Spector: An OpenCL FPGA Benchmark Suite

Quentin Gautier, Alric Althoff, Pingfan Meng and Ryan Kastner
University of California, San Diego

***Abstract*—High-level synthesis tools allow programmers to use OpenCL to create FPGA designs. Unfortunately, these tools have a complex compilation process that can take several hours to synthesize a single design. This creates a significant barrier for design optimization since even experts typically need to test many designs due to the non-obvious interactions between the different optimizations. Thus, understanding the design space, and guiding the optimization process is a crucial requirement for enabling the widespread adoption of these high-level synthesis tools. However this requires a significant amount of design space data that is currently unavailable or difficult to generate. To solve this problem, we present an OpenCL FPGA benchmark suite. We outfitted each benchmark with a range of optimization parameters (or *knobs*), compiled over 8300 unique designs using the Altera OpenCL SDK, executed them on a Terasic DE5 board, and recorded their corresponding performance and utilization characteristics. We describe the resulting design spaces, and perform a statistical analysis of the optimization configurations which provides valuable architecture insights to FPGA developers. We make the benchmarks and results completely open-source to give opportunities for the community to perform additional analyses and provide a repository of well-documented designs for follow-on research.**

I. Introduction

FPGA design was traditionally relegated to only experienced hardware designers, and required specifying the application using low-level hardware design languages. This provides opportunities to create highly specialized custom architectures; yet it is time consuming as every minute detail must be specified on a cycle-by-cycle basis. Recently, FPGA vendors have released high-level synthesis tools centered around the OpenCL programming model. The tools directly synthesize OpenCL *kernels* to programmable logic creating a custom hardware accelerator. They raise the level of abstraction of the programming model and increase the designer's productivity. Furthermore, the tools manage the transfer of data between the FPGA and the CPU host. This opens the door for more programmers to easily utilize FPGAs.

OpenCL is a open standard that provides a framework for programming heterogenous systems. The language extends C with features that specify different levels of parallelism and define a memory hierarchy. There exists OpenCL implementations for a variety of multicore CPUs, DSPs, and GPUs. More recently, commercial tools like Xilinx SDAccel [1] and the Altera OpenCL SDK [2] add FPGAs into the mix of supported OpenCL devices. This greatly simplifies the integration of FPGAs into heterogeneous systems, and provides a FPGA design entry point for a larger audience of programmers.

The OpenCL FPGA design process starts with implementing the application using OpenCL semantics. The designer then typically employs some combination of well-known optimizations (e.g. SIMD vectors, loop unrolling, etc.) and settles on a small set of designs that are considered optimal according to some metric of performance (resource utilization, power, etc.). Most designers will need multiple attempts with several optimization options to understand the design space. Unfortunately, a major drawback of these OpenCL FPGA tools is that the compilation time is long; it can take hours or even days. This severely limits the ability to perform a large scale design space exploration, and requires techniques to efficiently guide the designer to a good solution.

In many applications, it is difficult to predict the performance and area results, especially when optimization parameters interact with each other in unforeseen manners. As an example, increasing the number of compute units duplicates the OpenCL kernel, which should improve performance at the expense of FPGA resources. However, this is not always true as memory contention may limit the application's performance. Finding when this occurs requires a better understanding of the memory access patterns and how other optimizations alter it. Many other optimizations are also intertwined in non-intuitive ways as we describe throughout the paper.

We propose an OpenCL FPGA benchmark suite. Each benchmark is tunable by changing a set of *knobs* that modify the resulting FPGA design. We compiled over 8000 designs across 9 unique benchmarks using the Altera OpenCL SDK. All of our results are openly available and easily accessible [3]. This provides large set of designs to enable research on system-level synthesis for FPGAs. For example, researchers can use our results to evaluate their methods for improving the process of design space exploration. We provide our own analysis of the data as an example use-case in Section V; there is substantial follow-on work that can be done, and we encourage other researchers to use and extend our results.

The major contributions are:

- Designing and releasing an OpenCL FPGA benchmark suite
- Creating an optimization space for each benchmark and describing the parameters that define it.
- Performing a comprehensive set of end-to-end synthesis experiments, the result of which is over twenty thousands hours of compilation time.
- Providing a statistical analysis on the results to give insights on OpenCL FPGA design space exploration.

The remainder of the paper is organized as follows. In Section II, we motivate the need for this research, and discuss related work. In Section III we detail our benchmark design process and talk about how we obtained the results. In Section IV we describe the benchmarks, detail the tunable

978-1-5090-5602-6/16/$31.00

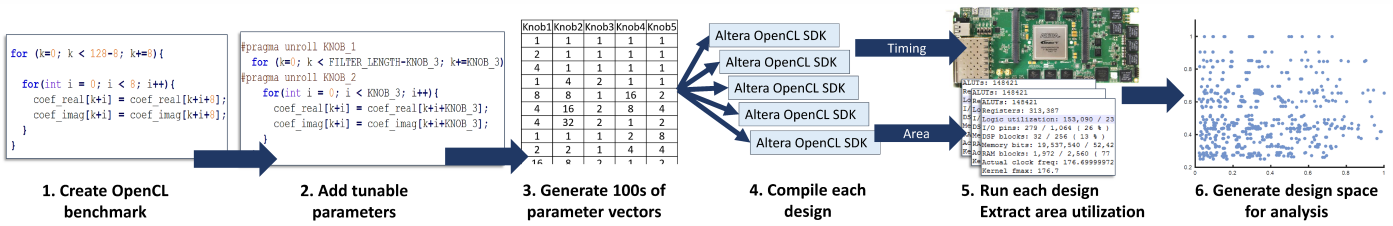

Fig. 1. Our workflow to generate each benchmark and then generate the design space results as presented in Section III.

knobs and their effect on the architectures, and present their design spaces. We give a statistical analysis of some of the results in Section V and conclude in Section VI.

II. Motivation

There are substantial number of application case studies for parallel computing, heterogeneous computing, and hardware design. One can start with reference designs from hardware vendors such as Intel, NVIDIA, Altera, and Xilinx. Unfortunately, these are often scattered across different websites, use different target compute platforms (CPU, GPU, FPGA), and they lack a common lingua franca in terms of optimization parameters. This makes them difficult to provide a fair comparison across the different applications. This is the general motivation for benchmark suites – they provide highly available, well documented, representative set of applications that can assess the effectiveness of different design strategies and optimizations.

Several open-source benchmarks for parallel applications currently exist. Many of these, e.g., the HPEC challenge benchmark suite [4] or Rodinia [5], focus on GPUs and multicore CPUs. FPGAs have a different compute model. Thus, while some of the applications in these benchmarks suites are applicable to studying the OpenCL to FPGA design flow, they require modifications to be useful. Several of our benchmarks are found in these existing benchmark suites. There are also a number of FPGA specific benchmarks suites. These generally target different parts of the design flow. For example, the applications in ERCBench [6] are written in Verilog and useful for studying RTL optimizations or as a comparison point for hardware/software partitioning. Titan [7] uses a customized workflow to create benchmarks to study FPGA architecture and CAD tools. The OpenCL dwarfs [8], [9] contain several OpenCL programs that have been optimized for FPGA. Unfortunately, they usually have a fixed architecture with little to no optimization parameters.

We seek to extend these benchmark suites by leveraging the existing OpenCL benchmarks and reference programs, and outfitting them with multiple optimization parameters. Each of these designs can be compiled with a commercial program or open-source tool to generate thousands of unique configurations. We make open-source our results that we obtained from the Altera software, and encourage the community to compile our benchmarks with different tools. One of our motivating factors in creating this benchmark suite was the lack of a common set of designs and optimization parameters for comparing different design space exploration (DSE) techniques. Machine learning techniques for DSE in particular can benefit from a large set of designs, e.g., [10] and [11] use machine learning approaches to explore design spaces and predict the set of Pareto designs without having to compile the entire space. These techniques could directly leverage our results to verify and perhaps improve their models. And in general, we believe that open repository of OpenCL FPGA designs will benefit this and other areas of research.

TABLE I. Number of successfully compiled designs.

BFS	507	Histogram	894	Normal estimation	696
DCT	211	Matrix Multiply	1180	Sobel filter	1381
FIR filter	1173	Merge sort	1532	SPMV	740

III. Methodology

We designed nine benchmarks that cover a wide range of applications. We selected benchmarks that are recurrent in FPGA accelerated applications (FIR filter, matrix multiply, etc.), but we also included code with more specific purpose to cover potential real-world programs (like histogram calculation and 3D normal estimation). These benchmarks come from various places: some were written directly without example source code, others come from GPU examples, and some come from FPGA optimized examples. In all cases, we started from programs that contained little to no optimization parameters, thus requiring us to define the optimization space.

For each benchmark, we proceeded as illustrated in Figure 1. First we created or obtained code that was partially or fully optimized for FPGA. It is important to note that we were not trying to reach a single "most optimal" design, but instead defining an optimization space that covers a wide range of optimizations. We studied which types of optimization would be relevant for each benchmark. Then we added several optimization *knobs*, which are values that we can tune at compile-time. These knobs can enable or disable code, or affect an optimization parameter (e.g., unrolling factor). We compiled several sample designs to ensure that the knobs we had chosen would have *some* impact on the timing and area. Each benchmark has a set of scripts to generate hundreds of unique designs, with all the possible combinations of knob values. In most cases we restricted the values of the knobs to a subset of the options (e.g., powers of two). We also removed values that were likely to use more resources than available, and filtered out further using pre-*place-and-route* estimations. All the benchmarks were written using standard OpenCL code with a C++ host program that can run and measure the time of execution. The OpenCL code works with the Altera SDK for FPGA, and can also be executed on GPU and CPU. Although we have not tested the programs with other commercial or open-source OpenCL-to-FPGA pipelines, we expect that little to no modifications are required to ensure compatibility.

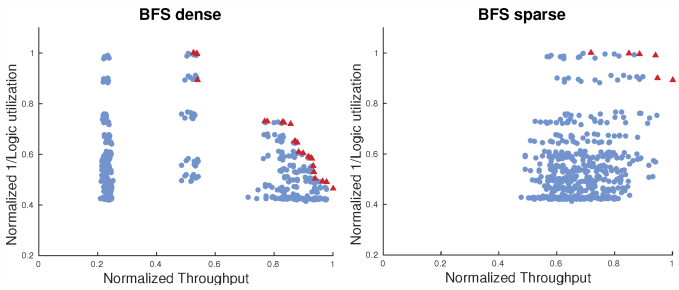

Fig. 2. BFS design spaces for dense and sparse inputs. We plot the inverse logic utilization against the throughput such that higher values are better. The Pareto designs are shown in red triangles.

Each design was then individually compiled using the Altera OpenCL SDK v14.1, with a compile time typically requiring 1 to 4 hours on a modern server, and occasionally taking more than 5 hours per design. In total we successfully compiled more than 8300 designs (see Table I), plus many more that went through almost the entire compilation process but failed due to device resource limits. We executed the successful designs on a Terasic DE5 board with a Stratix V FPGA to measure the running time for a fixed input. Some applications can behave differently given a different set of input data however, and the optimizations to use in these cases might vary. This is the case for algorithms like graph traversal or sparse matrix multiplication, where the sparsity of the input can have a significant impact on the design space. In both of these benchmarks we ran the program with two sets of inputs. We ran BFS with both a densely connected graph and one with sparse edges. Sparse matrix-vector multiplication was run with one matrix containing 0.5% of non-zero values and one only 50% sparse. We extracted the running time and area data (such as logic, block RAM, DSPs, etc.) for each run. The set of design spaces that we present in Figures 2 to 8 shows the logic utilization against the running time, as logic is usually the most important resource and often the limiting factor. These results are generated from the scripts `plot_design_space.m` and `plot_all_DS.m` available in our repository.

IV. Benchmarks Description

Here we describe the benchmarks and the knobs that we have chosen, so that the reader can interpret the design space results based on the design choices. First we explain some of the most common optimization types in OpenCL designs. Then we explain each benchmark in more details, followed by an overview of the shape of the design space.

Work-items: These are parallel threads with a common context on GPU. On FPGA, they can be interpreted as multiple iterations of an outer loop that is pipelined by the compiler. Using only one can give more flexibility to the programmer to control unrolling and pipelining, while using multiple can enable optimizations such as SIMD. **Work-groups**: This defines how many groups of work-items to use, each group using a different context (no shared memory). This is useful to enable the compute units optimization. **Compute units**: How many duplicates of the kernel are created on the FPGA chip. Compute units can run work-groups in parallel, however they all access the external memory and thus might be limited by the bandwidth. **SIMD**: Work-items can be processed simultaneously by increasing local area usage. It is only possible when there is no branching. **Unrolling**: By explicitly unrolling a loop, we can process multiple elements simultaneously by using more area, usually storing data in more local registers.

A. Breadth-First Search (BFS)

This code is based on the BFS FPGA benchmark from the OpenDwarfs project [8], [9], and originally based on the BFS benchmark from the Rodinia Benchmark Suite [5]. It is an iterative algorithm that simply traverses a graph starting at a specified node by performing a breadth-first traversal, and returns a depth value for each node. The algorithm iterates over two OpenCL kernels until all the reachable nodes have been visited. Each kernel launches work-items for each node in the graph and uses binary masks to enable computation. There are 6 knobs with varying values in these kernels.

- **Unroll factor** in kernel 1: Unrolls a loop to process multiple edges simultaneously. We enable additional code for edge cases only if the unroll factor is greater than 1.
- **Compute units** in kernel 1 and 2, **SIMD** in kernel 2.
- **Enable branch** in kernel 1: Describes how to check if a node was visited and how to update graph mask values. Either enables the code from OpenDwarfs with bitwise operators to avoid branching, or enables the code from Rodinia with regular *if* statement.
- **Mask type**: Number of bits used to encode the values of graph masks.

Design space: (Figure 2) The design space for the dense input is clearly divided along the timing axis. The left cluster represents the designs where the branching code is disabled. The more optimized branching code gets better performance as we increase the unrolling factor. The discontinuity between the middle and right clusters is caused by a jump in the knob values. The impact of the unrolling factor is however limited by the number of edges to process per node. This limitation is reflected in the sparse input design space that is more uniform.

B. Discrete Cosine Transform (DCT)

This algorithm is based on the NVIDIA CUDA implementation of a 2D 8x8 DCT [12]. The program divides the input signal into 8x8 blocks loaded into shared memory, then processed by calculating DCT for rows then columns with precalculated coefficients. Some knobs enable multiple blocks to be loaded in shared memory, and multiple rows and columns to be processed simultaneously. Each work-group processes one 8x8 block, and within the group each work-item processes one row and column. This can be altered by 9 tunable knobs:

- **SIMD** and **Compute units**.
- **Block size**: Number of rows/columns to process per work-item.
- **Block dim. X** and **Block dim. Y**: Number of blocks per work-group in X or Y direction.
- **Manual SIMD type**: Using OpenCL vector types, each work-item processes either multiple consecutive rows/columns, or processes multiple rows/columns from different blocks.

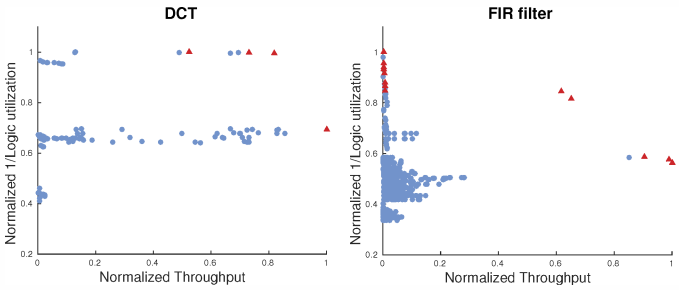

Fig. 3. DCT and FIR filter design spaces.

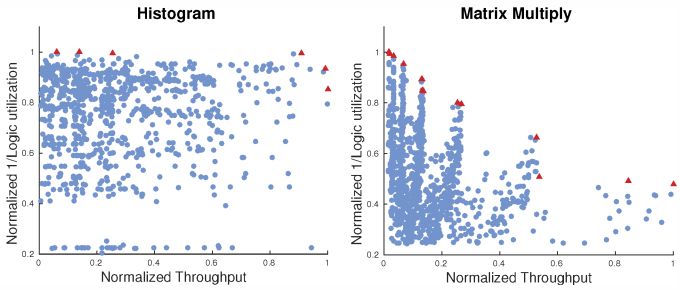

Fig. 4. Histogram and Matrix multiplication design spaces.

TABLE II. FIR FILTER PARETO OPTIMAL DESIGNS.

Coef. shift	8	8	1	8	1	1	8	1	8	1	8	1
Num. parallel	1	2	2	1	1	4	2	2	2	2	1	1
Unroll inner	32	32	32	32	32	1	2	2	1	1	2	2
Unroll outer	2	1	1	1	1	1	1	1	1	1	1	1
Work-items = Work-groups = SIMD = Compute units = 1												
Time (ms)	0.70	0.71	0.78	1.08	1.14	80.73	85.35	94.93	151.9	159.9	190.1	192.2
Logic	52%	50%	50%	36%	34%	34%	34%	33%	32%	31%	31%	30%

- **Manual SIMD**: Number of rows/columns for one work-item to process using SIMD vector types.
- **Unroll factor**: Unroll factor for the loops launching 8-point DCT on rows and columns.
- **DCT unroll**: Either use the loop version of 8-point DCT, or the manually unrolled 8-point DCT.

Design space: (Figure 3) The designs are clearly divided into clusters along the logic utilization axis. These clusters can be mostly explained by the *block size*, *manual SIMD* and *compute units* knobs that have a similar impact on logic. The combination of these knobs increases by steps, creating the clustering of the design space.

C. Finite Impulse Response (FIR) Filter

This benchmark is based on the Altera OpenCL design example of a Time-Domain FIR Filter, itself based on the HPEC Challenge Benchmark suite [4]. This code implements a complex single-precision floating point filter, where multiple filters are applied to a stream of input data using a sliding window. After a block of input data has been processed, it loads the next filter's coefficients while still shifting the sliding window to avoid too much branching complexity. The kernel is originally a single work-item task, but it has been extended to use multiple work-items. There are 8 tunable knobs:

- **Coefficient shift**: Number of filter coefficients to load at each loading iteration.
- **Num. parallel**: Number of FIR computations to perform in a single iteration. This extends the size of the sliding window.
- **Unroll inner**: Unroll factor for the FIR computation loop (for each coefficient).
- **Unroll outer**: Unroll factor for the main loop (for each input value).
- **Work-items** Number of work-items. This divides the input data into multiple blocks, each work-item works on one block.
- **Work-groups**: Number of work groups, same consequences as work-items, but also enable compute units.
- **SIMD** and **Compute units**.

Design space: (Figure 3) The FIR filter benchmark has this particularity to present a small group of outlier results that turn out to be the most efficient designs. Unsurprisingly, the values of the knobs correspond to the original code from Altera that is thoroughly optimized for FPGA. It's a single work-item sliding window with a fully unrolled filter computation, loading 8 complex numbers when loading a new filter (8x2x32 bits = 512 bits, the external memory width). The difference with the original is the unrolling or sliding window size that are bigger, allowing two elements per iteration to be processed. This is possible because we use a smaller filter size than the original. As we follow the Pareto front toward less logic utilization (Table II), we simply unroll less the computation, and decrease the sliding window size. From this design space we can also learn that, even though not Pareto optimal, the next most efficient designs come from using pipelining with multiple work-items. It is less efficient due to accesses to non-contiguous portions of the external memory.

D. Histogram

This code calculates the distribution histogram of unsigned 8-bits values by calculating the number of each of the 256 unique values. One OpenCL kernel counts the input values into a local histogram. If multiple work-items are used, we divide the input data and calculate multiple histograms that can be combined either in shared memory, or through global memory with a second kernel. The second kernel uses a single work-item to sum them locally and output the result. The first kernel can also process multiple values at the same time by using several local histograms. There are 7 tunable knobs:

- **Num. histograms**: This is the number of local histograms in the first kernel to compute simultaneously.
- **Histogram size**: Switches between local histogram storage in registers or in block RAM.
- **Work-items**: This will create intermediate results that need to be accumulated.
- **Work-groups**: If there are multiple work-groups, the intermediate results have to be accumulated in a second kernel (cannot use shared memory).
- **Compute units**.
- **Accum. shared memory**: Choose to accumulate the intermediate results in shared memory if possible (only one work-group), or in a second kernel.
- **Unroll factor** Unrolls the main loop over the input values.

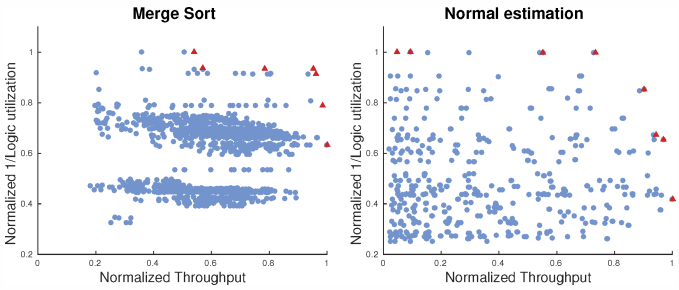

Fig. 5. Merge sort and Normal estimation design spaces.

Design space: (Figure 4) This is one of the few design spaces that appear relatively uniform. All the knobs seem to have a similar impact on the variations between designs, although a few design make the exception by using more logic. These designs set the *histogram size* such that the compiler will prefer to use registers instead of block RAMs, as they are using only one local histogram. But as opposed to some other uniform design spaces, the parameters cannot be linearly modeled, as presented in Section V.

E. Matrix Multiplication

This code is based on an Altera OpenCL example. It implements a simple matrix multiply $C = A * B$ with squared floating-point matrices. The matrix C is divided into blocks, each computed individually. The implementation includes several knobs to change the size of the blocks and to process multiple blocks at once. Each work-group takes care of one block of C. Each work-item takes care of one element in a block, including loading elements to local storage, multiplying one row of block A by one column of block B, and copying back to global storage. There are knobs that enable multiple block processing for work-groups and work-items, either by adding an inner loop, or by using OpenCL vector types for SIMD computation. There are 9 tunable knobs in this code:

- **Block dimension**: Width of blocks.
- **Sub-dimension X** and **Sub-dimension Y**: How many blocks of C in X or Y direction to process in one work-group. This adds a for loop so that each work-item processes this number of blocks.
- **Manual SIMD X** and **Manual SIMD Y**: How many blocks of C in X or Y direction to process in one work-group. This performs the inner matrix multiply with OpenCL vector types.
- **SIMD** and **Compute units**
- **Enable unroll**: Enables or disables unrolling loop on load and store operations.
- **Unroll factor**: Unroll factor for multiple loops.

Design space: (Figure 4) The matrix multiplication Pareto-optimal designs are a good example of an almost linear relationship between area and timing (in this optimization space). By looking at the knob values along the Pareto front, we can determine that it's mostly a combination of *block dimension*, *manual SIMD X*, *SIMD*, and *unroll factor* that can vary the results and create the trade-off between speed and area. The other knobs still have an impact on the design space, but tend to have a single optimal value for both area and timing. Typically, *manual SIMD Y* is not enabled in the optimal designs, as the data are organized along the X direction.

F. Merge Sort

This program applies the Merge Sort algorithm using loops, merging in local memory first, then in global memory:

```
1:  for each chunk of size localsortsize do
2:      copy the entire chunk into local memory
3:      for localchunksize = 2 to localsortsize do
4:          for each local chunk of the chunk do
5:              merge two halves of local chunk
6:          end for
7:          swap input/output buffers
8:      end for
9:  end for
10: for chunksize from localsortsize to inputsize do
11:     for each chunk of the input do
12:         merge two halves of the chunk
13:     end for
14:     swap input/output buffers
15: end for
```

- **Work-items**: Each work-item processes a different chunk in lines 4 and 11.
- **Local sort size**: Varies $localsortsize$.
- **Local use pointer**: Use pointers to swap buffers in local memory. This can force the use of block RAMs instead of registers.
- **Specialized code**: Enable a specialized code to merge chunks of size 2.
- **Work-groups**: Each work-group runs the algorithm on one portion of the input data. A second iteration of the kernel is launched to merge the final output.
- **Compute units**
- **Unroll**: Unroll factor for the loops copying data from/to local memory.

Design space: (Figure 5) This design space is mainly divided into 2 clusters, due to the *compute units* knob that can take the value 1 or 2. In this case, using multiple compute units has a large impact on the resource utilization, while the other knobs have a much smaller impact on resources, and are responsible for smaller variations within each cluster. Interestingly, the fastest designs use only one compute unit, but make use of the pipeline optimization from work-items.

G. 3D Normal Estimation

This code is inspired by an algorithm in the KinectFusion code from the PCL library [13]. It estimates the 3D normals for an organized 3D point cloud that comes from a depth map (vertex map). We can quickly estimate the normals for all the points by using the right and bottom neighbors on the 2D map, then calculate the cross-product of the difference between each neighbor and the current point, and normalize. If any of the vertices is null, the normal is null. One kernel does the entire computation using a small sliding window where the right neighbor is shifted at each iteration to be reused in the next iteration. As illustrated in Figure 6, a parameter can vary the window size so that multiple inputs are processed in one

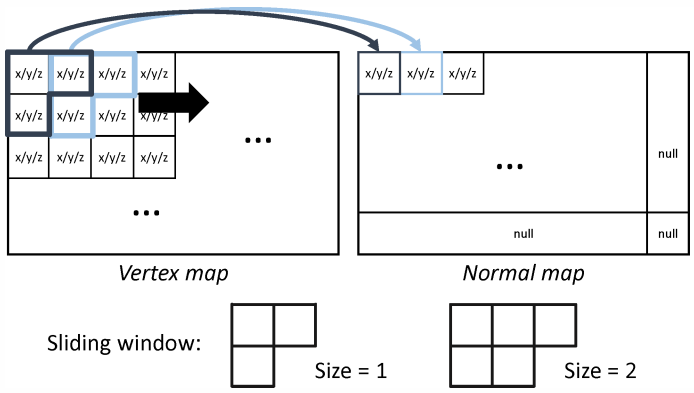

Fig. 6. Estimating 3D normals from 3D vertices organized on a 2D map. The top of the figure shows how the sliding window works in the algorithm. The bottom illustrates how the sliding window can be tuned.

iteration. If multiple work-items are used, the input data are cut into blocks of whole rows. There are 6 tunable knobs:

- **Work-items**, **Work-groups** and **Compute units**.
- **Unroll factor 1**: Unroll factor for the outer loop that iterates over all the input data.
- **Unroll factor 2**: Unroll factor for the loop that iterates over the elements within a sliding window.
- **Window size**: Size of the sliding window. *ie.* number of consecutive elements to process in one iteration.

Design space: (Figure 5) Normal estimation is another example of a fairly uniform design space. It is easier to create a model of the knobs (see Section V), and it is a good example of a optimization space where most parameters have an impact of similar importance on both the timing and the area utilization.

H. Sobel Filter

This code applies a Sobel filter on an input RGB image, based on the Altera OpenCL example.

```
1:  for each block on the input image do
2:    load pixel values from block in shared memory
3:    for each pixel in local storage do
4:      load the 8 pixels around from shared memory to
        registers
5:      convert pixels to grayscale
6:      apply the 3x3 filter in X and Y
7:      combine the X and Y results and apply threshold
8:      save result in global storage
9:    end for
10: end for
```

Work-group take care of blocks (line 1) and work-item take care of pixels within the block (line 3). The knobs can enable a sliding window within the blocks, SIMD computation, or make a work-item perform multiple computations. With the SIMD parameter, each work-item loads more pixels to registers to apply multiple filters by using OpenCL vector types. The sliding window parameter creates an inner loop (after line 3) where each work-item processes one pixel (or multiple with SIMD), then shifts the registers to load one new row or column of data from the local storage. There are 8 knobs in this code:

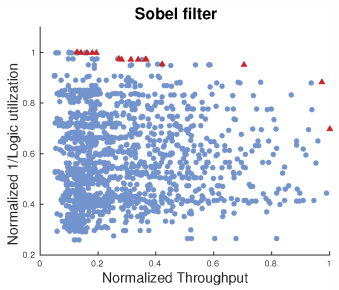

Fig. 7. Sobel filter design space.

- **Block dimension X** and **Block dimension Y**: Size of each block in X or Y.
- **Sub-dimension X** and **Sub-dimension Y**: Local sliding window size, moving in X or Y direction.
- **Manual SIMD X** and **Manual SIMD Y**: Number of elements to process as SIMD in X or Y direction.
- **SIMD** and **Compute units**.

Design space: (Figure 7) The Sobel filter is another example of a mostly uniform design space where all the knobs seem to have a similar impact on the output variables. A more detailed look at the knob values actually shows that along the timing axis, the *manual SIMD X* knob is one of the most important factors, and the most important on the Pareto front. This is a case where manually designing SIMD computation is better than using automatic SIMD, and this becomes apparent from the analysis of an entire optimization space.

I. Sparse Matrix-Vector Multiplication (SPMV)

This code is also based on an OpenDwarfs benchmark. It calculates $Ax+y$ where the matrix A is sparse in CSR format and the vectors x and y are dense. Each work-item processes one row of A to multiply the non-zero elements by elements of x. Two knobs control the number of elements processed simultaneously by one work-item: one unrolls a loop, and the other enables the use of OpenCL SIMD vector types to store and multiply the data. There are 4 tunable knobs:

- **Block dimension**: Number of work-items per work-group.
- **Compute units**.
- **Unroll factor**: This creates an inner loop over some number of elements and unrolls it so that elements can be processed simultaneously.
- **Manual SIMD width**: This is the size of the OpenCL vector type to use when processing elements. Elements are loaded and multiplied in parallel using this type.

Design space: (Figure 8) This benchmark is dependent on the type of input and behaves differently for more or less sparse matrices. This is reflected in the design spaces, where the most efficient designs for sparse matrices, and particularly the Pareto optimal designs, tend to have smaller values for *Unroll factor* and *Manual SIMD width*. When processing a denser matrix, the best designs tend to have a higher value for these knobs, as it allows simultaneous processing of elements in rows. For sparse matrices, the pipelining provided by *block dimension* is usually preferred to the SIMD and unroll optimizations.

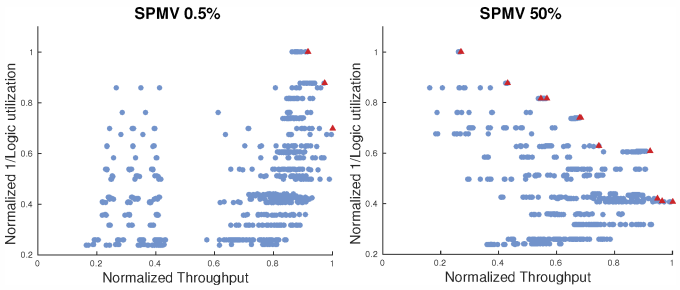

Fig. 8. SPMV design spaces for 0.5% sparse matrix and 50% sparse matrix.

V. Design Space Analysis

To demonstrate one potential use of our data, we perform an example analysis to determine the viability of multiple sparse linear regression to model design space performance and area. In mathematical form we compute model coefficients β in

$$f(x) = \beta_0 + \sum_{i=1}^{n} x_i\beta_i + \sum_{i=n+1}^{m}\sum_{j=i+1}^{m} x_i x_j \beta_{i\cdot m+j} \qquad (1)$$

where x is a vector of design space knob values, n is the number of design space knobs, and the number of entries in β is $m = n(n+1)/2$. In the following sections, we use the term "parameters" to refer to values of β and "realization" to refer to a single design (a single point in the design space plots of previous sections).

The purpose of this analysis is *not* to suggest that linear regression is a good idea when seeking to model a design space in general. Rather we observe that there are many cases where simple linear models are effective, *and equally many* where they are misleading and/or downright ridiculous. The lesson here is that DSE research involving parametric models should not overstate their generality, particularly where performance is concerned.

A. The Least Absolute Shrinkage and Selection Operator (LASSO)

The LASSO is a well-known statistical operator [14] useful for variable selection and sparse modeling. While we present its mathematical form in equation (2), LASSO is, in essence, ordinary least squares regression with a penalty forcing small variables toward zero. The operator parameter λ determines "small", and it is often—as it is in our case—selected via cross-validation. A small value at β_k indicates that variation of the kth parameter does not produce significant variation in the output. We prefer LASSO for this analysis because it tends to produce simpler and more interpretable models. The LASSO in mathematical form is

$$\min_{\beta} \|y - X\beta\|_2^2 + \lambda\|x\|_1 \qquad (2)$$
$$= \min_{\beta} \sum_{i=1}^{n}(y_i - \sum_{j=1}^{m} X_{ij}\beta_j)^2 + \lambda\sum_{i=1}^{m}|\beta_i|$$

where X_{ij} is the entry of the matrix X at row i and column j. In the remainder of this section β refers to the vector minimizing the LASSO for a λ minimizing the model mean squared error.

TABLE III. LASSO r^2 and $G(\beta)$ values for logic (ℓ) and timing (t) across benchmarks for complete and near-Pareto spaces

	Complete Space				Within 0.1 of Pareto			
Benchmark	r_ℓ^2	$G_\ell(\beta)$	r_t^2	$G_t(\beta)$	r_ℓ^2	$G_\ell(\beta)$	r_t^2	$G_t(\beta)$
BFS (Dense)	0.89	0.91	0.97	0.97	0.92	0.85	0.83	0.97
BFS (Sparse)	0.89	0.91	0.85	0.84	0.98	0.68	0.92	0.80
DCT	0.98	0.83	0.91	0.86	0.58	0.86	0.95	0.86
FIR	0.61	0.92	0.37	0.94	0.79	0.95	0.99	0.96
Histogram	0.73	0.90	0.04	0.80	-0.05	1.00	0.15	0.94
Matrix multiply	0.83	0.76	0.70	0.70	0.91	0.82	0.94	0.71
Normal estimation	0.90	0.81	0.91	0.57	0.98	0.72	0.98	0.69
Sobel	0.78	0.77	0.88	0.67	0.98	0.83	0.94	0.78
SPMV (Sparse)	0.88	0.93	0.29	0.71	0.90	0.89	0.24	0.93
SPMV (Dense)	0.88	0.93	0.24	0.82	0.90	0.94	0.68	0.80
Mergesort	0.91	0.89	0.43	0.68	0.95	0.92	0.74	0.81

Note: $G(\beta)$ values for which the associated $r^2 < 0.7$ are greyed out to indicate that they should be disregarded

While LASSO explicitly determines coefficients for a linear model, it is also useful for variable selection in nonlinear systems [15]. In this situation we do not read very deeply into β, but rather use it to detect when simple linear, perhaps even obvious, relationships exist between parameters and the realized design space. To summarize the LASSO results we compute the coefficient of determination—also known as the r^2 value—independently for throughput and area, denoted r_t^2 and r_ℓ^2 respectively. r^2 is a commonly used goodness-of-fit measure indicating the amount of variance in the data explained by the model. Alongside $r_{t,\ell}^2$ we also compute the Gini coefficient of β, $G_{t,\ell}(\beta)$, as a measure of model complexity. Note that if r^2 is small then $G(\beta)$ is a nearly worthless quantity. We do, however, include values for all design spaces in Table III for completeness.

B. Gini Coefficient

The Gini coefficient [16] G is a statistic frequently used to measure economic inequality. G takes values in the range $[0, 1]$. If $G(v) = 1 - \epsilon$ for a particular vector v and small ϵ, then there are a few elements of the set that are very large relative to others. If $G(v) = \epsilon$ then all vector elements have values that are close to each other.

Equation (3) describing G is calculated using equation (3) with bias correction from [17]

$$G(x) = \frac{n}{n-1} \cdot \frac{\sum_{i=1}^{n}(2i - n - 1)x_i}{n\sum_{i=1}^{n} x_i} \qquad (3)$$

where x is sorted beforehand. $G(\beta)$ indicates whether variation in the realized design space can be accounted for by a few parameters.

C. Example observations

To give the reader some idea of the sort of descriptive statistical information that can be gained from these data, we will consider only the the r^2 and Gini values from Table III for the *Histogram* and *Normal estimation* design spaces. A purely visual inspection of the design space graph, (see Figure 5 and 4 resp.) might suggest that there is a "grid-like" quality to the relationship between knobs and the realized design space. We demonstrate that this is not necessarily the case.

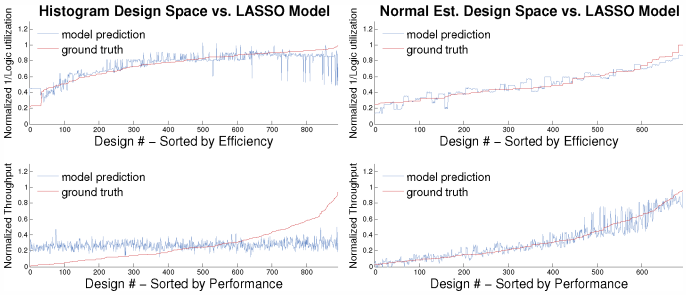

Fig. 9. In the above figure we have sorted the ground truth designs and plotted them alongside the LASSO model predictions. The worst designs begin on the left and progress toward the best designs on the right. The Histogram model predicts performance very poorly, and while the logic utilization model appears to follow rather closely for mediocre designs, the most efficient designs are poorly modeled. The opposite is true for Normal estimation: Near-Pareto designs are modeled more accurately than the remainder of the space.

Histogram: Considering the complete Histogram design space, $r^2_\ell = 0.73$ where r^2_ℓ indicates goodness-of-fit on the logic axis. This means that there is a $1 - r^2$ fraction of the total variance unaccounted for by the model, so 0.73 indicates a reasonable—but not excellent—fit for the LASSO model. $G_\ell(\beta) = 0.9$ tells us that the LASSO model, with learned parameter vector β, has a few dominant parameters, while the remainder have negligible influence over logic utilization. On the other hand, $r^2_t = 0.04$ means that the performance of the design is not well represented as a linear model of the input parameters. For this reason $G_t(\beta)$ should not be taken seriously as an indicator of parameter dominance. Examining these values for the subset of designs within 0.1 of the Pareto front tells a different story. r^2 for both logic and performance are extremely low. While r^2_t increased—meaning the design space becomes more amenable to linear modeling nearer to the Pareto front—logic utilization becomes far less explainable by our model.

Normal estimation: In sharp contrast to the Histogram design space, Normal estimation is very well modelled. r^2 for logic and performance both increase towards the Pareto front. Gini coefficients are split, timing becoming more attributable to a subset of parameters, while logic becomes less so. Altogether this implies that the model parameters are of the same order of magnitude in importance and have proportional (or inversely proportional) relationships to the resulting performance and area.

While these two design spaces are extreme examples on the spectrum of nonlinearity they demonstrate that inspection alone is insufficient to determine the knob-to-design mapping. Figure 9 shows the model predictions alongside the true performance and area results. This example analysis shows that researchers should be very cautious with parametric models in DSE. Even very general techniques such as Gaussian process regression (see [10]) have hyperparameters that must be carefully tuned.

VI. Conclusion

We have presented a set of OpenCL benchmarks targeted specifically at FPGA design space exploration for high-level synthesis. We hope that by releasing these benchmarks and our results to the community, we can expand our knowledge on how to improve design choices. We have analyzed the results to show that the variations between designs can be affected not only by individual parameters, but also by complex interactions between these parameters that are difficult to model mathematically. Yet we have barely scratched the surface of the information that we can gather from these data, and we hope that it will provide opportunities for everyone in the future, and particularly the machine learning community. People can also contribute by compiling these benchmarks on various toolchains, and we plan to expand our work to cover even more optimization types and values.

Acknowledgements

This work was supported in part by an Amazon Web Services Research Education grant.

References

[1] Xilinx sdaccel. Online: http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[2] Altera opencl sdk. Online: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

[3] Spector repository. Online: https://github.com/KastnerRG/spector

[4] R. Haney, T. Meuse, J. Kepner *et al.*, "The hpec challenge benchmark suite," in *HPEC 2005 Workshop*, 2005.

[5] S. Che, M. Boyer, J. Meng *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[6] D. W. Chang, C. D. Jenkins, P. C. Garcia *et al.*, "ERCBench: An open-source benchmark suite for embedded and reconfigurable computing," *Proceedings - 2010 International Conference on Field Programmable Logic and Applications, FPL 2010*, pp. 408–413, 2010.

[7] K. E. Murray, S. Whitty, S. Liu *et al.*, "Titan: Enabling large and complex benchmarks in academic cad," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–8.

[8] W.-c. Feng, H. Lin, T. Scogland *et al.*, "Opencl and the 13 dwarfs: A work in progress," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 291–294.

[9] "On the characterization of OpenCL dwarfs on fixed and reconfigurable platforms," *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, pp. 153–160, 2014.

[10] M. Zuluaga, A. Krause, P. Milder *et al.*, ""smart" design space sampling to predict pareto-optimal solutions," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES '12. New York, NY, USA: ACM, 2012, pp. 119–128.

[11] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, May 2013, pp. 1–7.

[12] A. Obukhov and A. Kharlamov, "Discrete cosine transform for 8x8 blocks with cuda," 2008.

[13] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[14] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[15] S. L. Kukreja, J. Löfberg, and M. J. Brenner, "A least absolute shrinkage and selection operator (lasso) for nonlinear system identification," *IFAC Proceedings Volumes*, vol. 39, no. 1, pp. 814–819, 2006.

[16] C. Gini, "Variabilità e mutabilità," *Reprinted in Memorie di metodologica statistica (Ed. Pizetti E, Salvemini, T). Rome: Libreria Eredi Virgilio Veschi*, vol. 1, 1912.

[17] C. Damgaard and J. Weiner, "Describing inequality in plant size or fecundity," *Ecology*, vol. 81, no. 4, pp. 1139–1142, 2000.