Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs

Yuan Zhou[1]*, Udit Gupta[2]★, Steve Dai[1], Ritchie Zhao[1], Nitish Srivastava[1], Hanchen Jin[1], Joseph Featherston[1], Yi-Hsiang Lai[1], Gai Liu[1], Gustavo Angarita Velasquez[3]★, Wenping Wang[4]★, Zhiru Zhang[1]*

[1] School of Electrical and Computer Engineering, Cornell University, USA
[2] Computer Science, Harvard University, USA
[3] Systems Engineering and Computer Science, National University of Colombia, Colombia
[4] Electronic and Information Engineering, Zhejiang University, China
*{yz882,zhiruz}@cornell.edu

ABSTRACT

Modern high-level synthesis (HLS) tools greatly reduce the turn-around time of designing and implementing complex FPGA-based accelerators. They also expose various optimization opportunities, which cannot be easily explored at the register-transfer level. With the increasing adoption of the HLS design methodology and continued advances of synthesis optimization, there is a growing need for realistic benchmarks to (1) facilitate comparisons between tools, (2) evaluate and stress-test new synthesis techniques, and (3) establish meaningful performance baselines to track progress of the HLS technology. While several HLS benchmark suites already exist, they are primarily comprised of small textbook-style function kernels, instead of complete and complex applications. To address this limitation, we introduce Rosetta, a realistic benchmark suite for software programmable FPGAs. Designs in Rosetta are fully-developed applications. They are associated with realistic performance constraints, and optimized with advanced features of modern HLS tools. We believe that Rosetta is not only useful for the HLS research community, but can also serve as a set of design tutorials for non-expert HLS users. In this paper we describe the characteristics of our benchmarks and the optimization techniques applied to them. We further report experimental results on an embedded FPGA device as well as a cloud FPGA platform.

ACM Reference Format:
Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *FPGA '18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174255

★ Udit, Gustavo, and Wenping conducted this research when they were affiliated with or visiting Cornell.

1 INTRODUCTION

Field-programmable gate arrays (FPGAs) have become an attractive option for realizing specialized accelerators thanks to their reconfigurability, massive fine-grained parallelism, and performance per watt advantage. With the extreme-scale integration of modern system-on-chip (SoC) and escalating design complexity of emerging applications, designing at a higher level of abstraction has become crucial to achieving high productivity. To address this challenge, high-level synthesis (HLS) tools have emerged to allow application developers to describe the hardware accelerator using common software programming languages like C/C++ by automatically generating RTL from behavioral descriptions [7, 14]. With the recent advances on HLS techniques and algorithms, modern HLS tools enable designers to explore optimization opportunities that are infeasible at the register-transfer level.

Programming FPGAs with HLS tools is drastically different from writing traditional software code. HLS users typically need to apply many optimization pragmas/directives to meet design constraints. The success of such manual optimization often requires nontrivial hardware design knowledge. For example, in image/video processing, the right combination of SRAM-based line buffers and shift registers is needed to achieve the ideal throughput and resource usage for pipelining the stencil code in hardware. With a more complex dataflow structure, the user needs to further calculate and specify the right FIFO depth to obtain the best pipeline rate without causing too much area overhead. However, these advanced HLS optimizations are rarely used or even required in the existing HLS benchmark suites (e.g., [11], [23]), which primarily include relatively small kernels that are designed to test some of the basic capabilities of an HLS tool such as the synthesis support of high-level language constructs. In addition, for HLS tool developers and the HLS research community at large, there is also a growing demand for a common set of realistic and complex designs to evaluate the efficacy of new synthesis techniques.

To this end, we introduce Rosetta[1] — a suite of realistic HLS benchmarks for software programmable FPGAs. Rosetta includes popular machine learning workloads such as logistic regression and neural network inference, as well as real-time video processing applications including image rendering and face detection. Unlike previous efforts, Rosetta presents fully developed applications instead of small kernel programs, and specifies realistic design constraints for each

[1]Rosetta gets the name following the convention of a plethora of "stone" benchmark suites. It also symbolizes that our benchmarks are specified in multiple languages (i.e., C++, OpenCL) and useful for evaluating HLS across different tools and platforms.

application. These design constraints are satisfied by applying advanced optimizations of state-of-the-art HLS tools, which are not exercised by existing benchmark suites. With these features, Rosetta is not only a set of practical benchmarks for the HLS community, but also a design tutorial on how to build specialized FPGA accelerators with advanced HLS optimizations. More concretely, our main contributions are threefold:

- We design and present Rosetta, which couples a range of realistic applications with real-world design constraints under different programming models. Current Rosetta designs are written in C++ and OpenCL. The synthesized hardware accelerators are tested on both embedded and cloud FPGA platforms.
- Rosetta demonstrates how to effectively apply advanced optimizations provided by modern HLS tools to meet the design constraints and achieve high quality of results. Examples of these optimizations include fixed-point optimization, dataflow pipelining, and data reuse through customized memory.
- The proposed benchmark suite is freely available in open-source format[2]. We plan to continuously improve Rosetta by strengthening current cases and adding new applications from other domains.

The rest of this paper is organized as follows: in Section 2, we introduce related work on HLS benchmarking and optimizations; Section 3 outlines the Rosetta applications and key HLS optimization techniques leveraged by them; details of each benchmark are described in Section 4; we show our experimental results in Section 5, and conclude this work in Section 6.

2 RELATED WORK

FPGA programming currently differs significantly from the common practice of software programming, even with the use of HLS tools. Instead of simply focusing on functional correctness and execution time, FPGA programmers often have to explore various complex design trade-offs involving performance, power, area, and cost. Therefore, traditional software benchmark suites cannot directly be applied to HLS evaluation. In response, a number of HLS-specific benchmark suites have been developed by the research community for evaluating various aspects of hardware synthesis techniques and tool flows. CHStone [11] is a widely used C-based HLS benchmark suite, which contains function kernels selected from application domains such as arithmetic, signal processing, and security. MachSuite [23] is another popular HLS benchmark suite, which includes a more diverse set of kernels and provides different algorithms for the same kernel to facilitate comparisons at the algorithmic level. A more recent effort, Spector [10], offers OpenCL benchmarks that are ready to be executed on Intel (formerly Altera) FPGA platforms. Kernels in Spector are designed to have large design spaces, which is useful for experimentation of automatic design space exploration (DSE) techniques. Additionally, HLS researchers have also adopted benchmarks from other communities. For example, Rodinia [5], originally designed for GPU benchmarking, has been used to test OpenCL-based HLS flows [29, 31]. Polybench [21] from the software compiler community has been adopted for assessing HLS-targeted polyhedral transformations [22, 34, 42] and DSE techniques [24, 29, 38, 40].

While the popular kernel benchmarks are simple to run and analyze, they are insufficient for evaluating the increased capabilities of HLS optimizations and new technology advances in FPGA devices. In particular, state-of-the-art HLS tools provide many advanced features for achieving high design quality. Examples include arbitrary-precision datatypes, parameterized hardware data structures (e.g., line buffers), and hierarchical dataflow pipelining. These features are often used in combination with other common HLS optimizations such as unrolling, loop pipelining [9, 15, 37], and array partitioning [30, 41]. Moreover, they are typically applied across multiple kernels exhibiting different characteristics to meet the stringent applicant-level design constraints.

We believe that a new set of full-application benchmarks is desirable to enable more realistic performance reporting of HLS tools and FPGA-based acceleration. Along this line, Liu et al. [16] conducted a comprehensive case study on an H.264 decoder, and they have open sourced their HLS implementation. Rosetta goes one step further by providing a suite of application benchmarks that can be used to (1) facilitate comparisons between HLS tools, (2) evaluate new synthesis techniques, and (3) establish meaningful baselines to track progress of the HLS and FPGA technologies. Each application in Rosetta includes a set of enforceable application-level design constraints based on real-world specifications. These constraints model the realistic use cases for FPGA-based hardware accelerators, which helps standardize the evaluation of future advancements in HLS tools. Furthermore, the applications in Rosetta leverage advanced features of HLS tools to achieve high quality of results (QoRs) across a distinct set of hardware designs. Hence these benchmarks can also serve as useful design tutorials for FPGA programmers to build high-performance hardware accelerators using HLS.

3 ROSETTA OVERVIEW

Rosetta currently contains six realistic benchmarks selected from machine learning and video processing fields, where FPGAs are competitive on energy efficiency compared to CPUs and GPUs.[3] For each Rosetta design, we provide the unoptimized software version, and the optimized HLS implementations written in either C++ or OpenCL. Table 1 lists the current Rosetta collection. Two of these benchmarks, binarized neural network and face detection, are adopted from our previously published work [25, 39], while the rest are new designs. Rosetta contains both compute-bound and memory-bound applications comprised of a rich set of kernels. These applications and kernels expose diverse sources of parallelism. Our current HLS implementations typically exploit instruction-level parallelism (ILP) through fine-grained pipelining, and in some cases also expose task-level parallelism (TLP) by overlapping the execution of different kernels. Additionally, each benchmark is associated with realistic design objectives — the machine learning applications require either low latency or high throughput depending on the use-case scenario, while video processing applications must meet a real-time throughput target of at least 30 frames per second. In order to achieve these application-level constraints, Rosetta designs are customized using a variety of HLS optimization techniques, which are concisely summarized as follows:

- **Datatype customization** – Customized data types such as fixed-point types allow an FPGA accelerator to compute at the desired numerical accuracy, and often lead to significant performance and area improvements over the design using full-precision floating-point types.

[2]Released on Github at https://github.com/cornell-zhang/rosetta

[3]For the time being, we are not targeting traditional benchmarks from cryptography (e.g., AES) and digital signal processing (e.g., DCT, FFT), since they are already included in several other benchmark suites [10, 23].

Table 1: The current set of the Rosetta applications — Rosetta contains both compute-bound and memory-bound applications with different workloads. Kernels in each application expose different sources of parallelism: SLP = subword-level parallelism; DLP = data-level parallelism; ILP = instruction-level parallelism. Different types of parallelism available in each compute kernel are listed in parentheses.

Application	Categorization	Major Compute Kernels	Major HLS Optimizations
3D Rendering	Video processing Compute bound Integer operation intensive	Integer arithmetics (ILP)	Dataflow pipelining Communication customization
Digit Recognition	Machine learning Compute bound Bitwise operation intensive	Hamming distance (SLP, DLP, ILP) KNN voting (ILP)	Loop unrolling Loop pipelining
Spam Filtering	Machine learning Memory bound Fixed-point arithmetic intensive	Dot product (DLP, ILP) Scalar multiplication (DLP, ILP) Vector addition (DLP, ILP) Sigmoid function (ILP)	Dataflow pipelining Datatype customization Communication customization
Optical Flow	Video processing Memory bound Floating-point arithmetic intensive	1D convolution (DLP, ILP) Outer product (DLP, ILP)	Dataflow pipelining Memory customization Communication customization
Binarized Neural Network (BNN) [39]	Machine learning Compute bound Bitwise operation intensive	Binarized 2D convolution (SLP, DLP, ILP) Binarized dot product (SLP, DLP, ILP)	Memory customization Datatype customization Communication customization
Face Detection [25]	Video processing Compute bound Integer arithmetic intensive	Image scaling (DLP, ILP) Cascaded classifiers (DLP, ILP)	Memory customization Datatype customization

- **Compute customization** – Compute customization improves the latency and/or throughput of the design through parallelization and pipelining. Loop unrolling, loop pipelining, and dataflow pipelining fall into this category.
- **Memory customization** – FPGA accelerators typically demand very high on-chip memory bandwidth to enable highly distributed control and computation. Therefore, it is critical to set up customized memory hierarchy to provide the required bandwidth through data reuse and memory banking.
- **Communication customization** – The limited data bandwidth between off-chip memories and the FPGA accelerators often becomes the performance bottleneck for memory-bound applications. Hence it is crucial to customize the communication channel and protocol used by the hardware accelerator to fully utilize off-chip memory bandwidth through proper data packing and careful design of the data layout.

4 BENCHMARK DESCRIPTION

This section discusses Rosetta applications in detail. For each benchmark, we first briefly introduce its functionality and design constraints; we then describe its major compute kernels, explain the rationale behind our categorizations in Table 1, and discuss the key HLS optimizations applied to this design.

4.1 3D Rendering

The 3D rendering benchmark renders 2D images from 3D triangle mesh models [20]. Taking in 3D coordinates of triangle vertices, the application projects the triangles onto a 2D image, and colors the image pixels according to the "altitude" of the projected triangle. Our implementation works on 256x256 images where pixels are represented with 8-bit integers. The provided dataset contains the coordinates of 3192 triangles. Target throughput is 30 frames per second.

```
TRIANGLES: for (int i = 0; i < NUM_3D_TRI; i++) {
  #pragma HLS dataflow
  // five stages for processing each 3D triangle
  projection(triangle_3ds, &triangle_2ds, angle);
  flag = rasterization1(triangle_2ds, max_min,
                        &triangle_2ds_same, max_index);
  size = rasterization2(flag, max_min, max_index,
                        triangle_2ds_same, fragment);
  size_pixels = zculling(i, fragment, size, pixels);
  coloringFB(i, size_pixels, pixels, frame_buffer);
}
```

Figure 1: Main loop for 3D Rendering. One triangle is processed by five image processing stages in each iteration.

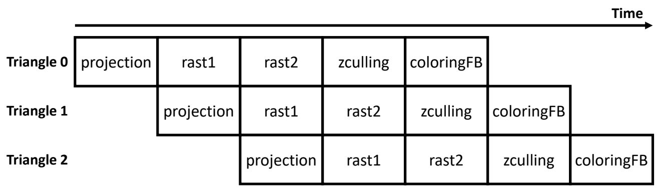

Figure 2: Dataflow optimization overlaps different pipeline stages in 3D rendering.

The HLS design contains a typical image processing pipeline as shown in Figure 1. The coordinates of each triangle go through four kernel functions before updating the output frame buffer in `coloringFB`. Integer operations form the primary workload inside the kernels: `projection` and `rasterization2` are rich in integer

```
 1 __local WholeDigitType training_set[NUM_TRAINING]
 2 __attribute__((xcl_array_partition(block,PAR_FACTOR,1)));
 3
 4 __attribute__((xcl_pipeline_loop))
 5 TRAINING_LOOP:
 6 for (int i = 0; i < NUM_TRAINING / PAR_FACTOR; i ++) {
 7   __attribute__((opencl_unroll_hint))
 8   LANES:
 9   for (int j = 0; j < PAR_FACTOR; j ++) {
10     // Read a new instance from the training set
11     int train_id = j * NUM_TRAINING / PAR_FACTOR + i;
12     WholeDigitType training_instance;
13     training_instance = training_set[train_id];
14     // Update the KNN set
15     update_knn(test_instance, training_instance,
16                &knn_set[j*K_CONST]);
17   }
18 }
```

Figure 3: Main compute loop nest for KNN calculation in OpenCL.

arithmetic, while `rasterization1` and `zculling` are heavy on integer comparisons. Each triangle requires a large amount of computation relative to its memory size. Therefore, the application is categorized as compute-bound.

3D rendering is a prime example of dataflow optimization, which is applied in the HLS code on line 2 of Figure 1. Dataflow optimization exploits task-level parallelism by overlapping different stages of the image processing pipeline, as shown in Figure 2. Although the latency of processing each triangle is not reduced, dataflow optimization improves throughput and ensures no hardware module in the pipeline is idle in the steady state.

Design parameters. We provide a switch in the source code to enable/disable dataflow optimization.

4.2 Digit Recognition

Digit recognition classifies hand-written digits using the K-nearest-neighbor (KNN) algorithm. The application works on a downsampled subset of the MNIST database [13], with 18000 training samples and 2000 test samples evenly split amongst the ten digit classes. Each MNIST image is downsampled to 14x14 and each pixel is represented as a single bit; thus, each image can be stored as a 196-bit unsigned integer. The KNN algorithm computes the Hamming distance between a test input and each training sample, stores the labels of the training samples with the `K` shortest distances, and votes among the `K` labels to decide the label of the test sample. The design objective for digit recognition is to minimize the total latency of classifying the 2000 test samples.

Digit recognition includes two major compute kernels: Hamming distance calculation and KNN voting. The Hamming distance kernel computes the Manhattan distance between two samples; as each sample is comprised of 1-bit pixels, this is done via bitwise XOR on the inputs, followed by computing a population count of the result. The kernel is therefore rich in bitwise logic. The Hamming distance must be calculated between a test input and every training sample. As a result, Hamming distance calculation is the dominant workload of digit recognition. The KNN voting kernel examines the list of Hamming distances to find the `K` nearest training samples, and outputs the classification result as the most frequent label amongst them. The main workload in this kernel is integer comparison and sorting. These two kernels have very different characteristics: while we can easily exploit the bit-level and data-level parallelism in the Hamming distance kernel, the KNN voting kernel is harder to parallelize.

Digit recognition has a high compute to communication ratio. For each test instance, Hamming distance calculation requires 100s-1000s of cycles depending on the parallelization factor, and KNN voting requires 10s-100s of cycles depending on `K` and the parallelization factor. The training samples and their labels are stored on-chip and reused for all test instances. As a result, digit recognition is a compute-bound application.

Figure 3 shows the main compute loop nest for KNN calculation, alongside key HLS optimizations. `TRAINING_LOOP` iterates over training samples, while the inner loop, `LANES`, instantiates different Hamming distance units. In addition to compute optimizations in the form of loop pipelining and unrolling (lines 4 and 7 of Figure 3), memory optimization is needed since the default implementation of on-chip array `training_set` only has two memory ports, it cannot supply `PAR_FACTOR` training instances per cycle. The `training_set` array is partitioned in line 2. With these optimizations, we can exploit the data-level parallelism between training instances.

Design parameters. The user can tune the following knobs:

- `K`: number of nearest neighbors.
- `PAR_FACTOR`: number of parallel Hamming distance units.

These two parameters present an interesting trade-off between classification accuracy, latency, and resource utilization. Increasing `PAR_FACTOR` reduces the latency of the Hamming distance kernel, but complicates the KNN voting kernel. Parallelization also causes frequency to drop. Furthermore, the complexity of both kernels increases with `K`. Additional results and analysis on the design space are presented in Section 5.

4.3 Spam Filtering

The spam filtering application uses stochastic gradient descent (SGD) to train a logistic regression (LR) model for spam email classification [19]. The input is a dataset containing 5000 emails, 4500 for training and 500 for testing [26]. Each email is represented as a 1024-dimensional vector whose elements are relative word frequencies stored as 16-bit fixed-point numbers. The SGD training process produces a vector of 32-bit fixed-point parameters for the LR model. We use five training epochs and a minibatch size of one; each epoch processes every training sample once and updates the parameters after each sample.

The performance target of spam filtering is to minimize training latency. Critical resource constraints are the number of hardened DSP blocks and the size of on-chip storage, which limits the level of compute parallelization and the amount of data stored on the FPGA. The SGD algorithm contains kernels commonly found in machine learning applications, including dot product, vector addition, and sigmoid.

Our spam filtering design exploits datatype customization and approximation of complex arithmetic operations on the FPGA. Figure 4 shows the optimized sigmoid function. Lines 1-3 show the customized datatypes used to avoid expensive floating-point arithmetic. We also eliminate most of the compute by taking advantage of the properties of the sigmoid function. Sigmoid asymptotically approaches one when the input is large and zero when the input is small (i.e. large negative). Sigmoid values when the input is between minus four and four are hardcoded in a look-up table.

```
typedef fixed<F_TWIDTH,F_IWIDTH> FeatureType;
typedef uint<LUT_TWIDTH> IdxFixed;
typedef fixed<LUT_TWIDTH, LUT_IWIDTH> LutInFixed;
// values of sigmoid function stored in a look-up table
FeatureType useLUT(LutInFixed in) {
  IdxFixed index;
  if (in < 0) {
    in = -in;
    index = LUT_SIZE - (in << (LUT_TWIDTH - LUT_IWIDTH));
  }
  else
    index = (in << (LUT_TWIDTH - LUT_IWIDTH));
  return lut[index];
}
// sigmoid function
FeatureType Sigmoid(FeatureType exponent) {
  if (exponent > 4)
    return 1.0;
  else if (exponent < -4)
    return 0.0;
  else {
    LutInFixed inLut = (LutInFixed)exponent;
    return useLUT(inLut);
  }
}
```

Figure 4: Datatype and compute optimization to the Sigmoid function — Specialized datatypes are used throughout the whole hardware function to avoid expensive floating-point arithmetic. We use a look-up table to store the values of the sigmoid function so that the complex arithmetic operations can be reduced. In our implementation F_TWIDTH = 32, F_IWIDTH = 13, LUT_TWIDTH = 12, LUT_IWIDTH = 4.

```
typedef uint<VDWIDTH> VectorDataType;
typedef fixed<D_TWIDTH, D_IWIDTH> DataType;
void read_data(VectorDataType*  data,
               DataType*        training,
               int tid)
{
  for (int i = 0; i < N_FEATURES/(VDWIDTH/D_TWIDTH); i++) {
    #pragma HLS pipeline
    // read in the data
    int idx = tid * N_FEATURES / (VDWIDTH/D_TWIDTH) + i;
    VectorDataType tmp = data[idx];
    // distribute into local buffer
    for (int j = 0; j < (VDWIDTH/D_TWIDTH); j++) {
      int loc_idx = i * (VDWIDTH/D_TWIDTH) + j;
      training[loc_idx] = tmp((j+1)*D_TWIDTH-1, j*D_TWIDTH);
  }}
}
```

Figure 5: Communication optimization for spam filtering — In our implementation D_TWIDTH = 16, D_IWIDTH = 4, N_FEATURES = 1024. Users can tune the VDWIDTH parameter to control the off-chip communication bandwidth.

Our target FPGA devices do not have sufficient on-chip memory to store the complete training set, necessitating the streaming of training instances from off-chip memory. Dataflow optimization (introduced in Section 4.1) is applied to overlap communication and compute. To fully utilize off-chip memory bandwidth, we apply element packing as shown in Figure 5. Data is transferred from off-chip storage as VectorFeatureType, which is a wide, custom-bitwidth

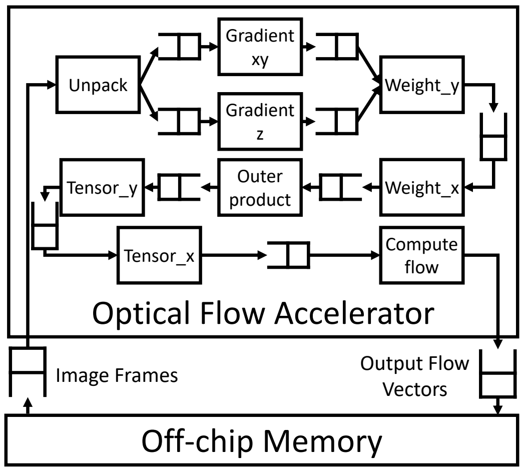

Figure 6: Hardware diagram for optical flow — The kernels are connected by FIFOs for streaming dataflow pipelining.

integer type. Inside the FPGA, the data is unpacked into 16-bit training vector elements, resulting in a communication throughput of multiple elements per cycle. Despite this optimization, the throughput of the dataflow pipeline is still determined by the communication latency because of the relatively simple and highly parallelized compute units for LR. Therefore, spam filtering is classified as a memory-bound application.

Design parameters. The design space of spam filtering consists of the following parameters:

- PAR_FACTOR: the parallelization factor of the vector compute kernels.
- VDWIDTH: the width of the packed vector data type, which controls the upper-bound of the off-chip communication bandwidth of the hardware function.

Our results and analysis on the design space are shown in Section 5.

4.4 Optical Flow

Optical flow captures the motion pattern of objects between consecutive image frames. It is an important step for object detection and is integrated into several image/video processing toolsets such as OpenCV and the Computer Vision toolbox of MATLAB. Our implementation is based on the Lucas-Kanade method which is friendly for FPGAs [32]. The output is a 2D vector field of the same size, where each vector shows the movement of the pixel in the input image frames. Currently, pixels of input images are represented with 8-bit integers, while the output and all intermediate results are represented with 32-bit floating-point numbers. We use the MPI Sintel dataset [4] for testing this benchmark. The resolution of the image frames in this dataset is 436x1024.

Optical flow must satisfy a real-time throughput constraint of 30 frames per second. In addition, the limited amount of on-chip storage prevents us from buffering the image frame on chip. Figure 6 shows the image processing pipeline with eight stages. The main compute kernel for stages Gradient, Weight, and Tensor is 1D convolution; the Outer product stage performs outer product of three-dimensional vectors. Output is generated in the Compute flow stage. Currently, we are using floating-point arithmetic in these kernels. Data packing optimization introduced in Section 4.3 is applied

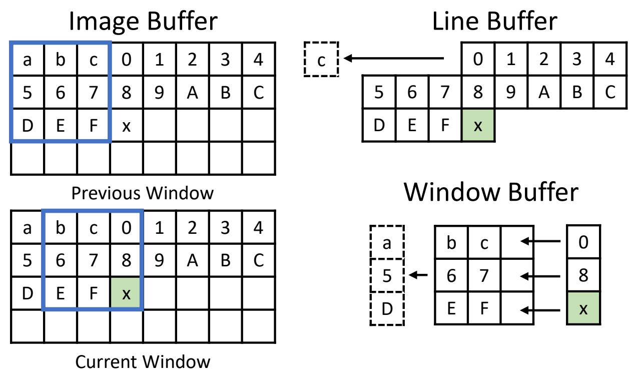

Figure 7: Example of a 2-row line buffer and a 3x3 window buffer — Pixels a, b, c and 0-F are already visited, while x is a new pixel. The line buffer stores pixels in the two most recently visited rows, and reads in one pixel from the image buffer every cycle. The 3x3 window buffer stores recently visited pixels in the 3x3 sliding window. When the sliding window shifts to the right, the left-most pixels in the window buffer are shifted out, while two pixels stored in the line buffer (0 and 8) and the new pixel x are shifted in. The new pixel x is also stored into the line buffer and pixel c is removed from the line buffer.

to avoid contention on the off-chip memory. Each packet contains one pixel from each image frame, and the Unpack stage distributes the pixels to on-chip FIFOs. Similar to 3D rendering, we use dataflow optimization to construct channels between stages of the image processing pipeline. The major difference between the two benchmarks is that all pipeline stages in optical flow produce and consume pixels in a strict sequential order. In addition, the pipeline stages have perfectly balanced rates. Therefore, the channels between pipeline stages can be implemented as fixed-depth FIFOs, as shown in Figure 6. The whole accelerator is a very deep, fine-grained pipeline with different stages perfectly overlapped.

Memory customization is also necessary for optical flow to achieve high throughput. Here we introduce the common specialized memory structures for image processing applications: line buffer and window buffer. Figure 7 gives a pictorial illustration of a 2-row line buffer and a 3x3 window buffer. The line buffer reads in one pixel per cycle and stores pixels in recently visited rows. The window buffer is completely partitioned into registers for parallel data access, and it consistently reads from the line buffer. These specialized memory structures exploit the data reuse in stencil applications with sliding processing windows, and minimize memory accesses to the next-level memory hierarchy. The convolution kernels in optical flow are good candidates for this memory customization. Figure 8 shows how we construct and maintain a line buffer and a window buffer in the `gradient_xy` kernel. Proper conditions need to be applied to avoid out-of-bound array accesses.

With the optimizations described above, we classify optical flow as a memory-bound application because the off-chip memory bandwidth directly determines the throughput of the streaming dataflow pipeline. However, this is because our current implementation does not exploit data reuse between input frames. We plan to further optimize this design to achieve a higher throughput.

```
void gradient_xy(pixel_t frame[MAX_HEIGHT][MAX_WIDTH],
    pixel_t gradient_x[MAX_HEIGHT][MAX_WIDTH],
    pixel_t gradient_y[MAX_HEIGHT][MAX_WIDTH])
{
  // specialized line buffer and window buffer
  hls::LineBuffer<5,MAX_WIDTH,pixel_t> buf;
  hls::Window<5,5,pixel_t> window;
  GRAD_XY_OUTER: for (int r = 0; r < MAX_HEIGHT + 2; r ++) {
    GRAD_XY_INNER: for (int c = 0; c < MAX_WIDTH + 2; c ++) {
      #pragma HLS pipeline II=1
      // fill the line buffer
      if (r < MAX_HEIGHT && c < MAX_WIDTH) {
        // shift up pixels in column c
        buf.shift_pixels_up(c);
        // insert new pixel into column c of the last row
        buf.insert_bottom_row(frame[r][c], c);
      } else if (c < MAX_WIDTH) {
        buf.shift_pixels_up(c);
        // zero padding
        buf.insert_bottom_row(0,c);
      }
      // fill the window buffer
      if (r < MAX_HEIGHT && c < MAX_WIDTH) {
        // shift pixels to the left
        window.shift_pixels_left();
        for (int i = 0; i < 4; i ++)
          // read from the line buffer
          // and insert to the right-most column
          window.insert_pixel(buf.getval(i, c), i, 4);
      } else {
        window.shift_pixels_left();
        for (int i = 0; i < 4; i ++)
          // zero padding
          window.insert_pixel(0, i, 4);
      }
      // compute
      // ......
}}}
```

Figure 8: Gradient kernel optimized with line buffer and window buffer — `hls::LineBuffer` and `hls::Window` classes provide parameterized implementations of line buffers and window buffers.

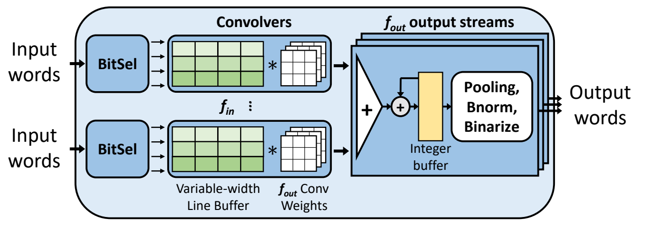

Figure 9: Hardware structure of the BNN accelerator (figure adapted from [39]).

4.5 Binarized Neural Network

Accelerating convolutional neural networks (CNNs) has become an important research topic for the FPGA community. Academic and industry researchers have implemented different CNN models on a variety of FPGA platforms [3, 18, 35, 36]. Recently, binarized neural networks (BNNs) were shown to be a natural fit for FPGA hardware [6, 27, 33, 39]. BNNs constrain weights and intermediate activations to +1 or -1; this converts most of its multiplies to binary XORs and takes full advantage of the FPGA logic fabric. We adopt an open-source implementation of BNN by Zhao et al. [39] as a representative neural network application in Rosetta.

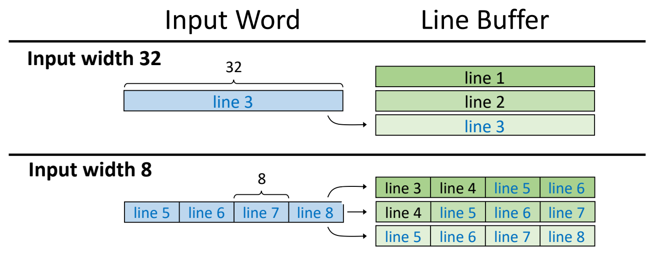

Figure 10: Example usage of variable-width line buffer for 8-wide and 32-wide feature maps (figure adapted from [39]).

Zhao et al. implement the BNN model described in [8], which operates on the CIFAR-10 dataset [12]. It contains six convolutional layers, three pooling layers, and three fully-connected layers. Figure 9 shows the hardware diagram of the BNN accelerator, which uses a configurable number of convolvers to exploit data-level parallelism in a scalable manner. The authors target a small FPGA device with limited on-chip storage. As a result, the BNN weights cannot fit on-chip and the accelerator must be invoked multiple times to classify an image; each time new weights are loaded from off-chip memory.

There are two major kernels in BNN: binarized convolution and binarized dot product. Both kernels are intensive of bitwise logic operations. Binarized convolution comprise the majority of operations in classifying an image, and is heavily parallelized as a result. In contrast, the binarized fully-connected layers, which use the dot product kernel, are limited by off-chip memory-bandwidth. We categorize BNN as compute-bound since latency improvement mostly comes from accelerating compute in the convolutional layers.

Since 2D convolutional layers have a sliding window access pattern, line buffers are used to exploit data locality. In particular, a variable-width line buffer (VWLB) is designed to keep the hardware convolvers fully utilized despite the varying sizes of the feature maps. Figure 10 shows how the VWLB works for different input widths. For input feature map with a width of 32, the VWLB operates identically to a conventional line buffer. For a smaller feature map with a width of 8, each row in the VWLB stores multiple rows of the input. The rows are carefully arranged in the VWLB so that the convolutional filter can slide through and produce correct results.

Design parameters. The BNN benchmark allows users to tune the number of convolvers in the accelerator. Other parameters such as the size of buffers are automatically scaled.

4.6 Face Detection

The face detection application is adopted from [25]. It uses the Viola-Jones algorithm [28] to detect human faces in a given image. More specifically, the accelerator takes an 320x240 greyscale image as input, which is scaled to construct an image pyramid; afterwards, an integral image is constructed from each image in the image pyramid, and a set of cascaded classifiers are applied to a fixed-size window which scans through the integral image; eventually, the positions and sizes of the human faces are returned.

As mentioned in [25], the throughput target for face detection is 30 frames per second. In addition, the application is subject to hardware constraints including limited on-chip storage and routing resources. The two major compute kernels in face detection are image scaling and cascaded classifiers. Image scaling is a common kernel in feature extraction applications such as SIFT [17], as well as the pooling layers of CNNs. The cascaded classifiers are the dominant workload for the face detection application. The authors of [25] parallelize the first three classifier stages and pipeline the rest of the stages to exploit data-level parallelism. This kernel also exposes an irregular memory access pattern — each classifier accesses either eight or twelve pixels, and the classifiers have different access patterns. This feature itself makes the kernel interesting for HLS memory optimization techniques. Customized memory partitioning is applied to improve kernel frequency and reduce routing effort [41].

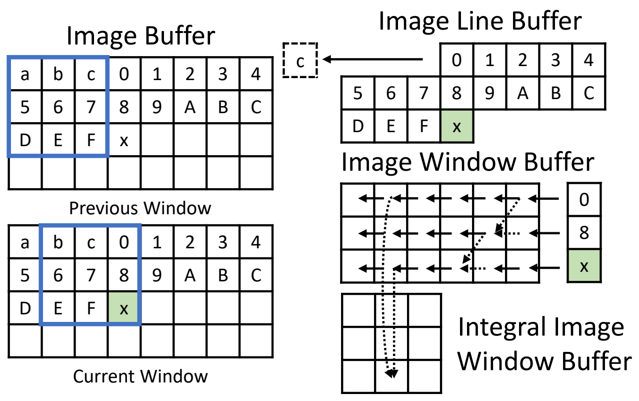

Figure 11: Specialized line buffer and window buffer for face detection [25] — Here we show a 3x3 example, but the actual implementation uses 25x25 windows. Solid arrows refer to normal register shifting, while dashed arrows refer to addition. The image window buffer accumulates the incoming pixels and construct the integral image on the fly. The integral image window buffer accesses the image window buffer for new data.

Table 2: Device capacity of the two FPGA platforms and the resource utilization of the platform logic (shell) on AWS F1 — The last row reports the average resource utilization of the shell, with the standard deviation in parentheses.

	# LUTs	# FFs	# BRAMs	# DSPs
AWS F1 Total	1181768	2363536	2160	6840
ZC706 Total	218600	437200	545	900
AWS F1 Shell	293209 (±3693)	381853 (±5138)	545 (±0)	12 (±0)

The cascaded classifiers operate on a sliding window of the integral image. As a result, face detection can also benefit from the line buffer and window buffer optimization introduced in Section 4.4. However, constructing the whole integral image before applying the classifiers would require a significant amount of on-chip storage and incur performance loss. Therefore, the authors of [25] modified the window buffer to construct the integral image efficiently. The operation of this buffer is depicted in Figure 11, where the modified image window buffer accumulates pixels on the diagonal to compute the pixel values in the integral image.

5 EXPERIMENTAL RESULTS

We have synthesized the Rosetta benchmarks targeting an embedded FPGA as well as a cloud FPGA instance. We use Xilinx ZC706 for the embedded platform, which contains a Kintex-7 FPGA with a

Table 3: Rosetta results on Xilinx ZC706 Platform — The Runtime column shows overall execution time. Resource numbers show the *total* resource usage of the designs, including both kernel function and shell logic. Bitstreams are generated by Xilinx SDSoC 2017.1.

Benchmark	# LUTs	# FFs	# BRAMs	# DSPs	Runtime (ms)	Throughput
3D Rendering	8893	12471	48	11	4.7	213 frames/s
Digit Recognition[1]	41238	26468	338	1	10.6	189k digits/s
Spam Filtering[2]	12678	22134	49	160	78.9	285k samples/s
Optical Flow	42878	61078	54	454	24.3	41.2 frames/s
Binarized Neural Network[3]	46899	46760	102	4	4995.2	200 images/s
Face Detection	62688	83804	121	79	33.0	30.3 frames/s

1. K = 3, `PAR_FACTOR` = 40. 2. Five epochs, `PAR_FACTOR` = 32, `VDWIDTH` = 512.
3. Eight convolvers, 1000 test images.

Table 4: Rosetta results on AWS F1 Platform — Kernel: execution time on the FPGA; Comm.: time of data transfer between host and global memory; Runtime: overall execution time. Performance-Cost Ratio is calculated based on the hourly rate (in US Dollar/$) of the AWS f1.2xlarge instance [1]. Resource numbers are for kernel functions only. Bitstreams are generated by Xilinx SDAccel 2017.1.

Benchmark	# LUTs	# FFs	# BRAMs	# DSPs	Kernel (ms)	Comm. (ms)	Runtime (ms)	Throughput	Performance-Cost Ratio
3D Rendering	6763	7916	36	11	3.6	0.19	4.4	227 frames/s	496k frames/$
Digit Recognition[1]	39971	33853	207	0	9.9	0.55	11.1	180k digits/s	393M digits/$
Spam Filtering[2]	7207	17434	90	224	25.1	4.8	30.9	728k samples/s	1.6G samples/$
Optical Flow	38094	63438	55	484	2.6	4.8	8.4	119 frames/s	260k frames/$
Face Detection	48217	54206	92	72	20.2	0.47	21.5	46.5 frames/s	101k frames/$

1. K = 3, `PAR_FACTOR` = 40. 2. Five epochs, `PAR_FACTOR` = 32, `VDWIDTH` = 512.

target clock frequency of 140MHz. For the cloud FPGA platform, we choose the AWS f1.2xlarge instance (F1), which is equipped with a Xilinx VU9P FPGA. The target clock frequency for our experiments on F1 is 250MHz. These two platforms have different memory systems — on ZC706, the FPGA shares the same DRAM with the embedded CPU, while on F1 the FPGA has its own on-board DRAM and communicates with the CPU through PCIe. In the rest of this section, we use the term *global memory* to refer to the DRAM on the FPGA side, and use *host memory* for the DRAM on the CPU side. The BNN benchmark is originally designed for embedded FPGA platforms and requires nontrivial effort to be retargeted to AWS F1. We leave this for future work, and will only present BNN results on ZC706 in this paper. For other benchmarks, the HLS code for the two platforms share the same optimization techniques, with some platform-dependent variances such as datatype and interface. Xilinx SDSoC 2017.1 is used to generate bitstream for ZC706, and SDAccel 2017.1 is used for F1.

We run the F1 applications remotely through the FPGA developer AMI flow provided by AWS, whereas the experiments on ZC706 are performed locally. Table 2 shows the available resource counts of the two platforms. On the F1 platform, the AWS platform logic (or shell) consumes a considerable amount of resources to provide peripheral connections for PCIe data transfer, DRAM access, and interrupts [2]. In the third row of Table 2, we report the statistics of the resource usage by this shell across different applications. For ZC706, Xilinx SDSoC also automatically generates shell logic for communications among accelerators, processors, and DRAM. However, the size of these shells greatly vary across designs, and are typically small compared to that of the core logic. Hence we choose to simply report the total resource utilization for ZC706 results.

Table 5: 3D rendering without dataflow on AWS F1.

# LUTs	# FFs	# BRAMs	#DSPs	Kernel (ms)
6323	7737	36	11	5.3

Tables 3 and 4 show our experimental results on the two platforms. All resource usage numbers are extracted from Vivado reports after place and route. Resource numbers in Table 3 show the *total* resource utilization of the designs on ZC706, while Table 4 reports resource usage on F1 without the shell logic. The total runtime of the applications, including hardware kernel time, communication time, and the overhead of necessary software function calls, are measured on both platforms. On AWS F1, we further break down the kernel and communication time with the help of the SDAccel profiler. Rosetta benchmarks generally have better performance on AWS F1 because of its higher frequency and off-chip memory bandwidth, except for digit recognition. For some applications, however, this performance gap is narrow due to the communication latency and additional overhead incurred by OpenCL runtime.

Since cost efficiency is an important aspect of platform selection and accelerator design, we further provide the performance-cost ratio as a metric for F1 applications based on the hourly rate of the f1.2xlarge instance (currently at $1.65 per hour).

In the remainder of this section we summarize the results for the four new benchmarks. As for BNN and face detection, interested readers can refer to [39] and [25], respectively, for more results and detailed performance analysis.

3D Rendering. For our test dataset, the total execution time of 3D rendering is 4.7 ms and 4.4 ms on the two platforms, respectively. Converting to throughput, our design achieves 213 frames per second

Table 6: Digit recognition accuracy vs. K value.

K	2	3	4	5
Accuracy (%)	92.9	93.9	94.3	94.3

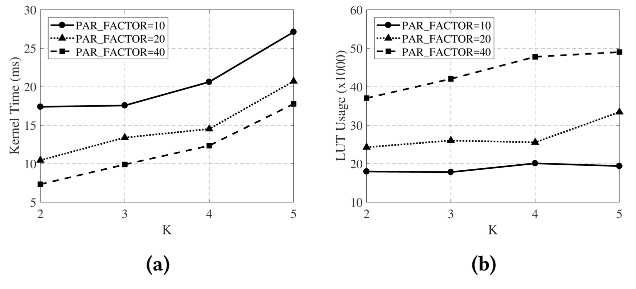

Figure 12: Digit recognition design space, results are for AWS F1 platform — (a) Kernel time vs. K value. Difference in kernel time is caused by variance in latency and kernel frequency. **(b)** LUT usage vs. K value.

on ZC706 and 227 frames per second on F1. While the throughput calculated with our test input is much higher than the target, both kernel time and communication time increase with more triangles in the input. Communication latency is not significant on F1, but the software API calls in OpenCL runtime incur a 0.6 ms overhead, which is not negligible for this specific application. These API calls initiate data transfer, enqueue the kernel function, and set proper kernel arguments.

Table 5 shows the resource utilization and kernel time of a baseline design where dataflow optimization is not applied. Comparing with the first row of Table 4, enabling dataflow optimization improves the kernel time by around 30% without significant resource overhead. This result demonstrates the efficacy of dataflow optimization in image processing pipelines.

Digit Recognition. In contrast to other benchmarks, the performance of digit recognition is currently slightly worse on F1 than ZC706. The overall throughput is 189k digits per second on ZC706 and 180k digits per second on F1. Although F1 has a shorter kernel time of 9.9ms, the latency of communication and other overhead in OpenCL runtime seem to have offset this advantage. According to our analysis, this is likely due to a missing feature in the specific version of the tool we are using, where `async_group_copy` is not pipelined to the full extent. Hence we expect to achieve a higher performance on F1 in the near future once this issue is resolved.

As mentioned in Section 4.2, digit recognition has a complex design space. Table 6 shows the classification accuracy of different K values. Figure 12 shows kernel time and resource utilization of different design points. We only show kernel time in Figure 12a because host-global memory communication time is not affected by kernel implementation. In Figure 12b, only the most critical resource LUT is shown. As we can see from Table 6 and Figure 12, the two design parameters expose interesting design trade-offs. Increasing the K value improves classification accuracy at a cost of significant increase in kernel time, which is caused by the frequency drop and the worsened latency of the KNN voting kernel. Additionally, the benefit of increasing `PAR_FACTOR` diminishes when `PAR_FACTOR` is

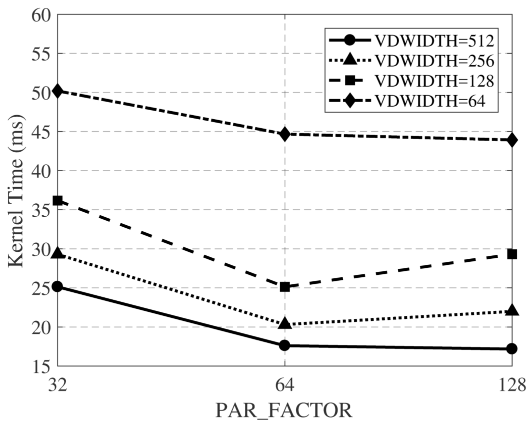

Figure 13: Spam filtering design space, results are for AWS F1 platform — Off-chip memory bandwidth is controlled by `VDWIDTH`. This parameter strictly limits the performance of the hardware kernel, showing that spam filtering is a memory-bound application.

already large. When the Hamming distance kernel is highly parallelized, the KNN voting kernel, which is highly sequential, becomes the performance bottleneck. The performance can be further improved by optimizing the KNN voting kernel, and finding an optimal combination of the K value and `PAR_FACTOR`.

Spam Filtering. The performance of spam filtering significantly differs on two platforms. The kernel time on F1 is 3.1x shorter than ZC706, and the total execution time on F1 is 2.6x shorter, despite the additional 4.8 ms latency for host-global memory communication. In addition to the frequency improvement, this performance gap is mainly caused by the difference in off-chip memory bandwidth. Since we apply dataflow optimization to overlap communication and compute, the overall latency of the design is determined by the maximum of compute and communication latency. Because the compute kernels are highly parallel, the low communication bandwidth on ZC706 results in a much longer latency of the dataflow pipeline.

Figure 13 shows the kernel time on AWS F1 with different combinations of `PAR_FACTOR` and `VDWIDTH`. Here `PAR_FACTOR` specifies the degree of parallelism in vector kernels, and `VDWIDTH` controls the off-chip communication bandwidth. With the same off-chip bandwidth, increasing `PAR_FACTOR` beyond 64 does not result in much performance gain, since the communication latency already dominates the compute latency. When off-chip bandwidth is reduced, communication latency further increases, and kernel time degrades for all `PAR_FACTOR` values we tested. The best-achievable performance improves with a higher off-chip memory bandwidth. These results confirm that spam filtering is a memory-bound application.

Optical Flow. The total execution time of optical flow is 8.4 ms on F1 and 24.3 ms on ZC706. Both implementations satisfy the throughput constraint. On the AWS F1 platform, host-global memory communication time takes up approximately 60% of the total execution time due to the large input/output data size. If we only consider kernel time, it is 9.3x shorter on F1 than on ZC706. Similar with spam filtering, this behavior is also caused by the difference in off-chip memory bandwidth. The optical flow accelerator is reading from and writing to the off-chip memory at the same time due to the streaming dataflow optimization. The F1 platform has multiple off-chip

DDR banks to handle concurrent read and write requests. On ZC706, however, these concurrent requests would cause contention on the off-chip memory, and the accelerator is often stalled due to the lack of input data.

6 CONCLUSIONS AND FUTURE WORK

We have presented Rosetta, an open-source, realistic benchmark suite for high-level synthesis targeting modern FPGA platforms. Rosetta is designed to be a collection of real applications which are optimized for performance and resource constraints. All Rosetta applications are ready to be executed on the supported embedded and cloud platforms. We believe that Rosetta can serve as a useful benchmark suite for HLS algorithms and tools, as well as a set of design tutorials for application developers interested in FPGA-based accelerated computing.

Rosetta will be continuously improved in the future. We will extend Rosetta to include more realistic applications from emerging domains. For the existing benchmarks, we plan to provide both C++ and OpenCL implementations for every benchmark to embrace different programming models commonly supported by HLS tools. The benchmarks will also be further optimized for achieving higher performance and resource efficiency.

ACKNOWLEDGEMENTS

This research was supported in part by a DARPA Young Faculty Award, NSF Awards #1337240, #1453378, and a research gift from Xilinx, Inc. We thank Dr. Sumit Roy from Xilinx for providing helpful feedback on the Rosetta designs. We also thank Ackerley Tng, Edgar Munoz, Wendian Jiang, Lin Wang, Yun Qing, Nithya Subramanian, Nikita Patil, Surabhi Singh, Judy Stephen, and Ian Thompson for their contributions to the baseline designs of digit recognition, 3D rendering, spam filtering, and optical flow.

REFERENCES

[1] Amazon Web Services. AWS FPGA Developer AMI. *https://aws. amazon. com/-marketplace/pp/B06VVYBLZZ*, Dec 2017.

[2] Amazon Web Services. AWS Shell Interface Specification. *https://github. com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md*, Dec 2017.

[3] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu. An OpenCL Deep Learning Accelerator on Arria 10. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[4] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A Naturalistic Open Source Movie for Optical Flow Evaluation. *European Conference on Computer Vision (ECCV)*, Oct 2012.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2009.

[6] P. Colangelo, R. Huang, E. Luebbers, M. Margala, and K. Nealis. Fine-Grained Acceleration of Binary Neural Networks Using Intel Xeon Processor with Integrated FPGA. *Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM)*, Apr/May 2017.

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[8] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to + 1 or -1. *arXiv preprint arXiv:1602.02830*, Mar 2016.

[9] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[10] Q. Gautier, A. Althoff, P. Meng, and R. Kastner. Spector: An OpenCL FPGA Benchmark Suite. *Int'l Conf. on Field Programmable Technology (FPT)*, Dec 2016.

[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-Based High-Level Synthesis. *Journal of Information Processing, Vol. 17*, pages 242–254, Oct 2008.

[12] A. Krizhevsky and G. Hinton. Learning Multiple Layers of Features from Tiny Images. *Technical report, University of Toronto*, Apr 2009.

[13] Y. LeCun. The MNIST Database of Handwritten Digits. *http://yann. lecun. com/exdb/mnist/*, Dec 2017.

[14] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen. High-Level Synthesis: Productivity, Performance, and Software Constraints. *Journal of Electrical and Computer Engineering*, 2012:1:1–1:1, Jan 2012.

[15] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017.

[16] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen. High Level Synthesis of Complex Applications: An H. 264 Video Decoder. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2016.

[17] D. G. Lowe. Object Recognition from Local Scale-Invariant Features. *Int'l Conf. on Computer Vision (ICCV)*, Oct 1999.

[18] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[19] K. P. Murphy. *Machine Learning: A Probabilistic Perspective.* MIT Press, 2012.

[20] J. Pineda. A Parallel Algorithm for Polygon Rasterization. *ACM SIGGRAPH Computer Graphics*, 22(4):17–20, 1988.

[21] L.-N. Pouchet. Polybench: The Polyhedral Benchmark Suite. *http://www. cs. ucla. edu/pouchet/software/polybench*, Dec 2017.

[22] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-Based Data Reuse Optimization for Configurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2013.

[23] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2014.

[24] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2014.

[25] N. K. Srivastava, S. Dai, R. Manohar, and Z. Zhang. Accelerating Face Detection on Programmable SoC Using C-Based Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[26] The Apache Software Foundation. Public Corpus. *http://spamassassin. apache. org/old/publiccorpus/*, Apr 2017.

[27] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[28] P. Viola, M. J. Jones, and D. Snow. Detecting Pedestrians using Patterns of Motion and Appearance. *International Journal of Computer Vision*, 63(2):153–161, Jul 2005.

[29] S. Wang, Y. Liang, and W. Zhang. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. *Design Automation Conf. (DAC)*, Jun 2017.

[30] Y. Wang, P. Li, and J. Cong. Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2014.

[31] Z. Wang, B. He, W. Zhang, and S. Jiang. A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs. *Int'l Symp. on High Performance Computer Architecture (HPCA)*, Mar 2016.

[32] Z. Wei, L. Dah-Jye, and B. E. Nelson. FPGA-Based Real-Time Optical Flow Algorithm Design and Implementation. *Journal of Multimedia*, 2:38–45, Sep 2007.

[33] H. Yonekawa and H. Nakahara. On-Chip Memory Based Binarized Convolutional Deep Neural Network Applying Batch Normalization Free Technique on an FPGA. *Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, May 2017.

[34] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2015.

[35] C. Zhang and V. K. Prasanna. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[36] J. Zhang and J. Li. Improving the Performance of OpenCL-Based FPGA Accelerator for Convolutional Neural Network. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[37] Z. Zhang and B. Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2013.

[38] J. Zhao, L. Feng, S. Sharad, W. Zhang, Y. Liang, and B. He. COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2017.

[39] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. B. Srivastava, R. Gupta, and Z. Zhang. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[40] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. Lin-Analyzer: A High-Level Performance Analysis Tool for FPGA-Based Accelerators. *Design Automation Conf. (DAC)*, Jun 2016.

[41] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang. A New Approach to Automatic Memory Banking using Trace-Based Address Mining. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[42] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong. Improving Polyhedral Code Generation for High-Level Synthesis. *Proc. of the 8th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep/Oct 2013.