SYCL Concurrency on GPU Platforms: Empirical Measurement

Thomas Applencourt [1] Abhishek Bagusetty [1] Ajay Panyala [2] Aksel Alpay [3]

[1]Argonne National Laboratory, [2]Pacific Northwest National Laboratory, [3]Heidelberg University

- Commands which can be scheduled in parallel should/may run faster than the same commands serialized

  *A **command** is a request to execute work that is submitted to a queue such as the invocation of a SYCL kernel function, the invocation of a host task or an asynchronous copy.*
  (SYCL 2020 Specification, Glossary)

- Overlaps is and optimization and is **not** required by the specification.

Which Commands can run Concurrently?

1. Multiple Low occupancy kernels
   - In this paper we use a traditional "clpeak" kernel (chain of FMA on `float` (C)) scheduled to use **one work-item**
2. Host to Device data-transfer + Device to Host data-transfer
   - In this paper data-transfer occur between host memory (either allocated via `malloc` (M), or pinned memory`sycl::malloc_host` (H)) and Device Memory (via `sycl::malloc_device` (D)).
     e.g.: H2D, D2M, ...
3. Low occupancy kernel + Host to/from Device data-transfer

This poster aims to report **on with conditions different SYCL runtimes achieve concurrency**. *Deep explanations of the results are out of the scope of this poster*. But we will be more than happy to discuss low-level details!

How to Achieve Concurrency?

The SYCL specification allow concurrent execution in two main scenarios:

- **Out Of Order Queue** Multiple SYCL commands submitted in a out of order queue can be executed concurrently ("OpenCL way")

```
sycl::queue Q{sycl::gpu_selector()};
for (auto& command: commands)
  do_work(Q,command);
Q.wait();
```

- **Multiple In Order Queues**. Multiple SYCL commands can be submitted in a multiple in-order queues. Each queue can be executed concurrently ("CUDA way")

```
const sycl::device D{sycl::gpu_selector()};
const sycl::context C(D);
std::vector<sycl::queue> Qs;
for (auto& : commands)
  Qs.push_back(sycl::queue(C, D,sycl::property::queue::in_order{}));
for (int i = 0; i < commands.size(); i++)
  do_work(Qs[i], commands[i]);
for (auto &Q : Qs)
 Q.wait();
```

Compiler & Drivers Versions Used

ID	Compiler Name	Commit
i	DPCPP	2022.1.0
ii	Intel/LLVM	03ff41f
iii	Intel/LLVM	1fe5eaa
iv	hipSYCL	258dc87

(a) Compiler Versions

ID	Drivers Name	Version
I	Compute Runtime	21.40.21182
II	CUDA	11.0.2
III	ROCM	4.5.2

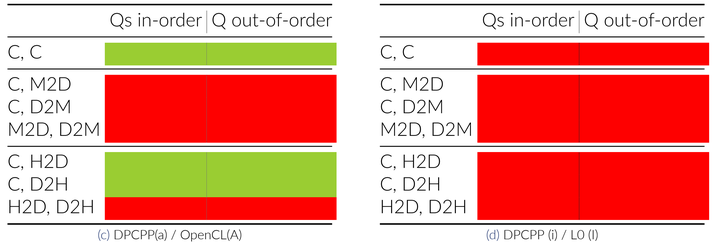
(b) Drivers Verions

Result Color Code

Concurrent Execution
Concurrent Execution but Serial Execution when using Profiling enable Queues
Serial Execution

Table 1. Color Scheme Legend

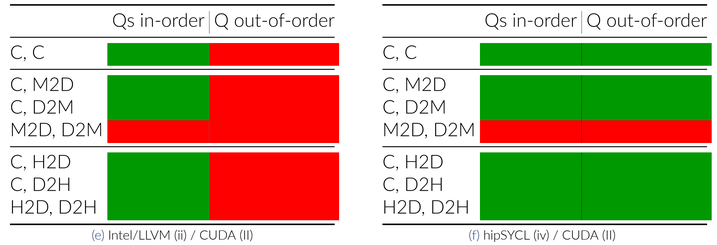
Intel GPU Platform (Iris Pro Graphics 580)



(c) DPCPP(a) / OpenCL(A)

(d) DPCPP (i) / L0 (I)

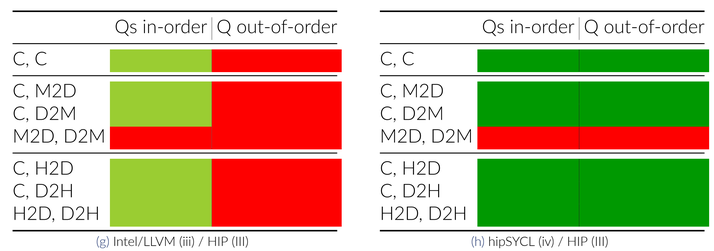
NVDIA Platform (A100)



(e) Intel/LLVM (ii) / CUDA (II)

(f) hipSYCL (iv) / CUDA (II)

AMD Platform (MI100)



(g) Intel/LLVM (iii) / HIP (III)

(h) hipSYCL (iv) / HIP (III)

Methodology

We run each step N times, and take the minimum time to reduce measurement noise

1. Run commands lists serially. Compute the maximum Theoretical Speedup $max\_command\_time/total\_time\_serial$
   - Commands parameters (number of FMA, size of the data-transfer) are auto-tuned so that each command take roughly the same time
   - By default memory transfers are "as big as possible" (≈ `D.get_info<sycl::info::device::max_mem_alloc_size>()`)
2. Run the list of commands in a mode who allow concurrency. Compute Empirical Speedup $total\_time\_concurrent/total\_time\_serial$
3. Verify that Theoretical speedup and Empirical speedup roughly match

Tool Output Example

```
$./sycl_con out_of_order C D2H
Performing Autotuning
Parameters tunned:
  tripcount_C: 466765
  globalsize_D2H: 1073739776
Best Total Time Serial: 610271us
  Best Time Command 0 (  C): 303298us
  Best Time Command 1 (D2H): 306973us
Maximum Theoretical Speedup: 1.98803x
Best Total Time //: 355451us
Speedup Relative to Serial: 1.71689x
SUCCESS: Close from Theoretical Speedup
```

Summary

1. We developed a empirical concurrency testing framework. People are encouraged to use it! *See url in the bottom right of the poster*
2. Using "pinned memory", `sycl::malloc_host`, may be required by drivers for concurrency
3. On Intel Platform, L0 need a little more love
   - Future version Intel/LLVM L0 backend is planned to use immediate command list. Should allow concurrency is many case.
4. On NVDIA and AMD Platform, hipSYCL delivers most reliable overlap of operations. Intel/LLVM has a hard time with out-of-order queue
   - Discussion on-going for Intel/LLVM CUDA,HIP back-end to implement an M:N mapping between streams and `sycl::queues`. This Should help out-of-order queues performance.
5. Using profiling queue, via `sycl::property::queue::enable_profiling`, can impact concurrency