Design and Performance Evaluation of Optimizations for OpenCL FPGA Kernels

Anthony M. Cabrera
Roger D. Chamberlain

Anthony M. Cabrera and Roger D. Chamberlain, “Design and Performance Evaluation of Optimizations for OpenCL FPGA Kernels,” in *Proc. of IEEE High-Performance Extreme Computing Conference* (HPEC), September 2020.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Design and Performance Evaluation of Optimizations for OpenCL FPGA Kernels

Anthony M. Cabrera and Roger D. Chamberlain
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
{acabrera, roger}@wustl.edu

***Abstract*—The use of FPGAs in heterogeneous systems are valuable because they can be used to architect custom hardware to accelerate a particular application or domain. However, they are notoriously difficult to program. The development of high level synthesis tools like OpenCL make FPGA development more accessible, but not without its own challenges. The synthesized hardware comes from a description that is semantically closer to the application, which leaves the underlying hardware implementation unclear. Moreover, the interaction of the hardware tuning knobs exposed using a higher level specification increases the challenge of finding the most performant hardware configuration. In this work, we address these aforementioned challenges by describing how to approach the design space, using both information from the literature as well as by describing a methodology to better visualize the resulting hardware from the high level specification. Finally, we present an empirical evaluation of the impact of vectorizing data types as a tunable knob and its interaction among other coarse-grained hardware knobs.**

I. Introduction

As the era of transistor scaling and Moore's law wanes, systems designers are looking beyond the horizon of general purpose processors towards heterogeneous systems that incorporate hardware accelerators like GPUs and FPGAs. FPGAs are particularly interesting in that they can be used as a platform to architect custom hardware without having to fabricate an ASIC. While the programmability of FPGAs has traditionally been out of reach for those without extensive hardware knowledge, advances in high level synthesis (HLS) have made FPGAs more accessible by allowing for those who want to program FPGAs to use a language that is semantically closer to the application to be accelerated. This, in turn, makes hardware-software co-design more accessible.

To this end, the hardware flexibility of FPGAs, paired with incrementally easier programmability through HLS, can be used to realize the vision of post-Moore systems that incorporate heterogeneous compute components. This improvement in programmability, however, is not without its own challenges. While FPGA hardware can be described using a higher level of abstraction, it is often unclear what hardware results from a specified kernel of computation. Often, the inclusion or exclusion of one line or even a keyword can imply a non-trivial amount of hardware and can have a large impact on the design that is inferred. Furthermore, the choice of design paradigm of an FPGA kernel comes with its own challenges, i.e., should a design be architected as a wide vectorized compute unit that executes multiple threads or a deeply pipelined compute unit that is controlled by a single thread? Each paradigm, additionally, comes with its own coarse-grained design knobs that are specific to that paradigm. Even when the best execution model is chosen, the knobs must be tuned for optimal performance along with other optimizations that may be applicable. We will show that, even for seemingly simple kernels, there are design choices and optimizations to be made whose interaction and performance are not immediately obvious. The situation is, in fact, analogous to the need to optimize codes for good cache performance in the HPC community, which is primarily an empirical task [14], even today [10].

The major contributions of this work are as follows: we use the conversion of EBCDIC to ASCII characters as a case study to architect an FPGA-accelerated instance of this application using the Intel FPGA SDK for OpenCL. We describe the the design of two versions of the kernel that reflect the two, aforementioned design paradigms. We discuss the qualifiers and attributes that are used for both design paradigms and those that are specific to each paradigm, and how those choices impact the hardware that is inferred. We contribute design, experimentation, and performance results to the currently sparse literature targeting the Intel HARPv2 platform using OpenCL, though our findings should be applicable to any OpenCL-compliant FPGA. This includes describing our methodology for overcoming the available tools for OpenCL kernel development on the platform and justifying design decisions through this methodology. We present a heuristic for pruning the design space, which can reduce development time. We empirically show the interactions between design choices in order to show find the most performant design given the implemented design features.

II. Background and Related Work

A. Intel HARPv2 CPU+FPGA Platform

The Intel Hardware Accelerator Research Program (HARP) tightly integrates a server-grade CPU and FPGA. The HARPv2 system, which serves as the target platform in this work, is the second iteration in this program, and combines a 14-core Intel Broadwell Xeon CPU with an Intel Arria 10 GX1150 in the same chip package. Both the CPU and FPGA share a common

off-chip system memory that can be accessed coherently by both devices. This is in contrast to traditional solutions in which an FPGA is connected via PCIe slot.

B. Designing Kernels with OpenCL

Traditionally, programming an FPGA requires domain specific knowledge of digital systems design, which is not a skill of most software developers. Also, the designs are historically expressed at the register-transfer level (RTL) using languages like VHDL, Verilog, or SystemVerilog. High level synthesis (HLS) addresses both of these issues. Specifically, we use the HLS framework provided by the Intel FPGA SDK for OpenCL [7]. HLS effectively allows a programmer to express a computational kernel at a higher abstraction level than RTL, allowing the programmer to focus on the functional specification. This kernel is then translated into an equivalent RTL description by the Intel tools which will be fed into the traditional FPGA synthesis flow. From this point forward, we will refer to the tools that take the OpenCL specified kernel to perform the high level synthesis, logic synthesis, place, and route steps collectively as the *hardware compiler*.

There are two main execution models for designing an OpenCL kernel to target synthesizable FPGA hardware. The first is the multiple work item (MWI) case, in which the work to be executed is divided among multiple threads that are to be scheduled for execution on one or more compute units. This execution model is frequently used on GPUs, whose compute units are comprised of many SIMD vector units that are well suited to take advantage of data-level parallelism. The other execution model is the single work item (SWI) case, in which one thread is responsible for executing the entirety of a compute kernel. This is often best for targeting FPGAs, in which the hardware compiler can account for non-trivial data dependencies and build a custom compute pipeline for the kernel. However, the choice between the two models is non-trivial, as evidenced by Jiang et al. [8].

Though there are many examples in the literature of using HLS frameworks to program FPGAs, we highlight instances that are most relevant to this work. Zohouri et al. focus on the portability aspect of using OpenCL kernels intended for GPUs on FPGAs [17]. Sanaullah et al. propose a framework for describing OpenCL kernels that relies on the stacking of optimizations that should apply generally to all kernels [12]. However, they prescribe that the most performant version of any OpenCL kernel will use the SWI design paradigm and ignore the MWI paradigm. In this work, we explore both paradigms and find that, in our particular application, that the MWI paradigm results in the best performance. Jin and Finkel perform a hardware design space search [9] similar to this work, but do not show the effect of varying the vectorized data types and their interaction with the available coarse-grained knobs. Additionally, they do not show the impact of scaling the input size. In all cases, none of these works target the Intel HARPv2 CPU+FPGA platform using OpenCL. There is a only a small body of literature showing case studies that use the Intel HARPv2 platform in this way [1], [3], [4], [13], [16].

```
1  __attribute__((num_compute_units(NUMCOMPUNITS)))
2  __attribute__((reqd_work_group_size(WGSIZE,1,1)))
3  __attribute__((num_simd_work_items(NUMSIMD)))
4  __kernel void
5  k_e2a(   __global const uchar* restrict src,
6           __global uchar* restrict dst) {
7    unsigned char e2a_lut[256] =
8    {
9      0, 1, 2, 3,156, 9,134,127, /* e2a chars 0-7 */
10     151,141,142, 11, 12, 13, 14, 15, /* 8-15 */
11     ...
12     48, 49, 50, 51, 52, 53, 54, 55, /* 240-247 */
13     56, 57,250,251,252,253,254,255 /* 248-255 */
14   };
15
16   unsigned int i = get_global_id(0);
17   uchar orig_char = src[i];
18   uchar xformd_char;
19
20   xformd_char = e2a_lut[orig_char];
21
22   dst[i] = xformd_char;
23 }
```

Listing 1: Baseline Implementation of the MWI `e2a` kernel using the OpenCL API and syntax.

III. Baseline Kernel Design

A. EBCDIC to ASCII Kernel

In this work, we use the transformation of 8-bit EBCDIC characters to 7-bit ASCII characters (taken from the Data Integration Benchmark Suite [2]) as our case study application for designing and optimizing an FPGA-based implementation. We will henceforth refer to this as the `e2a` application. The application proceeds by reading a 9.2 MB EBCDIC-encoded file from disk, counting the number of elements to be transformed, and then performing the conversion to ASCII characters. The conversion is the section of the original computation that we isolate as an OpenCL kernel, whose code is shown in Listing 1. The conversion is performed by using the EBCDIC character as an index (line 20, Listing 1) into a 256 character look up table (line 7-14, Listing 1) that maps the input EBCDIC character to the appropriate ASCII character.

It is clear that this application is relatively simple. It was chosen for precisely this reason: to enable us to investigate issues of configuration and parameter tuning without the additional issues present with a complex algorithm. Despite its simplicity, it is representative of a broad class of problems that are (1) embarrassingly parallel and (2) relatively straightforward, algorithmically. For example, a number of the applications (although clearly not all) from CommBench [15], DIBS [2], and MiBench [5] are of this type.

B. Baseline OpenCL MWI Kernel Design

1) Memory Access Hardware Compiler Hints: The `const` keyword is applied to the global input buffer `src` (line 5, Listing 1) to tell the hardware compiler that this buffer is read-only. The hardware compiler, in turn, will be given permission to perform more aggressive optimizations regarding loads from this buffer [6]. Both the `src` and `dst` (lines 5 and 6, Listing 1) global memory buffers are both preceded by the `restrict` keyword. This hints to the hardware compiler to "trust" the

```
1  __attribute__((max_global_work_dim(0)))
2  __kernel void
3  k_e2a(    __global const uchar* restrict src,
4            __global uchar* restrict dst),
5            unsigned int total_work_items) {
6    unsigned char e2a_lut[256] = { ... }
7    uchar orig_char, xformd_char;
8    unsigned int i;
9
10   #pragma unroll UNROLL
11   for (i = 0; i < total_work_items; ++i)
12   {
13     orig_char = src[i];
14     xformd_char = e2a_lut[orig_char];
15     dst[i] = xformd_char;
16   }
```

Listing 2: Implementation of SWI `e2a` kernel.

programmer's global memory accesses–this is a guarantee that there will be no pointer aliasing among these global buffers, and that there is no need to account for load and/or store dependencies between the buffers.

2) Choosing the Execution Model: As mentioned in Section II-B, selecting between MWI and SWI execution models is not trivial. To determine the most performant model, we implement a version for both and perform a design space search using the coarse-grained hardware knobs specific to each execution model, whose code is shown in Listings 1 and 2, respectively. For the MWI model, there are three knobs: number of compute unit replicates (`NUMCOMPUNITS`), the required work-group size (`WGSIZE`, i.e., the number of local work items that will belong to a work-group), and the SIMD factor (`NUMSIMD`, i.e., how many times to replicate the data path). These knobs are set in lines 1-3 of Listing 1.

The SWI kernel code, shown in Listing 2, is similar to the MWI kernel, even though their execution models are orthogonal. One difference is the extra argument that tells the kernel how many times to perform the data transformation (`total_work_items` in line 6, Listing 2). All of the work to be executed is wrapped in a for loop whose exit is conditioned on `total_work_items`. Another difference is that there is only one coarse-grained knob associated with this execution model: the loop unroll factor for the for loop in line 11 of Listing 2. This is supplied as a compiler hint set by the tunable parameter `UNROLL` in line 10 of Listing 2.

To perform the design space search, we proceed using the same methodology outlined by Cabrera and Chamberlain [1]. We first determine what values each knob can assume. The design space becomes `WG` $\in \{64, 128, 256, 512, 1024\}$, `NCU` $\in \{1, 2, 4, 8\}$, and `NS` $\in \{1, 2, 4, 8, 16\}$ for the MWI kernel, and `UNROLL` $\in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ for the SWI kernel. We then take the Cartesian product of all sets of assumable values and build each resulting kernel and empirically measure its performance. Through this process, we determine that the best version is achieved using the MWI design paradigm where `WGSIZE` $= 1024$, `NUMCOMPUNITS` $= 8$, and `NUMSIMD` $= 16$ for the input EBCDIC file size of 9.2 MB. We find that, generally, MWI kernels benefit mostly from increasing the knobs to their highest assumable values, which we will use as a design heuristic in Section IV. In particular, larger work-group sizes allow for work to be chunked in a spatially local way. Increasing `NUMCOMPUNITS` and `NUMSIMD` increases throughput by inferring multiple I/O interfaces and widening those interfaces, respectively. Additionally for the latter case, these wider interfaces allow for more data to be statically coalesced for access, which makes better use of the available bandwidth. This configuration now becomes the baseline kernel design from which we will make additional optimizations.

C. Overlapping Data Transfer and Execution

A key feature of the OpenCL environment specific to the Intel HARPv2 platform is that external memory is shared between the CPU and FPGA. This removes the problem of having to transfer data from host memory to FPGA memory and vice versa. This also allows for data transfer to directly overlap execution instead of waiting for explicit reads and writes between host and device memories. While it is possible to achieve data transfer and computation overlap in other OpenCL compliant FPGAs, the overlap in the HARPv2 platform is achievable without the need for breaking up a kernel instance and having to rely on asynchronous methods or double buffering. The interface to this memory is made available as an extension to the OpenCL 1.0 specification. Here, we allocate the `src` and `dst` buffers on the host side using the extension. Figure 1 shows the benefit of this method, which is congruent with related work [1].

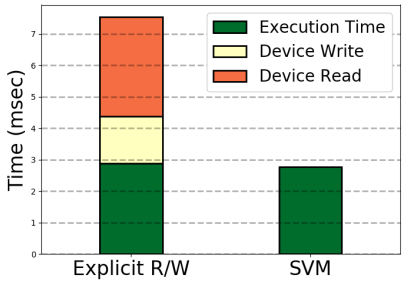

Fig. 1: Execution times of explicit reads and writes to memory and using the OpenCL 1.0 SVM extension.

D. Visualizing the Hardware

A challenge of using HLS to design hardware is the lack of ability to visualize what the hardware compiler will synthesize based on the OpenCL kernel that is authored. To this end, more recent versions of the Intel tools allow for an abstracted system level view of the hardware to be synthesized, by representing the operations to be executed as a control data flow diagram (CDFG) without having to fully synthesize a kernel. Historically in high level synthesis, viewing the abstracted hardware in this form is used to help reason about data dependencies and what cycle(s) to schedule operations on [11]. In this work, we will use this visualization as an aid

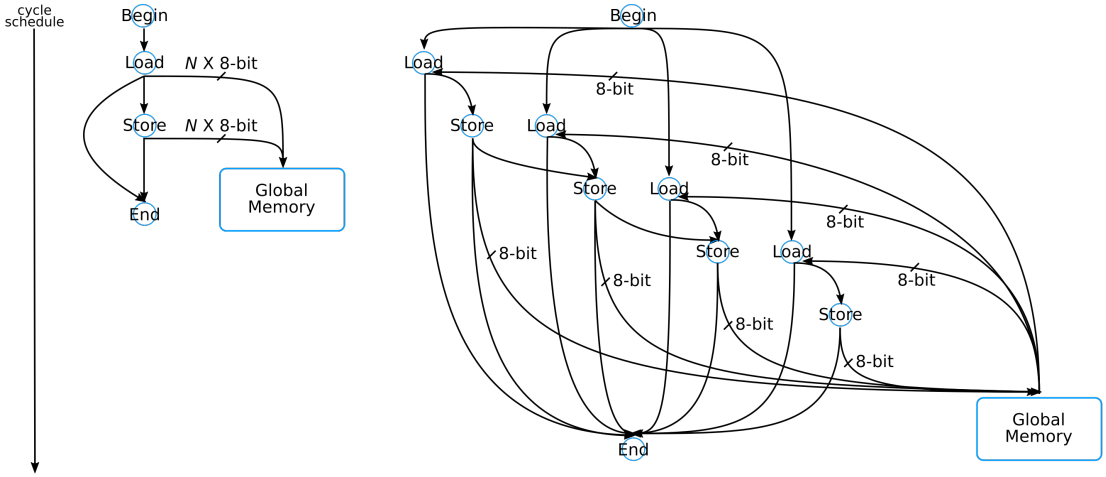

Fig. 2: Approximate cycle schedule of the control data flow graphs (CDFGs) that represent unbounded (left) and bounded (right) versions of the `e2a` kernel.

```
...
 if (i < total_work_items)
 {
  unsigned int i = get_global_id(0);
  uchar orig_char = src[i];
  uchar xformd_char;

  xformd_char = e2a_lut[orig_char];

  dst[i] = xformd_char;
 }
```

Listing 3: Using bounds checking to avoid the "loose ends."

to understand how a design choice made during the OpenCL kernel design process will impact the hardware that results. While this newer version of the tools is not supported by our target platform, we can still use them to effectively visualize design choices. This is valuable as the issue of tool versions is a general problem. We now show a use of this technique that allowed us to prune the design space and make an informed design decision by allowing us visualize a poor design choice made at the OpenCL kernel level and subsequently ignore it.

A requirement of MWI OpenCL kernels is that the work-group size evenly divides the number of global work-items (the total amount of work to be done). This is often not a naturally occurring feature when trying to accelerate applications. Recall that the optimal work-group size of the `e2a` kernel was found to be 1024. Since the global work-item size is not a multiple of 1024, this requirement is not met. In order to address the "loose ends," a common solution is to inflate the global work-item size to satisfy the requirement. In our case, we could pad the input file sizes with `NULL` characters until the input size is a multiple of 1024 and modify the `e2a` kernel to implement bounds checking to make sure that the kernel only processes meaningful input items. This is done by wrapping lines 16-22 of Listing 1 in an `if` statement conditioned on the true global work item size, as shown in Listing 3.

While this is a seemingly innocuous design choice for kernels targeting CPUs or GPUs with hardware support for conditional code, this is a costly operation when synthesizing hardware for the FPGA. Every operation (excluding dead code) specified in the kernel results in logic that gets synthesized into real hardware. The negative impact of this conditioned execution on the hardware may not be immediately obvious, so we leverage the system level viewer of a more recent version of the Intel tools to help better understand the impact of this choice.

Figure 2 shows the system level view of two versions of the MWI `e2a` kernel and the approximate cycle schedules of the CDFGs for an FPGA in the same product line (Arria 10) as our target platform. We will refer to the kernel version with no bounds checking as the unbounded case, and kernels with bounds checking as the bounded case. The left figure represents the CDFG and schedule for the unbounded case and the data path is replicated by a factor of `N` (i.e., `NUMSIMD = N`). It is also the same schedule for the bounded case, but only when `NUMSIMD = 1`. The right figure represents the bounded case where the data path is replicated 4 times.

The intuition behind this juxtaposition is that the schedule of the unbounded case does not change as the data path is replicated while the bounded case serializes accesses to global memory to maintain correctness. The total number of cycles for multiple replicas in the unbounded case scales well as `NUMSIMD` increases because the hardware compiler is able to infer a wider I/O interface to global memory. The bounded case shows that each replica of the data path requires an additional serial global memory access, thereby increasing the cycle count when a work-item is scheduled. Thus, by performing this visualization, we opted to design the MWI kernel using the unbounded approach and take care of the remaining global work-items on the host side.

IV. Widening the Data Type

We now build upon the baseline kernel configuration established in Section III-B2. In this section, we detail an

OpenCL design optimization to aid the hardware compiler in inferring even wider I/O interfaces and further statically coalescing memory accesses. This is accomplished by increasing the width of the data types in the `e2a` kernel, we we do by leveraging the OpenCL specification for vectorized data types. Specifically, we can modify the `uchar` type to `uchar{2,4,8,16}`. The modified kernel version using `uchar4` is shown in Listing 4, where the `src`, `dst`, `orig_char`, and `xformd_char` variables all reflect the new data type. While the kernel description is unaffected by the data type vectorization, this optimization implicitly modifies the global work item size by a factor of the new data type width and effectively creates additional "loose ends." We must account for this in the host side code.

```
...
__kernel void
k_e2a(    __global const uchar4* restrict src,
          __global uchar4* restrict dst)
{
  unsigned char e2a_lut[256] = { ... };

  unsigned int i = get_global_id(0);
  uchar4 orig_char = src[i];
  uchar4 xformd_char;

  xformd_char.s0 = e2a_lut[orig_char.s0];
  xformd_char.s1 = e2a_lut[orig_char.s1];
  xformd_char.s2 = e2a_lut[orig_char.s2];
  xformd_char.s3 = e2a_lut[orig_char.s3];

  dst[i] = xformd_char;
}
```

Listing 4: Kernel with vectorized `uchar` types.

In order to understand the effects of this optimization as it interacts with the existing knobs of the baseline MWI kernel, we create versions of the kernel with each available widened data type and use a reduced design space as guided by the heuristic developed in Section III-B2. The new design space becomes `WGSIZE` $\in \{512, 1024\}$, `NUMCOMPUNITS` $\in \{1, 2, 4, 8\}$, and `NUMSIMD` $\in \{1, 2, 4, 16\}$. In this case, there are 32 unique configurations that we consider in order to evaluate this optimization.

Once the most performant kernel is found, we measure the impact of input scaling on this kernel. This is done by using the original file to create differently-sized versions (roughly powers of 2 in file size) up to 1 GB.

V. Results

Figure 3 shows the result of the design space search detailed in Section IV. Each sub-graph represents a different data vectorization factor: Figures 3(a), 3(b), 3(c), 3(d), and 3(e), represent data vectorization factors of 1, 2, 4, 8, and 16, respectively. The x-axes show every kernel configuration for its respective vectorization factor, where each label represents: `WGSIZE-NUMCOMPUNITS-NUMSIMD`. On the *y*-axes are the observed data rates for each configuration. The two differently colored bars in Figure 3(e) represent configurations that could not be physically realized by the hardware compiler. The dotted black line represents the best data rate for a OpenCL MWI kernel targeting a Intel Core i7 Kaby Lake processor. The line is situated at 6.39 GB/s. (The Linux `mbw` utility reports that this Intel Core i7 machine achieves an average memory copy bandwidth of 11.6 GB/s.) Thus, any configuration whose respective bar is below the dotted line has a lower data rate than the multi-core CPU version and a higher data rate if a bar is above the line.

Figure 3(a) shows the results for no data type vectorization. All configurations in this group perform worse than the best CPU implementation. As shown in Table 1, we see that the most performant version without data type vectorization is $0.865\times$ the CPU data rate. However after the first level of type vectorization in Figure 3(b), we observe four configurations that perform better than the CPU. This indicates that while the kernel configuration with just the coarse-grained knobs has been tuned, there is still further room for improvement. Specifically, as the data types become wider in Figures 3(c), 3(d), and 3(e), we observe that additional configurations become more performant than the CPU case. This further validates our heuristic from Section III-B2 for pruning the design space for MWI kernels.

The main performance benefit comes from aiding the hardware compiler to statically coalesce memory accesses. Without this optimization, the widest interface that can be created for a single compute unit instance of a kernel is by replicating the data path up to 16 times. With the wider data type, a single data path can read and write `N uchars`, where `N` is the data vectorization factor. Additionally, this aids the burst-coalesced load-store unit (LSU) generated by the hardware compiler. Because of the vectorized types, a single address points to multiple data items, as opposed to just one data item. These addresses are queued up and coalesced in the LSU for a burst access. Thus, a burst access can grab up to `N`$\times$ more data in the best case when compared to the coarse-grained configuration without data type vectorization.

We also observe evidence supporting the heuristic in Table 1, which shows the Intel HARPv2 FPGA resource utilization, data rate, and speedup relative to the CPU for the two best configurations for each level of data type vectorization. We observe that the speedup for `uchar{8,16}` are only 4.7% and 3.0% slower, respectively, than the optimal configuration. This is a reasonable heuristic to follow when one is willing to make the tradeoff of the locally optimal configuration for one that is relatively close to optimal found in less time.

Finding the optimum requires more experimentation, as shown in Figure 3. When holding `WGSIZE` and `NUMCOMPUNITS` constant, we observe in Figures 3(a), 3(b), and 3(c) that increasing `NUMSIMD` results in a monotonically increasing data rate. However, this monotonic behavior ends when the data vectorization factor is set to 8 and 16. In this case, replicating the data path with wider types creates enough contention for the global memory resources such that the performance degrades by having to orchestrate these accesses.

From the table, we observe that the overall best performing configuration is (4, 512, 1, and 16) for the data vectorization factor, `WGSIZE`, `NUMCOMPUNITS`, and `NUMSIMD`, respec-

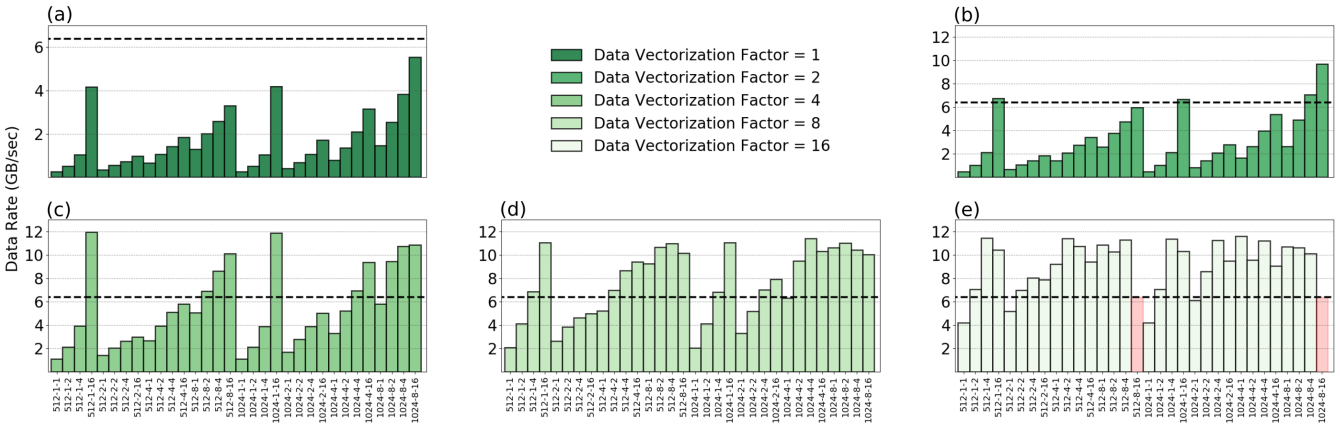

Fig. 3: Design space search for data vectorization factors $\{1, 2, 4, 8, 16\}$. The x-axes represent the coarse-grained configuration `WGSIZE-NUMCOMPUNITS-NUMSIMD` for the given data vectorization factor. The y-axes show the resulting data rate.

Data Vectorization	Work-group Size	# Compute Units	SIMD Factor	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	Data Rate (GB/s)	Speedup
1	1024	8	16	238.77	28%	8%	23%	5.529	0.865
	1024	1	16	280.19	24%	6%	16%	4.176	0.654
2	1024	8	16	237.19	29%	9%	28%	9.694	1.517
	1024	8	4	254.71	28%	7%	21%	7.071	1.107
4	512	1	16	268.81	24%	7%	17%	**11.933**	**1.868**
	1024	1	16	268.81	24%	7%	17%	11.856	1.856
8	1024	4	4	258.26	26%	8%	21%	11.400	1.784
	1024	1	16	247.77	24%	8%	20%	11.058	1.731
16	1024	4	1	282.56	26%	7%	19%	11.587	1.814
	512	1	4	255.75	24%	7%	17%	11.443	1.791

Table 1: Resource utilization and results for the two best coarse-grained configurations for each level of data type vectorization.

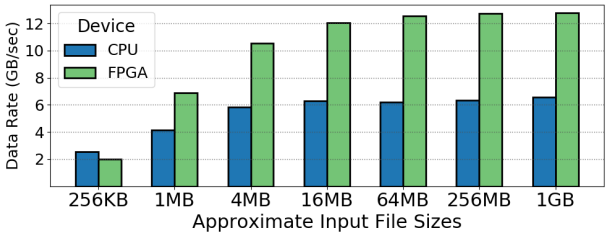

Fig. 4: Input File Size Sweep

tively. Its data rate is 11.933 GB/s–over one-third of the theoretical read/write bandwidth [4]–and a speedup of $1.868\times$ over the CPU implementation. We observe that the best result does not have the widest data vector type or any replicated compute units. In this case, there is less contention for global memory among compute unit replicates. Additionally, it is easier for the OpenCL runtime to schedule work-groups for execution because there is only one compute unit for which to issue commands. These system-level observations can be aided by observing the reported resource utilization numbers and maximum clock speeds in Table 1. (Related work historically has been able to account for differences in performance, in part, by using such results [1], [17].) Future work could include being able to incorporate this data to model the performance impact of interactions like these between design choices in order to more efficiently search the design space.

Figure 4 compares the performance of the best kernel configuration to the best CPU version when scaling the input size from approximately 256 KB to 1 GB. The CPU data rate performance starts to plateau at when the input file size is 16 MB, and the best achievable data rate is 6.55 GB/s. The Intel HARPv2 platform data rate begins to plateau when the input size is greater than 16 MB, and the best achievable data rate is 12.76 GB/s. The speedup factor of the Intel HARPv2 performance over the CPU is $1.95\times$. Although the kernel is relatively simple, input sizes of 16 MB and up are sufficient to stress the system into the asymptotic limit for data rate.

VI. Conclusions and Future Work

Using HLS frameworks like the Intel FPGA SDK for OpenCL make FPGAs clearly more accessible to accelerate applications in the dawn of a post-Moore compute landscape.

However, this is not without its own challenges. The expressiveness contained in as little as one keyword increases the difficulty of understanding the underlying hardware that results from the high level specification. To this end, we explicitly detail the design of an example kernel, including using CDFGs to visualize the pre-synthesized hardware in order to make more informed design decisions. We also develop a design heuristic for MWI kernels to prune the space design space, trading the optimal configuration for a near-optimal one using less development time. By sweeping the the target kernel's hardware knobs, we show that the interactions between knobs are non-trivial. Specifically, we show that there is a benefit to vectorizing data types for buffers that will be accessed contiguously. However, global memory contention induced when knob settings were near their maximum values necessitates a finer tuning of the configuration to achieve optimal performance. Finally, we show that scaling the input size in our case study stressed our platform enough to reach the asymptotic data rate.

Clearly, one of the limitations of this work is the algorithmic simplicity of the example kernel. It is not clear how well (or even if) our empirical results will generalize to a wider set of applications that have data dependencies and/or a greater amount of control complexity. Expanding the investigation to additional applications is the next step.

In addition, while this investigation is completely empirical (i.e., all performance data come from measurements of the example application executing on the physical machine), it is common practice for compilers in the software world to utilize performance modeling to guide compile-time decisions. A better understanding of the fundamental relationships between the configuration parameters and resulting performance would enable HLS synthesis tools to do a similar thing. The vision is for many of these tuning decisions, that are currently the responsibility of the application's author, become the responsibility of the tool chain.

Acknowledgment

Thanks to Intel for access to the CPU+FPGA system through the Hardware Accelerator Research Program. This work supported by NSF grant CNS-1763503.

References

[1] A. M. Cabrera and R. D. Chamberlain, "Exploring portability and performance of OpenCL FPGA kernels on Intel HARPv2," in *Proc. of Int'l Workshop on OpenCL*. ACM, Apr. 2019.

[2] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain, "DIBS: A data integration benchmark suite," in *Proc. of ACM/SPEC Int'l Conf. on Performance Engineering Companion*, Apr. 2018, pp. 25–28.

[3] C. J. Faber *et al.*, "Data integration tasks on heterogeneous systems using OpenCL," in *Proc. of Int'l Workshop on OpenCL*. ACM, 2019.

[4] T. Faict, E. D'Hollander, D. Stroobandt, and B. Goossens, "Exploring OpenCl on a CPU-FPGA heterogeneous architecture research platform," in *Proc. of Int'l Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC)*, 2019.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of 4th IEEE Int'l Workshop on Workload Characterization*, Dec. 2001, pp. 3–14.

[6] *Intel® FPGA SDK for OpenCL™ Pro Edition: Best Practices Guide*, Intel, April 2020.

[7] *Intel® FPGA SDK for OpenCL™ Pro Edition: Programming Guide*, Intel, April 2020.

[8] J. Jiang, Z. Wang, X. Liu, J. Gómez-Luna, N. Guan, Q. Deng, W. Zhang, and O. Mutlu, "Boyi: A systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs," in *Proc. of ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays*, 2020, pp. 299–309.

[9] Z. Jin and H. Finkel, "Performance-oriented optimizations for OpenCL streaming kernels on the FPGA," in *Proc. of Int'l Workshop on OpenCL*, 2018, pp. 1–8.

[10] J. Kurzak, Y. M. Tsai, M. Gates, A. Abdelfattah, and J. Dongarra, "Massively parallel automated software tuning," in *Proc. of 48th Int'l Conf. on Parallel Processing*, 2019.

[11] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[12] A. Sanaullah, R. Patel, and M. Herbordt, "An empirically guided optimization framework for FPGA OpenCL," in *Proc. of Int'l Conf. on Field-Programmable Technology*. IEEE, 2018, pp. 46–53.

[13] Y. Su, M. Anderson, J. I. Tamir, M. Lustig, and K. Li, "Compressed sensing MRI reconstruction on Intel HARPv2," in *Proc. of 27th Int'l Symp. on Field-Programmable Custom Computing Machines*. IEEE, 2019, pp. 254–257.

[14] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[15] T. Wolf and M. Franklin, "CommBench – a telecommunications benchmark for network processors," in *Proc. of IEEE Int'l Symp. on Performance Analysis of Systems and Software*, Apr. 2000, pp. 154–162.

[16] J. Zhang, S. Khoram, and J. Li, "Efficient large-scale approximate nearest neighbor search on OpenCL FPGA," in *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, 2018, pp. 4924–4932.

[17] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proc. of Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 409–420.