Automating Massively Parallel Heterogeneous Computing for Python Programmers

Vivek Sarkar
School of Computer Science
College of Computing
Georgia Institute of Technology

ESPM2 2020 Invited Talk

Outline

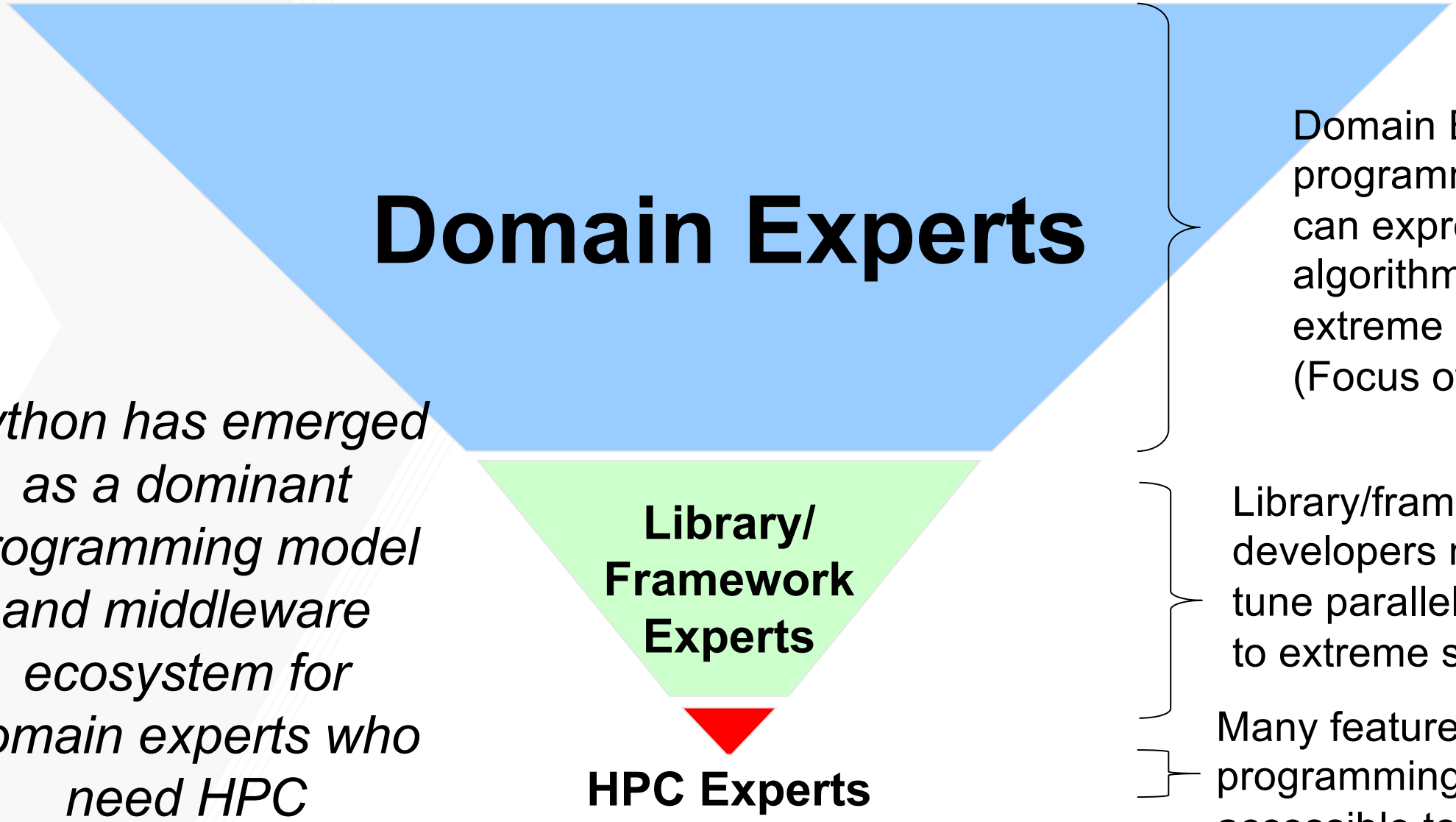
1. Motivation

2. Intrepydd -- an AOT tool chain for optimization & parallelization of Python programs

3. AMPHC – extend Intrepydd for Automating Massively Parallel Heterogeneous Computing using Python

4. Conclusions and Next Steps

Inverted Pyramid of HPC Developers



Python has emerged as a dominant programming model and middleware ecosystem for domain experts who need HPC

Domain Experts need programming models that can express new parallel algorithms for future extreme scale systems (Focus of today's talk)

Library/framework developers need to map and tune parallel algorithms on to extreme scale hardware

Many features in today's HPC programming models are only accessible to HPC experts

Extreme Scale = Extreme Heterogeneity

Heterogeneity crisis!

Compute capability and complexity is increasing at the intra-node level, while inter-node scaling is flat or declining

Significant challenge for Domain Experts to deal with this complexity at the Python level

Power wall (End of Dennard's scaling) — ~2005

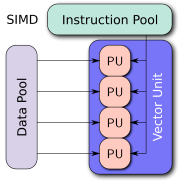
Utilization wall (Dark silicon) — ~2010

Locality wall — ~2015

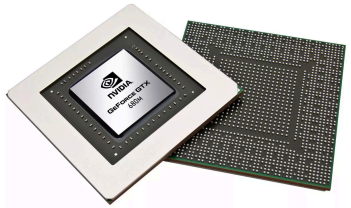
End of Moore's law

Multi-core processors

SIMD units

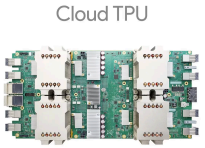


GPGPUs

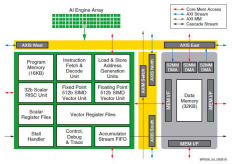


Light-weight many-core processors

Domain-specific accelerators

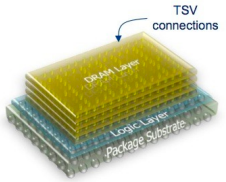


Specialized SIMD Units



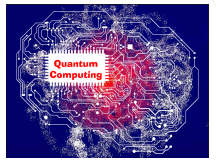
Migratory Architectures

3D stacked Processors



.....

Quantum Architectures



Neuromorphic computing

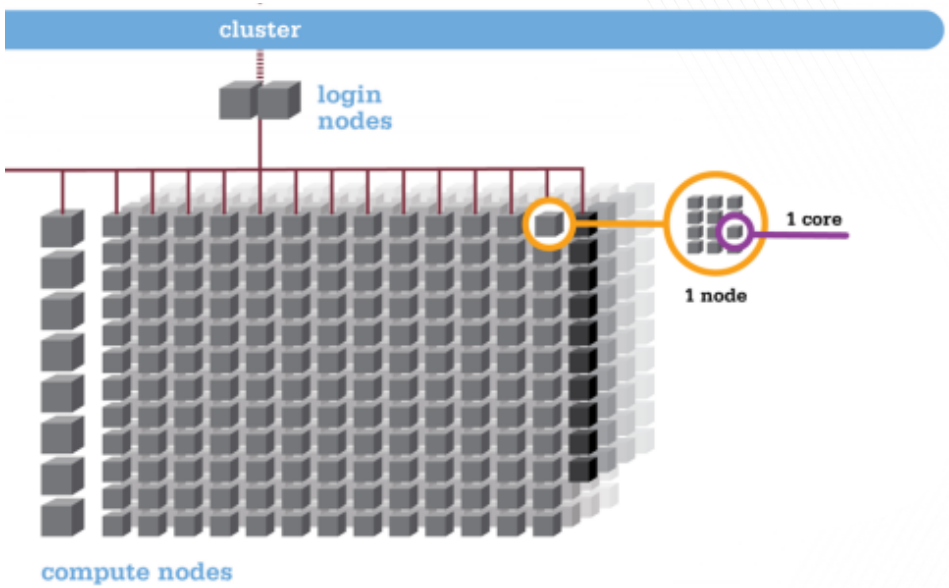
Georgia Tech
CREATING THE NEXT

Increasing Complexity with Increasing Parallelism for Python Programmers

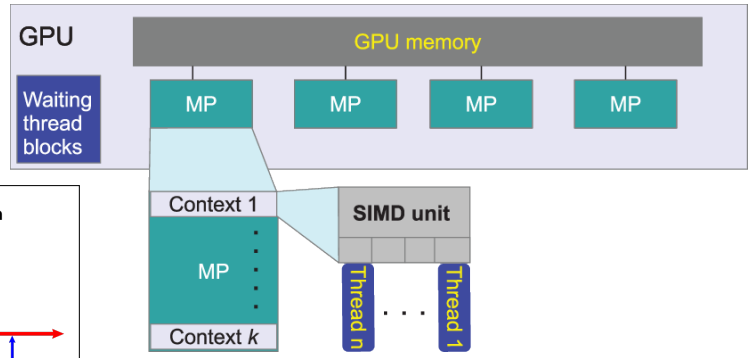
Parallelism cliffs: step function in programming and tuning efforts needed to enable applications to exploit the next stage of increasing parallelism

MPP Clusters

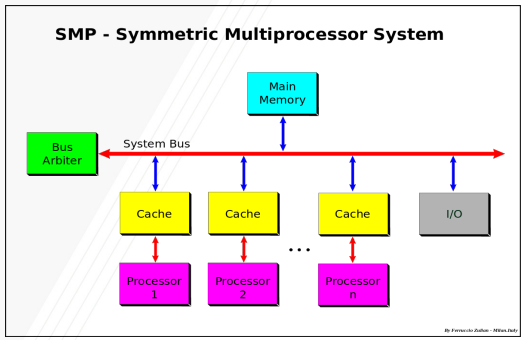
… Extreme Heterogeneity …



Single/multiple GPUs



Multicore



Single CPU

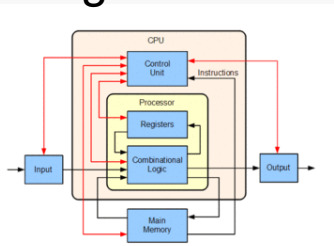


Image sources:
https://en.wikipedia.org/wiki/File:ABasicComputer.gif
https://en.wikipedia.org/wiki/Symmetric_multiprocessing
https://www.researchgate.net/figure/Hierarchical-hardware-parallelism-in-a-GPU_fig1_262176632
https://www.osc.edu/book/export/html/2782

Georgia Tech
CREATING THE NEXT

Outline

1. Motivation

2. Intrepydd -- an AOT tool chain for optimization & parallelization of Python programs

3. AMPHC – extend Intrepydd for Automating Massively Parallel Heterogeneous Computing using Python

4. Conclusions and Next Steps

Programming Systems for Data Science Applications: Current Approaches

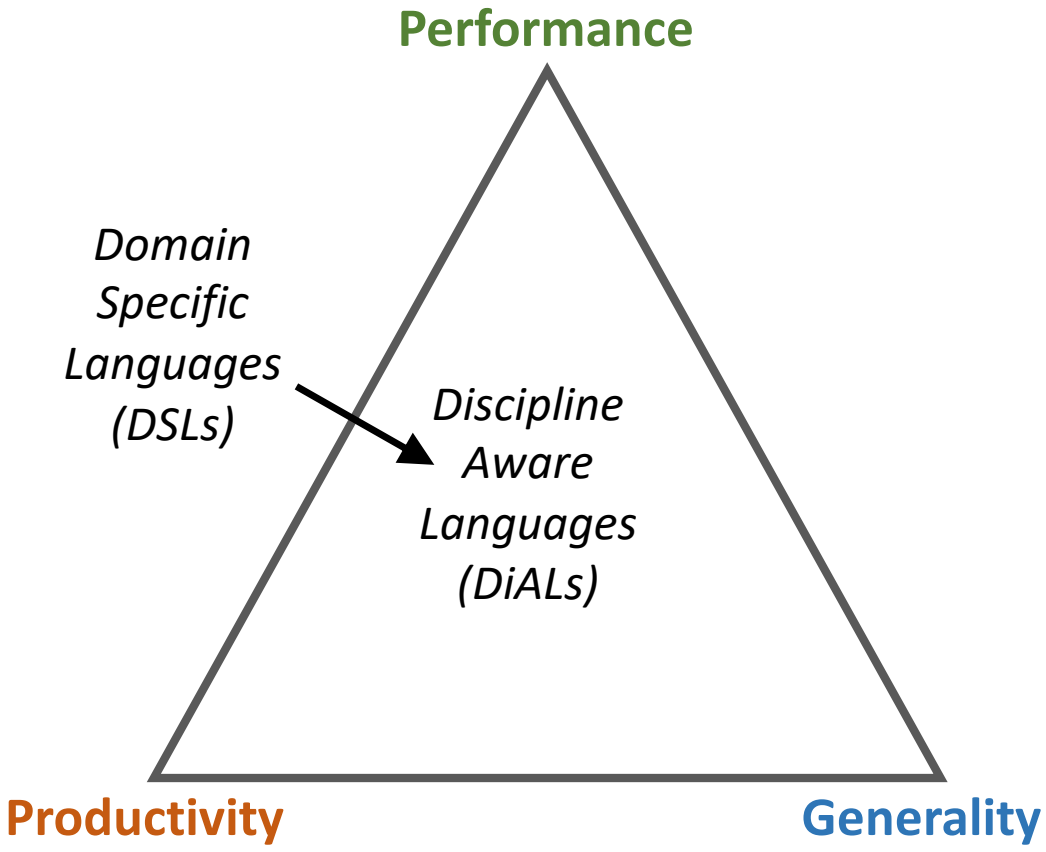
1. Augment general-purpose high-productivity languages (HPLs) with high-performance libraries
   - Examples: Python/Julia/Matlab with NumPy/SciPy/CuPy/PCT
   - Challenge: Library APIs may not adapt well to needs of new applications
2. Domain Specific Languages (DSLs) for target domains
   - Examples: TensorFlow for machine learning, Halide for image processing
   - Challenge: need an approach that includes multiple DSLs, as well as an HPL

Motivation for our work: combine the benefits of HPLs and DSLs in a single Discipline-Aware Language (DiAL) for Data Science

”Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels”. T.Zhou, J.Shirako, A.Jain, S.Srikanth, T.M.Conte, R.Vuduc, V.Sarkar. SPLASH Onward! ’20 conference.

Intrepydd Discipline-Aware Language

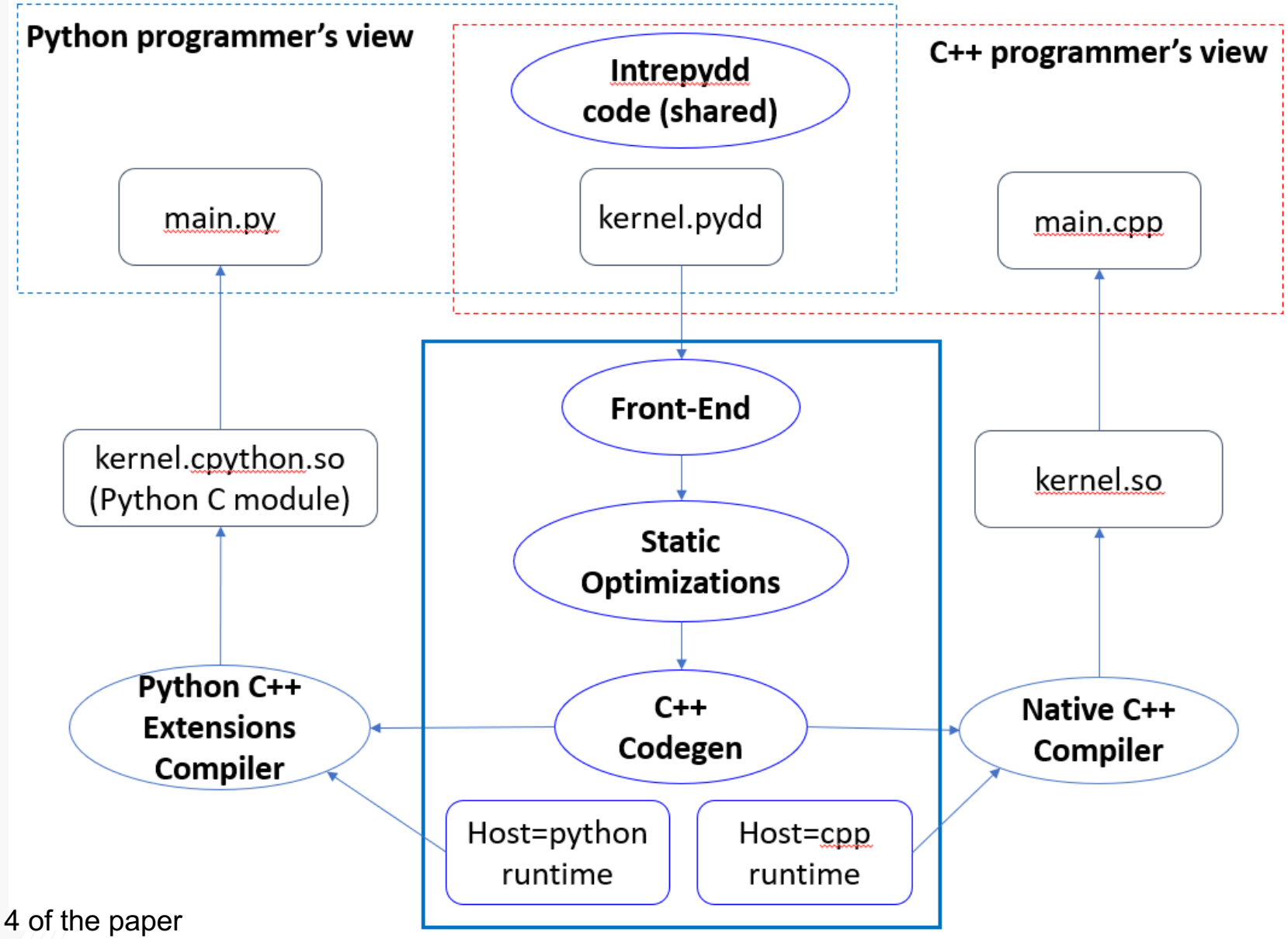
- Intrepydd programming system delivers performance, productivity, portability for kernels that span different data science domains
- Based on Python syntax for synergy with data science applications
- Simplifies programming for heterogeneous and post-Moore hardware
- Differs from Python in significant ways
  1. Language for writing computational kernels, not complete applications
  2. Designed for Ahead-Of-Time (AOT) compilation with high-level compiler optimizations and C++ code generation



Many reasons to not attempt AOT compilation and parallelization for Python …

- Python is designed for interactive programming, with a heavy use of native libraries for performance
- Dynamic typing
- Multithreading-unfriendly
  - The Global Interpreter Lock serializes computations
  - Not all extension modules support multithreading
- Multiprocessing module
  - Requires explicit launching of processes/jobs
- GPU programming options
  - Use the CuPy library
  - Write GPU native code, and link it in as a Python module
- Cluster programming options
  - Message passing, e.g., MPI
  - Distributed task runtimes, e.g., Ray

.. which is exactly why we decided to do it!



Details in Section 4 of the paper

Intrepydd Language Definition (Summary)

- Data Types
  - Boolean
  - Numeric: int32, int64, float32, float64
  - Collections: List(type), Array(type), SparseMat(type), Dict(type), Heap(type)
- Statements:
  - Function definitions, with types for parameters and return values
  - Assignment, Call/Return Statements
  - Control Flow Statements
  - Parallel Statements
- Type Inference
  - Static type inference is performed using types for parameters and return values
- Operators
  - Arithmetic: +, -, *, /, //, **
  - Comparison: ==, !=, <, >, <=, >=
  - Logical: and, or, not
  - Membership: in
- Library Functions
  - Reductions, unary/binary functions, dense/sparse linear algebra functions

Details in Section 3 of the paper

Sinkhorn WMD Kernel in Python, Intrepydd, and Julia

Python

```
1. def kernel(K,                M,
2.            X,                R,
3.            C,
4.
5.            ncols,            max_iter)
6.
7.    it = 0
8.    while it < max_iter:
9.        U = 1.0 / X
10.       V = C.multiply(1 / (K.T @ U))
11.       X = ((1/R) * K) @ V.tocsc()
12.       it += 1
13.   U = 1.0 / X
14.   V = C.multiply(1 / (K.T @ U))
15.   return (U * ((K * M) @ V)).sum(0)
```

Algorithm 1 Computation of $\mathbf{d} = [d_M^\lambda(r, c_1), \cdots, d_M^\lambda(r, c_N)]$, using Matlab syntax.

Input $M, \lambda, r, C := [c_1, \cdots, c_N]$.
$I = (r > 0); r = r(I); M = M(I, :); K = \exp(-\lambda M)$
$u = \texttt{ones}(\texttt{length}(r), N)/\texttt{length}(r);$
$\tilde{K} = \texttt{bsxfun}(\texttt{@rdivide}, K, r)$ % equivalent to $\tilde{K} = \textbf{diag}(1./r)K$
while u changes or any other relevant stopping criterion **do**
  $u = 1./(\tilde{K}(C./(K'u)))$
end while
$v = C./(K'u)$
$\mathbf{d} = \texttt{sum}(u. * ((K. * M)v))$ *(NeurIPS'13, #4927)*

```
8.   while it < max_iter:
9.       U = 1.0 / X
10.      V = C.multiply(1 / (K.T @ U))
11.      X = ((1/R) * K) @ V.tocsc()
12.      it += 1
13.  U = 1.0 / X
14.  V = C.multiply(1 / (K.T @ U))
15.  return (U * ((K * M) @ V)).sum(0)
```

Sinkhorn WMD Kernel in Python, Intrepydd, and Julia

Intrepydd

```
1. def kernel(K,                M,
2.         X,                R,
3.         C,
4.
5.         ncols,              max_iter)
6.
7.    it = 0
8.    while it < max_iter:
9.        U = 1.0 / X
10.       V = C.multiply(1 / (K.T @ U))
11.       X = ((1/R) * K) @ V.tocsc()
12.       it += 1
13.   U = 1.0 / X
14.   V = C.multiply(1 / (K.T @ U))
15.   return (U * ((K * M) @ V)).sum(0)
```

Sinkhorn WMD Kernel in Python, Intrepydd, and Julia

Intrepydd

```
1. def kernel(K: Array(float64, 2),   M: Array(float64, 2),
2.            X: Array(float64, 2),   R: Array(float64, 2),
3.            C,
4.
5.            ncols: int32,           max_iter: int32)
6.
7.    it = 0
8.    while it < max_iter:
9.        U = 1.0 / X
10.       V = C.multiply(1 / (K.T @ U))
11.       X = ((1/R) * K) @ V.tocsc()
12.       it += 1
13.   U = 1.0 / X
14.   V = C.multiply(1 / (K.T @ U))
15.   return (U * ((K * M) @ V)).sum(0)
```

Sinkhorn WMD Kernel in Python, Intrepydd, and Julia

Intrepydd

```
1. def kernel(K: Array(float64, 2),   M: Array(float64, 2),
2.            X: Array(float64, 2),   R: Array(float64, 2),
3.            data: Array(float64, 2),
4.            idx: Array(int32, 1),   ptr: Array(int32, 1),
5.            ncols: int32,           max_iter: int32)
6.    C = csr_to_spm(data, idx, ptr, ncols)
7.    it = 0
8.    while it < max_iter:
9.        U = 1.0 / X
10.       V = C.multiply(1 / (K.T @ U))
11.       X = ((1/R) * K) @ V.tocsc()
12.       it += 1
13.   U = 1.0 / X
14.   V = C.multiply(1 / (K.T @ U))
15.   return (U * ((K * M) @ V)).sum(0)
```

Sinkhorn WMD Kernel in Python, Intrepydd, and Julia

Intrepydd

```
1. def kernel(K: Array(float64, 2),   M: Array(float64, 2),
2.            X: Array(float64, 2),   R: Array(float64, 2),
3.            data: Array(float64, 2),
4.            idx: Array(int32, 1),   ptr: Array(int32, 1),
5.            ncols: int32,           max_iter: int32)
6.     C = csr_to_spm(data, idx, ptr, ncols)
7.     it = 0
8.     while it < max_iter:
9.         U = 1.0 / X
10.        V = C.multiply(1 / (K.T @ U))
11.        X = spmm_dense((1/R) * K, V)
12.        it += 1
13.    U = 1.0 / X
14.    V = C.multiply(1 / (K.T @ U))
15.    return (U * spmm_dense(K * M, V)).sum(0)
```

Sinkhorn WMD Kernel in Python, Intrepydd, and Julia

Intrepydd (future version)

```
1. def kernel(K: Array(float64, 2),   M: Array(float64, 2),
2.            X: Array(float64, 2),   R: Array(float64, 2),
3.            C: SparseArray(float64, 2),
4.
5.            ncols: int32,           max_iter: int32)
6.
7.    it = 0
8.    while it < max_iter:
9.        U = 1.0 / X
10.       V = C * (1 / (K.T @ U))
11.       X = ((1/R) * K) @ V
12.       it += 1
13.   U = 1.0 / X
14.   V = C * (1 / (K.T @ U))
15.   return (U * (K * M, V)).sum(0)
```

Sinkhorn WMD Kernel in Python, Intrepydd, and Julia

Julia

```
1. function kernel(K::Array{Float64, 2}, M::Array{Float64, 2},
2.                 X::Array{Float64, 2}, R::Array{Float64, 2},
3.                 row::Array{Int32, 1}, col::Array{Int32, 1},
4.                 data::Array{Float64, 1},
5.                 m::int64, n::int64, max_iter::int64)
6.     C = sparse(row, col, data, m, n)
7.     it = 0
8.     while it < max_iter:
9.         U = 1.0 ./ X
10.        V = C .* (1.0 ./ (transpose(K) * U))
11.        X = ((1.0 ./ R) .* K) * V
12.        it += 1
13.    end
14.    U = 1.0 ./ X
15.    V = C .* (1.0 ./ (transpose(K) * U))
16.    return sum(U .* ((K .* M) * V))
```

Using Intrepydd from Jupter notebooks

- Example of using Intrepydd from a Jupyter notebook
- Intrepydd interoperates with standard tools, such as compilers, profilers and timers
- Intrepydd compilation (pyddc) involves:

1. translating Intrepydd to C++, which is relatively quick (under 0.5 seconds for the examples that we evaluated
2. compiling the generated C++ code, which incurs the usual overhead of invoking a C++ compiler (6 to 12 seconds for the examples that we evaluated)

In [14]:
```
%%writefile opt.pydd
# opt.pydd

def update_centers(k: int64, X: Array(float64, 2), y: Array(int64)) \
                -> Array(float64, 2):
    m = shape(X, 0) # type: int64
    d = shape(X, 1) # type: int64
    centers = zeros((k, d), float64())
    counts = zeros(k, int64())

    # Sum each coordinate for each cluster
    # and count the number of points per cluster
    for i in range(m):
        c = y[i] # type: int64
        counts[c] += 1
        for j in range(d):
            centers[c, j] += X[i, j]

    # Divide the sums by the number of points
    # to get the average
    for c in range(k):
        n_c = counts[c] # type: int64
        for j in range(d):
            centers[c, j] /= n_c
    return centers

# eof
```

```
Overwriting opt.pydd
```

In [15]:
```
!../pyddc opt.pydd # Compile using Intrepydd
```

In [16]:
```
import opt
update_centers = opt.update_centers
kmeans(points, k, starting_centers=points[[0, 187], :], max_steps=50, ve
rbose=True)
```

Georgia Tech
CREATING THE NEXT

Experimental Methodology

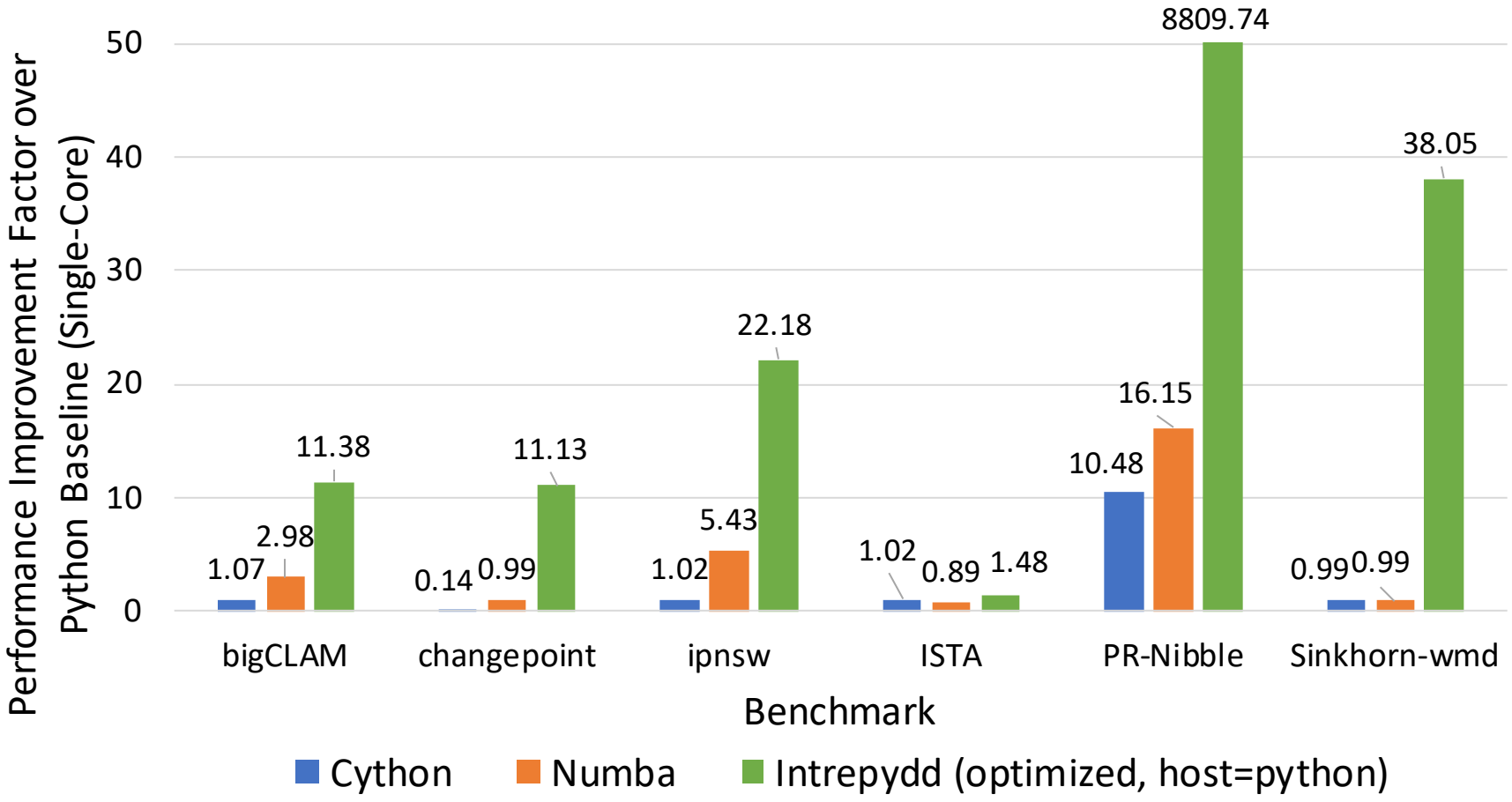
Benchmark Applications

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

Testbed

- Dual Intel Xeon Silver 4114 CPU @ 2.2GHz with 192GB of main memory and hyperthreading disabled
- Each benchmark run 11 times and average of later 10 runs reported
- Standard deviation between runs [0.06-3.6] percent of average
- Baseline idiomatic Python 3.7.6

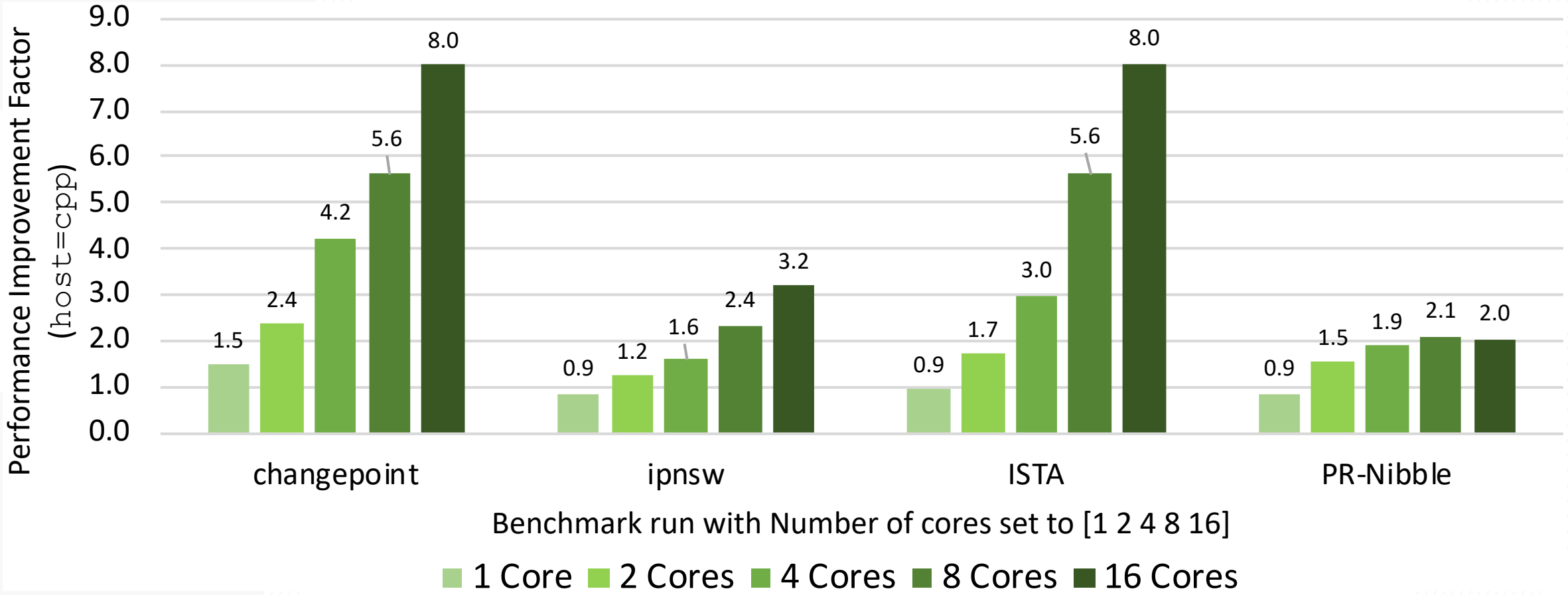
Intrepydd Single Core Performance



Intrepydd offers 20.07x speedup on average (harmonic mean) over baseline Python

Details in Section 6.3 of the paper

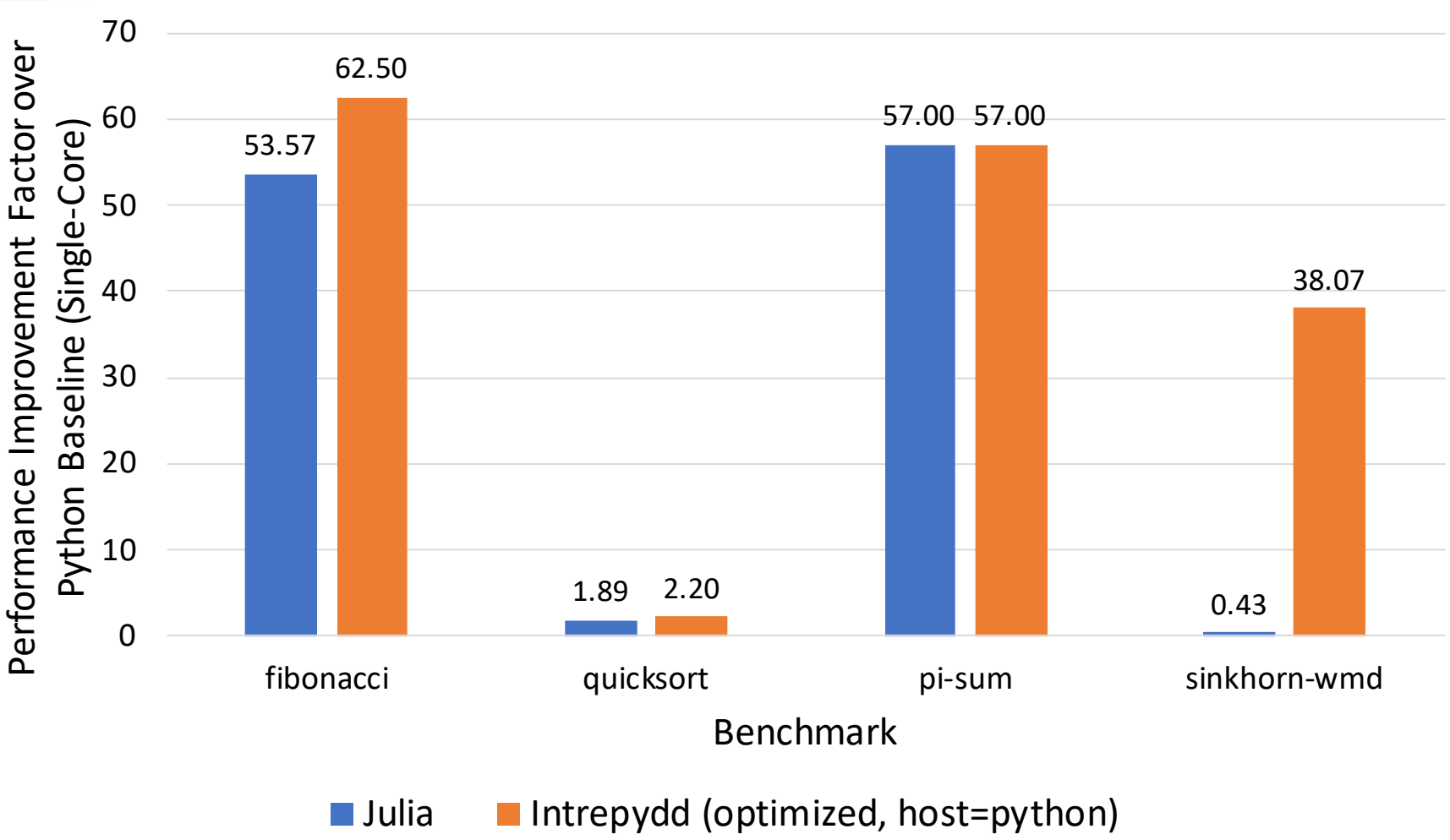
Multicore Scalability with user-specified pfor loops (improvement is for host=cpp relative to host=python)



Georgia Tech
CREATING THE NEXT

Details in Section 6.4 of the paper

Comparison with Julia



Intrepydd offers 88.5x speedup over Julia for Sinkhorn-wmd

Details in Section 6.5 of the paper

Performance benefits from Intrepydd to C++ translation

Intrepydd source code

```
def foo(xs: Array(double, 2)) -> double:
    ...
    for i in range(shape(xs, 0)):
        for j in range(shape(xs, 1)):
            a = 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) {
            a = xs.data()[i*pydd::shape(xs, 1)+j];
            ...
```

Details in Section 4 of paper

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**	**+LICM OPT**	**+Array OPT**	**+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	0.006
sinkhorn-wmd	47.612	47.395	1.225	1.220

Details in Sections 5 and 6.3 of paper

Code Optimization:
LICM and Dense/Sparse Array Operator Fusion

```
it = 0
while it < max_iter:
    u = 1.0 / x

    v = c.spm_mul(1 / (K.T @ u))

    x = spmm_dense((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

Georgia Tech
CREATING THE NEXT

Code Optimization: Array Allocation

```
it = 0
while it < max_iter:
    a = b + c   # all 2D arrays
    d = zeros_like(a)
    b = …       # b is not a loop invariant
    …
    it += 1
```

```
a = empty_like(b)
d = empty_like(a)
while it < max_iter:
    add(b, c, out=a)
    fill(d, 0)
    b = …
    …
    it += 1
```

Intrepydd source code

Transformed code

Georgia Tech
CREATING THE NEXT

Summary

- Intrepydd programming system
  - General Python-based semantics for data scientists
  - High performance through AOT compilation and high-level optimizations
  - High portability through support of Python and C++ host programs
    - Includes mapping to post-Moore accelerators and architectures
- Significant single-core performance improvements over Python
  - 11.1x - 8809.5x for non-library-dominated benchmarks
  - 1.5x improvement for a library-dominated benchmark
- Demonstration of multicore scalability with user-specified parallelism
- Next steps
  - Extend to Python-friendly distributed heterogenous runtime frameworks
  - Complete implementations for async, finish, isolated statements
  - Complete support for post-Moore accelerators and architectures

Outline

1. Motivation

2. Intrepydd -- an AOT tool chain for optimization & parallelization of Python programs

3. AMPHC – extend Intrepydd for Automating Massively Parallel Heterogeneous Computing using Python

4. Conclusions and Next Steps

AMPHC: Automating Massively Parallel Heterogeneous Computing (part of DARPA PAPPA program)

PI: Prof. Vivek Sarkar
School of Computer Science

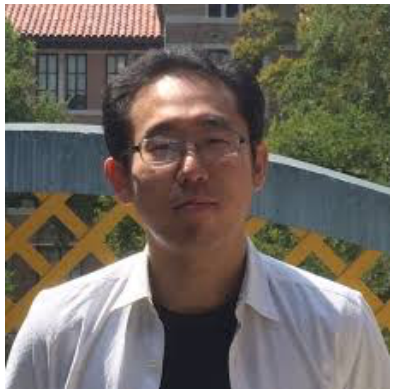
Co-PI: Prof. Taesoo Kim
School of Computer Science

Co-PI: Dr. Sukarno Mertoguno
Georgia Tech Research Institute

Co-PI: Prof. Alexey Tumanov
School of Computer Science

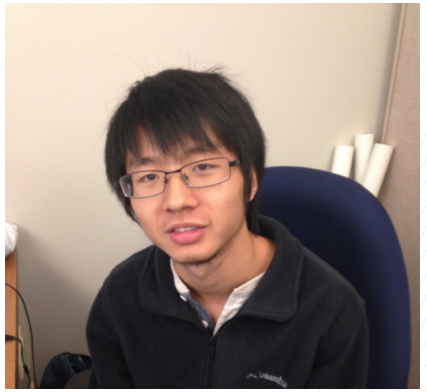
Shelby Allen
Georgia Tech Research Institute

Barry L Drake
Georgia Tech Research Institute



Dr. Jun Shirako
School of Computer Science

Mingyu Guan
School of Computer Science



Tong Zhou
School of Computer Science

Overview of our approach

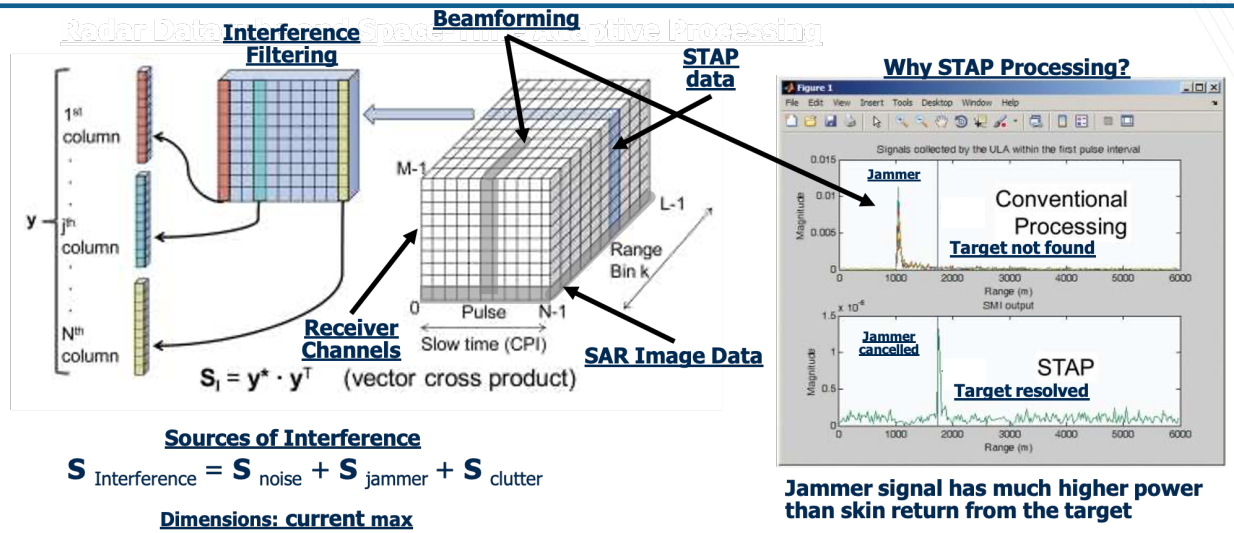
- **Programming Model extensions**
  - *Python:* standard Python with common ("messy") coding patterns
  - *Intrepydd-AMPHC:* partially typed ("clean") subset of Python suitable for AOT code generation
    - can be written directly or auto-generated from messy Python via "distillation"
- **New compiler technologies**
  - *Distillation* (input: Python code, output: distilled Python + Intrepydd-AMPHC code)
    - clean up code by removing serialization bottlenecks
    - identify types, match computational patterns to known libraries
  - *Annealing* (input: Intrepydd-AMPHC code, output: distributed heterogeneous code)
    - automatic two-level parallelization: 1) distributed Python wrapper code for execution on Ray-AMPHC runtime, and 2) intra-node heterogeneous native code for execution on HClib-AMPHC runtime
- **Runtime extensions**
  - *Ray-AMPHC:* extensions to Ray runtime to support compiler-generated distributed code with futures, actors, and heterogeneity
  - *HClib-AMPHC:* Extensions to Habanero-C runtime library (HClib) to support compiler-generated intra-node parallel code that interoperates with Ray-AMPHC

Focus Application Domains for AMPHC project

1. Radar Datacube Process Flow: Space-Time Adaptive Processing (STAP)

2. Scalable Distributed Data Analytics with Dataframes

NOTE: Both application domains currently lack productive programming systems for Massively Parallel Heterogeneous Computing



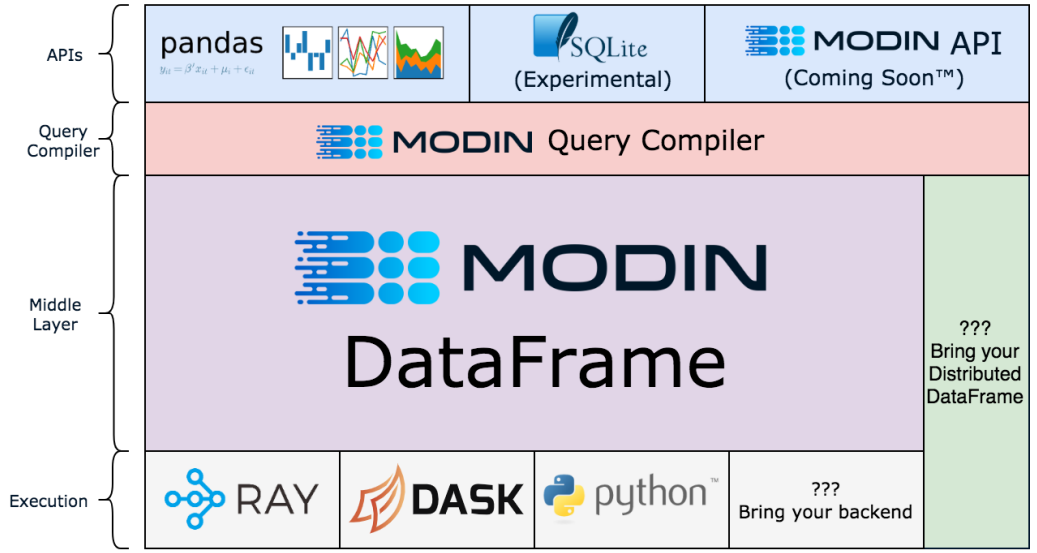


Image source: https://modin.readthedocs.io/en/latest/architecture.html#system-architecture

Intrepydd extensions for Distributed Heterogeneous Computing in AMPHC project

Python code

Intrepydd kernels

Intrepydd front-end

Addition of Python code generation pass

Python Runtime + Python Libraries + Intrepydd Libraries

High-Level Optimizations

Knowledge Base

output=cpp mode

output=python mode

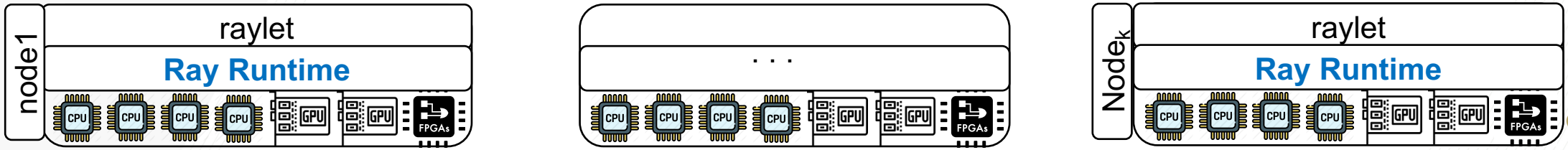
C++ code gen

Python code gen

C++ compiler

host=python mode

kernel-ray.py



Georgia Tech
CREATING THE NEXT

Example of Python code generation for Ray-AMPHC runtime

Input Function
(after distillation):

```
1. def read_array(file):
2.     # read ndarray “a”
3.     # from “file”
4.     return a


5. def add(a, b):
6.     return np.add(a, b)

7. a = read_array(file1)
8. b = read_array(file2)
9. sum = add(a, b)
```

Input Class
(after distillation):

```
1. class Counter(object):
2.     def __init__(self):
3.         self.value = 0
4.     def inc(self):
5.         self.value += 1
6.         return self.value

7. c = Counter()
8. c.inc()
9. c.inc()
```

Output code generated for Ray-AMPHC runtime

Function → Task

```
1. @ray.remote
2. def read_array(file):
3.     # read ndarray “a”
4.     # from “file”
5.     return a

6. @ray.remote
7. def add(a, b):
8.     return np.add(a, b)

9. id1 = read_array.remote(file1)
10.id2 = read_array.remote(file2)
11.id = add.remote(id1, id2)
12.sum = ray.get(id)
```

Object → Actor

```
1.  @ray.remote
2.  class Counter(object):
3.      def __init__(self):
4.          self.value = 0
5.      def inc(self):
6.          self.value += 1
7.          return self.value

8.  c = Counter.remote()
9.  c.inc.remote()
10. c.inc.remote()
```

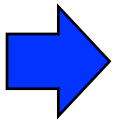
Automatic Distributed + Heterogeneous Parallelization of Intrepydd kernels

Space Time Adaptive Processing (STAP) kernel from signal processing application

```
1. def gen_proc_datacube(…):
2.     ... # Initializations
3.     beamforming = zeros((numPulses, numSamples), …)
4.     for idx in range(numPulses):
5.         dataCube = obtain_data_cube_slice(…)
6.         beamforming[idx,:] = squeeze(matmul(steerVector11,
7.                                             dataCube))
8.     d_X = fft.fft(beamforming, fftSize, axis=1)
9.     d_Y = d_X * d_matchFilterMultiply
10.    d_y = fft.ifft(d_Y, axis=1)
11.    d_yNorm = d_y / numSamples
12.    d_yNorm = fft.fftshift(d_yNorm, axes=1)

13.    d_ZTemp = fft.fft(d_yNorm, 4*numPulses, axis=0)
14.    d_Z = fft.fftshift(d_ZTemp, axes=0)
15.    return d_Z
```

Python program using NumPy arrays



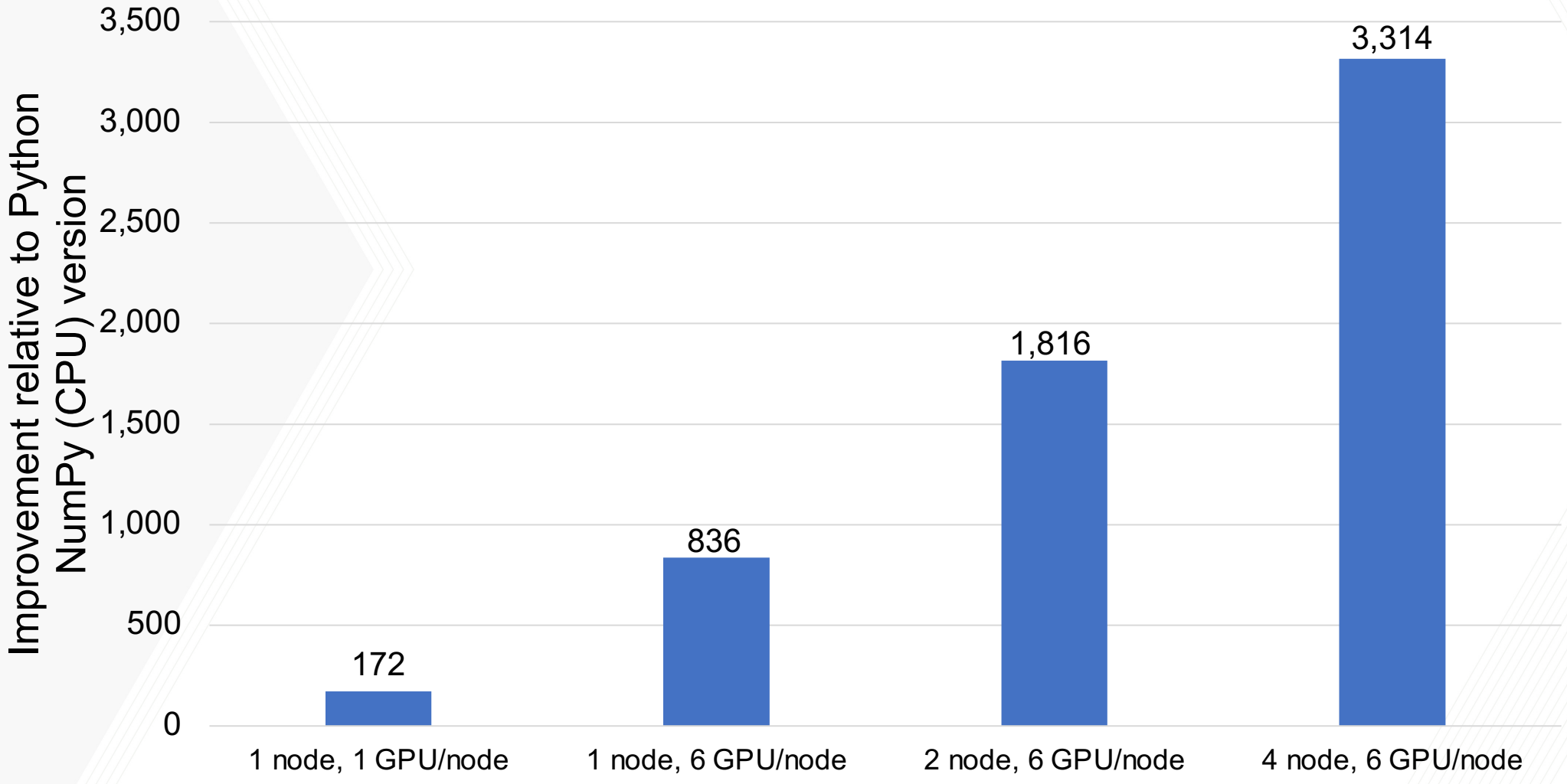
```
1.  def task1(steerVector11, dataCube, fftSize, numSamples,
    matchFilterMultiply):
2.      beamforming_1D = cp.squeeze(cp.matmul(steerVector11, dataCube))
3.      d_X_1D = cp.fft.fft(beamforming_1D, fftSize)
4.      d_Y_1D = d_X_1D * d_matchFilterMultiply[idx,:]
5.      d_y_1D = cp.fft.ifft(d_Y_1D)
6.      d_yNorm_1D = d_y_1D / numSamples
7.      return cp.fft.fftshift(d_yNorm_1D)
8.
9.  def gen_proc_datacube(…):
10.     ... # Initializations
11.     d_yNorm = cp.zeros((numPulses, fftSize), …)
12.     for idx in range(numPulses):
13.         dataCube = obtain_data_cube_slice(…)
14.         # Spawn a distributed Ray task
15.         d_yNorm[idx,:] = [task1.remote(steerVector11, dataCube, fftSize,
16.                                        numSamples, matchFilterMultiply[idx,:])]
17.     # Synchronize on all spawned tasks
18.     d_ZTemp = cp.fft.fft(ray.get_all(d_yNorm), 4*numPulses, axis=0)
19.     d_Z = cp.fft.fftshift(d_ZTemp, axes=0)
20.     return cp.asnumpy(d_Z)
```

Output Distributed-Parallel Heterogenous code using CuPy and Ray

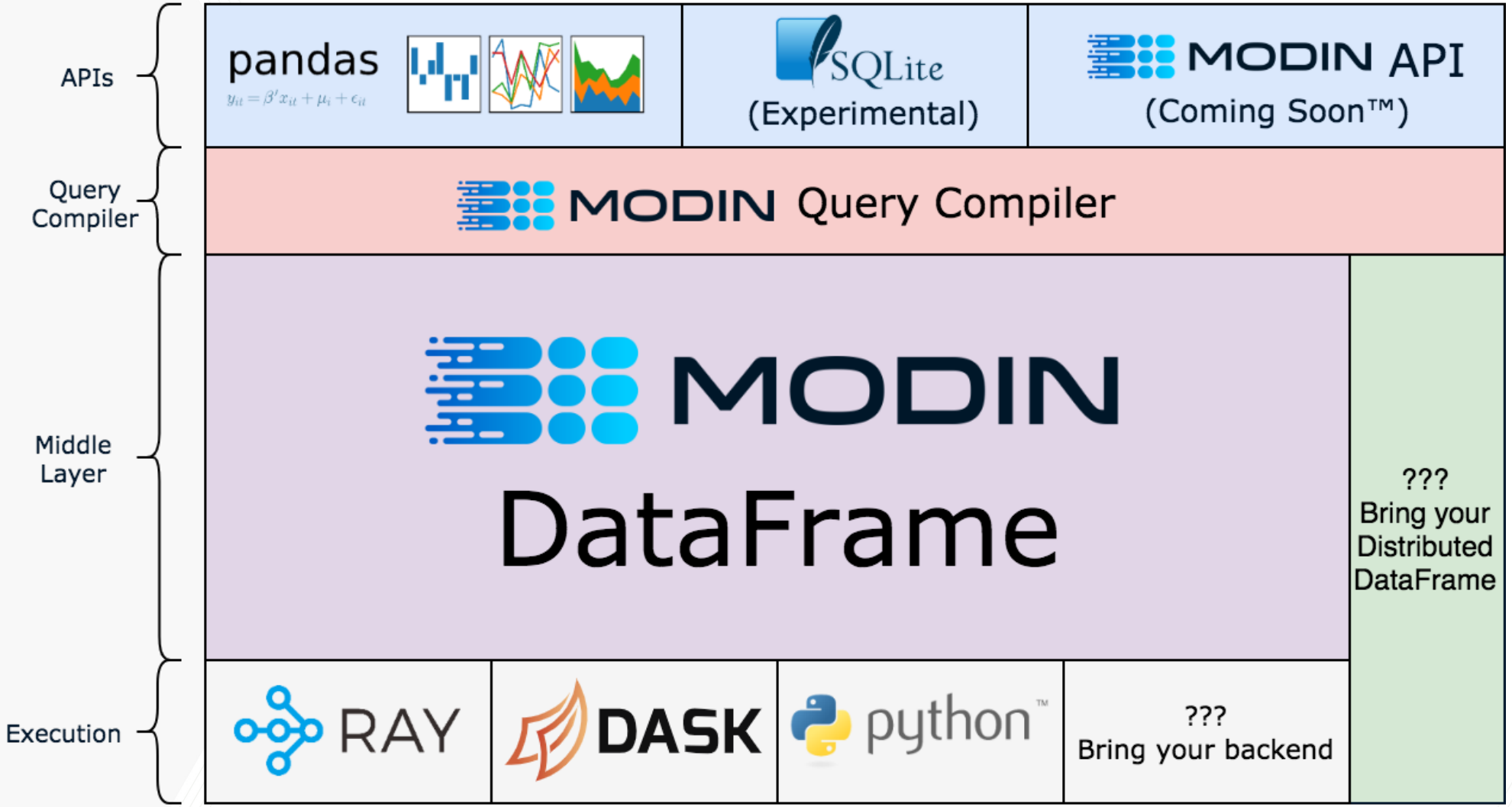
Experimental Setup for Figure 3.4 in Milestone 4 report

- NERSC Cori (GPU nodes)
  - Intel Xeon Skylake: 2/node, total 40 physical cores
  - NVIDIA Volta V100 GPUs: 8/node
- Problem size (data cube)
  - # pulses per cube = 100; # channels = 1,000; # samples per pulse = 30,000
    ➔ One data cube contains 3×10^9 elements
  - Total # data cubes processed = 64
- Compare performance of two variants of STAP Datacube Processing application
  1. NumPy code (original version, baseline for comparison)
  2. Ray Tasks with automatically generated parallel tasks with CuPy
     - Enable inter-node and intra-node parallelism via Ray tasks to use multiple GPUs
     - Each task invokes CuPy functions to run on a single GPU
- Timings are for entire application

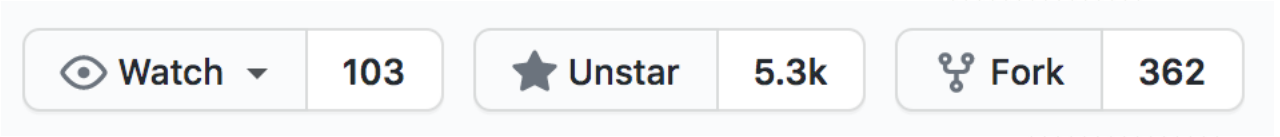
Parallelization of STAP Kernel on NERSC Cori GPU Nodes (relative to original NumPy version)



Using Intrepydd for Data processing at scale

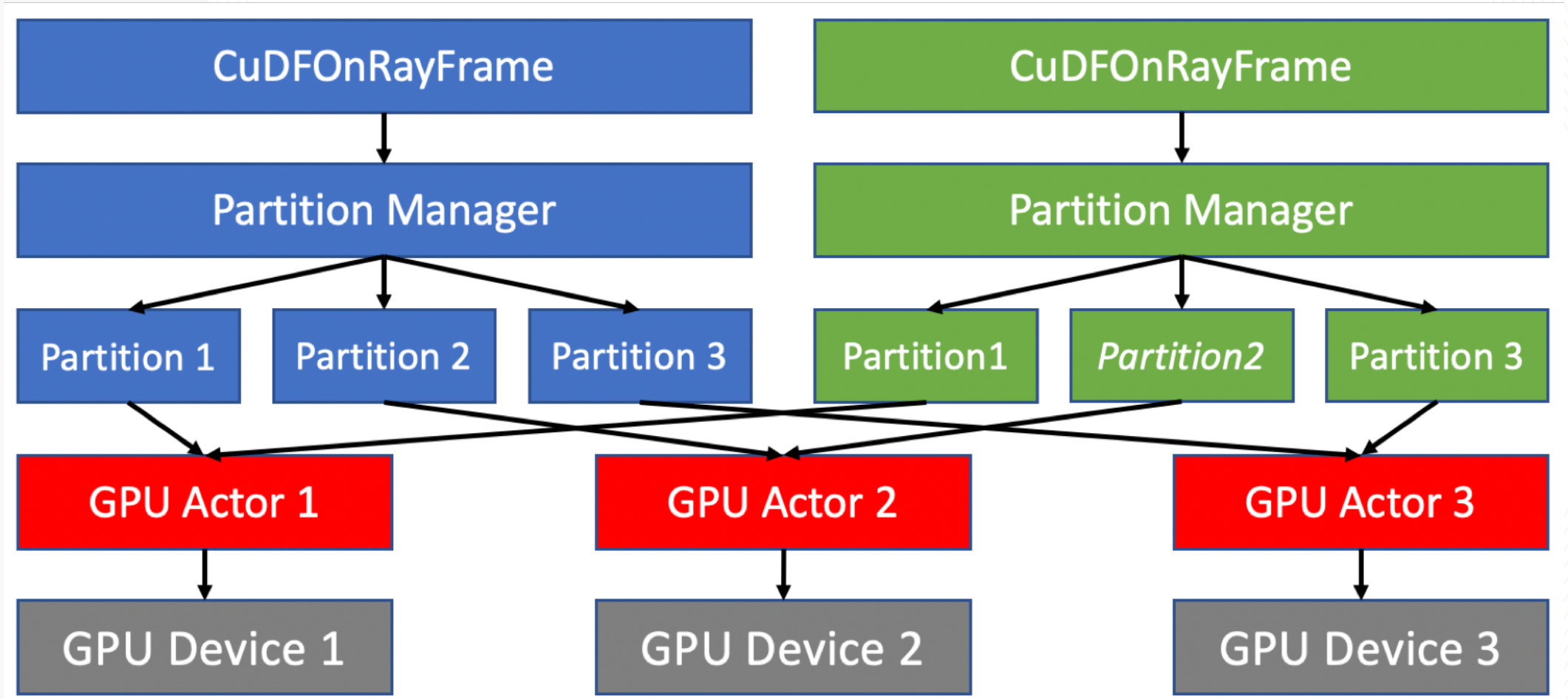


Modin: Current Impact



- Tesla
- DoD
- Oak Ridge National Lab
- Splunk
- NVIDIA
- Ford
- Intel
- 12+ other small groups
- 5,300+ stars
- 22k installs/month, 230k total (since July 2018 start)
- Reached overall trending on GitHub multiple times (top starred repos of the day)
- **This interest shows that Modin is solving problems for real users**
- **Real-world testbed, let's leverage this community to help data scientists use distributed heterogeneous parallelism**

Modin GPU Architecture Modification (Work in progress)



- While MODIN-CPU can rely on the shared-memory object store, we have to manage the partition placement.

End-to-end workflows

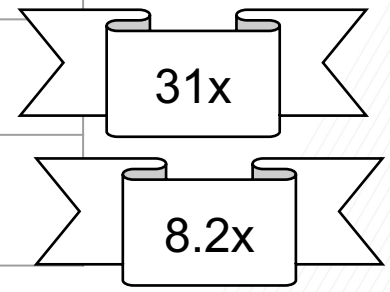
- Data Analysis for Network Security (from Kaggle)
  - Dataset: A CSV file with 100M rows and 5 columns
  - Use network flow data to uncover anomalous security events (10 questions)

- MovieLens Dataset Preprocessing (Adapted from an example in Pandas author's book)
  - Dataset: 2 CSV files with 25M rows and 60K rows respectively
  - Preprocessing the dataset for downstream machine learning tasks

Data Analysis for Network Security

- There is a preprocessing stage and a query stage in this workflow.
- Finished the preprocessing stage, and are currently working on the query stage.
- Finished 22 out of the 43 APIs needed by this workflow.

	Pandas	cuDF	MODIN-CPU (24 cores)	MODIN-GPU (8 GPUs)
Code Block 1	46.90	Out of memory	8.13	17.30
Code Block 2	75.00	Out of memory	36.6	**2.42**
Code Block 3	44.20	Out of memory	48.7	**5.33**

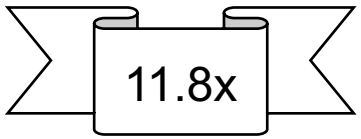


- The first code block is reading a CSV file. Partitions are read as Pandas DataFrames, then transferred to GPU. cuDF doesn't support directly reading large CSV files.

MovieLens Dataset Preprocessing

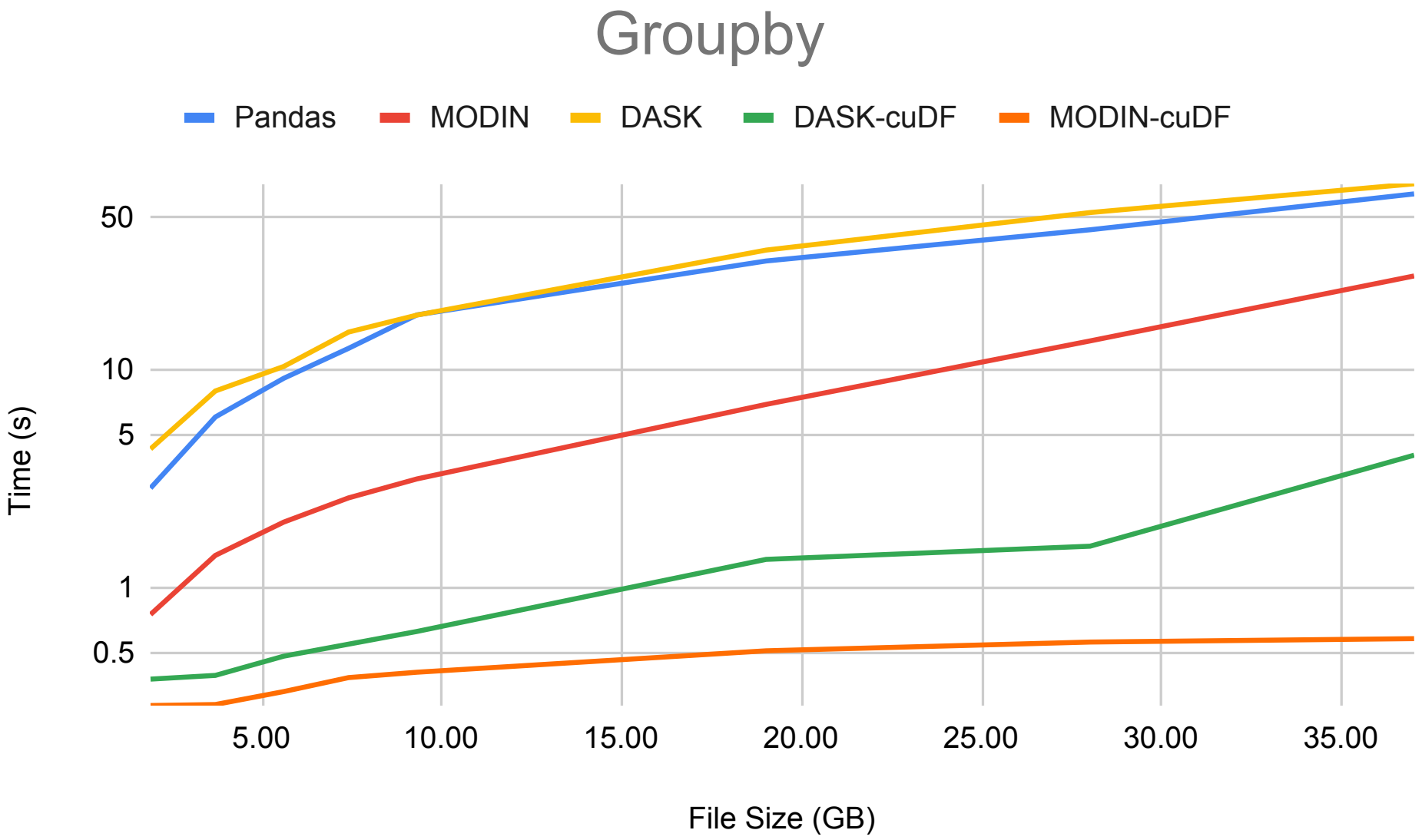
- This notebook is shorter, but it contains two key APIs missing from the previous workflow: **join/merge()** and **pivot_table()**.
- Here are some initial results for join/merge().

Data Size	pandas	cudf	modin-cpu	modin-gpu
60K Joined to 25M (25M rows x 3 columns) ROW PARTITION 4 GPUS	3.84 s	69.8 ms	2.76 s	1.41-1.91 sec w/ heuristic 2.6 sec w/o heuristic
12M Joined to 12M (12M rows x 19 columns) ROW PARTITION 8 GPUS	1 min 47 seconds	Out Of Memory	1 min 15 secs	9 seconds



- Heuristic:
  - Bypass the object store when 2 partitions are in the same device
  - Send smaller partition to the larger partition.
- MODIN-GPU pays the overhead w.r.t cuDF when the data fits in one GPU.

Groupby Microbencharks



Summary

- AMPHC is a new approach to Performant Automation of Parallel Program Assembly (PAPPA)
  - Performant and portable programming model based on Python
  - Compiler architecture to extract inter-node and intra-node parallelism
    - Distillation: clean up aspects of input code that may interfere with parallelization (guided by knowledge base)
    - Annealing: generate distributed + heterogeneous parallel code that can be executed on AMPHC runtime framework
  - Runtime framework that supports inter-node parallelism using Ray-AMPHC and intra-node parallelism using HClib-AMPHC
  - Two motivating application domains: signal processing, data analytics

Outline

1. Motivation

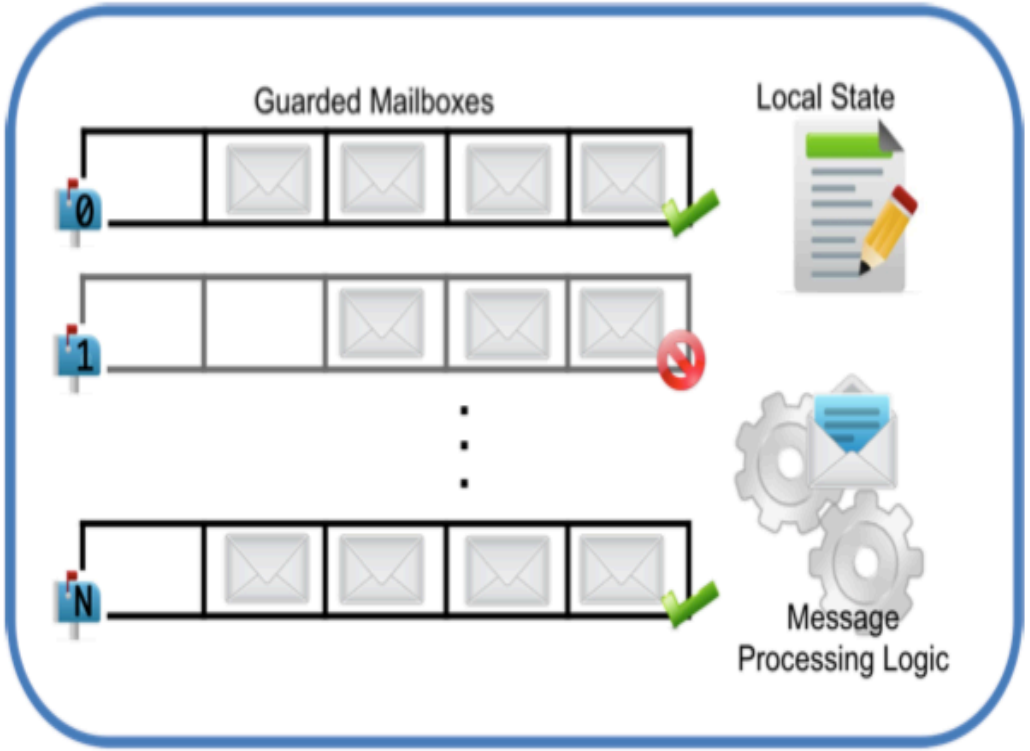
2. Intrepydd -- an AOT tool chain for optimization & parallelization of Python programs

3. AMPHC – extend Intrepydd for Automating Massively Parallel Heterogeneous Computing using Python

4. Conclusions and Next Steps

Exploring an actor-based model for workloads with short asynchronous messages

- Selector: Actor model extended with multiple mailboxes
  - A messages is sent to a specific mailbox in receiver actor
  - Each mailbox can be selectively enabled or disabled
  - Actor = Selector with one mailbox
- Symmetric Mailboxes: each selector has the same set of mailboxes
- Use of Conveyors for scalable communication with automatic message aggregation
- Automatic termination detection



Selectors: Actors with Multiple Guarded Mailboxes.
S M Imam, V Sarkar, Agere14

"A Scalable Actor-based Programming System for PGAS Runtimes". Sri Raj Paul, Kun Chen, Akihiro Hayashi, Vivek Sarkar.

Histogram example (current: C/C++, future: Python+Ray)

Ideal version (global view):

```
for(int i=0; i<n; i++)  histo[index[i]] += 1;
```

Conveyors version:

```
1.  convey_begin(c);
2.  int i=0, spot;
3.  while(convey_advance(c, i==n)) {
4.    for(;i<n;i++) {
5.      spot = index[i] / procs;
6.      PE = index[i] % PROCS;
7.      if(! convey_push(c, &spot, PE) break;
8.    }
9.    while(convey_pull( c,&spot, &from))
10.     histo[spot]++;
11. }
```

Selector version:

```
1.  HistoActor * h_actor = new HistoActor();
2.  for(int i=0; i < n; i++) {
3.    spot = index[i] / PROCS;
4.    PE = index[i]%PROCS;
5.    h_actor.send(PE, [=]() {lcounts[spot] += 1;});
6.  }
```

Evaluation

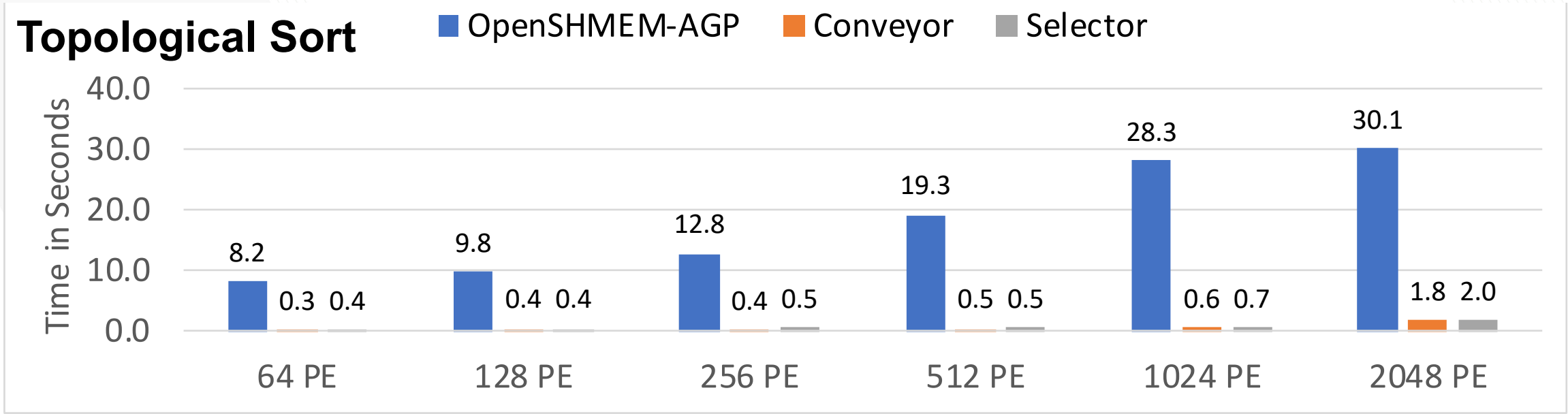
- Cray XC40™ Supercomputer @ NERSC (Cori)
  - Node
    - 2 Intel Xeon E5-2698 v3 @ 2.30GHz 16 cores
    - 128GB of RAM
- Cray Aries interconnect with Dragonfly topology with a global peak bisection bandwidth is 45.0 TB/s
- Maximum 64 Nodes with 32 OpenSHMEM PEs mapped to one node i.e., 2048 Pes
- Use of Habanero-C/C++ library (HClib)
  - One HClib worker thread per PE was used for these results (multiplexes computation and communication tasks)

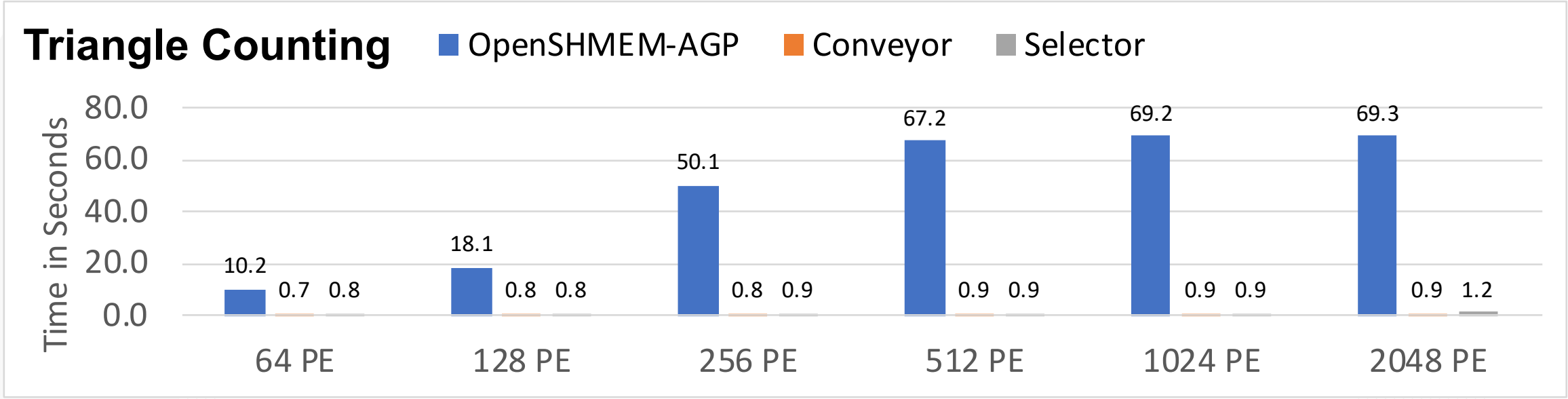
Motivation for use of Conveyors

		NB	Time
Histogram	OpenSHMEM NBI (cray-shmem 7.7.10)	Y	4.3
	UPC (Berkley-UPC 2020.4.0)	N	23.9
	MPI3-RMA (OpenMPI 4.0.2)	Y	88.9
	MPI3-RMA (cray-mpich 7.7.10)	Y	>300
	Charm++ (6.10.1, gni-crayxc w/ TRAM)	Y	9.7
	Conveyors (2.1 on cray-shmem 7.7.10)	Y	0.5
Index-gather	OpenSHMEM (cray-shmem 7.7.10)	N	35.5
	OpenSHMEM NBI (cray-shmem 7.7.10)	Y	4.2
	UPC (Berkley-UPC 2020.4.0)	N	22.6
	UPC NBI (Berkley-UPC 2020.4.0)	Y	19.7
	MPI3-RMA (OpenMPI 4.0.2)	Y	25.8
	MPI3-RMA (cray-mpich 7.7.10)	Y	8.3
	Charm++ (6.10.1, gni-crayxc w/ TRAM)	Y	21.3
	Conveyors (2.1 on cray-shmem 7.7.10)	Y	2.3

Absolute performance in seconds using best performing variants for Histogram and Index Gather on 2048 PEs (64 nodes with 32 cores (or PE) per node) in the Cori supercomputer which performs 2^{23} updates for Histogram and reads for Index Gather. (NB indicates if non-blocking communication was used.)

Performance results for Topological Sort and Triangle Counting Mini-Apps (Bale 2.1)





Conclusion: Abstraction without Apology!

Holy Grail:

- Domain expert specifies application and algorithm with declarative parallelism and semantics guarantees
- Compiler generates multi-version code for multiple target devices and inputs
- Runtime schedules compute and data movement tasks on distributed heterogeneous HPC platform

Exciting times for Extreme Scale Programming Models and Middleware:

- New applications (Deep Learning, Data Science, Real-time, …)
- New languages (Python, Rust, DSLs, …)
- New parallel hardware (clusters, multicore, accelerators, vector units, matrix units, …)