IWOCL 2024
The 12th International Workshop on OpenCL and SYCL

SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs

Luigi Crisci, University of Salerno
Lorenzo Carpentieri, Biagio Cosenza, University of Salerno
Peter Thoman, University of Innsbruck
Axel Alpay, Vincent Heuveline, University of Heidelberg

APRIL 8-11, 2024 | CHICAGO, USA | IWOCL.ORG
From SYCL 1.2.1 to SYCL 2020

- SYCL 1.2.1: high-level programming model on top of OpenCL
- Latest specification SYCL 2020 allow for third-party backends
  - NVIDIA CUDA, AMD ROCm, Intel LevelZero, OpenMP, TBB, etc.
- Several new features
  - Unified Shared Memory (USM)
  - Built-in parallel reduction support
  - Support for native API interoperability
  - Work group and subgroup common algorithm libraries
- Third-party backends + multiple compilers complicates validation
SYCL-Bench 2020

- Extend SYCL-Bench [1] with SYCL 2020-specific benchmark
  - Original work designed for SYCL 1.2.1
- Characterize SYCL 2020 features on HPC GPU hardware
- Evaluation of AdaptiveCpp and DPC++ implementations on data-center level GPUs

- 9 new benchmarks
- 44 different configurations
- Feature covered:
  - Unified Shared Memory
  - Kernel Reductions
  - Specialization constants
  - Group algorithms
  - In-order queue
  - Atomics

Experimental setup

- SYCL Implementations:
  - AdaptiveCpp (git eebfd4)
  - Intel DPC++ (git f43cd7b)

- Three vendor GPUs:
  - NVIDIA Tesla V100S (CUDA 12.1, driver 535.129.03)
  - AMD MI100 (ROCm 5.5.0, driver 505.302.01)
  - Intel Max 1100 (LevelZero driver 170.007.42)
Pattern 1: USM - Host-Device transfers

- Simulate different offloading scenarios

- **Benchmark:**
  - 2GB data size
  - *Instruction mix (IM):* host/device FLOP ratio
    - 1 to 6 IM
  - *Outer Loop (OL):* repeat the device and host kernels

- **Rationale:** Measure USM migration policies
Pattern 1: USM - Host-Device transfers

2GB, OL 1 iteration

Prefetch speedup over non-prefetched shared allocation
Pattern 1: USM - Host-Device transfers

- 4+ instruction mix to match host alloc
- Low performance from shared on AMD
- Shared alloc comparable to device alloc
- Prefetch speedup over non-prefetched shared allocation
- ~1.27x speedup with sycl::prefetch
Pattern 2: Reduction kernels & Group Reductions

- Two kind of SYCL reductions:
  - **Kernel reductions** (KR): Kernel level, cross-group
  - **Group reductions** (GR): WG or SG level
- Need to work for any SYCL supported type
  - KR cannot be trivially implemented in some cases
- **Benchmark:**
  - 150,000,000 elements
  - 4 types (int32, int64, fp32, fp64)
  - **Coarsening factor** (CF): element computed by each thread
  - Compared against *local memory reduction* w/ atomic (LM)
- **Rationale:** measure SYCL implementations reduction’s quality

```cpp
int sum = 0;
q.submit([&](handler& h) {
    auto r = reduction(&sum, h, sum<int>());
    accessor in(buf, h, access::read_only);
    h.parallel_for(range, [=](item<1> i, auto& op) {
        op.combine(in[i]);
    });
});
```

```cpp
q.submit([&](handler& h) {
    accessor in(buf, h, access::read_only);
    accessor out(out_v, h, access::write_only);
    h.parallel_for(nd_range, [=](nd_item<1> i){
        auto& group = i.get_group();
        out_v[i] = group_reduce(group, in[i], plus<int>());
    });
});
```

// Tree reduction to combine elements
Pattern 2: Reduction kernels & Group Reductions

![Benchmark Graphs]

- **Benchmark**
  - Kernel reduction CF1
  - Kernel reduction CF4
  - Reduce over group
  - Reduction local memory

- **SYCL impl.**
  - AdaptiveCpp-SMCP
  - Intel DPC++
Pattern 2: Reduction kernels & Group Reductions

- DPC++ applies coarsening automatically with `sycl::range` `parallel_for`
- Not possible with `nd_range`

DPC++ not influenced by coarsening

Low performance on Intel due to atomic

AdaptiveCPP requires manual coarsening
Pattern 2: Reduction kernels & Group Reductions

- DPC++ applies coarsening automatically with `sycl::range parallel_for`
- Not possible with `nd_range`

---

**DPC++ not influenced by coarsening**

AdaptiveCPP requires manual coarsening

**2x KR speedup w/ DPC++ and AdaptiveCpp!**

**DPC++ GR on par with LM**
Pattern 3: In order queues

- Command executed in FIFO order
- Optimization opportunities:
  - No dependency tracking needed (single queue)
  - SYCL Task graph could be omitted
- **Benchmark:** Measure USM vs Buffer kernel scheduling time
  - Schedule 3 USM or Accessor buffer
  - 50,000 addition kernels
- **Rationale:** check if implementations exploits optimizaitons to improve scheduling latency
Pattern 3: In order queues

**Overhead on DPC++ ROCm backend**

4x speedup!

**No effect on Accessors**
Pattern 4: Specialization constants

- Inject runtime values as constant in device kernel
- Kernel is JIT-compiled and optimized
- Requires recompilation for each specialization constant value change
- Implementation is backend-specific
- **Benchmark:**
  - Stencil code with *dynamic, constexpr*, and *specialization constant* parameters
    - *Inner Loop (IL)* param to increase computation
- **Rationale:** Measure the impact of const evaluation opt and JIT overhead

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;
static constexpr s::specialization_id<int> C;
int main(int n, char **) {
    [...]
    q.submit([&](handler &h) {
        h.set_specialization_constant<C>(runtime_value());
        accessor x(x_buf, h, access::read_only);
        h.parallel_for(num_items, [=](item i) {
            int val = h.get_specialization_constant<C>();
            x[i] = val * 0.5f;
        });
    });
}
```
Pattern 4: Specialization constants
Pattern 4: Specialization constants

SC perf comparable to constexpr!
Pattern 4: Specialization constants

SC perf comparable to constexpr!

Nothing happen on NVIDIA and AMD hardware
Pattern 4: Specialization constants

SC perf comparable to constexpr!

Up to 600x slowdown with 2ms kernel

Nothing happen on NVIDIA and AMD hardware
To summarize

- First benchmark suite for SYCL 2020
  - 9 new benchmark
  - 44 configurations
- The right USM allocation depends on the scenario
- In-order queue reduces scheduling time with USM
  - No effect with Accessors
- Specialization constant do not currently work on NVIDIA and AMD
- Compiler maturity is steadily improving
SYCL-Bench 2020: Benchmarking SYCL 2020 on AMD, Intel, and NVIDIA GPUs

https://github.com/unisa-hpc/sycl-bench/tree/sycl2020

Luigi Crisci
lcrisci@unisa.it

This project has received funding from the European Union’s HE research and innovation programme under grant agreement No. 101092877 (SYCLops) and from the European High-Performance Computing Joint Undertaking under grant agreement No. 956137 (LIGATE project). Additionally, it has received funding from the Austrian Research Promotion Agency (FFG) via the UMUGUC project (FFG \#4814683) and from the Italian Ministry of University and Research under PRIN 2022 grant No. 2022CC57PY (LibreRT project).
Backup slides
What’s SYCL?

- C++ royalty-free, cross-platform abstraction layer for heterogeneous computing
- Single-source, modern C++17 APIs
- Targets CPUs, GPUs, FPGAs, TPUs, etc. from multiple vendors
- Extension for Safety Critical environments (SYCL SC)

Credit: Kronos Group

---

**SYCL**

- C++ Libraries
- Standard C++ Application Code
- ML Frameworks

**SYCL Compiler**

- CPU Compiler
- Other Backends

**SYCL** is ideal for accelerating C++ based engines and applications with performance portability

Credit: Kronos Group
SYCL implementations

Major implementations

Additional implementations & extensions

Credits: Kronos Group
Unified Shared Memory

- Pointer-based, low-level memory API for handling memory allocations
- Lighter interface than sycl::buffer
- Common address space for both host and device
- Three types of allocation:
Unified Shared Memory

- Pointer-based, low-level memory API for handling memory allocations
- Lighter interface than sycl::buffer
- Common address space for both host and device
- Three types of allocation:
  - Host allocation

Device

Host

CPU

Host memory

Device memory

Unified address space

Accessible:
Allocated:

GPU

FPGA

TPU
Unified Shared Memory

- Pointer-based, low-level memory API for handling memory allocations
- Lighter interface than sycl::buffer
- Common address space for both host and device
- Three types of allocation:
  - Host allocation
  - Device allocation
Unified Shared Memory

- Pointer-based, low-level memory API for handling memory allocations
- Lighter interface than sycl::buffer
- Common address space for both host and device
- Three types of allocation:
  - Host allocation
  - Device allocation
  - Shared allocation
Unified Shared Memory

- Pointer-based, low-level memory API for handling memory allocations
- Lighter interface than sycl::buffer
- Common address space for both host and device
- Three types of allocation:
  - Host allocation
  - Device allocation
  - Shared allocation
- Each allocation suitable for different scenarios
USM benchmark results (1)

- **H-D: 3 copies to match Pinned alloc overhead**
- **D-H: no benefit from non-pinned memory**
USM: Benchmarks

1) **Task scheduling latency:**
   - Measure USM vs Buffer kernel scheduling time
     - Schedule 3 USM or Accessor buffer
     - 50,000 addition kernels

2) **Host-Device transfers:**
   - Measure USM migration policy
   - Simulate different offloading scenarios
     - *Instruction mix:* host/device FLOP ratio

3) **Pinned vs non-pinned memory:**
   - Measure host-device/device-host copy time when using pinned/non-pinned allocations
   - Host/device – device/host copies looped
Overhead on DPC++ ROCm backend

3.1x speedup!
Specialization constants

- Inject runtime values as constant in device kernel
- Kernel is JIT-compiled and optimized
- Requires recompilation for each specialization constant value change
- Implementation is backend-specific

**Benchmark:**
- Stencil code with `dynamic`, `constexpr`, and `specialization constant` parameters
  - *Inner Loop (IL)* param to increase computation
- **Rationale:** Measure the impact of const evaluation opt and JIT overhead

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl;
static constexpr s::specialization_id<int> C;

int main(int, char**) {
    constexpr size_t size = 10000;
    queue q{gpu_selector_v};
    std::vector<float> x_vec(size, 1.0f);
    buffer x_buf(x_vec.data());
    range<1> num_items{x_vec.size()};
    q.submit([&](handler& h) {
        h.set_specialization_constant<C>(runtime_value());
        accessor x(x_buf, h, access::read_only);
        h.parallel_for(num_items, [=](item<1> i) {
            int val = h.get_specialization_constant<C>();
            x[i] = val * 0.5f;
        });
    });
    // ... print results and returns
}
```
The graph shows the execution time relative to the median for different types and implementations on AMD MI100 and Intel Max 1100 platforms. The x-axis represents the number of runs, and the y-axis represents the execution time relative to the median.

On AMD MI100, the graph indicates that the execution time decreases with increasing runs, with slight variations depending on the type and implementation.

On Intel Max 1100, the execution time remains relatively constant across runs, with a slight increase in execution time for some implementations.

Notable observations:
- **Weird CAS loop behavior** highlighted on the graph, indicating unexpected performance behavior.
- **120x overhead on first run on int64** highlighted, suggesting a significant performance gap on the first run for int64 type.