Alexandru Calotoiu, Tal Ben-Nun, Grzegorz Kwasniewski, Johannes de Fine Licht,
Timo Schneider, Philipp Schaad, Torsten Hoefler

DaFlEx
Year 1
Report

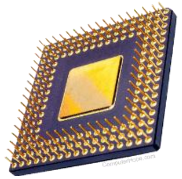
SPCL

DaFlEx – first year goal

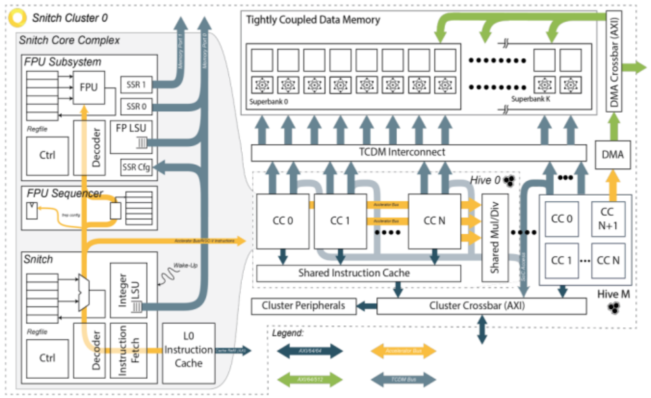
C2DaCe

Fortran

Dataflow extaction



Snitch



C2DaCe

```c
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000 + 0],
                             double B[2000 + 0])
{
  int t, i;

  for (t = 0; t < tsteps; t++)
  {

    for (i = 1; i < n - 1; i++)
    {
      B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1])
    }
    for (i = 1; i < n - 1; i++)
    {
      A[i] = 0.33333 * (B[i - 1] + B[i] + B[i + 1])
    }
  }
}
```

```c
inline void kernel_jacobi_1d_1_0_2(_jacobi_1d_t *__state, double* dace_A_3, double* dace_B_3, int dace_n_0, int dace_tsteps_0) {
    int tmp_for_3;
    int dace_t_0;

    for (tmp_for_3 = 0; (tmp_for_3 < dace_tsteps_0); tmp_for_3 = (tmp_for_3 + 1)) {
        {


            {
                #pragma omp parallel for
                for (auto tmp_for_4 = 1; tmp_for_4 < (dace_n_0 - 1); tmp_for_4 += 1) {
                    {
                        double A_1 = dace_A_3[(tmp_for_4 - 1)];
                        double A_2 = dace_A_3[tmp_for_4];
                        double A_3 = dace_A_3[(tmp_for_4 + 1)];
                        double B_out_1;

                        ///////////////////
                        B_out_1=0.33333*(A_1+A_2+A_3);
                        ///////////////////

                        dace_B_3[tmp_for_4] = B_out_1;
                    }
                }
            }
            {
                #pragma omp parallel for
                for (auto tmp_for_5 = 1; tmp_for_5 < (dace_n_0 - 1); tmp_for_5 += 1) {
                    {
                        double B_1 = dace_B_3[(tmp_for_5 - 1)];
                        double B_2 = dace_B_3[tmp_for_5];
                        double B_3 = dace_B_3[(tmp_for_5 + 1)];
                        double A_out_1;

                        ///////////////////
                        A_out_1=0.33333*(B_1+B_2+B_3);
                        ///////////////////

                        dace_A_3[tmp_for_5] = A_out_1;
                    }
                }
            }
```

From C to DaCe

```
for (int i = 0; i < N ; i++)
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++)
        y[i] +=  A[col_idx[j]] * x[j];
```

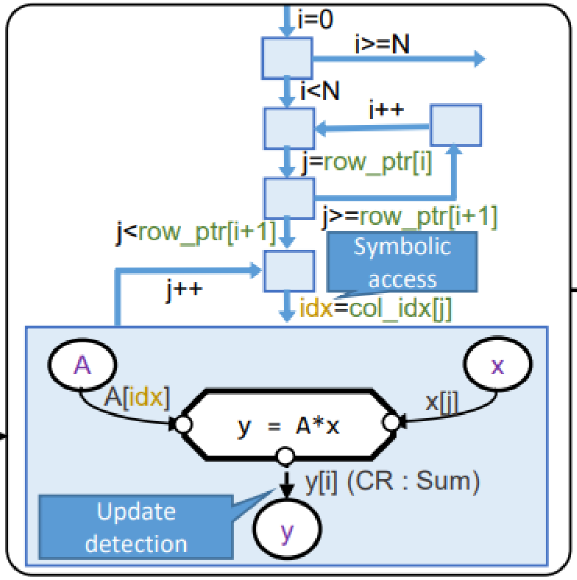
AST transformation

```
for (int i = 0; i < N ; i++)
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++)
    {
        int idx=col_idx[j];
        y[i] += A[idx] * x[j];
    }
```

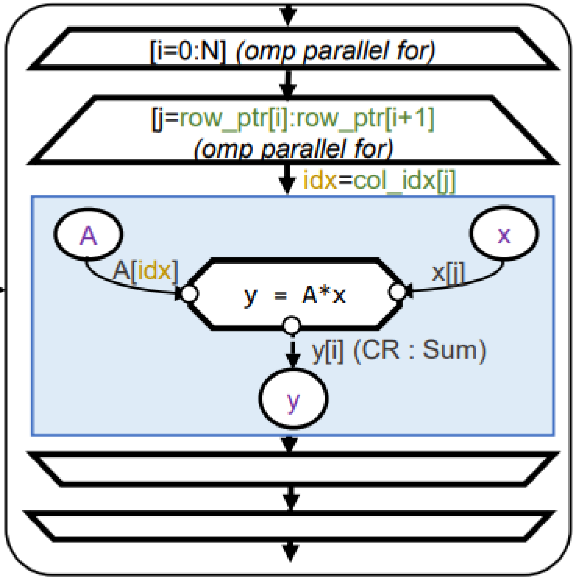
Index extraction

C-to-DaCe translation

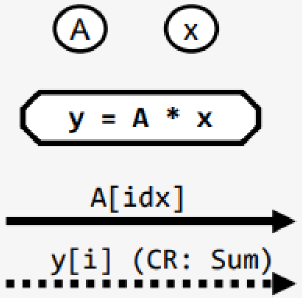
Dataflow coarsening



Dataflow optimization



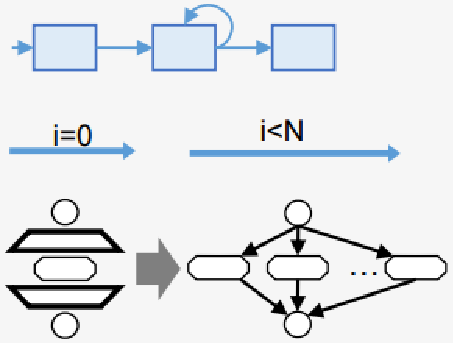
SDFG components



Data: Array containers

Tasklet: Fine-grained computation

Memlet: Data movement unit, with parallel write conflict resolution (CR) options



States: Control dependencies

Interstate edges: symbolic conditions and assignments dependencies

Map: Parametric parallelism scope

C2DaCe

- **Practical steps**
  - Clang (libclang with python bindings) creates initial AST representation
  - Transferred to internal AST representation (similar to the Python AST representation)
  - Apply canonicalization AST transformation passes
  - Create initial SDFG from canonical AST
  - Simplify SDFG
  - Coarsen dataflow
  - Identify parallelization opportunities
  - Generate code

- Written report: https://arxiv.org/abs/2112.11879
- Code prototype: https://github.com/spcl/c2dace
- DaCe transformations – already in production: https://github.com/spcl/dace

C – SDFG equivalence

C Language	SDFG Equivalent
Declarations and Types	
Primitive data type	Scalar data container
Array	Array data container
Pointer	Access node to existing data container, or new data container if pointing to newly allocated memory.
Expressions and Assignments	
Operators (Unary, Binary,...)	Tasklet with incoming and outgoing memlets for read/written operands
Array expression	Memlet
Statements	
Compound (blocks)	State
Branching (`if, ...`)	Branch conditions on state transition edges
Iteration (`for, while, ...`)	State for compound statement, with states and transitions for loop logic
Function flow (`break, continue, return`)	State transitions
`goto`	State transition
Functions	
Function calls (with source)	Nested SDFG for content, memlets reduce shape of inputs and outputs
External/Library calls	Tasklet with library state
Recursion	Unsupported
Function pointers	No equivalent, unsupported
Parallelism	
–	Parametric map scope

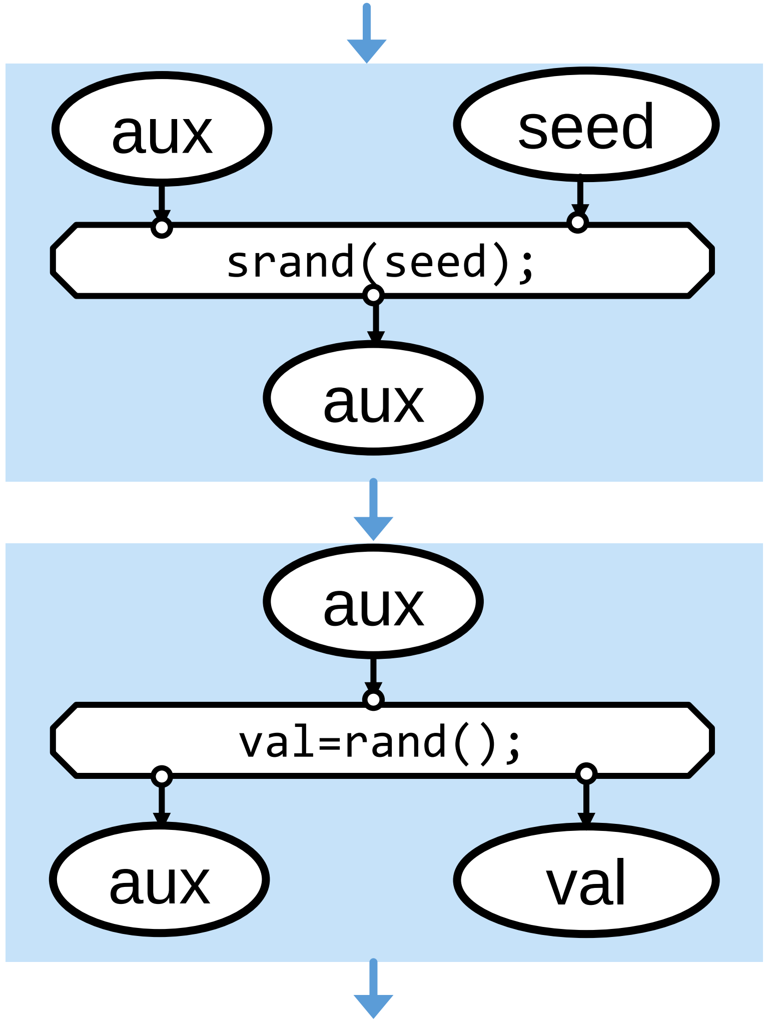
Whole program analysis allows elimination of almost all alias sources! Exception: opaque libraries and data originating within!

Important note:
The SDFG resulting from the initial translation from C is not a data-centric program! All control-centric constructs from C are unchanged – the process of simplification and optimization is only starting!

Datacentric program analysis can discover opportunities for parallelism where classical compilers cannot.

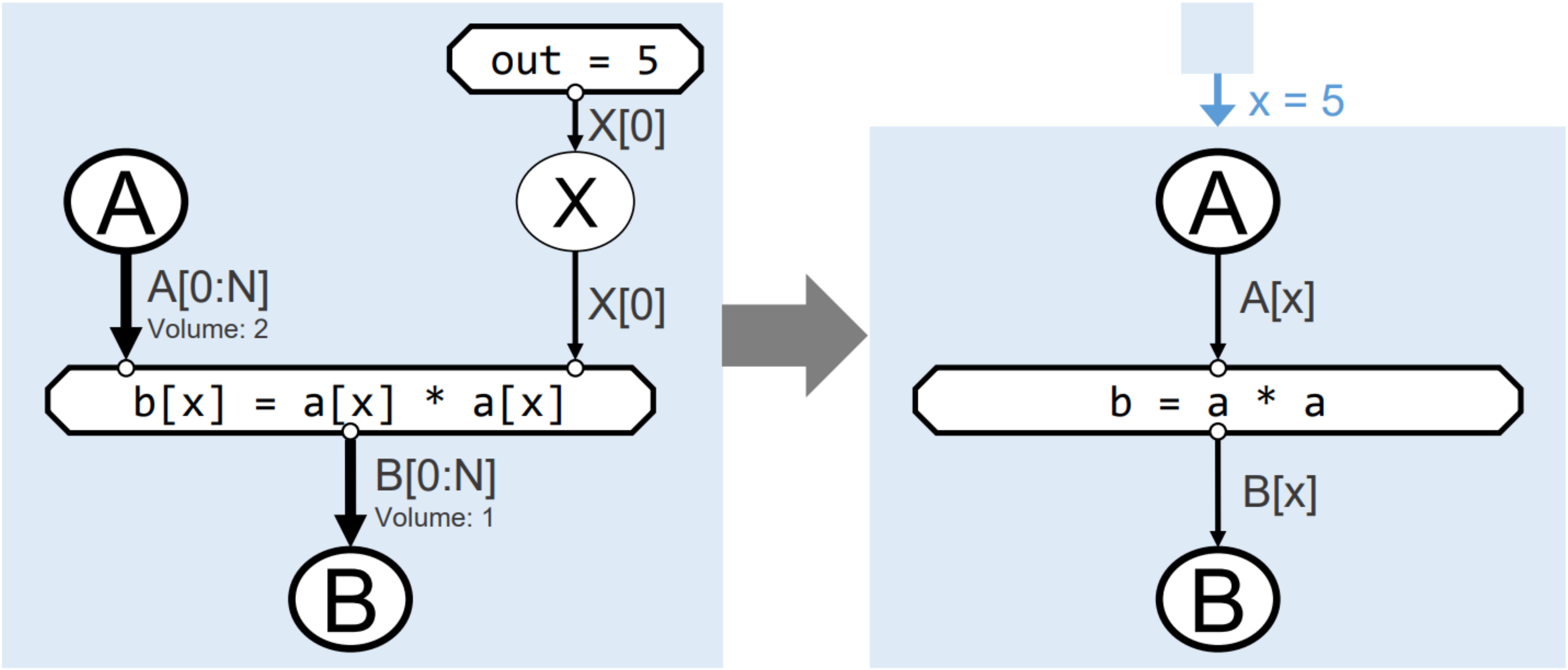
Enforcing order for data-less dependencies

- Global dictionary of states
- By default, all calls within a library should share a state
- Lists of stateless functions/libraries avoid needless complication of the SDFG

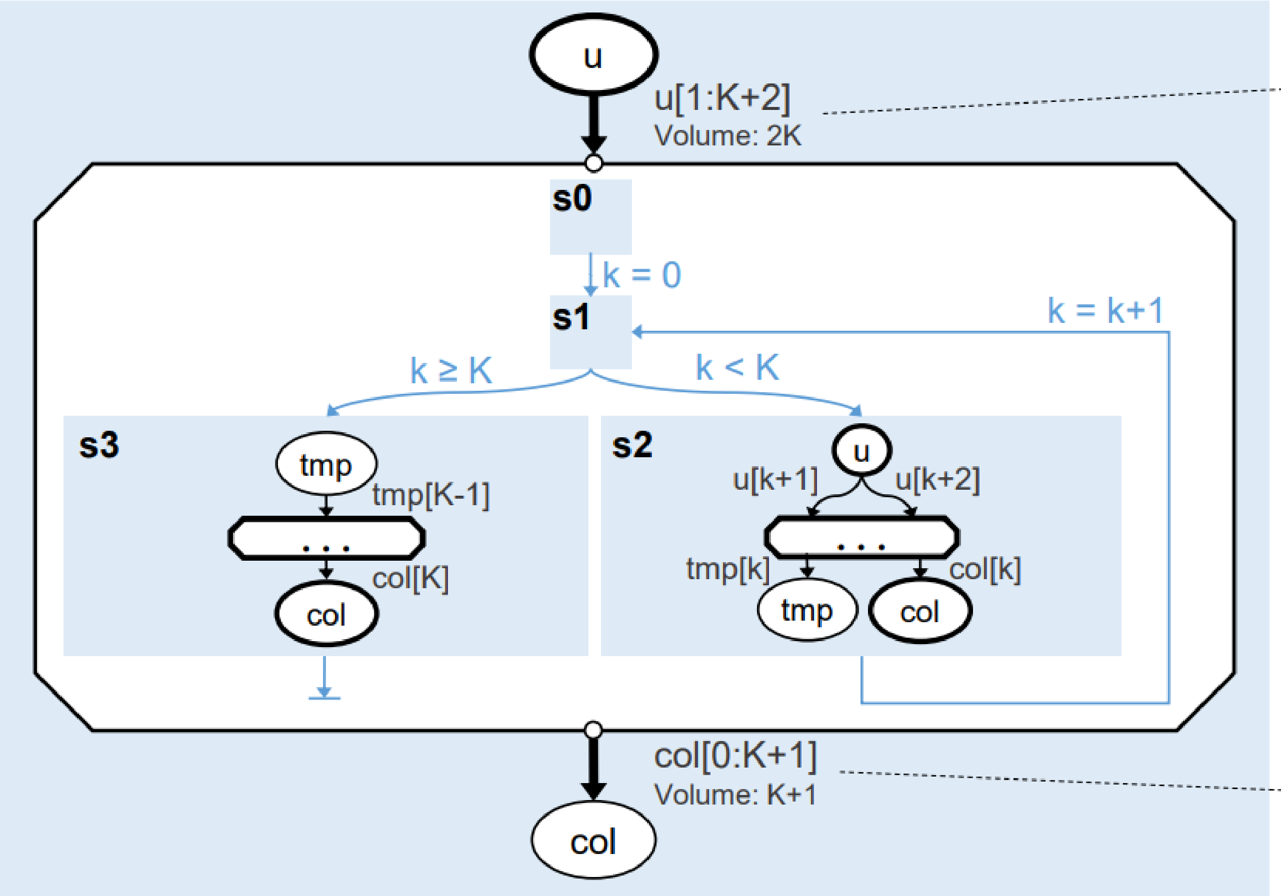


Symbolic scalar analysis

C code: `int x = 5; /*...*/ B[x] = A[x] * [x];`



Access pattern propagation



Subset: $\bigcup_{k=0}^{K}\{k+1, k+2\}$
Volume: $2 \cdot exec(s2)$

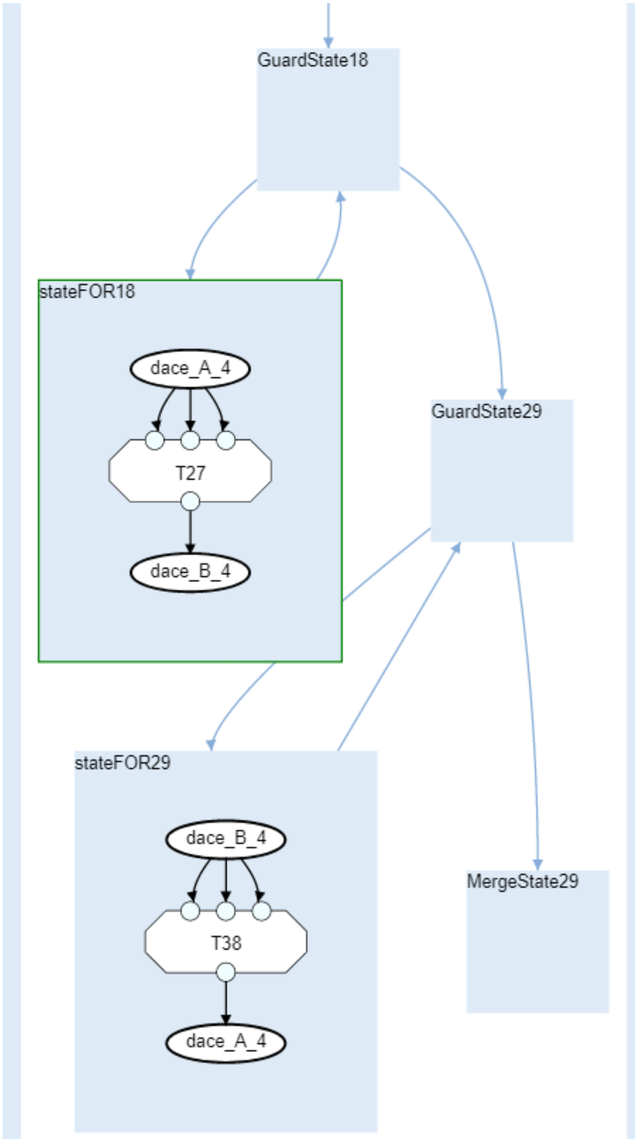
s0:
Executions = 1
Symbol Ranges = ∅

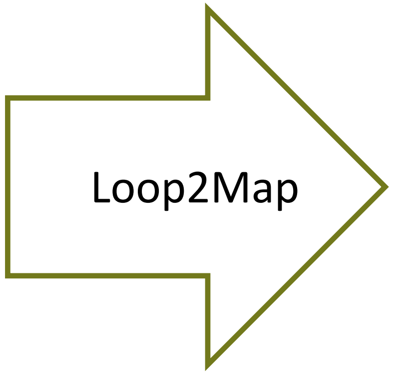
s3:
Executions = 1
Symbol Ranges = {k: K}

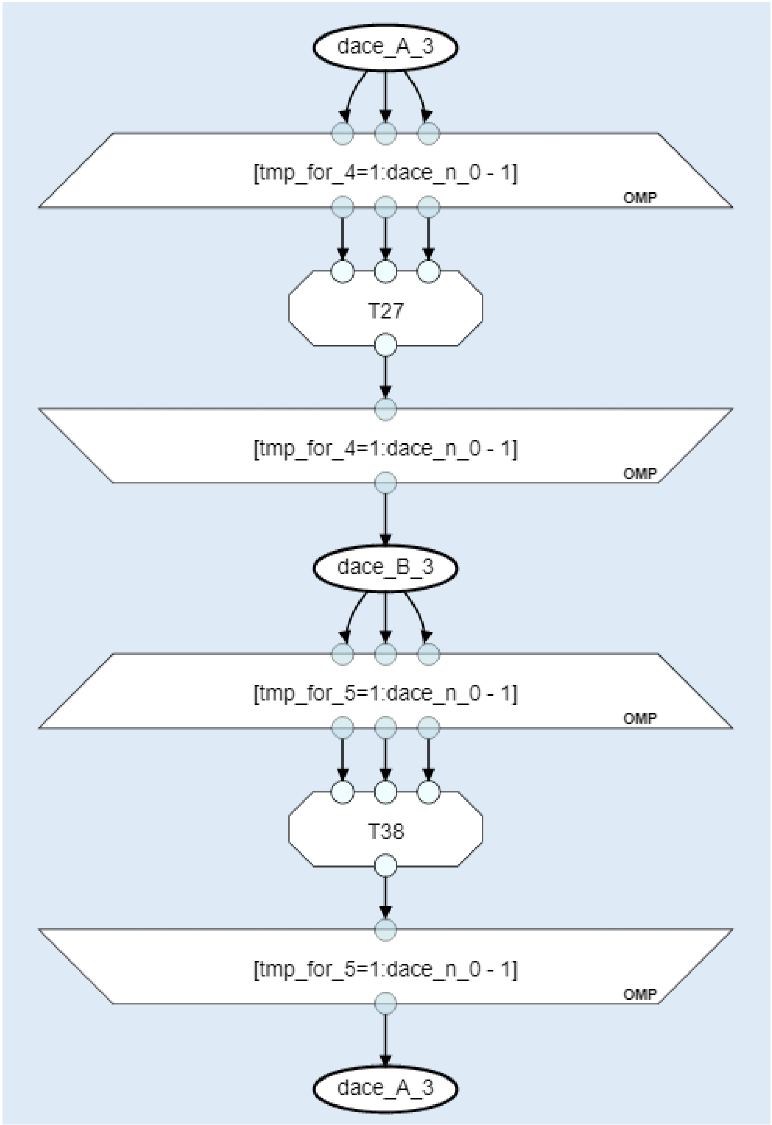
s1, s2:
Executions = K
Symbol Ranges = {k: 0:K}

Subset: $\bigcup_{k=0}^{K}\{k\} \cup \{K\}$
Volume: $1 \cdot exec(s2) +$
$1 \cdot exec(s3)$

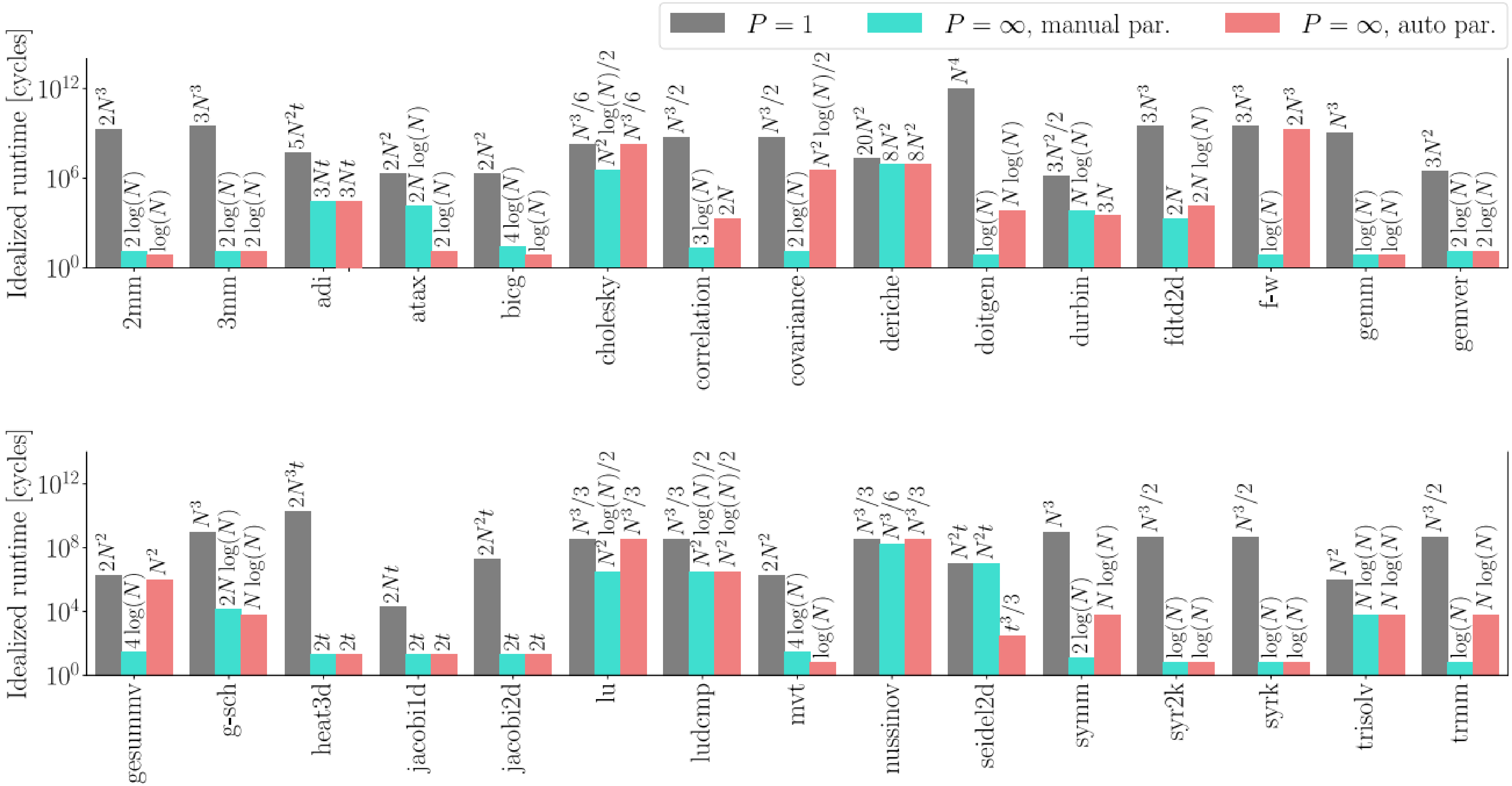
Automatic parallelization



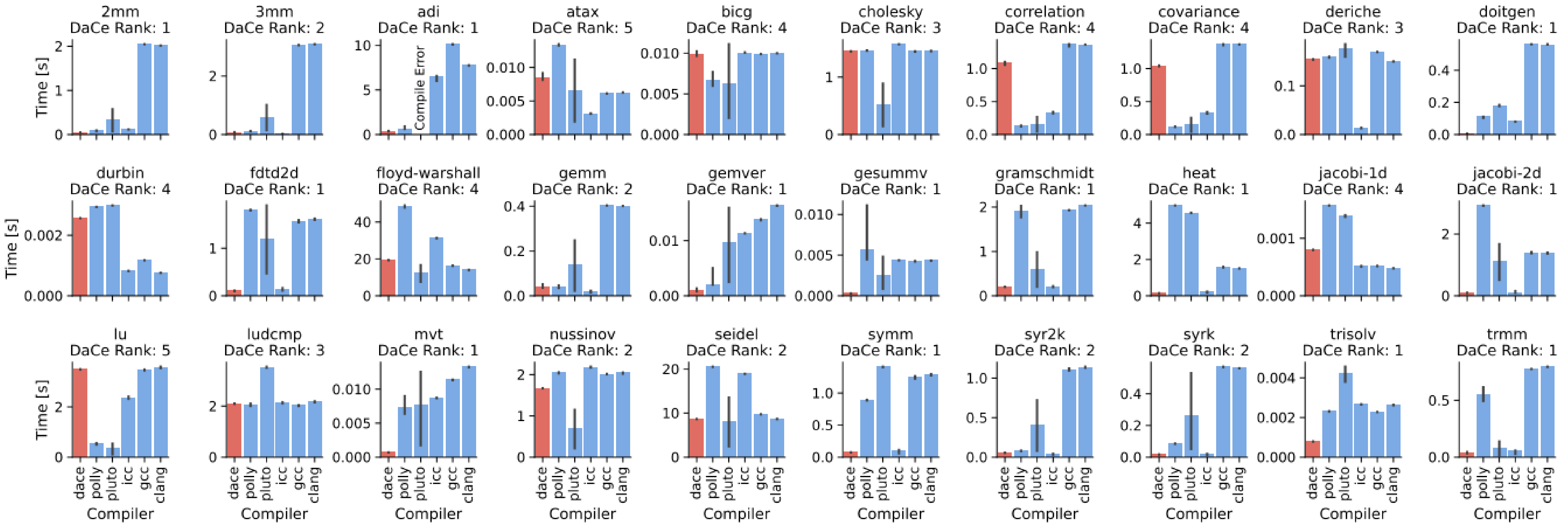




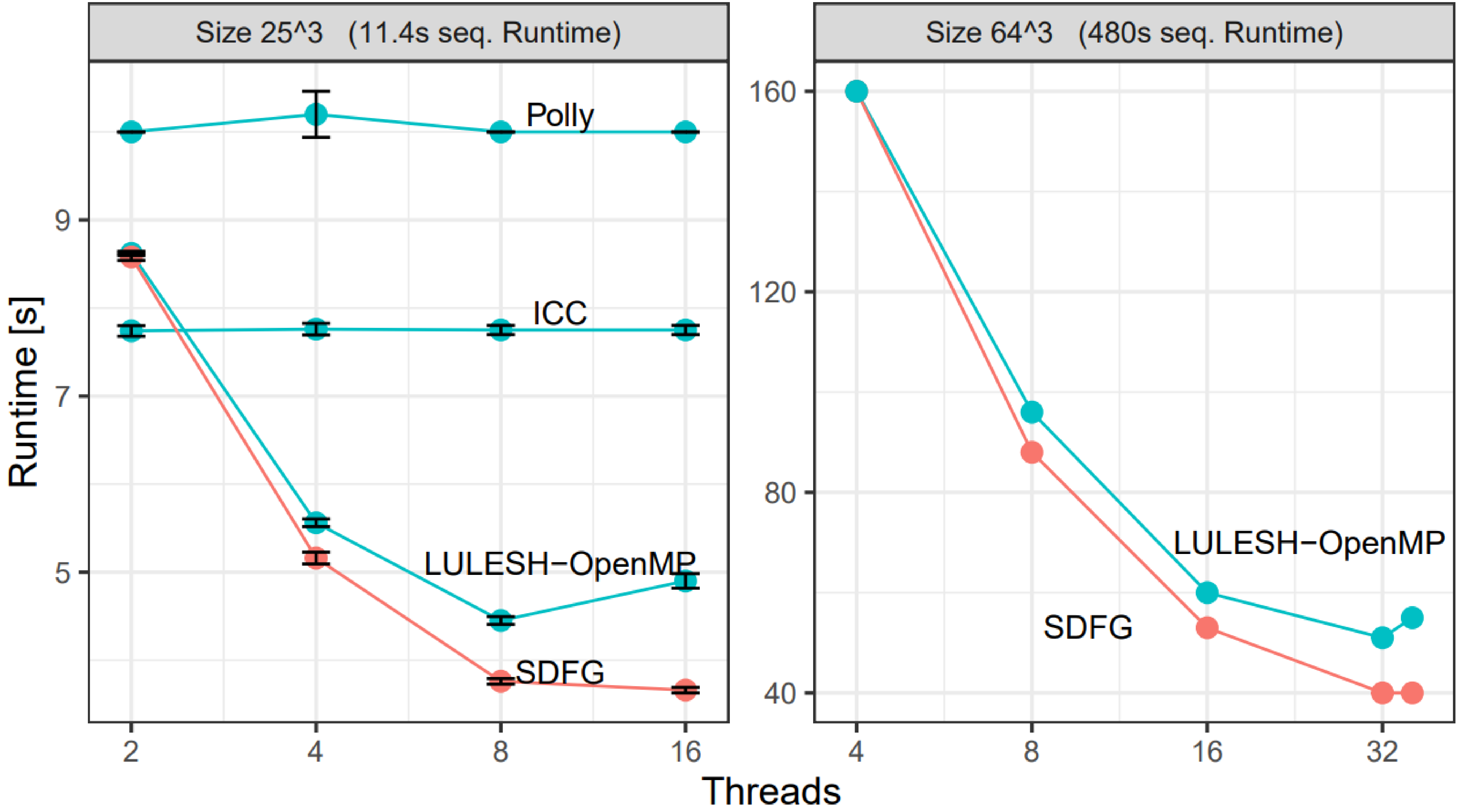
Automatic work-depth analysis



Results - Polybench



Results - LULESH

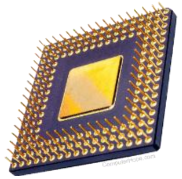


Next years?

C2DaCe

Fortran

Dataflow extaction



Snitch