SDIR Design

Berke Ates

23.06.2021

Contents

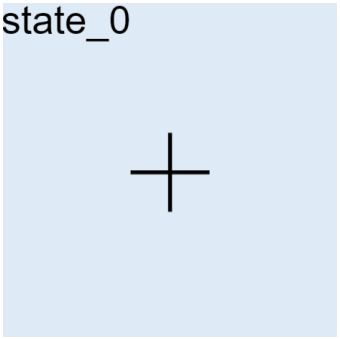
1 Preface

This document explains the representation of SDFG concepts in a MLIR dialect called SDIR (Stateful Dataflow Intermediate Representation). The goals were to constrain the supported SDFGs as little as possible, to stay as MLIR-esque as possible (to enable the use of MLIR tools) and to be as lossless as possible (a round-trip should result in the provided input). Keep in mind, that the shown code snippets have been adjusted to focus on the corresponding concept and to save space. Especially op types have been mostly omitted. `%0` and `%1` are used as constant zeros and ones. Comments regarding errors are welcome.

This is a work in progress. There are no correctness or completeness guarantees. Any liability will be declined.

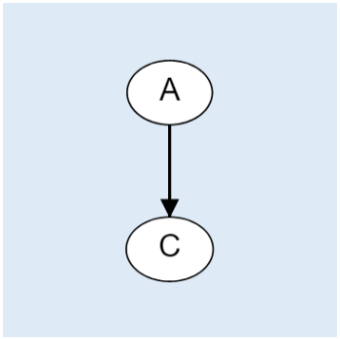
2 Single state

States are represented by an SSACFG region with the dialect-specific keyword `sdir.state`. See Figure 1. The state body contains all the runnable subgraphs. States need to use dialect ops for reading/writing global arrays and can therefore not access them directly. The op `sdir.get_access` is used to achieve this. This op is exclusive for the state region and may not be used by sub-regions. See Figure 2.




```
sdir.state @state_0{
    //state body
}
```

Figure 1: Single state




```
sdir.state @state_0{
    %a = sdir.get_access %A
            : !sdir.memlet<i32>
    %c = sdir.get_access %C
            : !sdir.memlet<i32>

    sdir.copy %a -> %c
}
```

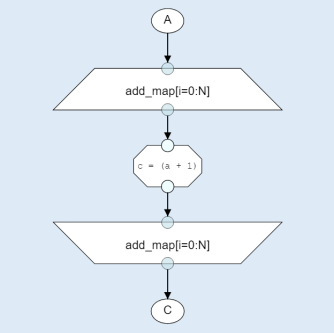
Figure 2: State input and output

2.1 Identifiers, keywords and types

SDIR introduces only one additional identifier with the symbol `$` (see Section 2.1.1).
Every op and region in SDIR is prefixed by the keyword `sdir` followed by a dot to clearly mark the end of the prefix. Some examples are: `sdir.tasklet{}`, `sdir.return{}`, `sdir.alloc()`.
SDIR contains only one additional type on top of the builtin types: `!sdir.memlet<>`. See Section 2.2.

2.1.1 Symbols

The op `sdir.alloc_symbol` creates a new symbol, but does not return anything. In order to access symbols the identifier `$()` can be used. See Figure 3. One can describe arithmetic expressions inside the identifier. Example: `$(2*N - 1)`. Parentheses can be omitted for a single symbol. `$N` is the same as `$(N)`.




```
sdir.alloc_symbol("N")

sdir.map $i = 0 to $N step 1 {
    %a = sdir.load %A[$i]
    %c = sdir.call @add_one(%a)
    sdir.store %c, %C[$i]
}
```

Figure 3: Symbol used for range in map

2.1.2 Symbolic sizes

Symbols (see Section 2.1.1) can be used to define the size of an `sdir.memlet`. One simply replaces the constants in the type by the symbol identifier `$()`.
Example: `%A = sdir.alloc() : !sdir.memlet<$(2*N)xi32>`.

2.1.3 Write conflict resolution

In some cases an op might write to a location that contains the result of an other op. To resolve this conflict any op that writes to a location has an optional attribute called `wcr`. Examples: `sdir.copy{wcr="add"} %a -> %c` or `sdir.store{wcr="max"} %1, %a : i32`. "overwrite" is the default if no `wcr` is provided. A function may be provided as well. The function must have the signature `old, new -> resolved` and the types must match the memlet types. See Figure 4.

```
sdir.func @add(%old: i32, %new: i32) -> i32{
    %res = sdir.addi %old, %1 : i32
    sdir.return %res : i32
}

sdir.state @state_0{
    %a = sdir.get_access %A : !sdir.memlet<i32>
    %c = sdir.get_access %C : !sdir.memlet<i32>
    sdir.copy{wcr=@add} %a -> %c
}
```

Figure 4: Write conflict resolution with custom function

2.1.4 Multiple basic blocks and branching

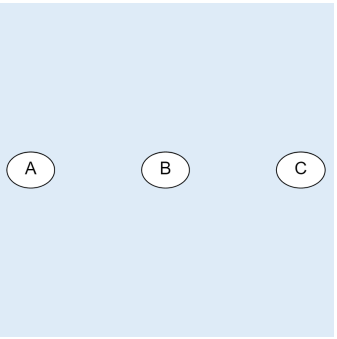
Tasklets (see Section 2.3) and `sdir.func` are the only regions that may have multiple basic blocks and branching. Every other region has only one basic block and therefore terminator operations can be omitted. SDFG regions (see Section 3) contain branches, but these are provided as attributes and therefore there is no need for branching with `sdir.br` or `sdir.cond_br`.

2.1.5 Unnamed outputs

Any output that is unnamed will be auto-generated with the prefix `__` (double underscore). Therefore explicitly naming any output with the same prefix is strictly forbidden for the user.

2.2 Memlets

Memlets are represented by using `sdir.memlet`, which extends the builtin `memref` to allow for attributes. See Figure 2. Using index and layout maps on `sdir.memlets` any subregion of the underlying array can be selected. Transient arrays get constructed using `sdir.alloc_transient()` as opposed to `sdir.alloc()` for global arrays. Arrays created inside of a state are only accessible by the same state whereas arrays created outside of all states are accessible by all states. See Figure 5.




```
%A = sdir.alloc_transient()
                : !sdir.memlet<16xi32>
%B = sdir.alloc_transient()
                : !sdir.memlet<16xi32>
%C = sdir.alloc_transient()
                : !sdir.memlet<16xi32>
```

Figure 5: Transient arrays

2.2.1 Parameterized sizes

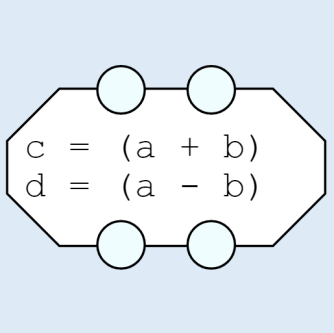
One can also pass parameters to `sdir.alloc()` to define the size at runtime as seen on here: `%A = sdir.alloc(%n) : !sdir.memlet<?xi32>`. In this case the question mark `?` gets replaced by the value of `%n`.

2.3 Tasklets

Tasklets are defined as pure functions working on primitive types or `sdir.memlets`. See Figure 6. In order to clearly mark the function as a tasklet the keyword `func` gets replaced by the dialect-specific keyword `sdir.tasklet`. In order to solve the problem of naming the outputs, an attribute gets added, which lists the output names in the order they get returned. This allows tasklets to work on primitive types directly and restricts reading from outputs or writing to inputs by default.

2.3.1 Language limitation

Tasklets must be written in MLIR despite SDFGs supporting many more (Python, C++, OpenCL, SystemVerilog). Currently there are no plans to support the full range of languages as that would complicate compilation quite a bit.




```
sdir.tasklet{outputs=["c", "d"]} @add(
    %a: i32, %b: i32) -> (i32, i32)
{
    %c = sdir.addi %a, %b : i32
    %d = sdir.subi %a, %b : i32
    sdir.return %c, %d : i32, i32
}
```

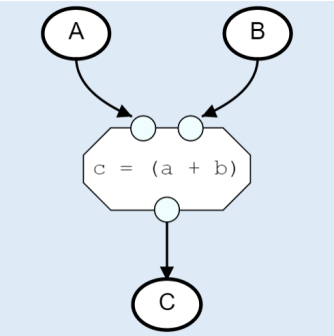
Figure 6: Tasklet

2.3.2 Calling tasklets

To call a tasklet an op called sdir.call gets added. It works similar to std.call but syntactically an arrow is used to separate the inputs and outputs instead of an assignment, since data is written to an sdir.memlet and not assigned to a new variable. This makes the distinction clear and as an added benefit visually shows the flow of data. Example: %c = sdir.call @add(%a, %b) .

2.3.3 Example SDFG

By combining all the concepts so far, one can represent simple SDFGs in SDIR. Such a simple SDFG is shown in Figure 7, which simply adds A and B and writes the result to C.

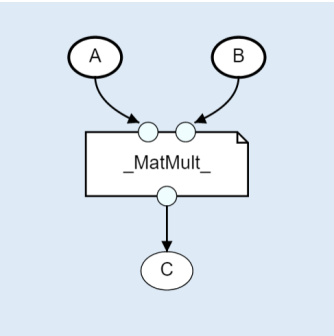



```
sdir.tasklet @add(%a: i32, %b: i32) -> i32{
    %c = sdir.addi %a, %b : i32
    sdir.return %c
}

sdir.state @state_0{
    %a = sdir.get_access %A
    %b = sdir.get_access %B
    %c = sdir.get_access %C

    %c = sdir.call @add(%a, %b)
}
```

Figure 7: Simple SDFG

2.4 Library nodes

Since a lossless translation is desired in order to translate back and forth, library nodes don't get expanded before translation. An op called `sdir.libcall` gets introduced. It works like `sdir.call` but calls a library function instead. The name of the function matches the specific library function name.

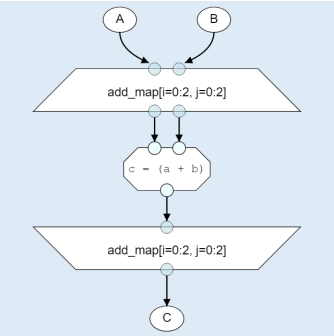



```
sdir.state @state_0{
    %a = sdir.get_access %A
    %b = sdir.get_access %B
    %c = sdir.get_access %C

    %c_t = sdir.libcall "dace.libraries
             .blas.nodes.Gemm" (%a, %b)
    sdir.store %c_t, %c
}
```

Figure 8: Library call

2.5 Map scopes

Maps get represented by a region labeled by `sdir.map`. A similar syntax to `affine.parallel` is being used. The region of the map has access to the `sdir.memlet` of the enclosing state and therefore can directly access them. Symbols are used for the map variables in order to ensure that they do not get edited in the map region as well as provide a way to perform simple arithmetic in a compact way. See Figure 9.




```
sdir.map ($i, $j) =
    (0, 0) to (2, 2) step (1, 1)
{
    %a = sdir.load %A[$i, $j]
    %b = sdir.load %B[$i, $j]
    %c = sdir.call @add(%a, %b)
    sdir.store %c, %C[$i, $j]
}
```

Figure 9: Map scope

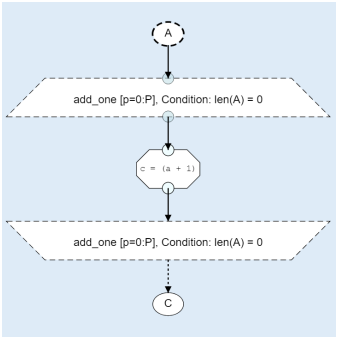
2.6 Streams and consume scopes

Streams use the type `sdir.stream`, which works similar to `sdir.memlet`, but it introduces two ops `sdir.stream_push` and `sdir.stream_pop`. To create streams one can use `sdir.alloc_stream()` for global and `sdir.alloc_transient_stream()` for transient streams, just like `sdir.memlet`. Since reading or writing directly to streams is forbidden, `sdir.stream` doesn't have reading or writing ops apart from push and pop. See Figure 10.

```
%A = sdir.alloc_stream() : !sdir.stream<i32>
%42 = sdir.constant 42 : i32
sdir.stream_push %42, %A : i32
%42_p = sdir.stream_pop %A : i32
```

Figure 10: Stream

Consume scopes are composed similarly to maps. However instead of ranged one needs to define the number of processing elements and the stopping condition with the attributes num_pes and condition . The condition function must have the signature stream -> boolean . The consume operation stops when the stream is empty. The first basic block gets called with the id of the processing element and the element retrieved by sdir.stream_pop . See Figure 11.




```
%A = sdir.alloc_stream()
                : !sdir.stream<i32>

sdir.func @empty(%A : !sdir.stream<i32>)
                                    -> i1
{
    %l = sdir.stream_length %A : i32
    %isZero = sdir.cmpi "eq", %l, %0 : i32
    sdir.cond_br %isZero, ^zero, ^one

    ^zero: sdir.return %1 : i1
    ^one: sdir.return %0 : i1
}

sdir.consume{num_pes=$P, condition=@empty}
    (%A: !sdir.stream<i32>)
{
    ^bb0(%p: i32, %a: i32):
        %c = sdir.call @add_one(%a)
        sdir.store{wcr="add"} %c, %C[%0]
}
```

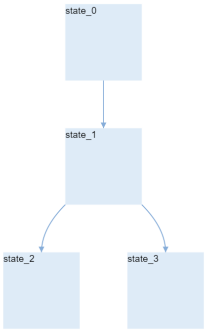
Figure 11: Consume scope

2.7 View nodes

A view node simply changes the view on a memory location. In SDIR this is the same as recasting a sdir.memlet . One may be inclined to use sdir.memlet_cast similar to memref_cast in the builtin dialect. However one might need sdir.memlet_cast outside of view nodes (such as selecting a subsection of the input-arrays). To clearly mark these casts as view nodes one instead uses the keyword sdir.view_cast , which works exactly like sdir.memlet_cast .

3 Multiple states

For multiple states a builtin graph region is used as states can form a cyclic graph. Transitions have either assignments or conditions. Both are implemented with the sdir.edge op using the assign and condition attributes. See Figure 12.




```
sdir.sdfg{entry=@state_0} {
    sdir.edge{assign=["i = 1"]}
        @state_0 -> @state_1
    sdir.edge{condition="i > 1"}
        @state_1 -> @state_2
    sdir.edge{condition="i <= 1"}
        @state_1 -> @state_3
}
```

Figure 12: Multiple states

3.1 Entry state

In order to run an SDFG with multiple states an entry state must be chosen. This is achieved by marking the sdfg with the attribute {entry=stateLabel} . See Figure 12.

3.2 Nested SDFGs

With the presented concepts so far any flat SDFG can be represented. In order to nest SDFGs one needs to give the SDFGs inputs/outputs and a name. After doing so, one can treat a nested SDFG like a tasklet.

```
sdir.sdfg{entry=@state_0, outputs=["C"]} @name(%A: !sdir.memlet<i32>,
                                               %C: !sdir.memlet<i32>)
{
    sdir.edge{assign=["i = 1"]} @state_0 -> @state_1
    sdir.edge{condition="i > 1"} @state_1 -> @state_2
    sdir.edge{condition="i <= 1"} @state_1 -> @state_3
}
```

Figure 13: Nested SDFG

4 File compatibility

In order to maintain metadata that is present in the SDFG every SDIR element has attributes, that describe the said metadata. However only non-default values are saved. The name of the attributes must match the names in the SDFG. See Figure 14.

```
sdir.state{nosync=false, instrument="No_Instrumentation"} @state_0{
    %a = sdir.get_access %A : !sdir.memlet<i32>
    %c = sdir.get_access %C : !sdir.memlet<i32>

    sdir.copy{dynamic=false, allow_oob=false} %a -> %c
}
```

Figure 14: Attributes for compatibility