Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study

Leonardo Solis-Vasquez
solis@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Edward Mascarenhas
edward.mascarenhas@intel.com
Intel Corporation
Santa Clara, USA

Andreas Koch
koch@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

ABSTRACT

In recent years, Intel introduced oneAPI as a unified and cross-architecture programming model based on the Data Parallel C++ (DPC++) language, which in turn, is based on the C++ and SYCL standard languages. In order to facilitate the migration of legacy CUDA code originally written for NVIDIA GPUs, developers can employ the Intel DPC++ Compatibility Tool, which aims to automatically migrate code from CUDA to SYCL. While this tool-assisted code migration is a good starting point for leveraging the Intel oneAPI ecosystem, manual steps for code completion and tuning are still required. In this paper, we present our experiences migrating AutoDock-GPU, a widely-used molecular docking application, from CUDA to SYCL. Our discussion focuses on: (1) the use of this automated source-code migration tool, (2) the required manual code refinement for functionality and optimization, and (3) the comparison of the performance achieved in this manner on multi-core CPUs as well as on high-end GPUs, such as NVIDIA A100 and the recently-launched Intel Data Center Max 1550 device.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Applied computing** → *Chemistry*.

KEYWORDS

Intel oneAPI, SYCL, molecular docking, AutoDock, CPU, A100 GPU, Intel Data Center GPU Max Series

ACM Reference Format:
Leonardo Solis-Vasquez, Edward Mascarenhas, and Andreas Koch. 2023. Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study . In *International Workshop on OpenCL (IWOCL '23), April 18–20, 2023, Cambridge, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3585341.3585372

1 INTRODUCTION

Intel created oneAPI to allow developers to program a wide spectrum of heterogeneous architectures using the common Data Parallel C++ language (DPC++) [22]. DPC++ is based on ISO C++, and incorporates the Khronos standard SYCL [16] along with community extensions to simplify data parallel programming. Current research efforts on leveraging and improving oneAPI are taking place at several recently-established *Centers of Excellence* in different academic institutions [8].

The Intel DPC++ Compatibility Tool [6] (from now on simply referred to as *Compatibility Tool*) released as part of oneAPI can be used to minimize the effort of code migration from CUDA to SYCL. According to Intel's estimates as of September 2021[1], Compatibility Tool is able to *automatically* migrate 90%-95% of CUDA code into human-readable SYCL code. In addition, Compatibility Tool provides inline comments to assist developers in *manually* completing the migration by pointing out required localized code changes.

Compatibility Tool can be leveraged in High Performance Computing (HPC), Artificial Intelligence, Medical Imaging, embedded applications, and other scenarios, where CUDA applications are traditionally employed for tackling many challenging tasks in science and engineering [19]. In fact, recent studies have assessed the effectiveness of Compatibility Tool at porting applications of different domains, e.g., tsunami simulation [2], ultrasound beamforming [31], sequence alignment [4], as well as those from the Rodinia and SHOC benchmark suites [1, 15]. In particular, studies in [1, 2, 4] report specific limitations of Compatibility Tool, agreeing in that most frequent tool-reported warnings point to code sections performing error handling of API return codes, as well as kernel invocation. To gain further insights into the capabilities and limitations of Compatibility Tool for more complex *irregular* algorithms, we are investigating its use on a state-of-the-art molecular docking code.

Molecular docking is an interesting application domain. Being a key method in computer-aided drug design, it simulates the close-distance interactions of two molecules of known three-dimensional structure, and aims to predict their binding poses (i.e., spatial arrangements) that are energetically strong. These two molecules are known as *ligand* (small molecule) and *receptor* (macromolecule). Molecular docking is used to identify ligands with anti-viral properties against a receptor modeling a given biological target (e.g., a protein or nucleid acid) [5]. One of the most widely-used molecular docking applications is AutoDock [18]. It explores the pose space through a systematic search consisting of multiple irregular nested loops with variable upper bounds. The search refinement is driven by the score of each pose, which quantifies the strength of the molecular interaction. The score is based on compute-intensive models and is typically evaluated 10^6 times within these search iterations.

[1]Based on measurements on a set of 70 HPC benchmarks and samples, with examples including Rodinia, SHOC, and PENNANT. Other results may vary.

IWOCL '23, April 18–20, 2023, Cambridge, United Kingdom

ACM ISBN 979-8-4007-0745-2/23/04…$15.00
https://doi.org/10.1145/3585341.3585372

These high-computing requirements of the molecular docking algorithm in AutoDock have spurred interest in parallelizing and accelerating this application. For that purpose, the official parallel-code development of AutoDock, called AutoDock-GPU [24], has been originally implemented in OpenCL [23], and afterwards migrated to CUDA to run on the *Summit* supercomputer for COVID-19 research [17]. Other authors have implemented a miniapp version of AutoDock-GPU using alternative heterogeneous programming languages, such as HIP and Kokkos [30]. Since its release, AutoDock-GPU has been under several performance enhancements for CPUs and GPUs [28, 29], as well as ported to different heterogeneous architectures including FPGAs [27] and vector processors [26]. As another example of its relevance, AutoDock-GPU is nowadays being used as the docking engine in *OpenPandemics: COVID-19*, a grid-computing project aiming to study proteins from the SARS-CoV-2 virus [32].

With its challenging code structure and high practical relevance, AutoDock-GPU is thus a promising case study for evaluating the automated CUDA-to-SYCL code migration, e.g., to target new Intel devices such as the Data Center Max 1550 GPU (code-named *Ponte Vecchio*). Our contributions are summarized as follows:

(1) We present our experiences migrating the molecular docking code of AutoDock-GPU from CUDA to SYCL. For this purpose, we employ Compatibility Tool, with the migrated (and manually refined) SYCL code being released as open source[2].
(2) We evaluate the performance of different code versions of AutoDock-GPU, i.e., the original OpenCL and CUDA as well as the newly migrated SYCL, on Intel Xeon Platinum 8360Y CPU, NVIDIA A100 GPU, and Intel Data Center Max 1550 GPU.
(3) We compare our work here to previous studies in terms of migration cases addressed when employing Compatibility Tool.

The remainder of this paper is structured as follows. Section 2 provides a background on AutoDock-GPU. Section 3 discusses the code migration process, while Section 4 evaluates the performance achieved on CPUs and GPUs. Section 5 compares our work to previous studies. Finally, Section 6 concludes this paper with a summary and outlook to future work.

2 BACKGROUND ON AUTODOCK-GPU

In this work, we use AutoDock-GPU v1.5.3 as our baseline, which corresponds to the latest stable release at the time of experimentation. Extensive descriptions on AutoDock-GPU's fundamentals and the evolution of the code base can be found in [23, 28, 29].

2.1 Functionality Overview

AutoDock-GPU performs a systematic search based on genetic evolution heuristics, where each of the ligand poses is treated as an individual of a population. Each individual is represented by its *genotype*, comprising in turn a set of *genes*, which describe the translation, orientation, and torsion experienced by the ligand during docking.

[2]https://github.com/ccsb-scripps/AutoDock-GPU/pull/183

The computational core of AutoDock-GPU is an irregular Lamarckian Genetic Algorithm (LGA) that performs a hybrid search combining a genetic algorithm (GA) and a local search (LS). Both LGA phases generate new individuals from the current population, however they employ different methods. The genetic algorithm applies genetic operations (i.e., crossover, mutation, and selection), while the local search aims to further improve (i.e., minimize) scores. As shown in Algorithm 1, AutoDock-GPU executes several independent LGA runs (default: $N^{\text{TOTAL}}_{\text{LGA-runs}} = 100$). AutoDock-GPU can enforce that each of these LGA runs terminates whenever any of the predefined upper bounds for the number of score evaluations (default: $N^{\text{MAX}}_{\text{score-evals}} = 25 \times 10^6$) or generations (default: $N^{\text{MAX}}_{\text{gens}} = 27 \times 10^3$) is reached.

Algorithm 1: Lamarckian Genetic Algorithm (LGA)

```
1 Function AutoDock-GPU
      /* Coarse-Level Parallelism */
2     for each LGA-run in N^TOTAL_LGA-runs do
3         while (N_score-evals < N^MAX_score-evals) and (N_gens < N^MAX_gens) do
              /* Medium-Level Parallelism */
4             GA (population)
5             LS (population)
```

For measuring the strength (i.e., energy) of molecular interactions, scores (expressed in kcal/mol) are computed for every pose during both LGA phases. Algorithm 2 shows the code structure of the scoring function (SF), which consists of three components. `PoseCalculation` transforms the genotypes into atomic coordinates, which are then used for computing the ligand-receptor (`InterScore`) and ligand-ligand (`IntraScore`) score components. The upper bounds of the corresponding loops depend on the molecular structure of the input, i.e., the number of elements in the rotation list ($N_{\text{rot-list}}$), the number of ligand atoms (N_{atom}), and the number of intramolecular contributor-pairs ($N_{\text{intra-contrib}}$).

Algorithm 2: Scoring Function (SF)

```
   /* Fine-Level Parallelism */
1  Function SF (genotype)
2      for each rot-item in N_rot-list do
3          PoseCalculation
4      for each lig-atom in N_atom do
5          InterScore
6      for each intra-pair in N_intra-contrib do
7          IntraScore
```

The most time-consuming phase of AutoDock-GPU is the local search, whose execution corresponds to >90% of the total execution time. AutoDock-GPU features several alternative methods that can be chosen for local search. Each of these methods minimizes the score through a number of adaptive iterations.

Similarly to the original AutoDock program, AutoDock-GPU implements as local search the method of Solis-Wets [25]. As shown in Algorithm 3, in each iteration, Solis-Wets starts generating a new genotype by adding small changes (i.e., a constrained random amount) to each gene of an initial genotype (Algorithm 3: lines 4, 5). Correspondingly, the scores of the aforementioned genotypes are

computed and compared (Algorithm 3: line 6). If the score is not minimized, a second new genotype is generated similarly as in the case above. In this case, however, a subtraction (Algorithm 3: lines 10, 11) is applied instead of an addition. The outcome of every comparison updates the number of successful and failed minimization attempts. Besides its divergent execution, Solis-Wets employs an irregular termination criterion (Algorithm 3: line 2) that depends on the maximum number of iterations (default: $N^{\text{MAX}}_{\text{LS-iters}} = 300$), as well as the minimum step change (default: $\text{step}^{\text{MIN}} = 0.01$).

Algorithm 3: Solis-Wets (SW) local search

```
   /* Fine-Level Parallelism */
1  Function SW (genotype)
2      while (N_LS-iters < N^MAX_LS-iters) and (step > step^MIN) do
3          delta = create-delta (step)
           // new-genotype1
4          for each gene in N_genes do
5              new-gene1 = gene + delta
6          if SF (new-genotype1) < SF (genotype) then
7              genotype = new-genotype1
8              success++; fail = 0
9          else
               // new-genotype2
10             for each gene in N_genes do
11                 new-gene2 = gene - delta
12             if SF (new-genotype2) < SF (genotype) then
13                 genotype = new-genotype2
14                 success++; fail = 0
15             else
16                 success = 0; fail++
17         step = update-step (success, fail)
```

Stepping forward with respect to the original AutoDock, AutoDock-GPU has incorporated improved local-search methods beyond Solis-Wets. Algorithm 4 describes one of these, ADADELTA [33], which generates a new genotype by using the gradients of the current genotype's score (Algorithm 4: line 4). Then, if the score of the new genotype is improved, the latter becomes the current genotype (Algorithm 4: line 6). ADADELTA terminates if the number of iterations reaches a maximum (default: $N^{\text{MAX}}_{\text{LS-iters}} = 300$).

Algorithm 4: ADADELTA (AD) local search

```
   /* Fine-Level Parallelism */
1  Function AD (genotype)
2      gradient = GC (genotype)
3      while (N_LS-iters < N^MAX_LS-iters) do
4          new-genotype = update-rule (genotype, gradient)
5          if SF (new-genotype) < SF (genotype) then
6              genotype = new-genotype
7          gradient = GC (genotype)
```

The gradient calculation (GC) employed by ADADELTA is described in Algorithm 5. The code structure resembles that of the scoring function in Algorithm 2. First, the `PoseCalculation` computes the atomic coordinates, which in turn, are used for computing the numerical (`InterGradient`) and analytical (`IntraGradient`) derivatives of the corresponding score components. At this point, such derivatives are expressed as a list of atomic contributions. However, as the overall LGA search works on genotypes, it is required to convert those atom-based into gene-based contributions. This conversion is achieved with `Gtrans`, `Grigidrot`, and `Grotbond` (Algorithm 5: lines 8-10), which are loops performing data-dependent operations for computing the translational, orientational, and rotational components of the gradient.

Algorithm 5: Gradient Calculation (GC)

```
   /* Fine-Level Parallelism */
1  Function GC (genotype)
       /* Gradients in atomic space */
2      for each rot-item in N_rot-list do
3          PoseCalculation
4      for each lig-atom in N_atom do
5          InterGradient
6      for each intra-pair in N_intra-contrib do
7          IntraGradient
       /* Convert from atomic into genetic space */
8      Gtrans // Translational gradients
9      Grigidrot // Rigid-body rotation gradients
10     Grotbond // Rotatable-bond gradients
```

2.2 Parallelization

As already indicated, AutoDock-GPU was originally developed in OpenCL [23], and thereafter ported to CUDA [17]. Both implementations follow a Single Instruction Multiple Thread (SIMT) programming style. Table 1 shows how AutoDock-GPU's computations are mapped onto OpenCL and CUDA processing elements at different paralellization levels.

In general, AutoDock-GPU performs $N^{\text{TOTAL}}_{\text{LGA-runs}}$ independent LGA runs with indices $\text{Run}_{\text{ID}} = \{0, 1, 2, \ldots, N^{\text{TOTAL}}_{\text{LGA-runs}} - 1\}$. In each LGA run, the genetic algorithm and local search process a population of Pop_{size} individuals with indexes $\text{Ind}_{\text{ID}} = \{0, 1, 2, \ldots, \text{Pop}_{\text{size}} - 1\}$. The main idea of AutoDock-GPU's parallelization is to process *simultaneously* individuals from different LGA runs. Based on this, $N^{\text{TOTAL}}_{\text{LGA-runs}} \times \text{Pop}_{\text{size}}$ individuals are mapped each to an OpenCL work-group (CUDA block), where the index of an OpenCL work-group (CUDA block) can be expressed as: $\text{WG}_{\text{ID}} = \text{Run}_{\text{ID}} \times \text{Pop}_{\text{size}} + \text{Ind}_{\text{ID}}$. Moreover, the fine-grained tasks such as the genotype generation and score evaluation are carried out by OpenCL work-items (CUDA threads).

Table 1: Mapping of AutoDock-GPU's computations onto OpenCL and CUDA processing elements. The corresponding parallelism is also indicated as comments in Algorithms 1, 2, 3

Computation	OpenCL (CUDA) element	Parallelization level
Genetic algorithm / Local search	Kernel	Coarse
Individual	Work-Group (Block)	Medium
Generation / Scoring	Work-Item (Thread)	Fine

3 MIGRATING AUTODOCK-GPU TO SYCL

In our case, we have two options to migrate to SYCL:

(1) Migrate from the original OpenCL version.
(2) Migrate from the recently-created CUDA version.

Since one of our goals is the performance comparison of new SYCL-programmable GPUs such as Intel's Data Center Max 1550 with state-of-the-art NVIDIA GPUs, we employed the second approach consisting of two main steps:

- First, using as much as possible the output of Compatibility Tool v2021.2.0[3], namely the SYCL migrated code created from the CUDA code of AutoDock-GPU (v1.5.3). As a baseline, common CUDA API concepts and their automatically migrated equivalents in SYCL[4]are shown in Table 2.
- Second, reviewing and manually completing the tool-assisted migration by using the tool guidelines provided as code comments.

In this section, we discuss the migration cases addressed in this work, categorized in two groups according to their purpose: functional correctness and performance optimization. Examples for the cases in these two groups are illustrated in listings showing the manually-edited SYCL code.

3.1 Migration Cases for Functional Correctness

3.1.1 Reductions. AutoDock-GPU employs integer reductions to keep track of the total number of score evaluations. In the CUDA version of AutoDock-GPU, these reductions are implemented as multi-line macros performing shuffles. However, Compatibility Tool could not migrate these correctly, warning that it could not handle mask options for shuffle (e.g., Listing 2: line 15). By inspecting the CUDA code, we realized that it was easier to manually use the equivalent built-in SYCL `reduce_over_group()` function directly. Listing 1 shows that by using this *collective* function, we greatly simplify this macro in the SYCL version. In general, collectives should be leveraged whenever possible.

3.1.2 Shuffles. The CUDA version of AutoDock-GPU employs a block of threads performing shuffles to find the minimum score. In this case, Compatibility Tool could not determine that a mask was used, and thus, it performed incorrect variable substitution. Listing 2 shows that to fix this, we had to manually insert the equivalent SYCL sub-group shuffles (lines 21, 22). Compatibility Tool reported the corresponding warning (line 15), which is due to the fact that the SYCL sub-group shuffle function used for migration does not support the extra mask argument required in the CUDA counterpart.

3.1.3 Synchronization. In the CUDA version of AutoDock-GPU, `__threadfence()` is always followed by `__syncthreads()` (e.g., Listing 1: lines 22, 23). Compatibility Tool leaves `__threadfence()` as is, while it migrates CUDA `__syncthreads()` into SYCL `work_item.barrier()`. From a semantics perspective, this migration step makes sense as a SYCL `barrier()` performs two things [3]. First, it ensures that each work-item within a work-group reaches the barrier call. In this way, the barrier synchronizes the work-group at a certain point in the code, acting equivalently to CUDA `__syncthreads()`. Second, the barrier emits a memory fence ensuring that the specified space is consistent across all work-items within the work-group, acting equivalently to CUDA `__threadfence()`. Therefore, we manually migrate `__threadfence()` as a no-op, and keep the tool-migrated `work_item.barrier()` statement (e.g., Listing 1: line 31). Moreover, `cudaDeviceSynchronize()` was manually migrated to `dpct::get_default_queue().wait_and_throw()`.

Listing 1: Migration of reduction macro

```
1  // CUDA
2  #define REDUCEINTEGERSUM(val, pAccumulator) \
3      if (threadIdx.x == 0) \
4      { \
5          *pAccumulator = 0; \
6      } \
7      __threadfence(); \
8      __syncthreads(); \
9      if (__any_sync(0xffffffff), val != 0)) \
10     { \
11         uint32_t tgx = threadIdx.x & cData.warpmask; \
12         val += __shfl_sync(0xffffffff, val, tgx^1); \
13         val += __shfl_sync(0xffffffff, val, tgx^2); \
14         val += __shfl_sync(0xffffffff, val, tgx^4); \
15         val += __shfl_sync(0xffffffff, val, tgx^8); \
16         val += __shfl_sync(0xffffffff, val, tgx^16); \
17         if (tgx == 0) \
18         { \
19             atomicAdd(pAccumulator, val); \
20         } \
21     } \
22     __threadfence(); \
23     __syncthreads(); \
24     val = *pAccumulator; \
25     __syncthreads();
26
27 // SYCL
28 #define REDUCEINTEGERSUM(val, pAccumulator) \
29     int myval = sycl::reduce_over_group(wi.get_group(), val,
           std::plus<>()); \
30     *pAccumulator = myval; \
31     wi.barrier(sycl::access::fence_space::local_space);
```

Listing 2: Migration of shuffles

```
1  // CUDA
2  #define WARPMINIMUMEXCHANGE(tgx, v0, k0, mask) \
3  { \
4      float v1 = v0; int k1 = k0; \
5      int otgx = tgx ^ mask; \
6      float v2 = __shfl_sync(0xffffffff, v0, otgx); \
7      int k2 = __shfl_sync(0xffffffff, k0, otgx); \
8      int flag = ((v1<v2) ^ (tgx>otgx)) && (v1!=v2); \
9      k0 = flag ? k1 : k2; \
10     v0 = flag ? v1 : v2; \
11 }
12
13 // SYCL
14 /*
15 DPCT1023:57: The DPC++ sub-group does not support mask options
       for shuffle.
16 */
17 #define WARPMINIMUMEXCHANGE(tgx, v0, k0, mask) \
18 { \
19     float v1 = v0; int k1 = k0; \
20     int otgx = tgx ^ mask; \
21     float v2 = wi.get_sub_group().shuffle(v0, otgx); \
22     int k2 = wi.get_sub_group().shuffle(k0, otgx); \
23     ...
24 }
```

3.1.4 Sub-group sizes. A SYCL sub-group represents a short range of consecutive work-items that are processed together as a SIMD

[3]Our migration of AutoDock-GPU to SYCL used the Intel DPC++ Compatibility Tool. This work predates the release of SYCLomatic [11], which is based on Compatibility Tool. Any improvements made to SYCLomatic through community efforts will be also incorporated in the Intel DPC++ Compatibility Tool product.
[4]Compatibility Tool generates 3D `nd_range` kernels by default as it works in general for migrating most CUDA kernels and does not incur a performance penalty compared to 1D.

Table 2: Examples for automatic CUDA-to-SYCL migrations using Intel DPC++ Compatibility Tool

CUDA	SYCL	
Thread	`sycl::nd_item`	(work-item)
Warp	`sycl::sub-group`	(sub-group)
Block	`sycl::group`	(work-group)
`threadIdx.x`	`item_ct1.get_local_id(2)`	(for three-dimensional range)
`blockDim.x`	`item_ct1.get_local_range().get(2)`	(for three-dimensional range)
`blockIdx.x`	`item_ct1.get_group(2)`	(for three-dimensional range)
`cudaMalloc`	`sycl::malloc_device`	
`cudaMallocManaged`	`sycl::malloc_shared`	
`cudaFree`	`sycl::free`	
`cudaMemcpy`	`dpct::get_default_queue().memcpy(. . .).wait()`	
`__shared__`	`sycl::accessor<int, 0, sycl::access::mode::read_write, sycl::access::target::local>`	
`kernel«<blocks, threadsPerBlock»>();`	`dpct::get_default_queue().submit([&](sycl::handler &cgh){ cgh.parallel_for (sycl::nd_range<3>( sycl::range<3>(1, 1, blocks) * sycl::range<3>(1, 1, threadsPerBlock), sycl::range<3>(1, 1, threadsPerBlock)), [=](sycl::nd_item<3> item_ct1){ kernel(item_ct1); }); });`	

vector [10]. Since SYCL allows alternative sub-group size configurations, Compatibility Tool does not generate any size. Hence, when SYCL sub-group functions are used (e.g., shuffles), sub-group sizes must be explicitly specified as in Listing 3 (line 8).

Listing 3: Specification of sub-group size

```
1  // SYCL
2  cgh.parallel_for(
3      sycl::nd_range<3>(
4          sycl::range<3>(1,1,blocks) *
5          sycl::range<3>(1,1,threadsPerBlock),
6          sycl::range<3>(1,1,threadsPerBlock)
7      ), [=](sycl::nd_item<3> wi)
8      [[intel::reqd_sub_group_size(32)]] {
9          gpu_gen_and_eval_newpops_kernel(...)
10         ...
11     }
12 );
```

3.1.5 Memory layout for vector data types. The CUDA version of AutoDock-GPU relies on memory allocated using `sizeof(float3)` (= `3*sizeof(float)`) while doing pointer arithmetic in kernels. While Compatibility Tool does correctly migrate CUDA `float3` to SYCL `sycl::float3`, a *different* number of bytes is allocated in each case: 12 for CUDA vs. 16 for SYCL. This particular discrepancy in the allocated memory between CUDA and SYCL led to a silent memory corruption overwriting portions of the shared memory reserved for other variables in AutoDock-GPU, which in turn, led to incorrect score evaluations. This error was pinpointed through extensive debugging, and corrected by explicitly propagating – from host to device – the size of the allocated memory (based on e.g., `sycl::float3` or `sycl::int3`) via the pointer to its corresponding SYCL accessor.

3.1.6 Query for available memory on device. The CUDA version of AutoDock-GPU calls `cudaMemGetInfo()` during host setup. However, Compatibility Tool does not migrate it. For this reason, we manually insert the `get_device_info()` and `get_global_mem_size()` calls.

3.1.7 Assembly code. The CUDA version of AutoDock-GPU contains few lines of inline PTX assembly code. Likely, this was carried out for performance optimization reasons, as inline PTX enables the access to instructions not exposed via CUDA intrinsics. Compatibility Tool does *not* support the migration of inline PTX. However, Listing 4 shows how we easily manually convert inline PTX in CUDA into SYCL.

Listing 4: Migration of assembly code

```
1  // CUDA
2  __device__ inline uint64_t llitoulli(int64_t l) {
3      uint64_t u;
4      asm("mov.b64 %0, %1;" : "=l"(u) : "l"(l));
5      return u;
6  }
7
8  // SYCL
9  inline uint64_t llitoulli(int64_t l) {
10     uint64_t u;
11     u = l;
12     return u;
13 }
```

3.2 Migration Cases for Performance Optimization

3.2.1 Atomics and barriers. Compatibility Tool assumes that the memory address space to perform atomic operations is always declared as *global*. However, this is not strictly required in all cases, and hence, *local* address space could be used instead to reduce the synchronization effort. Listing 5 shows how we manually specify the `local_space` (line 11) for the SYCL atomic addition on the `pAccumulator` variable. Moreover, Compatibility Tool makes assumptions on the memory order and scope. Similarly, the initially tool-migrated `acq_rel` order and `device` scope are manually replaced with `relaxed` (line 9) and `work_group` (line 10), respectively. As already described in Section 3.1.3, any CUDA `__syncthreads()` is automatically migrated as a SYCL `work_item.barrier()`. However, in order to properly configure it for *local* memory space, we have to manually add the `sycl::access::fence_space::local_space` specifier as an argument to the barrier call (Listing 1: line 31).

Listing 5: Migration of atomic operation

```
// CUDA
#define ATOMICADDI32(pAccumulator, value) \
    atomicAdd(pAccumulator, (value))

// SYCL
#define ATOMICADDI32(pAccumulator, value) \
    sycl::atomic_ref< \
        int, \
        sycl::memory_order::relaxed, \
        sycl::memory_scope::work_group, \
        sycl::access::address_space::local_space> \
        (*pAccumulator) += ((int)(value))
```

3.2.2 Native math functions. Compatibility Tool migrates CUDA single-precision math functions into their SYCL equivalents. For higher performance, the OpenCL version of AutoDock-GPU leverages *native* math functions as much as possible. For the same purpose, we manually replace the automatically-migrated calls to default SYCL math functions (e.g., `sycl::sqrt()`) with their native counterparts (e.g., `sycl::native::sqrt()`).

3.3 Summary of Migration Cases and Required Actions

Table 3 summarizes the generic actions, required from an application developer, for resolving each of the migration cases discussed in Sections 3.1 and 3.2.

On the one hand, most migration cases require manual code fixes for achieving a functionally-correct program. Of these, some cases are not enforced by Compatibility Tool: e.g., that of sub-group sizes (Section 3.1.4), for which the developer must understand if the code depends on a given feature, and manually add it, if needed. An outlier case is that of shuffle migration (Section 3.1.2), originally requiring us to manually insert SYCL sub-group shuffles for attaining working code. This is due to a limitation of v2021.2.0 of Compatibility Tool used in this work. However, newer versions of Compatibility Tool have resolved this issue, and thus, such manual fix is *not* necessary anymore. Furthermore, the migration of `cudaMemGetInfo()` (Section 3.1.6) could be resolved in the future, e.g., when the underlying software layers in drivers provide the function calls needed for querying device characteristics.

On the other hand, only two migration cases, i.e., atomics and barriers (Section 3.2.1) and native math (Section 3.2.2), aim for performance optimization. Since successfully dealing with these two cases requires a deeper understanding of the algorithmic needs of a program, Compatibility Tool cannot migrate them fully automatically, and hence, we need to fix them manually.

4 EVALUATION

In this section, we report the relative performance achieved with our migrated (and manually refined) SYCL version of AutoDock-GPU with respect to their original CUDA and OpenCL counterparts. The numbers reported here are interim[5] as our performance optimization work is still ongoing.

[5]This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

Table 3: Migration cases and required actions for AutoDock-GPU when using Compatibility Tool v2021.2.0

Migration cases for functional correctness		Required action
3.1.1	Reductions	Manual fix
3.1.2	Shuffles	Manual fix / None (for newer tool versions)
3.1.3	Synchronization	Manual fix
3.1.4	Sub-group sizes	Manual fix
3.1.5	Memory layout for vector data types	Manual fix
3.1.6	Query of available memory on device	Manual fix
3.1.7	Assembly code	Manual fix
Migration cases for performance optimization		**Required action**
3.2.1	Atomics and barriers	Manual fix
3.2.2	Native math functions	Manual fix

Table 4: Employed test cases

Ligand-Receptor Test Case	1ac8	1stp	3ce3	3tmn	7cpa
N_{rot}	0	5	5	1	15
N_{atom}	8	18	37	27	43

4.1 Experimental Setup

4.1.1 Dataset. We use a set of five ligand-receptor test cases from the AutoDock-GPU GitHub repository [24]. Table 4 shows these cases indicating their number of rotatable bonds (N_{rot}) and atoms (N_{atom}).

4.1.2 Accelerator devices. We compare the performance of OpenCL, CUDA, and SYCL versions of AutoDock-GPU achieved on datacenter-grade CPUs and GPUs. Table 5 lists relevant characteristics of the employed devices: Intel Xeon Platinum 8360Y CPU [7], NVIDIA A100 GPU [20], and Intel Data Center Max 1550 (from now on abbreviated as *Max 1550*) GPU [13].

4.2 Performance Comparison

For a fair comparison, we disregard the different host platforms holding the two GPUs. Specifically, we include only the GPU-side kernel configuration and execution, plus all required host-GPU data movements in our measurements. These time measurements are collectively reported as *docking* time. Host-side operations, such as file I/O and results processing, were not included and are considered as *idle* time (from the GPU perspective). All charts shown here report the docking time *ratios* achieved per ligand-receptor test case on all selected devices. Furthermore, these charts compare side-by-side the performance achieved by AutoDock-GPU's local-search methods: Solis-Wets (left) vs. ADADELTA (right).

We start comparing two hardware configurations for the Max 1550 GPU: 1S vs. 2S, which respectively leverage a single and both *Stacks* available in this GPU. From a hardware perspective, an X^e-stack contains up to four X^e-*slices* [12]. Each X^e-slice mainly contains 16 X^e-*cores*. Such core is the fundamental compute unit on the X^e GPU architecture. Therefore, a 2S configuration can apply double the hardware resources of the 1S one. Specifically, a 2S configuration contains two X^e-stacks, i.e., eight X^e-slices, which in turn, comprise a total of 128 X^e-cores (plus additional supporting

Table 5: Employed accelerator devices

Characteristics	Intel Xeon Platinum 8360Y CPU	NVIDIA A100 GPU	Intel Data Center Max 1550 GPU
Release Date	April 2021	June 2021	January 2023
Form Factor	-	PCIe Add-in-Card (AIC)	OCP Accelerator Module (OAM)
Architecture	Ice Lake-SP	Ampere	X^e-HPC
Process Size [nm]	10	7	Multi-tile package
TDP [W]	250	250	600
Frequency [GHz]	2.40 (base) 3.50 (boost)	0.76 (base) 1.41 (boost)	0.90 (base) 1.60 (boost)
Number of Cores	36 × 2 sockets (two threads per core)	108 SMs	128 X^e-cores
FP32 Performance [TFLOP/s]	5.5	19.5	52
Memory Type	DDR4	HBM2e	HBM2e
Memory Bandwidth [GB/s]	410	1555	3200
Memory Capacity [GB]	256	80	128
L1 Cache	64 kB (per core)	192 kB (per SM)	512 kB (per X^e-core)
L2 Cache	1 MB (per core)	40 MB (shared)	408 MB (shared)
L3 Cache	54 MB (shared)	-	-

units such as tracing units, hardware contexts, HBM2e controllers, and 16 X^e-links realizing a high-speed coherent fabric in multi-GPU configurations). From a software perspective, no changes to source code are required to run on both stacks; the default is to enable and run on both stacks. An environment variable (i.e., `ZE_AFFINITY_MASK`) can be set (to either `0.0` or `0.1`) in order to restrict runs to one stack [9].

Figure 1 reports the docking time ratios between these two configurations: higher 1S/2S ratios correspond to faster executions of the 2S configuration. For both Solis-Wets and ADADELTA, in all test cases, the usage of two X^e-stacks in the Max 1550 result in *faster* executions, achieving maximum speedup factors of 1.47× (Solis-Wets, 1stp) and 1.58× (ADADELTA, 7cpa). With respect to the ideal 1S/2S speedup factor of 2×, we attribute the achieved lower ratios to the required synchronization effort (Section 3.2.1) in compute-intensive regions, i.e., the scoring function (Algorithm 2) and the gradient calculation (Algorithm 5).

The following two charts compare the performance achieved with SYCL vs. that with the native parallel programming language of the device: OpenCL for Xeon 8360Y CPU (Figure 2), and CUDA for A100 (Figure 3).

Figure 2 reports the SYCL/OpenCL docking time ratios achieved on the Xeon 8360Y CPU. All test cases yield ratios lower than 1 (i.e., horizontal red line), indicating that on this CPU, for both Solis-Wets and ADADELTA, all SYCL executions are faster than OpenCL ones. We attribute the improved SYCL performance to optimization pipelines that are specific to SYCL, such as pre-SPIR-V optimizations, inlining heuristics, and others. The performance advantage factor of SYCL over OpenCL ranges within {∼1.06× (=$\frac{1}{0.94}$, for 7cpa), ∼1.20× (=$\frac{1}{0.84}$, for 1stp)} for Solis-Wets, and within {∼1.03× (=$\frac{1}{0.97}$, for 1ac8), ∼1.20× (=$\frac{1}{0.84}$, for 1stp)} when running ADADELTA.

Figure 3 reports the SYCL/CUDA docking time ratios achieved on the A100 GPU. Most test cases yield ratios higher than 1 (i.e., horizontal red line), which means that their respective SYCL executions are slower than the CUDA ones. The only exception occurs for Solis-Wets and the smallest test case 1ac8, where SYCL is ∼1.09× (=$\frac{1}{0.91}$) faster than CUDA. For all test cases in ADADELTA, SYCL is slower than CUDA in factors ranging within {1.24× (1ac8), 2.38× (7cpa)}.

In order to understand more where to focus optimizing our SYCL version on the A100, we profile AutoDock-GPU executions using the NVIDIA Nsight Compute tool [21]. For instance, Table 6 reports relevant metrics collected from a single ADADELTA local-search kernel execution. In this case, CUDA is 1.6× faster despite the fact SYCL achieves a slightly higher peak FP32 performance (11% vs. 10%). While not depicted here, Nsight Compute shows that both the SYCL and CUDA versions fall in the compute-bound region on the roofline chart. Based on the achieved arithmetic intensity (FLOP/byte) and performance (GFLOP/s), it seems that the SYCL version is performing more computations than its CUDA counterpart. We believe this is possibly due to a non-evident mismatch in the arithmetic precision employed in some parts of both versions.

Regarding the GPU scheduler statistics reported in Table 6, specifically on the number of theoretical warps, SYCL achieves only *half* of those by CUDA (4 vs. 8, respectively). This scheduler metric is limited by the launch configuration, in which we find noticeable differences: SYCL uses 1.9× more registers per threads as well as 95.8× more static shared memory. Surprisingly, while neither the original CUDA nor our migrated SYCL version allocate dynamic shared memory in kernels, Nsight Compute reports 1.8 kbytes/block being dynamically allocated from the SYCL code. Furthermore, the higher register pressure in SYCL (127 vs. 64) causes its lower kernel occupancy (19.3% vs. 38.8%). A way to alleviate this is to prevent the compiler from allocating too many registers. For SYCL code targeting NVIDIA GPUs, this is possible e.g., by passing down to the underlying nvcc compiler the –maxrregcount option, which allows setting the maximum number of registers to be allocated per thread. On this point, we will evaluate different –maxrregcount configurations seeking those increasing the SYCL performance.

A performance comparison between the selected GPU devices is shown in Figure 4, where the docking time ratios between the executions on Max 1550 (SYCL, 2S configuration) and A100 (CUDA) are depicted. For Solis-Wets, smaller test cases result in ∼1.42× (=$\frac{1}{0.70}$, for 1ac8) and ∼1.19× (=$\frac{1}{0.84}$, for 1stp) faster executions on the Max 1550, whereas larger test cases result in 1.12× (for 3ce3), 1.24× (for 3tmn), and 1.48× (for 7cpa) faster executions on the

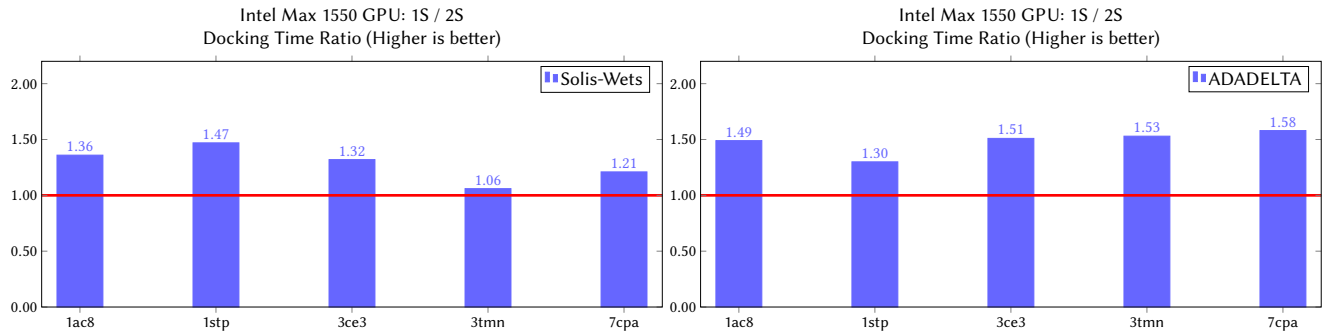

Figure 1: Docking Time Ratios (single [1S] stack / dual [2S] stack) achieved on Intel Max 1550 GPU. Ratios above 1 (i.e., horizontal red line) correspond to faster executions of the dual stack configuration

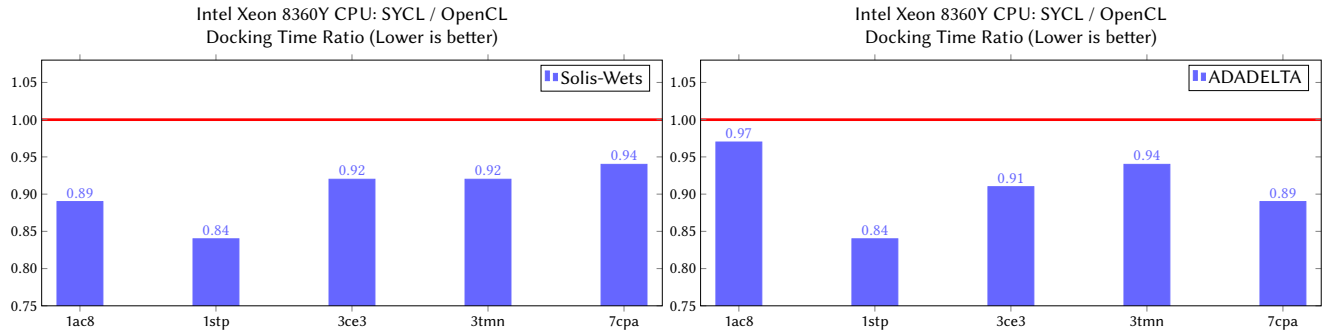

Figure 2: Docking Time Ratios (SYCL / OpenCL) achieved on Intel Xeon 8360Y CPU. Ratios below 1 (i.e., horizontal red line) correspond to faster executions of the SYCL version

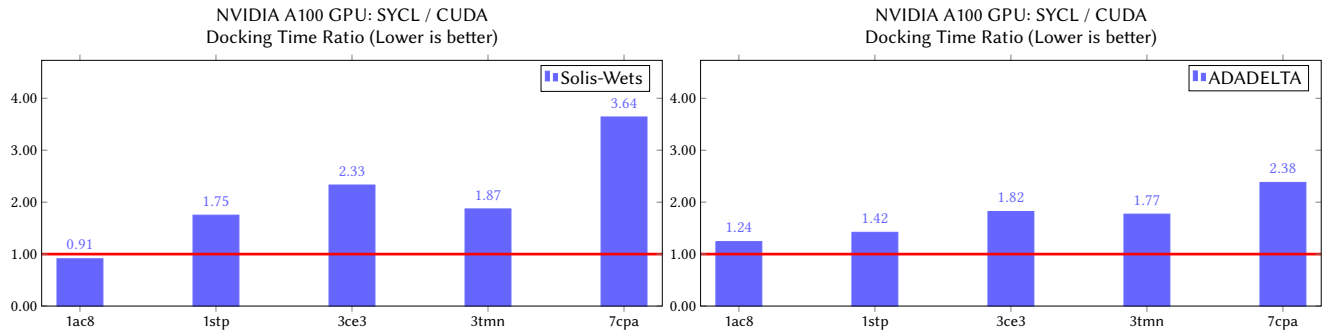

Figure 3: Docking Time Ratios (SYCL / CUDA) achieved on NVIDIA A100 GPU. Ratios below 1 (i.e., horizontal red line) correspond to faster executions of the SYCL version

A100. For ADADELTA, all test cases achieve faster executions on the Max 1550, where speedups range within {~1.16× (=$\frac{1}{0.86}$, for 7cpa), ~1.88× (=$\frac{1}{0.53}$, for 1ac8)}. All the achieved values are below the expected ratio of ~2.67× (=$\frac{52}{19.5}$), which is calculated using the theoretical FP32 performance capabilities (TFLOP/s) of both GPUs (Table 5). Based on the aforementioned compute boundness of AutoDock-GPU's algorithm, we can attribute these lower-than-expected ratios to a possible inefficient usage of compute resources in the SYCL version, specifically on the compute-intensive calls, i.e., scoring function (Algorithm 2) and gradient calculation (Algorithm 5).

As already described, these results are still preliminary, with the following limitations being present: While larger test cases provide more parallelism (i.e., more rotatable bonds and atoms) and typically result in higher speedups, this was not always the case in our experiments. Additionally, regarding the employed local-search methods (Solis-Wets and ADADELTA), we still need to investigate the performance impact of their algorithmic differences: Solis-Wets suffers from more thread divergence, while ADADELTA performs

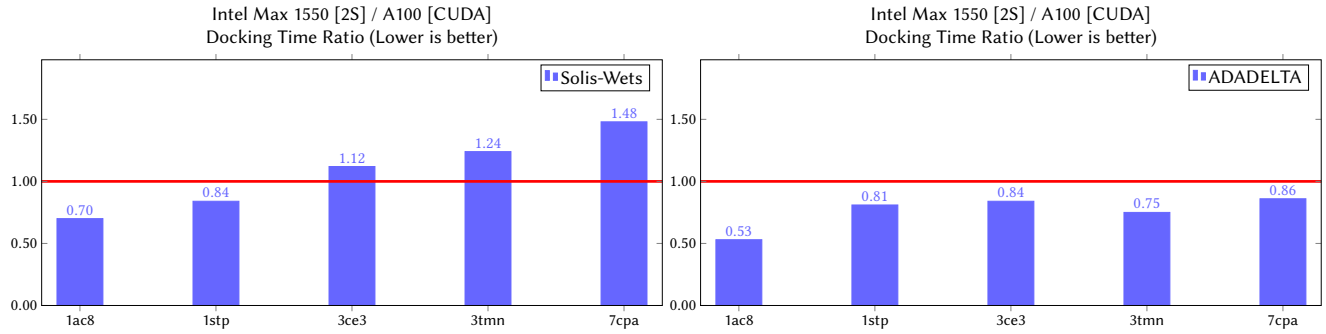

Figure 4: Docking Time Ratios: Intel Max 1550 [2S] / A100 [CUDA]. Ratios below 1 (i.e., horizontal red line) correspond to faster executions on the Intel Max 1550 GPU

Table 6: Profiles of a single ADADELTA local-search kernel execution (test case: 7cpa) on NVIDIA A100 GPU

Metrics	CUDA	SYCL	SYCL/CUDA Ratio
Time [ms]	96.4	150.9	1.6
Achieved Peak FP32 Performance [%]	10	11	1.1
Arithmetic Intensity [FLOP/byte]	989.8	1583.8	1.6
Performance [GFLOP/s]	1111	1228	1.1
Theoretical Warps per Scheduler	8	4	0.5
Registers per Thread	64	127	1.9
Static Shared Memory per Block [byte/block]	7.4	512	95.8
Dynamic Shared Memory per Block [kbyte/block]	0	1.8	$+\infty$
Achieved Occupancy [%]	38.8	19.3	0.5

more compute-intensive calls and requires heavier synchronization. These questions can be resolved by a more thorough performance analysis on the Max 1550 GPU. This is still work-in-progress, as the Intel profiling tools (e.g., roofline) still have some limitations for the very recent Max 1550 GPU, making such analysis difficult.

5 RELATED WORK

In this section, we compare our work to previous studies in terms of the migration cases addressed when employing Compatibility Tool. For this purpose, Table 7 includes different migration efforts involving code refinements for functional correctness and/or performance optimization.

As discussed in [1], the most frequently-encountered cases involve work-group/sub-group size setup (resulting in tool warnings pointing to kernel invocations) and error handling. Both cases have been addressed in three previous studies ([1, 2, 4]). In *our work here*, we explicitly address the case of work-group/sub-group sizes, but do not discuss the case of error handling. This is because Compatibility Tool correctly migrated the latter case, and thus, the corresponding migrated SYCL code required no manual intervention. Less frequent cases are synchronization, memory allocation and layout, queries of device properties, as well as atomics and barriers. All these cases are addressed in *our work here* as well as in two other studies per case. To the best of our knowledge, AutoDock-GPU presents two cases not previously discussed in the literature, namely the handling of assembly code and of native math functions.

Compatibility Tool is being continually enhanced. However, there is no single application that covers all possible migration cases. In fact, new issues in the tool, i.e., opportunities for tool enhancement, are discovered as more applications are migrated. For completeness, we show some examples of additional cases in Table 7 that were not encountered when migrating AutoDock-GPU.

Regarding time measurements, [2] replaces manually the CUDA events with `std::chrono` functions calls. This is not required for AutoDock-GPU, as it already uses that library. Image data is not processed in AutoDock-GPU, but [1, 4] require manual adjustments in that area, as SYCL supports only a 4-channel image format. Device selection code needed to be manually reviewed in [1], while for AutoDock-GPU, it was already indirectly handled with `dpct::get_default_queue()` (Section 3.1.3). Loop unrolling is employed in [15] to fully allocate data on a GPU execution unit. Furthermore, the SYCL extension called *explicit SIMD* is leveraged in [31] to gain more control over the generated code instead of relying on compiler optimizations. Migrating AutoDock-GPU did not require these two latter steps.

6 CONCLUSIONS

In this work, we have migrated AutoDock-GPU from CUDA to SYCL by employing the Intel DPC++ Compatibility Tool, which automates the migration of most CUDA into human-readable SYCL code. While Compatibility Tool greatly reduces the effort of code migration, for complex programs such as AutoDock-GPU, a developer still needs to closely analyze, manually complete, and fine-tune the migrated code. In particular, we show that the manual modifications we performed on the original tool-migrated SYCL code resulted in competitive performance on a multi-core CPU and high-end GPUs. On a 36-core × 2 sockets Intel Xeon 8360Y CPU, the migrated SYCL code is now *faster* than the original OpenCL for all local-search methods and test cases. On this CPU, the highest speedup factor achieved with SYCL compared to OpenCL was ~1.20×. For GPUs, we found that by using the two X^e-stacks available in the Intel Max 1550 GPU, it is possible to further reduce docking times by factors of up to 1.88× compared to those on an NVIDIA A100 GPU.

Table 7: Migration cases addressed in this work vs. those in previous studies

		Our work Molecular docking	[2] (2020) Tsunami simulation	[15] (2021) Rodinia & SHOC	[31] (2021) Ultrasound beamforming	[1] (2022) Rodinia	[4] (2022) Sequence alignment
Migration cases for functional correctness							
3.1.1, 3.1.2	Built-in functions (reductions, shuffles, etc)	✓	–	–	–	–	✓
3.1.3	Synchronization	✓	✓	✓	–	–	–
3.1.4	Work-group/sub-group sizes	✓	✓	–	–	✓	✓
3.1.5	Memory (allocation, layout, etc)	✓	✓	–	–	✓	–
3.1.6	Query of device properties	✓	–	–	–	✓	✓
3.1.7	Assembly code	✓	–	–	–	–	–
–	Error handling	–	✓	–	–	✓	✓
–	Timing measurements	–	✓	–	–	✓	–
–	Image format	–	–	–	–	✓	✓
–	Device selection	–	–	–	–	✓	–
Migration cases for performance optimization							
3.2.1	Memory space in atomics and barriers	✓	–	✓	–	–	✓
3.2.2	Native math functions	✓	–	–	–	–	–
–	Loop unrolling	–	–	✓	–	–	–
–	Explicit SIMD	–	–	–	✓	–	–

In future work, we plan to further increase the efficiency of the SYCL version on the Intel Max 1550 GPU, as well as to analyze its portability to a variety of high-end accelerator platforms. These porting efforts could be carried out, e.g., by also considering the interplay of docking algorithms and data-structures with the device-specific memory hierarchies, such as that on the recently-launched Intel Xeon CPU Max Series featuring on-package HBM2e memory (code-named *Sapphire Rapids HBM*) [14].

ACKNOWLEDGMENTS

This work has been supported by Intel under the oneAPI Center of Excellence Research Award granted to Technical University of Darmstadt.

REFERENCES

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

[2] Steffen Christgau and Thomas Steinke. 2020. Porting a Legacy CUDA Stencil Code to oneAPI. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 359–367. https://doi.org/10.1109/IPDPSW50202.2020.00070

[3] Codeplay. 2022. SYCL Guide. https://developer.codeplay.com/products/computecpp/ce/2.11.0/guides/sycl-guide

[4] Manuel Costanzo, Enzo Rucci, Carlos García-Sánchez, Marcelo Naiouf, and Manuel Prieto-Matías. 2022. Migrating CUDA to oneAPI: A Smith-Waterman Case Study. In *Bioinformatics and Biomedical Engineering (IWBBIO)*. Springer, 103–116. https://doi.org/10.1007/978-3-031-07802-6_9

[5] Inbal Halperin, Buyong Ma, Haim Wolfson, and Ruth Nussinov. 2002. Principles of docking: An overview of search algorithms and a guide to scoring functions. *Journal of Proteins: Structure, Function, and Bioinformatics* 47, 4 (2002), 409–443. https://doi.org/10.1002/prot.10115

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

[7] Intel. 2022. Intel Xeon Platinum 8360Y Processor. https://www.intel.com/content/www/us/en/products/sku/212459/intel-xeon-platinum-8360y-processor-54m-cache-2-40-ghz/specifications.html

[8] Intel. 2022. oneAPI Centers of Excellence. https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/academic-program/centers-of-excellence.htm

[9] Intel. 2022. oneAPI DPC++ Compiler Documentation - Considerations for Programming to Multi-Tile and Multi-Card under Level-Zero Backend. https://intel.github.io/llvm-docs/MultiTileCardWithLevelZero.html

[10] Intel. 2022. SYCL Thread Mapping and GPU Occupancy. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top/thread-mapping.html

[11] Intel. 2022. SYCLomatic: A New CUDA-to-SYCL Code Migration Tool. https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html

[12] Intel. 2022. X^e-HPC GPU Architecture. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top/xe-arch.html

[13] Intel. 2023. Intel Data Center GPU Max 1550. https://ark.intel.com/content/www/us/en/ark/products/232873/intel-data-center-gpu-max-1550.html

[14] Intel. 2023. Intel Launches 4th Gen Xeon Scalable Processors, Max Series CPUs. https://www.intel.com/content/www/us/en/newsroom/news/4th-gen-xeon-scalable-processors-max-series-cpus-gpus.htm

[15] Zheming Jin and Jeffrey Vetter. 2021. Evaluating CUDA Portability with HIPCL and DPCT. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 371–376. https://doi.org/10.1109/IPDPSW52791.2021.00065

[16] Khronos Group. 2022. SYCL Resources. https://www.khronos.org/sycl/resources

[17] Scott LeGrand, Aaron Scheinberg, Andreas F. Tillack, Mathialakan Thavappiragasam, Josh V. Vermaas, Rupesh Agarwal, Jeff Larkin, Duncan Poole, Diogo Santos-Martins, Leonardo Solis-Vasquez, Andreas Koch, Stefano Forli, Oscar Hernandez, Jeremy C. Smith, and Ada Sedova. 2020. GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research. In *Proceedings of the 11th International Conference on Bioinformatics, Computational Biology and Health Informatics*. ACM. https://doi.org/10.1145/3388440.3412472

[18] Garrett M. Morris, David S. Goodsell, Robert S. Halliday, Ruth Huey, William E. Hart, Richard K. Belew, and Arthur J. Olson. 1998. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry* 19, 14 (1998), 1639–1662. https://doi.org/10.1002/(SICI)1096-987X(19981115)19:14<1639::AID-JCC10>3.0.CO;2-B

[19] NVIDIA. 2022. Accelerated Apps Catalog. https://www.nvidia.com/en-us/gpu-accelerated-applications

[20] NVIDIA. 2022. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100

[21] NVIDIA. 2022. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute

[22] oneAPI. 2022. oneAPI Spec Elements. https://www.oneapi.io/spec

[23] Diogo Santos-Martins, Leonardo Solis-Vasquez, Andreas F Tillack, Michel F Sanner, Andreas Koch, and Stefano Forli. 2021. Accelerating AutoDock4 with GPUs and Gradient-Based Local Search. *Journal of Chemical Theory and Computation* 17, 2 (2021), 1060–1073. https://doi.org/10.1021/acs.jctc.0c01006

[24] Scripps Research. [n. d.]. AutoDock-GPU: AutoDock for GPUs and other accelerators. https://github.com/ccsb-scripps/AutoDock-GPU

[25] Francisco J. Solis and Roger J. B. Wets. 1981. Minimization by Random Search Techniques. *Journal of Mathematics of Operations Research* 6, 1 (1981), 19–30. https://doi.org/10.1287/moor.6.1.19

[26] Leonardo Solis-Vasquez, Erich Focht, and Andreas Koch. 2021. Mapping Irregular Computations for Molecular Docking to the SX-Aurora TSUBASA Vector Engine.

In *Proceedings of the 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 1–10. https://doi.org/10.1109/IA354616.2021.00008

[27] Leonardo Solis-Vasquez and Andreas Koch. 2018. A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software. In *Proceedings of the 5th International Workshop on FPGAs for Software Programmers (FSP)* (Dublin, Ireland). VDE Verlag, 1–10.

[28] Leonardo Solis-Vasquez, Diogo Santos-Martins, Andreas Tillack, Andreas F. Koch, Jérôme Eberhardt, and Stefano Forli. 2020. Parallelizing Irregular Computations for Molecular Docking. In *Proceedings of the 10th International Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 12–21. https://doi.org/10.1109/IA351965.2020.00008

[29] Leonardo Solis-Vasquez, Andreas F. Tillack, Diogo Santos-Martins, Andreas Koch, Scott LeGrand, and Stefano Forli. 2022. Benchmarking the performance of irregular computations in AutoDock-GPU molecular docking. *Parallel Comput.* 109 (2022), 102861. https://doi.org/10.1016/j.parco.2021.102861

[30] Mathialakan Thavappiragasam, Aaron Scheinberg, Wael Elwasif, Oscar Hernandez, and Ada Sedova. 2020. Performance Portability of Molecular Docking Miniapp On Leadership Computing Platforms. In *Proceedings of the International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 36–44. https://doi.org/10.1109/P3HPC51967.2020.00009

[31] 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 *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 360–370. https://doi.org/10.1109/IPDPSW52791.2021.00064

[32] World Community Grid. 2021. OpenPandemics - COVID-19 Now Running on Machines with Graphics Processing Units. https://www.worldcommunitygrid.org/about_us/viewNewsArticle.do?articleId=693

[33] Matthew D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. *arXiv* abs/1212.5701 (2012).

A AUTODOCK-GPU'S EXECUTION CONFIGURATION

For all executions, we set the number of LGA runs as follows: $N_{\text{LGA-runs}}^{\text{TOTAL}} = 100$. Additionally, we disable the default early-termination mechanisms of AutoDock-GPU [29], i.e., *autostop* and *heuristic*, in order to process similar workloads across executions using the same local-search method and test case. The SYCL version of AutoDock-GPU used here is available under https://github.com/ccsb-scripps/AutoDock-GPU/pull/183 (commit hash: 72e4309).