Benchmarking and Extending SYCL
Hierarchical Parallelism

Tom Deakin, Simon McIntosh-Smith (Uni. of Bristol, UK)
Aksel Alpay, Vincent Heuveline (Uni. of Heidelberg, Germany)
Presenters

Tom Deakin
University of Bristol
tom.deakin@bristol.ac.uk

Aksel Alpay
University of Heidelberg
aksel.alpay@uni-heidelberg.de
Expressing Parallelism in SYCL

• Data parallel loop:
  • All iterations independent, no barriers
  • `parallel_for(range<1>{1024}, [=](id<1> it) {...});`

• NDRange:
  • 2 (or 3) level hierarchy: work-items collected into work-groups (and sub-groups)
  • Work-group barriers allowed “anywhere”
  • `parallel_for(nd_range<1>{{1024}, {16}}, [=](nd_item<1> it) {...});`

• Hierarchical parallelism:
  • 2 level hierarchy: work-items in work-groups
  • Only implicit barriers at end of `parallel_for_work_item`
  • `parallel_for_work_group(range<1>{64}, [=](group<1> g) {
      g.parallel_for_work_item(range<1>{16}, [=](h_item<1> it) {...}); });`
Memory spaces

Q.submit([&](handler &cgh) {

local_accessor<T> locA { N, cgh }; // runtime size local memory

cgh.parallel_for_work_group( {64}, [=](group<1> g) {

int A[16]; // compile time size local memory
int i; // scalar in local memory
private_memory<int> j {g}; // scalar in private memory

g.parallel_for_work_item( {16}, [=](h_item<1> it) {

int k = it.get_global_id(0); // private memory
j(it) = k; // access private_memory
A[it.get_local_id(0)] = k; // access local memory
locA[it.get_local_id()] = k;
}); }); }); }};
Implementing SYCL

- Hierarchical Parallelism in SYCL supposed to be convenient syntax
  - Barrier locations restricted so different forward progress guarantees
  - On CPUs, thread per work-group, auto-vectorise work-items
    - Straightforward mapping for a library implementation
  - On GPUs, need extra support for serial execution in work-group scope and additional local memory barriers

- NDRange must allow work-items reach barriers
  - Execution of work-items must “yield” control to allow other work-items to reach barrier
  - On CPUs, work-items need C++ fibers, threads, etc, work-groups are threaded
  - On GPUs: maps naturally to underlying models (CUDA, HIP, OpenCL, …)
Scoped parallelism

• Extension in hipSYCL for nested parallelism

• Idea: User requests a number of logical work items which are then distributed across implementation-defined physical work items
  • Mapping is done using `distribute_items(group, [&](s_item<1> idx){...});`
  • Some implementation choices for logical parallelism:
    • sequential loop iterations, or actual physical work items (e.g. CUDA threads), or a mixture of both
  • Can be efficiently implemented on CPU as well as on accelerators like GPUs
    • No complex compiler transformations required (unlike hierarchical parallelism on GPUs)

• More control for users for less performance suprises:
  • Explicit barriers and collective group algorithms supported
  • user has control over private/local memory placement
Scoped parallelism II

• `distribute_groups(group, [&](auto subgroup){...})` can be used to subdivide a group into smaller subunits
  • Can be nested arbitrarily deep
  • Implementation may fall back to trivial scalar groups if no further subdivision is supported
  • Can e.g. be used for multi-level tiling

```cpp
distribute_groups(group, [&](auto subgroup){
    distribute_groups(subgroup, [&](auto subsubgroup){
        distribute_items(subsubgroup, [&](s_item<1> idx){...});
    });
    group_barrier(subgroup);
});
```
Scoped parallelism example

Q.submit([&](handler& cgh) {
    local_accessor<T> locA { N, cgh }; // runtime size local memory
    cgh.parallel( {64}, {16}, [=](auto g) {
        int i; // scalar in private memory of physical work item
        sycl::memory_environment(
            sycl::require_local_mem<int[16]>().
            require_private_mem<
                int()
            ()
        ),
        [&](auto& A, auto& j) {
            sycl::distribute_groups(g, [&] (auto subgroup) { // optional
                sycl::distribute_items(subgroup, [&] (sycl::s_item<1> idx) {
                    int k = idx.get_global_id(0); // private mem. of logical item
                    j(idx) = k; // access private_memory
                    A[idx.get_innermost_local_id(0)] = k; // access local memory
                    locA[idx.get_innermost_local_id(0)] = k;
                });
            });
        });
    });
});
Benchmarks

• DGEMM: simple (range, no tiling), ndrange, hierarchical and scoped
• SYCL-Bench:
  • Nbody, reduction and segmented reduction
  • ndrange, hierarchical and scoped
• Compilers:
  • hipSYCL: feature/revisit-scoped-parallelism (4615409) with LLVM 11 and GCC 10.3
  • DPC++: oneAPI v2021.3.0, dpcpp from August 19, 2021

<table>
<thead>
<tr>
<th>Processor</th>
<th>Details</th>
<th>Main memory bandwidth (GB/s)</th>
<th>FP64 TFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Cascade Lake Gold 6230</td>
<td>2x20 cores</td>
<td>282</td>
<td>2.688</td>
</tr>
<tr>
<td>NVIDIA V100</td>
<td>PCIe</td>
<td>900</td>
<td>7</td>
</tr>
</tbody>
</table>

Results collected on Isambard MACS: https://gw4.ac.uk/isambard/
SYCL-Bench on V100

### Table II

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>hipSYCL</td>
</tr>
<tr>
<td>nbody</td>
<td>ndrange</td>
<td>0.185</td>
</tr>
<tr>
<td></td>
<td>hierarchical</td>
<td>0.187</td>
</tr>
<tr>
<td></td>
<td>scoped</td>
<td>0.187</td>
</tr>
<tr>
<td>reduction</td>
<td>ndrange</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>hierarchical</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>scoped</td>
<td>0.002</td>
</tr>
<tr>
<td>segmented reduction</td>
<td>ndrange</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>hierarchical</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>scoped</td>
<td>0.002</td>
</tr>
</tbody>
</table>
SYCL-Bench on Cascade Lake

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Kernel</th>
<th>Runtime (s)</th>
<th>hipSYCL</th>
<th>LLVM</th>
<th>GCC</th>
<th>DPCPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbody</td>
<td>ndrange</td>
<td>4.053</td>
<td>3.478</td>
<td>0.700</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hierarchical</td>
<td>3.323</td>
<td>2.966</td>
<td>4.952</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>scoped</td>
<td>5.665</td>
<td>4.250</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reduction</td>
<td>ndrange</td>
<td>6.543</td>
<td>4.430</td>
<td>0.022</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hierarchical</td>
<td>0.033</td>
<td>0.044</td>
<td>0.457</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>scoped</td>
<td>0.033</td>
<td>0.029</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>segmented</td>
<td>ndrange</td>
<td>6.438</td>
<td>4.408</td>
<td>0.020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reduction</td>
<td>hierarchical</td>
<td>0.037</td>
<td>0.036</td>
<td>0.207</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>scoped</td>
<td>0.035</td>
<td>0.028</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the nbody benchmark on the CPU, the differences between the models are less significant in hipSYCL. This is because the nbody benchmark is highly compute-bound, with a high amount of work per work-item and few barriers. Performance is therefore mostly dependent on how well the compiler auto-vectorizes a particular formulation of the kernel. In addition, nbody is the only benchmark in this study that uses private_memory allocations in the hierarchical and scoped versions. At present, this is implemented with a dynamic memory allocation on the heap for every work group to allocate sufficient memory for each work item, which is avoidable in the DPCPP compiler-based implementation of SYCL.
Summary

• Hierarchical and Scoped parallelism give weaker forward progress guarantees
  • Allows for easier library-only implementations on CPU

• Performance can depend on SYCL implementation choices, and underlying compiler

• Scoped Parallelism:
  • Implementor flexibility to map to hardware
  • Programmer control of synchronisation
  • Exposes deeper hierarchy (sub-groups, sub-sub-groups)
  • Private memory by default prevents accidental use of local memory

• Try it out for yourself:
  • https://github.com/illuhad/hipSYCL/pull/619
  • https://github.com/UoB-HPC/sycl_dgemm
  • https://github.com/illuhad/sycl-bench/tree/scoped-parallelism