On the Three P's of Parallel Programming for Heterogeneous Computing: Performance, Productivity, and Portability

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

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

***Abstract*—As FPGAs and GPUs continue to make inroads into high-performance computing (HPC), the need for languages and frameworks that offer performance, productivity, and portability across heterogeneous platforms, such as FPGAs and GPUs, continues to grow. OpenCL and SYCL have emerged as frameworks that offer cross-platform *functional* portability between FPGAs and GPUs. While functional portability across a diverse set of platforms is an important feature of portable frameworks, achieving *performance* portability often requires vendor and platform-specific optimizations. Achieving performance portability, therefore, comes at the expense of productivity.**

This paper presents a quantification of the tradeoffs between performance, portability, and productivity of OpenCL and SYCL. It extends and complements our prior work on quantifying performance-productivity tradeoffs between Verilog and OpenCL for the FPGA. In addition to evaluating the performance-productivity tradeoffs between OpenCL and SYCL, this work quantifies the performance portability (PP) of OpenCL and SYCL as well as their code convergence (CC), i.e., a measure of productivity across different platforms (e.g., FPGA and GPU). Using two applications as case studies (i.e., edge detection using the Sobel filter, and graph link prediction using the Jaccard similarity index), we characterize the tradeoffs between performance, portability, and productivity. Our results show that OpenCL and SYCL offer complementary tradeoffs. While OpenCL delivers better performance portability than SYCL, SYCL offers better code convergence and a 1.6× improvement in source lines of code over OpenCL.

***Index Terms*—code convergence, FPGA, GPU, high-level synthesis (HLS), oneAPI, OpenCL, performance, portability, productivity, SLOC, SYCL, Verilog**

I. Introduction

In high-performance computing (HPC), there is an increasing need for easily programmable, highly performant, and naturally portable applications across different heterogeneous platforms (e.g., CPU, GPU, and FPGA). Over the past decade, the high-performance computing (HPC) community has witnessed the emergence of a cornucopia of heterogeneous computing frameworks and languages that offer *functional* portability[1] across a wide variety of heterogeneous platforms. Examples of such frameworks and languages include MetaMorph [1], Kokkos [2], RAJA [3], SYCL [4], OpenCL [5], and Chapel [6]. The above frameworks and languages generally offer functional portability across CPUs and GPUs only. For OpenCL and SYCL, however, parallel programs can be written and run on CPUs and GPUs as well as FPGAs. OpenCL, as a C-based high-level synthesis language, is supported by FPGA vendors and their associated toolchains, e.g., Intel's *FPGA SDK for OpenCL* [7] and Xilinx's *Vitis Unified Software Platform* [8]. For SYCL programs, Intel's data-parallel C++ (DPC++) compiler [9] can be used to target Intel CPUs, GPUs, and FPGAs and even NVIDIA GPUs.

While the aforementioned frameworks offer *functional* portability, achieving *performance* portability generally requires architecture-aware and vendor-specific optimizations. For example, our prior work [10] of a Sobel edge detection filter in OpenCL for an FPGA uses OpenCL's single-task kernel configuration [11] to exploit pipelined parallelism. In contrast, the OpenCL GPU implementation of the same application requires data-parallel designs using OpenCL's NDRange configuration [12] to achieve high performance. Thus, the transition from a baseline functionally portable implementation between an FPGA and GPU to a performance-portable implementation generally requires the introduction of platform-specific code. As a result, achieving performance portability across FPGAs and GPUs comes at the expense of the developer's productivity. In fact, Harrell et al. [13] refer to the introduction of platform-specific code as *code divergence* and propose a metric to evaluate it. Complementarily, Pennycook et al. use code convergence (CC) as a measure of similarity between programs that target a FPGA vs. a GPU [14].

Figure 1 shows examples of code divergence in OpenCL when targeting a GPU and an FPGA, respectively. When targeting a GPU (or CPU) platform, the OpenCL kernel can be built at runtime via just-in-time (JIT) compilation. Figure 1a

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.

[1] This is in contrast to *performance* portability, which we define as achieving a similar percentage of theoretical peak performance across two (or more) platforms.

shows such an example of building an OpenCL program object from source for a GPU (or CPU). When targeting an FPGA, a pre-compiled binary is typically used to build an OpenCL program object. Figure 1b shows such an example of using a pre-compiled binary to build an OpenCL program.

Figures 1c and 1d show vector addition kernels using OpenCL's NDRange and single-task configurations, respectively, and serve as examples of code divergence introduced due to performance considerations. Data-parallel designs are implemented with the NDRange configuration for the GPU. In contrast, pipelined designs and associated optimizations use the single-task configuration for the FPGA [15].

```
std::ifstream kernelFile(fileName, std::ios::in);
if (!kernelFile.is_open())
     {
     std::cerr << ”Failed to open file for reading: ” <<
      ↪ fileName << std::endl;
     return ;
     }
std::ostringstream oss;
oss << kernelFile.rdbuf();
std::string srcStdStr = oss.str();
const char *srcStr = srcStdStr.c_str();
program = clCreateProgramWithSource(context, 1,(const char**)&
   ↪ srcStr,NULL, NULL);

if (program == NULL)
     {
     std::cerr << ”Failed to create CL program from source.” <<
      ↪  std::endl;
     return ;
     }
clBuildProgram(program, 1, &device, NULL, NULL, NULL);
```

(a) GPU: Creating OpenCL program object from kernel

```
#define clBinaryProg(name) \
cl_program name; { \
FILE * f = fopen(#name”.aocx”, ”r”); \
fseek(f, 0, SEEK_END); \
size_t len = (size_t) ftell(f); \
const unsigned char * progSrc = (const unsigned char *) malloc
   ↪ (sizeof(char) * len); \
rewind(f); \
fread((void *) progSrc, len, 1, f); \
fclose(f); \
cl_int err; \
name = clCreateProgramWithBinary(context, 1, &device , &len, &
   ↪ progSrc, NULL, &err);}

clFinish(commandQueue);
clBinaryProg(program);
errNum=clBuildProgram(program, 1, &device, NULL, NULL, NULL);
```

(b) FPGA: Creating OpenCL program object from a pre-compiled binary

```
__kernel void kernel1(__global int* restrict device_input1,
__global int* restrict device_input2,
__global int* restrict device_output,
){
  int tid = get_global_id(0);
  device_output[tid]= device_input1[tid] + device_input2[tid]
}
```

(c) GPU: OpenCL vector add kernel with NDRange configuration

```
__kernel void kernel2(__global int* restrict device_input1,
__global int* restrict device_input2,
__global int* restrict device_output,
__gloabl int array_size)
){
  for (int i = 0; i< array_size; i++){
    device_output[tid]= device_input1[tid] + device_input2[tid
     ↪ ]
  }
}
```

(d) FPGA: OpenCL vector add kernel with single task configuration

Fig. 1: Examples of code divergence in OpenCL when targeting FPGAs and GPUs

From the example code listings in Figure 1, we observe that the development of performance-portable applications comes at the expense of higher code divergence and reduced productivity. While multiple studies exist that quantify the metrics of performance portability and code convergence for frameworks targeting CPUs and GPUs, a similar study on the rigorous quantification of the aforementioned metrics for FPGAs and GPUs has never been conducted.

Specifically, this paper quantifies the tradeoffs between the performance, portability, and productivity of OpenCL and SYCL on FPGA and GPU. It extends and complements our prior work on the characterization of the performance-productivity tradeoff for FPGAs [10]. In addition to measuring the performance-productivity tradeoffs between FPGA programming languages (i.e., Verilog and OpenCL), this paper explores the performance portability (PP) of OpenCL and SYCL as well as their code convergence (CC), i.e., a measure of productivity across different platforms (e.g., FPGA and GPU). Using two applications as case studies — (1) edge detection using the Sobel filter and (2) graph link prediction using the Jaccard similarity index — we characterize the tradeoffs between performance, portability, and productivity. In all, this paper makes the following contributions:

- Quantification and analysis of the performance-productivity tradeoffs between OpenCL and SYCL on GPU and FPGA.
- Quantification and analysis of productivity improvements of SYCL over OpenCL.
- Quantification and analysis of performance portability and code convergence [13], [14], [16] of OpenCL and SYCL.

The rest of the paper is organized as follows. §II highlights the related work on quantifying tradeoffs between performance, productivity, and portability. In §III, we present the metrics used to evaluate performance portability and productivity. In §IV, we discuss the implementations of our target applications (in short, Sobel filter and Jaccard similarity). §V presents the evaluation of performance, productivity, and portability metrics. We discuss future work in §VI and conclude in §VII.

II. Related Work

We present related work in three parts: (1) metrics for tradeoffs between performance, portability, and productivity, (2) prior work on our target applications, i.e., Sobel filter and Jaccard similarity, and (3) existing studies on performance portability.

A. Metrics for tradeoffs between performance, portability, and productivity

First, Pennycook et al. define performance portability as the harmonic mean of an application's performance efficiencies observed across a set of platforms [16]. Second, Harell et al. define metrics to quantify productivity in terms of code divergence, maintenance cost, and development cost platforms [17]. Third, Pennycook et al. incorporate the code divergence metric from [17] and expand upon their performance-portability evaluation [18].

Figure 2 shows how Pennycook et al. visualize the performance portability and code convergence of a particular code [18]. An ideal language is expected to have theoretical maximum values of one for performance portability and code convergence, but it is *not* realistic for a language to deliver both high performance portability and code convergence. Oftentimes, the use of platform-specific code is necessary to achieve high performance portability. However, while platform-specific code can improve performance portability, it comes at the expense of code convergence, as shown in Figure 2. In this work, we evaluate the performance portability of the OpenCL and SYCL implementations of a Sobel filter and code convergence for the OpenCL and SYCL implementations of a Sobel filter and Jaccard similarity.

In our prior work, we introduced the *performance productivity product* (Π) to evaluate performance-productivity tradeoffs between FPGA programming languages (e.g., Verilog) and high-level synthesis languages (e.g., OpenCL) [10]. This paper is a complementary extension of this prior work. Rather than focus on the Π metric of an FPGA, we more broadly evaluate performance, productivity, and portability metrics (and combinations thereof) on both the GPU and FPGA.

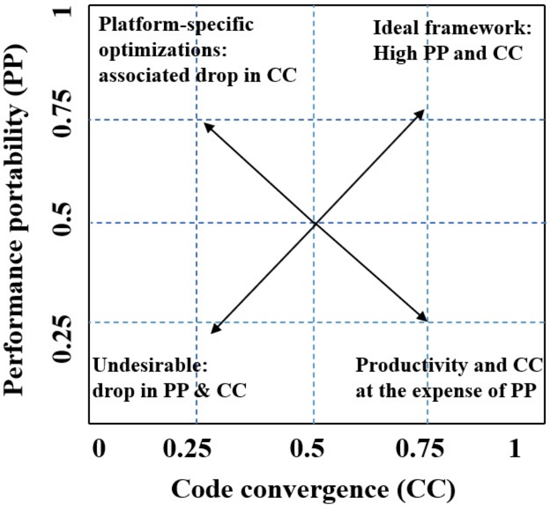

Platform-specific optimizations: associated drop in CC
Ideal framework: High PP and CC
Undesirable: drop in PP & CC
Productivity and CC at the expense of PP
Performance portability (PP)
0.25
0.5
0.75
1
0
0.25
0.5
0.75
1
Code convergence (CC)

Fig. 2: Visualization of performance portability vs. code convergence for portable languages and frameworks [18]

B. Prior work on target applications

We realize and evaluate two applications in this paper: (1) image edge detection using the Sobel filter and (2) link prediction in graph datasets using Jaccard similarity, hereinafter referred to as Sobel filter and Jaccard similarity for brevity.

1) Sobel filter: Since its introduction by Sobel [19], the Sobel filter has been optimized for both GPUs [12] and FPGAs [20], [21]. In our prior work, we quantified the performance-productivity tradeoffs between Verilog and OpenCL using the Sobel filter [10]. We complement and extend our prior work by including FPGAs *and* GPUs in our analysis as well as adding a more complex and irregular application to the mix, i.e., Jaccard similarity.

2) Jaccard similarity: In graph analytics, computing the intersection of neighborhood sets is a widely explored problem [22], [23]. In this work, we evaluate an instance of the set intersection problem to compute link prediction in graph datasets. Link prediction in graph datasets can be evaluated using a metric called *Jaccard similarity* (JS) [24]. Our analysis uses the edge-centric implementation of Jaccard similarity, introduced in [23].

C. Performance portability studies

We use the aforementioned target applications to evaluate performance portability and related metrics. Similar studies have been conducted for a wide variety of languages and frameworks. For example, Deakin et al. present a rigorous quantification of performance portability of OpenMP, Kokkos, CUDA, and OpenACC [25]. Deakin et al. extend their work by including SYCL and more applications in the analysis of performance portability [26]. None of the above studies include FPGAs in their evaluation.

III. Metrics for Performance, Productivity, and Portability

This section describes the metrics used in our evaluation.

A. Source lines of code (SLOC)

We use source lines of code (SLOC) as a baseline measure of productivity when writing programs in OpenCL and SYCL.

B. Performance-productivity product (Π)

As defined in [10], for a low-level language A and a high-level language B, $\Pi_{A\to B}$ evaluates the performance-productivity tradeoff for a transition from A to B. The metric $\Pi_{A\to B}$ is formulated such that it evaluates to zero in the ideal case, i.e., no difference. $\Pi_{A\to B}$, is defined as shown below:

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

where $\Delta T_{A\to B}$ evaluates the relative difference in the performance on a given platform when transitioning from a low-level language A to a high-level language B and where $\Delta P_{A\to B}$, evaluates the relative productivity improvement by incorporating both source lines of code (SLOC) and total development time (TDEV) of language A and language B.

$\Delta T_{A\to B}$ and $\Delta P_{A\to B}$, from Equation (1) are defined below.

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

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

where $0 \leq \alpha \leq 1$

We use the terms α and (1-α) to assign weights to the SLOC and TDEV metrics. The value of α can be varied between [0, 1], depending on the perceived importance of SLOC and

TDEV metrics. In this work, we evaluate $\Pi_{A \rightarrow B}$ by setting α to one (1). $\Pi_{A \rightarrow B}$ is designed such that it rewards a transition from a low-level framework to a high-level framework if the associated productivity improvement does not come at the cost of performance degradation. In contrast, the metric penalizes the same transition if productivity gains are not significant in comparison with performance degradation.

C. Composite metric for performance portability

For an application a, solving a problem p on a given set of platforms H, Pennycook et al. [16] define performance portability, as shown in Equation (4).

$$\Phi(a, p, H) = \begin{cases} \dfrac{|H|}{\displaystyle\sum_{i \in H} \dfrac{1}{e_i(a, p)}}, & \text{if } i \text{ is supported } \forall i \in H. \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

$|H|$ is the total number of platforms and $e_i(a, p)$ is the performance efficiency of application a on a platform i. It can be observed from Equation (4) that performance portability is defined as the harmonic mean of performance efficiencies on all supported platforms. Performance portability evaluates to zero if an application is not supported on any one (or more) of the target platforms. Pennycook et al. provide two definitions of performance efficiency: architectural efficiency and application efficiency [16]. Architectural efficiency is defined as achieved performance as a fraction of peak theoretical performance on a given platform . Application efficiency is measured as a fraction of performance relative to the best-observed performance on a given platform. In this paper, we use *application efficiency* to evaluate the performance portability for OpenCL and SYCL, as shown in Equation (4). The ideal value of performance portability is one (1), Φ (a, p, H) evaluates to 1 when the performance efficiencies across all platforms are equal to one, indicating that the framework achieves maximum efficiency on all target platforms.

D. Code convergence

Code convergence is a measure of similarity between portable programs targeting multiple platforms such as FPGAs and GPUs. Pennycook et al. [14] define code convergence as "1− code divergence" and evaluate code divergence, as defined by Harrell et al. [13]. Code divergence is defined as the average Jaccard distance between each pair of platforms [13]. For two platforms N and M, code divergence (CD) can be defined as shown in Equation (5), where $SLOC_i$ is the number of source lines of code when targeting platform i. In this paper, we compute code convergence (CC) as shown in Equation (6).

$$\mathbf{CD} = \frac{|\mathbf{SLOC_N} \cup \mathbf{SLOC_M}| - |\mathbf{SLOC_N} \cap \mathbf{SLOC_M}|}{|\mathbf{SLOC_N} \cup \mathbf{SLOC_M}|} \tag{5}$$

$$\mathbf{CC} = \frac{|\mathbf{SLOC_N} \cap \mathbf{SLOC_M}|}{|\mathbf{SLOC_N} \cup \mathbf{SLOC_M}|} \tag{6}$$

The ideal value of CC is 1, $CC = 1$ implies that the same code can be run on all target platforms without any platform-specific updates.

IV. Case Studies

This section describes the target applications — Jaccard similarity and Sobel filter — and their implementations in OpenCL and SYCL.

A. Jaccard similarity

In graph analytics, Jaccard similarity is used to measure link prediction between two or more vertex pairs. More generally, Jaccard similarity between any two sets, $\mathbf{A}$ and $\mathbf{B}$, is defined as shown in Equation (7).

$$\textbf{Jaccard similarity}(\mathbf{A}, \mathbf{B}) = \frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A} \cup \mathbf{B}|} \tag{7}$$

In graph datasets, Jaccard similarity for any two vertices is computed using the respective neighborhood sets of the vertices. Algorithm 1 describes our implementation of Jaccard similarity. Similar to prior work on Jaccard similarity for FPGA [23], we use binary search to compute the size of the intersection and union of neighborhood sets.

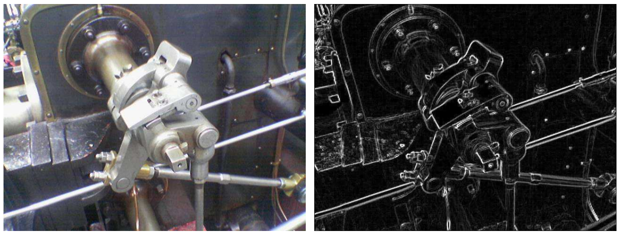
Algorithm 1: Edge-centric implementation of Jaccard similarity [23]

```
Input  : graph G(V, E) in edge-list format:
         source($|E|$), dest($|E|$),
         offsets($|V|$)
Output: Jaccard($|E|$)
Data: $|E|$
foreach edge $e$ from $E$ do
    s = source[e], d = dest[e]
    // Count neighbors
    $N_s$ = offsets[s+1]- offsets[s]
    $N_d$ = offsets[d+1]- offsets[d]
    if $N_s < N_d$ then
        ref = s, cur = d
    end
    else
        ref = d, cur = s
    end
    foreach destination $i$ from $ref$ do
        refCol = dest[i]
        // Binary search
        is_present = binary_search(refCol,
          neighbors(cur))
        if (is_present)
           Jaccard[e] += 1
        Jaccard[e] = Jaccard[e] / (($N_s$ + $N_d$) -
          Jaccard[e]);
    end
end
```

Implementation: Our implementation of Jaccard similarity in OpenCL and SYCL exploits edge-centric parallelism. Each work-item computes the Jaccard score for a unique vertex pair that forms an edge. Unlike the Sobel filter, we do not implement any platform-specific optimizations. Instead, we evaluate the performance of similar data-parallel designs in OpenCL and SYCL on both FPGA and GPU.

B. Sobel filter

We evaluate an integer variation of the Sobel filter [10]. Figure 3 shows an example of the Sobel filter on an RGB image from [27]. A 3×3 filter is used to compute gradients along the X and Y axes of the image. Algorithm 2 describes our baseline implementation of the Sobel filter.



(a) RGB image [27]

(b) Sobel Filter output

Fig. 3: Example of Sobel edge detection filter on an RGB image

Implementation: The OpenCL implementation for FPGA uses manual vectorization and loop unrolling in OpenCL's single-task configuration [15]. The SYCL implementation targeting FPGAs includes loop unrolling in a single-task kernel.

The GPU implementations in OpenCL and SYCL exploit data-level parallelism via NDRange and parallel-for kernel launch configurations in OpenCL and SYCL, respectively. To compute the application efficiency of OpenCL and SYCL on the FPGA, we use the performance of a Verilog implementation of the Sobel filter as our reference [10]. To compute the application efficiency of OpenCL and SYCL on the GPU, we designed a CUDA implementation of Algorithm 2 and used its performance as the best-observed performance on the GPU. Table I describes our OpenCL and SYCL implementations.

TABLE I: Implementations of target applications

Application	Language	FPGA		GPU	
		Implemenation	Description	Implemenation	Description
Sobel filter	OpenCL	NDRange	Data-parallel design	NDRange	Data-parallel design
		ST + LU [10]	Single task with unrolling		
		ST + LU + V [10]	Single task with unrolling and vectorization		
	SYCL	Parallel for	Data-parallel design	Parallel for	Data-parallel design
		ST + LU [10]	Single task with unrolling		
Jaccard similarity	OpenCL	NDRange	Data-parallel design	NDRange	Data-parallel design
	SYCL	NDRange	Data-parallel design	NDRange	Data-parallel design

ST: single-task kernel, LU: loop unrolling, V: vectorization

```
Algorithm 2: Sobel filter using 3×3 kernels
Input  : rgb_image[height*width]
Output: filtered_image[height*width]
Data: r, g, b, gx[3][3], gy[3][3]
$gx \leftarrow \{\{-1,-2,-1\},\{0,0,0\},\{1,2,1\}\}$
$gy \leftarrow \{\{-1,0,1\},\{-2,0,2\},\{-1,0,1\}\}$
for $k \leftarrow 0$ to $height * width - 1$ do
    x_grad, y_grad $\leftarrow 0$
    for $i \leftarrow 0$ to $2$ do
        for $j \leftarrow 0$ to $2$ do
            pixel = rgb_image[k + ((i-1)*width) + (j-1)]
            b = pixel & 0xff
            g = (pixel >> 8) & 0xff
            r = (pixel >> 16) & 0xff
            luma $\leftarrow rgb\_to\_Luma(r, g, b)$
            x_grad += luma * gx[i][j]
            y_grad += luma * gy[i][j]
        end
    end
    sum = abs(x_grad) + abs(y_grad)
    sum = min(255, sum)
    filtered_image[k] $\leftarrow sum$
end
```

V. Evaluation

This section presents the evaluation of our target applications and metrics discussed in §III. Relative to hardware, we evaluated the applications on the following hardware: Intel Arria 10 GX FPGA and NVIDIA RTX 3090 GPU. Intel's DPC++ compiler [9] was used to compile the SYCL implementations on target platforms.

A. Sobel filter

Tables II and III show the evaluation of performance portability (Φ) and code convergence (CC) for Sobel filter. We computed the application efficiency for OpenCL and SYCL on the Arria 10, relative to the performance of our optimized Verilog implementation from [10]. It is a stream-oriented implementation with the Sobel gradients along the X and Y axes being computed as the data is streamed from the host CPU to the FPGA. To ensure that the performance comparisons are consistent with our implementation of Sobel filter in Verilog, we used the total time to solution, including the time to transfer the data to and from the platform along with the kernel runtime. The total time to solution was used to compute the application efficiency for Sobel filter implementations. CUDA delivered the best-observed performance from our implementations on the RTX 3090 GPU, and therefore, we computed application efficiencies on the RTX 3090 relative to the CUDA performance. Similar to the visualizations presented by Pennycook et al. [18], we plot the Φ and CC of OpenCL in Figure 4. The NDRange kernel without any FPGA-specific optimizations offers the highest CC but poor Φ. As we introduce FPGA-specific optimizations,

TABLE II: Performance portability (Ⴔ) evaluation for Sobel filter on 8K (4320 × 7680) image

	FPGA (Intel Arria 10)				GPU NVIDIA RTX 3090				Application efficiency (Arria 10)	Application efficiency RTX 3090)	Performance portability (Ⴔ) (Arria 10, RTX 3090)
Language	Implementation	Kernel runtime (ms)	Total time to solution: kernel + data transfer time (ms)	**Throughput (frames/sec)**	Implementation	Kernel runtime (ms)	Total time to solution: kernel + data transfer time (ms)	**Throughput (frames/sec)**			
Verilog	LU + V [10]	-	34.43	**29.03**	**not portable on GPU**				1	not portable	**0**
CUDA	**not portable on FPGA**				CUDA equivalent of NDRange	0.34	17.77	**56.27**	not portable	1	**0**
OpenCL	NDRange	128.34	152.70	**6.54**	NDRange	0.35	22.26	**44.92**	0.22	0.79	**0.34**
	ST + LU [10]	108.25	180.88	**5.52**					0.19		**0.31**
	ST + LU + V [10]	6.86	41.34	**24.19**					0.83		**0.81**
SYCL	Parallel for	135.13	170.23	**5.87**	Parallel for	0.37	23.65	**42.28**	0.20	0.75	**0.31**
	ST + LU [10]	113.75	183.19	**5.45**					0.18		**0.29**

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

TABLE III: Code convergence and SLOC evaluation for Sobel filter

	FPGA (Intel Arria 10)		GPU (NVIDIA RTX 3090)		Common SLOC FPGA ∩ GPU	Total SLOC FPGA ∪ GPU	Code convergence (Arria 10, RTX 3090)
Language	Implementation	SLOC	Implementation	SLOC			
Verilog	UR +V [10]	429	**not portable on GPU**		**not portable on GPU**		
CUDA	**not portable on FPGA**		CUDA equivalent of NDRange	141	**not portable on FPGA**		
OpenCL	NDRange	257	NDRange	254	239	272	**0.87**
	ST + LU [10]	267			228	293	**0.77**
	ST + LU + V [10]	328			208	374	**0.55**
SYCL	Parallel for	135	Parallel for	135	133	137	**0.97**
	ST + LU [10]	139			128	142	**0.90**

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

we achieve higher Ⴔ, but it comes at the expense of reduced CC. Table IV shows the performance-productivity product (Π) for the OpenCL and SYCL implementations of our target applications. For the Sobel filter implementations with identical levels of optimizations, the transition from OpenCL to SYCL does not come at the expense of significant performance loss.

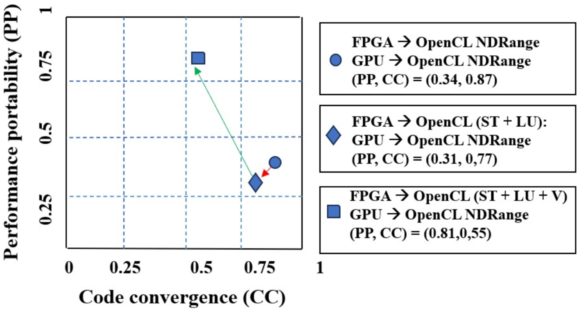


Fig. 4: Sobel filter: impact of platform-specific optimizations on the Ⴔ and CC of OpenCL

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

TABLE IV: $\Pi_{A\rightarrow B}$ evaluation for transitions from OpenCL to SYCL

Application	Platform	Implementation A	Implementation B	$\Pi_{A\rightarrow B}$ $\alpha = 1$
Sobel filter	Arria 10	OpenCL (ST + LU + V)	SYCL (ST + LU)	1.34
		OpenCL (ST +LU)	SYCL (ST +LU)	0.02
	RTX 3090	OpenCL (NDRange)	SYCL (Parallel for)	0.12
Jaccard similariy	Arria 10	OpenCL (NDRange)	SYCL (NDRange)	0.87
	RTX 3090	OpenCL (NDRange)	SYCL (NDRange)	0.76

Lower the value of $\Pi_{A\rightarrow B}$ the better
ST: Single-Task kernel, LU: Loop Unrolling, V:Vectorization

TABLE V: SLOC and code convergence for OpenCL and SYCL NDRange implementations of Jaccard similarity

Language	SLOC		Common SLOC	Total SLOC	Code convergence
	SLOC_{FPGA}	SLOC_{GPU}	$\text{SLOC}_{FPGA\ \cap\ GPU}$	$\text{SLOC}_{FPGA\ \cup\ GPU}$	CC
OpenCL	915	915	898	932	0.96
SYCL	721	721	720	722	0.99

B. Jaccard similarity

Tables V and VI present the evaluation of code convergence and performance of Jaccard similarity implementations, respectively. The RTX 3090 GPU outperforming the Xilinx Arria 10 FPGA is expected as we did not implement any FPGA-

TABLE VI: Performance evaluation for Jaccard similarity. Input graph: California road network from [28] with 2 million vertices and 5.6 million bidirectional edges

	FPGA (Intel Arria 10)		GPU (NVIDIA RTX 3090)	
Language	Implementation	Kernel runtime (ms)	Implementation	Kernel runtime (ms)
CUDA	Not portable on FPGA		CUDA equivalent of NDRange	0.25
OpenCL	NDRange	262.44	NDRange	0.26
SYCL	NDRange	322.15	NDrange	0.31

specific optimizations (yet). We observe that OpenCL offers better performance on both platforms compared to SYCL while SYCL offers higher code convergence and productivity.

Our OpenCL and SYCL implementations offer complementary tradeoffs, with OpenCL delivering higher Φ compared to SYCL and SYCL offering better code convergence and average 1.6× improvement in productivity in terms of SLOC.

VI. Future Work

Future work in this area can include the evaluation of performance portability and code convergence of applications that utilize multiple GPUs and/or FPGAs. We intend to incorporate more languages and frameworks, such as Kokkos, RAJA, Chapel, OpenMP, and OpenACC, to further evaluate performance portability and code convergence.

VII. Conclusion

This work quantifies the metrics for performance portability, code convergence, and performance-productivity tradeoffs when targeting FPGAs and GPUs. The platform-agnostic kernels from our examples achieve poor performance portability but have higher code convergence than kernels that incorporate platform-specific optimizations. With the case study on the Sobel filter, we show that achieving performance portability requires platform-specific optimizations. In terms of source lines of code (SLOC), we observe that developing target applications in SYCL is 1.6× more productive than OpenCL. SYCL offers better code convergence than OpenCL but the higher code convergence and productivity benefits come at the cost of performance portability.

References

[1] A. Helal, P. Sathre, and W. Feng, "MetaMorph: A Library Framework for Interoperable Kernels on Multi- and Many-core Clusters," in *SC'16: IEEE/ACM Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, Utah, USA, Nov. 2016.

[2] H. Edwards, C. Trott, and D. Sunderland, "Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns," *J. Parallel & Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[3] D. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. Kunen, O. Pearce, P. Robinson, B. Ryujin, and T. Scogland, "RAJA: Portable Performance for Large-Scale Scientific Applications," in *IEEE/ACM Int'l Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[4] The SyCL Specification. Khronos Group. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

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

[6] *The Chapel Parallel Programming Language*. [Online]. Available: https://chapel-lang.org/

[7] Intel. (2023) Intel FPGA SDK for OpenCL. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683846/22-4/overview.html

[8] Xilinx. (2023) Vitis Unified Software Platform. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[9] The oneAPI DPC++ Compiler documentation. Intel. [Online]. Available: https://intel.github.io/llvm-docs/index.html

[10] A. Gondhalekar, T. Twomey, and W. Feng, "On the Characterization of the Performance-Productivity Gap for FPGA," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–8.

[11] *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*, Intel. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683846/22-4/eol.html

[12] T. Sanida, A. Sideris, and M. Dasygenis, "A Heterogeneous Implementation of the Sobel Edge Detection Filter Using OpenCL," in *9th Int'l Conf. on Modern Circuits and Systems Technologies*, 2020, pp. 1–4.

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

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

[15] *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*, Intel. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683521/22-4/eol.html

[16] 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

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

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

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

[20] 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

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

[22] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static Graph Challenge: Subgraph Isomorphism," in *IEEE High Performance Extreme Computing Conference (HPEC)*, Sept. 2017. [Online]. Available: https://doi.org/10.1109%2Fhpec.2017.8091039

[23] P. Sathre, A. Gondhalekar, and W. Feng, "Edge-Connected Jaccard Similarity for Graph Link Prediction on FPGA," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–10.

[24] P. Jaccard, "The Distribution of the Flora in the Alpine Zone.1," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912. [Online]. Available: https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x

[25] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance Portability across Diverse Computer Architectures," in *IEEE/ACM Int'l Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*, 2019, pp. 1–13.

[26] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking Performance Portability on the Yellow Brick Road to Exascale," in *IEEE/ACM Int'l Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*, 2020, pp. 1–13.

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

[28] R. Rossi and N. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com