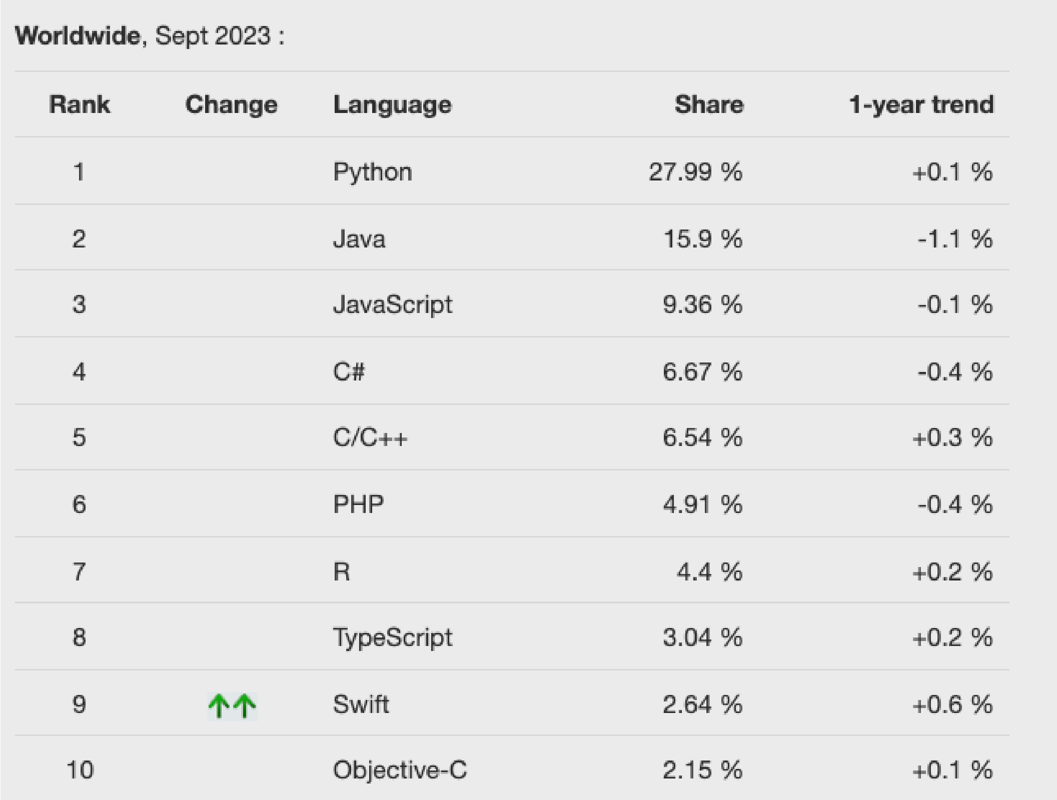
High-level Compiler Optimizations for Python Programs

Tong Zhou

Committee: Vivek Sarkar, Jun Shirako, Tushar Krishna, Santosh Pande, Rich Vuduc

Georgia Tech

Python is the most popular programming language today (according to the PyPL index)



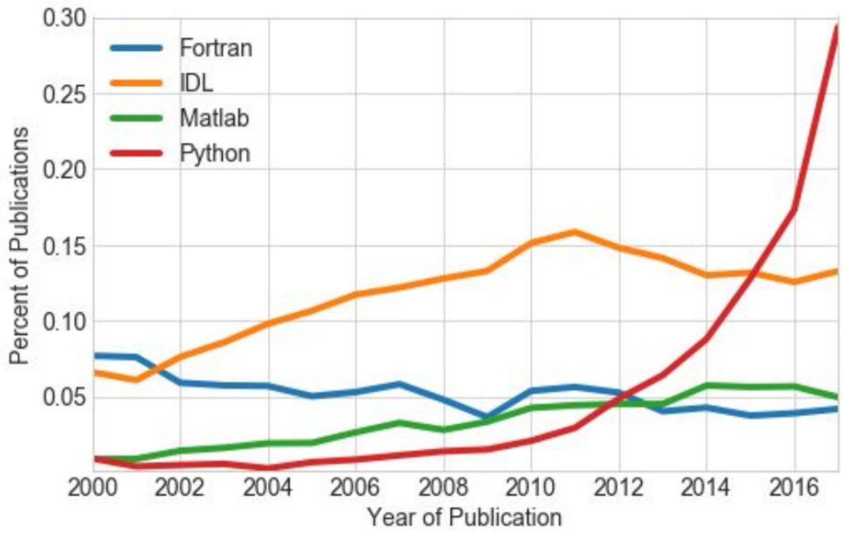
Worldwide, Sept 2023 :

Rank	Change	Language	Share	1-year trend
1		Python	27.99 %	+0.1 %
2		Java	15.9 %	-1.1 %
3		JavaScript	9.36 %	-0.1 %
4		C#	6.67 %	-0.4 %
5		C/C++	6.54 %	+0.3 %
6		PHP	4.91 %	-0.4 %
7		R	4.4 %	+0.2 %
8		TypeScript	3.04 %	+0.2 %
9	↑↑	Swift	2.64 %	+0.6 %
10		Objective-C	2.15 %	+0.1 %

https://pypl.github.io/PYPL.html
"The PYPL PopularitY of Programming Language Index is created by analyzing how often language tutorials are searched on Google."

Python is also widely used in scientific computing and data science

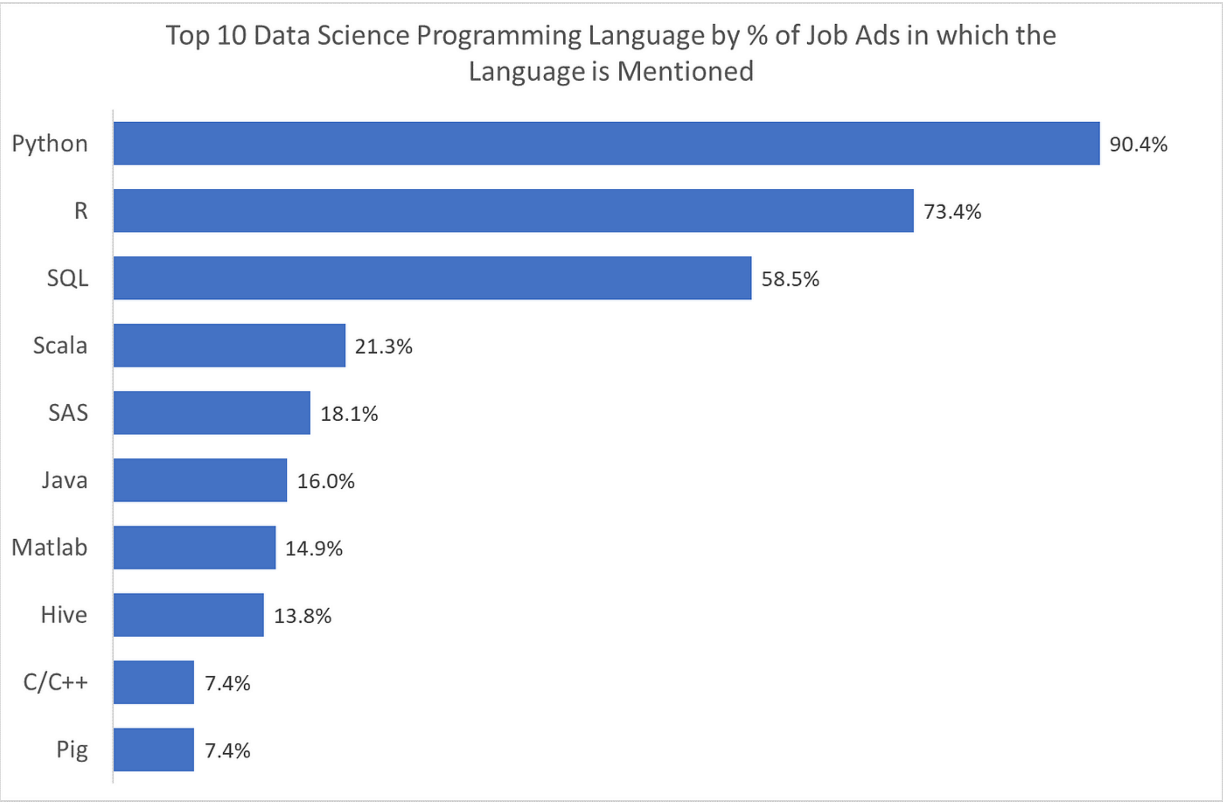
Mentions of Software in Astronomy Publications:



Compiled from NASA ADS (code).

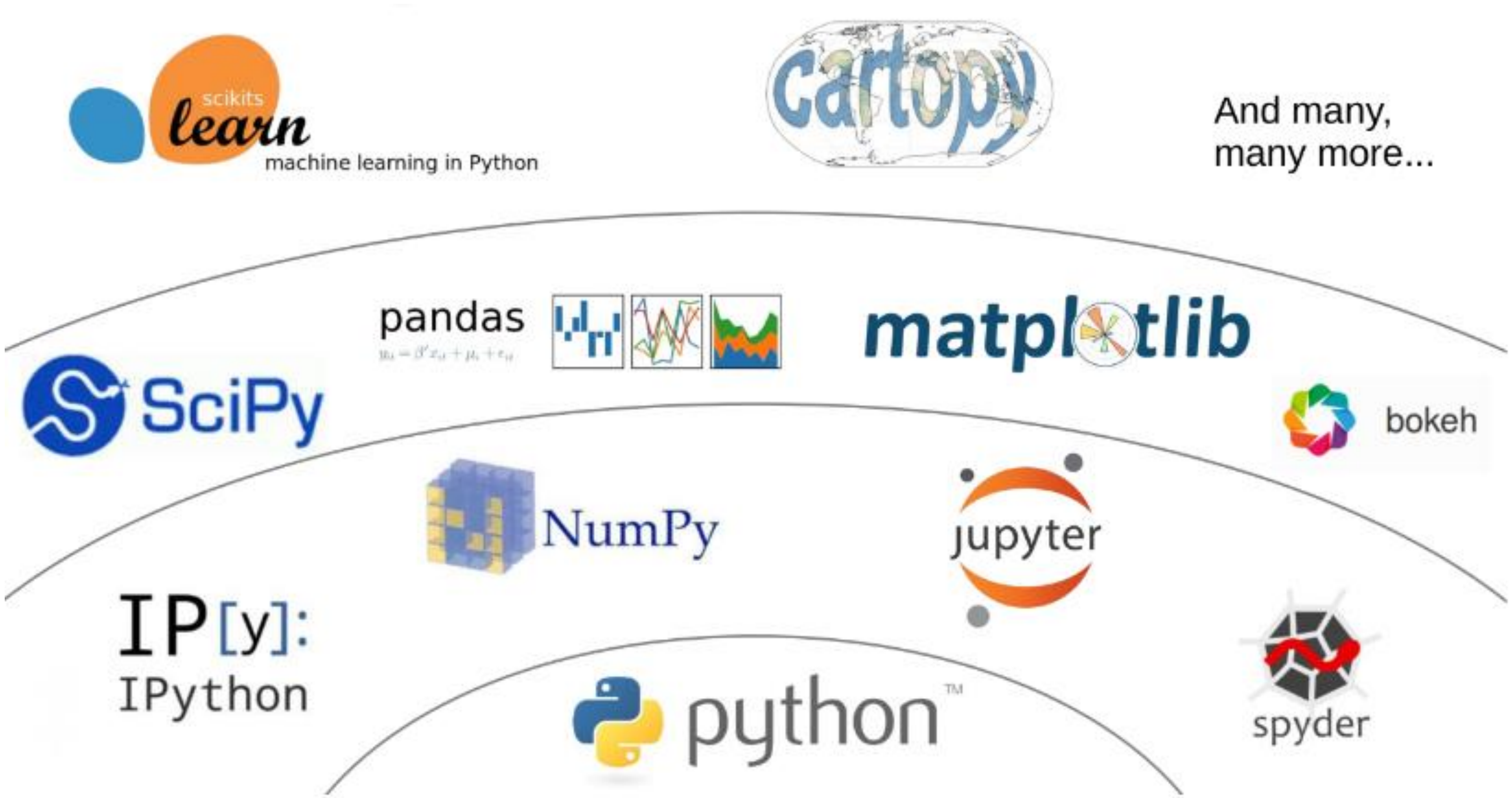
Thanks to Juan Nunez-Iglesias, Thomas P. Robitaille, and Chris Beaumont.

https://speakerdeck.com/jakevdp/the-unexpected-effectiveness-of-python-in-science?slide=32



https://towardsdatascience.com/which-programming-language-should-data-scientists-learn-first-aac4d3fd3038

Python’s rich ecosystem for scientific computing



And many,
many more...

https://speakerdeck.com/jakevdp/the-state-of-the-stack-scipy-2015-keynote

But, isn’t Python slow?

But, isn’t Python slow?

Python is great for HPC
with better compilers!

Thesis statement

Compilers that are aware of high-level operator and loop semantics can deliver improved performance for Python programs on CPUs and GPUs relative to past work

Thesis contributions

- APPy: Annotated Parallelism for Python on GPUs
  - [CC24] Parallelize Python loops and tensor expressions on GPUs
- ReACT: Redundancy-Aware Code Generation for Tensor Expressions
  - [PACT22] Redundancy elimination when fusing sparse/dense tensor operators
- Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels
  - [Onward!20] Compile Python/NumPy to C++ with high-level optimizations

Thesis contributions

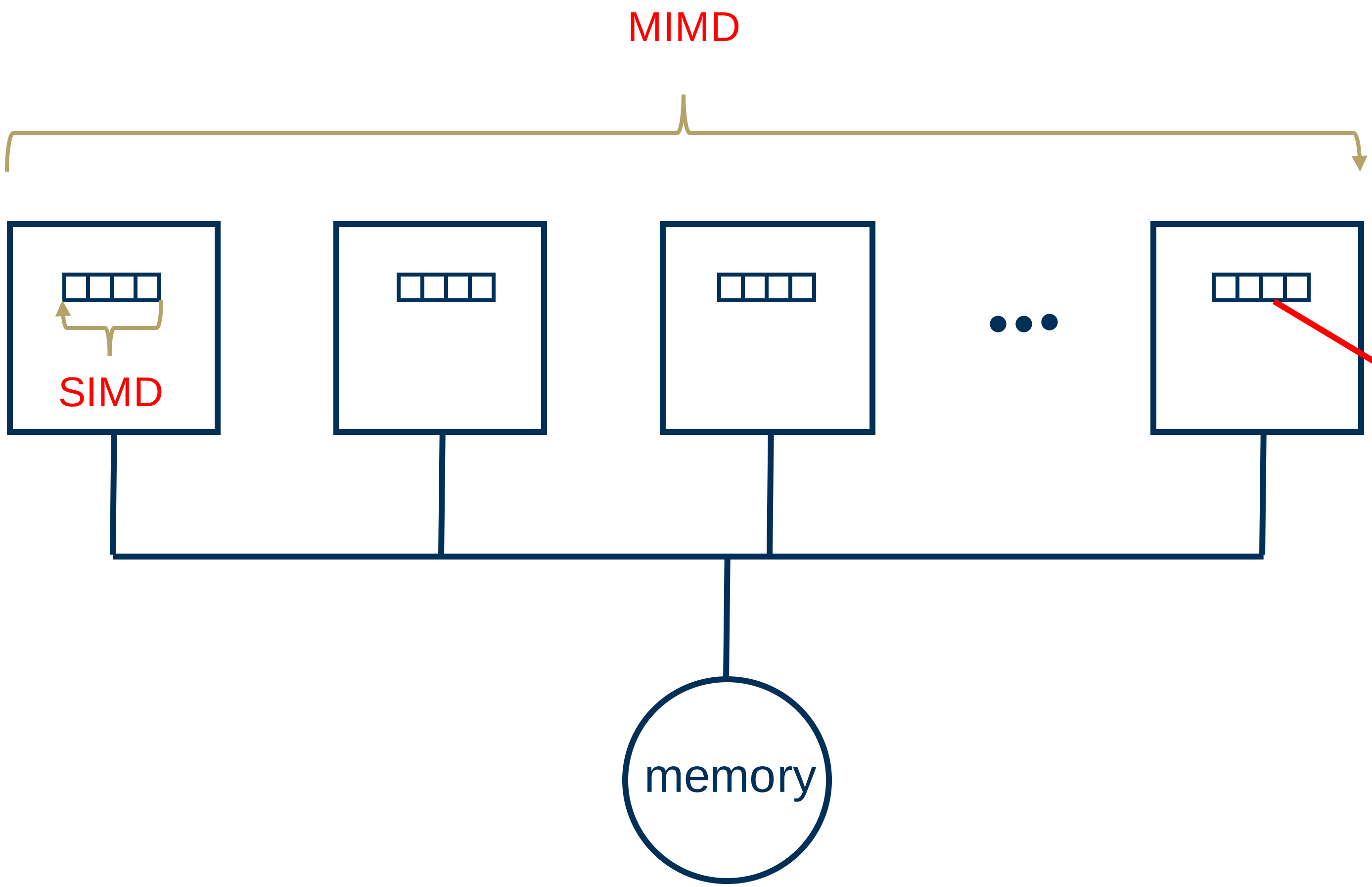
- APPy: Annotated Parallelism for Python on GPUs
  - [CC24] Parallelize Python loops and tensor expressions on GPUs
- ReACT: Redundancy-Aware Code Generation for Tensor Expressions
  - [PACT22] Redundancy elimination when fusing sparse/dense tensor operators
- Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels
  - [Onward!20] Compile Python/NumPy to C++ with high-level optimizations

Motivation for APPy

- Scientific Python programs can often benefit from using a GPU
- Two common approaches for GPU acceleration in Python
  - Library-based accelerations (e.g. CuPy), but many programs cannot be expressed using pre-defined operators alone
  - Creating custom CUDA/OpenCL kernels is challenging and time-consuming to get correctness and high performance
- Our solution (APPy)
  - Users write regular sequential Python code + annotate with simple pragmas
  - The compiler automatically generates GPU kernels from it

	CuPy	CUDA	APPy
Productivity	High	Low	High
Generality	Low	Very high	High

Abstract machine model: a multi-vector processor



Support cross-lane reduction
Maximum vector length:
appy.MVL

Support atomic update to memory locations

APPy compiler directives

- Annotations for loops
  - #pragma parallel for
  - #pragma parallel for single
  - #pragma simd
- Annotations for statements
  - #pragma atomic
- Annotations for tensor expressions
  - #pragma {dim}=>{properties}

- Difference from OpenMP codegen
  - OpenMP directly exposes the parallelism hierarchy of the GPUs and requires more complicated pragmas to generate GPU code
  - OpenMP does not recognize and compile tensor expressions

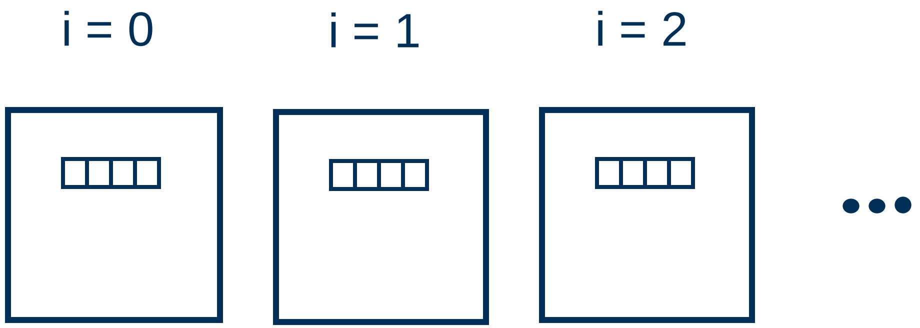
Vector addition with APPy

Software

```
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for
    for i in range(N):
        c[i] = a[i] + b[i]
```

N *workers* launched

Hardware
(abstract)



Utilize both layers of parallelism: parallel for + simd

N workers launched

```
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for
    for i in range(N):
        c[i] = a[i] + b[i]
```

Performance boost!

N / MVL workers launched

```
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for simd
    for i in range(N):
        c[i] = a[i] + b[i]
```

Utilize both layers of parallelism: parallel for + simd

N workers launched

```
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for
    for i in range(N):
        c[i] = a[i] + b[i]
```

Performance boost!

N / MVL workers launched

```
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for simd
    for i in range(N):
        c[i] = a[i] + b[i]
```

Compiler generated strip-mined loop

```
@appy.jit
def _generated(a, b, c, N):
    #pragma parallel for
    for i in range(0, N, MVL):
        c[i:i+MVL] = …
```

APPy allows you to use both loops and tensor expressions

Using loops is flexible, but sometimes it can be verbose …

Tensor operators can be more natural if applicable

Code simplified with tensor expressions

Use loop only

```
@appy.jit
def softmax_loop_oriented(a, b, M, N):
    #pragma parallel for
    for i in range(M):
        m = float('-inf’)
        #pragma simd
        for j in range(N):
            m = maximum(m, a[i,j])
        s = 0.0
        #pragma simd
        for j in range(N):
            s += exp(a[i,j] - m)
        #pragma simd
        for j in range(N):
            b[i,j] = exp(a[i,j] - m) / s
```

Use loop + tensor expressions

```
@appy.jit(auto_simd=True)
def softmax_tensor_oriented(a, b, M, N):
    #pragma parallel for
    for i in range(M):
        m = max(a[i,:N])
        s = sum(exp(a[i,:N] - m))
        b[i,:N] = exp(a[i,:N] - m) / s
```

The compiler automatically converts these tensor expressions into loops with operator fusion

Productivity improvement: 15 lines to 7 lines! (Also more readable)

Tensor-Oriented model

- Allows operating directly on tensors of arbitrary size as a whole
  - Tensor expressions need to be in the form of *sliced index notation*
    - C[:M, :N] = A[:M, :N] + B[:M, :N]
    - B[:M] = sum(A[:M, :N], axis=1)
    - A[:M, :N] = B[:M, None] + C[None, :N]
    - B[1:M-1, 1:N-1] = 0.2 * (A[1:M-1, 1:N-1] + A[1:M-1, :N-2] + A[1:M-1, 2:N] + …)
- Dimensions need to be annotated using syntax low:up=>prop1,prop2, …
  - Supported properties
    - Parallel, simd, reduction, le (small dimension optimization)
- More automatic compiler optimizations
  - Operator fusion
  - Synchronization reduction

Matrix vector multiplication using tensor expressions

- Loop order is determined by the order of the dimensions from left to right in the pragma
- The last dimension is automatically strip-mined with option auto_simd=True
- The optimal value of appy.MVL is automatically tuned from a list of common choices

```
@appy.jit(auto_simd=True)
def mv(alpha, A, x):
    M, N = A.shape
    #pragma :M=>parallel :N=>reduction(sum:y)
    y[:M] = mv(alpha * A[:M, :N], x[:N])
```

Compiler generated

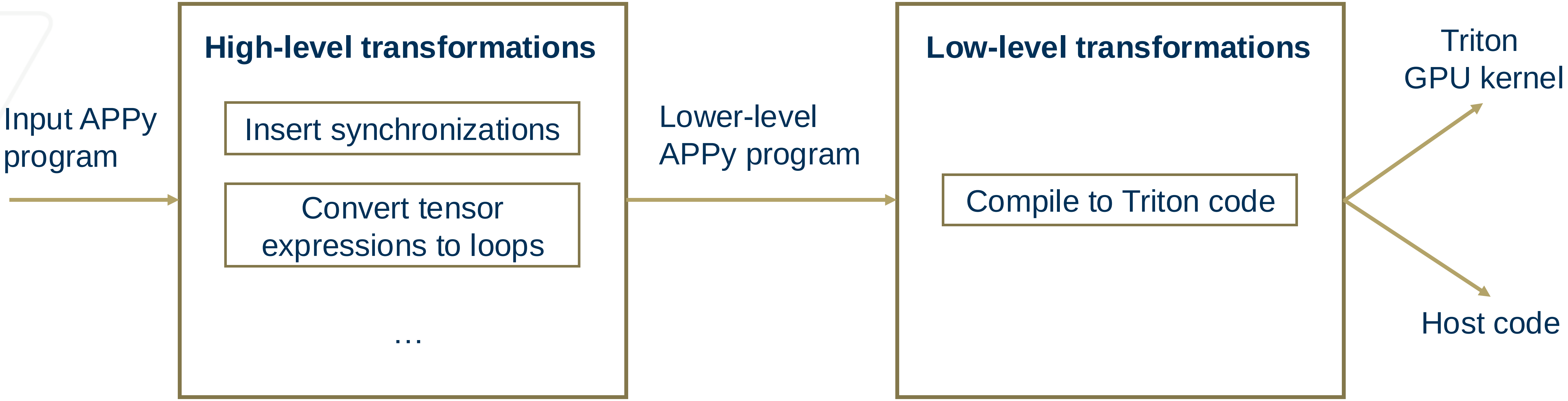
```
@appy.jit
def mv_generated(alpha, A, x):
    M, N = A.shape
    #pragma parallel for
    for _i0 in range(0, M, 1):
        y[_i0] = 0.0
        for _i1 in range(0, N, appy.MVL):
            _v1 = appy.vidx(_i1, appy.MVL, N)
            y[_i0] += sum(alpha * A[_i0, _v1] * x[_v1])
```

Map an APPy worker to GPU execution

- Mapping each worker to a thread block is more flexible, but their execution models don't match
  - APPy worker: statements execute sequentially
  - Thread block: multiple warps can execute asynchronously
- Solution: compiler automatically inserts thread synchronizations after memory operations to handle cross-thread dependence
  - Optimization 1: synchronizations are unnecessary if within loops generated from tensor expressions due to their regular computations
  - Optimization 2: if a tensor being written is never read in any other statements, then it cannot have data dependence with any other memory reads

Implementation

- All transformation passes are Python AST based



Input APPy program

High-level transformations

Insert synchronizations

Convert tensor expressions to loops

…

Lower-level APPy program

Low-level transformations

Compile to Triton code

Triton GPU kernel

Host code

A code generation example

```
@appy.jit(auto_simd=True)
def mv(alpha, A, x):
    M, N = A.shape
    #pragma :M=>parallel :N=>reduction(sum:y)
    y[:M] = mv(alpha * A[:M, :N], x[:N])
```

High-level transform

```
@appy.jit
def mv(alpha, A, x):
    M, N = A.shape
    #pragma parallel for
    for _i0 in range(0, M, 1):
        tmp = 0.0
        for _i1 in range(0, N, appy.MVL):
            _v1 = appy.vidx(_i1, appy.MVL, N)
            tmp += sum(alpha * A[_i0, _v1] * x[_v1])
        y[_i0] = tmp
```

Gen device code

```
@triton.jit
def _kernel(M, N, A, A_stride0, A_stride1, x, \
            x_stride0, y, y_stride0, MVL: tl.constexpr):
    _i0 = tl.program_id(0) * 1
    tmp = 0.0
    for _i1 in range(0, N, MVL):
        tmp += tl.sum(
            alpha * tl.load(
                    A + _i0*A_stride0 + \
                        _i1 + tl.arange(0, MVL),
                    mask=_i1 + tl.arange(0, MVL) < N
                ),
            tl.load(
                x + _i1 + tl.arange(0, MVL),
                mask=_i1 + tl.arange(0, MVL) < N
            )
        )
    tl.store(y + _i0, tmp)
```

Gen host code

```
def mv(alpha, A, x):
  M, N = A.shape
  MVL = 128; grid = (M,)
  _kernel[grid](M, N, A, A.stride(0), A.stride(1), \
    x, x.stride(0), y, y.stride(0), MVL)
```

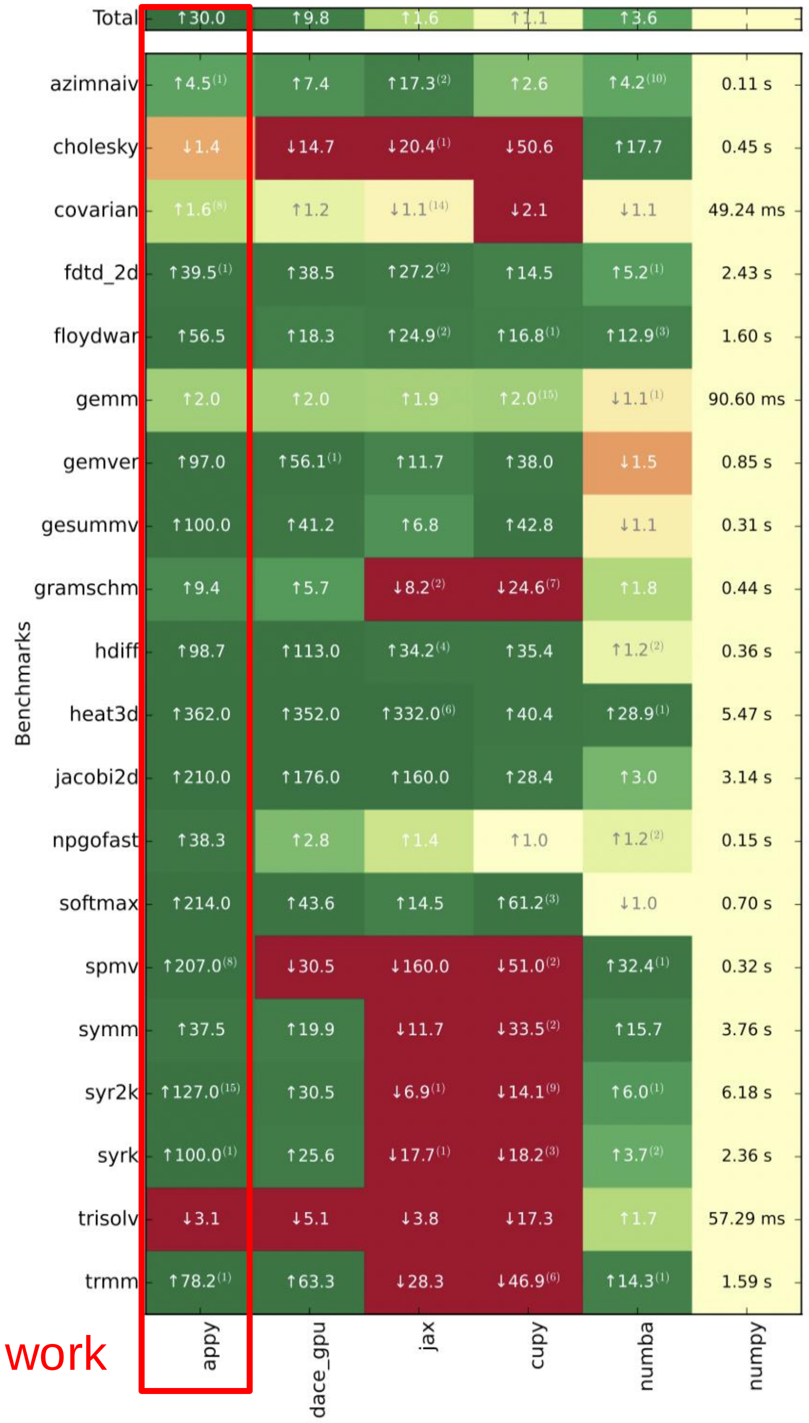
Performance evaluation

- CPU: Ryzen 7 5800X
  - 8 cores
  - Cache sizes
    - L1: 32K, L2: 512K, L3: 32M
- GPU: RTX 3090
  - 10496 cuda cores, 82 SMs
  - Cache sizes
    - L1: 128K, L2: 6M
- Benchmarking methodology
  - Each benchmark is run 10 times and report median
  - Each benchmark run is ~ 1 second
- Comparisons
  - NumPy (CPU library), CuPy (GPU library)
  - Numba (SOTA CPU compiler), JAX (SOTA JIT compiler with GPU backend), DaCe-GPU (SOTA GPU compiler)

- 20 kernels
  - azimint_naive
  - cholesky
  - covariance
  - fdtd_2d
  - floyd_warshall
  - gemm
  - gemver
  - gesummv
  - go_fast
  - gramschmidt
  - heat_3d
  - jacobi_1d
  - jacobi_2d
  - softmax
  - spmv
  - symm
  - syr2k
  - syrk
  - trisolv
  - trmm

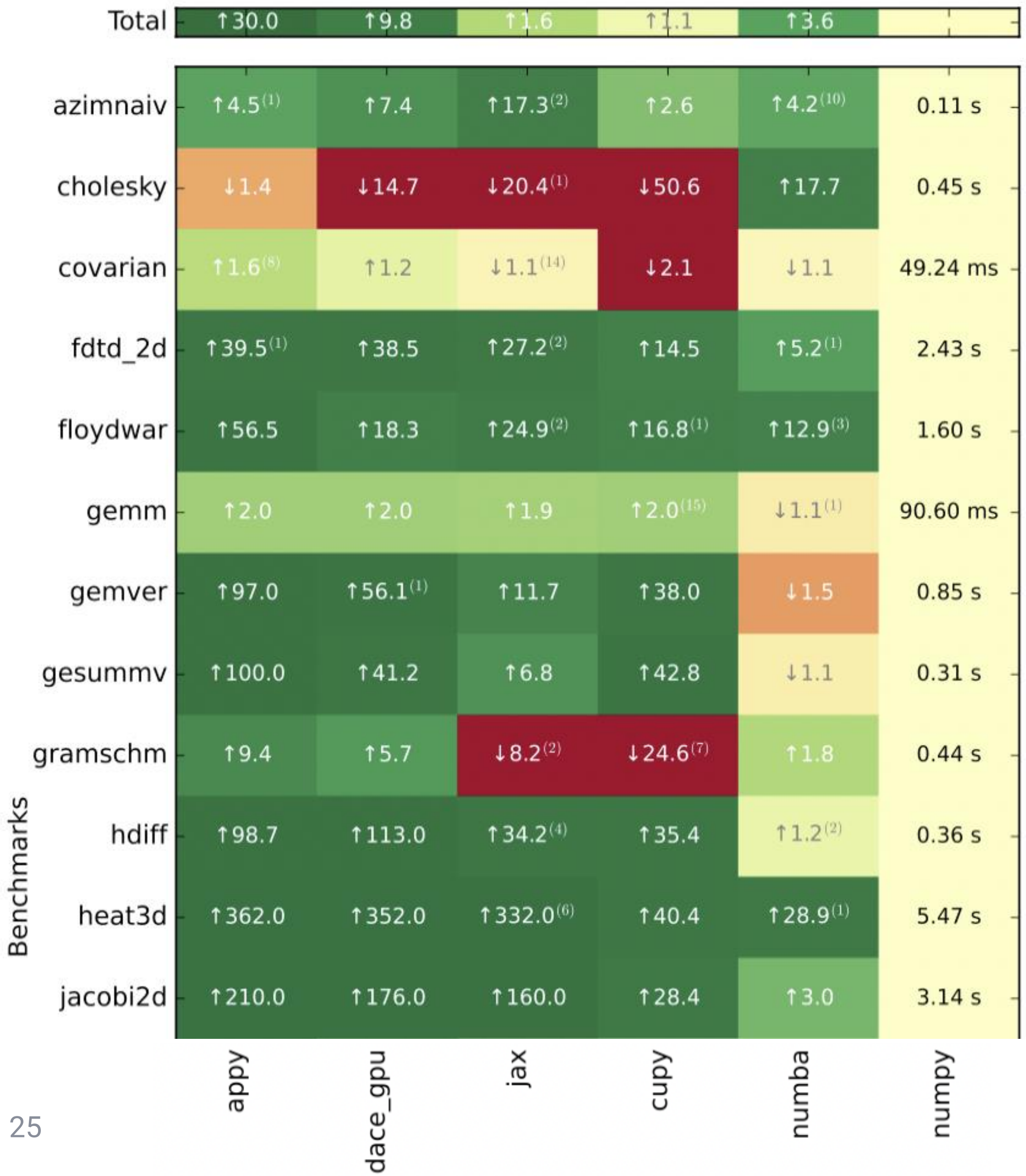
Performance results

- NumPy
  - Rightmost column shows absolute runtime
- Other frameworks: speedups/slowdown relative to NumPy
  - Acknowledgment: visualization script from npbench (ETH)
  - Up arrow indicates speedup (from light green to dark green)
  - Down arrow indicates slowdown (from orange to red)
- Summary of Appy's performance (geometric means)
  - 30x speedup over NumPy
  - 8.3x speedup over Numba
  - 30x speedup over CuPy
  - 18.8x speedup over JAX (with JIT)
  - 3.1x speedup over DaCe-GPU



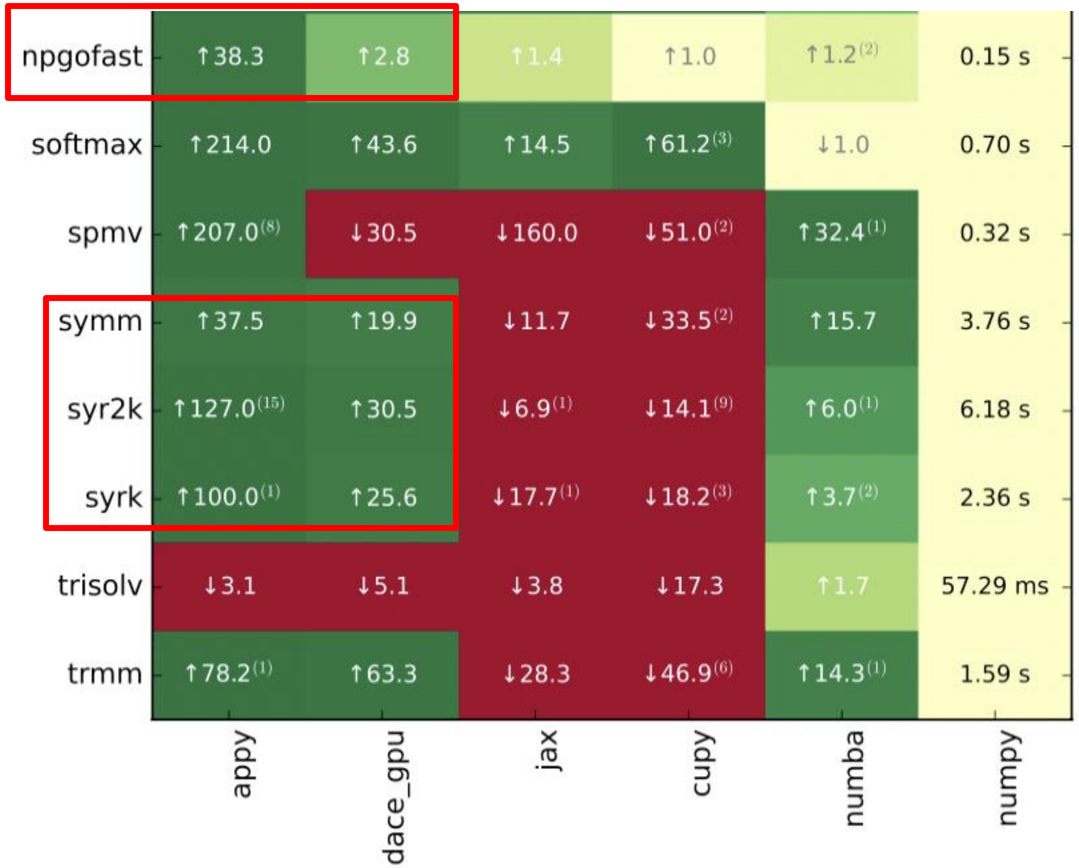
Benchmarks	appy	dace_gpu	jax	cupy	numba	numpy
Total	↑30.0	↑9.8	↑1.6	↑1.1	↑3.6	
azimnaiv	↑4.5$^{(1)}$	↑7.4	↑17.3$^{(2)}$	↑2.6	↑4.2$^{(10)}$	0.11 s
cholesky	↓1.4	↓14.7	↓20.4$^{(1)}$	↓50.6	↑17.7	0.45 s
covarian	↑1.6$^{(8)}$	↑1.2	↓1.1$^{(14)}$	↓2.1	↓1.1	49.24 ms
fdtd_2d	↑39.5$^{(1)}$	↑38.5	↑27.2$^{(2)}$	↑14.5	↑5.2$^{(1)}$	2.43 s
floydwar	↑56.5	↑18.3	↑24.9$^{(2)}$	↑16.8$^{(1)}$	↑12.9$^{(3)}$	1.60 s
gemm	↑2.0	↑2.0	↑1.9	↑2.0$^{(15)}$	↓1.1$^{(1)}$	90.60 ms
gemver	↑97.0	↑56.1$^{(1)}$	↑11.7	↑38.0	↓1.5	0.85 s
gesummv	↑100.0	↑41.2	↑6.8	↑42.8	↓1.1	0.31 s
gramschm	↑9.4	↑5.7	↓8.2$^{(2)}$	↓24.6$^{(7)}$	↑1.8	0.44 s
hdiff	↑98.7	↑113.0	↑34.2$^{(4)}$	↑35.4	↑1.2$^{(2)}$	0.36 s
heat3d	↑362.0	↑352.0	↑332.0$^{(6)}$	↑40.4	↑28.9$^{(1)}$	5.47 s
jacobi2d	↑210.0	↑176.0	↑160.0	↑28.4	↑3.0	3.14 s
npgofast	↑38.3	↑2.8	↑1.4	↑1.0	↑1.2$^{(2)}$	0.15 s
softmax	↑214.0	↑43.6	↑14.5	↑61.2$^{(3)}$	↓1.0	0.70 s
spmv	↑207.0$^{(8)}$	↓30.5	↓160.0	↓51.0$^{(2)}$	↑32.4$^{(1)}$	0.32 s
symm	↑37.5	↑19.9	↓11.7	↓33.5$^{(2)}$	↑15.7	3.76 s
syr2k	↑127.0$^{(15)}$	↑30.5	↓6.9$^{(1)}$	↓14.1$^{(9)}$	↑6.0$^{(1)}$	6.18 s
syrk	↑100.0$^{(1)}$	↑25.6	↓17.7$^{(1)}$	↓18.2$^{(3)}$	↑3.7$^{(2)}$	2.36 s
trisolv	↓3.1	↓5.1	↓3.8	↓17.3	↑1.7	57.29 ms
trmm	↑78.2$^{(1)}$	↑63.3	↓28.3	↓46.9$^{(6)}$	↑14.3$^{(1)}$	1.59 s

This work

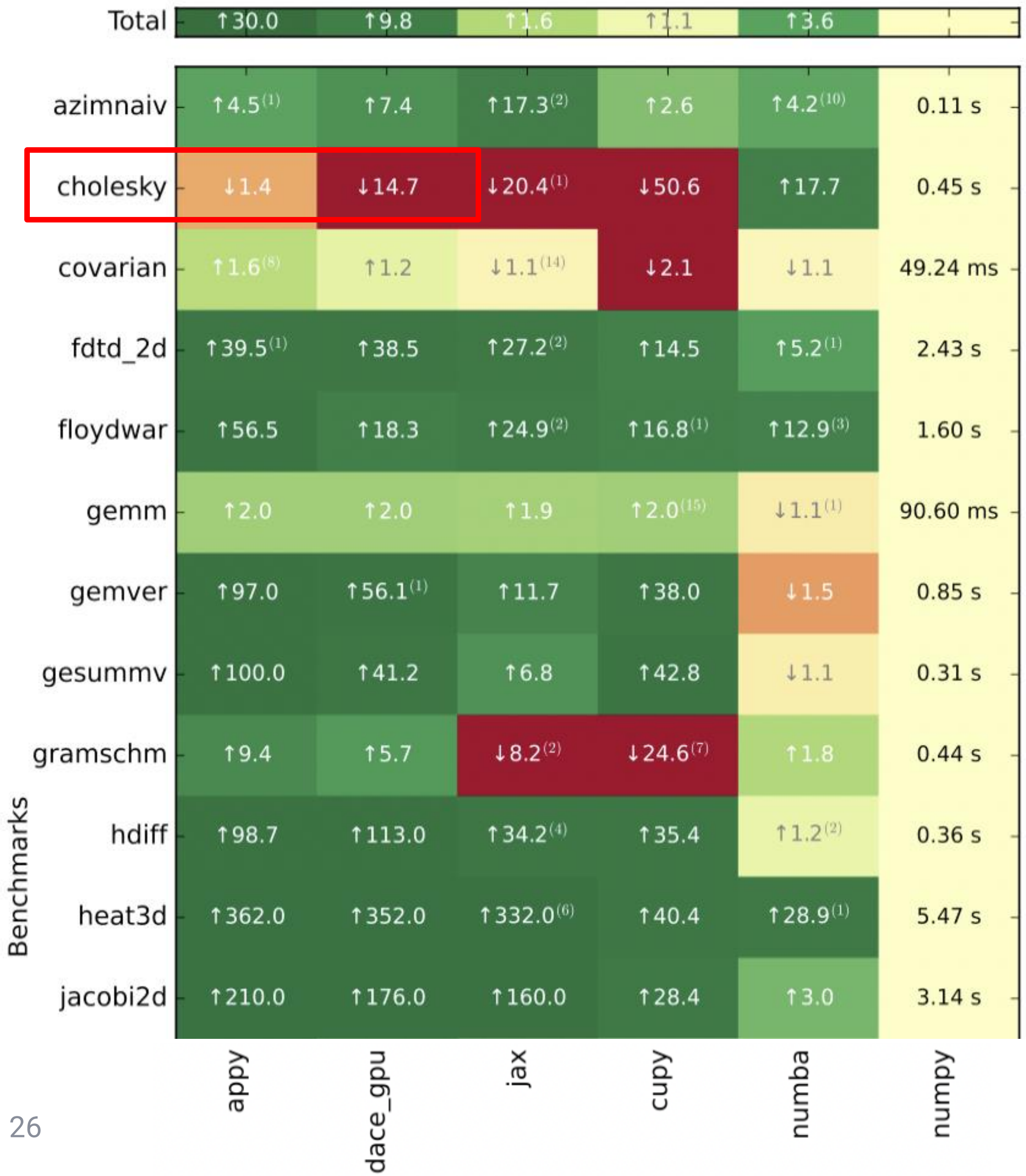


	appy	dace_gpu	jax	cupy	numba	numpy
Total	↑30.0	↑9.8	↑1.6	↑1.1	↑3.6	
azimnaiv	↑4.5$^{(1)}$	↑7.4	↑17.3$^{(2)}$	↑2.6	↑4.2$^{(10)}$	0.11 s
cholesky	↓1.4	↓14.7	↓20.4$^{(1)}$	↓50.6	↑17.7	0.45 s
covarian	↑1.6$^{(8)}$	↑1.2	↓1.1$^{(14)}$	↓2.1	↓1.1	49.24 ms
fdtd_2d	↑39.5$^{(1)}$	↑38.5	↑27.2$^{(2)}$	↑14.5	↑5.2$^{(1)}$	2.43 s
floydwar	↑56.5	↑18.3	↑24.9$^{(2)}$	↑16.8$^{(1)}$	↑12.9$^{(3)}$	1.60 s
gemm	↑2.0	↑2.0	↑1.9	↑2.0$^{(15)}$	↓1.1$^{(1)}$	90.60 ms
gemver	↑97.0	↑56.1$^{(1)}$	↑11.7	↑38.0	↓1.5	0.85 s
gesummv	↑100.0	↑41.2	↑6.8	↑42.8	↓1.1	0.31 s
gramschm	↑9.4	↑5.7	↓8.2$^{(2)}$	↓24.6$^{(7)}$	↑1.8	0.44 s
hdiff	↑98.7	↑113.0	↑34.2$^{(4)}$	↑35.4	↑1.2$^{(2)}$	0.36 s
heat3d	↑362.0	↑352.0	↑332.0$^{(6)}$	↑40.4	↑28.9$^{(1)}$	5.47 s
jacobi2d	↑210.0	↑176.0	↑160.0	↑28.4	↑3.0	3.14 s

Faster than DaCe due to some patterns are parallelized with APPy but sequentialized by DaCe

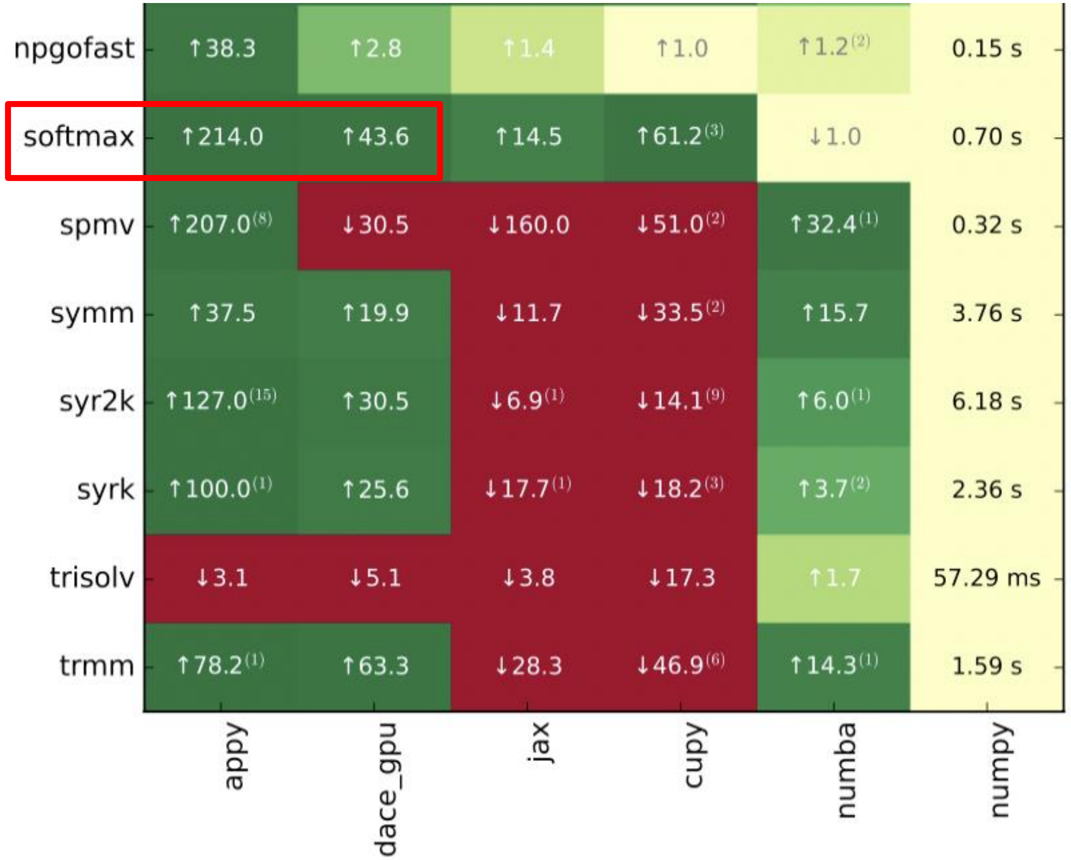


	appy	dace_gpu	jax	cupy	numba	numpy
npgofast	↑38.3	↑2.8	↑1.4	↑1.0	↑1.2$^{(2)}$	0.15 s
softmax	↑214.0	↑43.6	↑14.5	↑61.2$^{(3)}$	↓1.0	0.70 s
spmv	↑207.0$^{(8)}$	↓30.5	↓160.0	↓51.0$^{(2)}$	↑32.4$^{(1)}$	0.32 s
symm	↑37.5	↑19.9	↓11.7	↓33.5$^{(2)}$	↑15.7	3.76 s
syr2k	↑127.0$^{(15)}$	↑30.5	↓6.9$^{(1)}$	↓14.1$^{(9)}$	↑6.0$^{(1)}$	6.18 s
syrk	↑100.0$^{(1)}$	↑25.6	↓17.7$^{(1)}$	↓18.2$^{(3)}$	↑3.7$^{(2)}$	2.36 s
trisolv	↓3.1	↓5.1	↓3.8	↓17.3	↑1.7	57.29 ms
trmm	↑78.2$^{(1)}$	↑63.3	↓28.3	↓46.9$^{(6)}$	↑14.3$^{(1)}$	1.59 s



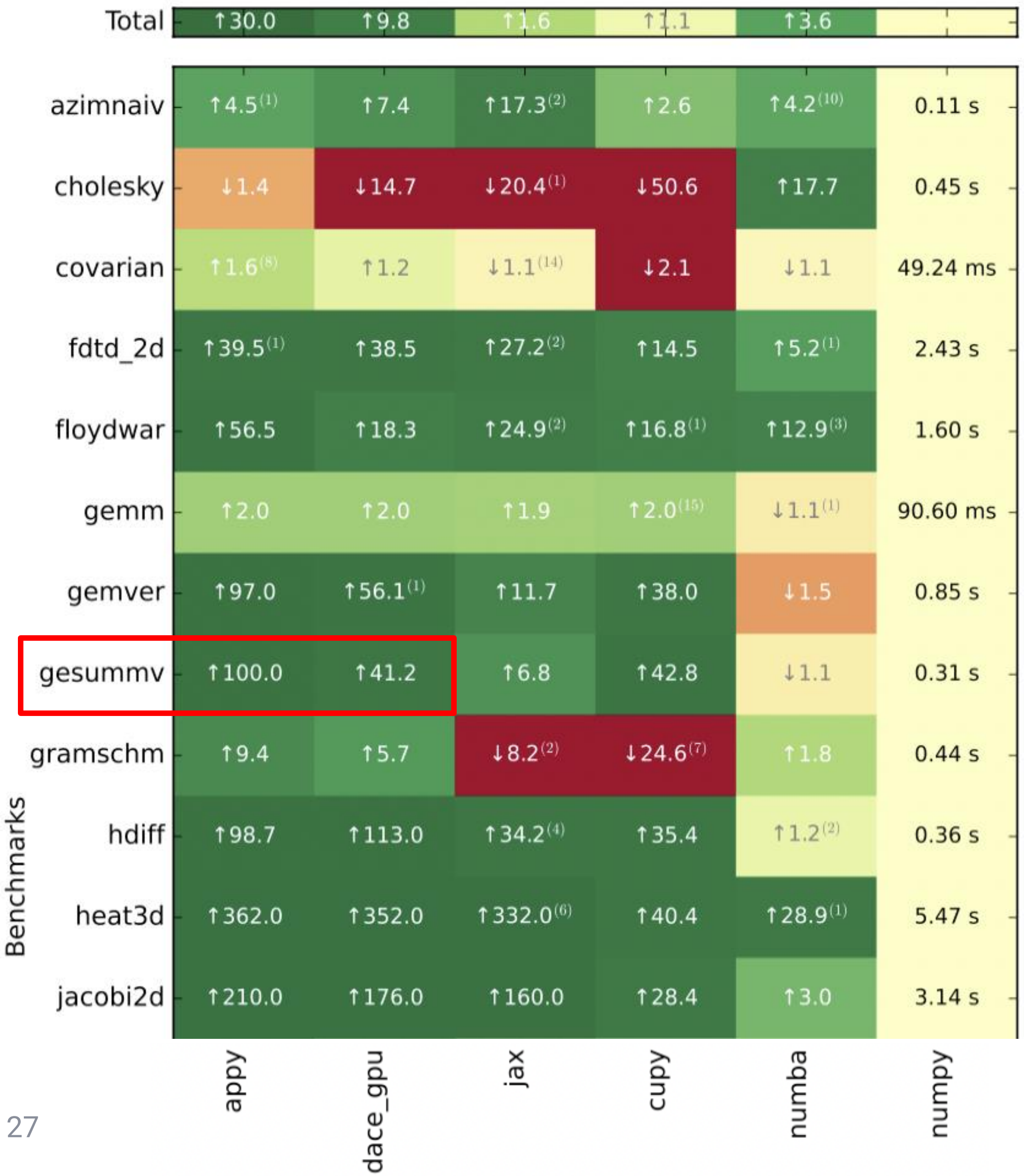
Benchmarks	appy	dace_gpu	jax	cupy	numba	numpy
Total	↑30.0	↑9.8	↑1.6	↑1.1	↑3.6	
azimnaiv	↑4.5$^{(1)}$	↑7.4	↑17.3$^{(2)}$	↑2.6	↑4.2$^{(10)}$	0.11 s
cholesky	↓1.4	↓14.7	↓20.4$^{(1)}$	↓50.6	↑17.7	0.45 s
covarian	↑1.6$^{(8)}$	↑1.2	↓1.1$^{(14)}$	↓2.1	↓1.1	49.24 ms
fdtd_2d	↑39.5$^{(1)}$	↑38.5	↑27.2$^{(2)}$	↑14.5	↑5.2$^{(1)}$	2.43 s
floydwar	↑56.5	↑18.3	↑24.9$^{(2)}$	↑16.8$^{(1)}$	↑12.9$^{(3)}$	1.60 s
gemm	↑2.0	↑2.0	↑1.9	↑2.0$^{(15)}$	↓1.1$^{(1)}$	90.60 ms
gemver	↑97.0	↑56.1$^{(1)}$	↑11.7	↑38.0	↓1.5	0.85 s
gesummv	↑100.0	↑41.2	↑6.8	↑42.8	↓1.1	0.31 s
gramschm	↑9.4	↑5.7	↓8.2$^{(2)}$	↓24.6$^{(7)}$	↑1.8	0.44 s
hdiff	↑98.7	↑113.0	↑34.2$^{(4)}$	↑35.4	↑1.2$^{(2)}$	0.36 s
heat3d	↑362.0	↑352.0	↑332.0$^{(6)}$	↑40.4	↑28.9$^{(1)}$	5.47 s
jacobi2d	↑210.0	↑176.0	↑160.0	↑28.4	↑3.0	3.14 s

Faster than DaCe due to small dimension optimization in APPy (cached in registers)

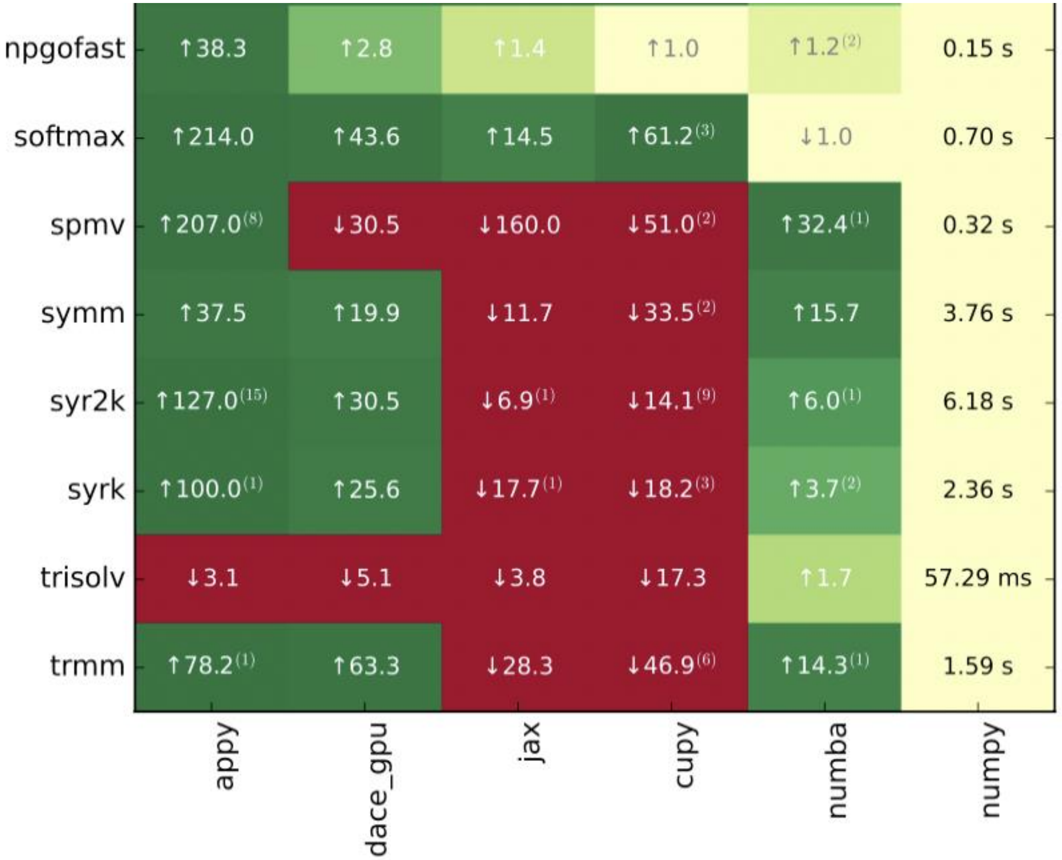


	appy	dace_gpu	jax	cupy	numba	numpy
npgofast	↑38.3	↑2.8	↑1.4	↑1.0	↑1.2$^{(2)}$	0.15 s
softmax	↑214.0	↑43.6	↑14.5	↑61.2$^{(3)}$	↓1.0	0.70 s
spmv	↑207.0$^{(8)}$	↓30.5	↓160.0	↓51.0$^{(2)}$	↑32.4$^{(1)}$	0.32 s
symm	↑37.5	↑19.9	↓11.7	↓33.5$^{(2)}$	↑15.7	3.76 s
syr2k	↑127.0$^{(15)}$	↑30.5	↓6.9$^{(1)}$	↓14.1$^{(9)}$	↑6.0$^{(1)}$	6.18 s
syrk	↑100.0$^{(1)}$	↑25.6	↓17.7$^{(1)}$	↓18.2$^{(3)}$	↑3.7$^{(2)}$	2.36 s
trisolv	↓3.1	↓5.1	↓3.8	↓17.3	↑1.7	57.29 ms
trmm	↑78.2$^{(1)}$	↑63.3	↓28.3	↓46.9$^{(6)}$	↑14.3$^{(1)}$	1.59 s

Faster than DaCe due to APPy generates fused code while DaCe does not

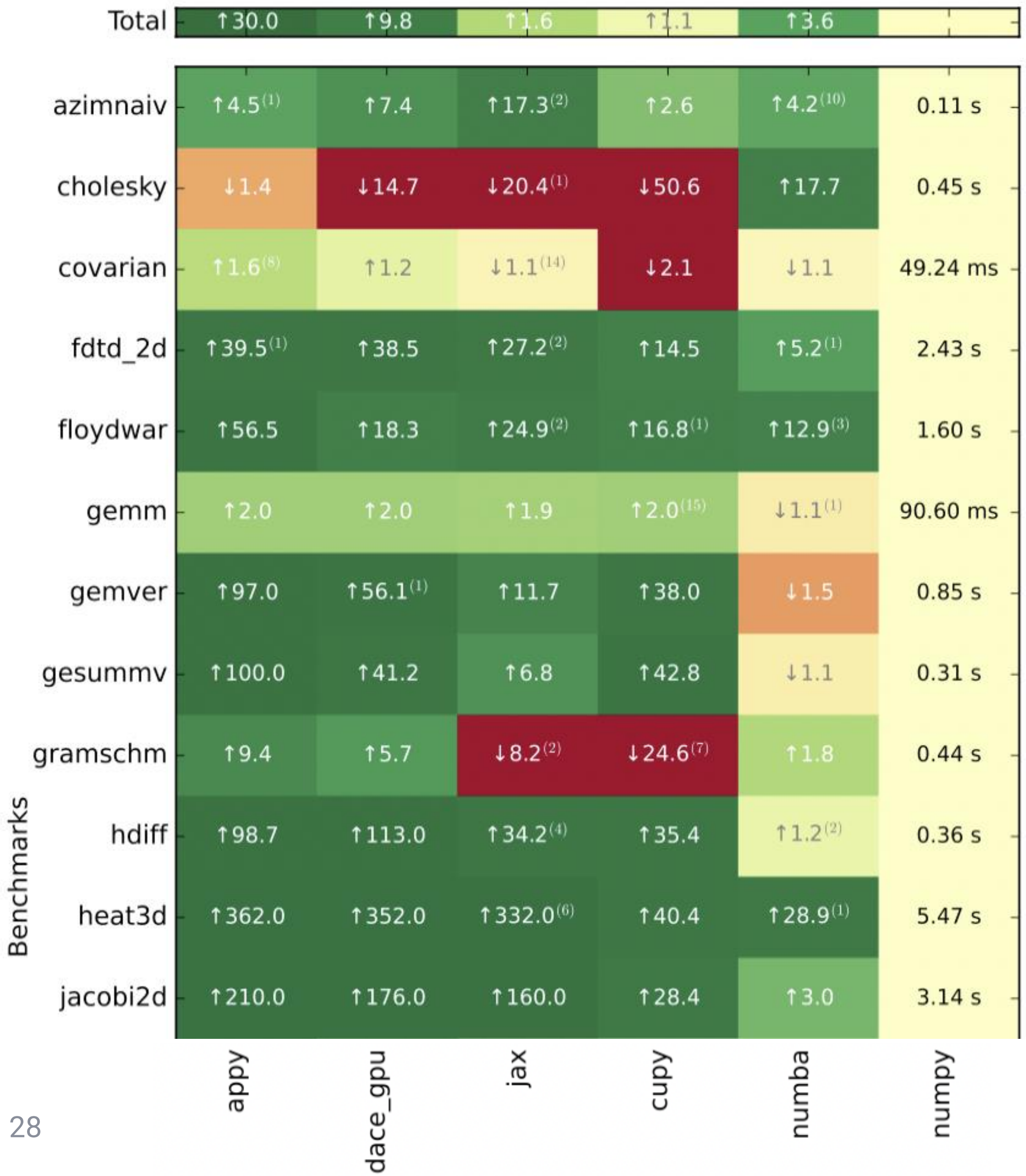


	appy	dace_gpu	jax	cupy	numba	numpy
Total	↑30.0	↑9.8	↑1.6	↑1.1	↑3.6	
azimnaiv	↑4.5(1)	↑7.4	↑17.3(2)	↑2.6	↑4.2(10)	0.11 s
cholesky	↓1.4	↓14.7	↓20.4(1)	↓50.6	↑17.7	0.45 s
covarian	↑1.6(8)	↑1.2	↓1.1(14)	↓2.1	↓1.1	49.24 ms
fdtd_2d	↑39.5(1)	↑38.5	↑27.2(2)	↑14.5	↑5.2(1)	2.43 s
floydwar	↑56.5	↑18.3	↑24.9(2)	↑16.8(1)	↑12.9(3)	1.60 s
gemm	↑2.0	↑2.0	↑1.9	↑2.0(15)	↓1.1(1)	90.60 ms
gemver	↑97.0	↑56.1(1)	↑11.7	↑38.0	↓1.5	0.85 s
gesummv	↑100.0	↑41.2	↑6.8	↑42.8	↓1.1	0.31 s
gramschm	↑9.4	↑5.7	↓8.2(2)	↓24.6(7)	↑1.8	0.44 s
hdiff	↑98.7	↑113.0	↑34.2(4)	↑35.4	↑1.2(2)	0.36 s
heat3d	↑362.0	↑352.0	↑332.0(6)	↑40.4	↑28.9(1)	5.47 s
jacobi2d	↑210.0	↑176.0	↑160.0	↑28.4	↑3.0	3.14 s

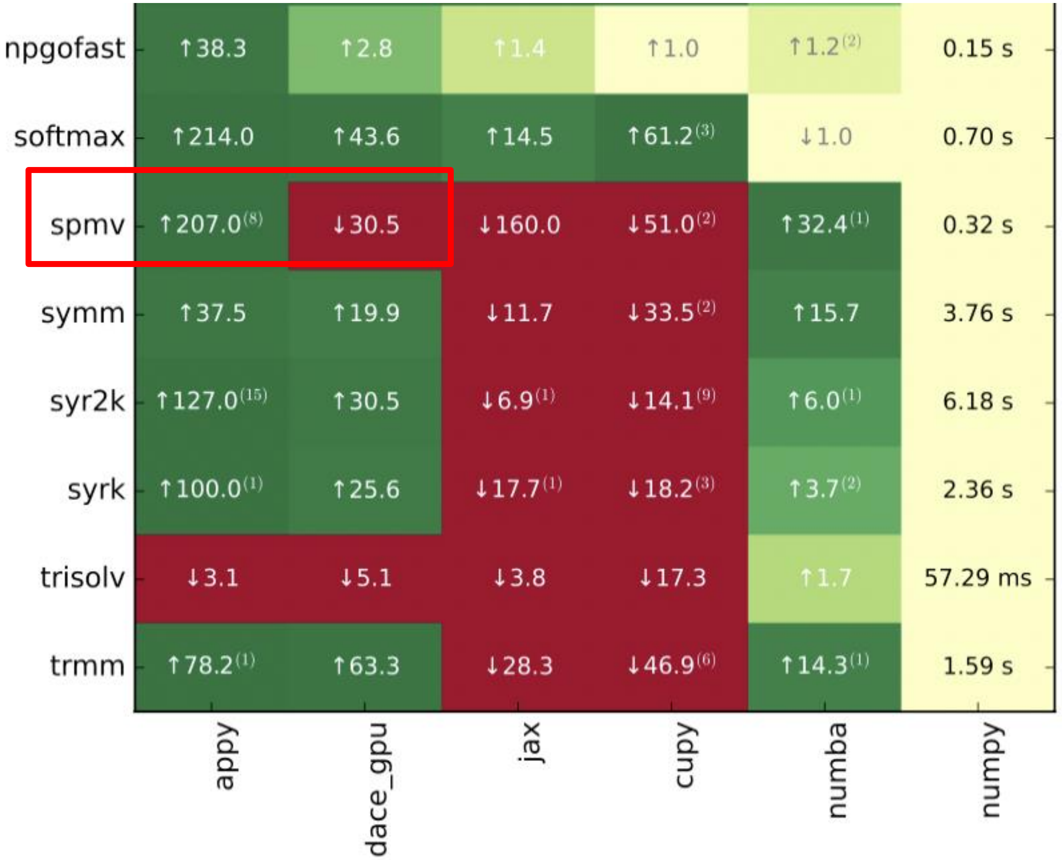


	appy	dace_gpu	jax	cupy	numba	numpy
npgofast	↑38.3	↑2.8	↑1.4	↑1.0	↑1.2(2)	0.15 s
softmax	↑214.0	↑43.6	↑14.5	↑61.2(3)	↓1.0	0.70 s
spmv	↑207.0(8)	↓30.5	↓160.0	↓51.0(2)	↑32.4(1)	0.32 s
symm	↑37.5	↑19.9	↓11.7	↓33.5(2)	↑15.7	3.76 s
syr2k	↑127.0(15)	↑30.5	↓6.9(1)	↓14.1(9)	↑6.0(1)	6.18 s
syrk	↑100.0(1)	↑25.6	↓17.7(1)	↓18.2(3)	↑3.7(2)	2.36 s
trisolv	↓3.1	↓5.1	↓3.8	↓17.3	↑1.7	57.29 ms
trmm	↑78.2(1)	↑63.3	↓28.3	↓46.9(6)	↑14.3(1)	1.59 s

Now show a code example: spmv



Benchmarks	appy	dace_gpu	jax	cupy	numba	numpy
Total	↑30.0	↑9.8	↑1.6	↑1.1	↑3.6	
azimnaiv	↑4.5$^{(1)}$	↑7.4	↑17.3$^{(2)}$	↑2.6	↑4.2$^{(10)}$	0.11 s
cholesky	↓1.4	↓14.7	↓20.4$^{(1)}$	↓50.6	↑17.7	0.45 s
covarian	↑1.6$^{(8)}$	↑1.2	↓1.1$^{(14)}$	↓2.1	↓1.1	49.24 ms
fdtd_2d	↑39.5$^{(1)}$	↑38.5	↑27.2$^{(2)}$	↑14.5	↑5.2$^{(1)}$	2.43 s
floydwar	↑56.5	↑18.3	↑24.9$^{(2)}$	↑16.8$^{(1)}$	↑12.9$^{(3)}$	1.60 s
gemm	↑2.0	↑2.0	↑1.9	↑2.0$^{(15)}$	↓1.1$^{(1)}$	90.60 ms
gemver	↑97.0	↑56.1$^{(1)}$	↑11.7	↑38.0	↓1.5	0.85 s
gesummv	↑100.0	↑41.2	↑6.8	↑42.8	↓1.1	0.31 s
gramschm	↑9.4	↑5.7	↓8.2$^{(2)}$	↓24.6$^{(7)}$	↑1.8	0.44 s
hdiff	↑98.7	↑113.0	↑34.2$^{(4)}$	↑35.4	↑1.2$^{(2)}$	0.36 s
heat3d	↑362.0	↑352.0	↑332.0$^{(6)}$	↑40.4	↑28.9$^{(1)}$	5.47 s
jacobi2d	↑210.0	↑176.0	↑160.0	↑28.4	↑3.0	3.14 s



	appy	dace_gpu	jax	cupy	numba	numpy
npgofast	↑38.3	↑2.8	↑1.4	↑1.0	↑1.2$^{(2)}$	0.15 s
softmax	↑214.0	↑43.6	↑14.5	↑61.2$^{(3)}$	↓1.0	0.70 s
spmv	↑207.0$^{(8)}$	↓30.5	↓160.0	↓51.0$^{(2)}$	↑32.4$^{(1)}$	0.32 s
symm	↑37.5	↑19.9	↓11.7	↓33.5$^{(2)}$	↑15.7	3.76 s
syr2k	↑127.0$^{(15)}$	↑30.5	↓6.9$^{(1)}$	↓14.1$^{(9)}$	↑6.0$^{(1)}$	6.18 s
syrk	↑100.0$^{(1)}$	↑25.6	↓17.7$^{(1)}$	↓18.2$^{(3)}$	↑3.7$^{(2)}$	2.36 s
trisolv	↓3.1	↓5.1	↓3.8	↓17.3	↑1.7	57.29 ms
trmm	↑78.2$^{(1)}$	↑63.3	↓28.3	↓46.9$^{(6)}$	↑14.3$^{(1)}$	1.59 s

Sparse matrix dense vector multiplication (SpMV)

CuPy version

```
def spmv(A_row, A_col, A_val, x):
    N = A_row.shape[0]
    y = empty([N - 1], dtype=A_val.dtype)
    for i in range(N - 1):
        cols = A_col[A_row[i]:A_row[i + 1]]
        vals = A_val[A_row[i]:A_row[i + 1]]
        y[i] = dot(vals, x[cols])
    return y
```

APPy version

10557x speedup! [1]

```
@appy.jit
def spmv(A_row, A_col, A_val, x):
    N = A_row.shape[0]
    y = empty([N - 1], dtype=A_val.dtype)
    #pragma parallel for
    for i in range(N - 1):
        y[i] = 0.0
        #pragma simd
        for j in range(A_row[i], A_row[1+i]):
            cols = A_col[j]
            y[i] += A_val[j] * x[cols]
    return y
```

Dynamic loop bounds are fine with #pragma simd

[1] Testing machine is a RTX 3090 GPU and the baseline NumPy runtime is ~0.3 seconds

More results explanation

- Why faster than JAX (with JIT)?
  - Parallelizable loops are parallelized by APPy but sequentialized by JAX
  - APPy fuses some operator sequence pattern that’s not fused by JAX
- Why faster than CuPy?
  - Loop-based CuPy kernels run the loops sequentially in the Python interpreter while APPy runs them in parallel in native code
  - Operator-based CuPy kernels have memory inefficiency due to the need to materialize intermediate results for a sequence of operators while APPy does operator fusion
- Why faster than NumPy/Numba?
  - GPUs are known to be more efficient than CPUs for data parallel applications

APPy summary

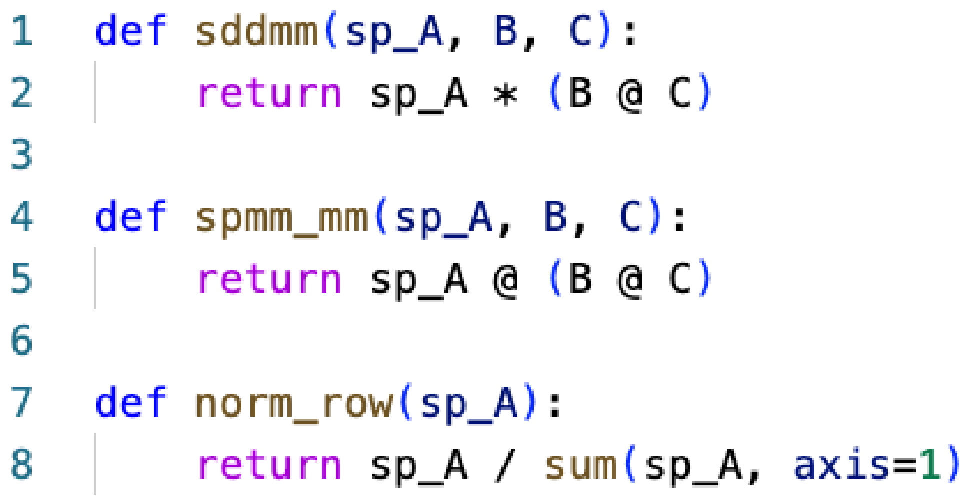
- We present APPy, a Python-based programming model and compiler that allows users to parallelize sequential Python code on GPUs using compiler directives
- We present the design of a loop-oriented programming model and a tensor-oriented programming model, and their implementations, including code generation and automatic compiler optimizations
- We evaluate the performance of APPy using 20 kernels from scientific computing and demonstrate significant speedup over CuPy (30× on average), JAX (18.8× on average), and DaCe-GPU (3.1× on average)

Thesis contributions

- APPy: Annotated Parallelism for Python on GPUs
  - [CC24] Parallelize Python loops and tensor expressions on GPUs
- ReACT: Redundancy-Aware Code Generation for Tensor Expressions
  - [PACT22] Redundancy elimination when fusing sparse/dense tensor operators
- Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels
  - [Onward!20] Compile Python/NumPy to C++ with high-level optimizations

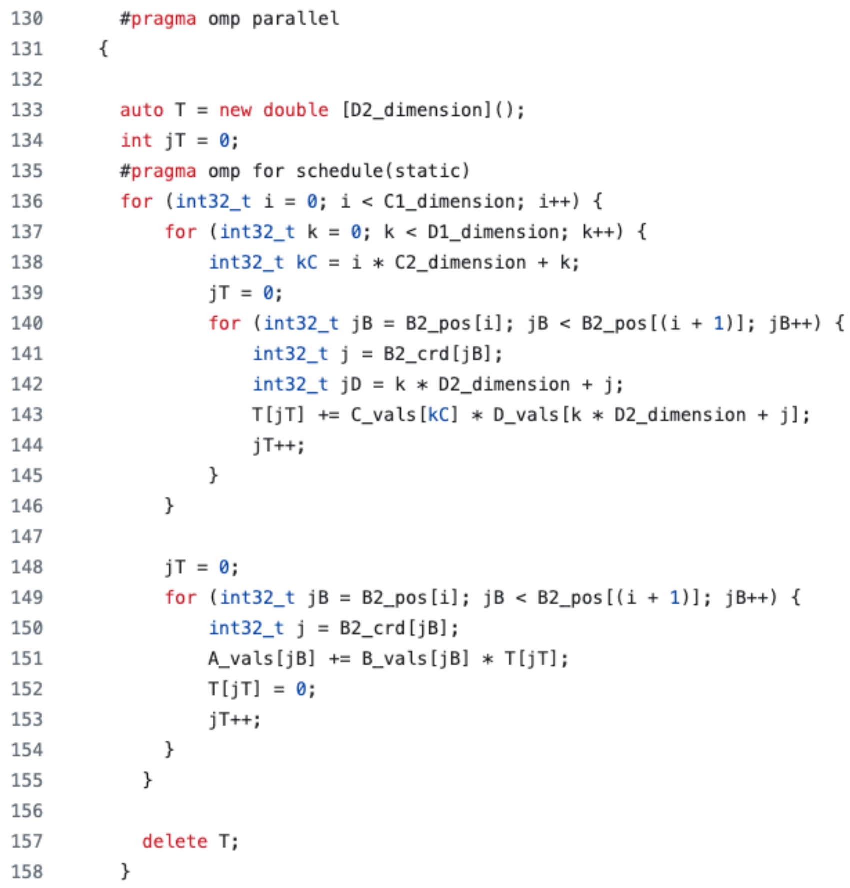
Problem statement: desired input and output

- Desired input: operator program in Python (can be sparse)



```python
def sddmm(sp_A, B, C):
    return sp_A * (B @ C)

def spmm_mm(sp_A, B, C):
    return sp_A @ (B @ C)

def norm_row(sp_A):
    return sp_A / sum(sp_A, axis=1)
```

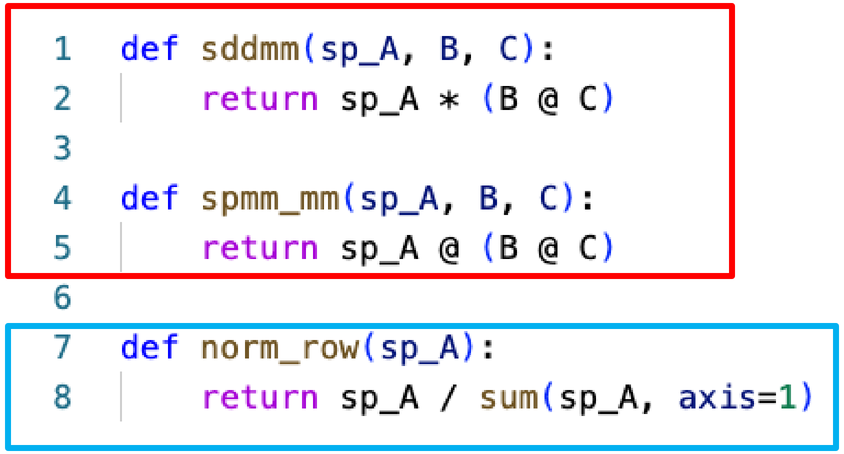
- Desired output: fused CPU kernel with reduced redundant memory accesses and computations



```cpp
    #pragma omp parallel
  {

    auto T = new double [D2_dimension]();
    int jT = 0;
    #pragma omp for schedule(static)
    for (int32_t i = 0; i < C1_dimension; i++) {
        for (int32_t k = 0; k < D1_dimension; k++) {
            int32_t kC = i * C2_dimension + k;
            jT = 0;
            for (int32_t jB = B2_pos[i]; jB < B2_pos[(i + 1)]; jB++) {
                int32_t j = B2_crd[jB];
                int32_t jD = k * D2_dimension + j;
                T[jT] += C_vals[kC] * D_vals[k * D2_dimension + j];
                jT++;
            }
        }

        jT = 0;
        for (int32_t jB = B2_pos[i]; jB < B2_pos[(i + 1)]; jB++) {
            int32_t j = B2_crd[jB];
            A_vals[jB] += B_vals[jB] * T[jT];
            T[jT] = 0;
            jT++;
        }
      }

      delete T;
    }
```

Limitations with State-of-the-art

- TACO
  - A code generator for arbitrary sparse/dense tensor algebra expressions
  - **maximal fusion** is implicit during code generation
- Limitations
  - Maximal fusion may introduce some types of redundant memory accesses and computations
  - Maximal fusion cannot properly fuse certain reduction expressions



```python
def sddmm(sp_A, B, C):
    return sp_A * (B @ C)

def spmm_mm(sp_A, B, C):
    return sp_A @ (B @ C)

def norm_row(sp_A):
    return sp_A / sum(sp_A, axis=1)
```

Maximal fusion does not work because
it requires the “/” operator to be distributive over a summation

Redundancy types identified

- **Type 1** (Reduction Redundancy): When multiple multiply-add operations are performed instead of multiple adds followed by a single multiply (distributive law).
- **Type 2** (Loop-Invariant Redundancy): When a loop invariant expression is introduced (could be invariant in a non-innermost loop) due to maximum fusion.
- **Type 3** (Load-Store Redundancy): When some values are stored and loaded in separate loops, and the loads/stores can be eliminated after fusion --- a classical benefit of loop fusion.
- **Type 4** (Dead-Value Redundancy): When some values are computed but not used later on (e.g., when multiplying with 0s in a sparse tensor) --- another classical benefit of loop fusion.

(Type 1) Reduction redundancy

Input: c = b * sum(A, axis=1)

With redundancy (due to maximal fusion)

```
for (int i = 0; i < NI; i++) {
  double s = 0;
  double bi = b[i];
  for (int j = 0; j < NJ; j++) {
    s += A[i,j] * bi;
    ...
  }
  ...
}
```

Without redundancy

```
for (int i = 0; i < NI; i++) {
  double s = 0;
  for (int j = 0; j < NJ; j++) {
    s += A[i,j];
    ...
  }
  s = s * B[i];
  ...
}
```

Reduced number of multiplications in the innermost loop!

(Type 2) Loop-Invariant redundancy

Input: A = (B + E) * (C @ D)

With redundancy (due to maximal fusion)

```
for (int i = 0; i < NI; i++)
  for (int k = 0; k < NK; k++)
    for (int j = 0; j < NJ; j++)
      A[i,j] += (B[i,j] + E[i,j]) * \ (C[i,k] * D[k,j]);
```

Without redundancy

```
double* T = new double[NJ];
for (int i = 0; i < NI; i++) {
  for (int j = 0; j < NJ; j++) {
    T[j] = B[i,j] + E[i,j];
  }
  for (int k = 0; k < NK; k++) {
    for (int j = 0; j < NJ; j++) {
      A[i,j] += T[j] * (C[i,k] * D[k,j]);
    }
  }
}
```

B[i,j] + E[i,j] is no longer repeatedly calculated for different *k* iterations!

(Type 3) Load-Store redundancy

Input: s = sum(A, axis=1); B = A / s[:, None]

With redundancy (due to no fusion)

```
double* s = new double[NI];
// Operator 1
for (int i = 0; i < NI; i++) {
  s[i] = 0;
  for (int j = 0; j < NJ; j++) {
    s[i] += A[i,j];
  }
}
// Operator 2
for (int i = 0; i < NI; i++) {
  for (int j = 0; j < NJ; j++) {
    B[i,j] = A[i,j] / s[i];
  }
}
```

Without redundancy

```
// Operator 1 and 2 fused
for (int i = 0; i < NI; i++) {
  double s = 0;
  for (int j = 0; j < NJ; j++) {
    s += A[i,j];
  }

  for (int j = 0; j < NJ; j++) {
    B[i,j] = A[i,j] / s;
  }
}
```

A[i,j] and s[i] now have reduced reuse distance, which leads to better locality!

(Type 4) Dead-Value redundancy

Input: B = where(A < 0, alpha * A, A)

With redundancy (due to no fusion)

```
// Operator 1
double* tmp = new double[NI];
for (int i = 0; i < NI; i++) {
  tmp[i] = alpha * A[i];
}
// Operator 2
for (int i = 0; i < NI; i++) {
  if (A[i] < 0) {
    B[i] = tmp[i];
  }
  else {
    B[i] = A[i];
  }
}
```

Not all values in array tmp are useful!

Without redundancy

```
// Operator 1 and 2 fused
for (int i = 0; i < NI; i++) {
  if (A[i] < 0) {
    B[i] = alpha * A[i];
  }
  else {
    B[i] = A[i];
  }
}
```

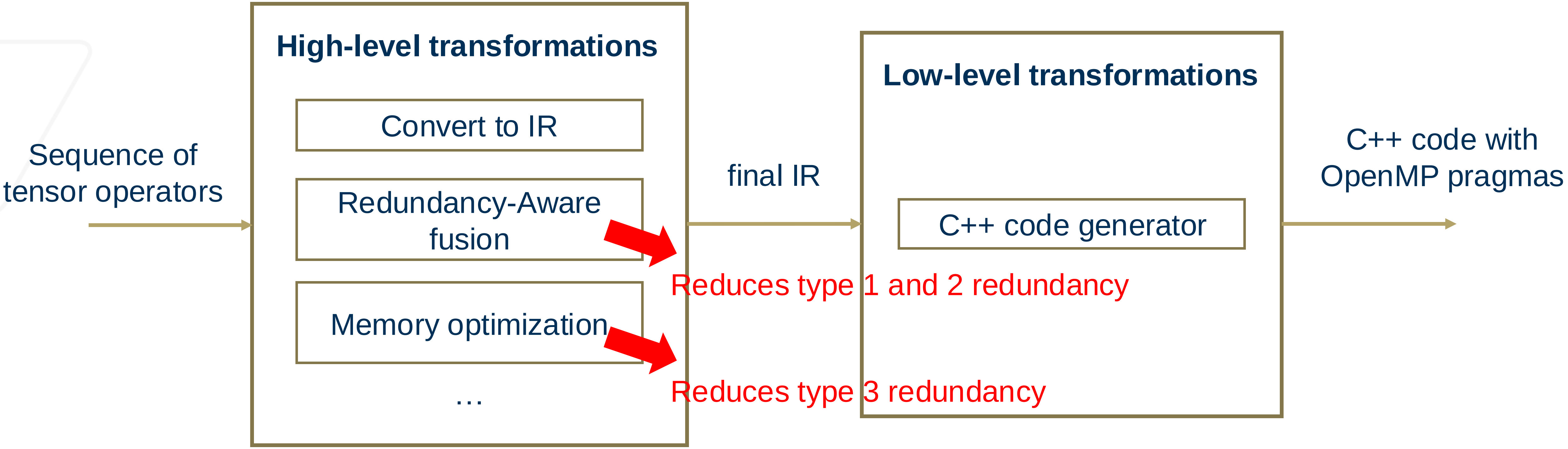
The use of tmp is now eliminated, which reduces redundant computations and memory accesses!

Redundancies eliminated by each approach

Redundancy type	ReACT (this work)	TACO	SciPy
Reduction (type 1)	Yes	No	Yes
Loop invariant (type 2)	Yes	No	Yes
Load store (type 3)	Yes	Partially	No
Dead value (type 4)	Yes	Yes	No

How is ReACT able to reduce these redundancies?

Transformation passes are redundancy-aware



Sequence of tensor operators

High-level transformations

Convert to IR

Redundancy-Aware fusion

Memory optimization

…

final IR

Low-level transformations

C++ code generator

C++ code with OpenMP pragmas

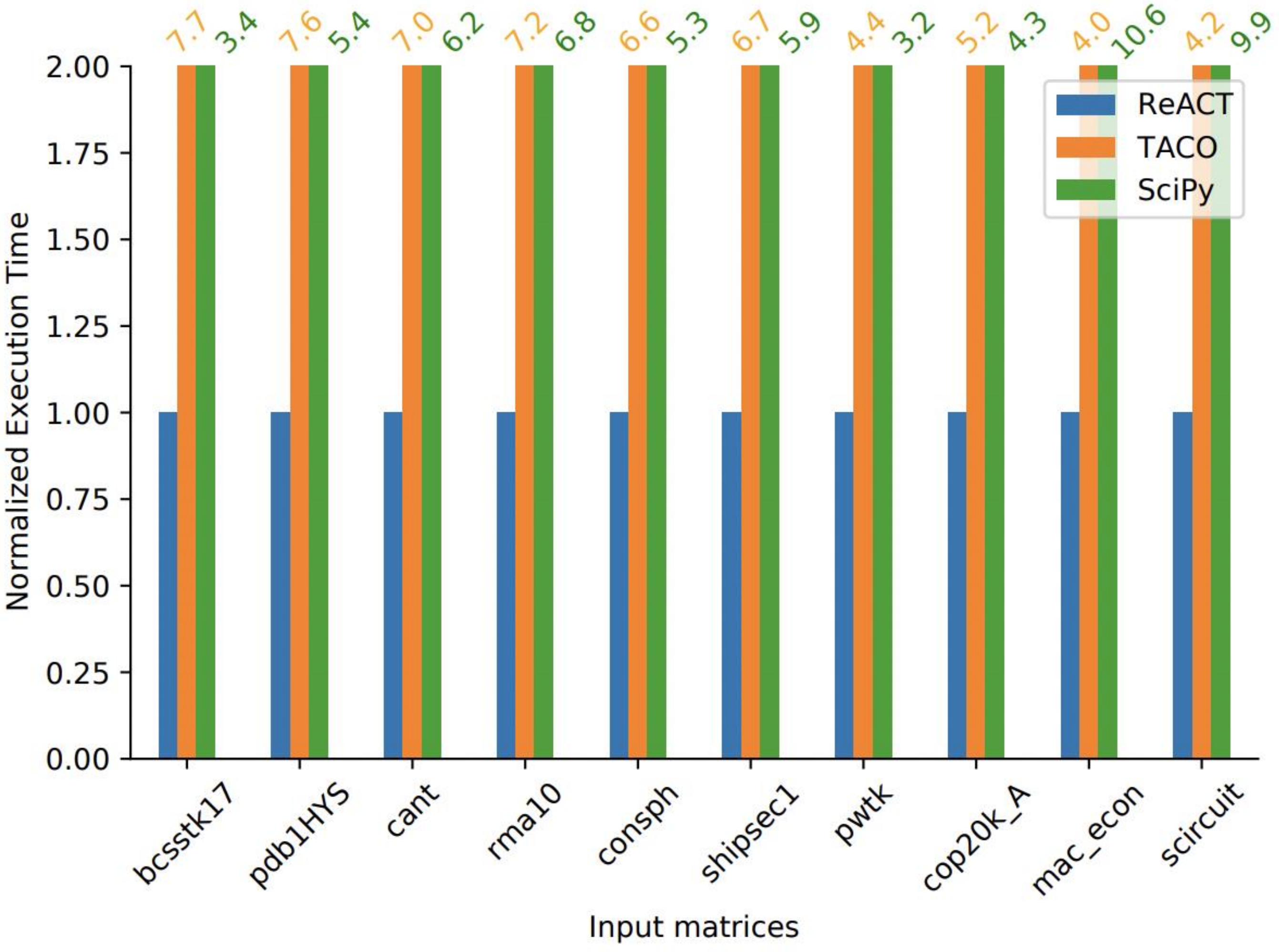
Reduces type 1 and 2 redundancy

Reduces type 3 redundancy

Performance evaluation

- Test machine
  - 16-core Intel(R) Xeon(R) 2.20GHz CPU
  - OMP_NUM_THREADS is set to 16
- Kernels (all kernels have at least 2 operators)
  - SpMM-MM (sparse-dense matmul followed by dense matmul)
  - SDDMM/Masked MM (a dense matmul followed by a dense-sparse element-wise mul)
  - Sparse-softmax (row-wise softmax on a sparse matrix)
    - Expressed using basic operators such as exp, sum, divide etc
- Sparse matrices
  - A collection of real-world matrices from SuiteSparse
  - All sparse matrices are in CSR format
- Comparisons
  - ReACT (our approach)
  - TACO (SOTA compiler)
  - SciPy.sparse (SOTA library)

SpMM-MM results – 5.9x faster than TACO



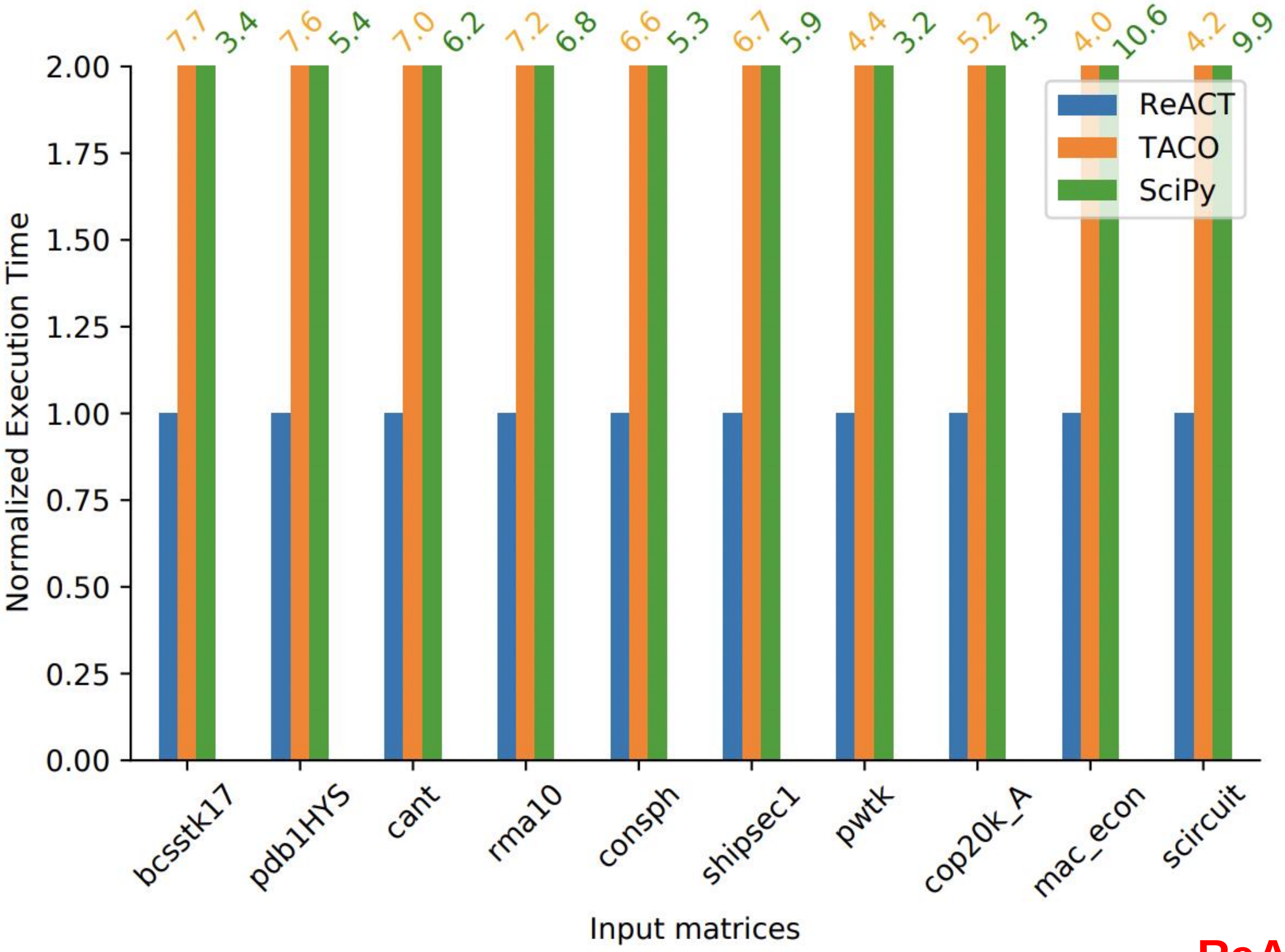
(b) GNN-kernel1 (NH=256, NJ=16)

“No” is good here!

Redundancy types present	TACO output	ReACT output
Type 1	Yes	No
Type 2	Yes	No
Type 3	No	No
Type 4	No	No

Code time complexity is reduced from $O(NNZ * NH * NJ)$ (TACO) to $O(NI * NH * NJ)$ (ReACT)

SpMM-MM results – 5.7x faster than SciPy



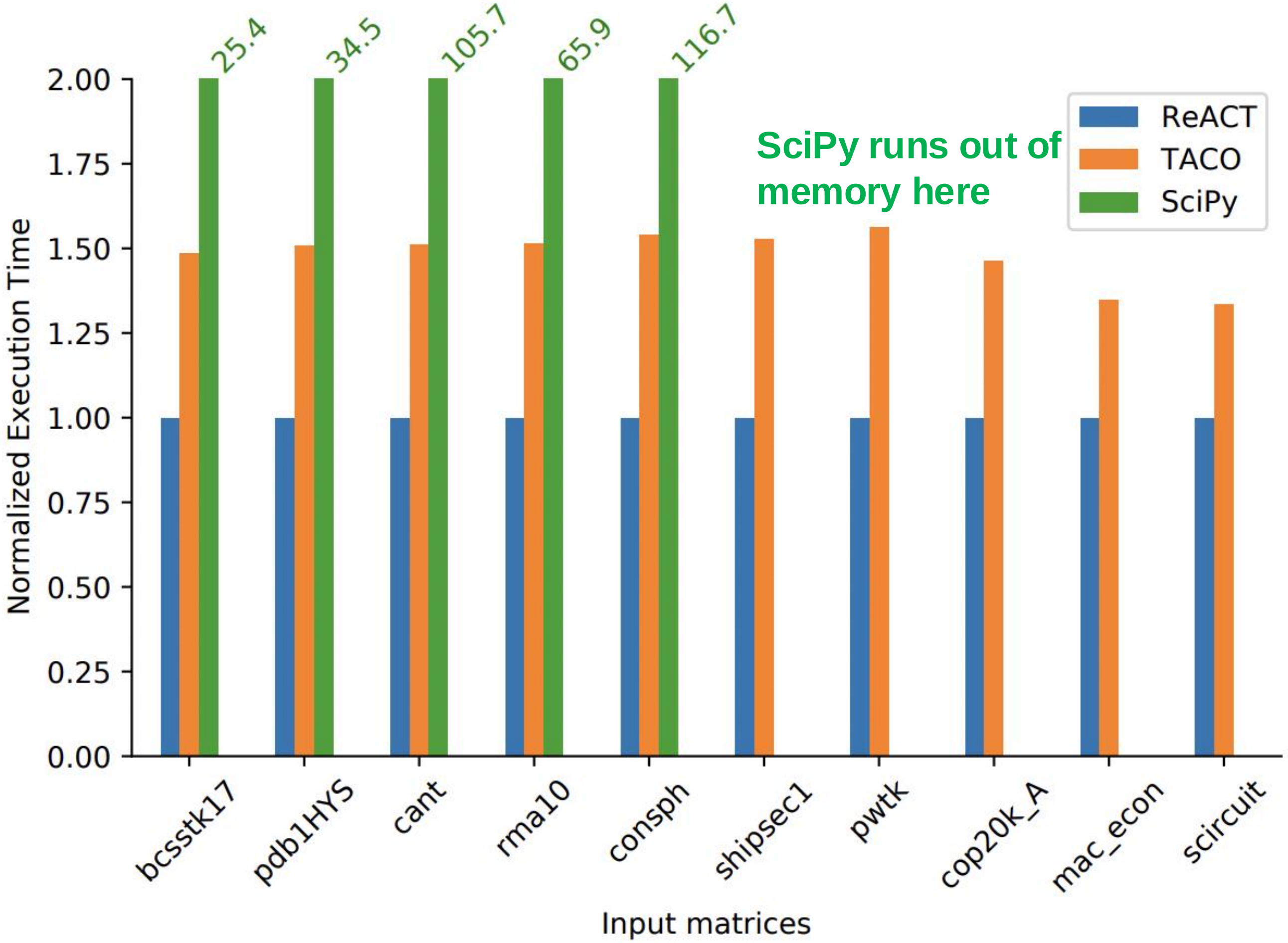
(b) GNN-kernel1 (NH=256, NJ=16)

Redundancy types present	SciPy	ReACT output
Type 1	No	No
Type 2	No	No
Type 3	No	No
Type 4	Yes	No

ReACT has better locality + more parallelism

Note: SciPy uses only a single thread for its SpMM implementation

SDDMM results – 1.5x faster than TACO

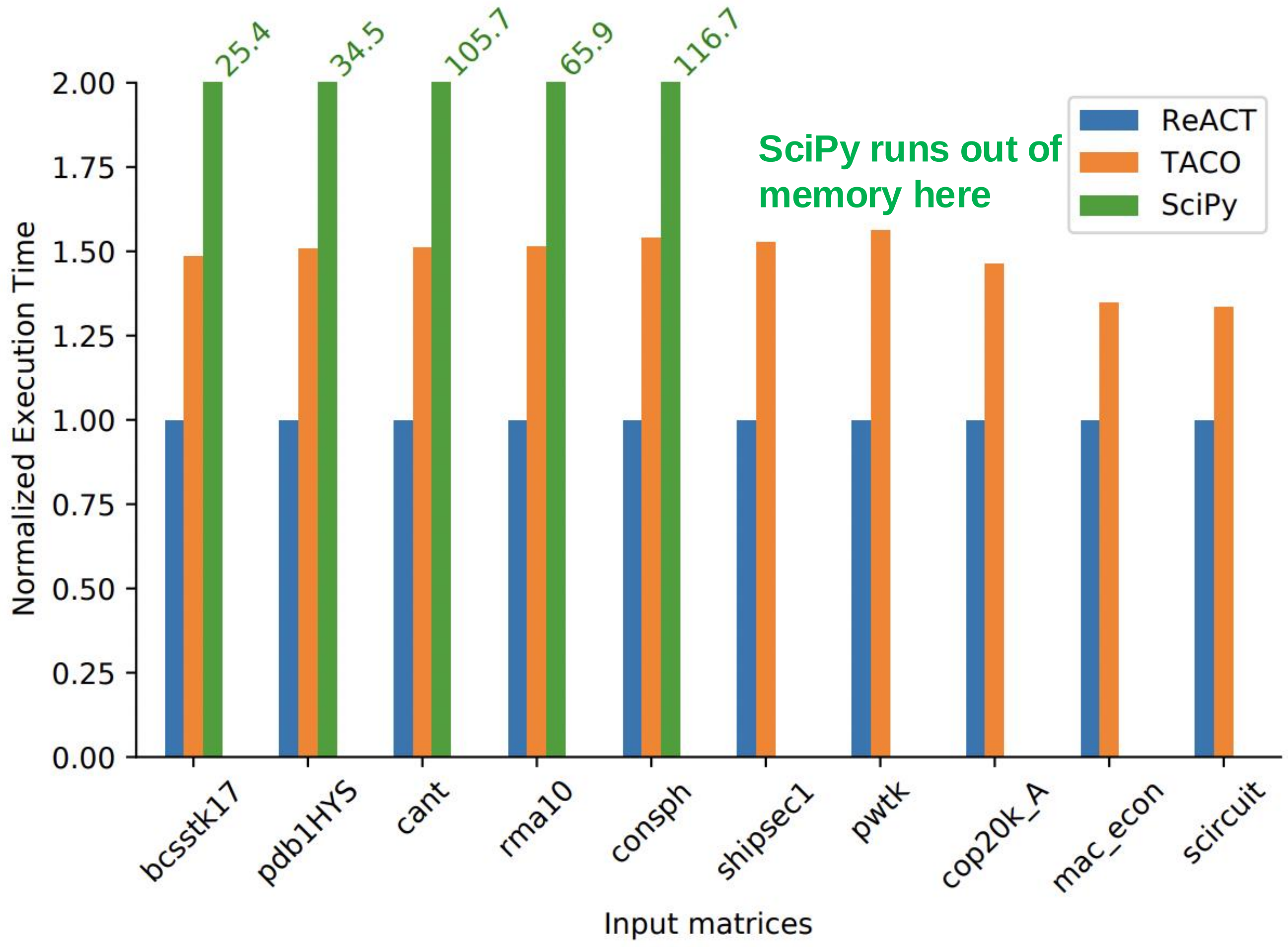


(a) SDDMM (NK=64)

Redundancy types present	TACO output	ReACT output
Type 1	Yes	No
Type 2	No	No
Type 3	No	No
Type 4	No	No

Both the amount of memory accesses and computations are reduced by eliminating type 1 redundancy.

SDDMM results – 57.3x faster than SciPy

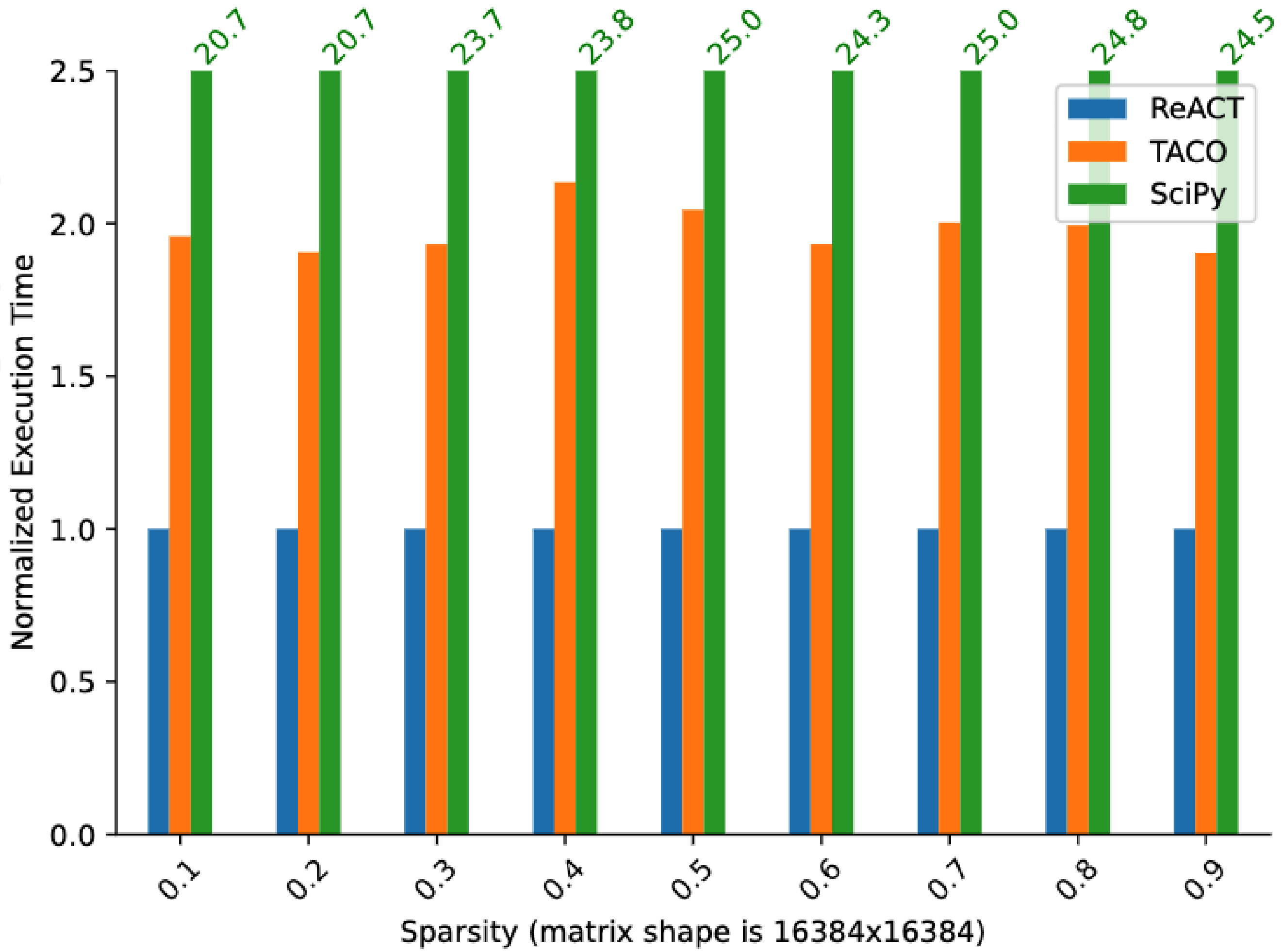


(a) SDDMM (NK=64)

Redundancy types present	SciPy	ReACT output
Type 1	No	No
Type 2	No	No
Type 3	Yes	No
Type 4	Yes	No

Many redundant computations are saved by eliminating type 4 (dead value) redundancies

Sparse-softmax results – 2.0x faster than TACO

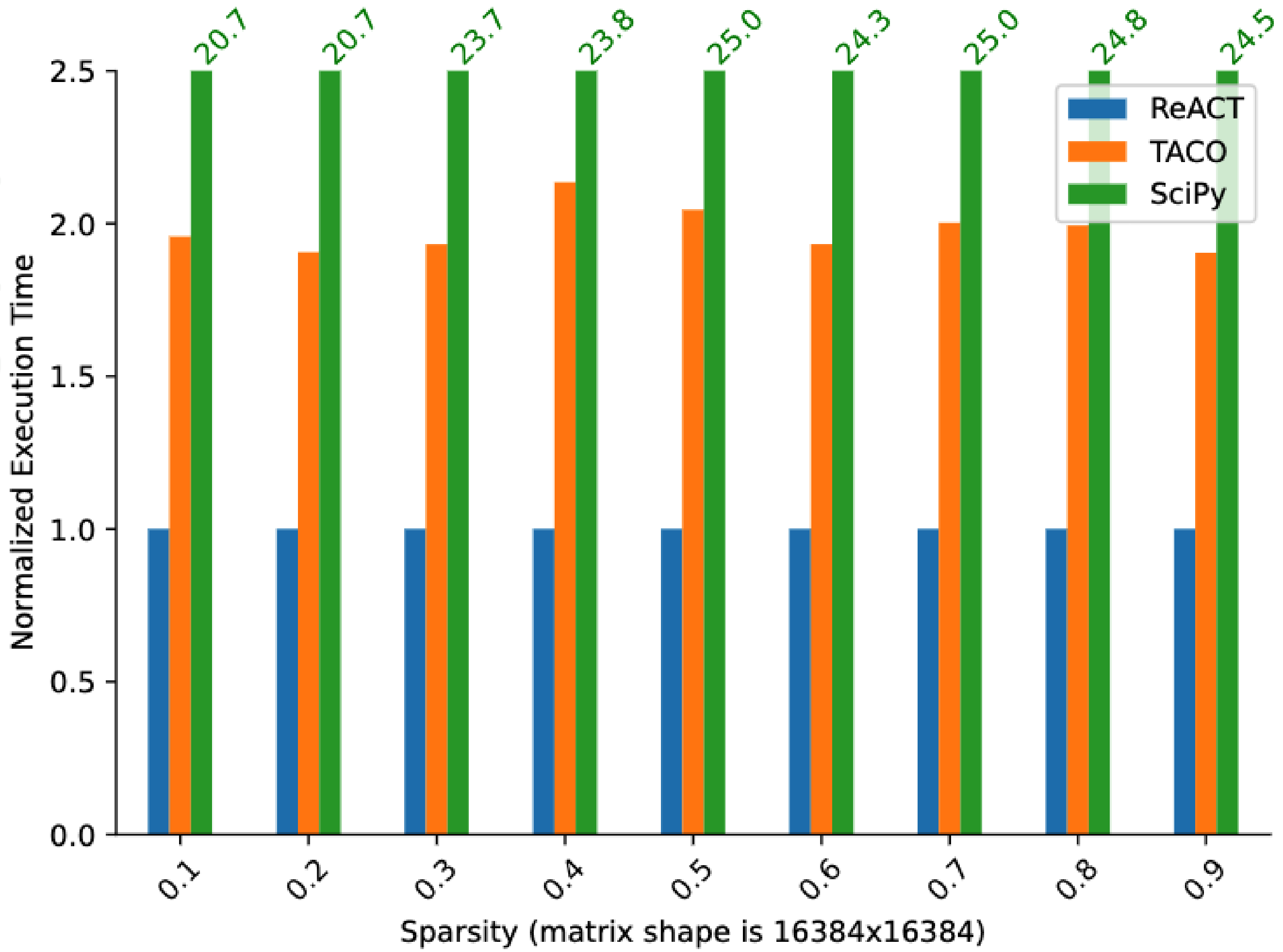


Redundancy types present	TACO output	ReACT output
Type 1	No	No
Type 2	No	No
Type 3	Yes	No
Type 4	No	No

Note: an AMD Ryzen 9 3900X was used for this experiment

TACO cannot fuse it into one single kernel while ReACT does, so ReACT has better locality

Sparse-softmax results – 23.5x faster than SciPy



Note: an AMD Ryzen 9 3900X was used for this experiment

Redundancy types present	TACO output	ReACT output
Type 1	No	No
Type 2	No	No
Type 3	Yes	No
Type 4	No	No

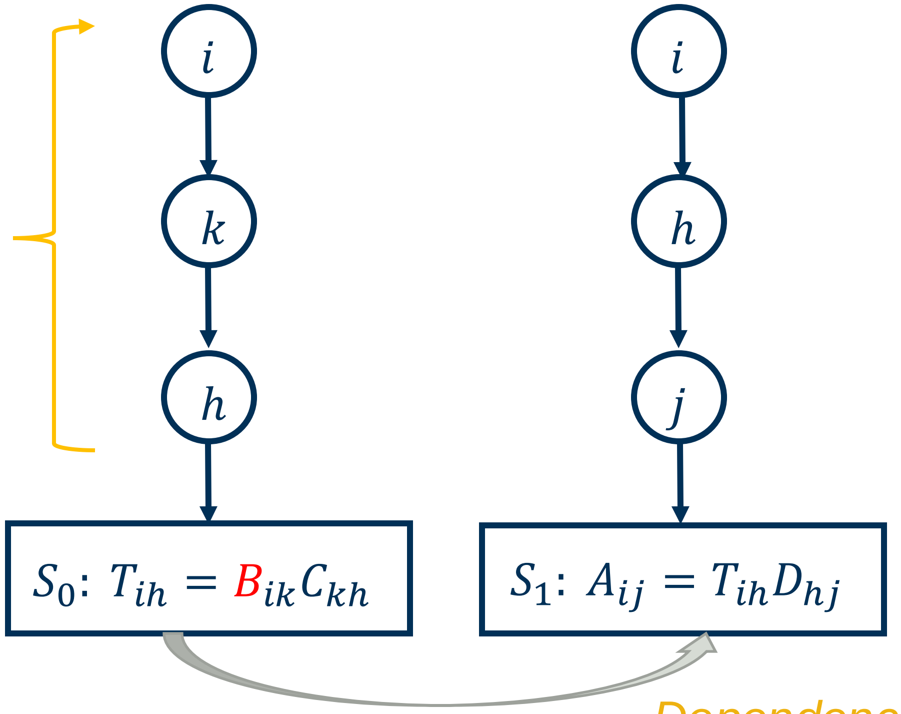
SciPy's sparse kernels are not parallelized
The operations are also not fused

Example: SpMM-MM

- Sparse-dense matmul followed by dense-dense matmul
  - Commonly used in graph neural networks
- Original input expression (sparse matrices are in red, assuming CSR format)
  - Python: $A = B \ @ \ C \ @ \ D$
- Transformations
  - Step 1: convert into *index notation* statements (each statement contains one operator)
    - S_0: $T_{ih} = B_{ik} \ @ \ C_{kh}$ (sparse-dense MM)
    - S_1: $A_{ij} = T_{ih} \ @ \ D_{hj}$ (dense-dense MM)
    - T_{ih} is compiler-generated temporary variable
  - Step 2: create an *index tree* from the index notation statements
    - Next slide

Index tree of SpMM-MM

- Two operations => create two subtrees
  - S_0: $T_{ih} = {\color{red}B_{ik}}\ @\ C_{kh}$
  - S_1: $A_{ij} = T_{ih}\ @\ D_{hj}$



Index node

Compute node

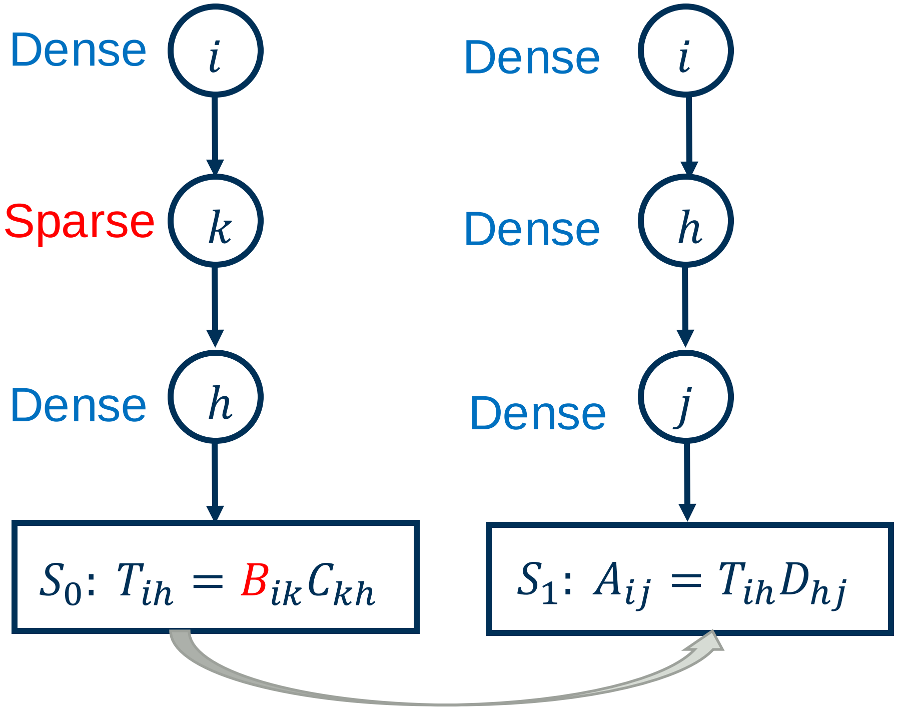
S_0: $T_{ih} = {\color{red}B_{ik}}C_{kh}$

S_1: $A_{ij} = T_{ih}D_{hj}$

Dependence edge

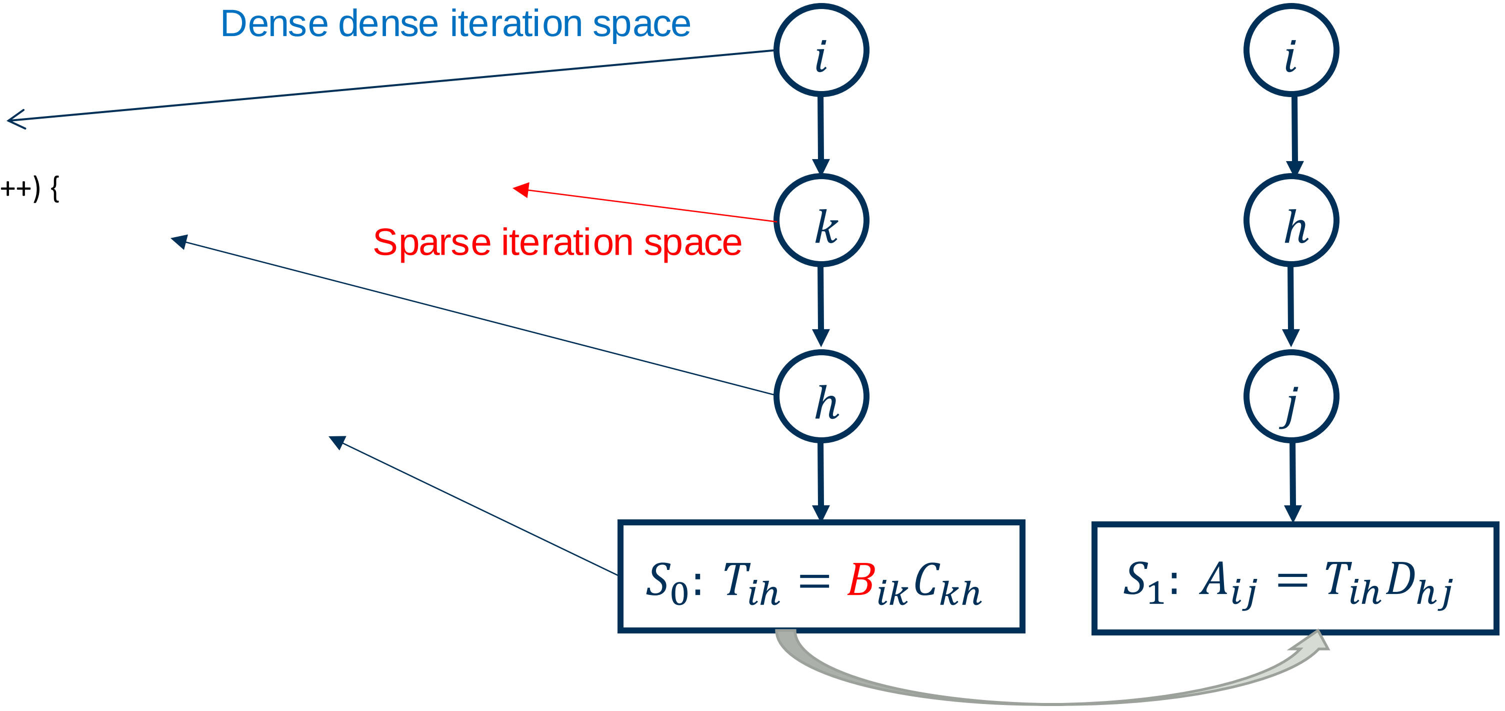
SpMM-MM index trees

- Annotate each index node as “Dense” or “Sparse”


Dense
i
Sparse
k
Dense
h
S_0: $T_{ih} = B_{ik}C_{kh}$
Dense
i
Dense
h
Dense
j
S_1: $A_{ij} = T_{ih}D_{hj}$

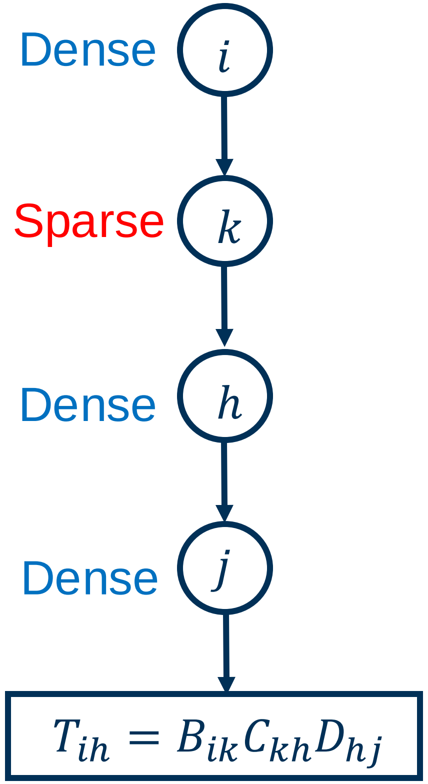
Index tree corresponding loop structure

```
for (int i = 0; i < NI; i++) {
  for (int k = B.rowptrs[i]; k < B.rowptrs[i+1]; k++) {
    for (int h = 0; h < NH; h++) {
      ...
      // T[i, h] += B[i, k] * C[k, h]
      T[i, h] += B.vals[k] * C[B.cols[k], h];
      ...
    }
  }
}
```


Dense dense iteration space
Sparse iteration space
i
k
h
$S_0\colon T_{ih} = B_{ik}C_{kh}$
i
h
j
$S_1\colon A_{ij} = T_{ih}D_{hj}$

SpMM-MM index trees: TACO (maximal fusion)

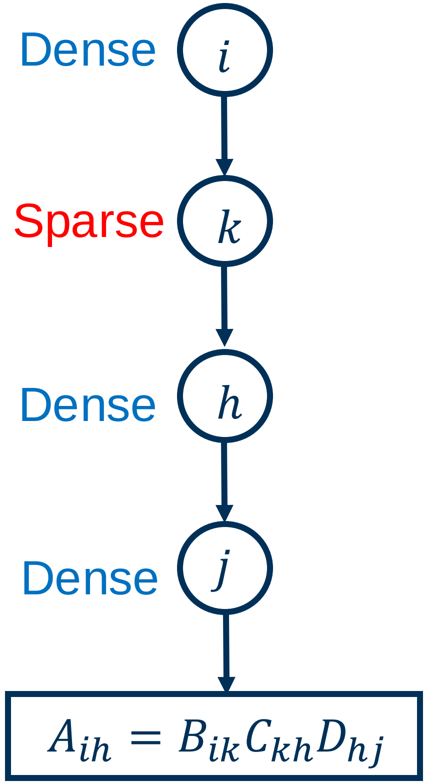
- Time: Bad, $O(NNZ_B * NH * NJ)$
  - Due to type 1 and 2 redundancies
- Intermediate space: Great, $O(1)$
- Locality: Great



SpMM-MM index trees: TACO (maximal fusion)

Generated code

```
for (int i = 0; i < NI; i++) {
  for (int k = B.rowptrs[i]; k < B.rowptrs[i+1]; k++) {
    for (int h = 0; h < NH; h++) {
      for (int j = 0; j < NJ; j++) {
        ...
        // A[i, h] += B[i, k] * C[k, h] * D[h, j]
        A[i, h] += B.vals[k] * C[B.cols[k], h] * D[h, j];
        ...
      }
    }
  }
}
```



Dense i

Sparse k

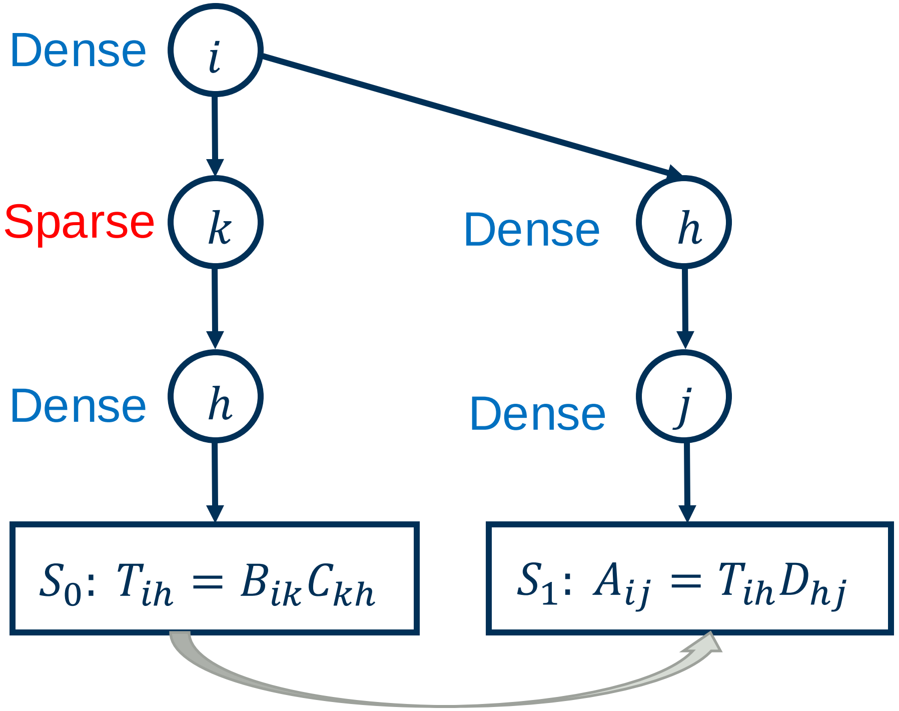
Dense h

Dense j

$A_{ih} = B_{ik}C_{kh}D_{hj}$

SpMM-MM index trees: ReACT (partial fusion)

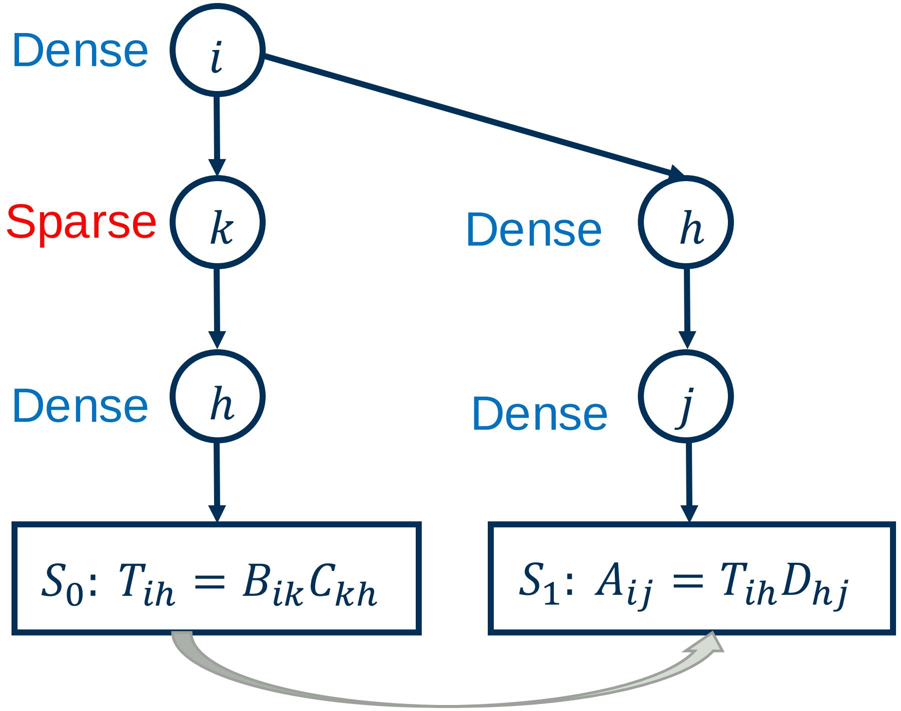
- Time: Good, $O(NNZ_B * NH + NI * NH * NJ)$
  - Typically much smaller than $O(NNZ_B * NH * NJ)$
- Intermediate space: Good, $O(NH)$
  - After memory optimization
- Locality: Good


Dense
i
Sparse
k
Dense
h
Dense
h
Dense
j
S_0: $T_{ih} = B_{ik}C_{kh}$
S_1: $A_{ij} = T_{ih}D_{hj}$

SpMM-MM index trees: ReACT (partial fusion)

Generated code

```
for (int i = 0; i < NI; i++) {
    for (int k = B.rowptrs[i]; k < B.rowptrs[i+1]; k++) {
        for (int h = 0; h < NH; h++) {
            ...
            // T[i, h] += B[i, k] * C[k, h]
            T[h] += B.vals[k] * C[B.cols[k], h];
            ...
        }
    }
    for (int h = 0; h < NH; h++) {
        for (int j = 0; j < NJ; j++) {
            ...
            // A[i, h] += T[i, h] * D[h, j]
            A[i, h] += T[h] * D[h, j];
            ...
        }
        T[h] = 0;
    }
}
```



Dense i

Sparse k

Dense h

Dense h

Dense j

$S_0\colon T_{ih} = B_{ik}C_{kh}$

$S_1\colon A_{ij} = T_{ih}D_{hj}$

ReACT summary

- We identify four common types of redundancies that can occur when generating code for a sequence of dense/sparse tensor operations
- We introduce ReACT, which consists of a set of redundancy-aware code generation techniques and can generate code with reduced redundancies
- Empirical evaluation on real-world applications such as SDDMM, GNN, Sparse-Softmax, and MTTKRP showed that our generated code with redundancy elimination resulted in 1.1× to orders-of-magnitude performance improvements relative to a state-of-the-art tensor algebra compiler (TACO) and library approaches such as scipy.sparse

Thesis contributions

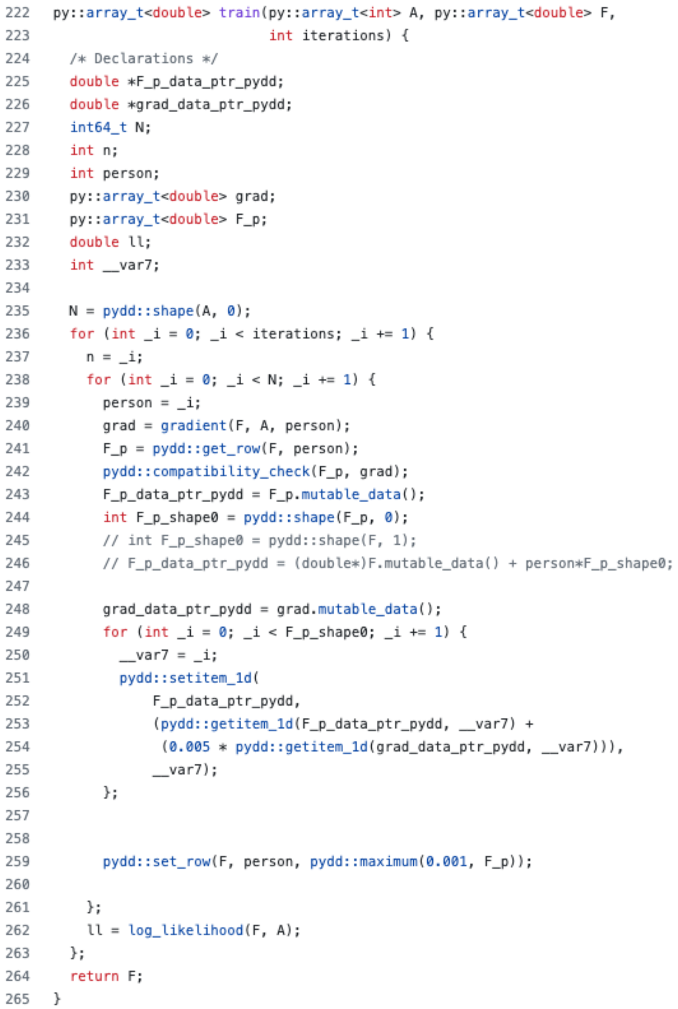
- APPy: Annotated Parallelism for Python on GPUs
  - [CC24] Parallelize Python loops and tensor expressions on GPUs
- ReACT: Redundancy-Aware Code Generation for Tensor Expressions
  - [PACT22] Redundancy elimination when fusing sparse/dense tensor operators
- Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels
  - [Onward!20] Compile Python/NumPy to C++ with high-level optimizations

Problem statement: desired input and output

- Desired input: whole kernel in Python (control flow is fine)

```
it = 0
while it < max_iter:
    u = 1.0 / x
    v = c * (1 / (K.T @ u))
    x = ((1 / r) * K) @ v
    it += 1
```

- Desired output: C++ code



```cpp
py::array_t<double> train(py::array_t<int> A, py::array_t<double> F,
                          int iterations) {
  /* Declarations */
  double *F_p_data_ptr_pydd;
  double *grad_data_ptr_pydd;
  int64_t N;
  int n;
  int person;
  py::array_t<double> grad;
  py::array_t<double> F_p;
  double ll;
  int __var7;

  N = pydd::shape(A, 0);
  for (int _i = 0; _i < iterations; _i += 1) {
    n = _i;
    for (int _i = 0; _i < N; _i += 1) {
      person = _i;
      grad = gradient(F, A, person);
      F_p = pydd::get_row(F, person);
      pydd::compatibility_check(F_p, grad);
      F_p_data_ptr_pydd = F_p.mutable_data();
      int F_p_shape0 = pydd::shape(F_p, 0);
      // int F_p_shape0 = pydd::shape(F, 1);
      // F_p_data_ptr_pydd = (double*)F.mutable_data() + person*F_p_shape0;

      grad_data_ptr_pydd = grad.mutable_data();
      for (int _i = 0; _i < F_p_shape0; _i += 1) {
        __var7 = _i;
        pydd::setitem_1d(
            F_p_data_ptr_pydd,
            (pydd::getitem_1d(F_p_data_ptr_pydd, __var7) +
             (0.005 * pydd::getitem_1d(grad_data_ptr_pydd, __var7))),
            __var7);
      };


      pydd::set_row(F, person, pydd::maximum(0.001, F_p));

    };
    ll = log_likelihood(F, A);
  };
  return F;
}
```

Compilation Pipeline: From Intrepydd to C++

Intrepydd source code

```
def foo(xs: Array(double, 2)) -> double:
        ...
  for i in range(shape(xs, 0)):
    for j in range(shape(xs, 1)):
      sum += xs[i, j]
                                      ...
```

Compilation Pipeline: From Intrepydd to C++

Intrepydd source code

```
def foo(xs: Array(double, 2)) -> double:
        ...
  for i in range(shape(xs, 0)):
    for j in range(shape(xs, 1)):
      sum += xs[i, j]
                              ...
```

Intrepydd compiler

Resulting C++ code

```
Array<double>* foo(Array<double>* xs) {
...
for (int i = 0; i < pydd::shape(xs, 0); i += 1) {
  for (int j = 0; j < pydd::shape(xs, 1); j += 1) {
    sum += xs.data()[i*pydd::shape(xs, 1)+j];
    ...
```

Code Optimization

- High-level Optimizations in AOT compilation
    - Loop invariant code motion (LICM OPT)
    - Dense & Sparse Array Operator Fusion (Array OPT)
    - Array allocation and slicing optimization (Memory OPT)

Code Optimization: LICM

c: sparse
K, u: dense

```
it = 0
while it < max_iter:
    u = 1.0 / x
    v = c * (1 / (K.T @ u)) # SDDMM
    x = ((1 / r) * K) @ v
    it += 1
```

Intrepydd source code (Sinkhorn)

```
it = 0
# Hoisted loop-invariant expressions
tmp1 = K.T
tmp2 = (1 / r) * K
while it < max_iter:
    u = 1.0 / x

    v = empty_like(c)
    # Fused loop iterating over non-zero elements
    for row, col, val in c.nonzero_elements():
        tmp3 = 0.0
        for idx in range(shape(tmp1, 1)):
            tmp3 += tmp1[row, idx] * u[idx, col]
        tmp4 = val * (1 / tmp3)
        spm_set_item(v, tmp4, row, col)

    x = spmm_dense(tmp2, v)

    it += 1
```

Transformed code

Code Optimization: Sparse Operator Fusion

c: sparse
K, u: dense

```
it = 0
while it < max_iter:
    u = 1.0 / x
    v = c * (1 / (K.T @ u)) # SDDMM
    x = ((1 / r) * K) @ v
    it += 1
```

SDDMM: masked matmul

Intrepydd source code (Sinkhorn)

```
it = 0
# Hoisted loop-invariant expressions
tmp1 = K.T
tmp2 = (1 / r) * K
while it < max_iter:
    u = 1.0 / x
    v = empty_like(c)
    # Fused loop iterating over non-zero elements
    for row, col, val in c.nonzero_elements():
        tmp3 = 0.0
        for idx in range(shape(tmp1, 1)):
            tmp3 += tmp1[row, idx] * u[idx, col]
        tmp4 = val * (1 / tmp3)
        spm_set_item(v, tmp4, row, col)

    x = spmm_dense(tmp2, v)

    it += 1
```

Transformed code

Code Optimization: Dense Operator Fusion

c: sparse
K, u: dense

```
it = 0
while it < max_iter:
    u = 1.0 / x
    v = c * (1 / (K.T @ u)) # SDDMM
    x = ((1 / r) * K) @ v
    it += 1
```

SDDMM: masked matmul

Intrepydd source code (Sinkhorn)

```
it = 0
# Hoisted loop-invariant expressions
tmp1 = K.T
tmp2 = (1 / r) * K
while it < max_iter:
    u = 1.0 / x
    v = empty_like(c)
    # Fused loop iterating over non-zero elements
    for row, col, val in c.nonzero_elements():
        tmp3 = 0.0
        for idx in range(shape(tmp1, 1)):
            tmp3 += tmp1[row, idx] * u[idx, col]
        tmp4 = val * (1 / tmp3)
        spm_set_item(v, tmp4, row, col)

    x = spmm_dense(tmp2, v)

    it += 1
```

Transformed code

Experimental Methodology

Benchmark Applications

- A subset of Python based data analytics applications from a recent DARPA program
- Mix of non-library call and library call dominated applications

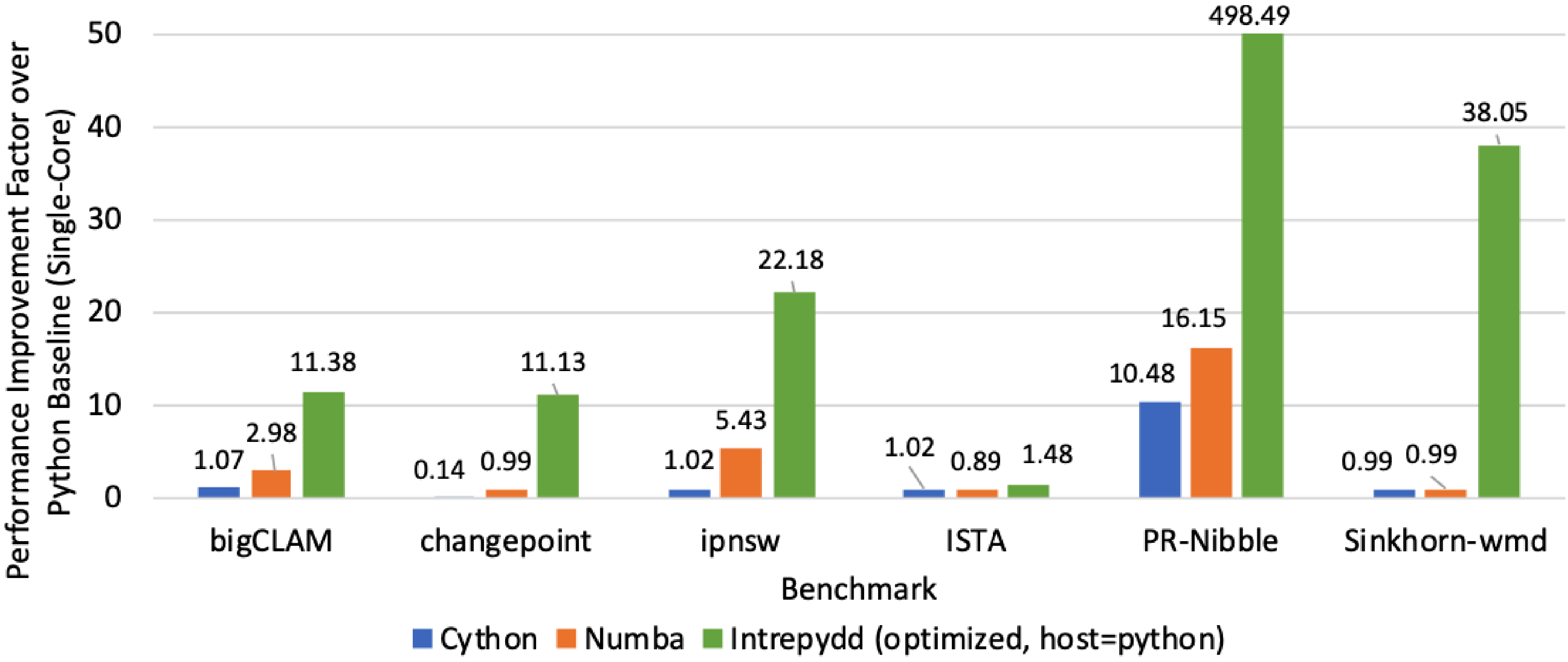
Test machine

- Dual Intel Xeon Silver 4114 CPU @ 2.2GHz with 192GB of main memory and hyperthreading disabled

Comparisons

- Baseline idiomatic Python 3.7.6
- Cython
- Numba

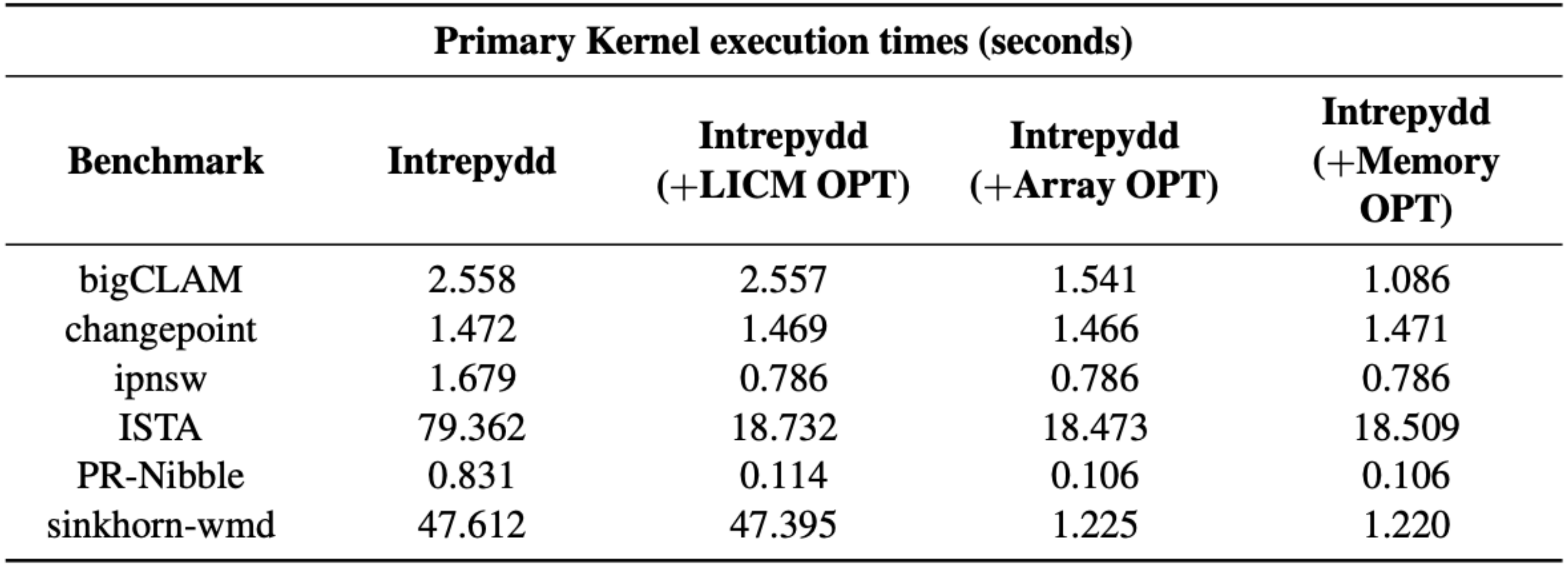
Intrepydd Sequential Performance



Intrepydd offers 20.7x speedup on average (geomean) over baseline Python

Code Optimization

- High-level Optimizations in AOT compilation
  - Loop invariant code motion (LICM OPT)
  - Dense & Sparse Array Operator Fusion (Array OPT)
  - Array allocation and slicing optimization (Memory OPT)
- Impact on performance by each OPT



Primary Kernel execution times (seconds)

Benchmark	Intrepydd	Intrepydd (+LICM OPT)	Intrepydd (+Array OPT)	Intrepydd (+Memory OPT)
bigCLAM	2.558	2.557	1.541	1.086
changepoint	1.472	1.469	1.466	1.471
ipnsw	1.679	0.786	0.786	0.786
ISTA	79.362	18.732	18.473	18.509
PR-Nibble	0.831	0.114	0.106	0.106
sinkhorn-wmd	47.612	47.395	1.225	1.220

Intrepydd summary

- We present Intrepydd, a Python-based programming system, which is designed to enable data scientists to write application kernels with high performance, productivity, and portability
- We implement a number of high-level compiler optimizations during the compilation
- We evaluate the performance of Intrepydd using 6 data science kernels and show significant single-core performance improvements of Intrepydd relative to vanilla Python/NumPy (1.5× to 498.5×), Cython (1.5× to 47.5×) and Numba (1.7× to 38.1×)

Thank you!

- APPy: Annotated Parallelism for Python on GPUs
    - [CC24] Parallelize Python loops and tensor expressions on GPUs
- ReACT: Redundancy-Aware Code Generation for Tensor Expressions
    - [PACT22] Redundancy elimination when fusing sparse/dense tensor operators
- Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels
    - [Onward!20] Compile Python/NumPy to C++ with high-level optimizations

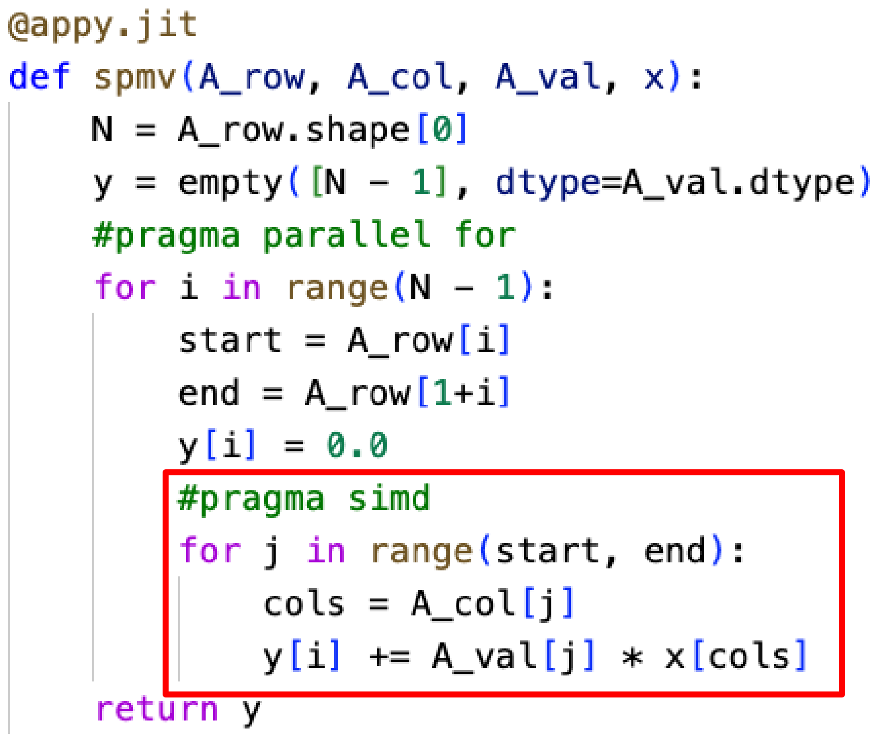
	APPy	ReACT	Intrepydd
Input	Python programs	Tensor DSL	Python programs
Output	Triton code for GPUs	C++ code for CPUs	C++ code for CPUs
Compilation	JIT	AOT	AOT
Requires type annotation	No	Yes	Yes
Requires compiler directives	Yes	No	Only for pfor
Parallel reduction	Yes via pragma	No	No
Operator fusion	Yes	Yes	Yes
LICM	No	Yes	Yes
Sparse redundancy elimination	No	Yes	Yes
General sparse codegen	No	Yes	No
Small tensor caching	Yes via pragma	No	No

APPy Backup

More complicated examples

- Sparse matrix dense vector multiplication

10830x speedup over CuPy (loop-based) [1]

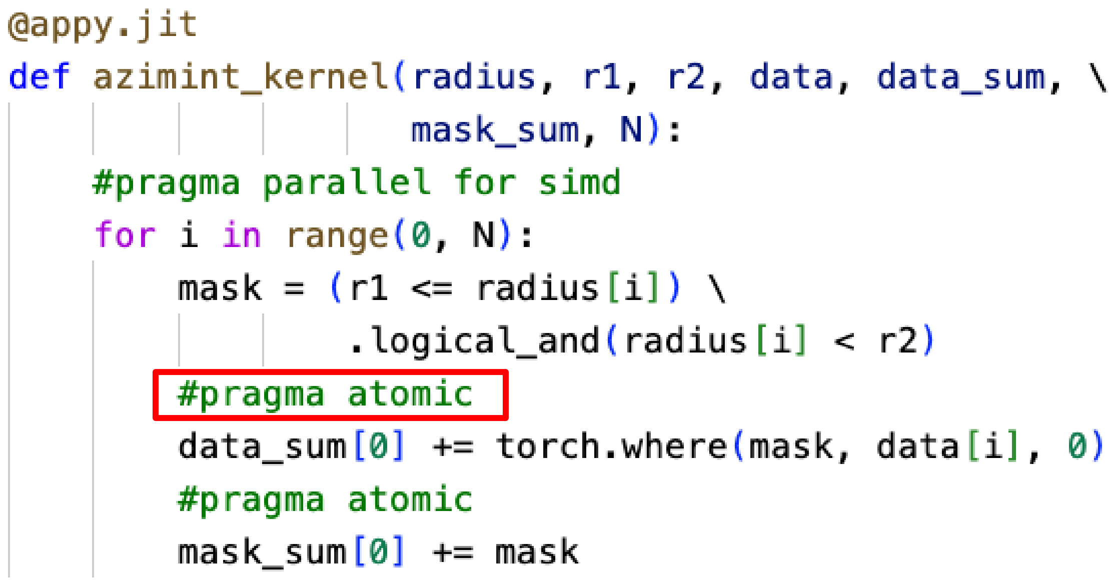


```
@appy.jit
def spmv(A_row, A_col, A_val, x):
    N = A_row.shape[0]
    y = empty([N - 1], dtype=A_val.dtype)
    #pragma parallel for
    for i in range(N - 1):
        start = A_row[i]
        end = A_row[1+i]
        y[i] = 0.0
        #pragma simd
        for j in range(start, end):
            cols = A_col[j]
            y[i] += A_val[j] * x[cols]
    return y
```

Dynamic loop bounds are fine with #pragma simd

- Azimuthal integration, related to X-ray images

1.8x speedup over CuPy (operator only) [1]

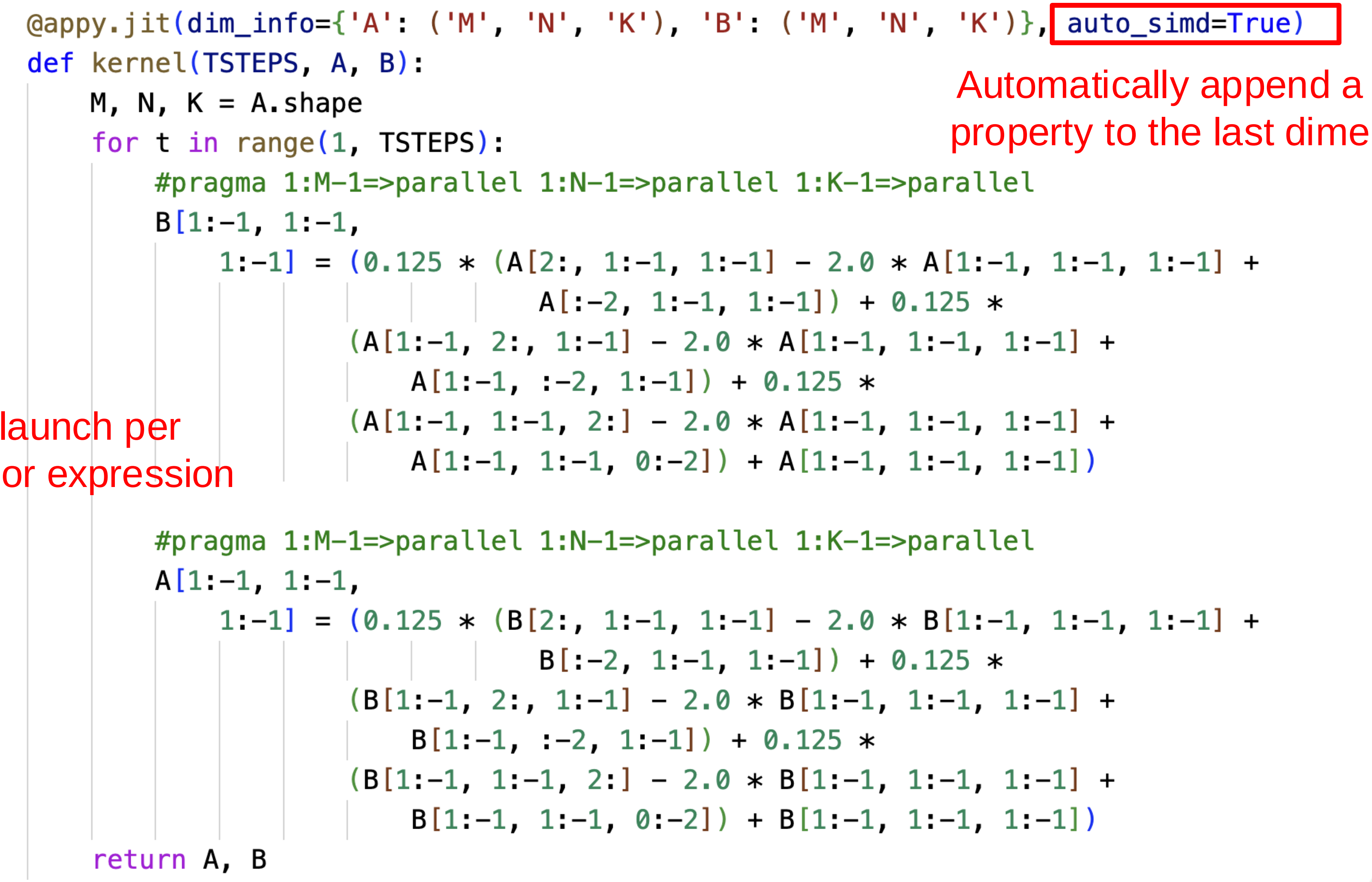


```
@appy.jit
def azimint_kernel(radius, r1, r2, data, data_sum, \
                   mask_sum, N):
    #pragma parallel for simd
    for i in range(0, N):
        mask = (r1 <= radius[i]) \
            .logical_and(radius[i] < r2)
        #pragma atomic
        data_sum[0] += torch.where(mask, data[i], 0)
        #pragma atomic
        mask_sum[0] += mask
```

Parallel reduction via atomic update

1. Testing machine is a RTX 3090 GPU and the baseline NumPy runtime is ~0.3 seconds

A stencil kernel “heat_3d” using tensor expressions



```
@appy.jit(dim_info={'A': ('M', 'N', 'K'), 'B': ('M', 'N', 'K')}, auto_simd=True)
def kernel(TSTEPS, A, B):
    M, N, K = A.shape
    for t in range(1, TSTEPS):
        #pragma 1:M-1=>parallel 1:N-1=>parallel 1:K-1=>parallel
        B[1:-1, 1:-1,
            1:-1] = (0.125 * (A[2:, 1:-1, 1:-1] - 2.0 * A[1:-1, 1:-1, 1:-1] +
                                A[:-2, 1:-1, 1:-1]) + 0.125 *
                    (A[1:-1, 2:, 1:-1] - 2.0 * A[1:-1, 1:-1, 1:-1] +
                        A[1:-1, :-2, 1:-1]) + 0.125 *
                    (A[1:-1, 1:-1, 2:] - 2.0 * A[1:-1, 1:-1, 1:-1] +
                        A[1:-1, 1:-1, 0:-2]) + A[1:-1, 1:-1, 1:-1])

        #pragma 1:M-1=>parallel 1:N-1=>parallel 1:K-1=>parallel
        A[1:-1, 1:-1,
            1:-1] = (0.125 * (B[2:, 1:-1, 1:-1] - 2.0 * B[1:-1, 1:-1, 1:-1] +
                                B[:-2, 1:-1, 1:-1]) + 0.125 *
                    (B[1:-1, 2:, 1:-1] - 2.0 * B[1:-1, 1:-1, 1:-1] +
                        B[1:-1, :-2, 1:-1]) + 0.125 *
                    (B[1:-1, 1:-1, 2:] - 2.0 * B[1:-1, 1:-1, 1:-1] +
                        B[1:-1, 1:-1, 0:-2]) + B[1:-1, 1:-1, 1:-1])
    return A, B
```

Automatically append a simd property to the last dimension

One kernel launch per annotated tensor expression

Utilize both layers of parallelism: parallel for + simd

```
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for simd
    for i in range(N):
        c[i] = a[i] + b[i]
```

```
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for
    for i in range(N):
        c[i] = a[i] + b[i]
```

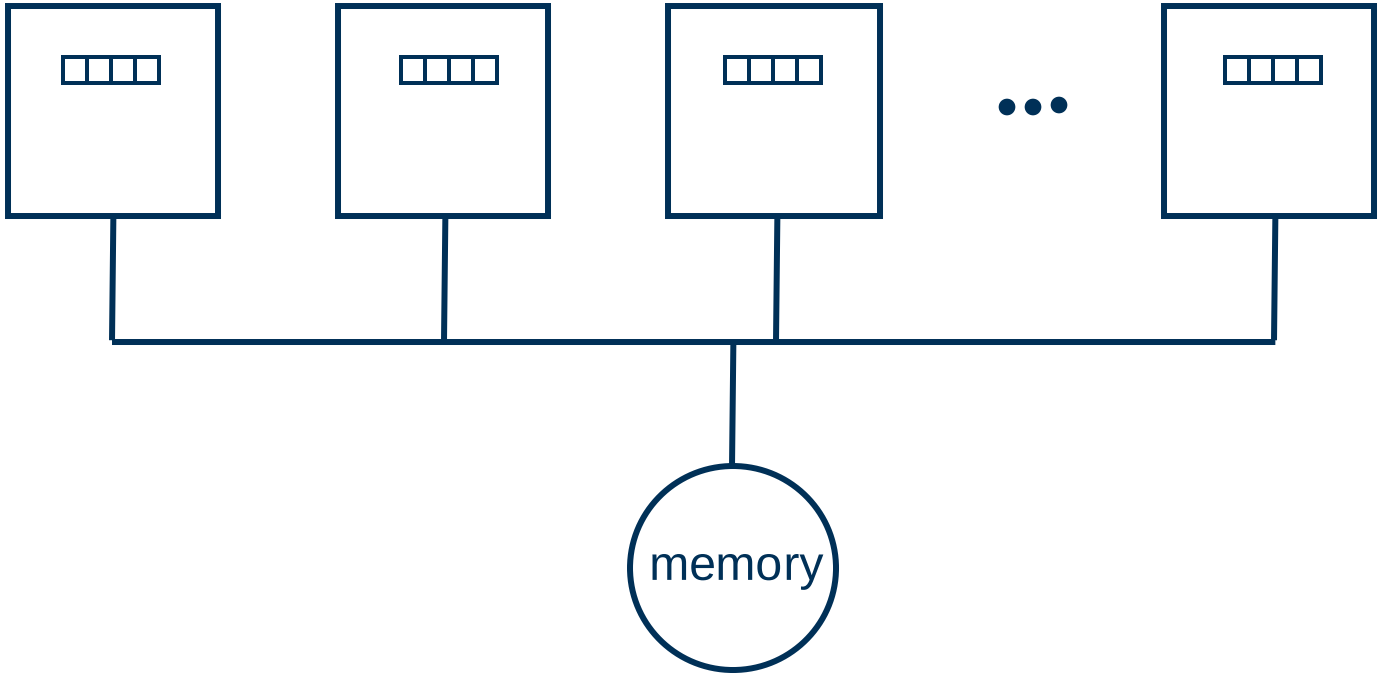
Performance boost!

Sliced index notation (inspired by Einstein notation)

- Two steps
  - Define index variables (dimension size)
  - Create sliced index notations
- Examples (assume “M, N = A.shape”)
  - Element-wise multiplication of A and B
    - C[:M, :N] = A[:M, :N] + B[:M, :N]
  - Row-wise summation of A
    - B[:M] = sum(A[:M, :N], axis=1)
  - Stencil pattern
    - B[1:M-1, 1:N-1] = 0.2 * (A[1:M-1, 1:N-1] + A[1:M-1, :N-2] + A[1:M-1, 2:N] + …)
  - Broadcast
    - A[:M, :N] = B[:M, None] + C[None, :N]

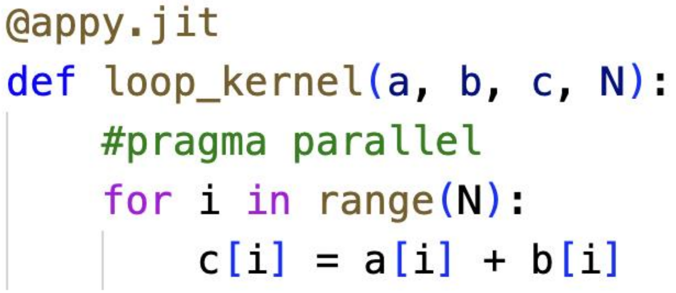
- Annotate each distinct dimension (slice) with a list of properties
  - :M=>parallel :N=>reduction(sum)
    - Indicate to the :M dimension should be processed in parallel and :N is a reduction dimension
  - 1:M-1=> parallel 1:N-1=>parallel
    - Indicate both dimensions should be processed in parallel

Abstract machine model: a multi-vector processor



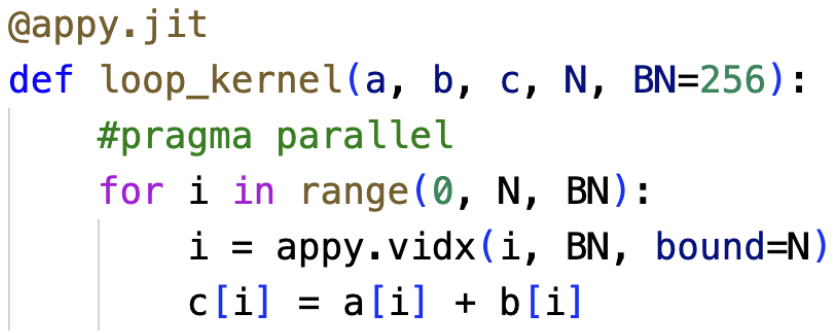
Loop-Oriented model

Higher performance can be achieved by working with a block of data per iteration



```
@appy.jit
def loop_kernel(a, b, c, N):
    #pragma parallel
    for i in range(N):
        c[i] = a[i] + b[i]
```

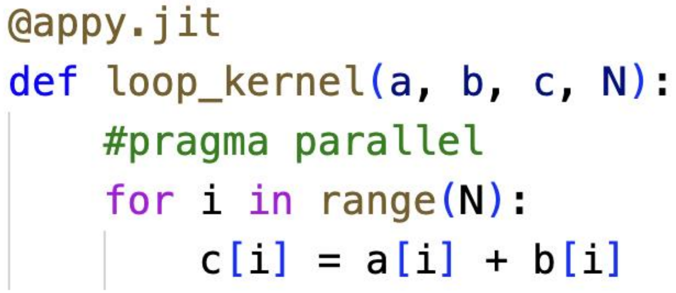
Performance boost!



```
@appy.jit
def loop_kernel(a, b, c, N, BN=256):
    #pragma parallel
    for i in range(0, N, BN):
        i = appy.vidx(i, BN, bound=N)
        c[i] = a[i] + b[i]
```

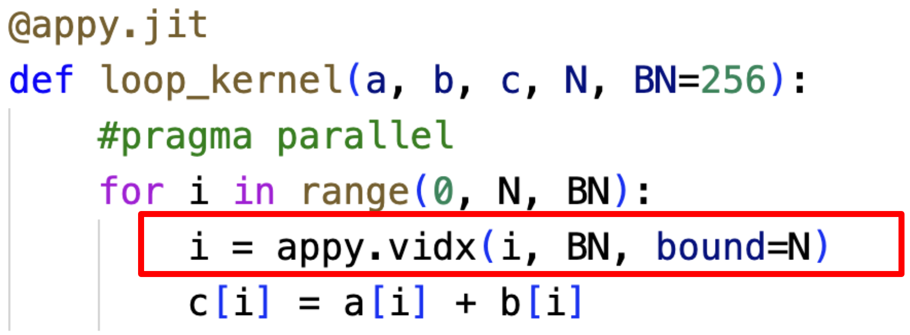
Loop-Oriented model

Higher performance can be achieved by working with a block of data per iteration



```
@appy.jit
def loop_kernel(a, b, c, N):
    #pragma parallel
    for i in range(N):
        c[i] = a[i] + b[i]
```

Performance boost!



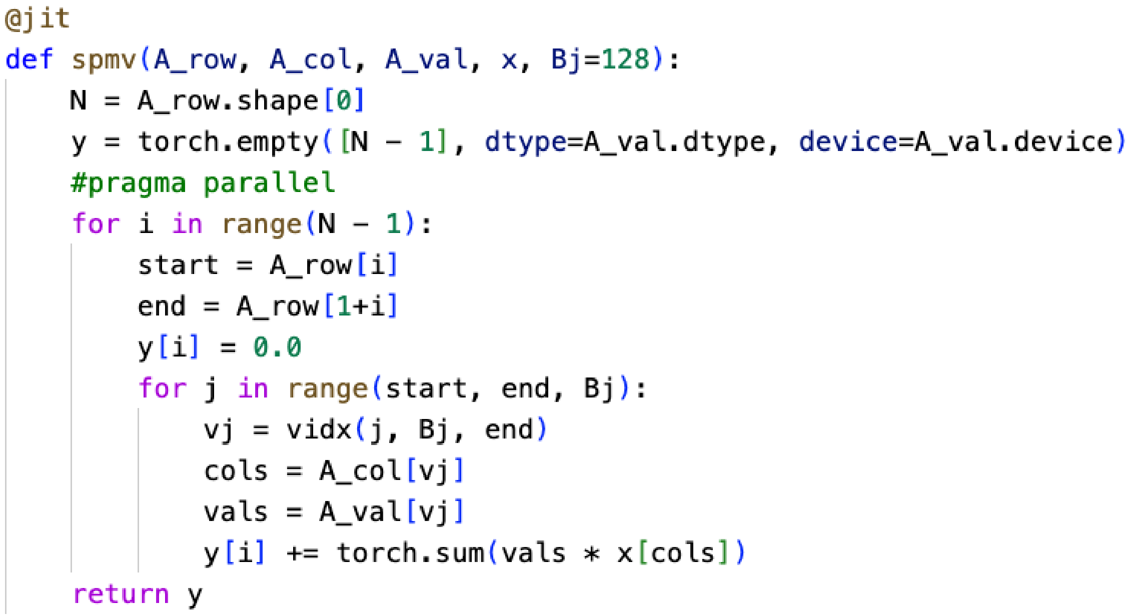
```
@appy.jit
def loop_kernel(a, b, c, N, BN=256):
    #pragma parallel
    for i in range(0, N, BN):
        i = appy.vidx(i, BN, bound=N)
        c[i] = a[i] + b[i]
```

A built-in function that returns a “vector of indices”, e.g. [i, i+1, i+2, …, i+BN-1]

More complicated examples

- Sparse matrix dense vector multiplication

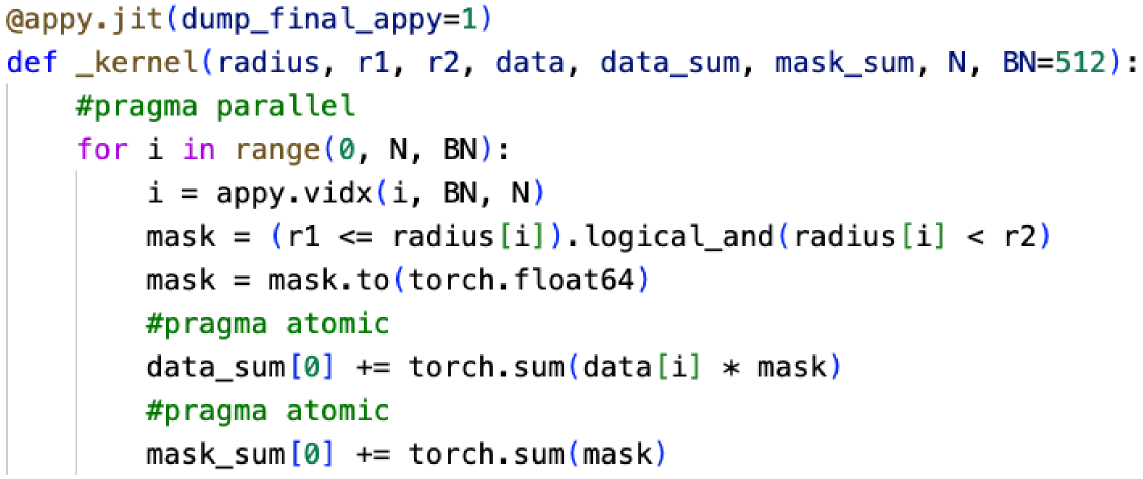
spmv



```
@jit
def spmv(A_row, A_col, A_val, x, Bj=128):
    N = A_row.shape[0]
    y = torch.empty([N - 1], dtype=A_val.dtype, device=A_val.device)
    #pragma parallel
    for i in range(N - 1):
        start = A_row[i]
        end = A_row[1+i]
        y[i] = 0.0
        for j in range(start, end, Bj):
            vj = vidx(j, Bj, end)
            cols = A_col[vj]
            vals = A_val[vj]
            y[i] += torch.sum(vals * x[cols])
    return y
```

- Azimuthal integration, related to X-ray images

azimint_naive

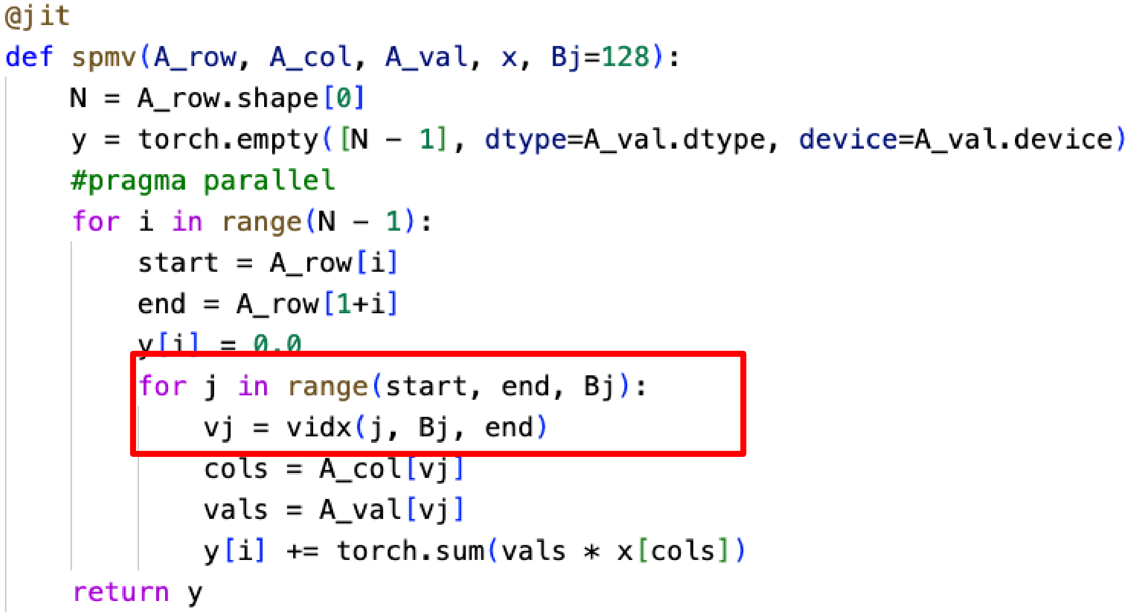


```
@appy.jit(dump_final_appy=1)
def _kernel(radius, r1, r2, data, data_sum, mask_sum, N, BN=512):
    #pragma parallel
    for i in range(0, N, BN):
        i = appy.vidx(i, BN, N)
        mask = (r1 <= radius[i]).logical_and(radius[i] < r2)
        mask = mask.to(torch.float64)
        #pragma atomic
        data_sum[0] += torch.sum(data[i] * mask)
        #pragma atomic
        mask_sum[0] += torch.sum(mask)
```

More complicated examples

- Sparse matrix dense vector multiplication
- Azimuthal integration, related to X-ray images

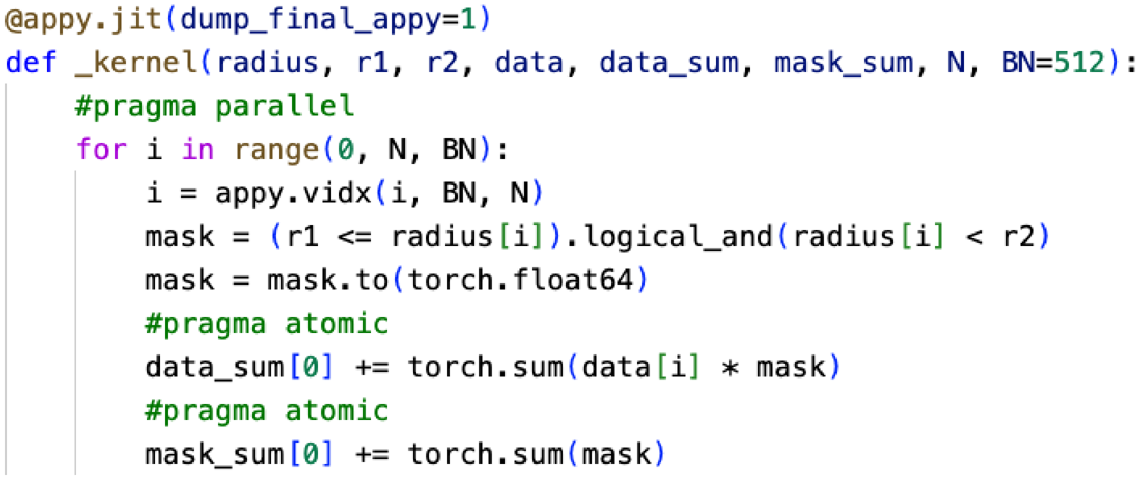
spmv



```
@jit
def spmv(A_row, A_col, A_val, x, Bj=128):
    N = A_row.shape[0]
    y = torch.empty([N - 1], dtype=A_val.dtype, device=A_val.device)
    #pragma parallel
    for i in range(N - 1):
        start = A_row[i]
        end = A_row[1+i]
        y[i] = 0.0
        for j in range(start, end, Bj):
            vj = vidx(j, Bj, end)
            cols = A_col[vj]
            vals = A_val[vj]
            y[i] += torch.sum(vals * x[cols])
    return y
```

Block size (Bj) must be a constant

azimint_naive

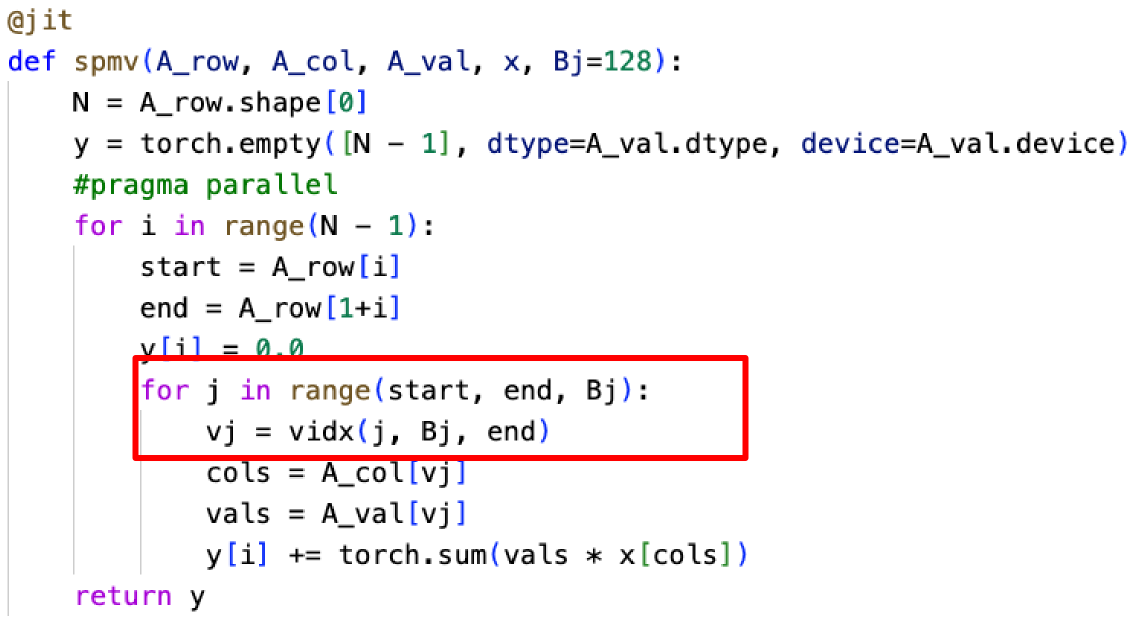


```
@appy.jit(dump_final_appy=1)
def _kernel(radius, r1, r2, data, data_sum, mask_sum, N, BN=512):
    #pragma parallel
    for i in range(0, N, BN):
        i = appy.vidx(i, BN, N)
        mask = (r1 <= radius[i]).logical_and(radius[i] < r2)
        mask = mask.to(torch.float64)
        #pragma atomic
        data_sum[0] += torch.sum(data[i] * mask)
        #pragma atomic
        mask_sum[0] += torch.sum(mask)
```

More complicated examples

- Sparse matrix dense vector multiplication
- Azimuthal integration, related to X-ray images

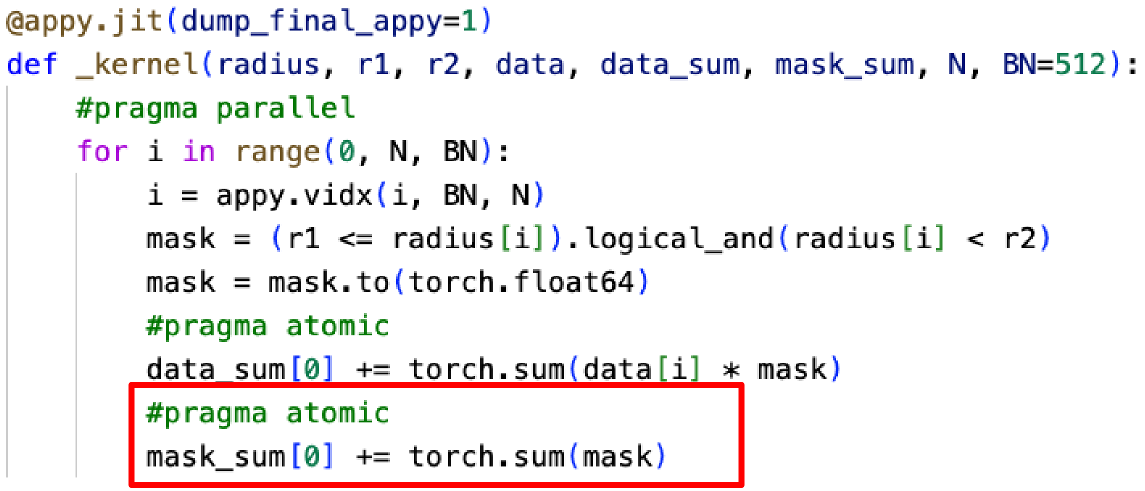
spmv



```
@jit
def spmv(A_row, A_col, A_val, x, Bj=128):
    N = A_row.shape[0]
    y = torch.empty([N - 1], dtype=A_val.dtype, device=A_val.device)
    #pragma parallel
    for i in range(N - 1):
        start = A_row[i]
        end = A_row[1+i]
        y[i] = 0.0
        for j in range(start, end, Bj):
            vj = vidx(j, Bj, end)
            cols = A_col[vj]
            vals = A_val[vj]
            y[i] += torch.sum(vals * x[cols])
    return y
```

Block size (Bj) must be a constant

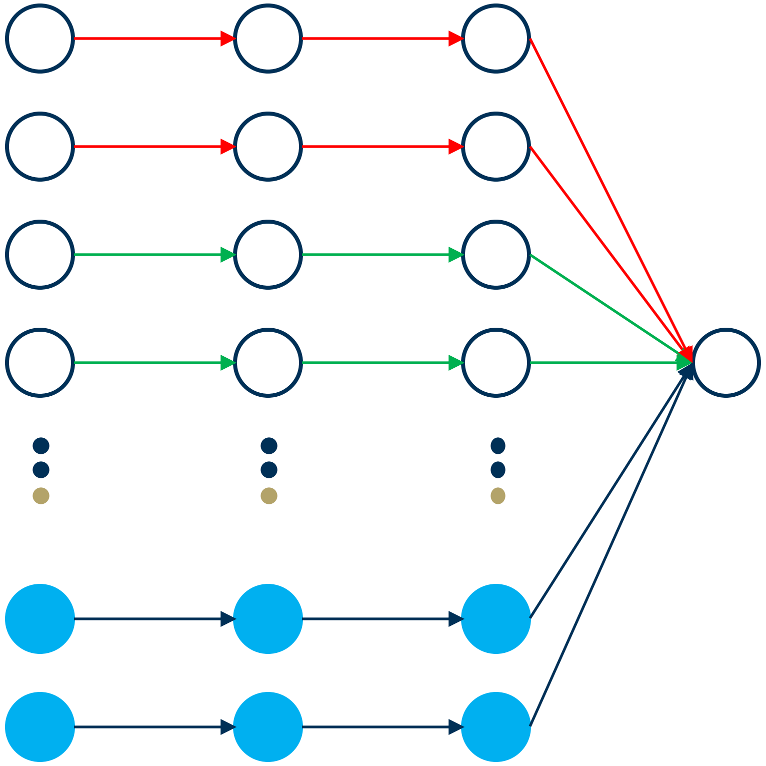
azimint_naive



```
@appy.jit(dump_final_appy=1)
def _kernel(radius, r1, r2, data, data_sum, mask_sum, N, BN=512):
    #pragma parallel
    for i in range(0, N, BN):
        i = appy.vidx(i, BN, N)
        mask = (r1 <= radius[i]).logical_and(radius[i] < r2)
        mask = mask.to(torch.float64)
        #pragma atomic
        data_sum[0] += torch.sum(data[i] * mask)
        #pragma atomic
        mask_sum[0] += torch.sum(mask)
```

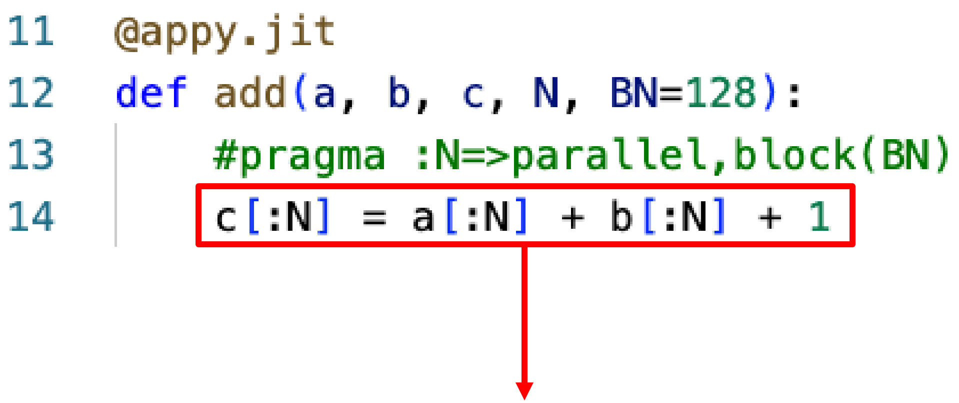
Indicates parallel reduction

Tensor expressions are inherently parallel



Tensor Oriented Programming Model

- Advantages
    - More concise
    - More automatic optimizations
        - Automatic loop fusion

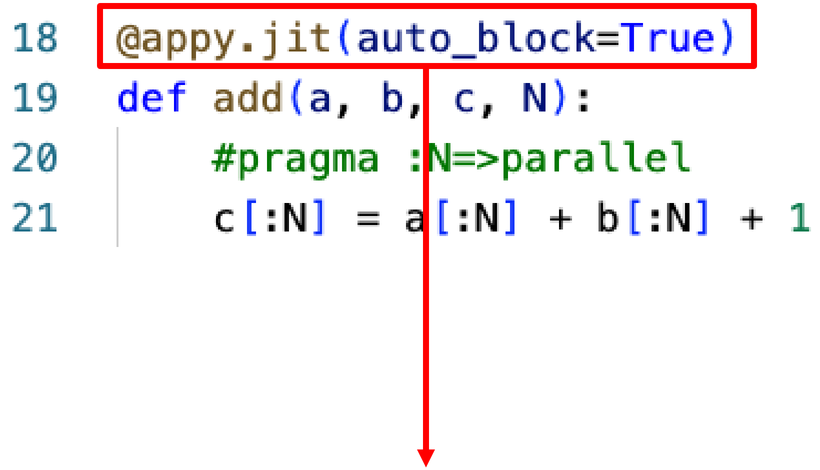


```python
@appy.jit
def add(a, b, c, N, BN=128):
    #pragma :N=>parallel,block(BN)
    c[:N] = a[:N] + b[:N] + 1
```

Two additions will be fused.

Tensor Oriented Programming Model

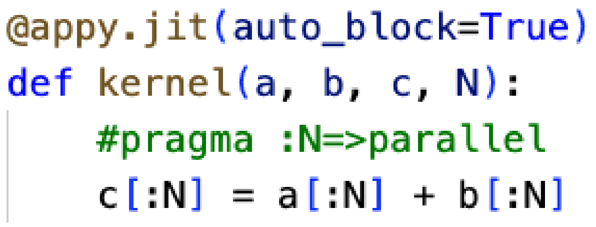
- Advantages
    - More concise
    - More automatic optimizations
        - Automatic loop fusion
        - Automatic loop tiling (a simple form)



```
@appy.jit(auto_block=True)
def add(a, b, c, N):
    #pragma :N=>parallel
    c[:N] = a[:N] + b[:N] + 1
```

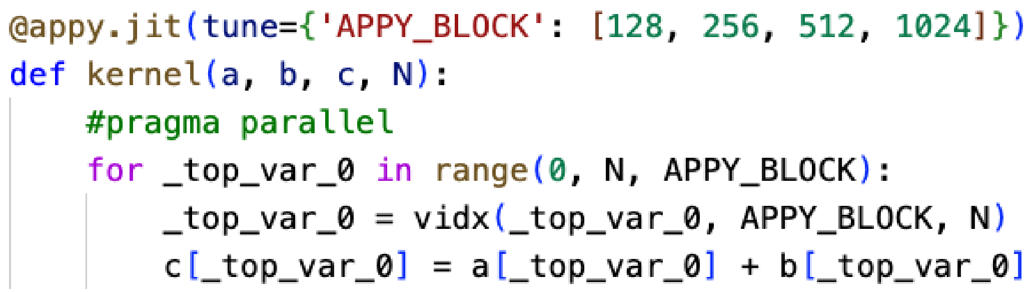
:N will be automatically blocked, and the optimal block size is auto-tuned from a set of common sizes

Example workflow for vector addition



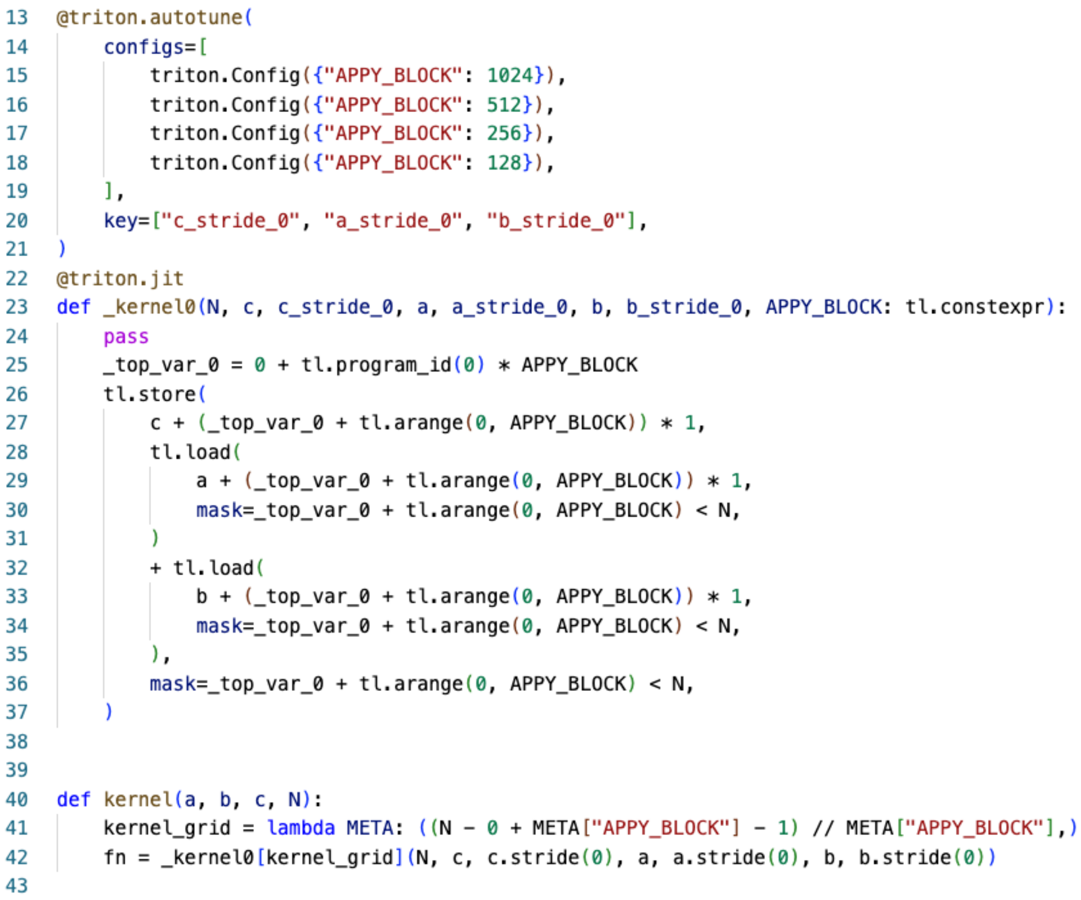
```
@appy.jit(auto_block=True)
def kernel(a, b, c, N):
    #pragma :N=>parallel
    c[:N] = a[:N] + b[:N]
```

High level transformations



```
@appy.jit(tune={'APPY_BLOCK': [128, 256, 512, 1024]})
def kernel(a, b, c, N):
    #pragma parallel
    for _top_var_0 in range(0, N, APPY_BLOCK):
        _top_var_0 = vidx(_top_var_0, APPY_BLOCK, N)
        c[_top_var_0] = a[_top_var_0] + b[_top_var_0]
```

Final code generation

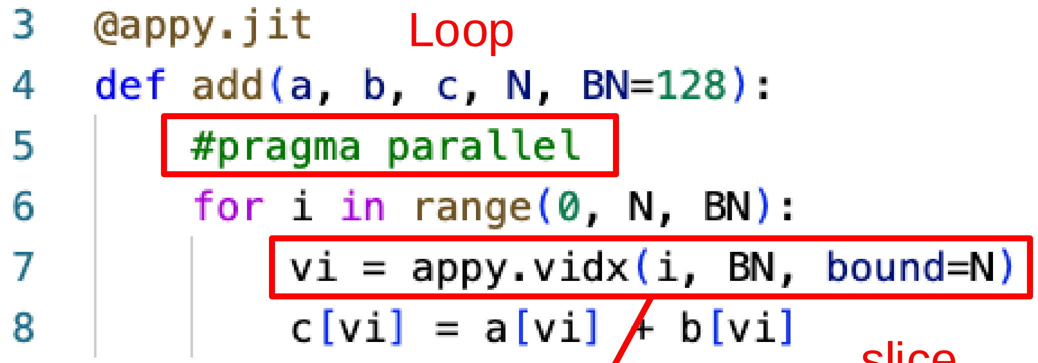


```
@triton.autotune(
    configs=[
        triton.Config({"APPY_BLOCK": 1024}),
        triton.Config({"APPY_BLOCK": 512}),
        triton.Config({"APPY_BLOCK": 256}),
        triton.Config({"APPY_BLOCK": 128}),
    ],
    key=["c_stride_0", "a_stride_0", "b_stride_0"],
)
@triton.jit
def _kernel0(N, c, c_stride_0, a, a_stride_0, b, b_stride_0, APPY_BLOCK: tl.constexpr):
    pass
    _top_var_0 = 0 + tl.program_id(0) * APPY_BLOCK
    tl.store(
        c + (_top_var_0 + tl.arange(0, APPY_BLOCK)) * 1,
        tl.load(
            a + (_top_var_0 + tl.arange(0, APPY_BLOCK)) * 1,
            mask=_top_var_0 + tl.arange(0, APPY_BLOCK) < N,
        )
        + tl.load(
            b + (_top_var_0 + tl.arange(0, APPY_BLOCK)) * 1,
            mask=_top_var_0 + tl.arange(0, APPY_BLOCK) < N,
        ),
        mask=_top_var_0 + tl.arange(0, APPY_BLOCK) < N,
    )


def kernel(a, b, c, N):
    kernel_grid = lambda META: ((N - 0 + META["APPY_BLOCK"] - 1) // META["APPY_BLOCK"],)
    fn = _kernel0[kernel_grid](N, c, c.stride(0), a, a.stride(0), b, b.stride(0))

```

“Loops + Slices”: a simple and flexible programming model

- No prior GPU programming experience is required
- Two key pieces
  - Identify parallel loops
    - Can be nested
  - Process a slice of elements per loop iteration
    - Typically 1-2048 elements

- Performance optimizations are manual
  - Manual loop tiling, fusion etc



```
@appy.jit
def add(a, b, c, N, BN=128):
    #pragma parallel
    for i in range(0, N, BN):
        vi = appy.vidx(i, BN, bound=N)
        c[vi] = a[vi] + b[vi]
```

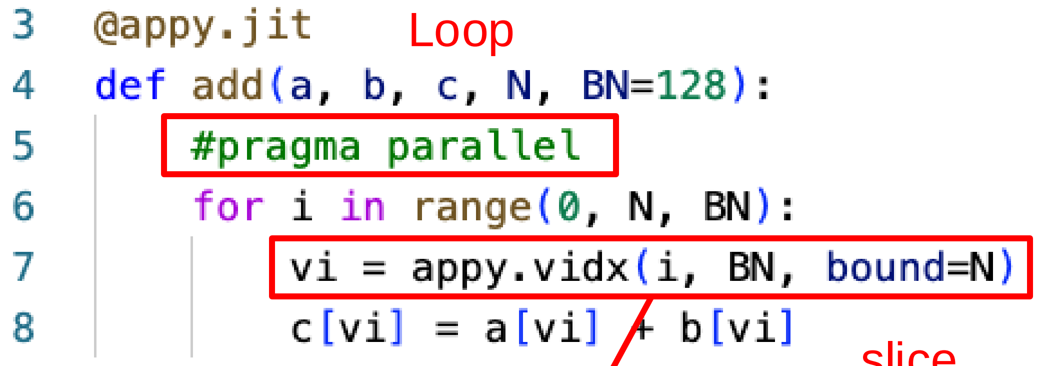
Loop

slice

A built-in function.
vidx” stands for “vector index”
Returns an array [i, i+1, i+2, …, i+BN-1]

“Loops + Slices”: two levels of parallelism

- No prior GPU programming experience is required
- Two levels of parallelism
  - Identify parallel loops
    - Loop iterations run in parallel
  - Process a slice of elements per loop iteration
    - Elements are processed in parallel
- Performance optimizations are manual
  - Manual loop tiling, fusion etc



```
@appy.jit
def add(a, b, c, N, BN=128):
    #pragma parallel
    for i in range(0, N, BN):
        vi = appy.vidx(i, BN, bound=N)
        c[vi] = a[vi] + b[vi]
```

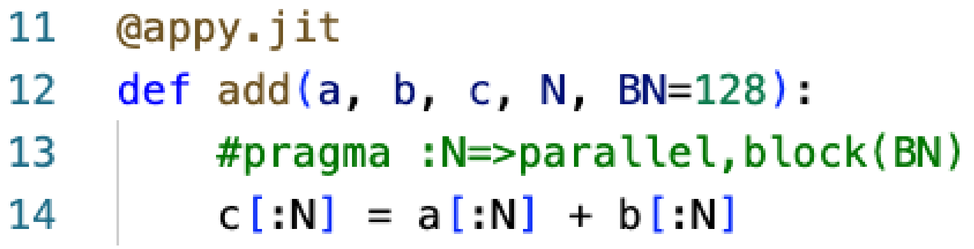
Loop

slice

A built-in function.
vidx” stands for “vector index”
Returns an array [i, i+1, i+2, …, i+BN-1]

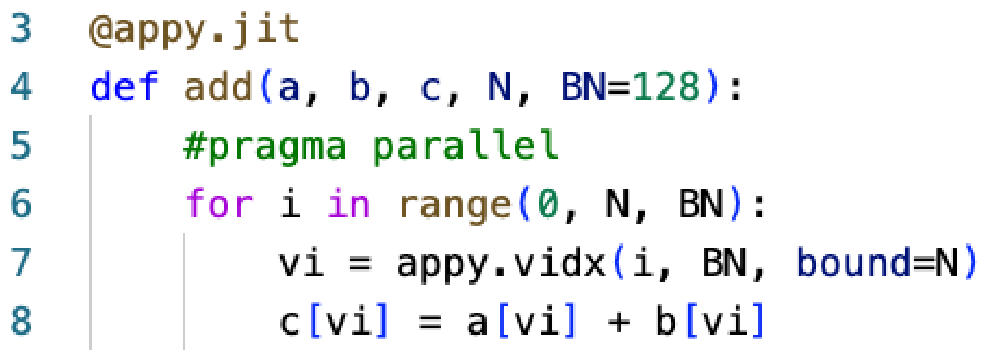
Tensor Oriented Programming Model

- Operate directly on tensors of arbitrary size
- Tensor expressions must be in the form of slicings with explicit upper bound
- User specifies the properties, e.g. parallelism, for each dimension, e.g. :N



```
@appy.jit
def add(a, b, c, N, BN=128):
    #pragma :N=>parallel,block(BN)
    c[:N] = a[:N] + b[:N]
```

=



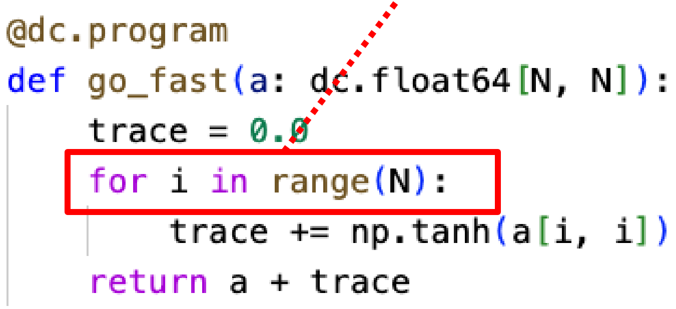
```
@appy.jit
def add(a, b, c, N, BN=128):
    #pragma parallel
    for i in range(0, N, BN):
        vi = appy.vidx(i, BN, bound=N)
        c[vi] = a[vi] + b[vi]
```

:N is the name of the dimension
“parallel,block(BN)” is the property of the dimension

Performance improvement over DaCe by category

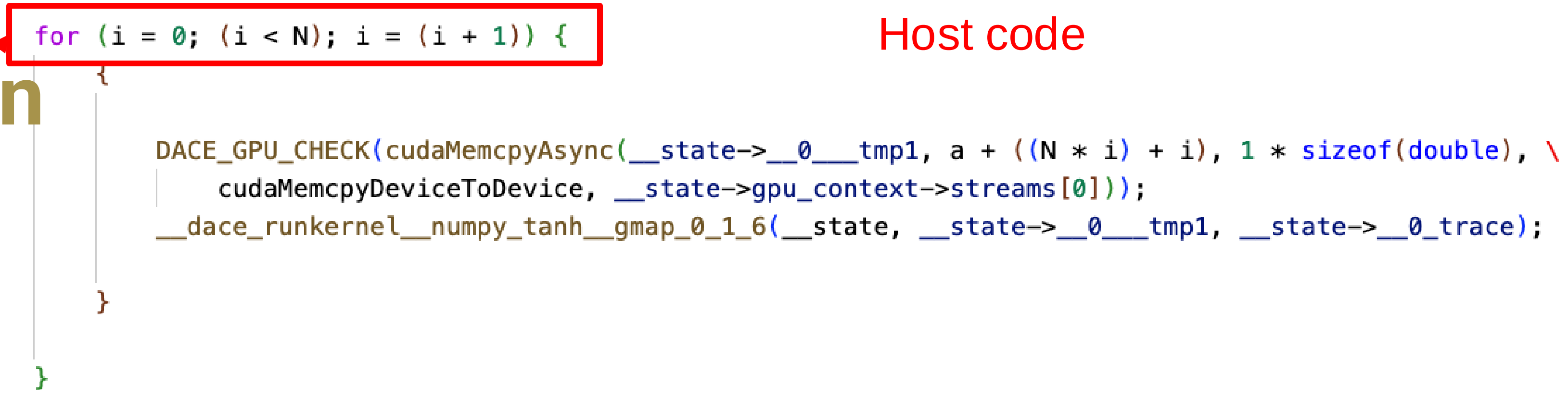
- Stencil
  - Tie with DaCe except for jacobi_1d where appy is ~5x slower
- Linear algebra (loop-based)
  - ~5x faster than DaCe
  - syrk, syr2k, spmv etc
- Solver
  - trisolv, cholesky
  - 2x and 12x faster than DaCe respectively
- Machine learning
  - Softmax
  - ~5x faster than DaCe

DaCe code generation for go_fast



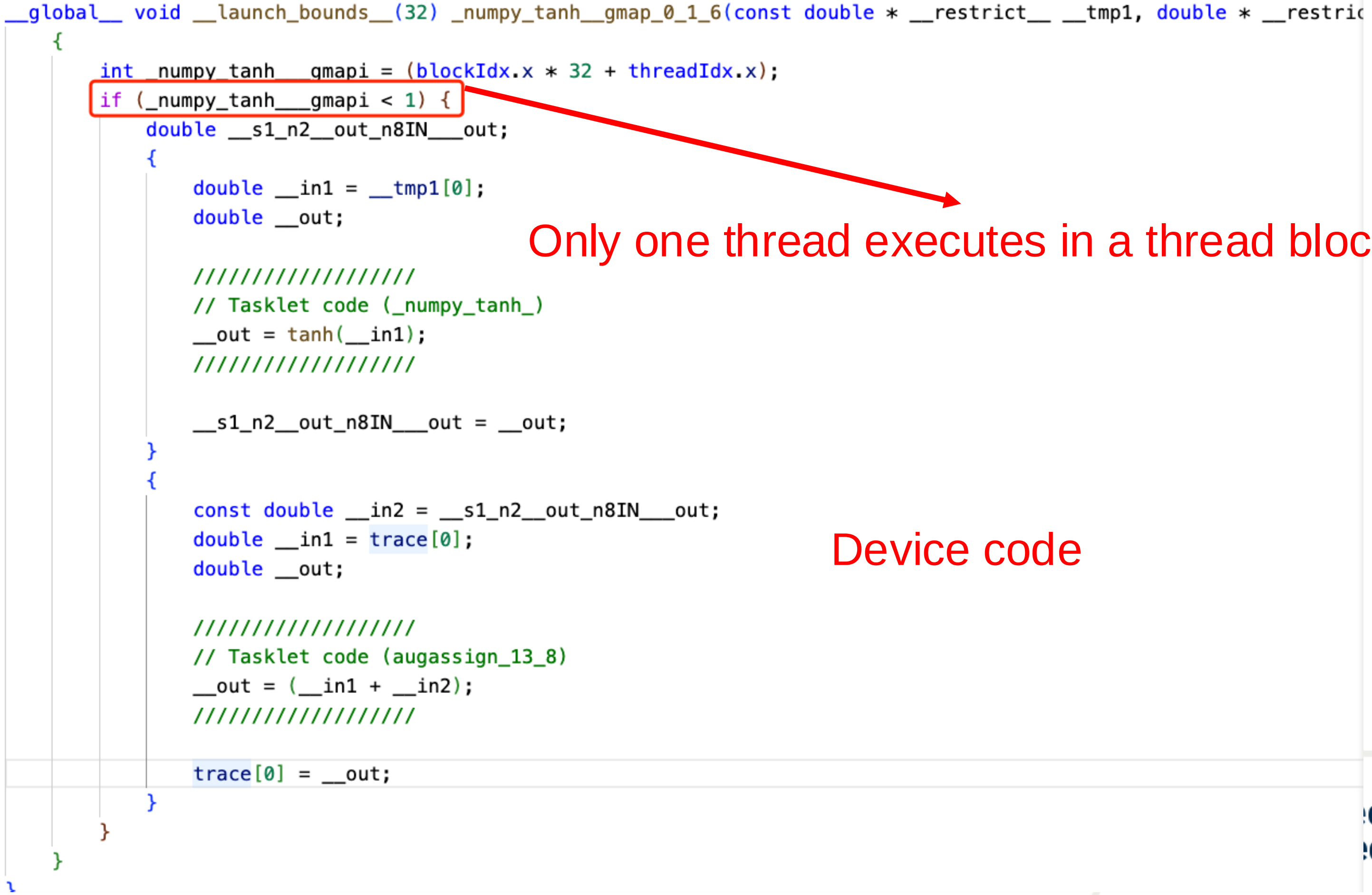
```python
@dc.program
def go_fast(a: dc.float64[N, N]):
    trace = 0.0
    for i in range(N):
        trace += np.tanh(a[i, i])
    return a + trace
```

i loop is sequential



```c
for (i = 0; (i < N); i = (i + 1)) {
    {

        DACE_GPU_CHECK(cudaMemcpyAsync(__state->__0___tmp1, a + ((N * i) + i), 1 * sizeof(double), \
            cudaMemcpyDeviceToDevice, __state->gpu_context->streams[0]));
        __dace_runkernel__numpy_tanh__gmap_0_1_6(__state, __state->__0___tmp1, __state->__0_trace);

    }

}
```

Host code

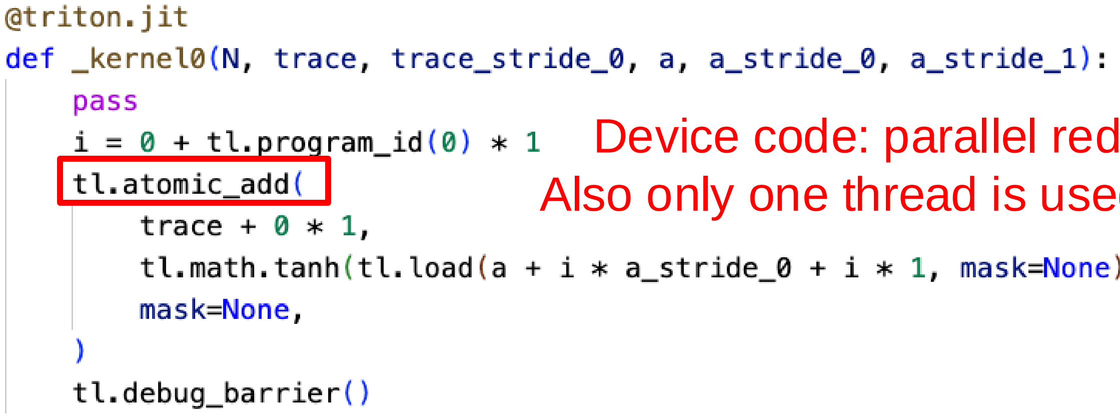


```c
__global__ void __launch_bounds__(32) _numpy_tanh__gmap_0_1_6(const double * __restrict__ __tmp1, double * __restric
    {
        int _numpy_tanh___gmapi = (blockIdx.x * 32 + threadIdx.x);
        if (_numpy_tanh___gmapi < 1) {
            double __s1_n2__out_n8IN___out;
            {
                double __in1 = __tmp1[0];
                double __out;

                ///////////////////
                // Tasklet code (_numpy_tanh_)
                __out = tanh(__in1);
                ///////////////////

                __s1_n2__out_n8IN___out = __out;
            }
            {
                const double __in2 = __s1_n2__out_n8IN___out;
                double __in1 = trace[0];
                double __out;

                ///////////////////
                // Tasklet code (augassign_13_8)
                __out = (__in1 + __in2);
                ///////////////////

                trace[0] = __out;
            }
        }
    }
}
```

Only one thread executes in a thread block

Device code

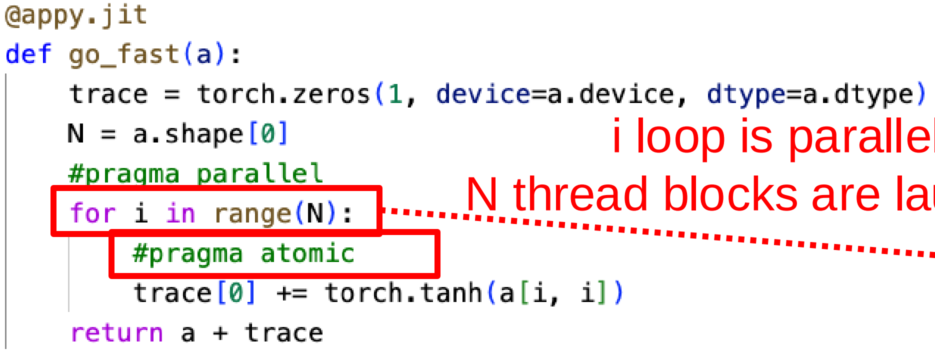
APPy code generation for go_fast



```
@triton.jit
def _kernel0(N, trace, trace_stride_0, a, a_stride_0, a_stride_1):
    pass
    i = 0 + tl.program_id(0) * 1
    tl.atomic_add(
        trace + 0 * 1,
        tl.math.tanh(tl.load(a + i * a_stride_0 + i * 1, mask=None)),
        mask=None,
    )
    tl.debug_barrier()
```

Device code: parallel reduction

Also only one thread is used though

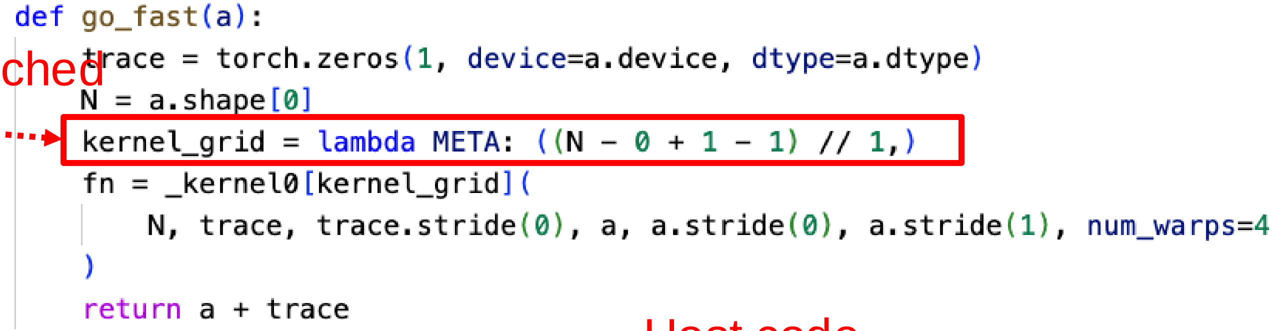


```
@appy.jit
def go_fast(a):
    trace = torch.zeros(1, device=a.device, dtype=a.dtype)
    N = a.shape[0]
    #pragma parallel
    for i in range(N):
        #pragma atomic
        trace[0] += torch.tanh(a[i, i])
    return a + trace
```

i loop is parallel

N thread blocks are launched

Indicates parallel reduction

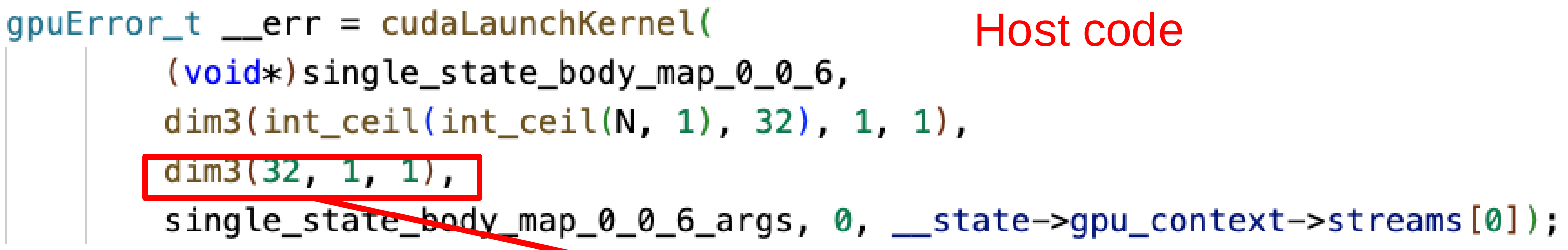


```
def go_fast(a):
    trace = torch.zeros(1, device=a.device, dtype=a.dtype)
    N = a.shape[0]
    kernel_grid = lambda META: ((N - 0 + 1 - 1) // 1,)
    fn = _kernel0[kernel_grid](
        N, trace, trace.stride(0), a, a.stride(0), a.stride(1), num_warps=4
    )
    return a + trace
```

Host code

13x faster than DaCe-GPU!

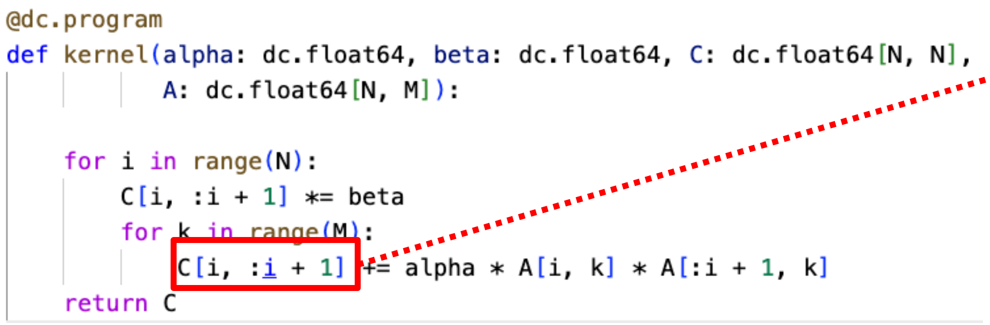
DaCe code generation for syrk



```
gpuError_t __err = cudaLaunchKernel(
        (void*)single_state_body_map_0_0_6,
        dim3(int_ceil(int_ceil(N, 1), 32), 1, 1),
        dim3(32, 1, 1),
        single_state_body_map_0_0_6_args, 0, __state->gpu_context->streams[0]);
```

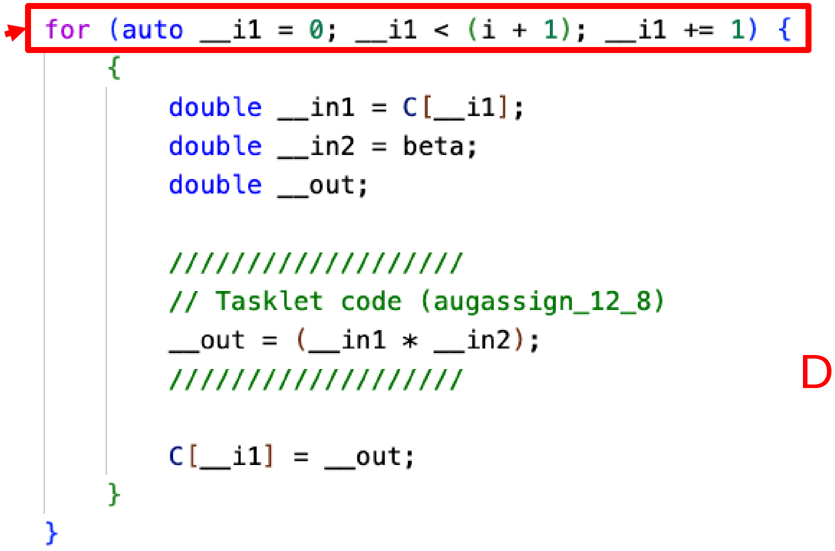
Host code

Kernel launch code: thread block size is fixed to 32



```python
@dc.program
def kernel(alpha: dc.float64, beta: dc.float64, C: dc.float64[N, N],
           A: dc.float64[N, M]):

    for i in range(N):
        C[i, :i + 1] *= beta
        for k in range(M):
            C[i, :i + 1] += alpha * A[i, k] * A[:i + 1, k]
    return C
```

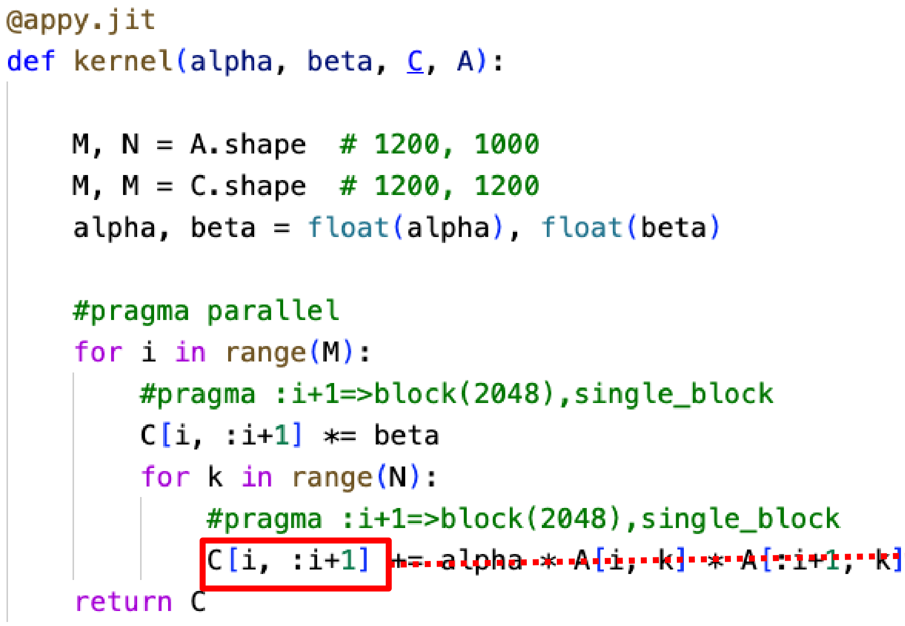
slice :i+1 is sequential



```
for (auto __i1 = 0; __i1 < (i + 1); __i1 += 1) {
    {
        double __in1 = C[__i1];
        double __in2 = beta;
        double __out;

        ///////////////////
        // Tasklet code (augassign_12_8)
        __out = (__in1 * __in2);
        ///////////////////

        C[__i1] = __out;
    }
}
```

Device code

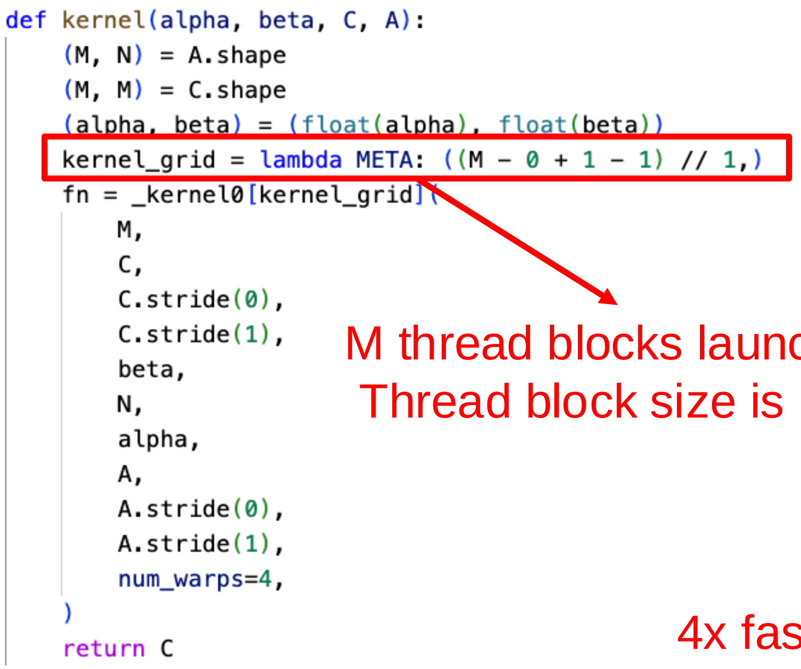
APPy code generation for syrk



```
@appy.jit
def kernel(alpha, beta, C, A):

    M, N = A.shape  # 1200, 1000
    M, M = C.shape  # 1200, 1200
    alpha, beta = float(alpha), float(beta)

    #pragma parallel
    for i in range(M):
        #pragma :i+1=>block(2048),single_block
        C[i, :i+1] *= beta
        for k in range(N):
            #pragma :i+1=>block(2048),single_block
            C[i, :i+1] += alpha * A[i, k] * A[:i+1, k]
    return C
```

slice :i+1 is parallelized

On top of that, an optimization (loop elimination) is applied for small slices



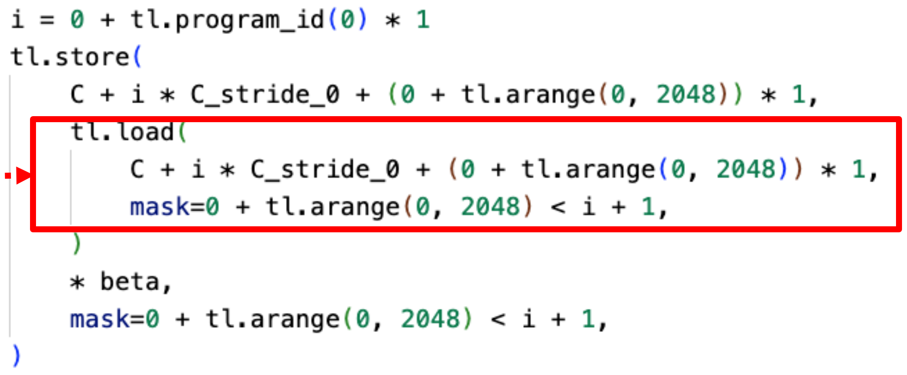
```
def kernel(alpha, beta, C, A):
    (M, N) = A.shape
    (M, M) = C.shape
    (alpha, beta) = (float(alpha), float(beta))
    kernel_grid = lambda META: ((M - 0 + 1 - 1) // 1,)
    fn = _kernel0[kernel_grid](
        M,
        C,
        C.stride(0),
        C.stride(1),
        beta,
        N,
        alpha,
        A,
        A.stride(0),
        A.stride(1),
        num_warps=4,
    )
    return C
```

Host code

M thread blocks launched

Thread block size is 128

4x faster than DaCe-GPU!



```
i = 0 + tl.program_id(0) * 1
tl.store(
    C + i * C_stride_0 + (0 + tl.arange(0, 2048)) * 1,
    tl.load(
        C + i * C_stride_0 + (0 + tl.arange(0, 2048)) * 1,
        mask=0 + tl.arange(0, 2048) < i + 1,
    )
    * beta,
    mask=0 + tl.arange(0, 2048) < i + 1,
)
```

Device code

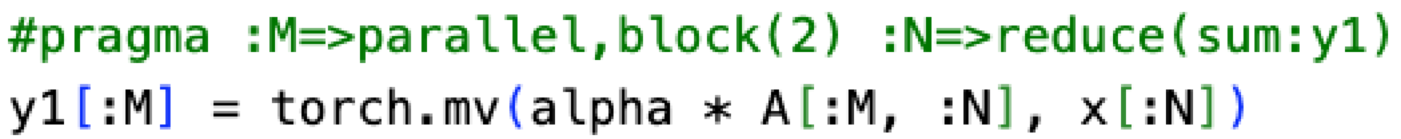
Automatic compiler optimizations

- On top of parallelization, the compiler also performs
  - Loop fusion
  - Loop tiling (via pragma)

Loop fusion case study: gesummv

- Memory footprint without fusion
  - T = alpha * A[:M, :N]
    - One load, one store of MxN matrix
  - mv(T, x)
    - One load of MxN matrix

- Memory footprint with fusion
  - One load of MxN matrix
  - ~3x speedup over CuPy is possible in principle!
  - Achieved speedup in practice: 2.5x



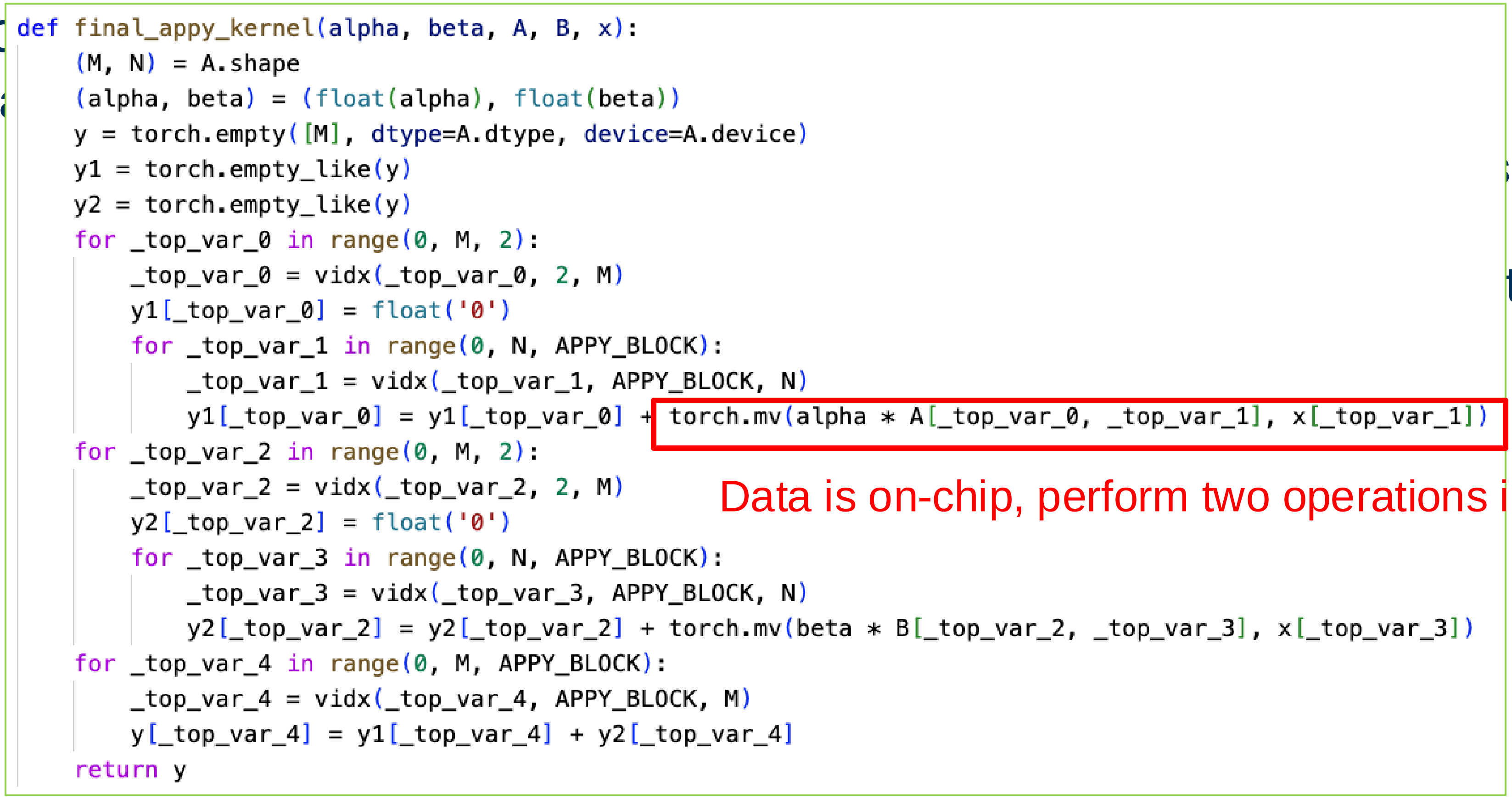
```
#pragma :M=>parallel,block(2) :N=>reduce(sum:y1)
y1[:M] = torch.mv(alpha * A[:M, :N], x[:N])
```

The final APPy code after automatic fusion

- Memory fo
  - T = alpha
    - One
  - mv(T, x)
    - One

possible in

tice: 2.5x



```
def final_appy_kernel(alpha, beta, A, B, x):
    (M, N) = A.shape
    (alpha, beta) = (float(alpha), float(beta))
    y = torch.empty([M], dtype=A.dtype, device=A.device)
    y1 = torch.empty_like(y)
    y2 = torch.empty_like(y)
    for _top_var_0 in range(0, M, 2):
        _top_var_0 = vidx(_top_var_0, 2, M)
        y1[_top_var_0] = float('0')
        for _top_var_1 in range(0, N, APPY_BLOCK):
            _top_var_1 = vidx(_top_var_1, APPY_BLOCK, N)
            y1[_top_var_0] = y1[_top_var_0] + torch.mv(alpha * A[_top_var_0, _top_var_1], x[_top_var_1])
    for _top_var_2 in range(0, M, 2):
        _top_var_2 = vidx(_top_var_2, 2, M)
        y2[_top_var_2] = float('0')
        for _top_var_3 in range(0, N, APPY_BLOCK):
            _top_var_3 = vidx(_top_var_3, APPY_BLOCK, N)
            y2[_top_var_2] = y2[_top_var_2] + torch.mv(beta * B[_top_var_2, _top_var_3], x[_top_var_3])
    for _top_var_4 in range(0, M, APPY_BLOCK):
        _top_var_4 = vidx(_top_var_4, APPY_BLOCK, M)
        y[_top_var_4] = y1[_top_var_4] + y2[_top_var_4]
    return y
```

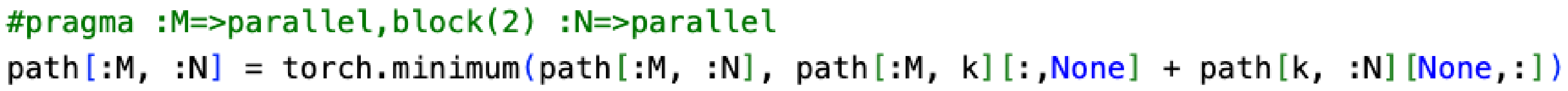
Data is on-chip, perform two operations in a row

```
#pragma :M=>parallel,block(2) :N=>reduce(sum:y1)
y1[:M] = torch.mv(alpha * A[:M, :N], x[:N])
```

Loop fusion case study: floyd_warshall

- Memory footprint without fusion
  - Add.outer
    - One store of MxN matrix
  - Minimum
    - Two loads and one store of MxN matrix
  - Assign
    - One load and one store

- Memory footprint with fusion
  - One load and one store of MxN matrix
  - Theoretical max speedup over CuPy: 3x
  - Actual achieved speedup: 3.3x

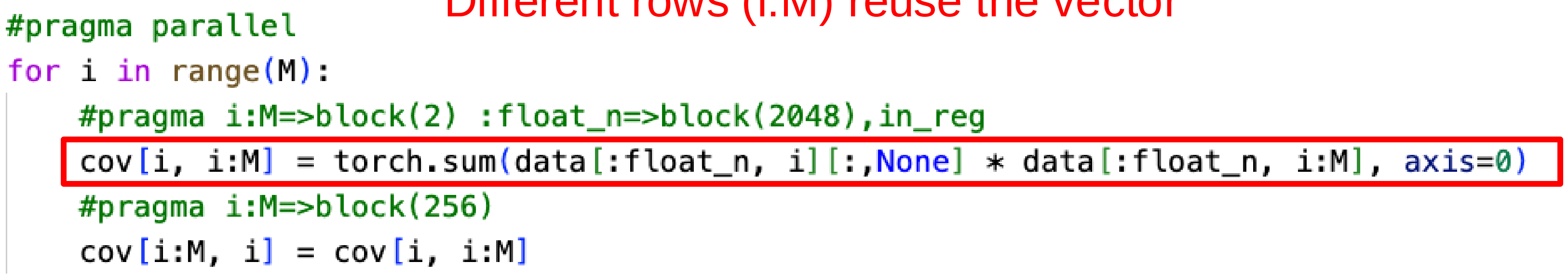


```
#pragma :M=>parallel,block(2) :N=>parallel
path[:M, :N] = torch.minimum(path[:M, :N], path[:M, k][:,None] + path[k, :N][None,:])
```

Loop tiling case study: covariance

A vector-matrix multiplication

Different rows (i:M) reuse the vector



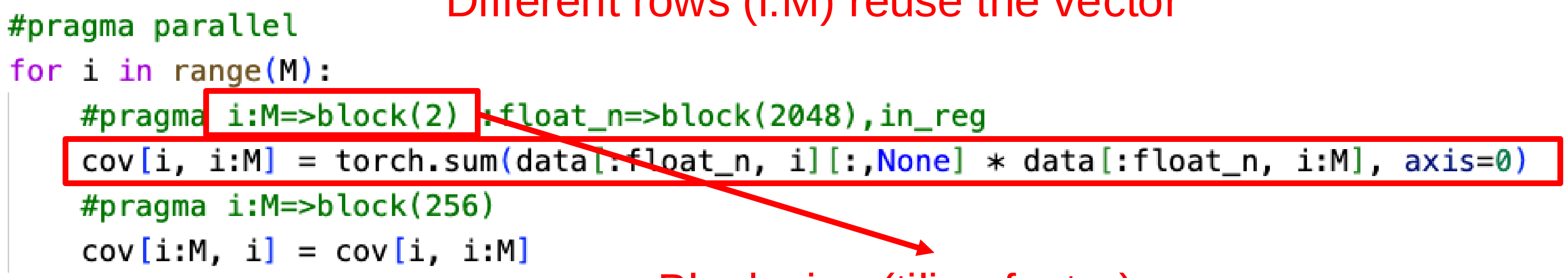
```
#pragma parallel
for i in range(M):
    #pragma i:M=>block(2) :float_n=>block(2048),in_reg
    cov[i, i:M] = torch.sum(data[:float_n, i][:,None] * data[:float_n, i:M], axis=0)
    #pragma i:M=>block(256)
    cov[i:M, i] = cov[i, i:M]
```

Loop tiling case study: covariance

- Blocking the i:M dimension enhances register reuse
  - data[:float_n, i] gets reused
- Equivalent to loop unrolling here

- Without blocking i:M
  - Runtime: 30ms
- With blocking i:M (block size is 2)
  - Runtime: 16ms

A vector-matrix multiplication
Different rows (i:M) reuse the vector



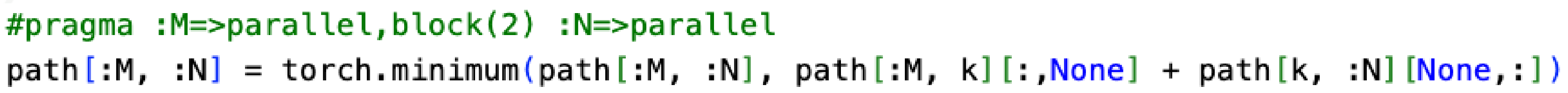
```
#pragma parallel
for i in range(M):
    #pragma i:M=>block(2) ,float_n=>block(2048),in_reg
    cov[i, i:M] = torch.sum(data[:float_n, i][:,None] * data[:float_n, i:M], axis=0)
    #pragma i:M=>block(256)
    cov[i:M, i] = cov[i, i:M]
```

Block size (tiling factor)

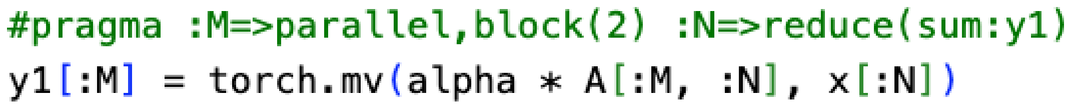
Loop tiling case study: floyd_warshall and gesummv

- Runtime of floyd_warshall
  - Without blocking
    - Runtime: 29ms
  - With blocking (block size is 2)
    - Runtime: 28ms

- Runtime of gesummv
  - Without blocking
    - 3ms
  - With blocking (block size is 2)
    - 3ms



```
#pragma :M=>parallel,block(2) :N=>parallel
path[:M, :N] = torch.minimum(path[:M, :N], path[:M, k][:,None] + path[k, :N][None,:])
```



```
#pragma :M=>parallel,block(2) :N=>reduce(sum:y1)
y1[:M] = torch.mv(alpha * A[:M, :N], x[:N])
```

Blocking is not helping much here, finer grain performance analysis is needed to diagnose why.

Evaluation

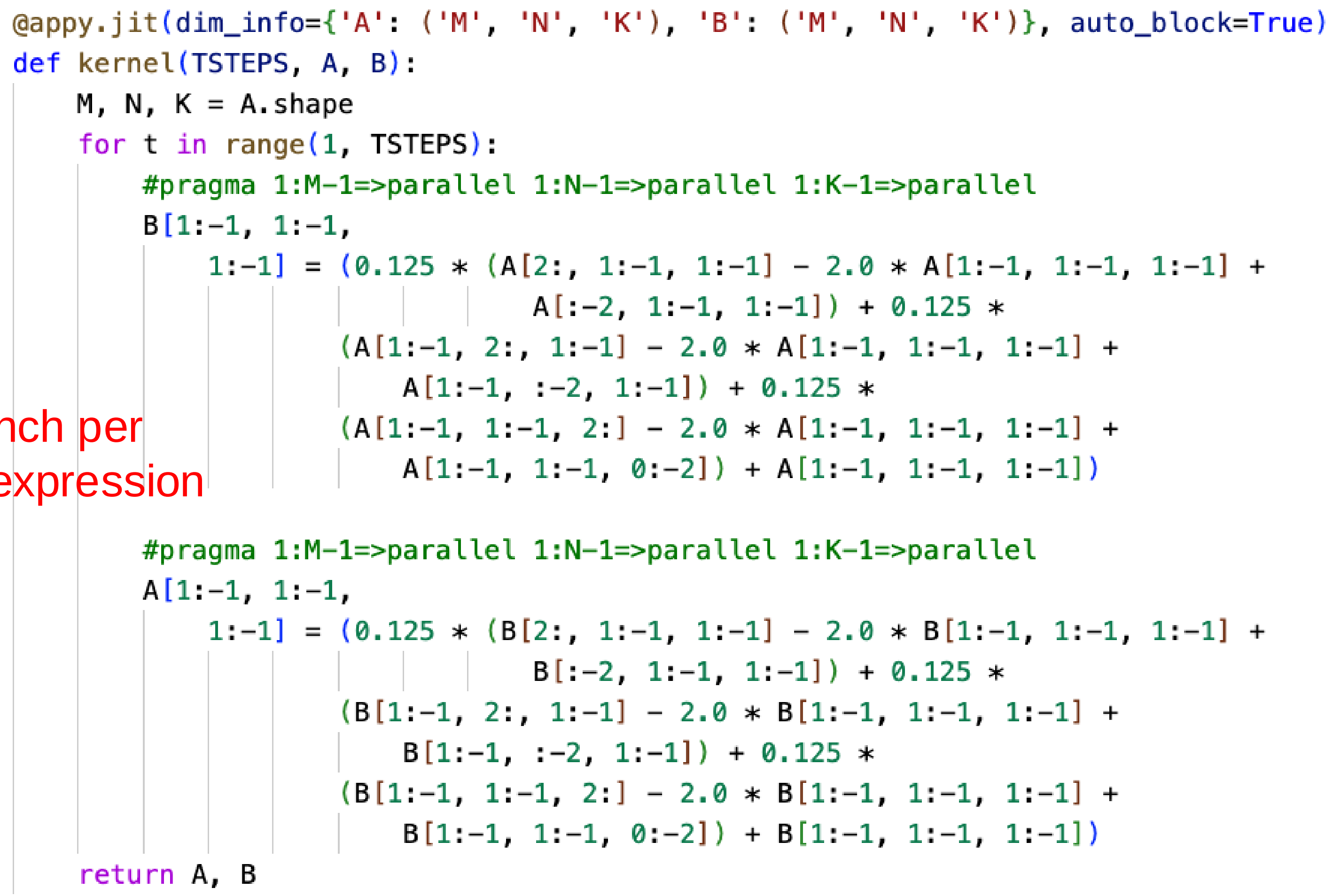
- Programmability evaluation
- Performance evaluation

Programmability evaluation

- Original program structure is kept as much as possible
  - We try to only add pragmas, and only change the program structure when necessary
- Programming model adoption stats
  - Use vanilla model only
    - 3/19
  - Use tensor expressions only
    - 8/19
  - Use loop + tensor expressions
    - 8/19

- The only benchmarks that had code adaptations besides annotations
  - Softmax
  - Spmv
  - Azimint_naive
- Other conventions
  - Parallel for loops must be a range loop
  - The result of parallel reduction must be an array, even if size is 1

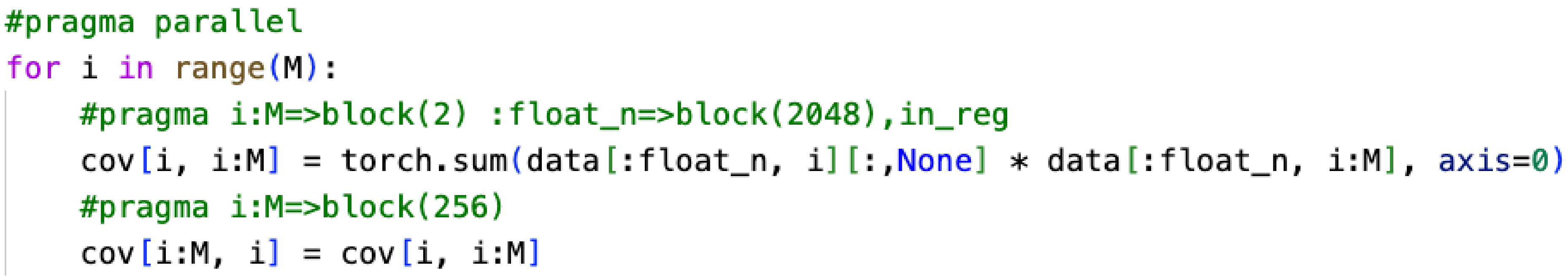
Typical stencil kernel: heat_3d



```
@appy.jit(dim_info={'A': ('M', 'N', 'K'), 'B': ('M', 'N', 'K')}, auto_block=True)
def kernel(TSTEPS, A, B):
    M, N, K = A.shape
    for t in range(1, TSTEPS):
        #pragma 1:M-1=>parallel 1:N-1=>parallel 1:K-1=>parallel
        B[1:-1, 1:-1,
            1:-1] = (0.125 * (A[2:, 1:-1, 1:-1] - 2.0 * A[1:-1, 1:-1, 1:-1] +
                            A[:-2, 1:-1, 1:-1]) + 0.125 *
                    (A[1:-1, 2:, 1:-1] - 2.0 * A[1:-1, 1:-1, 1:-1] +
                        A[1:-1, :-2, 1:-1]) + 0.125 *
                    (A[1:-1, 1:-1, 2:] - 2.0 * A[1:-1, 1:-1, 1:-1] +
                        A[1:-1, 1:-1, 0:-2]) + A[1:-1, 1:-1, 1:-1])

        #pragma 1:M-1=>parallel 1:N-1=>parallel 1:K-1=>parallel
        A[1:-1, 1:-1,
            1:-1] = (0.125 * (B[2:, 1:-1, 1:-1] - 2.0 * B[1:-1, 1:-1, 1:-1] +
                            B[:-2, 1:-1, 1:-1]) + 0.125 *
                    (B[1:-1, 2:, 1:-1] - 2.0 * B[1:-1, 1:-1, 1:-1] +
                        B[1:-1, :-2, 1:-1]) + 0.125 *
                    (B[1:-1, 1:-1, 2:] - 2.0 * B[1:-1, 1:-1, 1:-1] +
                        B[1:-1, 1:-1, 0:-2]) + B[1:-1, 1:-1, 1:-1])
    return A, B
```

One kernel launch per
annotated tensor expression

Typical loop-based kernel: covariance



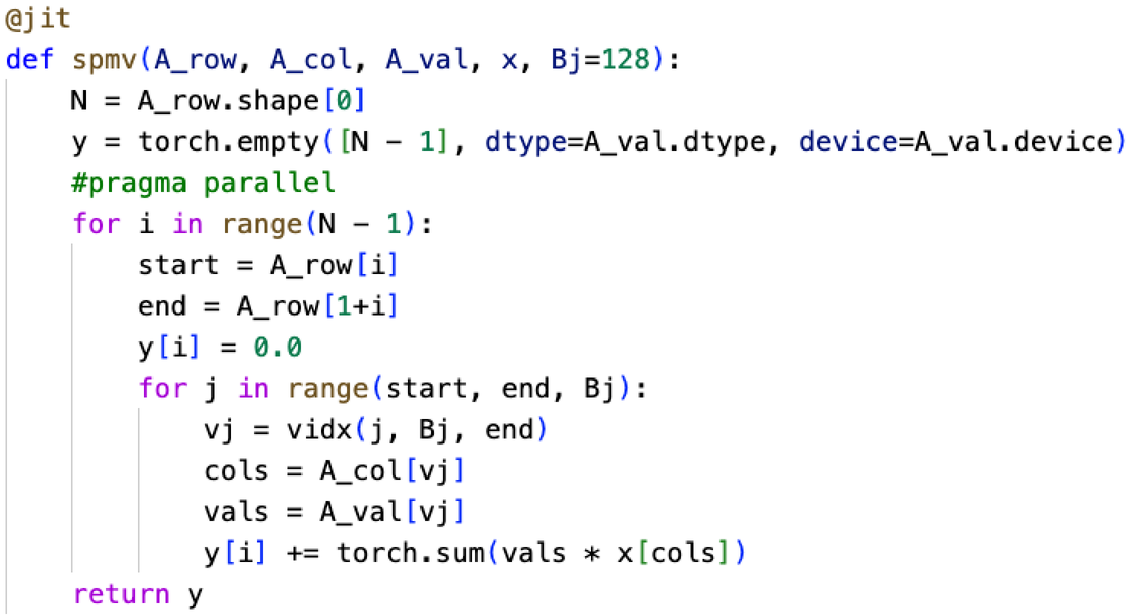
```
#pragma parallel
for i in range(M):
    #pragma i:M=>block(2) :float_n=>block(2048),in_reg
    cov[i, i:M] = torch.sum(data[:float_n, i][:,None] * data[:float_n, i:M], axis=0)
    #pragma i:M=>block(256)
    cov[i:M, i] = cov[i, i:M]
```

One kernel launch per parallel loop

Vanilla programming model alone

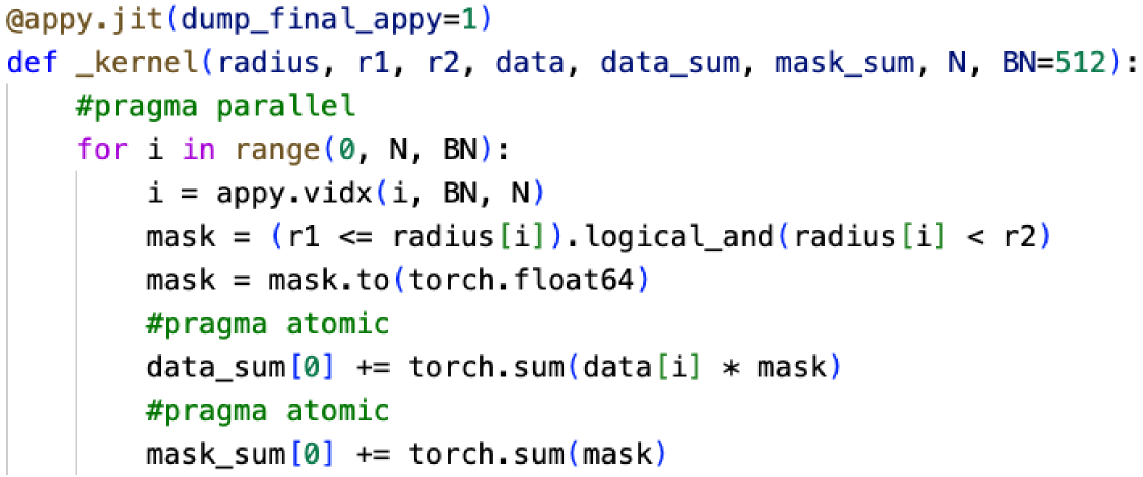
- Sometimes we use the vanilla model alone if more flexibility is needed, such as in spmv and azimint_naive

spmv



```
@jit
def spmv(A_row, A_col, A_val, x, Bj=128):
    N = A_row.shape[0]
    y = torch.empty([N - 1], dtype=A_val.dtype, device=A_val.device)
    #pragma parallel
    for i in range(N - 1):
        start = A_row[i]
        end = A_row[1+i]
        y[i] = 0.0
        for j in range(start, end, Bj):
            vj = vidx(j, Bj, end)
            cols = A_col[vj]
            vals = A_val[vj]
            y[i] += torch.sum(vals * x[cols])
    return y
```

azimint_naive



```
@appy.jit(dump_final_appy=1)
def _kernel(radius, r1, r2, data, data_sum, mask_sum, N, BN=512):
    #pragma parallel
    for i in range(0, N, BN):
        i = appy.vidx(i, BN, N)
        mask = (r1 <= radius[i]).logical_and(radius[i] < r2)
        mask = mask.to(torch.float64)
        #pragma atomic
        data_sum[0] += torch.sum(data[i] * mask)
        #pragma atomic
        mask_sum[0] += torch.sum(mask)
```

Comparison of the two programming models

- Block-oriented model
- Compose programs using loops + blocked tensor operations, only work with a small chunk of data at a time
- High flexibility
- Low productivity

- Tensor-oriented model
- Compose programs using tensor expressions, annotate each individual dimension as parallel or not
- Low flexibility
- High productivity

Memory consistency model implementation

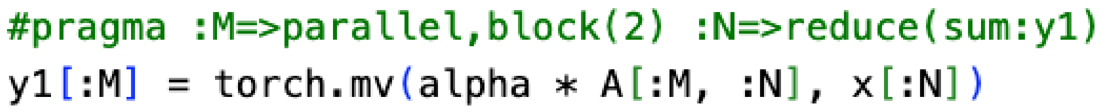
- Correctness condition: there must exist a __syncthreads() between any pair of memory operations that have data dependence
- A simple implementation: Insert a __syncthreads() after every memory load and store, except for tensors that are only ever loaded

Synchronization optimization

- Tensor expressions are “regular” operations so some extraneous thread synchronizations can be skipped
- Only necessary to insert one __syncthreads() before and after the loop, not within

Some constraints

- Multi-dimensional tensor expression is fine
- Each dimension must be uniquely named
- Every dimension must have an entry in the pragma
- A reduction dimension must be specified in the pragma

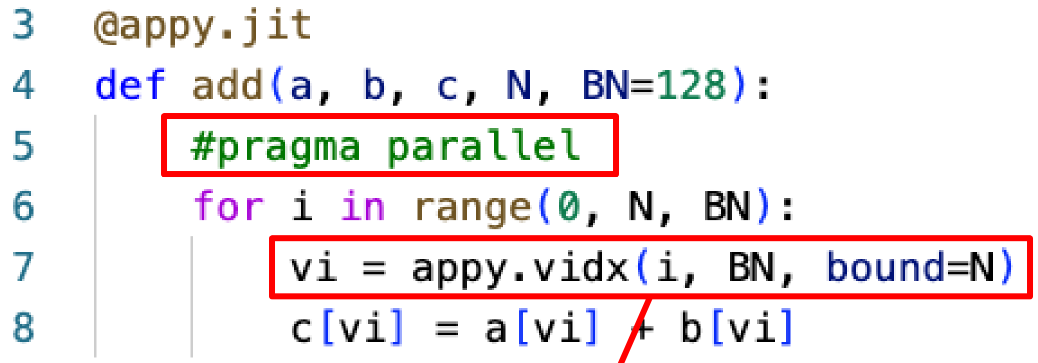


```
#pragma :M=>parallel,block(2) :N=>reduce(sum:y1)
y1[:M] = torch.mv(alpha * A[:M, :N], x[:N])
```

Two dimensions
:M and :N

Storage implication

- Arrays
  - Global memory
- Data block (variable)
  - On-chip storage, e.g. registers



```
@appy.jit
def add(a, b, c, N, BN=128):
    #pragma parallel
    for i in range(0, N, BN):
        vi = appy.vidx(i, BN, bound=N)
        c[vi] = a[vi] + b[vi]
```

Equivalent to

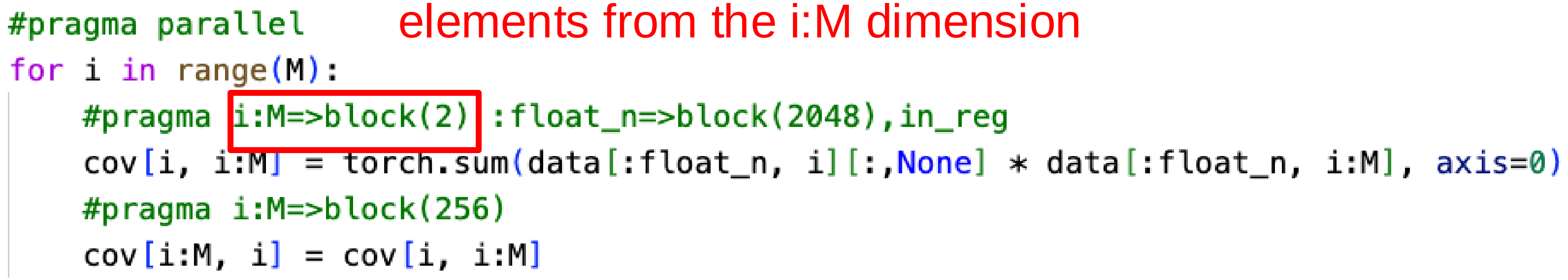
```
vi = range(i, min(i+BN, N))
```

Loop tiling case study: covariance

- Blocking the i:M dimension enhances register reuse
  - data[:float_n, i] gets reused
- Equivalent to loop unrolling here

- Without blocking i:M
  - Runtime: 30ms
- With blocking i:M (block size is 2)
  - Runtime: 16ms

Register reuse achieved. Each thread handles two elements from the i:M dimension



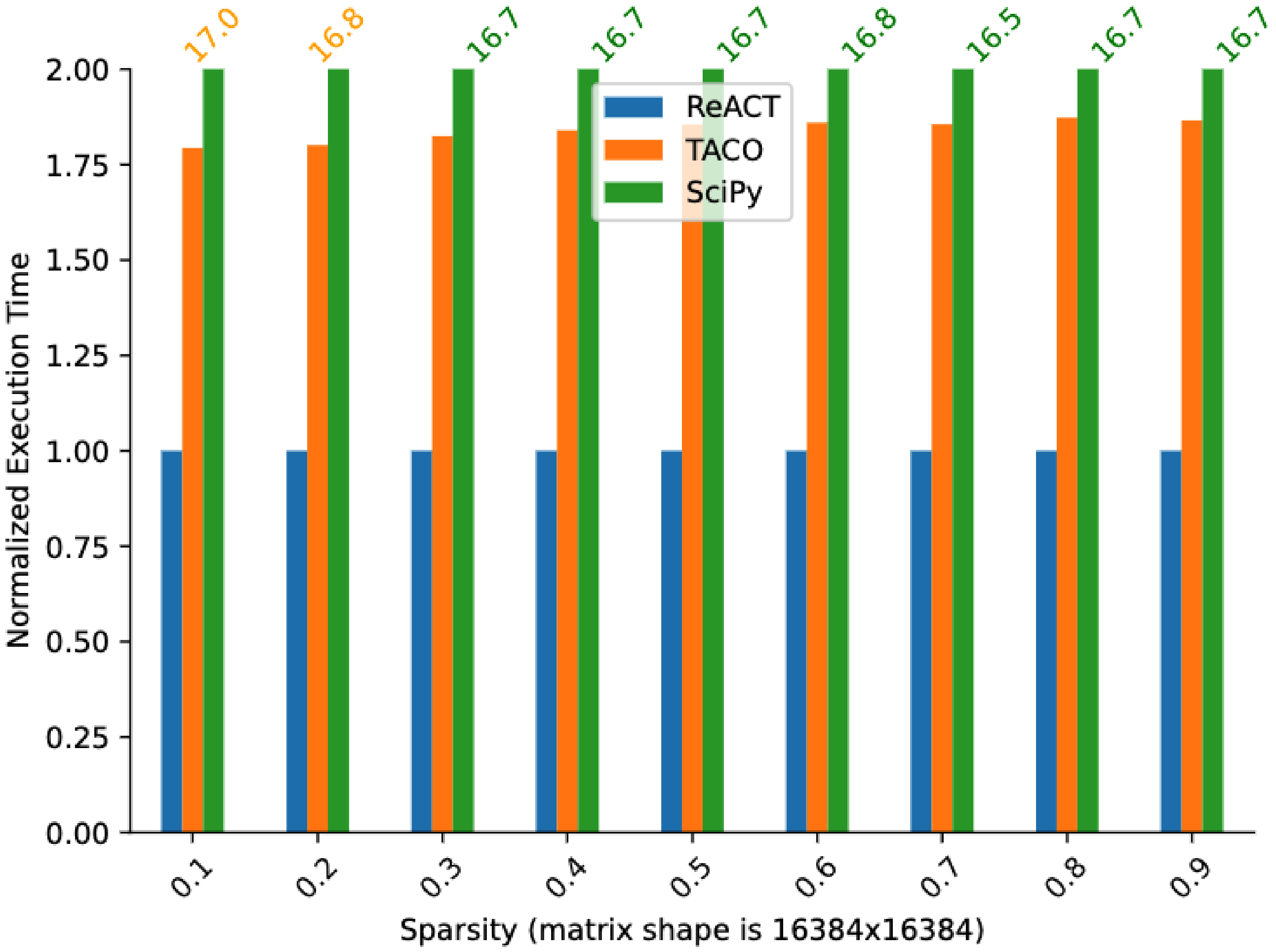
```
#pragma parallel
for i in range(M):
    #pragma i:M=>block(2) :float_n=>block(2048),in_reg
    cov[i, i:M] = torch.sum(data[:float_n, i][:,None] * data[:float_n, i:M], axis=0)
    #pragma i:M=>block(256)
    cov[i:M, i] = cov[i, i:M]
```

ReACT backup

How is ReACT able to reduce more redundancies?

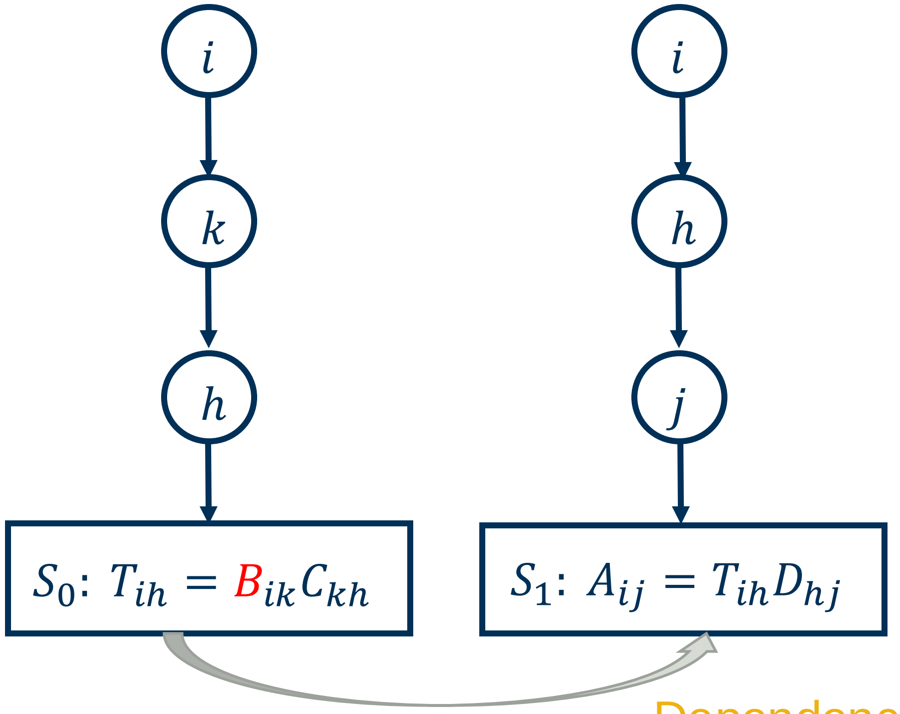
- It uses a tree-based intermediate representation (IR), and transforms the IR with redundancies-aware transformation passes (fully automatic)
  - A pass to perform partial fusion thus to reduce type 1 and 2 redundancy
  - A pass to reduce the intermediate storages to minimal sizes to reduce type 3 redundancy
  - …
- Let's look at some performance numbers before getting into *how* ReACT generates code with less redundancies

Sparse-softmax N=16384



Redundancy-Aware fusion via index tree

- Two operations => create two subtrees
  - S_0: $T_{ih} = B_{ik} * C_{kh}$ (sparse-dense MM, *B* is CSR format)
  - S_1: $A_{ij} = T_{ih} * D_{hj}$ (dense MM)

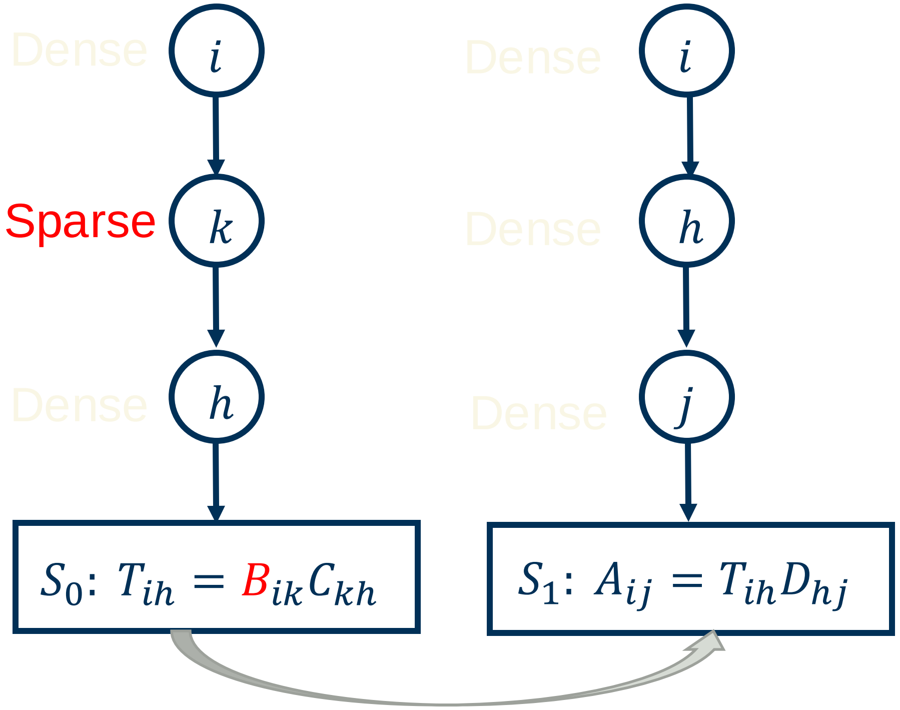


i
k
h
S_0: $T_{ih} = B_{ik}C_{kh}$
i
h
j
S_1: $A_{ij} = T_{ih}D_{hj}$

Dependence edge

SpMM-MM index trees

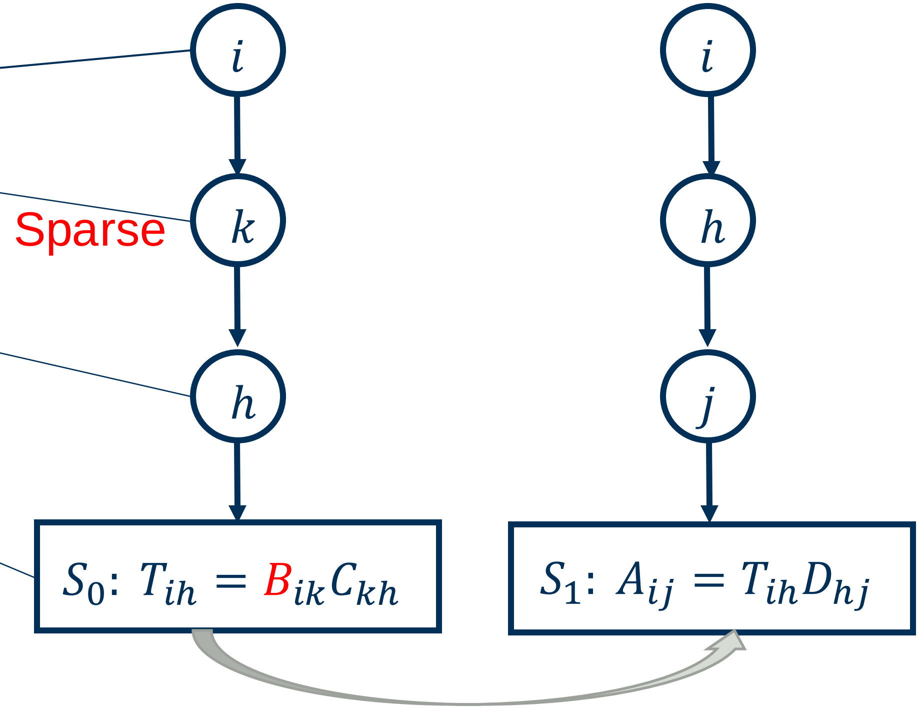
- Annotate each index node as “Dense” or “Sparse”


Dense
i
Sparse
k
Dense
h
S_0: $T_{ih} = B_{ik}C_{kh}$
Dense
i
Dense
h
Dense
j
S_1: $A_{ij} = T_{ih}D_{hj}$

Index tree corresponding loop structure

- k is a sparse (compressed) loop while i and h are dense loops.

```
for (int i = 0; i < NI; i++) {
    for (int k = B.rowptrs[i]; k < B.rowptrs[i+1]; k++) {
        for (int h = 0; h < NH; h++) {
            ...
            T[i,h] = B[i,k] * C[k,h];
            ...
        }
    }
}
```


i
k
Sparse
h
S_0: $T_{ih} = B_{ik}C_{kh}$
i
h
j
S_1: $A_{ij} = T_{ih}D_{hj}$

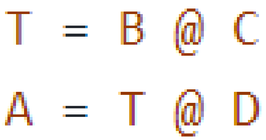
Redundancy-Aware fusion using index tree

- Library approach
  - No fusion
- TACO (a SOTA sparse tensor compiler)
  - Maximal fusion
- ReACT (our work)
  - Partial fusion

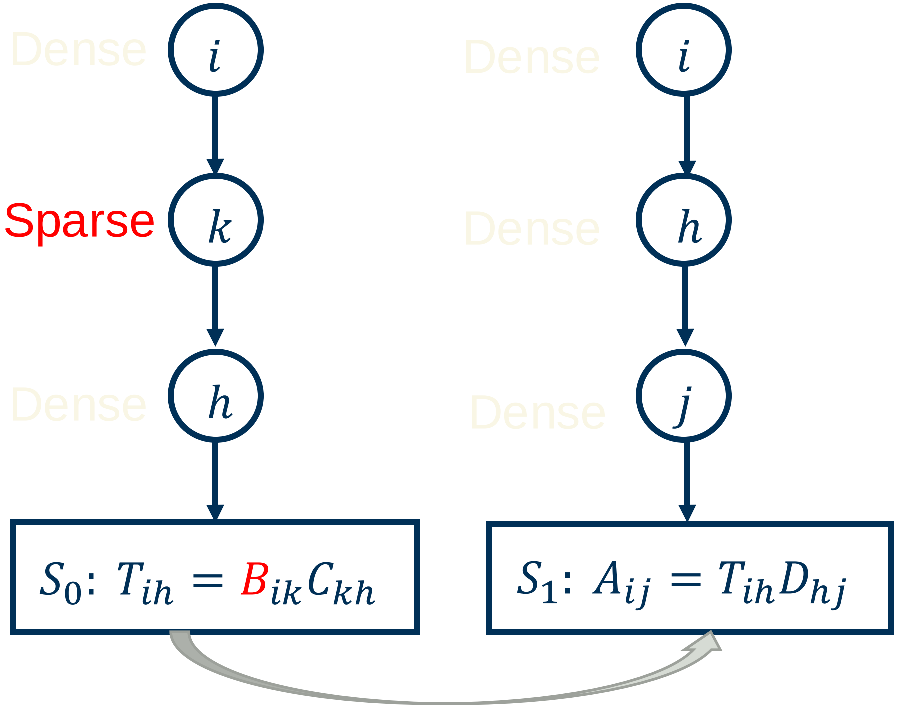
SpMM-MM index trees: no fusion

- Time: Good, $O(NNZ_B * NH + NI * NH * NJ)$
- Intermediate space: Poor, $O(NI * NH)$
- Locality: Poor

Generated code (library calls)



```
T = B @ C
A = T @ D
```


Dense
i
Sparse
k
Dense
h
$S_0\colon T_{ih} = B_{ik} C_{kh}$
Dense
i
Dense
h
Dense
j
$S_1\colon A_{ij} = T_{ih} D_{hj}$

Future work

- More optimizations
  - LICM is applicable for some benchmarks, such as syrk and covariance
- More autotuning
  - Now num_warps is fixed to 4 (128 threads), not always optimal
- Automatically add/search pragmas
  - Some pragmas may be inferred
- Fuse across tensor expressions
- Support multi-node distributed memory parallelism

Intrepydd backup

Code Optimization: Array Memory Recycling

```
it = 0
while it < max_iter:
    A = B + C   # all arrays
    …
    it += 1
```

Intrepydd source code

```
A = empty_like(B)
while it < max_iter:
    add(B, C, out=A)
    …
    it += 1
```

Transformed code

Code Optimization: Array Memory Recycling

```
it = 0
while it < max_iter:
    A = B + C   # all arrays
    …
    it += 1
```

```
A = empty_like(B)
while it < max_iter:
    add(B, C, out=A)
    …
    it += 1
```

This also reduces reference counting management overhead

Intrepydd source code

Transformed code

Code Optimization: Array Memory Recycling

- At an allocation site, and determine whose memory can be reused
- A variable's memory can only be reused if
    - It is a unique pointer of its memory
    - It is dead at this point
    - Namely, in the unique pointer set, but not in alive set
- Requires two data flow analysis: liveness analysis and unique-pointer analysis
    - A unique pointer set per program point
- Non-Aliasing-Creating statements:
    - Binary op
    - Unary op
    - All others are considered alias creating