Evaluating the Performance of Integer Sum Reduction in SYCL on GPUs

ZHEMING JIN

ACKNOWLEDGMENT

* Experimental Computing Laboratory at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
* Intel DevCloud
Overview

• Motivation
• Background
• Implementation
• Experiment
• Conclusion
Motivation

• SYCL is a promising programming model
## Background - SYCL

<table>
<thead>
<tr>
<th>Step</th>
<th>OpenCL</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Platform query</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Device query of a platform</td>
<td>Device selector class</td>
</tr>
<tr>
<td>3</td>
<td>Create context for devices</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Create command queue for context</td>
<td>Queue class</td>
</tr>
<tr>
<td>5</td>
<td>Create memory objects</td>
<td>Buffer class</td>
</tr>
<tr>
<td>6</td>
<td>Create program object</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Build a program</td>
<td>Lambda expressions</td>
</tr>
<tr>
<td>8</td>
<td>Create kernel(s)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Set kernel arguments</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Enqueue a kernel object for execution</td>
<td>Submit a SYCL kernel to a queue</td>
</tr>
<tr>
<td>11</td>
<td>Transfer data from device to host</td>
<td>Implicit via accessors</td>
</tr>
<tr>
<td>12</td>
<td>Event handling</td>
<td>Event class</td>
</tr>
<tr>
<td>13</td>
<td>Release resources</td>
<td>Implicit via destructor</td>
</tr>
</tbody>
</table>
Background – SYCL with CUDA support

Figure 1: CUDA target processing in the SYCL compiler [22]
Background – Integer sum reduction

```c
int numbers[M];
int sum = 0;
for ( int i = 0; i < M; i++ )
    sum += numbers[i];
```
SYCL Implementation 1 – Integer sum reduction

```cpp
1  cgh.parallel_for<class reduce>(
2     nd_range<1>(gws, lws), [=](nd_item<1> item) {
3     int gid = item.get_global_id(0);
4     int lid = item.get_local_id(0);
5     int WGS = item.get_local_size(0);
6     if (lid == 0) sum[0].store(0);
7     item.barrier(access::fence_space::local_space);
8     atomic_fetch_add(sum[0], input[gid]);
9     item.barrier(access::fence_space::local_space);
10    if (lid == WGS-1) {
11        int partial_sum = atomic_load(sum[0]);
12        atomic_fetch_add(out[0], partial_sum);
13    }
14});
```
SYCL Implementation 2 – Integer sum reduction

```cpp
1  cgh.parallel_for<class reduce>(
2     nd_range<1>(gws, lws), [=](nd_item<1> item) {
3       vec<int, N> vi;
4       int gid = item.get_global_id(0);
5       int lid = item.get_local_id(0);
6       int WGS = item.get_local_size(0);
7       vi.load(gid, input.get_pointer());
8       int r = vi.s0() + vi.s1() + ... + vi.sN-1();
9       if (lid == 0) sum[0].store(0);
10      item.barrier(access::fence_space::local_space);
11     atomic_fetch_add(sum[0], r);
12      item.barrier(access::fence_space::local_space);
13     if (lid == WGS-1) {
14       int partial_sum = atomic_load(sum[0]);
15       atomic_fetch_add(out[0], partial_sum);
16     }
17   });
```
SYCL Implementation 3 – Integer sum reduction

```cpp
1 cgh.parallel_for<class reduce>(
2   nd_range<1>(gws, lws), [=](nd_item<1> item) {
3     int gid = item.get_global_id(0);
4     int lid = item.get_local_id(0);
5     int blk = item.get_group(0);
6     int WGS = item.get_local_size(0);
7     if (lid == 0) sum[0].store(0);
8     item.barrier(access::fence_space::local_space);
9     int start = blk * WGS * L + lid;
10    int end = (blk+1) * WGS * L;
11    int r = 0;
12    for (int i = start; i < end; i = i + WGS)
13        r += input[i];
14    atomic_fetch_add(sum[0], r);
15    item.barrier(access::fence_space::local_space);
16    if (lid == WGS-1) {
17        int partial_sum = atomic_load(sum[0]);
18        atomic_fetch_add(out[0], partial_sum);
19    }
20});
```
Experimental Setup

- Intel P630, Nvidia P100 and V100 GPUs
- Number of integers is 1048576000, approximately 4 GB in memory size
- Timing (milliseconds) measured with the Intel OpenCL intercept layer and Nvidia performance profiler
- The average execution time of 100 invocations of a kernel for the reduction performance
Experimental Results

Minimum time: 110.9 ms (P630)

Minimum time: 9.197 ms (P100)

Minimum time: 4.671 ms (V100)
Comparison with Others

<table>
<thead>
<tr>
<th>Kernel time (ms)</th>
<th>P100</th>
<th>V100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reducer [1]</td>
<td>40.4</td>
<td>11.62</td>
</tr>
</tbody>
</table>

• The fastest SYCL implementations are approximately 3% and 0.3% faster than the templated reduction in Thrust and the device reduction in CUB, respectively.

• The fastest SYCL implementations are approximately 1.9% and 0.4% faster than the templated reduction in Thrust and the device reduction in CUB, respectively.

Conclusion and Future Work

• Tuning workgroup sizes, vector widths, and workload sizes are important for performance improvement

• Thrust and CUB are mature libraries for parallel reduction on an Nvidia GPU

• There is a large optimization space for the SYCL reducer class

• Investigate the performance of SYCL applications that contain reduction kernels
Thanks to SYCL developers