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

- Most papers look at two metrics
- Execution time per iteration (ms)
- Performance (FLOPS)

Baselines

- Sequential implementation
- OpenMP implementation
- sklearn implementation
- Results from prior papers
  - Lutz et. al (2018) [1]
  - Shahrezaei and Tavoli (2019) [2]
  - Yang et. al. (2020) [3]

Proposed Solution

- Implement single pass algorithm
  - Proposed by Lutz et al in 2018 [1]
  - Remove implicit barrier between calculating cluster assignment and recalculating cluster centers
- Vectorized loads and computation
- Texture memory

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 data
- Compare outputs and see if final cluster centers are within a tolerance

Dataset and Testbed

- Dataset
    - Randomly generated vectors
    - General acceleration for k-means, not specific use-cases
- Test System
    - College of Computing PACE Multi and GPU Clusters
    - Test Sequential, OpenMP, and sklearn baselines on 120 cores
    - Test GPU code on Tesla V100 GPU

Experiments and Potential Plots

- Line Plots
  - x-axis
    - Vary dimensionality of data (2, 4, 8, 16, 32, 64, 128, 256 dimensions)
    - Vary number of clusters (2, 4, 8, 16, 32, 64, 128, 256 clusters)
    - Vary number of points (1k, 5k, 10k, 20k, 50k, 100k, 250k, 500k)
  - y-axis
    - Measure time per iteration
    - Measure peak performance in FLOPS
- Breakdown plot
  - Time spent in different parts of the algorithm (cluster assignment, recentering, etc.)
  - Compare breakdowns for OpenMP and GPU implementations

References

[1] 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.

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

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

Thank You!

Accelerating Proximal Policy Optimization (PPO)

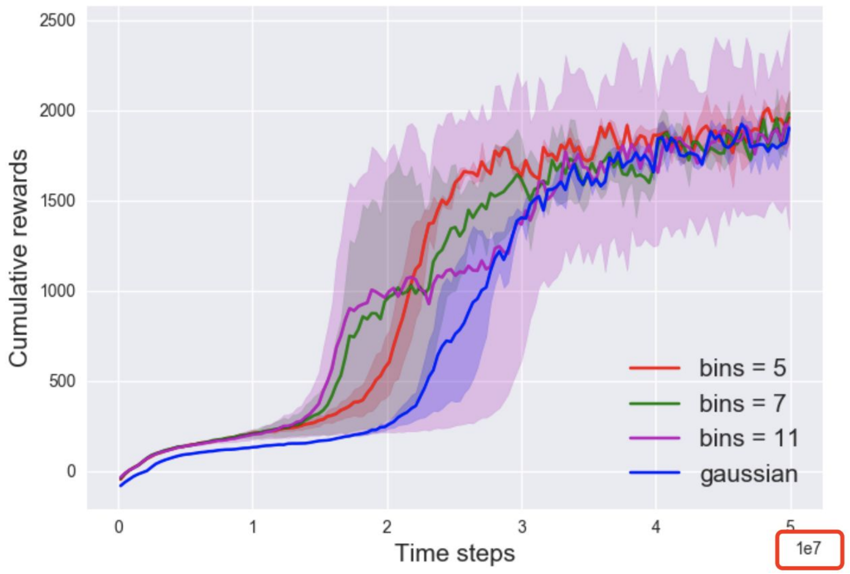
Akhil Goel, Matthew Woodward, Qingyu Xiao

Outline

- Problem Definition
- Project Category & Performance Metric
- Baselines & Dataset
- Proposed Solution
- Validation
- Experimental Design & Possible Roadblocks

Problem Definition

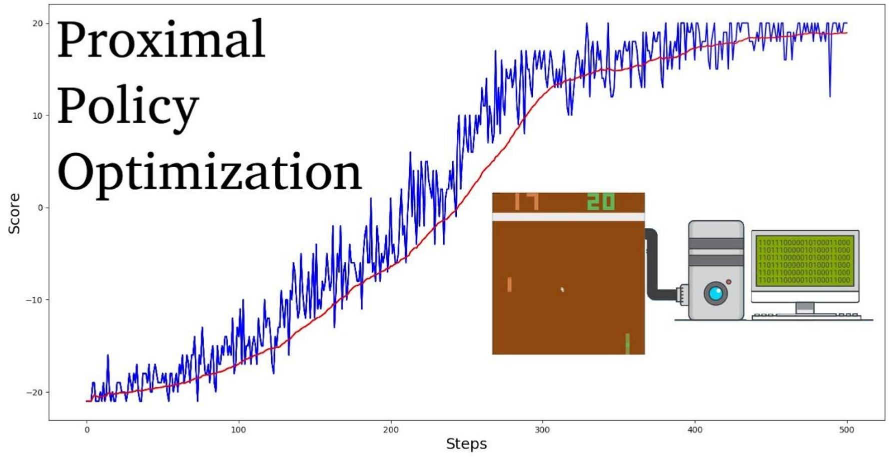
- Reinforcement Learning
- Accelerate with GPUs
- Scale with HPC Cluster?
- Profiling of bottlenecks
  - Compute Time
    - Environment Simulation
    - Parameter Updates
  - Memory Time
  - Communication Time



Training time steps is very large, roughly 5e7 time steps

Project Category & Performance Metric

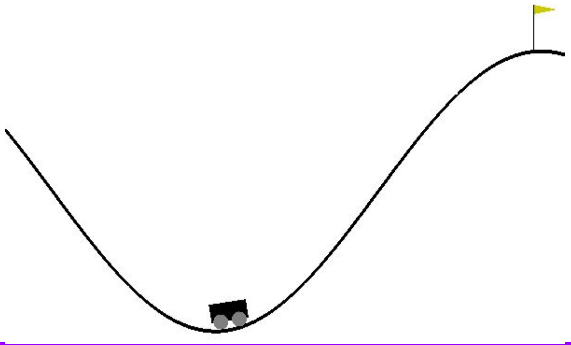
- Reproducibility: DD-PPO (https://arxiv.org/abs/1911.00357)
  - Decentralized Distributed Proximal Policy Optimization
- Performance Modeling: Scaling & Profiling Bottlenecks

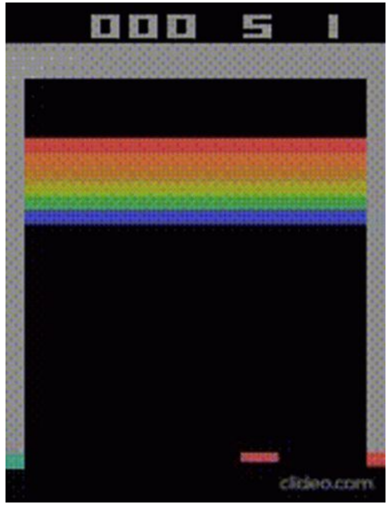


https://www.youtube.com/watch?v=lYP3cF2wqOY

Baselines & Dataset

- Classical PPO (https://arxiv.org/abs/1707.06347)
  - Single Worker
- OpenAI Gym RL Environments
  - Mountain Car
  - Atari (ported to GPU acceleration)





Proposed Solution

- DD-PPO
  - Decentralized synchronous update with worker pre-emption
  - GPU Acceleration of Environments (CuLE)

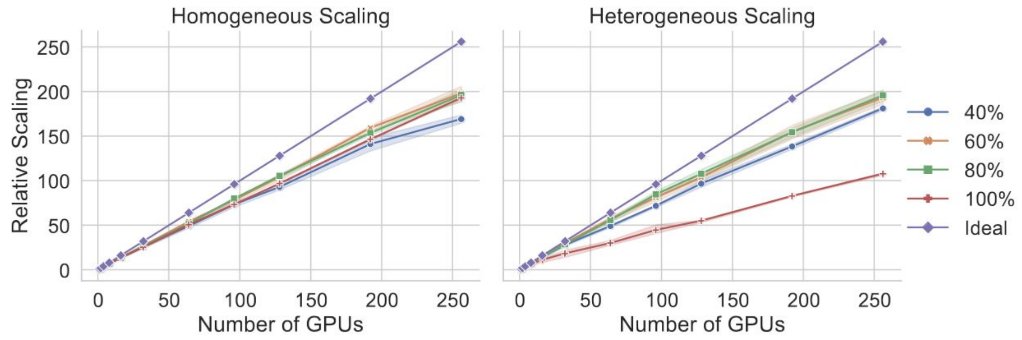


Figure 4: Scaling performance (in steps of experience per second relative to 1 GPU) of DD-PPO for various preemption threshold, $p\%$, values. Shading represents a 95% confidence interval.

NVlabs/**cule**

CuLE: A CUDA port of the Atari Learning Environment (ALE)

3 Contributors · 15 Issues · 213 Stars · 31 Forks

Validation

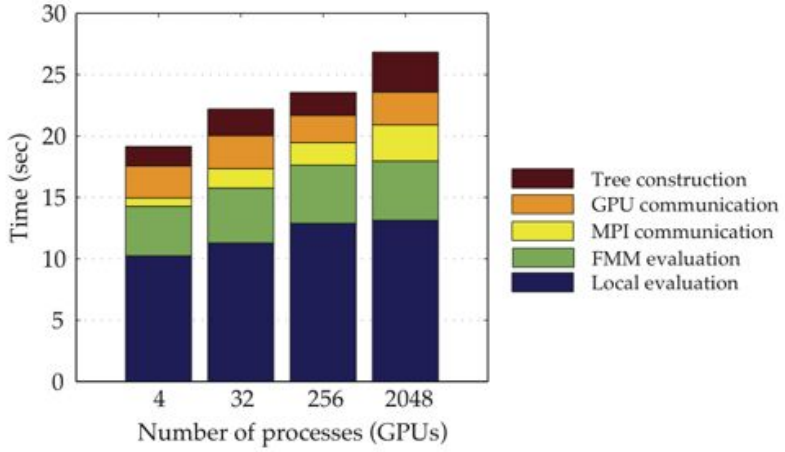
- PPO on Single GPU
- Performance Baseline
  - Fixed # of frames / samples to train on
  - Measure training time and samples of experience / second
- “Correctness” of results is difficult to measure
  - Training updates are stochastic in nature
  - Measuring “success” of agent is nebulous
- Isolate a small, deterministic example to compare weights

Why not centralized asynchronous PPO?

- "Asynchronous Distribution is notoriously difficult"
  - Unrealistic to implement
- Compute Resources
  - Limitations of scaling (synchronization strategy tradeoffs) difficult to test at our level of scale
- CPU vs GPU accelerated simulation of environment
  - Our environment is more computationally simple
  - Environment acceleration is preferred anyway

Experimental Design

- Test Bed: COC-ICE-GPU
- "Static" Variables
  - Environment Acceleration
- Independent Variables
  - Number of Workers
  - Pre-Emption Threshold
- Dependent Variables
  - Steps of experience per second
  - Compute time breakdown



https://www.bu.edu/exafmm/documentation/performance/

References

- Wijmans, E., Kadian, A., Morcos, A., Lee, S., Essa, I., Parikh, D., ... & Batra, D. (2019). DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames. arXiv preprint arXiv:1911.00357.
- Tang, Yunhao & Agrawal, Shipra. (2020). Discretizing Continuous Action Space for On-Policy Optimization. Proceedings of the AAAI Conference on Artificial Intelligence. 34. 5981-5988. 10.1609/aaai.v34i04.6059.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

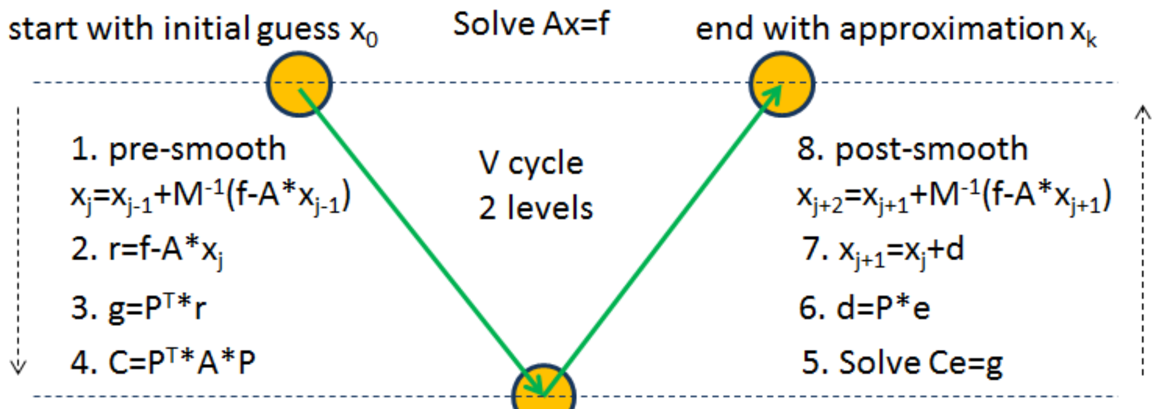
GPU-Accelerated Algebraic Multigrid Methods (AMG)

Bao Li

CSE 6230 – Spring 2023

Category

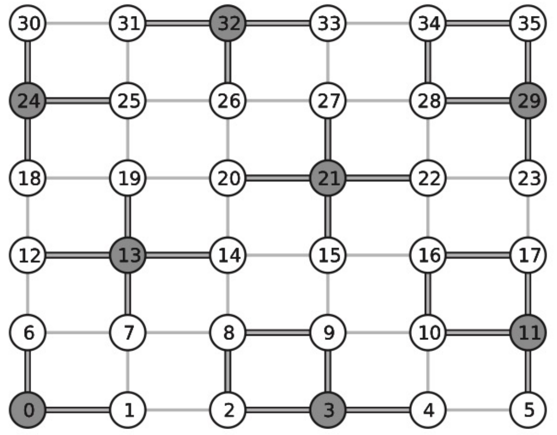
- Category:
  - AMG Solver (V-cycles): apply to structural optimization
  - the AMG algorithm solves the large (fine) linear system by cycling through levels composed of smaller (coarse) linear systems and finding updates that bring one closer to the exact solution

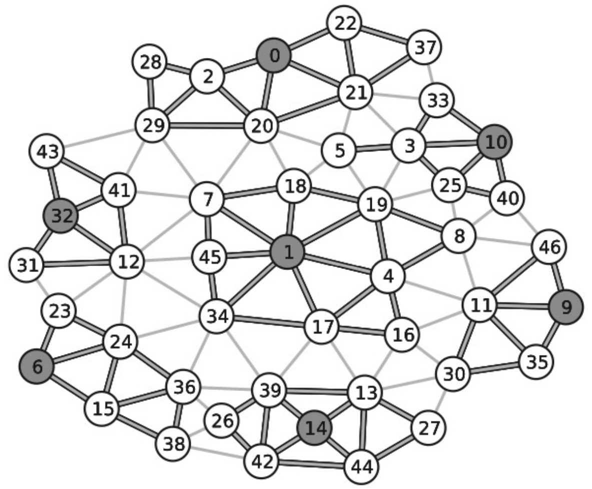


- M: preconditioner or smoothers that removes high frequency errors corresponding to matrix A
- $Ce = g$: coarse linear system
- P: prolongation and restriction matrices allow us to transition from coarse to fine and fine to coarse levels of the grid

Problem Statement

- Objective
  - **Large sparse** matrix from large scale structure optimization:
$$Ku = f$$
- Setup phase
  - Strength + aggregate + tentative + prolongation
- Galerkin product: $A_{k+1} = P_k^T A_k P_k$
  - Efficient sequential sparse matrix-matrix multiplication algorithms
  - Both space + computational efferent: fast and not large temporary storage


(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
  - Compare with existing GPU-accelerated solver
- Baseline:
  - Method based:
    - C++, pure sequentially AMG (V-cycle) method
  - GPU based:
    - cuSolverSP: Sparse LAPACK
    - GMRES Kokkos-based solver
  - AMGX

Proposed Solution

- OpenMP
- Kokkos + Kokkos Kernels:
  - Performance **portability** between GPUs and multicore CPUs.
  - It builds on top of parallel programming frameworks (such as CUDA, and OpenMP).
  - Similar to BLAS
  - Easy to implement
  - SPARSE 1, 2, 3 Kernels
- Thrust: C++ standard template library for CUDA based on the (STL)
  - Performance **portability** between GPUs and multicore CPUs.
  - It builds on top of parallel programming frameworks (such as CUDA, and OpenMP).
  - Hard to implement
  - + cuSPARSE

Proposed Solution

- Why Thrust
  - 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
- Thrust API
    - `thrust::transform`
    - `thrust::for_each`
    - `thrust::copy`
    - `thrust::sort`
    - `thrust::reduce`
    - `thrust::sequence`
    - `thrust::inner_product`
    - …

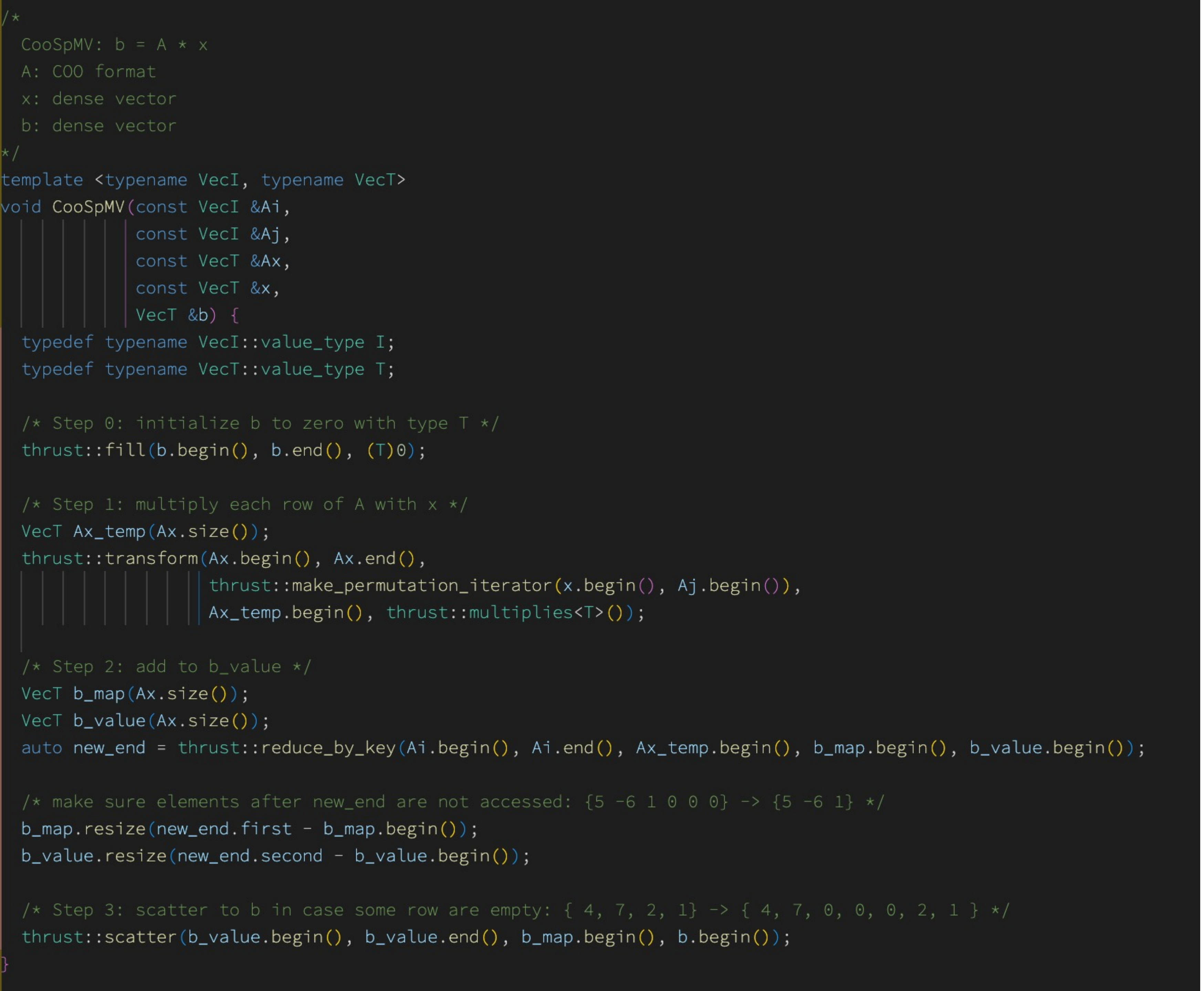
```
#include <thrust/copy.h>
#include <thrust/execution_policy.h>
...
struct is_even
{
__host__ __device__
bool operator()(const int x)
{
return (x % 2) == 0;
}
};

const int N = 6;
int V[N] = {-2, 0, -1, 0, 1, 2};
int result[4];
thrust::copy_if(thrust::host, V, V + N, result, is_even());
// V remains {-2, 0, -1, 0, 1, 2}
// result is now {-2, 0, 0, 2}
```

Datasets

- Data Store Format
  - Coordinate Format (COO): higher space complex
  - Block Compressed Sparse Row Format (BSR)

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix} \qquad \begin{array}{lcl} \text{cooValA} & = & [1.0 \; 4.0 \; 2.0 \; 3.0 \; 5.0 \; 7.0 \; 8.0 \; 9.0 \; 6.0] \\ \text{cooRowIndA} & = & [0 \; 0 \; 1 \; 1 \; 2 \; 2 \; 2 \; 3 \; 3] \\ \text{cooColIndA} & = & [0 \; 1 \; 1 \; 2 \; 0 \; 3 \; 4 \; 2 \; 4] \end{array}$$

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 & 0.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 & 0.0 \end{bmatrix} \qquad \begin{array}{lcl} \text{bsrValA} & = & [A_{00} \; A_{01} \; A_{10} \; A_{11} \; A_{12}] \\ \text{bsrRowPtrA} & = & [0 \; 2 \; 5] \\ \text{bsrColIndA} & = & [0 \; 1 \; 0 \; 1 \; 2] \end{array} \qquad A_b = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

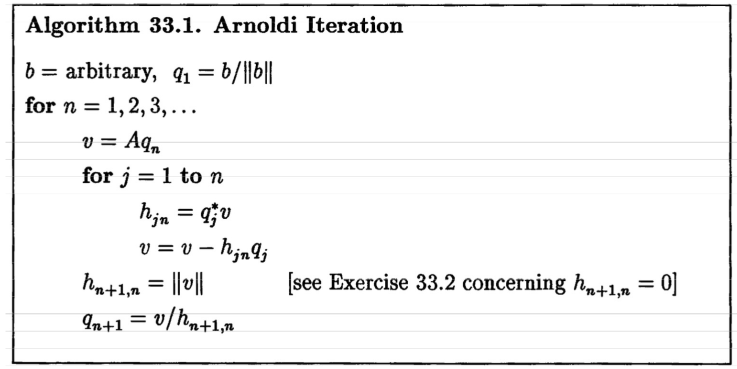


```
/*
  CooSpMV: b = A * x
  A: COO format
  x: dense vector
  b: dense vector
*/
template <typename VecI, typename VecT>
void CooSpMV(const VecI &Ai,
             const VecI &Aj,
             const VecT &Ax,
             const VecT &x,
             VecT &b) {
  typedef typename VecI::value_type I;
  typedef typename VecT::value_type T;

  /* Step 0: initialize b to zero with type T */
  thrust::fill(b.begin(), b.end(), (T)0);

  /* Step 1: multiply each row of A with x */
  VecT Ax_temp(Ax.size());
  thrust::transform(Ax.begin(), Ax.end(),
                    thrust::make_permutation_iterator(x.begin(), Aj.begin()),
                    Ax_temp.begin(), thrust::multiplies<T>());

  /* Step 2: add to b_value */
  VecT b_map(Ax.size());
  VecT b_value(Ax.size());
  auto new_end = thrust::reduce_by_key(Ai.begin(), Ai.end(), Ax_temp.begin(), b_map.begin(), b_value.begin());

  /* make sure elements after new_end are not accessed: {5 -6 1 0 0 0} -> {5 -6 1} */
  b_map.resize(new_end.first - b_map.begin());
  b_value.resize(new_end.second - b_value.begin());

  /* Step 3: scatter to b in case some row are empty: { 4, 7, 2, 1} -> { 4, 7, 0, 0, 0, 2, 1 } */
  thrust::scatter(b_value.begin(), b_value.end(), b_map.begin(), b.begin());
}
```

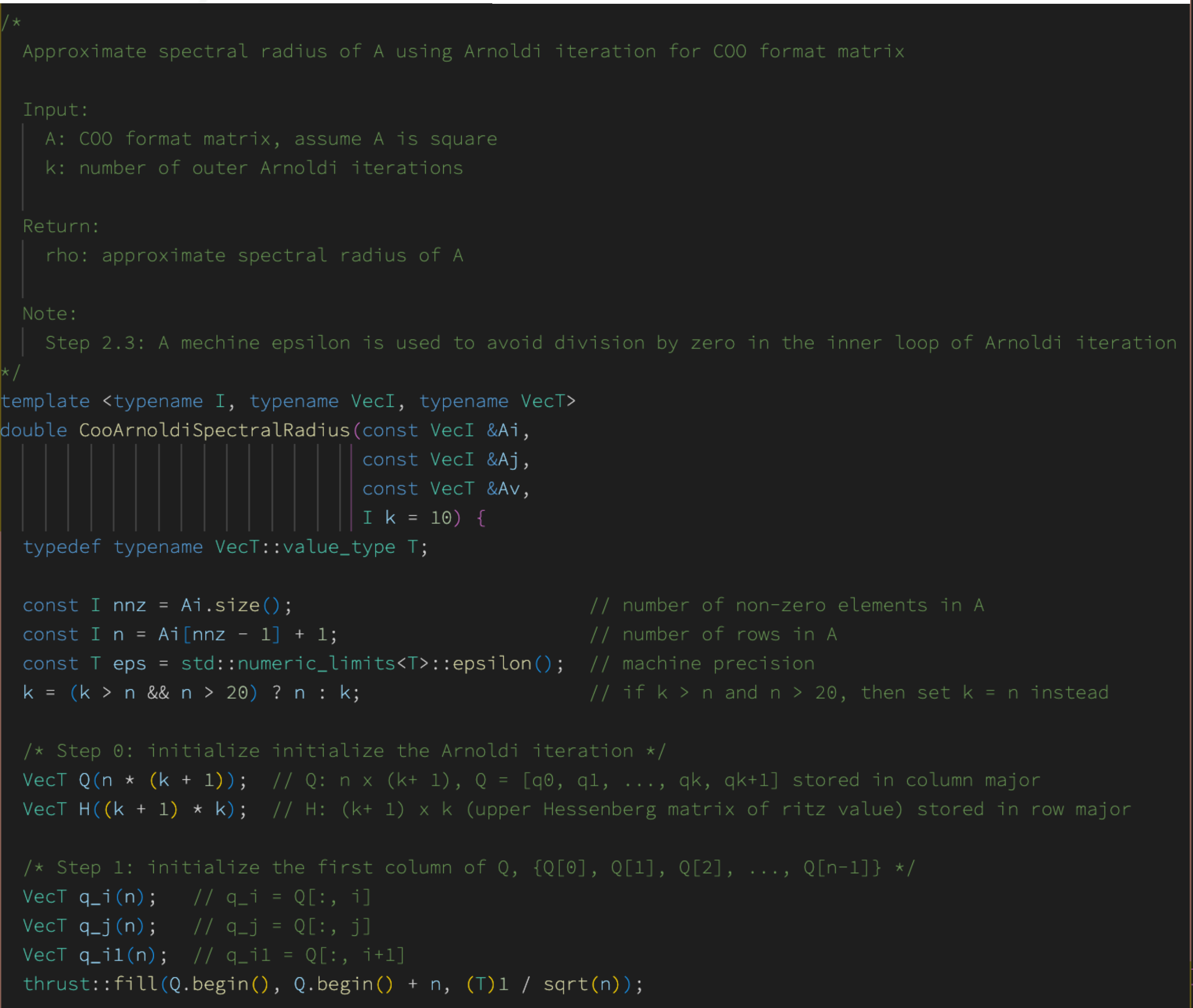
Reason For COO

- Easier to implement via thrust::iterator

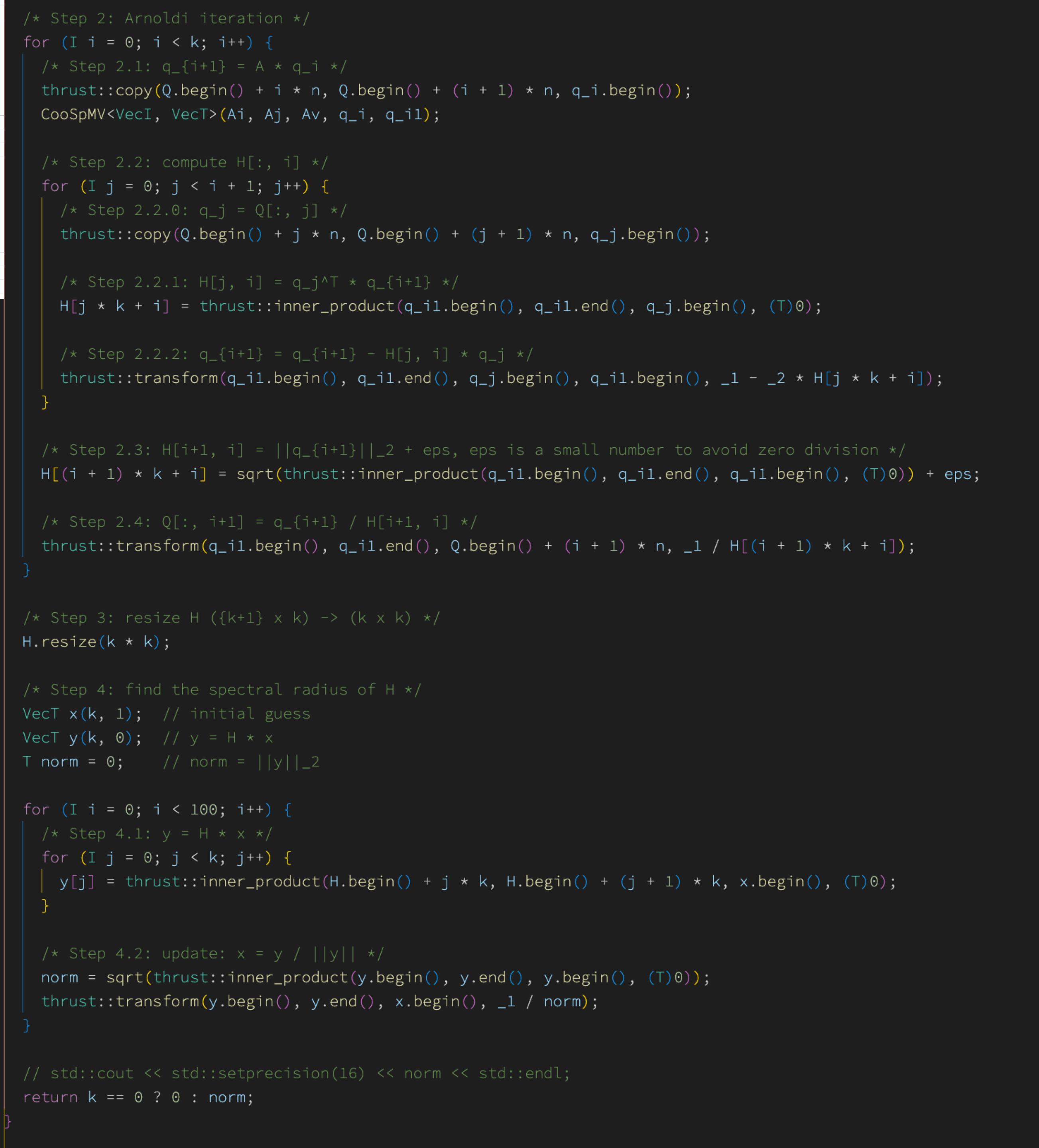
Georgia Tech


Algorithm 33.1. Arnoldi Iteration

```
$b =$ arbitrary, $q_1 = b/\|b\|$
for $n = 1, 2, 3, \ldots$
    $v = Aq_n$
    for $j = 1$ to $n$
        $h_{jn} = q_j^* v$
        $v = v - h_{jn} q_j$
    $h_{n+1,n} = \|v\|$        [see Exercise 33.2 concerning $h_{n+1,n} = 0$]
    $q_{n+1} = v/h_{n+1,n}$
```

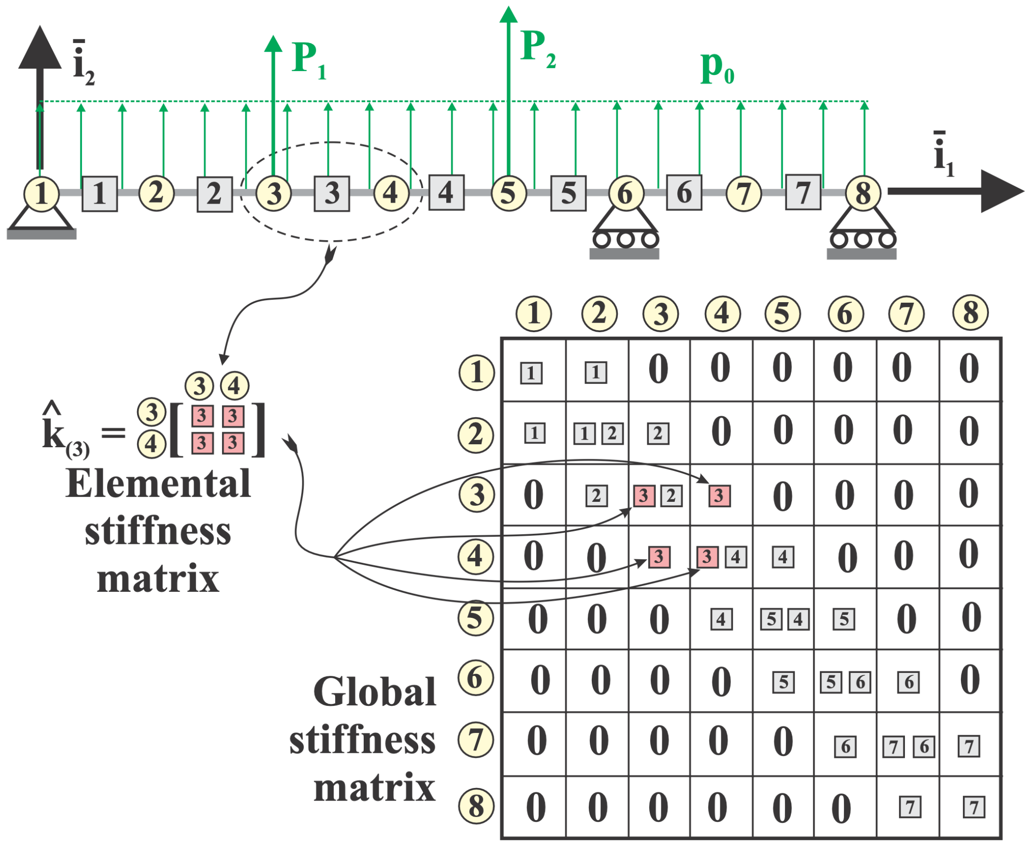


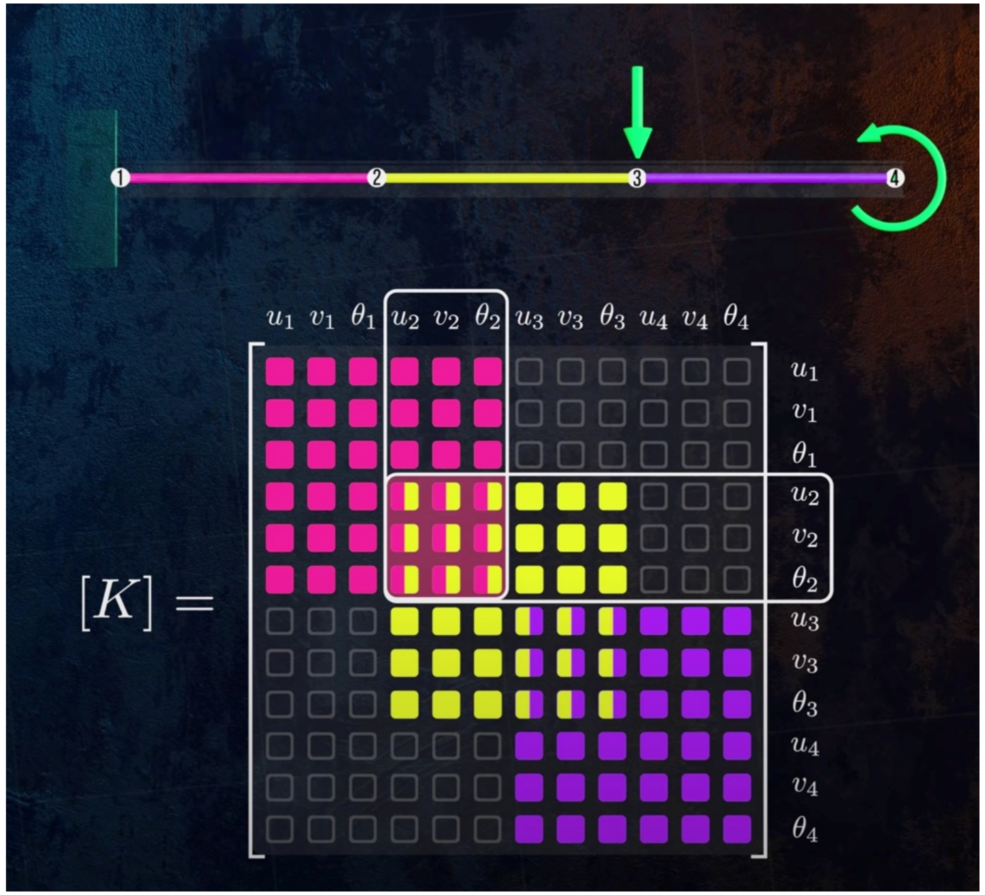
```cpp
/*
  Approximate spectral radius of A using Arnoldi iteration for COO format matrix

  Input:
    A: COO format matrix, assume A is square
    k: number of outer Arnoldi iterations

  Return:
    rho: approximate spectral radius of A

  Note:
    Step 2.3: A mechine epsilon is used to avoid division by zero in the inner loop of Arnoldi iteration
*/
template <typename I, typename VecI, typename VecT>
double CooArnoldiSpectralRadius(const VecI &Ai,
                                const VecI &Aj,
                                const VecT &Av,
                                I k = 10) {
  typedef typename VecT::value_type T;

  const I nnz = Ai.size();                          // number of non-zero elements in A
  const I n = Ai[nnz - 1] + 1;                      // number of rows in A
  const T eps = std::numeric_limits<T>::epsilon();  // machine precision
  k = (k > n && n > 20) ? n : k;                    // if k > n and n > 20, then set k = n instead

  /* Step 0: initialize initialize the Arnoldi iteration */
  VecT Q(n * (k + 1));  // Q: n x (k+ 1), Q = [q0, q1, ..., qk, qk+1] stored in column major
  VecT H((k + 1) * k);  // H: (k+ 1) x k (upper Hessenberg matrix of ritz value) stored in row major

  /* Step 1: initialize the first column of Q, {Q[0], Q[1], Q[2], ..., Q[n-1]} */
  VecT q_i(n);   // q_i = Q[:, i]
  VecT q_j(n);   // q_j = Q[:, j]
  VecT q_i1(n);  // q_i1 = Q[:, i+1]
  thrust::fill(Q.begin(), Q.begin() + n, (T)1 / sqrt(n));
```



```cpp
  /* Step 2: Arnoldi iteration */
  for (I i = 0; i < k; i++) {
    /* Step 2.1: q_{i+1} = A * q_i */
    thrust::copy(Q.begin() + i * n, Q.begin() + (i + 1) * n, q_i.begin());
    CooSpMV<VecI, VecT>(Ai, Aj, Av, q_i, q_i1);

    /* Step 2.2: compute H[:, i] */
    for (I j = 0; j < i + 1; j++) {
      /* Step 2.2.0: q_j = Q[:, j] */
      thrust::copy(Q.begin() + j * n, Q.begin() + (j + 1) * n, q_j.begin());

      /* Step 2.2.1: H[j, i] = q_j^T * q_{i+1} */
      H[j * k + i] = thrust::inner_product(q_i1.begin(), q_i1.end(), q_j.begin(), (T)0);

      /* Step 2.2.2: q_{i+1} = q_{i+1} - H[j, i] * q_j */
      thrust::transform(q_i1.begin(), q_i1.end(), q_j.begin(), q_i1.begin(), _1 - _2 * H[j * k + i]);
    }

    /* Step 2.3: H[i+1, i] = ||q_{i+1}||_2 + eps, eps is a small number to avoid zero division */
    H[(i + 1) * k + i] = sqrt(thrust::inner_product(q_i1.begin(), q_i1.end(), q_i1.begin(), (T)0)) + eps;

    /* Step 2.4: Q[:, i+1] = q_{i+1} / H[i+1, i] */
    thrust::transform(q_i1.begin(), q_i1.end(), Q.begin() + (i + 1) * n, _1 / H[(i + 1) * k + i]);
  }

  /* Step 3: resize H ({k+1} x k) -> (k x k) */
  H.resize(k * k);

  /* Step 4: find the spectral radius of H */
  VecT x(k, 1);  // initial guess
  VecT y(k, 0);  // y = H * x
  T norm = 0;    // norm = ||y||_2

  for (I i = 0; i < 100; i++) {
    /* Step 4.1: y = H * x */
    for (I j = 0; j < k; j++) {
      y[j] = thrust::inner_product(H.begin() + j * k, H.begin() + (j + 1) * k, x.begin(), (T)0);
    }

    /* Step 4.2: update: x = y / ||y|| */
    norm = sqrt(thrust::inner_product(y.begin(), y.end(), y.begin(), (T)0));
    thrust::transform(y.begin(), y.end(), x.begin(), _1 / norm);
  }

  // std::cout << std::setprecision(16) << norm << std::endl;
  return k == 0 ? 0 : norm;
}
```

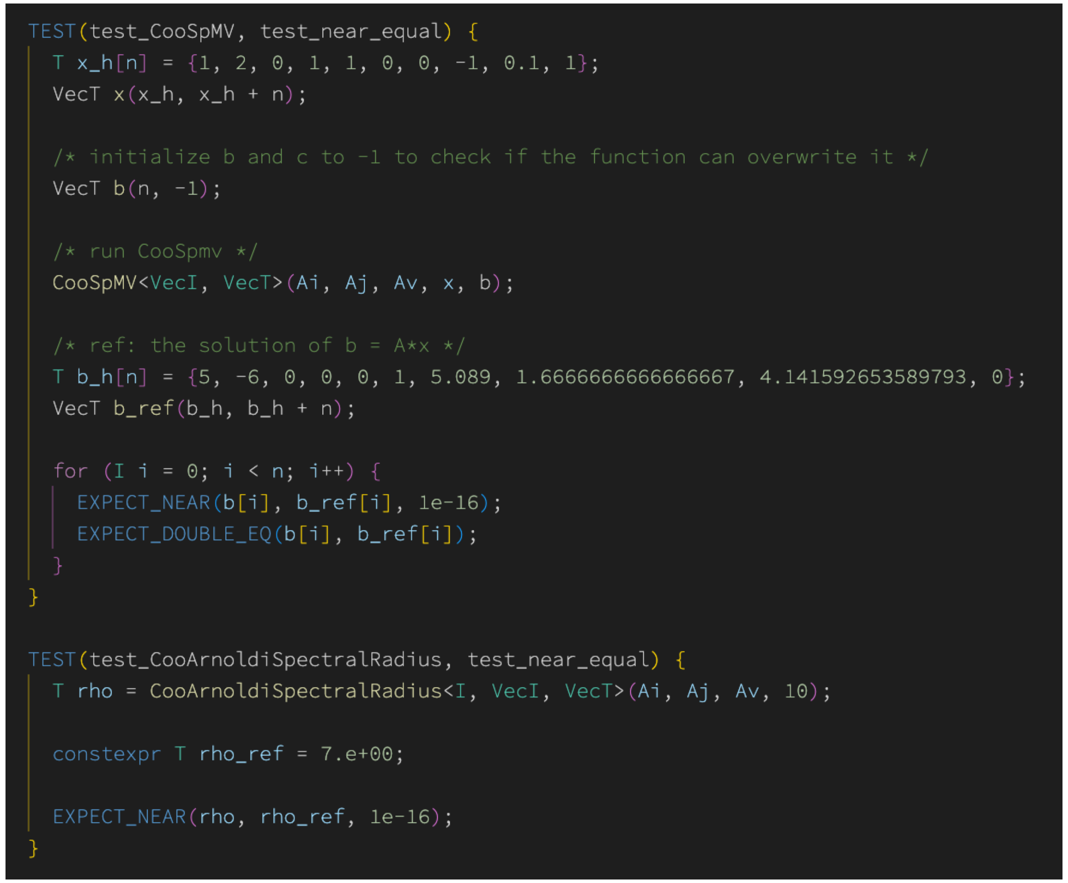
Reason For BRS: $Ku = f$



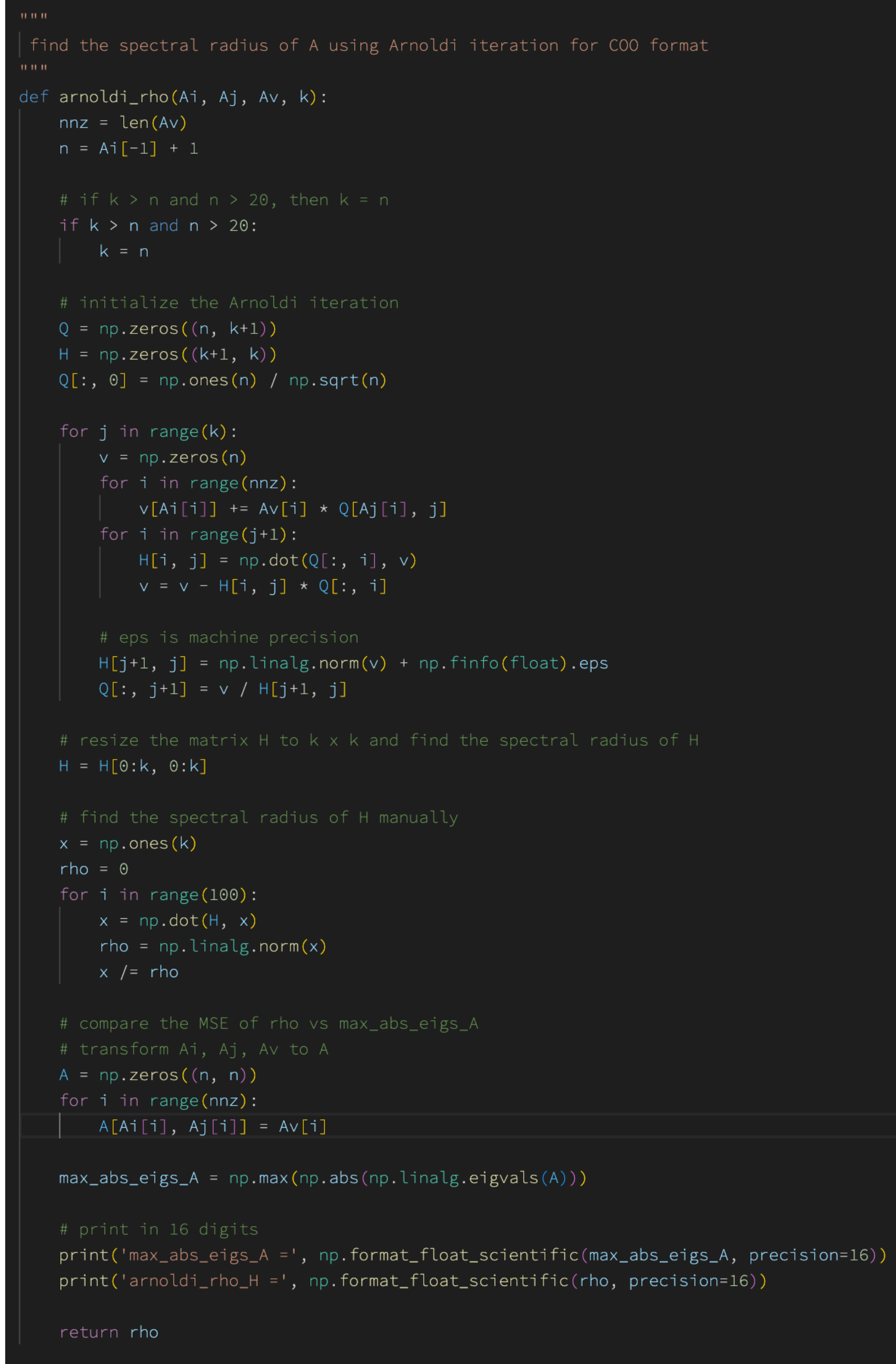


Validation of the proposed solution

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

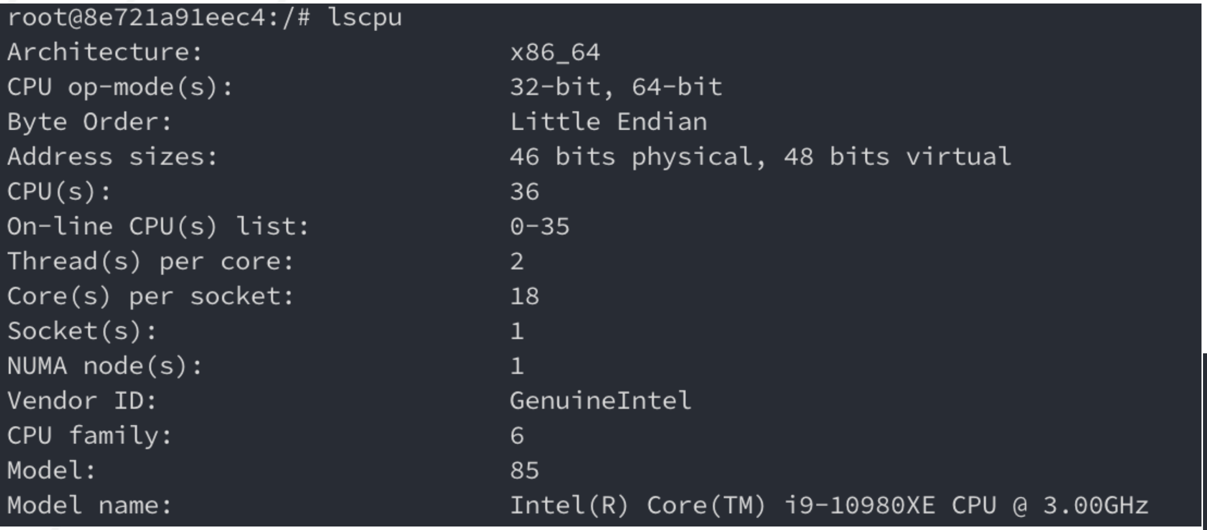


```cpp
TEST(test_CooSpMV, test_near_equal) {
  T x_h[n] = {1, 2, 0, 1, 1, 0, 0, -1, 0.1, 1};
  VecT x(x_h, x_h + n);

  /* initialize b and c to -1 to check if the function can overwrite it */
  VecT b(n, -1);

  /* run CooSpmv */
  CooSpMV<VecI, VecT>(Ai, Aj, Av, x, b);

  /* ref: the solution of b = A*x */
  T b_h[n] = {5, -6, 0, 0, 0, 1, 5.089, 1.6666666666666667, 4.141592653589793, 0};
  VecT b_ref(b_h, b_h + n);

  for (I i = 0; i < n; i++) {
    EXPECT_NEAR(b[i], b_ref[i], 1e-16);
    EXPECT_DOUBLE_EQ(b[i], b_ref[i]);
  }
}

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-16);
}
```



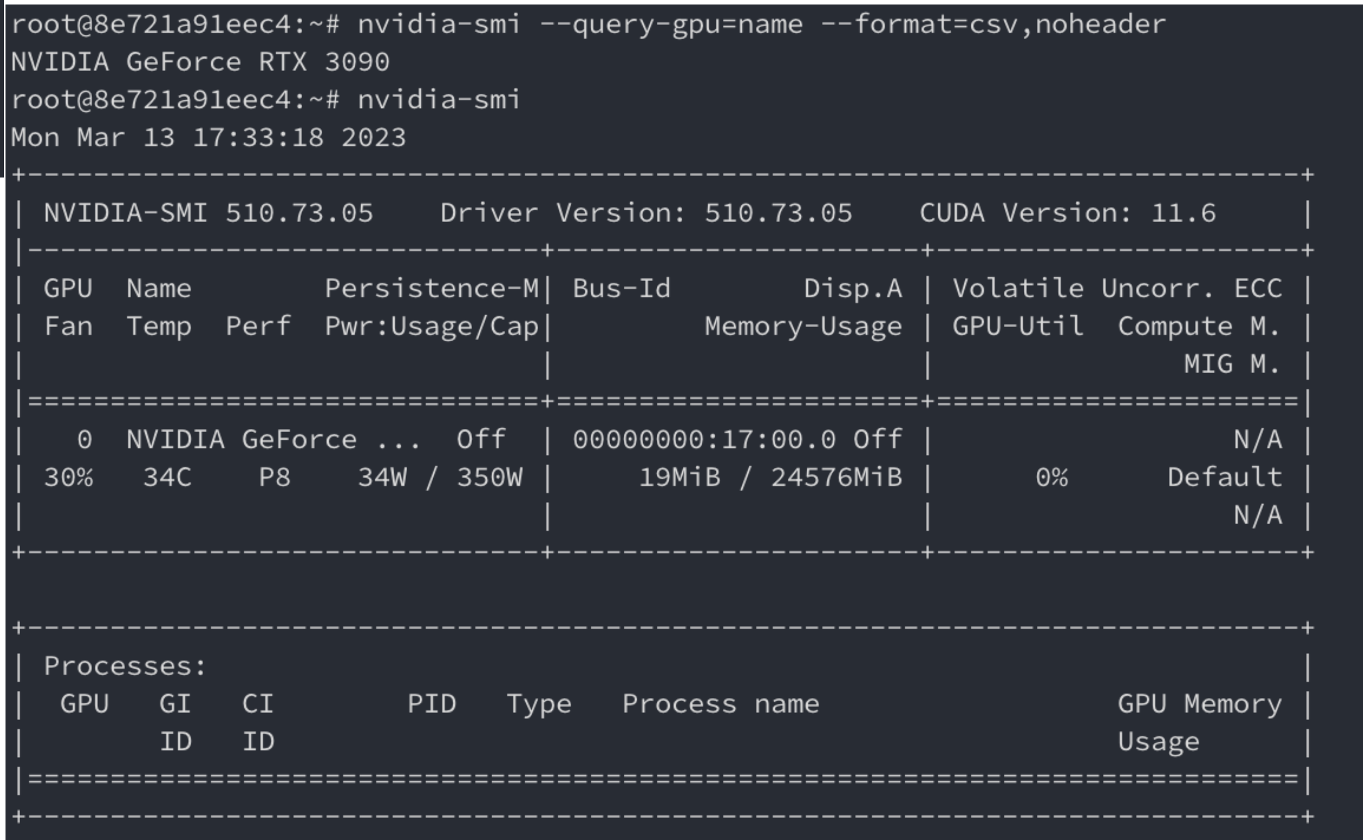
```python
"""
find the spectral radius of A using Arnoldi iteration for COO format
"""
def arnoldi_rho(Ai, Aj, Av, k):
    nnz = len(Av)
    n = Ai[-1] + 1

    # if k > n and n > 20, then k = n
    if k > n and n > 20:
        k = n

    # initialize the Arnoldi iteration
    Q = np.zeros((n, k+1))
    H = np.zeros((k+1, k))
    Q[:, 0] = np.ones(n) / np.sqrt(n)

    for j in range(k):
        v = np.zeros(n)
        for i in range(nnz):
            v[Ai[i]] += Av[i] * Q[Aj[i], j]
        for i in range(j+1):
            H[i, j] = np.dot(Q[:, i], v)
            v = v - H[i, j] * Q[:, i]

        # eps is machine precision
        H[j+1, j] = np.linalg.norm(v) + np.finfo(float).eps
        Q[:, j+1] = v / H[j+1, j]

    # resize the matrix H to k x k and find the spectral radius of H
    H = H[0:k, 0:k]

    # find the spectral radius of H manually
    x = np.ones(k)
    rho = 0
    for i in range(100):
        x = np.dot(H, x)
        rho = np.linalg.norm(x)
        x /= rho

    # compare the MSE of rho vs max_abs_eigs_A
    # transform Ai, Aj, Av to A
    A = np.zeros((n, n))
    for i in range(nnz):
        A[Ai[i], Aj[i]] = Av[i]

    max_abs_eigs_A = np.max(np.abs(np.linalg.eigvals(A)))

    # print in 16 digits
    print('max_abs_eigs_A =', np.format_float_scientific(max_abs_eigs_A, precision=16))
    print('arnoldi_rho_H =', np.format_float_scientific(rho, precision=16))

    return rho
```

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

- Novel and Challenge:
  - BSR Format
  - Kokkos + Kokkos Kernal
  - Thrust
- Application:
  - Large scale structure optimization problem: $Ku = f$
- Experiments (6 in total):
  - baseline using python: for checking the correctness
  - baseline using C++ vis BSR: for baseline performance
  - AMG + OpenMP + BSR
  - AMG + Kokkos + Kokkos Kernels + BSR (CPU, GPU version)
  - AMG + Thrust + cuSPARSE + COO (CPU, GPU version)
- Results:
  - Time to solve plot: strong scaling
  - Speed up plot vis baseline
  - Percentage performance reach via: cuSolverSP, GMRES Solver (Kokkos-Kernal)
  - Breakdown plot: computational cost for each function especially the Galerkin product

FLIP Fluid simulations on CUDA

Sorakrit Chonwattanagul

About FLIP

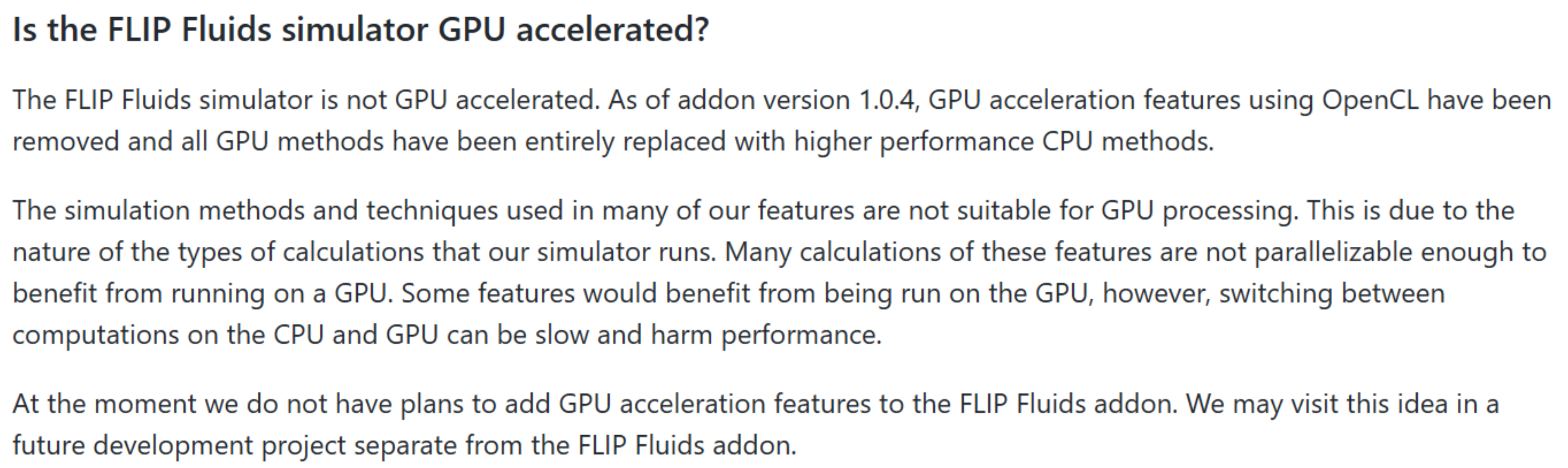
$$\frac{D\vec{u}}{Dt} + \frac{1}{\rho}\nabla p = \vec{g} + \nu\nabla \cdot \nabla\vec{u},$$

$$\nabla \cdot \vec{u} = 0$$

- Reproducibility project
- Fluid-Implicit-Particle (1988)
- Uses particles in a grid to simulate fluids and interaction with solids over time
- Approximates the incompressible Navier-Stokes equations
  - Advection equation
  - Body forces equation, e.g. gravity
  - Pressure equation (incompressibility)
- Can do liquids and gases
  - Bubbles and spray (diffuse particles) in water
- Often used in conjunction with particle-in-cell method (PIC; alone may add unintended viscosity/smoothness)

Existing implementations

- Industry standard for movie production, etc.
- Javascript example: https://matthias-research.github.io/pages/tenMinutePhysics/18-flip.html
- Blender Mantaflow/Houdini OpenCL



Is the FLIP Fluids simulator GPU accelerated?

The FLIP Fluids simulator is not GPU accelerated. As of addon version 1.0.4, GPU acceleration features using OpenCL have been removed and all GPU methods have been entirely replaced with higher performance CPU methods.

The simulation methods and techniques used in many of our features are not suitable for GPU processing. This is due to the nature of the types of calculations that our simulator runs. Many calculations of these features are not parallelizable enough to benefit from running on a GPU. Some features would benefit from being run on the GPU, however, switching between computations on the CPU and GPU can be slow and harm performance.

At the moment we do not have plans to add GPU acceleration features to the FLIP Fluids addon. We may visit this idea in a future development project separate from the FLIP Fluids addon.

Performance

- Time-to-solution
  - How long it takes to bake a simulation of a certain quality (simulation length, sub steps, grid resolution, number of particles)
- CPU: single-threaded, multi-threaded (OpenMP, threadpool)
- GPU: CUDA
- Compare solutions to single-threaded (sequential) simulation

Experiments

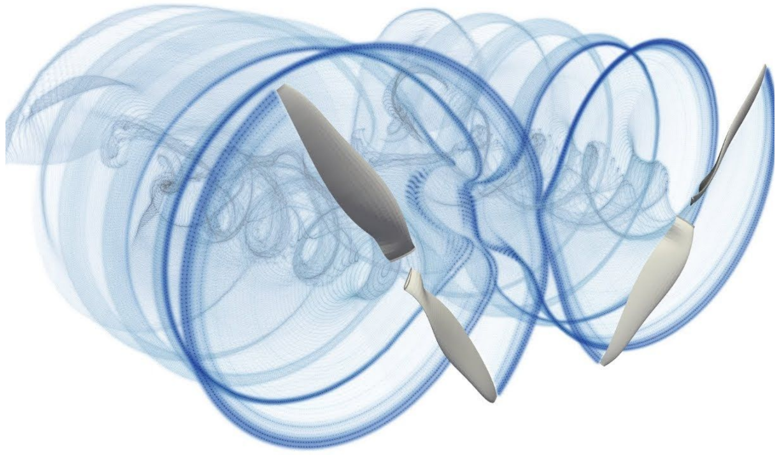
- Parameters
  - Simulation length
  - Sub steps
  - Grid resolution
  - Number of particles
- Testbed: most likely PACE output to VDB file
  - Maybe real-time simulation rendered on OpenGL (then laptop)
- Plots
  - Simulation time over quality parameters (above)

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 intend to reproduce this algorithm and optimize it for HPC GPU computing



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

Performance Metrics

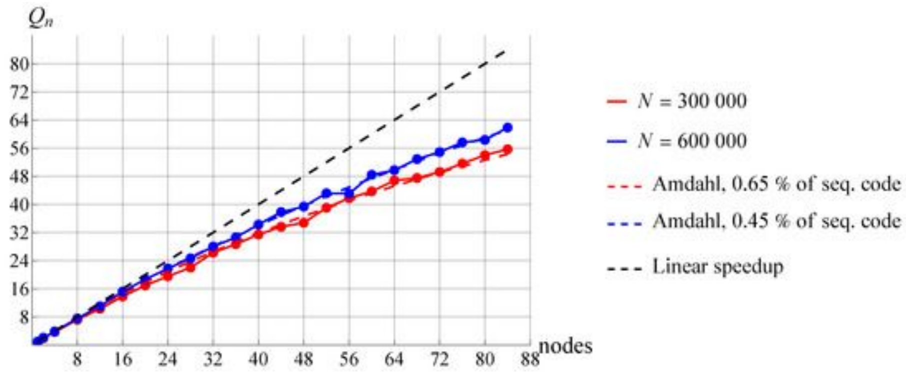
- Performance measure: simulation step time and algorithm scaling
  - Compare with varying particle counts
- Desired qualities of algorithm
  - Achieve good scaling on modern GPU hardware
  - Assess capability to conduct real-time simulations
    - Can we achieve real-time performance AND good results with lower-fidelity (e.g, smaller number of total particles) simulations?

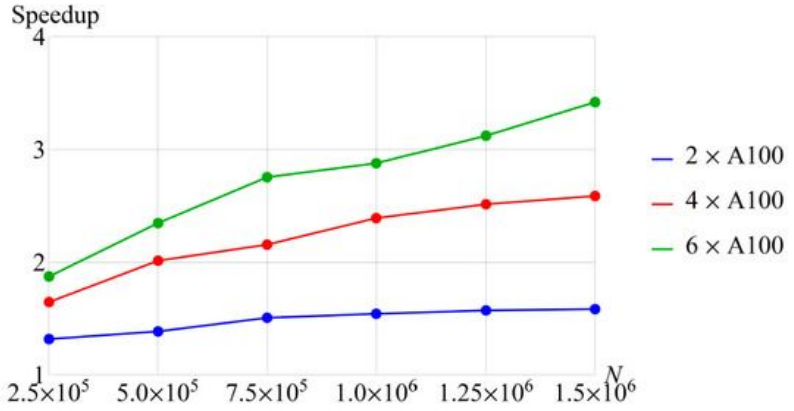
Plan for Solution

- Implement simple algorithm initially
  - At the beginning - keep it simple. Simplifying assumptions include
    - 2-dimensions only
    - Inviscid flow rather than viscous flow
- Use OpenACC to parallelize
- Validate and optimize solution
  - As project progresses, simplifying assumptions (2D and inviscid) could be relaxed depending on time constraints; however, more time is to be devoted to optimization of HPC implementation.
- Pitfalls
  - Numerical stability: vortex particle methods can have stability problems when two particles get too close together. To avoid this, many implementations “regrid” the particles onto a regularly-spaced lattice at periodic intervals to maintain numerical stability.
  - Clustering: vortex particle methods can also have issues where many of the particles cluster together in one region. The regridding should also help avoid this.

Baselines

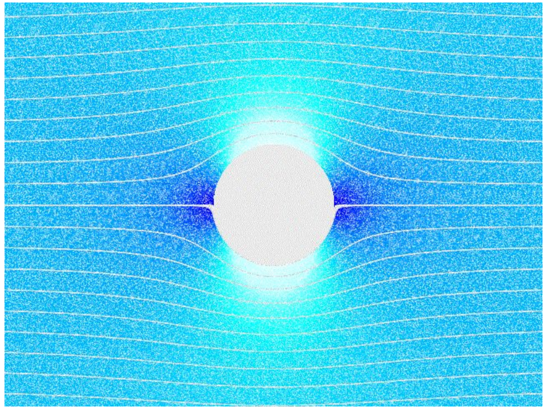
- Compare scaling and performance to existing implementations
  - VM2D
    - VM2D showed good scaling on CPUs, but not as good scaling on multiple GPUs
    - Figures on the right show scaling performance
  - CVortex
    - Open source code written in Julia
- Assess performance for many different problem sizes
  - Problem size is determined by number of particles simulated

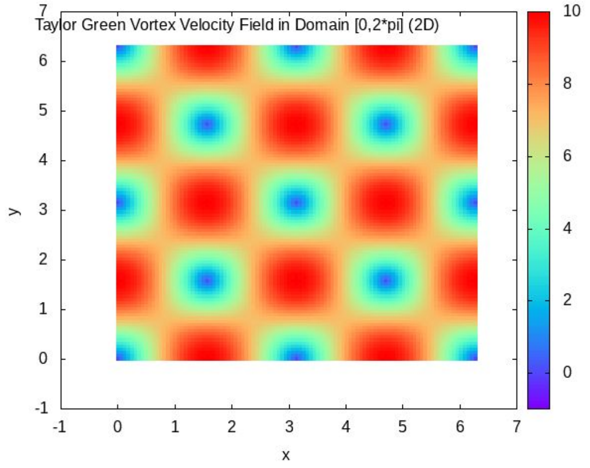




Validation

- Several canonical flows available for validation
  - Compare our computed solution to the known solution
- Canonical flows available
  - Flow over sphere
  - Flow over thin airfoil
  - Taylor-Green vortex (viscous only)
- Visualization
  - Rendering our results could be used for visual comparisons





Output Data for Validation

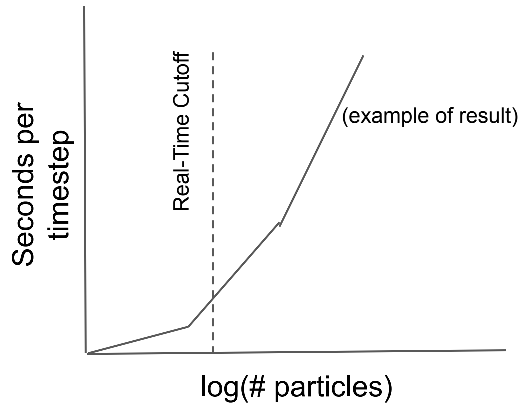
- Data output from simulations will be velocity field
  - Compare velocity contours to canonical reference solutions
- Additional potential output datasets:
  - Drag and lift coefficients for a thin airfoil simulation
- Postprocessing using widely-available visualization software
  - Paraview
  - Tecplot
- Since this is a particle-based method, not as easy to generate output slices over a grid
  - In order to do this, some interpolation will be required
  - To avoid excess interpolation, can combine dataset output with regridding procedure mentioned before.

Testbeds

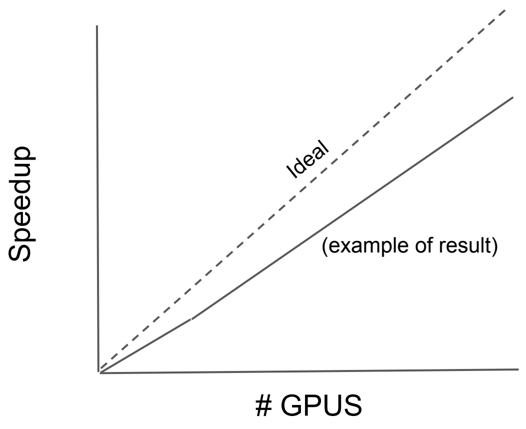
- Assess both single GPU capability and multi-GPU capability
- Single GPU: Run on local workstations/desktop computers
- Multi-GPU:
  - COC-ICE: Can parallelize over NVIDIA V100s
  - NVIDIA DGX A100 system available through Shreyas' research lab
    - 8x NVIDIA A100, interconnected through NVLink high-bandwidth connection
    - NVLink will reduce communication times between GPUs, drastically improving multi-GPU scaling

Potential Plots

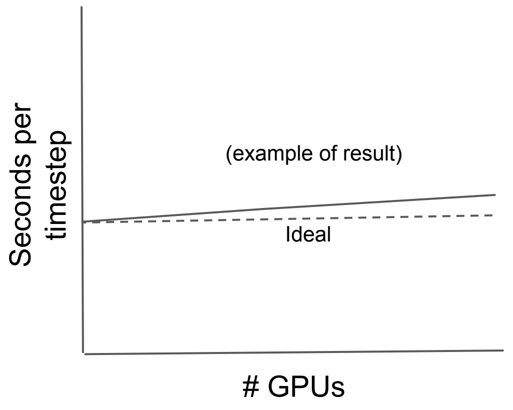
Single GPU Scaling Test



Multi-GPU Strong Scaling Test



Multi-GPU Weak Scaling Test



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=BrBlpeYNTMc

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