CUDA, OpenCL, SYCL, .. offer hierarchical execution models
The hierarchy is tailored to GPUs’ architecture
- Work-item (Thread)
- Sub-group (Warp)
- Work-group (Block ⇒ SM)
- Global (Grid)
High costs for work-group synchronization on CPUs!
TARGETING PERFORMANCE PORTABILITY WITH

nd_range parallel_for

- CUDA, OpenCL, SYCL, .. offer hierarchical execution models
- The hierarchy is tailored to GPUs’ architecture
  - Work-item (Thread)
  - Sub-group (Warp)
  - Work-group (Block ⇒ SM)
  - Global (Grid)
- High costs for work-group synchronization on CPUs!
• Multi-target SYCL Implementation by Uni Heidelberg
• Uses Clang HIP, CUDA and DPC++ augmented by plugin for GPU
• OpenMP CPU backend (traditionally library-only)
WORK-GROUP SYNCHRONIZATION

```cpp
1  cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
2     [=](sycl::nd_item<1> item) noexcept {
3     const auto lid = item.get_local_id(0);
4     scratch[lid] = acc[item.get_global_id()];
5     for(size_t i = group_size / 2; i > 0; i /= 2) {
6       item.barrier();
7       if(lid < i) scratch[lid] += scratch[lid+i];
8     }
9     if(lid == 0) acc[item.get_global_id()] = scratch[lid];
10    });
```
WORK-GROUP SYNCHRONIZATION

```cpp
1  cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
2      [=](sycl::nd_item<1> item) noexcept {
3      const auto lid = item.get_local_id(0);
4      scratch[lid] = acc[item.get_global_id()];
5      for(size_t i = group_size / 2; i > 0; i /= 2) {
6          item.barrier();
7          if(lid < i) scratch[lid] += scratch[lid + i];
8      }
9      if(lid == 0) acc[item.get_global_id()] = scratch[lid];
10  });
```
WORK-GROUP SYNCHRONIZATION — GPU

- Execution of many (mostly) independent threads
  → Forward-Progress guarantees
- __syncthreads()
WORK-GROUP SYNCHRONIZATION — CPU

- 1 thread : 1 work-item (< v0.9)
  - #pragma omp barrier
  - Many threads → OS / scheduling overhead
  - No vectorization across work-items
WORK-GROUP SYNCHRONIZATION — CPU

- 1 thread : 1 work-item (< v0.9)
  - #pragma omp barrier
  - Many threads → OS / scheduling overhead
  - No vectorization across work-items
- Boost.Fiber? (>= 0.9)
  - Lightweight threads + synchronization
  - Can optimize barrier-free kernels!
  - Scheduling overhead
  - Limited vectorization across work-items
Our work: Compiler-aided loop-fission

- Threading on the **work-group level**
- Loop over work-items in a single thread
- Compiler extension to split **work-item loop** at barriers
- Support targets that Clang supports OpenMP on, **no OpenCL runtime** required

Diagram:

1. SYCL code
2. SYCL compiler
3. SPIR-V
4. OpenCL
5. Compiler Transformations
6. Machine code

- Depend on OpenCL to transform to (hopefully) efficient CPU kernel
- Our work skips the additional dependency
- Compiler Transformations
POCL's Loop-Fission Approach
POCL'S LOOP-FISSION APPROACH

Diagram:
- A1
- A2
- Split

Flowchart:
- A1
- A2
POCL'S LOOP-FISSION APPROACH
CBS LOOP-FISSION APPROACH
CBS LOOP-FISSION APPROACH
CBS LOOP-FISSION APPROACH

A1
A2

Next: 0

0
A1
Next: 1

1
A2
Next: -1

Split

Next: 0

-1
POCL — SEMANTIC PROBLEM

```cpp
[=](sycl::nd_item<1> item) noexcept {
    const auto lid = item.get_local_id(0);
    scratch[lid] = acc[item.get_global_id()];
    item.barrier();
    for(size_t i = 0; i < 2 + lid; i++) {
        scratch[lid] += i;
        // only call the barrier if all work-items still run the loop.
        if(i < 2) item.barrier();
    }
    acc[item.get_global_id()] = scratch[lid];
}
```
POCL — SEMANTIC PROBLEM

```cpp
[=](sycl::nd_item<1> item) noexcept {
    const auto lid = item.get_local_id(0);
    scratch[lid] = acc[item.get_global_id()];
    item.barrier();
    for(size_t i = 0; i < 2 + lid; i++) {
        scratch[lid] += i;
        // only call the barrier if all work-items still run the loop.
        if(i < 2) item.barrier();
    }
    acc[item.get_global_id()] = scratch[lid];
}
```
BENCHMARK — SYSTEMS

- **Fujitsu A64FX "a64fx"**
  - 1x 1.8GHz 48-core CPU
  - 512bit SVE
  - 32GB HBM2 RAM

- **Marvell ThunderX2 "xci"**
  - 2x 2.1GHz 32-core, 128-threads CPU
  - NEON
  - 256GB DDR4-2666MHz

- **Intel Xeon Gold 6338 "ilake"**
  - 2x 2.00GHz 32-core, 64-threads Icelake CPU
  - AVX512
  - 256GB DDR4 RAM

- **AMD Epyc 7442 "rome"**
  - 2x 2.25GHz 64-core, 128-threads Rome CPU
  - AVX2
  - 256GB DDR4-3200Mhz

*Provided by*
- Isambard 2 UK
- National Tier-2 HPC Service
LLVM 14-git, Boost 1.77, Speedup of respective max. throughput
LLVM 14-git, Boost 1.77, Speedup of respective max. throughput
## POCL VS. CBS

<table>
<thead>
<tr>
<th>Feature</th>
<th>POCL</th>
<th>CBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYCL conforming semantic</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Linear control-flow</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Code size before vectorization</td>
<td>+14-45%</td>
<td></td>
</tr>
<tr>
<td>Independence of flags</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Expected maintenance cost</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Geomean speedup</td>
<td>23</td>
<td>34</td>
</tr>
</tbody>
</table>
IMPACT
Loop fission improves nd-range performance significantly
⇒ first SYCL implementation with competitive nd-range parallel-for without OpenCL runtime
IMPACT

Loop fission improves nd-range performance significantly
⇒ first SYCL implementation with competitive nd-range parallel-for without OpenCL runtime

CBS available in hipSYCL since v0.9.2
Loop fission improves nd-range performance significantly
⇒ first SYCL implementation with competitive nd-range parallel-for without OpenCL runtime

CBS available in hipSYCL since v0.9.2

Interest in having CBS upstreamed in LLVM?
Usable for OpenCL, SYCL, (CUDA, HIP,...?) CPU runtimes
(Interface: if kernel & barrier identified by annotation found, apply)
THANKS FOR WATCHING!

LOOKING FORWARD TO YOUR QUESTIONS AND FEEDBACK

Contact: Joachim Meyer
jmeyer@cs.uni-saarland.de
github.com/illuhad/hipSYCL