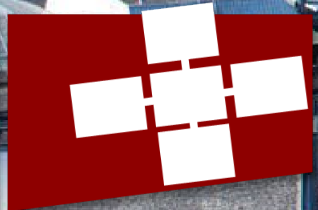
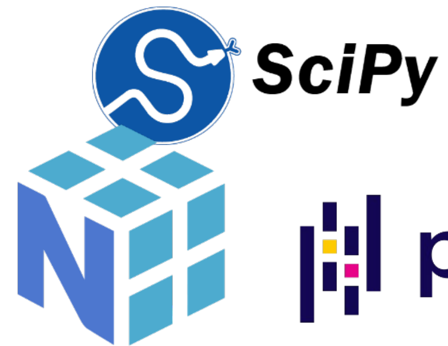
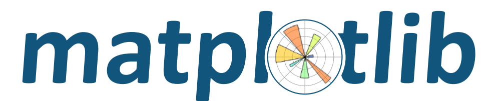
ALEXANDROS NIKOLAOS ZIOGAS, TAL BEN-NUN, TIMO SCHNEIDER, TORSTEN HOEFLER

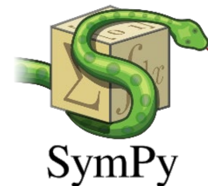
NPBench: A Benchmarking Suite for High-Performance NumPy



Python: The Scientific Language

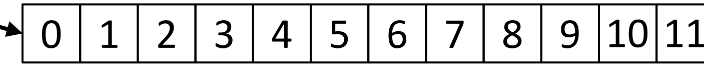






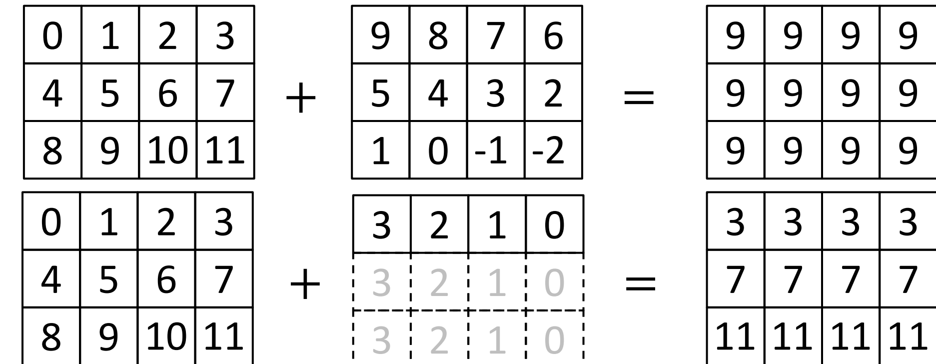
```
A = np.ndarray((3, 4), dtype=np.float64)
```

buffer
dtype
shape
strides = $(32, 8)$




```
B = np.ndarray((3, 4), dtype=np.float64)
A + B
```

```
x = np.ndarray((4,), dtype=np.float64)
A + x
```

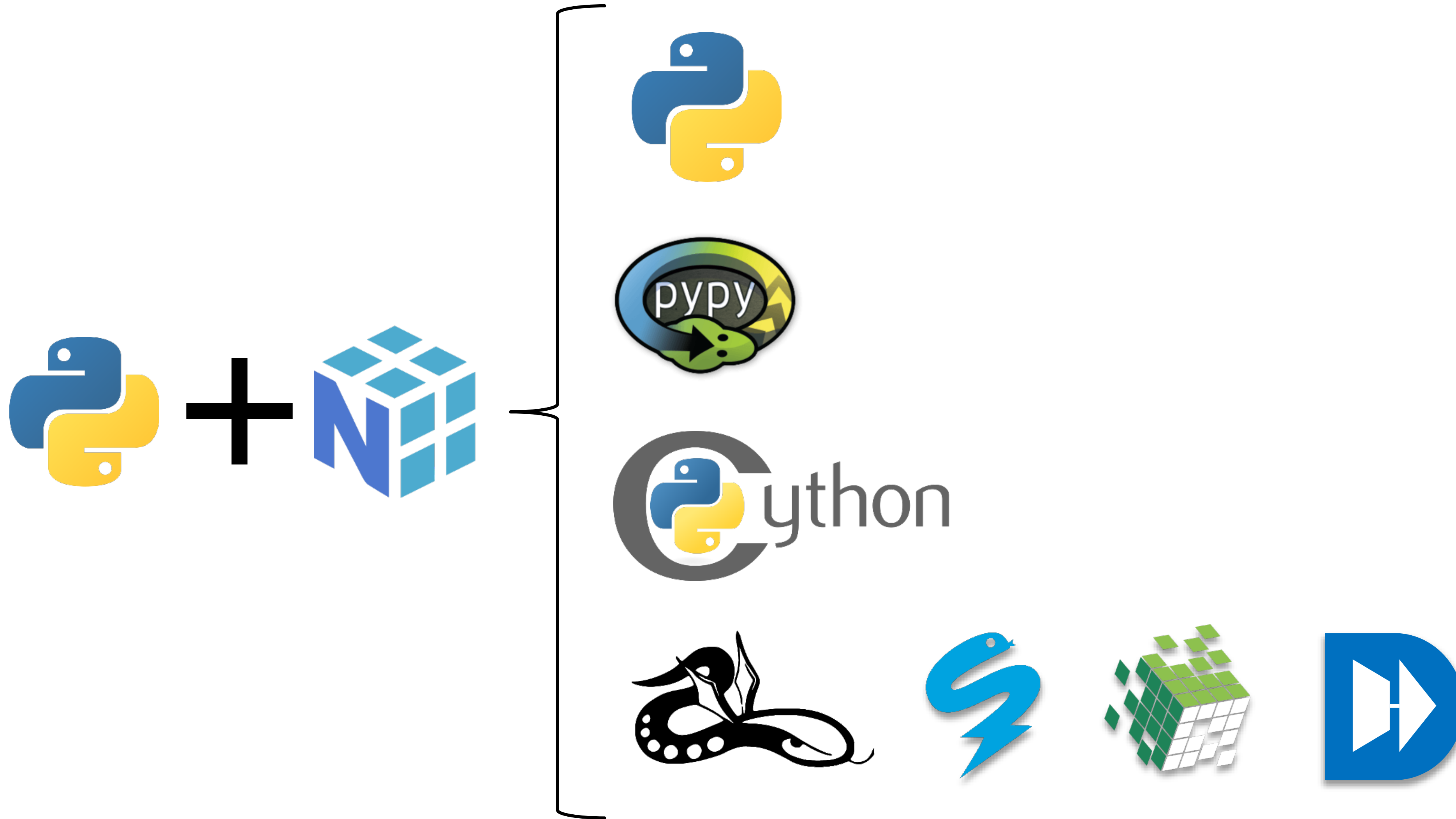



```
A @ B
np.linalg.cholesky(A)
```

```
cblas_dgemm
LAPACKE_dpotrf
```

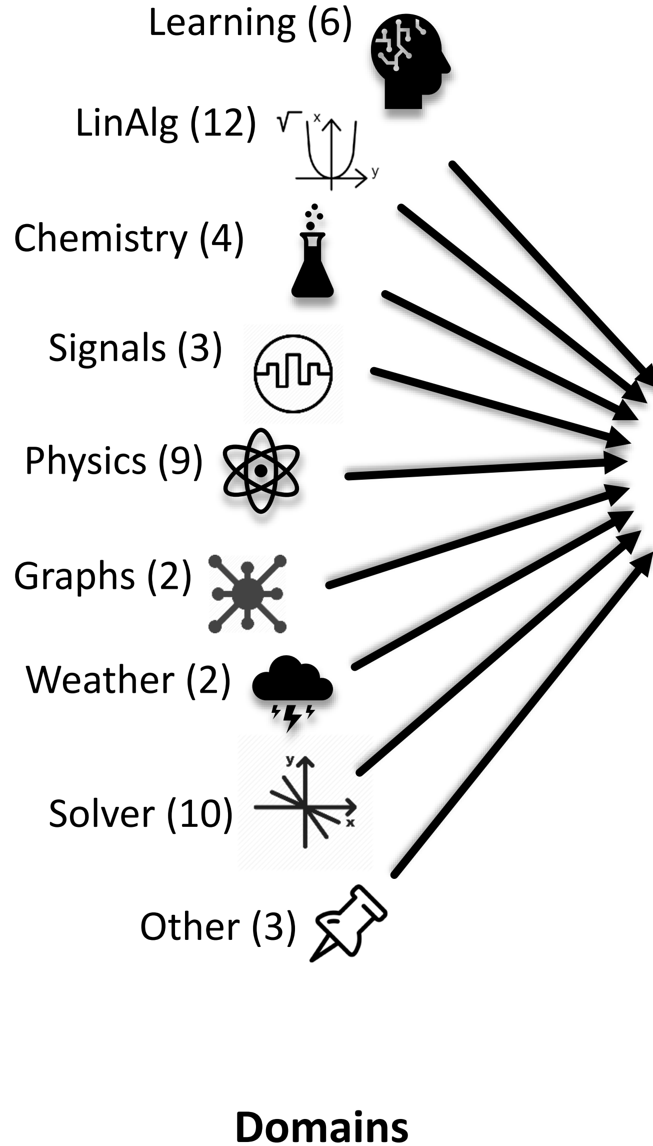
```
for i in range(3):
    for j in range(4):
        y += f(A[i, j])
```

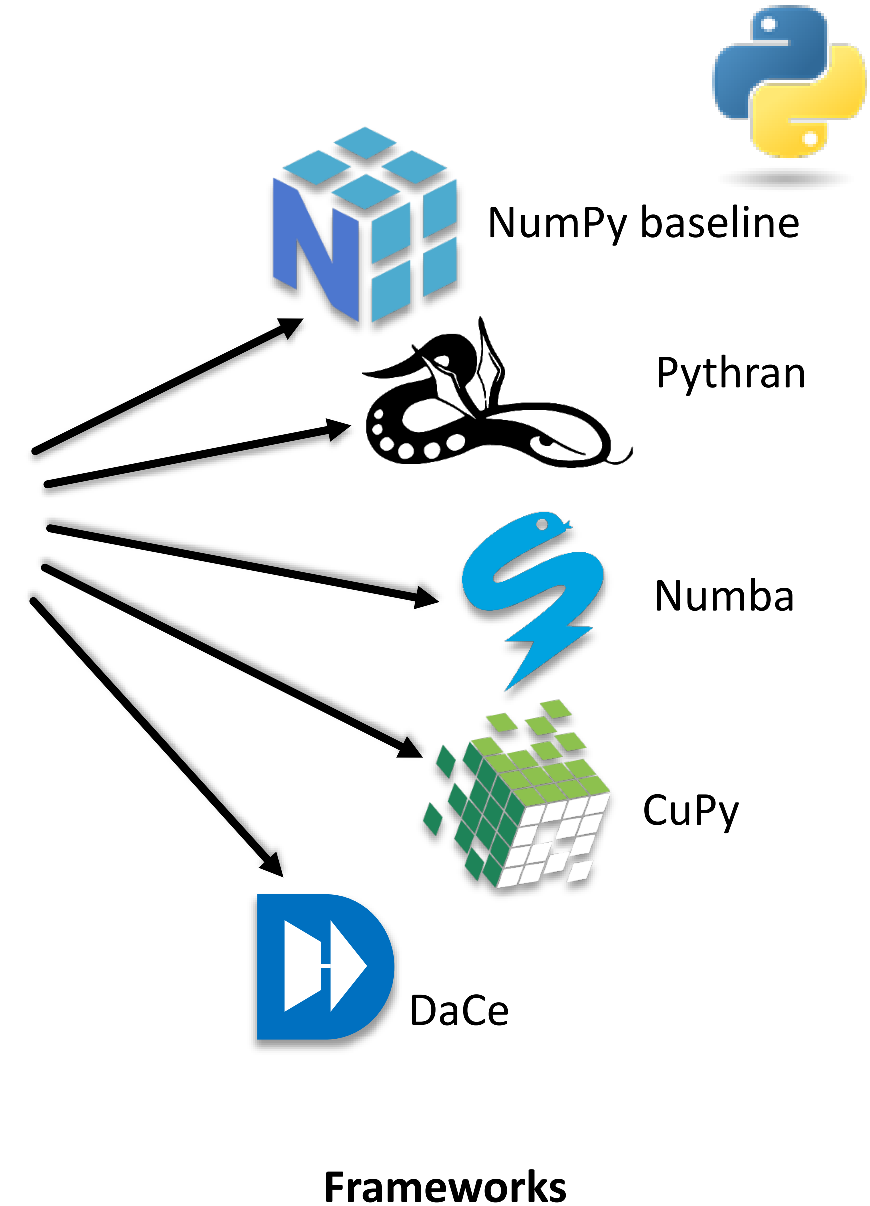
Many choices for executing Python and NumPy codes …



Many choices for executing Python and NumPy codes …

Let's make a benchmark!





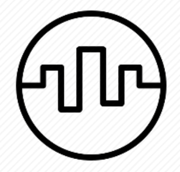
Domains **Infrastructure and Metrics** **Frameworks**

Scientific Domains

Chemistry

azimnaiv
azimhist **(PyFAI)**
doitgen
nussinov **(Polybench/C)**

Signals

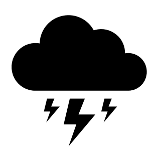


sthamfft **(KTH)**
clipping **(Cython)**
deriche **(Polybench/C)**

Graphs

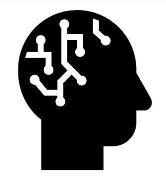
spmv
floydwar **(Polybench/C)**

Weather



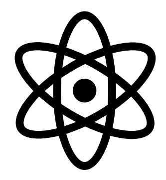
vadv
hdiff **(GT4Py)**

Learning



conv2d
softmax
mlp
lenet
resnet
correlate
covarian **(Polybench/C)**

Physics

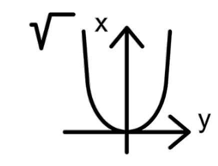


cavtflow
chanflow **(CFD Python)**
nbody **(nbody-python)**
coninteg
sselfeng **(OMEN)**
jacobi1d
jacobi2d
heat3d
fdtd2d **(Polybench/C)**

Other

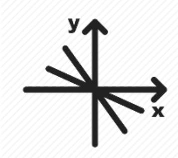
mandel1
mandel2 **(From Python to Numpy)**
crc16 **(oysstu)**

LinAlg



gemm
gemver
gesummv
symm
syr2k
syrk
trmm
2mm
3mm
atax
mvt **(Polybench/C)**
npgofast **(Numba)**

Solvers



seidel2d
durbin
trisolv
adi
bicg
gramschm
lu
ludcmp
cholesky
cholesky2 **(Polybench/C)**

NumPy Coverage: Simple Examples

```
def clipping(array_1, array_2, a, b, c):
    return np.clip(array_1, 2, 10) * a + array_2 * b + c
```

NumPy methods

array operations

array-scalar operations (broadcasting)

NumPy methods' keyword arguments

```
def batchnorm2d(x, eps=1e-5):
    mean = np.mean(x, axis=0, keepdims=True)
    std = np.std(x, axis=0, keepdims=True)
    return (x - mean) / np.sqrt(std + eps)
```

NumPy Coverage: Indexing

```
p[1:-1, 1:-1] = (((pn[1:-1, 2:] + pn[1:-1, 0:-2]) * dy**2 +
                  (pn[2:, 1:-1] + pn[0:-2, 1:-1]) * dx**2) /
                 (2 * (dx**2 + dy**2)) - dx**2 * dy**2 /
                 (2 * (dx**2 + dy**2)) * b[1:-1, 1:-1])
```

slicing start: stop: step

```
output[:, i, j, :] = np.sum(
    input[:, i:i + K, j:j + K, :, np.newaxis] *
    weights[np.newaxis, :, :, :],
    axis=(1, 2, 3),)
```

newaxis: adds dimension with length 1
used with broadcasting

```
for n in range(maxiter):
    I = np.less(abs(Z), horizon)
    N[I] = n
    Z[I] = Z[I]**2 + C[I]
```

advanced indexing with
boolean or integer arrays

NumPy Coverage: Routines and Sub-Modules

```
for z in int_pts:
    Tz = np.zeros((NR, NR), dtype=np.complex128)
    for n in range(slab_per_bc + 1):
        zz = np.power(z, slab_per_bc / 2 - n)
        Tz += zz * Ham[n]
    if NR == NM:
        X = np.linalg.inv(Tz)
    else:
        X = np.linalg.solve(Tz, Y)
```

Adapting Polybench/C

```
void cholesky(DATA_TYPE A[_PB_N][_PB_N]) {
    int i, j, k;
    for (i = 0; i < _PB_N; i++) {
        for (j = 0; j < i; j++) {
            for (k = 0; k < j; k++) {
                A[i][j] -= A[i][k] * A[j][k];
            }
            A[i][j] /= A[j][j];
        }
        for (k = 0; k < i; k++) {
            A[i][i] -= A[i][k] * A[i][k];
        }
        A[i][i] = SQRT_FUN(A[i][i]);
    }
}
```

```
def cholesky(A):
    for i in range(A.shape[0]):
        for j in range(i):
            for k in range(j):
                A[i, j] -= A[i, k] * A[j, k]
            A[i, j] /= A[j, j]
        for k in range(i):
            A[i, i] -= A[i, k] * A[i, k]
        A[i, i] = np.sqrt(A[i, i])
```

Adapting Polybench/C

```
def cholesky(A):
    for i in range(A.shape[0]):
        for j in range(i):
            for k in range(j):
                A[i, j] -= A[i, k] * A[j, k]
            A[i, j] /= A[j, j]
        for k in range(i):
            A[i, i] -= A[i, k] * A[i, k]
        A[i, i] = np.sqrt(A[i, i])
```

```
def cholesky(A):
    A[0, 0] = np.sqrt(A[0, 0])
    for i in range(1, A.shape[0]):
        for j in range(i):
            A[i, j] -= np.dot(A[i, :j], A[j, :j])
            A[i, j] /= A[j, j]
        A[i, i] -= np.dot(A[i, :i], A[i, :i])
        A[i, i] = np.sqrt(A[i, i])
```

Adapting Polybench/C

```python
def cholesky(A):
    for i in range(A.shape[0]):
        for j in range(i):
            for k in rang
                A[i, j]
            A[i, j] /= A[
        for k in range(i):
            A[i, i] -= A[i, k] * A[i, k]
        A[i, i] = np.sqrt(A[i, i])
```

```python
def cholesky(A):
    A[0, 0] = np.sqrt(A[0, 0])
    for i in range(1, A.shape[0]):
                      i):
                  np.dot(A[i, :j], A[j, :j])
            A[i, j] /= A[j, j]
        A[i, i] -= np.dot(A[i, :i], A[i, :i])
        A[i, i] = np.sqrt(A[i, i])
```

```python
def cholesky(A):
    A[:] = np.linalg.cholesky(A) + np.triu(A, k=1)
```

Python and NumPy Accelerators

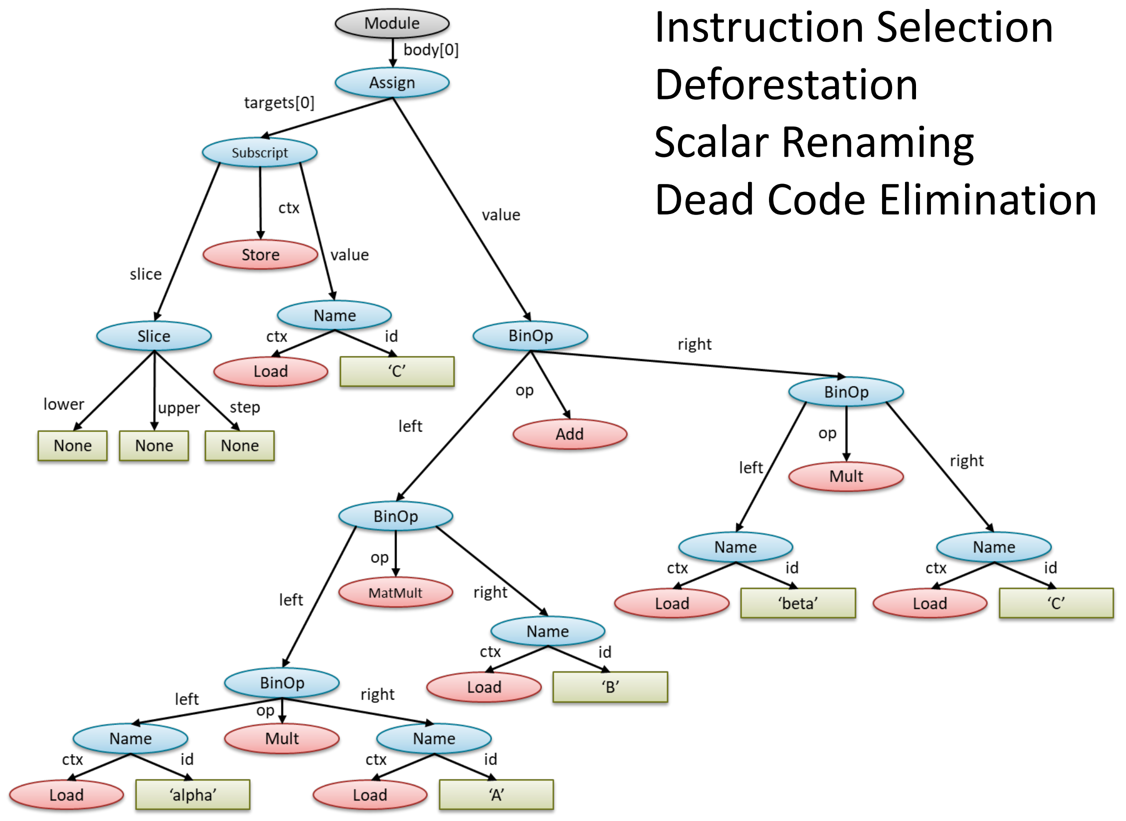
Clinostomus funduloides Rosyside Dace by Brian Gratwicke licensed under CC BY 2.0

Pythran

Type annotations

```python
# pythran export kernel(
    float64, float64, float64[:,:],
    float64[:,:], float64[:,:])
def gemm(alpha, beta, A, B, C):
    C[:] = alpha * A @ B + beta * C
```

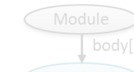
Python AST-based IR



- Interprocedural Constant Folding
- For-Based-Loop Unrolling
- Forward Substitution
- Instruction Selection
- Deforestation
- Scalar Renaming
- Dead Code Elimination

Pythran

Python AST-based IR

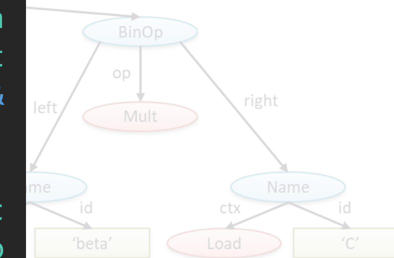


Interprocedural Constant Folding
For-Based-Loop Unrolling
Forward Substitution
Instruction Selection
Deforestation
Scalar Renaming
Dead Code Elimination

Type annotations

```
# pythran export kernel(
    float64, float64, float6
    float64[:,:], float64[:,
def gemm(alpha, beta, A, B,
    C[:] = alpha * A @ B + b
```

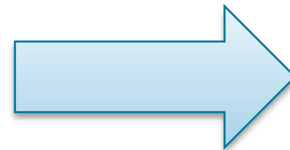
```
namespace __pythran_gemm
{
  struct kernel
  {
    typedef void callable;
    template <typename argument_type0 , typename argument_type1 , typenam
e argument_type2 , typename argument_type3 , typename argument_type4 >
  typename kernel::type<argument_type0, argument_type1, argument_type2, a
rgument_type3, argument_type4>::result_type kernel::operator()(argument_t
ype0&& alpha, argument_type1&& beta, argument_type2&& C, argument_type3&&
 A, argument_type4&& B) const
  {
    C[pythonic::types::contiguous_slice(pythonic::builtins::None,pythonic
::builtins::None)] = pythonic::operator_::add(pythonic::operator_::functo
r::matmul()(pythonic::operator_::mul(alpha, A), B), pythonic::operator_::
mul(beta, C));
    return pythonic::builtins::None;
  }
}
```



Numba

JIT decorator

```python
import numba as nb
@nb.jit(nopython=True, parallel=True,
        fastmath=True)
def gemm(alpha, beta, A, B, C):
    C[:] = alpha * A @ B + beta * C
```



Numba IR

```
------------------------IR DUMP: nopython_mode_parallel------------------------
label 0:
    alpha = arg(0, name=alpha)              ['alpha']
    beta = arg(1, name=beta)                ['beta']
    C = arg(2, name=C)                      ['C']
    A = arg(3, name=A)                      ['A']
    B = arg(4, name=B)                      ['B']
    $6binary_multiply.2 = alpha * A         ['$6binary_multiply.2'
    $10binary_matrix_multiply.4 = $6binary_multiply.2 <built-i
['$10binary_matrix_multiply.4', '$6binary_multiply.2', 'B']
    $16binary_multiply.7 = beta * C         ['$16binary_multiply.7
    $18binary_add.8 = $10binary_matrix_multiply.4 + $16binar
['$10binary_matrix_multiply.4', '$16binary_multiply.7', '$18bi
    $const22.10 = const(NoneType, None)     ['$const22.10']
    $const24.11 = const(NoneType, None)     ['$const24.11']
    $26build_slice.12 = global(slice: <class 'slice'>) ['$26build_s
    $26build_slice.13 = call $26build_slice.12($const22.10, $co
args=(Var($const22.10, gemm_numba.py:14), Var($const24.11, gemm_numba.py:14)), kws=(),
vararg=None) ['$26build_slice.12', '$26build_slice.13', '$const22.10', '$const24.11']
    C[$26build_slice.13] = $18binary_add.8   ['$18binary_add.8', '$26build_slice.13', 'C']
    $const30.14 = const(NoneType, None)     ['$const30.14']
    $32return_value.15 = cast(value=$const30.14) ['$32return_value.15', '$const30.14']
    return $32return_value.15               ['$32return_value.15']
```

- Loop Parallelization
- Loop Fusion
- Shortcut Deforestation
- CFG Simplification
- NumPy Canonicalization
- Array Analysis
- Copy Propagation
- Dead Code Eliminiation

Numba

Numba IR

JIT decorator

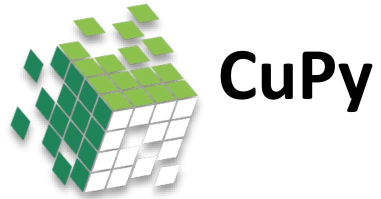
```
import numba as nb
@nb.jit(nopython=True, parallel
        fastmath=True)

def gemm(alpha, beta, A, B, C):
    C[:] = alpha * A @ B + beta
```

```
vector.body:                                          ; preds =
%vector.body.preheader, %vector.body
  %vector.recur = phi <4 x i64> [ %37, %vector.body ], [
%vector.recur.init, %vector.body.preheader ]
  %vec.ind = phi <4 x i64> [ %vec.ind.next.3, %vector.body ], [
%induction, %vector.body.preheader ]
  %niter = phi i64 [ %niter.nsub.3, %vector.body ], [ %unroll_iter,
%vector.body.preheader ]
  %7 = add <4 x i64> %vec.ind, <i64 1, i64 1, i64 1, i64 1>
  %8 = shufflevector <4 x i64> %vector.recur, <4 x i64> %7, <4 x i32>
<i32 3, i32 4, i32 5, i32 6>
  %9 = add <4 x i64> %8, %broadcast.splat22
  %10 = getelementptr double, double*
%arg._10binary__matrix__multiply_4.4, <4 x i64> %9
  %wide.masked.gather = call <4 x double>
@llvm.masked.gather.v4f64.v4p0f64(<4 x double*> %10, i32 8, <4 x i1>
<i1 true, i1 true, i1 true, i1 true>, <4 x double> undef)
  %11 = add <4 x i64> %8, %broadcast.splat24
  %12 = getelementptr double, double* %arg.C.4, <4 x i64> %11
  %wide.masked.gather25 = call <4 x double>
@llvm.masked.gather.v4f64.v4p0f64(<4 x double*> %12, i32 8, <4 x i1>
<i1 true, i1 true, i1 true, i1 true>, <4 x double> undef)
  %13 = fmul fast <4 x double> %wide.masked.gather25,
%broadcast.splat27
  %14 = fadd fast <4 x double> %13, %wide.masked.gather
```

```
.11, gemm_numba.py:14)), kws=(),
st22.10', '$const24.11']
.8', '$26build_slice.13', 'C']
n_value.15', '$const30.14']
```

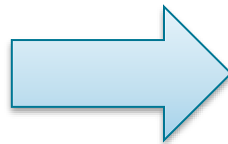
- Loop Parallelization
- Loop Fusion
- Shortcut Deforestation
- CFG Simplification
- NumPy Canonicalization
- Array Analysis
- Copy Propagation
- Dead Code Eliminiation



CuPy

```python
~~import numpy as np~~
import cupy as np

    C[:] = alpha * A @ B + beta * C
```

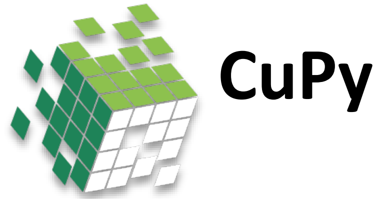


```python
cdef class ndarray:

    def __add__(x, y):
        if isinstance(y, ndarray):
            return _math._add(x, y)
        elif _should_use_rop(x, y):
            return NotImplemented
        else:
            return numpy.add(x, y)

    def __mul__(x, y):
        if isinstance(y, ndarray):
            return _math._multiply(x, y)
        elif _should_use_rop(x, y):
            return NotImplemented
        else:
            return numpy.multiply(x, y)

    def __matmul__(x, y):
        if not isinstance(y, ndarray) and _should_use_rop(x, y):
            return NotImplemented
        else:
            return cupy.linalg._product.matmul(x, y)
```



CuPy

```
cdef function.Function _get_simple_elementwise_kernel(
        tuple params, tuple arginfos, str operation, str name,
        _TypeMap type_map, str preamble, str loop_prep='', str after_loop='',
        tuple options=()):
    module_code = string.Template('''
    ${typedef_preamble}
    ${preamble}
    extern "C" __global__ void ${name}(${params}) {
      ${loop_prep};
      CUPY_FOR(i, _ind.size()) {
        _ind.set(i);
        ${operation};
      }
      ${after_loop};
    }
    ''').substitute(
        typedef_preamble=type_map.get_typedef_code(),
        params=_get_kernel_params(params, arginfos),
        operation=operation,
        name=name,
        preamble=preamble,
        loop_prep=loop_prep,
        after_loop=after_loop)
    module = compile_with_cache(module_code, options)
    return module.get_function(name)
```

```
cdef class ndarray:

    def __add__(x, y):
        if isinstance(y, ndarray):
            return _math._add(x, y)
        elif _should_use_rop(x, y):
            return NotImplemented
        else:
            return numpy.add(x, y)

    def __mul__(x, y):
        if isinstance(y, ndarray):
            return _math._multiply(x, y)
        elif _should_use_rop(x, y):
            return NotImplemented
        else:
            return numpy.multiply(x, y)

    def __matmul__(x, y):
        if not isinstance(y, ndarray) and _should_use_rop(x, y):
            return NotImplemented
        else:
            return cupy.linalg._product.matmul(x, y)
```

CuPy

```
cdef function.Function _get_simple_elementwise_kernel(
        tuple params, tuple arginfos, str operation, str name,
        _TypeMap type_map, str preamble, str loop_prep='', str after_loop='',
        tuple options=()):
    module_code = string.Template('''
    ${typedef_preamble}
    ${preamble}
    extern "C" __global__ void ${name}(${params}) {
      ${loop_prep};
      CUPY_FOR(i, _ind.size()) {
        _ind.set(i);
        ${operation};
      }
      ${after_loop};
    }
    ''').substitute(
        typedef_preamble=type_map.get_typedef_code(),
        params=_get_kernel_params(params, arginfos),
        operation=operation,
        name=name,
        preamble=preamble,
        loop_prep=loop_prep,
        after_loop=after_loop)
    module = compile_with_cache(module_code, options)
    return module.get_function(name)
```

```
cdef class ndarray:

    def __add__(x, y):
        if isinstance(y, ndarray):
            return _math._add(x, y)
        elif _should_use_rop(x, y):

    def __matmul__(x, y):
        if not isinstance(y, ndarray) and _should_use_rop(x, y):
            return NotImplemented
        else:
            return cupy.linalg._product.matmul(x, y)
```

```
elif dtype == numpy.float64:
    cublas.dgemmStridedBatched(
        handle,
        0,  # transa
        0,  # transb
        n, m, ka, one.ctypes.data,
        a.data.ptr, lda, strideA,
        b.data.ptr, ldb, strideB,
        zero.ctypes.data, out_view.data.ptr, ldout, strideC,
        batchCount)
```

DaCe

Symbolic sizes

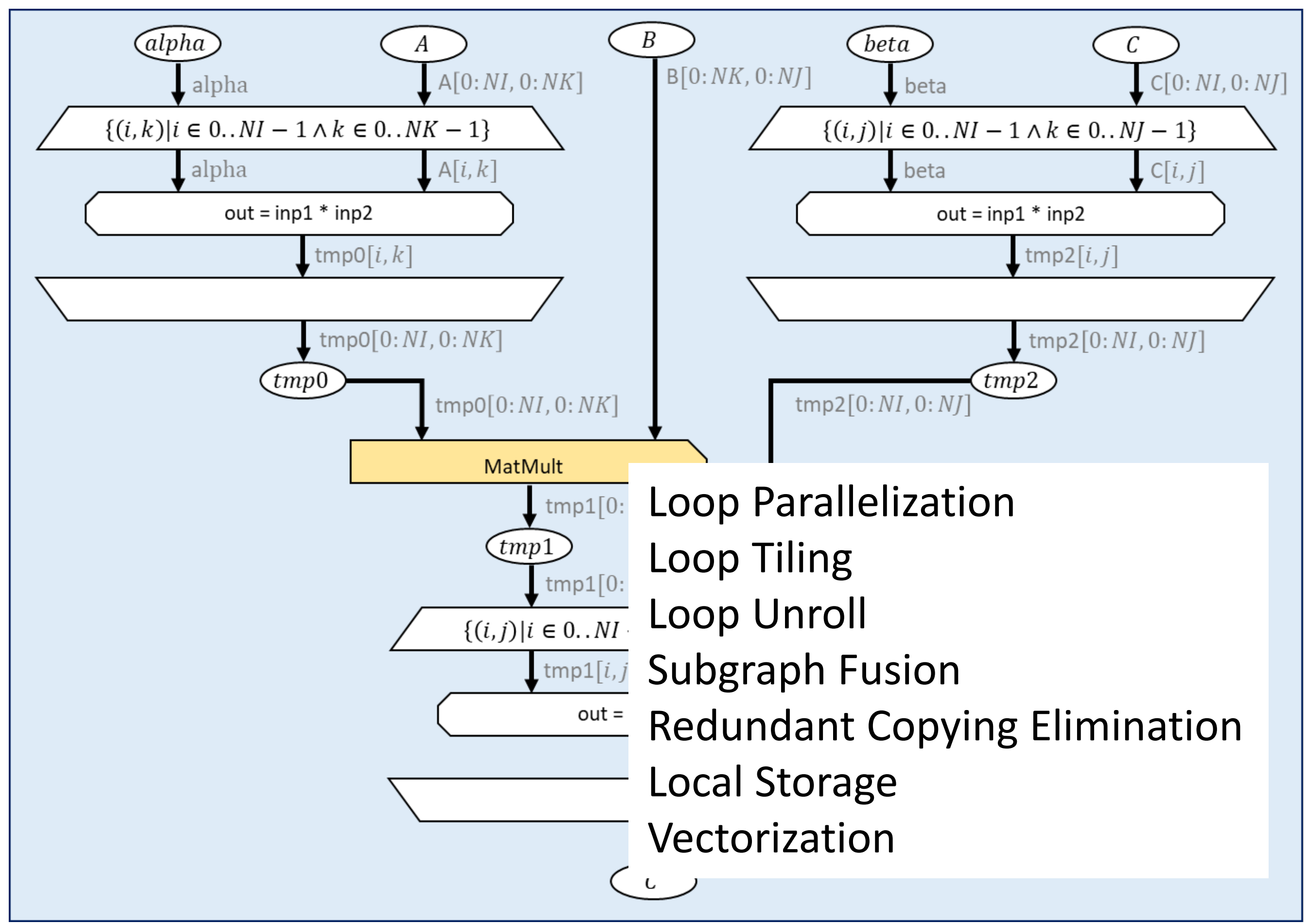
```
import dace as dc
NI, NJ, NK = (dc.symbol(s, dtype=dc.int64)
              for s in ('NI', 'NJ', 'NK'))
@dc.program
def gemm(alpha: dc.float64,
         beta: dc.float64,
         C: dc.float64[NI, NJ],
         A: dc.float64[NI, NK],
         B: dc.float64[NK, NJ]):

    C[:] = alpha * A @ B + beta * C
```

DaCe program decorator

Type annotations

Dataflow-based IR (SDFG)



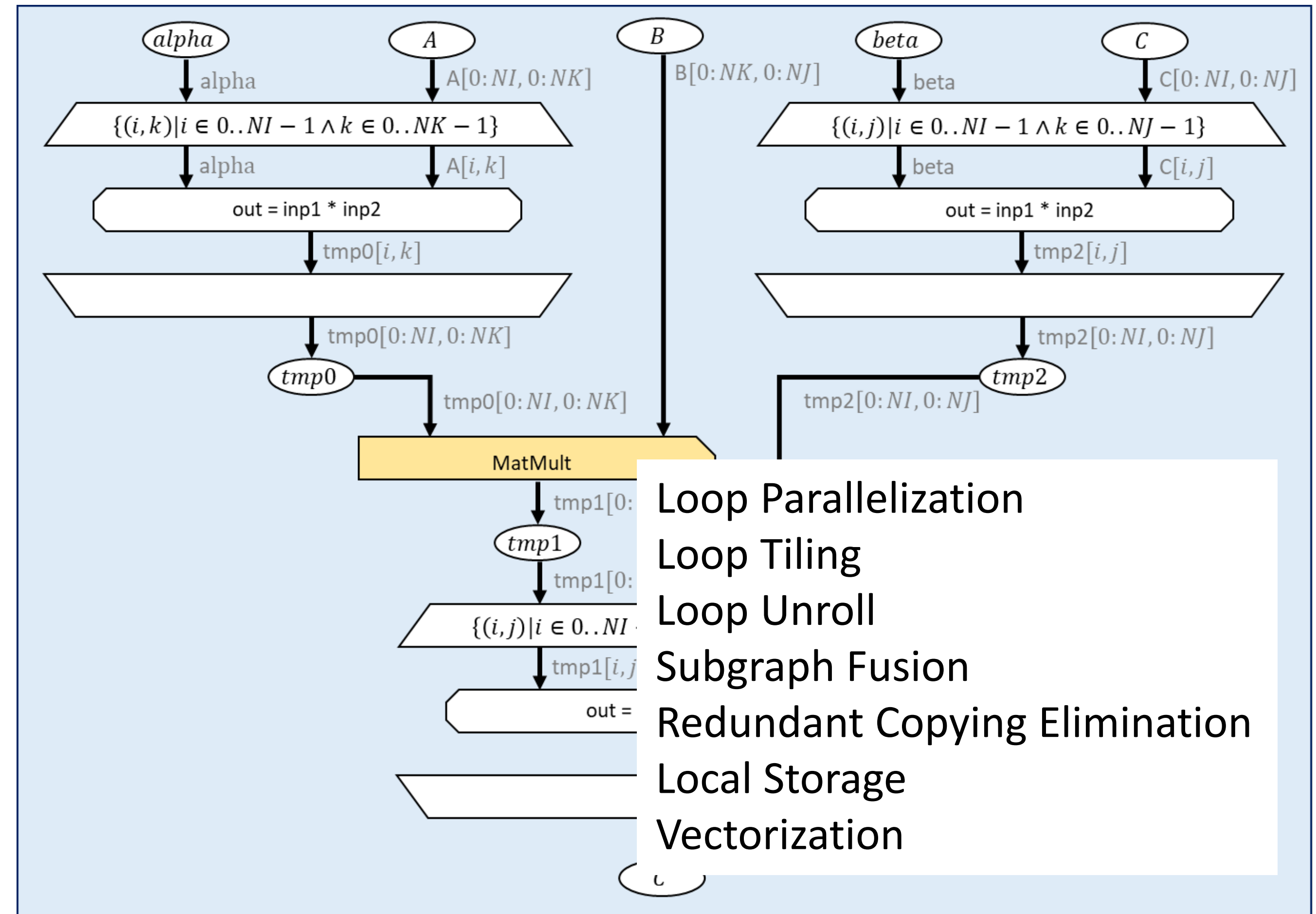
```
double *__tmp0;
__tmp0 = new double DACE_ALIGN(64)[(NI * NK)];
double *__tmp1;
__tmp1 = new double DACE_ALIGN(64)[(NI * NJ)];
{
    #pragma omp parallel for
    for (auto __i0 = 0; __i0 < NI; __i0 += 1) {
        for (auto __i1 = 0; __i1 < NK; __i1 += 1) {
            {
                double __in1 = alpha;
                double __in2 = A[((NK * __i0) + __i1)];
                double __out;

                ///////////////////
                // Tasklet code (_Mult_)
                __out = (__in1 * __in2);
                ///////////////////

                __tmp0[((NK * __i0) + __i1)] = __out;
            }
        }
    }
}
{
    double* _b = &B[0];
    double* _a = &__tmp0[0];
    double* _c = __tmp1;

    ///////////////////
    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                NJ, NI, NK, double(1.0), _b, NJ, _a, NK,
                double(0.0), _c, NJ);
    ///////////////////
```

Dataflow-based IR (SDFG)



```
double *__tmp0;
__tmp0 = new double DACE_ALIGN(64)[(NI * NK)];
double *__tmp1;
__tmp1 = new double DACE_ALIGN(64)[(NI * NJ)];
{
    #pragma omp parallel for
    for (auto __i0 = 0; __i0 < NI; __i0 += 1) {
        for (auto __i1 = 0; __i1 < NK; __i1 += 1) {
            {
                double __in1 = alpha;
                double __in2 = A[((NK * __i0) + __i1)];
                double __out;

                ///////////////////
                // Tasklet code (_Mult_)
                __out = (__in1 * __in2);
                ///////////////////

                __tmp0[((NK * __i0) + __i1)] = __out;
            }
        }
    }
}
{
    double* _b = &B[0];
    double* _a = &__tmp0[0];
    double* _c = __tmp1;

    ///////////////////
    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                NJ, NI, NK, double(1.0), _b, NJ, _a, NK,
                double(0.0), _c, NJ);
    ///////////////////
```

```
__dace_runkernel__Mult__map_0_0_0(__state, A, __state->__tmp0, alpha);
{
    double* _b = &B[0];
    double * _a = &__state->__tmp0[0];
    double* _c = __state->__tmp1;

    ///////////////////
    int __dace_current_stream_id = 0;
    cudaStream_t __dace_current_stream = __state->gpu_context->streams[
        __dace_current_stream_id];
    const int __dace_cuda_device = 0;
    cublasHandle_t &__dace_cublas_handle = __state->cublas_handle.Get
        (__dace_cuda_device);
    cublasSetStream(__dace_cublas_handle, __dace_current_stream);
    cublasDgemm(__dace_cublas_handle,
                CUBLAS_OP_N, CUBLAS_OP_N,
                NJ, NI, NK,
                __state->cublas_handle.Constants(
                    __dace_cuda_device).DoublePone(),
                (double*)_b, NJ,
                (double*)_a, NK,
                __state->cublas_handle.Constants(
                    __dace_cuda_device).DoubleZero(),
                (double*)_c, NJ);
    ///////////////////

}
__dace_runkernel__Mult__map_0_0_8(__state, C, __state->__tmp1, beta);
cudaStreamSynchronize(__state->gpu_context->streams[0]);
```

Evaluation

Measuring Burmese Python by Florida Fish and Wildlife licensed under CC BY-NC-ND 2.0

Adapting for Compatibility

```
import numpy as np
def floyd_warshall(path):
    for k in range(path.shape[0]):
        path[:] = np.minimum(path[:], np.add.outer(path[:, k], path[k, :]))
```

Unsupported by Numba

Rewritten in loop-form

```
import numpy as np
import numba as nb
@nb.jit(nopython=True, parallel=False, fastmath=True)
def floyd_warshall(path):
    for k in range(path.shape[0]):
        # path[:] = np.minimum(path[:], np.add.outer(path[:, k], path[k, :]))
        for i in range(path.shape[0]):
            path[i, :] = np.minimum(path[i, :], path[i, k] + path[k, :])
```

```
import numpy as np
def azimint_hist(data, radius, npt):
    histu = np.histogram(radius, npt)[0]
    histw = np.histogram(radius, npt, weights=data)[0]
    return histw / histu
```

Unsupported by Numba

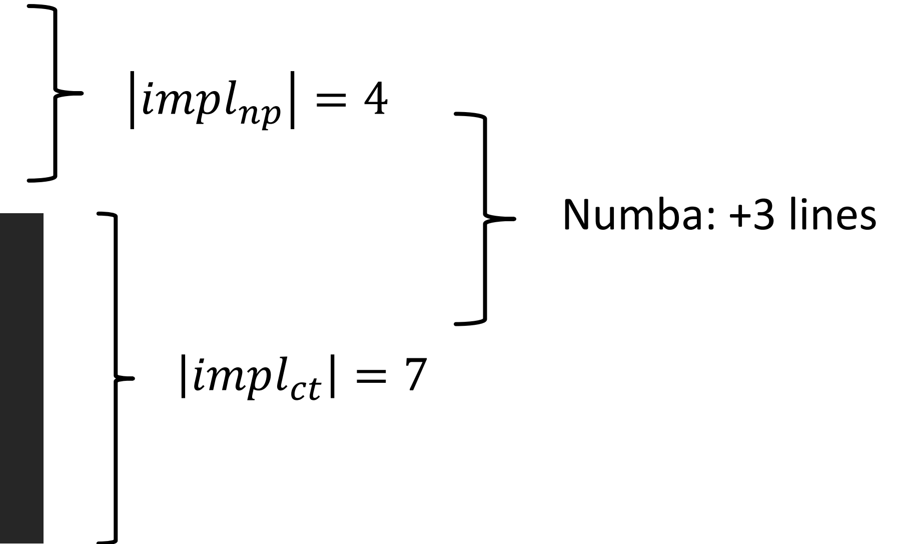
Rewritten as a separate method

```
import numpy as np
import numba as nb
@nb.jit(nopython=True, parallel=False, fastmath=True)
def azimint_hist(data, radius, npt):
    histu = np.histogram(radius, npt)[0]
    # histw = np.histogram(radius, npt, weights=data)[0]
    histw = histogram(radius, npt, weights=data)[0]
    return histw / histu
```

Measuring Productivity

```
import numpy as np
def floyd_warshall(path):
    for k in range(path.shape[0]):
        path[:] = np.minimum(path[:], np.add.outer(path[:, k], path[k, :]))
```

```
import numpy as np
import numba as nb
@nb.jit(nopython=True, parallel=False, fastmath=True)
def floyd_warshall(path):
    for k in range(path.shape[0]):
        # path[:] = np.minimum(path[:], np.add.outer(path[:, k], path[k, :]))
        for i in range(path.shape[0]):
            path[i, :] = np.minimum(path[i, :], path[i, k] + path[k, :])
```

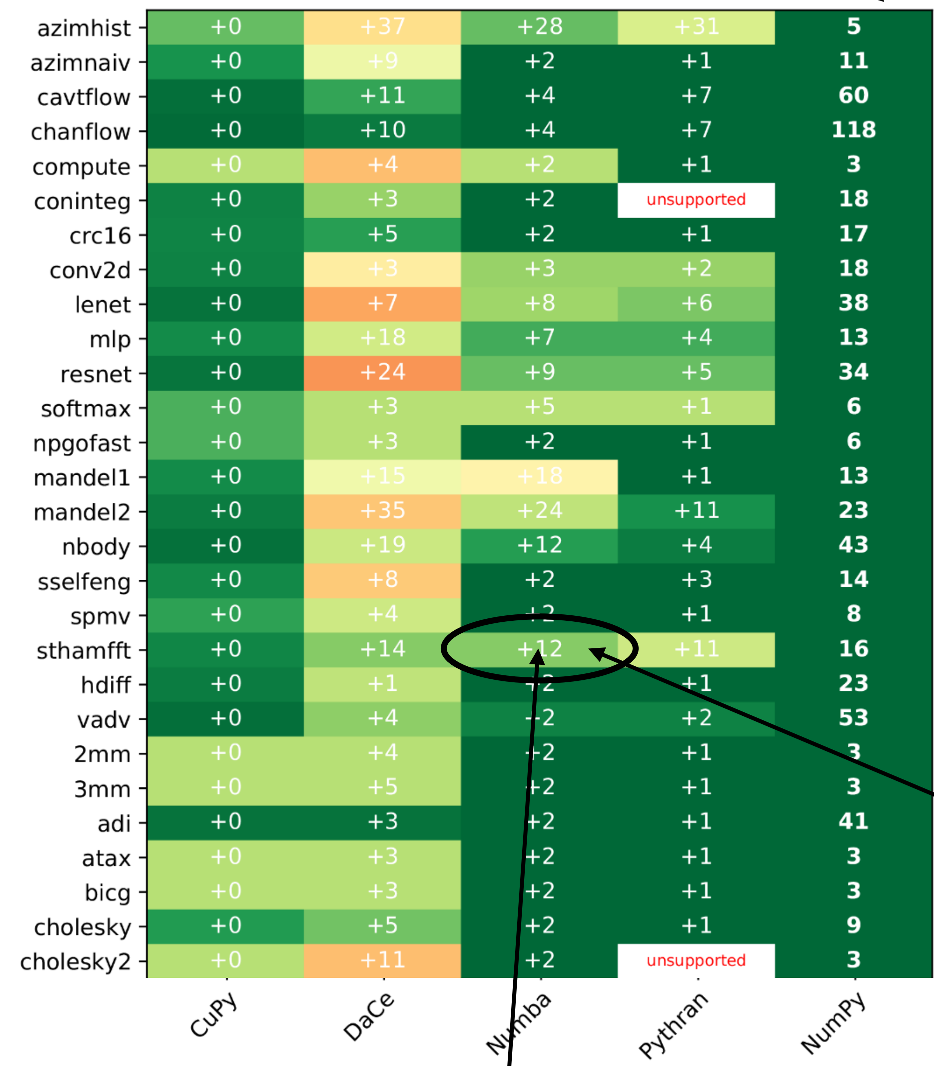


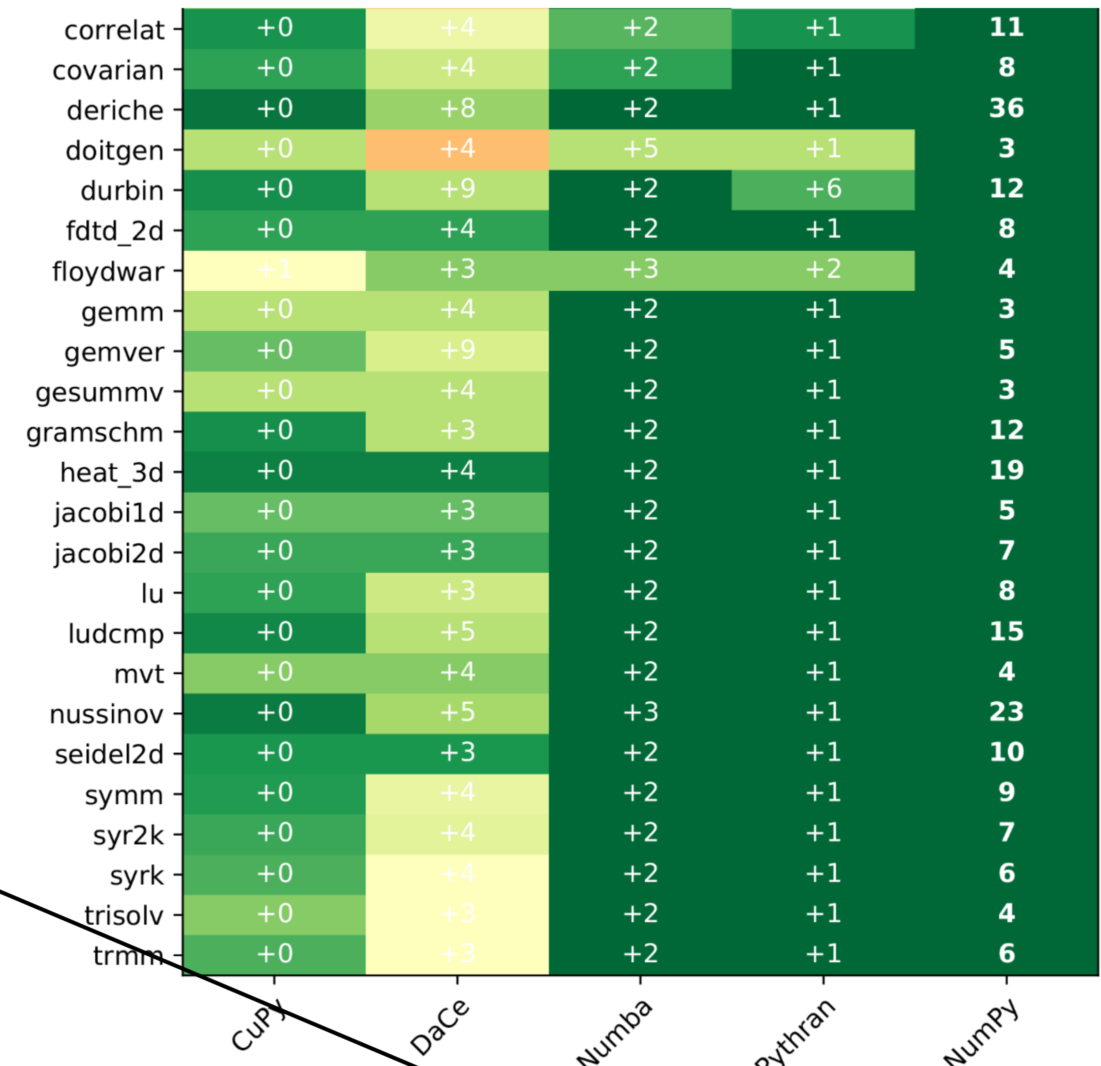
1 line of the reference code changed, i.e., $\mathrm{diff}(impl_{np}, impl_{ct}) = 1$

Numba: $\frac{\mathrm{diff}(impl_{np}, impl_{ct})}{|impl_{np}|} = 25\%$

Productivity

Reference code lines





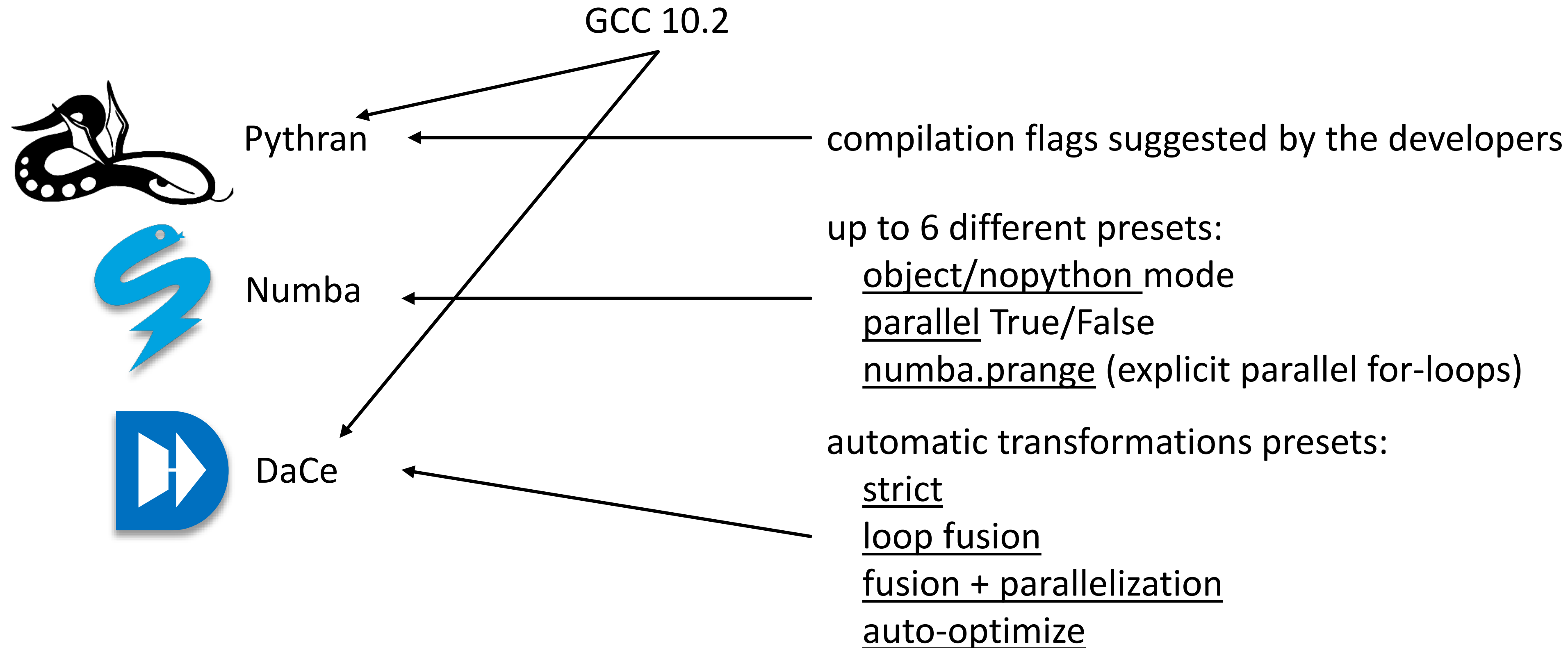
Number: Abs. diff. in number of lines

Color (green to red): Percentage of reference code changed

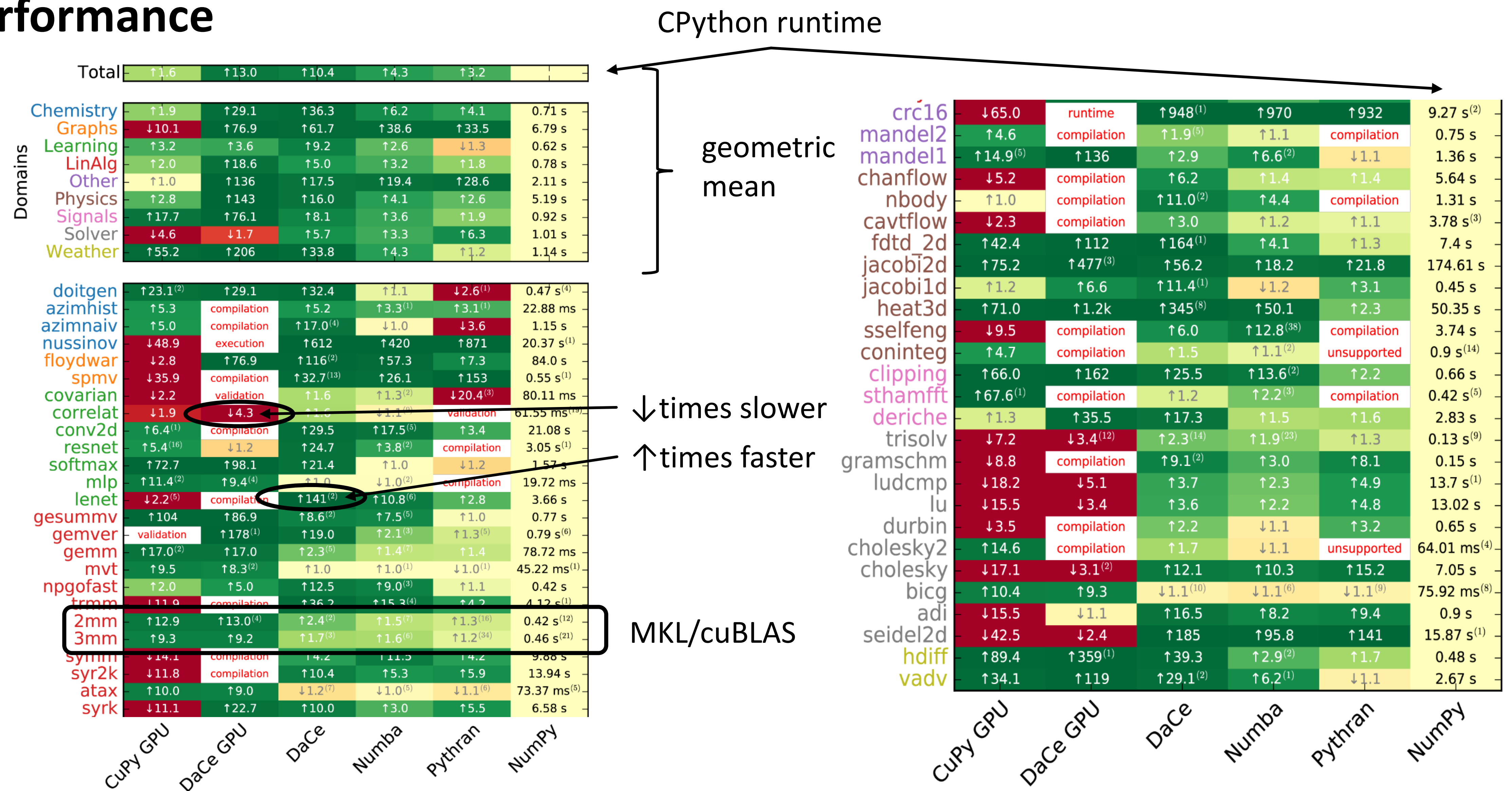
Measuring Performance

Machine:
CPU: 2x16-core Intel Xeon Gold 6130
GPU: Nvidia V100 32GB

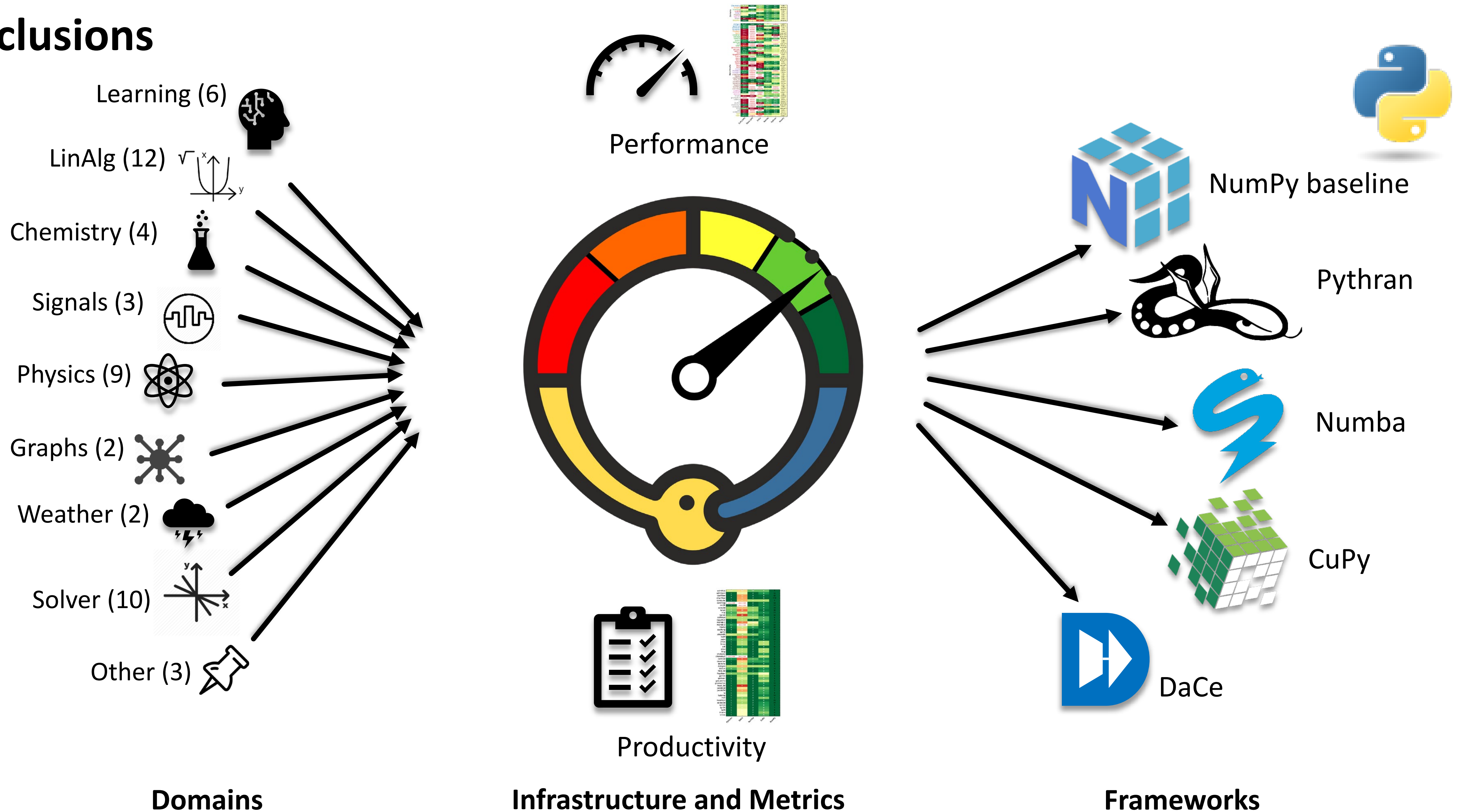
Optimization Presets (automatic):



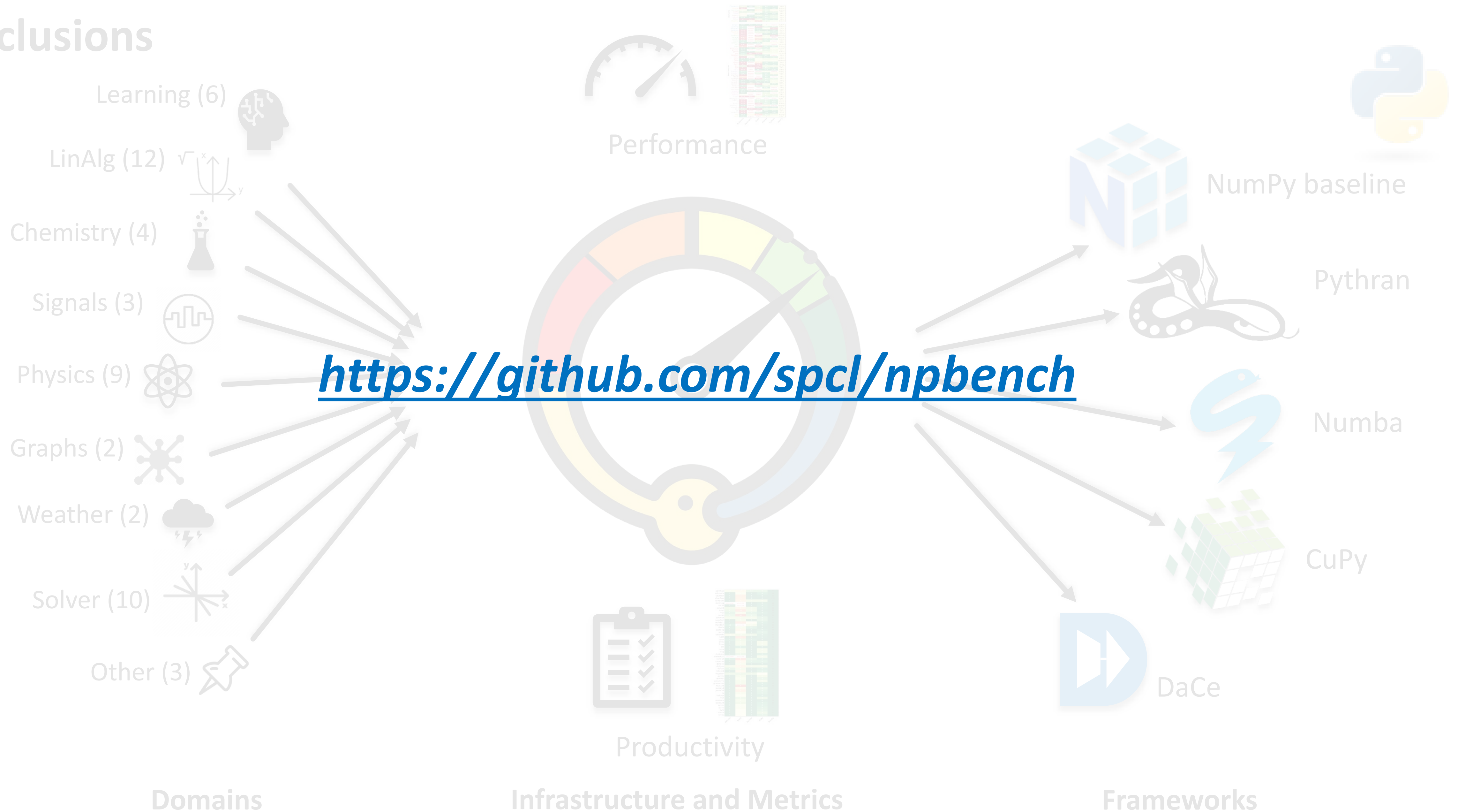
Performance



Conclusions



Conclusions



https://github.com/spcl/npbench