On the Characterization of the Performance-Productivity Gap for FPGA

Atharva Gondhalekar  
Department of ECE  
Virginia Tech  
Blacksburg, VA, USA  
atharva1@vt.edu

Thomas Twomey  
Department of ECE  
Virginia Tech  
Blacksburg, VA, USA  
twomey@vt.edu

Wu-chun Feng  
Department of CS and ECE  
Virginia Tech  
Blacksburg, VA, USA  
feng@cs.vt.edu

Abstract—Today, FPGA vendors provide a C++/C-based programming environment to enhance programmer productivity over using a hardware-description language at the register-transfer level. The common perception is that this enhanced productivity comes at the expense of significantly less performance, e.g., as much an order of magnitude worse.

To characterize this performance-productivity tradeoff, we propose a new composite metric, , that quantitatively captures the perceived discrepancy between the performance and productivity of any two given FPGA programming languages, e.g., Verilog vs. OpenCL. We then present the implications of our metric via a case study on the design of a Sobel filter (i.e., edge detector) using three different programming models — Verilog, OpenCL, oneAPI — on an Intel Arria 10 GX FPGA accelerator. Relative to performance, our results show that an optimized OpenCL kernel achieves 84% of the performance of an optimized Verilog version of the code on a 7680×4320 (8K) image. Conversely, relative to productivity, OpenCL offers a 6.1× improvement in productivity over Verilog, while oneAPI improves the productivity by an additional factor of 1.25× over OpenCL.

Index Terms—FPGA, hardware-description language (HDL), high-level synthesis (HLS), oneAPI, OpenCL, Verilog, performance, productivity, register-transfer level (RTL), SLOC

I. INTRODUCTION

Historically, the reconfigurable logic in an FPGA has been programmed in a hardware description language (HDL), such as Verilog or VHDL. An HDL provides fine-grained control over resource utilization and latency-sensitive datapaths. This control, however, comes at the cost of significantly longer code development time and more difficult debugging.

A typical development flow for an HDL-based kernel design consists of a multi-stage process, including simulation, timing analysis, and placement and routing. Oftentimes, errors identified at these stages can only be resolved by making appropriate changes in the HDL code. Therefore, writing functionally correct HDL code involves multiple time-consuming feedback loops from various compilation stages back to the HDL code.

To address this complex development flow, the introduction of high-level synthesis (HLS) tools allows developers to write code at a much higher level of abstraction. For example, the vendor-neutral OpenCL standard [1] is a C-based HLS framework, supported by FPGA vendors and their associated runtime systems, e.g., Intel’s FPGA SDK for OpenCL [2] and Xilinx’s SDAccel [3]. More recently, C++-based abstraction frameworks, such as Kokkos [4] and SyCL [5], are being increasingly adopted by the high-performance computing (HPC) community. These frameworks further improve productivity by raising the level of programming abstraction to C++. For example, oneAPI from Intel [6] uses SyCL to support heterogeneous computing across a diverse set of architectures, including CPU, GPU, and FPGA.

OpenCL and oneAPI offer C-based and C++-based code development, respectively, hiding the complexity of HDL and allowing programmers to write code that is more akin to typical software development with much greater productivity. In recent years, OpenCL-based HLS has been adopted in a number of application domains, e.g., deep learning [7], stencil computations [8], and graph processing [9]. Furthermore, HLS frameworks, such as OpenCL and oneAPI, deliver the added capability of deploying the same code to multiple types of accelerators, including CPUs and GPUs. Fig. 1 illustrates the productivity benefits of HLS approaches over HDL via a vector addition kernel.

What is effectively a single-line kernel in a single-source oneAPI code (Fig. 1a) becomes several lines of code in OpenCL (Fig. 1b) and an order of magnitude more lines of code in Verilog (Fig. 1c), which offers explicit control but at the expense of productivity and portability. Conversely, the development ease of HLS comes at a cost of less control over the FPGA hardware and potential performance penalties.

To concretely illustrate the above, we present a case study on the design of a Sobel filter in Verilog, OpenCL, and oneAPI. Through our experiments and the results from [10], Fig. 2 shows that there exists a performance-productivity gap between HDL and HLS frameworks. Achieving either high productivity or high performance generally comes at the expense of the other. While the aforementioned frameworks offer complementary tradeoffs, the performance-productivity gap between these frameworks has not been rigorously quantified to the best of our knowledge. In all, our contributions include

- A new metric, , to capture the performance-productivity tradeoff between FPGA programming abstractions.
edge detection using the Sobel filter has been extensively studied on both fixed (e.g., CPU, DSP, GPU) and reconfigurable architectures (e.g., FPGA).

A. Acceleration of Sobel Filter

Knap et al. use the Sobel filter as a kernel to evaluate the performance of unified virtual memory on Nvidia’s Pascal and Volta GPUs [12]. Nausheen et al. present an efficient implementation of a Sobel filter on an FPGA that achieves submillisecond latency when processing 512×512 images [13]. Zhao et al. realize real-time lane detection using a Sobel filter on an FPGA [14]. For input video with 720p (HD) resolution, they achieve a throughput of over 160 frames per second.

The work of Hill et al. [10] is the most similar to ours. They present a performance-productivity evaluation between OpenCL and HDL. In their evaluation of edge-detection filters, they report the development time of OpenCL to be 6.0× faster than HDL but at the expense of 1.4× higher resource utilization of FPGA hardware and up to 10% performance degradation when compared to HDL.

In contrast to [10], [13], [14], we evaluate our implementation of the Sobel filter on significantly larger images, ranging from 720p (1280×720) to 4K (3840×2160) to 8K (7680×4320). In addition, the optimizations for our Verilog reference implementation deliver a throughput of 549 frames per second for HD images, which is 3.43× better than the throughput reported in [14] and 1.75× better than the measured throughput for HD images in [10].

B. Studies on Performance, Portability, and Productivity

In recent years, there has been a growth in frameworks and languages for writing programs that are portable across diverse hardware platforms. Examples of such frameworks include OpenCL [1], oneAPI [6], and Kokkos [4]. However, achieving high performance on each of the diverse set of
platforms requires platform-aware optimizations and vendor-
specific extensions to the frameworks mentioned above. Thus,
achieving portability across a set of platforms while main-
taining an acceptable level of performance on each platform
remains a daunting challenge.

To quantify the “goodness” of a framework in achieving
performance portability, Pennycook et al. propose a metric
that is the harmonic mean of an application’s performance
efficiency observed across a set of platforms [15]. Harell et al.
propose metrics such as code divergence, maintenance cost,
and development cost to measure performance, productivity,
and portability across a set of platforms [16]. Pennycook et al.
continue to expand on their work on the performance portability
metric by incorporating code convergence [17], which is a
measure of similarity between the two programs written for
two or more heterogeneous platforms. Funk et al. propose
a metric called the relative development time productivity
(RDTP) of a parallel computing framework and define it as
the measured speedup (relative to the serial code) divided by
the “relative effort” Ψ, which, in turn, is the ratio of SLOC in
the parallel code to the SLOC in the serial code [18].

Compared to the metrics proposed in [15]–[17], our metric
applies to a single platform, and it incorporates both per-
formance and productivity in terms of source lines of code and
development time. Compared to [18] where the value of the
metric is measured relative to the performance and
productivity of a serial code, our metric seeks to capture the
performance-productivity tradeoff between any two languages.

III. QUANTIFYING PERFORMANCE VS. PRODUCTIVITY

Quantifying the performance-productivity tradeoffs between
two programming abstractions requires definitions of metrics
to evaluate performance and productivity. For performance, we
measure the performance of our Sobel filter implementations in
frames processed per second. The metrics used to quantify
productivity are described in §III-A.

A. Quantifying Productivity

Productivity is challenging to quantify. Many subjective
factors, like level of expertise of the developer, familiarity with
the programming language, and debugging time can impact a
developer’s productivity. In this paper, we first present two
widely-used productivity metrics — time to develop (TDEV)
and source lines of code (SLOC) — and then leverage these
productivity metrics to propose a new composite metric,
Π_{A→B}, that captures the tradeoff between performance and
productivity of a kernel when making a transition from
a low-level language A to a high-level language B [18].

1) Time to Develop the Application (TDEV): Under ideal
circumstances, productivity can be measured by performing
an exhaustive user study to evaluate developers’ efforts. The
Constructive Cost Model (COCOMO) [19] is one approach
for evaluating productivity. COCOMO incorporates several
qualitative metrics, such as the developer’s level of expertise
and the time required to complete the software project. Hill et
al. use the actual development time to evaluate the productivity
benefits of HLS over HDL. Similar to the aforementioned
work, we report the time to develop the program (TDEV) in
HDL and HLS. TDEV includes the time to plan, write, debug,
test, and optimize the Sobel filter program.

2) Source Lines of Code (SLOC): While the time required
to develop the program may provide insight its use for
objectively quantifying productivity is difficult because we
cannot normalize the qualitative metrics across developers with
varying levels of expertise. Furthermore, such a direct measure
of productivity involves logging the time taken by program-
mers to develop HDL and HLS kernels, respectively. In turn,
the evaluation of development time may be biased because
there can be significant variations in code development time,
depending on the FPGA developer’s HDL and HLS language
expertise. Therefore, to complement the development time,
which is a subjective metric, we use SLOC for an objective
comparison between the productivity of HLS and HDL.

B. Composite Metric for Measuring Performance-Productivity
Tradeoff

Here we seek to quantify the tradeoff between productivity and
performance. Unfortunately, several factors hinder an ac-
curate quantification of the performance-productivity tradeoff.
For instance, developers with varying levels of expertise in
HDLs may perceive the programming productivity of HDL
differently. Sacrificing performance for better productivity
by opting for HLS may not be a viable option for some
developers, e.g., mission-critical systems. On the other hand,
developers may prioritize the rapid prototyping offered by
HLS over achieving high performance via HDL. To assist
developers when making decisions on the choice of pro-
gramming language for the FPGA, we propose a metric,
Π_{A→B}, that captures the tradeoff between performance and
productivity. We have formulated Π_{A→B} such that it evaluates
to zero for the ideal case. The rationale behind our metric is
as follows:

For an application implemented using both a low-
level language A and a high-level language B, the metric
should

1) Reward a transition from A to B if higher pro-
ductivity in B can be achieved without significant
degradation in the performance compared to A.
The value of Π_{A→B} is closer to zero in this case.

2) Penalize a transition from A to B if higher
productivity in B is achieved with a significant
degradation in the performance compared to A.
The value of Π_{A→B} is considerably greater than
zero in such as case.

With this rationale in mind, we define our metric, Π_{A→B}, as
follows:

\[ Π_{A→B} = \frac{ΔT_{A→B}}{ΔP_{A→B}} \]

where the numerator, ΔT_{A→B}, is the relative difference in
the performance of a kernel when making a transition from
a low-level language A to a high-level language B and the
denominator, ΔP_{A→B}, is the relative productivity improve-
ment. It incorporates both SLOC and TDEV of language A
and language B. The numerator, $\Delta T_{A \to B}$, and the denominator, $\Delta P_{A \to B}$, from Equation (1) are given by the following equations, respectively.

$$\Delta T_{A \to B} = \frac{\text{Throughput}_A - \text{Throughput}_B}{\text{Throughput}_A}$$

(2)

$$\Delta P_{A \to B} = \alpha \left( \frac{\text{SLOC}_A - \text{SLOC}_B}{\text{SLOC}_A} \right) + (1 - \alpha) \left( \frac{\text{TDEV}_A - \text{TDEV}_B}{\text{TDEV}_A} \right)$$

(3)

where $0 \leq \alpha \leq 1$

$\alpha$ and $(1-\alpha)$ in the Equation (3) are weights assigned to relative improvements in SLOC and TDEV, respectively. The value of $\alpha$ can be varied depending on the perceived significance of SLOC and TDEV metrics. In this work, we evaluate the value of $\pi$ with the three values of $\alpha$, $(\alpha=1)$, $(\alpha=0)$, and $(\alpha=0.5)$. Setting $\alpha$ as one discards the subjective metric TDEV altogether in favor of an objective SLOC comparison. Setting $\alpha$ as zero discards SLOC and accounts for the development time entirely. When $\alpha$ is 0.5, equal weights are assigned to SLOC and TDEV. As shown in Fig. 3, SLOC and TDEV complement each other. SLOC provides an objective quantification of the effort needed to write the program, while TDEV accounts for the effort spent in the planning, verification, testing, and optimization phases of the program.

IV. CASE STUDY: SOBEL FILTER

In this paper, we explore image edge detection using a Sobel filter. We implement an integer variation of the Sobel Filter, which is representative of a structured-grid dwarf (or motif) from the OpenDwarfs benchmark suite [20]. Fig. 4 shows the application of our Sobel filter on an image taken from [21].

A. Design of Sobel Filter

A Sobel filter is a convolution filter that identifies pixels that occur at the edges of the objects in an image. Edge detection occurs by approximating the derivative of the luminosity or brightness across the pixel. Our implementation of the Sobel filter takes an array of integers, which represent RGB pixels, as input. It then performs a “floating-point free” conversion to a luminosity value, as is done in [22]. Finally, it employs a sliding-window approach to add the horizontal and vertical gradients and compute the output pixel value. Algorithm 1 describes our baseline implementation of the Sobel filter.

Each of our Sobel filters in Verilog, OpenCL, and oneAPI, respectively, implement the computational logic in Algorithm 1 and possess the same computational logic and optimizations. The memory models in our OpenCL and oneAPI implementations are batch-oriented in that all the relevant data is moved from the main memory of the CPU host to the DRAM on the FPGA card before the Sobel filter is applied. The memory model in our Verilog implementation is stream-oriented with the Sobel filter being applied as the data is “streamed” from the main memory of the CPU host to the internal FPGA memory. While the difference in the memory models is not ideal for a direct performance comparison, it allows us to measure the sensitivity of our metric to the variations in the level of optimizations of the two implementations under consideration.
TABLE I: Differences in the Tested Kernel Implementations

<table>
<thead>
<tr>
<th>Language</th>
<th>Memory Model</th>
<th>Loop Unrolling</th>
<th>Vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog</td>
<td>Stream mode</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Batch mode</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>oneAPI</td>
<td>Batch mode</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

✓: implemented. ×: not implemented

We compute the value of $\Pi$ for implementations with variations in memory models and optimizations. Table I shows our tested implementations. The selection of streaming approach for the Verilog implementation is also guided by the availability of existing open-source DMA interfaces [23]. The use of such interfaces simplifies the already complex memory management and kernel design in Verilog. Likewise, using HLS tools like OpenCL and oneAPI simplifies the management of data transfers to and from the device due to its high-level programming abstraction. Our implementation of the Verilog kernel uses the same general logic that is used in the HLS implementations. Most of the practical difficulty of the Verilog implementation lies in the host-to-kernel communication. Our Verilog implementation uses a module created by Emas et al. [23], [24], which provides a level of abstraction with a direct memory interface between host and kernel.

V. Optimizations

In this section, we describe the optimizations explored in the technology stacks. In our OpenCL and oneAPI implementations, we use the single work item configuration for kernel invocation and explore the optimizations for the single work item kernel.

A. Loop Unrolling

The clock speed of FPGAs is relatively low compared to CPUs and GPUs. Achieving high performance on FPGA necessitates high data reuse and parallelism. Parallelism on FPGAs can be realized with circuit replication and a deep pipeline. Loop unrolling allows the HLS compiler to implement deep pipelines. Loop unrolling is also used to infer specialized hardware such as a shift register [25]–[27]. In our implementation of a Sobel filter, we fully unroll the loops. Correct use of the #pragma unroll instructs the compiler to infer a shift register [25]–[27]. Shift registers enable the elimination of loop carried dependencies and allow for initiation intervals as low as one. Additionally, a shift register saves repeated accesses to local or global memory in a means highly conducive to pipelining. Shift registers are also an intuitive part of a rolling window-type image convolution such as the Sobel filter.

B. Vectorization

It has been shown that there is significant performance to be gained by fully saturating the memory bus between the FPGA chip and global memory [28]. The Arria 10 device that we use in this work has a 512-bit wide bus. This allows us to fetch OpenCL’s int16 vector type in a single load instruction. Similarly, the host-to-kernel memory interface used in the Verilog implementation has a 512-bit width. Vectorization enables a “wide” data-parallel pipeline. In the baseline approach, a single output pixel is determined from the eight values on the edge of the $3 \times 3$ convolution. In our vectorized implementation, we apply the Sobel filter to 16 elements at once.

We implement this in Verilog by using a series of generate statements to perform the same operations for all the interior pixels in the window. The same result is created in OpenCL using the OpenCL’s vector types.

VI. Productivity and Performance Evaluation

HDL development is significantly different from C and C++-based HLS development. Differences in the HDL and HLS workflows introduce deviations in the performance and productivity of HLS and HDL approaches. In order to evaluate $\Pi_{A \rightarrow B}$, we need to evaluate the relative difference in the performance and productivity of frameworks under consideration. In this section, we evaluate the performance and productivity of our implementations of Sobel filter in Verilog, OpenCL, and oneAPI.

1 The initiation interval is the number of clock cycles that the pipeline must stall before it can process the next loop iteration.
A. Productivity Evaluation

Table II shows the variation in the SLOC of our implementations. As expected, the Verilog implementation has the most SLOC among the three FPGA programming languages. Optimized OpenCL implementation (ST + LU + V) offers a 2.5× improvement in productivity over Verilog, while oneAPI improves productivity by an additional factor of 1.5 × over OpenCL.

Table III shows the efforts in terms of development time for HLS and HDL implementations. Table IV shows the development time for each of the tested implementations. Implementing the Sobel filter in HLS took significantly less time compared to Verilog.

Working for 20 hours per week, designing an optimized implementation of the Sobel filter required approximately four months, while implementing the same in OpenCL required three weeks. Compared to Hill et al. [10], where authors required six months of work for Verilog and one month for OpenCL, our Verilog and OpenCL implementations took 1.5× and 1.3× less time to develop, respectively. We believe the difference in the development time is due to the use of existing data transfer modules [23], [24] in Verilog and the availability of the reference implementation for OpenCL [22].

B. Performance Evaluation

We evaluate the performance of our implementation using three metrics. The first means of evaluating the performance is an attempt at measuring the on-device computational time. The second metric involves measuring the time between the beginning of the memory transfer from the host to the device and the end of the memory transfer of the computation results back to the host. The first metric abstracts away the different memory models used in Verilog and HLS implementations. It becomes less representative of the actual computational time as the amount of memory to be transferred to the device increases. The second measure of device computation time is more relevant for the case of larger kernels. For Verilog implementation, we cannot measure the on-device computation time accurately. This is because we cannot isolate computation time from the data transfer time in our streaming Verilog kernel, where input pixels are streamed from host to device. Therefore, for Verilog implementation, we only report the total time to solution, which includes the data transfer time and computation time. The third metric used for performance evaluation is throughput in terms of frames per second. We evaluate the value of frames per second by computing the reciprocal of the total execution time in seconds.

We perform our experiments on Intel Arria 10 GX FPGA. Intel(R) Xeon(R) Gold 6128 CPU with an operating frequency of 3.4 GHz is the host CPU. In both OpenCL and oneAPI, we explore the two(2) variants of the Sobel filter kernel. We begin our experiments with a baseline single-task (ST) implementation of the Sobel filter. Then, we evaluate the performance of the single-task version with fully unrolled loops (ST + LU). For OpenCL, we explore the impact of vectorization along with unrolling (ST + LU + V). In Verilog, we implement vectorization(V) by replicating compute-units by a factor of 16. This implementation generates 16 outputs pixels for every incoming stream of pixels.

1) Resource Utilization: The resource utilization report for our implementations is shown in the Table VI. We report the utilization for kernels compiled for 8K images. We observe that our Verilog implementation has higher register usage compared with all other implementations. Higher register usage in Verilog is primarily because of the difference in the memory model, where the Verilog implementation uses an optimized host-to_KERNEL pipeline from [23], [24].

2) Impact of Optimizations: Table V shows the performance of our Sobel filter implementations in Verilog, OpenCL, and OneAPI. For all three problem sizes, we observe that Verilog implementation is the fastest among the three. While the baseline single-task implementation is easier to develop, its performance is significantly lower than the optimized Verilog implementation (∼ 60,000× slower). After applying the optimizations discussed in §V, our OpenCL implementation

<table>
<thead>
<tr>
<th>Language</th>
<th>Implementation</th>
<th>Kernel SLOC</th>
<th>Host SLOC</th>
<th>Total SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog</td>
<td>LU + V</td>
<td>298</td>
<td>131</td>
<td>429</td>
</tr>
<tr>
<td>OpenCL</td>
<td>ST</td>
<td>58</td>
<td>76</td>
<td>134</td>
</tr>
<tr>
<td>OpenCL</td>
<td>ST + LU</td>
<td>61</td>
<td>76</td>
<td>137</td>
</tr>
<tr>
<td>OpenCL</td>
<td>ST + LU + V</td>
<td>94</td>
<td>76</td>
<td>170</td>
</tr>
<tr>
<td>oneAPI</td>
<td>ST</td>
<td>60</td>
<td>27</td>
<td>87</td>
</tr>
<tr>
<td>oneAPI</td>
<td>ST + LU</td>
<td>63</td>
<td>27</td>
<td>90</td>
</tr>
</tbody>
</table>

SLOC: Source Lines of Code
ST: Single-Task Kernel, LU: Loop Unrolling, V: Vectorization

<table>
<thead>
<tr>
<th>Implementation</th>
<th>TDEV (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog</td>
<td>305</td>
</tr>
<tr>
<td>OpenCL (ST)</td>
<td>20</td>
</tr>
<tr>
<td>OpenCL (ST + LU)</td>
<td>25</td>
</tr>
<tr>
<td>OpenCL (ST + LU + V)</td>
<td>50</td>
</tr>
<tr>
<td>oneAPI (ST)</td>
<td>12</td>
</tr>
<tr>
<td>oneAPI (ST + LU)</td>
<td>20</td>
</tr>
</tbody>
</table>

and computation time. The third metric used for performance evaluation is throughput in terms of frames per second. We evaluate the value of frames per second by computing the reciprocal of the total execution time in seconds.

We perform our experiments on Intel Arria 10 GX FPGA. Intel(R) Xeon(R) Gold 6128 CPU with an operating frequency of 3.4 GHz is the host CPU. In both OpenCL and oneAPI, we explore the two(2) variants of the Sobel filter kernel. We begin our experiments with a baseline single-task (ST) implementation of the Sobel filter. Then, we evaluate the performance of the single-task version with fully unrolled loops (ST + LU). For OpenCL, we explore the impact of vectorization along with unrolling (ST + LU + V). In Verilog, we implement vectorization(V) by replicating compute-units by a factor of 16. This implementation generates 16 outputs pixels for every incoming stream of pixels.

1) Resource Utilization: The resource utilization report for our implementations is shown in the Table VI. We report the utilization for kernels compiled for 8K images. We observe that our Verilog implementation has higher register usage compared with all other implementations. Higher register usage in Verilog is primarily because of the difference in the memory model, where the Verilog implementation uses an optimized host-to-kernel pipeline from [23], [24].

2) Impact of Optimizations: Table V shows the performance of our Sobel filter implementations in Verilog, OpenCL, and OneAPI. For all three problem sizes, we observe that Verilog implementation is the fastest among the three. While the baseline single-task implementation is easier to develop, its performance is significantly lower than the optimized Verilog implementation (∼ 60,000× slower). After applying the optimizations discussed in §V, our OpenCL implementation
delivers competitive performance that is within 20% of the performance of Verilog for an 8K image. A notable observation from the Table V is that the gap between the performance of Verilog, OpenCL/OneAPI lessens as the problem size increases. Identifying the exact cause behind this decrease in the gap and remains a subject of future study.

C. Evaluation of Performance-Productivity Tradeoff Using II

Table VII shows the variation in $\Pi_{A \rightarrow B}$, where an 8K image is used to measure the performance of $A$ and $B$. Depending on the choice of implementations of the Sobel filter for $A$ and $B$, we get a wide range of values for $\Pi_{A \rightarrow B}$. For example, our Verilog implementation is $\approx 60,000 \times$ faster than the baseline OpenCL single-task implementation. Value of $\Delta T_{A \rightarrow B}$ for this combination is almost equal to 1. While OpenCL (ST) offers considerable improvement in productivity over Verilog, the degradation in performance is far too significant. As a result, the value of $\Pi$ for this combination is greater than one for all three values of $\alpha$ as shown in Table VII.

The value of $\Pi$ between Verilog and our optimized OpenCL implementation, OpenCL (ST + LU + V) ranges from 0.199 to 0.275 depending on the value of $\alpha$. In this case, the relative improvement in productivity is more significant than the relative loss in performance. While we do not come across a case in Table VII where $\Pi$ evaluates to zero, we do notice that for identical implementations of OpenCL and oneAPI, the value of $\Pi$ is closer to zero. This observation highlights that for equivalent kernel implementations, oneAPI provides very nearly the same performance as OpenCL, with significantly-improved productivity.

VII. FUTURE WORK

As a subject of future study, we intend to evaluate the performance-productivity tradeoffs for FPGA accelerators in a wide range of applications. More specifically, we intend to explore the performance-productivity gap for applications where irregular memory accesses and workload imbalance significantly limit the optimization scope.

VIII. CONCLUSION

This work has quantified the performance-programmability gap between Verilog, OpenCL, and oneAPI using a case study on Sobel filter. We proposed a new metric for the evaluation of tradeoffs between performance and productivity of FPGA programming models. While performance parity may be out of reach, we can still get within an order of magnitude of Verilog in terms of performance with OpenCL and oneAPI, and in some cases within 20%. In comparison with Verilog and OpenCL, we observed that the modern C++-based oneAPI was the most productive for development in terms of SLOC and development time. oneAPI required $\approx 4 \times$ less SLOC than Verilog and $1.5 \times$ less SLOC than OpenCL. In terms of development time in hours, oneAPI required $7.6 \times$ less time than Verilog and $1.25 \times$ less time than OpenCL. For implementations with identical optimizations, the productivity improvements due to oneAPI did not come at the cost of significant performance degradation as event-based kernel execution time for oneAPI and OpenCL kernel implementations was roughly the same.
REFERENCES