels.

REFERENCES

[1] Amazon EC2 F1 Instance. https://aws.amazon.com/ec2/instance-types/f1/
[2] Intel SDK for OpenCL. https://software.intel.com/en-us/intel-opencl.
[3] Intel to Start Shipping Xeons With FPGAs in Early 2016. http://www.eweek.com/servers/intel-to-start-shipping-xeons-with-fpgas-in-early-2016
[4] Merlin Compiler. http://www.falcon-computing.com/index.php/solutions/merlin-compiler
[5] SDAccel Development Environment. http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[6] Xeon+FPGA Platform for the Data Center. https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf
[7] R. S. Bird. 2006. Improving Saddleback Search: A Lesson in Algorithm Design. In *Mathematics of Program Construction*. Springer.
[8] Y.-k. Choi et al. 2017. HLScope+: Fast and Accurate Performance Estimation for FPGA HLS. In *ICCAD*.
[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. 2014. Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications. In *FPGA*.
[12] J. Cong et al. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD*.
[13] J. Cong et al. 2017. Bandwidth optimization through on-chip memory restructuring for HLS. In *DAC*.
[14] J. Cong et al. 2012. Optimizing Memory Hierarchy Allocation with Loop Transformations for High-level Synthesis. In *DAC*.
[15] A. Hagiescu et al. 2009. A computing origami: Folding streams in FPGAs. In *DAC*.
[16] D. Koeplinger et al. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *ISCA*.
[17] A. Madhavan et al. 2014. Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms. In *ISCA*.
[18] S. B. Needleman et al. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *JMB*.
[19] J. Ouyang et al. 2014. Sda: Software-defined accelerator for largescale dnn systems. In *Hot Chips*.
[20] L. Page et al. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.
[21] N. K. Pham et al. 2015. Exploiting Loop-array Dependencies to Accelerate the Design Space Exploration with High Level Synthesis. In *DATE*.
[22] L.-N. Pouchet et al. 2013. Polyhedral-based Data Reuse Optimization for Configurable Computing. In *FPGA*.
[23] R. Prabhakar et al. 2016. Generating configurable hardware from parallel patterns. In *ASPLOS*.
[24] A. Putnam et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*.
[25] B. Reagen et al. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *IISWC*.
[26] Z. Wang et al. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *HPCA*.
[27] P. Zhang et al. 2015. CMOST: A System-level FPGA Compilation Framework. In *DAC*.
[28] J. Zhao et al. 2017. COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications. In *ICCAD*.
[29] G. Zhong et al. 2016. Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *DAC*.
[30] G. Zhong et al. 2017. Design Space Exploration of FPGA-based Accelerators with Multi-level Parallelism. In *DATE*.
[31] H. R. Zohouri et al. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *SC*.