Migrating AutoDock-GPU: a Drug Screening Code to Data Parallel C++
Tips and Techniques to make your CUDA Migration Effort Easier

Cross-Architecture Programming for Accelerated Compute, Freedom of Choice for Hardware

Edward Mascarenhas, PhD
Intel Corp.

October 26, 2021
NOTICES AND DISCLAIMERS

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

Results have been estimated or simulated.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All product plans and roadmaps are subject to change without notice.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
Agenda

1. oneAPI and Intel® DPC++ Compatibility Tool (DPCT) Brief Overview
2. AutoDock-GPU Overview
3. Migration Challenges
4. Getting to Functional Code
5. Key Takeaways
Agenda

• oneAPI and DPCT Brief Overview
• AutoDock-GPU Overview
• Migration Challenges
• Getting to Functional Code
• Key Takeaways
oneAPI
One Programming Model for Multiple Architectures and Vendors

Freedom to Make Your Best Choice
- Choose the best accelerated technology the software doesn’t decide for you

Realize all the Hardware Value
- Performance across CPU, GPUs, FPGAs, and other accelerators

Develop & Deploy Software with Peace of Mind
- Open industry standards provide a safe, clear path to the future
- Compatible with existing languages and programming models including C++, Python, SYCL, OpenMP, Fortran, and MPI
Data Parallel C++
Standards-based, Cross-architecture Language

DPC++ = ISO C++ and Khronos SYCL and community extensions

Freedom of Choice: Future-Ready Programming Model
- Allows code reuse across hardware targets
- Permits custom tuning for a specific accelerator
- Open, cross-industry alternative to proprietary language

DPC++ = ISO C++ and Khronos SYCL and community extensions
- Delivers C++ productivity benefits, using common, familiar C and C++ constructs
- Adds SYCL from the Khronos Group for data parallelism and heterogeneous programming

Community Project Drives Language Enhancements
- Provides extensions to simplify data parallel programming
- Continues evolution through open and cooperative development

The open source and Intel DPC++/C++ compiler supports Intel CPUs, GPUs, and FPGAs. Codeplay announced a DPC++ compiler that targets Nvidia GPUs.
Intel® DPC++ Compatibility Tool
Minimizes Code Migration Time

Assists developers migrating code written in CUDA to DPC++ once, generating human readable code wherever possible.

Typically, 90-95% of CUDA code automatically migrates to DPC++ code.*

Inline comments are provided to help developers finish porting the application.

*Intel estimates as of September 2021. Based on measurements on a set of 70 HPC benchmarks and samples, with examples like Rodinia, SHOC, PENNANT. Results may vary.
Migration Flow

Typical preparation steps for simple to complex projects

Prepare
intercept-build

Migrate
dpct

Review
Verify & Manually Edit

Vector-Add Example: DPCT migrates many CUDA APIs correctly

CUDA

```c
#include <cuda.h>
#include <stdio.h>
#define VECTOR_SIZE 256

__global__ void VectorAddKernel(float* A, float* B, float* C)
{
    A[threadIdx.x] = threadIdx.x + 1.0f;
    B[threadIdx.x] = threadIdx.x + 1.0f;
    C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
}
```

DPC++

```c
#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#include <stdio.h>
#define VECTOR_SIZE 256

void VectorAddKernel(float* A, float* B, float* C,
    sycl::nd_item<3> item_ct1)
{
    A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
    B[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
    C[item_ct1.get_local_id(2)] =
        A[item_ct1.get_local_id(2)] + B[item_ct1.get_local_id(2)];
}
```

https://github.com/oneapi-src/oneAPI-samples/tree/master/Tools/Migration/vector-add-dpct
Vector-Add Example: DPCT migrates many CUDA APIs correctly

CUDA

```c
int main()
{
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, VECTOR_SIZE*sizeof(float));
    cudaMalloc(&d_B, VECTOR_SIZE*sizeof(float));
    cudaMalloc(&d_C, VECTOR_SIZE*sizeof(float));
    VectorAddKernel<<<1, VECTOR_SIZE>>>(d_A, d_B, d_C);
    float Result[VECTOR_SIZE] = { }; 
    cudaMemcpy(Result, d_C, VECTOR_SIZE*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

DPC++

```c
int main()
{
    dpct::device_ext &dev_ct1 = dpct::get_current_device();
    sycl::queue &q_ct1 = dev_ct1.default_queue();
    float *d_A, *d_B, *d_C;
    d_A = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
    d_B = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
    d_C = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
    sycl::range<3> global = sycl::range<3>(1, 1, VECTOR_SIZE);
    sycl::range<3> local = sycl::range<3>(1, 1, VECTOR_SIZE);
    sycl::nd_item<3> item_ct1 = sycl::nd_range<3>(global, local);
    item_ct1.submit([&](sycl::handler &h) {
        VectorAddKernel(d_A, d_B, d_C, item_ct1);
    });
    float Result[VECTOR_SIZE] = { }; 
    sycl::memcpy(Result, d_C, VECTOR_SIZE * sizeof(float)).wait();
    sycl::free(d_A, q_ct1);
    sycl::free(d_B, q_ct1);
    sycl::free(d_C, q_ct1);
}
```

https://github.com/oneapi-src/oneAPI-samples/tree/master/Tools/Migration/vector-add-dpct
Agenda

- oneAPI and DPCT Brief Overview
- AutoDock-GPU Overview
- Migration Challenges
- Getting to Functional Code
- Key Takeaways
Overview of Molecular Docking

Virtual Screening (Optimization of Scoring function)

Score

Conformation

Optimal Docked position

Docking Data Preparation

Protein and Ligand Databases

Wet Lab Screening and Testing

Potential Drugs

AutoDock-GPU is a complex HPC code to migrate to DPC++.
AutoDock-GPU Original Code Organization can be maintained after migration

DPC++ code migrated from CUDA base using Intel® DPC++ Compatibility Tool.
AutoDock-GPU Migration: Apply the 3-step flow

Prepare
- git clone https://github.com/ccsb-scripps/AutoDock-GPU.git
- git checkout a544015a5d79c7da8b3e4443642fbf9dce680ea4
- intercept-build make DEVICE=CUDA NUNWI=64 GPU_INCLUDE_PATH=<path>/cuda-11.1/include
  GPU_LIBRARY_PATH=<path>/cuda-11.1/lib

Migrate
- dpct --in-root . --out-root=dpct_adock --cuda-include-path=<path>/cuda-11.1/include
  -p compile_commands.json
- Move migrated files to dpcpp, host, common directories
- Create Makefile.dpcpp for DEVICE=XeGPU

Review
- Review each diagnostic messages using reference and manually edit.
- Pay close attention to comments when DPCT indicates issues with migration as these changes are likely to need modification.
AutoDock-GPU kernel Example: DPCT does a great job of migrating this kernel

CUDA

```c
__global__ void
gpu_sum_evals_kernel()
{
    // The GPU global function sums the evaluation counter states
    // which are stored in evals_of_new_entities array foreach entity,
    // calculates the sums for each run and stores it in evals_of_runs array.
    // The number of blocks which should be started equals to num_of_runs,
    // since each block performs the summation for one run.
    __shared__ int sSum_evals;
    int partsum_evals = 0;
    int* pEvals_of_new_entities = cData.pMem_evals_of_new_entities + blockIdx.x
                              * cData.dockpars.pop_size;
    for (int entity_counter = threadIdx.x;
           entity_counter < cData.dockpars.pop_size;
           entity_counter += blockDim.x)
    {
        partsum_evals += pEvals_of_new_entities[entity_counter];
    }
    // Perform warp-wise reduction
    if (threadIdx.x == 0)
    {
        cData.pMem_gpu_evals_of_runs[blockIdx.x] += sSum_evals;
    }
}
```

DPC++

```c
gpu_sum_evals_kernel(sycl::nd_item<3> itemCtl, GpuData cData,
int *sSum_evals)
{
    int partsum_evals = 0;
    int* pEvals_of_new_entities = cData.pMem_evals_of_new_entities +
                                 itemCtl.get_group(2) * cData.dockpars.pop_size;
    for (int entity_counter = itemCtl.get_local_id(2);
         entity_counter < cData.dockpars.pop_size;
         entity_counter += itemCtl.get_local_range().get(2))
    {
        partsum_evals += pEvals_of_new_entities[entity_counter];
    }
    // Perform warp-wise reduction
    if (itemCtl.get_local_id(2) == 0)
    {
        cData.pMem_gpu_evals_of_runs[itemCtl.get_group(2)] += *sSum_evals;
    }
}
```

https://github.com/ccsb-scripps/AutoDock-GPU
AutoDock-GPU kernel Example: DPCT does a great job of migrating this kernel

CUDA

```c
void gpu_sum_evals(uint32_t blocks, uint32_t threadsPerBlock)
{
    gpu_sum_evals_kernel<<<blocks, threadsPerBlock>>>();
}
```

DPC++

```c
void gpu_sum_evals(uint32_t blocks, uint32_t threadsPerBlock)
{
    /*
     * DPCT1049:42: 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.
     */
    dpc::get_default_queue().submit([&](sycl::handler &cgh) {
        extern dpc::constant_memory<GpuData, 0> cData;
        cData.init();
        auto cData_ptr_ctl = cData.get_ptr();
        sycl::accessor<int, 0, sycl::access::mode::read_write,
        sycl::access::target::local>
        sSum_evals_acc_ctl(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_ctl) {
                gpu_sum_evals_kernel(
                    item_ctl, *cData_ptr_ctl,
                    sSum_evals_acc_ctl.get_pointer());
            });
    });
```
## DPCT migrates these cases plus more

<table>
<thead>
<tr>
<th>CUDA</th>
<th>DPC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread; Warp; Block</td>
<td>Work-item sycl::nd_item; Sub-group sycl::sub-group; Work-group sycl::group</td>
</tr>
<tr>
<td>threadIdx.x</td>
<td>item_ct1.get_local_id(2) for 3 dimensional range</td>
</tr>
<tr>
<td>blockDim.x</td>
<td>item_ct1.get_local_range().get(2) for 3 dimensional range</td>
</tr>
<tr>
<td>blockIdx.x</td>
<td>item_ct1.get_group(2) for 3 dimensional range</td>
</tr>
<tr>
<td>cudaMalloc</td>
<td>sycl::malloc_device</td>
</tr>
<tr>
<td>cudaMallocManaged</td>
<td>sycl::malloc_shared</td>
</tr>
<tr>
<td>cudaFree</td>
<td>sycl::free</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>dpct::get_default_queue().memcpy(...).wait()</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>GPU shared local memory</td>
</tr>
<tr>
<td></td>
<td>sycl::accessor&lt;int, 0, sycl::access::mode::read_write, sycl::access::target::local&gt;</td>
</tr>
<tr>
<td>kernel&lt;&lt;blocks,</td>
<td>dpct::get_default_queue().submit([&amp;](sycl::handler &amp;cgh) {</td>
</tr>
<tr>
<td>threadsPerBlock&gt;&gt;&gt;()</td>
<td>cgh.parallel_for (sycl::nd_range&lt;3&gt;(sycl::range&lt;3&gt;(1, 1, blocks) * sycl::range&lt;3&gt;(1, 1, threadsPerBlock), sycl::range&lt;3&gt;(1, 1, threadsPerBlock)), [=](sycl::nd_item&lt;3&gt; item_ctl) { kernel(item_ctl); }); });</td>
</tr>
</tbody>
</table>
Agenda

- oneAPI and DPCT Brief Overview
- AutoDock-GPU Overview
- Migration Challenges
- Getting to Functional Code
- Key Takeaways
AutoDock-GPU and getting to functional correctness

Code compiles and builds, but clearly functionally incorrect.
1. Reductions and Multi-Line Macros

```c
#define REDUCEINTEGERSUM(value, pAccumulator) 
{
  if (threadIdx.x == 0) 
  { 
    *pAccumulator = 0; 
  }
  __threadfence(); 
  __syncthreads(); 
  if (__any_sync(0xfffffffff, value != 0)) 
  { 
    uint32_t tgx = threadIdx.x & cData.warpmask; 
    value += __shfl_sync(0xfffffffff, value, tgx ^ 1); 
    value += __shfl_sync(0xfffffffff, value, tgx ^ 2); 
    value += __shfl_sync(0xfffffffff, value, tgx ^ 4); 
    value += __shfl_sync(0xfffffffff, value, tgx ^ 8); 
    value += __shfl_sync(0xfffffffff, value, tgx ^ 16); 
    if (tgx == 0) 
    { 
      atomicAdd(pAccumulator, value); 
    }
  }
  __threadfence(); 
  __syncthreads(); 
  value = *pAccumulator; 
  __syncthreads(); 
}
```

**DPCT Warnings**

/*DPCT1023:40: The DPC++ sub-group does not support mask options for sycl::ONEAPI::any_of.*/
/* DPCT1023:41: The DPC++ sub-group does not support mask options for shuffle.*/
/*DPCT1007:39: Migration of this CUDA API is not supported by the Intel(R) DPC++ Compatibility Tool.*/

Take advantage of built-in collectives, functions in DPC++ when possible.

2. Finding the Minimum

Subgroups
We explicitly set the sub-group size to 32. Migrated CUDA code implicitly assumes work-items will be executed in groups of size 32.
AutoDock-GPU and getting to functional correctness: Are we there yet?

Code compiles and builds, but clearly functionally incorrect. Evaluations are computed. That is Progress!
Atomics in local memory and sizeof(sycl::float3)

3. Atomics – edit to change to do atomics in local memory.

4. sycl::float3 vs float3 and sycl::int3 vs int3 memory size
   - AutoDock-GPU code implies float3 as 12 bytes whereas sycl::float3 is 16 bytes.
   - Memory layout of some arrays depend on such “incorrect” assumptions.
   - This causes silent memory corruption affecting energy calculations.
Using Shared Local Memory

- DPC++ use of shared local memory is through the `sycl::accessor` capability.
- CUDA `__shared__` declaration.
- For most cases DPCT handles this but for reduce accumulators and atomics DPCT assumed global memory.
Synchronization and Memory available calculation

5. Synchronization
   - DPCT migrated the CUDA `barrier()` to `work_item.barrier()`.
   - CUDA `__threadfence()` was manually migrated as a no-op (as it precedes a `__syncthreads()` call) whereas `__syncthreads()` is migrated as `work_item.barrier()`.
   - `cudaDeviceSynchronize()` was manually migrated to `dpct::get_current_device().queues_wait_and_throw()`.

6. Memory available calculation during setup
   - DPCT does not migrate `cudaMemGetInfo()` and this CUDA call was manually migrated using `get_device_info()` and `get_global_mem_size()`.
Error Handling and Assembly code

7. Error Handling
   - CUDA uses error checks from function call returns, whereas DPC++ uses exception handling.
   - DPCT’s handling of error checking reduced substantial effort of migrating error handling interspersed throughout the code.
   - Review the code where DPCT makes changes to handle errors.

8. Assembly code
   - DPCT does not migrate assembly code.
   - AutoDock-GPU has few lines of assembly which could be easily converted to C++ code manually.
AutoDock-GPU and getting to functional correctness

Code compiles and builds, but clearly functionally incorrect.

Some values are computed. Progress!

Both evaluations and energy calculated. Functionally correct!
• oneAPI and DPCT Brief Overview
• AutoDock-GPU Overview
• Migration Challenges
• Getting to Functional Code
• Key Takeaways
### Functional Correctness – 5 Test cases from GitHub repository

```
./bin/autodock_gpu_64wi -ffile input/3ce3/derived/3ce3_protein.maps.fld -lfile input/3ce3/derived/3ce3_ligand.pdbqt -nrun 100
```

<table>
<thead>
<tr>
<th>Protein/Ligand input test case</th>
<th>Best-Energy Result in kcal/mol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CUDA</td>
</tr>
<tr>
<td>1ac8</td>
<td>-5.70</td>
</tr>
<tr>
<td>1stp</td>
<td>-9.98</td>
</tr>
<tr>
<td>3ce3</td>
<td>-14.55</td>
</tr>
<tr>
<td>3tmn</td>
<td>-9.75</td>
</tr>
<tr>
<td>7cpa</td>
<td>-21.46</td>
</tr>
</tbody>
</table>

- The best-energy docking results for the CUDA version and the DPCPP version match closely.
- Other test cases were also executed with similar results.
Debugging: use gdb-oneapi batch feature

1. **Debug on host**
   - This is a key advantage of using DPC++. oneAPI code will run on host.

2. **gdb-oneapi**
   - gdb is enhanced to work with Xe GPUs, but there are some limitations at this time to debug on the device.
   - Full functionality is being worked on for Xe GPUs.

3. **Comparative results and key program run-time state from CUDA vs. DPC++**
   - Use gdb-oneapi batch feature to print variables at the same point in the program for offline comparison.
   - AutoDock-GPU has a feature to use same random number stream for reproducibility.
Debugging multi-page functions with printf

4. printf
   • Used printf liberally for offloaded kernel code debugging.
   • Two options:
     ▪ Experimental printf invoked as `sycl::ONEAPI::experimental::printf()`.
     ▪ Another way to print from an offloaded kernel is using `sycl::stream`.
   • Limitation: Host and Device printf are not in-order of execution.

5. Binary search and multi-page functions
   • For large multi-page compute functions -- comparative debugging is tedious.
   • Use binary search with printf or gdb batch feature to quickly locate the area of the code that is causing incorrect results.
Agenda

• oneAPI and DPCT Brief Overview
• AutoDock-GPU Overview
• Migration Challenges
• Getting to Functional Code
• Key Takeaways
Key Takeaways

- Intel® DPC++ Compatibility Tool (DPCT) saves substantial time and effort in migrating complex CUDA applications like AutoDock-GPU.
- Migrated AutoDock-GPU runs successfully on Intel dGPU.
- Manual review of DPCT diagnostics highly recommended. Edit code as required for correctness.
- Use suggested tips and techniques to get to functional correctness faster. Your feedback to improve oneAPI tools like DPCT is highly appreciated.
References

- Migrating your Existing CUDA Code to DPC++, IXPUG Webinar, July 2020.
- Intel DPC++ Compatibility Tool Developer Guide and Reference.
- Intel DPC++ Compatibility Tool diagnostic messages reference.
- Data Parallel C++ Book
- IWOCL2017_MolecularDocking_online.pdf – AutoDock-GPU GitHub
Questions
Thank you