APPy: Annotated Parallelism for Python on GPUs

Tong Zhou, Jun Shirako and Vivek Sarkar

Georgia Tech

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

Georgia Tech

A quick example of APPy

- An ordinary loop-based SpMV implementation in Python

```
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):
        y[i] = 0.0
        for j in range(A_row[i], A_row[1+i]):
            cols = A_col[j]
            y[i] += A_val[j] * x[cols]
    return y
```

A quick example of APPy

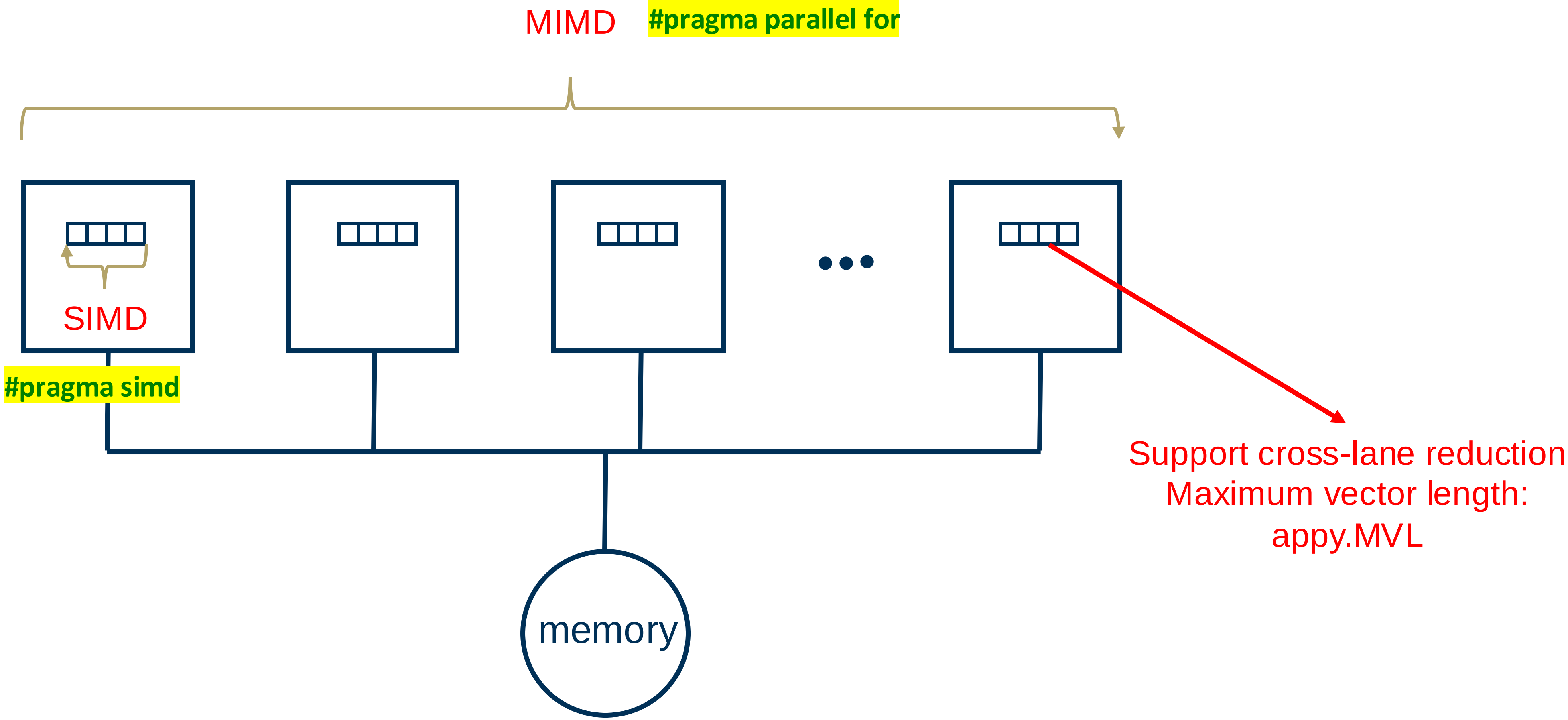
- An ordinary loop-based SpMV implementation in Python

```
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):
        y[i] = 0.0
        for j in range(A_row[i], A_row[1+i]):
            cols = A_col[j]
            y[i] += A_val[j] * x[cols]
    return y
```

- Ordinary SpMV parallelized with APPy

```
@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
```

Abstract machine model: a multi-vector processor



Support atomic update to memory locations

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}

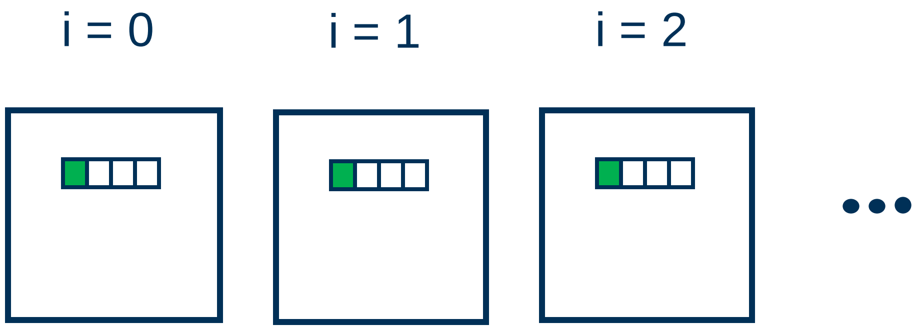
- Difference from OpenMP codegen
  - OpenMP directly exposes the parallelism hierarchy of the GPUs and requires more complicated pragmas to generate GPU code
  - OpenMP does not recognize and compile tensor expressions

Vector addition with APPy

Software

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

Hardware
(abstract)



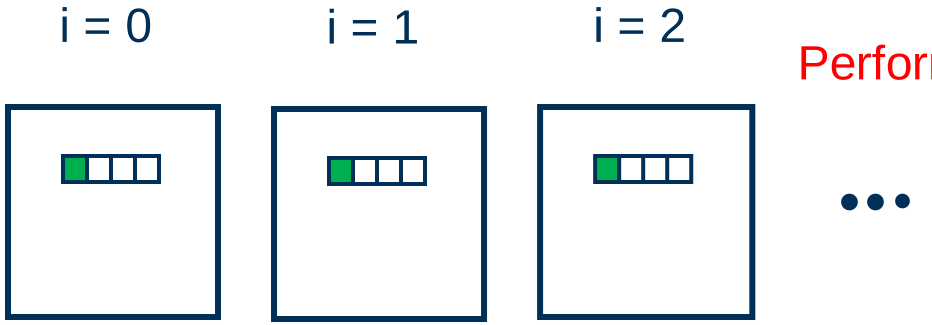
N *workers* launched

Utilize both layers of parallelism: parallel for + simd

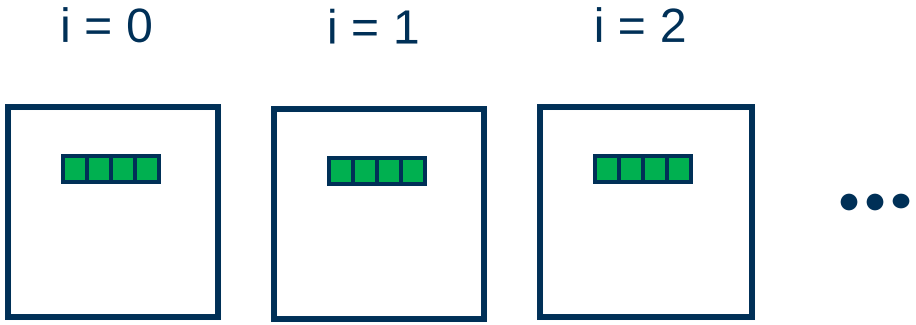
```python
@appy.jit
def vector_add(a, b, c, N):
    #pragma parallel for
    for i in range(N):
        c[i] = a[i] + b[i]
```

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

Performance boost!



N workers launched



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

Two ways to utilize vectorization

Annotate the loop with #pragma simd

```
@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
```

Write 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 strip-mined loops with operator fusion

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

Tensor expressions can be parallelized too

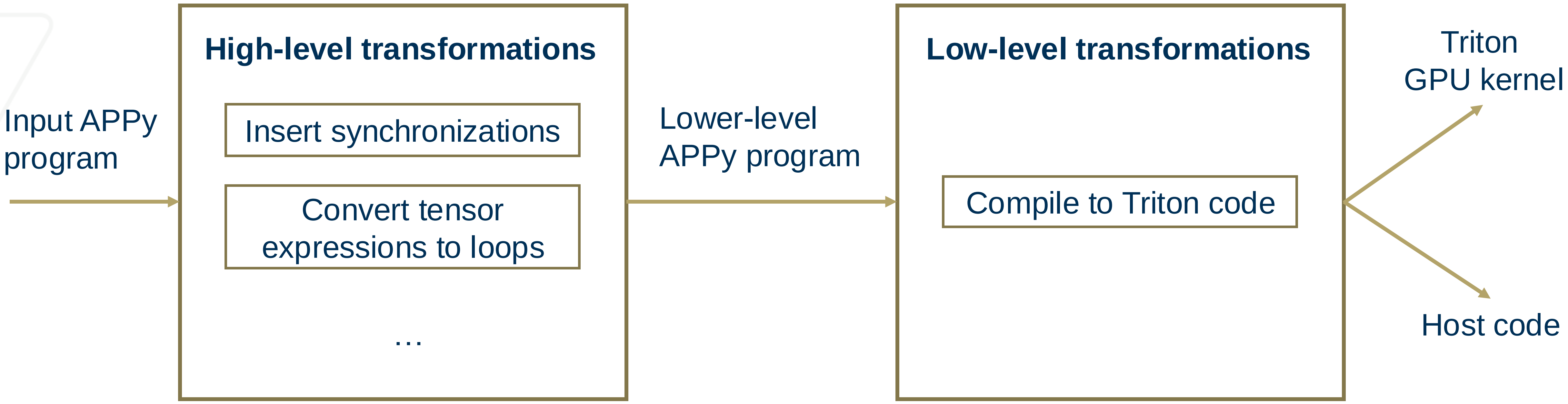
```
@appy.jit(auto_simd=True)
def gemv(alpha, A, x):
    M, N = A.shape
    #pragma :M=>parallel :N=>reduction(sum:y)
    y[:M] = mv(alpha * A[:M, :N], x[:N])
```

Compiler generated explicit loop

```
@appy.jit
def gemv_transformed(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])
```

Implementation

- All transformation passes are Python AST based



A code generation example

```
@appy.jit(auto_simd=True)
def gemv(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 gemv(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 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)
```

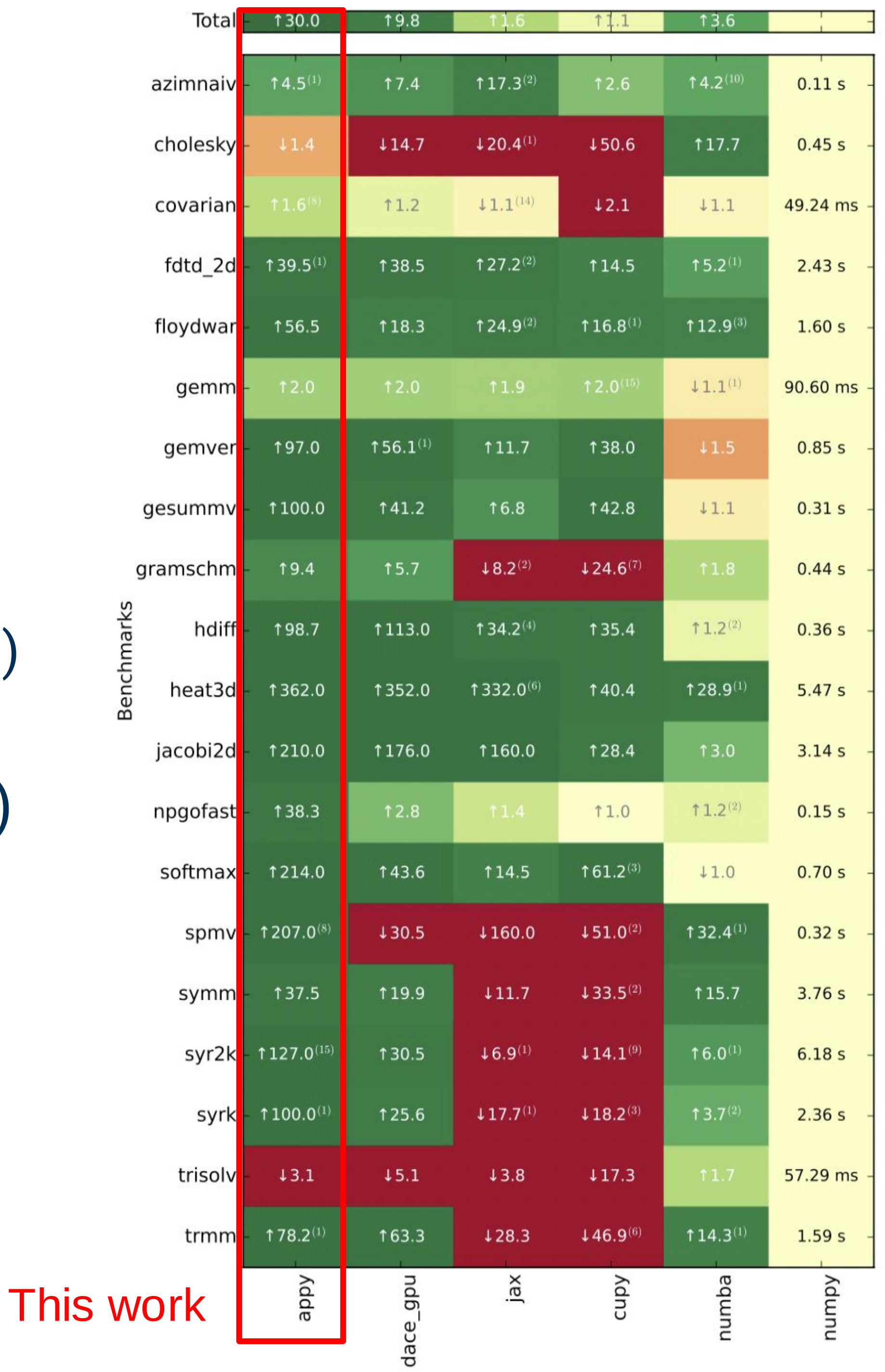
Performance evaluation

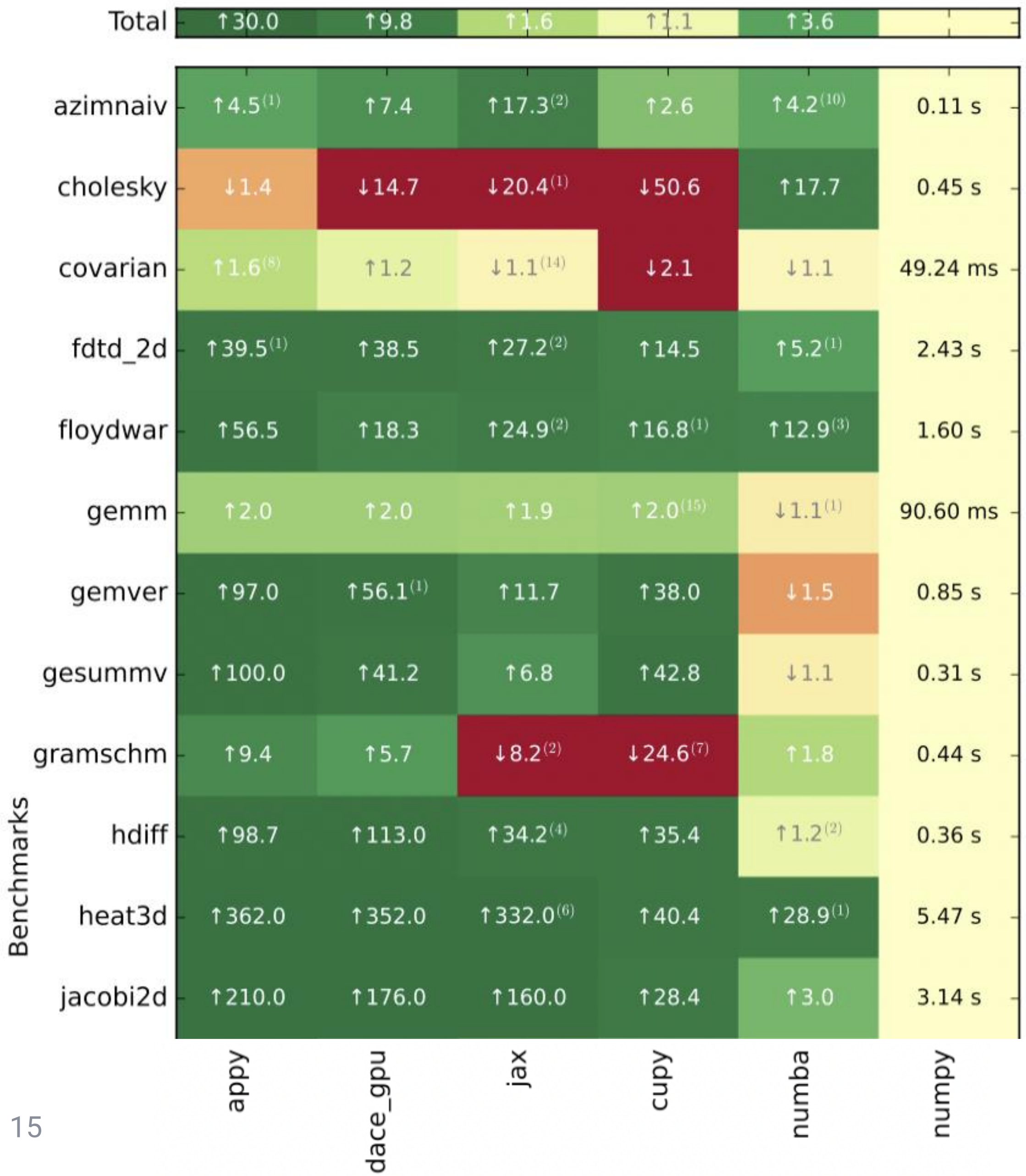
- CPU: Ryzen 7 5800X
  - 8 cores
  - Cache sizes
    - L1: 32K, L2: 512K, L3: 32M
- GPU: RTX 3090
  - 10496 cuda cores, 82 SMs
  - Cache sizes
    - L1: 128K, L2: 6M
- Benchmarking methodology
  - Each benchmark is run 10 times and report median
  - Each benchmark run is ~ 1 second
- Comparisons
  - 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

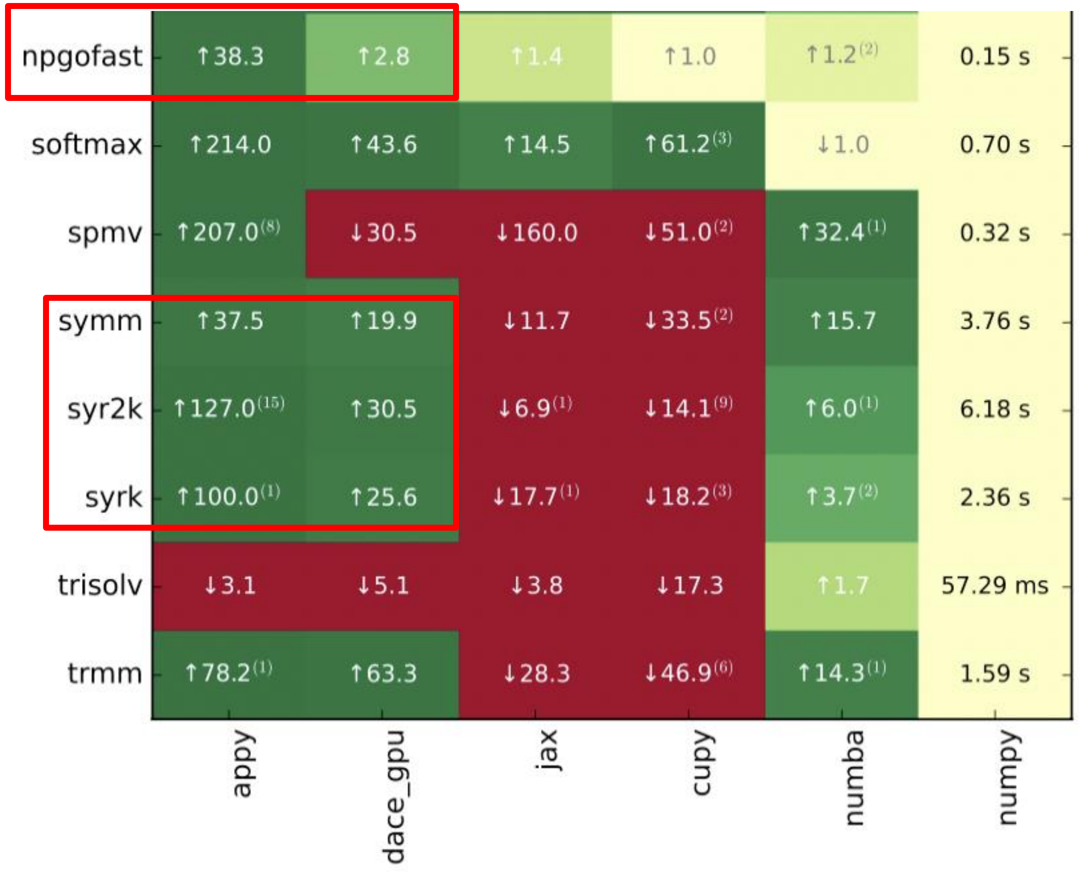
Performance results

- 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

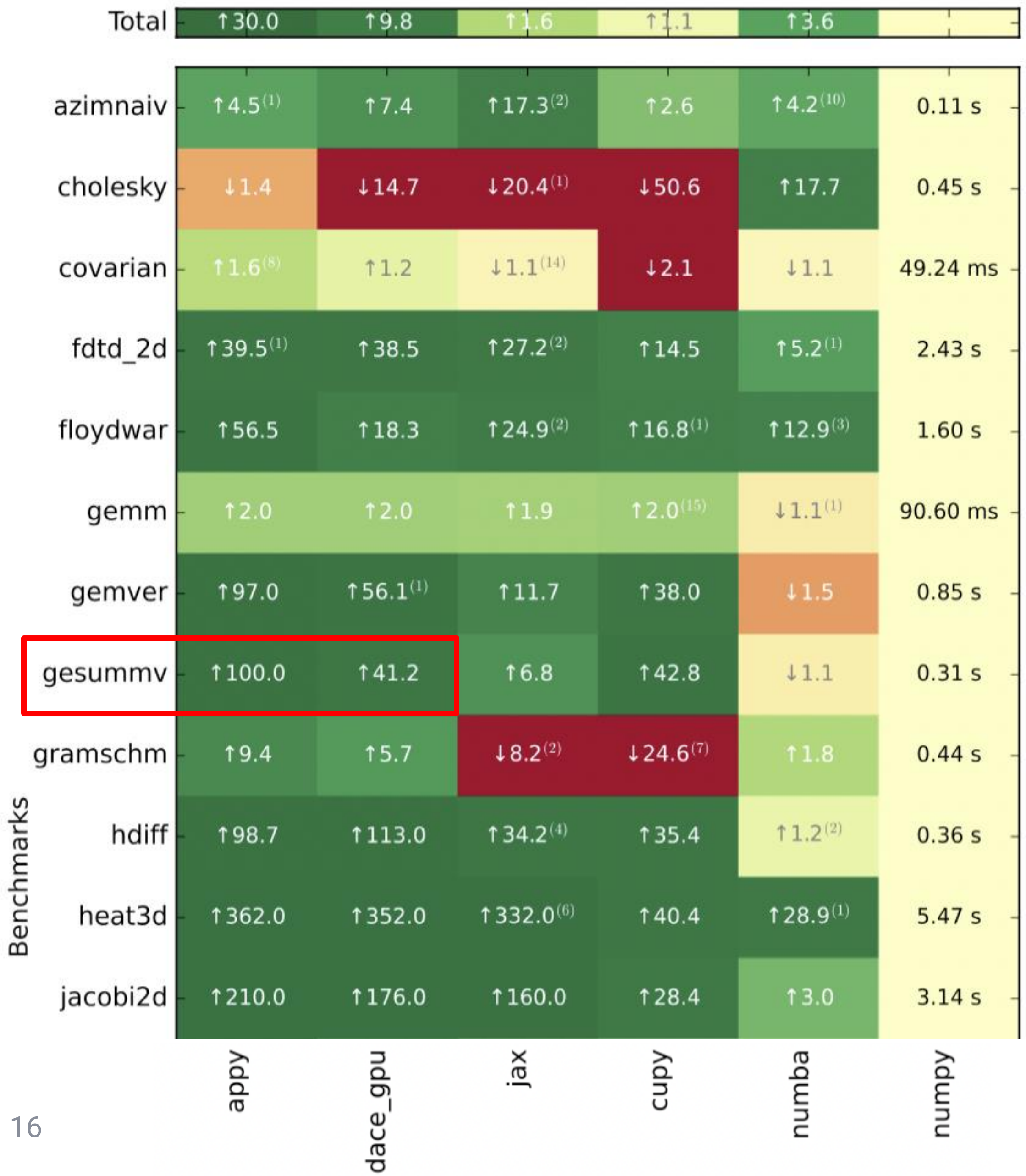


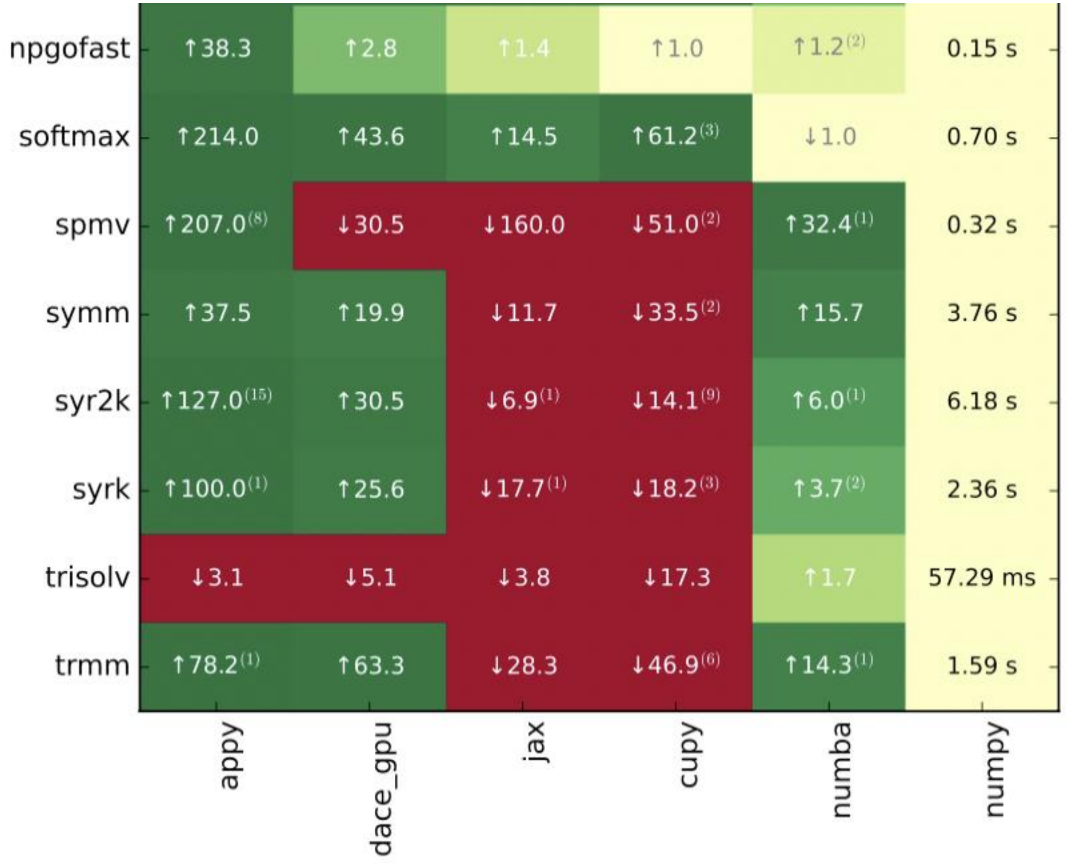


APPy is faster than DaCe due to more parallelism achieved

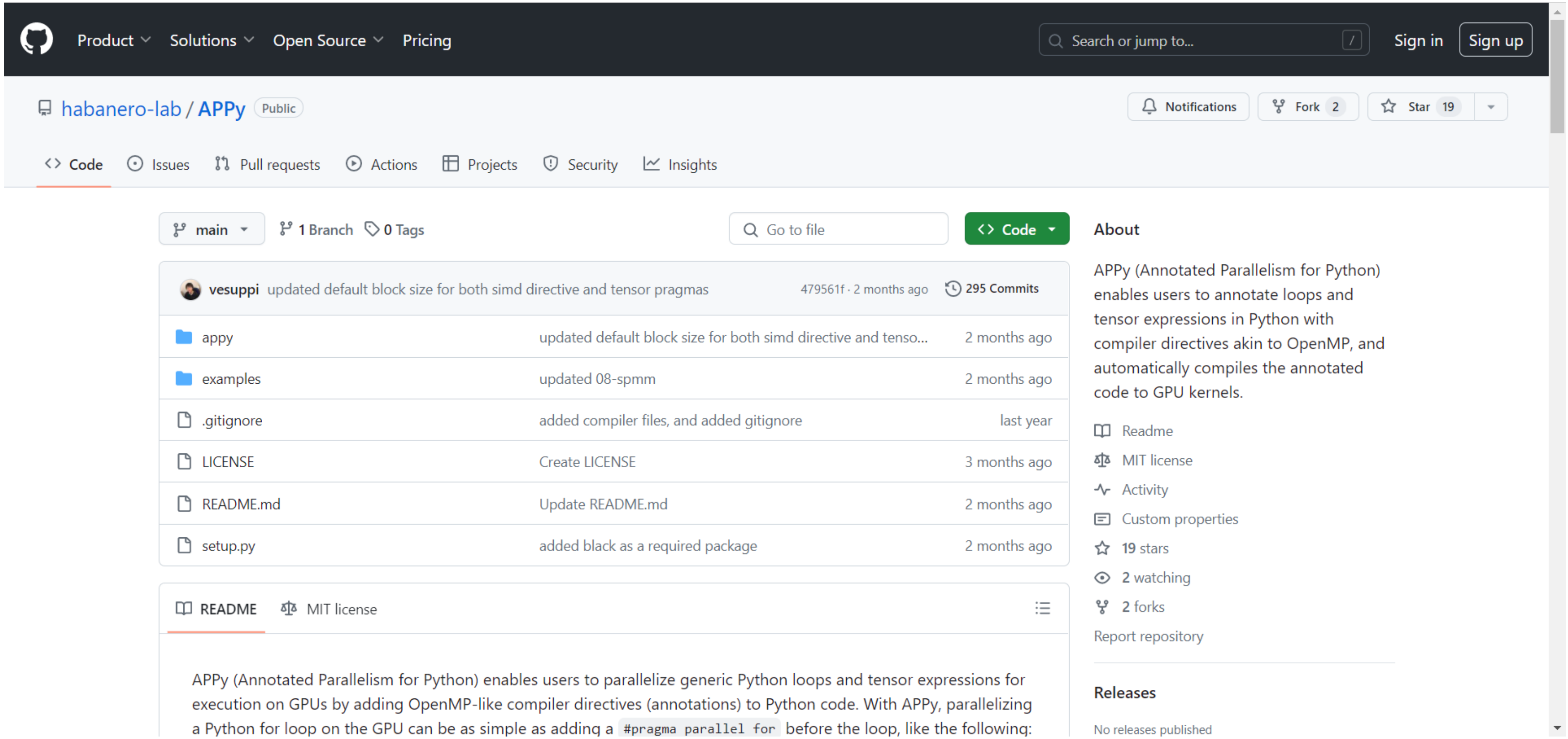


APPy is faster than DaCe due to more operator fusion and better locality





GitHub page



GitHub - habanero-lab/APPy

Georgia Tech