Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs

Christoph Weckert
Leonardo Solis-Vasquez
Julian Oppermann
Andreas Koch
Technical University of Darmstadt
Darmstadt, Germany

Oliver Sinnen
University of Auckland
Auckland, New Zealand

ABSTRACT

In this work, we introduce Altis-SYCL, a benchmark suite based on SYCL for GPUs and FPGAs. For developing Altis-SYCL, we leverage the oneAPI heterogeneous programming framework in two consecutive steps: 1) by using the modern Altis GPGPU benchmark suite as baseline and migrating it from CUDA to SYCL, and 2) by exploring several techniques to optimize the performance of the resulting SYCL code. Our migration-and-optimization methodology starts targeting GPUs and progressively moves towards FPGAs. In this process, we discuss the differences between device-specific strategies as well as detailing the required code refactoring and optimization efforts. The performance of Altis-SYCL was evaluated on Stratix 10 and Agilex FPGAs, and for some applications, their execution runtimes were competitive with those achieved on latest high-end GPUs. The corresponding code is released as open source under: https://github.com/esa-tu-darmstadt/altis_sycl.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**.

KEYWORDS

oneAPI, DPC++ Compatibility Tool, Altis, GPGPU benchmark, CUDA, SYCL, FPGA, Stratix 10, Agilex

ACM Reference Format:
Christoph Weckert, Leonardo Solis-Vasquez, Julian Oppermann, Andreas Koch, and Oliver Sinnen. 2023. Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs . In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3624542

1 INTRODUCTION

Since its initial announcement in 2018, Intel has been actively developing oneAPI [17], whose main objective is to provide a unified platform for developers to program CPUs, GPUs, and even FPGAs using a common language called Data Parallel C++. This language, DPC++, is built on the widely-used ISO C++ standard and incorporates the Khronos standard SYCL [19] along with community extensions.

The appeal of oneAPI lies in its foundation on C++, which is a preferred programming language in the High Performance Computing (HPC) community. Consequently, oneAPI has been generating significant interest within this community, particularly as ongoing research endeavors aim to efficiently execute GPU-native code on FPGAs [18, 22, 25]. This objective presents a formidable challenge, as even moderately-refactored Single-Instruction Multiple-Thread (SIMT) kernels may not perform optimally on FPGAs. Hence, achieving FPGA-specific optimization from a high-level design becomes crucial and deserves further exploration.

In this regard, recent research efforts have developed high-level benchmark suites for FPGA-based HPC systems. For instance, [20] proposed a parameterized OpenCL adaptation of the HPCC benchmark suite, while [26] discussed an optimized multi-FPGA implementation based on MPI and a C++ dialect (i.e., Vitis High Level Synthesis) of the HPCG benchmark suite. Moreover, [1] introduced HosNa, the first benchmark suite aiming to systematically evaluate optimization strategies on FPGAs using the SYCL language. While the SYCL-based implementation of HosNa provides advantages over previous studies (namely, holistic C++ programming view over OpenCL and code portability over Vitis HLS), it explores only Single-Task (i.e., single-threaded) implementations, and thus, it is not efficiently portable to widespread HPC systems such as those based on GPUs.

To fill this gap, we have developed Altis-SYCL: a benchmark suite for both GPUs and FPGAs derived from Altis [7], a modern GPGPU benchmark suite originally written in CUDA. In this work, we focus on leveraging the oneAPI ecosystem for migrating and optimizing Altis-SYCL. In particular, we describe our experience with employing Intel's DPC++ Compatibility Tool (DPCT) [9] that automates the code migration process from CUDA to SYCL, then improving this GPU-targeted SYCL, and adding FPGA-specific optimizations. With Altis-SYCL, besides providing a high-level benchmark suite for GPUs and FPGAs, we aim to present a comprehensive set of practical guidelines that can be applicable to efforts aiming to quickly deploy CUDA applications on HPC systems supporting SYCL. Our contributions are summarized as follows:

- We present our experience using DPCT for migrating the Altis benchmark suite from CUDA to SYCL.
- We discuss device-specific performance optimization techniques for both GPUs and FPGAs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624542

- We compare the performance achieved on CPUs, GPUs, and Intel FPGAs including Stratix 10 and Agilex.

2 BACKGROUND

2.1 The DPC++ Compatibility Tool

As part of Intel's oneAPI ecosystem [14], the DPC++ Compatibility Tool (DPCT) [9] assists developers by *automatically* migrating around 90%-95% of CUDA code into human-readable SYCL code. Such capabilities also include the migration of 1) API calls of common CUDA libraries (e.g., cuBLAS, cuDNN, cuFFT, NVIDIA's Thrust) and 2) commands for compiling the original CUDA project (e.g., in CMake files). If required, developers can *manually* complete the migration by following the hints in the inlined comments inserted by DPCT that point out required localized code changes.

2.2 The Altis Benchmark Suite

Altis [7] consists of a collection of GPGPU applications written in CUDA. By adopting and extending applications from existing benchmark suites such as Rodinia [4] and SHOC [5], as well as by adding new applications, Altis aims to better represent modern GPGPU workloads.

Altis offers three main advantages. First, it encompasses emerging domains, such as DNNs, which are either not covered or handled with outdated techniques in existing benchmark suites. Second, Altis addresses the dataset size issue comprehensively. It supports modern default sizes and provides the flexibility to use non-default sizes as needed. This approach overcomes the limitations of Rodinia, which lacks dataset size guidance, and SHOC, which lacks flexibility. And finally, Altis incorporates support for new CUDA features including: unified memory, events, HyperQ (enabling multiple independent CUDA kernels to run in parallel on the same GPU), nested parallelism, and grid-level synchronization.

The applications in Altis fall into four categories, of which we focus here on Level 2, the real-world application kernels shown in Table 1.

Table 1: Altis' Level 2 applications

Application	Description
CFD	Computational Fluid Dynamics. 3D Euler equation solver for compressible flow
DWT2D	Discrete Wavelet Transform. 2D transform for digital signal processing
FDTD2D	Finite Difference Time Domain. 2D Maxwell equation solver for electrodynamics
KMeans	Clustering algorithm for data mining
LavaMD	N-body particle interaction in a 3D space
Mandelbrot	Fractal image computation
NW	Needleman-Wunsch. Non-linear optimization for DNA sequence alignment
ParticleFilter (PF)	Statistical estimator of location of a target object
Raytracing	Image generator based on path-of-light tracing
SRAD	Partial differential equation solver for noise reduction
Where	Record filtering for data analytics

2.3 Related Work

Here, we discuss related studies leveraging oneAPI for GPUs/FPGAs, which we classify into two groups according to the adopted programming approach.

2.3.1 Direct SYCL Programming. these works study applications natively developed in SYCL. For instance, HosNa [1] and [6] followed a Single-Task (i.e., single-threaded) implementation approach, while [22] maintained ND-Range (i.e., multi-threaded) configurations for all kernel variants evaluated on FPGAs. In contrast, our work here evaluates both Single-Task and ND-Range implementations on FPGAs.

The studies in [18, 23] target not only FPGAs, but also CPUs and GPUs. On the one hand, [18] proposed a cooperative GPU-FPGA execution in which the kernels of the resulting unified SYCL application were conveniently partitioned between a GPU and an FPGA. This strategy aimed to leverage specific accelerators' strengths, while avoiding the cumbersome usage of different APIs (i.e., CUDA for GPU, OpenCL for FPGA). On the other hand, [23] indicated that on FPGA, executions of all ND-Range implementations were *faster* than their Single-Task counterparts, while being overall at least 4× slower wrt. those on a 18-core CPU. In contrast, as reported in Section 5, some applications in Altis-SYCL achieve higher performance on FPGAs over CPUs and GPUs.

2.3.2 DPCT-assisted Migration. similar to our work here, [3, 25] employed DPCT to automate the migration of applications originally written in CUDA. [25] proposed device-specific optimization techniques for ultrasound beamforming. Moreover, although not targeting FPGAs, [3] is included in this review due to the similarity between its case study (i.e., Rodinia) and ours (i.e., Altis). [3] reported a detailed DPCT-migration experience for Rodinia as well as a performance analysis on CPUs and GPUs. Our work here advances further by migrating the more modern Altis suite and benchmarking its performance also on Stratix 10 and Agilex FPGAs.

3 MIGRATING ALTIS FROM CUDA TO SYCL FOR GPUS

In this section, we present our methodology for migrating Altis (originally written in CUDA) to Altis-SYCL by using DPCT (oneAPI Base Toolkit v2022.3), initially targeting GPUs, as DPCT does not support directly migrating to FPGAs.

3.1 Experimental Environment

Table 2 lists all accelerator devices employed in this work: a Xeon Gold 6128 CPU, {RTX 2080, A100, Max 1100} GPUs, as well as {Stratix 10, Agilex} FPGAs.

The peak *theoretical* performance of FPGAs in FLOP/s can be calculated as follows: Peak FP32 = $N_{\mathrm{DSP}}^{\mathrm{Total}} \times 2 \times F_{\mathrm{Kernel}}^{\mathrm{Max}}$ [8], where $N_{\mathrm{DSP}}^{\mathrm{Total}}$ is the total number of DSPs on the FPGA (each supporting a FMA operation, and thus, the above factor of two) and $F_{\mathrm{Kernel}}^{\mathrm{Max}}$ is the maximum operating frequency of the kernel. Since some FPGA resources are utilized for the fixed board interface, and the achievable frequencies of SYCL kernels are much lower than the theoretical maxima, in Table 2, we report for the FPGA platforms their peak *attainable* performance instead. This is calculated using the above equation and considering the DSPs available for the user logic and the achieved frequency ranges of SYCL kernels. Therefore, the peak attainable performance is within {2.4 - 4.2} TFLOP/s for Stratix 10 and {2.3 - 5.0} TFLOP/s for Agilex.

Table 2: Employed Accelerator Devices

Device	Process [nm]	# Compute Units	Peak FP32 [TFLOP/s]	Peak Mem. BW [GB/s]
Xeon Gold 6128 CPU	14	6 Cores	1.1	128.0
RTX 2080 GPU	12	46 SMs	10.1	448.0
A100 GPU	7	108 SMs	19.5	1555.0
Max 1100 GPU (i.e., “Ponte Vecchio” [12])	10	56 X^e-cores	22.2	1229.0
Stratix 10 FPGA (BittWare 520N [2])	14	4713 DSPs (user logic)	2.4 (250 MHz) - 4.2 (450 MHz)	76.8
Agilex FPGA (DE10 Agilex [24])	10	4510 DSPs (user logic)	2.3 (250 MHz) - 5.0 (550 MHz)	85.3

3.2 Experiences with DPCT

Initially, we employ DPCT's `intercept-build` script to catch all compiler commands used in the regular build of an application project and to store them in a JSON file. Next, we run the `dpct` utility to migrate all source-code files to SYCL. DPCT supports the migration of most CUDA constructs employed in host and device code. Moreover, by using the above JSON file, DPCT can maintain the application's folder structure and even adjust its CMake files.

In this section, we address the main migration challenges of using DPCT in order to achieve a functionally correct execution of Altis-SYCL on the RTX 2080 GPU.

3.2.1 DPCT diagnostics. DPCT provides a number of warnings highlighting issues that need to be addressed by the user. Altis has roughly 40 k lines of code and DPCT inserted 2,535 warnings. After addressing them, ~70% of the migrated applications execute without errors on the RTX 2080. The most frequent warnings we encountered were:

Time measurements: DPCT translates measurements of kernel execution times from CUDA events to `std::chrono` calls. The latter also accounts for the kernel invocation overhead that involves several factors, including the just-in-time compilation for GPUs, or the FPGA reprogramming using the bitstream built ahead-of-time. Based on that, time measurements in the original CUDA and the migrated SYCL are *not* directly comparable, and thus, warnings are annotated by DPCT. Hence, we manually transform the `std::chrono` calls to SYCL events, which emulate the time measurements of CUDA events more accurately. However, we do this only when appropriate, as SYCL events are not usable in some cases, e.g., together with oneAPI library calls (Section 3.2.2).

Unified Shared Memory: USM is supported in all Altis applications. The corresponding warnings inserted by DPCT concern the calls to `mem_advise()` [10], which is designed to inform the SYCL runtime how different allocations will be used. The parameters for this function are device-dependent, and developers must identify their correct values for the specific target device. Furthermore, since the selected FPGA boards in Table 2 do not support USM (e.g., `sycl::malloc_host()` queries to both Stratix 10 and Agilex always return `nullptr`), we remove any USM usage from Altis-SYCL.

Barriers: their performance can be improved by narrowing the scope of synchronization. In practice, this can be achieved by explicitly specifying that the corresponding memory address space is *local*, e.g., `barrier(sycl::access::fence_space::local_space)`. Sometimes DPCT fails to detect whether this configuration is appropriate, and thus, assumes that the memory address space is instead *global* (i.e., by omitting the argument from the barrier call). Hence, we manually check whether the local scope can be used safely, and if so, we add the respective argument to the barrier call.

3.2.2 Miscellaneous. these encompass issues and migration cases not handled by DPCT. After addressing these, we achieve functional correctness in all migrated applications.

DPCT header files: these provide various functionalities comparable to those in CUDA such as device selection, constant-memory wrappers, etc. When using the DPCT header files, we found two problems. 1) The provided device-selection logic does *not* supply mechanisms to enable profiling for SYCL queues. Since such queue configuration is required for retrieving timing data from SYCL events, this prevented the accurate measurement of kernel execution times. 2) Occasionally, the data initialization in constant-memory wrapper objects occurred *after* their actual usage, causing segmentation faults. To address these problems, and the FPGA-specific one discussed later in Section 4, we opt to completely abandon the use of the DPCT header files for our code migration study, and thus, replace the invocation of their helper functions with standard SYCL code.

Dynamic memory allocation: the C++ `new` and `delete` operators can be used in CUDA kernels to allocate and free global memory, respectively. DPCT does *not* annotate that these operators are *not* supported in SYCL kernels. Hence, we manually move such memory allocations to the host side.

Polymorphism: DPC++ provides only *experimental* support for virtual functions when targeting CPUs. However, DPCT does *not* annotate this lack of support in the *standard* SYCL. For instance, the CUDA version of `Raytracing` uses virtual functions for objects and materials in the scene. Therefore, we completely remove such functions, and thus, have to significantly refactor `Raytracing`.

3.3 Optimizing SYCL performance on the RTX 2080 GPU

Once functional correctness is achieved, we perform a number of steps to create an optimized SYCL version having a similar performance to that of the original CUDA code.

Time measurements: for `FDTD2D`, we observed that the performance discrepancies between CUDA and SYCL were mostly the result of an inaccurate time measurement in the original CUDA code. We address this by adding the missing `cudaDeviceSynchronize()` device synchronization call, which brings the performance of both versions to comparable levels.

NVCC (CUDA) vs. Clang (SYCL): these are different compilers, and consequently, their behavior can diverge significantly in various situations. In our analysis of Altis, we discover that loop unrolling and inlining can have a different performance impact on CUDA and SYCL.

- Loop unrolling might increase the performance in CUDA, but it might also have the opposite effect in SYCL. For instance,

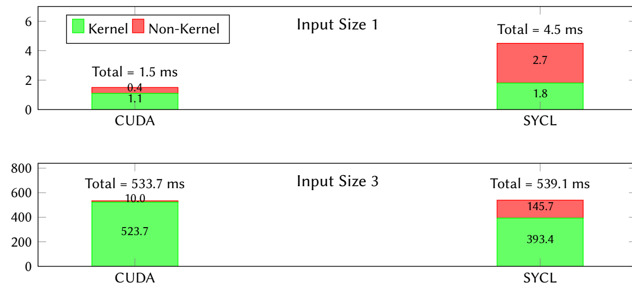


Figure 1: Execution-Time [ms] Decomposition of FDTD2D on the RTX 2080 GPU: CUDA vs. SYCL.

the SYCL version of CFD performed up to 3× worse when unrolling the main loop than without unrolling it. Hence, we remove unrolling to attain a more comparable performance wrt. CUDA.

- The Clang compiler for SYCL seems to act more cautiously when inlining functions: even if the kernel only calls a single function, this might *not* be automatically inlined if it contains more than a certain number of instructions. We increase such threshold by passing `-finlining-threshold = 10000` as a compiler option, and observe up to 2× higher performance for NW (and smaller improvements for other applications).

Power math function: during the migration of ParticleFilter Float, DPCT replaced a call to `pow(a,2)` with `a × a`. This seemingly minor change resulted in up to 6× higher performance of SYCL compared to CUDA. By applying such transformation back to the original CUDA codebase, we are able to bring both versions to a performance-comparable level.

Prefix-sum in oneDPL: Where uses a prefix-sum implementation from CUDA, which is migrated by DPCT to a corresponding implementation in the oneDPL library. Unfortunately, on the RTX 2080, this prefix-sum from oneDPL is 50% slower than that from CUDA. For GPUs, we decided to continue using this oneDPL implementation, while for FPGAs, we develop a custom prefix-sum version (Section 5.3).

Discussion: some Altis applications do not just benchmark the kernel execution, but instead time the entire program (i.e., including the kernel invocation overhead and host-device communication). Thus, in these cases, SYCL shows longer execution times than CUDA due to the higher overhead introduced by the oneAPI environment. This is illustrated in Figure 1, where the total execution time of FDTD2D is decomposed into kernel and non-kernel regions. For the *smaller* input size 1, the non-kernel region in SYCL is 1.5× (= $\frac{2.7}{1.8}$) longer than the corresponding kernel region and ~6.7× (= $\frac{2.7}{0.4}$) longer than its CUDA non-kernel counterpart. In contrast, for the *larger* input size 3, the kernel execution time in SYCL is ~2.7× longer (= $\frac{393.4}{145.7}$) than its corresponding non-kernel one. Profiling on the RTX 2080 reveals that this is mainly due to the migrated SYCL version invoking some extra underlying CUDA APIs for context/event management (also observed in [3]).

Figure 2 shows the speedups of SYCL over CUDA achieved on the RTX 2080 using the baseline (i.e., functionally correct but non-optimized) and optimized SYCL versions. In terms of performance, most of the optimized SYCL versions are similar (i.e., speedup = 1×) or even superior (e.g., Raytracing, speedup = ~21×) wrt. their CUDA counterparts, while only Where underperforms for all input sizes (i.e., speedup = ~0.3×), which as indicated above, is due to the usage of the prefix-sum from oneDPL. Overall, the geometric mean of the speedup achieved with the optimized SYCL wrt. CUDA is 1.0× (size 1), 1.1× (size 2), and 1.3× (size 3).

It is important to note that the resulting SYCL version of Raytracing differs significantly from the original CUDA, and hence, their execution times are *not* directly comparable. The two main reasons for that are 1) the different random number generator introduced by DPCT, namely oneMKL's philox4x32x10 [15], vs. the original cuRAND's XORWOW [21]), and 2) the significant manual code refactoring we required to cope with CUDA virtual functions (Section 3.2.2). Nevertheless, we consider this set of GPU-optimized SYCL versions a reasonable baseline for subsequent FPGA-specific refactoring.

4 CODE REFACTORING ALTIS-SYCL FOR SUCCESSFUL FPGA DESIGN GENERATION

In this section, we discuss the code refactoring applied to the ND-Range GPU-optimized Altis-SYCL kernels from Section 3 in order to successfully generate FPGA bitstreams.

DPCT header files: when included, the `memcpy` function in these header files is synthesized. Although the resource utilization of the corresponding kernel on the Stratix 10 is not significant (e.g., up to 1% of RAM and DSP, while <1% of other resources), this may still influence the FPGA bitstream generation success of complex designs. Hence, as already indicated in Section 3.2.2, we remove the usage of these header files from all our designs.

Congested memory ports: some Altis applications, e.g., DWT2D, perform numerous operations on a single shared-memory array. The correspondingly large number of required read and write ports, along with the arbiters needed to schedule the access to these ports, make the FPGA synthesis challenging. In general, the size of a shared-memory array depends on the employed work-group size. Hence, for applications whose syntheses incurred timing violations, we decrease their work-group sizes, and in turn, reduce the complexity of their generated memory systems.

Default work-group sizes: due to the SIMT nature of CUDA, all migrated Altis-SYCL kernels are implemented in an ND-Range fashion. For generating efficient FPGA hardware, the compiler must assume a maximum work-group size supported by the design. In the presence of barriers, this limit is automatically set to 128, while in other cases, the limit is determined by a default value that depends on compilation time and runtime constraints [13]. However, the default work-group sizes employed in Altis are generally larger than the preconfigured limits of the FPGA compiler, causing runtime errors. To fix this, we specify both the required and maximum number of work-items for a work-group in a kernel by inserting the `sycl::reqd_work_group_size(1, 1, BLOCK_SIZE)` and `intel::max_work_group_size(1, 1, BLOCK_SIZE)` kernel attributes, respectively.

Multiple kernel versions: some Altis applications contain alternative versions of a kernel for specific algorithms or problem sizes. By default, the compiler aims to synthesize all these kernel

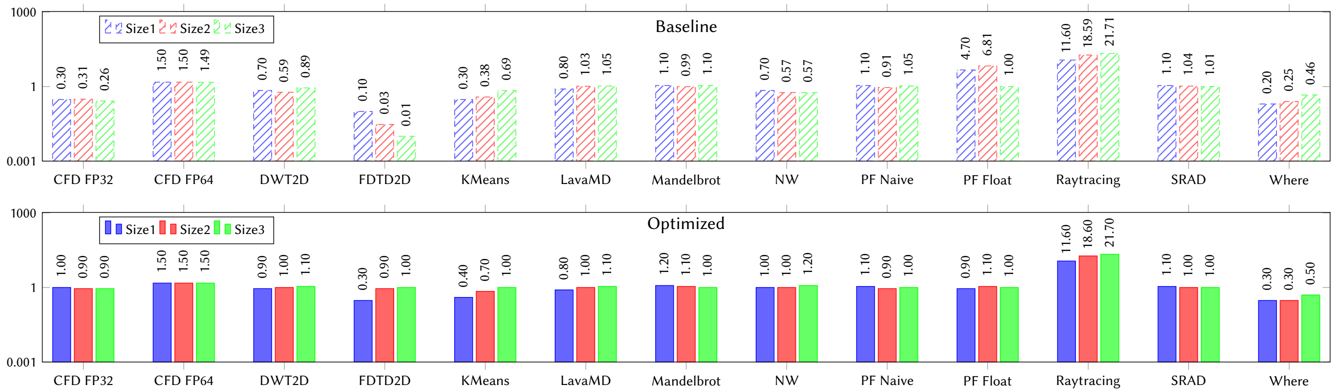


Figure 2: Speedup of Altis-SYCL over Altis (CUDA) achieved on the RTX 2080 GPU.

versions into a single FPGA binary. However, this is only possible for a limited set of kernels due to the constrained hardware resources. Therefore, for such applications, we synthesize only the required kernels for the intended use. For instance, DWT2D features a total of 14 kernels, from which only two are selected in order to handle the default algorithm and input size 3 with a given FPGA bitstream.

SYCL accessors: although they are the standard method for creating shared memory in SYCL, we found that accessors can cause issues when targeting FPGAs.

- The SYCL accessors introduced by DPCT are dynamically sized and cannot be statically defined at compile time. Hence, during synthesis, the compiler must assume a maximum size of 16 kB for each shared variable, which turned out to be resource-wasteful across all Altis applications employing shared memory. For instance, PF Float utilizes only a `double` shared scalar, for which the compiler generates a memory system of 16 kB instead of just 8 B. In the case of ND-Range kernels, a developer *cannot* control the degree of banking, replication, or private copies of the memory system. This limitation, coupled with the aforementioned high default size of shared memory, might hinder the success of placement.
- Sporadically, DPCT passes as a kernel argument a SYCL accessor object rather than a pointer to such accessor (i.e., of type `sycl::local_ptr<T>`). On CPUs and GPUs, we observe no performance difference between these two approaches. On FPGAs, however, passing an accessor object as kernel argument causes member functions of the accessor to be synthesized, leading to higher resource utilization. For instance, the initial design of SRAD passed eleven accessor objects to the kernel, which exceeded the resource limits of the Stratix 10 device. By passing instead pointers to the above accessors, we are able to successfully fit the design on FPGA.

5 OPTIMIZING ALTIS-SYCL FOR FPGAS

In this section, we apply techniques for increasing the performance on FPGAs. These techniques are grouped into 1) general optimizations, and those for 2) FPGA-refactored ND-Range kernels from Section 4, and for 3) Single-Task kernels reimplemented from the above ND-Range ones.

5.1 General Optimizations

Besides applying common techniques such as denoting non-aliasing pointers, precomputing constants on host and afterwards passing them to device, as well as loop unrolling, our focus here is on applying two techniques:

Replicating compute units: this can be beneficial to distribute work among replicated kernels, also called *compute units*. Depending on the type of kernel implementation, we perform any of the following actions:

- Single-Task kernels can be easily replicated by using the `SubmitComputeUnits` helper function from Intel's oneAPI samples repository [16].
- For ND-Range kernels, while Intel's *OpenCL* framework supports automatic replication, the above oneAPI samples repository does *not* provide such functionality. Therefore, we implement a custom helper class, which is used to instantiate ND-Range kernels a user-defined number of times, and then to distribute the work-items among them.

It should be noted that replication is often limited by the amount of available FPGA resources. For instance, kernels in CFD FP64 can be replicated at most twice. Our strategy is to initially optimize a *single* instance of a kernel before considering replication, and subsequently, to replicate the kernel as often as possible, while ensuring that each further replication attempt continues to provide substantial performance improvements.

Datatype optimizations: we noticed that the FPGA compiler often infers inefficient global and local memory systems when using a C++ class or struct featuring multiple member variables of different types. An example is the `material` class of Raytracing shown in Listing 1. For the original implementation, the compiler inferred a complex *non* stall-free memory system containing several arbiters as well as load and store ports. This was unexpected as the corresponding kernel employs only a *single* write access to the respective object, but the inferred hardware contained *three* store ports. Therefore, we optimize the `material` class by fusing all its

member variables into a single vector member, from which the compiler can then infer a stall-free memory system instead.

```
class material {    /* Original */
public:
    enum type: uint8_t {metal, dielectric, lambertian};
    type m_type;
    vec3 m_albedo;      // lambertian and metal (lam)
    float m_fuzz;       // metal (met)
    float m_ref_idx;};  // dielectic (die)

class material {    /* Optimized */
public:
//   data[0]: "fuzz" parameter
//   data[1]: "ref_idx" parameter
//   data[2:4]: "albedo" parameter
//   data[5]: material "type": met (0), die (1), lam (2)
//   data[6:7]: unused
     sycl::float8 data;};
```

Listing 1: Original and optimized `material` class.

5.2 Optimizing Migrated ND-Range Kernels

In many cases, ND-Range kernels can provide sufficient performance on FPGAs. Therefore, we choose to first optimize the migrated ND-Range kernels before considering their more laborious reimplementation in Single-Task fashion. Besides further restricting and tuning work-group sizes of all Altis applications, two more techniques have proven useful:

Vectorization: multiple work-items can be executed in a SIMD manner without manually vectorizing the kernel by specifying the `[[intel::num_simd_work_items(V)]]` attribute. In addition, in some cases, we have to rewrite conditional statements as ternary operators for achieving a successful vectorization. We notice that the resource utilization scales approximately linearly with the vectorization factor V, while the performance only scales accordingly when the available memory bandwidth is sufficient for the increased number of global memory accesses. For example, the performance of CFD FP32 only scales up to V = 2.

Shared memory: VTune Profiler reveals a higher pipeline activity in sections accessing shared memory. The following are the three situations we encountered when tackling such potential bottlenecks:

- Case 1: *access patterns allow efficient banking and replication of the memory system.* For instance, LavaMD, for which we successfully unroll by 30× a bottleneck loop (operating on shared memory), improves its performance almost linearly with the unrolling factor. Although further unrolling does not exceed the FPGA resource limitations, it leads to timing violations during synthesis.
- Case 2: *similar to Case 1, but the required resources exceed the available limit.* This was the case of SRAD, whose kernels employ *eleven* shared arrays. Unrolling a loop operating on this many memories, or entirely vectorizing the enclosing kernel, can lead to excessive resource utilization for large work-group sizes. For this application, we tested combinations of suitable work-group sizes and vectorization factors, e.g., a 64×64 work-group size with SIMD = 2 performs ~4× faster than a 16×16 work-group size with SIMD = 8.
- Case 3: *access patterns prevent efficient banking and replication.* In such situations (e.g., NW), the synthesis tool inserts arbiters to manage the shared-memory accesses. Furthermore, unrolling a loop accessing such memory is not a viable option, as it results in timing violations during the place and route process.

For Intel FPGAs, it is beneficial to replace the default SYCL accessors with `group_local_memory_for_overwrite` class objects. While this is vendor- and device-specific (via the oneAPI FPGA Toolkit, and not supported on CPUs/GPUs), it allows the implementation of shared memories with user-defined sizes, in contrast to default accessors (Section 4). We apply this to all Altis-SYCL ND-Range kernels employing shared memory, and thus, reduce their resource utilization.

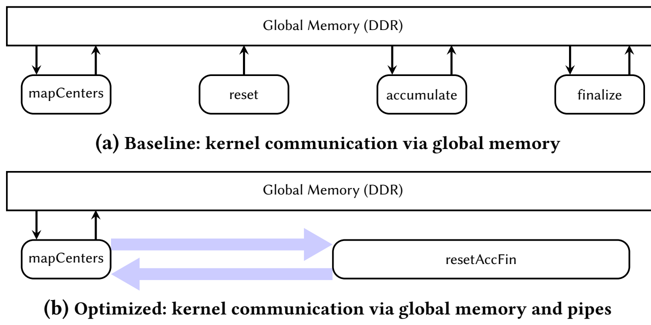
5.3 Rewriting ND-Range as Single-Task Kernels

We opt to reimplement some ND-Range kernels as Single-Task for two reasons: 1) to vectorize in cases where is not possible otherwise (e.g., PF, which contains many execution branches), and 2) to improve inter-kernel communication by using pipes instead of global memory (e.g., KMeans). The following techniques have been beneficial for kernels rewritten into Single-Task form.

Loop optimizations: in contrast to ND-Range kernels, loops in Single-Task kernels can be pipelined, which introduces two important loop attributes: initiation interval (ideally II = 1) and speculated iterations. In Altis, the exit condition of loops often becomes part of the critical path. To prevent clock-rate degradation, speculated iterations can be added to loop executions. Such iterations perform no useful work and their results are discarded as soon as the exit condition of previous iterations is satisfied. To put this into perspective, Mandelbrot performs two nested loops in the main calculation, each having 8,192 iterations. If the inner loop is scheduled with four speculated iterations (i.e., the compiler default), then no useful work is performed for up to 8,192×8,192×4 clock cycles. For both loop attributes, the compiler adopts conservative default values, which should be lowered whenever possible. Therefore, we utilize the `[[intel::initiation_interval(R)]]` and `[[intel::speculated_iterations(S)]]` directives on relevant loops. Moreover, for applications containing loops with large iteration counts (e.g., Mandelbrot), it is advisable to primarily apply the latter directive.

Pipes: Figure 3 shows both baseline and optimized designs of KMeans. In the baseline one, kernels execute sequentially, each passing its results to the downstream kernel via global memory. In the optimized one, the dataflow to/from global memory has been limited to the `mapCenters` kernel only. By utilizing pipes, the mapping of each data point is immediately passed from `mapCenters` to `resetAccFin`, and once finalized, the resulting center is fed back from `resetAccFin` to `mapCenters`. In other words, our optimized design is capable of minimizing the data traffic to/from global memory as well as of running simultaneously both kernels. For KMeans, the usage of pipes in the optimized design results in a 510× performance improvement relative to the baseline design.

Custom prefix-sum for FPGAs: as already discussed in Section 3.3, the SYCL and CUDA versions of Where employ different prefix-sum implementations. SYCL employs that from oneDPL, which on the RTX 2080, is 50% slower than its counterpart from the CUDA library. At the time of writing, oneDPL does not provide an FPGA-optimized implementation, and hence, instead of using the



(a) Baseline: kernel communication via global memory

(b) Optimized: kernel communication via global memory and pipes

Figure 3: Designs of KMeans.

GPU-specialized one from oneDPL, we develop a custom prefix-sum in SYCL for FPGAs, shown in Listing 2. On the Stratix 10, this custom implementation exhibits up to 100× performance improvement relative to the GPU version from oneDPL.

```
1  queue.submit([&](sycl::handler &cgh) {
2      sycl::accessor results
3      {results_buff, cgh, sycl::read_only};
4      sycl::accessor prefix
5      {prefix_buff, cgh, sycl::write_only, sycl::noinit};
6
7      cgh.single_task <class exclusive_scan_id >(
8          [=]() [[intel::kernel_args_restrict,
9                  intel::max_global_work_dim(0),
10                 intel::no_global_work_offset(1)]] {
11             prefix[0] = 0;
12             #pragma unroll 2
13             for (int i = 1; i < size; i++)
14                 prefix[i] = prefix[i - 1] + results[i];
15     }); });
```

Listing 2: Custom prefix-sum in SYCL for FPGAs.

5.4 Benchmarking on GPUs and Stratix 10 FPGA

While attempting to optimize the ND-Range implementation of DWT2D, we found that its shared memory suffered from high congestion that we were unable to remove. For increasing its performance on FPGAs, our impression is that a complete device-specific algorithmic rewrite would be required. Hence, for DWT2D on FPGAs, we currently provide only a *baseline* (i.e., fully functional, but non-optimized) version.

Figure 4 shows the performance improvements attained by employing the previously discussed optimization techniques (Sections 5.1, 5.2, 5.3) on the Stratix 10 (designs built with Quartus v19.2). Such improvements are moderate in some cases, e.g, CFD FP64 (2.1×, size 1) and SRAD (2.1×, size 1), but substantial in others, e.g., Mandelbrot (∼476×, size 3) and KMeans (∼510×, size 3). For each Altis-SYCL application, we apply a different combination of techniques, e.g., in CFD FP32 and CFD FP64, we decouple memory accesses by using pipes as well as replicate compute units. By doing so, we improve the performance of CFD FP32 on the Stratix 10 by up to 4.7× (size 3) wrt. the baseline (i.e., non-optimized) version. The geometric mean of the achieved optimized vs. baseline speedups on the Stratix 10 is ∼10.7× (size 1), ∼20.7× (size 2), and ∼35.6× (size 3).

Figure 5 shows the relative speedups (vs. the Xeon CPU, whose speedup = 1.0×) achieved on GPUs and FPGAs. At smaller sizes 1 and 2, for KMeans, LavaMD, PF Naive, PF Float, Where, the Stratix 10 exhibits comparable or superior performance wrt. the RTX 2080 and even the A100 and Max 1100. However, at the larger size 3, the advantage of the Stratix 10 diminishes. Profiling reveals that this is primarily caused by the growing demand of memory bandwidth for larger sizes. For CFD FP32 and CFD FP64, FPGA designs experience poor pipeline occupancy due to stalls in global memory access. Despite the fact that these were mitigated by employing pipes and replication, their performance on the Stratix 10 is lower compared to that on CPU and GPUs. Furthermore, at size 2 and 3, NW also exhibits half the performance wrt. the CPU. This is attributed to the complex access patterns to local memory, which forces the compiler to insert arbiters that can stall execution.

5.5 Retargeting from Stratix 10 to Agilex FPGAs

We use the code optimized for Stratix 10 as baseline for Agilex (designs built with Quartus v21.2). For fitting or increasing the performance on Agilex, we adjust some optimization parameters by, e.g., increasing the work-group size (16⟶32 in SRAD), scaling up/down the compute-unit replication factors (4×⟶8× in CFD FP32, 2×⟶4× and 20×⟶25× in Where, 16×⟶8× in NW, 10×⟶4× and 50×⟶24× in both PF Naive and PF Float), reducing either the unrolling factor (30×⟶16× in both LavaMD and Raytracing) or vectorization factor (2×⟶1× in CFD FP64). Furthermore, our execution attempts of Where with size 3 resulted in crashes on Agilex, and thus, the corresponding speedups are not shown in Figure 5. Overall, the geometric mean of the achieved relative speedups (vs. the Xeon CPU, whose speedup = 1.0×) on {RTX 2080, A100, Max 1100, Stratix 10, Agilex} is respectively {5.07×, 4.91×, 6.12×, 2.16× 2.55×} (size 1), {7.00×, 9.40×, 12.44×, 2.29× 2.25×} (size 2), and {8.61×, 23.14×, 21.11×, 1.44×, 1.48×} (size 3).

By comparing the selected FPGA devices (Table 2), we note that the Stratix 10 GX 2800 provides a larger total number of resources (+47.7% ALMs, +39.3% BRAMs, +21.7% DSPs) wrt. the Agilex AGF 014. As shown in Table 3, in *most* cases, the resource utilization percentage in Agilex is thus higher than that on the Stratix 10, while for *all* cases the designs targeting Agilex achieved a higher operating frequency. In particular, for Mandelbrot, we generate three different bitstreams, each specialized to the current input size combining a different number of compute units and unrolling factors.

6 CONCLUSIONS AND FUTURE WORK

In this work, we have leveraged oneAPI to migrate and optimize the CUDA-based Altis benchmark suite into Altis-SYCL. For automating the CUDA-to-SYCL migration process, we have employed the DPC++ Compatibility Tool and resolved the tool-reported code migration warnings. For performance optimization, we have applied a number of device-specific techniques targeting preliminarily GPUs, and then using this GPU-optimized code as baseline towards FPGA-adapted code.

Altis-SYCL aims to provide a portable C++-based benchmark suite useful for characterizing the performance of HPC systems based on GPUs as well as FPGAs. For some FPGA designs of Altis-SYCL, we observed that their performance was limited by the available platform memory bandwidth. Hence, as a future work, we plan to investigate the performance of Altis-SYCL on HBM-enabled Agilex FPGAs [11].

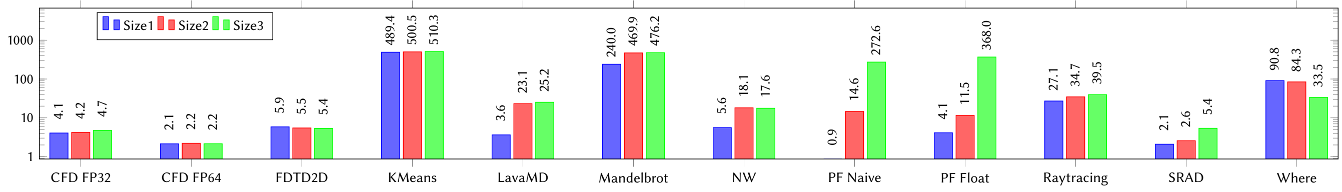


Figure 4: Speedup of the "FPGA Optimized" over "FPGA Baseline" implementations for Altis-SYCL achieved on Stratix 10.

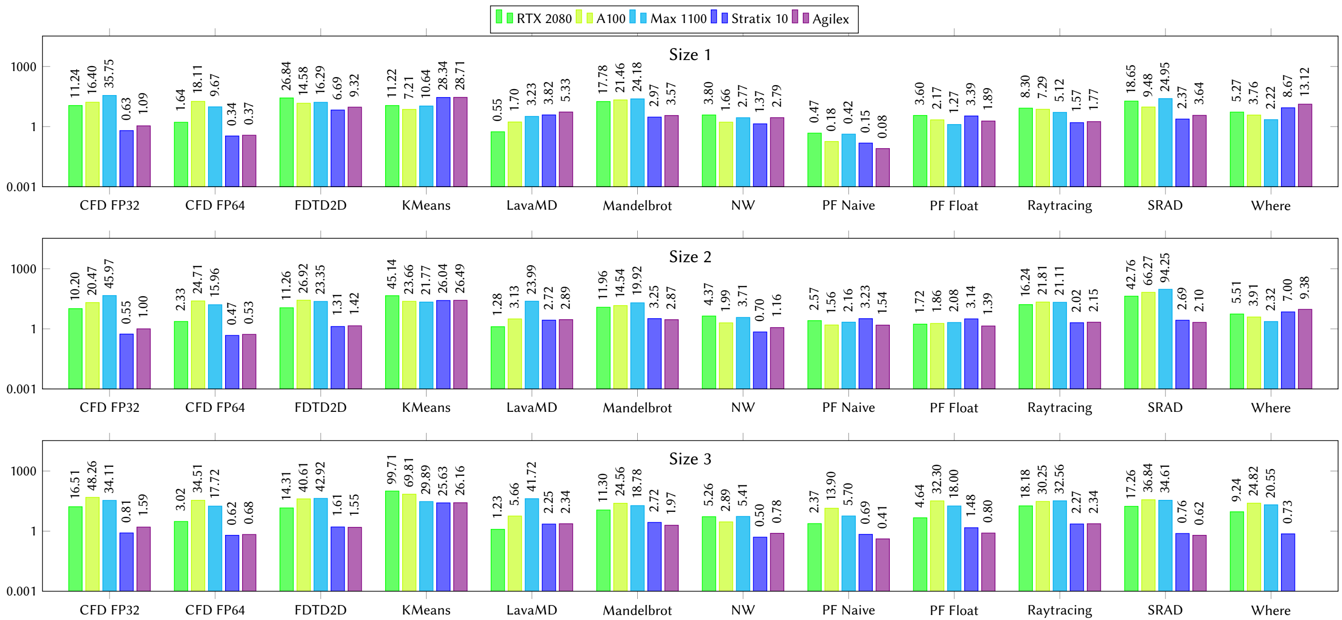


Figure 5: Relative speedup achieved on {RTX 2080, A100, Max 1100} GPUs and {Stratix 10, Agilex} FPGAs over Xeon CPU.

Table 3: Resource utilization (%) and frequency (MHz) achieved on Stratix 10 and Agilex FPGAs.

Application	ALM		BRAM		DSP		Freq (MHz)		Implementation
	Stratix 10 T: 933120	*Agilex* T: 487200	*Stratix 10* T: 11721	*Agilex* T: 7110	*Stratix 10* T: 5760	*Agilex* T: 4510	*Stratix 10*	*Agilex*	
CFD FP32	35.9%	79.7%	16.3%	43.7%	28.6%	70.4%	295.8	425.2	ND-Range & Single-Task
CFD FP64	65.7%	90.7%	30.0%	46.6%	21.7%	22.1%	256.3	373.3	ND-Range
FDTD2D	22.0%	28.6%	7.9%	15.7%	2.4%	3.1%	416.7	554.3	ND-Range
KMeans	25.3%	29.0%	7.0%	14.7%	10.8%	13.8%	347.5	370.6	Single-Task
LavaMD	76.7%	76.0%	15.0%	21.0%	22.9%	16.2%	320.8	519.2	ND-Range
Mandelbrot (size 1)	61.8%	58.8%	4.0%	14.2%	71.4%	39.7%	335.0	539.3	Single-Task
Mandelbrot (size 2)	48.4%	65.1%	3.6%	10.5%	71.2%	56.8%	379.2	539.3	
Mandelbrot (size 3)	45.3%	53.1%	3.9%	8.3%	71.1%	45.4%	375.0	544.4	
NW	45.6%	45.5%	63.9%	59.4%	1.5%	1.0%	216.0	414.1	ND-Range
PF Naive	48.3%	80.4%	26.3%	37.6%	0.0%	0.0%	107.8	108.4	Single-Task
PF Float	60.1%	67.9%	32.9%	31.2%	3.6%	4.5%	101.9	123.7	Single-Task
Raytracing	71.4%	84.2%	37.5%	43.2%	53.4%	40.0%	321.9	457.9	ND-Range
SRAD	31.9%	44.8%	46.4%	33.5%	3.5%	4.5%	280.0	463.2	Single-Task
Where	32.3%	60.2%	15.3%	51.8%	0.0%	0.0%	308.3	461.7	ND-Range & Single-Task

ACKNOWLEDGMENTS

We thank Paderborn Center for Parallel Computing (PC2) for providing access and support during our experiments on the Stratix 10 FPGA. The access to the Max 1100 GPU has been granted by Intel through the oneAPI Center of Excellence Research Award granted to Technical University of Darmstadt.

REFERENCES

[1] Najmeh Nazari Bavarsad, Hosein Mohammadi Makrani, Hossein Sayadi, Lawrence Landis, Setareh Rafatirad, and Houman Homayoun. 2021. HosNa: A DPC++ Benchmark Suite for Heterogeneous Architectures. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 509–516. https://doi.org/10.1109/ICCD53106.2021.00084

[2] BittWare. 2023. 520N Stratix 10 FPGA PCIe Board. https://www.bittware.com/products/520n

[3] Germán Castaño, Youssef Faqir-Rhazoui, Carlos García, and Manuel Prieto-Matías. 2022. Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing. *Journal of Parallel and Distributed Computing (JPDC)* 165 (2022), 120–129. https://doi.org/10.1016/j.jpdc.2022.03.017

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[5] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 63–74. https://doi.org/10.1145/1735688.1735702

[6] Atharva Gondhalekar, Thomas Twomey, and Wu-chun Feng. 2022. On the Characterization of the Performance-Productivity Gap for FPGA. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8. https://doi.org/10.1109/HPEC55821.2022.9926404

[7] Bodun Hu and Christopher J. Rossbach. 2020. Altis: Modernizing GPGPU Benchmarks. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–11. https://doi.org/10.1109/ISPASS48437.2020.00011

[8] Intel. 2019. Understanding Peak Floating-Point Performance Claims. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf

[9] Intel. 2021. Intel DPC++ Compatibility Tool. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html

[10] Intel. 2023. DPC++ Runtime - Runtime libraries for oneAPI DPC++. https://intel.github.io/llvm-docs/doxygen/classsycl_1_1__V1_1_1queue.html#abc43813da13473a0c47e92c7d334dc5a

[11] Intel. 2023. Intel Agilex 7 FPGA and SoC FPGA M-Series. https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7/m-series.html

[12] Intel. 2023. Intel Data Center GPU Max 1100. https://www.intel.com/content/www/us/en/products/sku/232876/intel-data-center-gpu-max-1100/specifications.html

[13] Intel. 2023. Intel FPGA Optimization Guide for Intel oneAPI Toolkits. https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/optimization-guide/2023-1/overview.html

[14] Intel. 2023. oneAPI: A New Era of Heterogeneous Computing. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

[15] Intel. 2023. oneMKL - Data Parallel C++ Developer Reference: oneapi::mkl::rng::philox4x32x10. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-dpcpp/2023-1/oneapi-mkl-rng-philox4x32x10.html

[16] Intel. 2023. Samples for Intel oneAPI Toolkits. https://github.com/oneapi-src/oneAPI-samples

[17] Intel. 2023. Why oneAPI? https://www.oneapi.io

[18] Ryuta Kashino, Ryohei Kobayashi, Norihisa Fujita, and Taisuke Boku. 2022. Multi-Hetero Acceleration by GPU and FPGA for Astrophysics Simulation on OneAPI Environment. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia)*. ACM, 84–93. https://doi.org/10.1145/3492805.3492817

[19] Khronos Group. 2023. SYCL. https://www.khronos.org/sycl

[20] Marius Meyer, Tobias Kenter, and Christian Plessl. 2020. Evaluating FPGA Accelerator Performance with a Parameterized OpenCL Adaptation of Selected Benchmarks of the HPCChallenge Benchmark Suite. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 10–18. https://doi.org/10.1109/H2RC51942.2020.00007

[21] NVIDIA. 2023. CUDA Toolkit v12.1.1: cuRAND. https://docs.nvidia.com/cuda/curand/host-api-overview.html#generator-types

[22] Paul Sathre, Atharva Gondhalekar, and Wu-chun Feng. 2022. Edge-Connected Jaccard Similarity for Graph Link Prediction on FPGA. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–10. https://doi.org/10.1109/HPEC55821.2022.9926326

[23] Christopher Siefert, Stephen L. Olivier, Gwendolyn Voskuilen, and Jeffrey Young. 2022. MultiGrid on FPGA Using Data Parallel C++. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 907–910. https://doi.org/10.1109/IPDPSW55747.2022.00147

[24] Terasic. 2023. DE10-Agilex Development Board. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=115&No=1252#contents

[25] Yong Wang, Yongfa Zhou, Qi Scott Wang, Yang Wang, Qing Xu, Chen Wang, Bo Peng, Zhaojun Zhu, Katayama Takuya, and Dylan Wang. 2021. Developing medical ultrasound beamforming application on GPU and FPGA using oneAPI. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 360–370. https://doi.org/10.1109/IPDPSW52791.2021.00064

[26] Alberto Zeni, Kenneth O'Brien, Michaela Blott, and Marco D. Santambrogio. 2021. Optimized Implementation of the HPCG Benchmark on Reconfigurable Hardware. In *Euro-Par 2021: Parallel Processing*. Springer, 616–630. https://doi.org/10.1007/978-3-030-85665-6_38