Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs

Christoph Weckert\textsuperscript{1}, Leonardo Solis-Vasquez\textsuperscript{1}, Julian Oppermann\textsuperscript{1}, Andreas Koch\textsuperscript{1}, Oliver Sinnen\textsuperscript{2}

1: \textsc{Technische Universität Darmstadt}
2: \textsc{University of Auckland}
Contents

Introduction to Altis

Migrating from CUDA to SYCL for GPUs

Porting and Optimizing for FPGAs
Altis-SYCL
Benchmark Suite for GPUs and FPGAs

Our contribution

Set of practical development guidelines for deploying CUDA applications on HPC systems supporting SYCL
The Altis Benchmark Suite
B. Hu and C. J. Rossbach [ISPASS 2020]

Collection of a wide-range of GPGPU applications written in CUDA
• Grouped into four categories

Aims to better represent modern GPGPU workloads
• By adopting and extending applications from Rodinia and SHOC

Increasing code complexity

Level 0
• Low Level Metrics

Level 1
• Basic Parallel Algorithms

Level 2
• Real World Applications

DNN Benchmarks
• Neural Network Layers in DNN models
Our Focus: Altis’ Level 2 Applications

- Computational Fluid Dynamics (CFD)
- Discrete Wavelet Transform (DWT2D)
- Finite Difference Time Domain (FDTD2D)
- Kmeans
- LavaMD
- Mandelbrot
- Needleman-Wunsch (NW)
- ParticleFilter (PF)
- Raytracing
- SRAD
- Where

Level 0
- Low Level Metrics

Level 1
- Basic Parallel Algorithms

Level 2
- Real World Applications

DNN Benchmarks
- Neural Network Layers in DNN models
Migration Methodology for Altis-SYCL

Migrating CUDA to SYCL

Optimizing for GPUs

Optimizing for FPGAs
Migration Methodology for Altis-SYCL

Migrating CUDA to SYCL

Optimizing for GPUs

Optimizing for FPGAs
Intel’s DPC++ Compatibility Tool (DPCT)
Usage Flow
Migrating CUDA to SYCL using DPCT

DPCT-inserted warnings domain distribution

- Error Handling: 56.2%
- Time Measurements: 15.0%
- USM: 9.0%
- Thread-Synchronization: 4.7%
- API-Calls: 3.0%
- Device Capabilities: 2.2%
- Templates: 1.8%
- Random Number Generators: 1.8%
- Group Synchronization: 1.4%
- Device Selection: 1.4%
- Control Flow: 0.9%
- Local memory: 0.7%

CUDA handles errors via function return, SYCL uses try-catch blocks instead

```c
/*
 DPCT1003:397: Migrated API does not return error code. (*, 0) is inserted. You may need to rewrite this code.
 */
 CUDA_SAFE_CALL((weights_GPU = sycl::malloc_device<double>(
   Nparticles, dpct::get_default_queue(0)),
```

DPCT inserted a total of 2535 diagnostic references when translating Altis Level 2
Migration Methodology for Altis-SYCL

Migrating CUDA to SYCL

Optimizing for GPUs

Optimizing for FPGAs
Optimizing for GPUs

Optimization on RTX 2080

- NVCC (CUDA) vs. Clang (SYCL)
- Power math function
**NVCC (CUDA) vs. Clang (SYCL)**

Loop Unrolling

Divergent behavior for same loop
- CUDA: might increase the performance
- SYCL: might have the *opposite* effect

[Computational Fluid Dynamics]
SYCL: we *disable* unrolling for the main loop → 3x faster (vs. unrolling)
NVCC (CUDA) vs. Clang (SYCL)

Different compilers →
Different behavior

Clang compiler (SYCL) acts more cautiously
• Even if kernel only calls a single function, it might not be automatically inlined

[Needleman-Wunsch]
• SYCL: we *increase* (#instructions) threshold: 2x faster (vs. default)
• -finlining-threshold = 10000

Function Inlining
**Power Math Function**

- DPCT replaced: $\text{pow}(a, 2) \rightarrow a \times a$
- SYCL: 6x faster than CUDA

We apply above transformation back to the original CUDA
- CUDA vs. SYCL: performance is on par
**SYCL vs. CUDA**

**Speedup on RTX 2080 GPU**

Predefined sizes in Altis:
Size 1 (small) → Size 2 → Size 3 (large)

**SYCL vs. CUDA**
**(After optimizations)**

**Speedup values > 1**
SYCL faster than CUDA

Most optimized SYCL and CUDA versions have comparable performance
Corner Case
[Raytracing]

DPCT introduced a different RNG
- cuRAND’s XORWOW
- oneMKL’s philox4x32x10

Significant code refactoring to cope with CUDA virtual functions

SYCL vs. CUDA Speedup (Optimized)

Size 1: 11.6x
Size 2: 18.6x
Size 3: 21.7x
Migration Methodology for Altis-SYCL

Migrating CUDA to SYCL

Optimizing for GPUs

Optimizing for FPGAs
Optimization Methodology for FPGAs

Apply general optimizations

Optimize the migrated ND-Range kernels

Performance acceptable?

No

Reimplement as Single-Task kernels

Yes
Optimization Methodology for FPGAs

1. Apply general optimizations
   - Datatype optimizations

2. Optimize the migrated ND-Range kernels

   - Performance acceptable?
     - Yes
     - No
       - No: Reimplement as Single-Task kernels

Besides compute unit replication
Datatype Optimizations
[Raytracing] <material> class: Original

FPGA compiler might infer *inefficient* global and local memory systems

E.g., C++ classes or structs featuring multiple member variables of different types

```cpp
class material {   /* Original */
public:
  enum type: uint8_t {metal, dielectric, lambertian};
  type m_type;
  vec3 m_albedo;      // lambertian and metal (lam)
  float m_fuzz;      // metal (met)
  float m_ref_idx;};  // dielectric (die)
```

Corresponding implementation:
- Performs only a *single* write access to such object
- But inferred hardware contains *three* store ports
Datatype Optimizations

[Raytracing] <material> class: Original $\rightarrow$ Optimized

class material {   /* Original */
    public:
        enum type: uint8_t {metal, dielectric, lambertian};
        type m_type;
        vec3 m_albedo;   // lambertian and metal (lam)
        float m_fuzz;    // metal (met)
        float m_ref_idx; // dielectric (die)
}

Arbiters: stallable memory
**Datatype Optimizations**

**[Raytracing] <material> class: Original → Optimized**

```cpp
class material { /* Original */
public:
    enum type: uint8_t {metal, dielectric, lambertian};
    type m_type;
    vec3 m_albedo;     // lambertian and metal (lam)
    float m_fuzz;      // metal (met)
    float m_ref_idx;   // dielectric (die)
};
```

```cpp
class material { /* Optimized */
public:
    // data[0]: "fuzz" parameter
    // data[1]: "ref_idx" parameter
    // data[2:4]: "albedo" parameter
    // data[5]: material "type": met (0), die (1), lam (2)
    // data[6:7]: unused
    sycl::float8 data;};
```

Fusing all class members into a single vector member

Arbiters: stallable memory

No arbiters: stall-free memory
Optimization Methodology for FPGAs

Apply general optimizations

Optimize the migrated ND-Range kernels

Performance acceptable?

Yes

No

Reimplement as Single-Task kernels

Besides vectorization

Apply general optimizations

Shared memory
**SYCL Accessors**

**Introduction**

- Standard method in SYCL for creating shared memory
- **DPCT**: inserts SYCL accessors
  - Dynamically sized
  - Cannot be statically defined at compile time
- Can cause issues when targeting FPGAs

```cpp
65 sycl::buffer<int> results_buff{ sycl::range(size)};
95 sycl::accessor results {results_buff, cgh, sycl::write_only, 
                         sycl::noinit};

100 cgh.parallel_for<mark_matches_cu>(
    sycl::nd_range<1>(grid_per_cu_k3[CU], block_dim),
    [=](sycl::nd_item<1> item) {
        [[intel::kernel_args_restrict, intel::num_simd_work_items(16),
        intel::no_global_work_offset(1),
        sycl::reqd_work_group_size(1, 1, g_thread_cnt),
        intel::max_work_group_size(1, 1, g_thread_cnt)]] {
            const int tid = item.get_global_id(0) + offset;

            for (int i = tid; i < size; i += g_thread_cnt * grid_range)
                results[tid] = (d_arr[tid] < coverage) ? 1 : 0;
        }
    });
```

[Where]
Shared Memory

<group_local_memory_for_overwrite> Class

- Allows the implementation of shared memories with user-defined sizes
  - Replaces default SYCL accessors

- Vendor and device specific
  - Only for Intel FPGAs, not supported on CPUs/GPUs
  - Available via oneAPI’s FPGA Toolkit

We apply this to all ND-Range kernels → reduces resource utilization

Can be used to allocate group-local memory at the kernel functor scope

```cpp
template <typename T, typename Group>
multi_ptr<T, access::address_space::local_space>
group_local_memory_for_overwrite(Group g);
```

```cpp
void kernel_gpu_cuda(par_str d_par_gpu, dim_str d_dim_gpu,
                     sycl::device_ptr<box_str> d_box_gpu,
                     sycl::device_ptr<four_vector> d_rv_gpu,
                     sycl::device_ptr<fp> d_qv_gpu,
                     sycl::device_ptr<four_vector> d_fv_gpu,
                     sycl::nd_item<1> item_ct1) {
  auto rA_shared_ptr =
      group_local_memory_for_overwrite<four_vector, number_par_per_box>(
          item_ct1.get_group());
}
```

[LavaMD]
Optimization Methodology for FPGAs

Apply general optimizations

Optimize the migrated ND-Range kernels

Performance acceptable?

No

Reimplement as Single-Task kernels

Yes

Besides loop pipelining & speculation

Custom prefix-sum for FPGAs

Pipes
Pipes

[Kmeans]: Optimization Process

Diagram showing the optimization process of the Kmeans algorithm, comparing the original and optimized versions.

Original:
- Global Memory (DDR)
- mapCenters
- reset
- accumulate
- finalize

Optimized:
- Global Memory (DDR)
- mapCenters
- resetAccFin
- pipe 1
- pipe 2
Pipes

[Kmeans]: Optimization Process

Dataflow to/from global memory is limited to `<mapCenters>` kernel only

Benefits of pipes
• Mapping of each data point is immediately passed between kernels

Performance improvement:
~510x
(pipes vs. no-pipes)
Custom Prefix-Sum for FPGAs

[Where]: prefix-sum

- CUDA: implementation from vendor library
- SYCL: DPCT incorporates oneDPL’s prefix sum
- SYCL: ~2x slower vs. CUDA on RTX 2080

oneDPL does *not* provide an FPGA-optimized implementation →

We develop a custom prefix-sum (Single-Task)

Performance improvement:
~100x

(Custom prefix-sum vs. oneDPL’s implementation)
FPGA Optimized vs. FPGA Baseline
Geometric Mean of Speedup on Stratix 10

\[
\text{Speedup} = \frac{\text{Runtime Baseline}}{\text{Runtime Optimized}}
\]

<table>
<thead>
<tr>
<th>Size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 1</td>
<td>10.7x</td>
</tr>
<tr>
<td>Size 2</td>
<td>20.7x</td>
</tr>
<tr>
<td>Size 3</td>
<td>35.6x</td>
</tr>
</tbody>
</table>
Stratix 10 → Agilex
FPGA Retargeting

Optimized code for Stratix 10 used as baseline for Agilex

We adjust some parameters for fitting or increasing performance

Resources: Stratix 10 > Agilex

Total Number of Hardware Resources

<table>
<thead>
<tr>
<th></th>
<th>Stratix 10</th>
<th>Agilex</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALM</td>
<td>933120</td>
<td>487200</td>
</tr>
<tr>
<td>BRAM</td>
<td>11721</td>
<td>7110</td>
</tr>
<tr>
<td>DSP</td>
<td>5760</td>
<td>4510</td>
</tr>
</tbody>
</table>

\[ \Delta = +47.7\% \] \quad \[ \Delta = +39.3\% \] \quad \[ \Delta = +21.7\% \]
Stratix 10 → Agilex
(Some) Parameter Adjustments

Increasing WG size

Reducing unrolling factor

Scaling up/down compute unit replication factor
## Evaluation Setup

<table>
<thead>
<tr>
<th>Device</th>
<th>Process [nm]</th>
<th># Compute Units</th>
<th>Peak FP32 [TFLOP/s]</th>
<th>Peak Mem. BW [GB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon Gold 6128</td>
<td>14</td>
<td>6 Cores</td>
<td>1.1</td>
<td>128.0</td>
</tr>
<tr>
<td><strong>GPUs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTX 2080</td>
<td>12</td>
<td>46 SMs</td>
<td>10.1</td>
<td>448.0</td>
</tr>
<tr>
<td>A100</td>
<td>7</td>
<td>108 SMs</td>
<td>19.5</td>
<td>1555.0</td>
</tr>
<tr>
<td>Max 1100</td>
<td>10</td>
<td>56 X$^e$-cores</td>
<td>22.2</td>
<td>1229.0</td>
</tr>
<tr>
<td><strong>FPGAs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stratix 10</td>
<td>14</td>
<td>4713 DSPs</td>
<td>2.4 – 4.2</td>
<td>76.8</td>
</tr>
<tr>
<td>Agilex</td>
<td>10</td>
<td>4510 DSPs</td>
<td>2.3 – 5.0</td>
<td>85.3</td>
</tr>
</tbody>
</table>
GPUs vs. FPGAs (wrt. CPU)
Geometric Mean of Relative Speedup

\[ \text{Speedup } X = \frac{\text{Runtime CPU}}{\text{Runtime } X} \]
GPUs vs. FPGAs (wrt. CPU)
Relative Speedup

\[ \text{Speedup} = \frac{\text{Runtime CPU}}{\text{Runtime X}} \]

Comparable performance between FPGAs and GPUs
GPUs vs. FPGAs (wrt. CPU) Relative Speedup

\[
\text{Speedup} = \frac{\text{Runtime CPU}}{\text{Runtime } X}
\]

Comparing performance between FPGAs and GPUs
Conclusions

Altis-SYCL: a modern and portable C++-based benchmark suite for GPUs and FPGAs
• Retains the advantages of Altis

Performance results
• Small input sizes: GPUs and FPGAs reach comparable performance for some applications
• Large input sizes: GPUs outperform FPGAs
  • Size 3: A100 vs. Agilex → geo. mean speedup: ~15x
Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs

https://github.com/esa-tu-darmstadt/altis_sycl

Acknowledgements

Paderborn Center for Parallel Computing
Stratix 10 FPGA

Intel oneAPI Center of Excellence Research Award
Max 1100 GPU

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