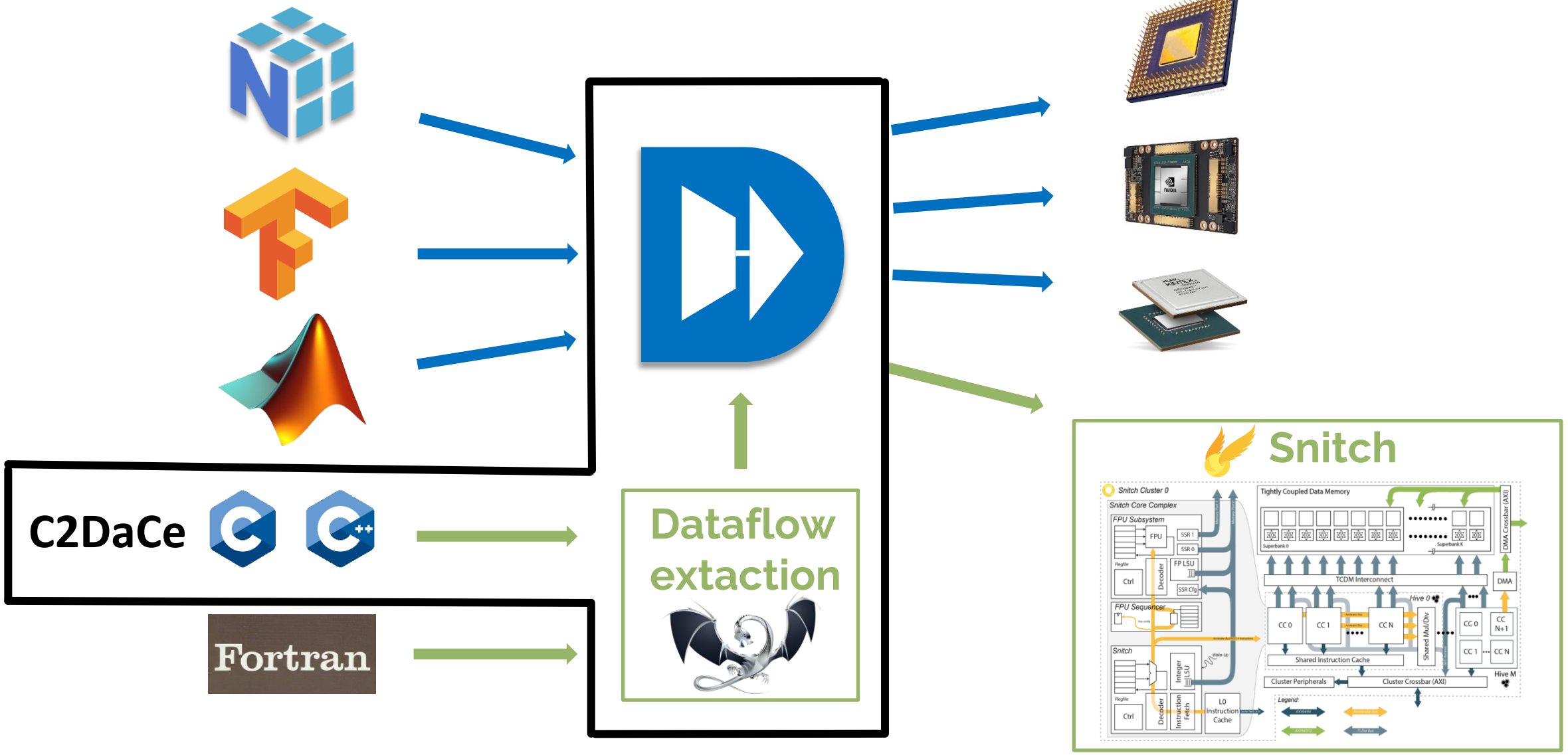
Alexandru Calotoiu

DaFlEx

DaFlEx – goals



C2DaCe

- **Paper submitted**
- **Initial prototype ready**
- **Lots of opportunity for expansion**
- **Next steps:**
  - Upload to Arxiv
  - Making the code available on Github

C2DaCe workflow

First parse the input file(s) into the clang AST:

```
import clang.cindex
from clang.cindex import Cursor, CursorKind, TypeKind, TokenKind
```

```
index = clang.cindex.Index.create()
print("parsing...")
tu = index.parse(filename)
```

Documentation is severely limited, so a lot of AST analysis is reversed engineering
Biggest issue: Clang AST is read-only

We wish to make data-centric friendly changes (transformation passes) on the AST
Solution: We create our own AST, templated after the python AST parser, visitor & transformer

```
own_ast = create_own_ast(tu.cursor, files)
```

C2DaCe workflow

```python
def create_own_ast(cnode, files):
    current_file = cnode.location.file

    if current_file is not None and str(current_file) not in files:
        return
    if cnode.kind in supported_cursor_kinds:
        node = supported_cursor_kinds[cnode.kind](cnode, files)
        if "type" in node._attributes and not hasattr(node, "type"):
            node.type = get_c_type_from_clang_type(cnode.type)

        if hasattr(node, "type"):
            if isinstance(node, TypeRef):
                return node.type
            if not isinstance(node.type, Type):
                a = 1
            assert isinstance(node.type, Type)

        return node

    elif cnode.kind in ignored_cursor_kinds:
        print(UserWarning("Ignored cursor kind", cnode.kind))
        return
    elif cnode.kind in unsupported_cursor_kinds:
        print(Warning(f"Unsupported {cnode.kind} at {cnode.location.line}"))
        return
    raise Exception("How did you even get here?")
```

What's a supported cursor kind?

```python
supported_cursor_kinds = {
    clang.cindex.CursorKind.TRANSLATION_UNIT: translation_unit,
    clang.cindex.CursorKind.TYPEDEF_DECL: typedef_decl,
    clang.cindex.CursorKind.TYPE_REF: type_ref,
    clang.cindex.CursorKind.FUNCTION_DECL: func_decl,
    clang.cindex.CursorKind.PARM_DECL: parm_decl,
    clang.cindex.CursorKind.COMPOUND_STMT: basic_block,
    clang.cindex.CursorKind.DECL_STMT: decl_stmt,
    clang.cindex.CursorKind.VAR_DECL: var_decl,
    clang.cindex.CursorKind.CXX_BOOL_LITERAL_EXPR: cxx_bool_literal,
    clang.cindex.CursorKind.INTEGER_LITERAL: int_literal,
    clang.cindex.CursorKind.FLOATING_LITERAL: float_literal,
    clang.cindex.CursorKind.STRING_LITERAL: string_literal,
    clang.cindex.CursorKind.CHARACTER_LITERAL: char_literal,
    clang.cindex.CursorKind.BINARY_OPERATOR: bin_op,
    clang.cindex.CursorKind.DECL_REF_EXPR: decl_ref_expr,
    clang.cindex.CursorKind.ARRAY_SUBSCRIPT_EXPR: array_subscript_expr,
    clang.cindex.CursorKind.UNEXPOSED_EXPR: unexposed_expr,
    clang.cindex.CursorKind.UNEXPOSED_DECL: unexposed_decl,
    clang.cindex.CursorKind.INIT_LIST_EXPR: init_list_expr,
    clang.cindex.CursorKind.UNARY_OPERATOR: unary_op,
    clang.cindex.CursorKind.FOR_STMT: for_stmt,
    clang.cindex.CursorKind.IF_STMT: if_stmt,
    clang.cindex.CursorKind.DO_STMT: do_stmt,
```

And many more…

C2DaCe workflow

What are the objects?

```
class ForStmt(Statement):
    _attributes = ()
    _fields = (
        'init',
        'cond',
        'body',
        'iter',
    )
```

How does the translation work?

```
def for_stmt(cnode, files):
    for_list = list(cnode.get_children())
    #print(cnode.location.line)
    return ForStmt(
        init=[create_own_ast(for_list[0], files)],
        cond=[create_own_ast(for_list[1], files)],
        iter=[create_own_ast(for_list[2], files)],
        body=[create_own_ast(for_list[3], files)],
        lineno=cnode.location.line,
    )
```

Novel – no
Difficult – no
Necessary – yes!
Why? – Transformations!

C2DaCe - Transforms

Some are canonicalization
Some are data-flow friendly
Some are SDFG friendly

```
transformations = [
    InsertMissingBasicBlocks,
    CXXClassToStruct,
    FlattenStructs,
    ReplaceStructDeclStatements,
    UnaryReferenceAndPointerRemover,
    CondExtractor,
    UnaryExtractor,
    UnaryToBinary,
    CallExtractor,
    MoveReturnValueToArguments,
    CompoundToBinary,
    IndicesExtractor,
    InitExtractor,
    ForDeclarer,
]

for transformation in transformations:
    changed_ast = transformation().visit(changed_ast)
```

C2DaCe Condition Extractor

Why? Because conditions must be symbolic expressions in SDFGs!

```python
class CondExtractorNodeLister(NodeVisitor):
    def __init__(self):
        self.nodes: List[Node] = []

    def visit_ForStmt(self, node: ForStmt):
        return

    def visit_IfStmt(self, node: IfStmt):
        self.nodes.append(node.cond[0])

    def visit_BasicBlock(self, node: BasicBlock):
        return
```

```python
class CondExtractor(NodeTransformer):
    def __init__(self, count=0):
        self.count = count

    def visit_IfStmt(self, node: IfStmt):

        if not hasattr(self, "count"):
            self.count = 0
        else:
            self.count = self.count + 1
        tmp = self.count

        cond = [
            BinOp(op="!=",
                  lvalue=DeclRefExpr(name="tmp_if_" + str(tmp - 1)),
                  rvalue=IntLiteral(value="0"))
        ]
        body_if = [self.visit(node.body_if[0])]
        if hasattr(node, "body_else"):
            body_else = [self.visit(node.body_else[0])]
            return IfStmt(cond=cond, body_if=body_if, body_else=body_else)
        else:
            return IfStmt(cond=cond, body_if=body_if)

    def visit_BasicBlock(self, node: BasicBlock):
        newbody = []

        for child in node.body:
            lister = CondExtractorNodeLister()
            lister.visit(child)
            res = lister.nodes
            temp = self.count
            if res is not None:
                for i in range(0, len(res)):
                    newbody.append(
                        DeclStmt(vardecl=[
                            VarDecl(name="tmp_if_" + str(temp), type=Int())
                        ]))
                    newbody.append(
                        BinOp(op="=",
                              lvalue=DeclRefExpr(name="tmp_if_" + str(temp)),
                              rvalue=res[i]))
            newbody.append(self.visit(child))

        return BasicBlock(body=newbody)
```

C2DaCe – from AST to SDFG

```
#sdfg = SDFG("_" + filecore + "_inner")
globalsdfg = SDFG("_" + filecore)
globalsdfg.add_symbol("_argcount", dace.int32)
name_mapping = NameMap()
name_mapping[globalsdfg]["argv_loc"] = "argv_loc"
name_mapping[globalsdfg]["argc_loc"] = "argc_loc"
name_mapping[globalsdfg]["c2d_retval"] = "c2d_retval"

globalsdfg.add_array(name_mapping[globalsdfg]["argv_loc"], ['_argcount'],
                     dace.int8,
                     transient=False)
globalsdfg.add_scalar(name_mapping[globalsdfg]["argc_loc"],
                      dace.int32,
                      transient=False)
globalsdfg.add_scalar(name_mapping[globalsdfg]["c2d_retval"],
                      dace.int32,
                      transient=True)
last_call_expression = [
    DeclRefExpr(name="argc_loc"),
    DeclRefExpr(name="argv_loc"),
    DeclRefExpr(name="c2d_retval")
]
translator = AST2SDFG(last_call_expression, globalsdfg, "main",
                      name_mapping)

translator.translate(changed_ast, globalsdfg)
```

C2DaCe - from AST to SDFG

```
self.ast_elements = {
    WhileStmt: self.while2sdfg,
    DoStmt: self.do2sdfg,
    RetStmt: self.ret2sdfg,
    IfStmt: self.ifstmt2sdfg,
    ForStmt: self.forstmt2sdfg,
    BasicBlock: self.basicblock2sdfg,
    FuncDecl: self.funcdecl2sdfg,
    BinOp: self.binop2sdfg,
    DeclStmt: self.declstmt2sdfg,
    VarDecl: self.vardecl2sdfg,
    ParmDecl: self.parmdecl2sdfg,
    TypeDecl: self.typedecl2sdfg,
    CallExpr: self.call2sdfg,
    BreakStmt: self.break2sdfg,
    ContinueStmt: self.continue2sdfg,
    AST: self.tu2sdfg
```

```
self.last_call_expression = {globalsdfg: last_call_expression}
```

```
self.libraries["printf"] = "print"
self.libraries["fprintf"] = "print"
self.libstates = ["print"]
```

C2DaCe – Functions context change

```python
def funcdecl2sdfg(self, node: FuncDecl, sdfg: SDFG):
    print("FUNC: ", node.name)
    if node.body is None:
        print("Empty function")
        return
    used_vars = [
        node for node in walk(node.body) if isinstance(node, DeclRefExpr)
    ]
    binop_nodes = [
        node for node in walk(node.body) if isinstance(node, BinOp)
    ]
    write_nodes = [node for node in binop_nodes if node.op == "="]
    write_vars = [node.lvalue for node in write_nodes]
    read_vars = copy.deepcopy(used_vars)
    for i in write_vars:
        if i in read_vars:
            read_vars.remove(i)
    write_vars = remove_duplicates(write_vars)
    read_vars = remove_duplicates(read_vars)
    used_vars = remove_duplicates(used_vars)
    write_names = []
    read_names = []
    for i in write_vars:
        write_names.append(i.name)
    for i in read_vars:
        read_names.append(i.name)
```

```python
internal_sdfg = substate.add_nested_sdfg(new_sdfg,
                                         sdfg,
                                         ins_in_new_sdfg,
                                         outs_in_new_sdfg,
                                         symbol_mapping=sym_dict)
```

```python
    memlet = generate_memlet(i, sdfg, self)
if local_name.name in write_names:
    add_memlet_write(substate, mapped_name, internal_sdfg,
                     self.name_mapping[new_sdfg][local_name.name],
                     memlet)
if local_name.name in read_names:
    add_memlet_read(substate, mapped_name, internal_sdfg,
                    self.name_mapping[new_sdfg][local_name.name],
                    memlet)
```

The most interesting bits!