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)

```python
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

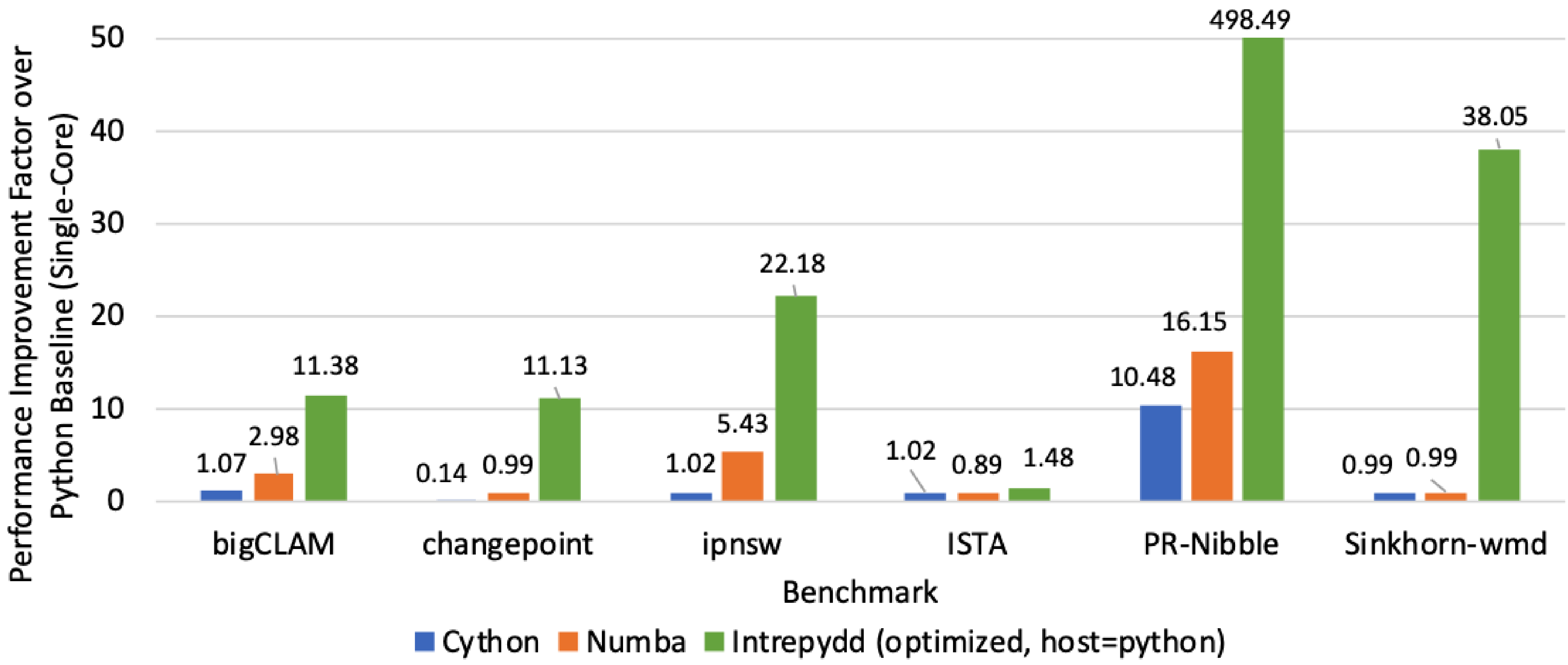
Test machine

- 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×)