

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

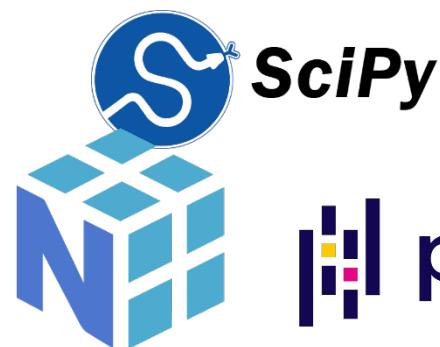
NPBench: A Benchmarking Suite for High-Performance NumPy



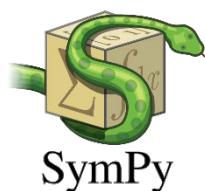
Python: The Scientific Language



IP[y]:
IPython



matplotlib



 NumPy

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

buffer
dtype
shape
strides = (32, 8)

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

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

0	1	2	3
4	5	6	7
8	9	10	11

9	8	7	6
5	4	3	2
1	0	-1	-2

9	9	9	9
9	9	9	9
9	9	9	9

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

0	1	2	3
4	5	6	7
8	9	10	11

3	2	1	0
3	2	1	0
3	2	1	0

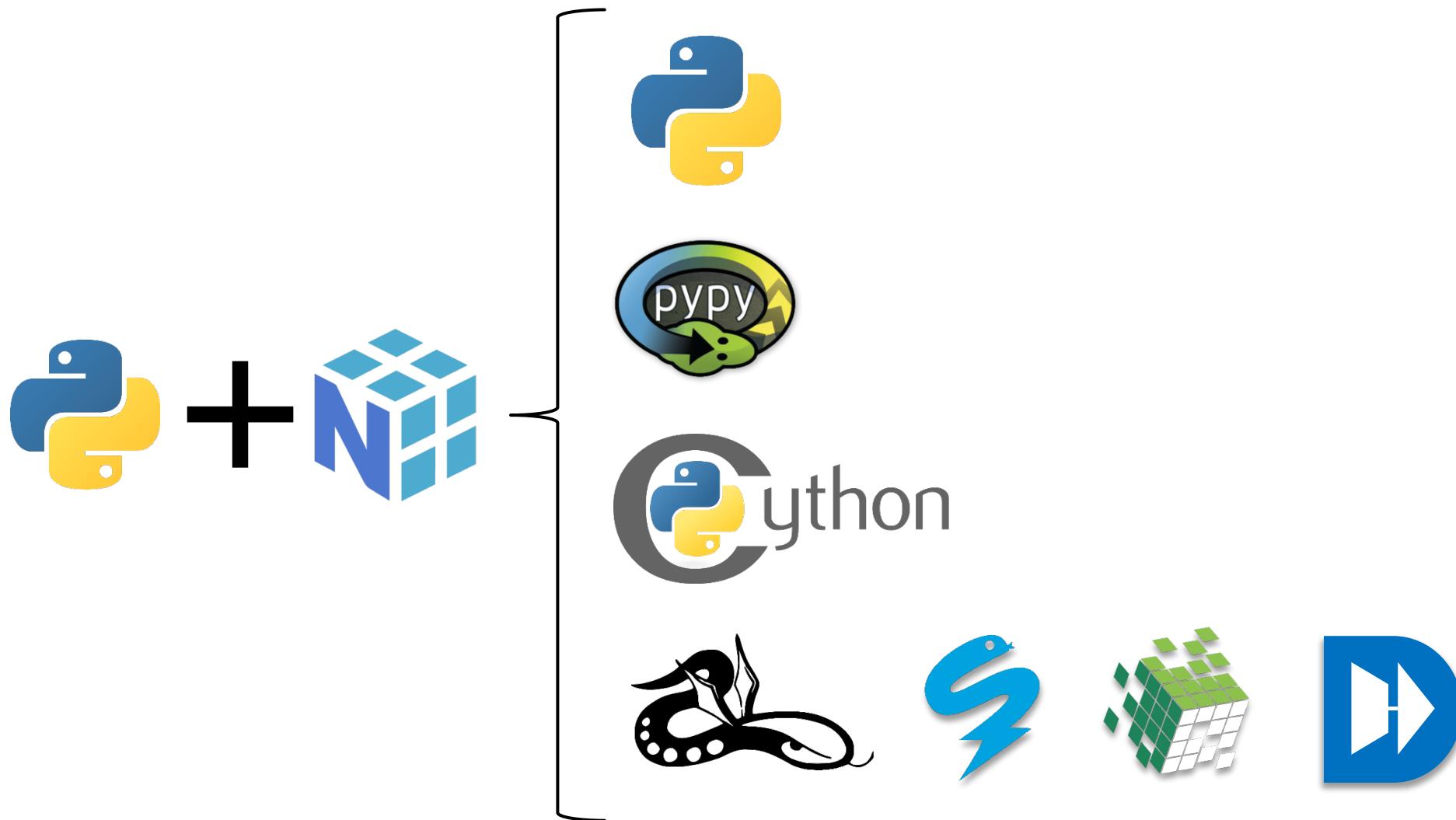
3	3	3	3
7	7	7	7
11	11	11	11

```
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])
```

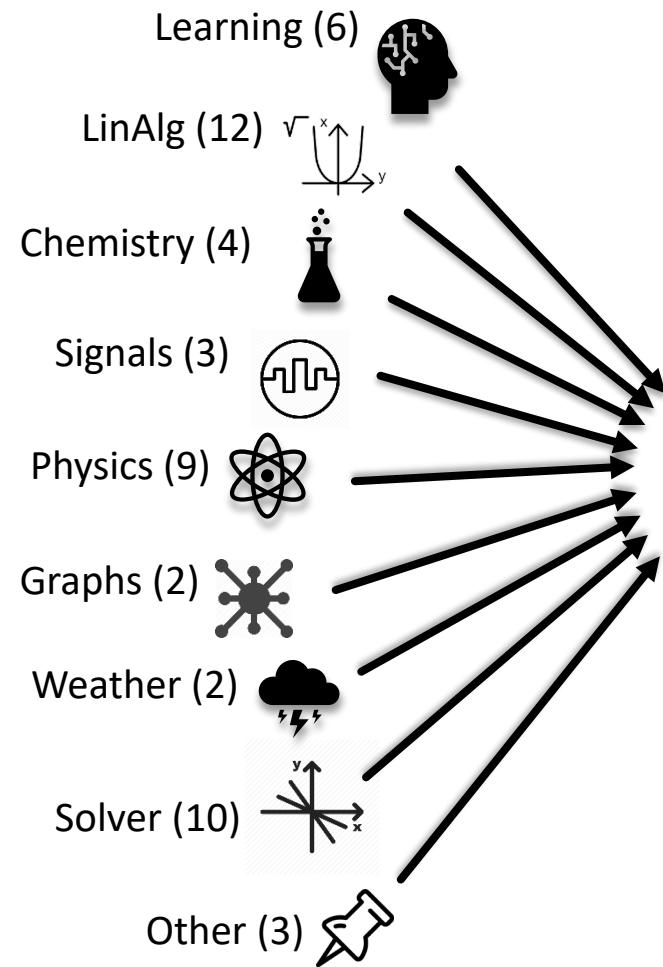
Many choices for executing Python and NumPy codes ...



Many choices for executing Python and NumPy codes ...



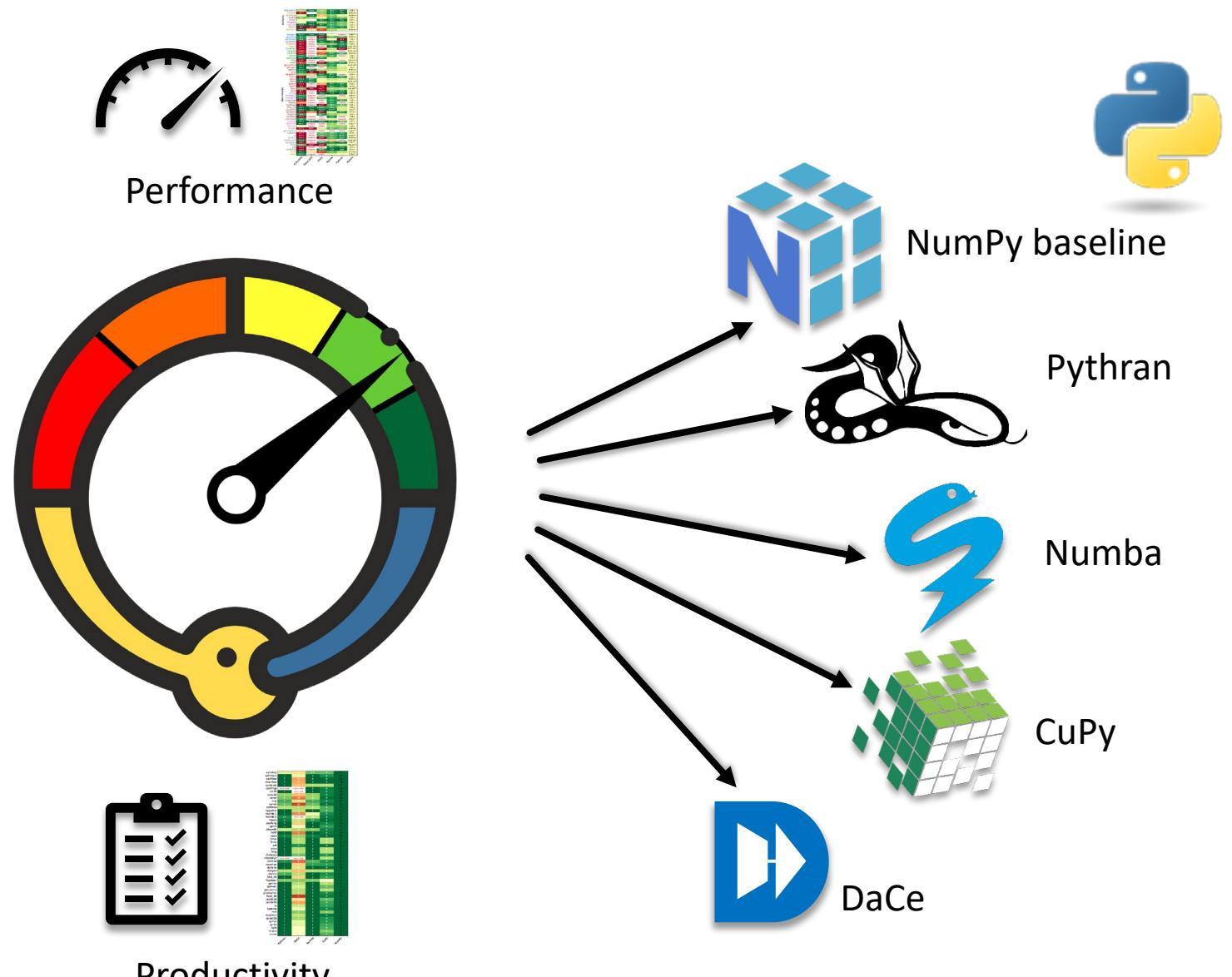
Let's make a benchmark!



Domains

Infrastructure and Metrics

Frameworks



Scientific Domains

Chemistry



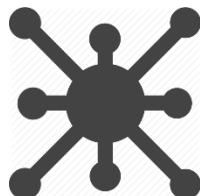
azimnaiv (PyFAI)
azimhist
doitgen (Polybench/C)
nussinov

Signals



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

Graphs



spmv
floydwar (Polybench/C)

Weather



vadv
hdifff (GT4Py)

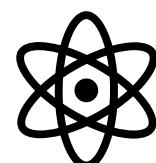
Learning



conv2d
softmax
mlp
lenet
resnet
correlate
covarian

(Polybench/C)

Physics



cavtflow
chanflow
nbody
coninteg
sselfeng
jacobi1d
jacobi2d
heat3d
fdtd2d

(Polybench/C)

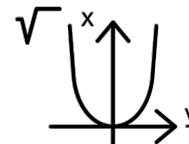
Other



mandel1
mandel2
crc16

(From Python to Numpy)
(oysstu)

LinAlg

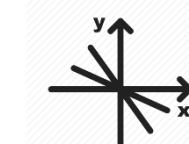


gemm
gemver
gesummv
symm
syr2k
syrk
trmm

(Polybench/C)

2mm
3mm
atax
mvt
npgofast (Numba)

Solvers



seidel2d
durbin
trisolv
adi
bicg (Polybench/C)

gramschm
lu
ludcmp
cholesky
cholesky2

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)
```

Linear algebra sub-module

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

```
def cholesky(A):
    for i in range(A.shape[0]):
        for j in range(i):
            for k in range(j+1, i):
                A[i, j] -= A[i, k] * A[i, 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])
```

Python and NumPy Accelerators



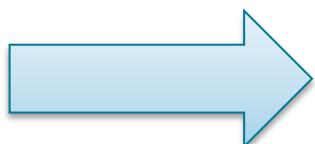
[Clinostomus funduloides Rosy-side Dace](#) by [Brian Gratwicke](#)
licensed under [CC BY 2.0](#)



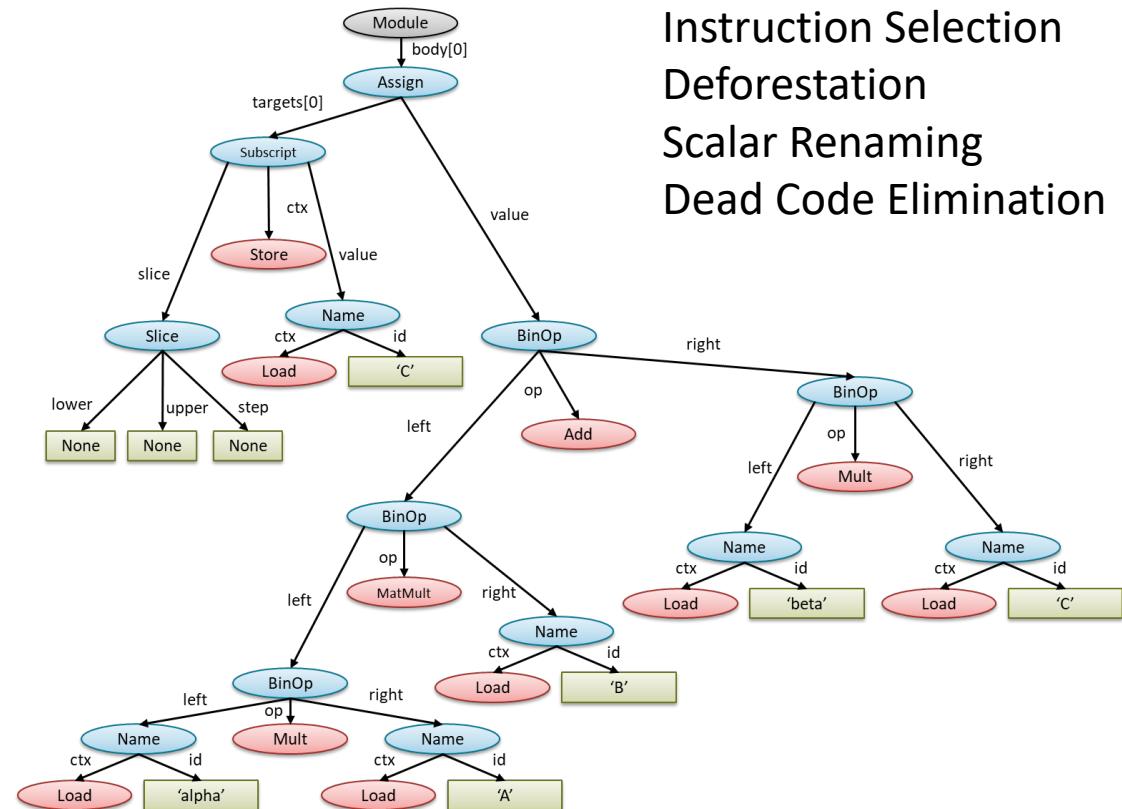
Pythran

Type annotations

```
# 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



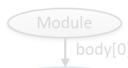
Pythran

Type annotations

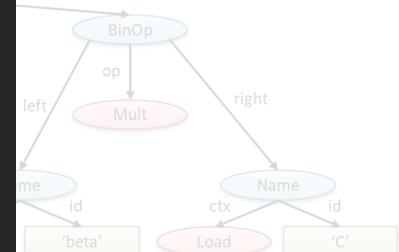
```
# pythran export kernel(
    float64, float64, float64
    float64[:, :], float64[:, :],
def gemm(alpha, beta, A, B,
    C[:] = alpha * A @ B + b
```

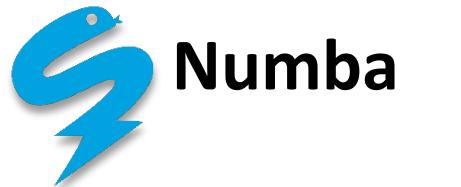
```
namespace __pythran_gemm
{
    struct kernel
    {
        typedef void callable;
        template <typename argument_type0 , typename argument_type1 , typename argument_type2 , typename argument_type3 , typename argument_type4 >
        typename kernel::type<argument_type0, argument_type1, argument_type2, argument_type3, argument_type4>::result_type kernel::operator()(argument_type0&& 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_::function::matmul()(pythonic::operator_::mul(alpha, A), B), pythonic::operator_::mul(beta, C));
            return pythonic::builtins::None;
        }
    }
}
```

Python AST-based IR



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





JIT decorator

```
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-in>
['$10binary_matrix_multiply.4', '$6binary_multiply.2', 'B']
$16binary_multiply.7 = beta * C ['$16binary_multiply.7']
$18binary_add.8 = $10binary_matrix_multiply.4 + $16binary_multiply.7
['$10binary_matrix_multiply.4', '$16binary_multiply.7', '$18binary_multiply.7']
$const22.10 = const(NoneType, None) ['$const22.10']
$const24.11 = const(NoneType, None) ['$const24.11']
$26build_slice.12 = global(slice: <class 'slice'>) ['$26build_slice.12']
$26build_slice.13 = call $26build_slice.12($const22.10, $const24.11)
args=(Var($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 Elimination



Numba

JIT decorator

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

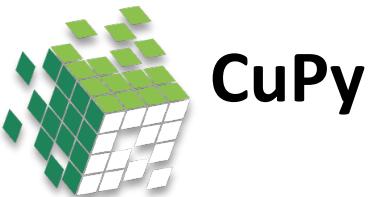
Numba IR

```
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
```

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

2'
 %i
y.7
ar
bi
 .s
co
 .11, gemm_numba.py:14)), kws=(),
 ist22.10', '\$const24.11']
 i.8', '\$26build_slice.13', 'C']

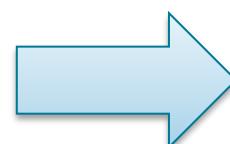
 n_value.15', '\$const30.14']
]



CuPy

```
import numpy as np
import cupy as np

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

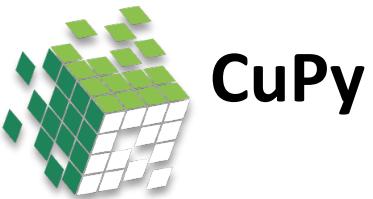


```
cdef class ndarray:

    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

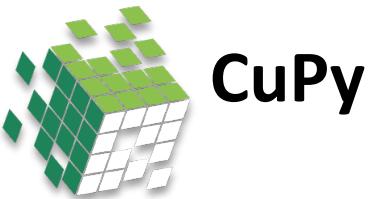
```
cdef function.Function _get_simple_elementwise_kernel(
    tuple params, tuple arginfos, str operation, str name,
    _TypeMap type_map, str preamble, str loop_prep='', str after_loop='',
    tuple options=()):
    module_code = string.Template('''
${typedef_preamble}
${preamble}
extern "C" __global__ 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):
            if 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)
        else:
            return NotImplemented

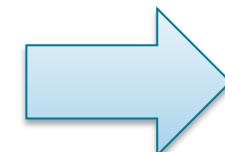
    def __matmul__(x, y):
        if not isinstance(y, ndarray) and _should_use_rop(x, y):
            return NotImplemented
        else:
            return cupy.linalg._product.matmul(x, y)
```



DaCe

Symbolic sizes

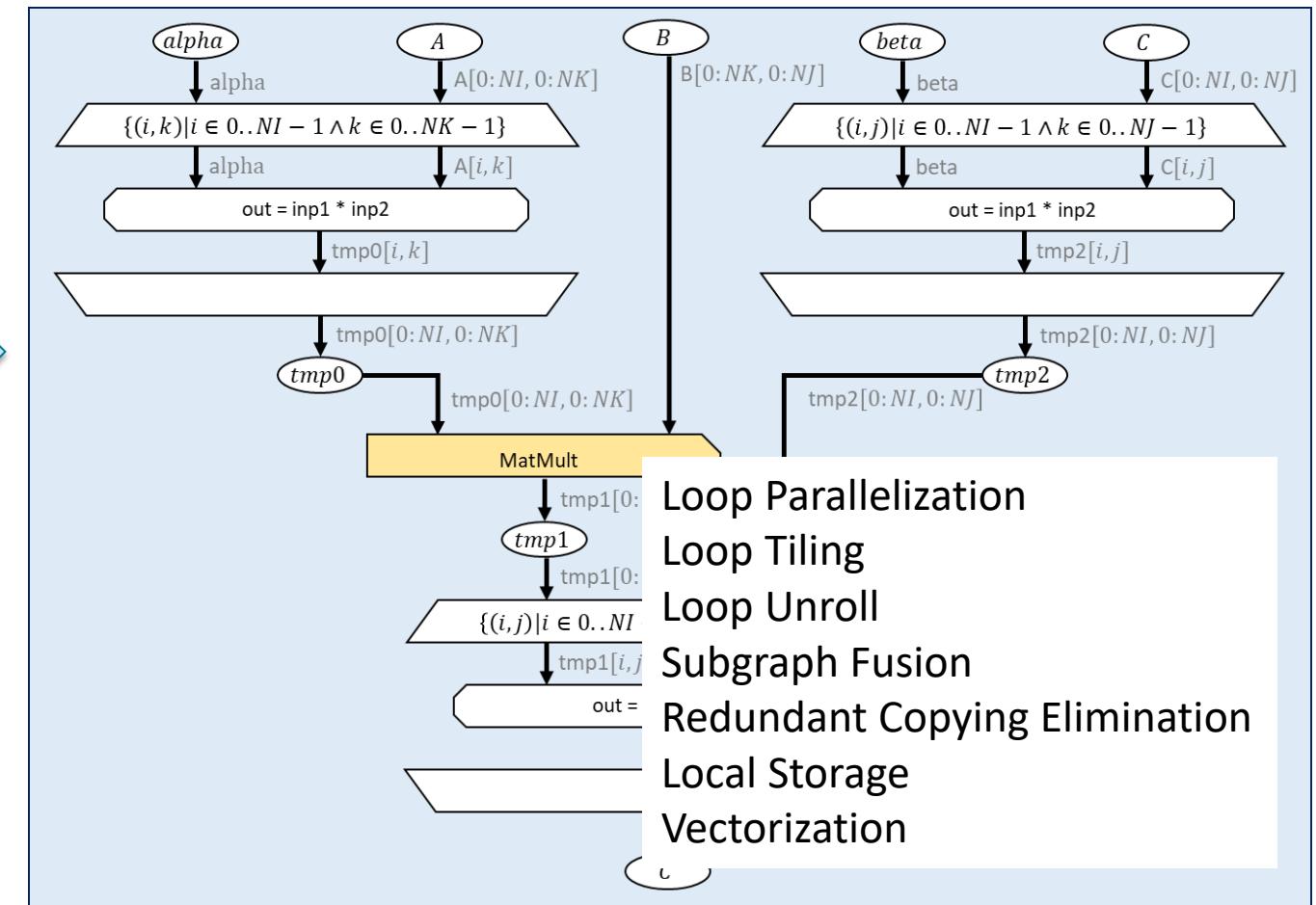
```
import dace as dc
NI, NJ, NK = (dc.symbol(s, dtype=dc.int64)
               for s in ('NI', 'NJ', 'NK'))
@dc.program
def gemm(alpha: dc.float64,
          beta: dc.float64,
          C: dc.float64[NI, NJ],
          A: dc.float64[NI, NK],
          B: dc.float64[NK, NJ]):
    C[:] = alpha * A @ B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

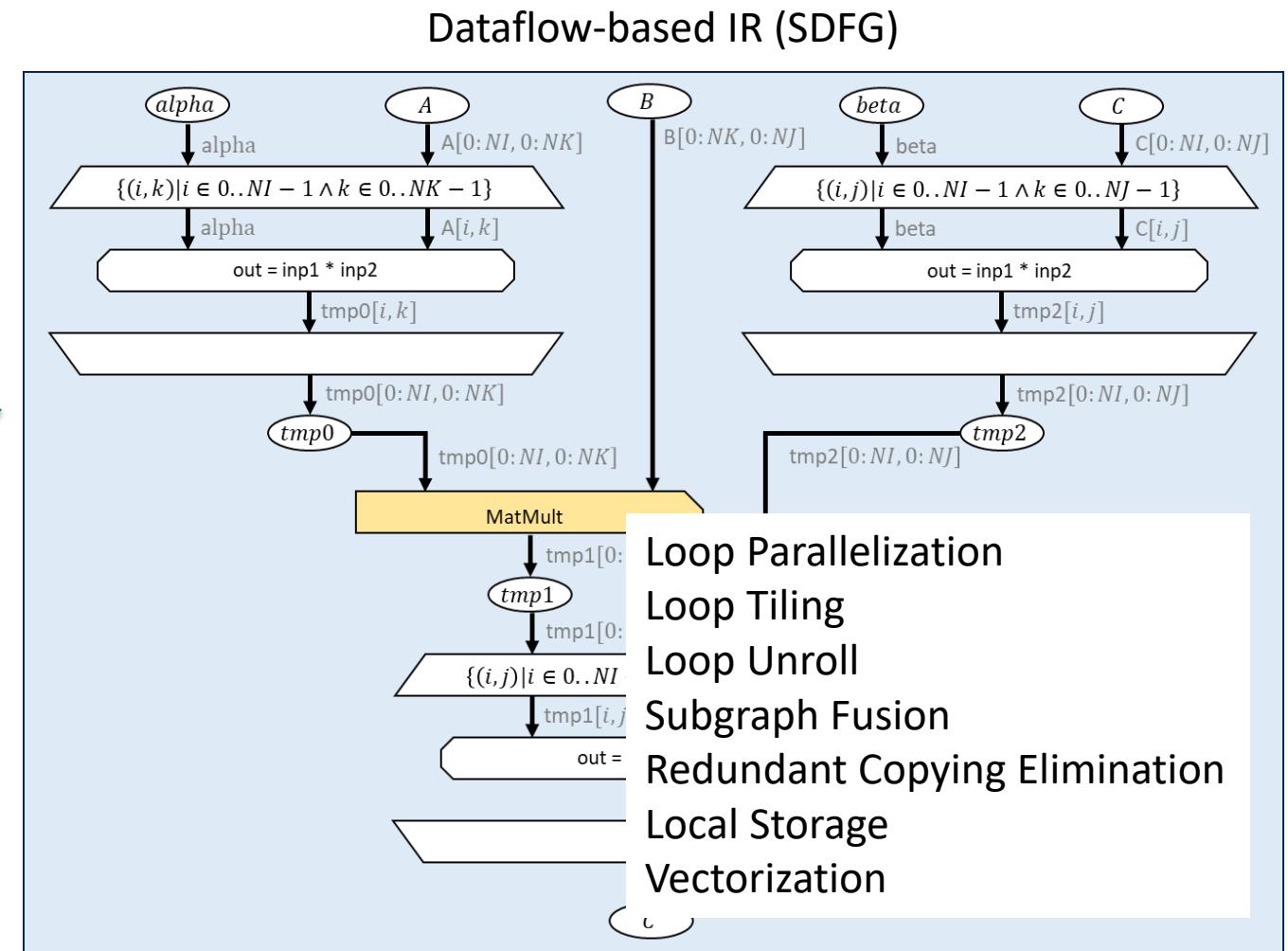
                /////////////////
                // Tasklet code (_Mult_)
                __out = (__in1 * __in2);
                ///////////////

                __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);
    ///////////////
}

```



```
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_marshall(path):
    for k in range(path.shape[0]):
        path[:] = np.minimum(path[:, :], np.add.outer(path[:, k], path[k, :]))
```

Unsupported by Numba

```
import numpy as np
import numba as nb
@nb.jit(nopython=True, parallel=False, fastmath=True)
def floyd_marshall(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, :])
```

Rewritten in loop-form

```
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
```

```
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
```

Unsupported by Numba

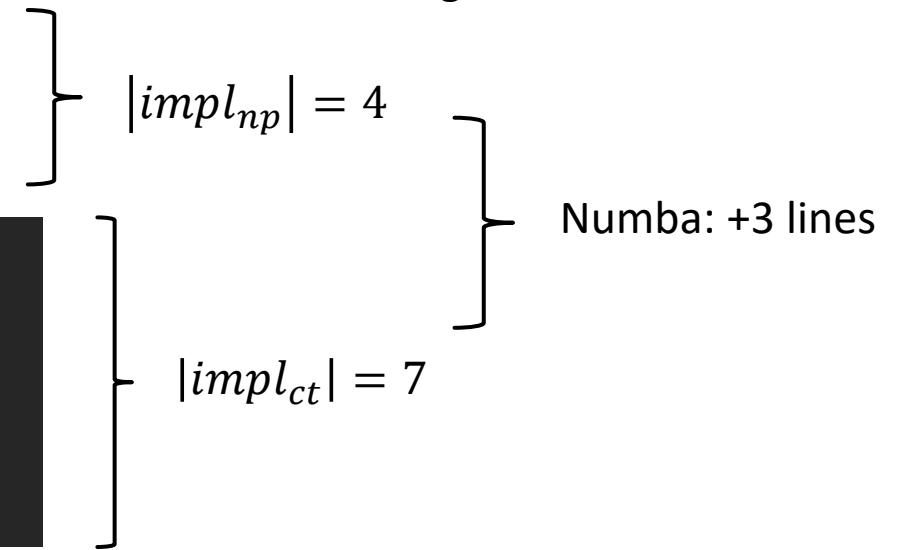
Rewritten as a separate method

Measuring Productivity

```
import numpy as np
def floyd_marshall(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_marshall(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, :])
```

Counting lines with SLOCCount

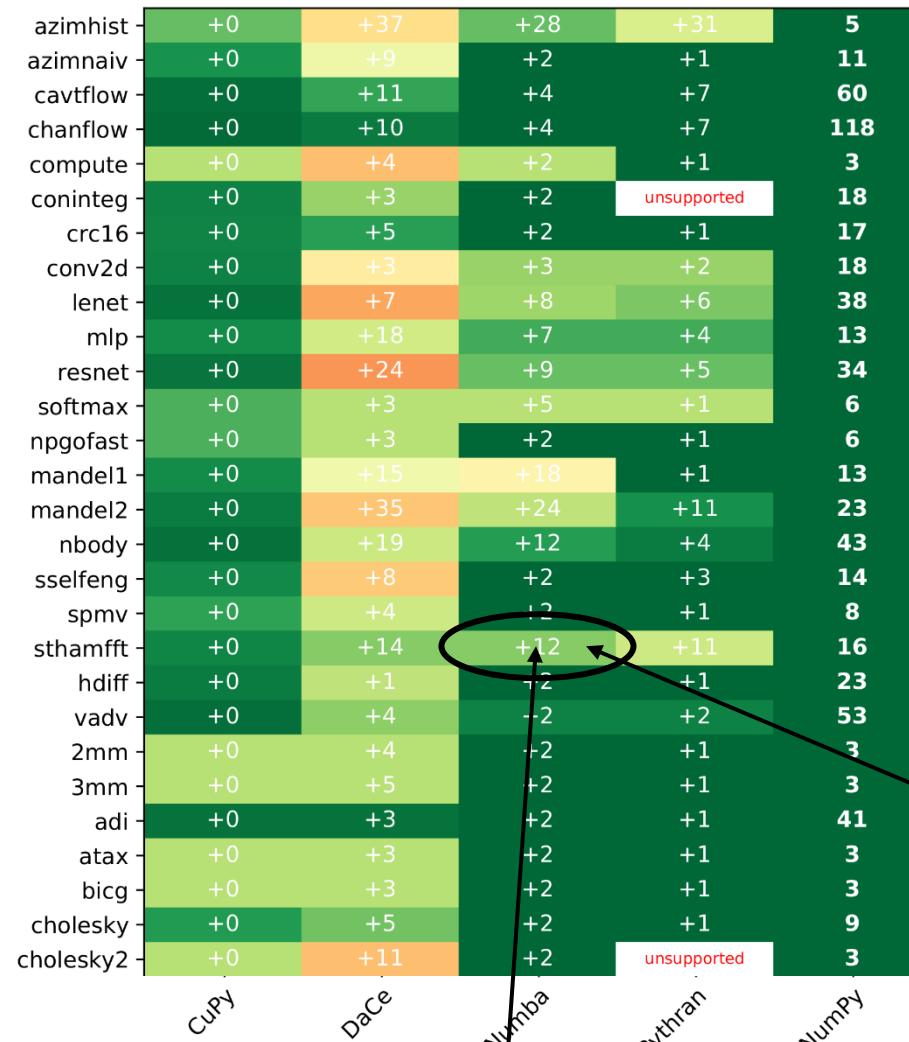


1 line of the reference code changed, i.e., $\text{diff}(impl_{np}, impl_{ct}) = 1$

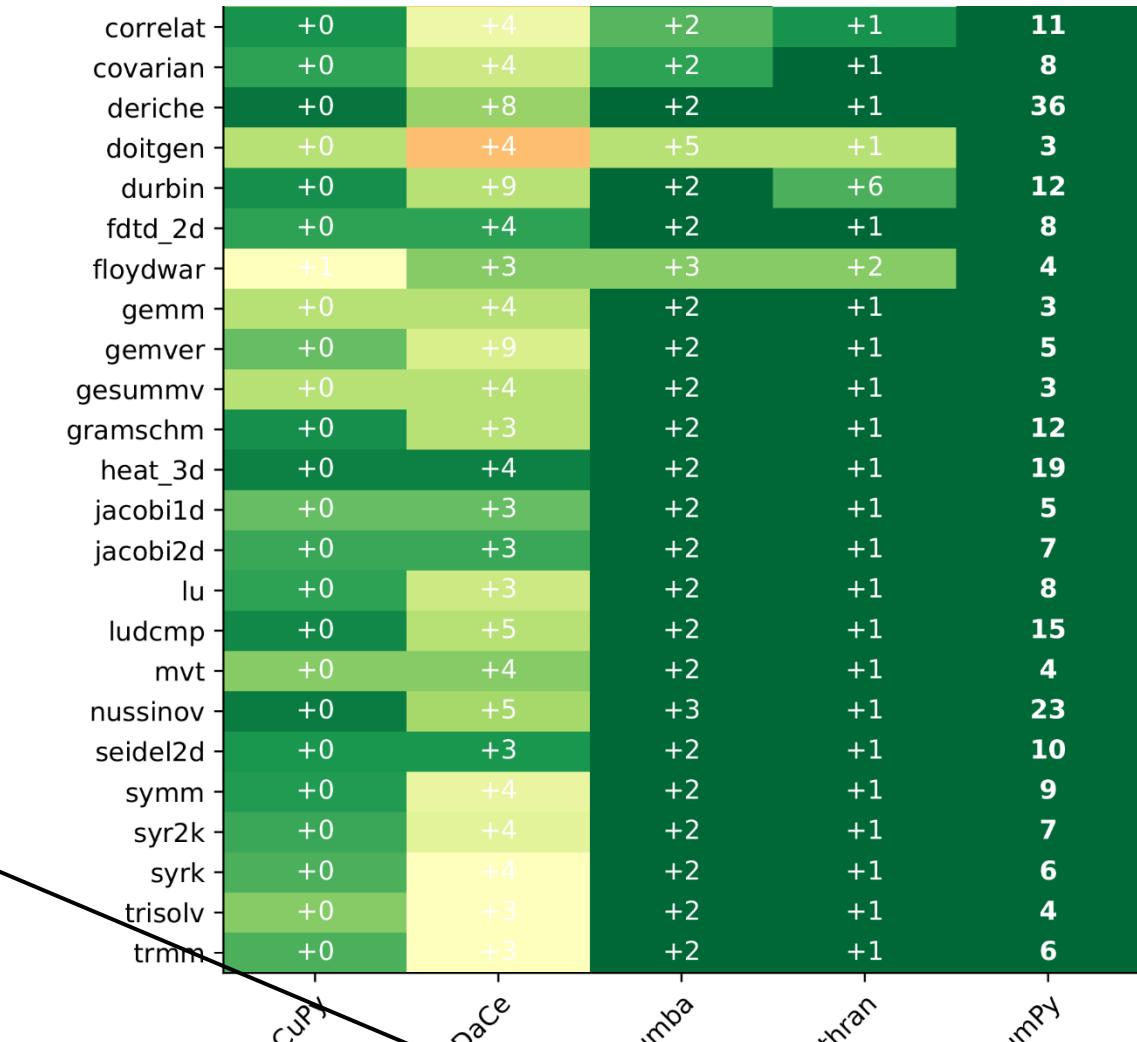
$$\text{Numba: } \frac{\text{diff}(impl_{np}, impl_{ct})}{|impl_{np}|} = 25\%$$

Productivity

Reference code lines



Number: Abs. diff. in number of lines

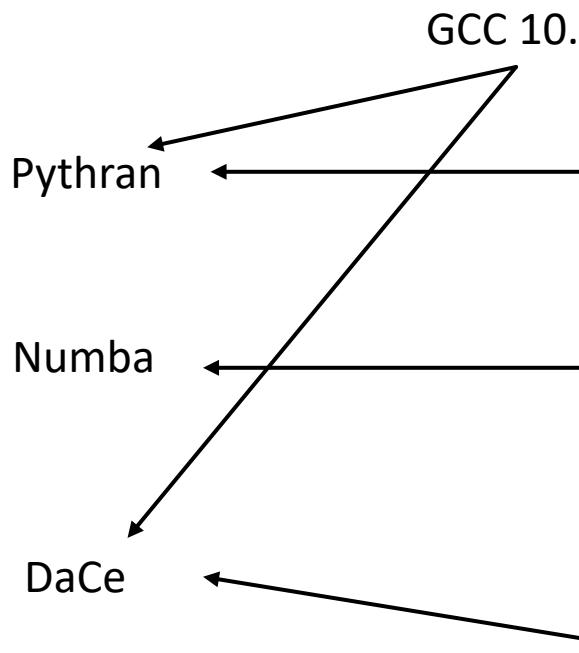


Color (green to red): Percentage of reference code changed

Measuring Performance

Machine:

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



Optimization Presets (automatic):

compilation flags suggested by the developers

up to 6 different presets:

object/nopython mode

parallel True/False

numba.prange (explicit parallel for-loops)

automatic transformations presets:

strict

loop fusion

fusion + parallelization

auto-optimize

Performance

	CuPy GPU	DaCe GPU	DaCe	Numba	Pythran	NumPy	Total
Chemistry	↑1.9	↑29.1	↑36.3	↑6.2	↑4.1	0.71 s	
Graphs	↓10.1	↑76.9	↑61.7	↑38.6	↑33.5	6.79 s	
Learning	↑3.2	↑3.6	↑9.2	↑2.6	↓1.3	0.62 s	
LinAlg	↑2.0	↑18.6	↑5.0	↑3.2	↑1.8	0.78 s	
Other	↑1.0	↑136	↑17.5	↑19.4	↑28.6	2.11 s	
Physics	↑2.8	↑143	↑16.0	↑4.1	↑2.6	5.19 s	
Signals	↑17.7	↑76.1	↑8.1	↑3.6	↑1.9	0.92 s	
Solver	↓4.6	↓1.7	↑5.7	↑3.3	↑6.3	1.01 s	
Weather	↑55.2	↑206	↑33.8	↑4.3	↑1.2	1.14 s	
doitgen	↑23.1 ⁽²⁾	↑29.1	↑32.4	↑1.1	↓2.6 ⁽¹⁾	0.47 s ⁽⁴⁾	
azimhist	↑5.3	compilation	↑5.2	↑3.3 ⁽¹⁾	↑3.1 ⁽¹⁾	22.88 ms	
azimnaiv	↑5.0	compilation	↑17.0 ⁽⁴⁾	↓1.0	↓3.6	1.15 s	
nussinov	↓48.9	execution	↑612	↑420	↑871	20.37 s ⁽¹⁾	
floydwar	↓2.8	↑76.9	↑116 ⁽²⁾	↑57.3	↑7.3	84.0 s	
spmv	↓35.9	compilation	↑32.7 ⁽¹³⁾	↑26.1	↑153	0.55 s ⁽¹⁾	
covarian	↓2.2	validation	↑1.6	↑1.3 ⁽²⁾	↓20.4 ⁽³⁾	80.11 ms	
correlat	↓1.9	↓4.3	↑1.0	↓1.1 ⁽ⁿ⁾	validation	61.55 ms ⁽³⁾	
conv2d	↑6.4 ⁽¹⁾	compilation	↑29.5	↑17.5 ⁽⁵⁾	↑3.4	21.08 s	
resnet	↑5.4 ⁽¹⁶⁾	↓1.2	↑24.7	↑3.8 ⁽²⁾	compilation	3.05 s ⁽¹⁾	
softmax	↑72.7	↑98.1	↑21.4	↑1.0	↓1.2	1.57 s	
mlp	↑11.4 ⁽²⁾	↑9.4 ⁽⁴⁾	↑1.0	↓1.0 ⁽²⁾	compilation	19.72 ms	
lenet	↓2.2 ⁽⁵⁾	compilation	↑141 ⁽²⁾	↑10.8 ⁽⁶⁾	↑2.8	3.66 s	
gesummv	↑104	↑86.9	↑8.6 ⁽²⁾	↑7.5 ⁽⁵⁾	↑1.0	0.77 s	
gemver	validation	↑178 ⁽¹⁾	↑19.0	↑2.1 ⁽³⁾	↑1.3 ⁽⁵⁾	0.79 s ⁽⁶⁾	
gemm	↑17.0 ⁽²⁾	↑17.0	↑2.3 ⁽⁵⁾	↑1.4 ⁽⁷⁾	↑1.4	78.72 ms	
mvt	↑9.5	↑8.3 ⁽²⁾	↑1.0	↑1.0 ⁽¹⁾	↓1.0 ⁽¹⁾	45.22 ms ⁽¹⁾	
npgofast	↑2.0	↑5.0	↑12.5	↑9.0 ⁽³⁾	↑1.1	0.42 s	
trmm	↓11.0	compilation	↑36.2	↑15.3 ⁽⁴⁾	↑1.2	4.12 s ⁽¹⁾	
2mm	↑12.9	↑13.0 ⁽⁴⁾	↑2.4 ⁽²⁾	↑1.5 ⁽⁷⁾	↑1.3 ⁽¹⁶⁾	0.42 s ⁽¹²⁾	
3mm	↑9.3	↑9.2	↑1.7 ⁽³⁾	↑1.6 ⁽⁶⁾	↑1.2 ⁽³⁴⁾	0.46 s ⁽²¹⁾	
Symini	↑14.1	compilation	↑4.2	↑1.1 ⁽²⁾	↑4.2	9.86 s	
syr2k	↓11.8	compilation	↑10.4	↑5.3	↑5.9	13.94 s	
atax	↑10.0	↑9.0	↓1.2 ⁽⁷⁾	↓1.0 ⁽⁵⁾	↓1.1 ⁽⁶⁾	73.37 ms ⁽⁵⁾	
syrk	↓11.1	↑22.7	↑10.0	↑3.0	↑5.5	6.58 s	

CPython runtime

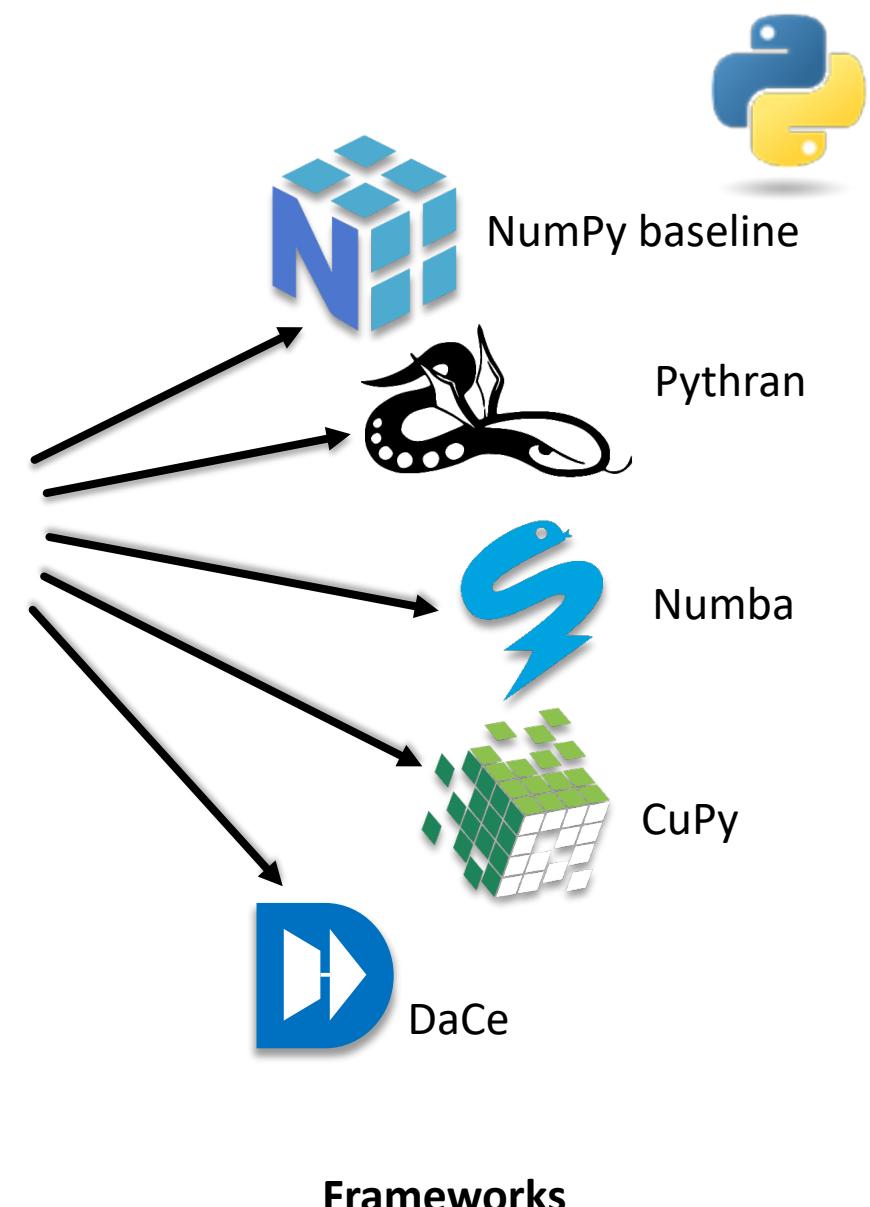
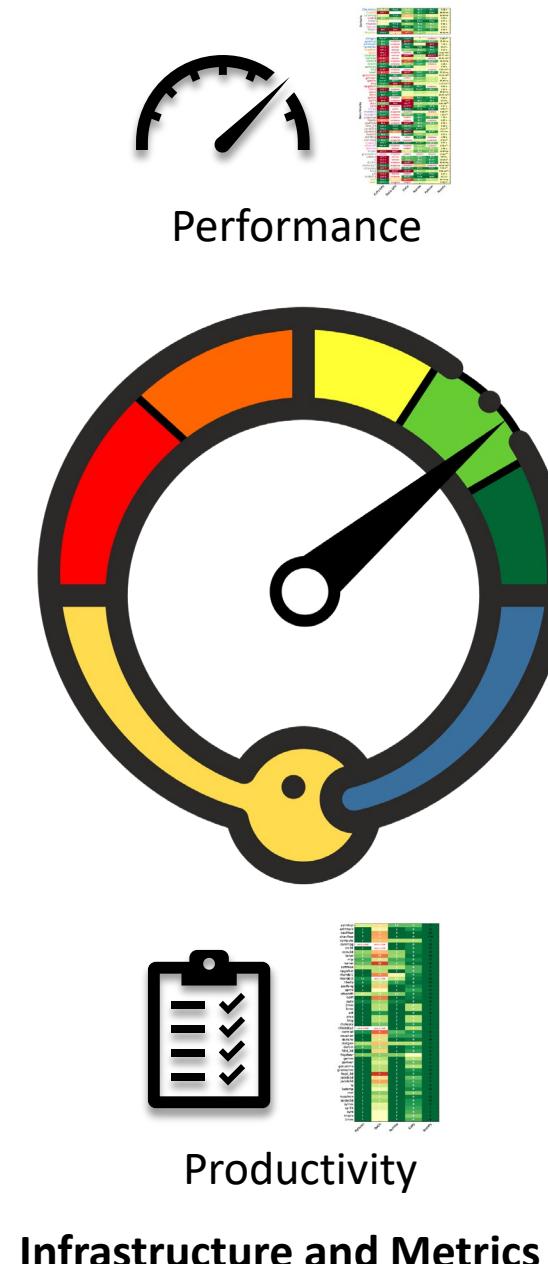
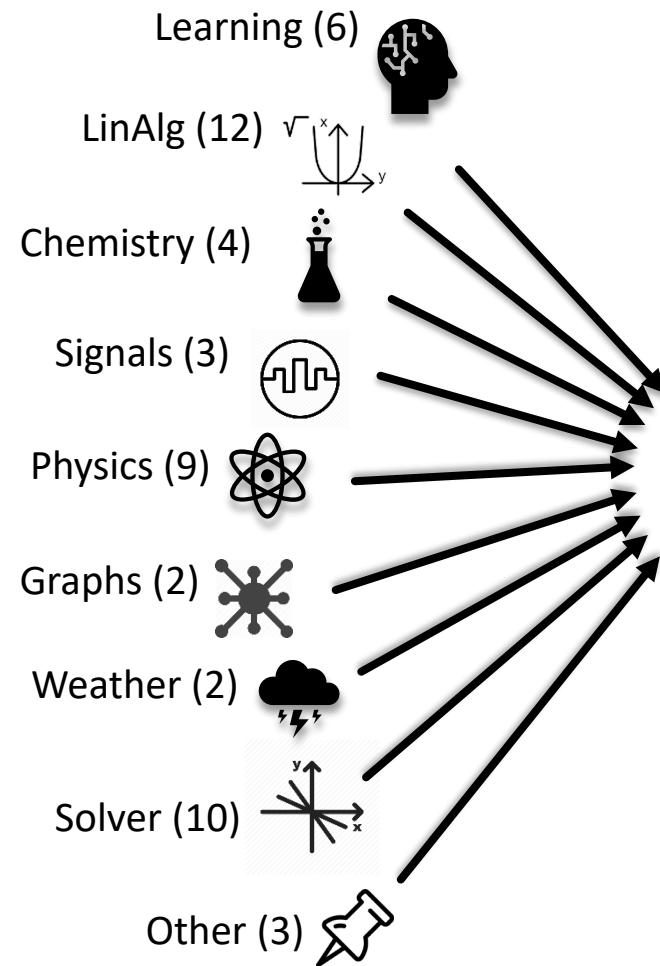
geometric mean

↓times slower
↑times faster

MKL/cuBLAS

	CuPy GPU	DaCe GPU	DaCe	Numba	Pythran	NumPy	
crc16	↓65.0	runtime	↑948 ⁽¹⁾	↑970	↑932	9.27 s ⁽²⁾	
mandel2	↑4.6	compilation	↑1.9 ⁽⁵⁾	↑1.1	compilation	0.75 s	
mandel1	↑14.9 ⁽⁵⁾	↑136	↑2.9	↑6.6 ⁽²⁾	↓1.1	1.36 s	
chanflow	↓5.2	compilation	↑6.2	↑1.4	↑1.4	5.64 s	
nbody	↑1.0	compilation	↑11.0 ⁽²⁾	↑4.4	compilation	1.31 s	
cavtflow	↓2.3	compilation	↑3.0	↑1.2	↑1.1	3.78 s ⁽³⁾	
fdtd 2d	↑42.4	↑112	↑164 ⁽¹⁾	↑4.1	↑1.3	7.4 s	
jacobi2d	↑75.2	↑477 ⁽³⁾	↑56.2	↑18.2	↑21.8	174.61 s	
jacobi1d	↑1.2	↑6.6	↑11.4 ⁽¹⁾	↓1.2	↑3.1	0.45 s	
heat3d	↑71.0	↑1.2k	↑345 ⁽⁸⁾	↑50.1	↑2.3	50.35 s	
sselfeng	↓9.5	compilation	↑6.0	↑12.8 ⁽³⁸⁾	compilation	3.74 s	
coninteg	↑4.7	compilation	↑1.5	↑1.1 ⁽²⁾	unsupported	0.9 s ⁽¹⁴⁾	
clipping	↑66.0	↑162	↑25.5	↑13.6 ⁽²⁾	↑2.2	0.66 s	
sthamfft	↑67.6 ⁽¹⁾	compilation	↑1.2	↑2.2 ⁽³⁾	compilation	0.42 s ⁽⁵⁾	
deriche	↑1.3	↑35.5	↑17.3	↑1.5	↑1.6	2.83 s	
trisolv	↓7.2	↓3.4 ⁽¹²⁾	↑2.3 ⁽¹⁴⁾	↑1.9 ⁽²³⁾	↑1.3	0.13 s ⁽⁹⁾	
gramschm	↓8.8	compilation	↑9.1 ⁽²⁾	↑3.0	↑8.1	0.15 s	
ludcmp	↑18.2	↓5.1	↑3.7	↑2.3	↑4.9	13.7 s ⁽¹⁾	
lu	↓15.5	↓3.4	↑3.6	↑2.2	↑4.8	13.02 s	
durbin	↓3.5	compilation	↑2.2	↓1.1	↑3.2	0.65 s	
cholesky2	↑14.6	compilation	↑1.7	↓1.1	unsupported	64.01 ms ⁽⁴⁾	
cholesky	↓17.1	↓3.1 ⁽²⁾	↑12.1	↑10.3	↑15.2	7.05 s	
bicg	↑10.4	↑9.3	↓1.1 ⁽¹⁰⁾	↓1.1 ⁽⁶⁾	↓1.1 ⁽⁹⁾	75.92 ms ⁽⁸⁾	
adi	↓15.5	↓1.1	↑16.5	↑8.2	↑9.4	0.9 s	
seidel2d	↓42.5	↓2.4	↑185	↑95.8	↑141	15.87 s ⁽¹⁾	
hdifff	↑89.4	↑359 ⁽¹⁾	↑39.3	↑2.9 ⁽²⁾	↑1.7	0.48 s	
vadv	↑34.1	↑119	↑29.1 ⁽²⁾	↑6.2 ⁽¹⁾	↓1.1	2.67 s	

Conclusions



Domains

Infrastructure and Metrics

Frameworks

Conclusions



<https://github.com/spcl/npbench>

