Introduction to GPU Computing and CUDA

Chik Him (Ricky) Wong

University of Wuppertal

August 29, 2024

Outline

- Part 1: GPU computing
  - What is GPU?
  - GPU Architecture
  - CUDA basics
  - Performance and Optimization
- Part 2: GPU Applications in HEP Research
  - Nuclear Astrophysics:
    - GraCCA: GPU-accelerated N-body simulations
    - AMReX : framework for AMR
  - Nuclear Physics:
    - Lattice QCD
    - Parton Shower calculation
  - Neural Network applications

GraCCA

Hsi-Yu Schive et al, 2007 NewAstron.13:418-435,2008 [arXiv:0707.2991]

- GraCCA (Graphic-Card Cluster for Astrophysics) is a specialized computing system designed for astrophysical simulations
- Gravity is long-ranged interaction
  $\Rightarrow$ N-body Simulation involves N^2 interaction calculations
- Hybrid model:
  - GPU : the acceleration and jerk on i-particles exerted by j-particles
  - CPU : for other tasks

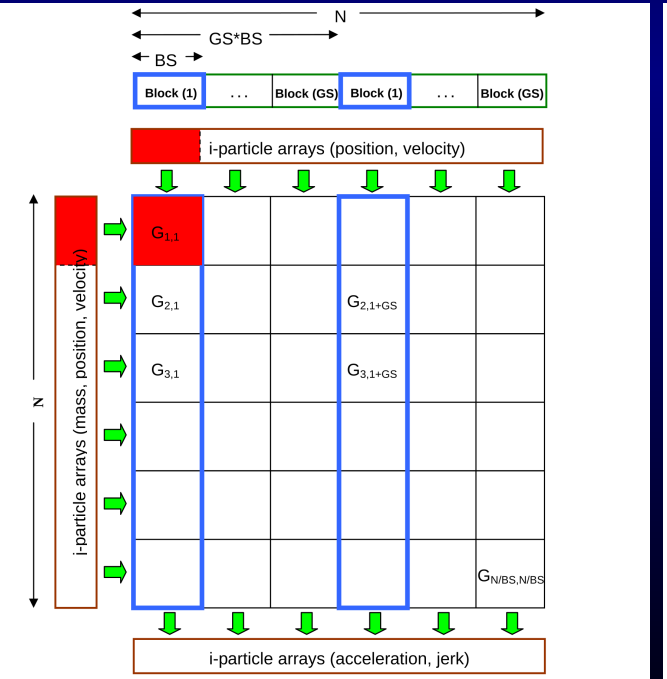


Fig. 6. The schematic diagram of our single-GPU implementation for acceleration and jerk calculations. The interaction groups computed by Block(1) are highlighted with blue border. The red regions in i-particle and j-particle arrays are the particles used to compute the group $G_{1,1}$.

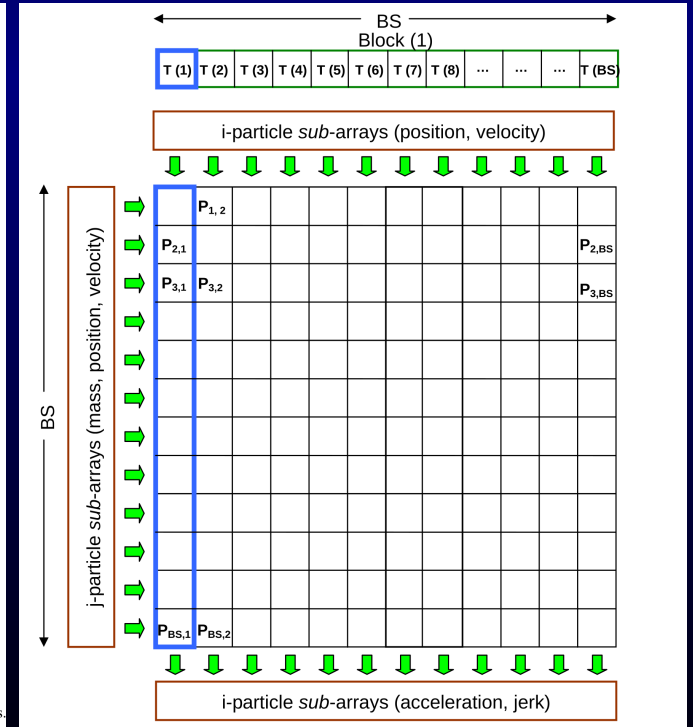
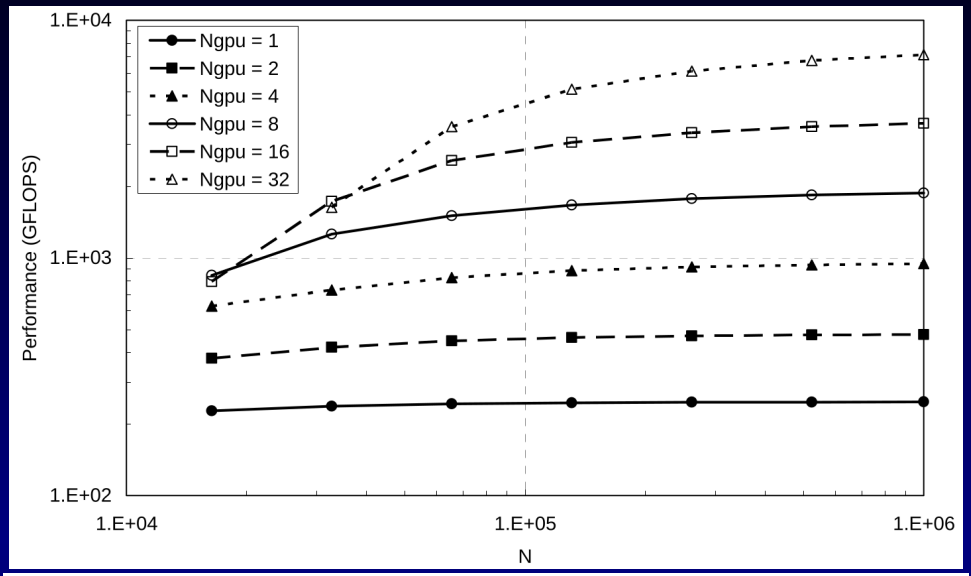
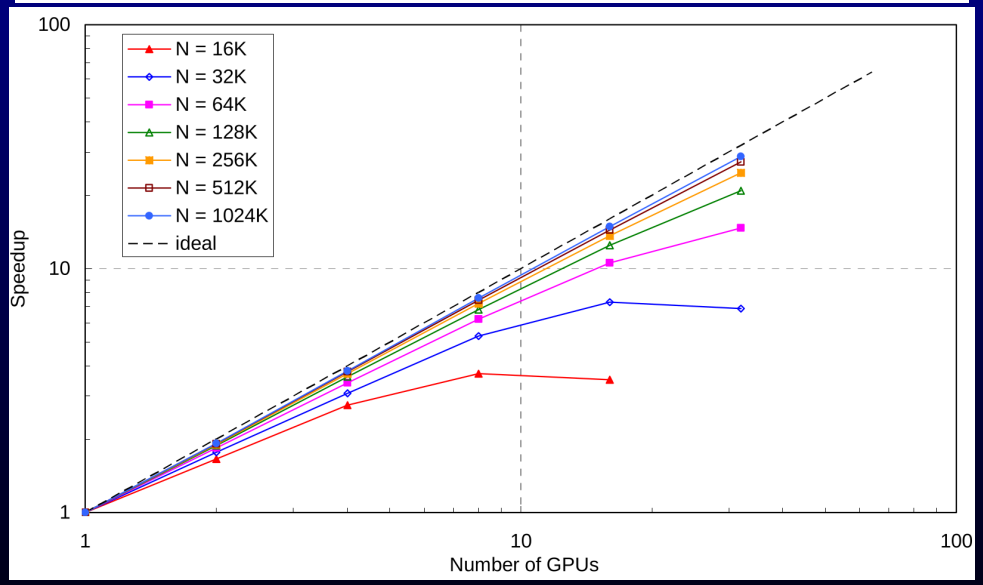


Fig. 7. The schematic diagram of the evaluation of group $G_{1,1}$ in Fig. 6. The T(s) stands for the sth thread within the Block(1). The interaction pairs computed by T(1) are highlighted with blue border.

GraCCA


1.E+04
1.E+03
1.E+02
Performance (GFLOPS)
1.E+04
1.E+05
1.E+06
N
Ngpu = 1
Ngpu = 2
Ngpu = 4
Ngpu = 8
Ngpu = 16
Ngpu = 32

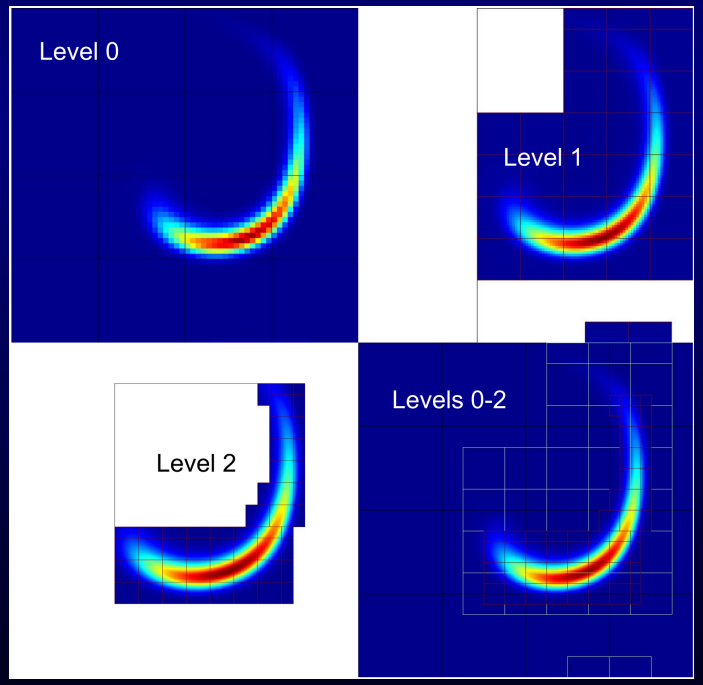

100
10
1
Speedup
1
10
100
Number of GPUs
N = 16K
N = 32K
N = 64K
N = 128K
N = 256K
N = 512K
N = 1024K
ideal

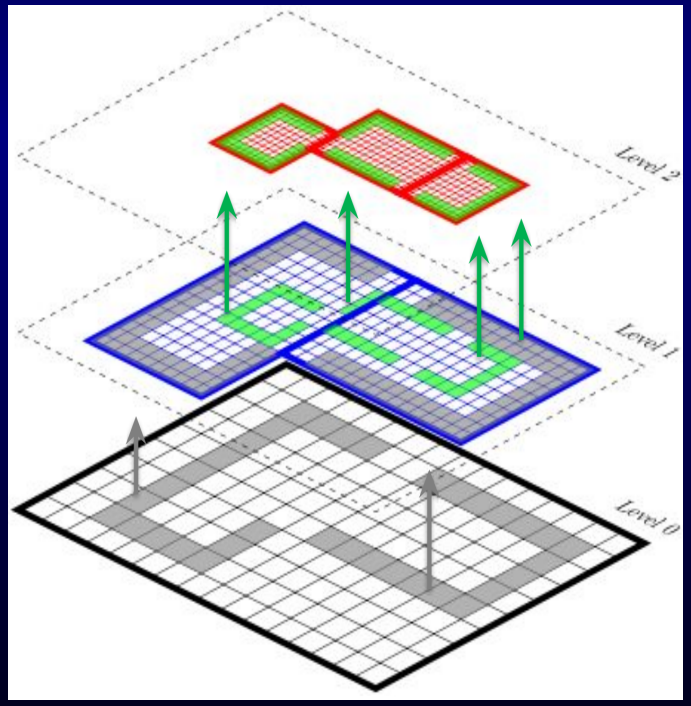
AMReX

W. Zhang et al, 2020, 2009.12009v1

https://drive.google.com/file/d/1-Fn6peoPj6zRc-iV-j1_Zc3YHoKZM2C9/view?usp=sharing

- Adaptive Mesh Refinement (AMR): Dynamically adjusts the resolution of the computational grid during a simulation, focusing computational resources on areas that require higher precision while using a coarser grid in regions of less interest

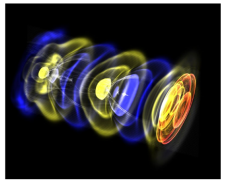


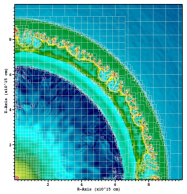


AMReX

A sampling of AMReX-based application codes

Astrophysics:
Castro (compressible)
MaestroEx (low-Mach)
SedonaEx (Monte Carlo radiation transport)
Emu (neutrino transport)
Quokka (radiation-hydrodynamics)



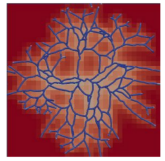


Incompressible Navier-Stokes:
IAMR
incflo

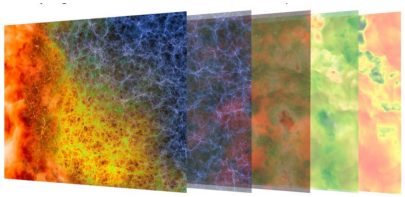
Solid Mechanics:
Alamo

Atmospheric science:
AMR-wind
ERF

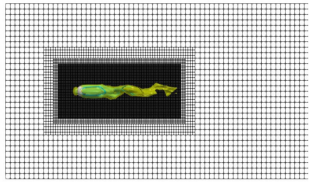
Biological cell modelling:
BoltzmanMFX
CCM



Cosmology:
Nyx



Combustion:
PeleC (Compressible)
PeleLM (Low Mach)



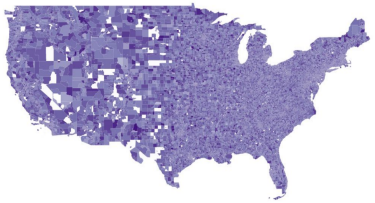
Multi-phase Flow:
MFIX-Exa

Accelerator Modelling:
WarpX
ImpactX
Hipace++

Multiscale Modelling and Stochastic Systems:
FHDeX

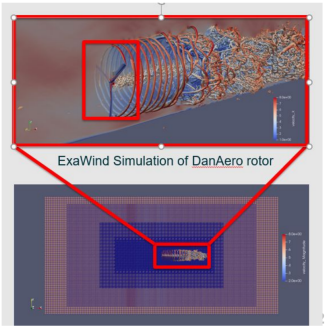
Electromagnetics:
ARTEMIS

Epidemiology:
ExaEpi



Magnetically-confined fusion:
GEMPIC

Ocean Modeling:
REMORA

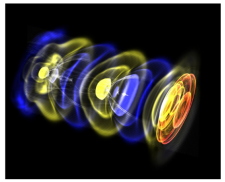


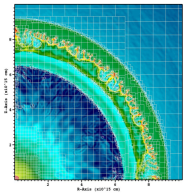
- AMReX is a publicly available software framework designed for building massively parallel block-structured AMR applications
  - MAESTROeX: Used for low Mach number astrophysics simulations
  - SedonaEX: Calculates radiation signatures of supernovae and other transient astrophysical phenomena
  - Emu: Focuses on neutrino quantum kinetics

AMReX

A sampling of AMReX-based application codes

Astrophysics:
Castro (compressible)
MaestroEx (low-Mach)
SedonaEx (Monte Carlo radiation transport)
Emu (neutrino transport)
Quokka (radiation-hydrodynamics)



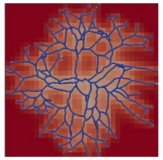


Incompressible Navier-Stokes:
IAMR
incflo

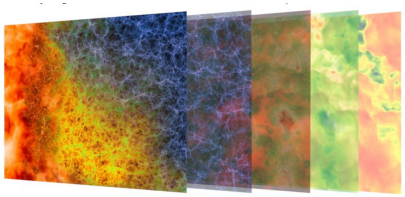
Solid Mechanics:
Alamo

Atmospheric science:
AMR-wind
ERF

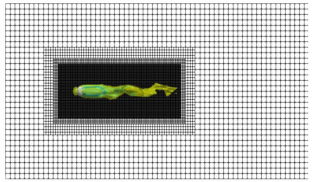
Biological cell modelling:
BoltzmanMFX
CCM



Cosmology:
Nyx



Combustion:
PeleC (Compressible)
PeleLM (Low Mach)



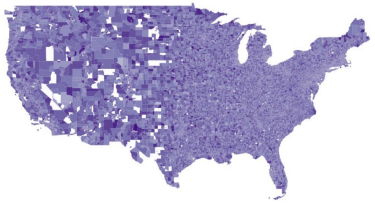
Multi-phase Flow:
MFIX-Exa

Accelerator Modelling:
WarpX
ImpactX
Hipace++

Multiscale Modelling and Stochastic Systems:
FHDeX

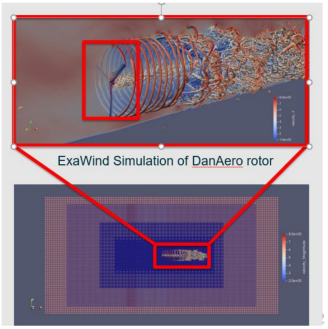
Electromagnetics:
ARTEMIS

Epidemiology:
ExaEpi



Magnetically-confined fusion:
GEMPIC

Ocean Modeling:
REMORA

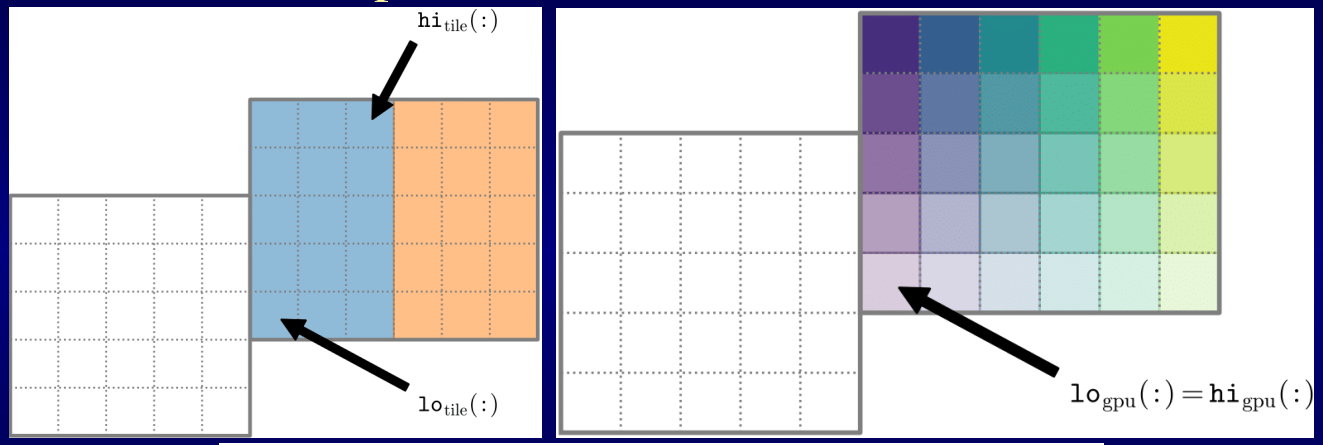


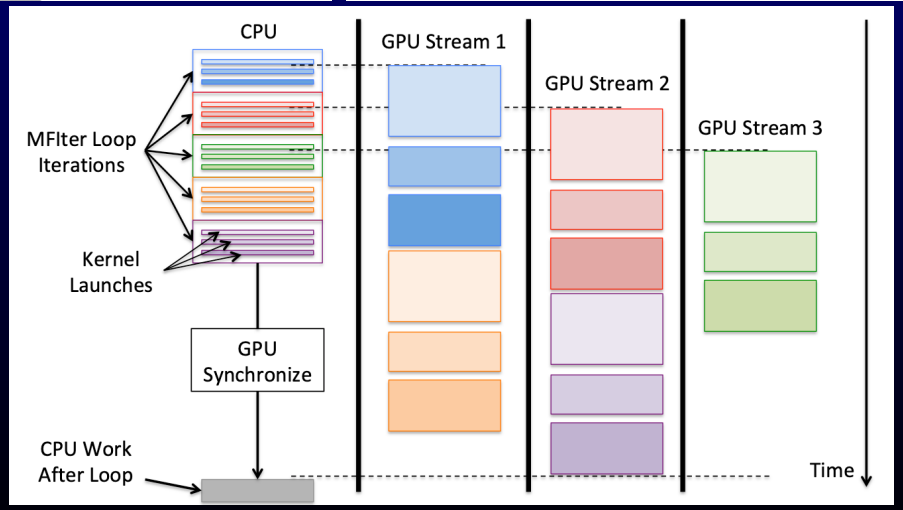
- C++ and Fortran interfaces
- 1-, 2- and 3-D support
- Parallelization via flat MPI, OpenMP, hybrid MPI/OpenMP, hybrid MPI/(CUDA or HIP or SYCL), or MPI/MPI

AMReX

- Each resolution level contains an array of "boxes"(domains) and keep track of the interations within and among them
- $\Rightarrow$ Main Task: Loop over these boxes and across levels
- Highly parallelizable $\Rightarrow$ Speed up by GPU

CPU(OpenMP) $\Rightarrow$ GPU



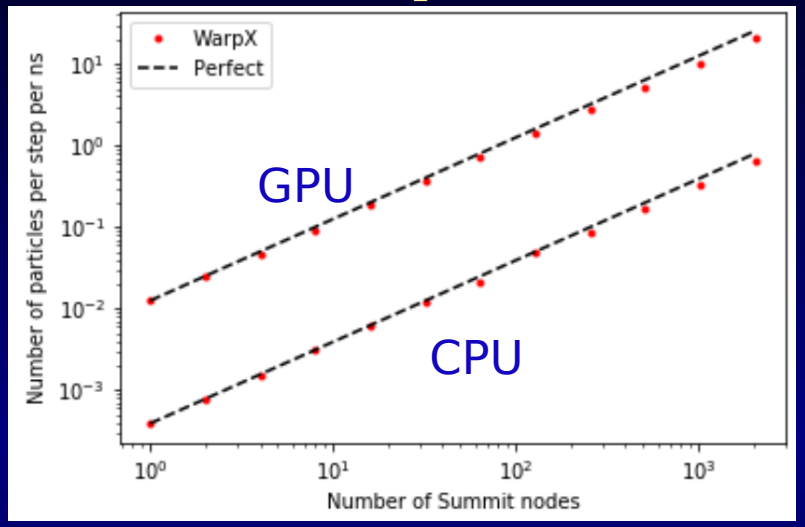


AMReX

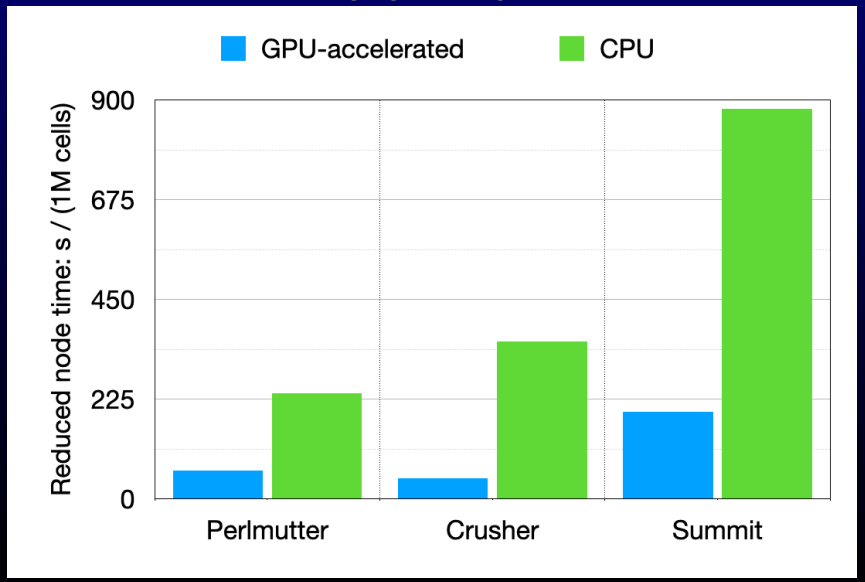
https://amrex-combustion.github.io/PeleLMeX/manual/html/Performances.html

- There is significant speed up by GPU:

WarpX



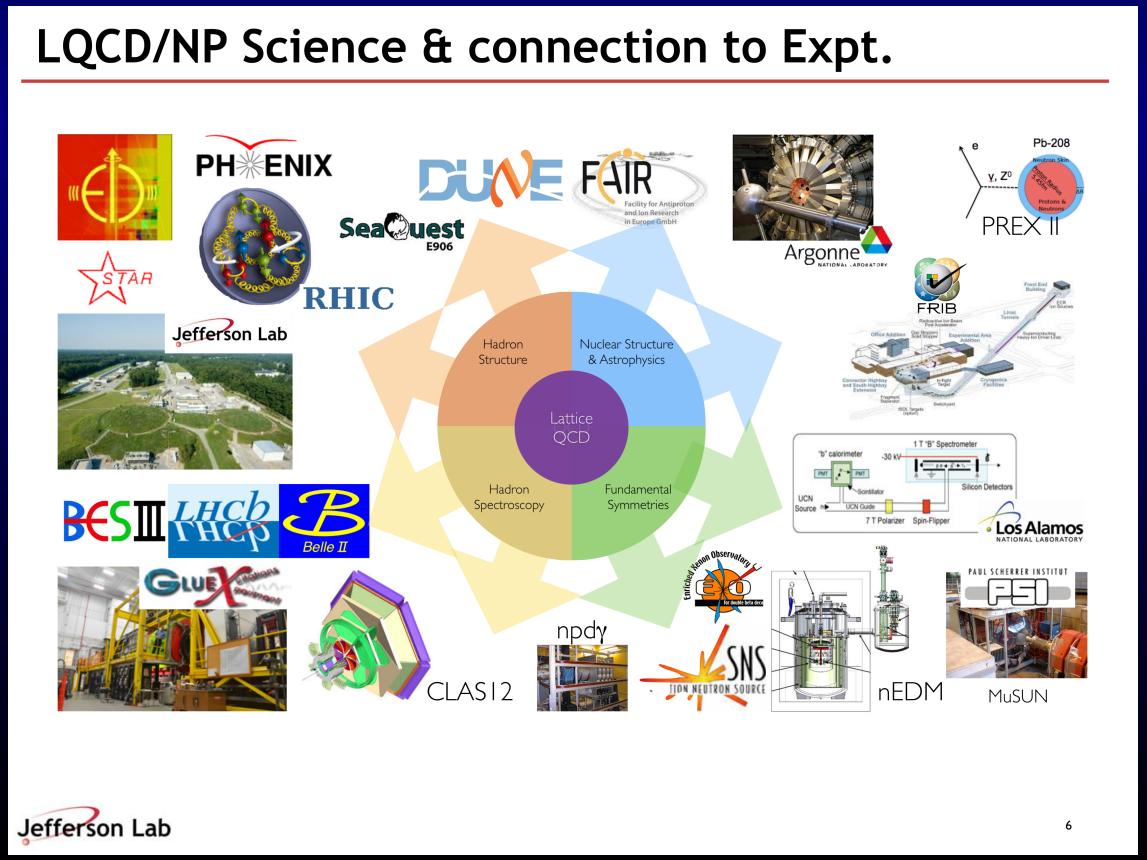
PeleLMeX



Lattice QCD

https://userweb.jlab.org/ edwards/talks/edwards_lqcd_scidac_19.pdf

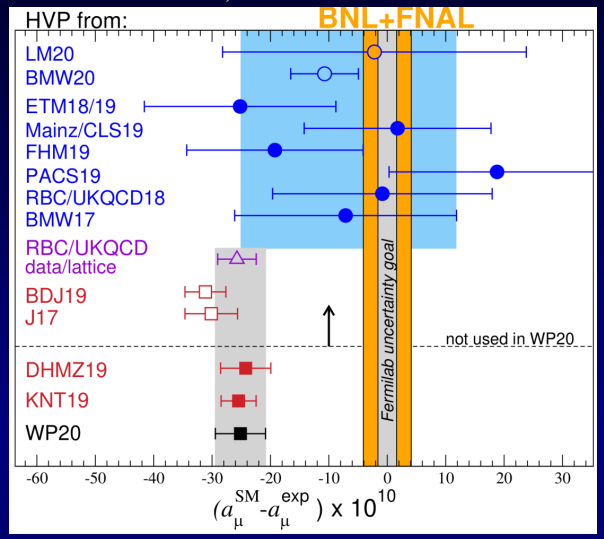
- Many important properties of QCD, e.g. Hadron Spectroscopy, Λ_{QCD} etc, involve computations at IR scales where perturbation is not applicable $\Rightarrow$ Non-perturbative methods are required
- Lattice QCD :
  First-principle computation of QCD on discretized finite-volume spacetime grids, then take continuum + infinite-volume limit



Lattice QCD

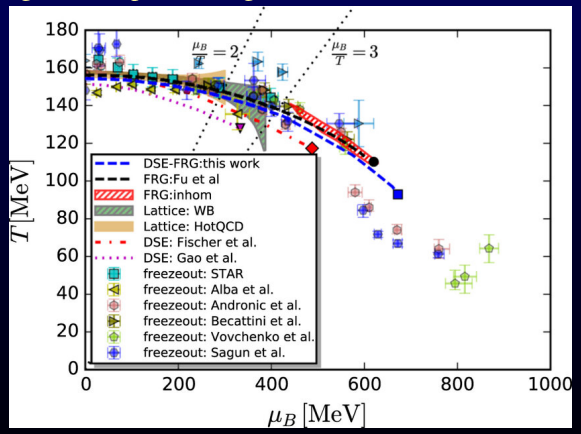
- $g_\mu - 2$ calculation

A. S. Kronfeld et al, 2207.07641v1



- QCD phase diagram

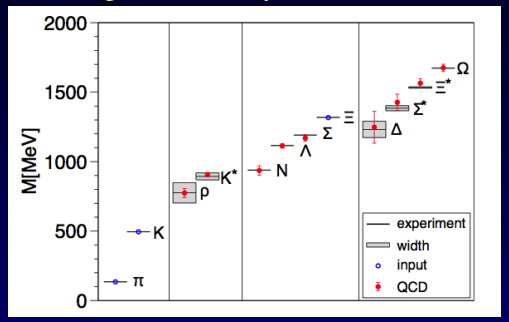
H.-T. Ding et al. https://doi.org/10.1007/978-981-19-4441-3_1



(2022)

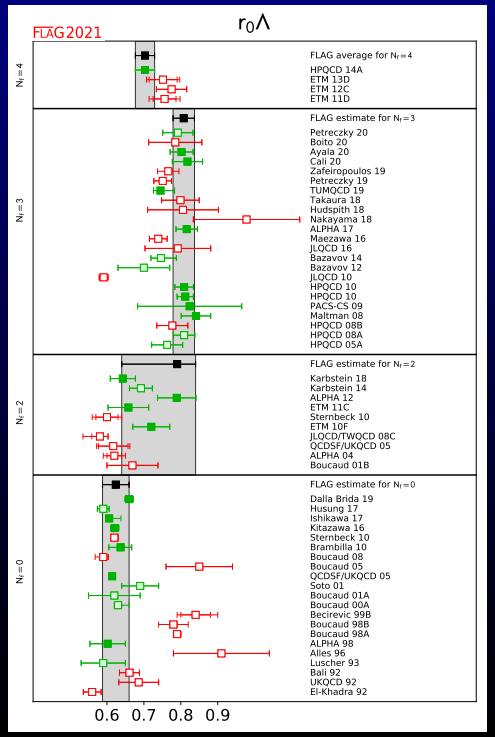
- Hadron Spectroscopy

M. Battaglieri, Acta Phys.Polon. B46 (2015) 2, 257



- Λ_{QCD} calculation

Y. Aoki et al, Eur.Phys.J.C 82 (2022) 10, 869



Lattice QCD

- Wick rotation: $t \to -i\tau$
- Partition function with action S_M rotated into Eucledian spacetime is mathematically equivalent to that of a Classical Statistical Mechanical system with Hamiltonian S_E

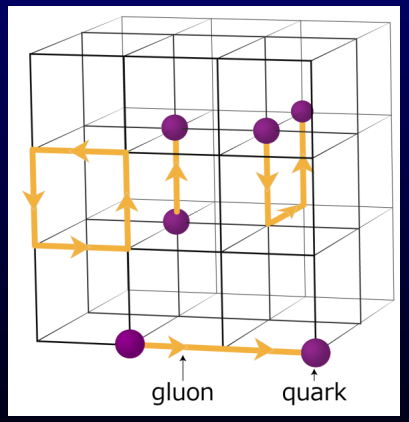
$$Z_M = \int_{\{\phi\}} e^{iS_M/\hbar} = \int_{\{\phi\}} e^{\frac{i}{\hbar}\int L_M(\{\phi\}x,t)d^3xdt}$$

$$\to Z_E = \int_{\{\phi\}} e^{-\beta S_E} = \int_{\{\phi\}} e^{-\beta\int L_E(\{\phi\},x,\tau)d^3xd\tau}$$

- If S_E is real:
  - Monte Carlo Integration over the fields is applicable
  - $\Rightarrow$ Simulation is viable
- If not, it is known as sign problem. Some tricks can be used to make it work to some extent (e.g. reweighting)
- If boson/fermion has periodic/antiperiodic temporal boundary conditions, the temporal lattice size corresponds to inverse temperature

Lattice QCD

- Discretization of QCD action involves a change of variables:
  - Gauge field: Link variables (SU(3) matrices)
    $A_\mu(x,t) \to U_\mu(x,\tau)$
  - Quark field: pseudo fermion fields (not Grassmann)
    $\psi(x,t) \to \psi(x,\tau)$
  - Inverse propagator: Dirac operator
    $\bar{\psi}(i\gamma^\mu D_\mu(A) - m)\psi \to \bar{\psi}(x_1,\tau_1)D(U;x_1,\tau_1;x_0,\tau_0)\psi(x_0,\tau_0)$
- All Grassmann degrees of freedom are integrated out into Dirac operator, and all observables are only in terms of U and Dirac operator
  $\Rightarrow$ A suitable distribution ($e^{-\beta S_E}$) of gauge fields (configurations) fully captures all physics required



Y. Bi et al, EPJ Web of Conferences 245, 09008 (2020)

Lattice QCD

- Ensembles of gauge fields are generated by Hybrid Monte Carlo algorithm
- Molecular Dynamics Evolution: Equation of motion of a fictitious fluid with Hamiltonian defined as S_E, evolving U along a fictitious time t_{MD}

$$\frac{dU}{dt_{\mathrm{MD}}} = P, \quad \frac{dP}{dt_{\mathrm{MD}}} = -\frac{\partial S_E}{\partial U}$$

- An initial random (or thermalized) gauge field is evolved according to Molecular Dynamics Evolution at discretized t_{MD}
- The Hamiltonian (S_E) is approximately conserved, but fluctuates due to discretization of t_{MD} (on purpose)
- At each MD step, accept the change with probability based on the change in S_E (a Metropolis accept-reject test)
- The result is a Markov-Chain of U with the desired distribution

Lattice QCD

- Typical scenario:
  - lattice size: $48^3 \times 96 \Rightarrow$ Operator size: $10^7 \times 10^7$
  - Link variables: 3×3 matrix , per direction per site $\approx 10^8$ entries $\Rightarrow$ $\approx GB$ per configuration
  - $O(10^3)$ configurations needed
- Dirac operator inversion is very computational intensive
- Action on lattice cannot preserve all continuum symmetries
  $\Rightarrow$ A denser grid or more sophisticated discretization is needed
- QCD is distorted by being trapped in a box
  $\Rightarrow$ A larger box is needed
- Sign problem, Overlap problem, Frozen Topology
  $\Rightarrow$ Better algorithms or more computational power are needed
- $\Rightarrow$ Bad News:
  Computation is overwhelmingly intensive
- Good News:
  The Lattice formalism makes it very parallelizable

Lattice QCD

- The most computational expensive part is the Dirac matrix, in which typical terms read

$$U_{\mu}^{aa'}(x)\psi_{\alpha,a'}(x+\hat{\mu})\ ,\ U_{\mu}^{aa'\dagger}(x-\hat{\mu})\psi_{\alpha,a'}(x-\hat{\mu})$$

- The discretized QCD action is (usually) local
  $\Rightarrow$ Sparse matrix that only involves neighboring sites
- The lattice can be naturally divided into local sublattices, whose interactions are only via boundary surfaces
  $\Rightarrow$ parallelization per sublattice
- Computation of each site can be independently done given that the data at neighboring sites
  $\Rightarrow$ parallelization per site

Lattice QCD

- Parallelization highly applicable ⇒ GPU can accelerate
- Many different GPU codes are developed:
  - QUDA (USQCD):
    A comprehensive CUDA library dedicated to Lattice QCD
  - OpenQCD (CERN):
    Focus on open boundary conditions and Huge lattice sizes
  - HiRep (U of Southern Denmark):
    Focus on simulation with fermions in higher representations and variable number of colors, in the context of BSM studies
  - Janko (U of Wuppertal):
    Focus on thermodynamics studies

Parton Shower Simulations

S. Höche, 1411.4085v2

- Event generators that simulate the hard interation between partons and soft interactions between hadrons is crucial in the understanding of the detected events in the colliders
- The evolution between the scales utilizes parton shower and hadronization models, and accurate simulation is essential
- Parton shower models the splitting of partons' energy and momenta into more partons and at lower scales in consecutive branching, while hadronization model combines the final state partons into hadrons
- Many simulated events are required to reduce the simulation uncertainty and allow exotic (very rare) events to be simulated.
- $\Rightarrow$ Such calculations, done by the so-called Monte Carlo Event Generators, is computationally expensive
- The ATLAS Detector's HL-LHC Roadmap document:
  - Event generators form around 14% of CPU usage
  - Conservative CPU usage cannot maintain a sustainable budget
- $\Rightarrow$ GPU acceleration is needed

Sudakov Veto Algorithm

M. H. Seymour, 2024, 2403.08692v1

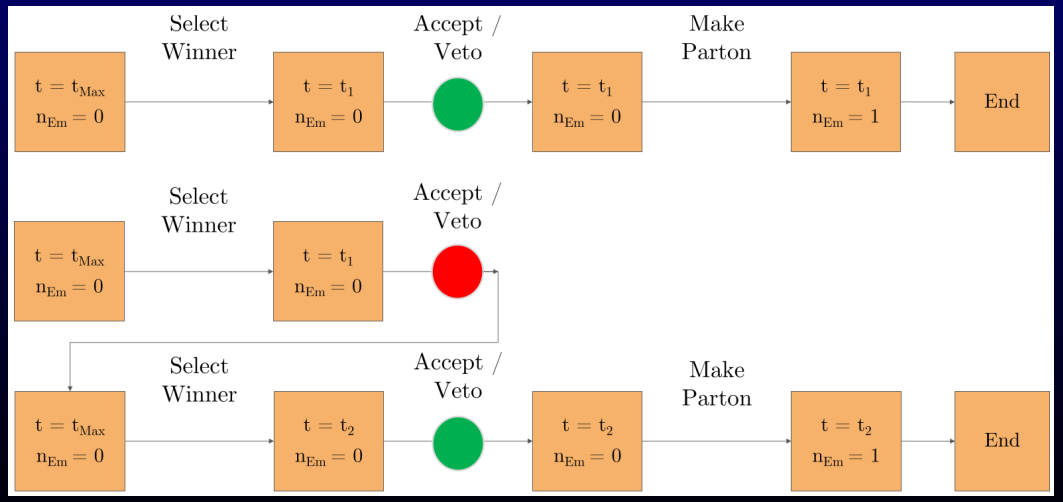
- An algorithm used for parton shower calculation
- Each branching of a parton $\tilde{ij}$ to partons i and j is characterized by a set of values (t, z, ϕ)
  - t: scale of momentum transfer
  - z: defines how energy is splitted between the children
  - ϕ: Azimuthal angle of the branching
- Task: Generate a distribution of (t, z, ϕ) such that the probability of the branching is dictated by known physics
- Sudakov form factor: the probability that no emissions occur between the initial scale of the system T and a smaller scale t

$$\Delta(t_0, t_1) = \exp\left[-\int_t^T dt' \left(\frac{1}{t'}\int_{z_-}^{z_+} dz \frac{\alpha_s(p_\perp^2(t', z))}{2\pi} P(z)\right)\right]$$

- The probability to branch at scale t with evolution starting at T is then given by Poisson statistics $P = d\Delta/d\ln t$
- It is not trivial to compute and invert the term within the integral $\rightarrow$ some modifications are needed to generate such distribution by Monte Carlo
- The resulting algorithm is known as Veto algorithm

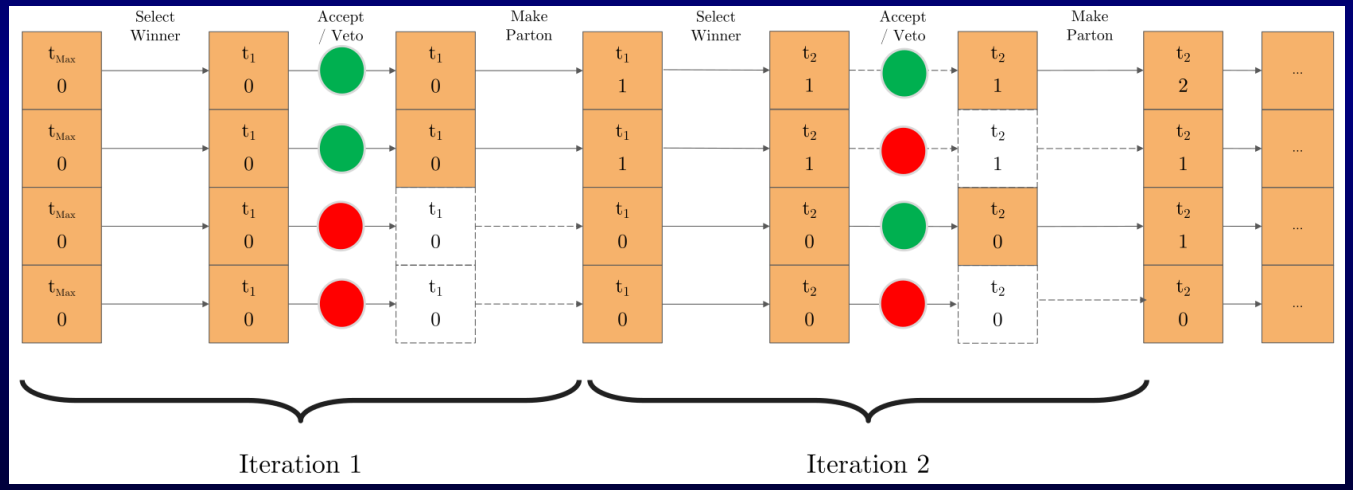
Sudakov Veto Algorithm

- Propose new (t, z, ϕ) from a wrong distribution, which can be analytically inverted
- Among all possible branchings considered, select a "Winner" who proposes the highest new t
- Accept the Winner with probability dictated by the ratio between correct and wrong distribution
- If accepted, generate the corresponding partons and go to the next iteration
- If rejected, replace t_0 with t_1 and propose new set of (t, z, ϕ)
- If $t < t_C$ for some cutoff scale allowed, set t back to t_0



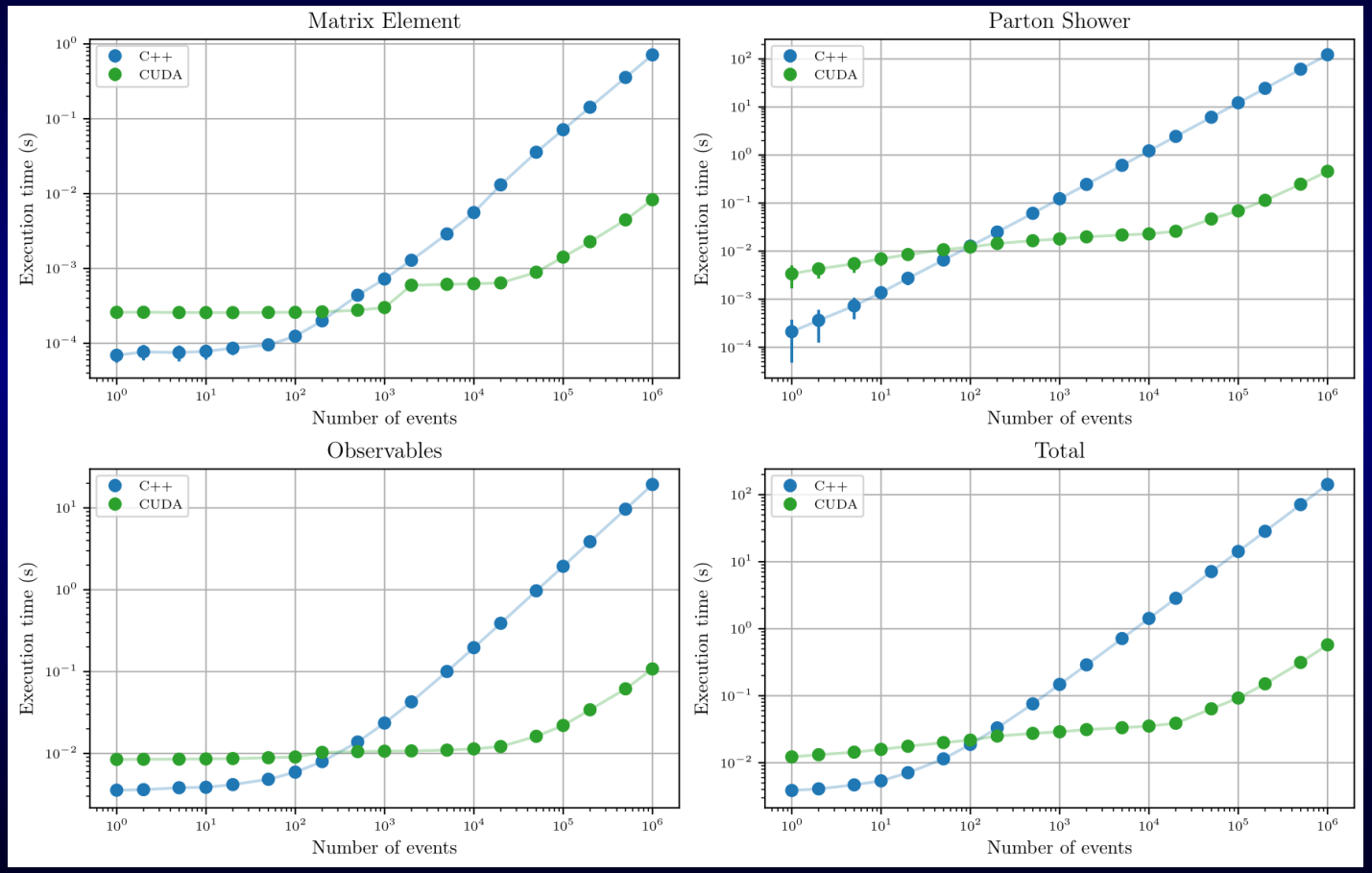
Sudakov Veto Algorithm

- This algorithm can be parallelized on GPU
- Threads could execute different commands independently, hence the effect of unpredictable termination of each event is not significant



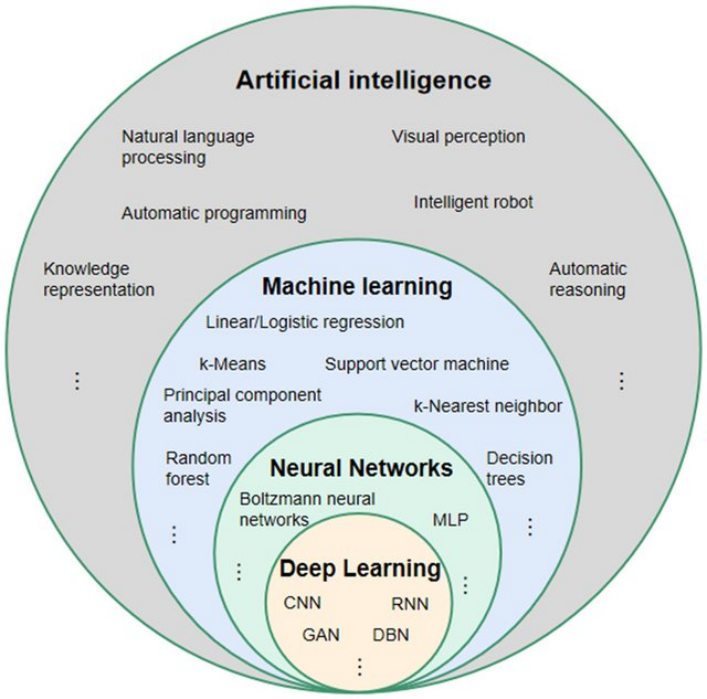
Sudakov Veto Algorithm

- There is significant speed up by GPU:

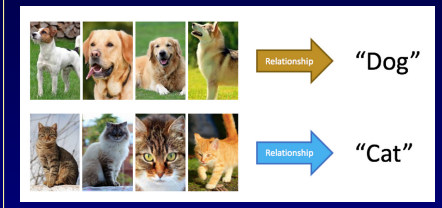
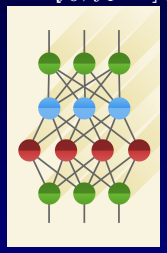
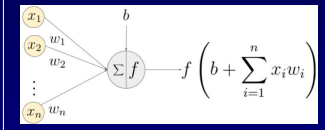
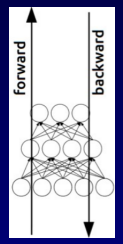


Machine Learning and Neural Network

- Artificial Intelligence:
  Machines carry out tasks in an intelligent way
- Machine Learning:
  Machines learn via data to acquire intelligence
- Neural Network:
  A type of Machine Learning mimicking biological neural networks
- Deep Learning:
  Many-layered Neural Networks are used

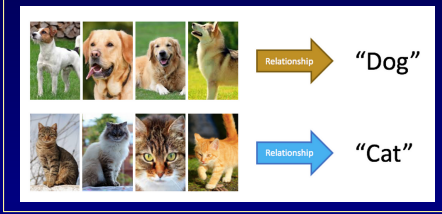
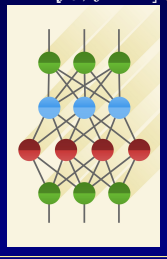
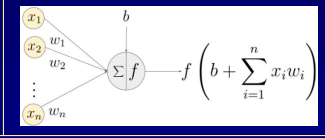
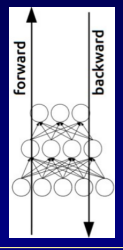


Machine Learning and Neural Network

Input	Ansatz	Parameters	Fitting	Inter/Extra-polation
$\{x=0, y=1\}, \{x=1, y=0.5\} \ldots$ 	$y = F(x) = a + bx$	a, b	$\min(\chi^2)$ 	$y = F(x = 0.13) = 3.14(2)$ 
Training data 	Model architecture Label = $F[f_0, f_1 \ldots]$(Img) 	Weights / Bias W , B 	Training $\min(L)$ 	Inference "Dog" (92% likely) 

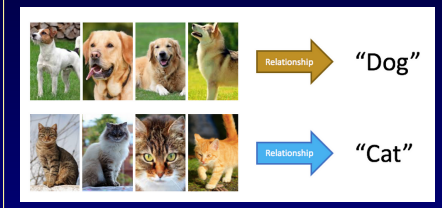
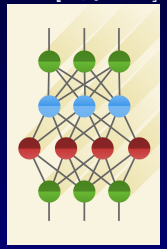
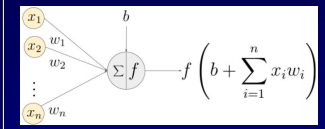
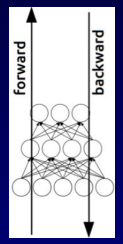
- Goal of Machine learning:
  To infer a mapping between input parameters and outputs to make predictions (a.k.a. Fitting data with a function)
- "Model" of a Neural Network:
  a fitting function with Overwhelmingly Large Number of Parameters.
- This distinguishes Neural Network from ordinary curve fittings
- Application in wide range of domains:
  Most functions in reality can be approximated by sufficiently large number of parameters
  $\Rightarrow$ Most systems can be fitted given enough data

Machine Learning and Neural Network

Input	Ansatz	Parameters	Fitting	Inter/Extra-polation
$\{x=0, y=1\}, \{x=1, y=0.5\}\ldots$ 	$y = F(x) = a + bx$	a, b	$\min(\chi^2)$ 	$y = F(x = 0.13) = 3.14(2)$ 
Training data 	Model architecture Label = $F[f_0, f_1\ldots]$(Img) 	Weights / Bias W , B 	Training $\min(L)$ 	Inference "Dog" (92% likely) 

- "Neural Network": a Model implicitly defined by an arrangement of connections of smaller components "Neurons" . "Neurons" are simplified sigmoid-like mathmatical functions with adjustable parameters (weights and bias), briefly resembling biological neural networks.
- "Train a model" : Adjust (fit) the parameters with existing input data or from feedback loops

Machine Learning and Neural Network

Input	Ansatz	Parameters	Fitting	Inter/Extra-polation
$\{x=0, y=1\}, \{x=1, y=0.5\}\ldots$ 	$y = F(x) = a + bx$	a, b	$\min(\chi^2)$ 	$y = F(x = 0.13) = 3.14(2)$ 
Training data 	Model architecture Label = $F[f_0, f_1\ldots](\text{Img})$ 	Weights / Bias W , B 	Training $\min(L)$ 	Inference "Dog" (92% likely) 

- Loss function:
  Defines the difference between predicted and actual outputs to be minimized, e.g.
  - Regression: Mean Squared Error (MSE)
    $L = \chi^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^{\text{data}})^2$
  - Catagorization: Cross Entropy
    $L = -\sum_x p(x)\ln q(x)$
  - Generative: Relative Entropy (Kullback-Leibler Divergence)
    $L = \sum_x p(x)\ln(p(x)/q(x))$
- Usually combined with a term proportional to absolute sum (L1) or squared sum (L2) of the weights as regularization to prevent overfitting

Training (Back Propagation) in Neural Networks

- Loss function:
  Defines the difference between predicted and actual outputs to be minimized
- Forward propagation: Computation of Loss function
- Back-Propagation:
  Updates the weights and biases according to corresponding gradients of the loss function, i.e. Gradient Descent method

$$w_{ij} \to w_{ij} - \alpha \cdot \frac{\partial L}{\partial w_{ij}}, \quad \frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial f_j} \cdot \frac{\partial f_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$$

$$b_j \to b_j - \beta \cdot \frac{\partial L}{\partial b_j}, \quad \frac{\partial L}{\partial b_j} = \frac{\partial L}{\partial f_j} \cdot \frac{\partial f_j}{\partial b_j}$$

- α, β: learning rates
- f_j: Activation function of neuron j:
  nonlinear function mapping input with output using w_{ij} and b_j
- z_j: weighted sum of inputs to neuron j

Parallelisms in Neural Networks

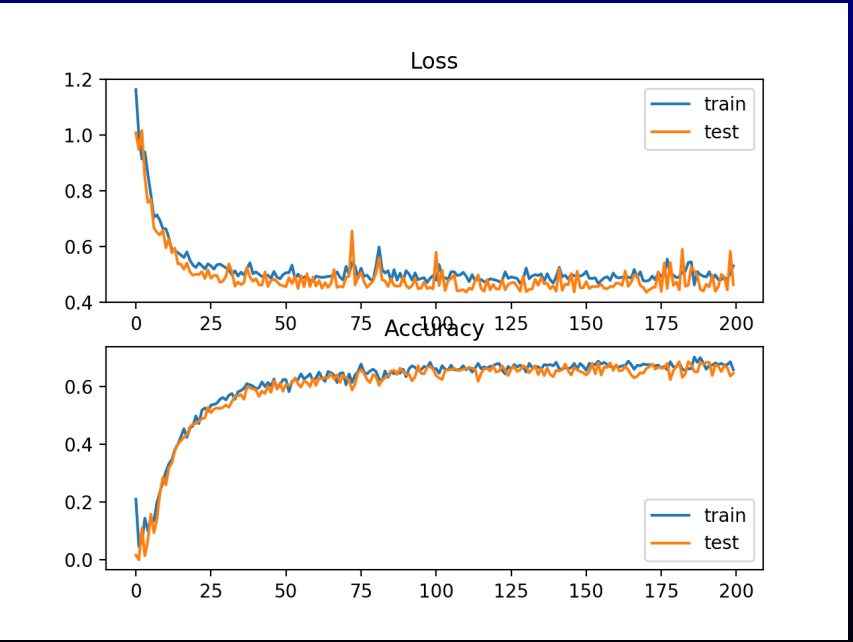
- "With great number of parameters comes great ... ":
  - Demand of Data : To avoid overfitting
  - Demand of Computational Power : For Training and Inference
- $\Rightarrow$ Parallelism is crucially needed
- Training:
  - Data Parallelism:
    Trained on seperate datasets and combined in occational full back-propagation
  - Task Parallelism:
    The model is broken down into different parts and trained concurrently (Distributable models, Mixture of Experts)
- Inference:
  - Data Parallelism:
    Handle multiple requests concurrently
  - Task Parallelism:
    Agents, Mixture of Experts
- Both Parallelisms can be sped up by GPUs

Using Neural Networks in Python

- Python:
  Popular language for machine learning and neural networks due to Huge and ever growing number of related libraries.
- Most relevant prerequites:
  - NumPy: Efficient numerical computations
  - Pandas: Data manipulation and analysis
  - Scikit-learn: Machine learning tools
  - Matplotlib/Seaborn: Data visualization
- Available Deep learning frameworks:
  - TensorFlow/Keras: High-level APIs, production-ready
  - PyTorch: Dynamic computation graphs, research-friendly
- These libraries are optimized in multi-GPU (and multi-CPU) environment

Using Neural Networks in Python

- Components of a neural network:
  - Input layer
  - Hidden layer(s)
  - Output layer
  - Activation functions (e.g., ReLU, Sigmoid)
- Steps:
  1. Data preparation and preprocessing
  2. Model definition
  3. Training (forward and backward propagation)
  4. Evaluation and prediction



Using Neural Networks in Python

```
import numpy as np
import pandas as pd
from sqlalchemy import create_engine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

#Database connection
engine = create_engine(
'postgresql://username:password@localhost:5432/mydatabase')
Read data from database
query = "SELECT feature1, feature2, ..., target FROM my_table"
df = pd.read_sql(query, engine)

#Prepare data
X = df.drop('target', axis=1).values
y = df['target'].values
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2, random_state=42)

#Normalize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Using Neural Networks in Python

```
#Define the model
model = Sequential([
Dense(64, activation='relu', input_shape=(X.shape,)),
Dense(32, activation='relu'),
Dense(1, activation='sigmoid')
])

#Compile and train
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_split=0.2)

#Evaluate
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy:.4f}")

#Make predictions
new_data =
pd.read_sql("SELECT feature1, feature2, ... FROM new_data_table", engine)
new_data_scaled = scaler.transform(new_data)
predictions = model.predict(new_data_scaled)
```

Using Neural Networks in Python

```
import torch
import torch.nn as nn
import torch.optim as optim
from mpi4py import MPI
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#Set up CUDA
gpu_id = rank % torch.cuda.device_count()
device = torch.device(f"cuda:{gpu_id}")

#Get data
...
# Convert to PyTorch tensors
data = torch.FloatTensor(X_train_scaled)
targets = torch.FloatTensor(y_train)

# Divide data among MPI processes
data_size = len(data)
local_data_size = data_size // size

local_data = data[rank*local_data_size:(rank+1)*local_data_size
].to(device)
```

Using Neural Networks in Python

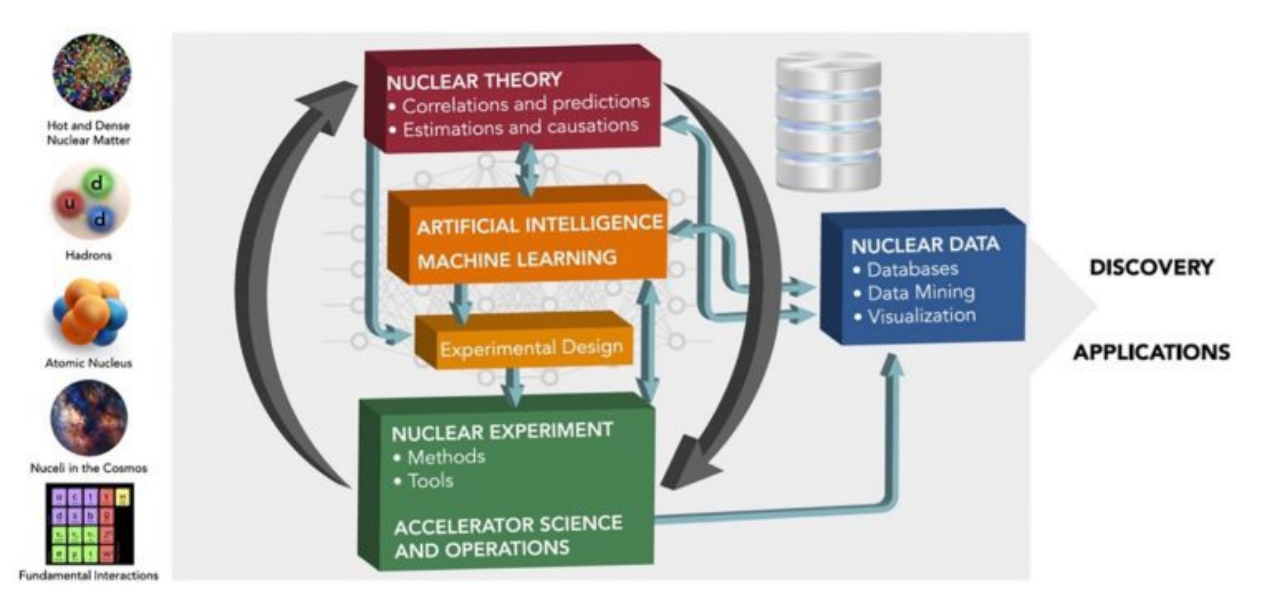
```
# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 5)
        self.fc2 = nn.Linear(5, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Create model and move to GPU
model = SimpleNN().to(device)
# Create optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Using Neural Networks in Python

```
# Training loop
for epoch in range(10):
    # Forward pass
    outputs = model(local_data)
    loss = nn.MSELoss()(outputs, local_targets)

    # Backward pass and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Aggregate gradients across all processes
    for param in model.parameters():
        comm.Allreduce(MPI.IN_PLACE, param.grad.data.numpy(), op=MPI.SUM)
        param.grad.data=torch.from_numpy(param.grad.data.numpy()/size)

    if rank == 0:
        print(f"Epoch {epoch+1}, Loss: {loss.item()}")

# Synchronize final model parameters
for param in model.parameters():
    comm.Bcast(param.data.numpy(), root=0)

if rank == 0:
    print("Training complete")
```

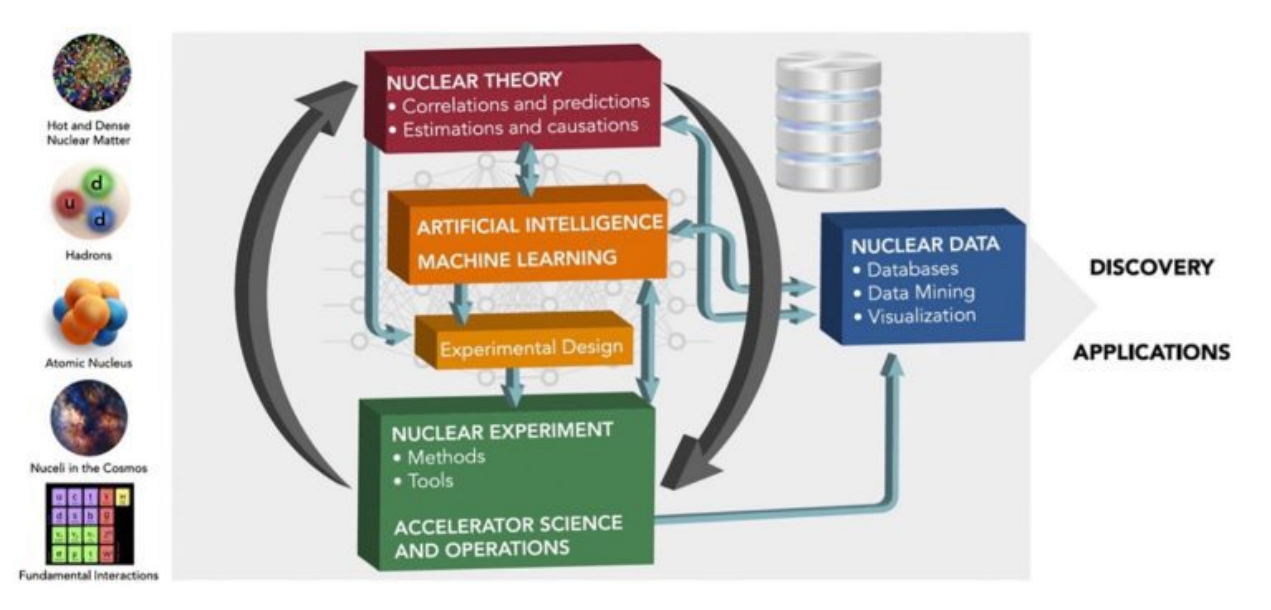
Machine Learning in HEP Research

S. Gleyzer et al, 1807.02876v3 W.-B. He et al, Nuclear Science and Techniques (2023) 34:88



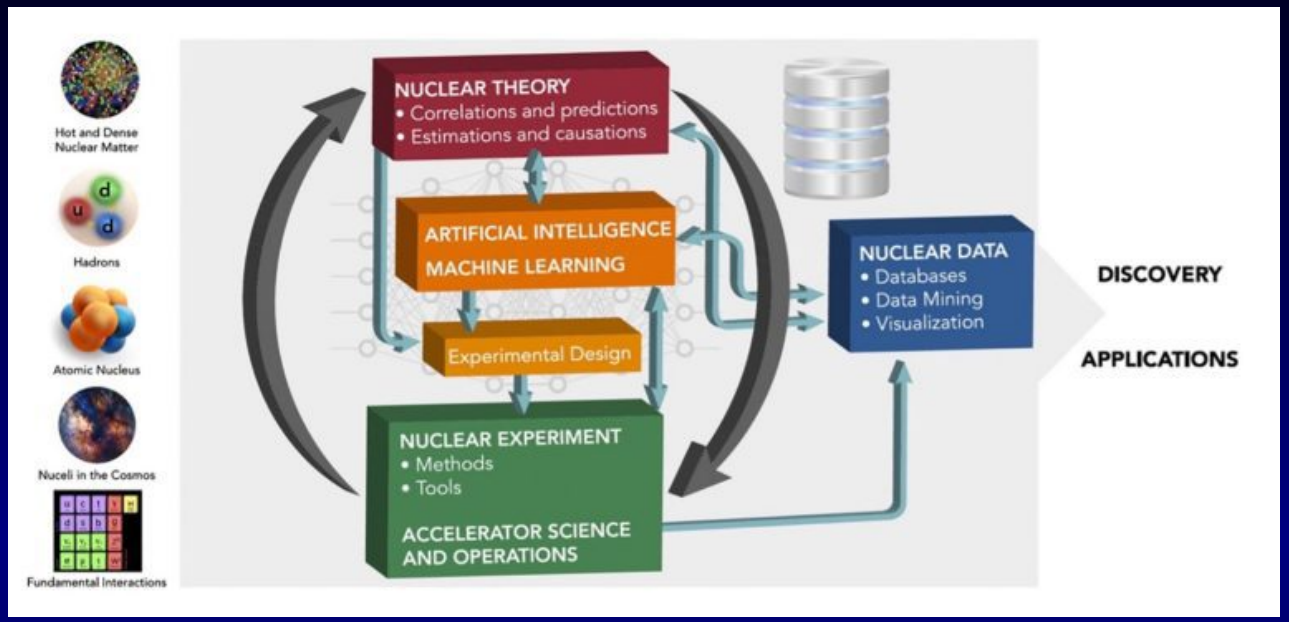
- Theoretical:
  - Simulation:
    ML techniques are employed to improve the accuracy and efficiency of simulations and SM calculations such as matrix elements
  - Object Reconstruction, Identification, and Calibration:
    ML methods enhance the reconstruction and identification of particles and events, as well as the calibration of detectors

Machine Learning in HEP Research



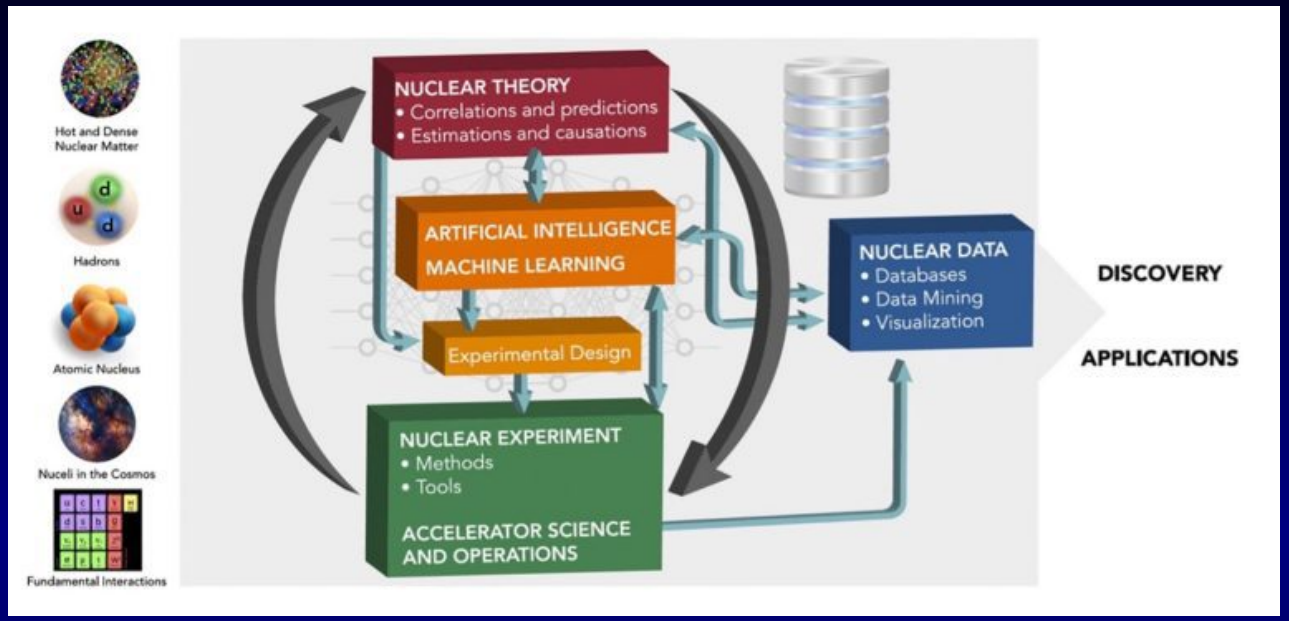
- Theoretical:
  - End-To-End Deep Learning:
    Potential of deep learning approaches to streamline the entire data processing pipeline, from raw data to physics analysis.
  - Theoretical Applications:
    ML is applied to improve model building and hypothesis testing.

Machine Learning in HEP Research



- Experimental:
  - Real-Time Analysis and Triggering:
    Integration of ML for real-time data analysis and event triggering is crucial for handling the large volumes of data generated by particle collisions.
  - Uncertainty Assignment:
    ML plays a role in quantifying uncertainties in measurements and predictions, which is vital for the reliability of experimental results.

Machine Learning in HEP Research



- Experimental:
  - Monitoring of Detectors and Maintenance:
    ML is used for monitoring detector performance, identifying hardware anomalies, and facilitating preemptive maintenance.
  - Computing Resource Optimization:
    The chapter emphasizes the importance of optimizing computing resources and managing workflows to handle the increasing data demands in HEP.

Machine Learning in Nuclear astrophysics

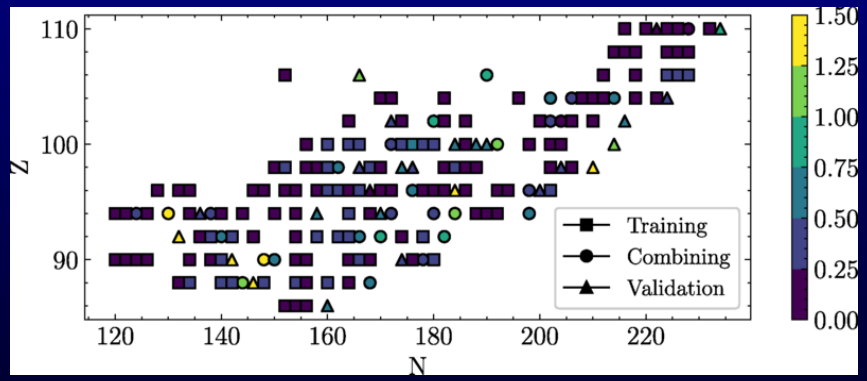
- Nuclear astrophysics
  - Nuclear Mass Predictions

    [e.g. Liquid Drop Model computation: X.-K. Le, Nuclear Physics A Volume 1038 (2023) 122707]
  - Equation of State Reconstruction for Neutron Star

    [e.g. F. Morawski et al, Astronomy&Astrophysics 642, A78 (2020)]
  - Nuclear Reactions in Astrophysical Simulations

    [e.g. MAESTROeX, in later slides]
  - Nuclear Charge Radii Predictions

    [e.g. S Akkoyun et al, J. Phys. G: Nucl. Part. Phys. 40 (2013) 055106]
  - R-Process Nucleosynthesis

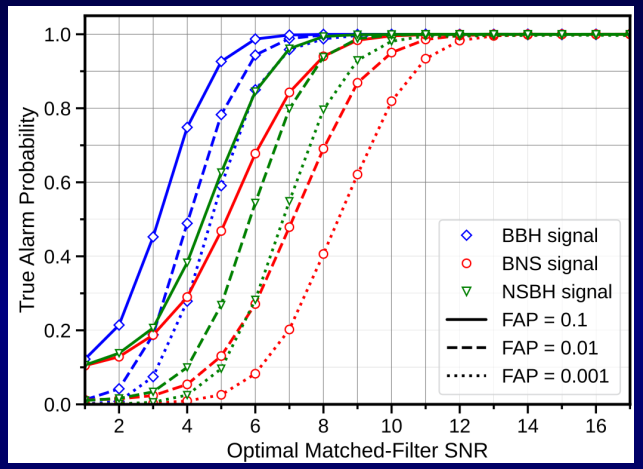


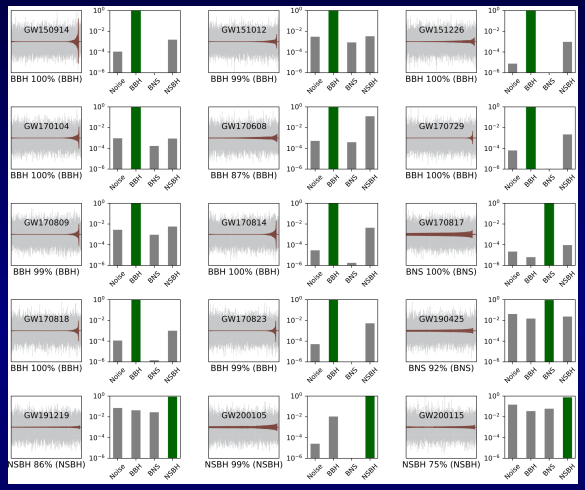
Potential energy and collective inertia can be obtained by expensive Nuclear Density Functional Theory (DFT)

$\Rightarrow$ Use NN to emulate instead [e.g. D. Lay et al, Phys. Rev. C 109, 044305 (2024)]

Machine Learning in Graviational Wave studies

- Graviational wave studies
  - Real-time Detection and Classification of Gravitational Wave signals





[e.g. R. Qiu et al, Physics Letters B Volume 840 (2023) 137850]

  - Alignment sensing and Control of Detectors

[e.g. N. Mukund et al, Physical Review Applied 20 (6), 064041 (2023)]

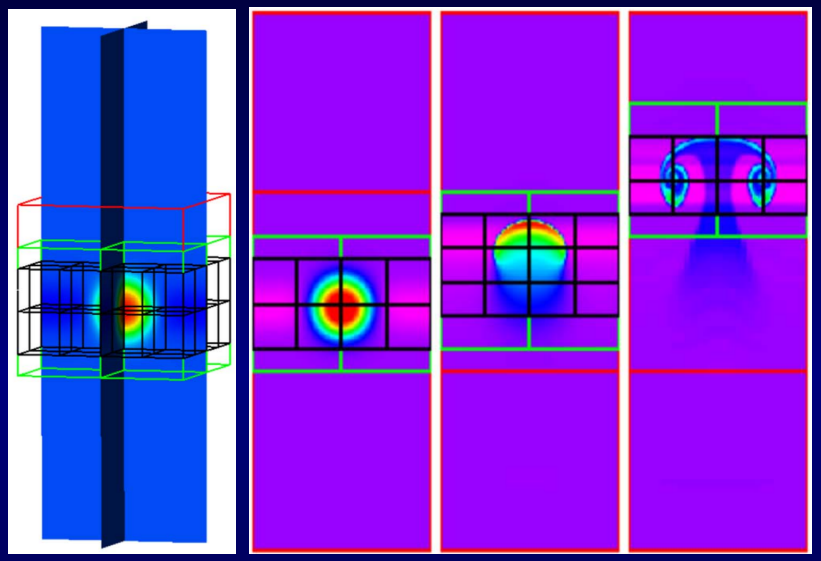
  - Continuous gravitational waves (CWs) are weak, longlasting and nearly-monochromatic waves emitted by nonaxisymmetric spinning neutron stars. Cheap DNN can be applied as a filter for expensive searches to increase sensitivity for the expected weak CWs signals.

[e.g. A. L. Miller et al, Phys. Rev. D 100, 062005]

Neural Networks in MAESTROeX

D. Fan et al, AstroPhysics Journal 887 212 (2019)

D. Fan et al, AstroPhysics Journal 940 134 (2022)

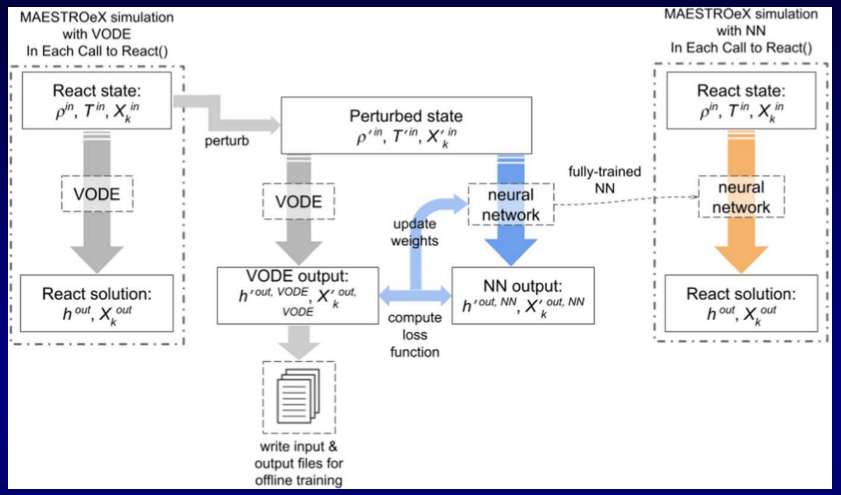


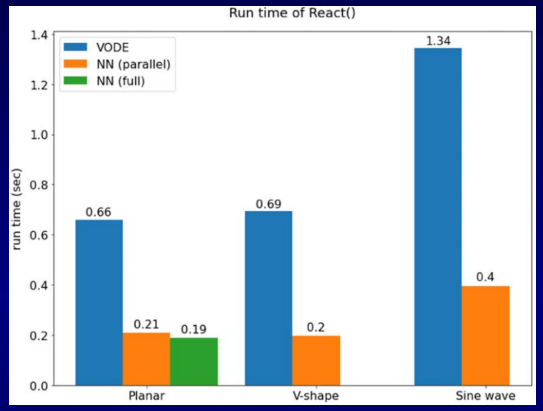
- MAESTROeX: Used for low Mach number astrophysics simulations, based on AMReX
- Suitable for modeling spherical stars as well as planar simulations of dynamics within localized regions of a star

Neural Networks in MAESTROeX

D. Fan et al, AstroPhysics Journal 887 212 (2019)

D. Fan et al, AstroPhysics Journal 940 134 (2022)





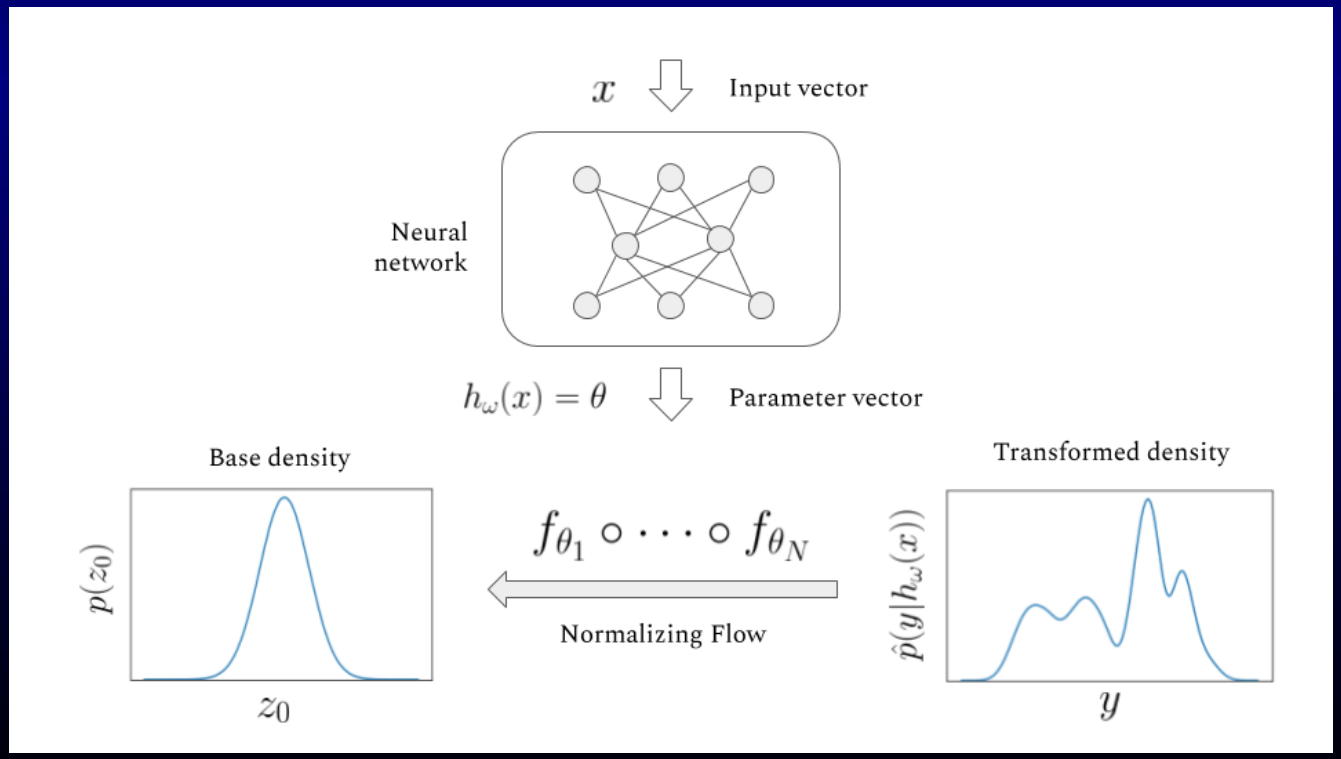
- Accelerate reaction steps in MAESTROeX by replacing stiff ODE integrator (VODE) with trained neural networks
- Inputs: density, temperature, mass fractions
- Outputs: updated mass fractions, nuclear energy generation
- Trained on standard MAESTROeX simulation data

Machine Learning and Lattice QCD

M. S. Albergo et al, Phys. Rev. D 100, 034515 (2019),

https://siboehm.com/articles/19/normalizing-flow-network

- Normalizing Flow:
  Replace HMC with Generative AI for gauge generation
  - Eliminates auto-correlations that persists for observables with long correlation length, e.g. topological charge
  - Uses Neural Network architectures to mimic actions that are very expensive/difficult to simulate



Conclusion

- Part 1: GPU computing
  - What is GPU?
  - GPU Architecture
  - CUDA basics
  - Performance and Optimization
- Part 2: GPU Applications in HEP Research
  - Nuclear Astrophysics:
    - GraCCA: GPU-accelerated N-body simulations
    - AMReX : framework for AMR
  - Nuclear Physics:
    - Lattice QCD
    - Parton Shower calculation
  - Neural Network applications