SOLID WORK-GROUP SYNCHRONIZATION ON CPUS

LLVM PERFORMANCE WORKSHOP @ CGO'23

JOACHIM MEYER, SEBASTIAN HACK (SAARLAND UNIVERSITY) AND
AKSEL ALPAY, HOLGER FRÖNING, VINCENT HEUVELINE (HEIDELBERG UNIVERSITY)

jmeyer@cs.uni-saarland.de

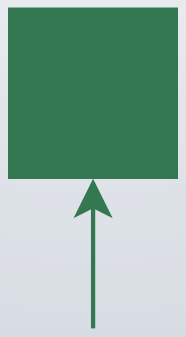
WHY SHOULD WE CARE?

• • •

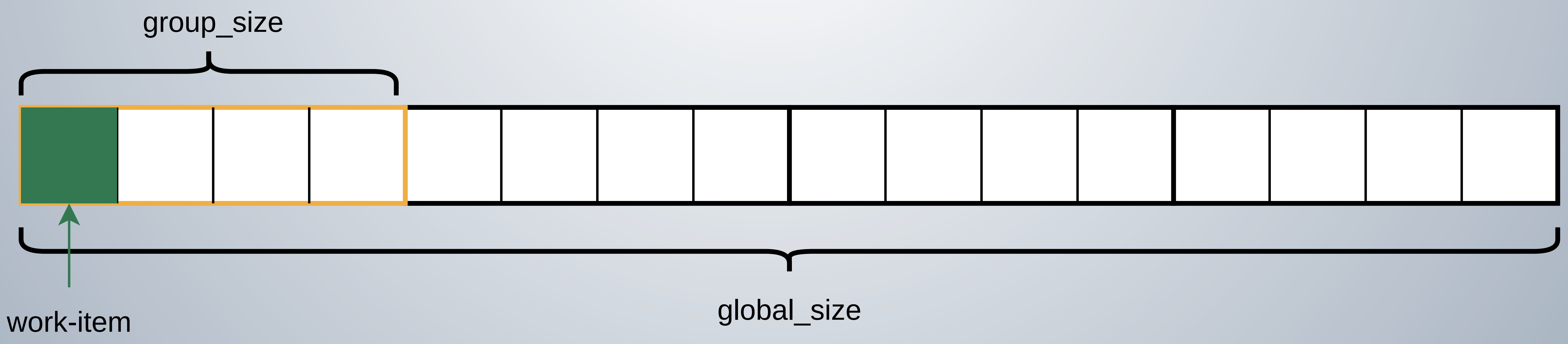
WORK-GROUP SYNCHRONIZATION IS RE-IMPLEMENTED TOO OFTEN

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
    const auto lid = item.get_local_id(0);
    scratch[lid] = acc[item.get_global_id()];
    for(size_t i = group_size / 2; i > 0; i /= 2) {
      item.barrier();
      if(lid < i) scratch[lid] += scratch[lid + i];
    }

    if(lid == 0) acc[item.get_global_id()] = scratch[lid];
  });
```

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
    const auto lid = item.get_local_id(0);
    scratch[lid] = acc[item.get_global_id()];
    for(size_t i = group_size / 2; i > 0; i /= 2) {
      item.barrier();
      if(lid < i) scratch[lid] += scratch[lid + i];
    }

    if(lid == 0) acc[item.get_global_id()] = scratch[lid];
  });
```

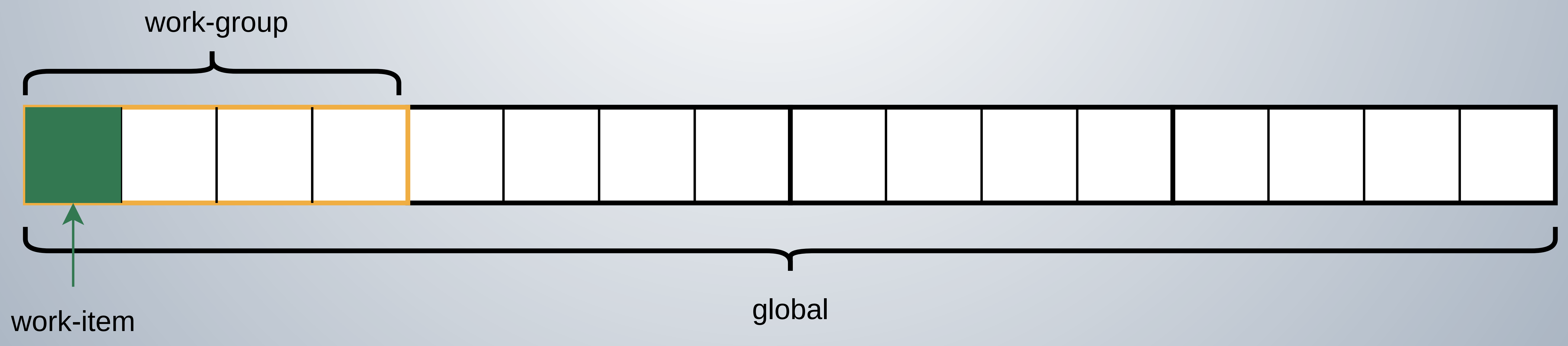


```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
    const auto lid = item.get_local_id(0);
    scratch[lid] = acc[item.get_global_id()];
    for(size_t i = group_size / 2; i > 0; i /= 2) {
      item.barrier();
      if(lid < i) scratch[lid] += scratch[lid + i];
    }

    if(lid == 0) acc[item.get_global_id()] = scratch[lid];
  });
```



THE BIG BLOCKER

```
cgh.parallel_for(sycl::nd_range<1>{global_size, group_size},
  [=](sycl::nd_item<1> item) noexcept {
    const auto lid = item.get_local_id(0);
    scratch[lid] = acc[item.get_global_id()];
    for(size_t i = group_size / 2; i > 0; i /= 2) {
      item.barrier();
      if(lid < i) scratch[lid] += scratch[lid + i];
    }

    if(lid == 0) acc[item.get_global_id()] = scratch[lid];
  });
```



THIS IS "SIMPLE" ON GPUS

- Execution of many (mostly) independent threads
  → Forward-Progress guarantees
- Hardware support for work-group barriers

HOW TO MAP THIS TO CPUS?

CONCURRENCY!

- 1 work-item : 1 thread
  - E.g. OpenMP parallel for + #pragma omp barrier

⚡Many threads → scheduling overhead

⚡No vectorization across work-items

CONCURRENCY!

- 1 work-item : 1 thread
  - E.g. OpenMP parallel for + #pragma omp barrier

⚡Many threads → scheduling overhead

⚡No vectorization across work-items

- 1 work-item : 1 fiber
  - Lightweight threads + synchronization
  - Can optimize barrier-free kernels!

⚡Context-switch overhead

⚡Limited vectorization across work-items

CONCURRENCY!

- 1 work-item : 1 thread
  - E.g. OpenMP parallel for + #pragma omp barrier

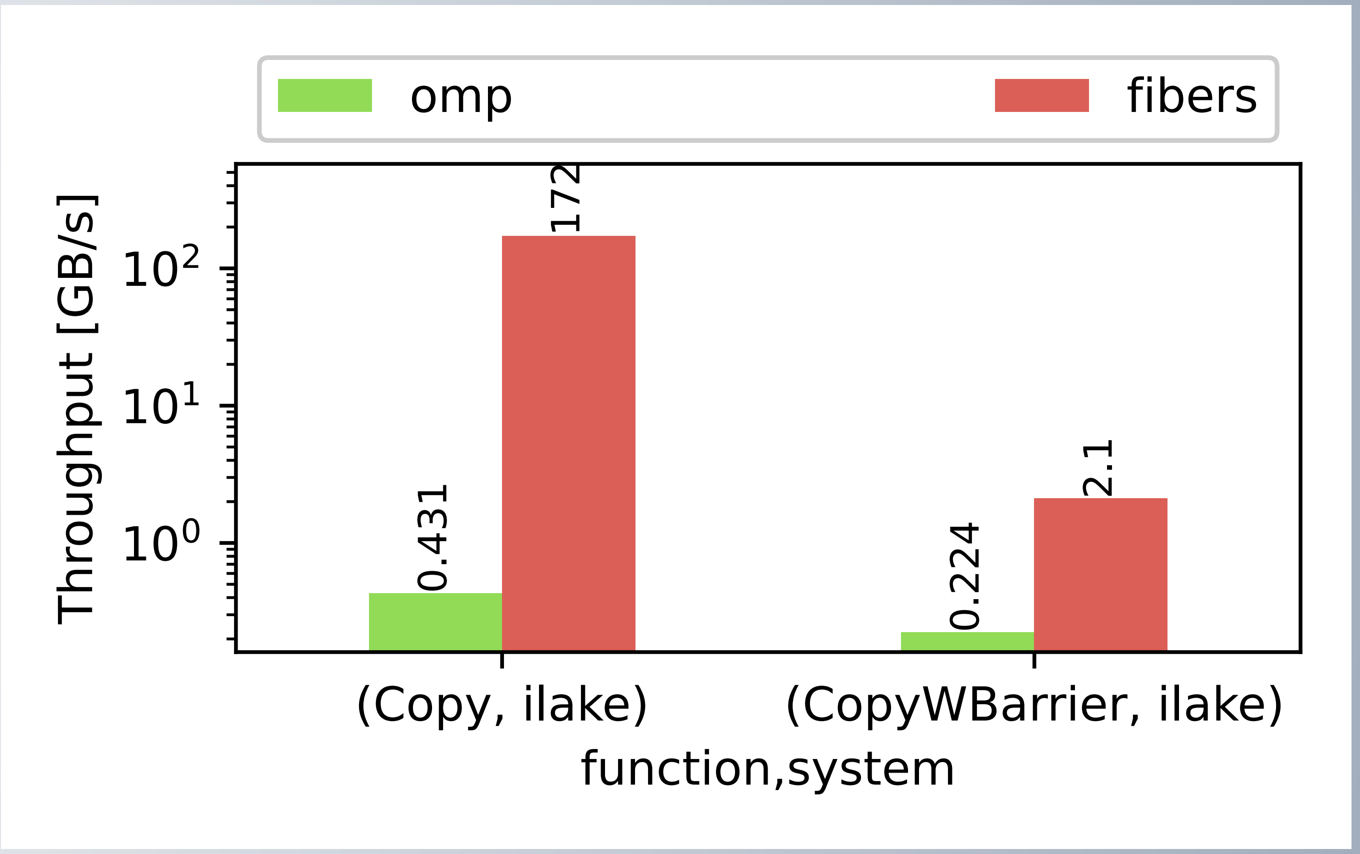
⚡Many threads → scheduling overhead

⚡No vectorization across work-items

- 1 work-item : 1 fiber
  - Lightweight threads + synchronization
  - Can optimize barrier-free kernels!

⚡Context-switch overhead

⚡Limited vectorization across work-items



github.com/UoB-HPC/BabelStream

CONCURRENCY!

- 1 work-item : 1 thread
  - E.g. OpenMP parallel for + #pragma omp barrier

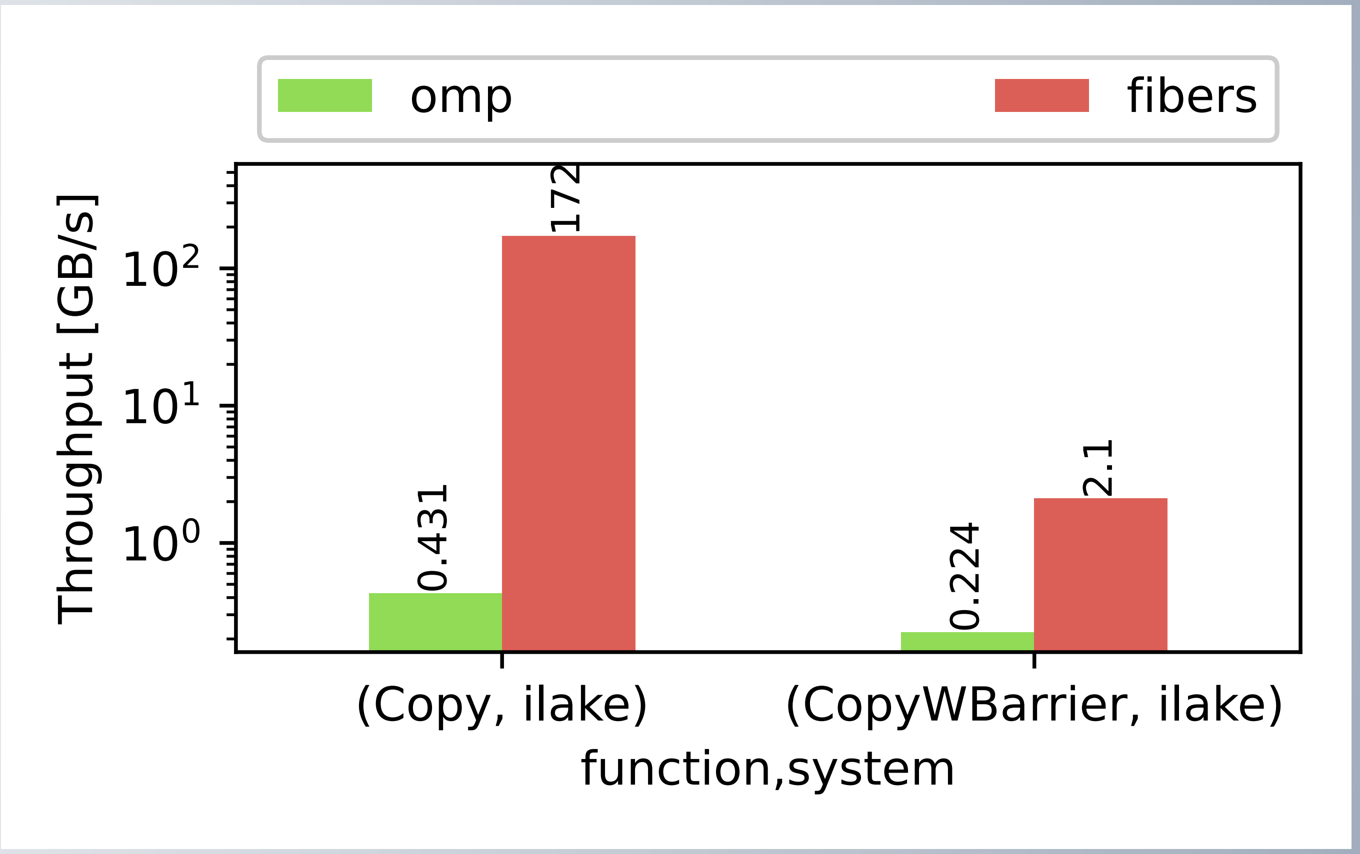
⚡Many threads → scheduling overhead

⚡No vectorization across work-items

- 1 work-item : 1 fiber
  - Lightweight threads + synchronization
  - Can optimize barrier-free kernels!

⚡Context-switch overhead

⚡Limited vectorization across work-items



github.com/UoB-HPC/BabelStream

✅ **Can be implemented without dedicated compiler support!**

NO, USE THE COMPILER!*

- Threading on the **work-group level**
- Loop over work-items in a single thread
- Compiler extension to **split kernel** at barriers

✅ Vectorization across work-items possible
✅ Improves performance over library-only by up to **several orders of magnitude**

```
#pragma omp parallel for
for(group : groups)
  #pragma omp simd
  for(item : itemsInGroup)
    kernel_before_barrier(nd_item{group, item})
  // implicit synchronization
  #pragma omp simd
  for(item : itemsInGroup)
    kernel_after_barrier(nd_item{group, item})
```

*After all, this is a compiler workshop 🤷‍♂️

DEEP LOOP FISSION (POCL)

```
[=](sycl::nd_item<1> item) {
  const auto lid = item.get_local_id(0);
  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier();
  for(size_t i = group_size / 2; i > 0; i /= 2) {
    if(lid < i) scratch[lid] += scratch[lid + i]; // B
    item.barrier();
  }

  if(lid == 0) acc[item.get_global_id()] = scratch[lid]; // C
}
```

```
for(lid : items[0:])
  // A
// barrier
for(i = group_size / 2; i > 0; i /= 2)
  // B (lid = 0)
  for(lid : items[1:])
    // B
  // barrier
for(lid : items[0:])
  // C
```

DEEP LOOP FISSION (POCL)

```
[=](sycl::nd_item<1> item) {
  const auto lid = item.get_local_id(0);
  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier();
  for(size_t i = group_size / 2; i > 0; i /= 2) {
    if(lid < i) scratch[lid] += scratch[lid + i]; // B
    item.barrier();
  }

  if(lid == 0) acc[item.get_global_id()] = scratch[lid]; // C
}
```

```
for(lid : items[0:])
  // A
// barrier
for(i = group_size / 2; i > 0; i /= 2)
  // B (lid = 0)
  for(lid : items[1:])
    // B
  // barrier
for(lid : items[0:])
  // C
```

DEEP LOOP FISSION (POCL)

```
[=](sycl::nd_item<1> item) {
  const auto lid = item.get_local_id(0);
  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier();
  for(size_t i = group_size / 2; i > 0; i /= 2) {
    if(lid < i) scratch[lid] += scratch[lid + i]; // B
    item.barrier();
  }

  if(lid == 0) acc[item.get_global_id()] = scratch[lid]; // C
}
```

```
for(lid : items[0:])
  // A
// barrier
for(i = group_size / 2; i > 0; i /= 2)
  // B (lid = 0)
  for(lid : items[1:])
    // B
  // barrier
for(lid : items[0:])
  // C
```

DEEP LOOP FISSION (POCL)

```
[=](sycl::nd_item<1> item) {
  const auto lid = item.get_local_id(0);
  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier();
  for(size_t i = group_size / 2; i > 0; i /= 2) {
    if(lid < i) scratch[lid] += scratch[lid + i]; // B
    item.barrier();
  }

  if(lid == 0) acc[item.get_global_id()] = scratch[lid]; // C
}
```

```
for(lid : items[0:])
  // A
// barrier
for(i = group_size / 2; i > 0; i /= 2)
  // B (lid = 0)
  for(lid : items[1:])
    // B
  // barrier
for(lid : items[0:])
  // C
```

DEEP LOOP FISSION (POCL)

```
[=](sycl::nd_item<1> item) {
  const auto lid = item.get_local_id(0);
  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier();
  for(size_t i = group_size / 2; i > 0; i /= 2) {
    if(lid < i) scratch[lid] += scratch[lid + i]; // B
    item.barrier();
  }

  if(lid == 0) acc[item.get_global_id()] = scratch[lid]; // C
}
```

```
for(lid : items[0:])
  // A
// barrier
for(i = group_size / 2; i > 0; i /= 2)
  // B (lid = 0)
  for(lid : items[1:])
    // B
  // barrier
for(lid : items[0:])
  // C
```

DEEP LOOP FISSION – SEMANTIC PROBLEM

```
[=](sycl::nd_item<1> item) noexcept {
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier();

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i; // B
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier();
  }
  acc[item.get_global_id()] = scratch[lid]; // C
}
```

```
for(lid : items[0:])
  // A
// barrier
for(i : [0,1])
  // B (lid = 0)
  for(lid : items[1:])
    // B
  if(i < 2)
    // barrier
for(lid : items[0:])
  // C
```

DEEP LOOP FISSION – SEMANTIC PROBLEM

```
[=](sycl::nd_item<1> item) noexcept {
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier();

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i; // B
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier();
  }
  acc[item.get_global_id()] = scratch[lid]; // C
}
```

```
for(lid : items[0:])
  // A
// barrier
for(i : [0,1])
  // B (lid = 0)
  for(lid : items[1:])
    // B
  if(i < 2)
    // barrier
for(lid : items[0:])
  // C
```

CONTINUATION-BASED SYNCHRONIZATION TO THE RESCUE

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

OpenSYCL

UNIVERSITÄT DES SAARLANDES

SIC Saarland Informatics Campus

```
[=](sycl::nd_item<1> item) noexcept { // 0
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier(); // 1

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i; // B
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier(); // 2
  }
  acc[item.get_global_id()] = scratch[lid]; // C
} // -1
```

Continuation-based Synchronization

```




    case 0:

        // A
        // barrier

    case 1:

        i = 0
        while(i < 2 + lid)
          // B
          if(i < 2) // barrier
          i++;
        // C


    case 2:

        i++;
        while(i < 2 + lid)
          // B
          if(i < 2)  // barrier
          i++;
        // C
```

CONTINUATION-BASED SYNCHRONIZATION TO THE RESCUE

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

OpenSYCL

UNIVERSITÄT DES SAARLANDES

SIC Saarland Informatics Campus

```
[=](sycl::nd_item<1> item) noexcept { // 0
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier(); // 1

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i; // B
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier(); // 2
  }
  acc[item.get_global_id()] = scratch[lid]; // C
} // -1
```

Continuation-based Synchronization

```
i[items] = alloca ..;



    case 0:
      for(lid : items[0:])
        // A
        // barrier

    case 1:
      for(lid : items[0:])
        i[lid] = 0
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) // barrier
          i[lid]++;
        // C


    case 2:
      for(lid : items[0:])
        i[lid]++;
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) // barrier
          i[lid]++;
        // C
```

CONTINUATION-BASED SYNCHRONIZATION TO THE RESCUE

```
[=](sycl::nd_item<1> item) noexcept { // 0
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier(); // 1

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i; // B
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier(); // 2
  }
  acc[item.get_global_id()] = scratch[lid]; // C
} // -1
```

Continuation-based Synchronization

```
i[items] = alloca ..;
next = 0;


    case 0:
      for(lid : items[0:])
        // A
        next = 1;

    case 1:
      cont1: for(lid : items[0:])
        i[lid] = 0
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) next = 2; goto cont1;
          i[lid]++;
        // C
        next = -1;

    case 2:
      cont2: for(lid : items[0:])
        i[lid]++;
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) next = 2; goto cont2;
          i[lid]++;
        // C
        next = -1;
```

CONTINUATION-BASED SYNCHRONIZATION TO THE RESCUE

```
[=](sycl::nd_item<1> item) noexcept { // 0
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier(); // 1

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i; // B
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier(); // 2
  }
  acc[item.get_global_id()] = scratch[lid]; // C
} // -1
```

Continuation-based Synchronization

```
i[items] = alloca ..;
next = 0;
while(next != -1) {
  switch(next) {
    case 0:
      for(lid : items[0:])
        // A
        next = 1;
      break;
    case 1:
      cont1: for(lid : items[0:])
        i[lid] = 0
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) next = 2; goto cont1;
          i[lid]++;
        // C
        next = -1;
      break;
    case 2:
      cont2: for(lid : items[0:])
        i[lid]++;
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) next = 2; goto cont2;
          i[lid]++;
        // C
        next = -1;
      break;
  }
}
```

CONTINUATION-BASED SYNCHRONIZATION TO THE RESCUE

```
[=](sycl::nd_item<1> item) noexcept { // 0
  const auto lid = item.get_local_id(0);

  scratch[lid] = acc[item.get_global_id()]; // A
  item.barrier(); // 1

  for(size_t i = 0; i < 2 + lid; i++) {
    scratch[lid] += i; // B
    // only call the barrier if all work-items still run the loop.
    if(i < 2) item.barrier(); // 2
  }
  acc[item.get_global_id()] = scratch[lid]; // C
} // -1
```

Deep Loop Fission

```
for(lid : items[0:])
  // A
// barrier
for(i : [0,1])
  // B (lid = 0)
  for(lid : items[1:])
    // B
  if(i < 2)
    // barrier
for(lid : items[0:])
  // C
```

Continuation-based Synchronization

```
i[items] = alloca ..;
next = 0;
while(next != -1) {
  switch(next) {
    case 0:
      for(lid : items[0:])
        // A
        next = 1;
      break;
    case 1:
      cont1: for(lid : items[0:])
        i[lid] = 0
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) next = 2; goto cont1;
          i[lid]++;
        // C
        next = -1;
      break;
    case 2:
      cont2: for(lid : items[0:])
        i[lid]++;
        while(i[lid] < 2 + lid)
          // B
          if(i[lid] < 2) next = 2; goto cont2;
          i[lid]++;
        // C
        next = -1;
      break;
  }
}
```

HOW ARE WORK-ITEM PRIVATE VALUES STORED?

✅ Dynamically-sized stack arrays with large alignment (64)

```
value[items] = alloca ..;
case 1:
  for(lid : items[0:])
    value1 = global[offset + lid];
    value[lid] = value1;
case 2:
  for(lid : items[0:])
    value2 = value[lid];
```

HOW TO AVOID STORING UNIFORM VALUES TO THOSE ARRAYS?

✅ Value shape analysis based on LLVM's SyncDependenceAnalysis

```
offset[items] = alloca ..;
case 1:
  for(lid : items[0:])
    offset1 = 0; // uniform
    offset[lid] = offset1;
case 2:

  for(lid : items[0:])
    offset2 = offset[lid];
```

⇒

```
offset = alloca ..;
case 1:
  for(lid : items[0:])
    offset1 = 0; // uniform
    offset = offset1;
case 2:
  offset2 = offset;
  for(lid : items[0:])
    // ..
```

HOW TO PROPAGATE VALUE CONTIGUITY TO THE LLVM OPTIMIZER?

✅ Value shape analysis + trace cont values to uniform values & wi-index + replicate trace after barrier

```
idx[items] = alloca ..;
case 1:
  for(lid : items[0:])
    idx1 = offset1 + lid; // contiguous
    idx[lid] = idx1
case 2:

  for(lid : items[0:])
    idx2 = idx[lid];
    // is this a contiguous access?
    ptr[idx2] = ..;
```

⇒

```
offset = alloca  ..; // uniform
case 1:
  for(lid : items[0:])
    idx1 = offset1 + lid; // contiguous
    offset = offset1
case 2:
  offset2 = offset
  for(lid : items[0:])
    idx2 = offset2 + lid;
    // this is a contiguous access!
    ptr[idx2] = ..;
```

HOW TO IMPLEMENT/MAP SUB-GROUP ALGORITHMS?

- Outer-loop vectorization of wi-loops + intrinsics (RV)
- Hierarchically split again at sub-group "barriers"

Scalar kernel

```
// A
item.barrier();
// B
cond = group_all_of(sg, x < 32); // barrier
// C
```

Outer-loop vectorized (RV)

```
for(lid : items[0::4])
  // A<4>
for(lid : items[0::4])
  // B<4>
  cond = rv_all(x < 32)
  // C<4>
```

HOW TO IMPLEMENT/MAP SUB-GROUP ALGORITHMS?

🟨 Outer-loop vectorization of wi-loops + intrinsics (RV)

🟨 Hierarchically split again at sub-group "barriers"

Scalar kernel

```
// A
item.barrier();
// B
cond = group_all_of(sg, x < 32); // barrier
// C
```

Outer-loop vectorized (RV)

```
for(lid : items[0::4])
  // A<4>
for(lid : items[0::4])
  // B<4>
  cond = rv_all(x < 32)
  // C<4>
```

HOW TO IMPLEMENT/MAP SUB-GROUP ALGORITHMS?

🟨 Outer-loop vectorization of wi-loops + intrinsics (RV)

🟨 Hierarchically split again at sub-group "barriers"

Scalar kernel

```
// A
item.barrier();
// B
cond = group_all_of(sg, x < 32); // barrier
// C
```

Outer-loop vectorized (RV)

```
for(lid : items[0::4])
  // A<4>
for(lid : items[0::4])
  // B<4>
  cond = rv_all(x < 32)
  // C<4>
```

HOW TO IMPLEMENT/MAP SUB-GROUP ALGORITHMS?

- 🟨 Outer-loop vectorization of wi-loops + intrinsics (RV)
- 🟨 Hierarchically split again at sub-group "barriers"

Scalar kernel

```
// A
item.barrier();
// B
cond = group_all_of(sg, x < 32); // barrier
// C
```

Outer-loop vectorized (RV)

```
for(lid : items[0::4])
  // A<4>
for(lid : items[0::4])
  // B<4>
  cond = rv_all(x < 32)
  // C<4>
```

Hierarchical split

```
c[max_sg_size] = alloca ..;
for(sg : sub_groups)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // A
for(sg : sub_groups)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // B
    c[sid] = x < 32
  cond = group_all_of(sg, c)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // C
```

HOW TO IMPLEMENT/MAP SUB-GROUP ALGORITHMS?

🟨 Outer-loop vectorization of wi-loops + intrinsics (RV)

🟨 Hierarchically split again at sub-group "barriers"

Scalar kernel

```
// A
item.barrier();
// B
cond = group_all_of(sg, x < 32); // barrier
// C
```

Outer-loop vectorized (RV)

```
for(lid : items[0::4])
  // A<4>
for(lid : items[0::4])
  // B<4>
  cond = rv_all(x < 32)
  // C<4>
```

Hierarchical split

```
c[max_sg_size] = alloca ..;
for(sg : sub_groups)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // A
for(sg : sub_groups)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // B
    c[sid] = x < 32
  cond = group_all_of(sg, c)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // C
```

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386 OpenSYCL
UNIVERSITÄT DES SAARLANDES SIC Saarland Informatics Campus

HOW TO IMPLEMENT/MAP SUB-GROUP ALGORITHMS?

🟨 Outer-loop vectorization of wi-loops + intrinsics (RV)

🟨 Hierarchically split again at sub-group "barriers"

Scalar kernel

```
// A
item.barrier();
// B
cond = group_all_of(sg, x < 32); // barrier
// C
```

Outer-loop vectorized (RV)

```
for(lid : items[0::4])
  // A<4>
for(lid : items[0::4])
  // B<4>
  cond = rv_all(x < 32)
  // C<4>
```

Hierarchical split

```
c[max_sg_size] = alloca ..;
for(sg : sub_groups)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // A
for(sg : sub_groups)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // B
    c[sid] = x < 32
  cond = group_all_of(sg, c)
  for(sid : sg.items)
    lid = sg.offset + sid;
    // C
```

PHI NODES ARE HARD.

❌ PoCL demotes PHI nodes to make transformations feasible

✅ CBS has a single flow change → keep PHIs

SO, DEEP LOOP FISSION OR CBS?

	DLF (PoCL)	CBS
Correct barrier semantic	✗	✓
Expected maintenance cost	✗	✓
Linearized control-flow	✓	✗
Geomean speedup/fiber SYCL	29	38
Performance in OpenCL	on par	

PROPOSAL

- Continuation-based synchronization as common infrastructure in LLVM upstream.

⇒No need to re-implement for new programming models / implementations

⇒New features benefit all implementations (sub-group support, uniformity analysis improvements, ...)

PROPOSAL

- Continuation-based synchronization as common infrastructure in LLVM upstream.

⇒No need to re-implement for new programming models / implementations

⇒New features benefit all implementations (sub-group support, uniformity analysis improvements, ...)

- Reuse our battle-tested open-source implementation from Open SYCL (formerly hipSYCL)
- Already used as fallback in PoCL upstream

DESIGN

- Add `llvm.spmd.barrier` intrinsic
- `cbs_kernel` annotation
- `llvm.spmd.group_size`, `llvm.spmd.local_id`
- CBS passes can be added to custom pipeline or extension points

THANK YOU FOR LISTENING!

LOOKING FORWARD TO QUESTIONS AND DISCUSSIONS

Contact: Joachim Meyer

jmeyer@cs.uni-saarland.de

github.com/OpenSYCL/OpenSYCL