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, Π, 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, Π, to capture the performance-productivity tradeoff between FPGA programming abstractions.

The work detailed herein has been supported in part by NSF I/UCRC CNS-1822080 via the NSF Center for Space, High-performance, and Resilient Computing (SHREC). The authors would also like to thank Intel DevCloud for providing access to the Arria 10 FPGA.

```
h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i];
  ↪ });
```

(a) oneAPI Kernel for Vector Addition

```
__kernel void vector_add(__global int *x,
__global int *y,
__global int *restrict z)
{
    // get index of the work item
    int index = get_global_id(0);

    // add the vector elements
    z[index] = x[index] + y[index];
}
```

(b) OpenCL Kernel for Vector Addition

```
module vector_add
  (
    input   clk,
    input   rst,

    // CCI-P signals
    input   rx,
    output  tx
    );
    localparam [127:0] afu_id = 'AFU_ACCEL_UUID;
    // User registers (memory mapped to address h0040, h0042, and h0044)
    logic [63:0]   in0_r, in1_r, out_r;

    always_ff @(posedge clk or posedge rst)
      begin
        if (rst)
          begin
            out_r <= '0;
          end
        else
          begin
            out_r <= in0_r + in1_r;
          end
      end
    always_ff @(posedge clk or posedge rst)
      begin
      // Code for reading rx from memory mapped I/O

                  ⋮

    end
    always_ff @(posedge clk or posedge rst)
      begin
      // Code for writing tx (out_r) to memory mapped I/O

                  ⋮

    end
  end module
```

(c) The Verilog Kernel for Vector Addition

Fig. 1: Vector Addition Kernel in OneAPI, OpenCL, and Verilog listed in the order of decreasing productivity

- A case study on the design and implementation of the Sobel edge detection filter in oneAPI, OpenCL, and Verilog.
- A quantification of the productivity offered by oneAPI and OpenCL over Verilog.
- A set of optimizations in OpenCL, including manual vectorization, unrolling, and inference of shift registers, aimed at making the performance competitive with Verilog.

The rest of the paper is organized as follows. In §II, we present related work. In §III, we present our new metric, Π, to capture the performance-productivity tradeoff. In §IV, we describe our implementation of a Sobel filter. In §V, we articulate the HLS optimizations that we used to enhance the performance of our kernels. In §VI, we evaluate the productivity and performance of our Sobel filter. We discuss future work in §VII and conclude in §VIII.

II. Related Work

The concept of image gradient detection using an isotropic 3×3 operator (also known as the Sobel operator or Sobel filter) was introduced by Sobel et al. [11]. Since its introduction,

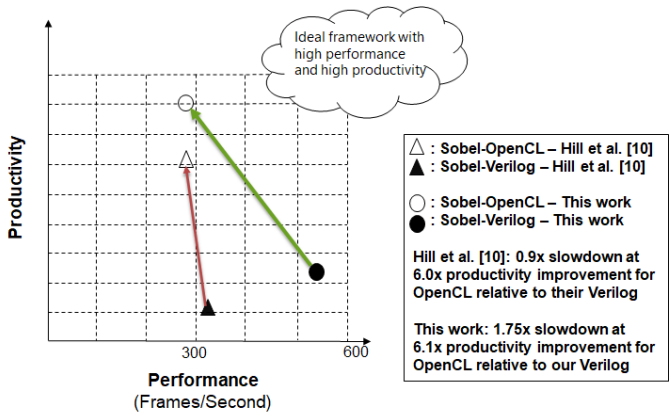


Fig. 2: Visualization of the Performance-Productivity Tradeoff using a Sobel Filter on an 1280 × 720 (HD) Image

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 sub-millisecond 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]. In §II-B, we discuss existing metrics that can are used to quantify the performance-portability-productivity tradeoffs between heterogeneous computing systems.

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 maintaining 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. further 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 performance 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, $\Pi_{A\rightarrow B}$, to quantify the tradeoff between the productivity and performance of language A vs. language B in §III-B.

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 programmers 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 accurate 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 programming language for the FPGA, we propose a metric, $\Pi_{A\rightarrow B}$, that captures the tradeoff between performance and productivity. We have formulated $\Pi_{A\rightarrow B}$ 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 productivity in B can be achieved without significant degradation in the performance compared to A. The value of $\Pi_{A\rightarrow B}$ 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 $\Pi_{A\rightarrow B}$ is considerably greater than zero in such as case.

With this rationale in mind, we define our metric, $\Pi_{A\rightarrow B}$, as follows:

$$\Pi_{A\rightarrow B} = \frac{\Delta T_{A\rightarrow B}}{\Delta P_{A\rightarrow B}} \tag{1}$$

where the numerator, $\Delta T_{A\rightarrow B}$, 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, $\Delta P_{A\rightarrow B}$, is the relative productivity improvement. It incorporates both SLOC and TDEV of language A

and language B. The numerator, $\Delta T_{A\rightarrow B}$, and the denominator, $\Delta P_{A\rightarrow B}$, from Equation (1) are given by the following equations, respectively.

$$\Delta T_{A\rightarrow B} = \frac{Throughput_A - Throughput_B}{Throughput_A} \tag{2}$$

$$\Delta P_{A\rightarrow B} = \alpha \left(\frac{SLOC_A - SLOC_B}{SLOC_A} \right) + (1-\alpha) \left(\frac{TDEV_A - TDEV_B}{TDEV_A} \right) \tag{3}$$

where $0 \leq \alpha \leq 1$

α and (1-α) in the Equation (3) are weights assigned to relative improvements in SLOC and TDEV, respectively. The value of α can be varied depending on the perceived significance of SLOC and TDEV metrics. In this work, we evaluate the value of π with the three values of α, (α=1), (α=0), and (α=0.5). Setting α as one discards the subjective metric TDEV altogether in favor of an objective SLOC comparison. Setting α as zero discards SLOC and accounts for the development time entirely. When α 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.

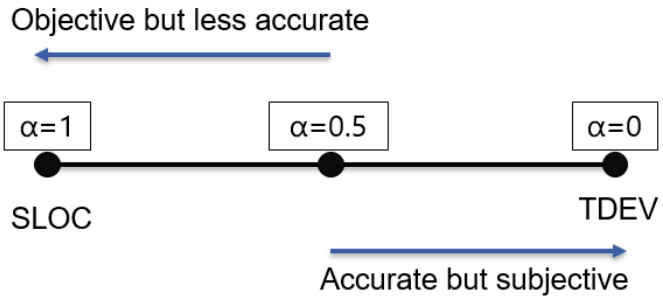


Fig. 3: The Spectrum of Productivity Metrics

C. Implications of $\Pi_{A\rightarrow B}$

We characterize $\Pi_{A\rightarrow B}$ across the following four regions of values: (1) $\Pi_{A\rightarrow B} > 1$, (2) $0 < \Pi_{A\rightarrow B} \leq 1$, (3) $\Pi_{A\rightarrow B} = 0$, and (4) $\Pi_{A\rightarrow B} < 0$.

1) $\Pi_{A\rightarrow B} > 1$*:* This implies that the relative difference in the performance of A and B is higher than the relative difference in the productivity of A and B. We deem this case as undesirable because the performance degradation is significant compared to productivity gains in going from language A to language B.

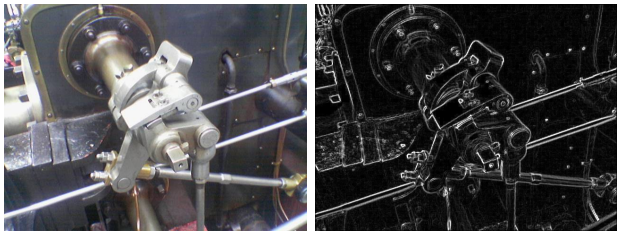
2) $0 < \Pi_{A\rightarrow B} <= 1$*:* A non-zero value of Π less than one implies that there is some performance cost associated with making a transition from a lower-level programming abstraction A to a higher one B.

3) $\Pi_{A\rightarrow B} = 0$*:* The implication here is that the productivity improvement of language B over language A does *not* come at the cost of any performance degradation.

4) $\Pi_{A\rightarrow B} < 0$*:* A negative value of $\Pi_{A\rightarrow B}$ implies that either $\Delta T_{A\rightarrow B}$ or $\Delta P_{A\rightarrow B}$ is negative. Both conditions are corner cases, and we do not come across these conditions in our evaluation.

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) Original RGB image [21] (b) Output of Sobel Filter

Fig. 4: An Example of Edge Detection Using a Sobel Filter

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.

Algorithm 1: Sobel Filter Using Sliding Window [22]

Input : $rgb\_pixels[height * width]$
Output: $sobel\_values[height * width]$
Data: $count, rows[2 * width + 3], gx[3][3], gy[3][3]$

$gx \leftarrow \{\{-1,-2,-1\},\{0,0,0\},\{1,2,1\}\}$
$gy \leftarrow \{\{-1,0,1\},\{-2,0,2\},\{-1,0,1\}\}$
$count \leftarrow -(2 * width + 3)$
while $count < height * width$ **do**
  for $i \leftarrow width * 2 + 2$ **to** 1 **do**
    $rows[i] \leftarrow rows[i-1]$
  end
  if $count \geq 0$ **then**
    $rows[0] \leftarrow rgbtoLuma(rgb\_pixels[count])$
  end
  $x\_dir, y\_dir \leftarrow 0$
  for $i \leftarrow 0$ **to** 2 **do**
    for $j \leftarrow 0$ **to** 2 **do**
      $x\_dir += rows[i * width + j] * gx[i][j]$
      $y\_dir += rows[i * width + j] * gy[i][j]$
    end
  end
  $sum = abs(x\_dir) + abs(y\_dir)$
  $sum = max(255, sum)$
  if $count \geq 0$ **then**
    $sobel\_values[count] \leftarrow sum$
  end
  $count \leftarrow count + 1$
end

TABLE I: Differences in the Tested Kernel Implementations

Language	Implementation		
	Memory Model	Loop Unrolling	Vectorization
Verilog	Stream mode	✓	✓
OpenCL	Batch mode	✓	✓
oneAPI	Batch mode	✓	×

✓: implemented, × : not implemented

We compute the value of Π 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[1] 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 3×3 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 $\Pi_{A \to B}$, 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.

TABLE II: Productivity Comparison between the Verilog, OpenCL and oneAPI Implementations of Sobel Filter

Language	Implementation	Kernel SLOC	Host SLOC	Total SLOC
Verilog	LU + V	298	131	429
OpenCL	ST	58	76	134
OpenCL	ST + LU	61	76	137
OpenCL	ST + LU + V	94	76	170
oneAPI	ST	60	27	87
oneAPI	ST + LU	63	27	90

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

TABLE III: Development Time in Hours for HLS and Verilog Implementations

	Time taken (Hours)			
Task	RTL	OpenCL	OneAPI	Status
Funtional Kernel development	85	5	2	Correctness tests passed in simulation
Correctness tests on FPGA	180	20	10	Verified the correctness of the kernel on FPGA Verified interactions between host and FPGA device.
Optimizations	40	25	8	Implemented optimizations for RTL and HLS
Total	305	50	20	

TABLE IV: Total Development Time (TDEV) for Tested Implementations

Implementation	TDEV (hours)
Verilog	305
OpenCL (ST)	20
OpenCL (ST + LU)	25
OpenCL (ST + LU + V)	50
oneAPI (ST)	12
oneAPI (ST + LU)	20

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 V: Sobel Filter: Performance Comparison between Verilog, OpenCL and oneAPI

Language	Implementation	Image size								
		HD (1280 × 720)			4K (3840 × 2160)			8K (7680 × 4320)		
		Event-based Kernel Runtime μS	Total time to solution μS	Throughput Frames/Sec.	Event-based Kernel Runtime μS	Total time to solution μS	Throughput Frames/Sec.	Event-based Kernel Runtime μS	Total time to solution μS	Throughput Frames/Sec.
Verilog	LU + V	-	1819	549.75	-	7540	132.62	-	34438	29.03
OpenCL	ST	33774130	33783121	0.029	262735483	262778854	0.003	2071574849	2071836869	0.00048
OpenCL	ST + LU	6486	12437	80.40	25760	45388	22.03	108257	180884	5.52
OpenCL	ST + LU +V	495	3452	289.68	1886	11670	85.68	6865	41340	24.189
oneAPI	ST	33504870	33513991	0.029	259019005	259036160	0.003	209378013	2097346330	0.00047
oneAPI	ST + LU	7202	15388	64.98	27402	46655	21.43	113759	183193	5.45

ST: Single-Task kernel, LU: Loop Unrolling, V:Vectorization

TABLE VI: Sobel Filter: Resource Usage Summary of 8k (7680 x 4320) Images

Language	Implementation	Logic Utilization	FFs / Registers	RAMs	Frequency (MHz)
Verilog	LU + V	73159 (18%)	187843	223 (9%)	292
OpenCL	ST	67407 (16%)	114308	389 (14%)	250
	ST + LU	66884 (16%)	113245	390 (14%)	320
	ST + LU + V	71526 (17%)	123600	398 (15%)	324
oneAPI	ST	80126 (20%)	116618	379 (14%)	246
	ST + LU	79626 (19%)	115613	381 (14%)	297

ST: Single-Task kernel, LU: Loop Unrolling, V:Vectorization

TABLE VII: Evaluation of Π for Various Implementations of Sobel Filter on an 8K Image

A	B	$\Pi_{A\to B}$ $\alpha = 1$	$\Pi_{A\to B}$ $\alpha = 0$	$\Pi_{A\to B}$ $\alpha = 0.5$
Verilog	OpenCL (ST)	1.454	1.070	1.233
Verilog	OpenCL (ST + LU)	1.190	0.881	1.013
Verilog	OpenCL (ST+ LU +V)	0.275	0.199	0.231
Verilog	oneAPI (ST)	1.253	1.041	1.137
Verilog	oneAPI (ST + LU)	1.028	0.869	0.942
OpenCL (ST)	oneAPI (ST)	0.057	0.050	0.053
OpenCL (ST +LU)	oneAPI (ST + LU)	0.035	0.060	0.044

ST: Single-Task Kernel, LU: Loop Unrolling, V:Vectorization

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 Π

Table VII shows the variation in $\Pi_{A\to B}$, 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 $\Pi_{A\to B}$. For example, our Verilog implementation is $\approx$60,000$\times$ faster than the baseline OpenCL single-task implementation. Value of $\Delta T_{A\to B}$ 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 Π for this combination is greater than one for all three values of α as shown in Table VII.

The value of Π between Verilog and our optimized OpenCL implementation, OpenCL (ST + LU + V) ranges from 0.199 to 0.275 depending on the value of α. 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 Π evaluates to zero, we do notice that for identical implementations of OpenCL and oneAPI, the value of Π 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 filer. 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

[1] A. Munshi, "The OpenCL Specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, 2009, pp. 1–314.

[2] Intel. (2021) Intel FPGA SDK for OpenCL. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[3] Xilinx. (2015) The Xilinx SDAccel Development Environment. [Online]. Available: https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[5] (2020) The sycl specification. Khronos Group. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[6] (2020) The oneapi specification. Intel. [Online]. Available: https://spec.oneapi.io/versions/latest/index.html

[7] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL™ Deep Learning Accelerator on Arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 55–64. [Online]. Available: https://doi.org/10.1145/3020078.3021738

[8] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 153–162. [Online]. Available: https://doi.org/10.1145/3174243.3174248

[9] X. Chen, R. Bajaj, Y. Chen, J. He, B. He, W.-F. Wong, and D. Chen, "On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-Based FPGAs," in *29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 67–73.

[10] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative Analysis of OpenCL vs. HDL with Image-Processing Kernels on Stratix-V FPGA," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015, pp. 189–193.

[11] I. Sobel, "An Isotropic 3x3 Image Gradient Operator," *Presentation at Stanford A.I. Project 1968*, 02 2014.

[12] M. Knap and P. Czarnul, "Performance Evaluation of Unified Memory with Prefetching and Oversubscription for Selected Parallel CUDA Applications on NVIDIA Pascal and Volta GPUs," *The Journal of Supercomputing*, vol. 75, 11 2019.

[13] N. Nausheen, A. Seal, P. Khanna, and S. Halder, "A FPGA based implementation of Sobel edge detection," *Microprocessors and Microsystems*, vol. 56, pp. 84–91, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933116302289

[14] J. Zhao, B. Xie, and X. Huang, "Real-time Lane Departure and Front Collision Warning System on an FPGA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–5.

[15] S. Pennycook, J. Sewall, and V. Lee, "Implications of a Metric for Performance Portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17300559

[16] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey, "Effective Performance Portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 24–36.

[17] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Navigating Performance, Portability, and Productivity," *Computing in Science Engineering*, vol. 23, no. 5, pp. 28–38, 2021.

[18] A. Funk, J. Kepner, V. Basili, and L. Hochstein, "A Relative Development Time Productivity Metric for HPC Systems," 05 2012.

[19] B. W. Boehm, "Software engineering economics," 1981.

[20] K. Krommydas, W. Feng, C. D. Antonopoulos, and N. Bellas, "OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures," *J. Signal Process. Syst.*, vol. 85, no. 3, p. 373–392, Dec. 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-1051-z

[21] S. contributor. (2021) Valve Original PNG Image. [Online]. Available: http://en.wikipedia.org/wiki/File:Valve_original_%281%29.PNG

[22] (2020) Sobel filter design example. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/sobel-filter.html

[23] (2020) Float Pipeline Intel Training Module. ARC Lab University of Florida. [Online]. Available: https://github.com/ARC-Lab-UF/intel-training-modules/blob/master/RTL/exercises/float_pipeline/solution/hw/afu.sv

[24] M. N. Emas, A. Baylis, and G. Stitt, "High-Frequency Absorption-FIFO Pipelining for Stratix 10 HyperFlex," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 97–100.

[25] *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*, Intel. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html

[26] *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*, Intel. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html

[27] (2020) Intel® oneAPI DPC++ FPGA Optimization Guide. Intel. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top.html

[28] Z. Jin and H. Finkel, "Optimizing an Atomics-Based Reduction Kernel on OpenCL FPGA Platform," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 532–539.