Accelerating Physics: Multi-GPU Scaling for Science!

Talk by: Paul Mekhedjian

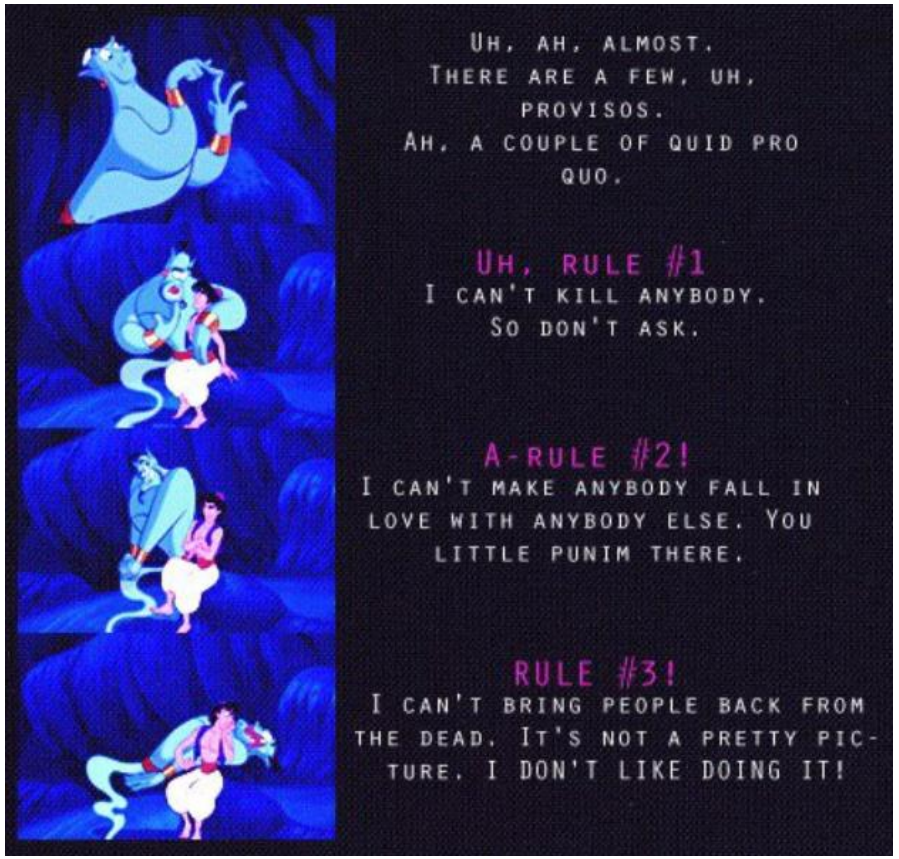
Room 107, SCaLE 23x
Pasadena, CA

Friday, March 6th, 2025
12:30 – 1:30 pm

Talk Objectives and Agenda

- Points of Order, Introductions, Motivation
- Recap of last year's work
- New progress made and paths forward
- Next Steps

Points of Order… a few, uh, provisos…



4. Views are my own, and not that of my employer.
5. I am **not** a programmer. I am **not** a software developer.
   🎵 ***I am a problem solver***. 🎶
6. With some work and effort, maybe you can do this, too? 😊
7. I don’t work for Nvidia, but I use their hardware and software, so apologies in advance.
8. My choice of Nvidia GPUs for this presentation is only due to availability.
9. I am not promoting any one idea, but this is simply what I have done.
10. ***Takeaway***: If you have a novel idea, use **anything** that works for you and gets you **results**!

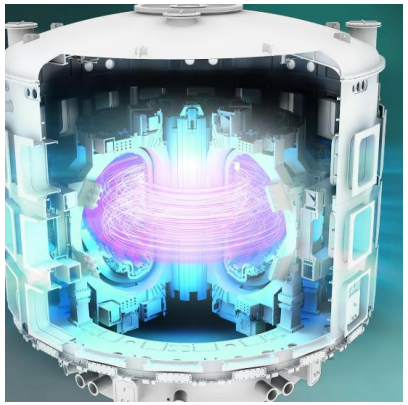
Where I  *am coming from…*

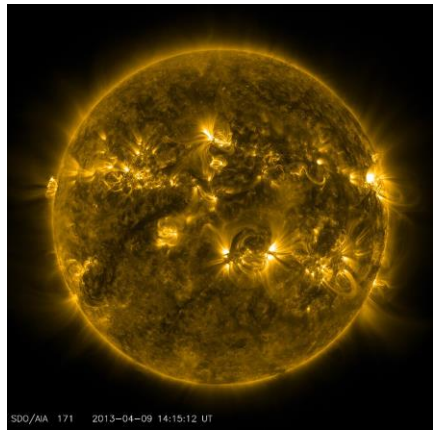
- Los Angeles native, husband, and dad! ~~hiker~~, ~~camper~~
- B.S. + M.Sc. in Physics, Ph.D. (in progress) in Aerospace Engineering
- 6th SCaLE, 2nd talk ever
- Linux power user (at home)
- (Former) HPC Linux engineer (at work)
- Since Dec ‘25, I work with RF signal processing

Python Motivation

- I want to port personal and academic projects to **Python** and take advantage of new **hardware**
- Python is **easy** to learn, and it's a **myth** that it is **slow** at everything (quod postea ostendam)
- Python is a **dynamically-typed*** language and higher level than Fortran or C/C++
- **Nothing** wrong with Fortran or C/C++, but I honestly don't have the time.
- One ***can*** speed up using **Cython** and **Numba**, but I seek scalability onto multiple compute targets (not just CPU cores).

My Motivations and Hopes for the Future





- I am a PhD student, and I research **turbulent** fluid flows, as seen in **nuclear fusion** environments as well as in **astrophysical** settings
- Turbulence is a **nonlinear** and **multi-scale** problem
- There are **interdisciplinary** benefits to improving our understanding of this

Why Multi-GPU?

- **Sizing**
  - **Memory constraints – Not all problems fit on 1 GPU**
- **Scalability**
  - Most systems only have enough space, power for one
- As power supplies, motherboards, and servers continue to grow **multi-GPUs will become more commonplace**
  - If not multi-GPU on one system, then multi-GPU across networked nodes
- Complexity rooted in questions (a la **non-linearity** of real-life problems) demands complexity in answers (i.e. models will **not be linear**)
- Examples of multi-GPU compute include:
  - **Datacenters**: NVIDIA B200, H200, H100, A100
  - **Workstation/Pro**: RTX 6000, RTX A5000
  - **Consumer/Gaming**: 4090, 5090
    - **Do not** support NVLink or NVSwitch

CUDA Python suite

- **CuPy**: NumPy/SciPy-compatible array library for GPUs
  - A **drop-in replacement** for NumPy that runs on GPUs
- CUDA Python (from Nvidia) = CuPy + nvmath + Numba-CUDA
- Of course, to leverage **CuPy**, programs **must** be written in Python
  - But this is a good thing for us domain specialists…

- **Multi-GPU** capabilities also available in C++ and Fortran via CUDA
  - Makes it easier to port applications from CUDA {C, Fortran} to CuPy
  - Multi-GPU, Multi-Node scaling is possible via CUDA's NCCL library

Conditions and Requirements:

- How CUDA is structured must be well understood if you want to solve engineering problems in this paradigm
- **Everything** must be seen and considered through the eyes of CPU (host) vs. GPU (device)
- Failing to do this means adding overhead, complexity, and compilation/runtime errors to your codebase
  - **Most projects** looking to use CUDA **will suffer a quick death** in this way
- How can I solve a simple math problem using a GPU? How can I use CUDA to do it?
  - Architecting your programs from the very beginning to do this is easier than later

Bringing CUDA's GPU capabilities to my headspace

- I am not a computer scientist, but I consider myself a problem solver.
    - I know that to solve the hardest problems, I need clever solutions
- ***Running*** Python for complex simulations can be slow, but ***using*** Python is a delight, and makes the code feel ***familiar*** to me
- So, how can we make the experience better and make this awesome tech accessible?
    - Have your cake and eat it, too!

Recap of main points from last talk

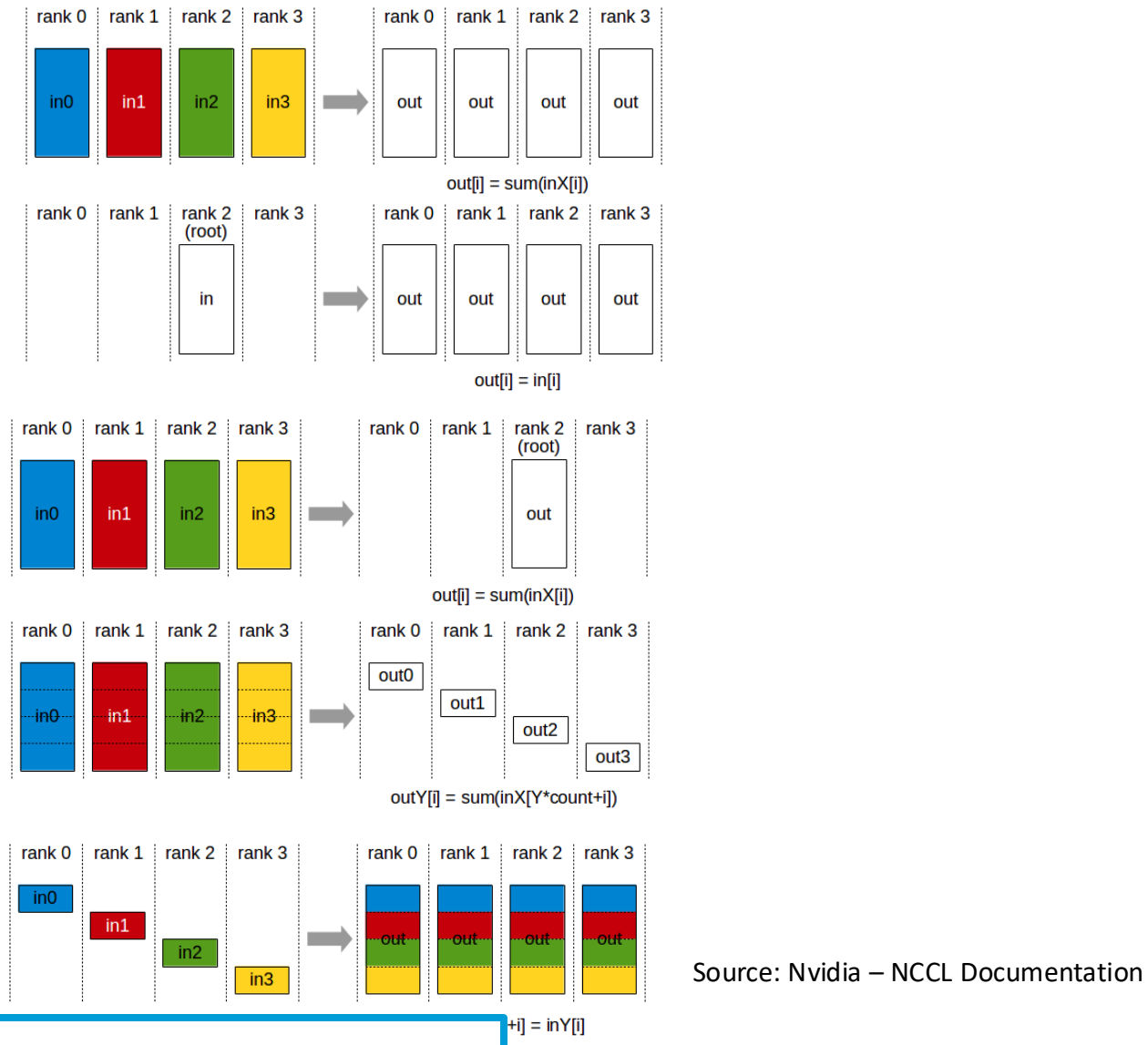
CUDA: NVIDIA Collective Communications Library (NCCL)

- CUDA NCCL:
  - Optimized primitives for collective multi-GPU communication
  - Enables GPU compute for/on:
    - Multiple GPUs (on **same** system)
    - Multi-Node GPUs (on **different** systems)
    - Multi-Node Multi-GPU – MNMG (on **different** systems AND **multi-GPUs**)

- NCCL compatible with **any multi-GPU model**:
  - Single-thread control of all GPUs
  - Multi-threaded (e.g. using one thread per GPU)
  - Multi-process (e.g. MPI)

Canonical Problem 2 – NCCL Operations

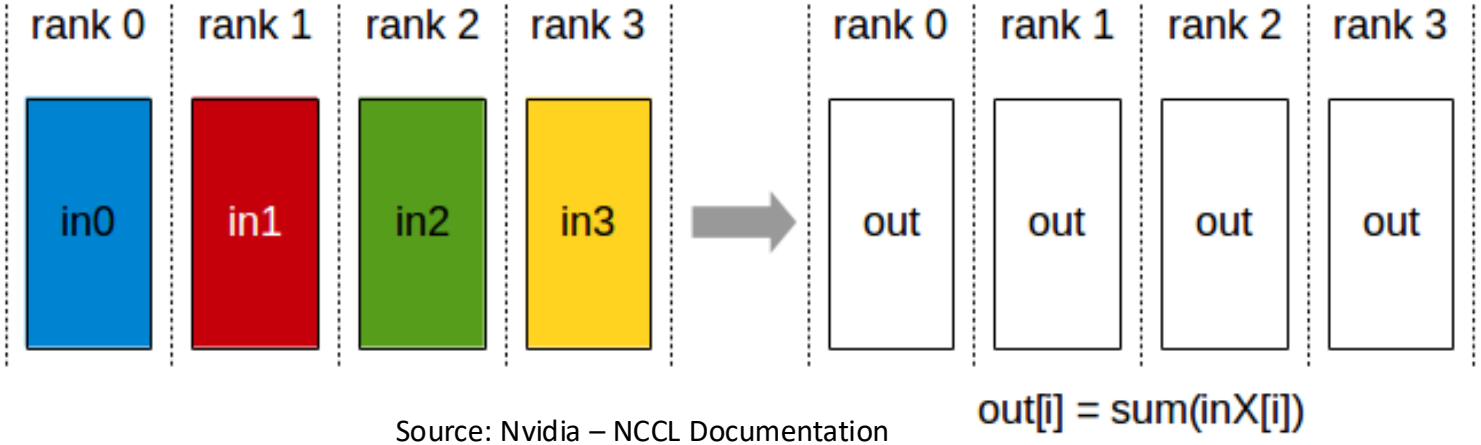
- AllReduce
- Broadcast
- Reduce
- ReduceScatter
- AllGather



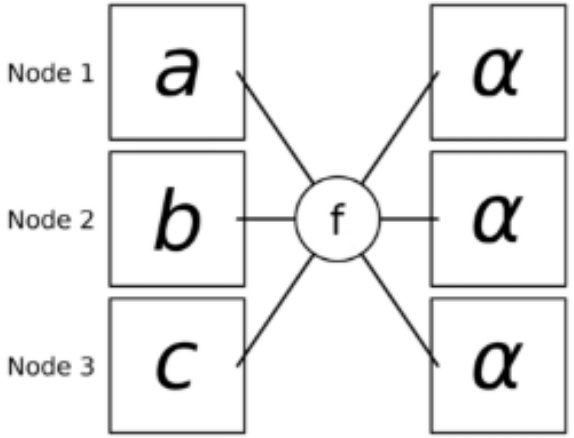
Source: Nvidia – NCCL Documentation

Canonical Problem 2 – NCCL Operations

- AllReduce



Source: Nvidia – NCCL Documentation



Source: Wikipedia – Collective operation

- AllReduce comes with an associative operator **f** that performs

  some **operations** or **reductions** on the data

  (e.g. sum, min, max, product, logical/bitwise AND/ORs)

Canonical Problem 2 – NCCL cont'd

- Defining Advection class:
  - Starting with an array of 1's

```python
import cupy as cp
import cupy.cuda.nccl as nccl
import multiprocessing as mp
```

```python
class Advection:
    def __init__(self):
        self.data = cp.ones(10)

    def print_data(self):
        print(self.data)
```

```python
def run_example(example_name, example_func):
    """
    Spawns multiple processes (one per GPU rank) to run the communication operation.
    Returns a dictionary (rank -> numpy array) for the main process to plot.
    """
    size = 8
    unique_id = nccl.get_unique_id()

    manager = mp.Manager()
    results_dict = manager.dict()

    processes = []
    for rank in range(size):
        p = mp.Process(target=example_func, args=(rank, size, unique_id, results_dict))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    return dict(results_dict)
```

```python
def allreduce_example(rank, size, unique_id, results):
    """
    Perform AllReduce (sum) on the Advection data and store
    the resulting array into the shared dictionary (results)
    so it can be plotted later by the main process.
    """
    cp.cuda.Device(rank).use()
    comm = nccl.NcclCommunicator(size, unique_id, rank)

    adv1 = Advection()
    reduced_data = cp.empty(10, dtype=cp.float64)

    print(f"Rank {rank}: Performing AllReduce operation")
    comm.allReduce(
        adv1.data.data.ptr,
        reduced_data.data.ptr,
        reduced_data.size,
        nccl.NCCL_FLOAT64,
        nccl.NCCL_SUM,
        cp.cuda.Stream.null.ptr
    )

    print(f"Rank {rank}: AllReduce result:")
    print(reduced_data)

    # Move result to host so we can store it in results dict
    results[rank] = reduced_data.get()

    cp.cuda.Device().synchronize()
    cp.get_default_memory_pool().free_all_blocks()
```

Canonical Problem 2 – NCCL Operations cont'd

- At runtime:
  - When it runs, GPUs all churn…

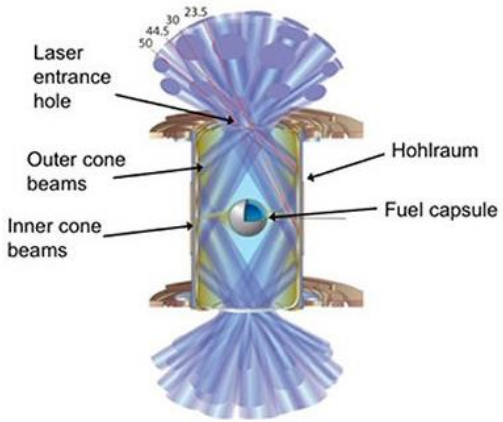
DEV	TYPE	GPU	GPU MEM	CPU	HOST MEM
7	Compute	2%	1022MiB	1%	58%	388MiB
6	Compute	2%	1022MiB	1%	64%	388MiB
5	Compute	2%	1022MiB	1%	61%	388MiB
4	Compute	2%	1022MiB	1%	59%	388MiB
3	Compute	2%	1022MiB	1%	60%	388MiB
2	Compute	2%	1022MiB	1%	60%	388MiB
1	Compute	2%	1022MiB	1%	77%	388MiB
0	Compute	2%	1022MiB	1%	56%	388MiB

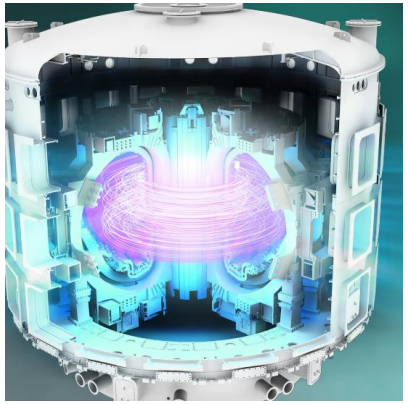
  - Outputs to screen (stdout)

```
Running AllReduce example:
Rank 7: Performing AllReduce operation
Rank 3: Performing AllReduce operation
Rank 0: Performing AllReduce operation
Rank 4: Performing AllReduce operation
Rank 5: Performing AllReduce operation
Rank 1: Performing AllReduce operation
Rank 2: Performing AllReduce operation
Rank 6: Performing AllReduce operation
Rank 5: AllReduce result:
Rank 1: AllReduce result:
Rank 4: AllReduce result:
Rank 7: AllReduce result:
Rank 3: AllReduce result:
Rank 2: AllReduce result:
Rank 6: AllReduce result:
Rank 0: AllReduce result:
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
[8. 8. 8. 8. 8. 8. 8. 8. 8. 8.]
```

But, how can I apply this to engineering?

Turbulent Fluids in Computational Fluid Dynamics (CFD)





Turbulent Fluids in Computational Fluid Dynamics (CFD)

- A field that has NOT benefitted from GPU computing as much as AI/ML
- Modern hardware provides more compute capability than scientists and engineers are **currently** leveraging.
  - Problems **can** lend themselves to be solved on GPUs efficiently.
- Turbulence problems are **nonlinear** and **multi-scale**, and that complexity demands high compute capacity.

Canonical Problem – 2D Poisson Equation

- 2D Poisson equation from physics

$$\nabla^2 \varphi = f$$

- Equation is discretized to be solvable on a 2D grid
  in x, y: central finite difference

$$\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} = 0 \qquad \nabla^2 p = f$$

- Assuming uniform spatial discretization: When paired with incompressibility condition and pressure:

$$(\nabla^2 u)_{ij} = \frac{1}{\Delta x^2}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) = g_{ij}$$

- Discrete Poisson coincidentally arises in the theory of Markov chains (probability theory)

Canonical Problem – 2D Poisson Equation

- Jacobi iteration is used to solve discrete 2D Poisson equation

```python
def jacobi_iteration(u, f, dx, dy, iters=1000):
    """
    Perform Jacobi iterations to solve the Poisson equation:
      ∇²u = f
    where u is the unknown potential (e.g. pressure),
    and f is a source term.
    """
    nx, ny = u.shape
    unew = cp.empty_like(u)
    for _ in range(iters):
        unew[1:-1, 1:-1] = 0.25 * (
            u[1:-1, :-2] + u[1:-1, 2:] +
            u[:-2, 1:-1] + u[2:, 1:-1] -
            f[1:-1, 1:-1] * dx * dy
        )
        # Swap references
        u, unew = unew, u
    return u
```

- Solver function then calls the Jacobi iterator using CUDA (via CuPy):

```python
def solve_poisson_multi_gpu(domain_np, iters=1000):
    """
    Demonstrates splitting the domain across two GPUs using CuPy
    and performing Jacobi iterations for the Poisson equation.
    """
    # Split domain into two halves along y dimension
    ny, nx = domain_np.shape
    half = ny // 2

    # Domain sub-blocks for each GPU
    domain_np_0 = domain_np[:half, :]
    domain_np_1 = domain_np[half:, :]

    # Two devices (assumes at least 2 GPUs are available)
    dev0 = cp.cuda.Device(0)
    dev1 = cp.cuda.Device(1)

    # Allocate arrays on each GPU
    with dev0:
        u0 = cp.zeros_like(cp.asarray(domain_np_0))
        f0 = cp.asarray(domain_np_0)
    with dev1:
        u1 = cp.zeros_like(cp.asarray(domain_np_1))
        f1 = cp.asarray(domain_np_1)

    dx = 1.0
    dy = 1.0
```

Canonical Problem – 2D Poisson Equation (2DPE)

- Then, after running the program multiple times with different nx, ny (problem sizes)

```python
def main():
    # Problem size
    nx, ny = 512, 512
    domain_np = initialize_domain(nx, ny)

    # Multi-GPU solve
    u_multi, time_multi = solve_poisson_multi_gpu(domain_np, iters=200)
    print(f"Multi-GPU Time: {time_multi:.3f} s")

    # Single GPU solve (for comparison)
    domain_gpu = cp.asarray(domain_np)
```

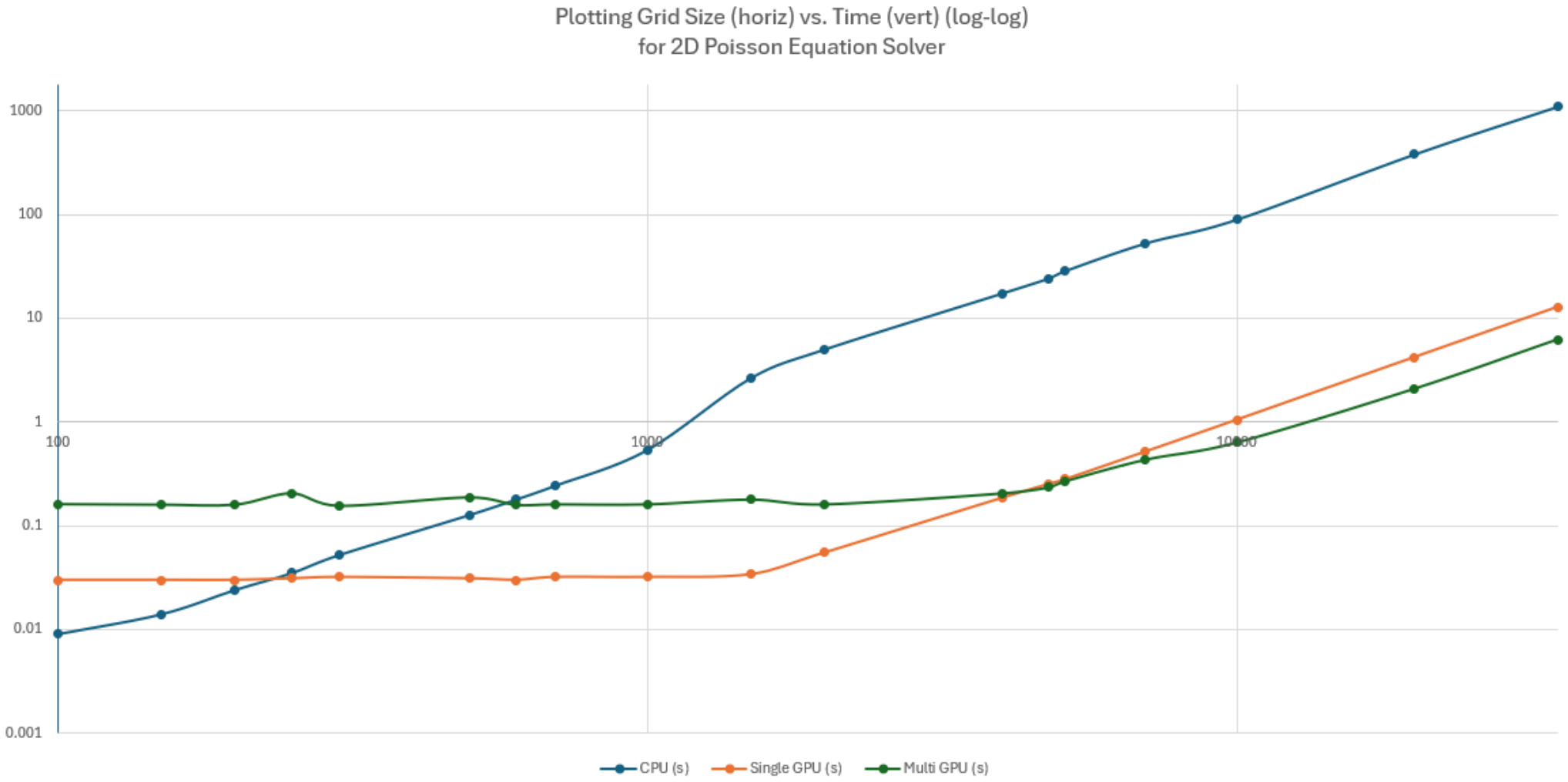
- The grid is split and each piece goes to a GPU separately to work in parallel

- Results

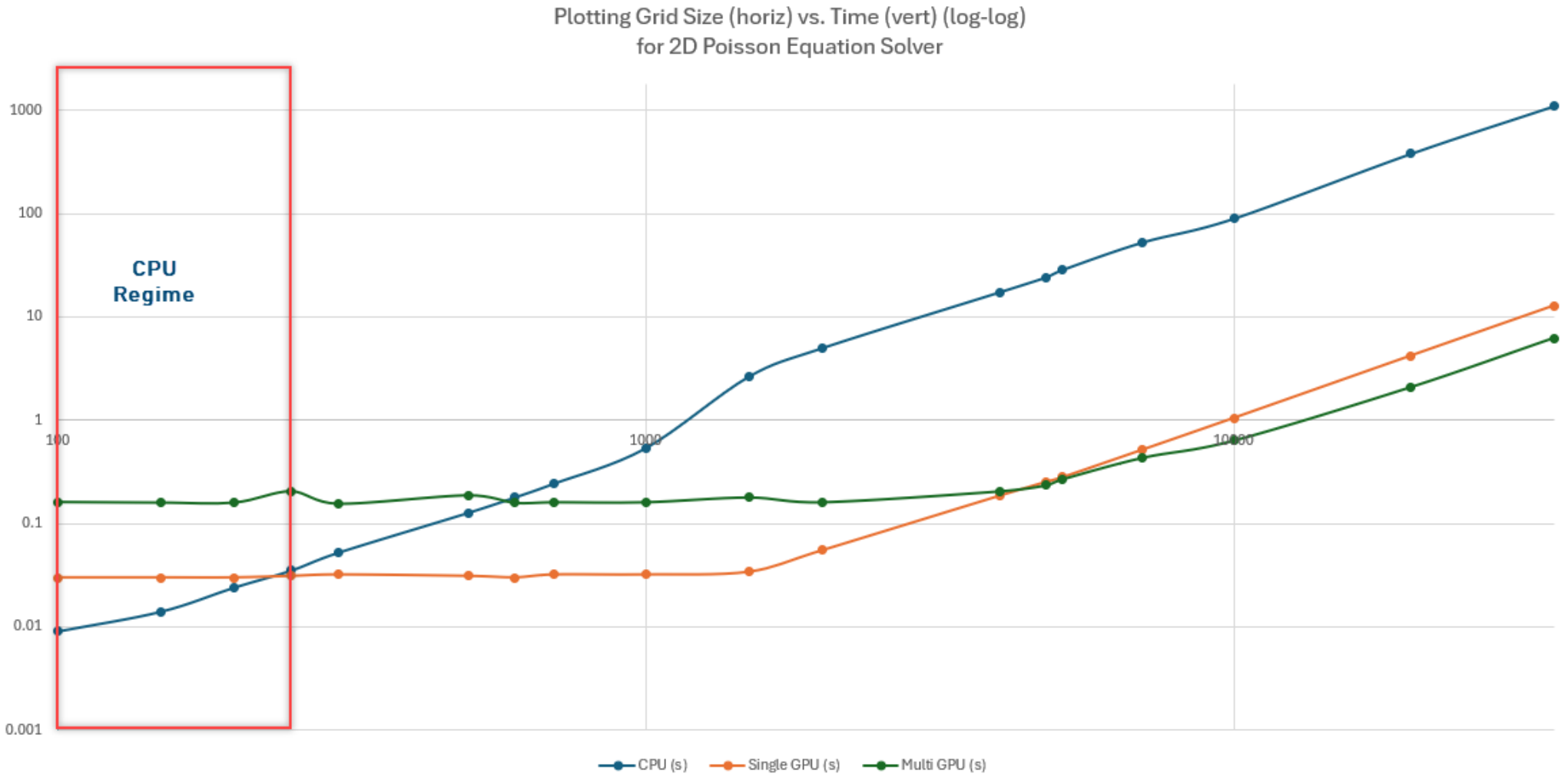
where Grid Size N is nx=ny=N

Grid Size N (nx=ny=N)	CPU (s)	Single GPU (s)	Multi GPU (s)
100	0.009	0.03	0.162
150	0.014	0.03	0.16
200	0.024	0.03	0.16
250	0.035	0.031	0.206
300	0.052	0.032	0.156
500	0.127	0.031	0.188
600	0.179	0.03	0.159
700	0.244	0.032	0.161
1000	0.534	0.032	0.161
1500	2.648	0.034	0.179
2000	5.002	0.055	0.161
4000	17.335	0.185	0.205
4800	24.188	0.252	0.234
5120	28.549	0.28	0.268
7000	52.625	0.517	0.432
10000	89.236	1.048	0.633
20000	378.458	4.205	2.08
35000	1093.5	12.836	6.209

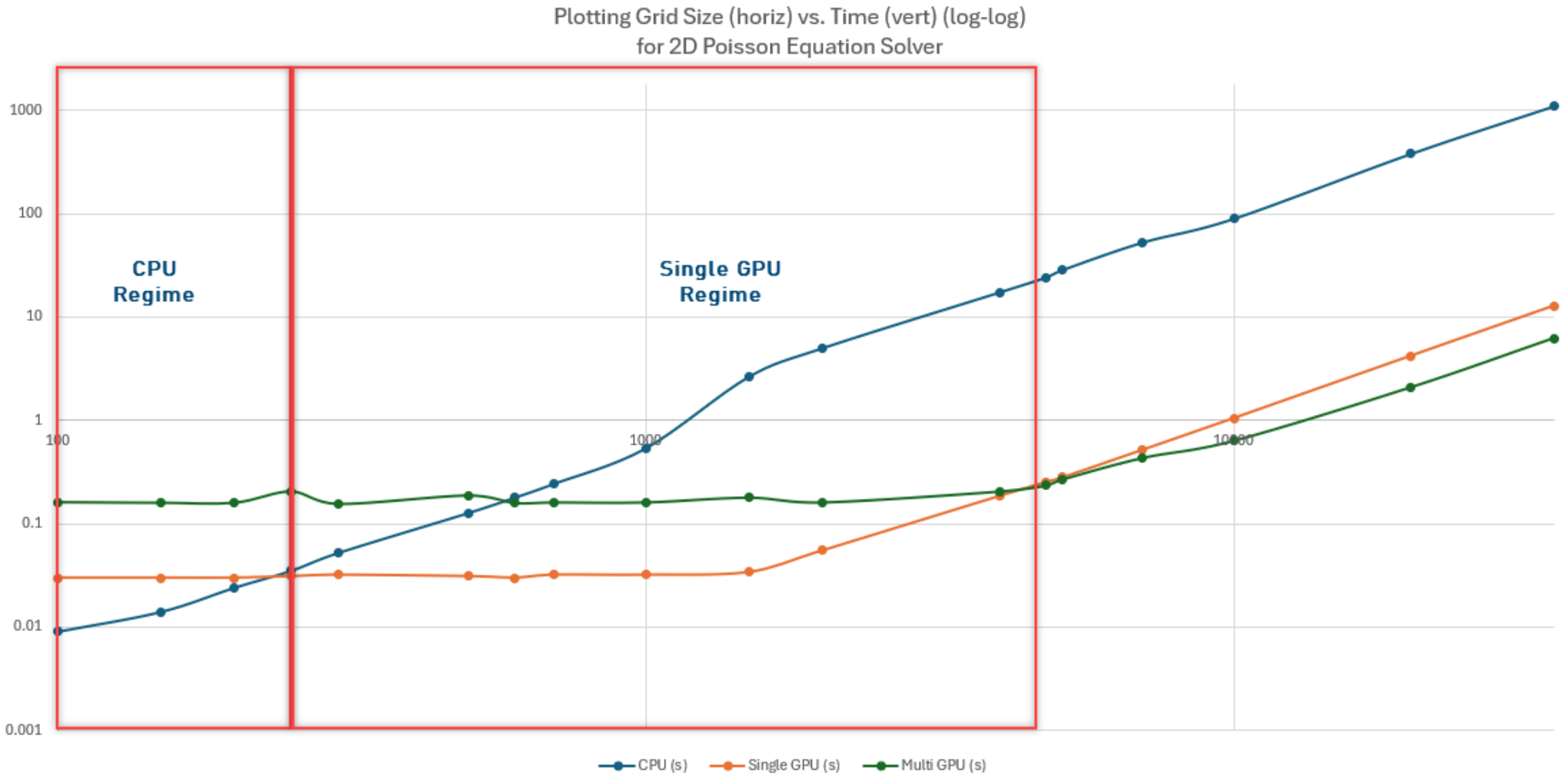
Canonical Problem 3 – 2D Poisson Equation (2DPE) (CPU vs. 1 GPU vs. 2 GPUs)



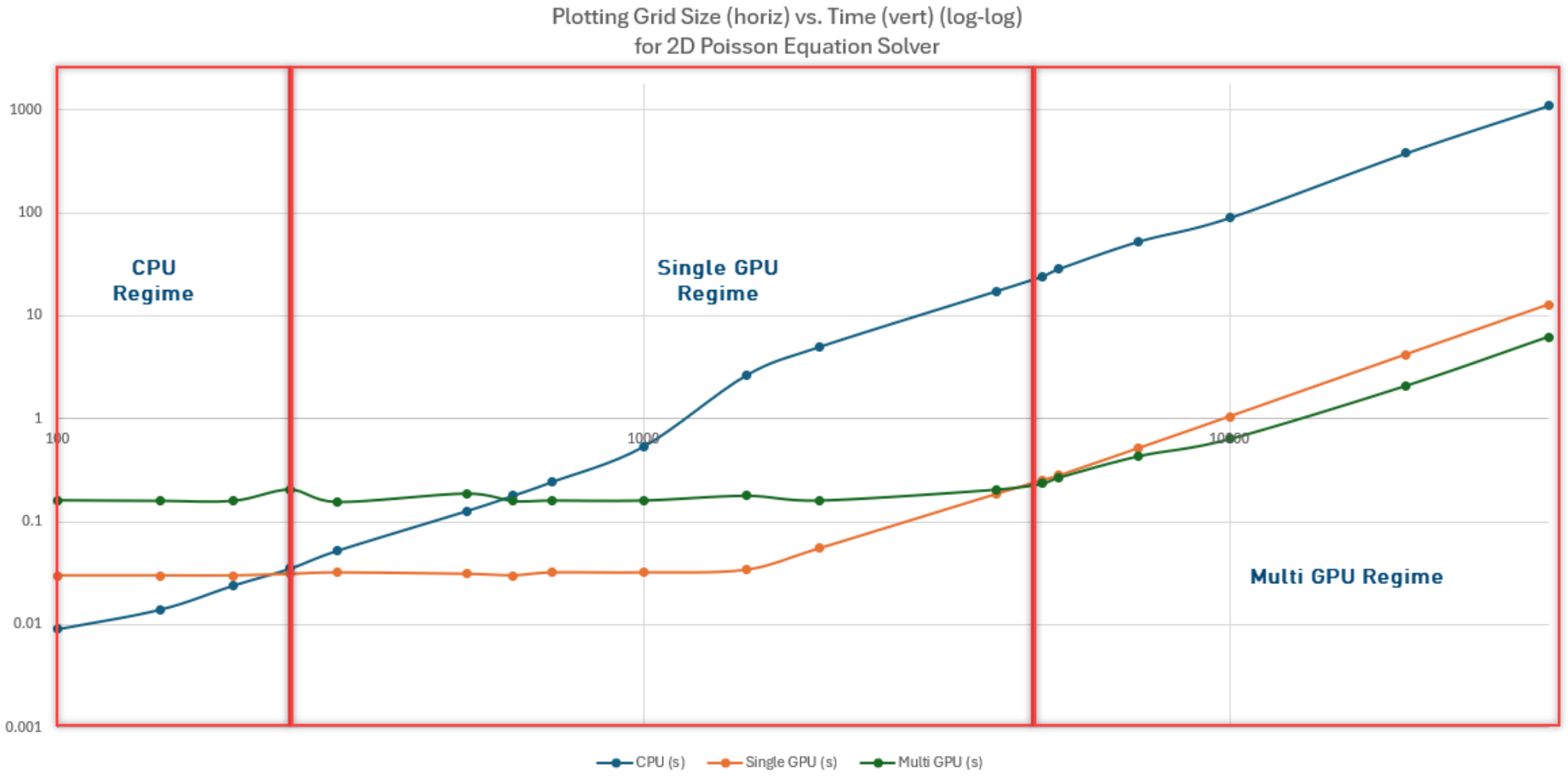
Canonical Problem 3 – 2D Poisson Equation (2DPE) (CPU vs. 1 GPU vs. 2 GPUs)



Canonical Problem 3 – 2D Poisson Equation (2DPE) (CPU vs. 1 GPU vs. 2 GPUs)



Canonical Problem 3 – 2D Poisson Equation (2DPE) (CPU vs. 1 GPU vs. 2 GPUs)



Looking again at the 2D Poisson Equation (2DPE)

- 2D Poisson equation from physics

$$\nabla^2 \varphi = f$$

- Equation is discretized to be solvable on a 2D grid in x, y: finite difference

$$\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} = 0 \qquad \nabla^2 p = f$$

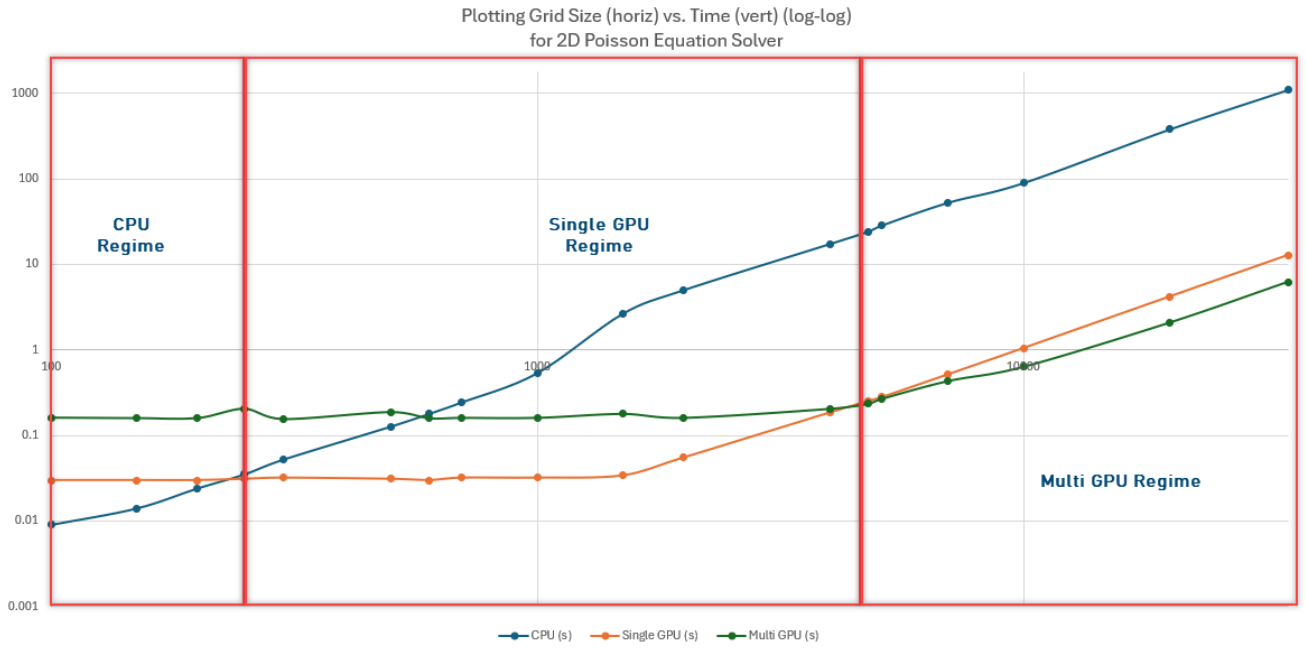
- Assuming uniform spatial discretization: When paired with incompressibility condition and pressure:

$$(\nabla^2 u)_{ij} = \frac{1}{\Delta x^2}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) = g_{ij}$$

- Discrete Poisson coincidentally arises in the theory of Markov chains (probability theory)

Canonical Problem 3 – 2D Poisson Equation (2DPE)

- Looking at the data from last year…



- Characteristic choices in grid size:

Grid Size N (nx=ny=N)	CPU (s)	Single GPU (s)	Multi GPU (s)
100	0.009	0.03	0.162
150	0.014	0.03	0.16
200	0.024	0.03	0.16
250	0.035	0.031	0.206
300	0.052	0.032	0.156
500	0.127	0.031	0.188
600	0.179	0.03	0.159
700	0.244	0.032	0.161
1000	0.534	0.032	0.161
1500	2.648	0.034	0.179
2000	5.002	0.055	0.161
4000	17.335	0.185	0.205
4800	24.188	0.252	0.234
5120	28.549	0.28	0.268
7000	52.625	0.517	0.432
10000	89.236	1.048	0.633
20000	378.458	4.205	2.08
35000	1093.5	12.836	6.209

This year I considered Burgers' Equation…

Johannes (Jan) Martinus Burgers
1895 - 1981

- Jan Burgers researched fluid dynamics
  - Worked on theory of turbulence

- Burgers' equation in one spatial (1D) and transient (*t*) dimension:
  - 1D (in *x*) **dissipative** system (when *v* is non-zero):

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

  - Nonlinear due to convective/adjective term ($u\partial u/\partial x$).
  - More precisely, is "quasilinear" due to *u* term. u *can* be c(u).
  - When *v* term is absent (no viscosity), this is a special case:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

2D Burgers' Equation (2DBE)

- 2DBE in vector form (bold is vectorized):

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = \nu \nabla^2 \mathbf{u}$$

- Expanded into components, keeping in mind that u=u(x,y):

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}$$

- In two dimensions (x, y) this looks like:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right)$$

- Discretization (to be computable) follows…

$$(\nabla^2 u)_{ij} = \frac{1}{\Delta x^2}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) = g_{ij}$$

- 1D: 1 nonlinear term in 1DBE
  2D: 4 nonlinear terms in 2DBE
  3D: 9 nonlinear terms in 3DBE

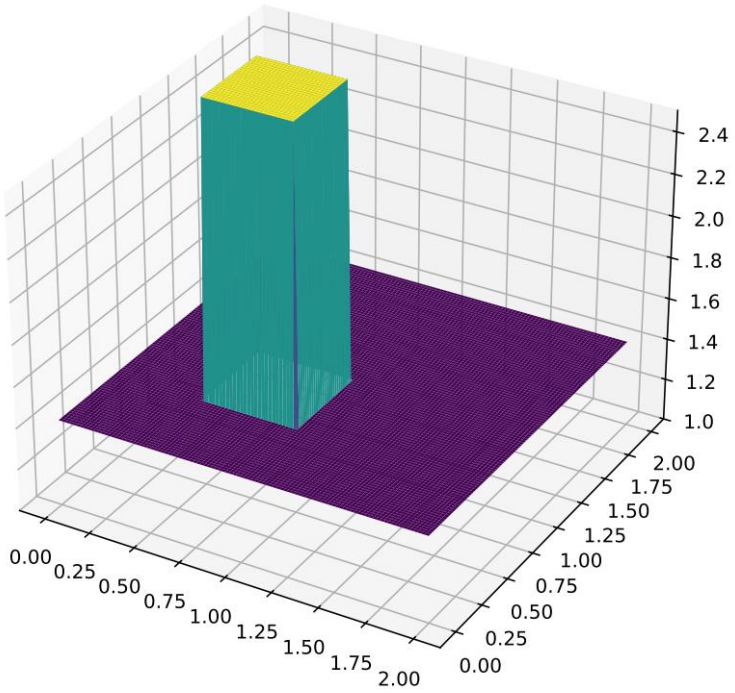
2D Poisson vs. 2D Burgers'

	2D Poisson	2D Burger
Type	Elliptic PDE	Parabolic/Hyperbolic PDE
Time dependence?	No (steady state)	Yes
Linearity	Linear	Nonlinear
Viscosity?	No	Yes
Jacobi iteration?	Yes (due to linearity)	No (must be computed directly)

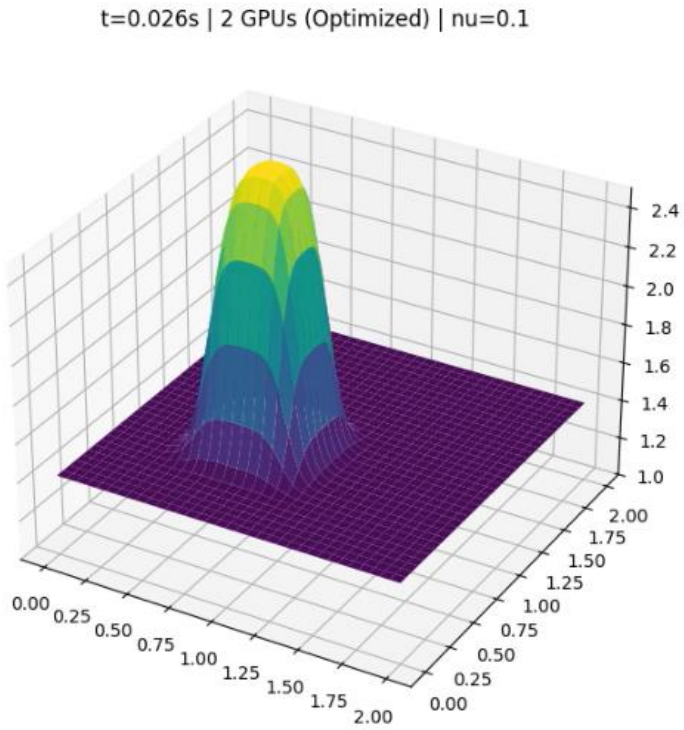
I am **intentionally complicating** the math by introducing **time dependence** and adding **nonlinear terms**, in order to push the limits of the **types of compute** that GPUs handle best (or worst).

What do solutions to 2D Burgers' look like?

Initial condition:



burgers_220537.png



When solved using a Python+CuPy solver, we see:

```
for i in range(n_gpus):
    with cp.cuda.Device(i):
        u, v, un, vn, stream = u_gpu[i], v_gpu[i], un_gpu[i], vn_gpu[i], streams[i]
        stream.synchronize()

        un[:] = u[:]; vn[:] = v[:]

        u[1:-1, 1:-1] = (un[1:-1, 1:-1] -
            dt / dx * un[1:-1, 1:-1] * (un[1:-1, 1:-1] - un[0:-2, 1:-1]) -
            dt / dy * vn[1:-1, 1:-1] * (un[1:-1, 1:-1] - un[1:-1, 0:-2]) +
            nu * dt / dx**2 * (un[2:, 1:-1] - 2 * un[1:-1, 1:-1] + un[0:-2, 1:-1]) +
            nu * dt / dy**2 * (un[1:-1, 2:] - 2 * un[1:-1, 1:-1] + un[1:-1, 0:-2]))

        v[1:-1, 1:-1] = (vn[1:-1, 1:-1] -
            dt / dx * un[1:-1, 1:-1] * (vn[1:-1, 1:-1] - vn[0:-2, 1:-1]) -
            dt / dy * vn[1:-1, 1:-1] * (vn[1:-1, 1:-1] - vn[1:-1, 0:-2]) +
            nu * dt / dx**2 * (vn[2:, 1:-1] - 2 * vn[1:-1, 1:-1] + vn[0:-2, 1:-1]) +
            nu * dt / dy**2 * (vn[1:-1, 2:] - 2 * vn[1:-1, 1:-1] + vn[1:-1, 0:-2]))

        bc_val_u = cp.array(1, dtype=u.dtype)
        bc_val_v = cp.array(1, dtype=v.dtype)

        if i == 0: u[1, :] = bc_val_u; v[1, :] = bc_val_v
        if i == n_gpus - 1: u[ny_local, :] = bc_val_u; v[ny_local, :] = bc_val_v
        u[:, 0] = bc_val_u; u[:, -1] = bc_val_u
        v[:, 0] = bc_val_v; v[:, -1] = bc_val_v
```

- Time to completion (in minutes), with increasing grid size (*nx* x *ny*)

# of GPUs	2048^2	4096^2	8192^2
2	**8.6**	115.6	1552
4	15.3	**61.5**	895
6	22.5	88.8	606
8	29.8	118.1	**473**

When solved using a CUDA kernel inside Python with CuPy, we see:

```
extern "C" __global__
void burgers_step(
    double* u, double* v, const double* un, const double* vn,
    int ny_local_padded, int nx,
    double dt_dx, double dt_dy, double nu_dt_dx2, double nu_dt_dy2
) {
    int i = blockIdx.y * blockDim.y + threadIdx.y + 1;  // y index (skip halo)
    int j = blockIdx.x * blockDim.x + threadIdx.x + 1;  // x index (skip boundary)

    if (i < ny_local_padded - 1 && j < nx - 1) {
        int idx = i * nx + j;
        int idx_im1 = (i - 1) * nx + j;  // y-1
        int idx_ip1 = (i + 1) * nx + j;  // y+1
        int idx_jm1 = i * nx + (j - 1);  // x-1
        int idx_jp1 = i * nx + (j + 1);  // x+1

        double un_c = un[idx];
        double vn_c = vn[idx];

        u[idx] = un_c
            - dt_dx * un_c * (un_c - un[idx_jm1])
            - dt_dy * vn_c * (un_c - un[idx_im1])
            + nu_dt_dx2 * (un[idx_jp1] - 2.0 * un_c + un[idx_jm1])
            + nu_dt_dy2 * (un[idx_ip1] - 2.0 * un_c + un[idx_im1]);

        v[idx] = vn_c
            - dt_dx * un_c * (vn_c - vn[idx_jm1])
            - dt_dy * vn_c * (vn_c - vn[idx_im1])
            + nu_dt_dx2 * (vn[idx_jp1] - 2.0 * vn_c + vn[idx_jm1])
            + nu_dt_dy2 * (vn[idx_ip1] - 2.0 * vn_c + vn[idx_im1]);
    }
}
```

- Time to completion (in minutes), with increasing grid size (*nx* x *ny*)

# of GPUs	2048^2	4096^2	8192^2
2	**1.3**	**5.3**	78
4	2.3	9.5	**40**
6	3.3	13.8	52
8	4.4	17.4	67

When we compare without CUDA kernel vs. with CUDA kernel

- Time to completion (in minutes), with increasing grid size (*nx* x *ny*)

without

# of GPUs	2048^2	4096^2	8192^2
2	**8.6**	115.6	1552
4	15.3	**61.5**	895
6	22.5	88.8	606
8	29.8	118.1	**473**

with

# of GPUs	2048^2	4096^2	8192^2
2	**1.3**	**5.3**	78
4	2.3	9.5	**40**
6	3.3	13.8	52
8	4.4	17.4	67

- Ratio of speedup = Old Time / New Time

# of GPUs	2048^2	4096^2	8192^2
2	6.6x	21.91x	19.90x
4	6.6x	6.47x	22.38x
6	6.81x	6.43x	11.65x
8	6.77x	6.79x	7.06x

- Percent speedup = (Old Time – New Time) / New Time

# of GPUs	2048^2	4096^2	8192^2
2	562%	2081%	1889%
4	565%	547%	2138%
6	581%	543%	1065%
8	577%	578%	606%

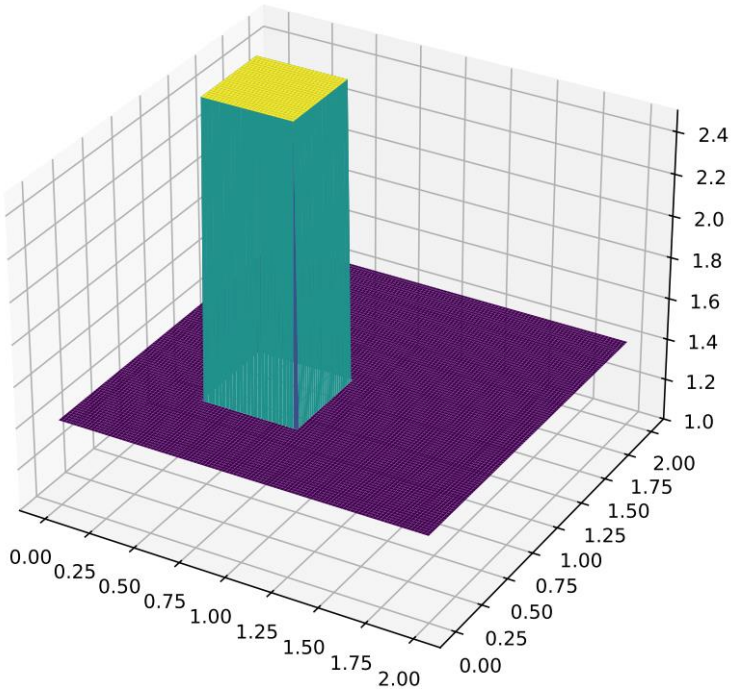
Maybe you're thinking…

- "Yeah, sure Python is able to run the simulation. So what?"
- "Yeah, sure Python can leverage GPU compute via CuPy"
- "And yeah, wow, okay cool, thanks to CuPy and all the hard work there, you can access lower-level CUDA API runtime and decrease your time to compute by offloading bottlenecks to the GPU. So what?"
- "BUT… this is not pro quality and to get real performance for high fidelity simulations, this HAS to be written in C/C++ or Fortran."

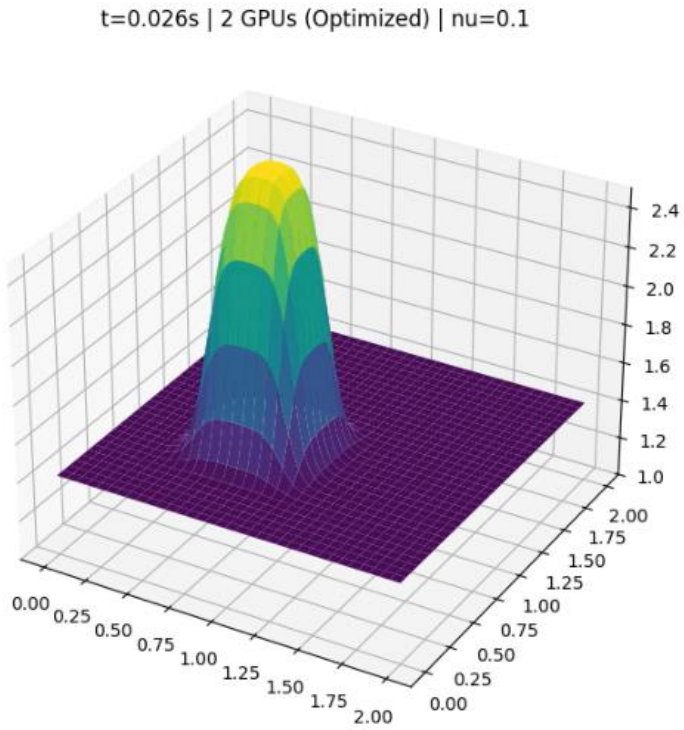
- Prompt-based development / AI-assisted engineering was crucial for me to mock up a port of my Python CuPy code to native C++. *Shoutout to Anthropic's Claude Opus 4.6*.
- My initial testing at home (on 1 GPU) showed similar compute times for Python+CuPy with CUDA fused kernel as the native C++ with CUDA calls.
  - I consider this groundbreaking and really exciting.

What do solutions to 2D Burgers' look like?

Initial condition:



burgers_220537.png



In summary

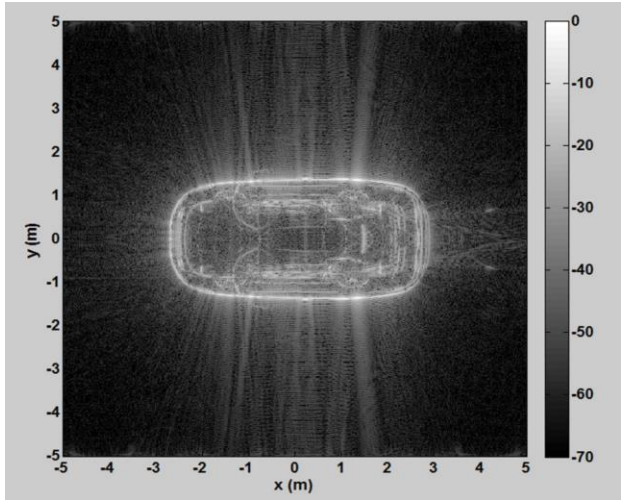
- **Traditional profiling can reveal performance bottlenecks, which can/should be resolved ASAP in a scientific project**
  - AI/LLMs can also assist in performing code review & analysis
- **Offload compute-intensive tasks from CPU to GPU kernels**
- **Kernel fusion reduces cross-talk and chatter even further**
  - Can be further optimized if you fine-tune grid and block dimensions

What I actually accomplished since last year…

- **Built custom device (CUDA) kernels: `cupy.RawKernel()` and in CUDA C**
- **I accelerated (my) physics!**
- Determined which problem- and domain-specific algorithms can be designed **from the beginning to use multi-GPU**
- **Took what I have learned and made it useful for my PhD program**

What remains to be done?

- Run the “optimized” Python+CuPy+CUDA kernel stack in conjunction with new, native C++ with CUDA runtime for more comparison + scrutiny
- Learn more CUDA C + CUDA Fortran
- Finish my proposal and write my dissertation
- Maybe after that, I can use GPUs to do *more* things with image processing for my radar job



Thanks to…

- Dr. Denis Aslangil
  - Project mentor and PhD advisor
- The Aerospace Corporation
  - Support in Graduate Student Fellowship program
- Leo Fang, Ash Vardanian, Ben van Werkhoven, Floris-Jan Willemsen, Stijn Heldens
- SCaLE 23x, CuPy team, and the whole open-source community!

THE UNIVERSITY OF ALABAMA® | College of Engineering

COLORADO SCHOOL OF MINES

References

1. **GPU-Accelerated Boussinesq Model Using Compute Unified Device Architecture FORTRAN** – *Journal of Coastal Research (Journal article)*

2. ***CUDA by Example: An Introduction to General-Purpose GPU Programming*** *(Authors: Sanders, Kandrot)*

3. ***CUDA Fortran for Scientists and Engineers*** *(Greg Ruetsch & Massimiliano Fatica)*

4. ***An Easy Introduction to CUDA C and C++****:* *https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/*

5. ***An Even Easier Introduction to CUDA****:* *https://developer.nvidia.com/blog/even-easier-introduction-cuda/*

6. ***CUDA C Programming Guide****: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ as PDF here*

7. **CUDA Fortran Programming Guide**, Release 26.1: https://docs.nvidia.com/hpc-sdk/pdf/hpc261cudaforug.pdf

8. **Simplifying GPU Application Development with Heterogeneous Memory Management** (HMM) - https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/
    1. For CUDA 12.2, nvidia-open, Linux kernel > v6.1

9. **Koushik Naskar** - https://github.com/Koushikphy/Intro-to-CUDA-Fortran

10. **GPU-Accelerated Computing with Python**: https://developer.nvidia.com/how-to-cuda-python

11. **Tridiagonal Matrix Solver via Thomas Algorithm:** https://www.quantstart.com/articles/Tridiagonal-Matrix-Solver-via-Thomas-Algorithm/

12. **Overview of NCCL**: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html

13. Leo Fang: **CuPy: Painlessly Accelerating Python Programs Using GPU** - https://leofang.github.io/assets/cupy.pdf

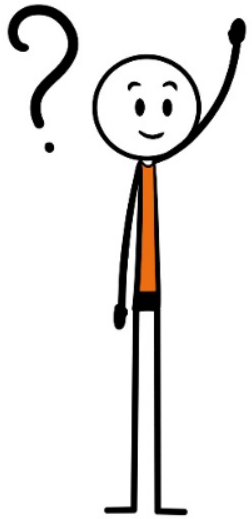
Questions?

Email:

Paul.Mekhedjian@gmail.com

Web:

https://paulmekhedjian.com



Thank You!

I really appreciate you joining me!