# Comparative Analysis of Executing GPU Applications on FPGA: HLS vs. Soft GPU Approaches

Chihyo Ahn
Georgia Institute of Technology
Atlanta, USA
ahnch@gatech.edu

Shinnung Jeong
Yonsei University
Seoul, Republic of Korea
shin0403@yonsei.ac.kr

Liam Paul Cooper
Georgia Institute of Technology
Atlanta, USA
lpc@gatech.edu

Nicholas Parnenzini
Georgia Institute of Technology
Atlanta, USA
nparnenzini3@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
Atlanta, USA
hyesoon@cc.gatech.edu

***Abstract*—With the development of the GPU, parallel languages are widely used for developing modern parallel applications. Given its low energy cost and programmable hardware, the FPGA emerges as a promising candidate to run GPU applications. Therefore, executing applications described in GPU programming languages on FPGA can offer new opportunities in terms of performance and energy efficiency. However, the gap between GPU programming languages and hardware description languages (HDL) poses a significant challenge for this transition. To overcome this problem, existing works have attempted to bridge this gap through high-level synthesis (HLS) or soft GPU. In this paper, we examine how HLS and soft GPU compile GPU languages for FPGA by discussing the detailed compilation and execution flow of two representative works: Intel FPGA SDK for OpenCL and Vortex. This paper also evaluates the coverage of both approaches and discusses methods for addressing the challenges each approach faces. Consequently, this paper explores the challenges HLS and GPU encounter, aiming to identify new problems and opportunities each approach introduces.**

## I. Introduction

The importance of running GPU applications efficiently has been increasing. With their low energy cost and programmable hardware, FPGAs are promising candidates for running GPU applications. However, FPGAs are traditionally programmed with hardware description languages (HDL), which require hardware expertise to synthesize and execute the desired applications. To improve the programmability of running GPU applications on FPGAs, several efforts have been made to raise the abstraction level. The first approach involves using high-level synthesis (HLS). HLS allows users to synthesize digital circuits by writing in software programming languages (e.g., C++). Major FPGA manufacturers support OpenCL as an input to their HLS pipeline. Another method to execute GPU programs on FPGAs without modifying the source code involves employing a GPU implemented as a softcore processor (soft GPU). This approach differs from HLS, which targets the implementation of specific OpenCL kernels. Instead, a soft GPU uses an FPGA as a substrate to implement the GPU architecture. Unlike HLS, a soft GPU runs a GPU application binary on a GPGPU, allowing for a more flexible flow of execution.

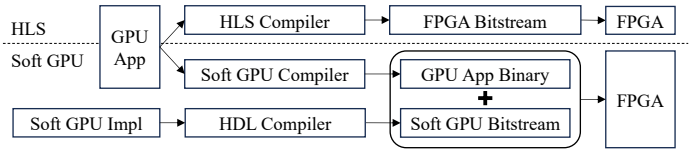



Fig. 1. Running a GPU application on an FPGA using HLS and soft GPU

The overview of the two different compilation and execution pipelines is shown in Figure 1. The first approach involves an HLS compiler that compiles OpenCL kernels into Verilog HDL for synthesis. The second approach, the soft GPU, begins with synthesizing the soft GPU bitstream using an HDL compiler. Next, the GPU application source code is compiled into a binary using a soft GPU compiler to be executed on the FPGA. Using HLS to synthesize the kernel code, or utilizing a soft GPU, are two approaches that require less hardware expertise from users. Neither method requires modification of the original GPU-friendly source code. However, each approach exhibits different strengths in terms of coverage and performance, as well as different challenges to overcome for complete support. In this work, we characterize the distinct characteristics of using HLS (Intel FPGA SDK for OpenCL) and a soft GPU (Vortex [1], [2], [3]) for running the parallel language (OpenCL), and the challenges of each approach for fluent support.

### A. Motivation

Although several studies have explored various methods of executing GPU code on FPGAs [4], [5], [6], [7], [8], few have directly compared the characteristics and challenges of

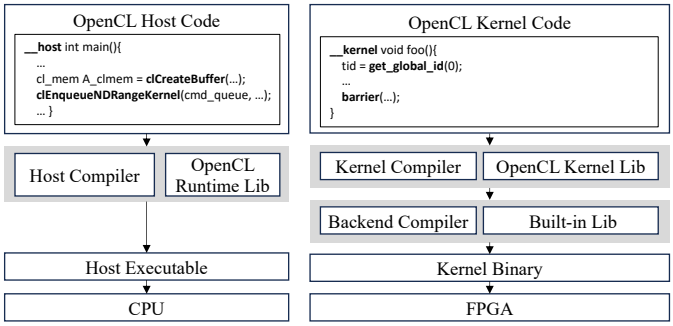



Fig. 2. General compilation flow for executing OpenCL program on FPGA

soft GPU implementations versus the HLS approach. Particularly, when it comes to supporting the diverse features of parallel programming languages, each method introduces unique hardware and software challenges that are not widely documented. For instance, implementing a software stack for a soft GPU entails significant effort to support parallel language features, such as atomic instructions, due to the multi-layered complexity of the software stack for soft GPUs. By comparing the current state and challenges of these two approaches using identical source code, we aim to provide a deeper understanding and facilitate broader support and advancement in this area.

In this paper, we compare HLS versus a soft GPU for executing GPU applications. Given that the performance of both platforms heavily relies on the quality of HLS compiler optimizations and the GPU softcore, this paper primarily concentrates on examining their coverage and challenges.

This paper makes the following key contributions:

- We compare the execution of GPU applications on FPGAs between soft GPU and HLS approaches in terms of coverage and performance.
- We identify and discuss the challenges of supporting a wide range of applications on an FPGA using both HLS and soft GPU approaches.

## II. Background

### A. Running OpenCL on FPGA

To execute OpenCL applications on FPGAs, existing works [6], [9], [10], [11], [7], [4], [5], [12] often adopt two approaches: high-level synthesis (HLS) or soft graphics processing units (soft GPUs). High-level synthesis converts from an abstract behavioral specification of a digital system to a register-transfer level (RTL) structure that implements that behavior [13]. Existing works utilizing HLS [6], [9], [10], [11], [7] start from OpenCL code and generate hardware description language (HDL) code, such as Verilog. A soft GPU is a GPU-like parallel processor on an FPGA, offering FPGA compute power in a very flexible, GPU-like tool flow [5]. Previous soft GPU works [4], [5], [12] employ OpenCL as a front-end language and support the compiler process to generate binaries consisting of their own instruction set architecture (ISA).

Both approaches use host and kernel compilers to generate program binaries for deploying programs on FPGAs, as shown in Figure 2. First, OpenCL host codes are compiled into a host executable by linking OpenCL communication functions, such as kernel launch and memory copy operations. Second, OpenCL kernel codes undergo a two-step compilation process to be transformed into kernel binaries. In the first step, the kernel compiler employs the OpenCL Kernel Library to lower OpenCL kernel functions to access kernel information and manage program flow. Subsequently, the back-end compiler generates an FPGA bitstream with a predefined compute unit (HLS) or produces a kernel executable compatible with the soft GPU ISA.

During the compilation process, the most crucial consideration is the effective deployment of the programmer's parallel tasks onto the FPGA while achieving low latency and energy efficiency. Although both the HLS compiler and the soft GPU compiler transform code into binaries through a similar high-level process, they diverge in their approaches during the kernel transformation phase. The HLS compiler primarily focuses on pipelining and pipeline duplication, whereas the soft GPU compiler emphasizes work distribution and ISA generation. An example of a widely accepted HLS compiler is the Intel FPGA SDK for OpenCL, which includes the AOC compiler [14]. This compiler supports the compilation of OpenCL source code, abstracting hardware details through HLS and synthesizing the corresponding FPGA bitstream. Vortex [1], [2], [3] represents another significant soft GPU work that supports OpenCL. This paper analyzes the coverage and challenges presented by HLS and soft GPU through these two representative examples.

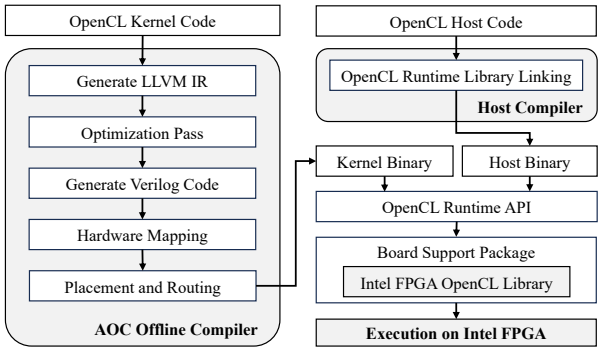



Fig. 3. Intel HLS for OpenCL compilation pipeline

### B. Intel FPGA SDK for OpenCL

The Intel FPGA SDK for OpenCL [14] offers a highly abstracted programming model, sparing users the need to delve into hardware specifics. The key components of the compilation pipeline are encapsulated within the SDK's AOC compiler. Figure 3 illustrates the Intel FPGA SDK's compilation pipeline. Initially, the AOC (kernel) compiler transforms kernel source code into LLVM IR, which is then subjected to LLVM optimization passes. Subsequently, this

intermediate representation is converted to RTL (Verilog), leading to hardware mapping and the placement/routing process that generates the final FPGA bitstream. The compilation of host code does not diverge from the standard OpenCL code compilation process (using GCC/Clang). However, at runtime, the linked Intel FPGA OpenCL library (installed with the board support package) enables OpenCL host calls to be directed to the board. Contrary to the soft GPU approach or conventional GPU execution, the AOC compiler facilitates parallelism by creating pipeline stages at any given moment [15]. To optimize the pipeline window, Intel advises that the device kernel operate as a single work item with a size of (1,1,1). Under this configuration, the AOC compiler seeks to implement pipelined parallelism on loops in kernel functions. Nevertheless, users may still execute multi-work item kernels. In such instances, kernels are executed using an N-Dimensional Range (NDRange)-based approach, where a deeply pipelined kernel representation allows for concurrent execution of multiple work items, thereby achieving pipeline parallelism. In this paper, we adopt the NDRange iterative work item issue approach to avoid modifying the GPU-friendly kernel code.

### C. Vortex: Open-source RISC-V GPGPU

Vortex [1], [2], [3] is an open-source RISC-V GPU that supports single-instruction, multiple-threads (SIMT) execution through extensions to the RISC-V ISA. Figure 4 shows the microarchitecture of Vortex. Vortex has been synthesized and run on FPGAs configured with up to 64 cores with a peak clock of over 200 MHz. Included with Vortex is an extensive software stack that supports OpenCL. The six-stage pipeline design is extensible and highly configurable, making Vortex a fitting research platform for design space exploration versus other methods of computing via FPGA.

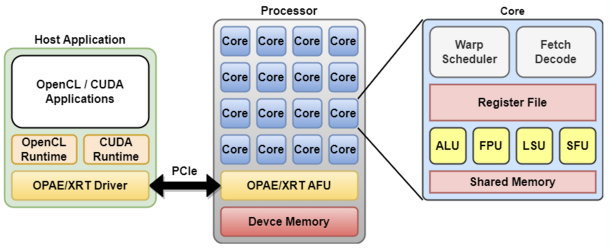



Fig. 4. Vortex microarchitecture [1], [2], [3]

### D. Vortex Software Stack for OpenCL

Figure 5 shows an overview of the Vortex software stack for OpenCL. To support OpenCL language, Vortex employs the Portable Computing Language (PoCL) [16] project, which implements extendable compiler and runtime support for OpenCL. Within the Vortex software stack, the Kernel Compiler and the OpenCL Runtime Library, as depicted in Figure 2, are substituted with the PoCL compiler and PoCL runtime, respectively. Vortex is able to use GCC and Clang as host compilers with PoCL runtime.

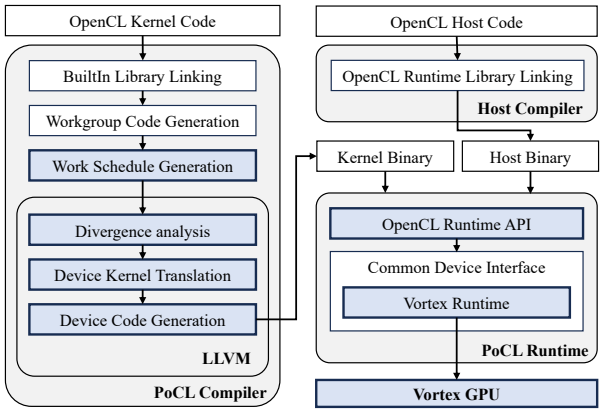



Fig. 5. Vortex Software Stack for OpenCL

The blue box in Figure 5 illustrates the extension to support Vortex. First, the PoCL compiler has to be extended to generate Vortex kernel binaries with the LLVM compiler [17]. The PoCL compiler is extended to add a pass that generates a kernel with work scheduling that reflects Vortex hardware. Then, LLVM is used to generate the kernel binary, which is compliant with the Vortex ISA. Second, the PoCL runtime is also extended using the Vortex runtime functions, including a communication interface with Vortex hardware. By extending the PoCL project, the Vortex software stack can support running the OpenCL program on Vortex.

Vortex uses a modified version of LLVM for the RISC-V compiler to support the extended ISA. The Vortex ISA extension is specifically designed to manage divergence within a warp through the introduction of four new instructions [1], [2], [3]. Vortex introduces the SPLIT instruction to indicate the divergent branch and the JOIN instruction to indicate the reconvergence point; the PRED instruction indicates the loop exit and prevents termination on the last iteration of the loop, and the TMC instruction changes the active threads. To use binaries that leverage this ISA supported by Vortex hardware, we need to modify the LLVM compiler to accommodate the Vortex ISA with divergence analysis, device kernel translation, and device code generation. Figure 5 illustrates the extended LLVM compilation flow to support Vortex.

## III. Evaluation

In this section, we compare the two aforementioned approaches to run GPU applications on FPGAs with respect to benchmark coverage and performance. As mentioned in previous sections, both Vortex and the Intel FPGA SDK utilized identical source code (both host and kernel), differing only in the kernel binaries loaded for all benchmarks. For our experiments, we selected two models from Intel's Stratix 10 FPGA family [14]: the SX2800 and the MX2100. The most notable difference between these two boards is that the MX2100 is equipped with HBM2 memory, whereas the SX2800 relies solely on DDR4 off-chip memory. Among the two boards, the Intel FPGA SDK approach was synthesized

TABLE I
BENCHMARK COVERAGE TABLE (LEFT:VORTEX, RIGHT:INTEL HLS)

| Benchmark Name | Vortex | Intel SDK | Reason to Fail |
|---|---|---|---|
| Vecadd | O | O | |
| Sgemm | O | O | |
| Psort | O | O | |
| Saxpy | O | O | |
| Sfilter | O | O | |
| Dotproduct | O | O | |
| SPMV | O | O | |
| Cutcp | O | O | |
| Stencil | O | O | |
| Lbm | O | X | Not enough BRAM |
| OCLPrintf | O | O | |
| Blackscholes | O | O | |
| Matmul | O | O | |
| Transpose | O | O | |
| Kmeans | O | O | |
| Nearn | O | O | |
| Gaussian | O | O | |
| BFS | O | O | |
| Backprop | O | X | Not enough BRAM |
| Streamcluster | O | O | |
| pathfinder | O | O | |
| nw | O | O | |
| B+tree | O | X | Not enough BRAM |
| LavaMD | O | O | |
| Hybridsort | O | X | Atomics |
| Particlefilter | O | O | |
| Dwd2d | O | X | Not enough BRAM |
| LUD | O | X | Not enough BRAM |

on the MX2100, and Vortex was synthesized on the SX2800. Although these two boards may yield slightly different performance results due to their differing off-chip memory configurations, coverage and challenges are the primary focus of this paper. Therefore, we believe that comparing the two approaches will provide insightful observations.

### A. Coverage Comparison: By Benchmark

To compare the benchmark coverage of the Intel SDK for OpenCL and Vortex, we tested 28 benchmarks from the Rodinia benchmark suite [18] and NVIDIA OpenCL SDK Code Samples [19]. Table I presents the overall results of benchmark coverage, indicating that six benchmarks failed to synthesize with the Intel SDK, in contrast to Vortex, which supported every benchmark. Vortex demonstrated broader benchmark coverage compared to the Intel SDK because soft GPUs are designed to utilize ISA and execute binaries on solid hardware.

In the *hybridsort* benchmark, the histogram kernel function includes an integer atomic add operation. Although the Intel SDK supports 32-bit integer atomic functions, it was unable to synthesize the kernel source code due to the heterogeneous memory system of the target FPGA, leading to synthesis failures. For other benchmarks experiencing synthesis errors, their area reports indicated that the required hardware resources exceeded limits, particularly in terms of BRAM blocks. A key reason for this is that each array access in the kernel code was synthesized into 32 load units. To mitigate hardware resource constraints, modifications can be made either to the type of Load Store Unit (LSU) or by enhancing data reuse at the source code level. The case study presented in Section III-B provides detailed methods to address these issues.

### B. Case Study: Increasing the Coverage of HLS

In HLS, one prevalent challenge encountered by users when compiling GPU-optimized OpenCL source code for FPGA deployment relates to the constrained hardware resources of FPGAs, particularly block RAM (BRAM). For instance, applications such as *lbm* and *backprop*, as illustrated in Table I, encounter synthesis obstacles due to insufficient BRAM, with error messages indicating the inability to synthesize the device source code because of inadequate BRAM resources. This issue is accentuated by the increasing complexity of modern applications undergoing FPGA adaptation. Specifically, the *backprop* application initially demanded 12,898 BRAM blocks, which exceeds the available BRAM capacity of the Stratix 10 FPGA by 188%. In this section, we aim to elucidate basic area optimization strategies to assist users in efficiently compiling their source code with HLS. It is important to note, however, that the application of these optimization techniques may impact overall performance due to the necessitated temporal reallocation of the limited hardware resources.

TABLE II
BACKPROP SYNTHESIS AREA REPORT USING INTEL HLS

| Optimization step | ALUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|
| Original code | 1,000,388 | 2,158,459 | 12,898 | 17 |
| Variable reuse (O1) | 826,993 | 1,587,827 | 9,882 | 9 |
| Pipelined load (O2) | 451,395 | 1,051,467 | 5,694 | 11 |

*1) O1: Variable Reuse:* In Figure 6, we present three code listings from the kernel function `bpnn`. Listing 1 shows the original device code, while Listing 2 and Listing 3 demonstrate the effects of applying two cumulative optimization techniques. Upon reviewing the original device source code, we identified that certain calculated values, such as `delta[index_x] * ETA`, were redundantly computed multiple times within the kernel function. To mitigate unnecessary computations and memory loads, additional local variables were introduced, enabling these values to be loaded once and reused throughout the function (Lines 7-9 in Listing 2). This optimization significantly reduced the number of block RAMs (BRAMs) utilized, from 12,898 (188%) to 9,882 (144%), as detailed in Table II.

*2) O2: Load Unit Pipelining:* Despite the substantial reduction in BRAM usage achieved through variable reuse, an additional reduction was necessary to accommodate all required hardware resources within the FPGA board. The synthesis report from O1 revealed that each line in lines 7-9 of Listing 2 involved 32 operating load units for array data retrieval, consuming over 1,000 BRAM blocks per line. To decrease the count of LSUs, `__pipelined_load` directives (Lines 7-9 in Listing 3) were implemented on the load operations. This pipelining strategy, recommended by Intel for its area efficiency at the expense of performance in non-consecutive access patterns, further reduced the BRAM count from 9,882 (144%) to 5,664 (83%).

```
__kernel void  bpnn(...){
  ...
  int index =  ( hid + 1 ) * HEIGHT *
       gid.y + ( hid + 1 ) * lid.y +
       lid.x + 1 + ( hid + 1 ) ;
  int index_y = HEIGHT * gid.y + lid.y
       + 1;
  int index_x = lid.x + 1;

  w[index] += ((ETA * delta[index_x] *
       ly[index_y]) + (MOMENTUM *
      oldw[index]));
  oldw[index] = ((ETA * delta[index_x]
       * ly[index_y]) + (MOMENTUM *
      oldw[index]));




  ...}
```

Listing 1. Original device code

```
__kernel void  bpnn(...){
  ...
  int index =  ( hid + 1 ) * HEIGHT *
       gid.y + ( hid + 1 ) * lid.y +
       lid.x + 1 + ( hid + 1 ) ;
  int index_y = HEIGHT * gid.y + lid.y
       + 1;
  int index_x = lid.x + 1;

  float delta_value = delta[index_x] *
       ETA;
  float ly_value = ly[index_y];
  float oldw_value = oldw[index] *
       MOMENTUM;

  float delta_by_ly = delta_value *
       ly_value + oldw_value;
  w[index] += delta_by_ly;
  oldw[index] = delta_by_ly;

  ...}
```

Listing 2. O1. Variable reuse

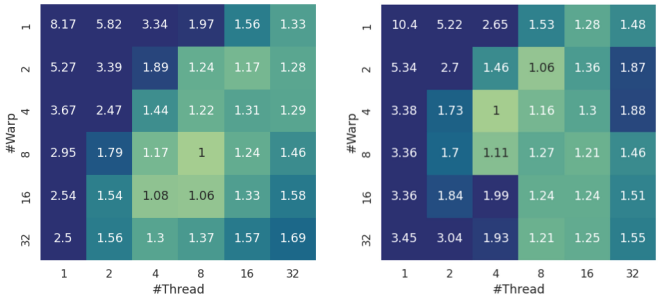
```
__kernel void  bpnn(...){
  ...
  int index =  ( hid + 1 ) * HEIGHT *
       gid.y + ( hid + 1 ) * lid.y +
       lid.x + 1 + ( hid + 1 ) ;
  int index_y = HEIGHT * gid.y + lid.y
       + 1;
  int index_x = lid.x + 1;

  float delta_value = __pipelined_load
       (delta + index_x) * ETA;
  float ly_value = __pipelined_load(ly
       +index_y);
  float oldw_value = __pipelined_load(
       oldw + index) * MOMENTUM;

  float delta_by_ly = delta_value *
       ly_value + oldw_value;
  w[index] += delta_by_ly;
  oldw[index] = delta_by_ly;
  ...}
```

Listing 3. O2. Pipelined load

Fig. 6. Back propagation benchmark kernel codes with cumulative optimizations




(a) Vector Addition (b) Transpose

Fig. 7. Cycle comparison for different warp and thread sizes with two benchmarks, Vector Addition and Transpose, on the Vortex simulator with four cores. The cycle is normalized to the minimum cycle result. The light color has a lower cycle compared to the dark color.

### C. Case Study: Performance Optimization of Vortex

In this case study, we demonstrate the performance impact of varying hardware configurations on Vortex. This analysis provides insights into how tuning hardware configurations, such as the number of warps or threads, is crucial for enhancing Vortex's performance. Increasing the number of threads necessitates an expansion in the register file size, as well as the number of arithmetic logic unit (ALU) lanes and floating-point unit (FPU) lanes. Additionally, augmenting the number of warp sizes leads to an expansion in the warp information table size. It is important to note that the application's latency does not invariably decrease with an increase in the number of warps and threads. This is because the application can encounter other bottlenecks, such as memory bandwidth limitations or pipeline unit stalls.

Figure 7 illustrates the performance variations of two benchmarks, *vector addition* and *transpose*, with different thread and warp sizes. These benchmarks differ in their memory access patterns, the number of loads per kernel, and the computation instructions count. The vector addition benchmark involves loading two floating-point values from two arrays indexed by the thread ID, adding these values, and then updating a third array also indexed by the thread ID. In contrast, the transpose benchmark works with a two-dimensional array, swapping values at opposite locations. This evaluation was conducted using the Vortex simulator with four cores.

The benchmarks not only reach their optimal performance at different warps and thread sizes but also exhibit varied overall cycle distributions. For vector addition, the best performance is observed with eight threads and eight warps. In contrast, transpose achieves optimal performance with four threads and four warps. This disparity arises because transpose—when saving the element— traverses the y-axis, taking less advantage of the cache in the core, incurs more LSU stalls with a higher number of threads and warps per core. If vector addition is configured with four warps and four threads, its performance could decrease by 44%. Similarly, if transpose is configured with eight threads and eight warps, its performance could decrease by 27%. The suboptimal configuration for both benchmarks—eight warps and four threads—results in performance reductions of 17% and 11%, respectively. This underscores the necessity of testing multiple hardware configurations to identify the optimal setup for each benchmark, highlighting the importance of targeted optimization.

### D. FPGA Usage

In this section, we compare the synthesized area reports for both the HLS and the soft GPU approaches. Each instance in the Intel SDK approach was synthesized for every benchmark, in contrast to Vortex, which had multiple synthesis instances with different hardware configurations that could be applied across all benchmarks. Table III presents the area reports from the synthesized kernel source code for selected benchmarks using HLS. Although the number of BRAMs was the common bottleneck in the AOC compiler approach, all types of hardware resources reflected the complexity of the benchmarks, ranging from relatively simple benchmarks (such as *Vecadd*

and *Matmul*) to more complex ones (*Gauss* and *BFS*), with the exception of DSP units, which showed relatively low usage across benchmarks.

TABLE III
SYNTHESIS AREA REPORT USING INTEL HLS

| Benchmark name | ALUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|
| Vecadd | 83,792 | 263,632 | 1,065 | 1 |
| Matmul | 250,218 | 415,893 | 2,696 | 5 |
| Gauss | 537,571 | 1,174,446 | 6,384 | 10 |
| BFS | 256,690 | 1,172,664 | 5,892 | 6 |

TABLE IV
SYNTHESIS AREA REPORT FROM VORTEX

| C | W | T | ALUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|---|---|
| 2 | 4 | 16 | 332,143 | 459,349 | 1,275 | 896 |
| 2 | 8 | 16 | 336,568 | 459,353 | 1,299 | 896 |
| 2 | 16 | 16 | 341,134 | 478,735 | 1,299 | 896 |
| 4 | 8 | 16 | 617,748 | 793,976 | 2,235 | 1,792 |
| 4 | 16 | 16 | 626,688 | 827,757 | 2,235 | 1,792 |

Similar to Table III, Table IV displays the synthesis area results for various hardware configurations from Vortex. Three indices (C, W, T) represent the number of cores, the number of warps per core, and the number of threads per warp, respectively. As expected, a greater number of cores, warps, and threads translate to increased hardware resource usage in FPGA synthesis. However, larger Vortex configurations do not always guarantee better performance across different benchmarks, as discussed in Section III-C.

When comparing Table III with Table IV, the soft GPU approach exhibited a broader range of hardware usage, providing users with more options to configure their FPGA within their area constraints without the need for source code optimizations. However, for simpler applications like *vecadd*, the HLS approach demonstrated more area-efficient synthesis results compared to soft GPU.

## IV. DISCUSSION

In this section, we discuss the challenges of running GPU applications on FPGAs using both HLS and soft GPU approaches. We explored two approaches attempting to bridge the gap between GPU applications and FPGA execution. However, as illustrated in Section III, both methods face challenges, including the need for kernel code modification in HLS due to inadequate BRAM, and the necessity for performance optimization in Vortex through hardware configuration exploration. Therefore, we discuss the challenges presented by these two approaches and explore potential solutions to address them.

### A. Challenges of Supporting OpenCL in Vortex

The primary challenge for Vortex is identifying the best configuration for a given workload, which is essential to achieve optimal performance. As demonstrated in Section III-C, the performance of applications depends significantly on both the hardware configuration and the applications' characteristics. However, testing all the hardware combinations in the hardware needs resynthesizing and effort. Thanks to its own simulator, Simx, Vortex can expedite the exploration of various hardware configurations. Simx is a C++ cycle-level simulator that achieves cycle accuracy within 6% compared to the Verilog model and helps rapidly explore various hardware configurations [3]. Therefore, a valuable opportunity exists for research aimed at minimizing or circumventing the exploration space by leveraging the application's characteristics and proposing an analytical model for Vortex's performance.

The second challenge involves both hardware and software modifications to introduce new features within the complex, multi-layered Vortex software stack. First, introducing new features requires hardware updates, such as adding support for barriers and atomic intrinsics. Second, supporting software features in the front-end language requires modifying the compiler's lowering process, accompanied by a range of design considerations. For instance, atomic functions in OpenCL need to be processed through the front-end, middle-end, and back-end of the compiler via an intermediate representation (IR) at each stage. The compiler can implement atomic functions in multiple ways, such as using atomic instructions or a combination of branch and load/store operations, and a decision is required on how to integrate these atomics—via IR, Vortex kernel function calls, or built-in libraries—into the lowering process. Thirdly, adding a new feature may necessitate updates in the host runtime library, such as incorporating a communication function for new GPU kernel capabilities like printing. A thorough abstraction and evaluation of the software stack simplifies maintenance and improves extensibility.

Additional challenges are not extensively addressed in this paper, including efficient divergence control and the effective distribution of work items to hardware parallel units. One of the strengths of Vortex is its hardware-supported divergence control logic, exposing SPLIT, JOIN, and PRED instructions [1], [2], which enables supporting benchmarks with complex control flows. However, these operations require additional computation cycles, indicating that compiler optimizations, such as uniform statement analysis or duplicating parts of the control flow, could further enhance performance. Another significant challenge involves the efficient mapping of work items and groups onto the Vortex hardware architecture, particularly in barrier support. Since mapping influences memory access patterns and pipeline unit stalls, devising adaptive mapping strategies based on application characteristics emerges as an area for research.

**Vortex Challenges** ❶ Identifying the optimal hardware configuration for applications. ❷ Simplifying the integration of new hardware and software capabilities into the Vortex ecosystem. ❸ Enhancing performance via compiler optimization to control divergence effectively. ❹ Identifying the optimal work item distribution on Vortex hardware.

### B. Challenges of Supporting OpenCL in HLS

One of the most important challenges HLS encounters is the difference in focus between HLS technology and GPU languages. HLS focuses on pipeline parallelism, while GPU languages are designed to express data-level parallelism. According to an Intel white paper [15], the Intel SDK recommends using a single work item and presents optimizations such as loop speculation and loop fusion for loops within a single work item. Although the Intel SDK supports multiple work items to bridge the gap between different types of parallelism, the maximum number of concurrent work items must be statically determined at compile time. Additionally, the bitstream generated by HLS does not contain a mechanism to distribute large work items into smaller parallel units, which can necessitate resynthesis. Furthermore, the characteristics demonstrated by the Intel SDK indicate that the potential advantages of GPU language abstractions are not fully exploited. It is believed that employing compiler techniques, such as mapping software-defined parallel units to hardware with limited parallelism (e.g., flat collapsing [16], [20]), can bridge this gap more effectively.

The second significant challenge in HLS is its poor programmability, necessitating an in-depth understanding of both HLS optimization techniques and FPGA architecture. This issue is evident in Section III-B, where a BRAM-intensive synthesis process renders FPGA execution impractical without modifications to the loading process. Given that the HLS compiler makes extensive use of pragmas, employing these pragmas demands a greater level of knowledge than is typically required for conventional GPU programming. Moreover, with certain GPU features being either unsupported or not recommended, programmers are compelled to identify alternative strategies. For instance, if a programmer employs a kernel function with barriers [15], the code must be adapted to utilize a single work item due to the lack of support for barriers. This challenge underscores the need for further compiler optimizations to enhance automation and feature support.

The final challenge with HLS is its tendency to prolong the development process since even small changes to a kernel require resynthesizing the entire bitstream. For example, the *backpropagation* benchmark mentioned in Section III-B took up to 10.4 hours for a successful synthesis, with unsuccessful attempts taking 1.5 and 1.2 hours. Repeating this process multiple times, especially alongside the second challenge, significantly slows development.

**HLS Challenges** ❶ Narrowing the gap between the parallelism approaches of HLS technology and GPU languages focused on pipeline and data parallelism, respectively. ❷ Improving poor programmability that requires a deep understanding of HLS optimizations and FPGA architecture. ❸ Streamlining lengthy development process coming from re-synthesizing kernel.

## V. Related Work

### A. FPGA vs GPU

Cong et al. compare FPGAs and GPUs using the Rodinia benchmark and Xilinx Virtex 7 FPGA versus NVIDIA K40c GPU [6], showing that FPGAs have 28% lower power use and lower runtime in six out of 15 kernels, despite lower clock speed and limited parallelism.

The work from Ejjeh et al. on the Intel FPGA SDK for OpenCL's impact on image processing in the ISP pipeline highlights the pros and cons of using high-level synthesis (HLS) tools [9]. This demonstrates that compiler optimizations can boost performance but require significant programmer effort. The study also points out the limitations of the HLS toolchain, including the need for extensive tuning and manual implementation of optimizations not automatically handled by the compiler.

Ahangari et al. developed a framework in SystemC, deviating from the prevalent use of OpenCL in Intel FPGA development [10]. This shift addresses a major issue with OpenCL: Its abstraction simplifies programmability versus HDLs but often reduces hardware design efficiency, especially in memory bandwidth usage. Their framework combines cycle-accurate implementation with high programmability and introduces a reusable template that eases the design process for non-hardware experts. Currently, its use is limited to graph-processing algorithms, facing challenges with larger graphs and lacking support for edge weighting.

Jin et al. [11] explore resource and peak memory bandwidth usage on FPGA implementations using the Intel SDK for vector addition on the Nallatech 3856A board. Their work compares compute unit duplication, with separate memory access, against SIMD vectorization, which duplicates only the data path.

Zohouri et al. investigate the Intel SDK's optimization on Stratix V FPGA, focusing on power and Rodinia benchmarks performance [7]. They discovered that applying FPGA-specific optimizations was more effective than directly porting GPU-optimized code to the FPGA. Their study shows FPGAs can outperform GPUs in efficiency by up to 3.2x and also exceed CPUs in performance and energy efficiency, offering greater adaptability. However, they note challenges such as the steep learning curve of hardware programming and the necessity for understanding low-level programming paradigms.

### B. Soft GPU

The DO-GPU framework [4] streamlines soft GPU development by offering automation and customization tools for building application-specific soft GPUs with "macro units". It automates soft GPU hardware generation by integrating these macro units into a flexible architecture, streamlining FPGA programming. Building on the PDL-FGPU, DO-GPU introduces standard interfaces for macro units and architectural enhancements, including scoreboarding for improved memory latency management and a VLIW scheduling approach for

simultaneous scalar and macro instruction execution, surpassing the previous single instruction per cycle limitation. Its performance was assessed using the Intel Stratix 10 (S10).

Duarte et al. developed FGPU [5], a 32-bit, multi-core, FPGA-based processor optimized for the OpenCL SIMT model, featuring area and energy efficiency with a 2.9x higher compute density and 11.2x lower energy use than MicroBlaze, and 4x faster than Cortex-A9 with NEON. Managed via a Python API, FGPU offers significant benefits but encounters scalability issues on smaller FPGAs due to high area demands, memory and speed limitations for intensive tasks, and the intricacies of FPGA-specific programming and timing management. These challenges are exacerbated by FPGA resource constraints and limited parallelism.

SCRATCH [12], building on the MIAOW [21] implementation, expands its instruction set from 42 to 156 and introduces features like a separate clock domain and prefetch system to reduce data latency. It includes a compile-time tool to customize the GPGPU architecture to specific applications and optimize resource use and execution precision. This tool streamlines the OpenCL compilation, design adjustment, and execution process on Xilinx FPGAs, increasing computational capacity by supporting more compute units and cores. SCRATCH facilitates native OpenCL code use on FPGAs, allows for architecture customization, and supports advanced image classification algorithms like CNNs, demonstrating its utility in high-level computing tasks.

## VI. Conclusion

In this paper, we presented a study comparing two approaches—HLS and soft GPU—that attempts to bridge the gap between parallel programming languages and the execution of GPU applications on FPGA devices. Our analysis of each approach's compilation and execution pipelines uncovered inherent challenges. Specifically, the HLS exhibited poor programmability, especially when running complex applications, often requiring kernel code modifications for FPGA compatibility. On the other hand, the optimal hardware configuration in the soft GPU was found to be application-dependent. This underscores the need for a more sophisticated approach, such as an analytical model, to identify the optimal soft GPU configuration for a more intuitive user experience.

## Acknowledgment

The authors would like to thank AMD, BAH for supporting this research work.

## References

[1] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021.

[2] B. Tine, V. Saxena, S. Srivatsan, J. R. Simpson, F. Alzammar, L. Cooper, and H. Kim, "Skybox: Open-source graphic rendering on programmable risc-v gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023.

[3] F. Elsabbagh, B. Tine, P. Roshan, E. Lyons, E. Kim, D. E. Shim, L. Zhu, S. K. Lim, and H. kim, "Vortex: OpenCL Compatible RISC-V GPGPU," Feb. 2020.

[4] R. Ma, J.-C. Hsu, T. Tan, E. Nurvitadhi, R. Vivekanandham, A. Dasu, M. Langhammer, and D. Chiou, "DO-GPU: Domain Optimizable Soft GPUs," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. Dresden, Germany: IEEE, Aug. 2021, pp. 140–144.

[5] M. A. Kadi, B. Janssen, J. Yudi, and M. Huebner, "General-Purpose Computing with Soft GPUs on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 1, pp. 1–22, Mar. 2018.

[6] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding Performance Differences of FPGAs and GPUs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Boulder, CO, USA: IEEE, Apr. 2018, pp. 93–96.

[7] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT: IEEE, Nov. 2016, pp. 409–420.

[8] M. Meyer, T. Kenter, and C. Plessl, "In-depth FPGA accelerator performance evaluation with single node benchmarks from the HPC challenge benchmark suite for Intel and Xilinx FPGAs using OpenCL," *Journal of Parallel and Distributed Computing*, vol. 160, pp. 79–89, Feb. 2022.

[9] A. Ejjeh, V. Adve, and R. Rutenbar, "Studying the Potential of Automatic Optimizations in the Intel FPGA SDK for OpenCL," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2020, pp. 318–318.

[10] H. Ahangari, M. M. Özdal, and Ö. Öztürk, "HLS-based High-throughput and Work-efficient Synthesizable Graph Processing Template Pipeline," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, pp. 1–24, Mar. 2023.

[11] Z. Jin, K. Yoshii, H. Finkel, and F. Cappello, "Evaluation of the Single-precision Floatingpoint Vector Add Kernel Using the Intel FPGA SDK for OpenCL," Tech. Rep. ANL/ALCF–17/2, 1357902, Apr. 2017.

[12] P. Duarte, P. Tomas, and G. Falcao, "Scratch: An end-to-end application-aware so-gpgpu architecture and trimming tool," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2017, pp. 165–177.

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

[14] I. Corporation, "Intel fpga sdk for opencl pro edition," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/software-kit/782424/intel-fpga-sdk-for-opencl-pro-edition-software-version-23-2.html

[15] I. Corporation., "Intel fpga sdk for opencl pro edition: Best practices guide," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683521/22-4/eol.html

[16] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," in *International Journal of Parallel Programming*, 2015.

[17] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[19] Nvidia, "Open computing language opencl." [Online]. Available: https://developer.nvidia.com/opencl

[20] R. Han, J. Lee, J. Sim, and H. Kim, "Cox: Exposing cuda warp-level functions to cpus," sep 2022. [Online]. Available: https://doi.org/10.1145/3554736

[21] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Miaow - an open source rtl implementation of a gpgpu," in *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*, April 2015, pp. 1–3.