

APPy: Annotated Parallelism for Python on GPUs

Tong Zhou, Jun Shirako and Vivek Sarkar



Motivation

- 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/Numba CUDA 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

A quick example of APPy

- An ordinary loop-based SpMV implementation in Python

```
1. def spmv(A_row, A_col, A_val, x):
2.     N = A_row.shape[0]
3.     y = empty([N - 1], dtype=A_val.dtype)
4.     for i in range(N - 1):
5.         y[i] = 0.0
6.         for j in range(A_row[i], A_row[1+i]):
7.             cols = A_col[j]
8.             y[i] += A_val[j] * x[cols]
9.     return y
```

A quick example of APPy

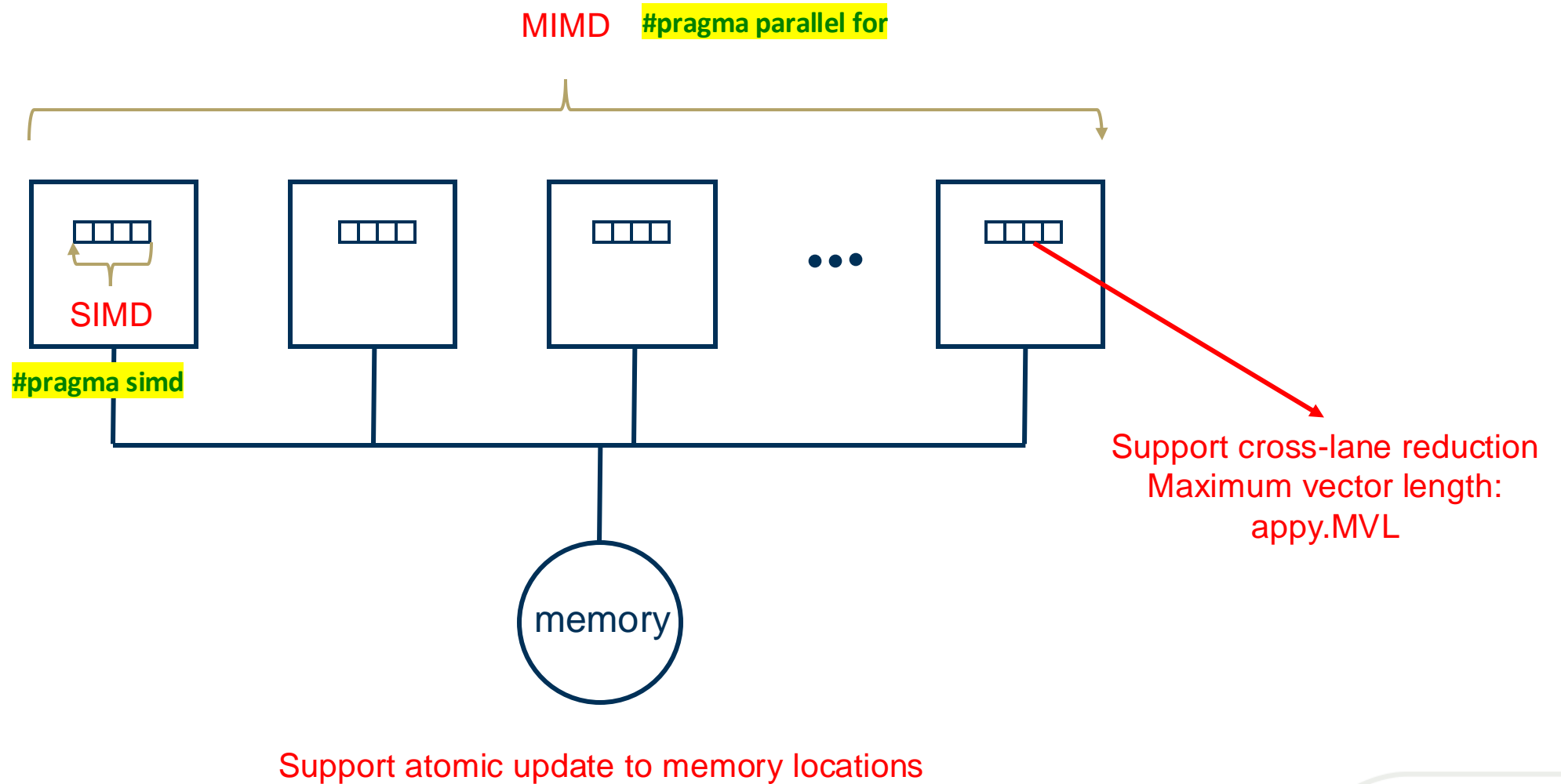
- An ordinary loop-based SpMV implementation in Python

```
1. def spmv(A_row, A_col, A_val, x):
2.     N = A_row.shape[0]
3.     y = empty([N - 1], dtype=A_val.dtype)
4.     for i in range(N - 1):
5.         y[i] = 0.0
6.         for j in range(A_row[i], A_row[1+i]):
7.             cols = A_col[j]
8.             y[i] += A_val[j] * x[cols]
9.     return y
```

- Ordinary SpMV parallelized with APPy

```
1. @appy.jit
2. def spmv(A_row, A_col, A_val, x):
3.     N = A_row.shape[0]
4.     y = empty([N - 1], dtype=A_val.dtype)
5.     #pragma parallel for
6.     for i in range(N - 1):
7.         y[i] = 0.0
8.         #pragma simd
9.         for j in range(A_row[i], A_row[1+i]):
10.             cols = A_col[j]
11.             y[i] += A_val[j] * x[cols]
12.     return y
```

Abstract machine model: a multi-vector processor



APPy compiler directives

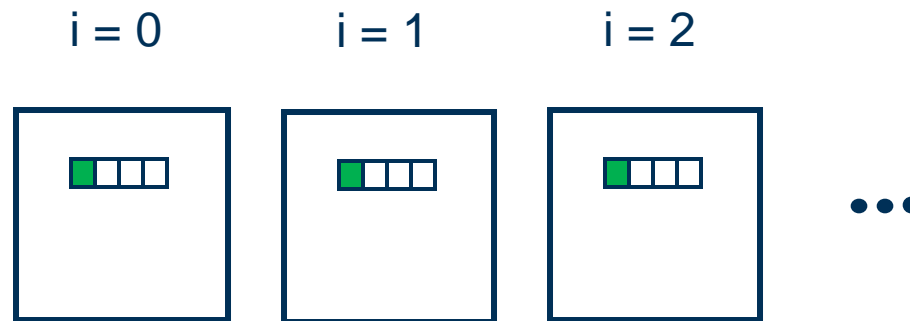
- Annotations for loops
 - #pragma parallel for
 - #pragma sequential for
 - #pragma simd
 - Annotations for statements
 - #pragma atomic
 - Annotations for tensor expressions
 - #pragma {dim}=>{properties}
- Difference from OpenMP codegen
 - OpenMP directly exposes the parallelism hierarchy of the GPUs and requires more complicated pragmas to generate GPU code
 - OpenMP does not recognize and compile tensor expressions

Vector addition with APPy

Software

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

Hardware
(abstract)

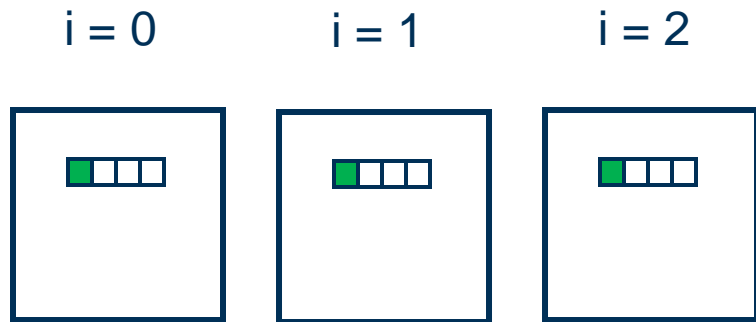
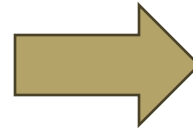


N workers launched

Utilize both layers of parallelism: parallel for + simd

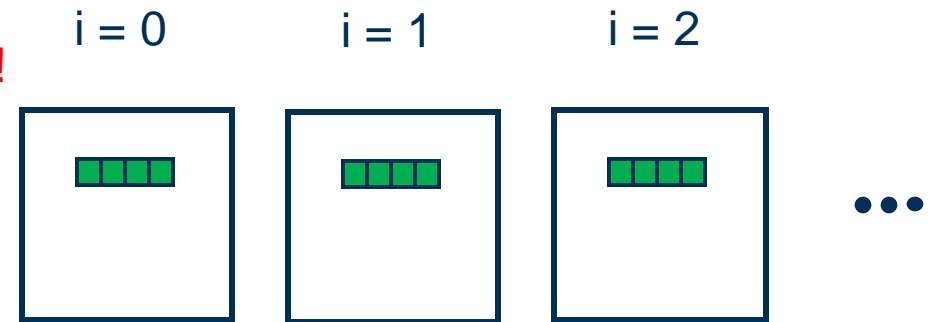
```
1. @appy.jit
2. def vector_add(a, b, c, N):
3.     #pragma parallel for
4.     for i in range(N):
5.         c[i] = a[i] + b[i]
```

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



N workers launched

Performance boost!



$\frac{N}{\text{vector length}}$ workers launched

Two ways to utilize vectorization

Annotate the loop with
`#pragma simd`

```
1. @appy.jit
2. def softmax_loop_oriented(a, b, M, N):
3.     #pragma parallel for
4.     for i in range(M):
5.         m = float('-inf')
6.         #pragma simd
7.         for j in range(N):
8.             m = maximum(m, a[i,j])
9.         s = 0.0
10.        #pragma simd
11.        for j in range(N):
12.            s += exp(a[i,j] - m)
13.        #pragma simd
14.        for j in range(N):
15.            b[i,j] = exp(a[i,j] - m) / s
```



Write tensor expressions

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

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

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

Tensor expressions can be parallelized too

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

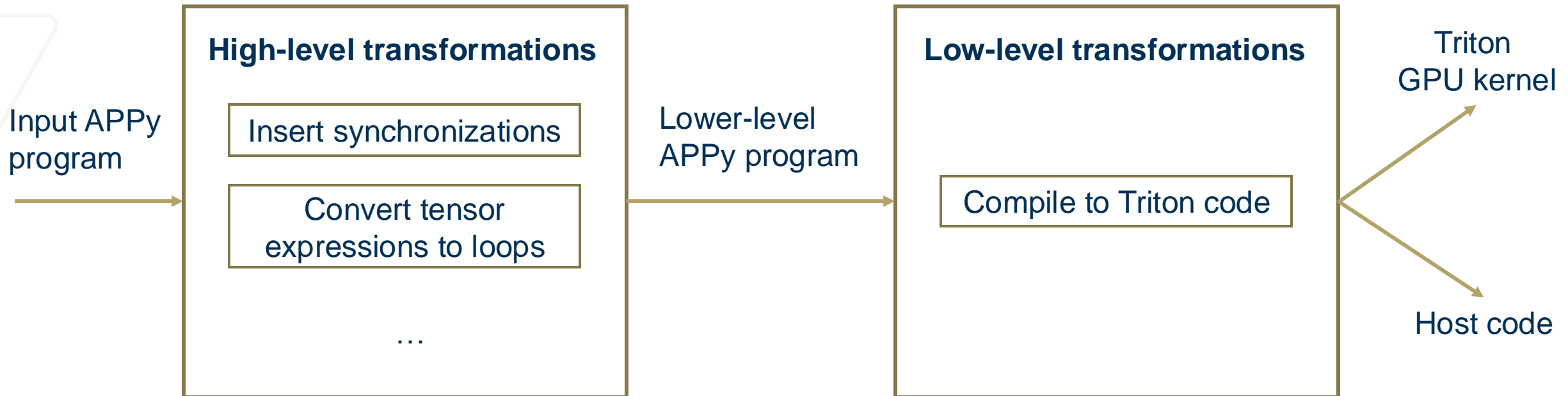


Compiler generated explicit loop

```
1. @appy.jit
2. def gemv_transformed(alpha, A, x):
3.     M, N = A.shape
4.     #pragma parallel for
5.     for _i0 in range(0, M, 1):
6.         y[_i0] = 0.0
7.         for _i1 in range(0, N, appy.MVL):
8.             _v1 = appy.vidx(_i1, appy.MVL, N)
9.             y[_i0] += sum(alpha * A[_i0, _v1] * x[_v1])
```

Implementation

- All transformation passes are Python AST based



A code generation example

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

High-level transform

```
1. @appy.jit
2. def gemv(alpha, A, x):
3.     M, N = A.shape
4.     #pragma parallel for
5.     for _i0 in range(0, M, 1):
6.         tmp = 0.0
7.         for _i1 in range(0, N, appy.MVL):
8.             _v1 = appy.vidx(_i1, appy.MVL, N)
9.             tmp += sum(alpha * A[_i0, _v1] * x[_v1])
10.        y[_i0] = tmp
```

Gen device code

```
1. @triton.jit
2. def _kernel(M, N, A, A_stride0, A_stride1, x, \
3.            x_stride0, y, y_stride0, MVL: tl.constexpr):
4.     _i0 = tl.program_id(0) * 1
5.     tmp = 0.0
6.     for _i1 in range(0, N, MVL):
7.         tmp += tl.sum(
8.             alpha * tl.load(
9.                 A + _i0*A_stride0 + \
10.                    _i1 + tl.arange(0, MVL),
11.                 mask=_i1 + tl.arange(0, MVL) < N
12.             ) *
13.             tl.load(
14.                 x + _i1 + tl.arange(0, MVL),
15.                 mask=_i1 + tl.arange(0, MVL) < N
16.             )
17.         )
18.     tl.store(y + _i0, tmp)
```

Gen host code

```
def gemv(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

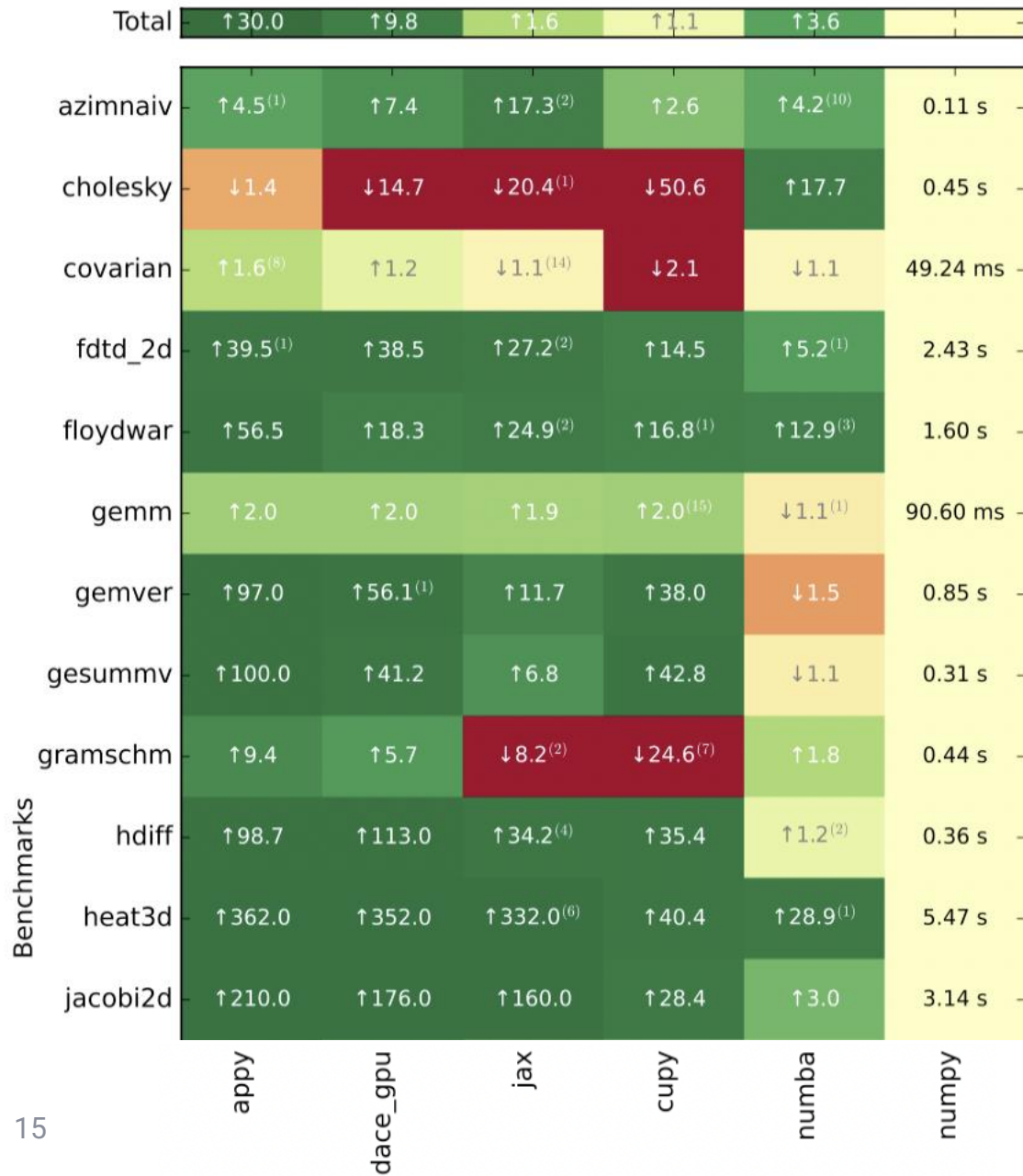
- NumPy
 - Rightmost column shows absolute runtime
- Other frameworks: speedups/slowdown relative to NumPy
 - Acknowledgment: visualization script from npbench (ETH)
 - Up arrow indicates speedup (from light green to dark green)
 - Down arrow indicates slowdown (from orange to red)
- Summary of APPy's 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

	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	
fddtd_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	
		↑30.0	↑9.8	↑1.6	↑1.1	↑3.6	

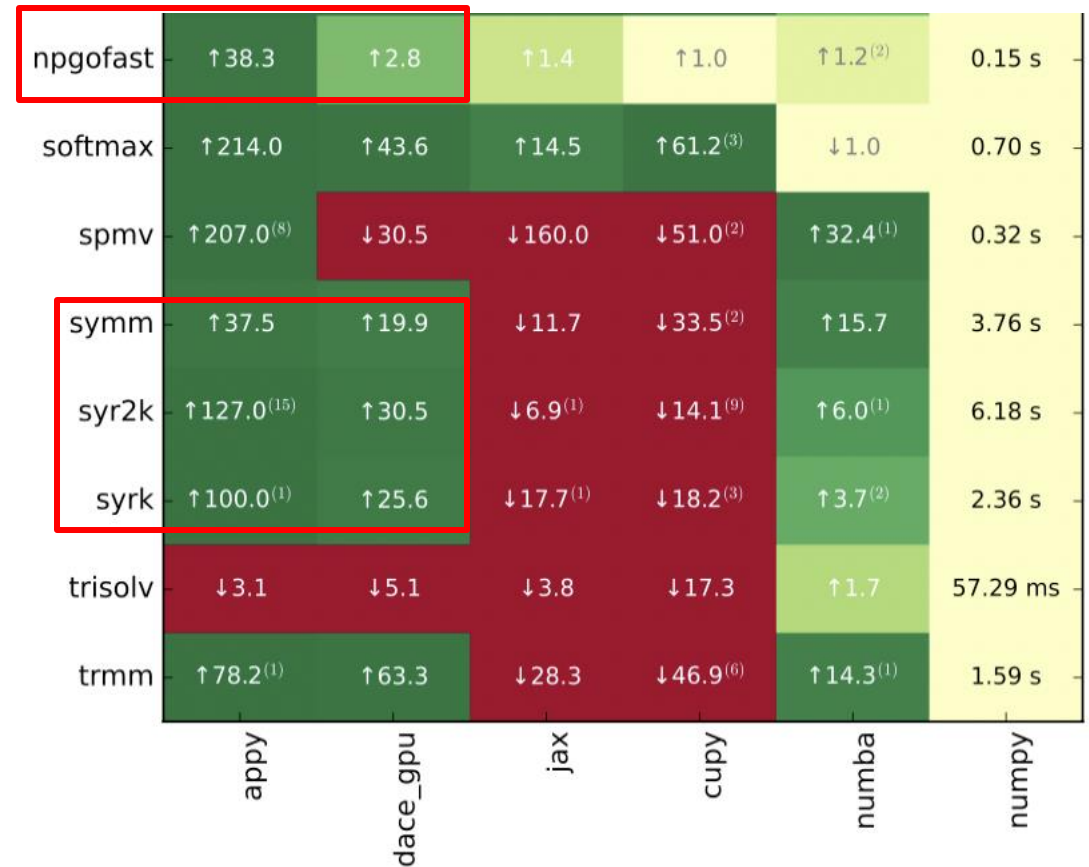
Benchmarks

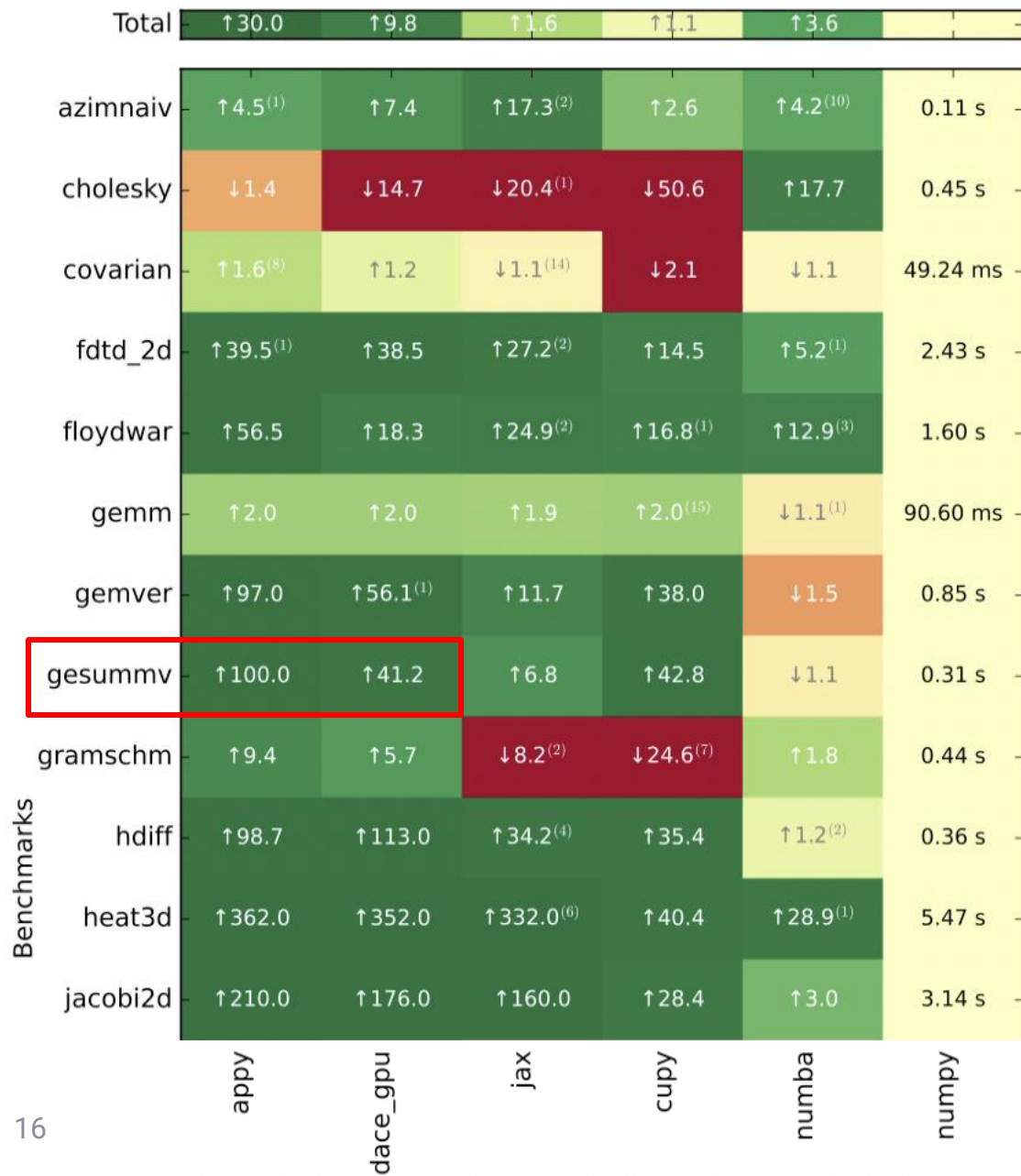
This work

appy
dace_gpu
jax
cupy
numba
numpy

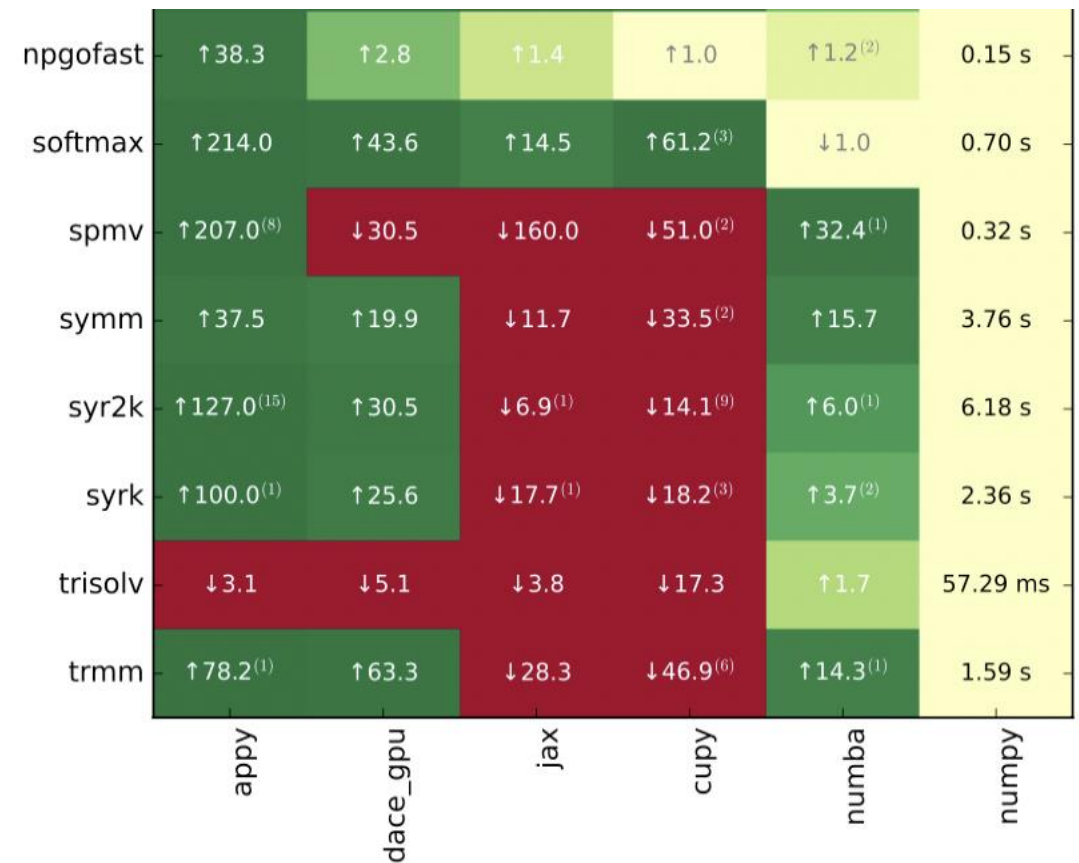


APPy is faster than DaCe due to more parallelism achieved





APPy is faster than DaCe due to more operator fusion and better locality



GitHub page

The screenshot displays the GitHub interface for the repository `habanero-lab/APPy`. At the top, there is a navigation bar with links for Product, Solutions, Open Source, and Pricing, along with a search bar and Sign in/Sign up buttons. Below this, the repository name and public status are shown, along with buttons for Notifications, Fork (2), and Star (19). The main navigation tabs include Code, Issues, Pull requests, Actions, Projects, Security, and Insights. The repository is currently on the `main` branch, with 1 branch and 0 tags. A search bar for files and a Code button are also present. The file list shows the following items:

File	Commit Message	Time
app	updated default block size for both simd directive and tenso...	2 months ago
examples	updated 08-spm	2 months ago
.gitignore	added compiler files, and added gitignore	last year
LICENSE	Create LICENSE	3 months ago
README.md	Update README.md	2 months ago
setup.py	added black as a required package	2 months ago

The README section is visible, showing the project description: "APPy (Annotated Parallelism for Python) enables users to parallelize generic Python loops and tensor expressions for execution on GPUs by adding OpenMP-like compiler directives (annotations) to Python code. With APPy, parallelizing a Python for loop on the GPU can be as simple as adding a `#pragma parallel for` before the loop, like the following:"

On the right side, the "About" section provides a brief description: "APPy (Annotated Parallelism for Python) enables users to annotate loops and tensor expressions in Python with compiler directives akin to OpenMP, and automatically compiles the annotated code to GPU kernels." Below this, there are links for Readme, MIT license, Activity, Custom properties, 19 stars, 2 watching, 2 forks, and Report repository. The "Releases" section shows "No releases published".

[GitHub - habanero-lab/APPy](https://github.com/habanero-lab/APPy)