Porting numerical integration codes from CUDA to oneAPI: a case study*

Ioannis Sakiotis[1 [0000000219880314]], Kamesh Arumugam[2 [0000000264826237]], Marc Paterno[1,3 [0000000308088388]], Desh Ranjan[1 [0000000282987093]], Balša Terzić[1 [0000000296468155]], and Mohammad Zubair[1 [0000000254491779]]

[1] Old Dominion University, Norfolk, VA 23529, USA
[2] NVIDIA, Santa Clara, CA 95051-0952, USA
[3] Fermi National Accelerator Laboratory, Batavia, IL 60510

Abstract. We present our experience in porting optimized CUDA implementations to oneAPI. We focus on the use case of numerical integration, particularly the CUDA implementations of PAGANI and m-Cubes. We faced several challenges that caused performance degradation in the oneAPI ports. These include differences in utilized registers per thread, compiler optimizations, and mappings of CUDA library calls to oneAPI equivalents. After addressing those challenges, we tested both the PAGANI and m-Cubes integrators on numerous integrands of various characteristics. To evaluate the quality of the ports, we collected performance metrics of the CUDA and oneAPI implementations on the Nvidia V100 GPU. We found that the oneAPI ports often achieve comparable performance to the CUDA versions, and that they are at most 10% slower.

1 Introduction

Historically, general-purpose GPU programming has been characterized by divergent architectures and programming models. A lack of widely adopted common standards led to the development of different ecosystems comprised of compilers and tools that were practically exclusive to specific GPU architectures. Most importantly, the emergent architectures themselves were not compatible with all ecosystems. Portability could only be achieved through the maintenance of multiple code bases. Traditionally, the proprietary CUDA programming model has been the most popular but is exclusively targeted to Nvidia GPUs.

In the absence of universally adopted standards, a viable solution for achieving general portability is to rely on platform-agnostic programming models that target multiple architectures via a unifying interface. This enables the execution of a single code base across various architectures. These programming models would ideally enable the utilization of platform-specific low-level features on their native hardware. This would allow highly-optimized implementations in such portable programming models to remain competitive with platform-specific

* code available at https://github.com/marcpaterno/gpuintegration

alternatives. Without these capabilities, use cases with extreme performance requirements would disqualify the use of such portable models.

The need for performant multi-platform execution is only increasing with the emergence of exascale supercomputers such as Frontier and Aurora that do not carry Nvidia GPUs. Projects requiring computing cores at that scale must develop new software solutions compatible with non-Nvidia GPUs or port existing CUDA implementations without significant loss of performance.

Portable programming models such as RAJA, Kokkos, and oneAPI have been in development and are already available for use. These portable alternatives lack maturity when compared to proprietary alternatives. As such, applications requiring portable solutions must be evaluated to quantify any necessary concessions.

In this paper, we discuss the porting process of two numerical integration implementations, PAGANI and m-Cubes, from CUDA to Data Paralllel C++ (DPC++), which is oneAPI's SYCL implementation. The oneAPI ecosystem provides a suite of compilers, libraries, and software tools, including Intel® DPC++ Compatibility Tool (DPCCT), that automates the majority of the porting process. Reliance on the C++ and SYCL standards as well as the capability to quickly port large CUDA implementations, places oneAPI at the forefront of the portability initiative.

We faced challenges during the porting process due to the lack of support for certain libraries utilized by the CUDA implementation. For example, the CUDA implementation of PAGANI uses the Nvidia Thrust library to perform common parallel operations on the host side, such as inner product and min-max. Even though there is a multitude of library options in oneAPI, we encountered difficulties with the DPCCT mapping of Nvidia Thrust library calls, which were not fully supported on all backends.

We also observed performance degradation for the ported oneAPI implementations. We conducted numerous experiments with integrands of various characteristics to identify the issues. Most of these issues pertained to optimization differences between the NVCC and Clang compilers, and time differences when executing mathematical functions. After addressing these challenges, the oneAPI ports were at most 10% slower than the optimized CUDA versions. We observe that the cases with the highest performance penalties for the oneAPI ports, require significantly more registers than the CUDA originals. This decreases the occupancy in the oneAPI implementation and causes performance degradation. When the number of registers is similar to the CUDA version, we observe penalties lower than 5%.

The remainder of this paper is structured as follows. First, we provide background information on oneAPI and other portability solutions in section 2. Then, we discuss the two numerical integration CUDA implementations in section 3. Section 4 details the porting process and challenges we faced using DPCCT and the oneAPI platform. In section 5, we present a performance comparison of the CUDA and oneAPI implementations of PAGANI and m-Cubes. We finish in section 6 with a discussion of our conclusions regarding the oneAPI platform's

viability and ease of use. We demonstrate that the oneAPI implementation does not induce significant performance penalties and that it is a viable platform for attaining performance on Nvidia GPUs.

2 Background

There are multiple programming models targeting different architectures. Among the most prominent, are OpenCL [24], [16], OpenACC [5], OpenMP [1], RAJA, Alpaka [30], and Kokkos [10]. The Khronos group was the first to address portability by developing the OpenCL standard to target various architectures. The same group later followed with the SYCL standard. SYCL is a higher-level language that retained OpenCL features but significantly improved ease of use with the utilization of C++ and the adoption of a single-source model. There are multiple implementations of SYCL such as DPC++, ComputeCpp, HipSYCL, and triSYCL [28]. DPC++ is conformant to the latest SYCL and C++ standards and is integrated into the oneAPI ecosystem [2].

2.1 oneAPI and SYCL

oneAPI provides a programming platform with portability across multiple architectures at the core of its mission. Intel's implementation of oneAPI includes an oneAPI Base Toolkit that includes various tools along with the DPC++ language which was based on the SYCL and C++ standards [8]. The reliance on these open standards that are intended to evolve over time is one of the most attractive features of DPC++. Such evolution is facilitated by DPC++ extensions with various features that can be later introduced to the standards after periods of experimentation. Such examples include the use of *Unified Memory* and *filtered Device selectors*, which were missing from SYCL 1.2.1 but were later included in the SYCL 2020 standard. DPC++ achieves execution platform portability through its use of SYCL and various backends (implemented as shared libraries) that interface with particular instruction sets such as PTX for Nvidia GPUs and SPIR-V for Intel devices. It is worth noting that there is no reliance on OpenCL, which is instead one of several available backends. As such, DPC++ implementations can target various CPUs, GPUs, and FPGAs. This is a similar approach to Kokkos, Alpaka, and RAJA.

2.2 CUDA-backend for SYCL

While CUDA is the native and most performant programming model for Nvidia GPUs, Nvidia provided support to the OpenCL API [25]. As a result, non-CUDA implementations could be executed on Nvidia GPUs. The ComputeCpp implementation of SYCL by CodePlay, provided such functionality through OpenCL, but its performance was not comparable to native CUDA as not all functionality was exposed [3].

As such, CodePlay developed the CUDA backend for DPC++, which is part of the LLVM compiler project. CUDA support is not enabled by default and is at an experimental stage. To enable the backend, we must build the LLVM compiler project for CUDA. This can be achieved through easy-to-follow instructions that involve CUDA-specific flags, and the use of *clang++* instead *dpcpp* to compile source code. As a result, DPC++ code can generate PTX code by using CUDA directly instead of relying on the OpenCL backend. This approach not only enables the use of Nvidia libraries and profiling tools with DPC++ but also the capability to theoretically achieve the same performance as CUDA.

2.3 Related Work

The oneAPI programming model may not be as mature as CUDA but the literature already includes several examples of utilizing DPC++. The authors of [12] validated the correctness of a DPC++ tsunami simulator ported from CUDA. A Boris Particle Pusher port from an openMP version was discussed in [27], where a DPC++ implementation was 10% slower than the optimized original. In [14], CUDA and DPC++ implementations of a matrix multiplication kernel were compared on different matrix sizes; the execution time on an Nvidia GPU was slower with DPC++ code by 7% on small problem sizes but as much as 37% on larger ones. On the contrary, [15] and [13] included experiments where a DPC++ biological sequence alignment code showed no significant performance penalty compared to CUDA, and even a case of 14% speedup. Spare matrix-vector multiplication kernels and *Krylov* solvers in [26] reached 90% of a CUDA version's bandwidth. There were also cases with non-favorable performance for DPC++ ports. In [17] a bioinformatics-related kernel performed twice as fast in CUDA and HIP than in DPC++. In [11] DPC++ versions generally reported comparable performance to CUDA but there were multiple cases where the penalty ranged from $25 - 190\%$.

There seems to be a deviation in the attainable performance. This is reasonable due to the variety of applications and the relatively early stage of development for the oneAPI ecosystem. We also expect that the level of optimization in CUDA implementations is an important factor. In our experience, highly optimized codes typically yield performance penalties in the range $(5 - 10\%)$. There are multiple cases displaying approximately 10% penalty compared to native programming models. This indicates that DPC++ can achieve comparable performance to CUDA, though careful tuning and additional optimizations may be needed.

3 Numerical Integration Use Case

Numerical integration is necessary for many applications across various fields and especially physics. Important examples include the simulation of beam dynamics and parameter estimation in cosmological models [21] [6] [9]. Even ill-behaving integrands (oscillatory, sharply peaked, etc.) can be efficiently integrated with

modest computational resources, as long the integration space is low dimensional (one or two variables). On the contrary, solving medium to high-dimensional integrands is often infeasible on standard computing platforms. In such cases, we must execute on highly parallel architectures to achieve performance at scale. There are a few GPU-compatible numerical integration algorithms [22] [23] [7] [19] [29]. Unfortunately, exploration of execution-platform portability has been limited, with CUDA being the most common choice. Since CUDA is a proprietary language, such optimized implementations cannot be executed on non-Nvidia GPUs. To our knowledge, the only mentions of potential portability in numerical integration libraries are found in [22] where a Kokkos implementation of the PAGANI integrator is briefly mentioned to be in development and in [23] which compares the CUDA implementation of m-Cubes with an experimental Kokkos version.

3.1 PAGANI

PAGANI is a deterministic quadrature-based algorithm designed for massively parallel architectures. The algorithm computes an integral by evaluating the quadrature rules, which are a series of weighted summations of the form $\sum_{i=1}^{f_{eval}} w_i \cdot f(x_i)$. The computation involves an integrand function f which we invoke at the d-dimensional points x_i. Each point x_i has a corresponding weight w_i and there are f_{eval} such points in the summation. PAGANI computes an initial integral and error estimate, and it progressively improves its accuracy until reaching a user-specified threshold. The accuracy improvement is achieved by applying the quadrature rules in smaller regions of the integration space and accumulating those values to get the integral estimate.

The most computationally intense kernel of PAGANI is the EVALUATE method (listed in Algorithm 2 of [22]) which consistently takes more than 90% of total execution time. Its function is to compute an integral/error estimate for each region and select one of the dimensional axes for splitting. As such, it can be viewed as the core of PAGANI, both from an algorithmic and performance standpoint. For the remainder of this paper, we will refer to this method as PAGANI-KERNEL.

In PAGANI-KERNEL, each thread-group processes a different region and uses all threads in the group to parallelize the integrand function evaluations. The function evaluations are then accumulated accumulated through a reduction operation. Since all threads in a thread-group operate on the same region, we can store region data in shared memory. The same data, which is needed for each function evacuation is broadcast to all threads, avoiding repeated access to the slower global memory. There are additional read-only arrays needed for function evaluations, which are stored in global memory due to their larger size. Finally, thread zero of each group writes the computed integral and error estimate of the region in the corresponding output arrays.

The CUDA implementation was optimized for the Nvidia *V100* GPU. The kernel is launched in groups of 64 threads and the functions evaluations are performed in iterated in a strided fashion. This allows the threads to coalesce accesses to the read-only arrays in global memory. For those reads, the kernel

relies on the "ldg" intrinsic, suggesting to the compiler their placement in the read-only cache.

3.2 *m*-Cubes

m-Cubes is a probabilistic Monte Carlo algorithm based on the VEGAS integrator [20]. It operates by randomizing the sample generation across the integration space to solve integrands and relies on the standard deviation of the Monte Carlo estimate to produce error estimates for the computation. Just like VEGAS, *m*-Cubes utilizes importance and stratified sampling to accelerate the Monte Carlo rate of convergence. The algorithm partitions the integration space into m sub-cubes that are sampled separately. The sub-division resolution is dictated by a user-specified number of samples per iteration.

While the main kernel of *m*-Cubes, which we will refer to as MCUBES-KERNEL is detailed in [23]), we describe some important characteristics. Each thread is assigned a number of sub-cubes and processes them serially. During the sampling of those cubes, the threads randomly generate a series of d-dimensional points within certain bin boundaries and evaluate an integrand f at those points. The magnitude of each function evaluation must be stored in d corresponding memory locations that represent the d bins used to generate the point. The kernel uses atomic addition to perform these memory writes because there are possible collisions due to a lack of $1-1$ mapping between bins and threads. Once the threads in a group have evaluated all their points across all their assigned sub-cubes, a reduction operation accumulates the function evaluations within a thread-group. Then, the results of each all thread-groups are accumulated through atomic addition. This provides an integral and error estimate.

The CUDA implementation was optimized for the *V100* GPU. The kernel consisted of 128 threads per block and utilized 500 bins per dimensional axis. The reduction operates on local memory and utilizes warp-level primitives, though limited shared memory is used to accumulate the values from the different warps. The first warp, completes the final reduction in the thread-group through warp-level primitives.

4 Porting Process

The maturity of the CUDA programming model along with the more widespread utilization of highly performant Nvidia GPUs make CUDA an intuitive choice for high-performance applications. As such, PAGANI and *m*-Cubes were designed and optimized for CUDA on a V100 Nvidia GPU [22] [23]. This makes DPCCT the most appropriate tool to facilitate the porting process from CUDA to DPC++.

4.1 Intel® DPC++ Compatibility Tool

DPCCT is intended to automate the majority of CUDA code migration to DPC++, instead of performing a total conversion [4]. In our experience as well

as those reported in [14], [18] and many others, DPCCT functions exactly as intended. An easy-to-complete conversion process requires few manual code insertions. When manual editing is needed, DPCCT displays helpful suggestions in the form of comment blocks to guide the user. There were certain code segments that were functional but needed simple fixes to improve performance. e.g. use of 3D *nd_item* instead of 1D equivalent. In our experience, those cases were few and we expect that such effects will be less pronounced as oneAPI and DPCCT evolve. Expert users are anticipated to produce higher-quality implementations than automated tools, but even then DPCCT greatly facilitates the porting process by automating the tedious and often error-prone translation of API calls and indexing schemes.

4.2 Challenges

Errors in Automated Code Migration A source of errors for DPCCT generated code was our use of C++ structures to encapsulate input/output data that resided in the device memory space. We used C++ to automate allocation, deallocation, and initialization for much of the data needed by our CUDA kernels. The constructors and destructors of these non-trivial C++ structures included calls to the CUDA API e.g (cudaMalloc, cudaFree), while member functions involved host-side processing and even invoked other CUDA kernels to perform parallel operations.

DPCCT translated the API calls from CUDA to SYCL without errors for all of our C++ structures. The problem arises when passing members of those structures as parameters to the lambda expressions that define the parallel code. The SYCL standard requires that all objects copied between host and device are trivially copyable. Since the lambdas will be copied to the device, any objects captured by the lambda must be trivially copayble as well. Our C++ structures are not trivially-copyable because they have user-defined destructors to free their device-allocated data. Even though we do not use the objects themselves in the parallel code, but only to conveniently pass their members as parameters, they are captured nonetheless and cause a static assert error.

We demonstrate this in Listing 1.1, where the SYCL wrapper-function brings the *regions* object into scope as a pointer (line 8, causes the pointer to be copied instead of the object itself. This is in contrast to the CUDA wrapper where there is no capture and *regions* is passed by reference (line 1), while the *leftcoord* member, which is a pointer, is passed by value (line 4) to the kernel. In SYCL, passing the C++ object as a pointer removes the trivially-copyable related compilation error, but accessing the pointer in the parallel code causes an *illegal access* run-time error (line 19). To solve this issue, we must store any data that we want to be captured by our lambda, into scope-local variables (line 10).

```
1 void cuda_wrapper(const Sub_regions& regions){
2     const size_t nBlocks = regions.size;
3     const size_t nThreads = 64;
4     kernel<<<nBlocks, nThreads>>>(regions.leftcoord);
5     cudaDeviceSynchronize();
```

```
}

void sycl_wrapper(Sub_regions* regions){
    sycl::queue q(sycl::gpu_selector());
    T* leftcoord = regions->leftcoord;
    const size_t nBlocks = regions->size;
    const size_t nThreads = 64;

    q.submit([&](sycl::handler& cgh){
        cgh.parallel_for(
            sycl::nd_range<1>(sycl::range<1>(nBlocks) * sycl
    ::range<1>(nThreads),
                sycl::range<1>(nThreads)),
            [=](sycl::nd_item<1> item_ct1){
                double x = regions->leftcoord[0]; //run-time
    error: illegal access
                double y = leftcoord[0]; //ok, local var
    leftcoord captured
            });
        });
    q.wait_and_throw();
}
```

Listing 1.1: Kernel Launch

Another issue we encountered in the DPCCT converted code was the incorrect conversion of atomic addition in our parallel code. DPCCT converted the *atomicAdd* CUDA function call to *dpct::atomic_fetch_add*. The use of this particular function triggers an *unresolved extern* error for the *__spirv_AtomicFAddEXT* function. This is an improper mapping for atomic addition from the DPCCT implementation to the CUDA backend; the same command works on Intel devices. We resolve this problem by using the correct atomic function directly from the SYCL namespace (see Listing 1.2).

```
//ptxas fatal   : Unresolved extern function
dpct::atomic_fetch_add<double,
    sycl::access::address_space::generic_space>(
    &result_dev[0], fbg);

//functional replacement
auto v = sycl::atomic_ref<double,
    sycl::memory_order::relaxed,
    sycl::memory_scope::device,
    sycl::access::address_space::global_space>(result_dev[0])
    ;
v += fbg;
```

Listing 1.2: Atomic Addition

Porting Issues with Nvidia Thrust Library PAGANI uses Thrust to perform common parallel operations on the host side, such as reduction, dot-product, prefix sum, and finding the minimum/maximum value in a list. DPCCT successfully automates the translation of these Thrust library calls to SYCL, mainly through the use of equivalent functions in the *DPCT* namespace. The one exception where DPCCT fails to provide a function call is the *minmax_element* function, where an appropriate warning for the unsuccessful code migration is provided. Instead, we used the *min_max* function from the oneMKL library's *Summary Statistics* domain. This function worked on Intel GPUs and CPUs but had no mapping for the CUDA backend and yielded an *undefined reference* error. To solve this issue, we used the *iamax* and *iamin* routines from the oneMKL library's BLAS domain.

We faced a similar CUDA-backend mapping issue with the *dpct::inner_product* method, which caused a *no matching function* compilation error. We found the *row_major::dot* method as an alternative in the oneMKL library but it was not implemented for the CUDA backend. Instead, we the equivalent routine in the *column_major* namespace worked for both Intel and NVIDIA devices. The only limitation of the oneMKL routine was that the dot-product operation requires the two input lists to be of the same type. In contrast, the Thrust routine allows the user to compute the dot-product between floating point and integer lists. In most cases, any performance impact would be negligible, but the impact on memory can be critical for PAGANI which is a memory hungry algorithm that uses memory-saving routines when the available memory is close to being exhausted. Using floating-point type instead of integer-types for certain lists, can trigger costly memory-saving routines in the oneAPI implementation sooner than the CUDA original and slightly degrade performance.

Performance Degradation We encountered more difficulties when attempting to achieve comparable performance to the original CUDA implementations. The parallel codes for SYCL and CUDA were near-identical, yet we found differences in terms of register pressure and shared memory allocation size. These factors contributed to degraded performance in the SYCL implementations, with execution times often being more 50% larger than the CUDA originals. The critical optimization that on average minimized execution times to within 10% of the CUDA implementation, was manual loop-unrolling but only after setting the inline-threshold to $10,000$ during compilation. In our initial SYCL versions, the default inline-threshold prevented code inlining and loop-unrolling even when manually set. While the increased inline-threshold further increased register usage, it allowed better optimizations by the compiler and better pipeline utilization which improved performance.

Another method to limit register usage in SYCL, though to a lesser extent than code inlining, was the use of one-dimensional *nd_item* objects (used for indexing and coordinating threads in a group). DPCCT defaults to using 3D *nd_item* even when converting CUDA code that does not utilize multi-dimensional indexing. This is the case for both PAGANI and m-Cubes which

organize multi-dimensional data in one-dimensional lists and thus have no need for 2D grids. Using the one-dimensional *nd_item* in m-Cubes decreased register usage by 10 and yielded small $(1-2\%)$ but consistent performance improvement. The same technique did not impact performance in PAGANI.

Additionally, we found that using a custom function to perform work-group reduction through shared memory was faster than the built-in *reduce_over_group*. Computing a six-dimensional integral where PAGANI used the built-in reduction, increased the register count from 100 to 132 and the execution time from 757 ms to 778 ms.

Another challenge in our attempt to achieve comparable performance to CUDA was deviations in the performance of SYCL and CUDA mathematical functions. There is no guarantee that the mathematical functions in SYCL have the same implementations as the functions in the CUDA Math API. In some cases, we must use different functions (e.g. *sycl::pow* instead *pow*) which could make small deviations unavoidable. Exponential functions displayed comparable performance on benchmark kernels. On the contrary, we observed a slowdown of various degrees in SYCL when using *power* or trigonometric functions. This is most likely attributed to the compilers utilizing different optimizations. We did not use any *fast-math* flags, since high accuracy is critical in numerical integration use cases.

Finally, the use of atomic addition in m-Cubes caused orders of magnitude slowdown on both the MCUBES-KERNEL and benchmark kernels. This was attributed to the lack of an architecture-specific flag that must be set to enable efficient atomics when supported. After setting the Volta architecture flag, atomic addition was as performant as in the native CUDA implementation.

Software Engineering Issues We faced non-intuitive compilation errors due to our use of the Catch2 testing framework. Header inclusion for the oneDPL library caused compilation errors only for testing code. We observed demo programs that did not use catch2 headers could compile and execute without issue. On the contrary, codes such as the example in Listing 1.3 causes compilation error. Removing the oneDPL header at line 4 eliminates the issue in this example. The same issue occured when replacing the headers at lines $3-4$ with *dpct/dpct.hpp* and *dpct/dpl_utils.hpp* which were the headers that DPCCT automatically included to use the parallel policies of standard library functions such as *std::exclusive_scan* and *std::reduce*.

```
1 #define CATCH_CONFIG_MAIN
2 #include "catch2/catch.hpp"
3 #include <oneapi/dpl/execution>
4 #include <oneapi/dpl/algorithm>
5
6 //error: ranges/nanorange.hpp:3303:46: error: reference to '
    match_results' is ambiguous
7
8 TEST_CASE("TEST HEADER INCLUSION")
9 {
```

```
  sycl::queue q;
}
```

Listing 1.3: Header Inclusion Issues with Catch2 Testing Framework

```
# For Intel P60 GPU
find_package(MKL REQUIRED)
add_executable(exec_name filename.cpp)
target_link_libraries(exec_name PUBLIC MKL::MKL_DPCPP)

# For CUDA backend
//we must store the path to oneMKL library in the CMake
    variable ONEMKL_DIR
//store GPU architeture in CMake variable TARGET_ARCH
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fsycl -fsycl-targets
    =nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-
    arch=${TARGET_ARCH}")
add_executable(exec_name filename.cpp)
target_link_directories(exec_name PUBLIC "${ONEMKL_DIR}")
target_compile_options(exec_name PRIVATE "-lonemkl")
```

Listing 1.4: Using CMake

Utilization of the Catch2 testing framework and CMake was largely successful but more error-prone when building for the CUDA backend. We had to include separate CMake commands and flags when building for Nvidia GPUs instead of utilizing CMake packages which were viable when building for other backends. As illustrated in Listing 1.4, we needed additional flags for the CUDA backend (line 9). Using the oneMKL library, required the *-lonemkl* flag to CMake's *target_compile_options* and the oneMKL location to the *target_link_directories* command, which had to be manually set. Building for the *P630* Intel GPU was simpler. We did not need any flags to compile a target, and the oneMKL the CMake package made the utilization of the library less verbose. Our supplement of extra flags for the CUDA backend does not follow standard CMake practices. As support for the CUDA-backend exits its experimental stage, we expect such software engineering issues will be less pronounced.

5 Experimental Results

We conducted a series of experiments to evaluate the performance and correctness of the oneAPI ports relative to the optimized CUDA implementations of PAGANI and m-Cubes. We used a single node with a V100 Nvidia GPU and a 2.4 GHz Intel Xeon R Gold 6130 CPU. We also used the Devcloud environment to verify that the DPC++ implementations were portable and could be executed on a P630 Intel GPU. Due to the V100 GPU having significantly more computing cores than the P630, we do not make any performance comparisons between the two GPUs. Instead, we focus on the attainable performance of DPC++ on NVIDIA hardware.

When executing the CUDA implementations, we used gcc 8.5 and CUDA 11.6. For the CUDA-backend execution, we used the same environment but compiled with clang 15, an inline threshold of 10000, and the following compilation flags: "-fsycl -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend –cuda-gpu-arch=sm_70". We verified the correctness of our ports, by comparing the results on both the Nvidia (V100) and Intel (P630) GPUs, to the results generated by the CUDA originals on a V100 GPU.

In terms of evaluating performance, we chose the same benchmark integrands originally used to evaluate PAGANI and m-Cubes in [22] and [23]. These functions belong to separate integrand families with features that make accurate estimation challenging. We list those integrands in equations 1 to 6. All experiments use the same integration bounds $(0, 1)$ on each dimensional axis. Similar to [22] and [23], we perform multiple experiments per integrand.

We deviate from [22] and [23] in that we do not execute the PAGANI and m-Cubes methods in their entirety. Instead, we execute their main kernels PAGANI-KERNEL and MCUBES-KERNEL, which is where more than 90% of execution is spent. With this approach, we can evaluate the effectiveness of each programming model in terms of offloading workloads to the device. It allows us to separate kernel evaluation from memory management operations (allocations, copies, etc.) and library usage. This comparison of custom kernel implementations is a better indicator of performance implications when porting CUDA codes to DPC++.

$$f_{1,d}(x) = \cos\left(\sum_{i=1}^{d} i\, x_i\right) \tag{1}$$

$$f_{2,d}(x) = \prod_{i=1}^{d} \left(\frac{1}{50^2} + (x_i - 1/2)^2\right)^{-1} \tag{2}$$

$$f_{3,d}(x) = \left(1 + \sum_{i=1}^{d} i\, x_i\right)^{-d-1} \tag{3}$$

$$f_{4,d}(x) = \exp\left(-625 \sum_{i=1}^{d} (x_i - 1/2)^2\right) \tag{4}$$

$$f_{5,d}(x) = \exp\left(-10 \sum_{i=1}^{d} |x_i - 1/2|\right) \tag{5}$$

$$f_{6,d}(x) = \begin{cases} \exp\left(\sum_{i=1}^{d} (i+4)\, x_i\right) & \text{if } x_i < (3+i)/10 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

5.1 Offloading Mathematical Computations to Kernels

A critical stage in PAGANI-KERNEL and M-CUBES-KERNEL is the invocation of the integrand at various d-dimensional points. Integrands with trigonometric or

exponential functions and table look-ups will have larger execution times compared to other simple integrands that only contain basic mathematical operations. To attain satisfactory performance, both the invocation of the integrand functions and the remaining operations within the kernels must achieve comparable performance to the CUDA implementation.

We tested the efficiency of the integrand oneAPI implementations with a simple kernel that performs a series of invocations on many d-dimensional points. The points are randomly generated on the host and then copied to device memory. Each thread invokes the integrand serially 1 million times and writes its accumulated results to global memory. Writing the results prevents the NVCC and Clang compilers from disregarding the integrand computations due to optimization.

We first tested simple integrands that contained only a particular function such as *sin, pow, powf, sycl::exp, sycl::pow, sycl::pown*. We invoked these mathematical functions with d arguments that comprise each d-dimensional point. We did not use fast-math flags as accuracy is critical in numerical integration. We observed small but consistent penalties of at most 2% when invoking the *power* and *exponential* functions. On the contrary, trigonometric functions are approximately 40% slower on the CUDA backend.

We performed the same experiment on the six benchmark integrands for dimensions 5 to 8. We summarize the results in Table 1. The timings in CUDA and oneAPI columns are the means of 10 kernel executions per integrand. The ratio of those timings shows that the oneAPI version is at most 4% slower. The largest penalty is observed in the $f1$ integrand which makes use of the *cos* function. The remaining integrands only make use of *exponential* and *power* functions and yield small penalties.

These experiments on the execution time of the integrand invocations demonstrate that the user-defined computations do not display significant performance penalties. The one exception is the extended use of trigonometric functions. None of the benchmark integrands make extended use of trigonometric functions ($f1$ has one call to *cos* per invocation). As such, we do not expect any slowdown larger than 5% in either PAGANI or m-Cubes to be attributed to the integrand implementations.

5.2 Benchmark Integrands Performance Comparison

Another set of experiments involved the invocation of the PAGANI-KERNEL and MCUBES-KERNEL on the benchmark integrands. To address different degrees of computational intensity, we vary the number of thread-blocks used to launch the kernels. For the MCUBES-KERNEL, we achieve this effect by varying the required number of samples per iteration in the range $(1e8, 3e9)$. This leads to different block sizes per kernel. For PAGANI-KERNEL, the number of thread blocks corresponds to the number of regions being processed. We perform high-resolution uniform splits to generate region lists of different sizes and supply them to the PAGANI-KERNEL for evaluation.

Table 1: mean (μ) and standard deviation (σ) of execution times for invoking $5-8D$ benchmark integrands

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
f1	1866.4	1952.4	13.3	21.4	1.04
f2	8413.9	8487.3	5012.5	5042.9	1.009
f3	1812.4	1828.3	18.5	27.1	1.009
f4	11416.1	11410.1	2184.9	2148.1	0.99
f5	634.3	654.4	73.5	67.3	1.03
f6	300.4	300.8	32.05	32.6	1.001

We report the penalty of using oneAPI for the benchmark integrands, in the ratio columns of Tables 2 and 3. We used four thread-block sizes for each integrand for the kernel executions. Each kernel configuration (number of thread groups) was repeated 100 times to provide a statistical mean and standard deviation for the execution times.

Across our experiments, the average execution time ratio ($\frac{oneAPI}{CUDA}$) is in the range $(0-10\%)$. The $f2$ and $f4$ integrands which make repeated use of the *power* function display the largest performance penalties for both PAGANI and m-Cubes. It is worth noting that both $f2$ and $f4$ display the largest execution times among the benchmark integrands for both integrators.

Table 2: m-Cubes: mean (μ) and standard deviation (σ) of execution times for 8D benchmark integrands

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
f1	286.7	286.7	2.1	0.9	1.0
f2	402.1	443.1	2.6	0.9	1.1
f3	284.5	285.8	1.6	1.4	1.0
f4	385.7	423.5	2.4	0.5	1.1
f5	284.3	285.9	2.1	1.7	1.0
f6	283.8	285.4	1.9	1.6	1.0

5.3 Simple Integrands Performance Comparison

In addition to the benchmark integrands, we also evaluate integrands that only perform a summation of the arguments ($\sum_{i=1}^{d} x_i$) where d is the number of

Table 3: PAGANI: mean (μ) and standard deviation (σ) of execution times for 8D benchmark integrands

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
f1	172.3	177.5	0.9	1.2	1.02
f2	1500.4	1651.0	0.3	2.1	1.1
f3	286.4	290.7	0.8	0.4	1.01
f4	1434.7	1524.9	0.4	1.9	1.06
f5	166.5	170.7	0.6	0.4	1.03
f6	136.8	139.4	0.4	0.2	1.02

dimensions. This avoids any bias in the comparison by avoiding mathematical functions that could either call different implementations, cause differences in register usage or lead to different optimizations. The ratios in Tables 4 and 5, display timings on addition integrands for dimensions five to eight. Once more, we observe penalties smaller than 10% and for both integrators these penalties decrease on higher dimensionalities.

Table 4: m-Cubes: mean (μ) and standard deviation (σ) of execution times for addition integrands ($\sum_{i=1}^{d} x_i$)

id	μ CUDA (ms)	μ oneAPI (ms)	σ CUDA	σ oneAPI	$\frac{\mu\ oneAPI}{\mu\ CUDA}$
5D	206.1	214.5	2.1	1.7	1.04
6D	214.1	217.2	2.2	1.0	1.01
7D	234.1	235.2	1.8	0.9	1.005
8D	284.7	285.7	1.9	1.9	1.005

5.4 Factors Limiting Performance

Both PAGANI-KERNEL and MCUBES-KERNEL, are compute bound, performing thousands of computations for each byte of accessed memory. The number of registers per thread is a factor limiting the number of concurrent threads that can be executed; the amount of shared memory and registers per thread limit warp/work-group occupancy, which in turn degrades performance.

In most cases, the oneAPI implementations assigned more registers to each thread compared to their CUDA equivalents. We illustrate the magnitude of this difference in registers per thread in Figures 1 and 2. We observe the largest difference in integrands $f2$ and $f4$, which make extended use of the *power function*.

Table 5: PAGANI: mean (μ) and standard deviation (σ) of execution times for addition integrands ($\sum_{i=1}^{d} x_i$)

id	CUDA (ms)	oneAPI (ms)	Std. CUDA	Std. oneAPI	$\frac{oneAPI}{CUDA}$
5D	1.5	1.7	0.05	0.06	1.1
6D	24.8	26.7	0.3	1.4	1.1
7D	129.8	131.6	0.7	0.2	1.01
8D	137.4	137.6	1.3	1.0	1.001

It is the same functions that display the two largest execution time penalties for the benchmark integrands in Tables 2 and 3.

We observe a similar pattern on the simple addition integrands (Table 4 and 5). In those cases, there are no mathematical functions (*pow*, *exp*, etc.) and the integrands only perform a summation. The difference in registers decreases on higher dimensions, leading to degraded performance on low dimensions. This is evident in tables 4 and 5 where higher-dimensional integrands have smaller values in the $\frac{oneAPI}{CUDA}$ column. The same pattern is observed for the benchmark integrands, where the high dimensional versions perform better than the low dimension equivalents. It can be seen in Figure 1, that this effect is more prominent in m-Cubes, since it displays a larger deviation across all dimensions. These observations lead us to believe that register difference and its effect on occupancy is the main reason behind the performance degradation.

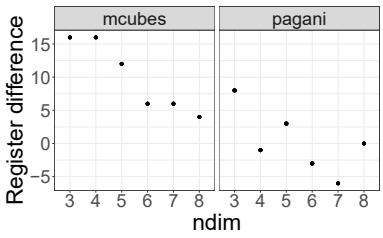

Fig. 1: Register difference on simple addition integrands ($\sum_{i=1}^{d} x_i$). The y-axis displays the number of additional registers per thread in the DPC++ implementation.

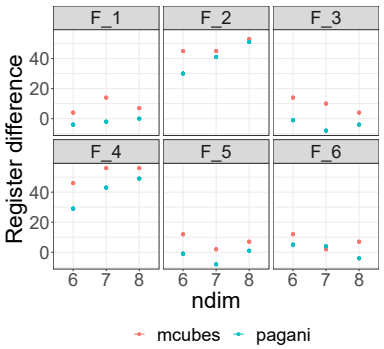

Fig. 2: Register difference on the benchmark integrands. The y-axis displays the number of additional registers per thread in the DPC++ implementation.

6 Conclusion

We presented our experience of porting two numerical integration implementations, PAGANI and m-Cubes, from CUDA to DPC++ . We utilized Intel's DPCCT to automate the conversion process from CUDA to SYCL and successfully attained the capability to execute the same implementation on both Intel and Nvidia GPUs. We experimented with various workloads consisting of different mathematical functions. We found that the assigned registers per thread can deviate in oneAPI and CUDA codes. This affects occupancy which in turn can negatively impact performance, particularly in compute-bound kernels. We faced additional challenges with mapping library calls to oneAPI equivalents, matching compiler optimizations of NVCC with Clang, and using build and testing libraries like CMake and Catch2. We addressed those challenges and demonstrated that the performance penalty of using oneAPI ports instead of optimized CUDA implementations can be limited to 10% on Nvidia GPUs. Additionally, numerous cases exhibited comparable performance to the original CUDA implementations, with execution time differences in the $1 - 2\%$ range. We compared oneAPI and CUDA implementations on the same Nvidia V100 GPU. We were able to execute on an Intel P630 GPU but we did not compare these timings with those on the V100 GPU due their significant difference in computing power. In the future, we plan to execute on the high end Intel Ponte Vecchio GPU and compare performance metrics with Nvidia high end GPUs such as A100.

The vast array of libraries, ease of portability, and small margin of performance degradation, make oneAPI an appropriate software solution for the use case of numerical integration.

7 Acknowledgements

The authors would like to thank Intel Corporation and Codeplay for providing technical support in the code migration process. The authors are also grateful for the support of the Intel oneAPI Academic Center of Excellence at Old Dominion University.

Work supported by the Fermi National Accelerator Laboratory, managed and operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes. FERMILAB-CONF-23-007-LDRD-SCD.

We acknowledge the support of Jefferson Lab grant to Old Dominion University 16-347. Authored by Jefferson Science Associates, LLC under U.S. DOE Contract No. DE-AC05-06OR23177 and DE-AC02- 06CH11357.

References

1. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
2. Argonne leadership computing facility, https://www.alcf.anl.gov/support-center/aurora/sycl-and-dpc-aurora#:~:text=DPC%2B%2B%20(Data%20Parallel%20C,versions%20of%20the%20SYCL%20language
3. Computecpp™ community edition, https://developer.codeplay.com/products/computecpp/ce/2.11.0/guides/#computecpp
4. Migrate cuda* to dpc++ code: Intel® dpc++ compatibility tool, https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html#gs.lx007q
5. What is openacc?, https://www.openacc.org/
6. et al., G.: Dark energy survey year 3 results: Redshift calibration of the maglim lens sample from the combination of sompz and clustering and its impact on cosmology (2022)
7. Arumugam, K., Godunov, A., Ranjan, D., Terzic, B., Zubair, M.: A memory efficient algorithm for adaptive multidimensional integration with multiple gpus. In: 20th Annual International Conference on High Performance Computing. pp. 169–175. IEEE (2013)
8. Ashbaugh, B., Bader, A., Brodman, J., Hammond, J., Kinsner, M., Pennycook, J., Schulz, R., Sewall, J.: Data parallel c++: Enhancing sycl through extensions for productivity and performance. In: Proceedings of the International Workshop on OpenCL. IWOCL '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3388333.3388653, https://doi.org/10.1145/3388333.3388653
9. Bridle, S., Dodelson, S., Jennings, E., Kowalkowski, J., Manzotti, A., Paterno, M., Rudd, D., Sehrish, S., Zuntz, J.: Cosmosis: a system for mc parameter estimation. Journal of Physics: Conference Series **664**(7), 072036 (dec 2015). https://doi.org/10.1088/1742-6596/664/7/072036, https://dx.doi.org/10.1088/1742-6596/664/7/072036
10. Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of parallel and distributed computing **74**(12), 3202–3216 (2014)
11. Castaño, G., Faqir-Rhazoui, Y., García, C., Prieto-Matías, M.: Evaluation of intel's dpc++ compatibility tool in heterogeneous computing. Journal of Parallel and Distributed Computing **165**, 120–129 (2022). https://doi.org/https://doi.org/10.1016/j.jpdc.2022.03.017, https://www.sciencedirect.com/science/article/pii/S0743731522000727
12. Christgau, S., Steinke, T.: Porting a legacy cuda stencil code to oneapi. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 359–367 (2020). https://doi.org/10.1109/IPDPSW50202.2020.00070
13. Costanzo, M., Rucci, E., García-Sánchez, C., Naiouf, M., Prieto-Matías, M.: Migrating cuda to oneapi: A smith-waterman case study. In: Bioinformatics and Biomedical Engineering, pp. 103–116. Lecture Notes in Computer Science, Springer International Publishing, Cham (2022)
14. Costanzo, M., Rucci, E., Sanchez, C.G., Naiouf, M.: Early experiences migrating cuda codes to oneapi (2021)

15. Costanzo, M., Rucci, E., Sánchez, C.G., Naiouf, M., Prieto-Matías, M.: Assessing opportunities of sycl and intel oneapi for biological sequence alignment (2022)
16. Doerfert, J., Jasper, M., Huber, J., Abdelaal, K., Georgakoudis, G., Scogland, T., Parasyris, K.: Breaking the vendor lock-performance portable programming through openmp as target independent runtime layer. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2022)
17. Haseeb, M., Ding, N., Deslippe, J., Awan, M.: Evaluating performance and portability of a core bioinformatics kernel on multiple vendor gpus. In: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 68–78 (2021). https://doi.org/10.1109/P3HPC54578.2021.00010
18. Jin, Z., Vetter, J.: Evaluating cuda portability with hipcl and dpct. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 371–376 (2021). https://doi.org/10.1109/IPDPSW52791.2021.00065
19. Kanzaki, J.: Monte carlo integration on gpu. The European physical journal. C, Particles and fields **71**(2), 1–7 (2011)
20. Peter Lepage, G.: A new algorithm for adaptive multidimensional integration. Journal of Computational Physics **27**(2), 192–203 (1978). https://doi.org/https://doi.org/10.1016/0021-9991(78)90004-9, https://www.sciencedirect.com/science/article/pii/0021999178900049
21. Ranjan, N., Terzić, B., Krafft, G., Petrillo, V., Drebot, I., Serafini, L.: Simulation of inverse compton scattering and its implications on the scattered linewidth. Physical review. Accelerators and beams **21**(3), 030701 (2018)
22. Sakiotis, I., Arumugam, K., Paterno, M., Ranjan, D., Terzić, B., Zubair, M.: PAGANI: A Parallel Adaptive GPU Algorithm for Numerical Integration. Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3458817.3476198
23. Sakiotis, I., Arumugam, K., Paterno, M., Ranjan, D., Terzić, B., Zubair, M.: m-cubes: An efficient and portable implementation of multi-dimensional integration for gpus. In: High Performance Computing, pp. 192–209. Lecture Notes in Computer Science, Springer International Publishing, Cham (2022)
24. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering **12**(3), 66–73 (2010)
25. Su, C.L., Chen, P.Y., Lan, C.C., Huang, L.S., Wu, K.H.: Overview and comparison of opencl and cuda technology for gpgpu. In: 2012 IEEE Asia Pacific Conference on Circuits and Systems. pp. 448–451 (2012). https://doi.org/10.1109/APCCAS.2012.6419068
26. Tsai, Y.M., Cojean, T., Anzt, H.: Porting sparse linear algebra to intel gpus. In: Euro-Par 2021: Parallel Processing Workshops, pp. 57–68. Lecture Notes in Computer Science, Springer International Publishing, Cham (2022)
27. Volokitin, V., Bashinov, A., Efimenko, E., Gonoskov, A., Meyerov, I.: High performance implementation of boris particle pusher on dpc++. a first look at oneapi. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp. 288–300. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021)
28. Wong, M., Liber, N., Bassini, S., Richards, A., Butler, M., McVeigh, J., Cook, B., Sugimoto, H., Cordoba, C., Fahringer, T., et al.: Sycl - c++ single-source heterogeneous programming for acceleration offload (Jan 2014), https://www.khronos.org/sycl/

29. Wu, H.Z., Zhang, J.J., Pang, L.G., Wang, Q.: Zmcintegral: A package for multi-dimensional monte carlo integration on multi-gpus. Computer Physics Communications **248**, 106962 (2020). https://doi.org/https://doi.org/10.1016/j.cpc.2019.106962, https://www.sciencedirect.com/science/article/pii/S0010465519303121
30. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka - an abstraction library for parallel kernel acceleration. In: arXiv.org. Cornell University Library, arXiv.org, Ithaca (2016)