A Study of Integer Sum Reduction using SYCL

Zheming Jin Hal Finkel

Email: zjin@anl.gov

Argonne NATIONAL LABORATORY

Introduction

The SYCL standard is a cross-platform abstraction layer for the programming of heterogeneous computing system using standard C++.

In Open Computing Language (OpenCL) programming model, host and device code are written in *different* languages.

The SYCL programming model can combine host and device (e.g., graphics processing unit) code for an application in a type-safe way to improve *development productivity.*

Sum reduction is a fundamental data parallel primitive. In this study, we focus on the implementations of integer sum reduction using the SYCL programming model, and evaluate their performance on an Intel processor with a central processing unit (CPU) and an integrated GPU. We choose an Intel processor for our study because the mainstream SYCL compilers, which provide a conformant implementation of the SYCL 1.2.1 Khronos specification [1], support Intel devices.

SYCL

- Most of the abstraction features of C++, such as templates, classes, and operator overloading are available for a kernel function in SYCL.

.

- Some C++ language features, such as virtual functions, virtual inheritance, throwing/catching exceptions, function pointers, and runtime type-information, are not allowed inside kernels due to the capabilities of the underlying OpenCL standard.

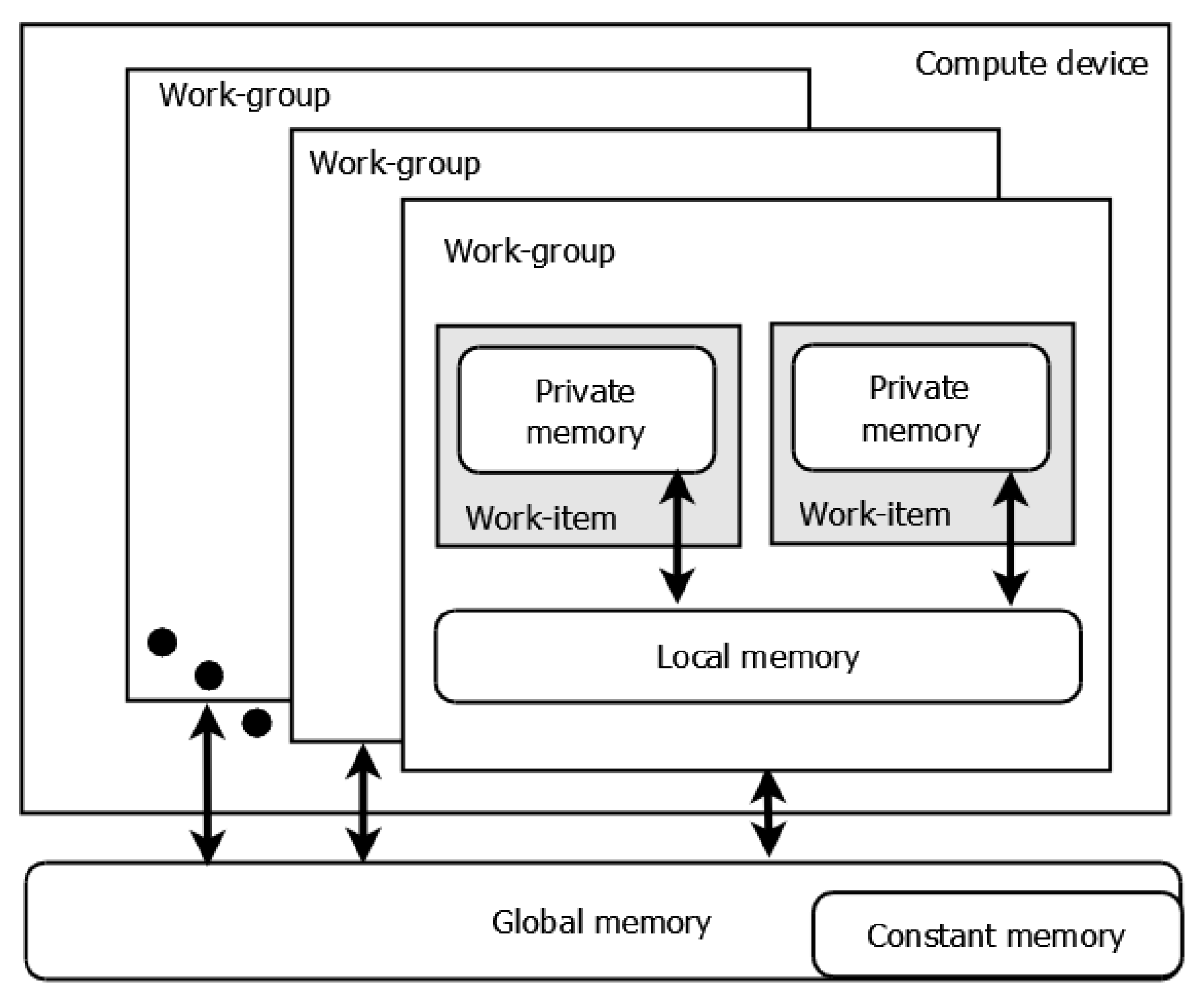
- These features, however, are available outside the kernel scope.

- A SYCL application is logically structured into three scopes:
  - Kernel scope: a single kernel function that will be executed on a device after compilation.
  - Command-group scope: a unit of work that will comprise of a kernel function and buffer accessors.
  - Application scope: all other code outside of a command-group scope.

- A SYCL kernel function may be defined by the body of a lambda function, by a function object or by the binary generated from an OpenCL kernel string.

Memory Model

- Global memory
  - Shared between all processing elements on the device and CPU
- Local memory
  - Shared by all work-items in a work-group
- Private memory
  - Unique to each work-item



Sum Reduction

Sequential version described in C

```
int sum = 0;
for ( int i = 0; i < M; i++ )
   sum += input[i];
```

Operations can be done in any order due to the associativity and commutativity of the integer add.
Take advantage of associativity to divide the reduction into independent partial sums, and then combine results from the partial sums.
The idea can be generalized to reductions of arbitrary size.

Parallel implementations described in SYCL

1. Reduction over global memory

```
cgh.parallel_for<class reduce>(
  numOfItems,[=](id<1> wiID) {
  atomic_fetch_add(accessorC[0], accessorA[wiID]);
});
```

- There are a total of “numOfItems” work-items to sum up the input
- “id<dimensions>” is a vector of dimensions that is used to represent an index into a global range.
- An accessor provides access to the data managed by a buffer or to shared local memory allocated by the runtime.
- Call the SYCL API function “cl::sycl::atomic_fetch_add()” which is equivalent to the function “std::atomic<int>::fetch_add()” in C++.

2. Hierarchical reduction using local memory

```
cgh.parallel_for<class reduce>(
  nd_range<1>(numOfItems, range<1>(WGS)),
  [=](nd_item<1> item) {
  size_t gid = item.get_global_linear_id();
  ushort lid = item.get_local_linear_id();
  if (lid == 0) {  // for the first work-item
    sum[0].store(0);
  }
  item.barrier(access::fence_space::local_space);
  atomic_fetch_add(sum[0], accessorA[gid]);
  item.barrier(access::fence_space::local_space);
  if (lid == WGS-1) {
    int partial_sum = atomic_load(sum[0]);
    atomic_fetch_add(accessorC[0], partial_sum);
  }
});
```

- The first work-item in each work-group resets the local sum.
- A barrier synchronizes the work-items to use the initialized local sum.
- After the atomic addition is performed by all work-items in a work-group, the last work-item in the group atomically adds the partial sum to the output.
- The total number of work-items equal the length of the input vector.

3. Hierarchical reduction with vectorized memory accesses

```
cgh.parallel_for<class reduce>(
  nd_range<1>(numOfWorkItems, range<1>(WGS)),
  [=](cl::sycl::nd_item<1> item) {
  vec<int,n> vi;
  size_t gid = item.get_global_linear_id();
  ushort lid = item.get_local_linear_id();
  vi.load(gid, accessorA.get_pointer());
  int r = vi.s0() + vi.s1() + … + vi.sn-1;
  if (lid == 0) sum[0].store(0);
  item.barrier(access::fence_space::local_space);
  atomic_fetch_add(sum[0], r);
    …
});
```

- The vectorized memory load reduces the global work size by a factor of “n”.

Experimental Setup

Hardware

CPU: Intel Xeon E3-1284L v4 (2.90 GHz, 8 compute units)
GPU: Intel Iris Pro P6300 Gen8 (1.15 GHz, 48 compute units)

Software

Intel SYCL compiler (clang 9.0.0) from the open-source repository [2], and
Codeplay’s SYCL compiler (ComputeCpp community edition, version 1.1.4) on Ubuntu 18.04

Input array length

The input array has 32 million elements (128 MB), and it is initialized with random data. We report the average execution time of 16 runs..

Experimental Results

Implementation	**CPU Time (ms)**		**GPU Time (ms)**	
	Intel wgs=512	Codeplay wgs=512	Intel wgs=256	Codeplay wgs=256
reduce_gmem	845	791	77	816
reduce_lmem	320	206	74	103
reduce_lmem_v2	221	207	72	99
reduce_lmem_v4	183	198	75	97
reduce_lmem_v8	174	199	89	99
reduce_lmem_v16	317*	210	145	99
reduce_baseline [3]	400*	208*	70*	99

Notes: gmem: global memory, lmem: local memory, *vn*: *n*-lane vectorization,
wgs: work-group size
* Assuming the reduction results are correct

- Intel SYCL compiler can optimize the reduction over global memory on the GPU.
- Reduction over global memory should be avoided. Using local memory to reduce the overhead of global memory accesses.
- The vectorized memory accesses are most effective on the CPU when the kernel functions are compiled with the Intel compiler.

Conclusion

- Reduction is an important data parallel primitive for acceleration.
- Demonstrate the SYCL implementations of the reduction using local memory, atomics, and vectorized memory accesses.
- Evaluate the performance of the integer sum reduction on an Intel CPU and GPU using Intel and Codeplay SYCL compilers.
- The compilers are maturing and we are interested in evaluating more SYCL programs.

Reference

- [1] https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
- [2] https://github.com/intel/llvm
- [3] https://github.com/codeplaysoftware/computecpp-sdk

U.S. DEPARTMENT OF ENERGY

2019 C++ Conference

Acknowledgement
I sincerely appreciate the reviewers’ comments. Results presented were obtained using the Chameleon testbed supported by the National Science Foundation. This research was supported under Contract DE-AC02-06CH11357.