Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study

Leonardo Solis-Vasquez, TU Darmstadt
Edward Mascarenhas, Intel Corporation
Andreas Koch, TU Darmstadt
Molecular Docking

- Key method in computer-aided drug design
  - Simulates the close-distance interaction of two molecules
  - Aims to predict their energetically-strong binding poses

Binding between a ligand and a receptor of the “3ptb” molecular complex
AutoDock-GPU

- A parallel molecular docking application co-developed at TU Darmstadt
  - OpenCL: original version [IWOCL 2017, JCTC 2021]
  - CUDA: COVID-19 research on the Summit supercomputer [BCB 2020]

- Challenging code structure and high practical relevance

- Promising case study for evaluating automated CUDA-to-SYCL migration
  - Employing DPC++ Compatibility Tool (aka SYCLomatic)
Contents

• Migrating AutoDock-GPU to SYCL

• Evaluation on CPUs and GPUs
MIGRATING AUTODOCK-GPU TO SYCL
Migration Approach to SYCL: Using CUDA as Baseline

• Our strategy
  • Using as much as possible the output of Compatibility Tool v2021.2.0
  • Reviewing and manually completing the tool-assisted migration
    • Using tool guidelines provided as code comments

• Our classification of migration cases
  • Functional correctness
  • Performance optimization
Functional Correctness
Summary of Migration Cases

• Reductions

• Shuffles

• Synchronization

• Memory layout for vector data types
Reductions Compatibility Tool in Action

- Used to keep track of the number of score evaluations
- Implemented in CUDA as multi-line macros performing shuffles
- Compatibility Tool cannot migrate shuffles correctly, warning:

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

/*
DPCCT1023:57: The DPC++ sub-group does not support mask options for shuffle.
*/
Reductions
Manually Completing the Migration

We use the equivalent built-in SYCL `reduce_over_group()` function directly, instead of fixing the shuffles.

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

SYCL collective functions (e.g., `reduce_over_group()`) should be leveraged whenever possible.
Shuffles

Compatibility Tool in Action

- Block of CUDA threads perform shuffles to find minimum score

```c
#define WARPMINIMUMEXCHANGE(tgx, v0, k0, mask) \
{ \
    float v1 = v0; int k1 = k0; \
    int otgx = tgx ^ mask; \
    float v2 = __shfl_sync(0xffffffff, v0, otgx); \
    int k2 = __shfl_sync(0xffffffff, k0, otgx); \
    int flag = ((v1<v2) ^ (tgx>otgx)) && (v1!=v2); \
    k0 = flag ? k1 : k2; \
    v0 = flag ? v1 : v2; \
}
```

- Compatibility Tool performs incorrect variable substitution
  - Warning: SYCL sub-group shuffle does not support extra mask argument
Shuffles
Manually Completing the Migration

• We insert the SYCL sub-group shuffles and fix the incorrect variable use

```cpp
#define WARPMINIMUMEXCHANGE(tgx, v0, k0, mask) \
{
  float v1 = v0; int k1 = k0; \
  int otgx = tgx ^ mask; \
  float v2 = __shfl_sync(0xffffffff, v0, otgx); \
  int k2 = __shfl_sync(0xffffffff, k0, otgx); \
  int flag = ((v1<v2) ^ (tgx>otgx)) && (v1!=v2); \
  k0 = flag ? k1 : k2; \
  v0 = flag ? v1 : v2; \
}
```

```cpp
#define WARPMINIMUMEXCHANGE(tgx, v0, k0, mask) \
{
  float v1 = v0; int k1 = k0; \
  int otgx = tgx ^ mask; \
  float v2 = wi.get_sub_group().shuffle(v0, otgx); \
  int k2 = wi.get_sub_group().shuffle(k0, otgx); \
  ...
}
```
Synchronization Compatibility Tool in Action

CUDA version of AutoDock-GPU:

- `__threadfence()` is always followed by `__syncthreads()`

Compatibility Tool:

- Leaves `__threadfence()` as is
- Migrates `__syncthreads()` to a SYCL barrier()
Synchronization
Manually Completing the Migration

• A SYCL `barrier()` ensures that
  • The specified memory space is consistent across all work-items within a work-
    group → equivalent to `__threadfence()`
  • Each work-item within a work-group reaches a barrier call → equivalent to `__syncthreads()`

• Our actions
  • Keep the tool-migrated `barrier()`
  • Migrate the untouched `__threadfence()` as a no-op
Memory Layout for Vector Data Types Compatibility Tool in Action

• CUDA version of AutoDock-GPU:
  • Host: allocates memory using sizeof(float3)
  • Device: performs pointer arithmetic relying on above memory allocation

• Compatibility Tool **correctly migrates** CUDA `float3` to SYCL `sycl::float3`  

• However, a different number of bytes is allocated in each case
  • 12 bytes (CUDA) vs. 16 bytes (SYCL)
Memory Layout for Vector Data Types
Manually Completing the Migration

• Discrepancy in allocated memory between CUDA and SYCL versions
  • Problem: incorrect score evaluations
  • Reason: silent memory corruption overwriting portions of shared memory reserved for other variables

We correct it by explicitly propagating (from host to device) …
  • … the size of the allocated memory (based on `sycl::float3`) …
  • … via pointer to its corresponding SYCL accessor
Performance Optimization

Summary of Migration Cases

- Atomics
- Barriers
- Native math functions
Atomics
Compatibility Tool in Action

• Compatibility Tool
  • Assumes the memory order / scope / address space …
    • .. are always declared as acq_rel / device / global respectively
  • Uses such configurations in the migrated SYCL code
    • But these are not strictly required in all cases

• More-relaxed configurations may be safe to use instead
  • Reduce synchronization effort
Atomics
Manually Tuning the Migration

```c
#define ATOMICADDI32(pAccumulator, value) \natomicAdd(pAccumulator, (value))
```

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

- `acq_rel → relaxed`
- `device → work-group`
- `global → local`
Barriers
(Similar to Atomics)

• Compatibility Tool
  • Assumes that the memory address space is always declared as **global**
  • Uses that configuration in the migrated SYCL code
• A local address space may be safe to use instead

```cpp
__syncthreads();

wi.barrier(sycl::access::fence_space::local_space);
```
Native Math Functions

- Compatibility Tool **correctly migrates** math functions

- CUDA and OpenCL versions of AutoDock-GPU
  - Leverage reduced-precision and native math functions

- We replace automatically-migrated calls with **native** counterparts
  - E.g.: `sycl::sqrt()` → `sycl::native::sqrt()`
EVALUATION ON CPUS AND GPUS
Summary of Performance Comparison

1. Intel Xeon Platinum 8360Y CPU
   SYCL vs. OpenCL

2. NVIDIA A100 80GB GPU
   SYCL vs. CUDA

3. Intel Max 1550 GPU
   [“Ponte Vecchio”] (SYCL)
   vs.
   NVIDIA A100 80GB (CUDA)
Input Molecules

<table>
<thead>
<tr>
<th></th>
<th>1ac8</th>
<th>1stp</th>
<th>3ce3</th>
<th>3tmn</th>
<th>7cpa</th>
</tr>
</thead>
<tbody>
<tr>
<td># Rotatable bonds</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td># Atoms</td>
<td>8</td>
<td>18</td>
<td>37</td>
<td>27</td>
<td>43</td>
</tr>
</tbody>
</table>

[https://github.com/ccsb-scripps/AutoDock-GPU]
Local Search in AutoDock-GPU

- **AutoDock-GPU = [Genetic Algorithm] + [Local Search]**
- **Two alternative methods for Local Search**

- **Local search is driven by score optimization**
  - > 90% total AutoDock-GPU’s execution time
Intel Xeon Platinum 8360Y CPU
Docking Time Ratios: SYCL / OpenCL

<table>
<thead>
<tr>
<th>Input molecules</th>
<th>Solis-Wets</th>
<th>ADADELT A</th>
<th>Ratio = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ac8</td>
<td>0.89</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>1stp</td>
<td>0.84</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>3ce3</td>
<td>0.92</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>3tmn</td>
<td>0.92</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>7cpa</td>
<td>0.94</td>
<td>0.89</td>
<td></td>
</tr>
</tbody>
</table>
Intel Xeon Platinum 8360Y CPU
Docking Time Ratios: SYCL / OpenCL

**Ratios below 1:**
SYCL is faster than OpenCL

**Optimization pipelines specific to SYCL:**
- pre-SPIR-V optimizations
- inlining heuristics, etc

**Input molecules**
- 1ac8
- 1stp
- 3ce3
- 3tmn
- 7cpa

**Maximum Speed Up (SYCL wrt. OpenCL)**
= ~1.2x (1 / 0.84)
NVIDIA A100 80GB GPU
Docking Time Ratios: SYCL / CUDA

<table>
<thead>
<tr>
<th>Input molecules</th>
<th>Solis-Wets</th>
<th>ADADELTa</th>
<th>Ratio = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ac8</td>
<td>0.91</td>
<td>1.24</td>
<td>1</td>
</tr>
<tr>
<td>1stp</td>
<td>1.75</td>
<td>1.42</td>
<td>2.33</td>
</tr>
<tr>
<td>3ce3</td>
<td>2.33</td>
<td>1.82</td>
<td>1.87</td>
</tr>
<tr>
<td>3tmn</td>
<td>1.87</td>
<td>1.77</td>
<td>1.77</td>
</tr>
<tr>
<td>7cpa</td>
<td>3.64</td>
<td>2.38</td>
<td>2.38</td>
</tr>
</tbody>
</table>
### NVIDIA A100 80GB GPU

Docking Time Ratios: SYCL / CUDA

<table>
<thead>
<tr>
<th>Input molecules</th>
<th>Solis-Wets</th>
<th>ADADELT A</th>
<th>Ratio</th>
<th>CUDA is faster than SYCL</th>
<th>SYCL faster than CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ac8</td>
<td>0.91</td>
<td>1.24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1stp</td>
<td>1.75</td>
<td>1.42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3ce3</td>
<td>2.33</td>
<td>1.82</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3tmn</td>
<td>1.87</td>
<td>1.77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7cpa</td>
<td>3.64</td>
<td>2.38</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Ratios above 1:**

- Solis-Wets + 1ac8: SYCL is ~1.1x (1 / 0.91) faster than CUDA
SYCL and CUDA versions are compute bound (roofline analysis)

SYCL is apparently performing more computations than CUDA

- Based on [FLOP/byte] and [GFLOP/s] numbers
- Possibly due to a mismatch in the arithmetic precision
Profiling Using Nsight Compute
ADADELTA (7cpa) on A100 80GB GPU

<table>
<thead>
<tr>
<th>Metrics (For a single kernel execution)</th>
<th>CUDA</th>
<th>SYCL</th>
<th>SYCL / CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical Warps per Scheduler</td>
<td>8</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Registers per Thread</td>
<td>64</td>
<td>127</td>
<td>1.9</td>
</tr>
<tr>
<td>Achieved Occupancy [%]</td>
<td>38.8</td>
<td>19.3</td>
<td>0.5</td>
</tr>
</tbody>
</table>

- # theoretical warps: SYCL = $\frac{1}{2}$ CUDA
Profiling Using Nsight Compute
ADADELTA (7cpa) on A100 80GB GPU

<table>
<thead>
<tr>
<th>Metrics (For a single kernel execution)</th>
<th>CUDA</th>
<th>SYCL</th>
<th>SYCL / CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical Warps per Scheduler</td>
<td>8</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Registers per Thread</td>
<td>64</td>
<td>127</td>
<td>1.9</td>
</tr>
<tr>
<td>Achieved Occupancy [%]</td>
<td>38.8</td>
<td>19.3</td>
<td>0.5</td>
</tr>
</tbody>
</table>

- # theoretical warps: SYCL = ½ CUDA
- Register pressure: SYCL is 1.9x higher than CUDA
- Higher register pressure → lower kernel occupancy: SYCL = ½ CUDA
  - Could be prevented by setting the maximum number of registers per thread
    - NVCC compiler option: -maxrregcount
**Max 1550 (SYCL) vs. A100 80GB (CUDA) Docking Time Ratios**

<table>
<thead>
<tr>
<th>Input molecules</th>
<th>Solis-Wets</th>
<th>ADADELT A</th>
<th>Ratio = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ac8</td>
<td>0.70</td>
<td>0.53</td>
<td>1</td>
</tr>
<tr>
<td>1stp</td>
<td>0.84</td>
<td>0.81</td>
<td>1</td>
</tr>
<tr>
<td>3ce3</td>
<td>1.12</td>
<td>0.84</td>
<td>1</td>
</tr>
<tr>
<td>3tmn</td>
<td>1.24</td>
<td>0.75</td>
<td>1</td>
</tr>
<tr>
<td>7cpa</td>
<td>1.48</td>
<td>0.86</td>
<td>1</td>
</tr>
</tbody>
</table>
Max 1550 (SYCL) vs. A100 80GB (CUDA) Docking Time Ratios

Ratios below 1: Max 1550 is faster than A100 80GB

Solis-Wets
Maximum Speed Up
(Max 1550 wrt. A100 80GB)
= ~1.42x (1 / 0.70)

Input molecules

<table>
<thead>
<tr>
<th>Input molecules</th>
<th>Solis-Wets</th>
<th>ADADELT</th>
<th>Ratio = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ac8</td>
<td>0.53</td>
<td>0.70</td>
<td></td>
</tr>
<tr>
<td>1stp</td>
<td>0.84</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>3ce3</td>
<td>1.12</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>3tmn</td>
<td>1.24</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>7cpa</td>
<td>1.48</td>
<td>0.86</td>
<td></td>
</tr>
</tbody>
</table>
## Max 1550 (SYCL) vs. A100 80GB (CUDA)

### Docking Time Ratios

<table>
<thead>
<tr>
<th>Input molecules</th>
<th>ADADELTASolis-Wets Ratio</th>
<th>ADADELTASolis-Wets Ratio</th>
<th>ADADELTASolis-Wets Ratio</th>
<th>ADADELTASolis-Wets Ratio</th>
<th>ADADELTASolis-Wets Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ac8</td>
<td>0.70</td>
<td>0.53</td>
<td>1.12</td>
<td>1.24</td>
<td>1.48</td>
</tr>
<tr>
<td>1stp</td>
<td>0.84</td>
<td>0.81</td>
<td>0.84</td>
<td>0.75</td>
<td>0.86</td>
</tr>
<tr>
<td>3ce3</td>
<td>1.12</td>
<td>1.12</td>
<td>1.12</td>
<td>1.12</td>
<td>1.12</td>
</tr>
<tr>
<td>3tmn</td>
<td>1.24</td>
<td>1.24</td>
<td>1.24</td>
<td>1.24</td>
<td>1.24</td>
</tr>
<tr>
<td>7cpa</td>
<td>1.48</td>
<td>1.48</td>
<td>1.48</td>
<td>1.48</td>
<td>1.48</td>
</tr>
</tbody>
</table>

**Ratios below 1:**
Max 1550 is faster than A100 80GB

**ADADELTAs Maximum Speed Up (Max 1550 wrt. A100 80GB):**

\[
\text{Maximum Speed Up} = \frac{1}{0.53} = 1.88x
\]
SYCL and CUDA versions are both compute bound

Expected speed up: based on theoretical FP32 performance capabilities

- Max 1550 / A100 80GB = 52 [TFLOP/s] / 19.5 [TFLOP/s] = ~2.6x

Maximum achieved speed ups are lower than expected one

- = ~1.4x (Solis-Wets), ~1.8x (ADADELTA)
- Possibly due to inefficient usage of compute resources …
- … in compute-intensive calls, i.e., score and gradient calculations
Conclusion

- We have manually modified the Compatibility-Tool-migrated SYCL code
  - For higher performance (atomics, barriers, native math functions)
- Preliminary results: still competitive on CPUs and GPUs
  - Optimization is ongoing!

**Maximum achieved performance ratios**

<table>
<thead>
<tr>
<th>Xeon 8360Y CPU</th>
<th>A100 80GB GPU</th>
<th>A100 80GB vs. Max 1550</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCL</td>
<td>SYCL</td>
<td>CUDA</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>3.6</td>
<td>1.8</td>
</tr>
</tbody>
</table>

- SYCL faster
- CUDA faster
- Max 1550 faster
Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study

https://github.com/emascarenhas/AutoDock-GPU

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

Leonardo Solis-Vasquez
solis@esa.tu-darmstadt.de