A Data-Centric Optimization Workflow for the Python Language

Alexandros Nikolaos Ziogas, Torsten Hoefler

Motivation

Python is the language of choice for scientific computing due to its high productivity. However, its execution is often slow. How do we make Python the language of choice for writing highly performant code too?

From Python …

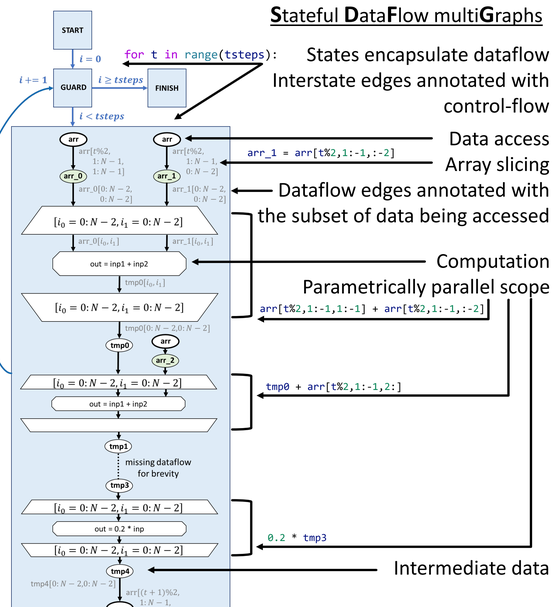
Python with type annotations for Ahead-Of-Time compilation

```
@dace.program
def jacobi2d(tsteps: dace.int32, arr: dace.float64[2, N, N]):
    for t in range(tsteps):
        arr[(t+1)%2, 1:-1, 1:-1] = 0.2 * (arr[t%2, 1:-1, 1:-1] +
                                          arr[t%2, 1:-1, :-2] +
                                          arr[t%2, 1:-1, 2:] +
                                          arr[t%2, 2:, 1:-1] +
                                          arr[t%2, :-2, 1:-1])
```

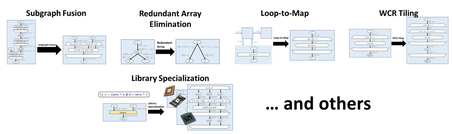
MPI-compatible syntax for distributed-memory programming

```
@dace.program
def jacobi2d_distr(tsteps: dace.int32, arr: dace.float64[2,Nx+2,Ny+2]):
    req = np.empty((8,), dtype=dace.comm.Request)
    for t in range(tsteps):
        dace.comm.Isend(arr[t%2, 1, 1:-1], north, 0, req[0])
        ...
        dace.comm.Irecv(arr[t%2, 1:-1, -1], east, 2, req[7])
        dace.comm.Waitall(req)
        arr[(t+1)%2, 1+noff:-1-soff, 1+woff:-1-eoff] = 0.2 * (...)
```

… to a Data-Centric Representation …

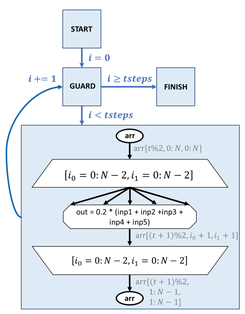


… optimized via Graph Transformations …



Optimization driven by:

- ❑ User/performance engineer
- ❑ Automatic optimization heuristics
- ❑ Performance modeling



… generating Architecture-Specific Code

CPU

```
for (t = 0; (t < tsteps); t = t + 1) {
    #pragma omp parallel for
    for (auto __i0 = 0; __i0 < (N - 2); __i0 += 1) {
        for (auto __i1 = 0; __i1 < (N - 2); __i1 += 1) {
            double __in1 = arr[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1) + 1)];
            double __in2_0_0 = arr[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1)];
            double __in2_1 = arr[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1) + 2)];
            double __in2_0 = arr[((((N * N) * (t % 2)) + (N * (__i0 + 2))) + __i1) + 1)];
            double __in2 = arr[((((N * N) * (t % 2)) + (N * __i0)) + __i1) + 1)];
            double __out;
            __out = (0.2 * ((((__in1 + __in2_0_0) + __in2_1) + __in2_0) + __in2));
            arr[((((N * N) * ((t + 1) % 2)) + (N * (__i0 + 1))) + __i1) + 1)] = __out;
        }
    }
}
```

GPU

```
__global__ void outer_fused_0_0_2(double * __restrict__ gpu_arr, int N, long long t) {
    int __i1 = (blockIdx.x * 32 + threadIdx.x);
    int __i0 = (blockIdx.y * 1 + threadIdx.y);
    if (__i1 < (N - 2)) {
        double __in1 = gpu_arr[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1) + 1)];
        double __in2_0_0 = gpu_arr[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1)];
        double __in2_1 = gpu_arr[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1) + 2)];
        double __in2_0 = gpu_arr[((((N * N) * (t % 2)) + (N * (__i0 + 2))) + __i1) + 1)];
        double __in2 = gpu_arr[((((N * N) * (t % 2)) + (N * __i0)) + __i1) + 1)];
        double __out;
        __out = (0.2 * ((((__in1 + __in2_0_0) + __in2_1) + __in2_0) + __in2));
        gpu_arr[((((N * N) * ((t + 1) % 2)) + (N * (__i0 + 1))) + __i1) + 1)] = __out;
    }
}
```

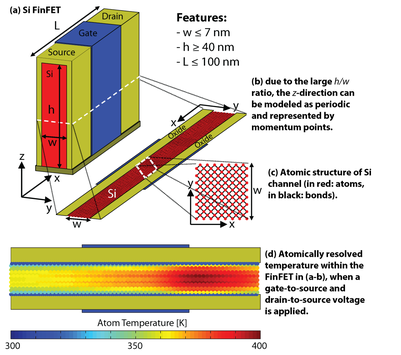
FPGA

```
for (t = 0; (t < __tsteps_in); t = (t + 1)) {
    for (int __i0 = 0; __i0 < (N - 2); __i0 += 1) {
        for (int __i1 = 0; __i1 < (N - 2); __i1 += 1) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_FLATTEN
            {
                double __in1 = __arr_in[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1) + 1)];
                double __in2_0_0 = __arr_in[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1)];
                double __in2_1 = __arr_in[((((N * N) * (t % 2)) + (N * (__i0 + 1))) + __i1) + 2)];
                double __in2_0 = __arr_in[((((N * N) * (t % 2)) + (N * (__i0 + 2))) + __i1) + 1)];
                double __in2 = __arr_in[((((N * N) * (t % 2)) + (N * __i0)) + __i1) + 1)];
                double __out;
                __out = (0.2 * ((((__in1 + __in2_0_0) + __in2_1) + __in2_0) + __in2));
                *(__arr_out + ((((N * N) * ((t + 1) % 2)) + (N * (__i0 + 1))) + __i1) + 1)) = __out;
            }
        }
    }
}
```

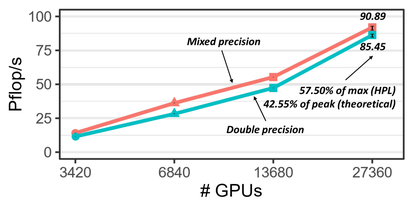
Case Studies

Quantum Transport Simulation
(ACM Gordon Bell Award 2019)

Implementation in Python and optimization of Quantum Transport Solver OMEN (original application written in C++).



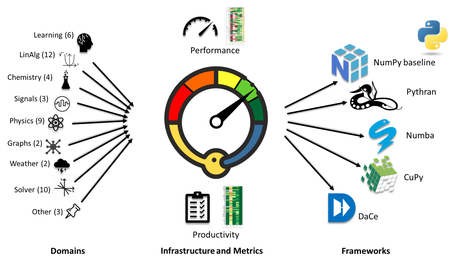
OMEN simulates the electro-thermal properties of nanodevices.

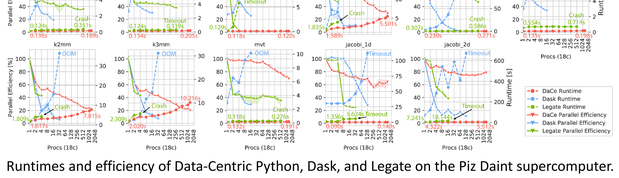


Strong scaling of Data-Centric OMEN on the Summit supercomputer.

NPBench

A benchmarking suite for Python, with over 50 code samples across a wide range of scientific domains.





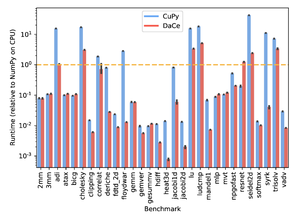
Runtimes and efficiency of Data-Centric Python, Dask, and Legate on the Piz Daint supercomputer.

Improvement in parallel efficiency compared to

DASK **2.5x**

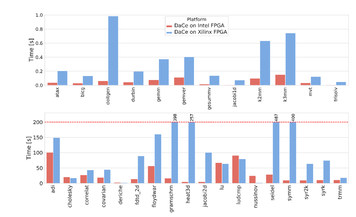
LEGATE **1.7x**

Speedups of Data-Centric Python and CuPy compared to the reference Python execution on NVIDIA V100 GPU.

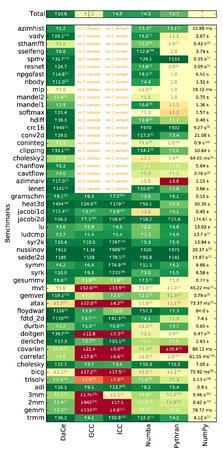


Speedup compared to

CuPy **3.75x**



Runtimes of Data-Centric Python on Intel Stratix 10 2800 GX and Xilinx Alveo U250 FPGAs.



Reference Python (NumPy) runtime on two Intel Xeon Gold 6130 CPUs. Speedups (green) and slowdowns (red) of Python-accelerating solutions compared to the reference execution.

Speedup compared to

CPython+NumPy **10.6x**

Numba **2.47x**

Pythran **3.93x**

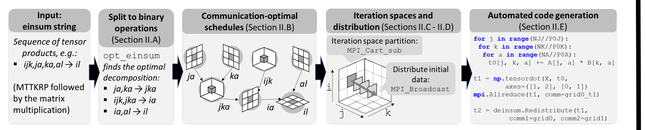
GCC **9.27x**

ICC **2.27x**

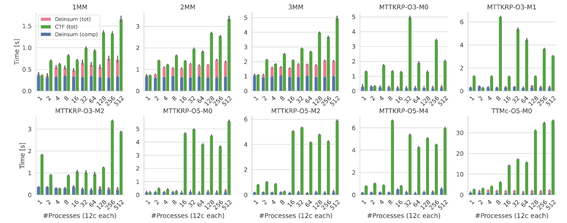
(subset with a C-baseline)

Multilinear Algebra and Einsums

Using Data-Centric Python to implement Deinsum, an automated framework for distributed multilinear algebra computations expressed in Einstein notation.



Deinsum's workflow



Runtimes of Deinsum and CTF on the Piz Daint supercomputer (CPU).

up to 19x faster than CTF

Runtimes of Deinsum and CTF on the Piz Daint supercomputer (GPU).

For references, resources, and other information, please follow the link below: