Accelerating K-means Clustering

Sreemanth Prathipati and Sooraj Karthik

Background and Motivation

- Very widely used
  - Image compression
  - Video recommendation systems
  - Insurance fraud detection
- Algorithm
  - Randomly initialize cluster centers
  - Assign points to centers based on distance
  - Adjust centers to mean of assigned points
- Potential for GPU acceleration
  - Most of the algorithm is embarrassingly parallel

Problem Category and Definition

- Problem
  - Accelerate K-means Clustering using GPU
- Categories
  - Reproduce results from prior publications
  - See if we can discover any novel approaches to solve the problem more efficiently

Performance Metrics and Baselines

- Metric: Most papers look at execution time per iteration (ms)
- Baselines:
  - sklearn implementation
  - OpenMP implementation
  - kmcuda (popular open source implementation) [1]

Solution - General Algorithm

- Each thread calculates assignment for one point
- Then we accumulate point values for each cluster
- Finally divide each accumulation by the number of points in the cluster

Solution - Two Approaches

- Separate kernels for calculating cluster assignment and accumulating point values per cluster
- One kernel that calculates assignment and accumulates values per cluster
- Both then need to divide accumulation by number of points in each cluster to get the recentered centroid values

Solution - Shared Memory

- Load a chunk of the centroids into shared memory
- Two parameters:
  - SHM_K: number of centroids to load (we used 16)
  - SHM_DIM: number of dimensions to load (we used 16)
- Tiled loop over centroids with step size SHM_K
  - Tiled loop over dimensions with step size SHM_DIM
    - Load data for the current dimensions into shared memory
    - For each centroid in shared memory, accumulate distances from point to each dimension of centroid
  - Update the point’s assignment

Solution - Privatization

- With just one accumulator, each thread will attempt to add its point’s value to the accumulator at the same time
  - Major performance drawback due to synchronization from atomic adds
- Solution: Create multiple private copies of the accumulator and reduce all private copies down to one at the end
- What is a good number of private copies?
  - We used 8 since any more couldn’t fit in cache
- How do we assign threads to accumulators?
  - Round-robin by warp so that memory access is still coalesced if possible

Solution - Memory Layout

- Row major order - Points are contiguous in memory
- Column major order - Dimensions are contiguous in memory
- Assignment - Column major worked best
- Accumulation - Row major worked best
- Hybrid Solution
  - Input points in column major order
  - Input centroids in row major order

Solution - Kernels

- Assignments
- Accumulation
- Fused (Assignment + Accumulation)
- Private copy reduction
- Divide accumulation by number of points in cluster

Validation

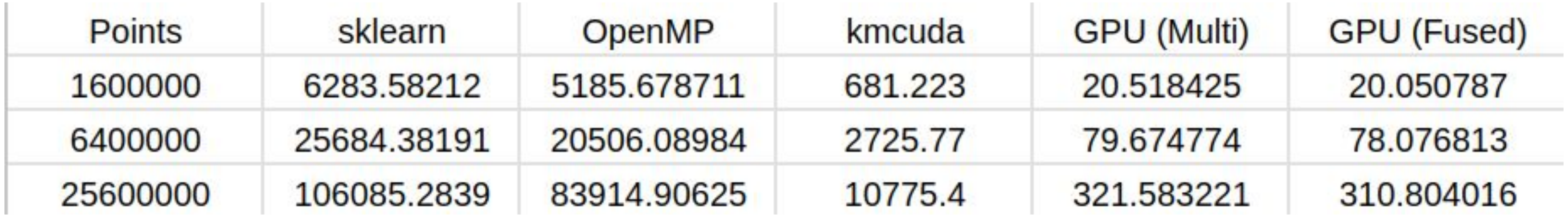
- K-means algorithm is deterministic once initial cluster centers are chosen
- Fix the initial clusters to some predetermined values
- Run accelerated and sequential algorithms on same inputs and see if the outputs are the same

Dataset and Testbed

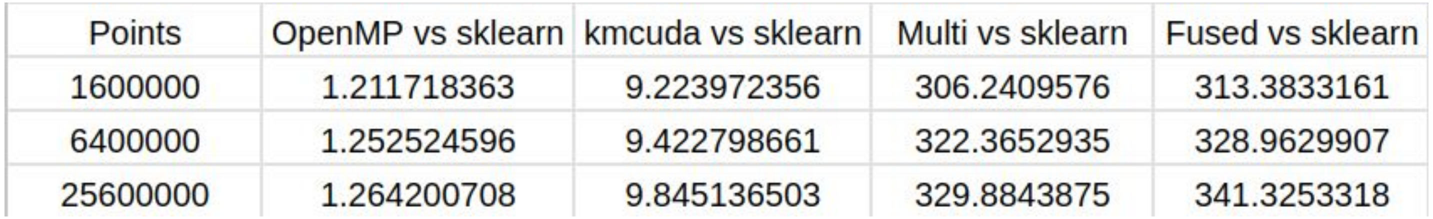
- Dataset
    - Randomly generated vectors
    - General acceleration for k-means, not specific use-cases
- Test System
    - College of Computing PACE GPU Clusters
    - sklearn and OpenMP baselines on 24 cores
    - Test GPU code on Tesla V100 GPU

Experiments and Plots

- Line Plots
  - x-axis
    - Vary number of points (100k, 200k, 400k, ..., 25.6m)
    - Vary number of clusters (16, 32, 64, ..., 4096)
    - Vary dimensionality of data (16, 32, 64, ..., 4096)
  - y-axis
    - Measure time per iteration
- Breakdown plot
  - Time spent in different parts of the algorithm (assignment, accumulation, etc.)

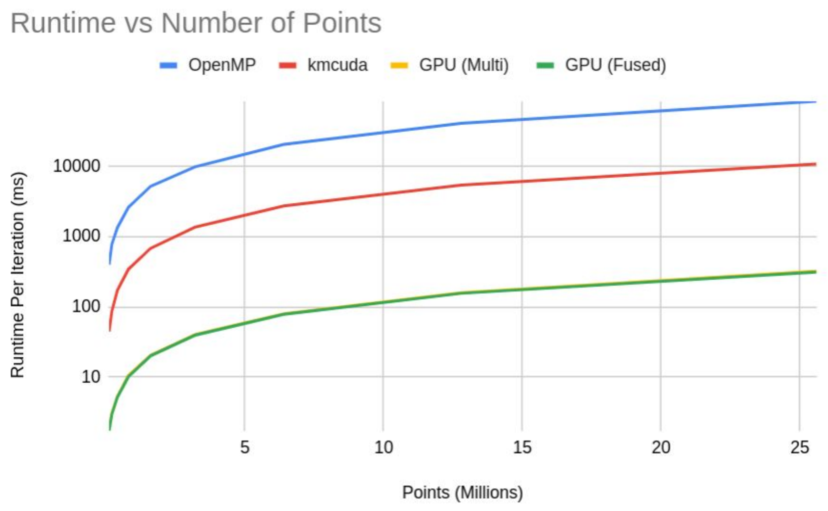


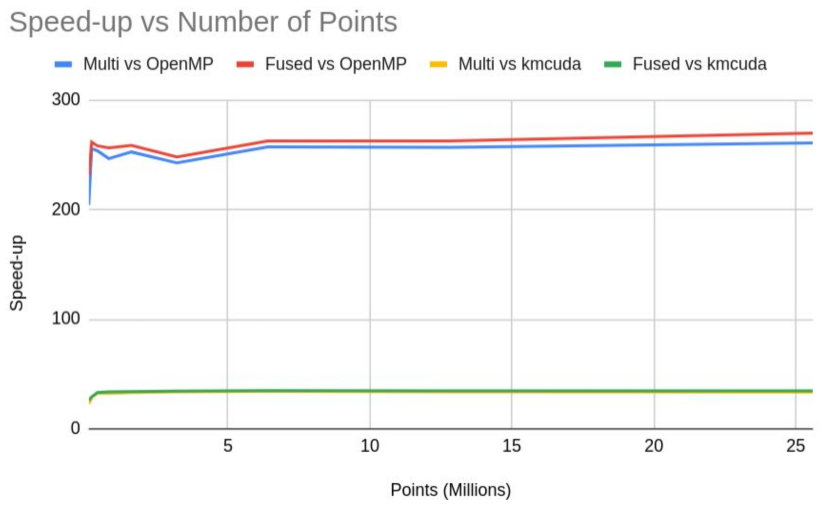
Points	sklearn	OpenMP	kmcuda	GPU (Multi)	GPU (Fused)
1600000	6283.58212	5185.678711	681.223	20.518425	20.050787
6400000	25684.38191	20506.08984	2725.77	79.674774	78.076813
25600000	106085.2839	83914.90625	10775.4	321.583221	310.804016



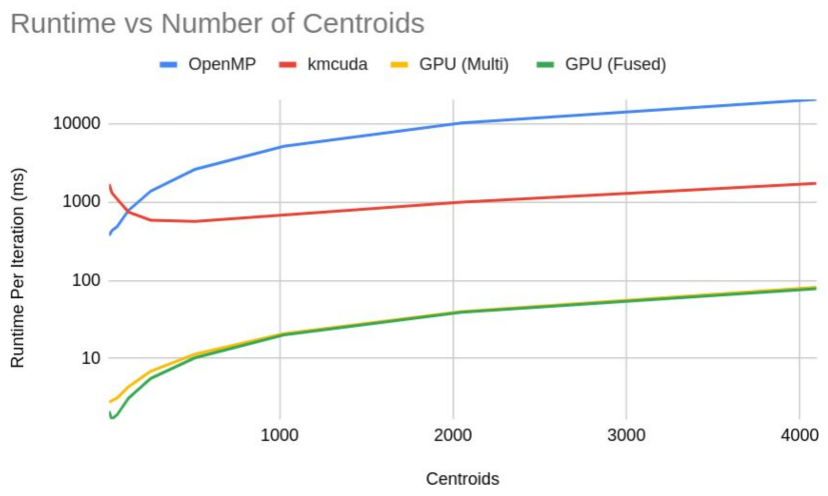
Points	OpenMP vs sklearn	kmcuda vs sklearn	Multi vs sklearn	Fused vs sklearn
1600000	1.211718363	9.223972356	306.2409576	313.3833161
6400000	1.252524596	9.422798661	322.3652935	328.9629907
25600000	1.264200708	9.845136503	329.8843875	341.3253318

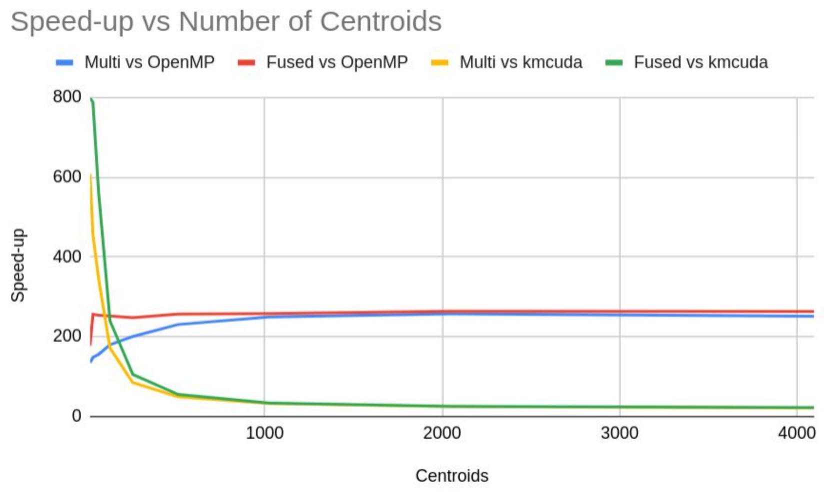
Speed-up vs sklearn
(for 1024 centroids, 32 dims)



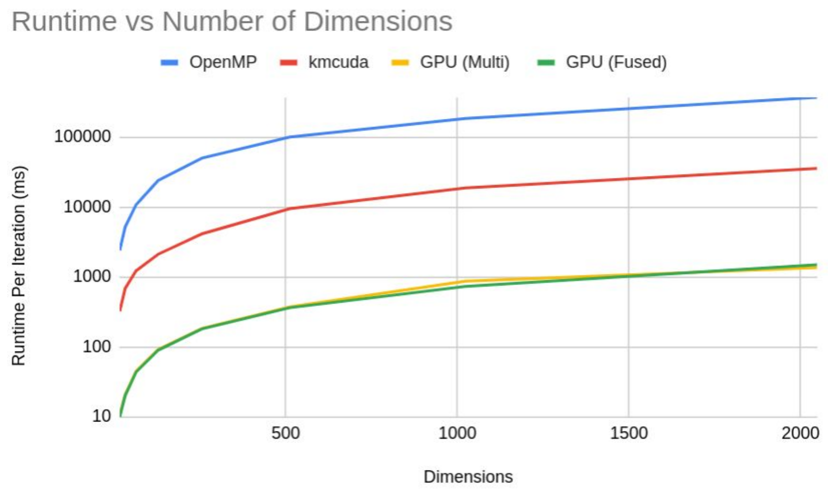


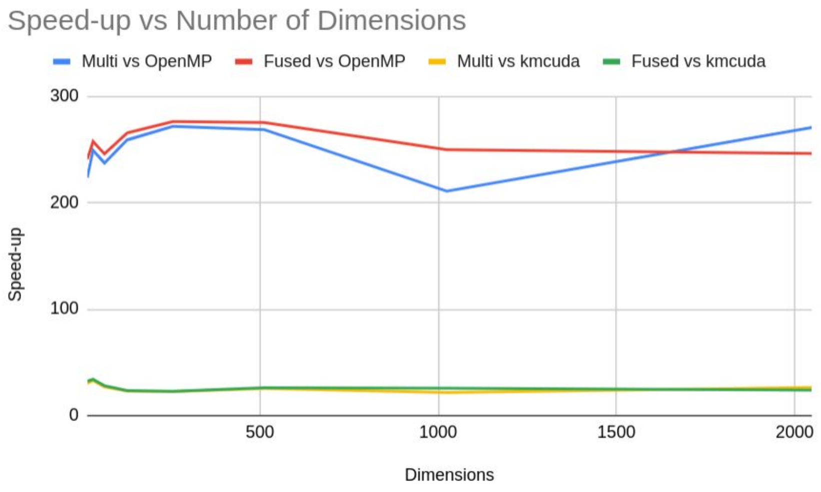
Varying number of points (for 1024 centroids, 32 dimensions)



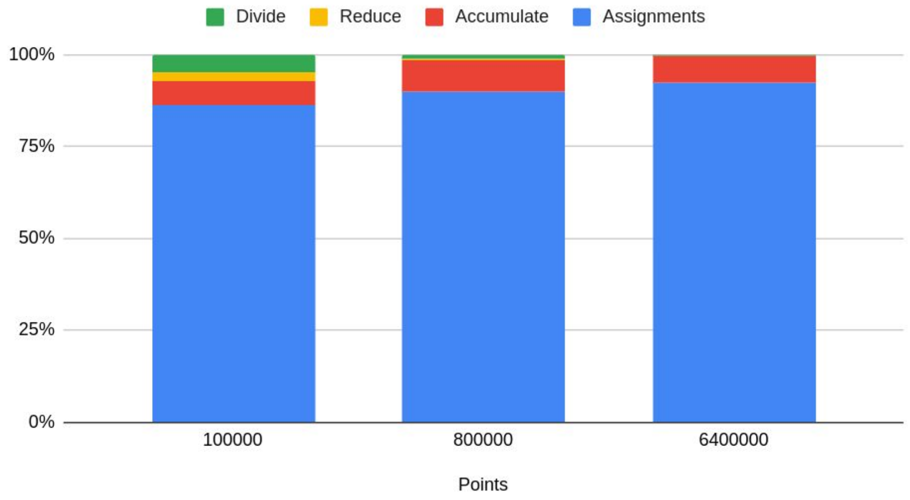


Varying number of centroids (for 1.6m points, 32 dimensions)

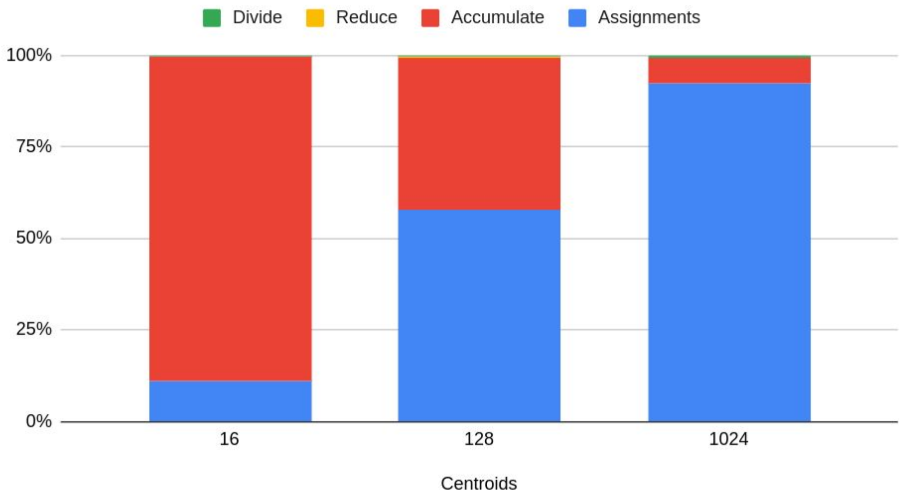




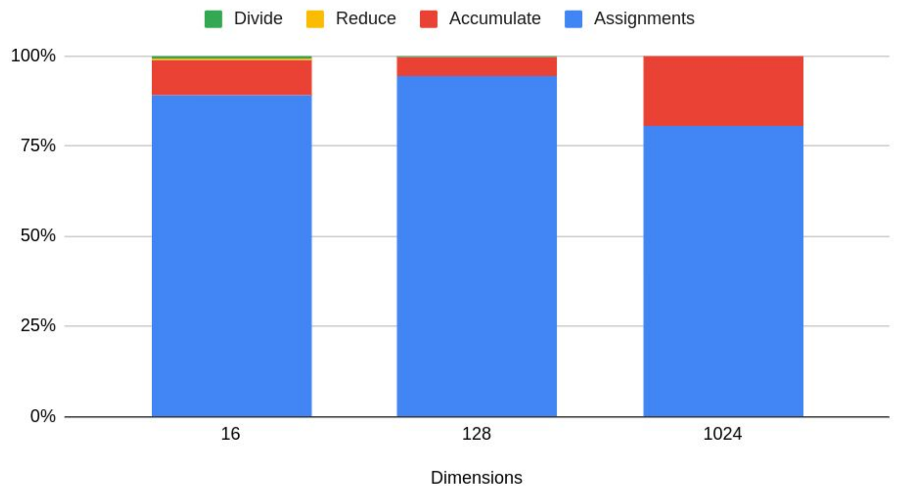
Varying number of dimensions
(for 1.6m points, 1024 centroids)



Breakdown for varying points for multi kernel (for 1024 centroids, 32 dimensions)

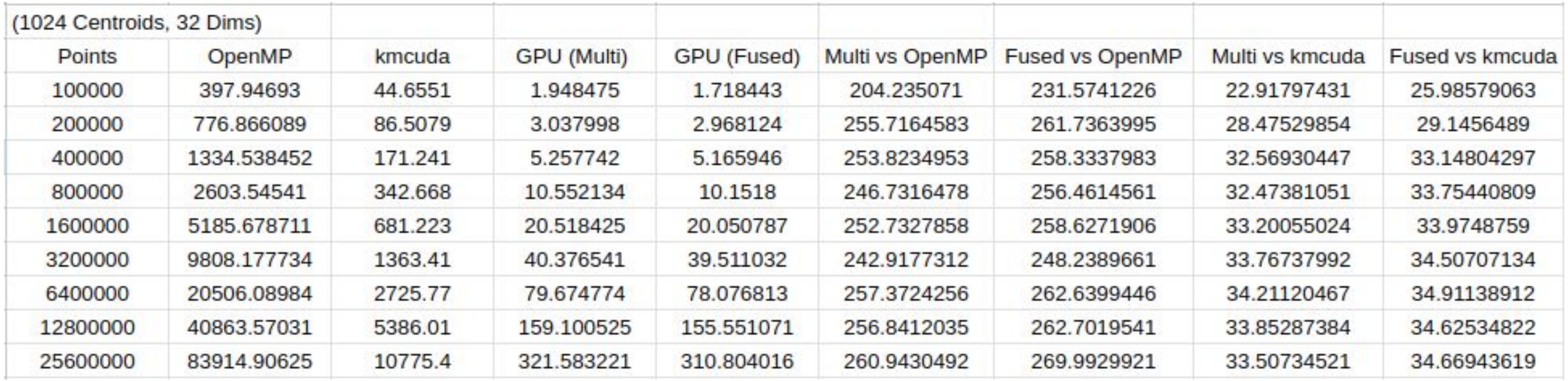


Breakdown for varying centroids for multi kernel (for 1.6m points, 32 dimensions)



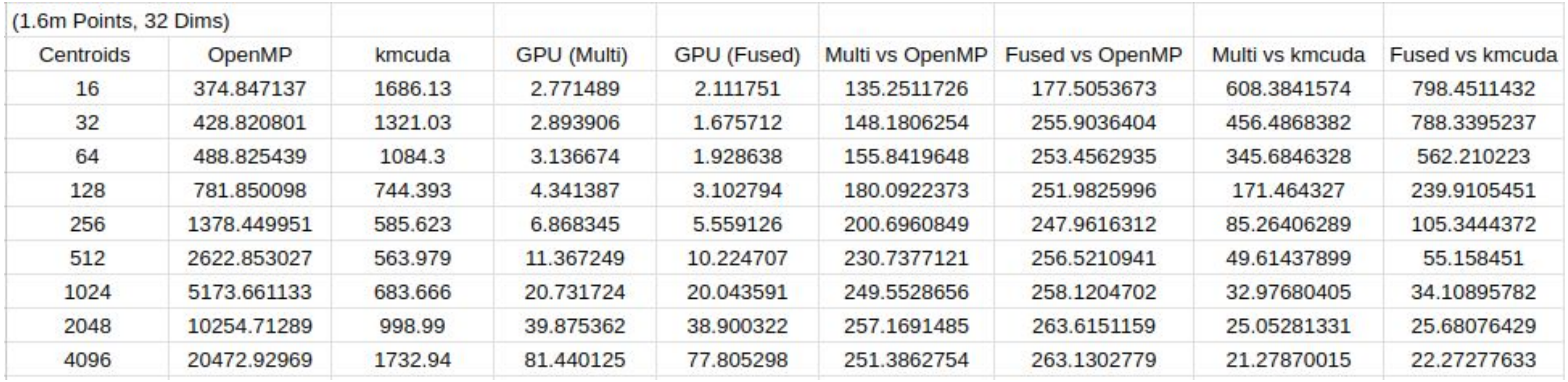
Breakdown for varying dimensions for multi kernel (for 1.6m points, 1024 centroids)

Thank You!



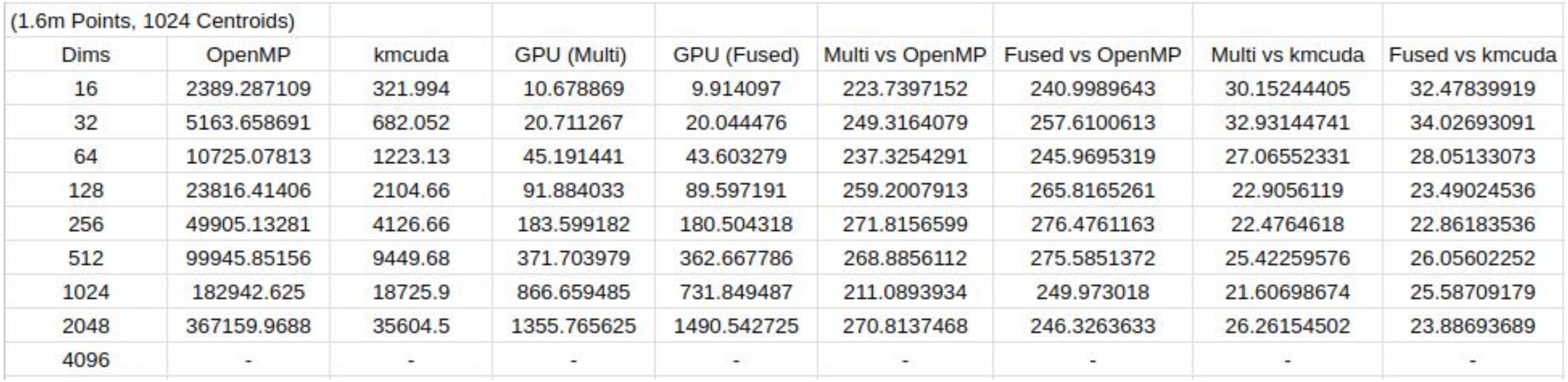
(1024 Centroids, 32 Dims)								
Points	OpenMP	kmcuda	GPU (Multi)	GPU (Fused)	Multi vs OpenMP	Fused vs OpenMP	Multi vs kmcuda	Fused vs kmcuda
100000	397.94693	44.6551	1.948475	1.718443	204.235071	231.5741226	22.91797431	25.98579063
200000	776.866089	86.5079	3.037998	2.968124	255.7164583	261.7363995	28.47529854	29.1456489
400000	1334.538452	171.241	5.257742	5.165946	253.8234953	258.3337983	32.56930447	33.14804297
800000	2603.54541	342.668	10.552134	10.1518	246.7316478	256.4614561	32.47381051	33.75440809
1600000	5185.678711	681.223	20.518425	20.050787	252.7327858	258.6271906	33.20055024	33.9748759
3200000	9808.177734	1363.41	40.376541	39.511032	242.9177312	248.2389661	33.76737992	34.50707134
6400000	20506.08984	2725.77	79.674774	78.076813	257.3724256	262.6399446	34.21120467	34.91138912
12800000	40863.57031	5386.01	159.100525	155.551071	256.8412035	262.7019541	33.85287384	34.62534822
25600000	83914.90625	10775.4	321.583221	310.804016	260.9430492	269.9929921	33.50734521	34.66943619

Varying number of points (for 1024 centroids, 32 dimensions)



(1.6m Points, 32 Dims)								
Centroids	OpenMP	kmcuda	GPU (Multi)	GPU (Fused)	Multi vs OpenMP	Fused vs OpenMP	Multi vs kmcuda	Fused vs kmcuda
16	374.847137	1686.13	2.771489	2.111751	135.2511726	177.5053673	608.3841574	798.4511432
32	428.820801	1321.03	2.893906	1.675712	148.1806254	255.9036404	456.4868382	788.3395237
64	488.825439	1084.3	3.136674	1.928638	155.8419648	253.4562935	345.6846328	562.210223
128	781.850098	744.393	4.341387	3.102794	180.0922373	251.9825996	171.464327	239.9105451
256	1378.449951	585.623	6.868345	5.559126	200.6960849	247.9616312	85.26406289	105.3444372
512	2622.853027	563.979	11.367249	10.224707	230.7377121	256.5210941	49.61437899	55.158451
1024	5173.661133	683.666	20.731724	20.043591	249.5528656	258.1204702	32.97680405	34.10895782
2048	10254.71289	998.99	39.875362	38.900322	257.1691485	263.6151159	25.05281331	25.68076429
4096	20472.92969	1732.94	81.440125	77.805298	251.3862754	263.1302779	21.27870015	22.27277633

Varying number of centroids
(for 1.6m points, 32 dimensions)



(1.6m Points, 1024 Centroids)								
Dims	OpenMP	kmcuda	GPU (Multi)	GPU (Fused)	Multi vs OpenMP	Fused vs OpenMP	Multi vs kmcuda	Fused vs kmcuda
16	2389.287109	321.994	10.678869	9.914097	223.7397152	240.9989643	30.15244405	32.47839919
32	5163.658691	682.052	20.711267	20.044476	249.3164079	257.6100613	32.93144741	34.02693091
64	10725.07813	1223.13	45.191441	43.603279	237.3254291	245.9695319	27.06552331	28.05133073
128	23816.41406	2104.66	91.884033	89.597191	259.2007913	265.8165261	22.9056119	23.49024536
256	49905.13281	4126.66	183.599182	180.504318	271.8156599	276.4761163	22.4764618	22.86183536
512	99945.85156	9449.68	371.703979	362.667786	268.8856112	275.5851372	25.42259576	26.05602252
1024	182942.625	18725.9	866.659485	731.849487	211.0893934	249.973018	21.60698674	25.58709179
2048	367159.9688	35604.5	1355.765625	1490.542725	270.8137468	246.3263633	26.26154502	23.88693689
4096	-	-	-	-	-	-	-	-

Varying number of dimensions
(for 1.6m points, 1024 centroids)

References

[1] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Yinyang K-Means: A Drop-In Replacement of the Classic K-Means with Consistent Speedup. In ICML.

[2] Clemens Lutz, Sebastian Breß, Tilmann Rabl, Steffen Zeuch, and Volker Markl. 2018. Efficient and Scalable k-Means on GPUs. In Datenbank-Spektrum volume 18, pages 157–169.

[3] Maliheh Heydarpour Shahrezaei and Reza Tavoli. 2019. Parallelization of Kmeans++ using CUDA. arXiv:1908.02136.

[4] Can Yang, Yin Li, and Fenhua Cheng. 2020. Accelerating k-Means on GPU with CUDA Programming. doi:10.1088/1757-899X/790/1/012036.

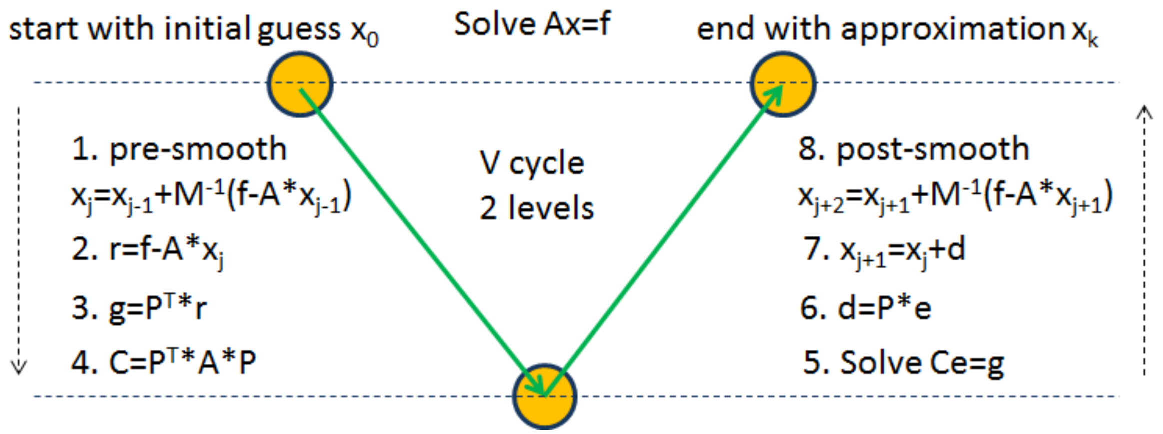
GPU-Accelerated Algebraic Multigrid Methods (AMG)

Bao Li

CSE 6230 – Spring 2023

Category

- Category:
  - AMG Solver (V-cycles): apply to structural optimization
  - the AMG algorithm solves the large (fine) linear system by cycling through levels composed of smaller (coarse) linear systems and finding updates that bring one closer to the exact solution

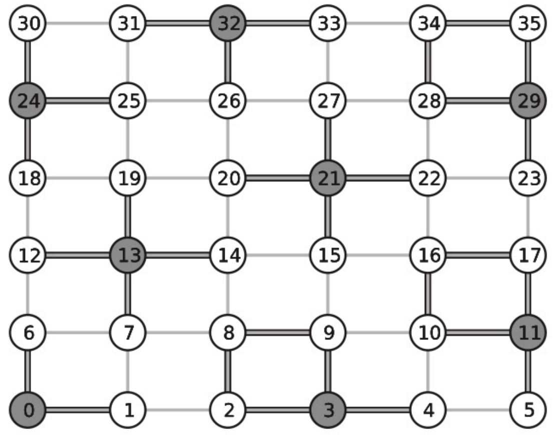


Problem Statement

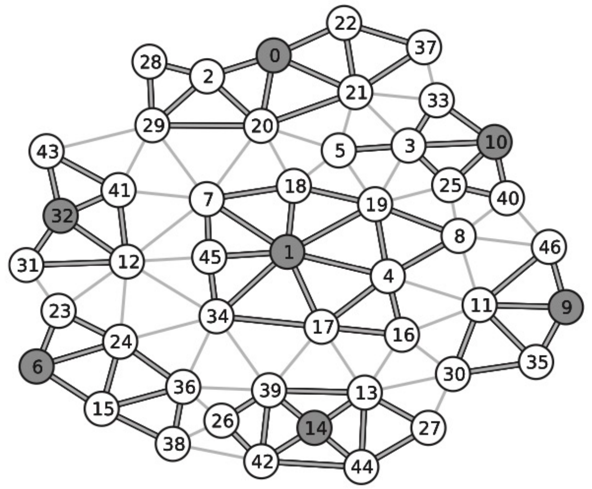
- Objective
  - **Large sparse** matrix from large scale structure optimization:

$$Ku = f$$

- Setup phase
  - Strength-of-connection
  - Aggregation
  - Construction of the tentative prolongation
  - Prolongation smoothing
- Solve phase
  - Pre-smooth
  - Solve
  - Post-smooth



(a) Structured mesh aggregates



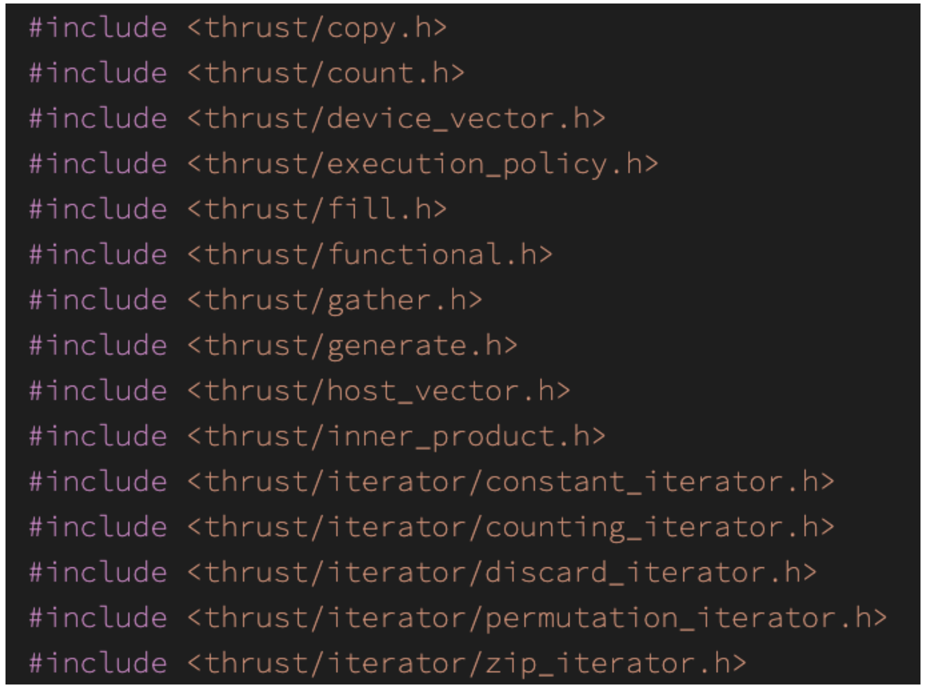
(b) Unstructured mesh aggregates

Performance Metric && Baselines

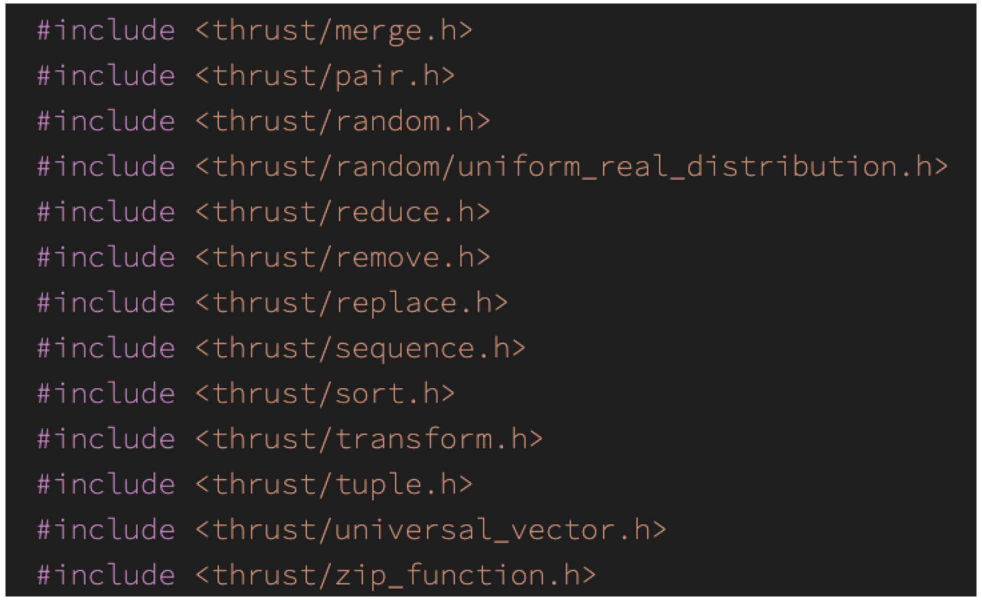
- Performance Metric
  - Timing: the speed of solving lager sparse linear system
  - Scaling: difference number of degree freedom for the structure
  - Breakdown: plot for setup and solve phase
- Baseline:
  1. Python: x = np.linalg.solve(A, b)
  2. C++: pure sequentially AMG with single CPU
- Accelerated:
  1. AMG + OpenMP
  2. AMG + GPU:Thrust

Proposed Solution

- AMG + OpenMP
- AMG + GPU:Thrust
  - Thrust: C++ standard template library for CUDA based on the (STL)



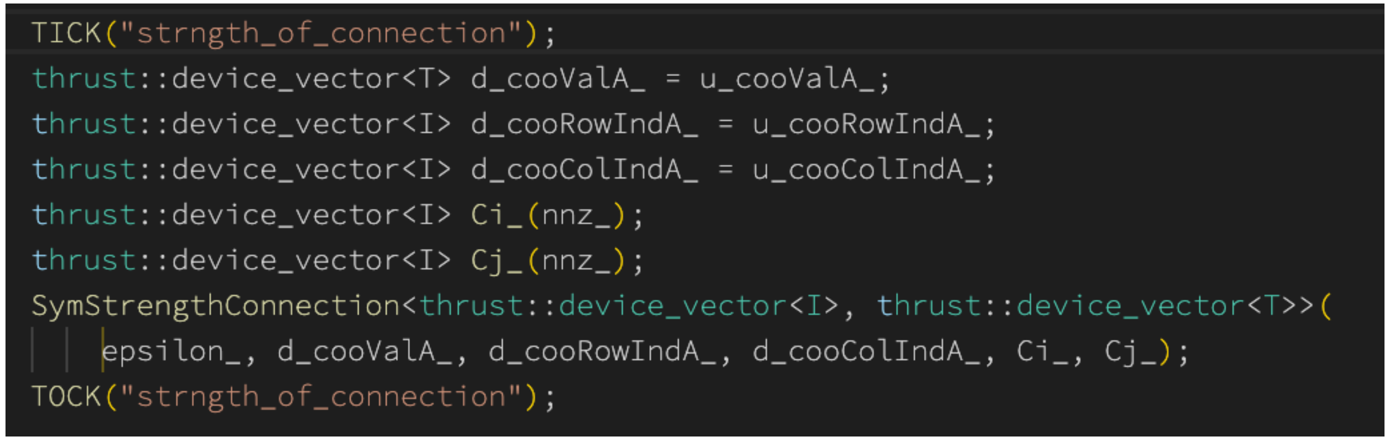
```
#include <thrust/copy.h>
#include <thrust/count.h>
#include <thrust/device_vector.h>
#include <thrust/execution_policy.h>
#include <thrust/fill.h>
#include <thrust/functional.h>
#include <thrust/gather.h>
#include <thrust/generate.h>
#include <thrust/host_vector.h>
#include <thrust/inner_product.h>
#include <thrust/iterator/constant_iterator.h>
#include <thrust/iterator/counting_iterator.h>
#include <thrust/iterator/discard_iterator.h>
#include <thrust/iterator/permutation_iterator.h>
#include <thrust/iterator/zip_iterator.h>
```



```
#include <thrust/merge.h>
#include <thrust/pair.h>
#include <thrust/random.h>
#include <thrust/random/uniform_real_distribution.h>
#include <thrust/reduce.h>
#include <thrust/remove.h>
#include <thrust/replace.h>
#include <thrust/sequence.h>
#include <thrust/sort.h>
#include <thrust/transform.h>
#include <thrust/tuple.h>
#include <thrust/universal_vector.h>
#include <thrust/zip_function.h>
```

Proposed Solution

- Data containers:
  - `thrust::host_vector<T>` stored in host memory
  - `thrust::device_vector<T>` lives in GPU device memory
  - `thrust::universal_vector<T>` both GPU and CPU can allocate
  - iterator: begin, end
  - the "=" operator can be used to copy data

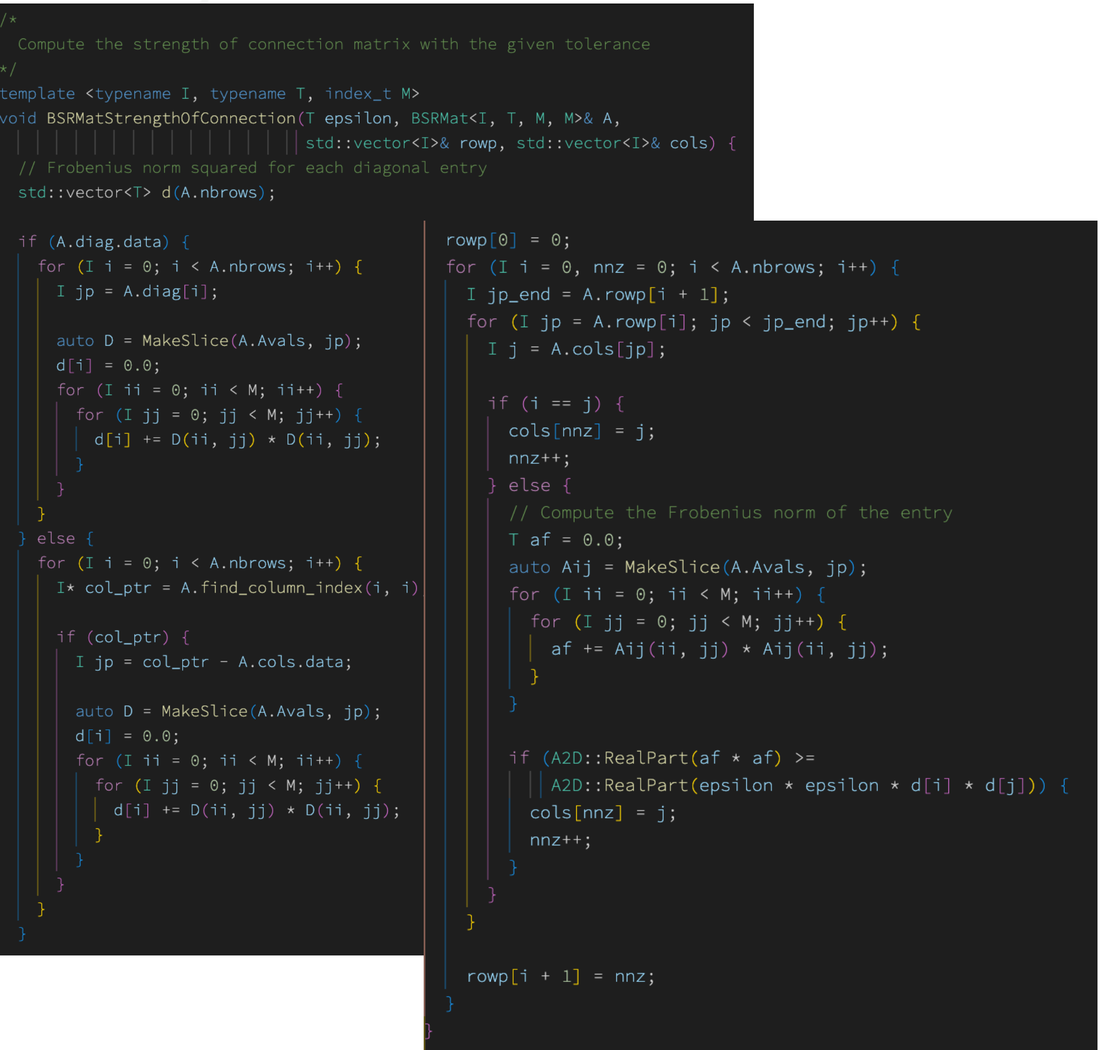


```
TICK("strngth_of_connection");
thrust::device_vector<T> d_cooValA_ = u_cooValA_;
thrust::device_vector<I> d_cooRowIndA_ = u_cooRowIndA_;
thrust::device_vector<I> d_cooColIndA_ = u_cooColIndA_;
thrust::device_vector<I> Ci_(nnz_);
thrust::device_vector<I> Cj_(nnz_);
SymStrengthConnection<thrust::device_vector<I>, thrust::device_vector<T>>(
    epsilon_, d_cooValA_, d_cooRowIndA_, d_cooColIndA_, Ci_, Cj_);
TOCK("strngth_of_connection");
```

Proposed Solution

```
ALGORITHM 2. Strength of connection: strength.

  parameters:  $A_k \equiv (I, J, V)$,  COO sparse matrix
  return:  $C_k \equiv (\hat{I}, \hat{J}, \hat{V})$,  COO sparse matrix

  $\mathcal{M} = \{0, \ldots, nnz(A) - 1\}$
  $D \leftarrow 0$
1 for $n \in \mathcal{M}$                                  {extract diagonal}
      if $I_n = J_n$
          $D(I_n) \leftarrow V_n$
2 for $n \in \mathcal{M}$                                  {check strength}
      if $|V_n| > \theta\sqrt{|D(I_n)| \cdot |D(J_n)|}$
          $(\hat{I}_{\hat{n}}, \hat{J}_{\hat{n}}, \hat{V}_{\hat{n}}) \leftarrow (I_n, J_n, V_n)$
```

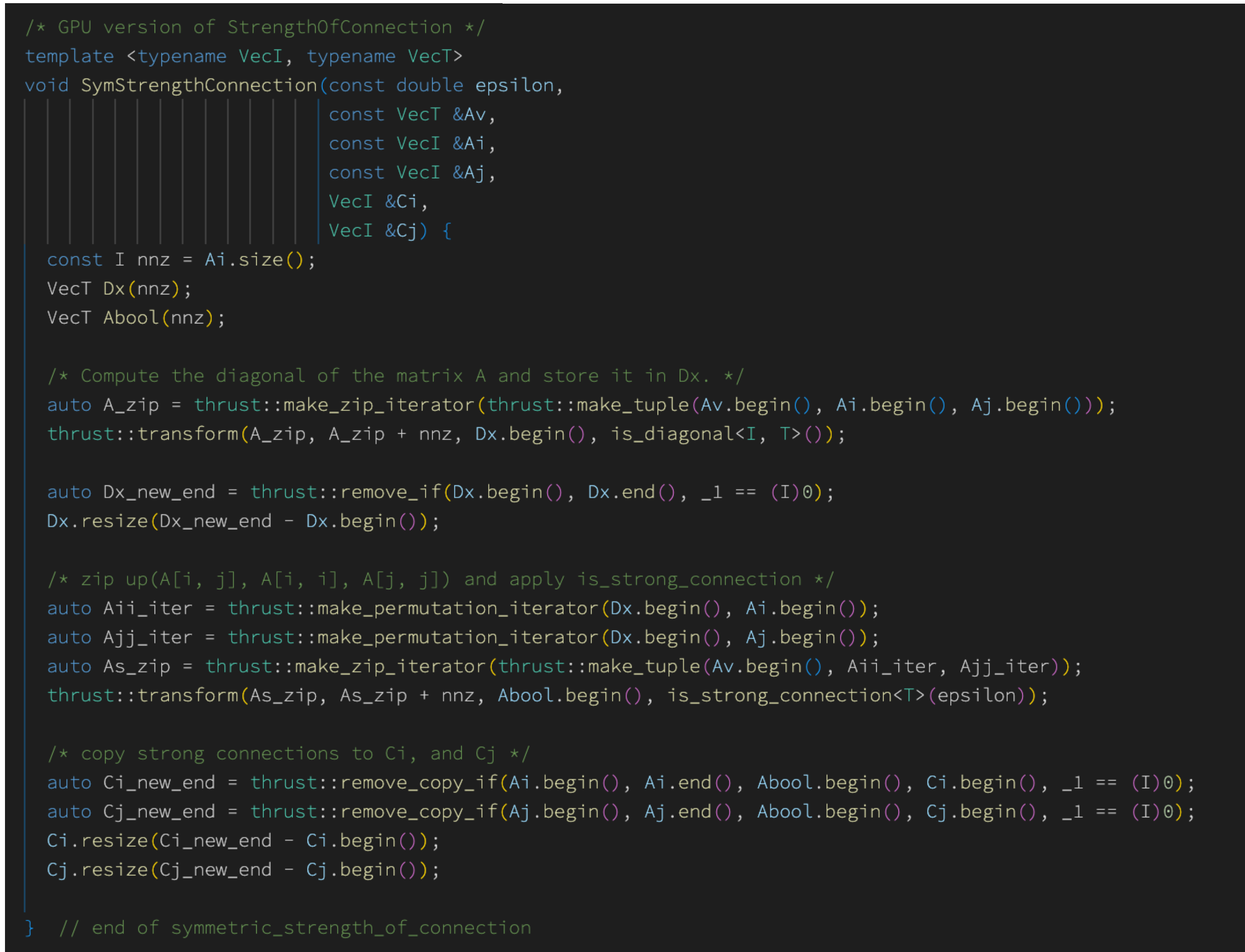
CPU+BRS format



```
/*
  Compute the strength of connection matrix with the given tolerance
*/
template <typename I, typename T, index_t M>
void BSRMatStrengthOfConnection(T epsilon, BSRMat<I, T, M, M>& A,
                                std::vector<I>& rowp, std::vector<I>& cols) {
  // Frobenius norm squared for each diagonal entry
  std::vector<T> d(A.nbrows);

  if (A.diag.data) {
    for (I i = 0; i < A.nbrows; i++) {
      I jp = A.diag[i];

      auto D = MakeSlice(A.Avals, jp);
      d[i] = 0.0;
      for (I ii = 0; ii < M; ii++) {
        for (I jj = 0; jj < M; jj++) {
          d[i] += D(ii, jj) * D(ii, jj);
        }
      }
    }
  } else {
    for (I i = 0; i < A.nbrows; i++) {
      I* col_ptr = A.find_column_index(i, i)

      if (col_ptr) {
        I jp = col_ptr - A.cols.data;

        auto D = MakeSlice(A.Avals, jp);
        d[i] = 0.0;
        for (I ii = 0; ii < M; ii++) {
          for (I jj = 0; jj < M; jj++) {
            d[i] += D(ii, jj) * D(ii, jj);
          }
        }
      }
    }
  }
```

```
  rowp[0] = 0;
  for (I i = 0, nnz = 0; i < A.nbrows; i++) {
    I jp_end = A.rowp[i + 1];
    for (I jp = A.rowp[i]; jp < jp_end; jp++) {
      I j = A.cols[jp];

      if (i == j) {
        cols[nnz] = j;
        nnz++;
      } else {
        // Compute the Frobenius norm of the entry
        T af = 0.0;
        auto Aij = MakeSlice(A.Avals, jp);
        for (I ii = 0; ii < M; ii++) {
          for (I jj = 0; jj < M; jj++) {
            af += Aij(ii, jj) * Aij(ii, jj);
          }
        }

        if (A2D::RealPart(af * af) >=
            A2D::RealPart(epsilon * epsilon * d[i] * d[j])) {
          cols[nnz] = j;
          nnz++;
        }
      }
    }

    rowp[i + 1] = nnz;
  }
}
```

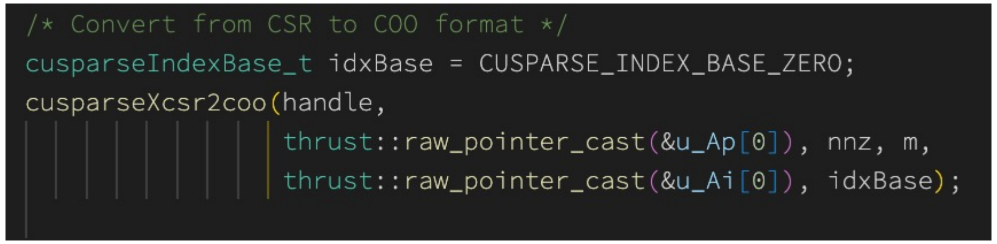
GPU+COO format



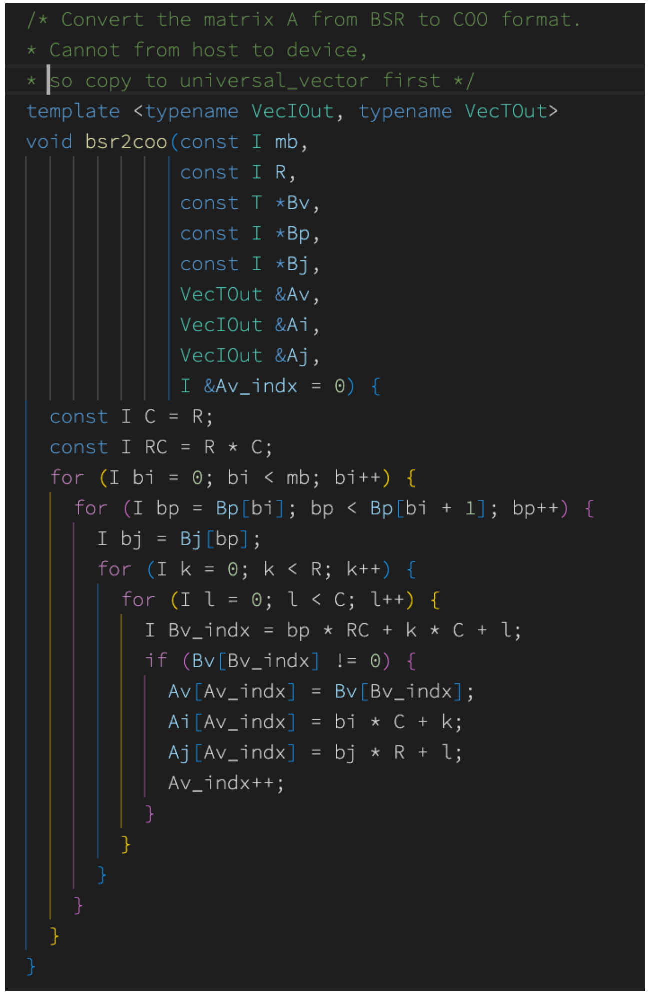
```
/* GPU version of StrengthOfConnection */
template <typename VecI, typename VecT>
void SymStrengthConnection(const double epsilon,
                           const VecT &Av,
                           const VecI &Ai,
                           const VecI &Aj,
                           VecI &Ci,
                           VecI &Cj) {
  const I nnz = Ai.size();
  VecT Dx(nnz);
  VecT Abool(nnz);

  /* Compute the diagonal of the matrix A and store it in Dx. */
  auto A_zip = thrust::make_zip_iterator(thrust::make_tuple(Av.begin(), Ai.begin(), Aj.begin()));
  thrust::transform(A_zip, A_zip + nnz, Dx.begin(), is_diagonal<I, T>());

  auto Dx_new_end = thrust::remove_if(Dx.begin(), Dx.end(), _1 == (I)0);
  Dx.resize(Dx_new_end - Dx.begin());

  /* zip up(A[i, j], A[i, i], A[j, j]) and apply is_strong_connection */
  auto Aii_iter = thrust::make_permutation_iterator(Dx.begin(), Ai.begin());
  auto Ajj_iter = thrust::make_permutation_iterator(Dx.begin(), Aj.begin());
  auto As_zip = thrust::make_zip_iterator(thrust::make_tuple(Av.begin(), Aii_iter, Ajj_iter));
  thrust::transform(As_zip, As_zip + nnz, Abool.begin(), is_strong_connection<T>(epsilon));

  /* copy strong connections to Ci, and Cj */
  auto Ci_new_end = thrust::remove_copy_if(Ai.begin(), Ai.end(), Abool.begin(), Ci.begin(), _1 == (I)0);
  auto Cj_new_end = thrust::remove_copy_if(Aj.begin(), Aj.end(), Abool.begin(), Cj.begin(), _1 == (I)0);
  Ci.resize(Ci_new_end - Ci.begin());
  Cj.resize(Cj_new_end - Cj.begin());

}  // end of symmetric_strength_of_connection
```

Datasets

- Data Store Format
  - Coordinate Format (COO)
  - Block Compressed Sparse Row Format (BSR)



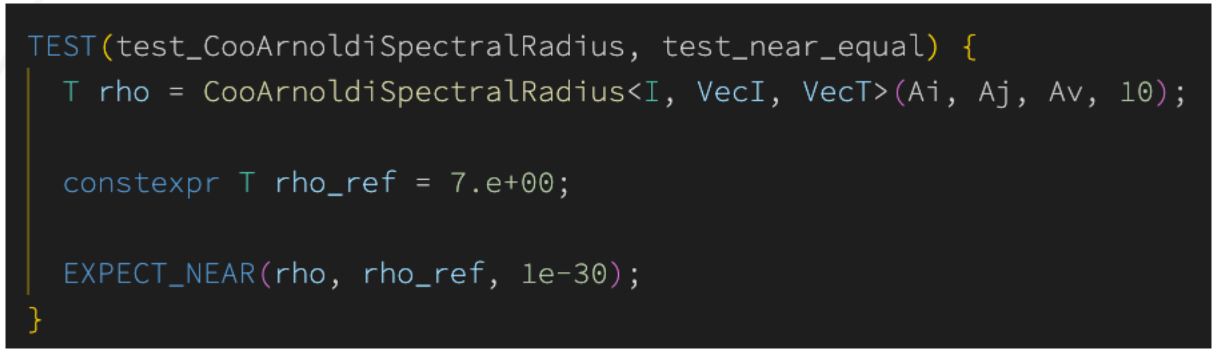
```
/* Convert from CSR to COO format */
cusparseIndexBase_t idxBase = CUSPARSE_INDEX_BASE_ZERO;
cusparseXcsr2coo(handle,
                 thrust::raw_pointer_cast(&u_Ap[0]), nnz, m,
                 thrust::raw_pointer_cast(&u_Ai[0]), idxBase);
```



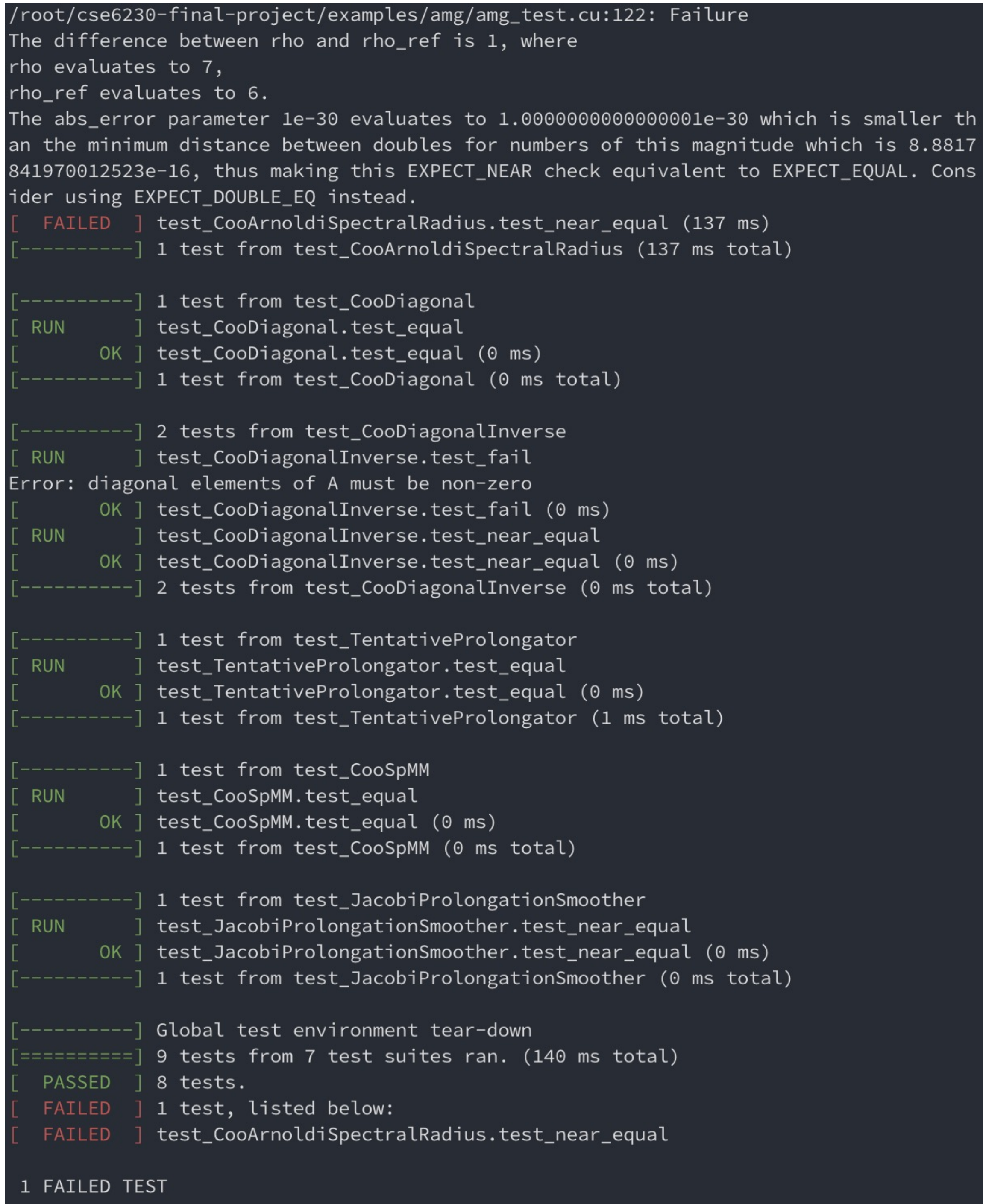
```
/* Convert the matrix A from BSR to COO format.
 * Cannot from host to device,
 * so copy to universal_vector first */
template <typename VecIOut, typename VecTOut>
void bsr2coo(const I mb,
             const I R,
             const T *Bv,
             const I *Bp,
             const I *Bj,
             VecTOut &Av,
             VecIOut &Ai,
             VecIOut &Aj,
             I &Av_indx = 0) {
  const I C = R;
  const I RC = R * C;
  for (I bi = 0; bi < mb; bi++) {
    for (I bp = Bp[bi]; bp < Bp[bi + 1]; bp++) {
      I bj = Bj[bp];
      for (I k = 0; k < R; k++) {
        for (I l = 0; l < C; l++) {
          I Bv_indx = bp * RC + k * C + l;
          if (Bv[Bv_indx] != 0) {
            Av[Av_indx] = Bv[Bv_indx];
            Ai[Av_indx] = bi * C + k;
            Aj[Av_indx] = bj * R + l;
            Av_indx++;
          }
        }
      }
    }
  }
}
```

Validation of solution

- GoogleTest
  - GoogleTest is Google’s C++ testing and mocking framework
  - `#include <gtest/gtest.h>`

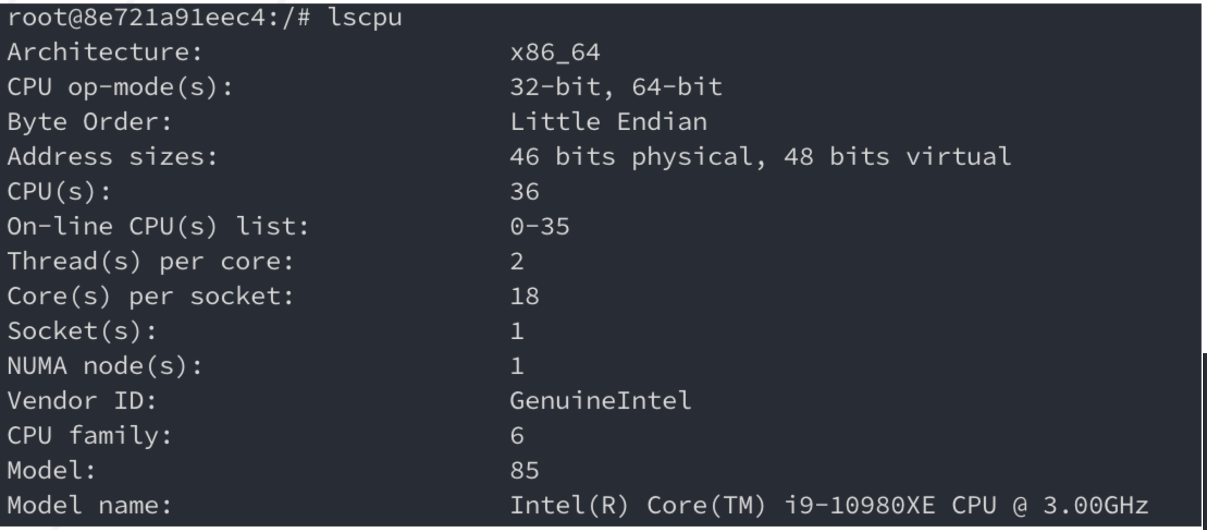


```
TEST(test_CooArnoldiSpectralRadius, test_near_equal) {
  T rho = CooArnoldiSpectralRadius<I, VecI, VecT>(Ai, Aj, Av, 10);

  constexpr T rho_ref = 7.e+00;

  EXPECT_NEAR(rho, rho_ref, 1e-30);
}
```



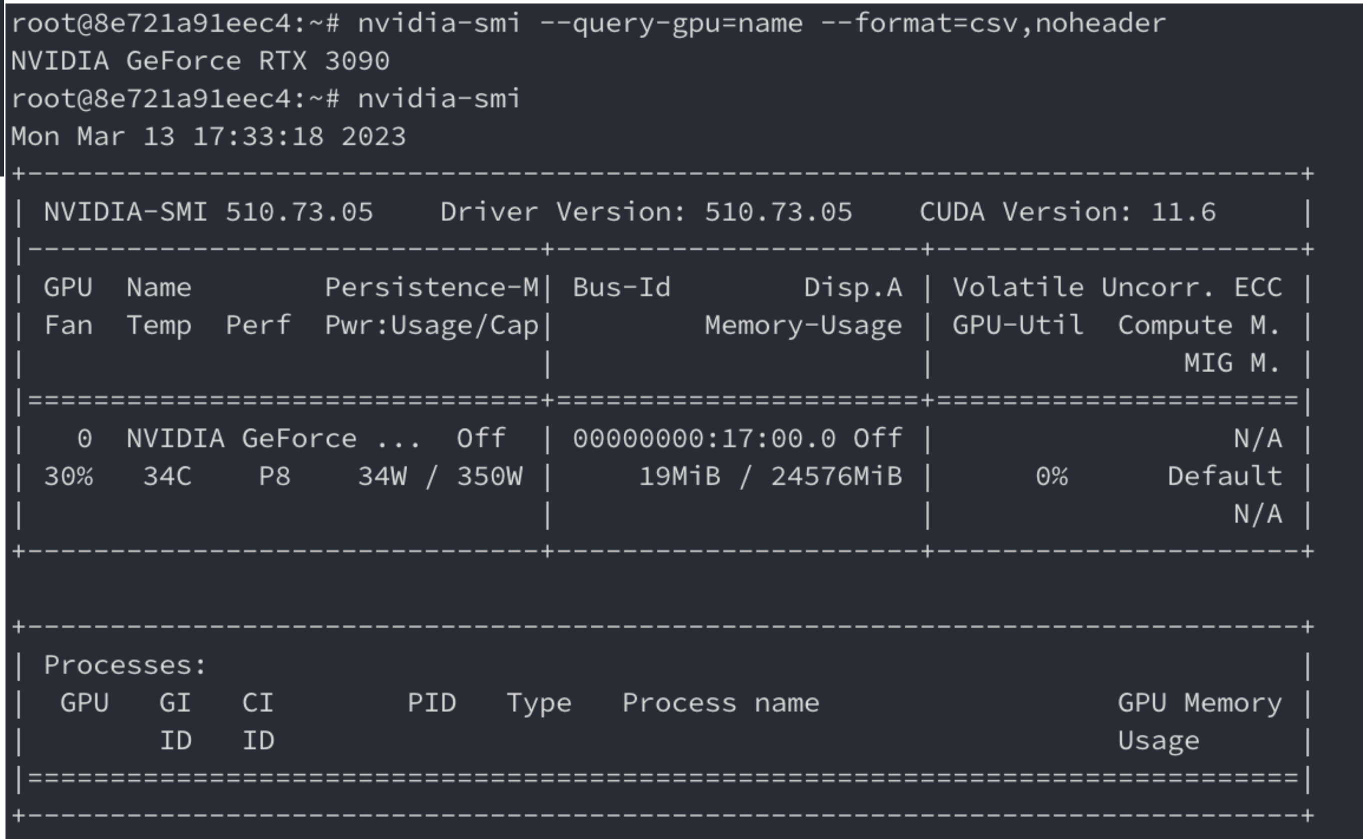
```
/root/cse6230-final-project/examples/amg/amg_test.cu:122: Failure
The difference between rho and rho_ref is 1, where
rho evaluates to 7,
rho_ref evaluates to 6.
The abs_error parameter 1e-30 evaluates to 1.0000000000000001e-30 which is smaller th
an the minimum distance between doubles for numbers of this magnitude which is 8.8817
841970012523e-16, thus making this EXPECT_NEAR check equivalent to EXPECT_EQUAL. Cons
ider using EXPECT_DOUBLE_EQ instead.
[  FAILED  ] test_CooArnoldiSpectralRadius.test_near_equal (137 ms)
[----------] 1 test from test_CooArnoldiSpectralRadius (137 ms total)

[----------] 1 test from test_CooDiagonal
[ RUN      ] test_CooDiagonal.test_equal
[       OK ] test_CooDiagonal.test_equal (0 ms)
[----------] 1 test from test_CooDiagonal (0 ms total)

[----------] 2 tests from test_CooDiagonalInverse
[ RUN      ] test_CooDiagonalInverse.test_fail
Error: diagonal elements of A must be non-zero
[       OK ] test_CooDiagonalInverse.test_fail (0 ms)
[ RUN      ] test_CooDiagonalInverse.test_near_equal
[       OK ] test_CooDiagonalInverse.test_near_equal (0 ms)
[----------] 2 tests from test_CooDiagonalInverse (0 ms total)

[----------] 1 test from test_TentativeProlongator
[ RUN      ] test_TentativeProlongator.test_equal
[       OK ] test_TentativeProlongator.test_equal (0 ms)
[----------] 1 test from test_TentativeProlongator (1 ms total)

[----------] 1 test from test_CooSpMM
[ RUN      ] test_CooSpMM.test_equal
[       OK ] test_CooSpMM.test_equal (0 ms)
[----------] 1 test from test_CooSpMM (0 ms total)

[----------] 1 test from test_JacobiProlongationSmoother
[ RUN      ] test_JacobiProlongationSmoother.test_near_equal
[       OK ] test_JacobiProlongationSmoother.test_near_equal (0 ms)
[----------] 1 test from test_JacobiProlongationSmoother (0 ms total)

[----------] Global test environment tear-down
[==========] 9 tests from 7 test suites ran. (140 ms total)
[  PASSED  ] 8 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] test_CooArnoldiSpectralRadius.test_near_equal

 1 FAILED TEST
```

Test Bed



```
root@8e721a91eec4:/# lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          36
On-line CPU(s) list:             0-35
Thread(s) per core:              2
Core(s) per socket:              18
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           85
Model name:                      Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz
```

Intel(R) Core(TM) i9-10980XE CPU
18 Core 36 threads



```
root@8e721a91eec4:~# nvidia-smi --query-gpu=name --format=csv,noheader
NVIDIA GeForce RTX 3090
root@8e721a91eec4:~# nvidia-smi
Mon Mar 13 17:33:18 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.73.05    Driver Version: 510.73.05    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:17:00.0 Off |                  N/A |
| 30%   34C    P8    34W / 350W |     19MiB / 24576MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

NVIDIA GeForce RTX 3090
CUDA Version: 11.6

Results:

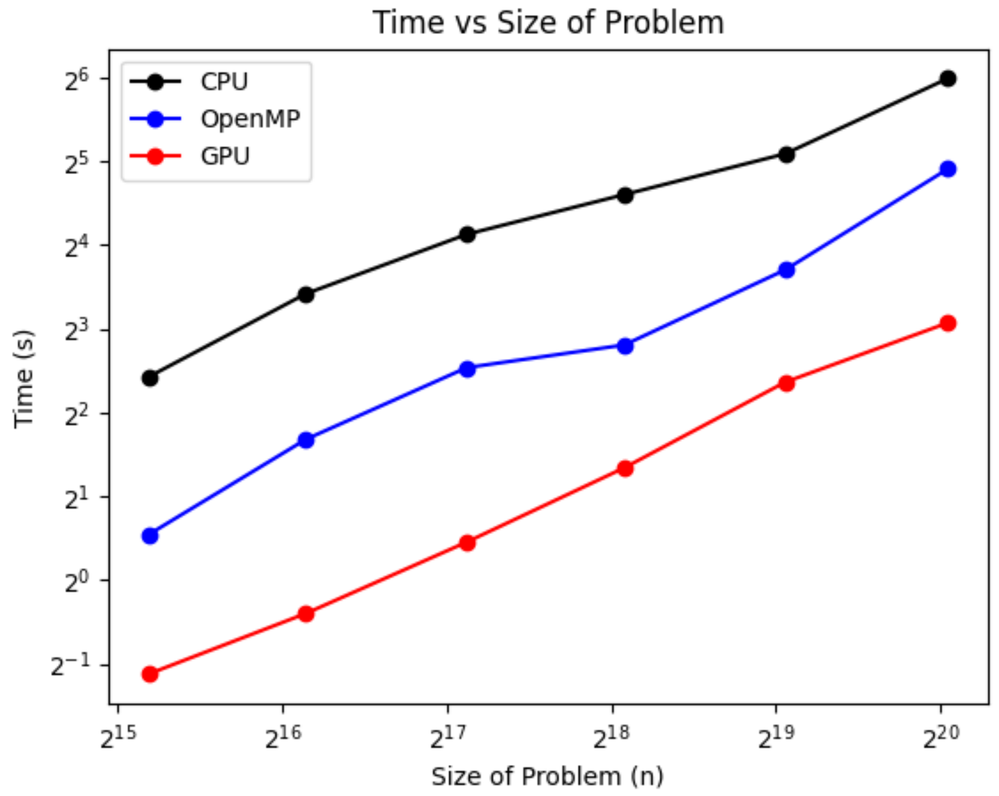
- Single CPU vs OpenMP vs GPU:Thrust
  - For the case: n = 37281, nnz = 924385 (non-zero elements)

Method	Time (s)
np.linalg.solve	103.57
Single CPU	5.36431
OpenMP	1.45374
GPU:Thrust	0.456715

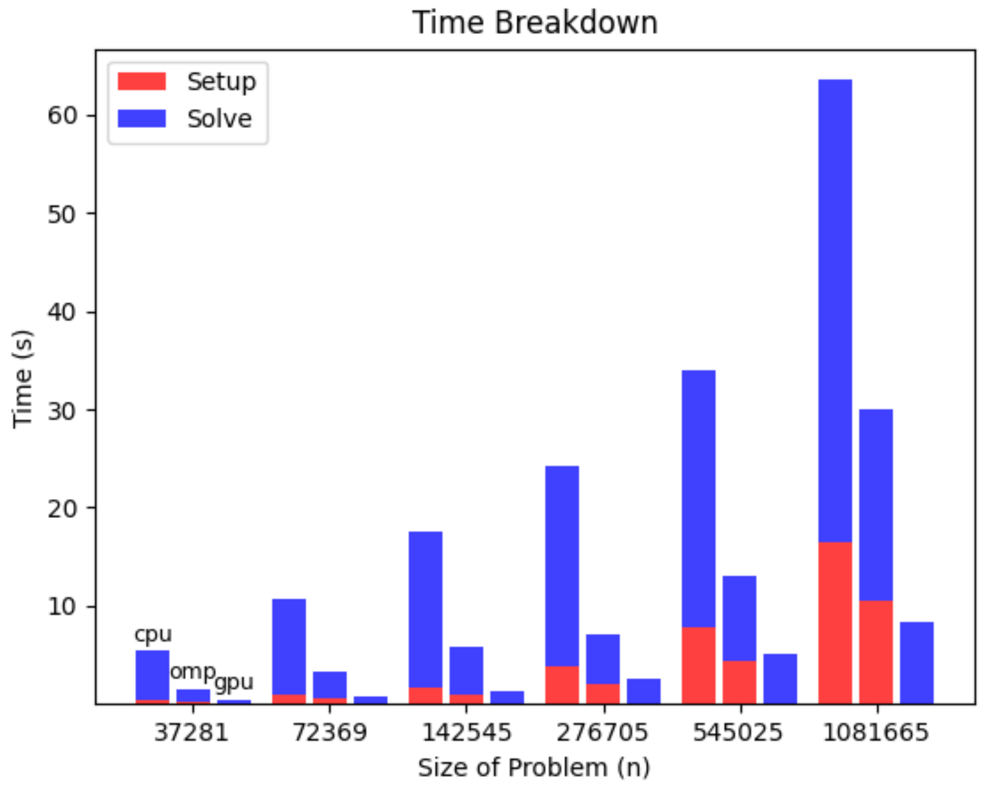
It is not possible to achieve 100% parallelization of the AMG algorithm.

Result

Strong Scaling



Breakdown: setup vs solve phase



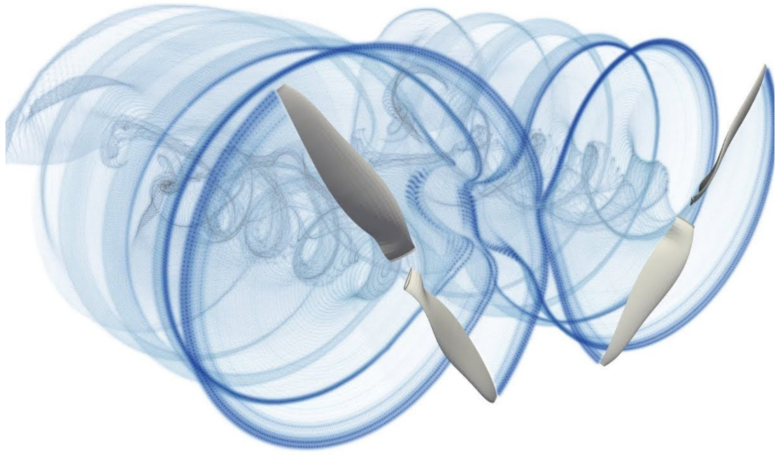
Setup phase have deeper parallelization on a GPU

GPU-Accelerated Vortex Particle Method (VPM)

Shreyas Ashok, Anand Radhakrishnan, Russell Newton

Introduction

- Vortex particle method (VPM) is a Computational Fluid Dynamics (CFD) technique used to solve the Euler or Navier-Stokes fluid equations of motion.
- Lagrangian approach—track individual particles of vorticity
  - In contrast with traditional Eulerian approach - discretize domain into a grid
- We reproduced simple 2D VPM using optimization techniques for single CPU and single GPU



Vortex Particle Method used for Multirotor Interaction Simulation

Alvarez, E. J., and Ning, A., "Development of a Vortex Particle Code for the Modeling of Wake Interaction in Distributed Propulsion," AIAA Applied Aerodynamics Conference, Atlanta, GA, Jun. 2018. doi:10.2514/6.2018-3646

Baseline and Performance Metrics

Baseline:

- Sequential N-Body simulation of Taylor-Green Vortex on CPU
- Validated against analytical Taylor-Green Vortex solution

Performance Metrics

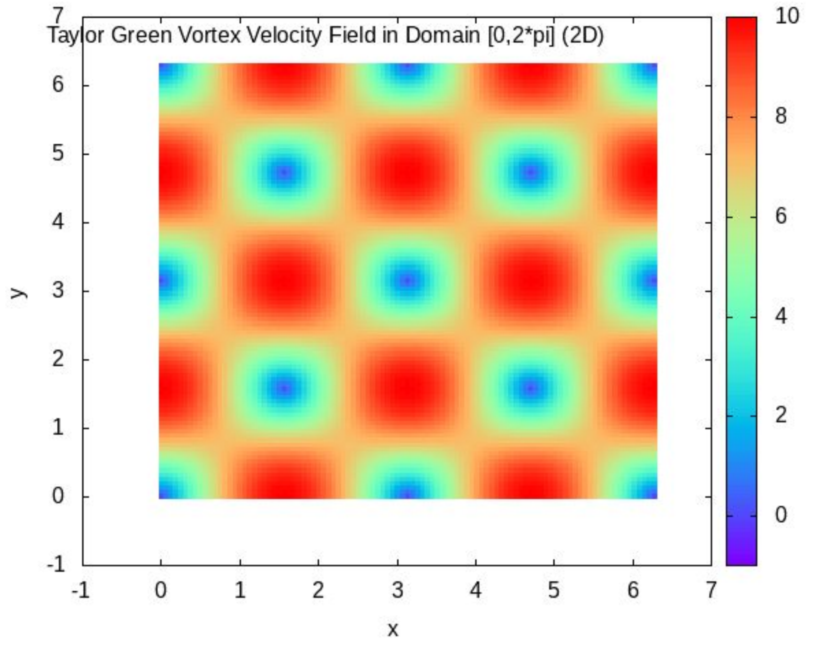
- Time per step of simulation
- Speedup versus baseline sequential CPU code
- Accuracy to analytical solution
- Accuracy of reduced-order problem versus full N-body solution

Baseline - Taylor Green Vortex

- The Taylor-Green Vortex is an analytical solution to the Navier-Stokes equations

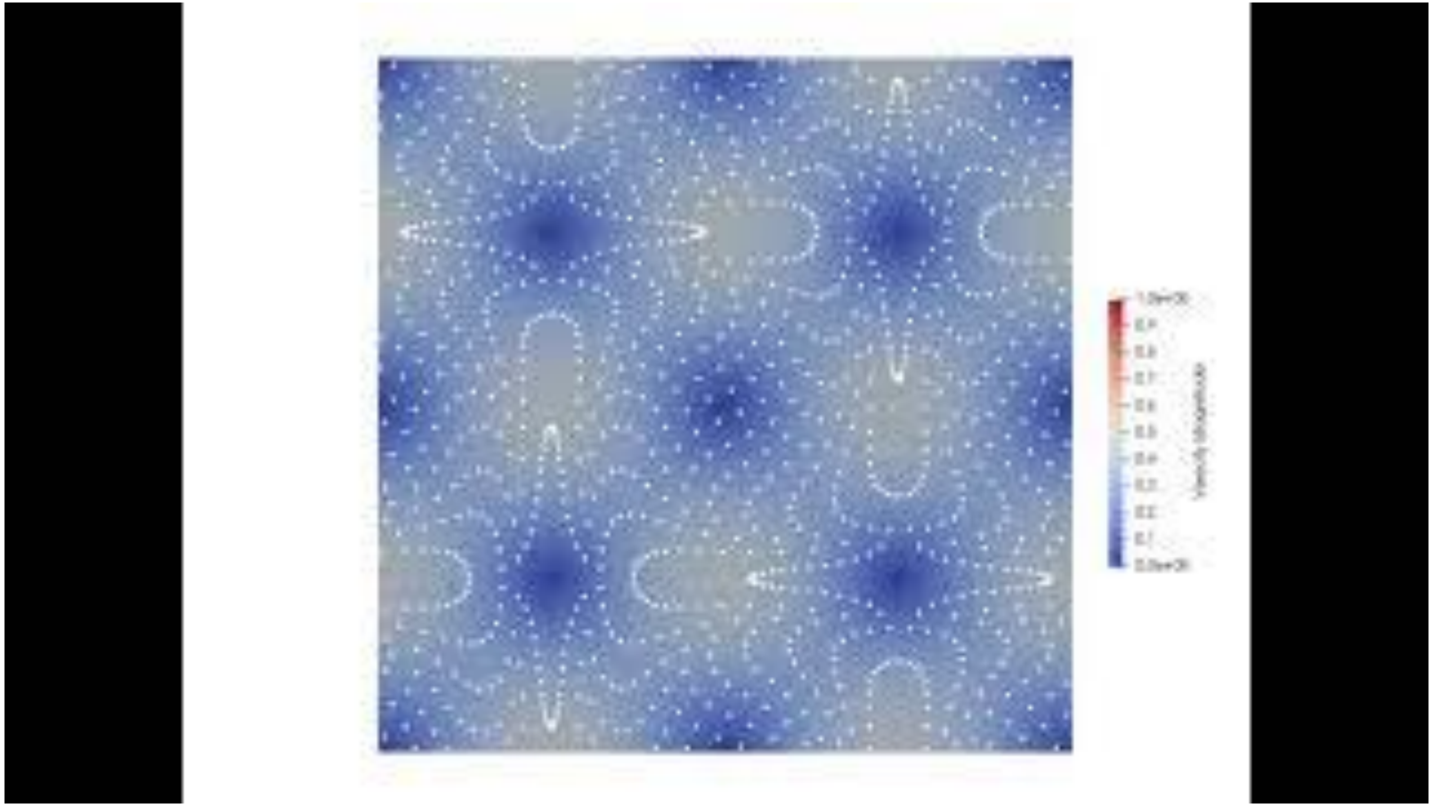
- 2D unsteady flow, periodic boundary conditions

- Simple boundary conditions and known analytical solution make this the perfect flow on which to test our VPM code!



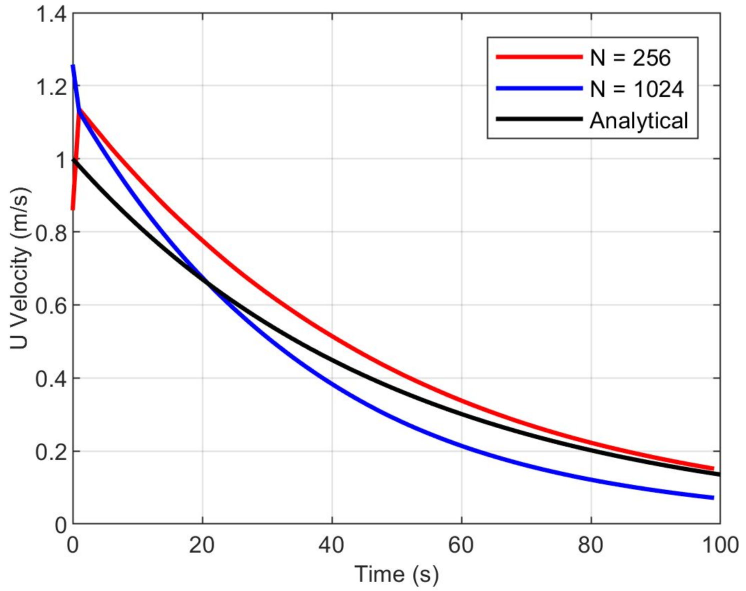
Taylor-Green Vortex Visualization
(Reproduced from Wikipedia)

Solution Video (link)



Validation of Solution

- VPM code represents expected behavior of Taylor-Green Vortex well!
    - Velocity contours visualized with Paraview and compared to known solution
- Dissipation rate is roughly in line with analytical values, although not perfect
    - Plot on right shows U Velocity at selected measurement point in domain
- Issue we encountered: “particle volume”
    - This parameter was not well-defined in the literature we used as reference to write the code; we probably have implemented this slightly wrong
    - Had to add a magic “multiplier value” of 0.1 to our particle volume to achieve roughly correct dissipation rates
- Issues don’t affect HPC side of things



Solution

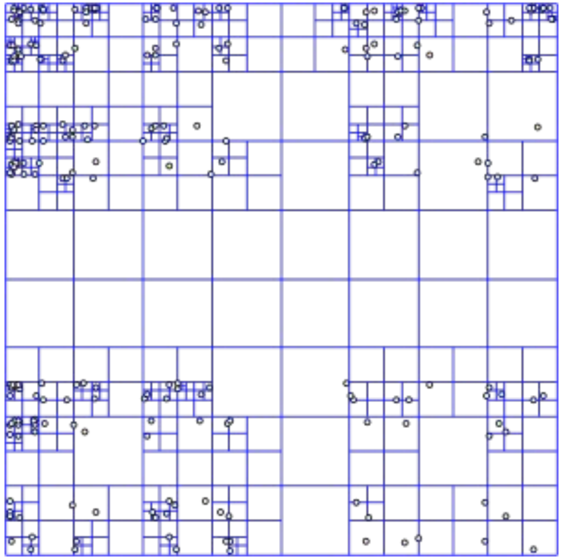
- Baseline solution: sequential code on CPU
- Accelerated solutions:
  - Originally wanted to try a distributed memory approach, but got complicated really fast
    - In raw form, VPM is an N-Body problem; a distributed memory approach of the raw form would require all-to-all communication of all N particles → **hugely impractical!**
  - To make VPM practical on distributed memory → need to reduce the order of the problem.
  - Tree-code approach, which helps group particles together into clusters, reduces the problem to O(NlogN).
  - On distributed memory system, theoretically possible to limit communication to higher branches of the tree

Solution

- Baseline solution: sequential code on CPU
- Accelerated solutions:
    - Implemented a tree code approach on CPU
    - Implemented n-body problem on GPU
        - It’s great for doing simple things fast
    - Attempted to implement the tree code on GPU
        - In progress
    - Did not achieve distributed memory solution in the time available; however, the code we developed can serve as a base for a distributed implementation.

Tree Code

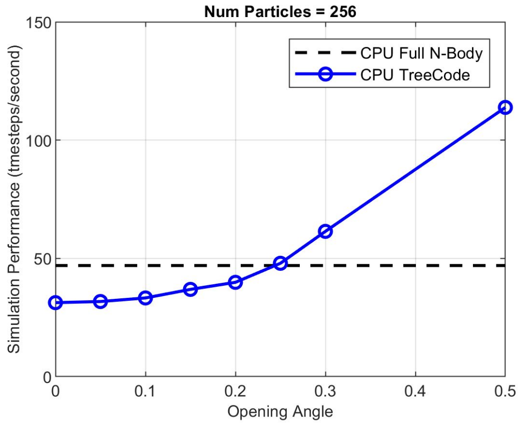
- Split the domain into quadrants
- Keep subdividing until each particle is in its own quadrant
- Subdivisions define a tree that can then be searched down
  - Inner nodes contain the centroidal position and vorticity of all particles in their children
- The approximation with inner nodes can be used instead of an individual particle if (quadrant size) / (distance to centroid) < (threshold)
- Changes the problem from quadratic time to loglinear time

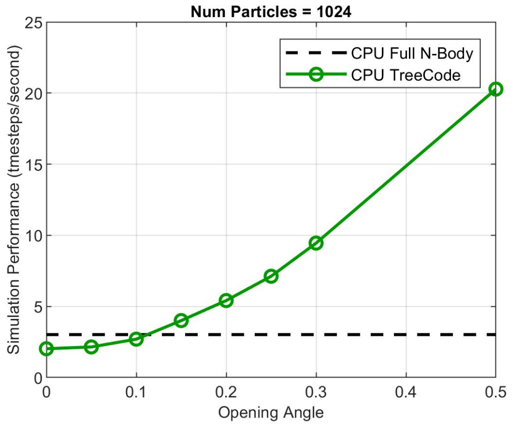


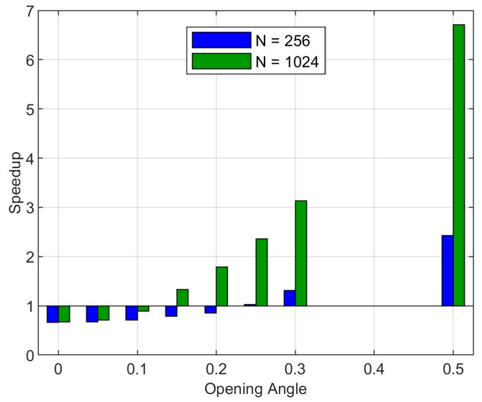
Quadtree Graphical Representation
(Reproduced from Wikipedia)

CPU Tree Code: Performance Results

- For larger particle numbers and larger groupings, tree code provides large speedup.
- Overhead of building the tree is not worth it for small numbers of particles and small opening angles (less clustering)





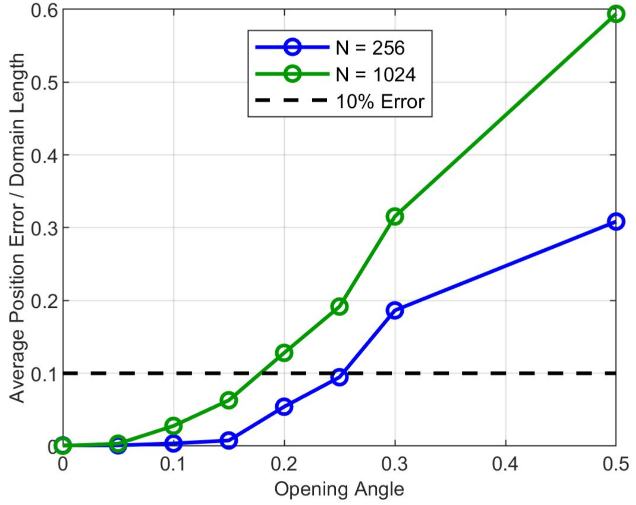


CPU Tree Code: Accuracy Results

- Tree approach is inherently an approximation of the true N-body problem
- Results show average particle position error versus opening angle, as compared to full N-body simulation.
    - How much accuracy did we lose because of the tree approximation?

- Opening Angle = 0.2 seems to be the limit at which error starts to grow dramatically.

- Around Opening Angle = 0.3, the simulation goes unstable for higher particle counts.



GPU Implementation (N-body problem)

- GPU offloading via OpenACC with NVHPC 22 compiler
- Simplify data structures to enable compiler optimizations
- Struct-of-arrays preferred over Array-of-structs for GPUs
- Nested parallelization with OpenACC gangs and vectors
- Manual inlining of subroutines for higher speedups
- ~1000X speedup using V100 GPU over Xeon Gold CPU on PACE (Phoenix)
- 67.81% SM utilization for GPU kernel with N = 4096
- Compute bound with large arithmetic intensity; 30% of Peak FLOPs
- High L1 (77%) and L2 (99%) Cache hit rate

GPU speedups; NVIDIA V100 v/s Intel Xeon Gold (-O2)

N	CPU (ms)	GPU (ms)	**Speedup**
256	2.45	0.02108	**115**
625	14.086	0.0386	**365**
1024	37.0801	0.07024	**528**
1600	88.8086	0.14184	**627**
2500	213.625	0.30168	**708**
4096	732.165	0.7404	**990**

GPU + Treecode

- Treecode data structures inherently sequential; not suitable for accelerators
- Need to conduct Depth-First Search (DFS) prior to GPU parallelization
- Current implementation does not warrant efficient utilization of GPUs

Alternate approach (work in progress)

- Rather than implement the full tree on GPUs, just implement a single level grid by dividing the domain into a 16x16 region (note: decomposition amount is tunable).
- Assign one thread per region, loop through all N particles, and assign each one to a box.
- Continue as per the tree code approach.
- This approach leads to less clustering efficiency due to there only being one level of the tree, but may be more effective on GPUs due to better parallelization.
- Initial implementation is in progress–still working out some bugs…

Complications

- After getting a successful tree code implementation, we weren’t able to finish a distributed approach
  - How to keep each process from requiring access to the entire tree
  - What does that communication look like?
- Compiler errors with nvhpc when trying to validate the GPU code
  - Streams weren’t behaving correctly
- Getting the treecode to work on the GPU required a complete revamp of the data structure
  - Big array with integer indices to children
- Lots of segfaults

Testbeds

- GPU Computing Results conducted on Tesla V100 on Phoenix PACE cluster

- CPU results taken on Intel® Core™ i9-10980XE CPU @ 3.00GHz, on lab workstation

Accuracy Errors

- We used simple Euler first-order forward integration in time
    - Runge-Kutta would have likely improved our results considerably
- The tree code seems to reduce overall energy dissipation
    - Possible our approximation of vorticity at cell center of mass is slightly off
- Papers we found didn’t explain a few things very well:
    - Particle mass and volume
    - We had to introduce a fudge factor for initial vorticity

References

Meldgaard, A., Darkner, S., & Erleben, K. (2022). Fast Vortex Particle Method for Fluid-Character Interaction. Graphics Interface 2022. Retrieved from https://openreview.net/forum?id=BrBIpeYNTMc

Marchevsky, I., Sokol, K., Ryatina, E., & Izmailova, Y. (2023). The VM2D Open Source Code for Two-Dimensional Incompressible Flow Simulation by Using Fully Lagrangian Vortex Particle Methods. Axioms, 12(3). doi:10.3390/axioms12030248

He, C., & Zhao, J. (2009). Modeling Rotor Wake Dynamics with Viscous Vortex Particle Method. AIAA Journal, 47(4), 902–915. doi:10.2514/1.36466

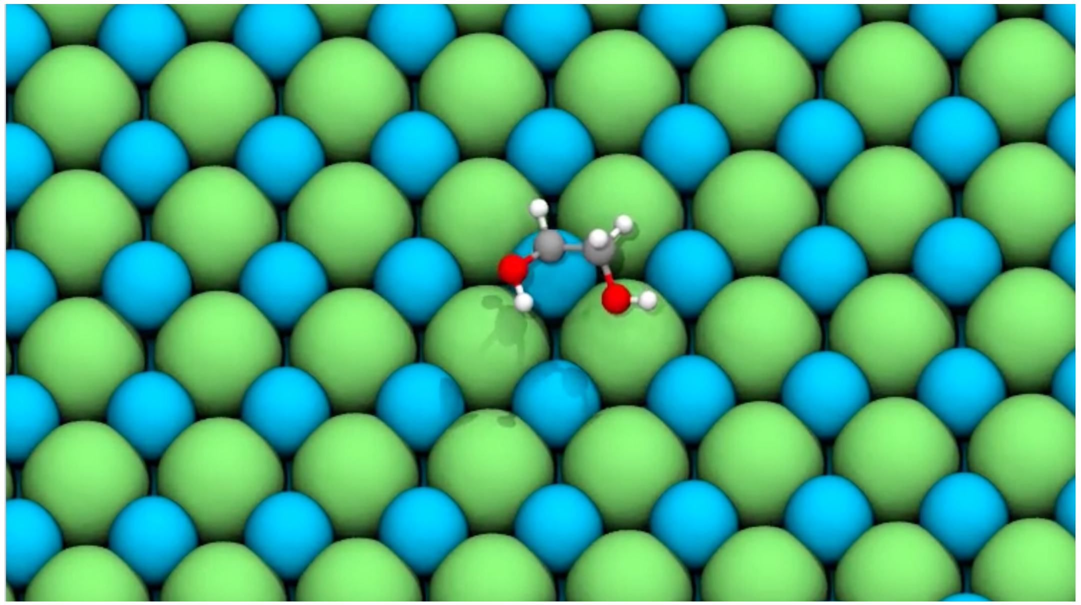
Alvarez, E. J., and Ning, A. (2018). Development of a Vortex Particle Code for the Modeling of Wake Interaction in Distributed Propulsion. AIAA Applied Aerodynamics Conference, Atlanta, GA, Jun. 2018. doi:10.2514/6.2018-3646

Tan, J., and Wang, H. (2013). Simulating unsteady aerodynamics of helicopter rotor with panel/viscous vortex particle method. *Aerospace Science and Technology*, 30(1), 255-268. doi:10.1016/j.ast.2013.08.010

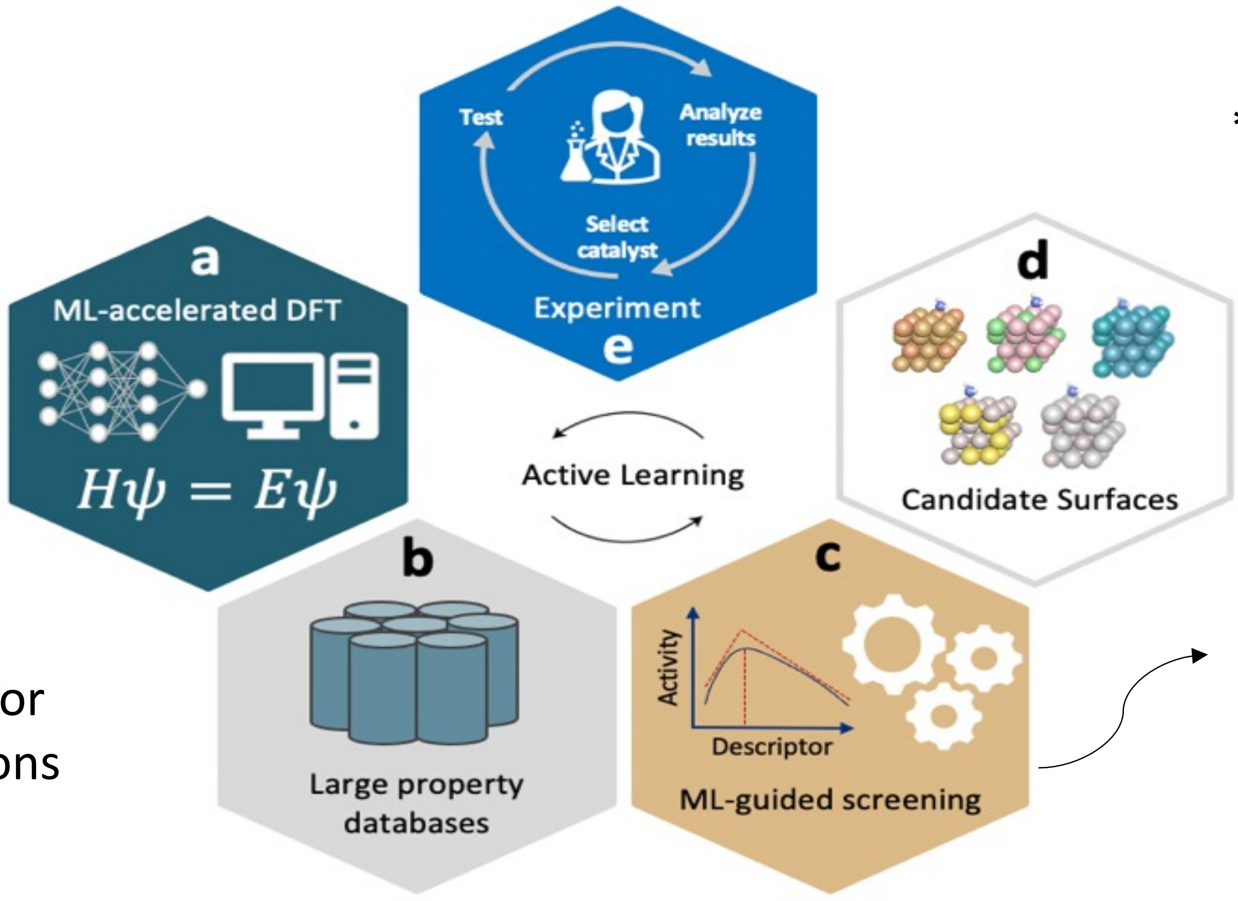
CSE6320: High Performance Parallel Computing

Distributed Hyperparameter Tuning for Machine Learning Models in Catalysis

Project Presentation
April 18, 2023
Omar Jiménez



Motivation: Data-Driven Catalyst Discovery Workflow



*many variations, this is just one example

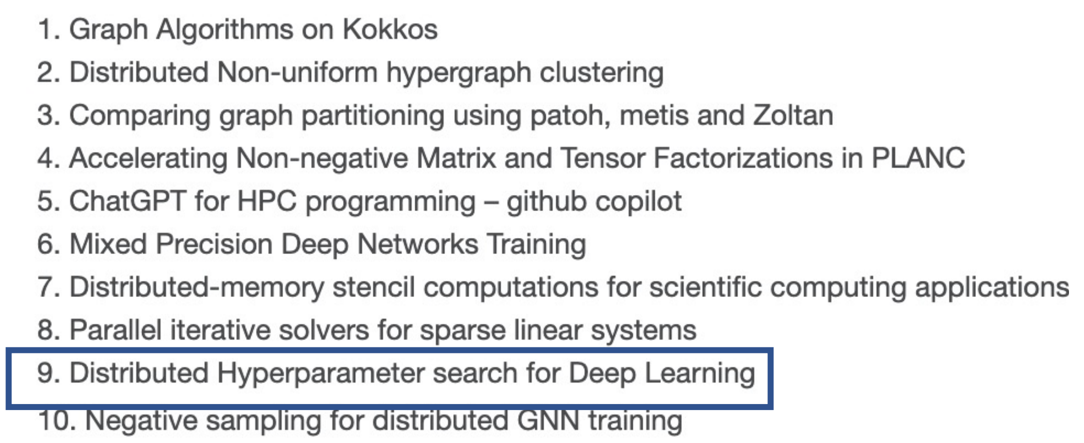
ML -> Training -> Hyperparameters

*ML Potentials = models for (energy) property predictions of chemical systems

Example at end of presentation

Project

Original idea from Piazza post on 03/08/23



1. Graph Algorithms on Kokkos
2. Distributed Non-uniform hypergraph clustering
3. Comparing graph partitioning using patoh, metis and Zoltan
4. Accelerating Non-negative Matrix and Tensor Factorizations in PLANC
5. ChatGPT for HPC programming – github copilot
6. Mixed Precision Deep Networks Training
7. Distributed-memory stencil computations for scientific computing applications
8. Parallel iterative solvers for sparse linear systems
9. Distributed Hyperparameter search for Deep Learning
10. Negative sampling for distributed GNN training

Project Type: ***Application.*** I will integrate distributed hyperparameter search algorithms into training of ML potentials relevant to data-driven catalyst discovery workflows

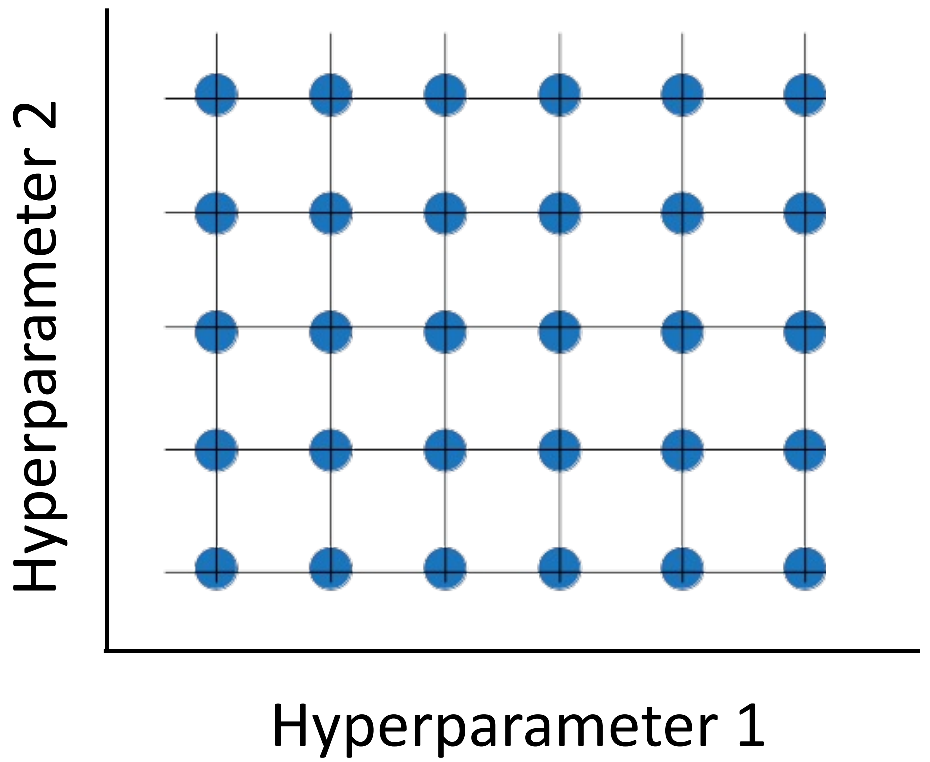
Problem Statement

Problem: Depending on the search space, search algorithms for hyperparameter tuning can be *computationally expensive* (from hours to days). State-of-the-art packages for training of ML potentials (e.g., SchNetPack, AMPTorch, etc.) do not support distributed hyperparameter tuning.

Solution: *Distributed* Hyperparameter Search

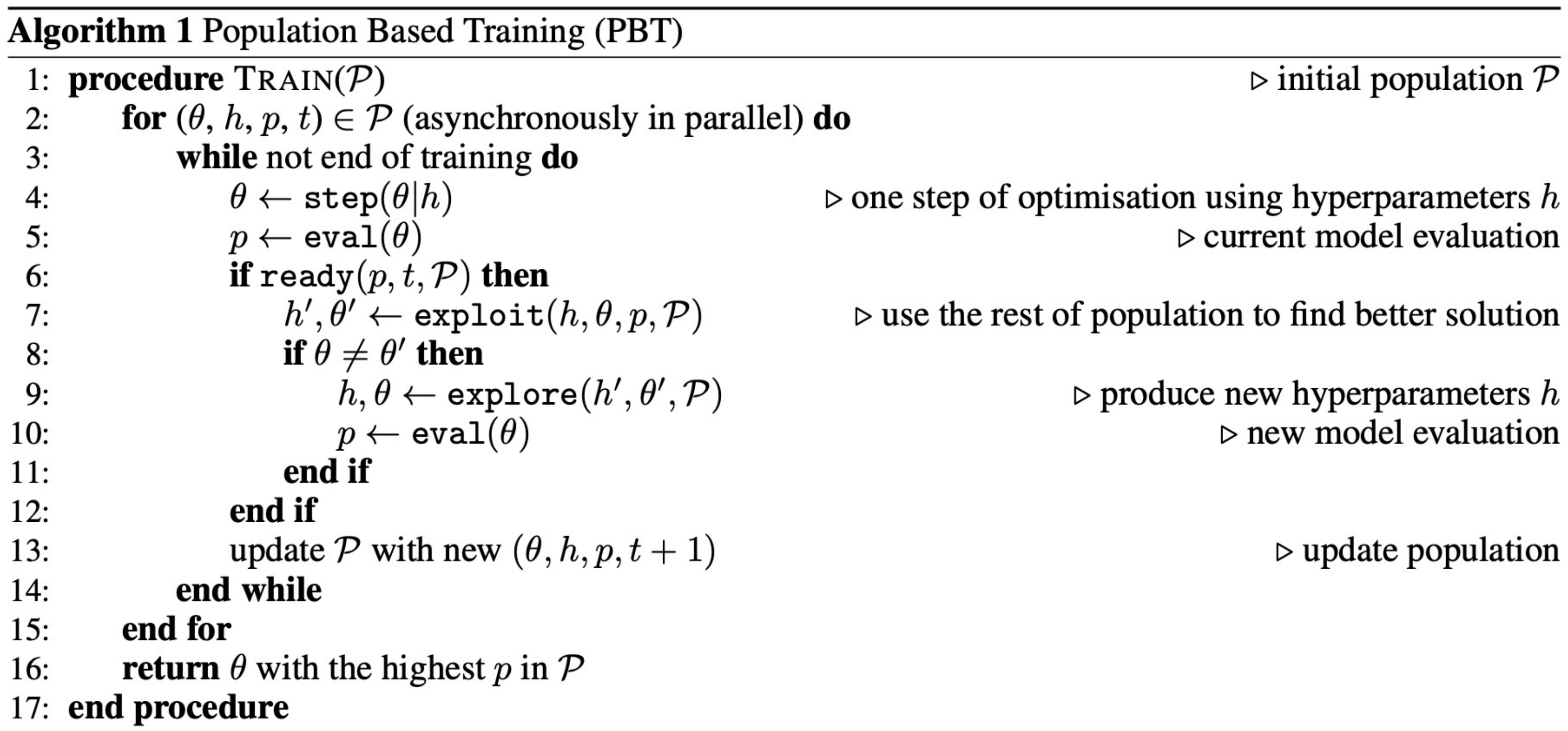
Implemented Algorithms

1. Grid Search: classical example and very easy to distribute (no dependency between tasks). No communication, each NN is trained to convergence and best set is selected



Implemented Algorithms

2. Population Based Training (PBT): originally proposed by Jaderberg et. Al (2017). Similar to EA



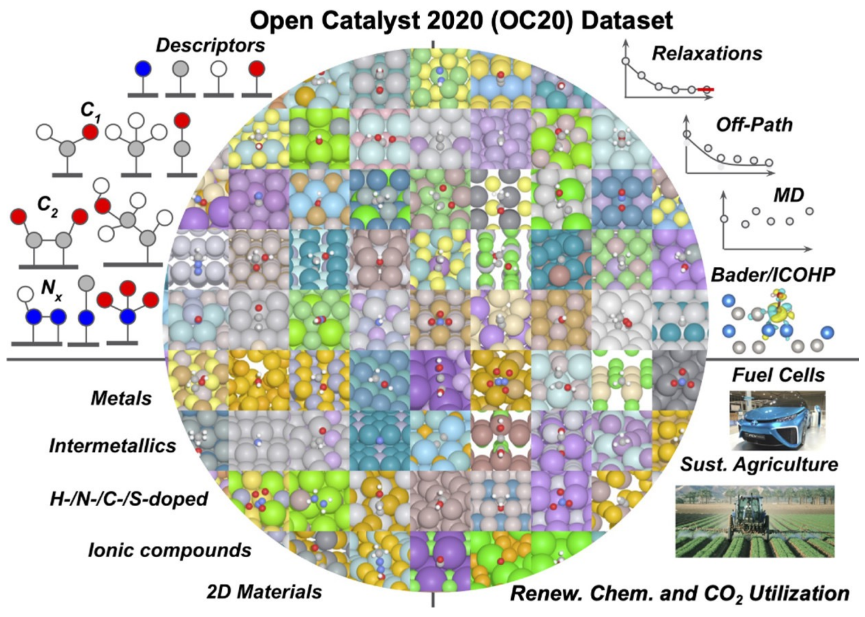
Algorithm 1 Population Based Training (PBT)

```
procedure TRAIN($\mathcal{P}$)                                    ▷ initial population $\mathcal{P}$
    for $(\theta, h, p, t) \in \mathcal{P}$ (asynchronously in parallel) do
        while not end of training do
            $\theta \leftarrow$ step$(\theta|h)$          ▷ one step of optimisation using hyperparameters $h$
            $p \leftarrow$ eval$(\theta)$                 ▷ current model evaluation
            if ready$(p, t, \mathcal{P})$ then
                $h', \theta' \leftarrow$ exploit$(h, \theta, p, \mathcal{P})$   ▷ use the rest of population to find better solution
                if $\theta \neq \theta'$ then
                    $h, \theta \leftarrow$ explore$(h', \theta', \mathcal{P})$   ▷ produce new hyperparameters $h$
                    $p \leftarrow$ eval$(\theta)$         ▷ new model evaluation
                end if
            end if
            update $\mathcal{P}$ with new $(\theta, h, p, t+1)$   ▷ update population
        end while
    end for
    return $\theta$ with the highest $p$ in $\mathcal{P}$
end procedure
```

Experiments

Dataset

- Open Catalyst 2020 (OC20), dataset of catalysis DFT calculations
- **Small chunk** of it. Full dataset has *millions* of data points which is not feasible given resource constraints



Testbed

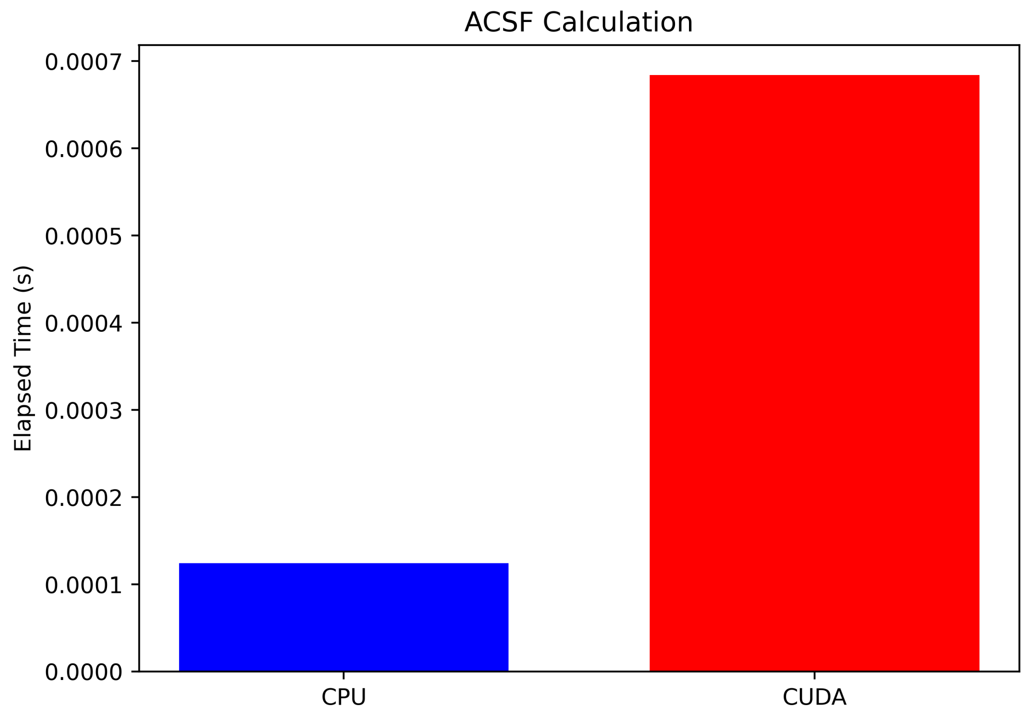
- Ray AWS Cluster

1. ACSF Calculations GPU Acceleration (attempt)

$$G_i^2 = \sum_{j=1}^{N_{\text{atom}}} e^{-\boxed{\eta}(R_{ij}-\boxed{R_s})^2} f_c(R_{ij})$$

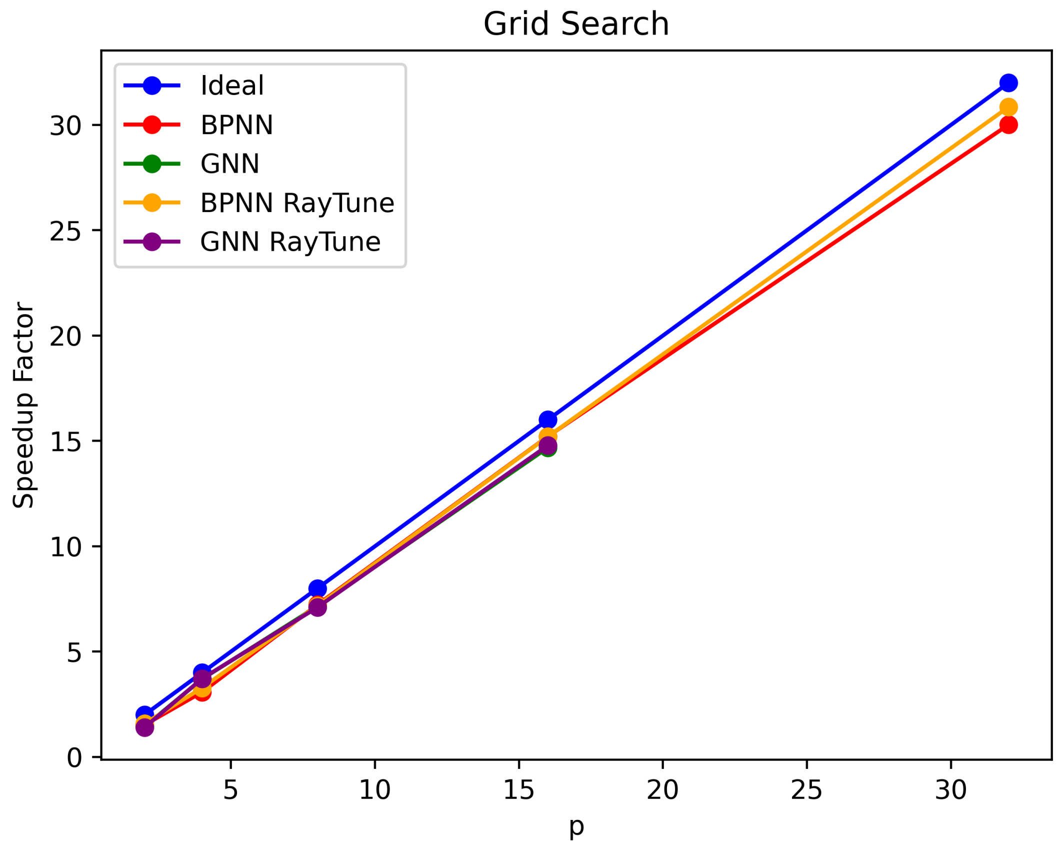
Hyperparameters

$$G_i^4 = 2^{1-\boxed{\zeta}} \sum_{j,k\neq i} [(1+\boxed{\gamma}\cos\theta_{ijk})^{\boxed{\zeta}} e^{-\eta(R_{ij}^2+R_{ik}^2+R_{jk}^2)} f_c(R_{ij})$$
$$f_c(R_{ik})\, f_c(R_{jk})]$$

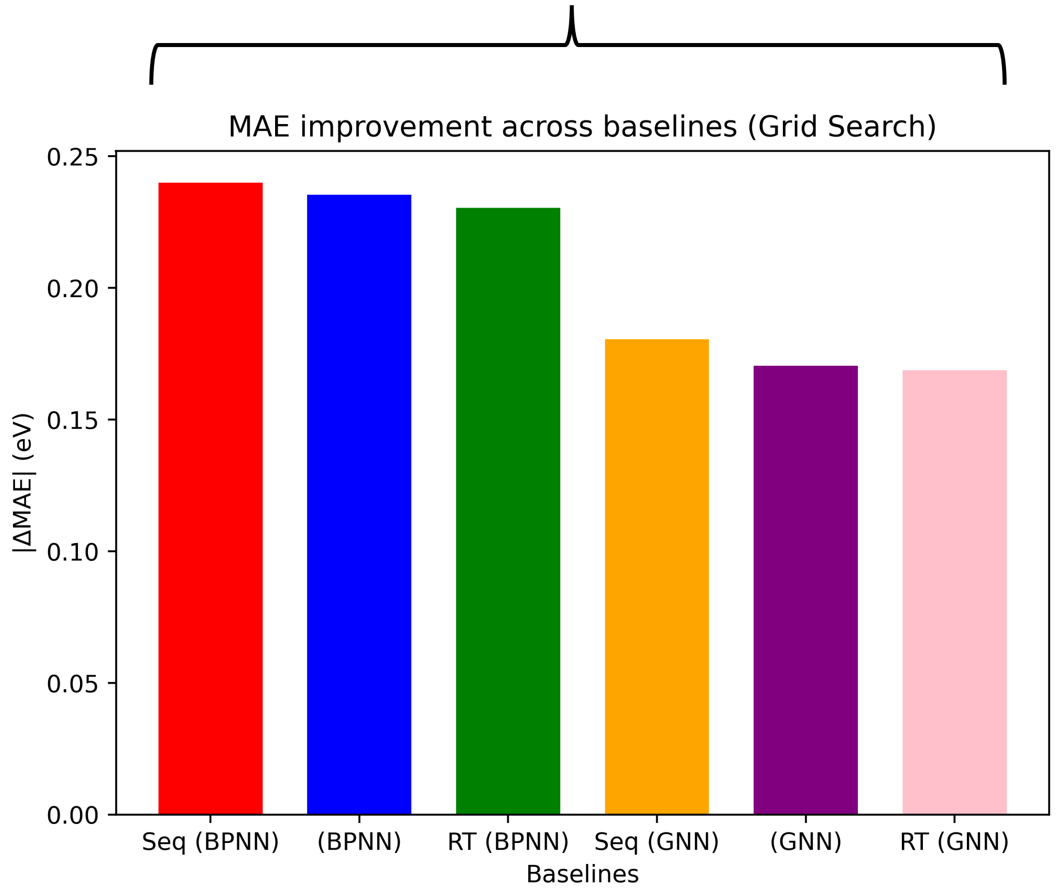


Not really expected to work just by looking at equations, computation is not linear algebra heavy (tensors are small, $N\times N$ where typically $N < 100$)

2. Grid Search Results



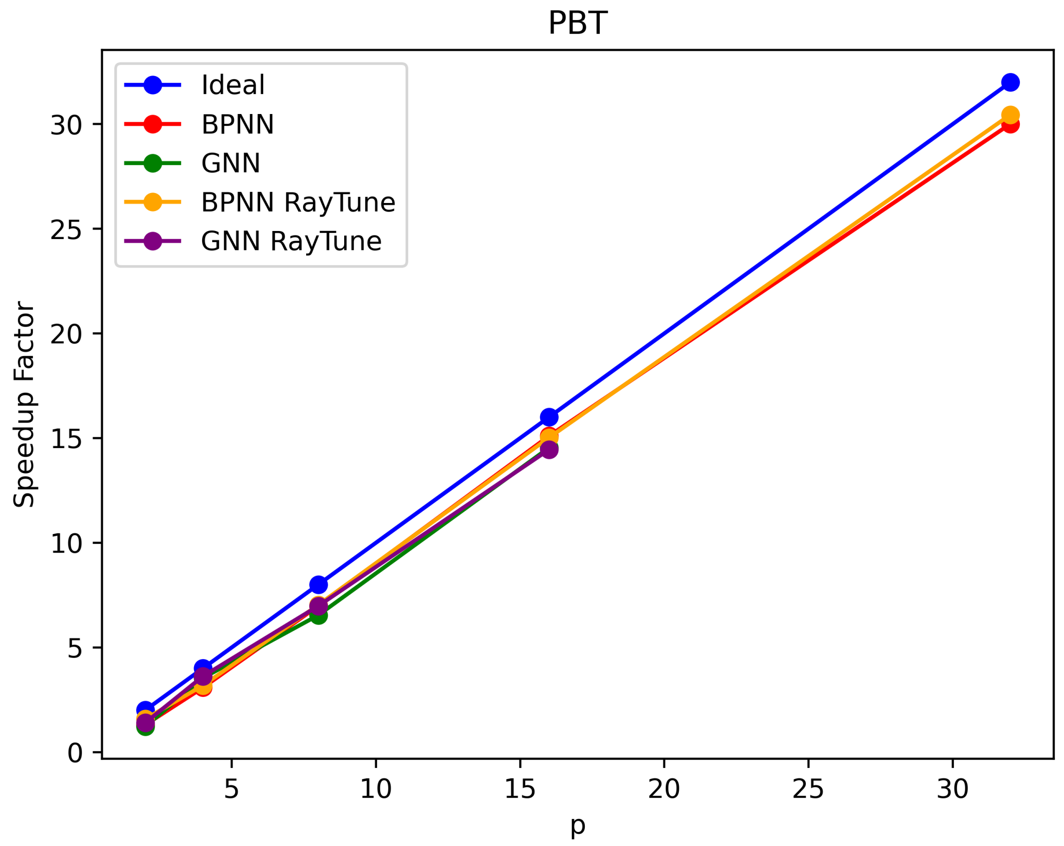
Validation (correctness)

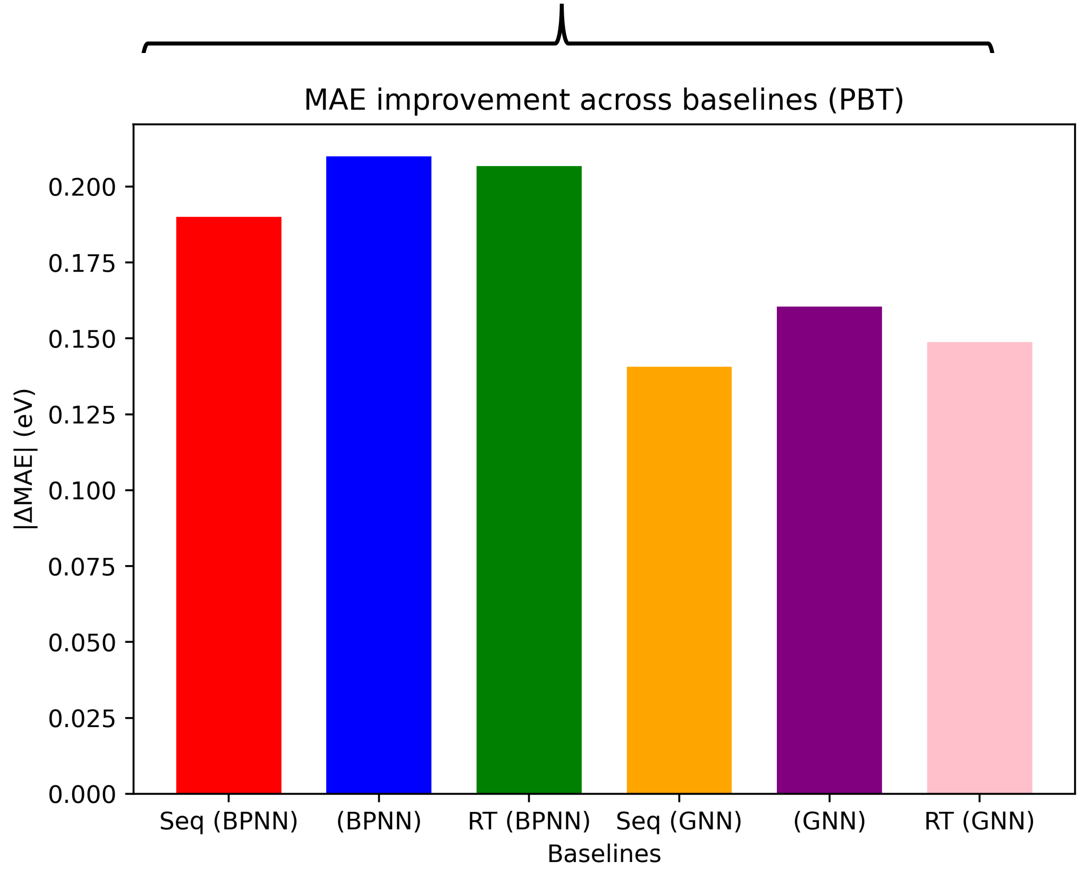


Distributed Hyperparameter Tuning is embarrassingly parallel by nature

3. PBT Results

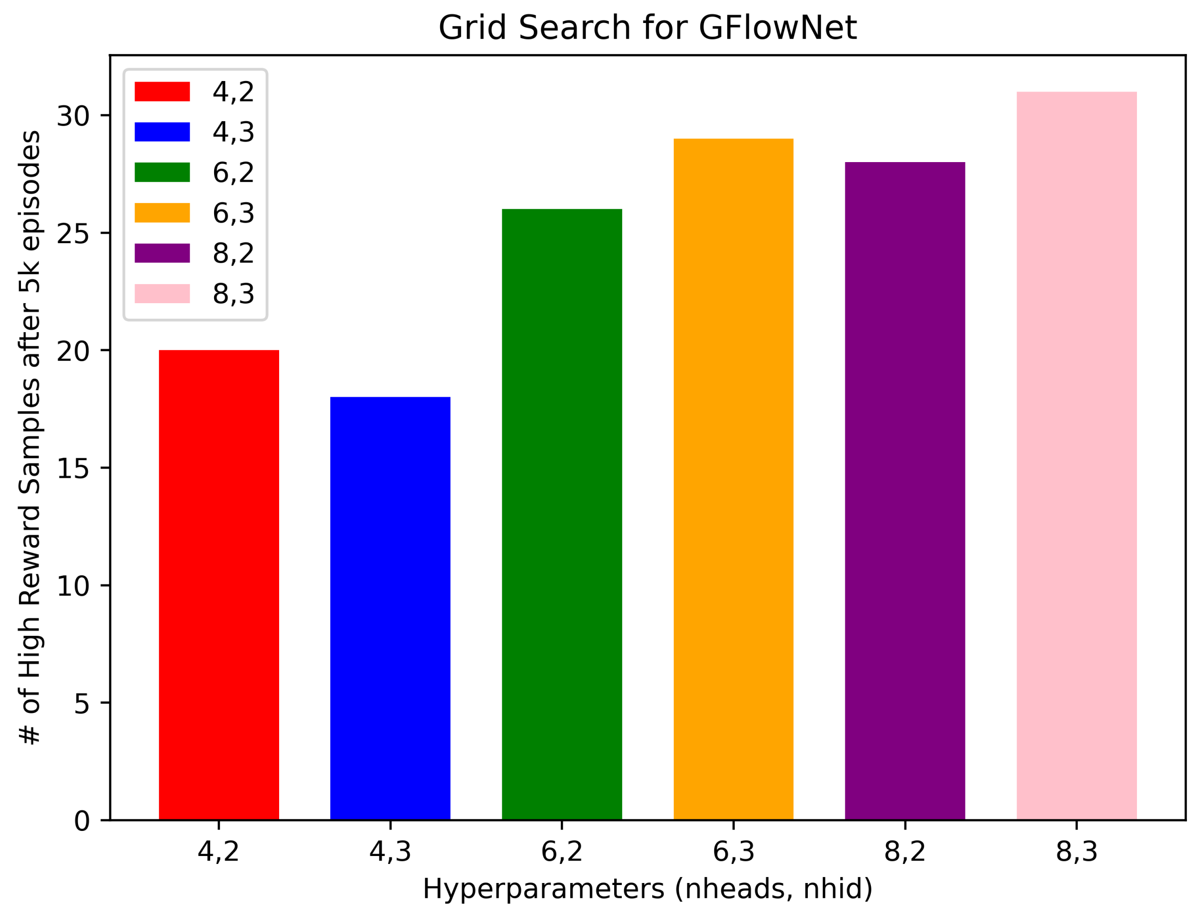
Validation (correctness)





Distributed Hyperparameter Tuning is embarrassingly parallel by nature

4. Generative Model Performance



Scaling obvious by now (+ expensive in this case), focus on "performance" improvement