Parallelizing Applications With Indirect Memory Writes in MLIR

Pablo Antonio Martinez
pablo.antonio.martinez@huawei.com

Huawei Technologies R&D (UK)
Cambridge Research Centre – Compiler Lab
Date: 10/04/24

HUAWEI

Introduction

What are indirect memory writes? Why are they hard to parallelize?

Indirect memory writes involve writing to an array where the access is based on indices stored in another array.

For example (histogram computation):

```
for (int i=0; i < N*N; i++) {
  const unsigned int idx = img[i];
  if (histo[idx] < UINT8_MAX) {
    histo[idx]++;
  }
}
```

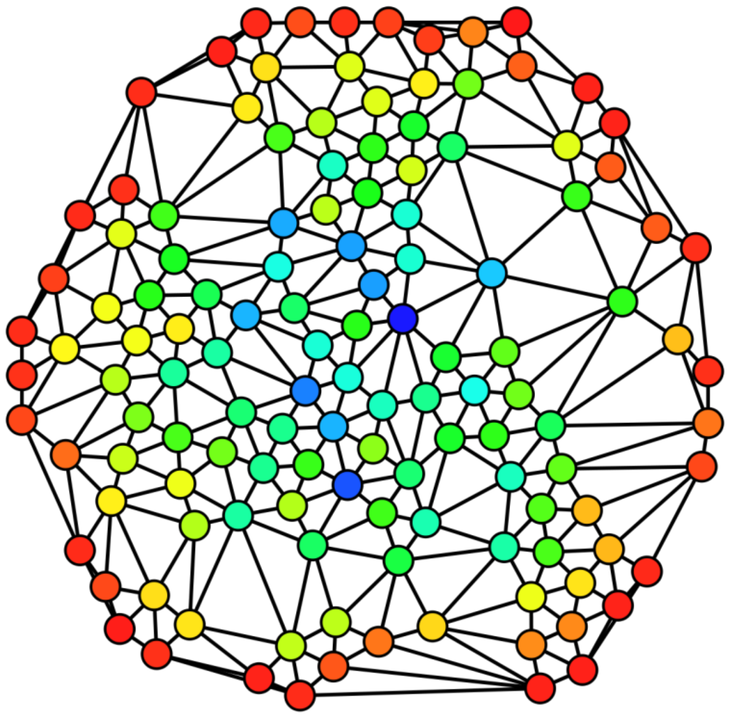
It **CAN'T** be parallelized like this!

```
#pragma omp parallel for reduction(+:histo)
for (int i=0; i < N*N; i++) {
  const unsigned int idx = img[i];
  if (histo[idx] < UINT8_MAX) {
    histo[idx]++;
  }
}
```

Motivation

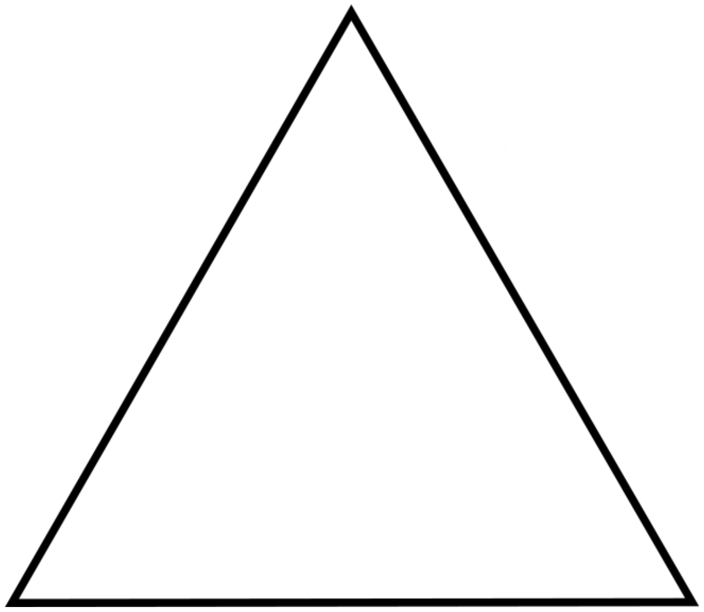
Why are indirect memory writes important?

This access pattern appears in many AI and HPC applications like:

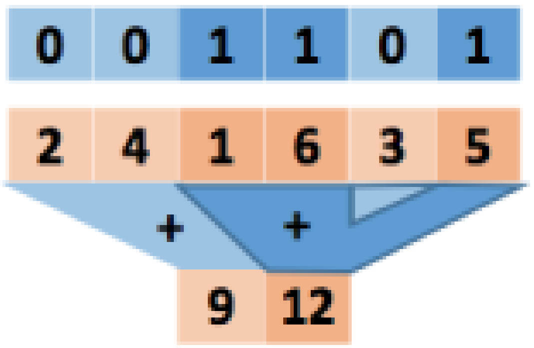


Betweenness Centrality

Credit: Claudio Rocchini, Wikipedia



Triangle Counting



Unsorted Segment Sum

Credit: Mindspore docs, Huawei

Challenges

- In MLIR we usually parallelize code using `structured.tile_using_forall` (limited to linalg ops)

```
func.func @tile_output_multi_1d_static(%IN1: tensor<100xf32>, %IN2: tensor<100xf32>,
                                        %OUT1: tensor<100xf32>, %OUT2: tensor<100xf32>)
                                        -> (tensor<100xf32>, tensor<100xf32>) {
  %res1, %res2 = linalg.generic { indexing_maps = [...], iterator_types = ["parallel"] }
    ins(%IN1, %IN2 : tensor<100xf32>, tensor<100xf32>)
    outs(%OUT1, %OUT2 : tensor<100xf32>, tensor<100xf32>) {
    ^bb0(%a1: f32, %a2: f32, %a3: f32, %a4: f32):
      %1 = arith.addf %a1, %a3 : f32
      %2 = arith.addf %a2, %a4 : f32
      linalg.yield %1, %2 : f32,f32
  } -> (tensor<100xf32>, tensor<100xf32>)
  return %res1, %res2 : tensor<100xf32>, tensor<100xf32>
}
module attributes {transform.with_named_sequence} {
  ...
  transform.structured.tile_using_forall %0 num_threads [7] ...
  ...
}
```

- An indirect memory write cannot be represented with a `linalg.generic`!

Challenges

Representing an indirect memory write with loops:

```
%0 = scf.for %arg0 = %c0 to %1 step %c1 iter_args(%arg1 = %2) ->
(tensor<?xi8>) {
  %extracted = tensor.extract %indices[%arg0] : tensor<?xi32>
  %idx = arith.index_cast %extracted : i32 to index
  %4 = tensor.extract %buff[%idx] : tensor<?xi32>
  %5 = arith.addi %4, %c1 : i32
  %inserted = tensor.insert %5 into %buff[%idx] : tensor<?xi32>
  ...
  scf.yield %inserted : tensor<?x32>
}
```

Indirect read

Indirect write

Indirect memory write example in MLIR (expressed with loops)

Proposal

We add:

1. Tiling at the loop level (`loop.tile_using_forall`)
2. `privatize_buffers` option

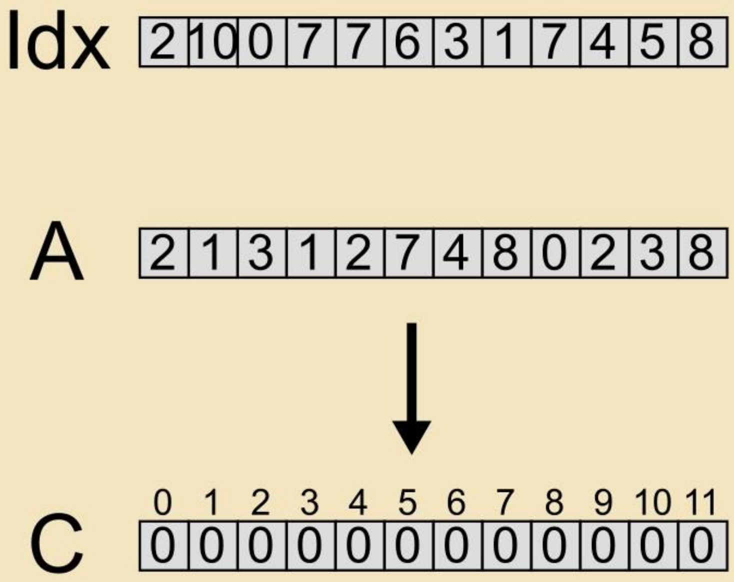
```
%0 = scf.for %arg0 = %c0 to %1 step %c1 iter_args(%arg1 = %2) -> (tensor<?xi8>) {
  %extracted = tensor.extract %arg2[%arg0] : tensor<?xi32>
  %3 = arith.index_cast %extracted : i32 to index
  %4 = tensor.extract %arg1[%3] : tensor<?xi32>
  %5 = arith.addi %4, %c1 : i32
  %inserted = tensor.insert %5 into %arg1[%3] : tensor<?xi32>
  ...
  scf.yield %inserted : tensor<?x32>
}

...

transform.sequence  failures(propagate) {
^bb0(%arg0: !transform.any_op):
  %0 = transform.structured.match ops{["arith.addi"]} in %arg0
  %1 = transform.get_parent_op %0 {op_name="scf.for"}
  %2:2 = transform.loop.tile_using_forall %1 num_threads = 8 {privatize_buffers=true}
}
```

Proposal (high-level idea)

Assume we want to add +1 to each element

`privatize_buffers:` **OFF**

(1 thread)

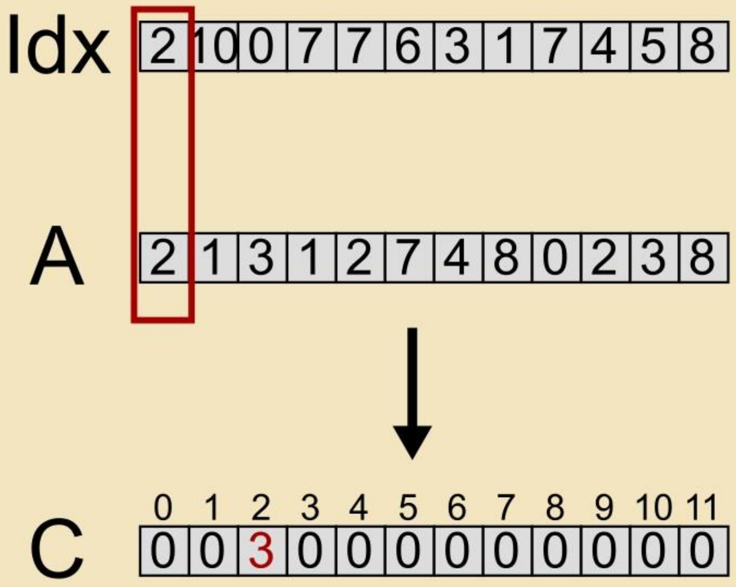


Proposal (high-level idea)

Assume we want to add +1 to each element

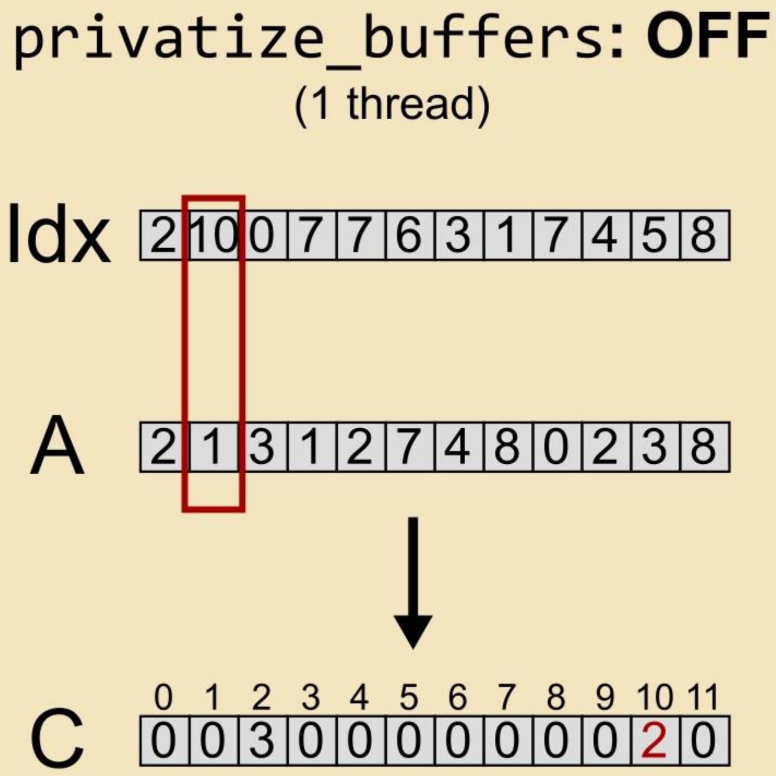
`privatize_buffers`: **OFF**

(1 thread)



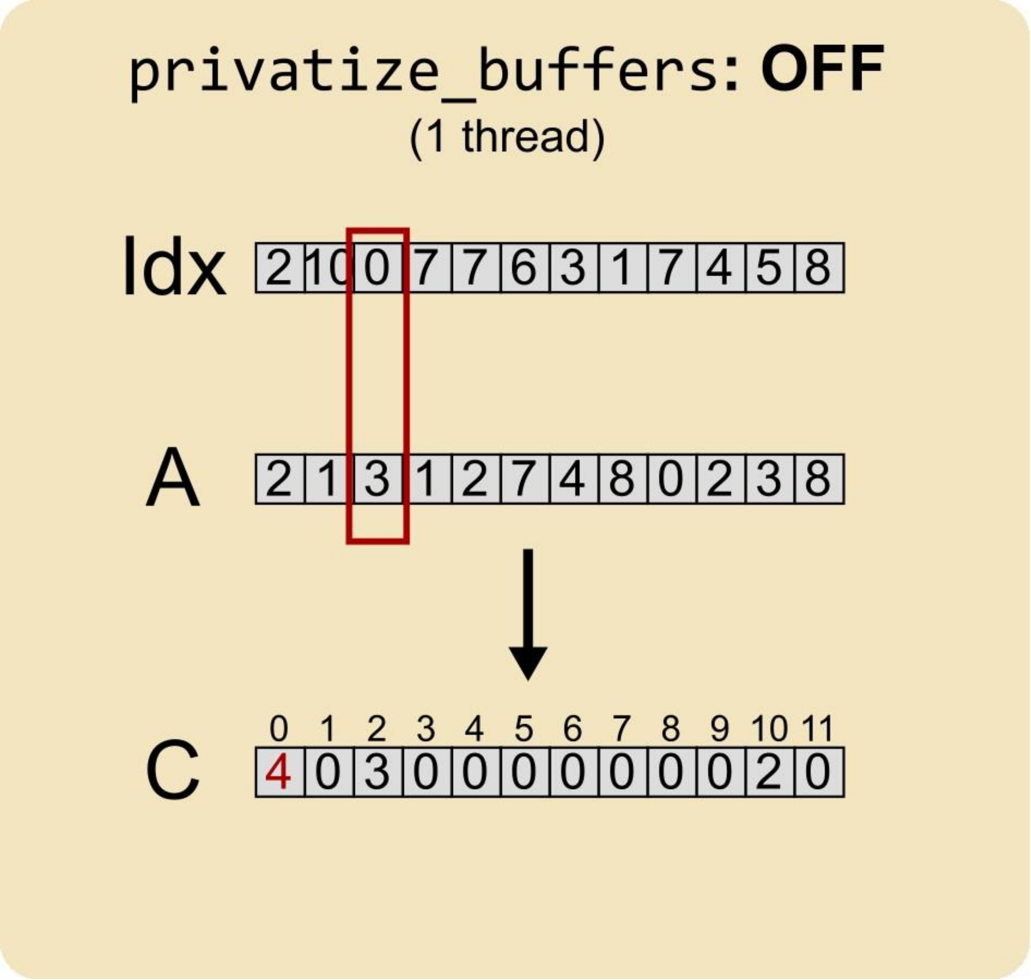
Proposal (high-level idea)

Assume we want to add +1 to each element



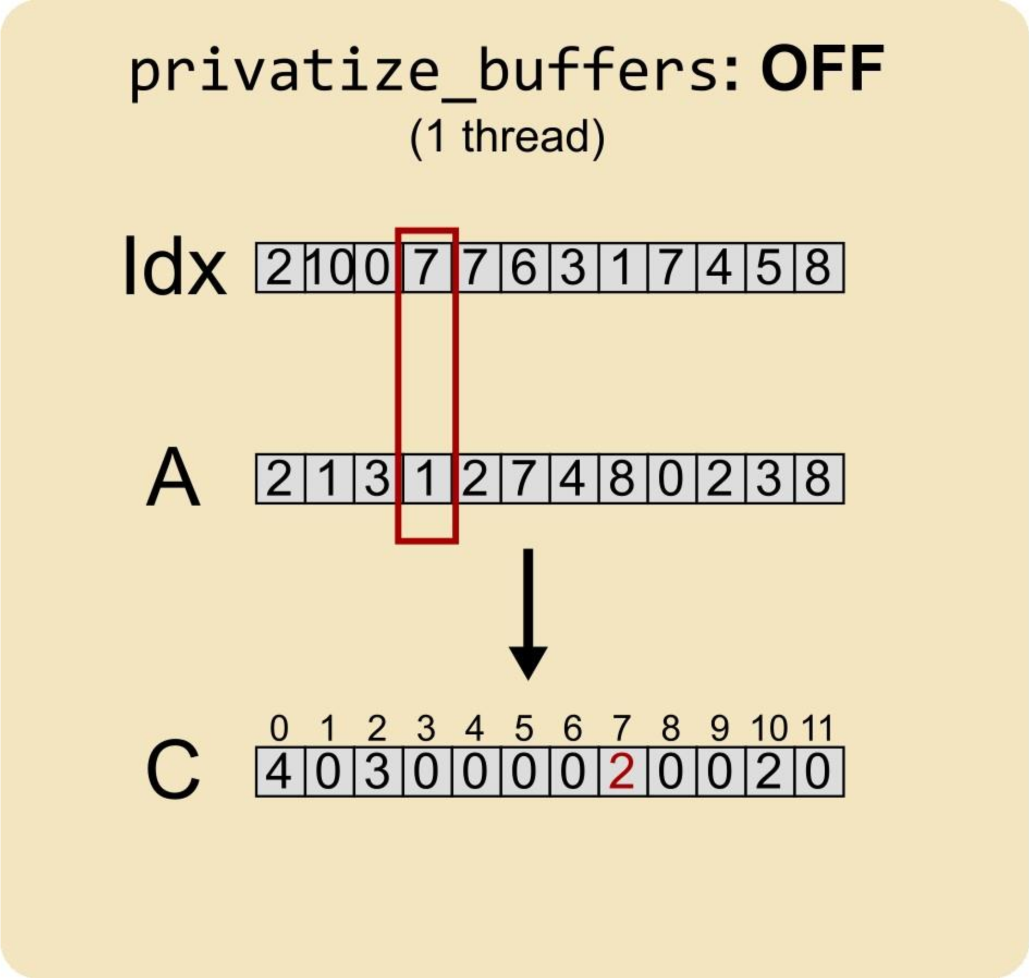
Proposal (high-level idea)

Assume we want to add +1 to each element



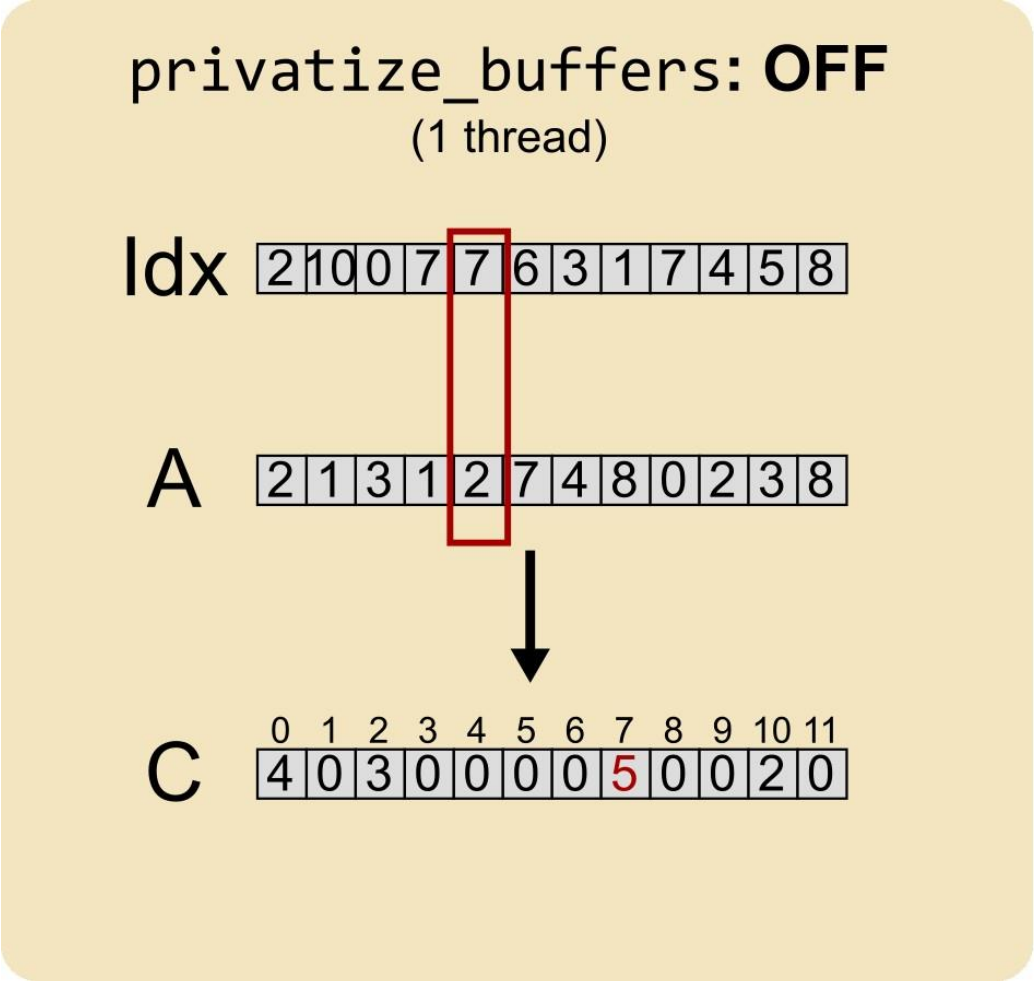
Proposal (high-level idea)

Assume we want to add +1 to each element



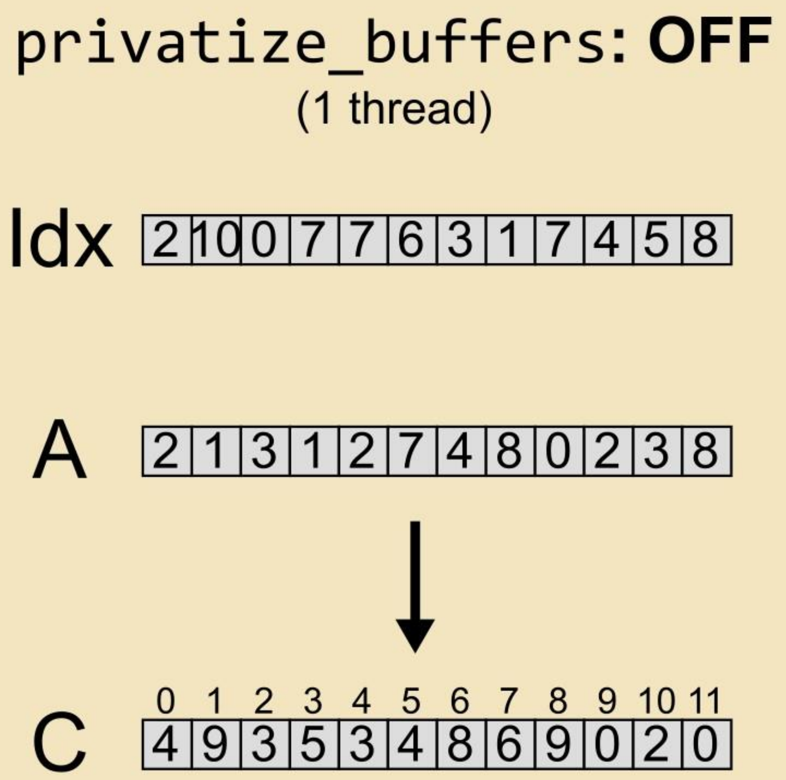
Proposal (high-level idea)

Assume we want to add +1 to each element



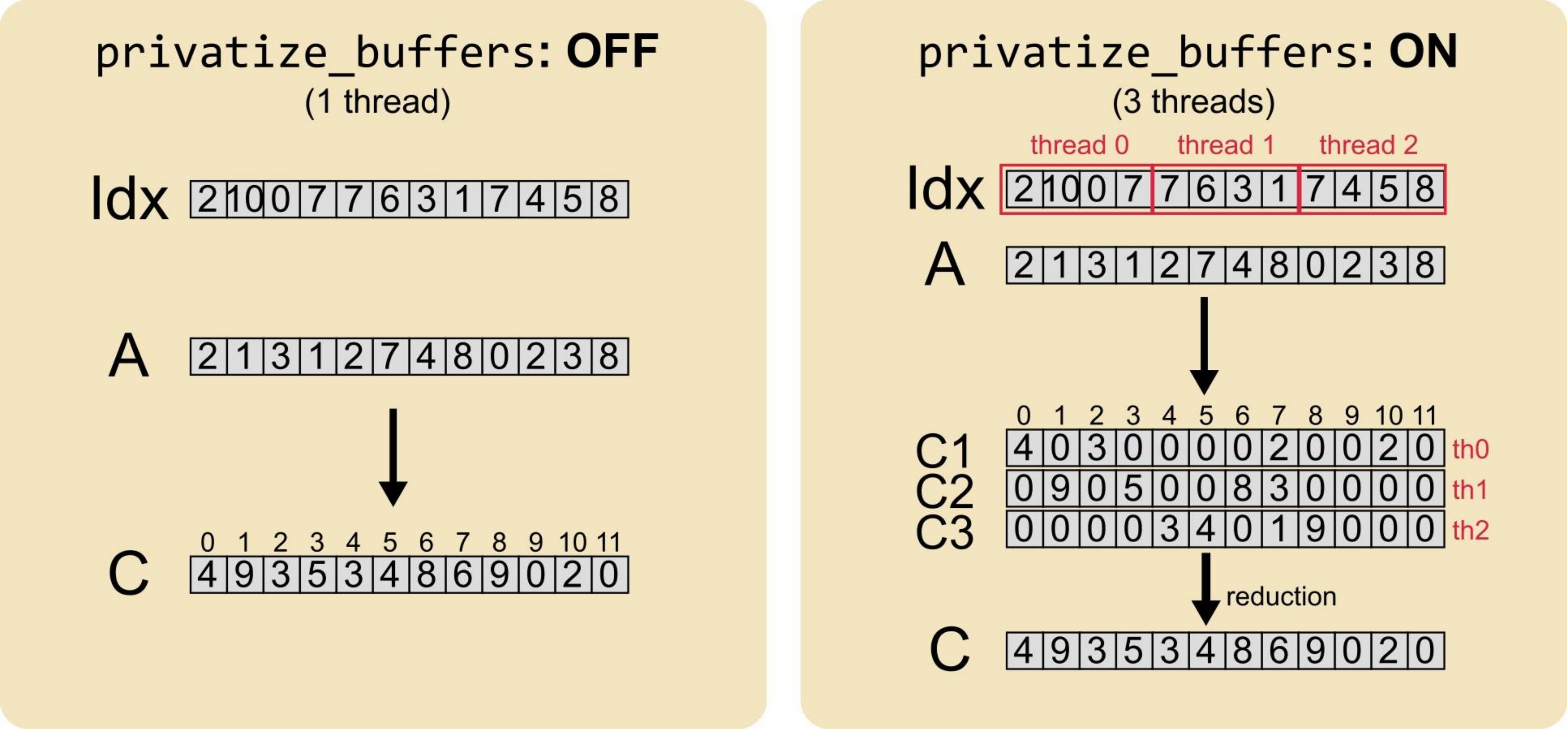
Proposal (high-level idea)

Assume we want to add +1 to each element



Proposal (high-level idea)

Assume we want to add +1 to each element



Proposal (code example)

1. Create new buffer (with a new dimension equal to the number of threads) to store thread-private data.

```
%d = tensor.dim %0, %c0 : tensor<?x32>
%1 = tensor.empty(%d) : tensor<8x?xi32>
%2 = scf.forall (%a1) in (8) shared_outs(%a2 = %1) -> (tensor<8x?xi32>) {
  %ex = tensor.extract_slice %a2[%a1,0][1,%d][1, 1] : tensor<8x?xi32> to tensor<?xi32>
  %3 = linalg.fill ins(%cst : f32) outs(%ex : tensor<?xi32>) -> tensor<?xi32>
  %4 = scf.for %a4 = %5 to %10 step %c1 iter_args(%a3 = %3) -> (tensor<?xi32>) {
    %extracted = tensor.extract %a3[...] : tensor<?xi32>
    ...
  }
  scf.forall.in_parallel { ... }
}
%red = linalg.reduce ins(%2: tensor<8x?xi32>) outs(%out: tensor<?xi32>) dimensions=[0]
  (%in: i32, %init: i32) {
    %5 = arith.addi %in, %init : i32
    linalg.yield %5 : i32
}
```

Proposal (code example)

2. Create `scf.forall`, move the loop body inside and create `scf.forall.in_parallel`

```
%d = tensor.dim %0, %c0 : tensor<?x32>
%1 = tensor.empty(%d) : tensor<8x?xi32>
%2 = scf.forall (%a1) in (8) shared_outs(%a2 = %1) -> (tensor<8x?xi32>) {
  %ex = tensor.extract_slice %a2[%a1,0][1,%d][1, 1] : tensor<8x?xi32> to tensor<?xi32>
  %3 = linalg.fill ins(%cst : f32) outs(%ex : tensor<?xi32>) -> tensor<?xi32>
  %4 = scf.for %a4 = %5 to %10 step %c1 iter_args(%a3 = %3) -> (tensor<?xi32>) {
    %extracted = tensor.extract %a3[...] : tensor<?xi32>
    ...
  }
  scf.forall.in_parallel { ... }
}
%red = linalg.reduce ins(%2: tensor<8x?xi32>) outs(%out: tensor<?xi32>) dimensions=[0]
  (%in: i32, %init: i32) {
    %5 = arith.addi %in, %init : i32
    linalg.yield %5 : i32
}
```

Proposal (code example)

3. Extract each thread-private slice from the new buffer (created in Step 1) and fill it with the identity element.

```
%d = tensor.dim %0, %c0 : tensor<?x32>
%1 = tensor.empty(%d) : tensor<8x?xi32>
%2 = scf.forall (%a1) in (8) shared_outs(%a2 = %1) -> (tensor<8x?xi32>) {
  %ex = tensor.extract_slice %a2[%a1,0][1,%d][1, 1] : tensor<8x?xi32> to tensor<?xi32>
  %3 = linalg.fill ins(%cst : f32) outs(%ex : tensor<?xi32>) -> tensor<?xi32>
  %4 = scf.for %a4 = %5 to %10 step %c1 iter_args(%a3 = %3) -> (tensor<?xi32>) {
    %extracted = tensor.extract %a3[...] : tensor<?xi32>
    ...
  }
  scf.forall.in_parallel { ... }
}
%red = linalg.reduce ins(%2: tensor<8x?xi32>) outs(%out: tensor<?xi32>) dimensions=[0]
  (%in: i32, %init: i32) {
    %5 = arith.addi %in, %init : i32
    linalg.yield %5 : i32
}
```

Proposal (code example)

4. Remap all the code to use the thread-private slice (%ex) instead of the global output (%out)

```
%d = tensor.dim %0, %c0 : tensor<?x32>
%1 = tensor.empty(%d) : tensor<8x?xi32>
%2 = scf.forall (%a1) in (8) shared_outs(%a2 = %1) -> (tensor<8x?xi32>) {
  %ex = tensor.extract_slice %a2[%a1,0][1,%d][1, 1] : tensor<8x?xi32> to tensor<?xi32>
  %3 = linalg.fill ins(%cst : f32) outs(%ex : tensor<?xi32>) -> tensor<?xi32>
  %4 = scf.for %a4 = %5 to %10 step %c1 iter_args(%a3 = %3) -> (tensor<?xi32>) {
    %extracted = tensor.extract %a3[...] : tensor<?xi32>
    ...
  }
  scf.forall.in_parallel { ... }
}
%red = linalg.reduce ins(%2: tensor<8x?xi32>) outs(%out: tensor<?xi32>) dimensions=[0]
  (%in: i32, %init: i32) {
    %5 = arith.addi %in, %init : i32
    linalg.yield %5 : i32
}
```

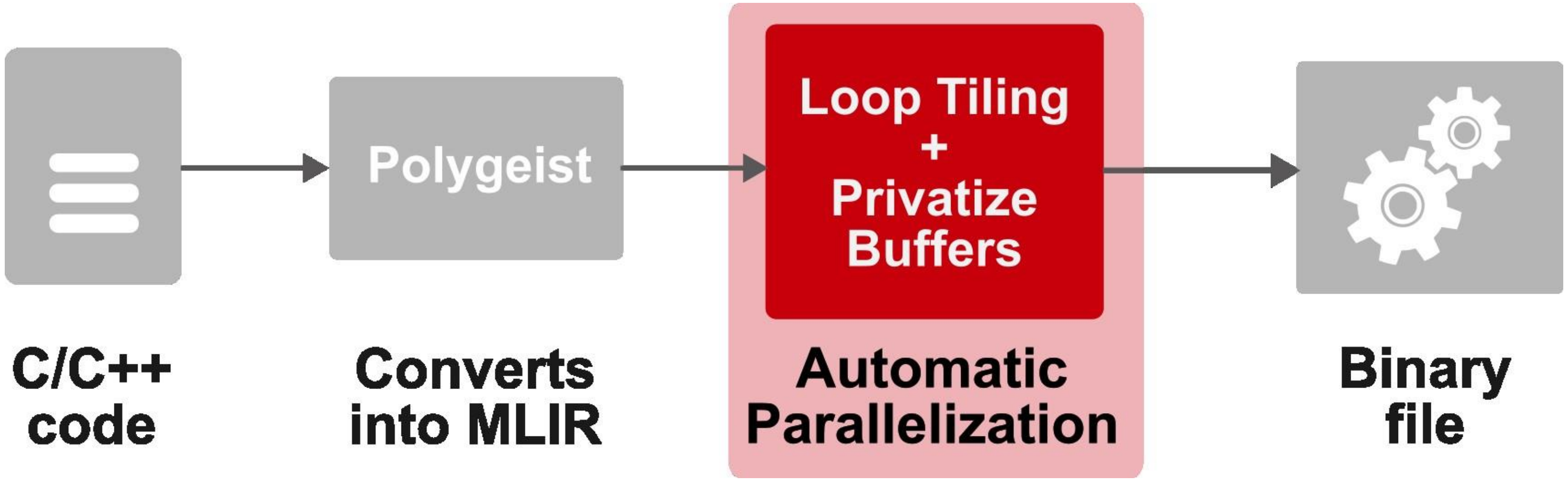
Proposal (code example)

5. Accumulate each thread-private result using a reduction at the end of the `scf.forall`

```
%d = tensor.dim %0, %c0 : tensor<?x32>
%1 = tensor.empty(%d) : tensor<8x?xi32>
%2 = scf.forall (%a1) in (8) shared_outs(%a2 = %1) -> (tensor<8x?xi32>) {
  %ex = tensor.extract_slice %a2[%a1,0][1,%d][1, 1] : tensor<8x?xi32> to tensor<?xi32>
  %3 = linalg.fill ins(%cst : f32) outs(%ex : tensor<?xi32>) -> tensor<?xi32>
  %4 = scf.for %a4 = %5 to %10 step %c1 iter_args(%a3 = %3) -> (tensor<?xi32>) {
    %extracted = tensor.extract %a3[...] : tensor<?xi32>
    ...
  }
  scf.forall.in_parallel { ... }
}
%red = linalg.reduce ins(%2: tensor<8x?xi32>) outs(%out: tensor<?xi32>) dimensions=[0]
  (%in: i32, %init: i32) {
    %5 = arith.addi %in, %init : i32
    linalg.yield %5 : i32
}
```

Evaluation

- Triangle Counting (**TC**): CRONO benchmark suite
- Betweenness Centrality (**BC**): CRONO benchmark suite
- Unsorted Segment Sum (**USS**): manual implementation



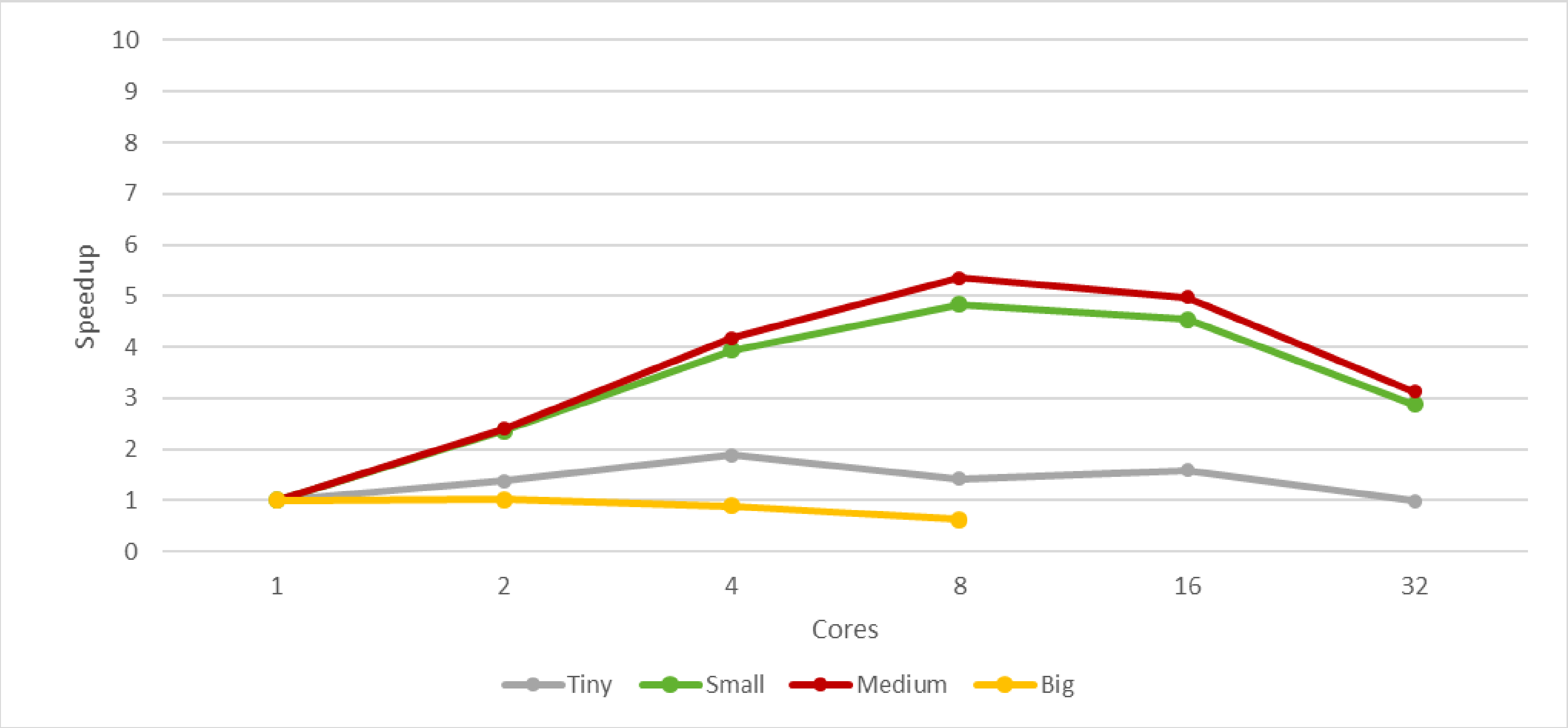
Test bed:

- Kunpeng 920 (32 cores, ARMv8, 2.6 GHz)
- 128 GB RAM DDR4 @ 2933 MHz

* The values shown are the average over 5 independent runs.

Evaluation

Results (Triangle Counting)

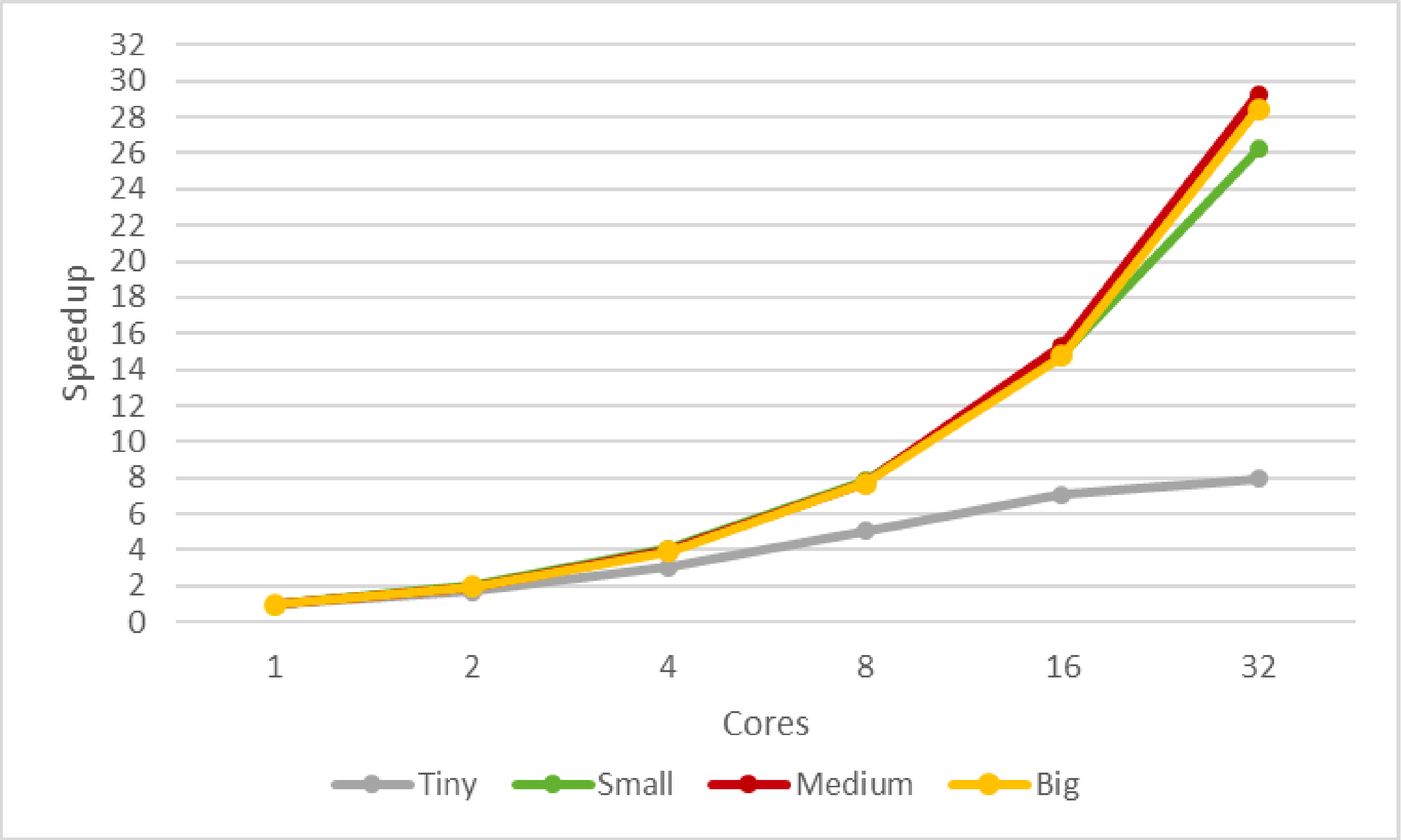


- *Big input on 16 and 32 cores give out-of-memory
- Huge buffers, so performance is mostly memory-bound
- Sweet spot around 8 threads

HUAWEI

Evaluation

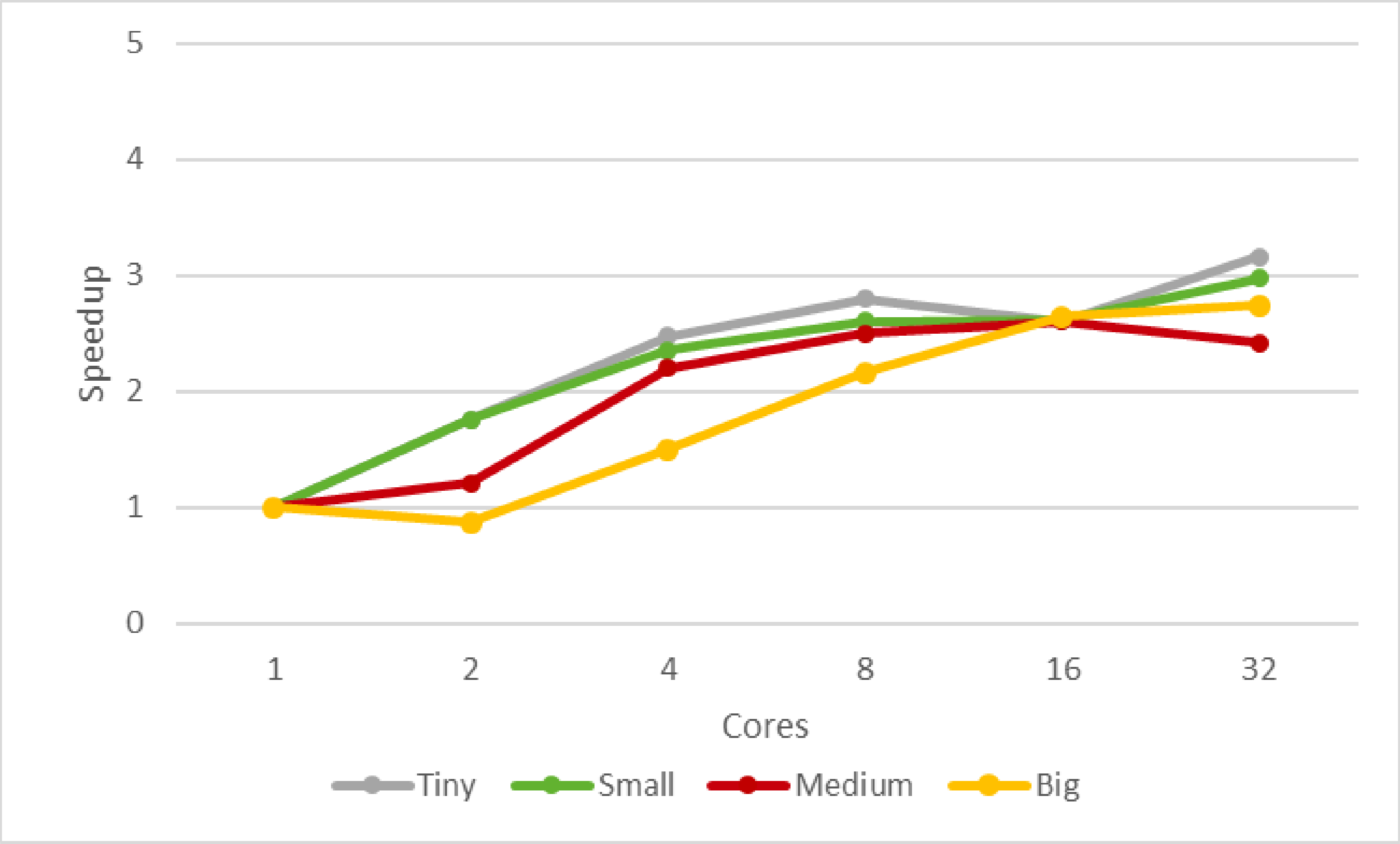
Results (Betweenness Centrality)



- Near optimal scalability!
- Very quick reduction, which helps achieving high speedups

Evaluation

Results (USS)



- Overall good scalability in all input sizes
- Big buffers, limited scalability compared to BC

Limitations

- **Does not support conditional writes:** Consider the following example (histogram):

```
const unsigned int idx = img[i];
if (histo[idx] < UINT8_MAX) {
  histo[idx]++;
}
```

loop tiling (with 4 threads) with privatized buffers generates:

```
linalg.reduce ins(%alloc : memref<4x?xi32>) outs(%arg2 : memref<?xi32>) dimensions = [0]
  (%in: i32, %init: i32) {
    %3 = arith.addi %in, %init : i32
    linalg.yield %3 : i32
}
```

which does not check if the accumulated value is greater than UINT8_MAX

Limitations

- **Memory usage is increased** (can be dangerous if original buffer is large, like in Triangle Counting).

For example, with 32 threads:

```
%alloc = memref.alloc(%dim) {alignment = 64 : i64} : memref<32x?xi32>
```

- **The reduction has a big impact in the overall execution time**

Possible solution: use `transform.structured.tile_reduction_using_forall`

Conclusions

- Loop tiling and buffer privatization can enable the automatic parallelization of indirect memory write programs.
- This approach has some limitations which we need to be aware of.
- Speedup varies depending on the size of the final reduction: with a light reduction we may achieve almost perfect scalability.

Ongoing / Future work:

- Add support for conditional writes
- Incorporate automatic parallelization of the reduction
- **Upstreaming!**

Thank you.

Questions?

2024 EURO LLVM
Vienna Austria
DEVELOPERS' MEETING

HUAWEI