Introduction to GPU Computing

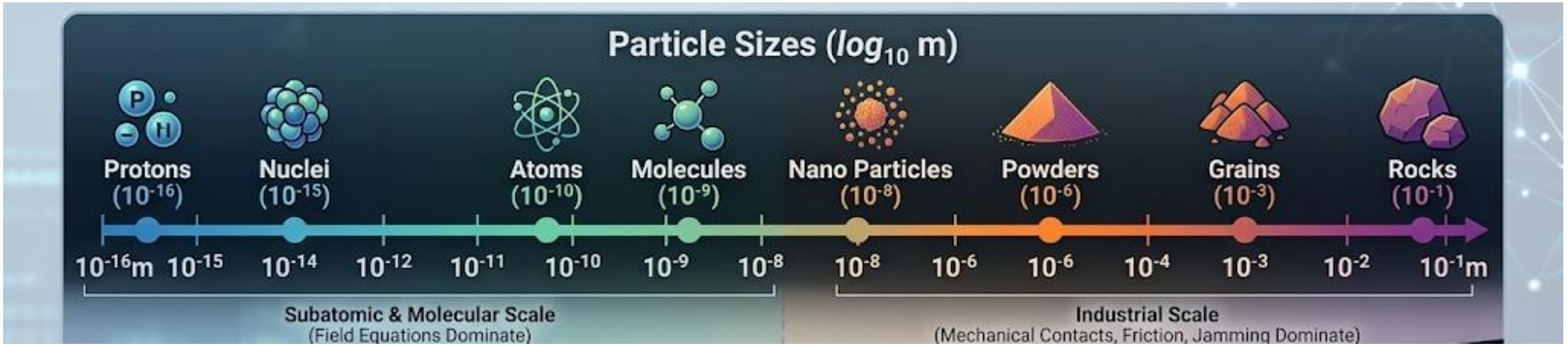
Belfast, 24 March 2026

Prof Nicolin Govender

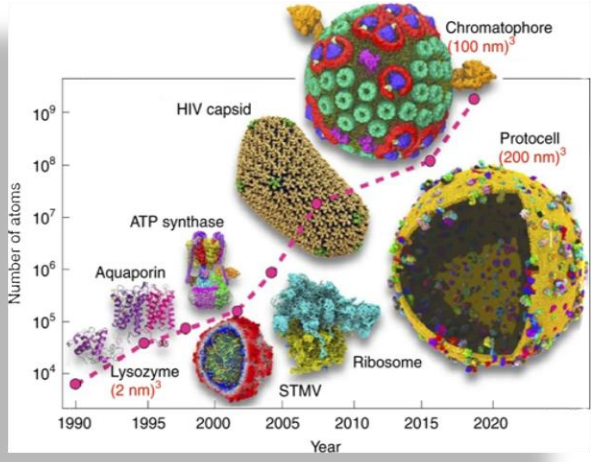
Advanced Research Computing
University College London

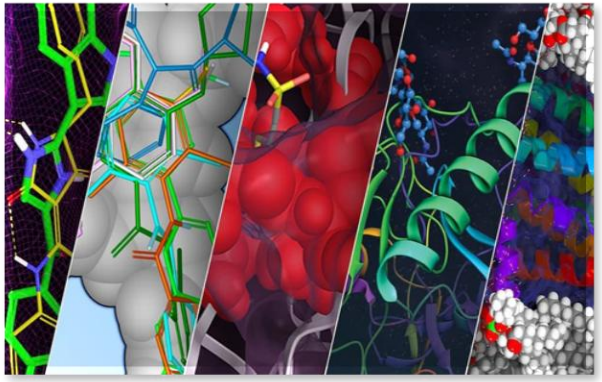
PI: Nvidia GPU research center
Nvidia Inception Program

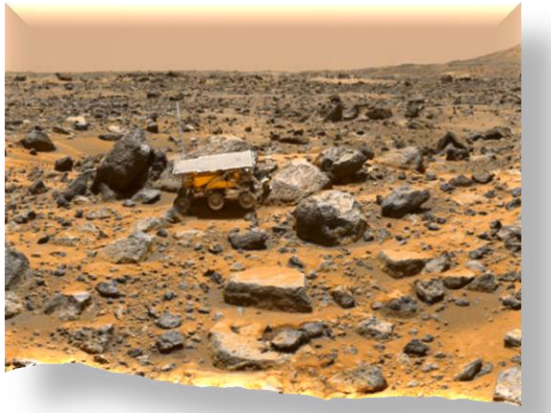
A Particle View of the World



For 70 years, we have mastered the extremes: peering down into the subatomic world of quarks and reaching out across the light-years to galaxies







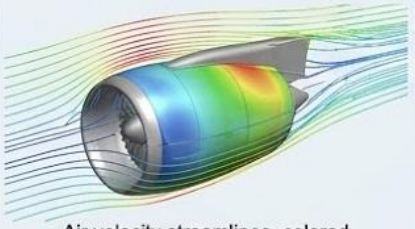
The World of CAE

(1951) (1956) (1979) (2020)

I. CONTINUUM-BASED MULTI-SCALE METHODS

CFD *(Computational Fluid Dynamics)*

Solves governing partial differential equations over structured/unstructured grids.

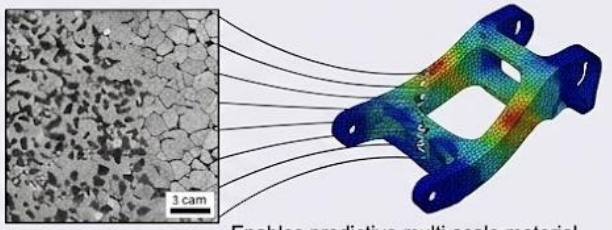


Air velocity streamlines, colored pressure field plot.

(1951)

FEM *(Finite Element Method)*

Discretizes complex geometries into **multi-scale finite elements,** from microscopic grain structure to macroscopic mechanical systems.

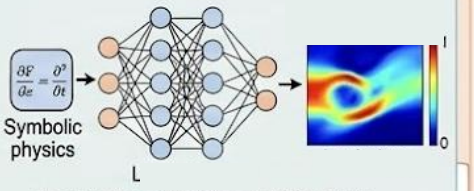


Enables predictive multi-scale material property modeling and mechanical response.

(1956)

PINN *(Physics-Informed Neural Networks)*

Leverages deep learning to **learn governing physical laws,** complementing traditional continuum solvers for complex physics.



Accelerates simulation and discovers hidden physical parameters.

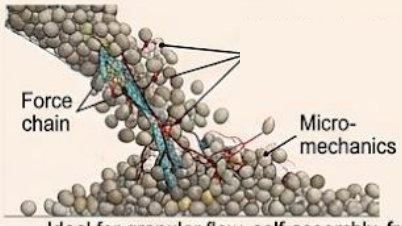
(2020)

Integrated Continuum Multi-Scale Approach: Treats material as a continuous field (micro to macro). Tracks field variables with increasingly sophisticated efficiency and fidelity.

II. DISCRETE-PARTICLE-BASED METHODS

DEM *(Discrete Element Method)*

Explicitly simulates individual discrete particle motion and interactions. Fundamental Lagrangian method.



Ideal for granular flow, self-assembly, fracture, and discontinuous material mechanics.

(1979)

Treats material as distinct, interacting particles. Tracks individual particle properties (position, velocity, force) and discontinuous interactions.

In recent years, several continuum methods such as MPM, SPH have been applied to granular materials using representative volume elements to account for discontinuous behavior.

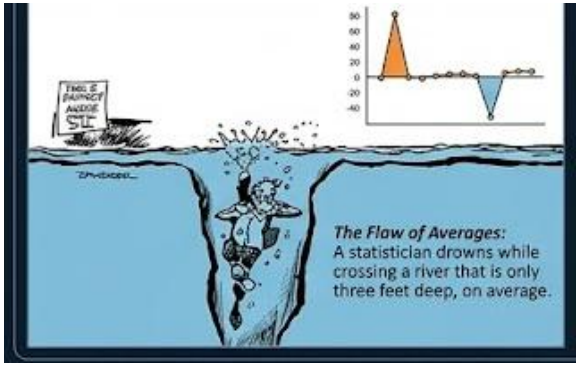
Limited applications of bulk flows have shown success where there is little internal stress differences.

Why: Industry

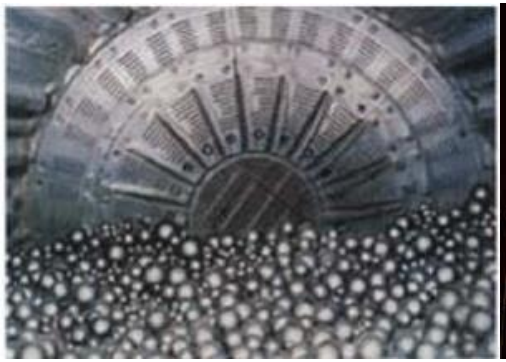
Over/under engineering devices results increased costs and reduced throughput.

Deal with issues when they happen

Over reliance on historical data



Industrial equipment is harsh so sensor information is limited



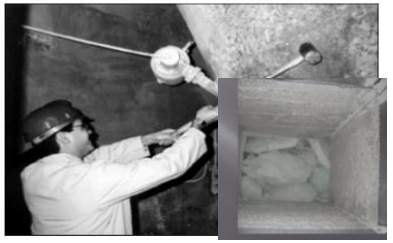
Granular Materials

Ubiquity of Granular Material
Nearly every industry processes granular material at some point.

Complex Behavior:
Granular material exhibits diverse behavior due to the vast range of sizes and shapes.

Lack of Universal Formulas:
The absence of general empirical formulae necessitates reliance on experiments or numerical modeling.

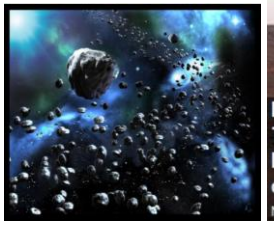
Unpredictability:
Material behavior can change with each batch, leading to variability in processing.



Fluid to Solid

Solid to Fluid

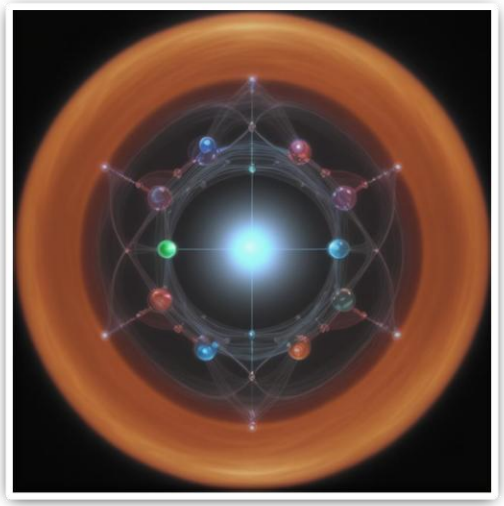
Significance:
Granular material is the second most manipulated substance on the planet after water.



Granular Material Challenge

User: Can you predict particle behavior

LLM: Here you go



User: Discrete particles

LLM: Sorry about that, here you go

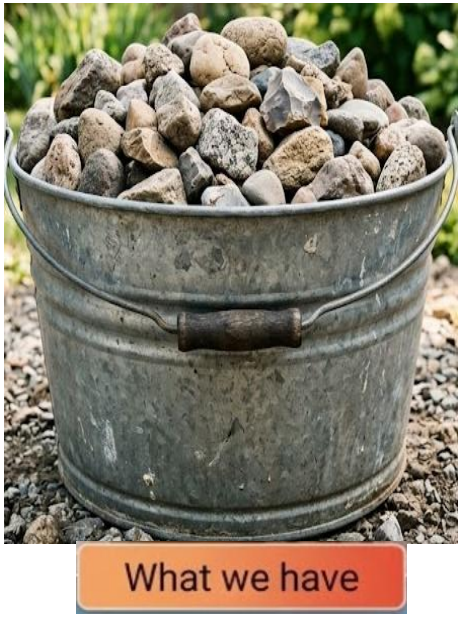
User: Particles like this image

LLM: Sorry, I have only been trained on
1.8 Trillion parameters

Now we know why

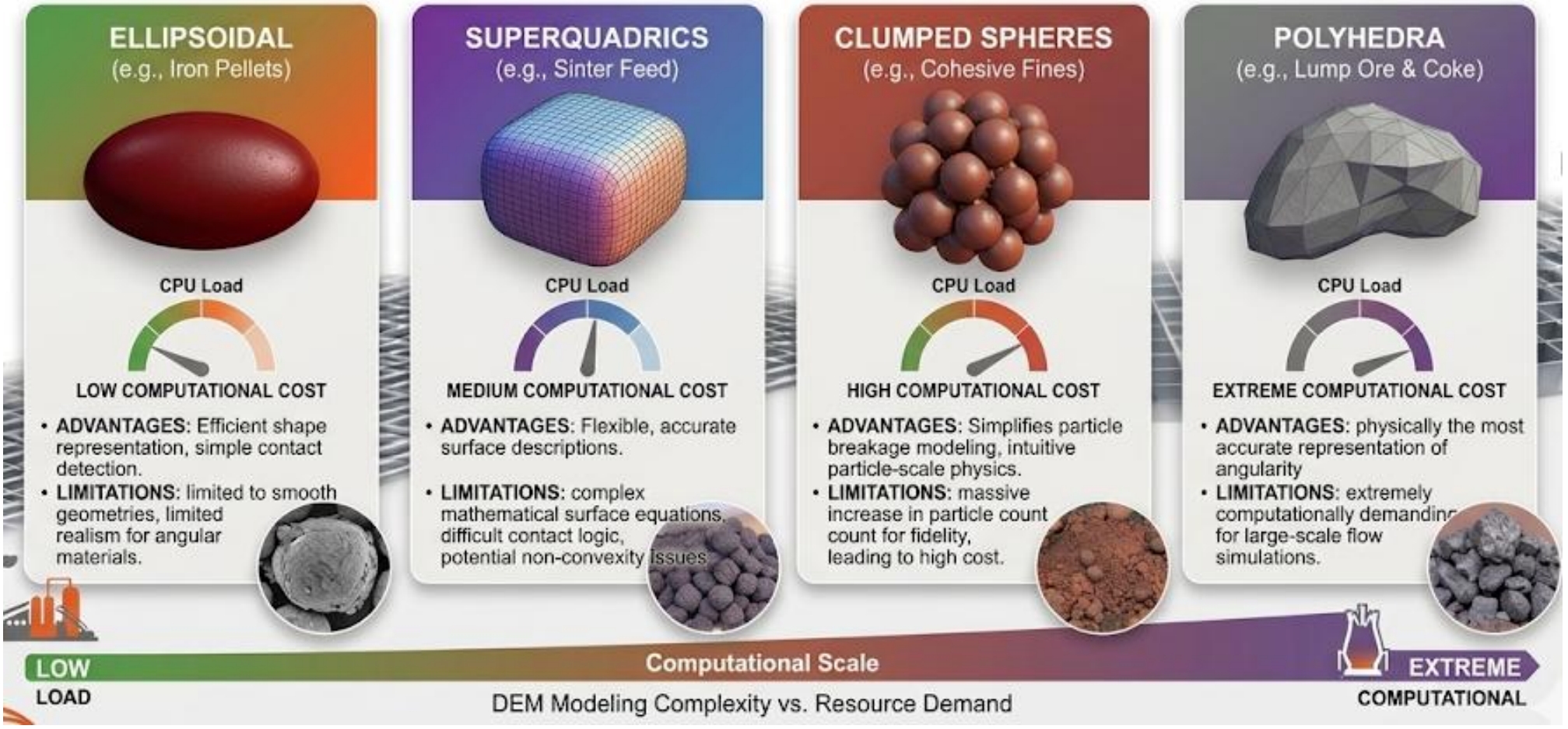
Discrete Challenge: Particle Number

Particle Number: *A full scale device contains millions to billions of particles*

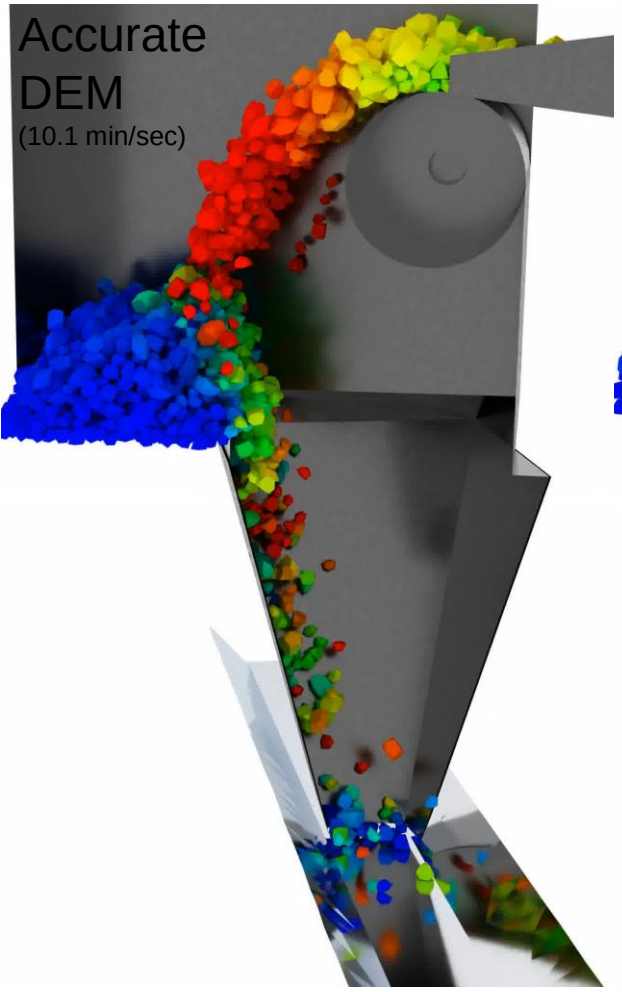


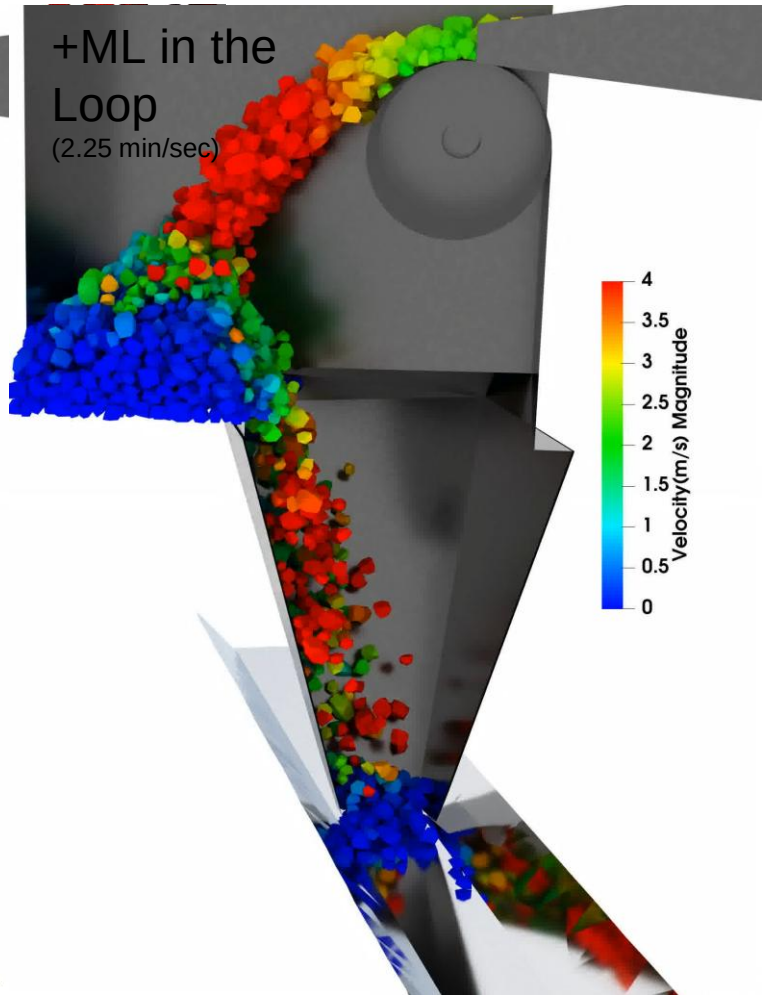
Discrete Challenge: Particle shape

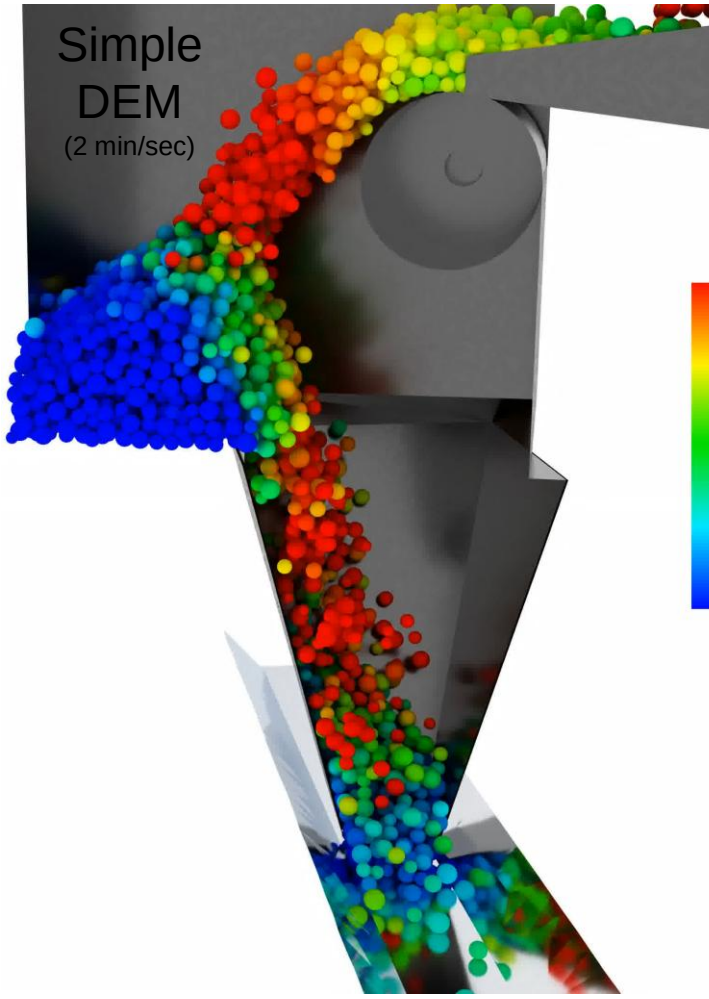
DEM Particulate Matter Challenge: Even when using a small number of particles, traditional CPU-based solvers struggle to balance complex particle shapes with finite computational resources. Multi-Scale understanding is critical for optimization

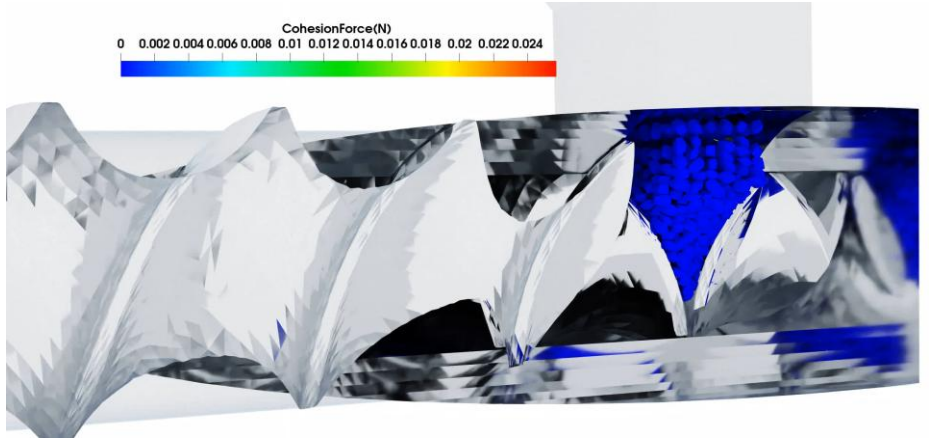


Why should you care about GPUs ?



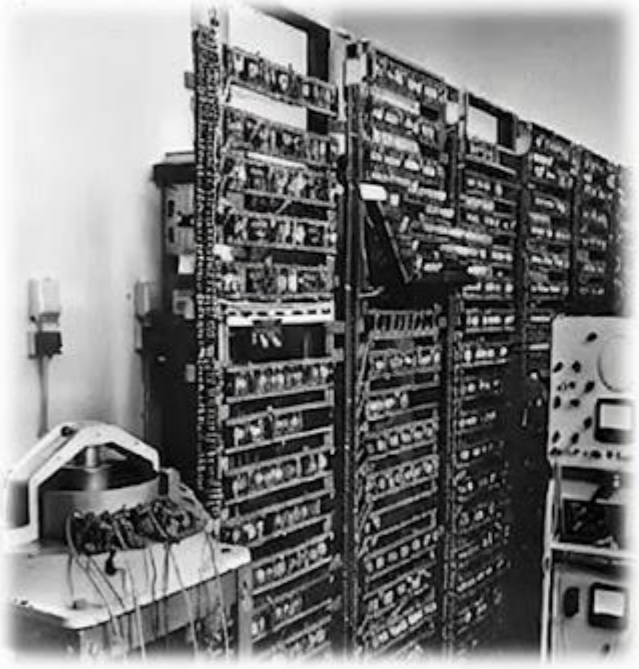






HISTORY

FROM CPU TO GPU: THE JOURNEY OF ACCELERATION



Vintage Mainframe: Era of the All-Purpose CPU

EARLY COMPUTING (Pre-1990s)

Logic

Processing

CPU as the Sole Processor

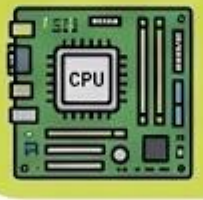
Computers required Central Processing Units (CPUs) to handle all complex logic and data processing tasks.

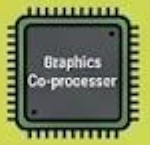
THE VISUAL REVOLUTION (Early 1990s)

GUI & Game Demands

Graphical User Interfaces (GUIs), detailed applications, and early 3D games dramatically increased the demand for sophisticated graphical processing.

SPECIALIZED CO-PROCESSORS (Mid-1990s)

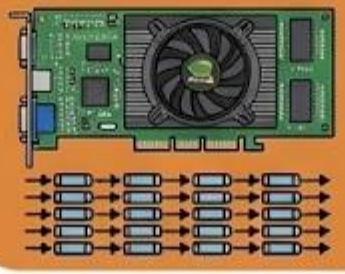




Dedicated Accelerators

Specialized co-processors were developed to **offload specific graphical tasks**, reducing CPU load and improving graphics performance.

THE BIRTH OF THE GPU (1999)

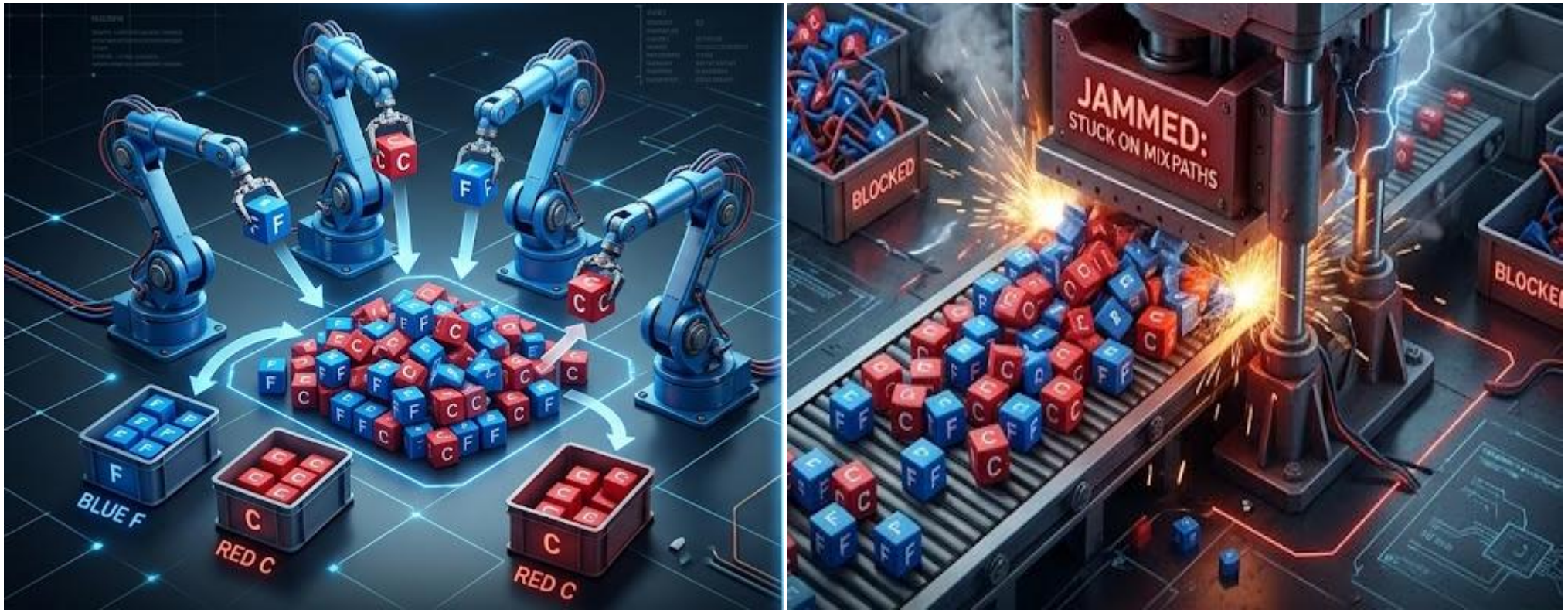


World's First Consumer GPU

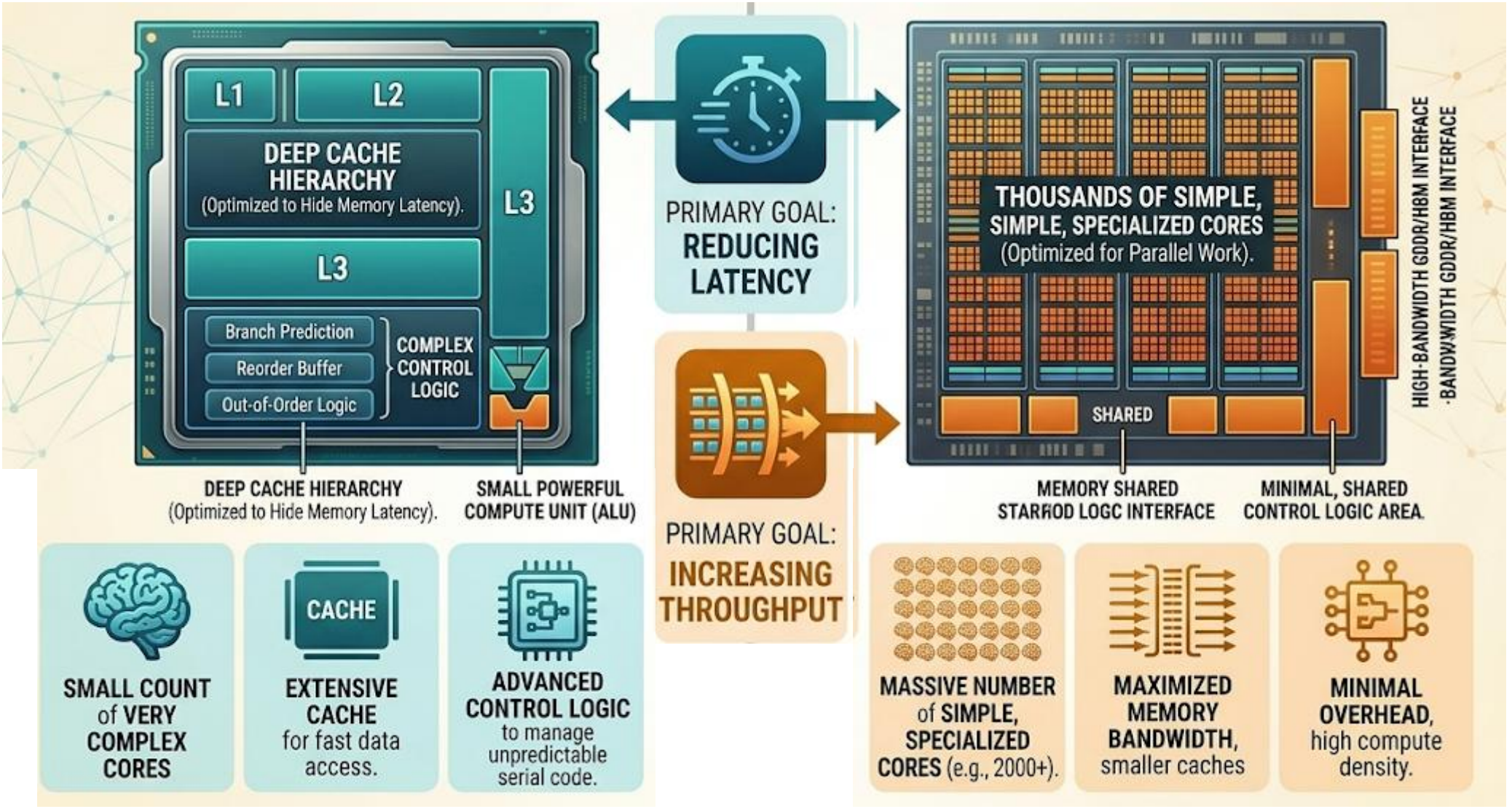
NVIDIA released the GeForce 256, the world's first consumer Graphics Processor Unit (GPU), designed for massive parallel graphical task execution.

Introducing key parallel processing capabilities.

Match the Chip

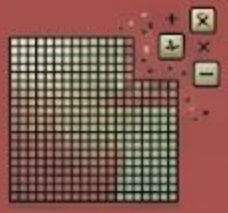


Chip Design

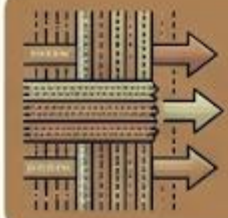


SUMMARY: CPU is a serial-code generalist, allocating most area to memory and control.
GPU is a parallel-math specialist, maximizing area for arithmetic. (Architecture is destiny.)

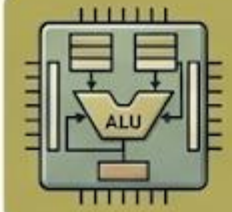
How They Work



PIXEL-LEVEL CALCULATIONS: Rendering images involves millions of simple arithmetic (±, ×) operations for EVERY pixel. Modern resolutions are demanding.



MASSIVE PARALLEL DEMAND: With millions of pixels needing SIMULTANEOUS updates, the processor must be massively parallel.



GPU HARDWARE OPTIMIZATION: To meet parallel math demand, the GPU architecture is composed primarily of thousands of simple ALUs.

CPU'S SEQUENTIAL DESIGN: In contrast, the CPU is a generalist designed for complex, sequential tasks (like running an OS). Its goal is single-task speed.

Think of it like this

THE CPU (LATENCY OPTIMIZED)

PURPOSE: Minimize time for a **SINGLE** task.

ANALOGY: The fast car moves a small group very quickly.
Max 4 people @ 120km/h, < 6 min trip.

TECHNICAL LINK: Handles complex, serial logic (e.g., OS operations, branching).

THE GPU (THROUGHPUT OPTIMIZED)

PURPOSE: Maximize work done per **UNIT OF TIME**.

ANALOGY: The large bus moves many people, even if slower.
Max 60 people @ 20km/h, 1 hour trip.

TECHNICAL LINK: Handles massive parallel operations (e.g., million-pixel display update).

CPUs **REDUCE LATENCY** for complex tasks.

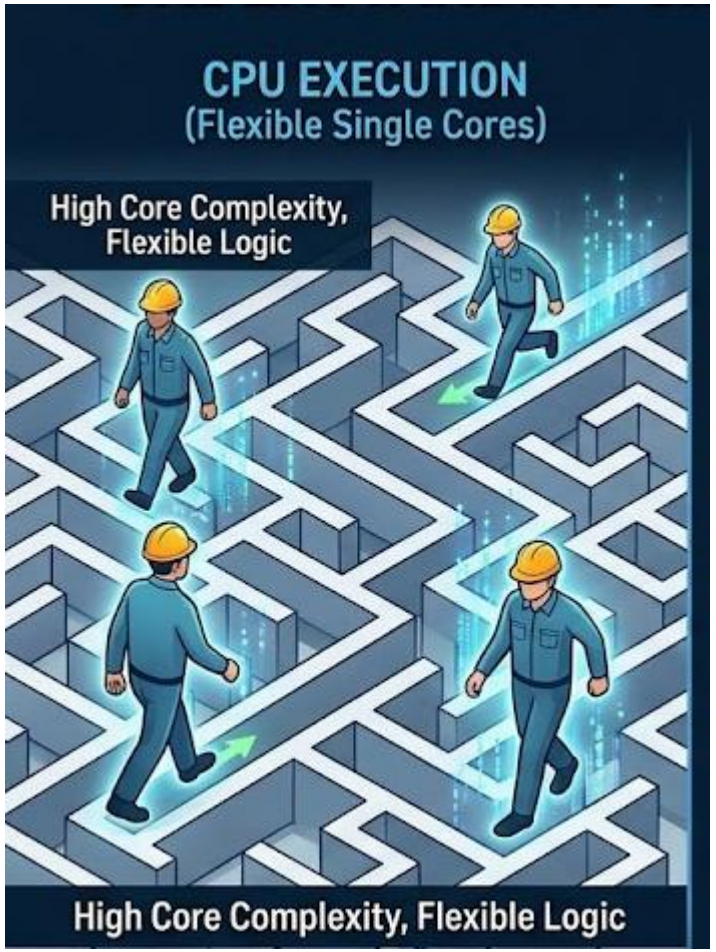
Car (40 people/hour) vs. Bus (60 people/hour)

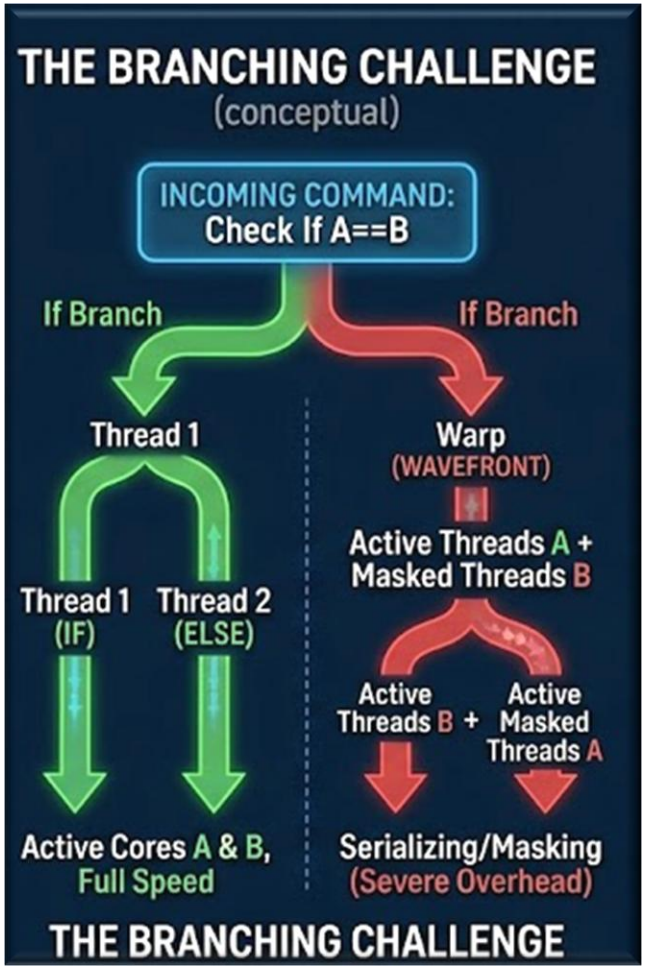
LATENCY VS THROUGHPUT

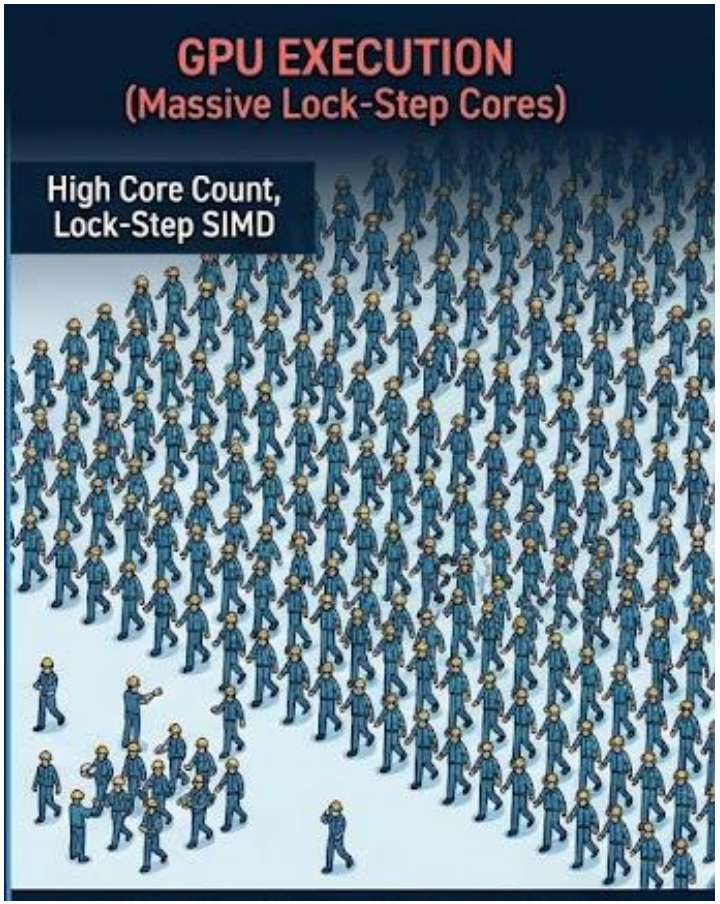
GPUs **INCREASE THROUGHPUT** for parallel tasks.

Car (40 people/hour) vs. Bus (60 people/hour)

What tasks are suited to the GPU

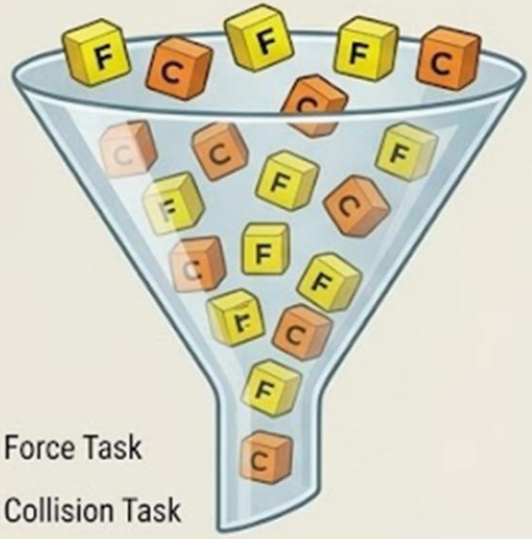






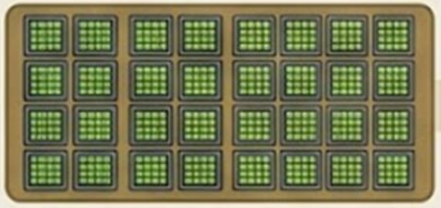
Lets think of it has a hopper with particles ;-)

SCENARIO 1: PROCESSING DIVERSE, MIXED TASKS (Complex Logic & Branching)



CPU COMPUTATION

Highly Efficient: Optimized for complex, serial branching logic. Quickly handles **diverse decisions** (e.g., AI).

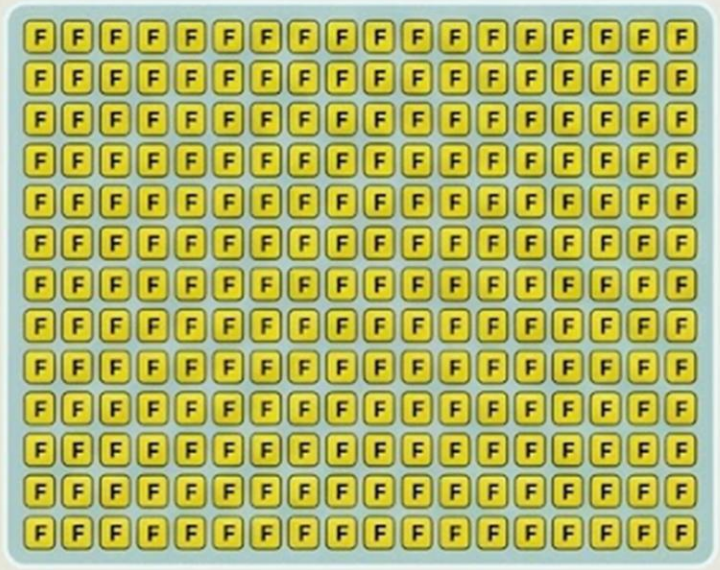


GPU COMPUTATION

Underutilized: Limited by branching overhead. Diverse tasks cause threads to wait, losing massive parallelism.

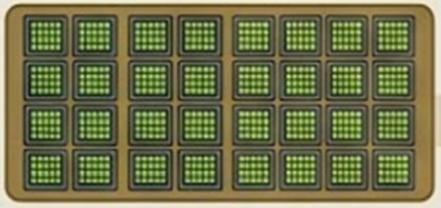
SCENARIO 2: PROCESSING MASSIVELY PARALLEL UNIFORM ARITHMETIC

(Identical Tasks & Data-Parallelism)

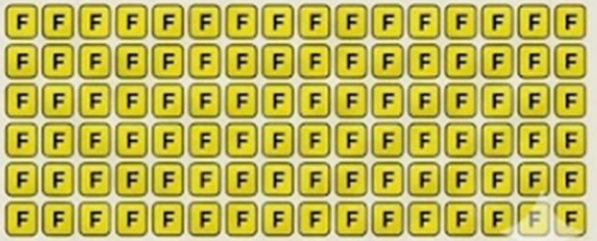


CPU COMPUTATION

Serial Bottleneck: A few powerful cores cannot process massive uniform task counts quickly.



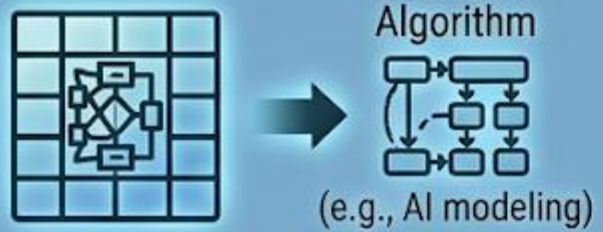
GPU COMPUTATION



Massive Throughput: Ideal for thousands of identical tasks (e.g., pixel rendering). Utilizes all available parallel power. Up to 2048 threads per core cluster (SM).

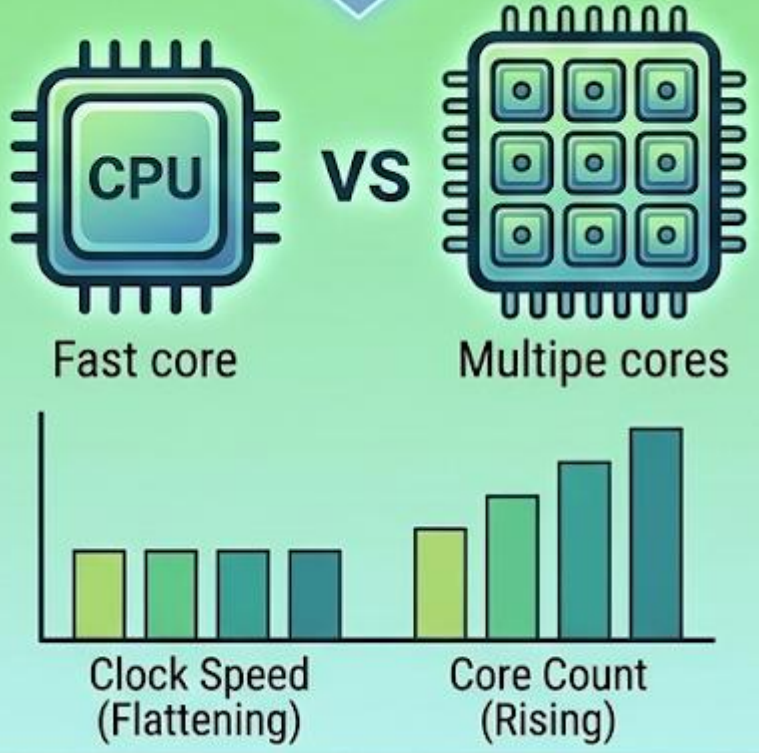
Why Parallel Computing

THE COMPUTE POWER SHIFT



1. THE ESCALATING WORKLOAD

Modern workloads are **compute-bound**. **Task complexity** continues to **grow**, demanding exponential computational power.

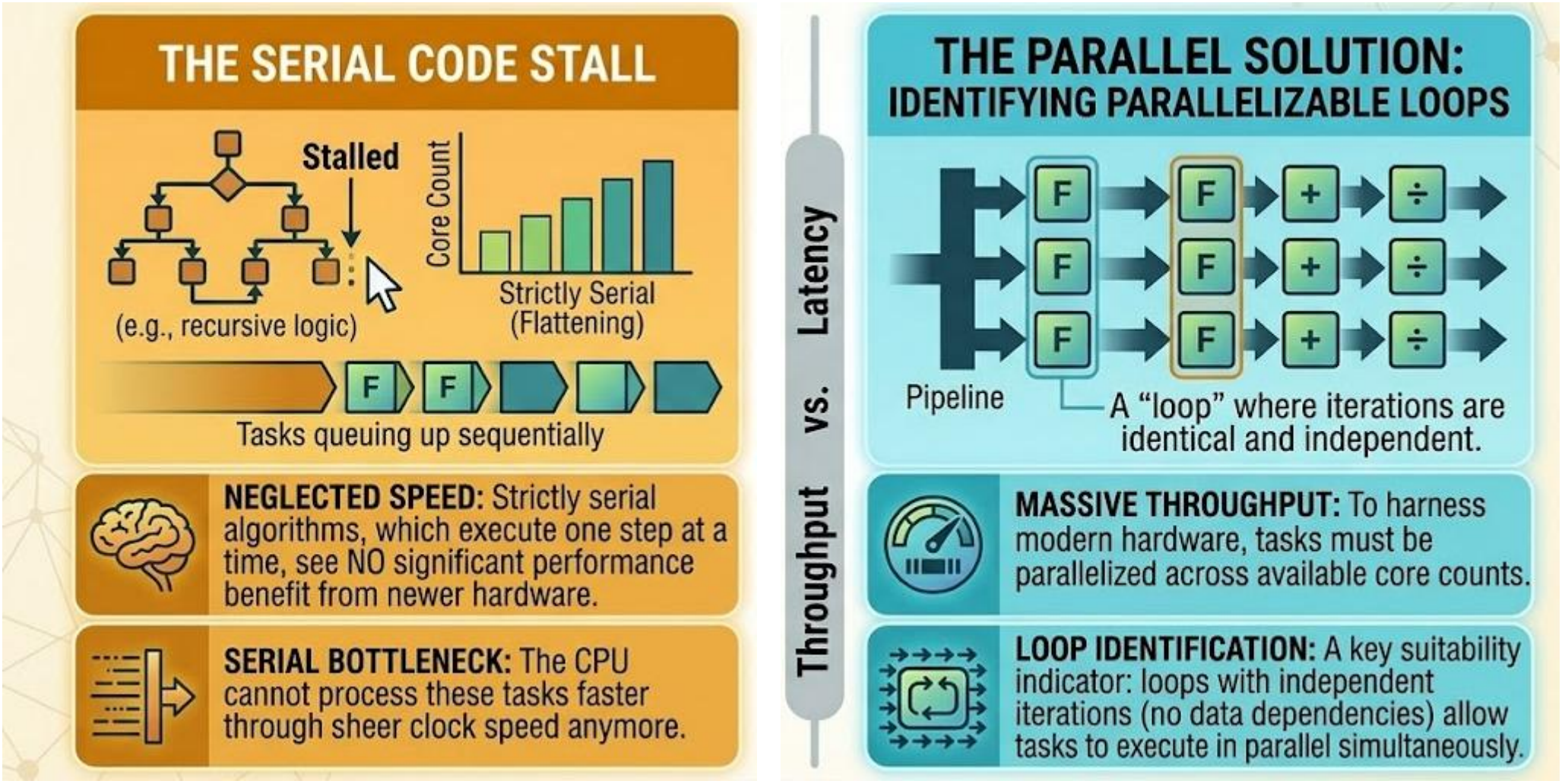


2. THE CLOCK SPEED LIMIT & SHIFT IN SCALING

CLOCK SPEED STALL: Traditional single-core scaling halted due to physical limitations of materials and physics.

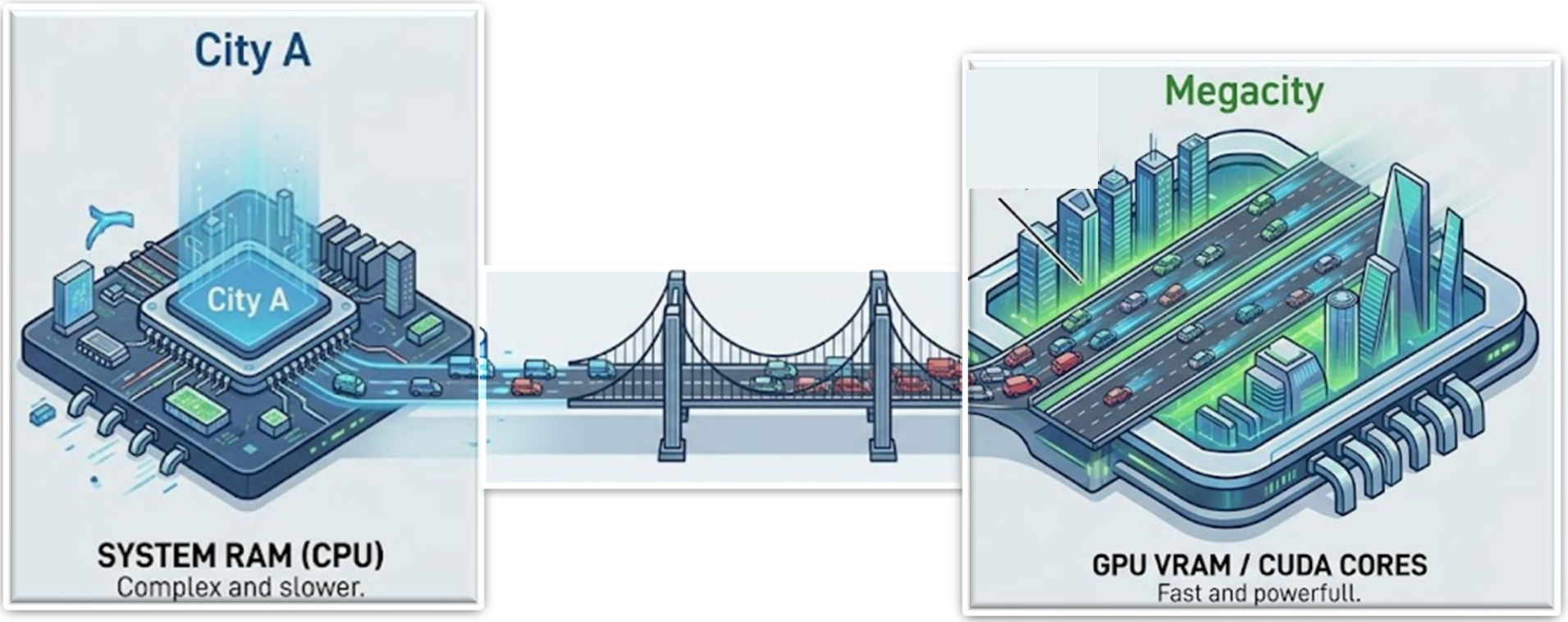
MULTICORE SCALING: Computational power (Moore's Law) now scales via **INCREASING CORE COUNT** per chip.

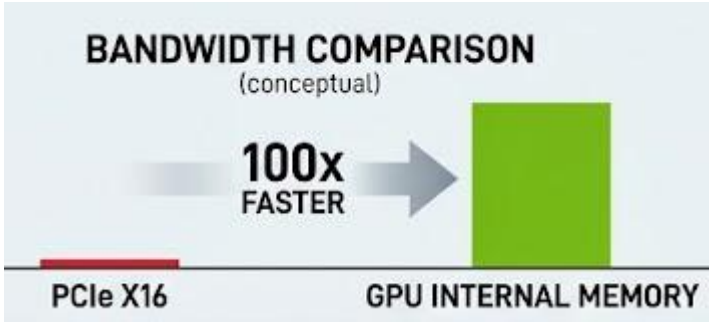
Consider our force and compute problem again



SUMMARY: Identify and refactor bottlenecks into parallel pipelines to achieve massive throughput on modern chips.

Heterogenous Computing





A simple example

Suppose the task at hand is to compute the sum of N numbers.

Canonical form: **float** sum = **0** for (; begin != end; ++begin) sum += *begin;

Not something that is suited to the parallel threaded nature of the GPU right ?

C++

```
std::vector<float> numbers(1024 * 1024 * 256);
std::fill(numers.begin(), numbers.end(), 1.0); auto
sum = std::accumulate(numers.begin(),
numbers.end(), 0.0);
```

OpenMP

```
float operator()(float const * begin, float
const * end) noexcept { float sum = 0;
size_t const n = end - begin; #pragma omp
parallel for default(shared) reduction(+ : sum) for
(size_t i = 0; i != n; i++) sum += begin[i];
return sum; }
```

AVX2

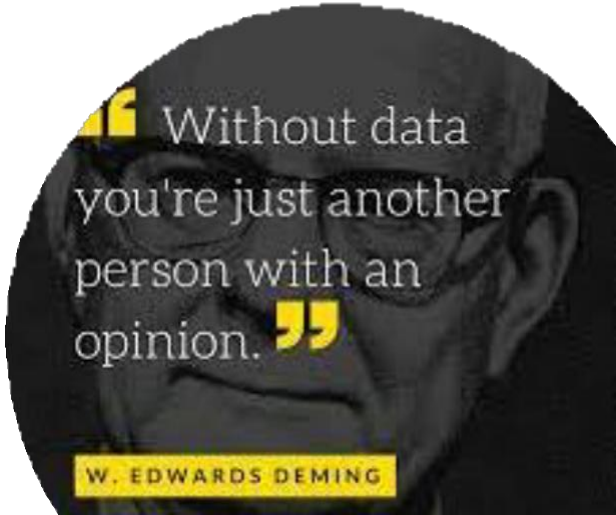
```
float operator()(float const * begin, float
const * end) noexcept { auto it = begin; //
SIMD-parallel summation stage auto sums =
_mm256_set1_ps(0); auto compensations =
_mm256_set1_ps(0); auto t =
_mm256_set1_ps(0); auto y =
_mm256_set1_ps(0); for (; it + 8 < end; it
+= 8) { y =
_mm256_sub_ps(_mm256_loadu_ps(it),
compensations); t = _mm256_add_ps(sums,
y); compensations =
_mm256_sub_ps(_mm256_sub_ps(t, sums),
y); sums = t; } // Cross-lane horizontal
reduction sums = _mm256_hadd_ps(sums,
_mm256_permute2f128_ps(sums, sums, 1));
sums = _mm256_hadd_ps(sums, sums);
sums = _mm256_hadd_ps(sums, sums); auto
sum = _mm256_cvtss_f32(sums); // Serial
summation of the remainder for (; it != end;
++it) sum += *it; return sum; }
```

CUDA

```
thrust::reduce(numers.begin(), numbers.end(),
float(0), thrust::plus<float>());
```

1. 5.40 GB/s
2. 10.9 GB/s
3. 80.0 GB/s
4. 743 GB/s

1. C++ std
2. SIMD-AVX2
3. OpenMP
4. CUDA

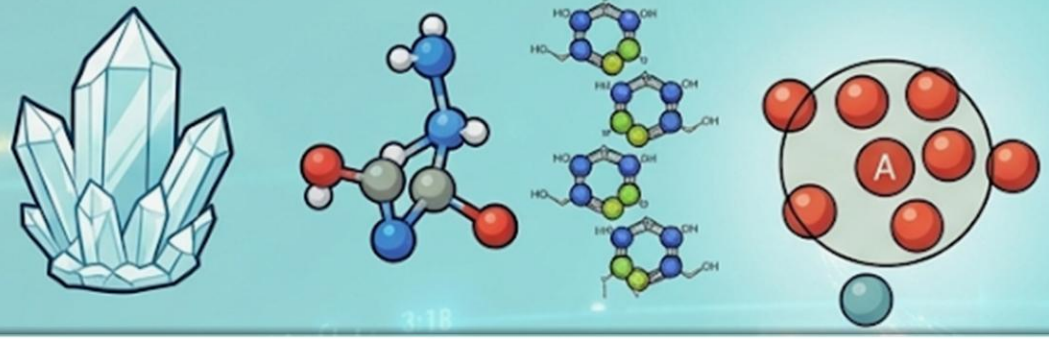


Discrete Numerical Methods

EVENT-BASED METHODS (e.g., Monte Carlo)

- **THROUGHPUT FOCUS**
  High volume of independent trials.
- **MASSIVE PARALLEL POTENTIAL**
- **PARALLEL DIVERGENCE: LOW**
  Minimal shared state; mostly consistent work per particle, despite random paths.
- Tally Count and Path-length focus.

PROXIMITY-BASED METHODS (e.g., Molecular Dynamics, SPH)



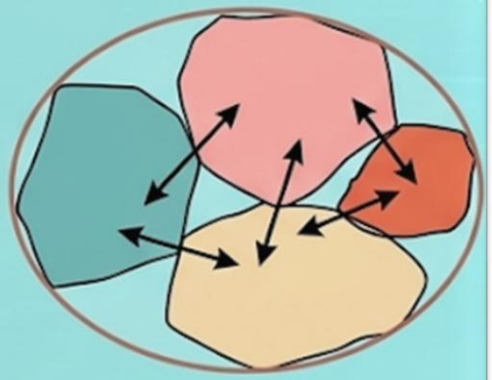
AT PARTICLE LEVEL:
Massively highly parallelizable (simultaneous molecular interactions).

LOW PARALLEL DIVERGENCE: Instruction complexity is very consistent across particles.

TASK SUITABILITY: IDENTICAL → Optimised for standard GPU paths

MASSIVE THROUGHPUT:

CONTACT-BASED METHODS (e.g., DEM, Impulse)



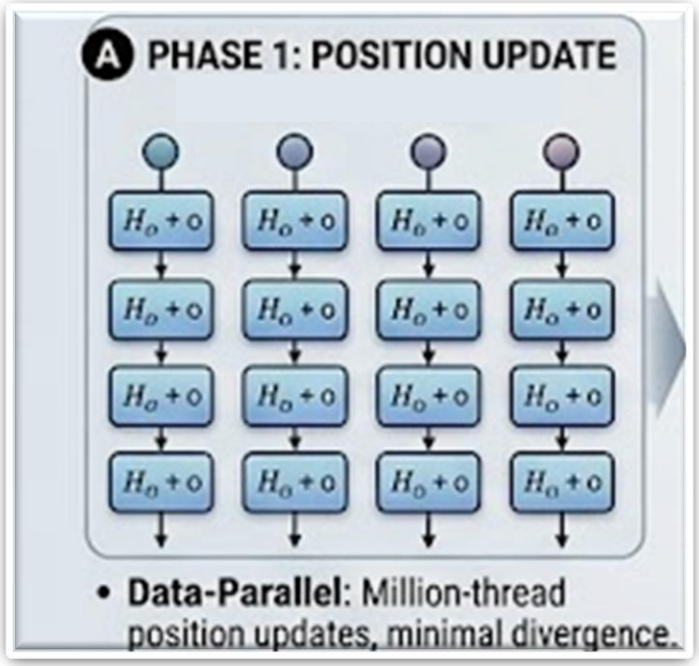
AT PARTICLE LEVEL: Large-scale potential, but highly complex computational load.

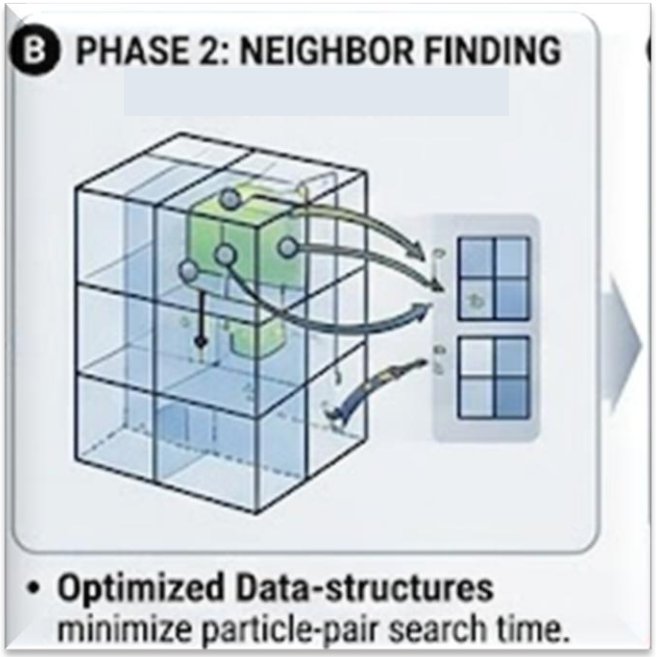
HIGH PARALLEL DIVERGENCE: Complex shape collision checks (geometry dependent) and varied contact states (sliding, rolling, static) create non-uniform branching logic.

DIVERSE TASK CHALLENGE:
Code execution paths vary significantly.

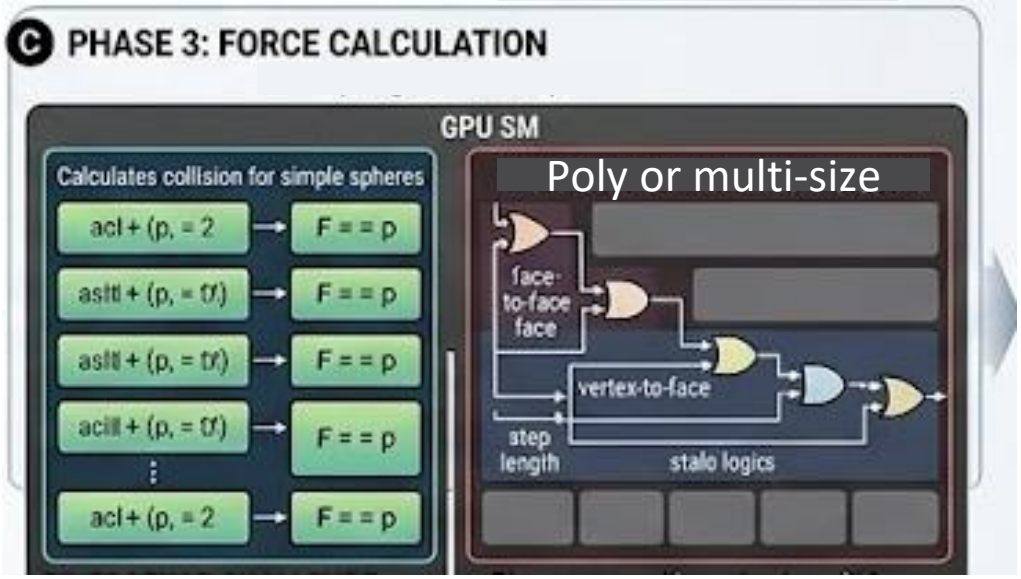
PARALLEL BOTTLENECK:

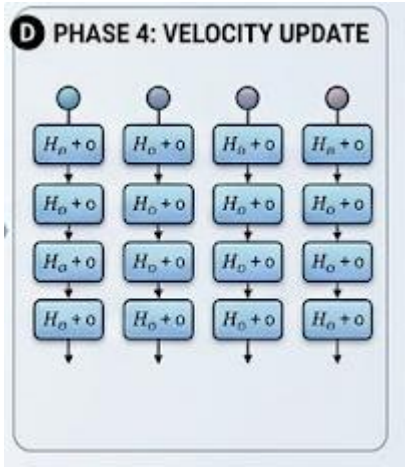
Why is DEM bad for GPUs?





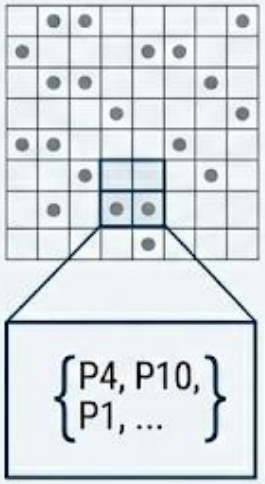
Proximity	Contact (DEM)
Parallelism	Parallelism
Divergence	Divergence





Bad on a GPU is still better than great on a CPU

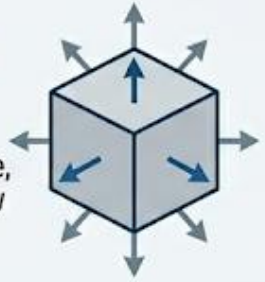
UNIFORM GRID PARTITIONING for Particles



1. Divide the simulation domain into a uniform grid of voxels.
2. Each particle is assigned to a single cell (the "bucket").
3. Neighbor search only checks particles within adjacent cells.

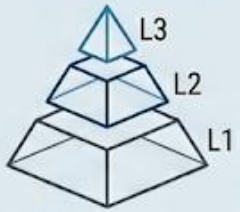
CRITICAL CHANGE:

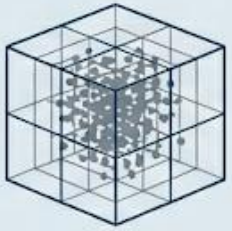
1. Requires access to all particles within a directional cell.
2. Requires linear storage, but lookup can be slow with large cell counts.

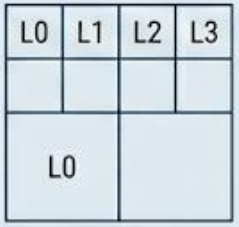


* Scales with Large Particle Counts

MULTI-GRID & ADAPTIVE SPATIAL PARTITIONING



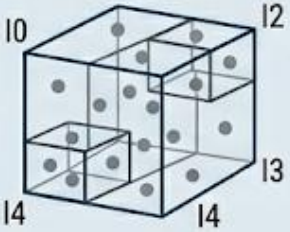




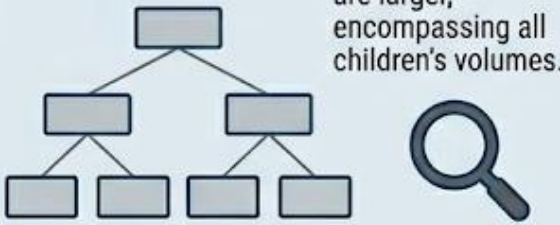
1. Hierarchical approach for handling varying density.
2. Start with a coarse grid (Level 0, L0) and subdivide for high density (L3).
3. Assign particles based on spatial location and resolution level.
4. Facilitates efficient long-range interactions (multigrid solver).

BOUNDING VOLUME HIERARCHY (BVH) for Particles

1. Sort particles along a space-filling curve (e.g., Morton-Z curve).
2. Build a Bounding Volume Hierarchy (BVH) over the sorted particles.

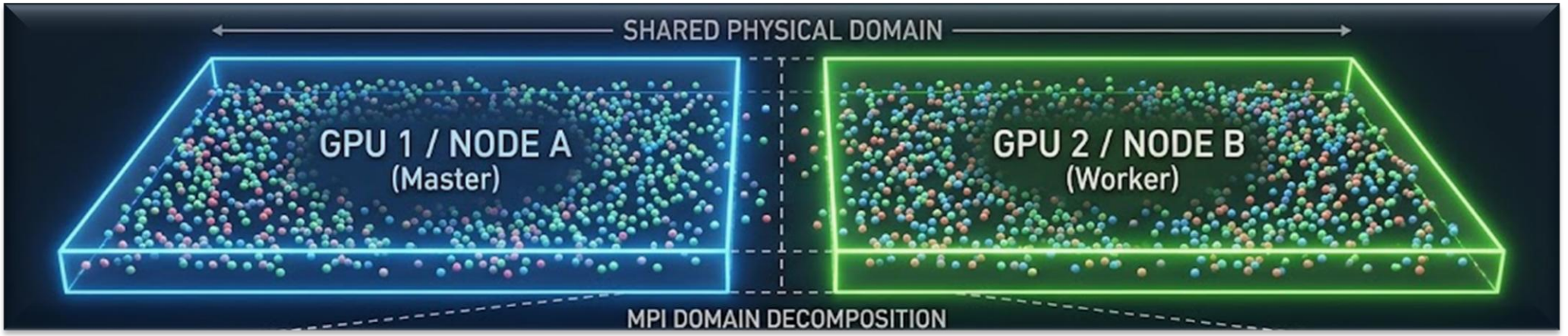


3. Leaf nodes contain small numbers of particles.
4. Intermediate nodes are larger, encompassing all children's volumes.



* Enables Fast Neighbor Searches

Multi Node Computing



Example 1: Making Loops Parallel on GPU

1. Let's compute the 3D grid positions for *posCOM* [*numParticles*].

```
for (uint indexP = 0; indexP < numParticles; indexP++)
{
  Real3 vecPos =  floor(posCOM[indexP ] - WOrigin)/cellSize);
  uint3 gridPos3D = make_uint3((uint)VecPos.x,(uint)VecPos.y,(uint)VecPos.z)
}
```

```
/* A] Parallel execution using std */
std::transform(std::execution::par, posCOM, posCOM +
numParticles, gridPos3D, [&](const Real3& pos)
{
Real3 vecPos = floor(pos - WOrigin) / cellSize;
return make_uint3(vecPos.x, vecPos.y, vecPos.z); }     );
```

```
/* B] Perform computation using OpenMP parallel for */
 #pragma omp parallel for
for (uint indexP = 0; indexP < numParticles; indexP++)
 {
   Real3 vecPos =  floor(posCOM[indexP ] - WOrigin)/cellSize);
  uint3 gridPos3D = make_uint3((uint)VecPos.x,(uint)VecPos.y,(uint)VecPos.z)
 }
```

```
/* C] Launch kernel using OpenMP target directive on GPU */
#pragma omp target teams distribute parallel for
for (uint indexP = 0; indexP < numParticles; indexP++)
{
  Real3 vecPos =  floor(posCOM[indexP ] - WOrigin)/cellSize);
 uint3 gridPos3D = make_uint3((uint)VecPos.x,(uint)VecPos.y,(uint)VecPos.z)
}
```

How can you code for GPUs ?

We write “Kernels”

```
__global__ void computeGridPosKernel(Real3* posCOM, uint3* gridPos3D, int numParticles, Real3 WOrigin,
Real cellSize)
{
// 1. Calculate the global thread ID (indexP)
    int indexP = blockIdx.x * blockDim.x + threadIdx.x;

    // 2. Ensure we don't read out of bounds
    if (indexP < numParticles) {

        // 3. Compute the grid position (Math from Slide 12)
        // Extract the particle's position
        Real3 pos = posCOM[indexP];

        // Perform the calculation per component
        Real3 vecPos;
        vecPos.x = floorf((pos.x - WOrigin.x) / cellSize);
        vecPos.y = floorf((pos.y - WOrigin.y) / cellSize);
        vecPos.z = floorf((pos.z - WOrigin.z) / cellSize);

        // Cast to unsigned int and store
        uint3 gridPos;
        gridPos.x = (unsigned int)vecPos.x;
        gridPos.y = (unsigned int)vecPos.y;
        gridPos.z = (unsigned int)vecPos.z;

        gridPos3D[indexP] = gridPos;
    }
}
```

```
// --- Set Device & Allocate ---
cudaSetDevice(0);
size_t bytes = numParticles * sizeof(Real3);
size_t gridBytes = numParticles * sizeof(uint3);

Real3 *d_posCOM, *h_posCOM;
uint3 *d_gridPos3D, *h_gridPos3D;

h_posCOM = (Real3*)malloc(bytes);
h_gridPos3D = (uint3*)malloc(gridBytes);

cudaMalloc((void**)&d_posCOM, bytes);
cudaMalloc((void**)&d_gridPos3D, gridBytes);
```

```
// --- Copy Data to GPU ---
cudaMemcpy(d_posCOM, h_posCOM, bytes, cudaMemcpyHostToDevice);
```

```
// --- Grid config & Launch ---
int threadsPerBlock = 256;
int blocksPerGrid = (numParticles + threadsPerBlock - 1) / threadsPerBlock;

computeGridPosKernel<<<blocksPerGrid, threadsPerBlock>>>(d_posCOM, d_gridPos3D, numParticles, WOrigin,
cellSize);

// Wait for GPU to finish
cudaDeviceSynchronize();
```

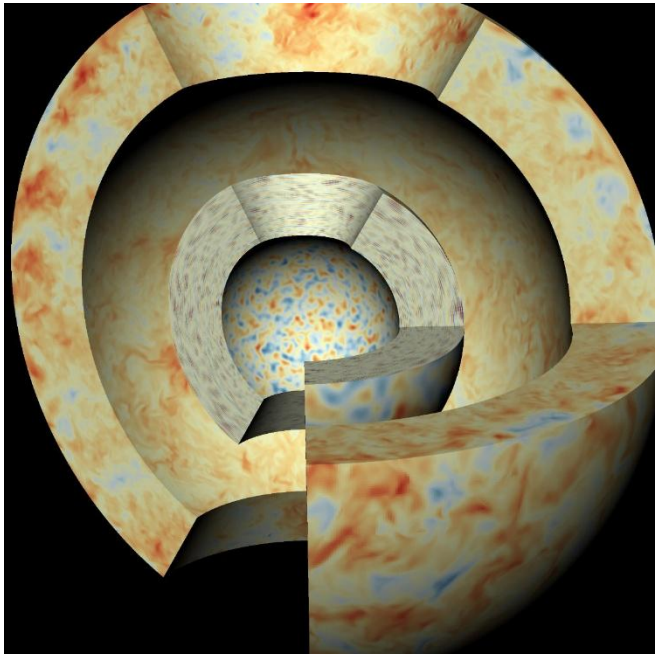
```
// --- Copy Results back to CPU ---
cudaMemcpy(h_gridPos3D, d_gridPos3D, gridBytes, cudaMemcpyDeviceToHost);

// --- Free Memory (Device & Host) ---
cudaFree(d_posCOM);
cudaFree(d_gridPos3D);

free(h_posCOM);
free(h_gridPos3D);
```

These days life is much easier

White Board Time

GPU Feasibility

Multidimensional Stellar Implicit Code (MUSIC)



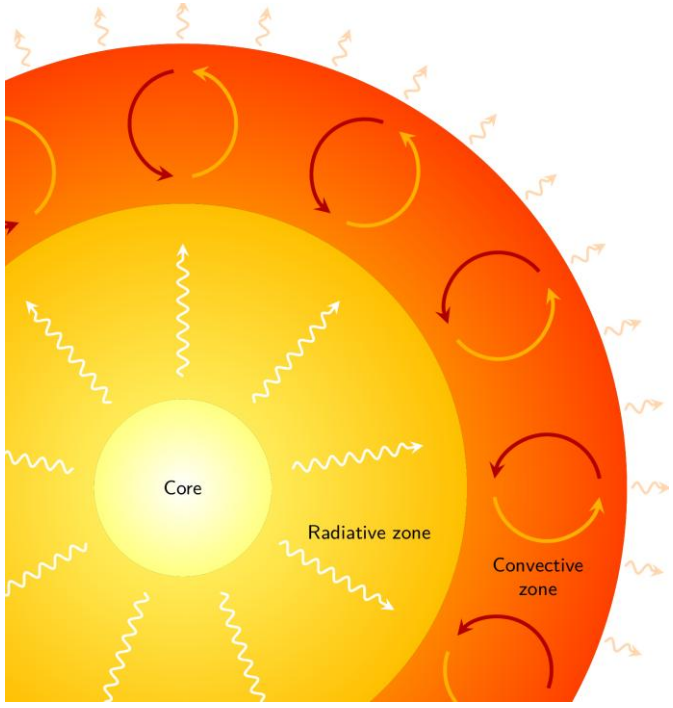
DiRAC

ExCALIBUR

UCL Advanced Research Computing

UKRI Science and Technology Facilities Council

The Physics: Understand the internal structure of stars



- **What is answered :**
  - Origin and amplification of magnetic fields
  - Angular momentum transport
  - Chemical mixing and Lithium depletion
  - Interplay of waves and convection

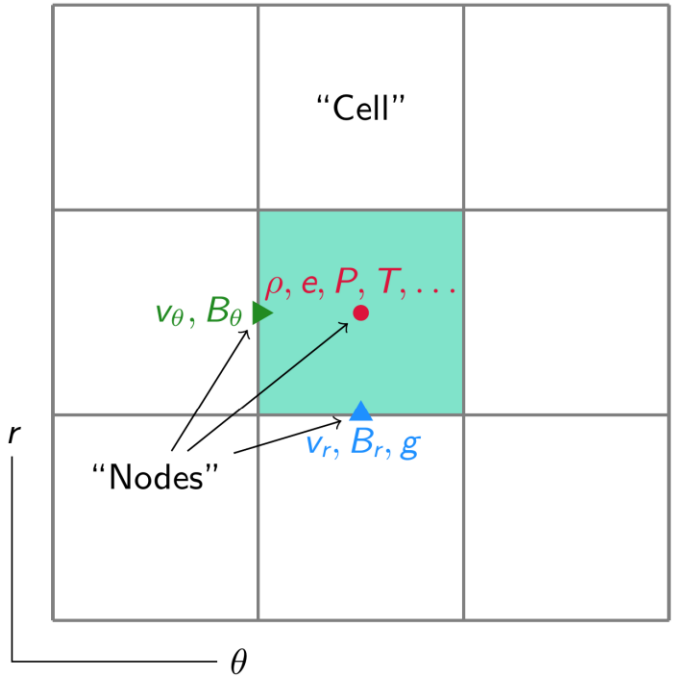
- **What needs to be simulated (DOFs)**
  - Convection & turbulence
  - Acoustic waves, internal waves
  - Tran
  - Mag

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{v}),$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} = -\nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v}) - \nabla p + \rho \mathbf{g},$$

$$\frac{\partial \rho e}{\partial t} = -\nabla \cdot (\rho e \mathbf{v}) - p \nabla \cdot \mathbf{v} + \nabla \cdot (\chi \nabla T) + Q_{\mathrm{nuc}},$$

The Physics: Compressible Hydro Dynamics



Staggered grid $\Rightarrow$ accurate buoyancy and $\vec{\nabla} \cdot \vec{v}$, constrained transport for MHD

Finite-volume scheme for hydrodynamics

$$\partial_t(\rho e) + \underbrace{\nabla \cdot (\rho e \boldsymbol{v})}_{\text{Transport term}} = \underbrace{-p\nabla \cdot \boldsymbol{v} + \nabla \cdot (\chi \nabla T) + \ldots}_{\text{Source terms}}$$

- Conservative upwinded transport term using van Leer or WENO reconstruction
- Source terms evaluated directly using finite volume

Resulting semi-discrete scheme:

$\partial_t \boldsymbol{X} = \boldsymbol{R}[\boldsymbol{X}, t]$, $\boldsymbol{R}$ is the nonlinear "RHS"

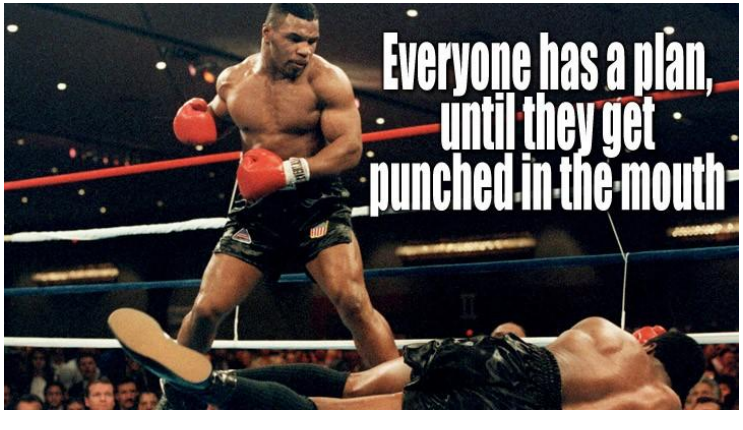
The initial plan

- Understand the code from the maintainers point of view.
- Get a suitable test case from the maintainers.
- Verify the assumed bottleneck in the code via profiling.

- Determine key issues for GPU offloading using further profiling.

  These issues could be
  - Lots of data transforms
  - Frequent communications between MPI ranks
  - Many operations over small blocks of memory

- Assess our approach and draft a template process for GPU feasibility assessment



The Code

Multidimensional **S**tellar **I**mplicit **C**ode, simulates hydrodynamics of stellar interiors.

- Written in Object Oriented Fortran for the time propagation solution with the spatial solution of the sparse matrix system done via the C++ backend called Trilinos.

- Precondition (Equations) in Fortran CG in an iterative manner with the Device using GMRES .

- The preconditioned sparse matrix is then transferred to the Device
  - The system is solved using TPetra for the physical quantities of interest.
  - These matrices/vectors are then transferred back to the CPU to have the physics evaluated and the residual calculated in a predictor corrector step.

The Code

Jacobian-free Newton–Krylov (JFNK) Method

Knoll & Keyes 2004, Park+2009; for MUSIC: Viallet+2016; Goffrey+2017

Time step loop

Advance in time $t^n \to t^{n+1}$ by solving implicit nonlinear problem $\boldsymbol{F}[\boldsymbol{X}] = 0$ for next state $\boldsymbol{X}$

Newton-Raphson iterations for $\boldsymbol{F}[\boldsymbol{X}] = 0$ from initial guess for $\boldsymbol{X}$

- Solve linear system $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}} \cdot \delta \boldsymbol{X} = -\boldsymbol{F}$ for correction $\delta \boldsymbol{X}$, and correct $\boldsymbol{X} \leftarrow \boldsymbol{X} + \delta \boldsymbol{X}$
- $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}}$ is a *huge* matrix, but we can avoid forming it explicitly

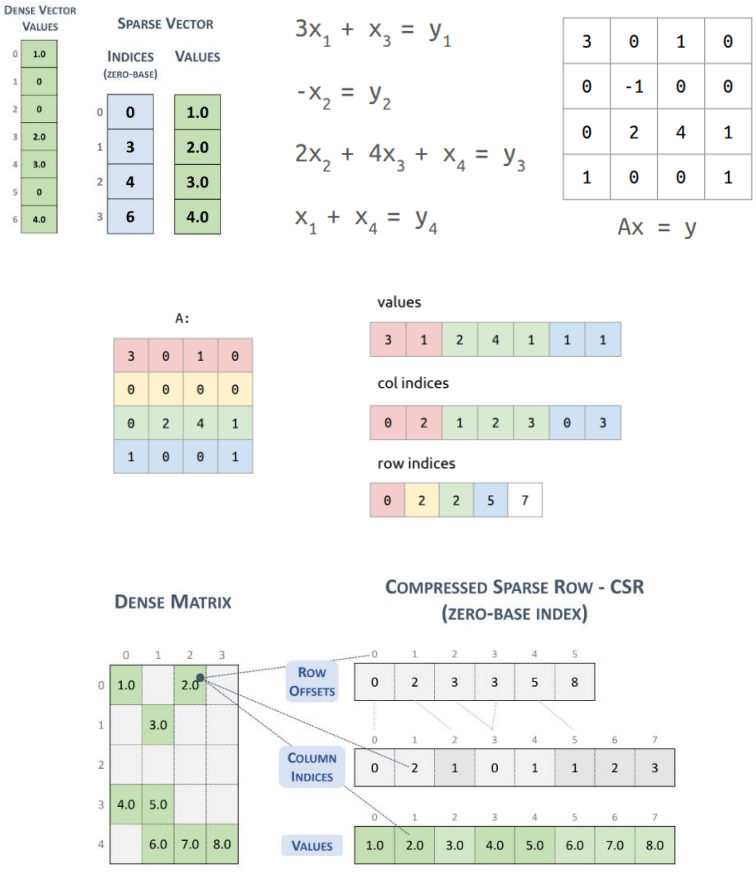
Krylov (GMRES) iterations for $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}} \cdot \delta \boldsymbol{X} = -\boldsymbol{F}$

- Krylov methods use repeated application of linear operator (here $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}}$)
- Key JFNK idea: approximate: $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}} \cdot \boldsymbol{V} \approx \frac{1}{\epsilon} \left(\boldsymbol{F}[\boldsymbol{X} + \epsilon \boldsymbol{V}] - \boldsymbol{F}[\boldsymbol{X}] \right)$
- Krylov methods are very sensitive to conditioning

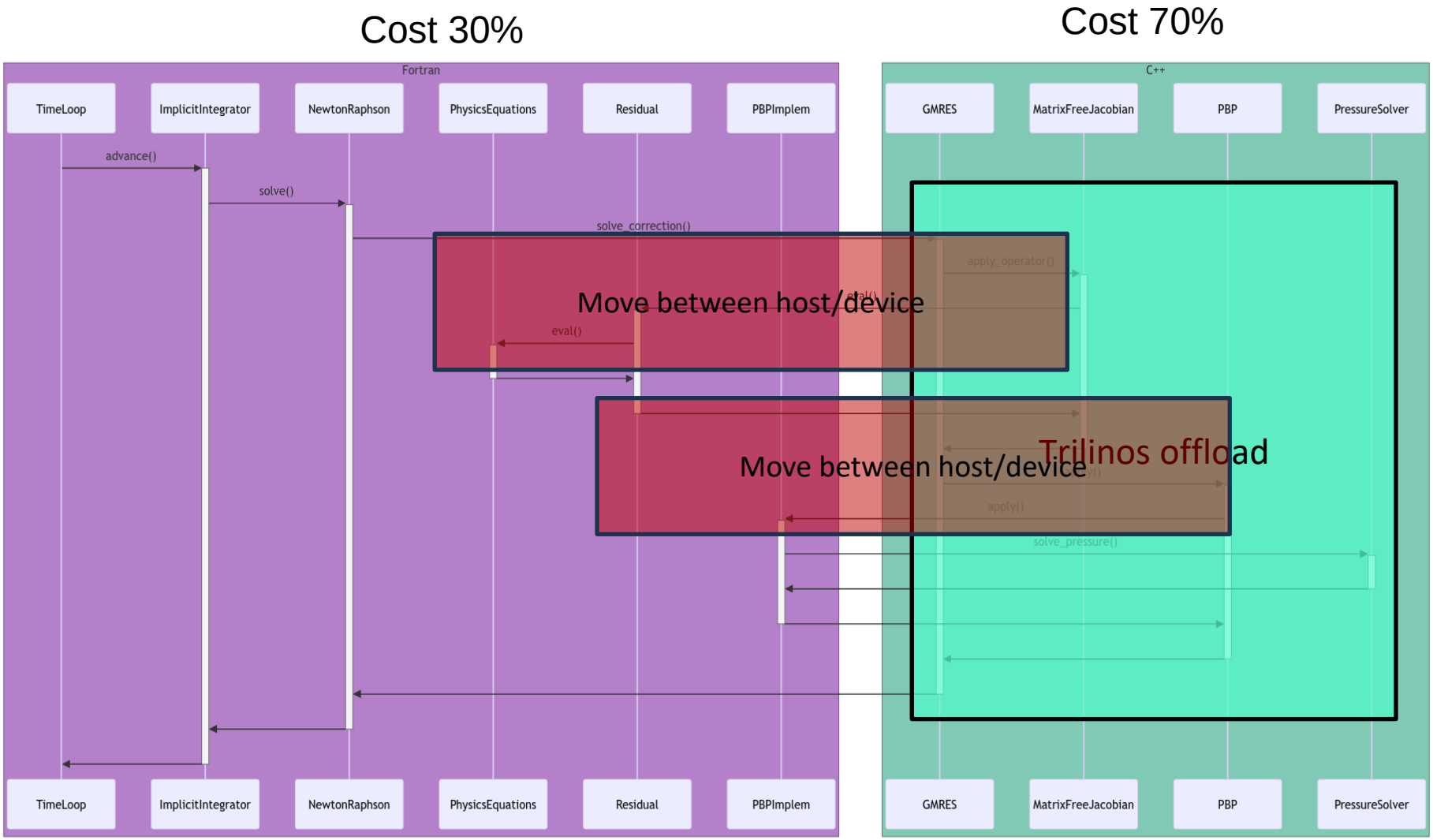
Physics-based preconditioner: semi-implicit scheme [Park+2009]

$\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}}$ contains "slow" (advective) and "fast" (acoustic) processes

- Advection: explicit formulation
- Acoustic waves: linearization + implicit formulation
  Pressure equation (elliptic): GMRES, with AMG preconditioner



The Architecture



Backend comparison:

13th Gen Intel® Core™ i7-13700F
NVIDIA RTX A6000

1 MPI, Single Node

PERF: Time loop.....: 3.203E+02 [100.0 %]
PERF: IO.....: 4.845E-03 [0.0 %]
PERF: Particles.....: 0.000E+00 [0.0 %]
PERF: Ghostcells.....: 3.390E+00 [1.1 %]
PERF: JFNK.....: 3.171E+02 [99.0 %]
PERF: GMRES.....: 2.685E+02 [83.8 %]
PERF: Residual.....: 8.285E+01 [25.9 %]
PERF: Gravity .....: 1.444E-03 [0.0 %]
PERF: PBP.....: 1.360E+02 [42.5 %]
PERF: PBP GMRES...: 1.166E+02 [36.4 %]
PERF: Diagnostics.....: 1.863E-01 [0.1 %]
DIAGNOSTICS: HYDRO: time= 2.50000000E-03, Eint= 2.50160270E+09, Ekin= 1.26248360E+03, Emag= 2.55946271E-11
DIAGNOSTICS: MHD: time= 2.50000000E-03, divB_L2= 5.86671691E-19, divB_Linf= 7.36579851E-18, Emag= 2.55946271E-11

1 MPI, 4 OMP, Single Node

PERF: Time loop.....: 2.991E+02 [100.0 %]
PERF: IO.....: 5.752E-03 [0.0 %]
PERF: Particles.....: 0.000E+00 [0.0 %]
PERF: Ghostcells.....: 3.422E+00 [1.1 %]
PERF: JFNK.....: 2.959E+02 [98.9 %]
PERF: GMRES.....: 2.482E+02 [83.0 %]
PERF: Residual.....: 8.297E+01 [27.7 %]
PERF: Gravity .....: 1.241E-03 [0.0 %]
PERF: PBP.....: 1.176E+02 [39.3 %]
PERF: PBP GMRES...: 9.731E+01 [32.5 %]
PERF: Diagnostics.....: 1.817E-01 [0.1 %]
DIAGNOSTICS: HYDRO: time= 2.50000000E-03, Eint= 2.50160270E+09, Ekin= 1.26248360E+03, Emag= 2.55946271E-11
DIAGNOSTICS: MHD: time= 2.50000000E-03, divB_L2= 5.86975020E-19, divB_Linf= 7.47083059E-18, Emag= 2.55946271E-11

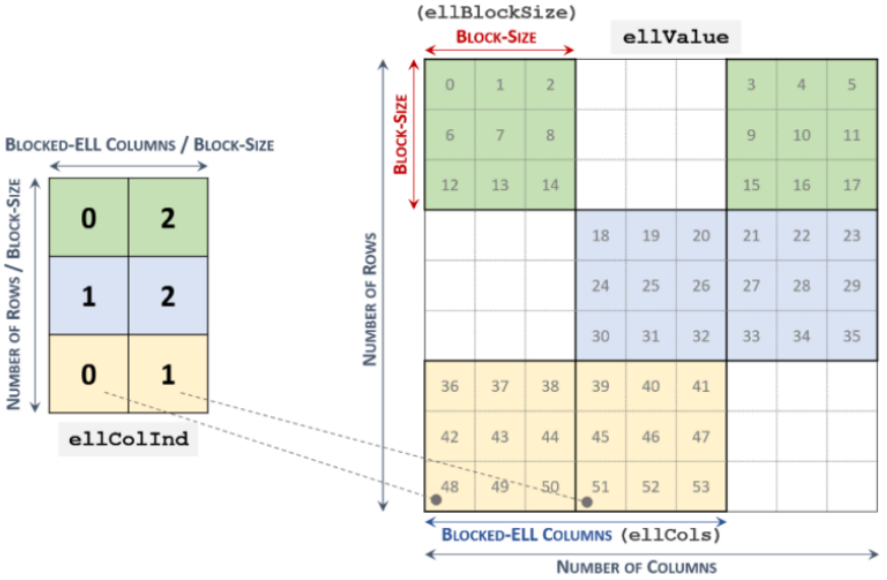
1 MPI, 1 GPU, Single Prec, Single Node

PERF: Time loop.....: 2.104E+02 [100.0 %]
PERF: IO.....: 4.639E-03 [0.0 %]
PERF: Particles.....: 0.000E+00 [0.0 %]
PERF: Ghostcells.....: 3.368E+00 [1.6 %]
PERF: JFNK.....: 2.072E+02 [98.5 %]
PERF: GMRES.....: 1.417E+02 [67.4 %]
PERF: Residual.....: 8.349E+01 [39.7 %]
PERF: Gravity .....: 1.394E-03 [0.0 %]
PERF: PBP.....: 4.442E+01 [21.1 %]
PERF: PBP GMRES.....: 1.415E+01 [6.7 %]
PERF: Diagnostics.....: 1.770E-01 [0.1 %]
DIAGNOSTICS: HYDRO: time= 2.50000000E-03, Eint= 2.50160270E+09, Ekin= 1.26248360E+03, Emag= 2.55946271E-11
DIAGNOSTICS: MHD: time= 2.50000000E-03, divB_L2= 5.86615548E-19, divB_Linf= 7.53520510E-18, Emag= 2.55946271E-11

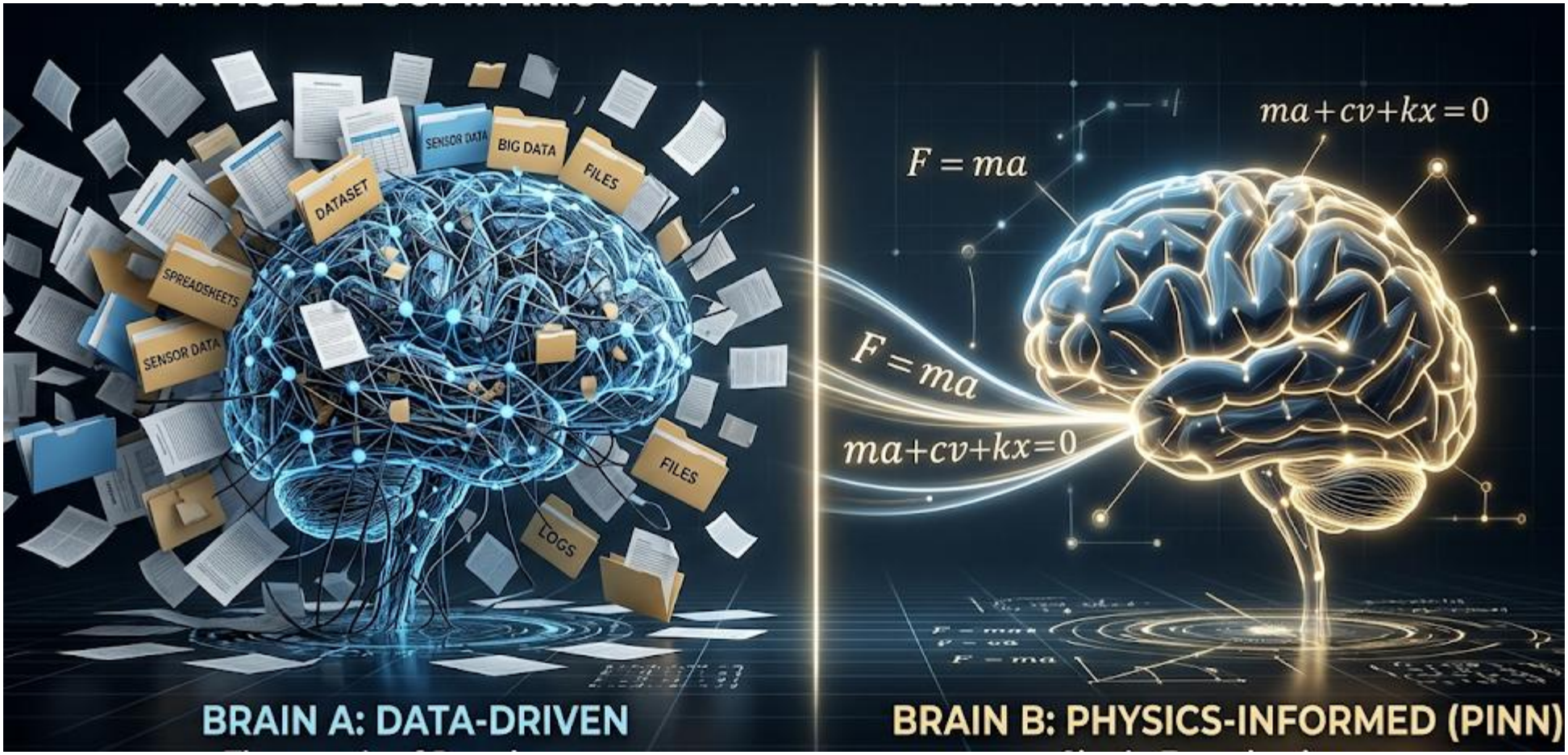
Test case: 3dim/roberts/dynamo With resol=64 and tfinal=0.0025s

Recommendations

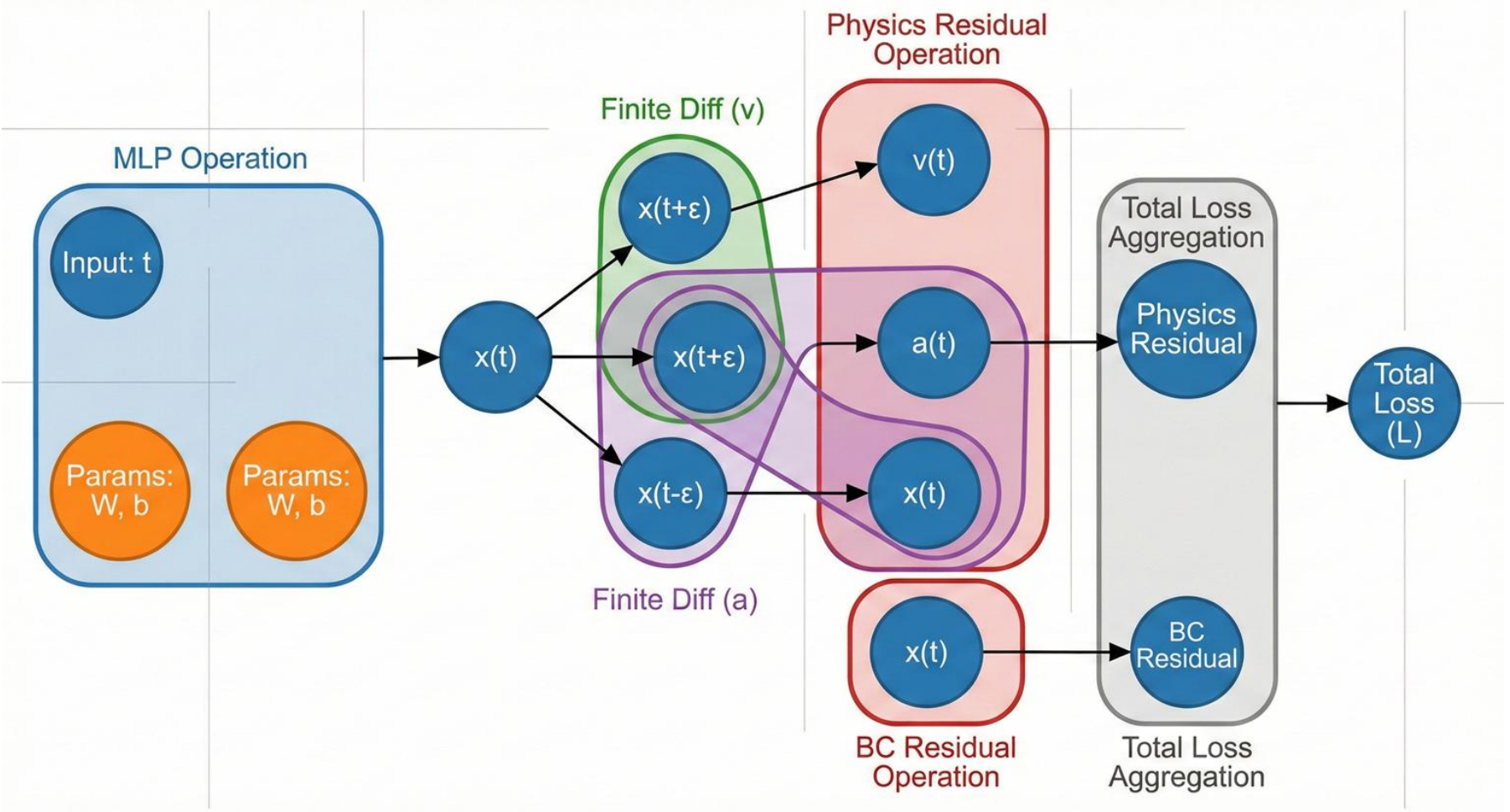
- MUSIC performs many data transfers to and from the device.
  - Reducing these could improve GPU performance.
- Choose Sparse matrix format based on the problem with GPU optimal layout
- Is there a reason to evaluate equations on the CPU.
- Is Trilinos-Kokkos needed or direct callings to AMGx, cublas, cusparse, CUSP*. (Less vendor lock-in than kokkos format).
- Exploit tensor cores, eg cuSPARSELt.



Have to include “AI” in everything



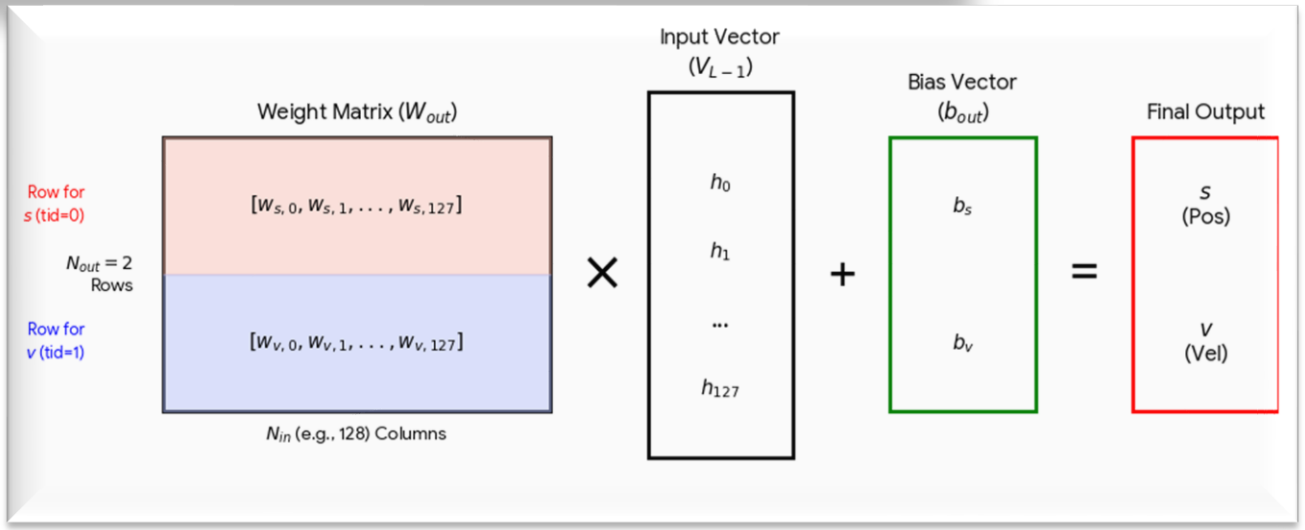
How the “AI experts” view the problem



The Reality

```
          WEIGHT MATRIX (W)                    INPUTS (I)      VALUE OUTPUT (V)
        [4 rows (neurons) x 3 cols]          [3 rows x 1 col]  [4 rows x 1 col]

      +-----------------------------------+     +---------+     +----------------+
Neuron 1 |  W_n1,i1   W_n1,i2   W_n1,i3   |     |   I_1   |     |   V_neuron1    |
      |                                   |     |         |     |                |
Neuron 2 |  W_n2,i1   W_n2,i2   W_n2,i3   |  x  |   I_2   |  =  |   V_neuron2    |
      |                                   |     |         |     |                |
Neuron 3 |  W_n3,i1   W_n3,i2   W_n3,i3   |     |   I_3   |     |   V_neuron3    |
      |                                   |     +---------+     |                |
Neuron 4 |  W_n4,i1   W_n4,i2   W_n4,i3   |                     |   V_neuron4    |
      +-----------------------------------+                     +----------------+
```

$$\begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & w_{0,3} \\ w_{1,0} & w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,0} & w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,0} & w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \cdot \begin{bmatrix} v_{prev,0} \\ v_{prev,1} \\ v_{prev,2} \\ v_{prev,3} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} v_{new,0} \\ v_{new,1} \\ v_{new,2} \\ v_{new,3} \end{bmatrix}$$



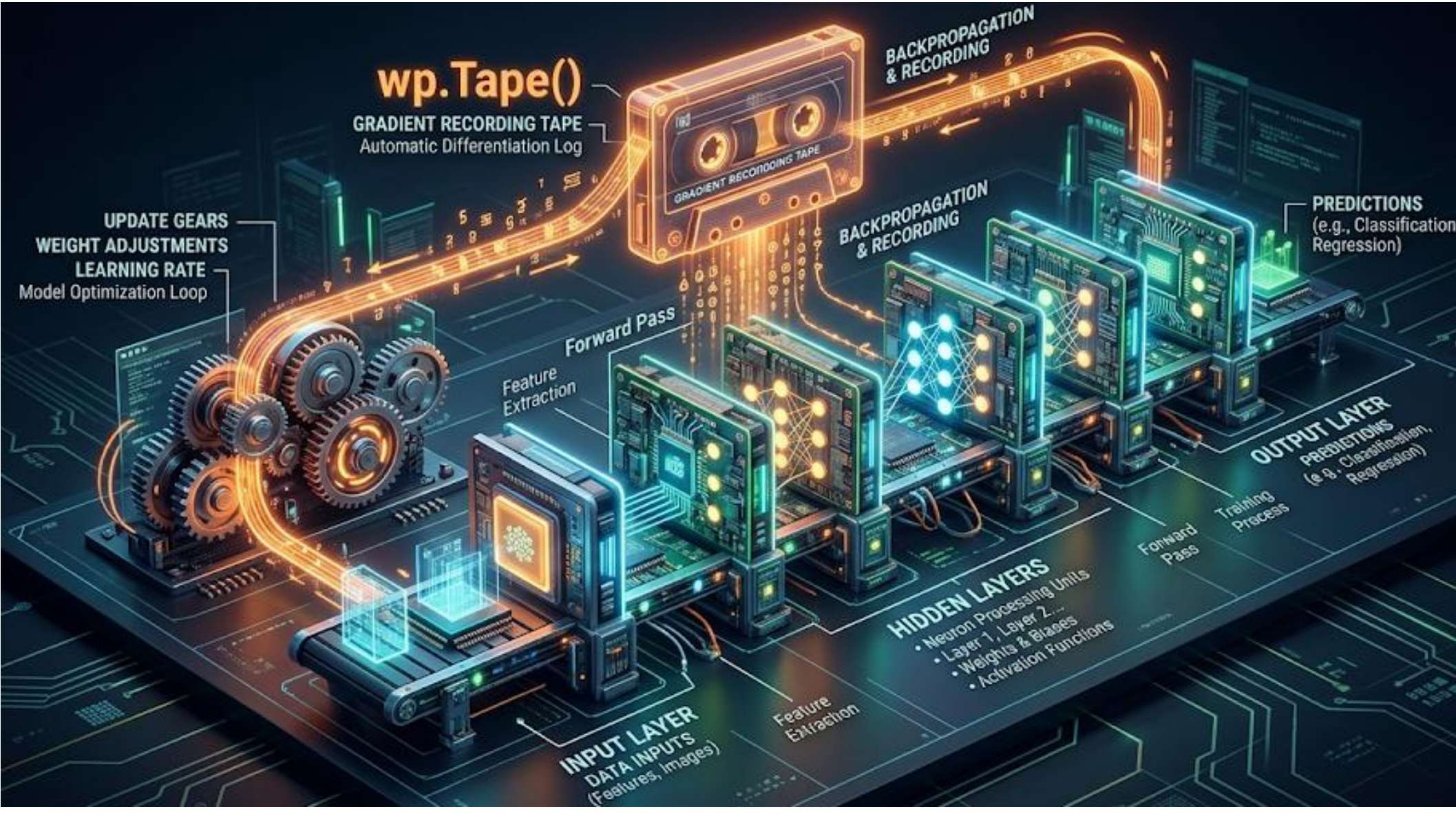
Code Repo

https://github.com/nicolin/PINN_CUDA_Demo/blob/main/1_NGPINNDemo.py

What the code does

Concept	Associated Function / Keyword	Approx. Line Numbers	Key Point
1. The "Physics Loss" Loop	CUDAKern_Loss_PINN()	Lines 86–104	**Concept: Residuals over Data.** s and v are calculated *on the fly* (Line 95) using physical laws instead of being loaded from a dataset. The loss is the difference between the network's prediction and the immutable laws of physics.
	CUDAKern_Loss_HMO()	Lines 110–143	
2. Smooth Manifolds (Activation)	@wp.func	Lines 42–45	**Concept: Differentiability.** PINNs require calculating gradients (derivatives) of the network with respect to its inputs to satisfy differential equations like ma+cv+kx=0. Jagged activation functions break this process.
	def swish(x: float):		
3. Memory Safety in Parallel	wp.atomic_add()	Line 72	**Concept: Race Conditions.** When stepping from Python to CUDA, thousands of threads run simultaneously. If they all update lossTot at once, data is corrupted. atomic_add queues these operations safely at the hardware level.
		Line 142	
4. The Autodiff Tape Recorder	tape = wp.Tape()	Lines 255–256	**Concept: Automatic Differentiation.** wp.Tape() context manager records every forward operation (matrix multiplications, activations). When tape.backward() is called, it replays the tape in reverse to calculate the exact gradients for the SGD update.
	tape.backward(...)	Line 281	
5. JIT Compilation Portal	@wp.kernel	Line 33	**Concept: Warp's JIT Compiler.** Standard Python is too slow for physics simulations. The @wp.kernel decorator takes the Python AST (Abstract Syntax Tree) and compiles it into highly optimized C++/CUDA code before execution.
	wp.init()	Throughout	

CUDA Auto Diff Magic



Did it learn ?

Structural Identity Over Numerical Noise: Traditional data matching fails if sensor noise exists or if variables have different names (e.g., t vs. time_step).

Solution: We ignore the raw numbers and look at the topology of the laws. If Mesh A and Mesh B are isomorphic, the physics is identical, regardless of the coordinate system or units used.

Pre-Training Verification: Structural Checks Save GPU Cycles Solution: Before running 50,000 training epochs, the checker verifies that the PINN is wired correctly for the problem

Gravity vs Gravity: Both meshes show a linear flow of second-order integration (a→v→x). Gravity vs HMO: The checker would identify a feedback loop (cycle) in a spring-mass system that doesn't exist in Gravity, preventing a mismatch. Ensures the model is mathematically capable of solving the target physics before you hit "Train".

Plug-and-Play Inference: Once a PINN is trained on a "Gravity" hypergraph, the .npz weight file can be safely used on any isomorphic mesh.