Bandwidth Optimization Through On-Chip Memory Restructuring for HLS

Jason Cong Peng Wei Cody Hao Yu Peipei Zhou
University of California, Los Angeles
{cong, peng.wei.prc, hyu, memoryzpp}@cs.ucla.edu

ABSTRACT

High-level synthesis (HLS) is getting increasing attention from both academia and industry for high-quality and high-productivity designs. However, when inferring primitive-type arrays in HLS designs into on-chip memory buffers, commercial HLS tools fail to effectively organize FPGAs' on-chip BRAM building blocks to realize high-bandwidth data communication; this often leads to sub-optimal quality of results. This paper addresses this issue via automated on-chip buffer restructuring. Specifically, we present three buffer restructuring approaches and develop an analytical model for each approach to capture its impact on performance and resource consumption. With the proposed model, we formulate the process of identifying the optimal design choice into an integer non-linear programming (INLP) problem and demonstrate that it can be solved efficiently with the help of a one-time C-to-HDL (hardware description language) synthesis. The experimental results show that our automated source-to-source code transformation tool improves the performance of a broad class of HLS designs by averagely 4.8x.

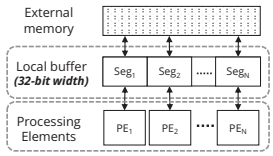
1. INTRODUCTION

The demand for high-performance, energy-efficient computing stimulates the adoption of field programmable gate arrays (FPGAs), which can accelerate a broad class of computational kernels while supplying flexibility of reconfiguration in datacenters. Leading datacenter operators, such as Microsoft and Baidu, have harnessed FPGA accelerators in important datacenter applications, e.g., search engines [1] and neural networks [2]. The Amazon Elastic Compute Cloud (EC2) also introduces FPGAs in its F1 compute instance [3]. Intel, with its $16.7 billion acquisition of Altera, predicts that 30% of servers will have FPGAs by 2020 [4], suggesting that FPGAs are becoming a mainstream acceleration technology. However, the long development cycle of the FPGA circuit poses a serious challenge to meeting the time-to-market requirement of an application. The situation may become even worse if the computational kernel to accelerate is prone to obsolescence.

High-level synthesis (HLS) [5], which allows synthesis of circuit designs from high-level programming languages, has captured increased attention from both academia and industry in an attempt to gain fast design delivery. Commercial tools, such as Xilinx Vivado HLS [6] and Intel SDK for OpenCL [7], can infer accelerator circuits directly from behavior-level descriptions and thus free designers from bothering with the register-transfer level (RTL) details. However, state-of-the-art HLS tools still demand a fair amount of manual effort to ensure quality of results. Specifically, a considerable number of directives in the form of specialized coding styles or annotations are necessary to guide HLS tools to infer high-quality RTL designs from high-level descriptions. Designers also need to manually conduct design space exploration, i.e., trying to find the best design choice among a great number of options. Although FPGA vendors and researchers are constantly making efforts, the automation of generating high-quality RTL designs is still far away from a perfect push-button process, and many issues are still open for research.

```
1  void kernel(int size, int *in, int *out) {
2    int buf_in_x[Npe][B_SIZE];
3    int buf_in_y[Npe][B_SIZE];
4    int buf_out_x[Npe][B_SIZE],
5    int buf_out_y[Npe][B_SIZE];
6    for (int i=0; i<size/B_SIZE+1; i++) {
7      // double-buffer scheduling
8      if (i % 2 == 0) {
9        load_data(buf_in_x, in+i*B_SIZE);
10       compute(buf_in_y, buf_out_y);
11       store_data(out+i*B_SIZE, buf_out_x);
12     }
13     else {
14       load_data(buf_in_y, in+i*B_SIZE);
15       compute(buf_in_x, buf_out_x);
16       store_data(out+i*B_SIZE, buf_out_y);
17     }
18   }
19 }
20 void compute(int** buf_in, int** buf_out) {
21   for (int i=0; i<Npe; i++) {
22   #pragma HLS unroll
23     // simplified due to page limit
24     ... = buf_in[i][j];
25   }
26 }
```

(a) HLS-C code



(b) Accelerator architecture

Figure 1: Motivating HLS-C example design.

In this paper we focus on one of such issues that has received less attention. Specifically, we notice that a primitive-type local array in HLS-C [8] will be inferred to a low-width on-chip buffer by C-to-HDL (hardware description language) synthesis, which results in poor bandwidth in the DRAM-BRAM data transfer. We use a simple HLS-C example design in Figure 1 to illustrate this phenomenon.[1] The *kernel* function describes an FPGA accelerator design which specifies two external memory buffers, *in* and *out*, for input and output, respectively. The *kernel* function also includes two sets of on-chip memory buffers, *buf_in_x(y)* (line 2) and *buf_out_x(y)* (line 3), with 32-bit width that is inferred from the 32-bit integer type. The buffers are all partitioned into *Npe* partitions to enable concurrent data supply, where *Npe* denotes the number of processing elements (PEs). The accelerator iteratively loads a certain size of data from the external to on-chip memory (line 7 or 12), processes them in parallel (line 8 or 13), and stores the output back to DRAM (line 9 or 14). We assume that common optimization strategies have already been applied, such as double buffering (lines 6-15), parallel computing with duplicated PEs (line 21) and memory partitioning for *buf_in_x(y)* and *buf_out_x(y)* to enable parallel access for each PE. The issue we study occurs in the data load and store phases. Since the local buffers are of 32-bit data width, only 32-bit data per cycle can be read (written) from (to) the external memory, even though the bus between the local buffers and the external memory can have up to 512-bit width in our experimental platform. In other words, the utilized bandwidth of DRAM-BRAM data transfer is only $\frac{32}{512} = 6.25\%$ of the max-

[1] Although being illustrated in HLS-C, the issue is generic to other HLS languages.

DAC '17, June 18-22, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062208

imum achievable bandwidth, which means that the design performance could be easily bounded by data communication.

An intuitive approach to address this issue is to let designers manually replace all primitive-type arrays with specialized large-width data types, such as the *ap_int<W>* type in HLS-C, where *W* is a customized data width. Such a transformation restructures the on-chip buffers to enable a larger data transfer width — thus a higher bandwidth. However, it often results in inefficient BRAM utilization and more BRAM consumption to enlarge the data width of a BRAM buffer with a certain capacity. Given such a trade-off between performance and resource consumption, designers have to run HLS tools a considerable number of times in order to identify the optimal solution.

In this paper we aim to provide an automatic on-chip buffer restructuring solution which identifies the optimal bit-width under resource constraints and performs code transformation. We make the following contributions:

- **Approaches.** We propose three buffer restructuring approaches, coarse-grained, fine-grained and hybrid, with detailed analysis of each to trade-off performance and resource consumption.
- **Analytical Models.** We develop analytical models for all three approaches to quantify the performance-resource trade-off, formulate the process of identifying the optimal solution into an INLP problem, and demonstrate that the problem can be efficiently solved by running the C-to-HDL synthesis only once per design.
- **Automation Tool.** To further relieve the burden of manual code transformation to realize the optimal solution, we implement a source-to-source transformation tool to automatically generate the improved HLS-C design. Our experimental results show that the automated tool improves the performance of a broad class of benchmarks by an average of 4.8x.

The remainder of the paper is organized as follows. Section 2 presents the three buffer restructuring approaches and analyzes the performance-resource trade-off on each approach. Section 3 delivers the analytical models for the three approaches. Section 4 presents the automated tool implementation and experimental results. Section 5 summarizes the related work. Section 6 concludes the paper and presents possible future work.

2. APPROACHES

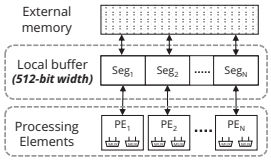
Section 1 presented the issue of inefficient DRAM bandwidth utilization due to the low-width on-chip buffers inferred from primitive-type arrays. In this section we propose three buffer restructuring approaches to address this issue, and perform a detailed analysis of the trade-off between performance and resource consumption for each approach.

2.1 Fine-Grained Approach

Since the source of the problem is that primitive-type arrays are inferred into low-width buffers, a natural solution is to declare and use local arrays as large-width data types. We call this the *fine-grained* approach. Figure 2(a) illustrates this approach through a code update of the example design in Figure 1, and Figure 2(b) shows the updated accelerator architecture. This update features two major changes. First, the two sets of local arrays, *buf_in_x(y)* and *buf_out_x(y)*, are both updated as the 512-bit data type – *ap_int<512>*. From the perspective of architecture, this code modification leads to a width increase in local buffers from 32 bits to 512 bits. As a result, the data transfer throughput between the external memory and the local buffers can reach up to 512 bits per cycle, maximizing the DRAM bandwidth utilization. Second, each array indexing operation is replaced with a complex switch statement (lines 15-21). In detail, to retrieve the *j-th* integer from the buffer *buf_in[i]* inside the *compute* function, the switch statement first locates the 512-bit data that contains the integer *buf_in[i][j/16]* and makes a 512-to-32 selection to extract the integer out. Such a switch statement is inferred into a multiplexer circuit to realize range selection by the HLS tool, as shown in Figure 2(b).[2]

```
1  void kernel(int size, ap_uint<512>* in,
2   ap_uint<512>* out) {
3   ap_uint<512> buf_in_x[Npe][B_SIZE/16];
4   ap_uint<512> buf_in_y[Npe][B_SIZE/16];
5   ap_uint<512> buf_out_x[Npe][B_SIZE/16];
6   ap_uint<512> buf_out_y[Npe][B_SIZE/16];
7   for (int i=0; i<size/B_SIZE+1; i++) {
8    /* The double-buffer scheduling remains
          unchanged, thus omitted */
9    ...
10  }
11 }
12
13 void compute(ap_uint<512>** buf_in,
14  ap_uint<512>** buf_out) {
15 #pragma HLS unroll
16  for (int i=0; i<Npe; i++) {
17   switch (j) {
18   case 0:
19   ... = buf_in[i][j/16].range(31, 0);
20   // intermediate cases are omitted
21   case 15:
22   ... = buf_in[i][j/16].range(511, 480);
23   }
24  }
25 }
```

(a) HLS-C code



(b) Accelerator architecture

Figure 2: Example of fine-grained approach.

In terms of throughput, the accelerator performance of data communication is significantly improved (by up to 16x in the example design) by applying the fine-grained approach; the computation time remains almost unchanged. Since the load, compute and store phases are overlapped (because of double buffering), the overall throughput will probably be improved if the accelerator is bounded by data communication. On the other hand, the fine-grained approach leads to extra LUT and BRAM consumption. For example, each switch statement for array indexing in the accelerator design is inferred into a multiplexer, which leads to extra LUT consumption; BRAM buffers with the same capacity but different bit-widths may experience up to 15x difference in BRAM consumption.

2.2 Coarse-Grained Approach

The fine-grained approach achieves performance improvement at the cost of LUT and BRAM consumption, and it requires modification to the PE's HLS-C description. An alternative approach avoids such modification and does not consume extra LUTs, with the price of extra Flip-Flop (FF) consumption as well as larger consumption of BRAM. We call this the *coarse-grained approach*, as illustrated in Figure 3.

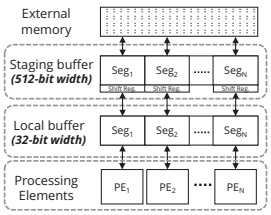
Compared to the fine-grained approach, the coarse-grained approach has two major differences. First, instead of enlarging the width of the local buffers, the coarse-grained approach adds an intermediate layer of BRAM buffers, named *staging buffers*, between the local buffers and the external memory, as shown in Figure 3(b). This layer, consisting of large-width buffers (512-bit in the example design), supplies a high DRAM-BRAM data transfer speed. The staging buffers, same as the local buffers, are partitioned into Npe partitions, where Npe denotes the number of PEs. This makes the data movement between the staging and local buffers in parallel. The other difference is illustrated in the *compute* function in Figure 3(a). Instead of letting PEs directly fetch data from large-width buffers, the coarse-grained approach requires a BRAM-BRAM data copy between the staging and local buffers before and after computation, which leaves the PE logic unchanged. While this leads to an addition overhead, it can be alleviated through the parallel data movement, and thus does not considerably affect the overall performance.

In terms of resource consumption, the coarse-grained approach consumes more BRAM blocks because of the existence of both staging and local buffers, but less LUTs since no extra multiplexer is needed. Meanwhile, it consumes extra FFs to realize data ex-

[2]Using the *range* API with variable lower and upper bounds can reduce the switch statement to one line of code, but causes over 10x LUT consumption.

```
1  void kernel(int size, ap_uint<512>* in,
2    ap_uint<512>* out) {
3    ap_uint<512> buf_in_x[Npe][B_SIZE/16];
4    ap_uint<512> buf_in_y[Npe][B_SIZE/16];
5    ap_uint<512> buf_out_x[Npe][B_SIZE/16];
6    ap_uint<512> buf_out_y[Npe][B_SIZE/16];
7    for (int i=0; i<size/B_SIZE+1; i++) {
8      /* The double-buffer scheduling remains
           unchanged, thus omitted */
9      ...
10   }
11 }
12
13 void compute(ap_uint<512>** kernel_buf_in,
14   ap_uint<512>** kernel_buf_out) {
15   int buf_in[Npe][B_SIZE];
16   int buf_out[Npe][B_SIZE];
17 #pragma HLS unroll
18   for (int i=0; i<Npe; i++) {
19     copy_data(buf_in[i], kernel_buf_in[i]);
20     ... = buf_in[i][j];
21     copy_data(kernel_buf_out[i], buf_out[i]);
22   }
23 }
```

(a) HLS-C code



(b) Accelerator architecture

Figure 3: Example of coarse-grained approach.

change (though usually not the bottleneck) between the staging and local buffers. Table 1 summarizes the impact of both approaches on performance and resource consumption.

Table 1: Impact of fine- and coarse-grained approaches on performance and resource consumption.

Approach	Fine-Grained	Coarse-Grained
Load Perf.	+	+
Compute Perf.	=	-
Store Perf.	+	+
LUT Consumption	+	=
BRAM Consumption	+	++
FF Consumption	=	+

2.3 Hybrid Approach

Section 2.1 and 2.2 make it clear that both the fine-grained and coarse-grained approaches have their pros and cons. The former establishes a positive correlation between the LUT consumption and the DRAM-BRAM bandwidth, and thus does not perform well for LUT-consuming accelerators. The latter consumes more on-chip memory and FFs, and introduces an overhead on computation. To fully cover the design space of both approaches and discover more design choices to find potentially better solutions, we propose a *hybrid* approach, as illustrated in Figure 4. Compared to the fine-grained approach, the hybrid approach decouples the correlation between the LUT consumption and the DRAM-BRAM bandwidth, thus allowing designs to have both a large DRAM-BRAM bandwidth and a low LUT consumption. Compared to the coarse-grained approach, the computation overhead exerted by the data exchange between the staging and local buffers can be alleviated with an enlarged local buffer width. As illustrated in the example, the staging-local data exchange is 4x faster than that in Figure 3, since the width of the local buffers is increased from 32 to 128 bits.

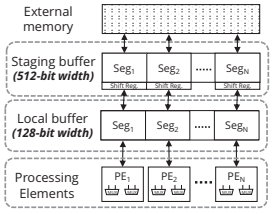
In summary, the hybrid approach enables more design choices to better balance the LUT, BRAM and FF consumption, and thus further improve the qualities of accelerator designs. However, such an advantage comes at the price of requiring designers to run HLS synthesis millions of times to try all possible combinations of (bw_c, bw_f, N_{PE}), where bw_c, bw_f and N_{PE} denote the width of the staging buffers, that of the local buffers, and the number of PE duplicates, respectively. To relieve this certainly overwhelming burden, this paper establishes analytical models for all three buffer restructuring approaches (Section 3), and implements a source-to-source transformation tool to recognize the optimal solution by running HLS synthesis only once (Section 4).

2.4 Problem Formulation

In this paper we target a HLS-C program with N_{PE} processing elements (PEs). Each PE has N_{ref} array references. The input data are processed in a coarse-grained three-stage pipeline, where the three stages are load, compute and store. In the load stage, the

```
1  void kernel(int size, ap_uint<512>* in,
       ap_uint<512>* out) {
2    ap_uint<512> buf_in_x[Npe][B_SIZE/16],
       buf_in_y[Npe][B_SIZE/16];
3    ap_uint<512> buf_out_x[Npe][B_SIZE/16],
       buf_out_y[Npe][B_SIZE/16];
4    for (int i=0; i<size/B_SIZE+1; i++) {
5      /* The double-buffer scheduling remains
           unchanged, thus omitted */
6      ...
7    }
8  }
9
10 void compute(ap_uint<512>** kernel_buf_in,
11   ap_uint<512>** kernel_buf_out) {
12   ap_uint<128> buf_in[Npe][B_SIZE /4];
13   ap_uint<128> buf_out[Npe][B_SIZE/4];
14 #pragma HLS unroll
15   for (int i=0; i<Npe; i++) {
16     copy_data(buf_in[i], kernel_buf_in[i]);
17     case 0:
18     ... = buf_in[i][j/4].range(31, 0);
19     // intermediate cases are omitted
20     case 3:
21     ... = buf_in[i][j/4].range(127, 96);
22     }
23     copy_data(kernel_buf_out[i], buf_out[i]);
24   }
25 }
```

(a) HLS-C code



(b) Accelerator architecture

Figure 4: Example of hybrid approach.

accelerator loads a certain size (denoted by S_{in}) of data with N_{task} independent tasks, which are processed in parallel by all PEs. The output size of the store stage is denoted by S_{out}. For ease of description, we assume the input and output data share the same bitwidth, denoted by bw_{ori}, but our methods are applicable to designs with non-uniform input/output types as well. Our objective is to derive the optimal combination of (bw_c, bw_f, N_{PE}) for each proposed approach to maximize the overall performance. Accordingly, the problem is defined as follows:

Problem Definition: Given a HLS-C program, determine the values of bw_c, bw_f and N_{PE} under resource constraints so that the performance is maximized.

Eq. 1 presents the overall execution cycle of the accelerator design.

$$C = max(C_{comp}, C_{load}, C_{store}) \tag{1}$$

where C_{comp}, C_{load}, and C_{store} are the execution cycles of the compute, load and store phases. Note that we assume the input design is implemented with double buffering so that the load, compute and store phases are able to be overlapped. As a result, the execution cycle is dominated by the most time-consuming phase.

Eq. 2 presents the execution cycle for each phase.

$$\begin{aligned} C_{comp} &= \frac{C_{PE} \times N_{task}}{N_{PE}} \\ C_{load} &= C_{init} + \frac{S_{in}}{bw_{ori}} \\ C_{store} &= C_{init} + \frac{S_{out}}{bw_{ori}} \end{aligned} \tag{2}$$

where C_{PE} is the latency of each PE in doing one individual job, and C_{init} is the initialization overhead of each external memory access, which is platform-dependent. Specifically, the value of C_{init} is 100 cycles in our experimental platform.

Meanwhile, the design has to meet the resource constraints, as shown in Eq. 3. B, L, and FF denote the use of on-chip BRAM blocks, LUTs, and flip-flops, respectively. Note that we ignore the DSP constraint since none of the proposed approachs will increase the DSP consumption.

$$\begin{aligned} B &\leq B_{total} \\ L &\leq L_{total} \\ FF &\leq FF_{total} \end{aligned} \tag{3}$$

To find the optimal solution to this problem, in Section 3 we discuss building an analytical model for each approach to capture its impact on performance and resource utilization.

3. ANALYTICAL MODELS

In this section, we model performance and resource utilization of proposed approaches. According to the models, we formulate the problem of identifying the optimal design choice to an integer non-linear programming (INLP) problem and demonstrate that this INLP problem can be solved efficiently by running the HLS synthesis only once.

3.1 Fine-Grained Model

The fine-grained approach affects the consumption of both LUTs and BRAMs, so we model both as follows. The cost of BRAM blocks to store a certain size of data in a certain width of BRAM buffer is modeled by Eq. 4.

$$V(s, bw) = \lceil \frac{s}{N_{blk}(bw) \times 18K} \rceil \times N_{blk}(bw) \tag{4}$$

where s denotes the size of data to store, and $N_{blk}(bw)$ represents the BRAM consumption of a unit bw-width buffer. The constant, 18K, is the capacity of a BRAM block. $N_{blk}(bw)$ is further modeled in Eq. 5, where 36 is the maximum achievable bit-width of a BRAM block.

$$N_{blk}(k) = \lceil \frac{k}{36} \rceil \tag{5}$$

We then use the above equations to model the total BRAM block usage after applying the fine-grained approach, which is the original block usage plus the extra BRAM blocks consumed by all transformed local buffers with larger bit-widths:

$$B = B_0 + N_{PE} \times \left[B_{PE} + \sum_{i \in buf} (V(S_i, bw_f) - V(S_i, bw_{ori})) \right] \tag{6}$$

where B_{PE} is the number of BRAM blocks within each PE, B_0 denotes other unaffected BRAM blocks, and buf is a set of all transformed local buffers. As can be seen, the difference between enlarged bit-width (bw_f) and the original bit-width (bw_{ori}) has a positive correlation to the additional BRAM block consumption.

In addition, as is mentioned in Section 2.1, the fine-grained approach consumes extra LUTs to form multiplexers for indexing a specific data. The multiplexer is implemented as a LUT-tree structure to select the necessary bit hierarchically. It means that if we want to index m bits from a buffer with n bit-width, we have to create m T-to-1 multiplexers ($T = \lceil \frac{n}{m} \rceil$), each of which forms a $\lceil log_4 T \rceil$-level LUT-tree. As a result, Eq. 7 quantifies the number of LUTs needed to construct an *n*-to-*m* multiplexer.

$$M(n, m) = m \times \sum_{i=1}^{\lceil log_4 T \rceil} \lceil \frac{T}{4^i} \rceil \tag{7}$$

Based on Eq. 7, we model the extra LUT consumption of the entire accelerator design in Eq. 8. Here, within each PE, N_{ref} is the number of indexing operations and L_{PE} is the original LUT consumption for each PE.

$$L = L_0 + N_{PE} \times (L_{PE} + N_{ref} \times M(bw_f, bw_{ori})) \tag{8}$$

The overhead of extra LUT consumption is positively related to the number of array indexing operations in a PE, and thus will become severe if a PE includes a large number of such operations. For example, the AES kernel (Advanced Encryption Standard, see Section 4) used in our experiments has 35 array references with 8-bit per reference. It implies that the LUT consumption will be increased by around 166× if we use 512-bit as the local buffer bit-width. As a result, the fine-grained approach is able to benefit the designs with a small number of array accesses.

In addition, the number of saved data transfer cycles is proportional to the ratio of enlarged local buffer bit-width to the original one (bw_f / bw_{ori}), as shown in Eq. 9.

$$\begin{aligned} C_{load} &= C_{init} + \frac{S_{in}}{bw_f} \\ C_{store} &= C_{init} + \frac{S_{out}}{bw_f} \end{aligned} \tag{9}$$

Finally, we want to mention that since the fine-grained approach does not fundamentally change the PE design, the PE latency remains unchanged.

3.2 Coarse-Grained Model

Since the coarse-grained method reconstructs on-chip memory by inserting a staging buffer with a larger bit-width between the external DRAM and PEs, it reduces data transfer cycles by increasing the memory bandwidth. Specifically, the number of saved data transfer cycles is proportional to the ratio of increased bit-width bw_c compared with bw_{ori}.

$$\begin{aligned} C_{load} &= C_{init} + \frac{S_{in}}{bw_c} \\ C_{store} &= C_{init} + \frac{S_{out}}{bw_c} \end{aligned} \tag{10}$$

Eq. 10 implies that the benefit of applying coarse-grained approach becomes observable when data size S is large enough.

On the other hand, different from the fine-grained approach, additional data transfer that required for the inserted staging buffer causes performance overhead, as shown in Eq. 11. The first term of Eq. 11 is the cycle number for transferring data from the staging buffer to the local buffer. As we have mentioned in Section 2.2, the data transfer can be done in parallel and will not be a bottleneck.

$$C_{comp} = \frac{S_{in} + S_{out}}{N_{PE} \times bw_{ori}} + \frac{C_{PE} \times N_{task}}{N_{PE}} \tag{11}$$

Meanwhile, FFs are used for implementing shift registers that are inferred by the HLS tool for laying out data from staging buffers to local buffers with relative small bit-widths. We model the cost of flip-flops for shift registers as follows.

$$FF = FF_0 + N_{PE} \times (FF_{PE} + N_{buf} \times bw_c) \tag{12}$$

On the other hand, similar to the fine-grained approach, we model the required on-chip memory for an inserted staging buffer as follows.

$$B = B_0 + N_{PE} \times \left(B_{PE} + \sum_{i \in buf} V(S_i, bw_c) \right) \tag{13}$$

Since the total on-chip memory capacity is usually a few megabytes, the value of bw_c may not be large if the design has already utilized most BRAM blocks.

3.3 Hybrid Method Model and Optimization

In order to apply both approaches simultaneously (hybrid method) and deal with the trade-off, we consider the pros and cons of both approaches as shown in Table 1. Clearly, since both approaches require more or less BRAM blocks, determining the bit-widths of staging buffers and local buffers forms a trade-off which highly depends on the design.

We formulate the trade-off as an Integer Non-Linear Programming (INLP) problem that is defined in Eq. 14-Eq. 25. Eq. 14-Eq. 19 are objective functions while Eq. 20-Eq. 25 are constraints. The binary variable t in Eq. 26 is used to indicate if the staging buffer is necessary to be inserted.

This formulation is an INLP problem because 1) several equations are not linear due to the *products* and *ceilings* and 2) all variables are integers. Although the INLP problem is NP-hard, the values of most variables are able to be inferred from either the design or hardware platform. Specifically, we categorize all variables into three types as shown in Table 2. Design-dependent variables can be retrieved via static analysis. Platform-dependent variables, such as resource capacities, can be acquired by the hardware specification. Both-dependent variables represent the baseline performance and resource utilization of the design on a specific platform, which can also be obtained by one-time C-to-HDL synthesis.

$$min: C = max\left(C_{comp}, C_{load}, C_{store}\right) \tag{14}$$

subject to

$$C_{load} = C_{init} + \frac{S_{in}}{bw_c} \tag{15}$$

$$C_{store} = C_{init} + \frac{S_{out}}{bw_c} \tag{16}$$

$$C_{comp} = t \times \frac{S_{in} + S_{out}}{N_{PE} \times bw_f} + \frac{C_{PE} \times N_{task}}{N_{PE}} \tag{17}$$

$$L = L_0 + N_{PE} \times (L_{PE} + N_{ref} \times M(bw_f, bw_{ori})) \tag{18}$$

$$B = B_0 + N_{PE} \times [B_{PE} + \sum_{i \in buf} (t \times V(S_i, bw_c) + V(S_i, bw_f) - V(S_i, bw_{ori}))] \tag{19}$$

$$FF = FF_0 + N_{PE} \times (FF_{PE} + t \times N_{buf} \times bw_c) \tag{20}$$

$$L \leq L_{total} \tag{21}$$

$$B \leq B_{total} \tag{22}$$

$$FF \leq FF_{total} \tag{23}$$

$$N_{PE} \geq 1 \tag{24}$$

$$bw_c \geq bw_f \geq bw_{ori} \tag{25}$$

$$t = \lceil \frac{bw_c - bw_f}{bw_c} \rceil \tag{26}$$

Table 2: Variable dependency categorization.

Type	Variables
Design	$S_{in}, S_{out}, N_{task}, N_{Ref}, B_{buf}, bw_{ori}, N_{PE}$
Platform	$L_{total}, B_{total}, FF_{total}, C_{init}, bw_{BRAM}$
Both	$C_{PE}, L_0, L_{PE}, B_0, B_{PE}, FF_{PE}, FF_0$

Consequently, we can find that as long as we are able to run the C-to-HDL synthesis once per design, the number of target variables is reduced to three (N_{PE}, bw_c and bw_f). As a result, the complexity of the design space exploration process is $O(PE \times max(bw_c) \times max(bw_f))$. In addition, we can further reduce the design options of solving the INLP problem by introducing other hardware specific constraints.

PE number limitation: Since the maximum PE number in a design is limited by the hardware resource (e.g. BRAM), the maximum PE number in a design is at most 2940 — the number of BRAM blocks on the FPGA fabric (although it is almost impossible to have 2940 PEs).

Bit-width limitation: The bit-width of an on-chip BRAM buffer that has interface to DRAM is limited to 512-bit as it is the maximum bit-width supported by in the experimental platform. This affects both bw_c and bw_f.

Efficient staging buffer bit-width: While restricted to no larger than 512-bit, the most efficient bw_c value is always dividable by 36:

THEOREM 1. *A design with* $\boldsymbol{x}$*-width staging buffers achieves the highest performance when* $\boldsymbol{x}$ *is dividable by 36.*

PROOF. Let the BRAM consumption of the x-width staging buffer be B_x. Assuming x is not dividable by 36, then there must exist another y-width staging buffer where $y > x \wedge y = 36 \times k (k \in \mathbb{Z})$ such that $B_y = B_x$. According to Eq. 10, the staging buffer bit-width is proportional to the performance when data size is large enough and the design is communication-bounded, so a design with y-width staging buffer has a higher performance than that of a design with x-width staging buffer. This contradicts the supposition. □

According to the theorem, we only need to explore $\lfloor \frac{512}{36} \rfloor = 14$ values of bw_c. Based on the these constraints, the solution space is reduced to $2940 \times 14 \times 512 \simeq 21M$. This number of design options is able to be explored exhaustively by a modern CPU in a few seconds.

4. EXPERIMENT

4.1 Implementation

We implement the proposed approaches via a source-to-source code transformation tool that is compatible with the Xilinx HLS design flow, as shown in Figure 5. Our implementation of code transformation is based on the ROSE compiler infrastructure [9] with Merlin compiler front-end APIs [10] which is developed by Falcon Computing Solutions [11] and based on the CMOST [12] compilation flow developed at UCLA. The framework takes a HLS-C program as an input and perform the C-to-HDL synthesis only once to collect necessary performance and resource utilization parameters listed in Table 2 to initialize the model. Then the design space exploration is performed for identifying the optimal design point (bw_c, bw_f, N_{PE}) for all approaches. Finally, the framework performs code transformation using the parameters to generate the optimized HLS-C design.

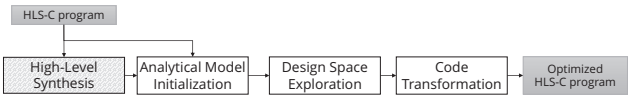


Figure 5: Overall execution flow.

4.2 Experimental Setup

This paper demonstrates the performance improvement lead by the automated tool using a set of HLS-C accelerator benchmarks in MachSuite [13]. MachSuite is a benchmark suite that consists of a broad class of HLS-C synthesizable accelerator designs. We apply a series of performance optimization strategies, including data tiling, pipelining, PE duplication and double buffering, to derive a set of designs with decent performance as the baseline. Table 4 presents a brief description as well as the resource utilization of each baseline design [3]. We can see that at least one type of resource has been fully utilized. The baseline and transformed designs are both synthesized using the Xilinx SDAccel development environment(v2015.4) [14], and executed on a Xilinx Virtex-7 board (Alpha Data ADM-PCIE-7V3) [15] that is equipped with a XC7VX690T-2 FPGA board, and a DDR3-1600 DRAM (12.8GB/s or 64 bytes/cycle off-chip bandwidth).

4.3 Results

Table 3 shows the performance improvement of the best designs in all three approaches. Based on the experimental results, we categorize the benchmarks into four classes, and discuss them as follows.

Coarse-grained preferred. The AES benchmark belongs to this category. We can see from Table 4 that the AES design is heavily LUT-bounded. The coarse-grained approach that features no extra LUT consumption hence serves as a natural fit for such a LUT-bounded design. As a result, the coarse-grained approach leads to the optimal design choice.

Fine-grained preferred. The NW, SORT and SPMV benchmarks belong to this category. These benchmarks are all BRAM-bounded, so the fine-grained approach that essentially trades BRAM with LUTs is better fitted and leads to the optimal design choice.

Hybrid preferred. The KMP kernel belongs to this category. We can see from Table 4 that the consumption of LUTs and BRAM are fairly balanced, i.e., it is not obviously bounded by a certain type of resource. As a result, the hybrid approach that carefully balances the LUT and BRAM consumption leads to the optimal solution.

Computation bounded. The VITERBI kernel belongs to this

[3] The percentage is based on the resources available for users, which is around 70% of those provided by the FPGA fabric.

Table 3: Performance comparison for proposed three approaches.

	Coarse-Grained			Fine-Grained			Hybrid			
	bw_c	$\Delta N_{PE}(\%)$	Speedup	bw_f	$\Delta N_{PE}(\%)$	Speedup	bw_f	bw_c	$\Delta N_{PE}(\%)$	Speedup
AES	504	0.0%	7.0x	96	30.0%	5.8x	N/A	504	0.0%	7.0x
KMP	108	65.9%	10.6x	64	60.9%	8.0x	36	108	65.9%	10.8x
NW	36	0.0%	2.4x	72	0.0%	2.5x	72	N/A	0.0%	2.5x
SORT	180	47.5%	1.9x	125	43.4%	2.1x	125	N/A	43.4%	2.1x
SPMV	504	50.0%	4.3x	432	50.0%	5.3x	432	N/A	50.0%	5.3x
VITERBI	32	0.0%	1.0x	32	0.0%	1.0x	32	32	0.0%	1.0x

Table 4: Benchmarks and resource utilization of baseline designs.

Bench.	Description	LUT	BRAM	FF
AES	Advanced encryption standard	99.4%	6.8%	30.9%
KMP	String matching	93.4%	97.8%	22.1%
NW	Needleman-Wunsch alignment	36.1%	96.4%	43.1%
SORT	Merge sort	81.6%	97.8%	49.6%
SPMV	Sparse matrix-vector mult.	30.0%	98.0%	25.6%
VITERBI	Viterbi DP algorithm	98.6%	85.9%	41.5%

category. The compute phase of VITERBI takes longer time than the load and store phases even before any buffer restructuring. Our solution is targeted at communication-bounded kernels and cannot benefit the kernel in this category.

In summary, the fine-grained approach is more suitable for the BRAM-bounded kernels, and the coarse-grained approach for the LUT-bounded kernels. The hybrid approach may find better design choices if the kernel is not clearly bounded by any individual type of resource.

5. RELATED WORK

High-Level synthesis optimization. Prior work proposed automatic transformation for on chip buffer data reuse, memory partitioning or memory merging [16] to improve performance [17] and minimize off-chip data volume [18, 19]. Polyhedral models are used to expose dependencies and data access in the loops and customized on-chip buffers are automatically generated. However, these work have not explored the DRAM-BRAM bandwidth optimization by altering data transfer bit-width using buffer reorganization. Our work is orthogonal to theirs and can be applied to their solutions to further increase the system performance. [20, 21] characterized DRAM-BRAM bandwidth with data transfer bit-width. They found that larger bit-width achieves higher bandwidth even when the data access size is the same. However, there are no analytic models on resource usage and cycle time with the bit-width, and data transfer bit-width are set as 512 as default in their design, which is not optimal for other applications.

Memory abstraction. LEAP [22], CoRAM [23], LMC [24] provided abstract memory management and hid data transfer and memory interface from users. LEAP [22] provided general-purpose cache like memories and built hierarchies for host memory, FPGA on-board memory and on-chip RAM buffers. LMC [24] was based on LEAP and provided compiler flow to do static program analysis to optimize memory access among different applications. CoRAM [23] provided high-level C-based language for application users to invoke memory operations called control actions which abstract away the detailed hardware support. Though the programmability and design portability are improved in these work, design choice that achieves the highest performance under resource constraints in terms of memory system are not discussed. Our work can potentially serve as the back-end support in these frameworks for better utilization of FPGA resources and higher performance.

6. CONCLUSION AND FUTURE WORK

In this paper, we target at the issue of inefficient DRAM bandwidth utilization due to primitive-type arrays in HLS-C, and propose three approaches to address them. With a carefully-design analytical model and automation tool, we are able to efficiently identify the optimal design choice and perform the code transformation automatically, supplying a nearly push-button experience to end users. The proposed approach is based on the execution pattern where the input data are processed in a tiled fashion so that a load-compute-store pipeline is formed. Hence, it does not cover the designs with extensive random accesses to a large memory footprint. This remains as possible future work.

Acknowledgments

This work is partially supported by the Center for Domain-Specific Computing under the NSF InTrans Award CCF-1436827, funding from CDSC industrial partners including Baidu, Fujitsu Labs, Google, Huawei, Intel and Mentor Graphics; C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We also want to thank Xilinx for FPGA boards and software donation.

7. REFERENCES

[1] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014.
[2] J. Ouyang *et al.*, "Sda: Software-defined accelerator for largescale dnn systems," in *Hot Chips*, 2014.
[3] "Amazon ec2 f1 instance," 2016. https://aws.amazon.com/ec2/instance-types/f1/.
[4] "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.html.
[5] J. Cong *et al.*, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.
[6] "Xilinx Vivado HLS." http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm.
[7] "Intel FPGA SDK for OpenCL." https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.
[8] "Vivado Design Suite User Guide: High-Level Synthesis." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf.
[9] "Rose Compiler Infrastructure." http://rosecompiler.org/.
[10] J. Cong, M. Huang, P. Pan, Y. Wang, and P. Zhang, "Source-to-source optimization for HLS," in *FPGAs for Software Programmers*, Springer International Publishing, 2016.
[11] "Falcon Computing Solutions, Inc." http://falcon-computing.com/.
[12] P. Zhang *et al.*, "CMOST: A system-level fpga compilation framework," in *DAC*, 2015.
[13] B. Reagen *et al.*, "Machsuite: Benchmarks for accelerator design and customized architectures," in *IISWC*, 2014.
[14] "SDAccel Development Environment." http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[15] "Alpha Data ADM-PCIE-7V3 datasheet."
[16] C. Pilato *et al.*, "System-level memory optimization for high-level synthesis of component-based socs," in *CODES*, 2014.
[17] W. Zuo *et al.*, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *FPGA '13*.
[18] L.-N. Pouchet *et al.*, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA '13*.
[19] C. Alias *et al.*, "Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for fpga," in *DATE '13*.
[20] Y.-k. Choi *et al.*, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *DAC '16*.
[21] C. Zhang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD '16*.
[22] M. Adler *et al.*, "Leap scratchpads: Automatic memory and cache management for reconfigurable logic," in *FPGA '11*.
[23] E. S. Chung *et al.*, "Coram: An in-fabric memory architecture for fpga-based computing," in *FPGA '11*.
[24] H.-J. Yang *et al.*, "Lmc: Automatic resource-aware program-optimized memory partitioning," in *FPGA '16*.