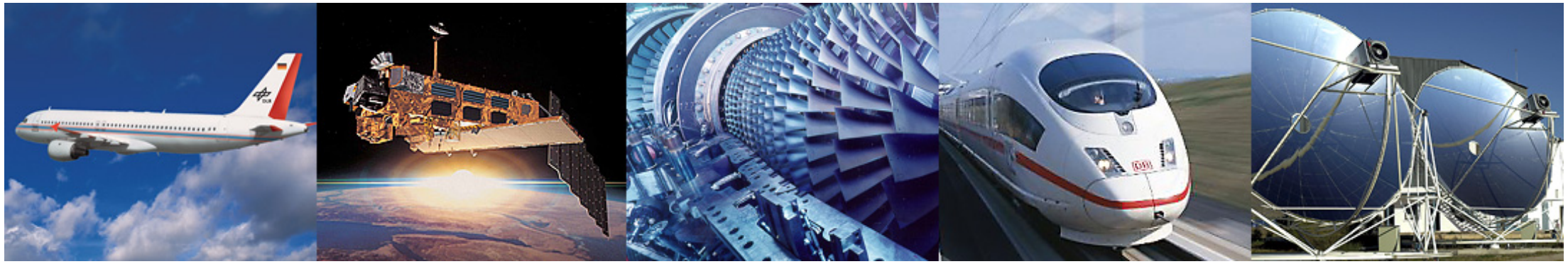
Performance and Productivity of Parallel Python Programming — A study with a CFD Test Case

Achim Basermann, Melven Röhrig-Zöllner and Joachim Illmer
German Aerospace Center (DLR)
Simulation and Software Technology
Linder Höhe, Cologne, Germany

DLR

DLR
German Aerospace Center

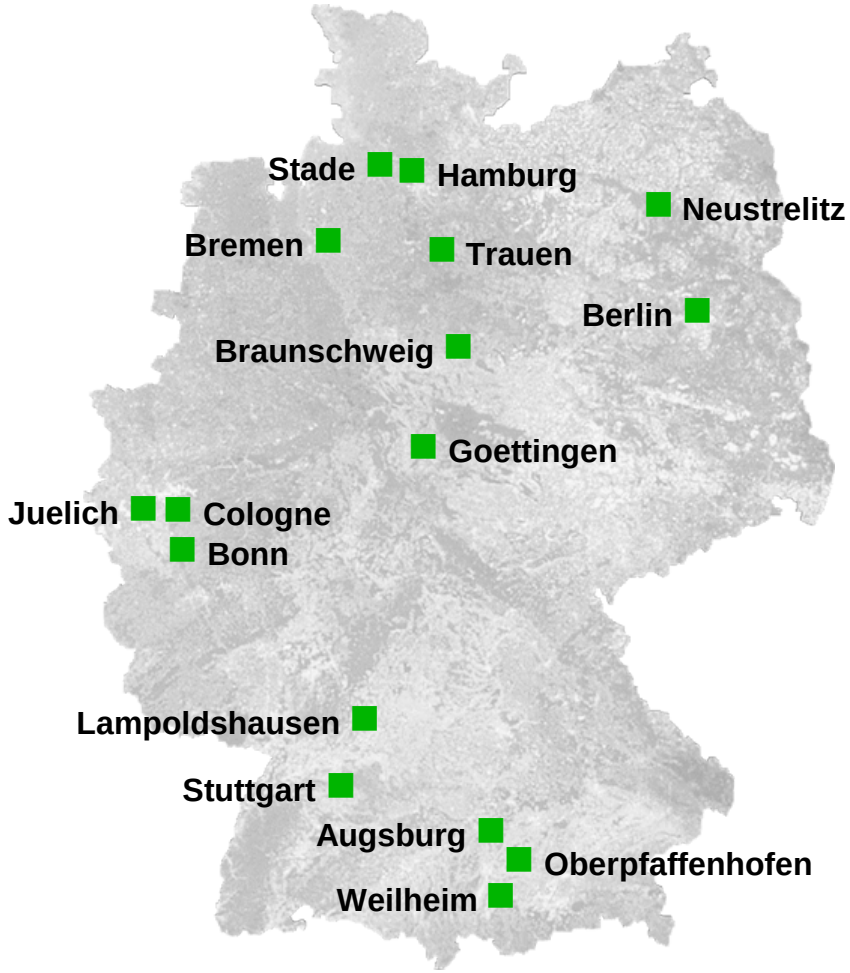


- Research Institution
- Space Agency
- Project Management Agency

DLR Locations and Employees

Approx. 8000 employees across 33 institutes and facilities at
- 16 sites.

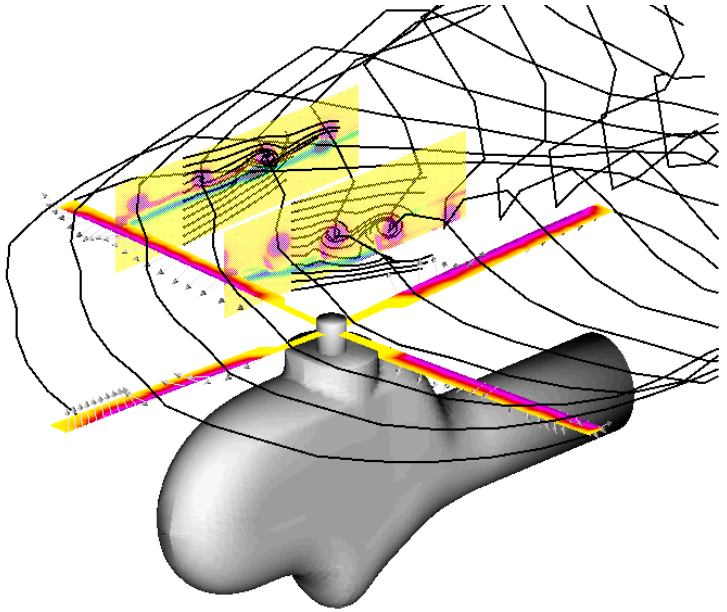
Offices in Brussels, Paris, Tokyo and Washington.



Survey

- Background rotor simulation test case
- Python libraries which support parallel programming
- Python implemenations of a rotor simulation kernel
- Performance analysis
  - Performance modelling
  - Performance measurements
- Conclusions

DLR Project Free-Wake



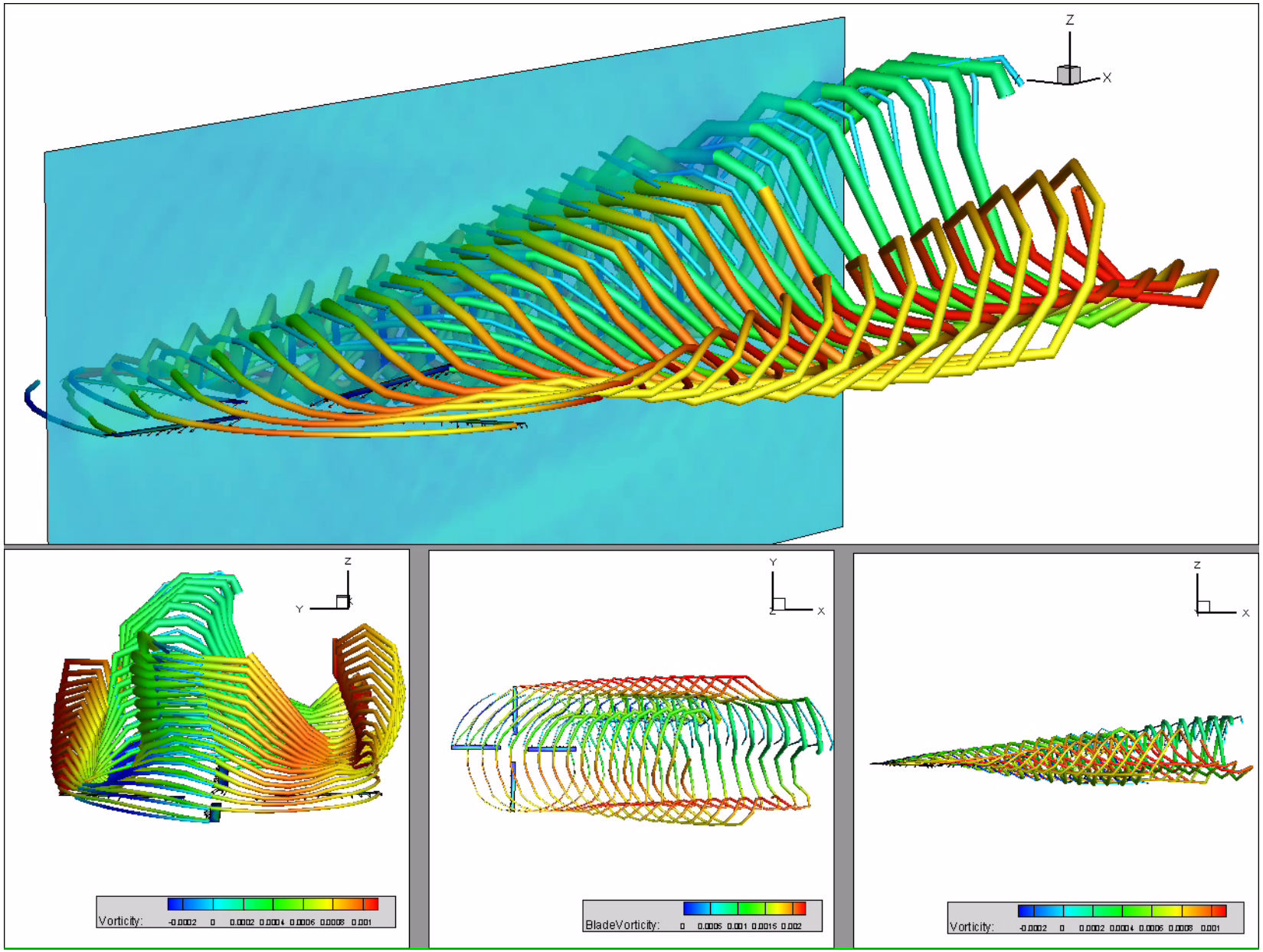
- Developed 1994-1996
- Simulates the flow around a helicopter’s rotor
- Vortex-Lattice method
- Discretizes complex wake structures with a set of vortex elements
    - Account for vortex aging, stretching and compression
- Based on experimental data (from the international HART-program 1995)
- MPI-parallel implementation in Fortran
- Loose coupling with the DLR rotor simulation code S4

GPU Parallelization with OpenACC

- Directive based
- Similar to OpenMP
- Explicit data movement between host and GPU (bottleneck!)
- Supported by CAPS-, CRAY- and **PGI-compilers** (C, C++, Fortran)
- Recently: Also available for GCC

```
program main
   integer :: a(N)
   …
!$acc data copyout(a)
   ! computation on the GPU in several loops:
!$acc parallel loop
   do i = 1, N
      a(i) = 2*a(i)
   end do
!$acc parallel loop
   …
!$acc end data
   ! Now results available on the CPU
   …
end program main
```

Vortex Visualization



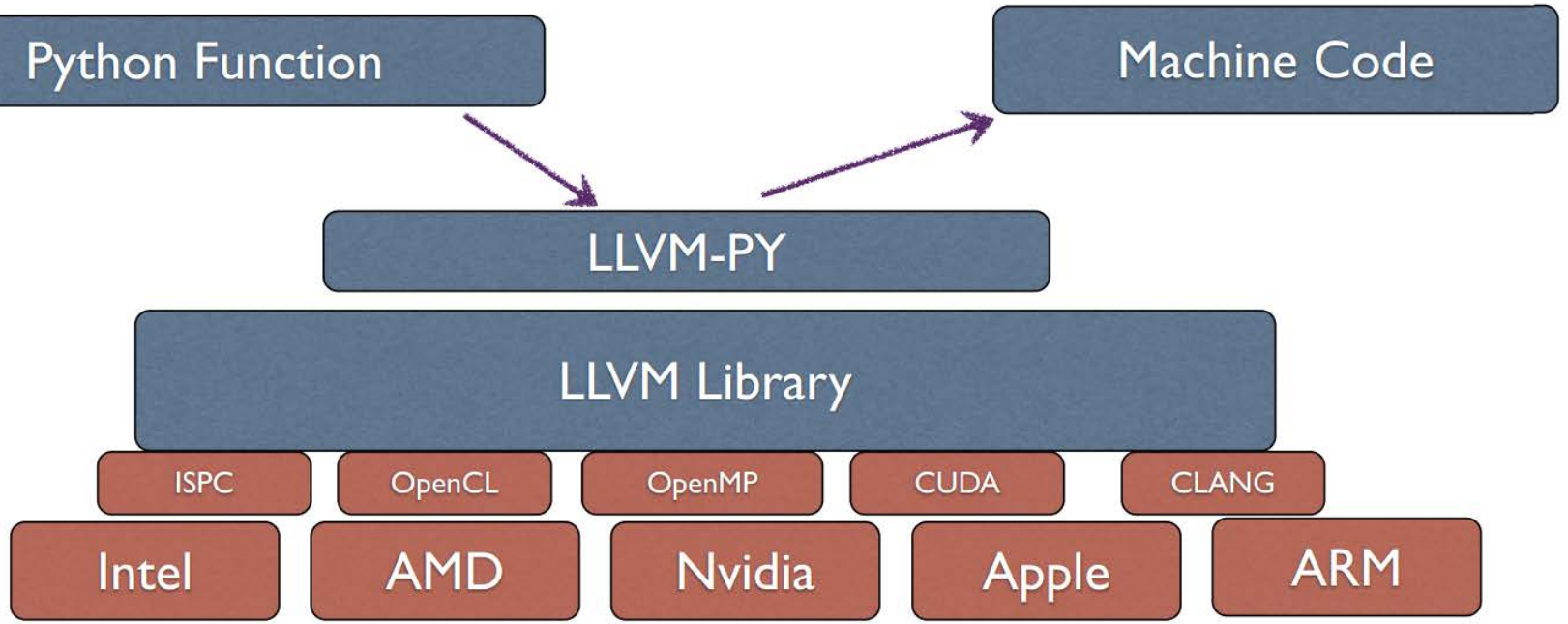
Python Libraries Which Support Parallel Programming (1)

- Cython
  - translates Python code into C code
  - then compilation by a C compiler
  - offers simple use of OpenMP for shared memory parallelization

- NumPy
  - efficient use of large N-dimensional arrays
  - comfortable use of e.g. linear algebra routines
  - hidden optimization or parallelization of basic routines possible

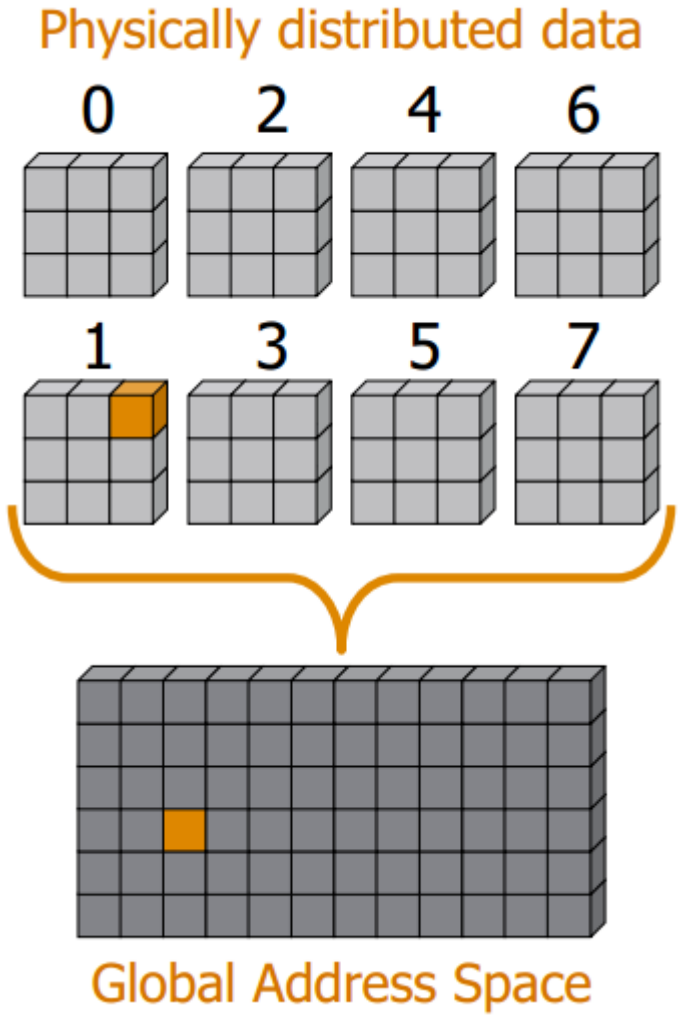
Python Libraries Which Support Parallel Programming (2)

- Numba
  - open source compiling environment for Python and NumPy
  - generates optimized machine code by use the LLVM compiler
  - supports OpenMP, OpenCL and CUDA



Python Libraries Which Support Parallel Programming (3)

- Python Bindings for Global Array Toolkit
  - allows access to physically distributed data via a shared memory interface
  - extension of MPI for message exchange between parallel processes
  - in the same program, switching between the shared memory interface of the Global arrays and the distributed memory interface of MPI possible



Standard Python Implementation of a Rotor Simulation Kernel

Call of the function *wilin*

```
for iblades in range(numberOfBlades):
    for iradial in range(1, dimensionInRadialDirection):
        for iazimutal in range(dimensionInAzimualDirectionTotal):
            for i1 in range(len(vx[0])):
                for i2 in range(len(vx[0][0])):
                    for i3 in range(len(vx[0][0][0])):
                        #wilin call 1
for iblades in range(numberOfBlades):
    for iradial in range(dimensionInRadialDirection):
        for iazimutal in range(1,
            ↪ dimensionInAzimualDirectionTotal):
            for i1 in range(len(vx[0])):
                for i2 in range(len(vx[0][0])):
                    for i3 in range(len(vx[0][0][0])):
                        #wilin call 2
for iDir in range(3):
    for i in range(numberOfBlades):
        for j in range(dimensionInRadialDirection):
            for k in range(dimensionInAzimualDirectionTotal):
                x[iDir][i][j][k] = x[iDir][i][j][k] + dt * vx[iDir
                    ↪ ][i][j][k]
```

Function *wilin* for velocity induction

```
def wilin(dax, day, daz, dex, dey, dez, ga, ge, wl, vx, vy, vz):
    rcq = 0.1
    daq = dax**2 + day**2 + daz**2
    deq = dex**2 + dey**2 + dez**2
    da = math.sqrt(daq)
    de = math.sqrt(deq)
    dae = dax * dex + day * dey + daz * dez
    sqa = daq - dae
    sqe = deq - dae
    sq = sqa + sqe
    rmq = daq * deq - dae**2
    fak = ((da + de) * (da * de - dae) * (ga * sqe + ge * sqa) +
        ↪  (ga - ge) * (da - de) * rmq) / (sq * (da * de + eps
        ↪ ) * (rmq + rcq * sq))
    fak = fak * wl / math.sqrt( sq )
    vx = vx + fak * (day * dez - daz * dey)
    vy = vy + fak * (daz * dex - dax * dez)
    vz = vz + fak * (dax * dey - day * dex)
    return vx, vy, vz
```

Cython Implementation of the Rotor Simulation Kernel

- Parallelization with the function *prange*
- realizes loop parallelization with OpenMP
- Change of loop order for optimization

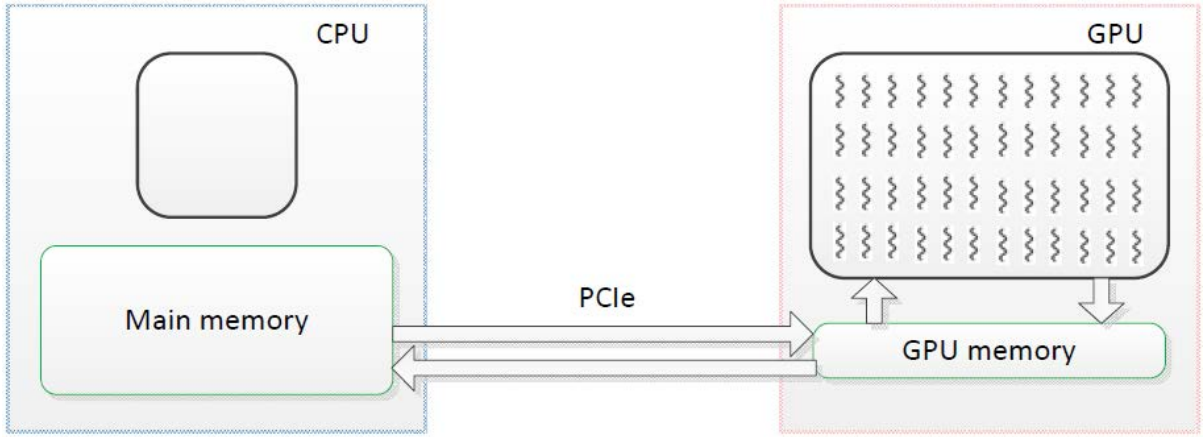
```
for i2 in prange(dimensionInRadialDirection):
    for i1 in xrange(numberOfBlades):
        for i3 in xrange(dimensionInAzimualDirectionTotal):
            for iblades in xrange(numberOfBlades):
                for iradial in xrange(1,
                    ↪ dimensionInRadialDirection):
                    for iazimutal in xrange(
                        ↪ dimensionInAzimualDirectionTotal):
```

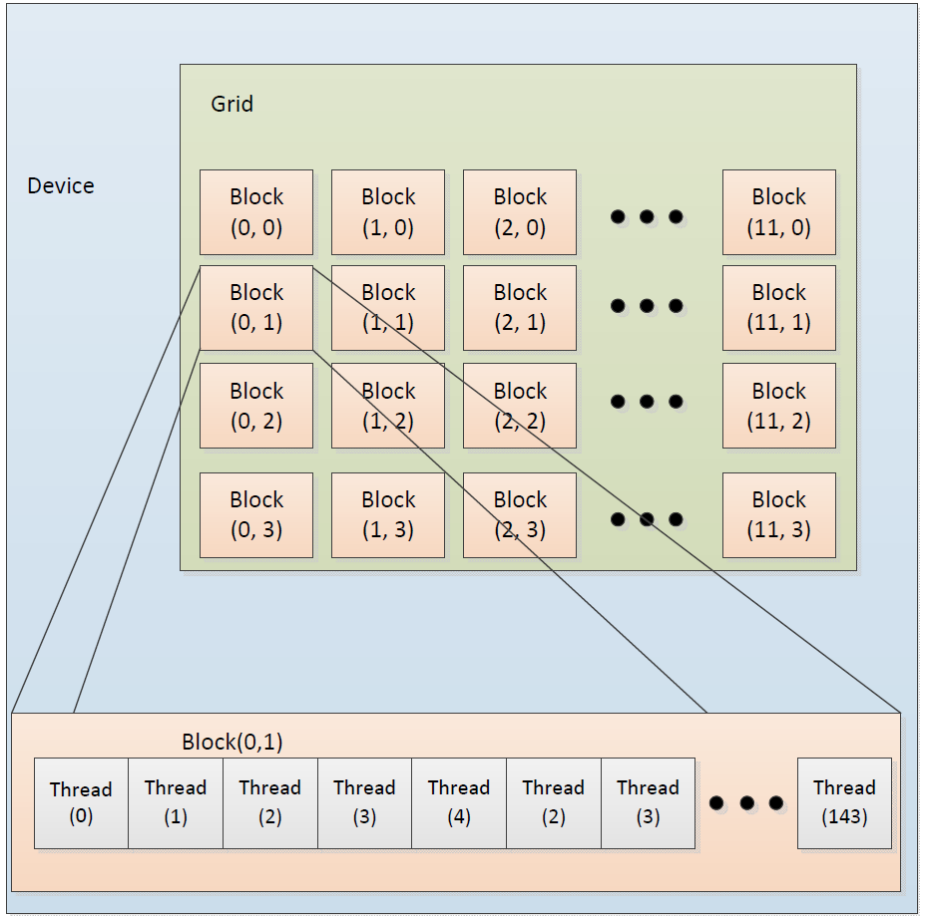
NumPy Implementation of the Rotor Simulation Kernel

- Exploitation of NumPy´s fast array operations
- generate NumPy arrays rather than standard Python lists
- call *wilin* with array parameters
- exploit fast NumPy functions like *numpy.sqrt* rather than *math.sqrt*

Numba Implementation of the Rotor Simulation Kernel

- Single-core optimization in the accelerated *nopython* mode
- Annotation *autojit* triggers compilation to machine code
- Multi-core optimization via *prange* command or *Vectorize* function; both were not applicable in our tests
- GPU optmization with the *CUDA JIT* annotation
- NUMBA offers a machine-oriented CUDA entry point.
- Synchronization with *cuda.syncthreads()*






```
i1 = cuda.blockIdx.x
i2 = cuda.blockIdx.y
i3 = cuda.threadIdx.x
```

Global Arrays Implementation of the Rotor Simulation Kernel

- Bases on the NumPy implementation
- Generation of Global Arrays (*g_*) of same dimension and type as the corresponding NumPy arrays necessary

```
g_x = ga.create(ga.C_DBL, [dim1, dim2, dim3, dim4])
```

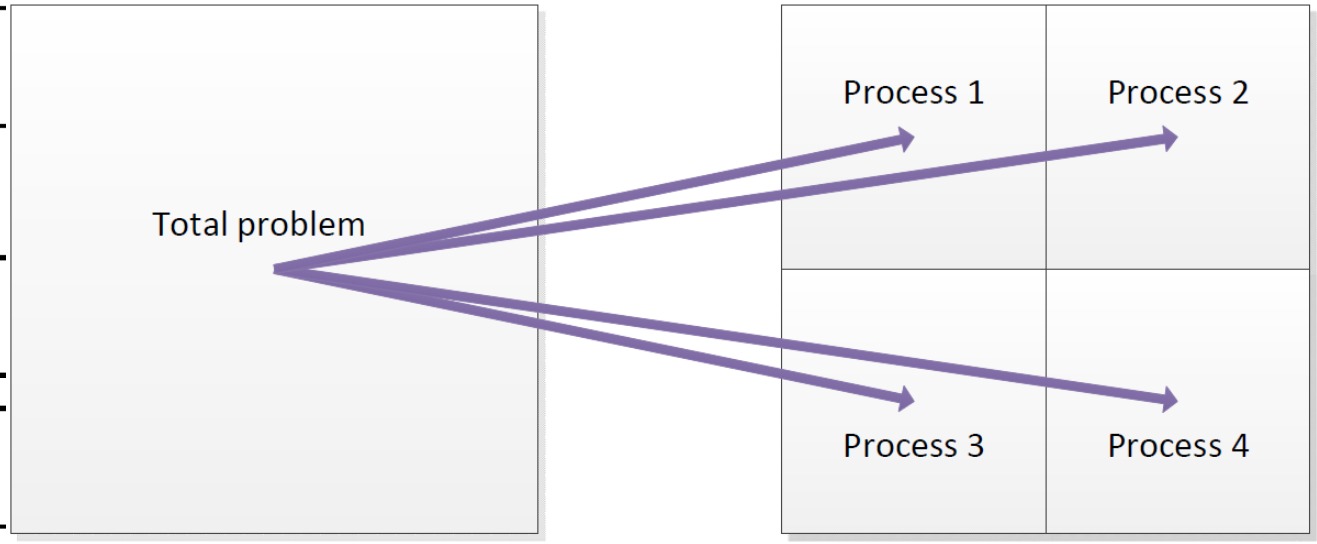
- Determine the scope of a Global Array administrated on a compute node

```
lo,hi = ga.distribution(g_x, node)
```

- Perform local computations with NumPy arrays (*l_*)

```
l_x = ga.get(g_x,lo,hi)
```

```
ga.put(g_x, l_x, lo,hi)
```



Performance Modelling: Roofline Performance Model

- Machine balance: $$B_m = \frac{\text{memory bandwidth [GBytes/s]}}{\text{peak performance [GFLOP/s]}}$$
- Code balance: $$B_c = \frac{\text{data traffic [Bytes]}}{\text{floating point operations [FLOP]}}$$
- Maximum kernel Performance: peak performance x $$l = \min\left(1, \frac{B_m}{B_c}\right)$$
- Intel Xeon E5645 CPU, 6 cores: $$B_m = \frac{19\,[\text{GByte/s}]}{28.8\,[\text{GFLOP/s}]} = 0.6597$$
- NVIDIA Tesla C2075 GPU, 448 CUDA cores: $$B_m = \frac{144\,[\text{GByte/s}]}{515\,[\text{GFLOP/s}]} = 0.2796$$

All codes with

a code balance larger than the machine balance

are bandwidth limited on this machine.

Performance Modelling of the Rotor Simulation Kernel

- Code balance of the rotor simulation kernel: $B_c = \frac{550,000[Bytes]}{6,500,000,000[FLOP]} = 0.00008 \frac{Bytes}{FLOP}$

- Performance factors on CPU or GPU:

$$l_{CPU} = \min\left(1, \frac{0.6597}{0.00008}\right) = 1$$

$$l_{GPU} = \min\left(1, \frac{0.2796}{0.00008}\right) = 1$$

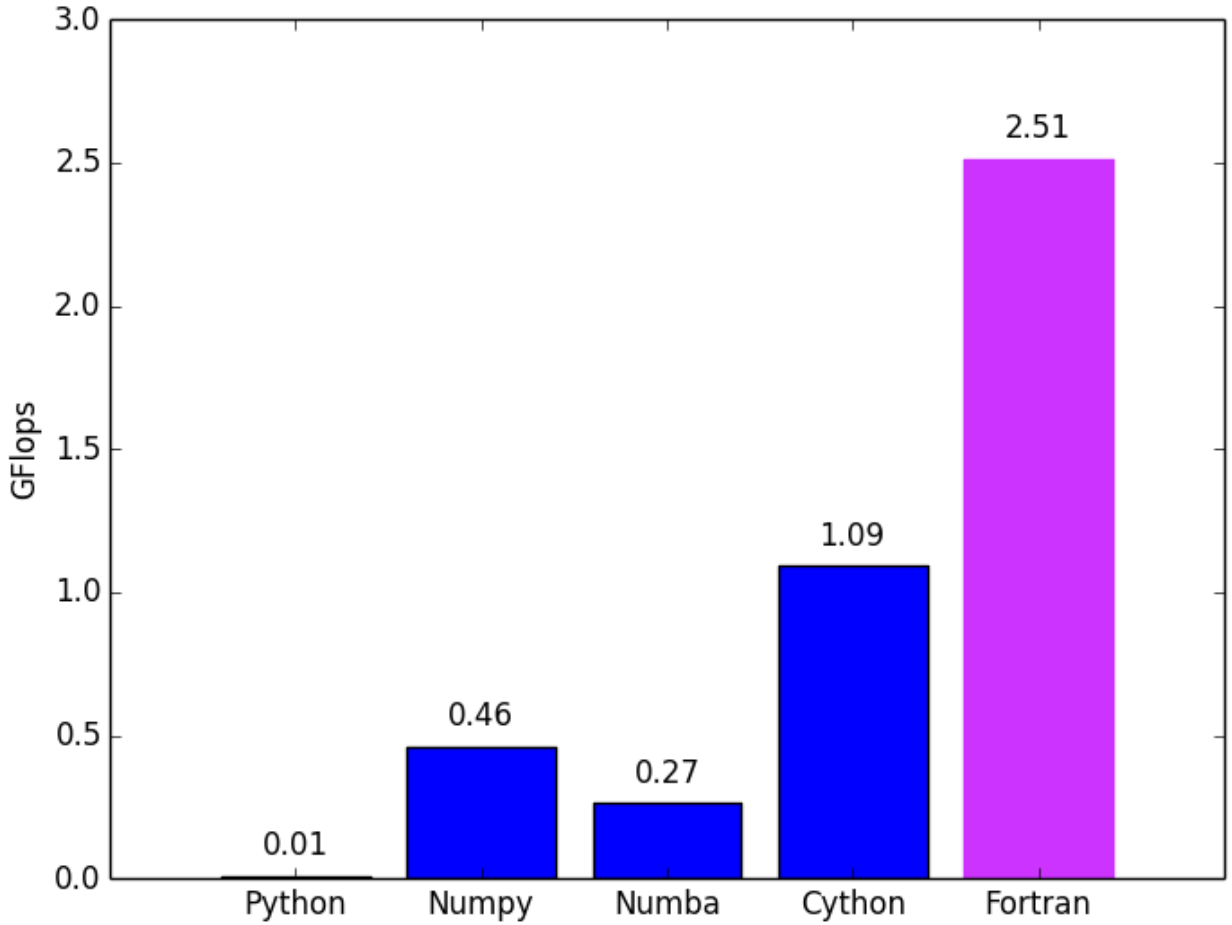
Kernel is able to exploit the computational peak performance of CPU or GPU!

Runtime estimations from the model:

$$T_{CPU} = \frac{6,500,000,000}{28,800,000,000 \frac{1}{s}} = 0.23s \qquad T_{CPUseriell} = \frac{6,500,000,000}{4,800,000,000 \frac{1}{s}} = 1.35s$$

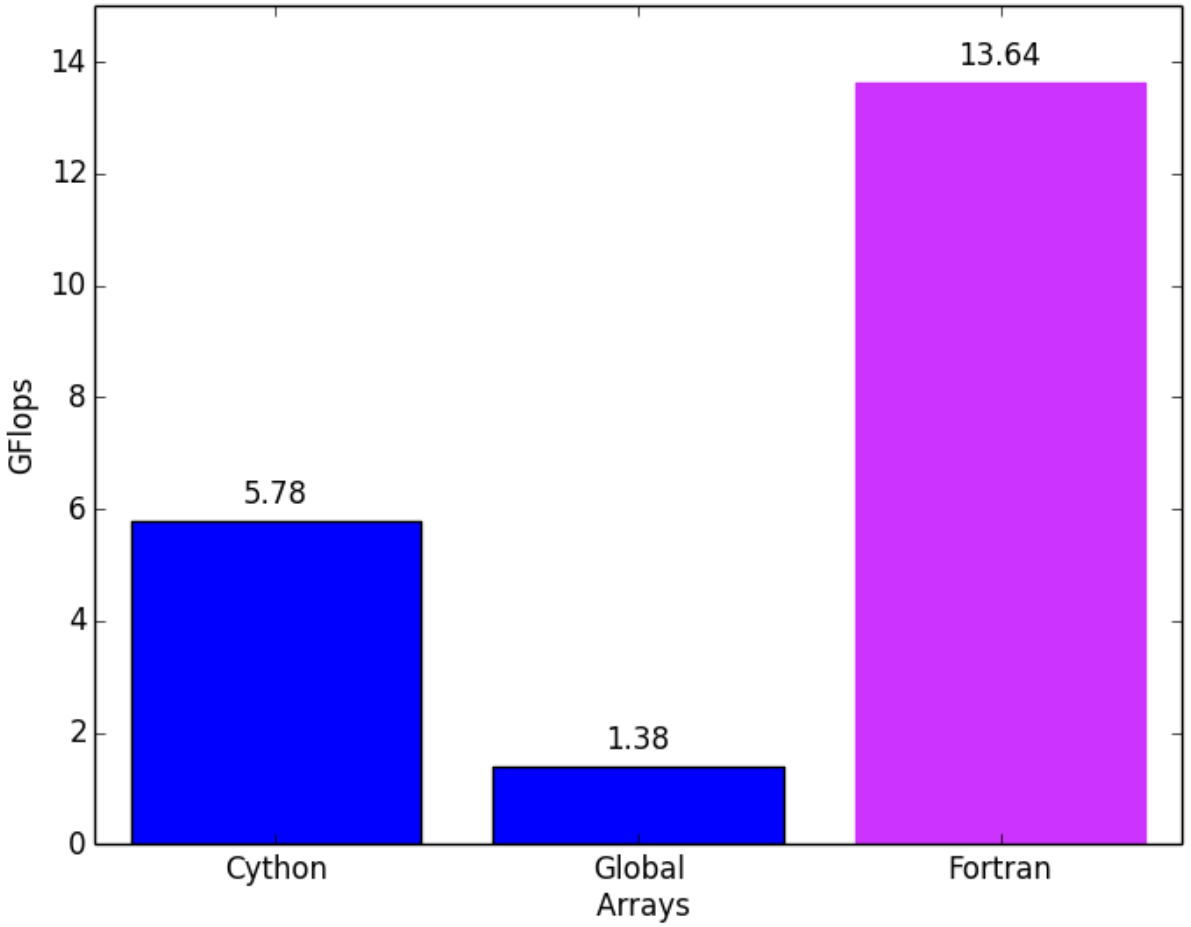
$$T_{GPU} = \frac{6,500,000,000}{515,000,000,000 \frac{1}{s}} = 0.0126s$$

Performance Measurements: CPU Single-Core Performance



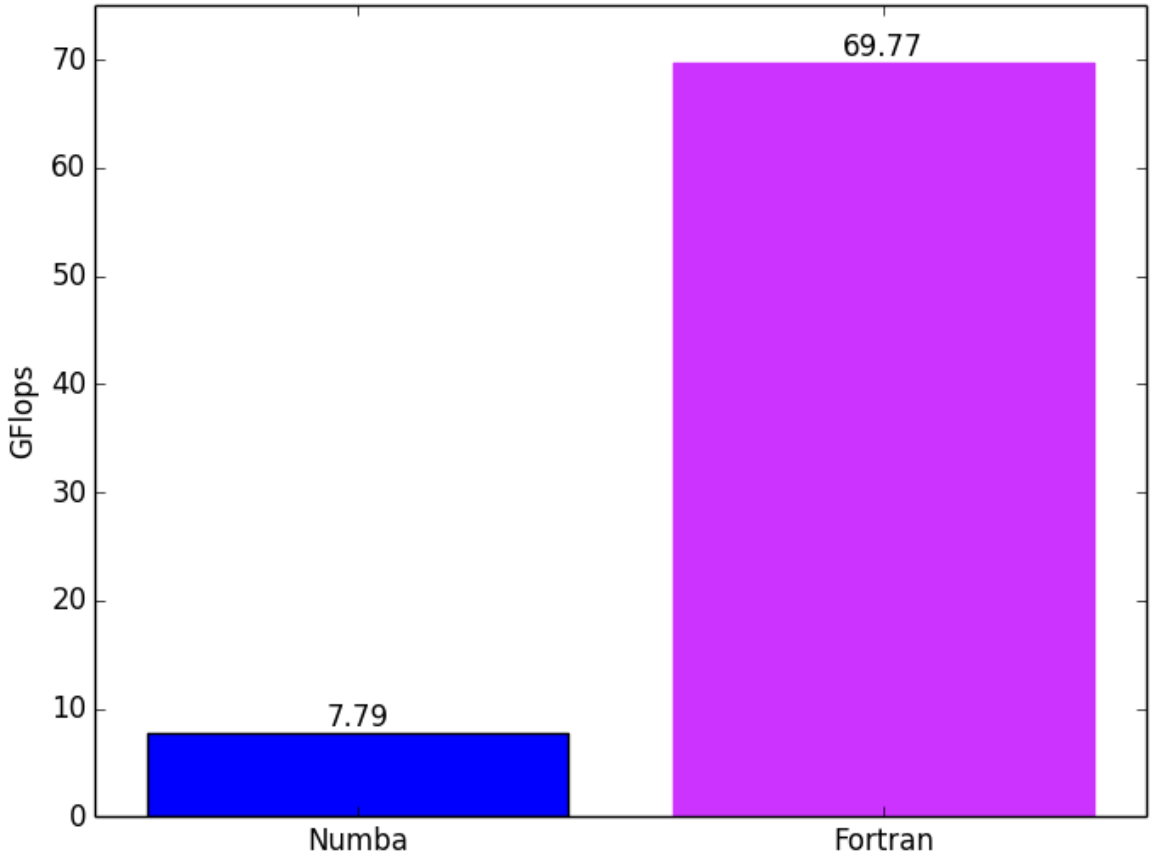
Implementation	Time
Python	638s
NumPy	12.99s
Numba	22.62s
Cython	5.50s
Fortran	2.38s

Performance Measurements: CPU Multi-Core Performance



Implementation	Time	Speed-Up
Cython	1.03s	5.3
Global Arrays	4.34s	3.7
Fortran	0.440s	5.4

Performance Measurements: GPU Performance



Implementation	Time
Numba	0.770s
Fortran + OpenACC	0.086s

Conclusions

- Python: high productivity for parallel programming but performance deficits compared with low-level languages like Fortran or C
- High parallel programming abstraction level for
  - CPU: Python Bindings for Global Array Toolkit
  - GPU: Numba
- Distinctly higher performance for
  - Cython with OpenMP
- → Acceptable performance with Cython but C-style programming
- Expectation: Communities behind NumPy, Numba, Cython and Global Array projects, e.g., will further improve their solutions and make Python increasingly more efficient on modern parallel hardware.

Many thanks for your attention!

Questions?

Dr.-Ing. Achim Basermann

German Aerospace Center (DLR)

Simulation and Software Technology

Department Distributed Systems and Component Software

Team High Performance Computing

Achim.Basermann@dlr.de

http://www.DLR.de/sc