Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels

Vivek Sarkar
Professor & Chair, School of Computer Science
College of Computing
Georgia Institute of Technology

CRNCH Summit, Jan 28-29, 2021

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

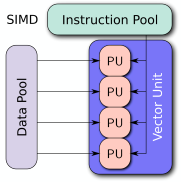
Utilization wall (Dark silicon) — ~2010

Locality wall — ~2015

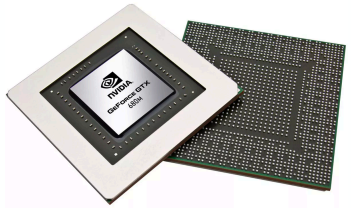
End of Moore's law

Multi-core processors

SIMD units

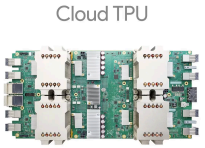


GPGPUs

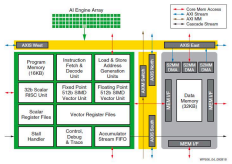


Light-weight many-core processors

Domain-specific accelerators

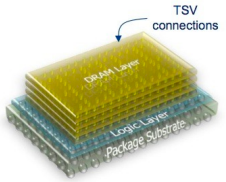


Specialized SIMD Units



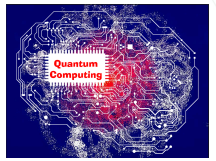
Migratory Architectures

3D stacked Processors



.....

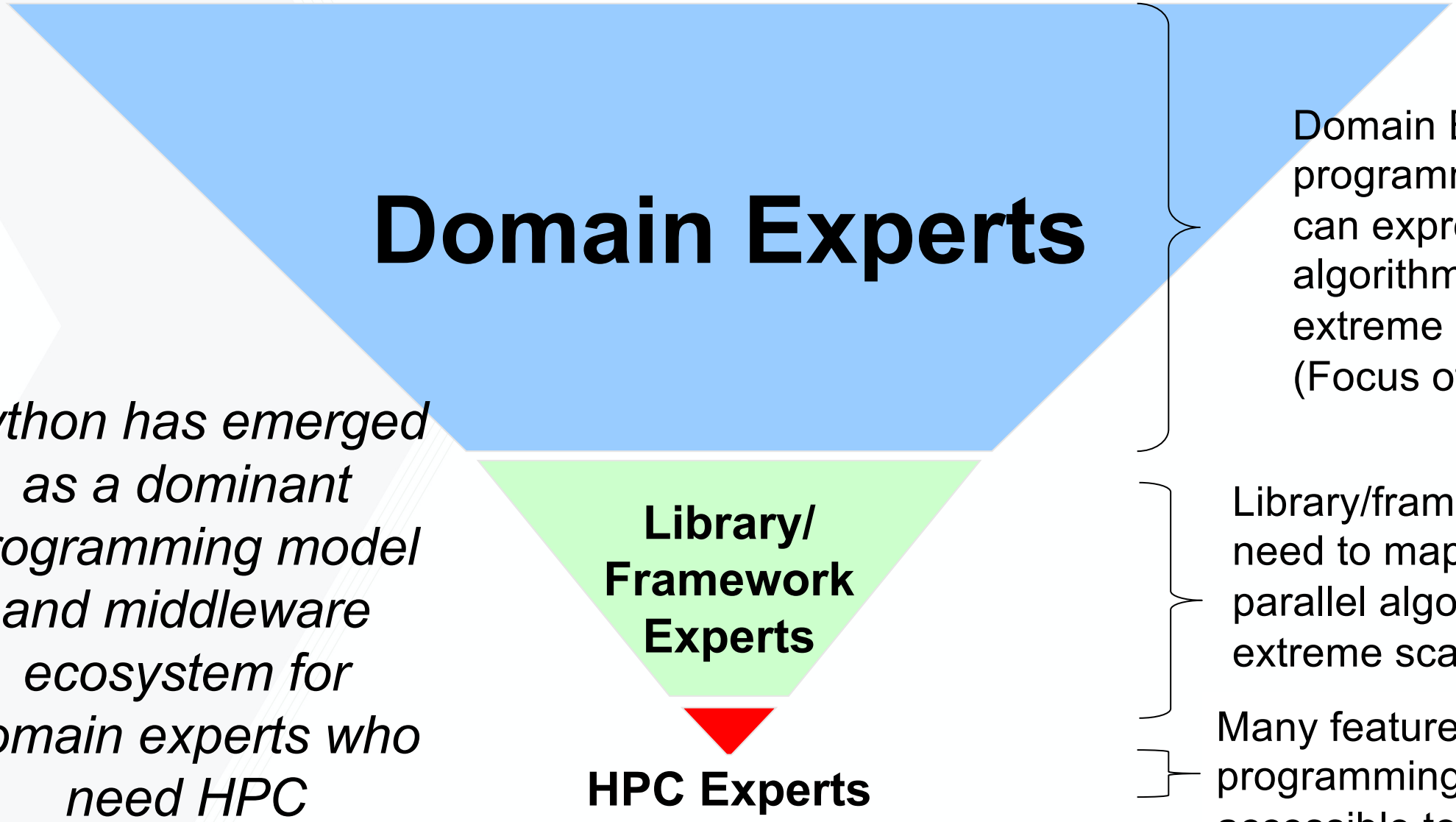
Quantum Architectures



Neuromorphic computing

Georgia Tech
CREATING THE NEXT

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 current parallel algorithms on to extreme scale hardware

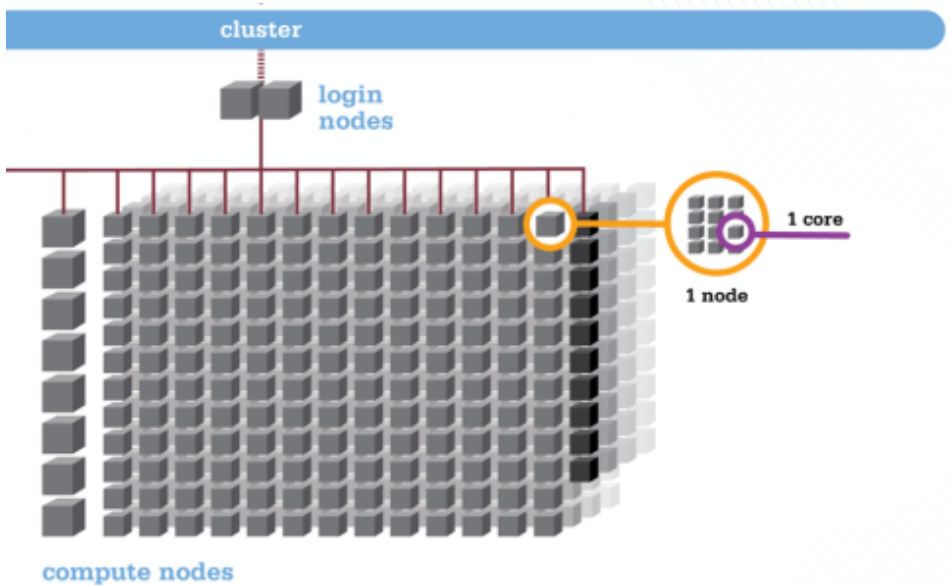
Many features in today's HPC programming models are only accessible to HPC experts

Increasing Complexity with Increasing Parallelism for Python Programmers

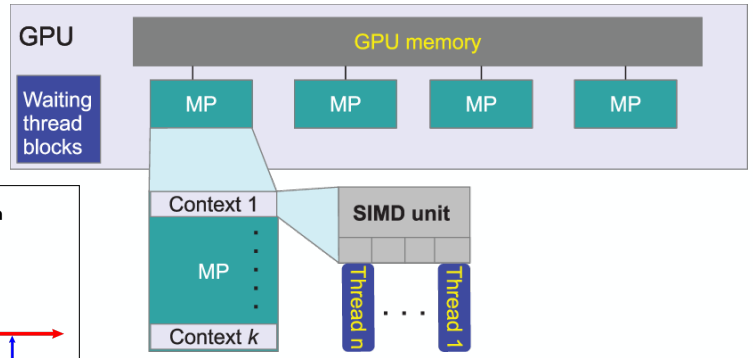
Parallelism cliffs: step function in programming and tuning efforts needed to enable applications to exploit the next stage of increasing parallelism

MPP Clusters

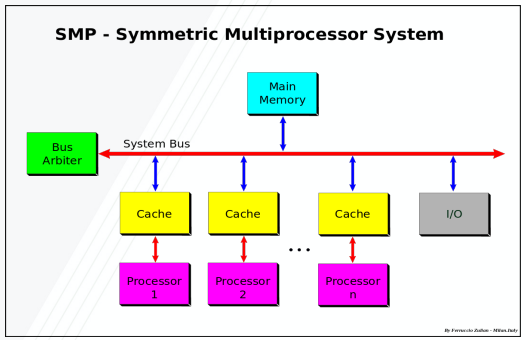
… Extreme Heterogeneity …



Single/multiple GPUs



Multicore



Single CPU

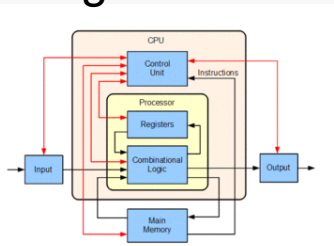


Image sources:
https://en.wikipedia.org/wiki/File:ABasicComputer.gif
https://en.wikipedia.org/wiki/Symmetric_multiprocessing
https://www.researchgate.net/figure/Hierarchical-hardware-parallelism-in-a-GPU_fig1_262176632
https://www.osc.edu/book/export/html/2782

Georgia Tech
CREATING THE NEXT

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

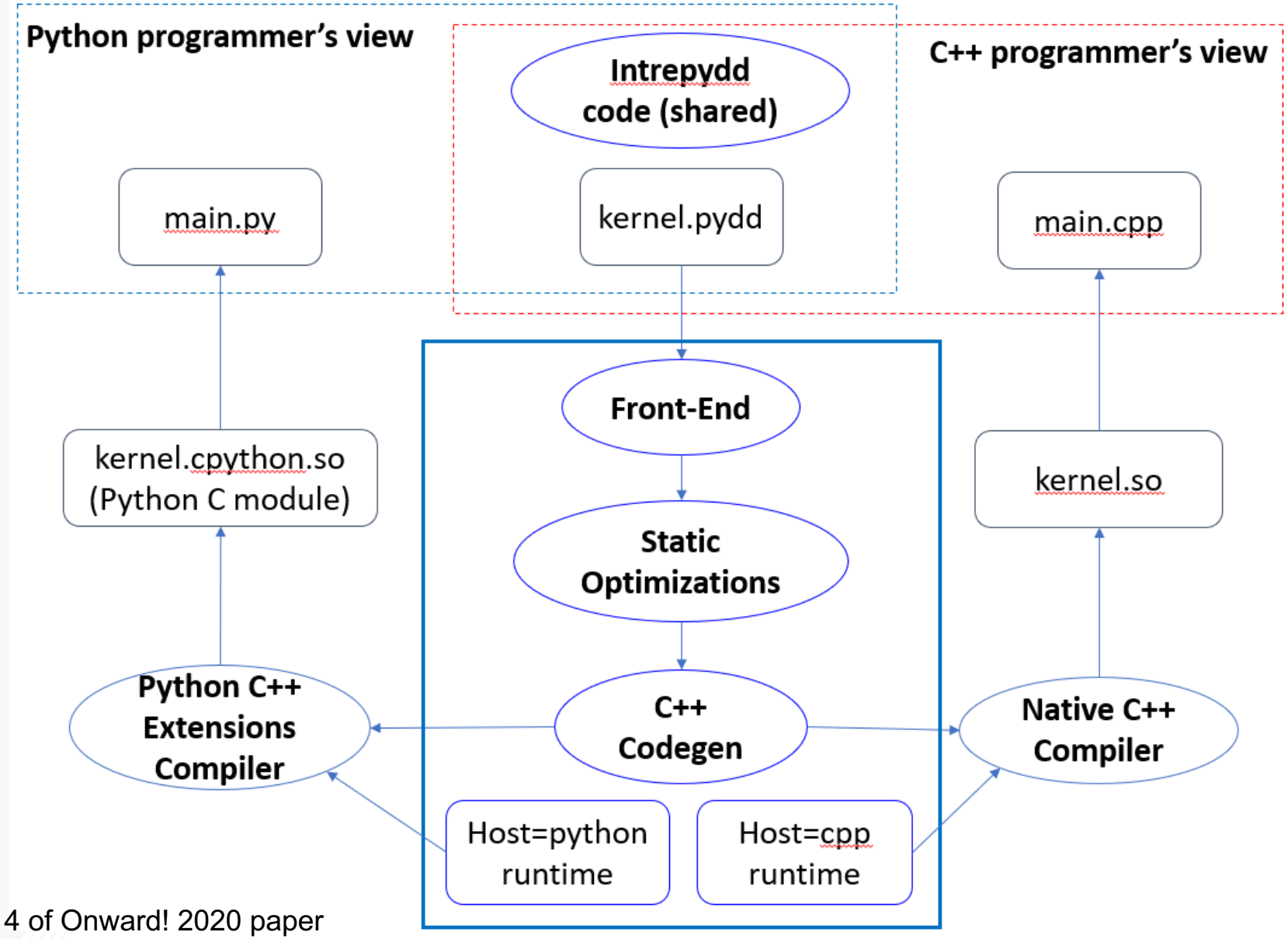
Motivation for our work: combine the benefits of HPLs and DSLs in a single Discipline-Aware Language (DiAL) for Data Science

”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.

Many reasons to not attempt static 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 Onward! 2020 paper

Intrepydd Language Definition (Summary)

- Data Types
  - Boolean
  - Numeric: int32, int64, float32, float64
  - Collections: List(type), Array(type), SparseMat(type), Dict(type), Heap(type)
  - Data distributions
- 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

Sinkhorn WMD Kernel in Python vs. Intrepydd

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 vs. Intrepydd

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 vs. Intrepydd

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 vs. Intrepydd

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 vs. Intrepydd

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 vs. Intrepydd

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)
```

Using Intrepydd from Jupyter 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

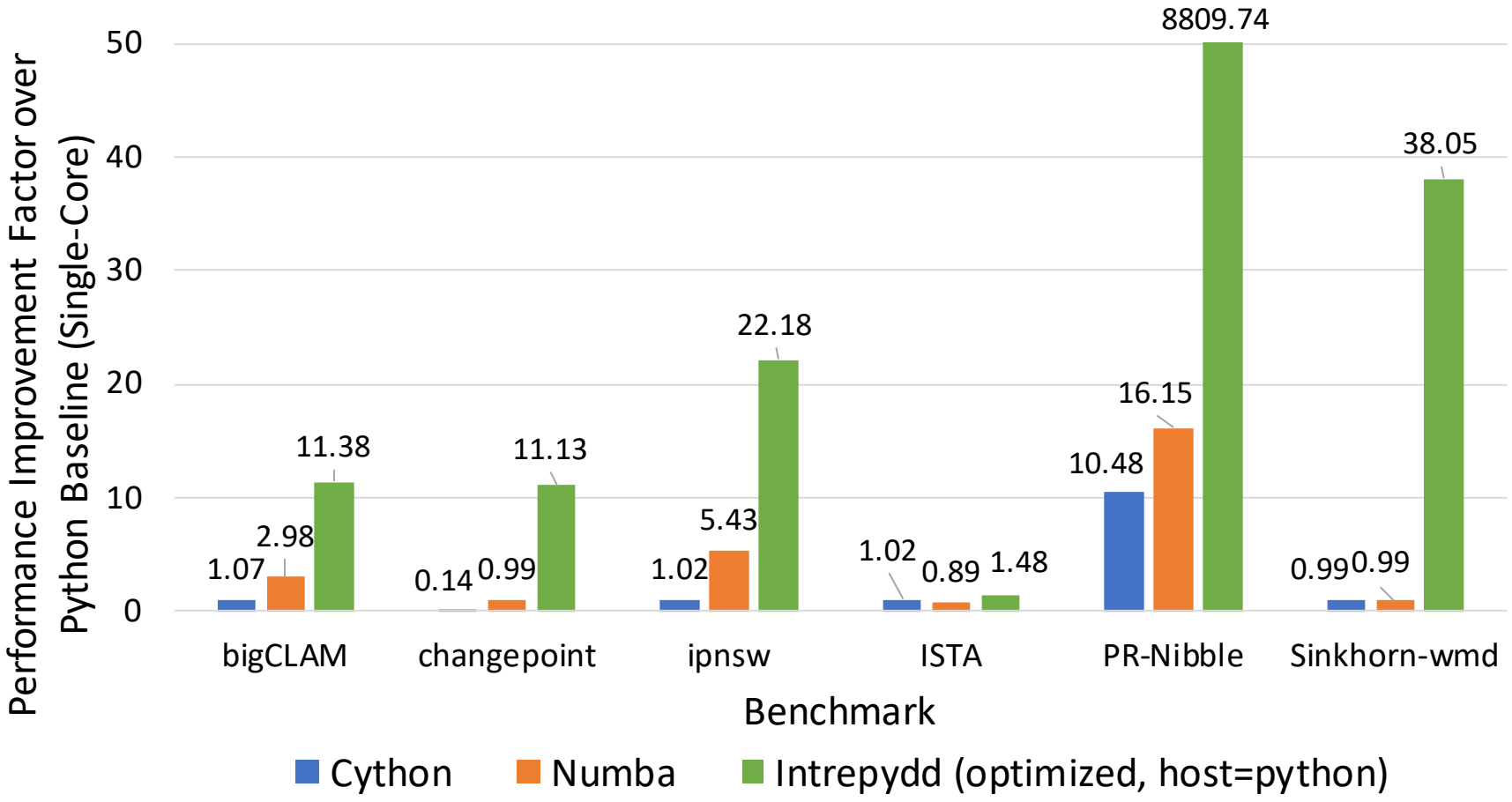
Benchmark Applications

- 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

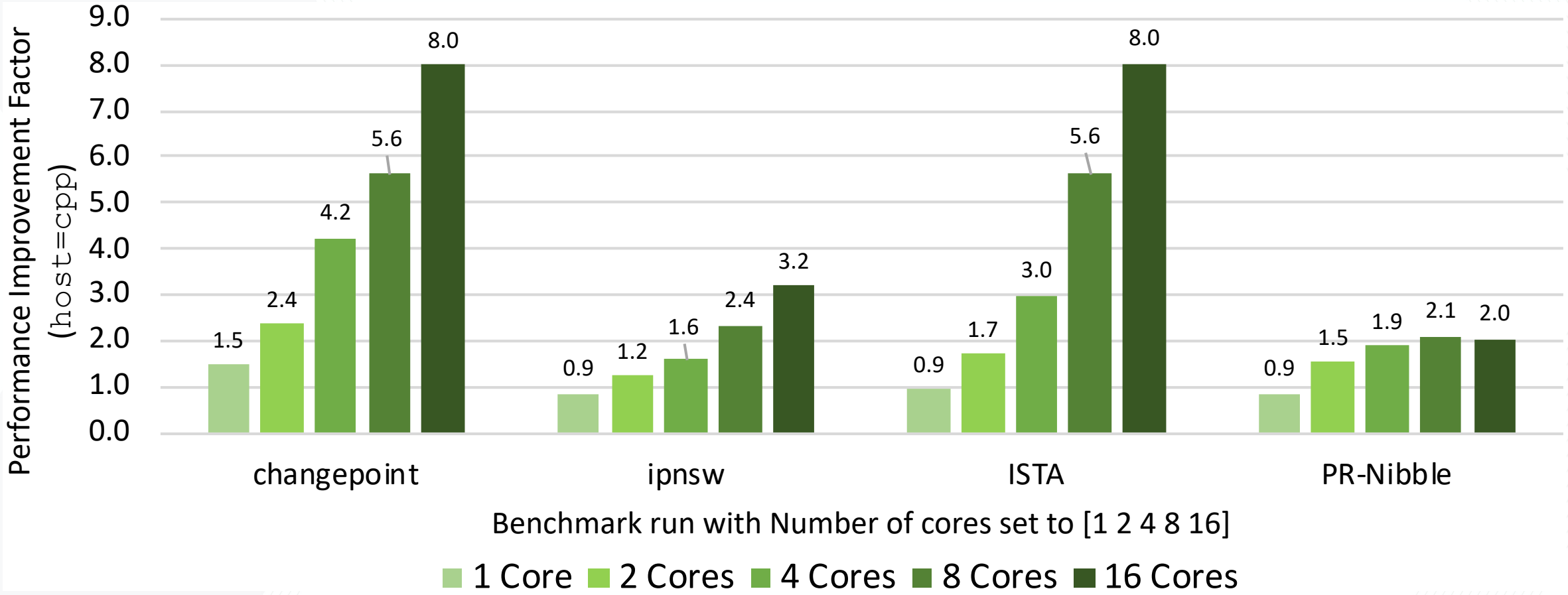
Intrepydd Single Core Performance



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

Details in Section 6.3 of Onward! 2020 paper

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

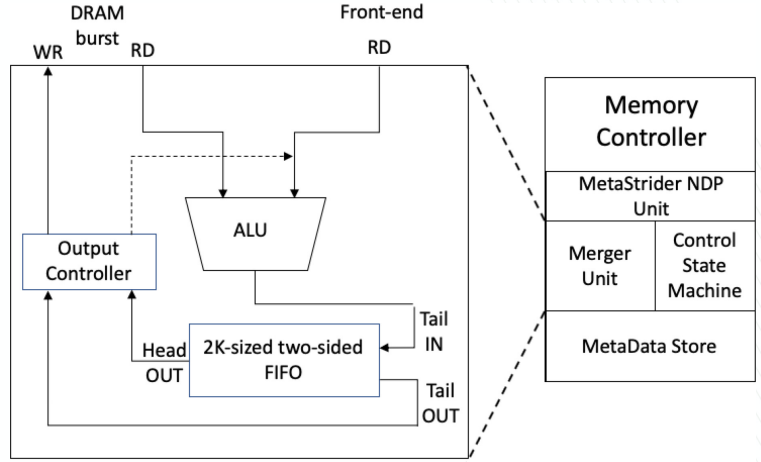


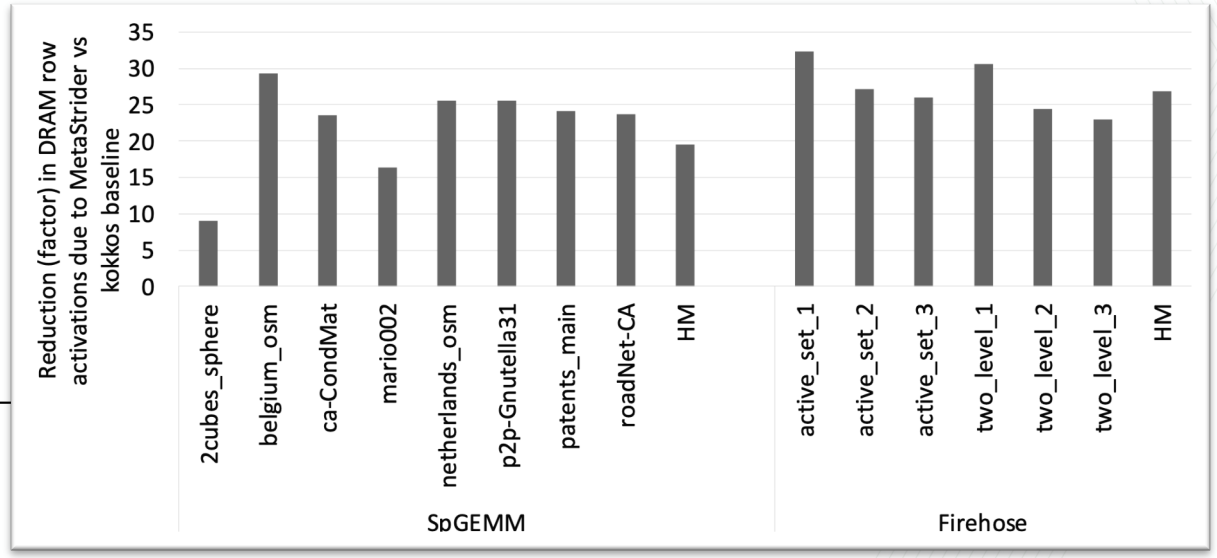
Georgia Tech
CREATING THE NEXT

Details in Section 6.4 of Onward! 2020 paper

Looking ahead: using Intrepydd for MetaStrider* sparse reduction accelerator

```
# A and B may be stored in conventional sparse formats or in the
# accelerator format, abstracted as Intrepydd sparse matrix type.
def SpGEMM_OuterProduct(A: SparseMat, B: SparseMat) -> SparseMat:
    C = empty_spm(A.shape(0), B.shape(1))
    Y = A.getNonZeroColumnIndices()     # Set of valid column indices
    Z = B.getNonZeroRowIndices()        # Set of valid row indices
    X = sorted(Y & Z)                   # Sorted list of valid indices
    finish:
        for x in X:
            colA = A.getCol(x) # List of non-zeros (index, value)
            rowB = B.getRow(x) # List of non-zeros (index, value)
            for (i, A_ix) in colA:
                for (j, B_xj) in rowB:
                    partial = A_ix * B_xj
                    async:
                        isolated:
                            C[i,j] += partial
    return C
```





** Sriseshan Srikanth, Anirudh Jain, Joseph M Lennon, Thomas M Conte, Erik Debenedictis, and Jeanine Cook. 2019. MetaStrider: Architectures for Scalable Memory-centric Reduction of Sparse Data Streams. ACM Transactions on Architecture and Code Optimization (TACO) 16, 4 (2019), 1–26.*

Details in Section 7.2 of Onward! 2020 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 Onward! 2020 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 Onward! 2020 paper

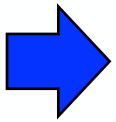
Automatic Distributed + Heterogeneous Parallelization of Intrepydd kernels

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)
```

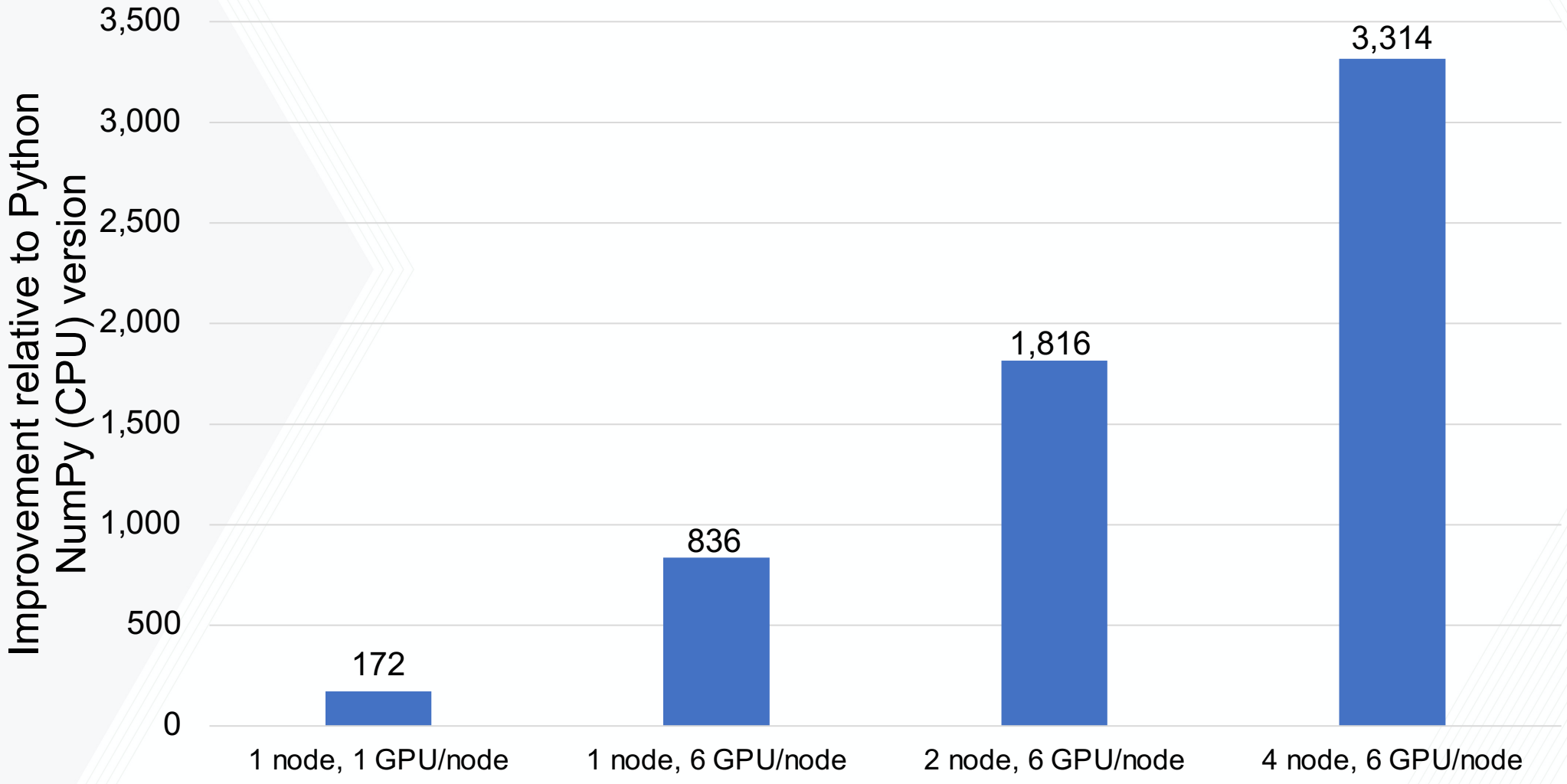
Output Distributed-Parallel Heterogenous code using CuPy and Ray

Acknowledgment: Team members on Georgia Tech DARPA PAPPA project. Shelby Allen, Barry Drake, Mingyu Guan, Taesoo Kim, Sukarno Mertoguno, Vivek Sarkar (PI & POC), Jun Shirako, Alexey Tumanov, Tong Zhou.

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)



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, analog, neuromorphic, …)

Bring us your wired, your core, your huddled hardware yearning to breathe free …