





# Early Experiences Migrating CUDA codes to oneAPI

Manuel Costanzo<sup>1</sup>, Enzo Rucci<sup>1</sup><sup>\*</sup>, Carlos García-Sánchez<sup>2</sup>, and Marcelo Naiouf<sup>1</sup>

<sup>1</sup> III-LIDI, Facultad de Informática, UNLP – CIC.  
La Plata (1900), Bs As, Argentina

{mcostanzo,erucci,mnaiouf}@lidi.info.unlp.edu.ar

<sup>2</sup> Dpto. Arquitectura de Computadores y Automática, Universidad Complutense de Madrid. Madrid (28040), España  
garsanca@dacya.ucm.es

**Abstract.** The heterogeneous computing paradigm represents a real programming challenge due to the proliferation of devices with different hardware characteristics. Recently Intel introduced oneAPI, a new programming environment that allows code developed in DPC++ to be run on different devices such as CPUs, GPUs, FPGAs, among others. This paper presents our first experiences in porting two CUDA applications to DPC++ using the oneAPI `dpct` tool. From the experimental work, it was possible to verify that `dpct` does not achieve 100% of the migration task; however, it performs most of the work, reporting the programmer of possible pending adaptations. Additionally, it was possible to verify the functional portability of the DPC++ code obtained, having successfully executed it on different CPU and GPU architectures.

**Keywords:** oneAPI · SYCL · GPU · CUDA · Code portability

## 1 Introduction

In the last decade, the quest to improve the energy efficiency of computing systems has fueled the trend toward heterogeneous computing and massively parallel architectures [1]. One effort to face some of the programming issues related to heterogeneous computing is SYCL<sup>3</sup>, a new open standard from Khronos Group. SYCL is a domain-specific embedded language that allows the programmer to write single-source C++ host code including accelerated code expressed as functors. In addition, SYCL features asynchronous task graphs, buffers defining location-independent storage, automatic overlapping kernels and communications, interoperability with OpenCL, among other characteristics [2].

Recently, Intel announced the *oneAPI* programming ecosystem that provides a unified programming model for a wide range of hardware architectures. At the core of the oneAPI environment is the Data Parallel C++ (DPC++) programming language, which can be summarized as C++ with SYCL. Additionally,

<sup>\*</sup> Corresponding author.

<sup>3</sup> <https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>

DPC++ also features some vendor-provided extensions that might be integrated into these standards in the future [3].

Today, GPUs can be considered the dominant accelerator and CUDA is the most popular programming language for them [4]. To tackle CUDA-based legacy codes, oneAPI provides a compatibility tool (`dpct`) that facilitates the migration to the SYCL-based DPC++ programming language. In this paper, we present our experiences from porting two original CUDA apps to DPC++ using `dpct`. Our contributions are: (1) the analysis of the `dpct` effectiveness for CUDA code migration, and (2) the analysis of the DPC++ code’s portability, considering different target platforms (CPU and GPUs).

## 2 The oneAPI Programming Ecosystem

oneAPI<sup>4</sup> is an industry proposal based on standard and open specifications, that includes the DPC++ language and a set of domain libraries. Each hardware vendor provides its own compatible implementations targeting different hardware platforms, like CPUs and accelerators. The Intel oneAPI implementation consists of the Intel DPC++ compiler, the Intel `dpct` tool, multiple optimized libraries, and advanced analysis and debugging tools [5].

## 3 Experimental Work and Results

### 3.1 Migrating CUDA Codes to oneAPI

`dpct` assists developers in porting CUDA code to DPC++, generating human readable code wherever possible. Typically, `dpct` migrates 80-90% of code in automatic manner. In addition, inline comments are provided to help developers finish migrating the application. In this work, we have selected two CUDA applications from the CUDA Demo Suite (CDS)<sup>5</sup>. Both codes were translated from CUDA to DPC++ using the `dpct` tool.

**Matrix Multiplication (MM)** This app computes a MM using shared memory through tiled approach. Fig. 1 shows an example of the memory transference translations. Because `checkCudaErrors` is a utility function (it is not part of the CUDA core), `dpct` inserts a comment to report this situation. Then, the programmer must decide whether to remove the function or redefine it.

Fig. 2 shows the kernel invocations. At the top, the original CUDA kernel’s call and, at the bottom, the migrated DPC++ code (only a portion is included due to the lack of space). On the one hand, `dpct` adds comments informing the programmer that it is possible that the size of the *work-group* exceeds the maximum of the device, being his responsibility to prevent this from happening. On the other hand, the resulting code is longer and more complex than the

<sup>4</sup> <https://www.oneapi.com/>

<sup>5</sup> <https://docs.nvidia.com/cuda/demo-suite/index.html>

```

// copy host memory to device
checkCudaErrors(cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice));

// copy host memory to device
/*
DPCT1003:3: Migrated API does not return error code. (*, 0) is inserted. You may need to rewrite this code.
*/
checkCudaErrors((q_ctl.memcpy(d_A, h_A, mem_size_A).wait(), 0));

```

Fig. 1: MM memory transference. Up: Original CUDA code. Down: Resultant DPC++ code.

```

for (int j = 0; j < nIter; j++) {
  if (block_size == 16) {
    MatrixMulCUDA<16> <<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
  } else {
    MatrixMulCUDA<32> <<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
  }
}

for (int j = 0; j < nIter; j++) {
  if (block_size == 16) {
    /* DPCT1049:1: The workgroup size passed to the SYCL kernel may exceed the limit. To get the device limit,
    query info::device::max_work_group_size. Adjust the workgroup size if needed. */
    q_ctl.submit({s}(sycl::handler{cgh}) {
      sycl::range<2> As_range_ctl(16, 16); sycl::range<2> Bs_range_ctl(16, 16);

      sycl::accessor<float, 2, sycl::access::mode::read_write, sycl::access::target::local> As_acc_ctl(As_range_ctl, cgh);
      sycl::accessor<float, 2, sycl::access::mode::read_write, sycl::access::target::local> Bs_acc_ctl(Bs_range_ctl, cgh);

      cgh.parallel_for<class kernel3>(sycl::nd_range<3>(grid * threads, threads),
        [=](sycl::nd_item<3> item_ctl) { MatrixMulCUDA<16>(d_C, d_A, d_B, dimsA[2], dimsB[2], item_ctl,
          dpct::accessor<float, dpct::local, 2>(As_acc_ctl, As_range_ctl),
          dpct::accessor<float, dpct::local, 2>(Bs_acc_ctl, Bs_range_ctl));
        });
    });
  } else {
  }
}

```

Fig. 2: MM kernel call. Up: Original CUDA code. Down: Resultant DPC++ code (portion).

CUDA original code. However, it is important to remark that this code is the result of an automatic translation. By following the DPC++ conventions, it could be significantly simplified.

Finally, Fig. 3 shows part of the kernel bodies, resulting in very similar codes. `dpct` manages to correctly translate the local memory usage, although it defines the arrays outside the loop as opposed to the CUDA case. In addition, it can be noted that `dpct` effectively translates the `unroll` directive and the synchronization barriers.

**Reduction (RED)** This app computes a parallel sum reduction of large arrays of values. The CUDA code includes several important optimization strategies like reduction using shared memory, `__shfl_down_sync`, `__reduce_add_sync` and `cooperative_groups::reduce`.

In this case, `dpct` is not able to translate advanced functionalities such as *CUDA Cooperative Groups*. Fig. 4 presents the comment inserted by `dpct` to inform the programmer about this issue. Even so, the tool manages to translate most of the original CUDA code, leaving little work to the programmer.

```

// Loop over all the sub-matrices of A and B required
// to compute the block sub-matrix
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
  // Declaration of the shared memory array Aa/Ba used
  // to store the sub-matrix of A/B
  __shared__ float Aa[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float Ba[BLOCK_SIZE][BLOCK_SIZE];
  // Load the matrices from device memory to shared memory;
  // each thread loads one element of each matrix
  Aa[ty][tx] = A[a + wA * ty + tx];
  Ba[ty][tx] = B[b + wB * ty + tx];
  // Synchronize to make sure the matrices are loaded
  __syncthreads();
  // Multiply the two matrices together;
  // each thread computes one element of the block sub-matrix
  #pragma unroll
  for (int k = 0; k < BLOCK_SIZE; ++k) {
    Csub += Aa[ty][k] * Bb[k][tx];
  }
  // Synchronize to make sure that the preceding computation
  // is done before loading two new
  // sub-matrices of A and B in the next iteration
  __syncthreads();
}

```

```

// Loop over all the sub-matrices of A and B required
// to compute the block sub-matrix
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
  // Declaration of the shared memory array Aa/Ba used
  // to store the sub-matrix of A/B
  // Load the matrices from device memory to shared memory;
  // each thread loads one element of each matrix
  Aa[ty][tx] = A[a + wA * ty + tx];
  Bb[ty][tx] = B[b + wB * ty + tx];
  // Synchronize to make sure the matrices are loaded
  item_ctl_barrier();
  // Multiply the two matrices together;
  // each thread computes one element of the block sub-matrix
  #pragma unroll
  for (int k = 0; k < BLOCK_SIZE; ++k) {
    Csub += Aa[ty][k] * Bb[k][tx];
  }
  // Synchronize to make sure that the preceding computation
  // is done before loading two new
  // sub-matrices of A and B in the next iteration
  item_ctl_barrier();
}

```

Fig. 3: MM kernel. Left: Original CUDA code. Right: Resultant DPC++ code.

```

cg::thread_block_tile<32> tile32 = cg::tiled_partition<32>(cta);

```

```

/* DPC100710: Migration of this CUDA API is not supported by the Intel(R) DPC++ Compatibility Pool. */
cg::thread_block_tile<32> tile32 = cg::tiled_partition<32>(cta);

```

Fig. 4: RED kernel. Up: Original CUDA code. Down: Resultant DPC++ code.

### 3.2 Experimental Results

Two hardware platforms were used for the experimental work. The first comprises an Intel Core i3-4160 3.60GHz processor, 16GB main memory and a NVIDIA GeForce RTX 2070 GPU. The second has an Intel Core i9-10920X 3.50GHz processor, 32GB main memory, and an Intel Iris Xe MAX Graphics GPU, from the Intel DevCloud <sup>6</sup>. oneAPI and CUDA versions are 2021.2 and 10.1, respectively. In addition, different workloads were configured for MM ( $nIter = 10$ ;  $wA, wB, hA, hB = \{4096, 8192, 16384\}$ ). Finally, to run DPC++ code on NVIDIA GPUs, several modifications had to be made to the build, as it is not supported by default <sup>7</sup>.

Table 1 shows the execution times of MM (CUDA and DPC++ versions) on the different experimental platforms. Before analyzing the execution times, it is important to remark that the DPC++ code was successfully executed on all the selected platforms and that the results were correct in all cases.

On the RTX 2070, the DPC++ code presents some overhead compared to the original code. However, it should be noted that these results are not final since the oneAPI support for NVIDIA GPUs is still experimental <sup>8</sup>. In fact, currently the code generation does not consider any particular optimization passes.

The DPC++ code was compiled and successfully executed on two different Intel devices: a CPU and a GPU. In this way, we verified its functional portability

<sup>6</sup> <https://software.intel.com/content/www/us/en/develop/tools/devcloud.html>

<sup>7</sup> <https://intel.github.io/llvm-docs/GetStartedGuide.html>

<sup>8</sup> <https://www.codeplay.com/portal/news/2020/02/03/codeplay-contribution-to-dpcpp-brings-sycl-support-for-nvidia-gpus.html>

Table 1: MM execution times on the target platforms

| Size  | NVIDIA RTX 2070<br>(CUDA) | NVIDIA RTX 2070<br>(oneAPI) | Intel Core i9-10920X | Intel Iris Xe MAX |
|-------|---------------------------|-----------------------------|----------------------|-------------------|
| 4096  | 1.3                       | 1.4                         | 9.2                  | 6.3               |
| 8192  | 11.1                      | 15.3                        | 102.8                | 50.4              |
| 16384 | 89.3                      | 122.9                       | 919.5                | 401.1             |

on different architectures. Little can be said about its performance due to the absence of an optimized version for both Intel devices. However, there is probably significant room for improvement considering that the ported code was compiled and executed with minimal programmer intervention.

## 4 Conclusions and Future Work

In this paper, we present our first experience migrating CUDA code to DPC++ using the Intel oneAPI environment. First, we were able to test the effectiveness of `dpct` for the selected test cases. Despite not translating 100% of the code, the tool does most of the work, reporting the programmer of possible pending adaptations. Second, it was possible to verify the functional portability of the obtained DPC++ code, by successfully executing it on different CPU and GPU architectures.

As future work, we are interested in deepening the experimental work. In particular, we want to include other test cases, hardware architectures, and metrics (like performance portability).

## References

- [1] H. Giefers et al. “Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA”. In: *2016 IEEE ISPASS*. 2016, pp. 46–56.
- [2] Ronan Keryell and Lin-Ya Yu. “Early Experiments Using SYCL Single-Source Modern C++ on Xilinx FPGA”. In: *Proceedings of the IWOCCL ’18*. Oxford, UK: ACM, 2018. DOI: 10.1145/3204919.3204937.
- [3] S. Christgau and T. Steinke. “Porting a Legacy CUDA Stencil Code to oneAPI”. In: *2020 IEEE IPDPSW*. May 2020, pp. 359–367. DOI: 10.1109/IPDPSW50202.2020.00070.
- [4] Manuel Costanzo et al. “Comparison of HPC Architectures for Computing All-Pairs Shortest Paths. Intel Xeon Phi KNL vs NVIDIA Pascal”. In: *Computer Science – CACIC 2020*. Vol. 1409. 2021, pp. 37–48. DOI: 10.1007/978-3-030-75836-3\_3.
- [5] Nikita Hariharan et al. “Heterogeneous Programming using OneAPI”. In: *Parallel Universe* 39 (2020), pp. 5–18.