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
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

- Unssorted Segment Sum
  Credit: Mindspore docs, Huawei
Challenges

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

```mlir
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:

1. %0 = scf.for %arg0 = %c0 to %1 step %c1 iter_args(%arg1 = %2) -> (tensor<?xi8>) {
2. %extracted = tensor.extract %indices %arg0 : tensor<?xi32>
3. %idx = arith.index_cast %extracted : i32 to index
4. %4 = tensor.extract %buff[ %idx ] : tensor<?xi32>
5. %5 = arith.addi %4, %c1 : i32
6. %inserted = tensor.insert %5 into %buff[ %idx ] : tensor<?xi32>
7. ...
8. scf.yield %inserted : tensor<?x32>
9. }

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
Proposal (high-level idea)

Assume we want to add +1 to each element

<table>
<thead>
<tr>
<th>privati*ze_buffers: OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 thread)</td>
</tr>
</tbody>
</table>

|Idx| 2 1 0 0 7 7 6 3 1 7 4 5 8 |

| A | 2 1 3 1 2 7 4 8 0 2 3 8 |

| C | 0 0 0 0 0 0 0 0 0 0 0 0 |

```markdown
<table>
<thead>
<tr>
<th>Idx</th>
<th>2 1 0 0 7 7 6 3 1 7 4 5 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2 1 3 1 2 7 4 8 0 2 3 8</td>
</tr>
<tr>
<td>C</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
```
Proposal (high-level idea)

Assume we want to add +1 to each element

```
privatize_buffers: OFF
(1 thread)
```

<table>
<thead>
<tr>
<th>Idx</th>
<th>2 1 0 0 7 7 6 3 1 7 4 5 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2 1 3 1 2 7 4 8 0 2 3 8</td>
</tr>
<tr>
<td>C</td>
<td>0 0 3 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Proposal (high-level idea)

Assume we want to add +1 to each element

```plaintext
privatize_buffers: OFF
(1 thread)

Idx: 2 1 0 0 7 7 6 3 1 7 4 5 8
A: 2 1 3 1 2 7 4 8 0 2 3 8
C: 0 0 3 0 0 0 0 0 0 0 0 2 0
```
Proposal (high-level idea)

Assume we want to add +1 to each element

 privatize_buffers: OFF
    (1 thread)

Idx  2 1 0 0 7 7 6 3 1 7 4 5 8

A   2 1 3 1 2 7 4 8 0 2 3 8

C   0 1 2 3 4 5 6 7 8 9 10 11

   4 0 3 0 0 0 0 0 0 0 2 0
Proposal (high-level idea)

Assume we want to add +1 to each element

```
privatize_buffers: OFF
(1 thread)

Idx 2 1 0 0 7 7 6 3 1 7 4 5 8
A 2 1 3 1 2 7 4 8 0 2 3 8
C 0 1 2 3 4 5 6 7 8 9 10 11
    4 0 3 0 0 0 0 2 0 0 2 0
```
Proposal (high-level idea)

Assume we want to add +1 to each element

```
privatize_buffers: OFF
(1 thread)
```

<table>
<thead>
<tr>
<th>Idx</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>7</th>
<th>7</th>
<th>6</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>4</th>
<th>5</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Proposal (high-level idea)

Assume we want to add +1 to each element.

<table>
<thead>
<tr>
<th>privatize_buffers: OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 thread)</td>
</tr>
</tbody>
</table>

| Idx | 2 1 0 0 7 7 6 3 1 7 4 5 8 |
| A   | 2 1 3 1 2 7 4 8 0 2 3 8   |
| C   | 0 1 2 3 4 5 6 7 8 9 10 11 |
|     | 4 9 3 5 3 4 8 6 9 0 2 0   |
Proposal (high-level idea)

Assume we want to add +1 to each element

privatizeBuffers: OFF
(1 thread)

\[
\begin{align*}
\text{Idx} & & 2 & 1 & 0 & 0 & 7 & 7 & 6 & 3 & 1 & 7 & 4 & 5 & 8 \\
\text{A} & & 2 & 1 & 3 & 1 & 2 & 7 & 4 & 8 & 0 & 2 & 3 & 8 \\
\text{C} & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
& & 4 & 9 & 3 & 5 & 3 & 4 & 8 & 6 & 9 & 0 & 2 & 0 
\end{align*}
\]

privatizeBuffers: ON
(3 threads)

\[
\begin{align*}
\text{Idx} & & 2 & 1 & 0 & 0 & 7 & 7 & 6 & 3 & 1 & 7 & 4 & 5 & 8 \\
\text{A} & & 2 & 1 & 3 & 1 & 2 & 7 & 4 & 8 & 0 & 2 & 3 & 8 \\
\text{C} & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
& & 4 & 9 & 3 & 5 & 3 & 4 & 8 & 6 & 9 & 0 & 2 & 0 
\end{align*}
\]
Proposal (code example)

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

```hls
%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<?x?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`

```plaintext
%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<?x?xi32>
    %3 = linalg.fill ins(%cst : f32) outs(%ex : tensor<?x?xi32>) -> tensor<?x?xi32>
    %4 = scf.for %a4 = %5 to %10 step %c1 iter_args(%a3 = %3) -> (tensor<?x?xi32>) {
        %extracted = tensor.extract %a3[...] : tensor<?x?xi32>
        ...
    }
    scf.forall.in_parallel { ... } }
}
%red = linalg.reduce ins(%2: tensor<8x?xi32>) outs(%out: tensor<?x?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.

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

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

```plaintext
%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<?x?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?ixi32>) 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

```plaintext
%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<8x?xi32>
  %3 = linalg.fill ins(%cst : f32) outs(%ex : tensor<?x32>) -> tensor<?x32>
  %4 = scf.for %a4 = %5 to %10 step %c1 iter_args(%a3 = %3) -> (tensor<?x32>) {
    %extracted = tensor.extract %a3[...] : tensor<?x32>
    ...
  }
  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.
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
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):

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

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

```c
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:

```c
%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?