Machine learning-driven conservative-to-primitive conversion in hybrid piecewise polytropic and tabulated equations of state

Semih Kacmaz[1,2], Roland Haas[1,2] and E. A. Huerta[1,3,4]

[1] Department of Physics, University of Illinois Urbana-Champaign, Urbana, Illinois 61801, USA

[2] National Center for Supercomputing Applications, University of Illinois Urbana-Champaign, Urbana, Illinois 61801, USA

[3] Data Science and Learning Division, Argonne National Laboratory, Lemont, Illinois 60439, USA

[4] Department of Computer Science, The University of Chicago, Chicago, Illinois 60637, USA

E-mail: skacmaz2@illinois.edu

Abstract. We present a novel machine learning (ML) method to accelerate conservative-to-primitive inversion, focusing on hybrid piecewise polytropic and tabulated equations of state. Traditional root-finding techniques are computationally expensive, particularly for large-scale relativistic hydrodynamics simulations. To address this, we employ feedforward neural networks (NNC2PS and NNC2PL), trained in PyTorch and optimized for GPU inference using NVIDIA TensorRT, achieving significant speedups with minimal accuracy loss. The NNC2PS model achieves L_1 and L_∞ errors of 4.54×10^{-7} and 3.44×10^{-6}, respectively, while the NNC2PL model exhibits even lower error values. TensorRT optimization with mixed-precision deployment substantially accelerates performance compared to traditional root-finding methods. Specifically, the mixed-precision TensorRT engine for NNC2PS achieves inference speeds approximately 400 times faster than a traditional single-threaded CPU implementation for a dataset size of 1,000,000 points. Ideal parallelization across an entire compute node in the Delta supercomputer (Dual AMD 64 core 2.45 GHz Milan processors; and 8 NVIDIA A100 GPUs with 40 GB HBM2 RAM and NVLink) predicts a 25-fold speedup for TensorRT over an optimally-parallelized numerical method when processing 8 million data points. Moreover, the ML method exhibits sub-linear scaling with increasing dataset sizes.We release the scientific software developed, enabling further validation and extension of our findings. This work underscores the potential of ML, combined with GPU optimization and model quantization, to accelerate conservative-to-primitive inversion in relativistic hydrodynamics simulations.

1. Introduction

In numerical relativity, accurately modeling astrophysical systems such as neutron star mergers [1–14] relies on solving the equations of relativistic hydrodynamics, which

involve the inversion of conservative-to-primitive (C2P) variable relations [15–17]. This process typically requires computationally expensive root-finding algorithms, such as Newton-Raphson methods, and interpolation of complex, multi-dimensional equations of state (EOS) tables [18,19]. These methods, while robust, incur significant computational costs and can lead to inefficiencies, particularly in large-scale simulations, where up to billions of C2P calls may be required per time step.

In view of these considerations, and taking into account the advent of GPU-based exascale supercomputers such as Aurora and Frontier, and ongoing efforts to port relativistic hydrodynamics software into GPUs [20–22], this work explores the use of machine learning (ML) algorithms that leverage GPU-accelerated computing for C2P conversion. CPU based algorithms for C2P conversion typically involve an iterative non-linear root finder, for which the number of iterations required to achieve a given target accuracy depends on the input data, resulting in different runtimes for different points of numerical grid. This limits the potential to use SIMD (for CPUs) or SIMT (for GPUs) parallelism, reducing the effective rate of conversion achievable using these schemes. A ML approach with its more predictable runtime and regular memory access pattern may help alleviate these issues. Indeed, this work is motivated by recent studies that have explored the potential of ML to replace traditional root-finding approaches for C2P inversion [23]. Specifically, neural networks have shown promise in accelerating the C2P inversion process while maintaining high accuracy [23]. Building on this, the present work introduces a novel approach that leverages ML to accelerate the recovery of primitive variables from conserved variables in relativistic hydrodynamics simulations, with particular focus on hybrid piecewise polytropic and tabulated EOS. These EOS models provide more realistic descriptions of the dense interior of neutron stars, yet their complexity makes the traditional C2P procedure very computationally expensive.

To help address these computational challenges, we present a suite of feedforward neural networks trained to directly map conserved variables to primitive variables, bypassing the need for traditional iterative solvers. In particular, we employ a hybrid approach, utilizing the flexibility of neural networks to handle the challenges posed by complex EOS models. Our models are implemented using modern deep learning tools, such as PyTorch, and optimized for GPU inference with NVIDIA TensorRT [24, 25]. Through comprehensive performance benchmarking, we demonstrate that our approach significantly outperforms traditional numerical methods in terms of speed, particularly when using mixed-precision deployment on modern hardware accelerators like NVIDIA A100 GPUs in the Delta supercomputer.

We evaluate the scalability of our ML models by comparing their inference performance against a single-threaded CPU implementation of a traditional numerical method from the RePrimAnd library [26]. The benchmark was conducted on a Delta supercomputer compute node, featuring dual AMD 64-core 2.45 GHz Milan processors, 8 NVIDIA A100 GPUs (40 GB HBM2 RAM), and NVLink. For dataset sizes ranging from 25,000 to 1,000,000 points, the numerical method exhibited linear scaling of inference time. In contrast, TensorRT-optimized and TorchScript-based neural networks achieved

substantially faster inference, typically demonstrating sub-linear scaling. We investigate two feedforward neural network architectures: a smaller network (NNC2PS) and a larger one (NNC2PL). Notably, mixed-precision TensorRT engines delivered impressive performance, with the NNC2PS engine processing 1,000,000 points in 8.54 ms, compared to 3,490 ms for the numerical method. Ideal parallelization across the entire node (64 CPU cores that support up to 128 threads, and 8 GPUs) suggests a 25-fold speedup for TensorRT over the optimally parallelized numerical method when processing 8 million points. These results demonstrate the scalability and efficiency of our ML-based methods, offering significant improvements for high-throughput numerical relativistic hydrodynamics simulations.

This article is structured as follows. Section 2 introduces the EOS considered in this study, along with the methodologies employed for designing, training, validating, and testing the ML models. In Section 3, we present our key results, including an assessment of the accuracy of the ML models across different model types and quantization schemes. Additionally, we provide a comparison of the computational performance of the ML models relative to traditional root-finding methods. Finally, Section 4 offers a summary of the findings and outlines potential avenues for future research.

2. Methods

We present a ML-based model with the potential to accelerate recovery of primitive variables from conserved variables in general relativistic hydrodynamics (GRHD) simulations, specifically focusing on scenarios employing hybrid piecewise polytropic EOS, and tabulated EOS. As in traditional approaches, this conversion requires inverting the conservative-to-primitive map, a process often reliant on computationally expensive root-finding algorithms. While previous work has demonstrated the success of machine learning for this task with the Γ−law EOS [23], here we investigate its application to hybrid piecewise polytropic EOS, which offers a more realistic representation of neutron star interiors, as well as the tabulated EOS, which incorporate current nuclear physics model of neutron matter. To evaluate the performance of our neural network, we use a traditional CPU-based root-finding algorithm (provided by the RePrimAnd library) as a baseline for comparison. Our aim is to demonstrate the speed advantages of the neural network approach for conservative-to-primitive variable conversion. Our network is implemented using PyTorch, and the inference speed tests are performed using libtorch and NVIDIA TensorRT's C++ API.

In general relativity, the equations of relativistic hydrodynamics can be expressed in a conservation form suitable for numerical implementation. Specifically, in a flat spacetime, they constitute the following first-order, flux-conservative hyperbolic system:

$$\frac{1}{\sqrt{-g}}\left[\frac{\partial \sqrt{\gamma}\,\mathbf{u}}{\partial x^0} + \frac{\partial \sqrt{-g}\,\mathbf{F}^i(\mathbf{u})}{\partial x^i}\right] = \mathbf{0}, \tag{1}$$

where $g = \det(g_{\mu\nu})$ is the metric determinant, and $\gamma = \det(\gamma_{ij})$ is the determinant of the

three-metric induced on each spacelike hypersurface. The state vector of the conserved variables is $\mathbf{u} = (D, \mathbf{S}, \tau)$, and the flux vector is given by

$$\mathbf{F}^i = \left(D \left(v^i - \frac{\beta^i}{\alpha} \right), S_j \left(v^i - \frac{\beta^i}{\alpha} \right) + p\delta^i_j, \tau \left(v^i - \frac{\beta^i}{\alpha} \right) + pv^i \right), \quad (2)$$

where α is the lapse function and β^i the spacelike shift vector: two kinematic variables describing the evolution of spacelike foliations in spacetime as in a typical 3 + 1 (ADM) formulation.

The five quantities satisfying Eq. 1, all measured by an Eulerian observer sitting at a spacelike hypersurface, are the relativistic rest-mass density, D, the three components of the momentum density, S_j, and the energy density relative to the rest mass density, $\tau = E - D$, respectively. These are related to the primitive variables; rest-mass density, ρ, three-velocity, v_i, specific internal energy, ϵ, and pressure, p through

$$\begin{aligned} D &= \rho W, \\ S_j &= \rho h W^2 v_j, \\ \tau &= \rho h W^2 - p - D, \end{aligned} \quad (3)$$

where $W = 1/\sqrt{1 - \gamma_{ij} v^i v^j}$ is the Lorentz factor, and $h = 1 + \epsilon + p/\rho$ is the specific enthalpy.

Incorporating the EOS into the picture provides the thermodynamical information linking the pressure to the fluid's rest-mass density, and internal energy, which combined with the definitions above, closes the system of equations given in Eq. 1 [27–29].

We will first focus on the hybrid piecewise polytropic EOS. The hybrid piecewise polytropic EOS was introduced for simplified simulations of stellar collapse to model the stiffening of the nuclear EOS at nuclear density, and include thermal pressure during the postbounce phase [30]. In gravitational-wave science, it is more commonly used as described in Read et al. [31], where it enables gravitational-wave parameter estimation and waveform modeling by effectively capturing macroscopic neutron star observables with minimal parameters. The structure of this EOS consists of multiple cold polytropes, defined by parameters $K_0, K_1, \cdots, K_{\text{segments}-1}$ and $\Gamma_0, \Gamma_1, \cdots, \Gamma_{\text{nsegments}-1}$, where nsegments denotes the total number of segments. Additionally, it includes a thermal $\Gamma-$law component characterized by Γ_{th}. Continuity of pressure and internal energy across segments, in accordance with the first law of thermodynamics, is ensured after appropriately setting initial values for the polytropic indices, density breakpoints (denoted ρ_{breaks}), and other relevant parameters. In this context, pressure and specific internal energy components in each density interval are given by

$$\begin{aligned} p_{\text{cold}} &= K_i \rho^{\Gamma_i}, \\ \epsilon_{\text{cold}} &= a_i + \frac{K_i}{\Gamma_i - 1} \rho^{\Gamma_i - 1}, \end{aligned} \quad (4)$$

$$p_{\text{th}} = (\Gamma_{\text{th}} - 1)\rho(\epsilon - \epsilon_{\text{cold}}),$$
$$p = p_{\text{th}} + p_{\text{cold}},$$

where a_i is segment-specific constant, and the rest mass density, ρ, is assumed to fall into the segments specified by each of ρ_{breaks}.

In addition to the hybrid piecewise polytropic EOS-based model, we will train a separate network to infer the conservative-to-primitive transformation utilizing the tabulated EOS data. Specifically, we will use the Lattimer-Swesty EOS with a compressibility parameter $K = 220$ (hereafter will be referred to as, LS220 EOS), due to its prevalence and historical significance. Our training dataset is based on a modern, updated version of LS220 EOS constructed and made available by Schneider, Roberts, and Ott in a more recent study [32].

Below, we outline the dataset preparation, model architecture, training process, and methods used in inference speed testing with libtorch and NVIDIA TensorRT to evaluate computational efficiency.

2.1. Data

2.1.1. Piecewise Polytropic EOS Based Model Data We generate a dataset of 500,000 samples using geometrized units where $G = c = M_\odot = 1$. Without loss of generality, we furthermore use a Minkoswki metric $g_{\mu\nu} = \text{diag}(-1, +1, +1, +1)$. The rest-mass density, ρ, is sampled uniformly from $[2\times10^5, 2\times10^{-3}]$, and the fluid's three-velocity is assumed one-dimensional along the x-axis, sampled uniformly from $v_x \in (0, 0.721)$. Following Ref. [31], we use an SLy four-segment piecewise polytropic EOS with segment-wise polytropic indices $\Gamma = [1.3569, 3.0050, 2.9880, 2.8510]$. The first segment's polytropic constant, K_0, is set to 8.9493×10^{-2}. Subsequent polytropic constants, K_i, are determined by enforcing pressure continuity. Similarly, the first segment's constant, a_0, is set to zero, while subsequent a_i values ensure continuity of internal energy. The density breaks for the segments are specified at $\rho = 2.3674 \times 10^{-4}$, 8.1147×10^{-4}, and 1.6191×10^{-3}. The thermal component has an adiabatic index of $\Gamma_{\text{th}} = 5/3$. Additionally, the thermal component of the specific internal energy, ϵ_{th}, is sampled uniformly from $[0, 2]$. A structured dataset is then constructed by converting the primitive variables to conserved variables using the standard relativistic hydrodynamic relations given in Eq. 3. In this dataset, conserved variables serve as input features, and the pressure is the target variable. The resulting dataset is then split into training, validation, and test sets, with each set fully standardized to zero mean and unit variance to ensure equal contribution of all features during neural network training (Fig. 1).

2.1.2. Tabulated EOS Based Model Data To generate the training data for the tabulated EOS-based model, we sample from a provided EOS table and follow a procedure similar to the one described in Section 2.1.1. We begin by reading in the EOS table, which contains the variables electron fraction (Y_e), temperature (T), rest-mass density (ρ), specific internal energy (ϵ), and pressure (p). These quantities are stored

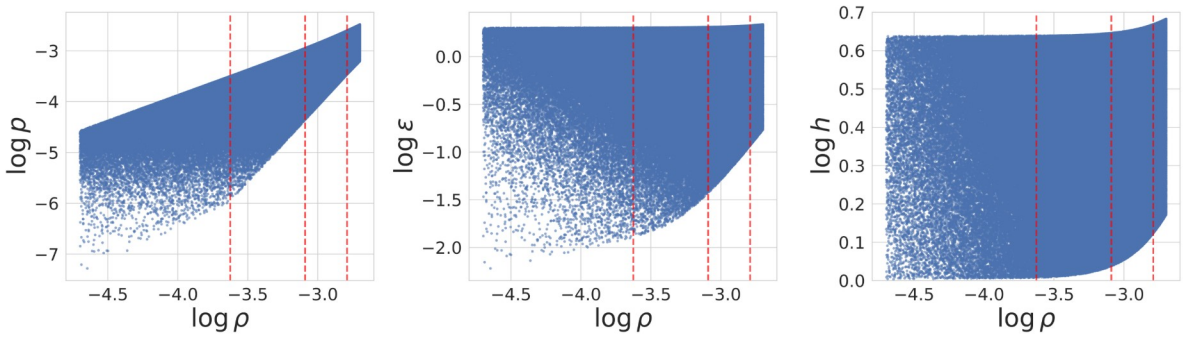

Figure 1. Visualization of the thermodynamic relations based on the complete training data generated for the four-segment piecewise polytropic EOS-based model. From left to right: pressure (p) vs. rest-mass density (ρ), specific internal energy (ϵ) vs. rest-mass density (ρ), and specific enthalpy (h) vs. rest-mass density (ρ). All quantities are plotted on a logarithmic scale. The distinct segments of the piecewise polytropic EOS are delineated by the red vertical lines.

in logarithmic form in the table and are extracted accordingly. For each data point, a random one-dimensional three-velocity, v_x, is sampled from the interval $(0, 0.721)$, and values are randomly chosen for electron fraction, temperature, and density from among the tabulated values. Using these, the corresponding values of ρ, ϵ, and p are then fetched from the EOS table. The primitive variables are then converted into conserved variables using standard relativistic hydrodynamics relations given in Eq. 3. A total of 1,000,000 datapoints are generated using this process[33]. Similar to the hybrid piecewise polytropic EOS-based model, the data is split into training, validation, and test sets, with each set fully standardized to zero mean and unit variance before being used for neural network training.

2.2. Model architecture

2.2.1. Piecewise Polytropic EOS Based Model

For the hybrid piecewise polytropic EOS-based model, we tested two feedforward neural networks of varying complexity to represent the conservative-to-primitive variable transformation. Each network takes as input the three conserved variables (D, S_x, τ) (Eq. 3) and outputs the pressure p (Eq. 4), assuming the remaining momentum density components are zero for simplicity. After experimenting with multiple multi-layer perceptron (MLP) architectures, we identified two models that offered a good balance between accuracy, speed, and trainability. The smaller model, NNC2PS, features two hidden layers with 600 and 200 neurons, while the larger model, NNC2PL, contains five hidden layers with 1024, 512, 256, 128, and 64 neurons (Fig 2).

ReLU activation functions were applied to the hidden layers to introduce nonlinearity, with the output layer kept linear. We found these models strike an effective balance between complexity and performance, making them well-suited for our task.

2.2.2. Tabulated EOS Based Model For the tabulated EOS-based model, we use a single feedforward neural network, NNC2P Tabulated, to achieve an inherently equivalent task with minor differences. This model takes as input the log-scaled variables ($\log D$, $\log S_x$, $\log \tau$, $\log Y_e$) and outputs the log-scaled pressure, $\log p$ (Eq. 4), assuming S_y and S_z are zero for simplicity as before. Using log-scaled inputs and outputs aligns with the format of the tabulated EOS values, which are also stored in logarithmic form to accommodate the typically large values of these physical quantities. This approach reduces the range of feature magnitudes, facilitating more stable learning dynamics and better alignment with the source data.

We explored several MLP architectures, varying in parameters, layers, and training strategies, to identify an optimal design for our task. Among these, a structure resembling NNC2PL, detailed in Section 2.2.1 above, emerged as a robust choice. This architecture effectively balanced capacity and efficiency, enabling accurate learning of log-scaled pressure from tabulated EOS data (Fig. 2).

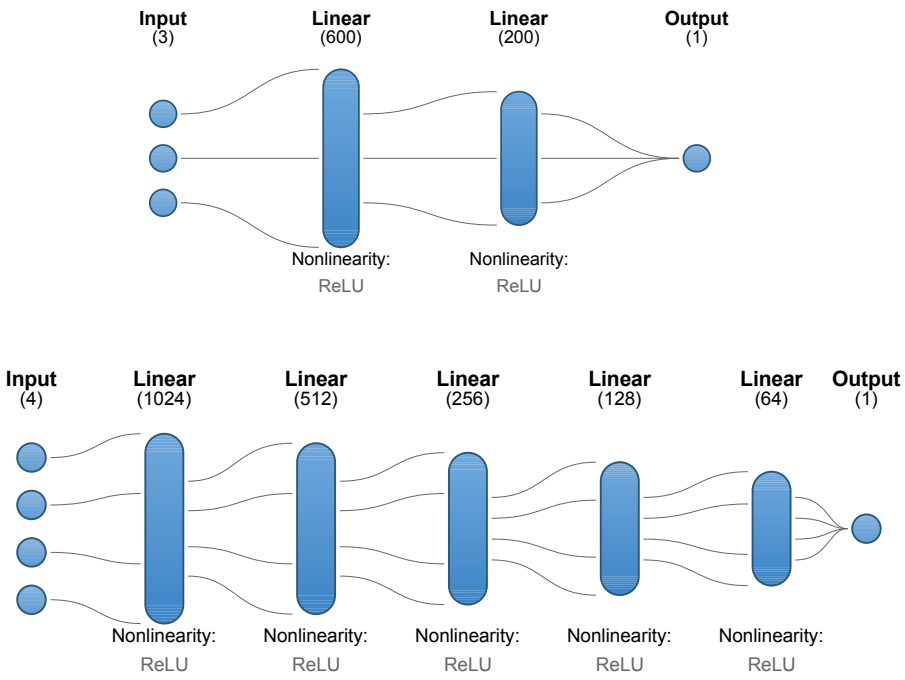

Figure 2. Architectures of the neural networks used for conservative-to-primitive variable mapping. Top: The NNC2PS network takes conserved variables D, S_x, and τ as input and outputs the pressure p. Bottom: The NNC2P Tabulated network uses the logarithm of conserved variables $\log D$, $\log S_x$, and $\log \tau$, along with the electron fraction Y_e, as input, outputting the logarithm of pressure $\log p$. The NNC2PL network (not shown) shares a similar architecture to NNC2P Tabulated but with the same input/output structure as NNC2PS.

2.3. Training approach

We use a similar procedure to optimize all neural networks: NNC2PS, NNC2PL, and the tabulated baseline model, NNC2P Tabulated, with minor tweaks. Training was performed on a single NVIDIA A100 GPU on the Delta cluster. For the hybrid piecewise polytropic EOS-based models (NNC2PS and NNC2PL), we employed a custom, physics-informed loss function that penalizes negative pressure predictions. This loss function is a modified mean-squared error:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i(\theta) - y_i)^2 + q \cdot \sum_{i=1}^{n} \mathrm{ReLU}(-N^{-1}(\hat{y}_i(\theta))) , \quad (5)$$

where $\hat{y}_i(\theta)$ represents the network's estimation for feature i, y_i is the corresponding target value, ReLU is the familiar rectified linear unit defined by $\mathrm{ReLU}(x) = \max(0, x)$, and $N^{-1}(\cdot)$ represents an inverse normalization procedure based on the training data statistics. The penalty factor, q, was optimized for each model, with q = 150 for NNC2PS and q = 350 for NNC2PL. These values consistently suppressed negative pressure predictions on the test set. For the tabulated EOS model (NNC2P Tabulated), the structure of the data precluded negative predictions, so a standard mean-squared error loss function was used.

All models were trained using the Adam optimizer with an initial learning rate of 3×10^{-4}. A learning rate scheduler reduced the learning rate by a factor of 0.5 if the validation loss failed to improve for five consecutive epochs. NNC2PS and NNC2PL were trained for 85 epochs, while NNC2P Tabulated required 250 epochs. For each epoch, the model was set to training mode, and data was loaded in batches of 32 onto the GPU. This batch size was chosen based on experimentation to balance the number of epochs and overall time to convergence. While training with larger batches and multiple GPUs (using PyTorch's DataParallel module or other approaches) is possible, we found no significant advantage regarding the total time to convergence and ultimately opted for this simpler, more portable approach. For each batch, optimizer gradients were reset before generating predictions, and the loss was computed using respective loss functions. Backpropagation was then performed to update the model parameters.

After completing the training phase for each epoch, the model's performance is evaluated on the validation dataset, accumulating the validation loss similarly to the training loss. Both losses are normalized by the size of the respective datasets and stored for further analysis, specifically for clues of potential overtraining.

2.4. Inference Speed Tests

In our inference speed tests, we evaluated two main approaches for efficient deployment: a TorchScript model, and NVIDIA's TensorRT optimized engines. These tests were conducted to measure and compare inference speed under typical deployment conditions, aiming to take advantage of the A100 GPU on Delta.

2.4.1. TorchScript Deployment To prepare models for inference with TorchScript, we first saved a scripted version of the model, which is compatible with PyTorch's JIT compiler, optimizing runtime execution without modifying the model's core structure. TorchScript's scripting provides some degree of optimization, enabling faster model execution than standard PyTorch models but without the hardware-level optimizations that TensorRT offers.

2.4.2. TensorRT Deployment For TensorRT, we explored both FP32 (unquantized) and FP16-quantized engines, ultimately deciding not to pursue INT8 quantization due to accuracy degradation observed in initial tests. After extensive testing, we opted for dynamic engine building with a batch size determined by the total size of the expected dataset, as this approach provided the best balance between performance and flexibility for our hardware and model structure. It must be noted that constructing an optimal engine in TensorRT is a nuanced process, influenced by multiple factors including model architecture, hardware specifications, intended batch sizes during inference, and input data. Therefore, achieving the best results often involves iterative tuning and profiling to adapt the engine to the specific deployment environment and workload requirements. Below, we summarize the overall engine-building process we followed in detail:

- **Model Export to ONNX:** First, we exported the PyTorch model to the ONNX format. This conversion enables interoperability with TensorRT, which uses ONNX as its primary model input format.
- **TensorRT Engine Building:** Using TensorRT's Python API, we constructed both FP32 and FP16 engines. A logger was initialized for verbose logging to capture potential issues during engine building. With the TensorRT Builder, we created a network definition with explicit batch handling, which is essential for dynamic batching configurations.
- **Parsing and Validating the ONNX Model:** We loaded the ONNX model into TensorRT, where the OnnxParser validated and parsed the model. Parsing errors, if any, were logged for troubleshooting, ensuring a valid model structure before optimization.
- **Configuration and Optimization Profiles:** The BuilderConfig was set with a 40 GB workspace memory limit, providing more than enough headroom for dynamic batch sizes while maintaining stable performance. We set up a dynamic optimization profile specifying minimum, optimal, and maximum batch sizes within 10 per cent margin of our typical usage, granting flexibility to handle both smaller and larger input volumes efficiently.
- **Engine Serialization:** Finally, we serialized and saved the engine, creating a portable and optimized binary that can be loaded for deployment. This step encapsulates the model's architecture, weights, and optimizations, ensuring it is ready for fast inference.

Overall, the process of optimizing and saving models using both TorchScript and TensorRT gave us insight into balancing flexibility, accuracy, and performance. For larger batch sizes and greater computational demands, TensorRT's dynamic engine approach in FP16 is often more effective, even for models as simple as ours, while TorchScript remains a reliable fallback and simpler alternative.

For the actual inference speed test procedure, we implemented two distinct workflows on a single GPU for both approaches. The TorchScript-based approach allowed for a straightforward configuration, primarily requiring the definition of batch sizes and the pre-loading of data onto the GPU. It then used libtorch for efficient GPU deployment and batch execution.

In contrast, the TensorRT-based approach demanded several additional configurations. The model, after being converted into an optimized engine, was loaded using TensorRT's C++ API. This included the manual pre-loading of input data into GPU memory before execution, and was followed by manual setup of input and output buffers for TensorRT's executeV2 function, and careful management of CUDA resources. While this setup was more involved, it leveraged hardware-specific optimizations to deliver substantial gains in inference speed.

3. Results

3.1. Accuracy

Table 1 summarizes the accuracy results based on L_1 and L_∞ error metrics for each model variant—NNC2PS, NNC2PL, and NNC2P Tabulated—including both the unquantized and quantized TensorRT engines built from them.

Table 1. Accuracy Results for All Models

Model	L_1 Error	L_∞ Error
NNC2PS (PyTorch)	4.54×10^{-7}	3.44×10^{-6}
NNC2PS (TensorRT)	4.54×10^{-7}	3.43×10^{-6}
NNC2PS (TensorRT–FP16)	6.39×10^{-7}	8.98×10^{-6}
NNC2PL (PyTorch)	2.75×10^{-7}	2.61×10^{-6}
NNC2PL (TensorRT)	2.88×10^{-7}	2.69×10^{-6}
NNC2PL (TensorRT–FP16)	5.32×10^{-7}	9.84×10^{-6}
NNC2P Tabulated (PyTorch)	8.02×10^{-3}	3.54×10^{-1}
NNC2P Tabulated (TensorRT)	8.16×10^{-3}	3.45×10^{-1}
NNC2P Tabulated (TensorRT–FP16)	1.38×10^{-2}	7.44×10^{-1}

The NNC2PS model trained in PyTorch achieves very high accuracy with an L_1 error of 4.54×10^{-7} and an L_∞ error of 3.44×10^{-6}. When the model is converted to a TensorRT engine, the accuracy remains nearly identical, with an L_1 error of 4.54×10^{-7} and an L_∞ error of 3.43×10^{-6}, indicating minimal loss in precision due to TensorRT

optimization. However, when FP16 quantization is applied, the error rates increase to an L_1 error of 6.39×10^{-7} and an L_∞ error of 8.98×10^{-6}, revealing an obvious side-effect of reduced precision.

The larger NNC2PL model, rather expectedly, achieves lower L_1 and L_∞ errors than NNC2PS, with an L_1 error of 2.75×10^{-7} and an L_∞ error of 2.61×10^{-6}. The corresponding TensorRT engine preserves this high level of accuracy, showing only a slight and negligible increase to an L_1 error of 2.88×10^{-7} and L_∞ error of 2.69×10^{-6} respectively. The FP16 quantized version, however, sees a notable rise in error metrics, with an L_1 error of 5.32×10^{-7} and an L_∞ error of 9.84×10^{-6}.

The NNC2P Tabulated model exhibits an L_1 error of 8.02×10^{-3} and an L_∞ error of 3.54×10^{1}. It should be noted that the apparent order-of-magnitude difference in its accuracy does not indicate inferiority or failure of the model; in fact, it is the result of the simple fact that this model is trained on a completely different dataset constructed from the LS220 EOS table to estimate the logarithmic pressure values. The TensorRT engine version also shows only a slight increase in L_1 error to 8.16×10^{-3}. With FP16 quantization, the L_1 error rises, again, more noticeably to 1.38×10^{-2}.

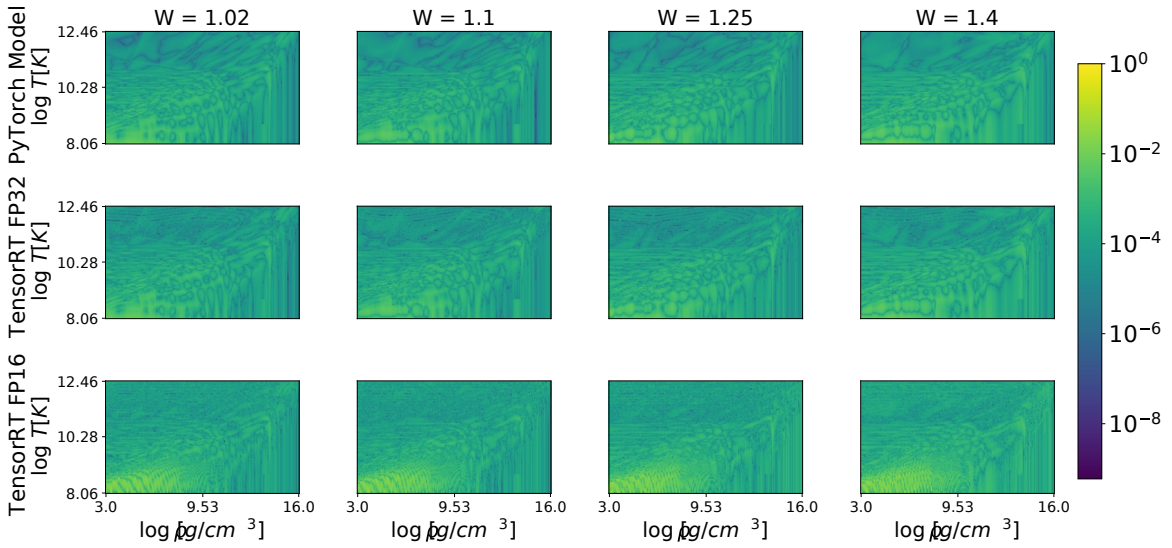

Figure 3. Relative error of the NNC2P Tabulated model for various Lorentz factors (W) with $Y_e \approx 0.1$. The plots highlight the accuracy trends across different regions of the LS220 EOS table, showing larger relative errors in low-density and low-temperature regions, reflecting the inherent complexities of the EOS in this region. This behavior is consistent across the tested W values of 1.02, 1.1, 1.25, and 1.4 and is more pronounced for the FP16 precision TensorRT engine.

Additionally, we examined the relative accuracy of the NNC2P Tabulated model for parameters W = 1.02, 1.1, 1.25, and 1.4 with $Y_e \approx 0.1$ (See Fig. 3). The relative error, defined as the absolute error divided by the true value for each point in a specific parameter set, was not uniform across the parameter space. Larger relative errors were observed in the lowest density and temperature regions of the EOS table, while

slightly smaller errors occurred in the high-temperature regions. This accuracy trend was consistent across all tested Lorentz factor (W) values and even more emphasized for the FP16 precision TensorRT engine. The LS220 EOS, as provided by [32], transitions from detailed treatment at high densities to simplified approximations at lower densities, which may contribute to these disparities. Low-density regions are inherently challenging due to the dominance of thermal effects, non-uniform phase transitions, and the treatment of nuclear matter surfaces, which can exacerbate modeling errors [32,34]. These characteristics likely explain the reduced accuracy in these regions, where variations in the nuclear matter's phase state are more pronounced.

The overall results show that TensorRT's optimizations maintain accuracy across models when using full precision. FP16 quantization, while accelerating inference (as will be discussed further below), introduces higher error rates, particularly in certain models. The potential trade-off between the inference speed and precision can be especially important in relativistic hydrodynamics simulations, where the accuracy of small-scale structures and wave propagation can critically impact the fidelity of predictions. For such simulations, even slight deviations due to quantization can influence results, making full-precision TensorRT inference particularly valuable when accuracy is paramount. Conversely, FP16 quantization may be suitable for faster, lower-fidelity simulations where minor accuracy trade-offs are acceptable.

3.2. Inference Speed Analysis

The inference performance of various methods was evaluated using a single NVIDIA A100 GPU for neural network models and a single-threaded CPU implementation of the traditional numerical method from the RePrimAnd library. The CPUs used in this study were dual AMD 64 core 2.45 GHz Milan processors on the Delta cluster, which can support up to 128 threads. Each configuration was tested across five dataset sizes, ranging from 25,000 to 1,000,000 data points, with ten inference runs conducted per configuration to ensure result stability and consistency.

The numerical method exhibited linear scaling of inference time with respect to the dataset size. In contrast, both TensorRT and TorchScript models generally maintained relatively stable inference times across the dataset sizes. Notably, the full-precision TensorRT engine for the smaller network, NNC2PS, showed a faster-than-expected processing time at certain intermediate dataset sizes, as observed in Fig. 4(a). This behavior may be attributed to favorable thread block utilization and the kernel selection mechanism of TensorRT for this particular network size. A more detailed profiling study is needed to fully elucidate the underlying cause. The accuracy characteristics of these models remained consistent, as indicated in Table 1.

The numerical method required significantly more time than the neural network-based approaches. On average, the numerical method took 103.8 ms to process 25,000 data points, with runtime scaling almost linearly to 3,490 ms for 1,000,000 data points. In contrast, the neural network models demonstrated substantially faster inference

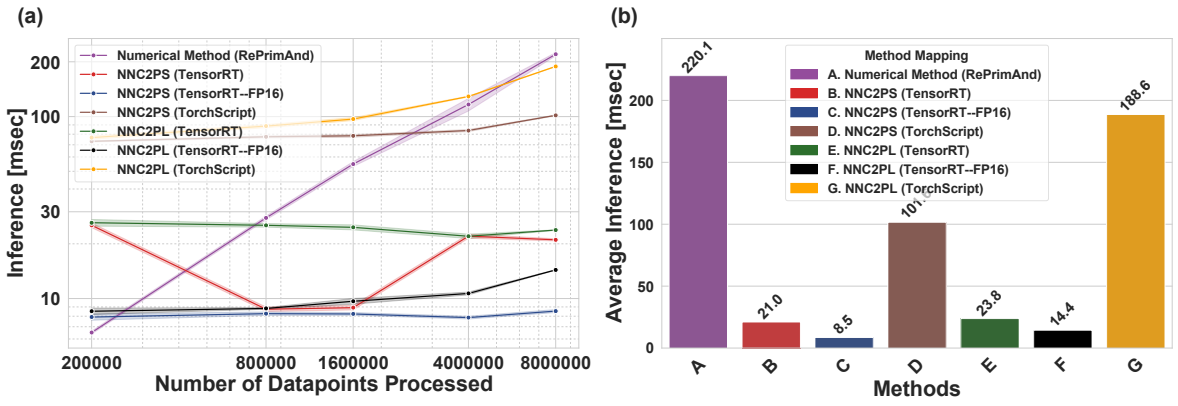

Figure 4. Ideal scaling comparison of various C2P inversion methods under the assumption of perfect parallelization. (a) Projected inference time as a function of dataset size for a traditional numerical solver (RePrimAnd utilizing 128 CPU threads on a single node of the Delta cluster) and two neural network models (NNC2PS and NNC2PL) using TensorRT (FP32 and FP16 precision) and TorchScript across 8 NVIDIA A100 GPUs. (b) Projected inference speed comparison for a dataset of 8 million points, highlighting the significant scalability and efficiency gains achieved by TensorRT engines, particularly with FP16 optimization. The mixed-precision TensorRT engine for NNC2PS achieves approximately a 25-fold reduction in processing time compared to the numerical method, showcasing the potential for TensorRT-based methods to convincingly outperform traditional numerical solvers at scale.

times. Specifically, the mixed-precision TensorRT engine built from NNC2PS required 7.92 ms for 25,000 data points and 8.54 ms for 1,000,000 data points. Its full-precision counterpart exhibited similar performance, with runtimes of 25.17 ms for 25,000 data points and 21.06 ms for 1,000,000 data points. The TorchScript variant showed slower performance, but still maintained sub-linear scaling, with runtimes averaging 72.79 ms for 25,000 points and 101.74 ms for 1,000,000 points.

A similar trend was observed for the NNC2PL models, with TensorRT engines consistently outperforming their TorchScript counterparts. The mixed-precision TensorRT engine for NNC2PL processed 25,000 data points in 8.32 ms and 1,000,000 points in 14.35 ms. In comparison, the full-precision TensorRT engine required 25.85 ms for 25,000 points and 23.87 ms for 1,000,000 points. The TorchScript model averaged 73.18 ms for 25,000 points and 102.04 ms for 1,000,000 points.

Figure 4 presents the ideal scaling achieved under the assumption of perfect parallelization, providing a theoretical performance benchmark. This scenario assumes optimal workload distribution, minimal communication overhead, and negligible synchronization delays, representing the upper bound of scalability. For the numerical method, the figure reflects the full computational capacity of a single CPU node on the Delta cluster, utilizing 128 threads. For the neural networks, it represents the use of 8 A100 GPUs within a single GPU node. Under these ideal conditions, the processing time of the numerical method per data point is projected to decrease by a factor of 128, allowing for the processing of 8 million points in approximately 218

ms (Fig. 4(b)). Similarly, all neural network methods are expected to achieve linear inference scaling with similar per-GPU efficiency. Under this scenario, TensorRT-based methods—particularly the mixed-precision engine for NNC2PS—show a 25-fold reduction in processing time for 8 million points compared to the numerical method running at full capacity on the CPU node. Furthermore, the scaling trend strongly favors TensorRT for even larger datasets.

The results presented above underscore the substantial performance gains achievable through the use of TensorRT-optimized neural networks, particularly in the context of conservative-to-primitive inversion in relativistic hydrodynamics simulations. By leveraging the parallel processing power of modern GPUs, these methods offer significant speedups compared to traditional CPU-based numerical approaches, even in large-scale simulations involving millions of data points. As demonstrated, TensorRT optimizations enable more efficient and scalable solutions, with the potential to dramatically reduce the computational cost of C2P operations. This work highlights the clear advantage of integrating ML-driven methods with GPU acceleration to address the computational challenges of high-throughput simulations. Moving forward, the next step is to incorporate these optimized approaches into full-scale hydrodynamics simulations, where their impact on both performance and scalability can be fully realized.

4. Conclusions

This work introduces a novel ML-driven method for accelerating C2P inversions in relativistic hydrodynamics simulations, with a focus on hybrid piecewise polytropic and tabulated equations of state. By employing feedforward neural networks optimized with TensorRT, we achieve substantial performance improvements over traditional CPU solvers, offering a compelling alternative to computationally expensive iterative methods, while maintaining high accuracy. Our results demonstrate that the TensorRT-optimized neural networks can process large datasets significantly faster, achieving up to 25 times the inference speed of traditional methods.

Future work will explore several key directions to refine and expand this approach. First, adapting the models to handle a broader range of equations of state will improve the versatility of this method across different simulation contexts. Second, exploring alternative network architectures could further enhance both accuracy and inference speed. Additionally, continued optimization of TensorRT, including advanced parallelization strategies and scaling across multiple GPUs, promises even greater reductions in computational time, enabling simulations of larger and more complex astrophysical systems. These improvements will be critical for advancing high-resolution simulations in numerical relativistic hydrodynamics.

We believe that ML-driven methods, particularly those incorporating TensorRT optimization, will play an essential role in advancing the field of general relativistic hydrodynamics and numerical relativity more broadly. To facilitate further validation and extension of these findings, we have made the software developed for this study

publicly available at the following GitHub repository [35].

5. Acknowledgements

E.A.H. gratefully acknowledges support from National Science Foundation award OAC-1931561 and OAC-2209892. R.H. gratefully acknowledges support from Natural Science Foundation award OAC-2103680, OAC-2004879, OAC-2310548, OAC-2005572, and OAC-2411068. This research used the Delta advanced computing and data resource which is supported by the National Science Foundation (award OAC 2005572) through ACCESS-CI [36] allocation PHY160053. Delta is a joint effort of the University of Illinois Urbana-Champaign and its National Center for Supercomputing Applications. This research used the DeltaAI advanced computing and data resource, which is supported by the National Science Foundation (award OAC 2320345) and the State of Illinois. DeltaAI is a joint effort of the University of Illinois Urbana-Champaign and its National Center for Supercomputing Applications. We further acknowledge the use of Matplotlib [37] and Seaborn [38] for the generation of figures in this publication.

References

[1] Radice D, Bernuzzi S and Perego A 2020 *Annual Review of Nuclear and Particle Science* **70** 95–119 ISSN 1545-4134 URL http://dx.doi.org/10.1146/annurev-nucl-013120-114541

[2] Ciolfi R, Kastaun W, Giacomazzo B, Endrizzi A, Siegel D M and Perna R 2017 *Phys. Rev. D* **95** 063016 (*Preprint* 1701.08738)

[3] Kiuchi K 2024 (*Preprint* 2405.10081)

[4] Siegel D M and Metzger B D 2017 *Phys. Rev. Lett.* **119**(23) 231102 URL https://link.aps.org/doi/10.1103/PhysRevLett.119.231102

[5] Sun L, Ruiz M, Shapiro S L and Tsokaros A 2022 *Phys. Rev. D* **105** 104028 (*Preprint* 2202.12901)

[6] Tsokaros A, Ruiz M, Shapiro S L and Ury¯u K 2022 *Phys. Rev. Lett.* **128** 061101 (*Preprint* 2111.00013)

[7] Fernández R, Tchekhovskoy A, Quataert E, Foucart F and Kasen D 2019 *Monthly Notices of the Royal Astronomical Society* **482** 3373–3393

[8] Foucart F, Haas R, Duez M D, O'Connor E, Ott C D, Roberts L, Kidder L E, Lippuner J, Pfeiffer H P and Scheel M A 2016 *Physical Review D* **93** 044019

[9] Camilletti A, Chiesa L, Ricigliano G, Perego A, Lippold L C, Padamata S, Bernuzzi S, Radice D, Logoteta D and Guercilena F M 2022 *Mon. Not. Roy. Astron. Soc.* **516** 4760–4781 (*Preprint* 2204.05336)

[10] Dietrich T, Hinderer T and Samajdar A 2021 *Gen. Rel. Grav.* **53** 27 (*Preprint* 2004.02527)

[11] Agathos M, Meidam J, Del Pozzo W, Li T G F, Tompitak M, Veitch J, Vitale S and Van Den Broeck C 2015 *Phys. Rev. D* **92** 023012 (*Preprint* 1503.05405)

[12] Bauswein A, Baumgarte T W and Janka H T 2013 *Phys. Rev. Lett.* **111**(13) 131101 URL https://link.aps.org/doi/10.1103/PhysRevLett.111.131101

[13] Oertel M, Hempel M, Kl¨ahn T and Typel S 2017 *Rev. Mod. Phys.* **89**(1) 015007 URL https://link.aps.org/doi/10.1103/RevModPhys.89.015007

[14] Alford M G, Schmitt A, Rajagopal K and Sch¨afer T 2008 *Rev. Mod. Phys.* **80**(4) 1455–1515 URL https://link.aps.org/doi/10.1103/RevModPhys.80.1455

[15] Noble S C, Gammie C F, McKinney J C and Del Zanna L 2006 *Astrophys. J.* **641** 626–637 (*Preprint* astro-ph/0512420)

[16] Faber J A and Rasio F A 2012 *Living Reviews in Relativity* **15** 1–83

[17] Duez M D, Liu Y T, Shapiro S L and Stephens B C 2005 *Phys. Rev. D* **72** 024028 (*Preprint* astro-ph/0503420)
[18] Font J A 2000 *Living Reviews in Relativity* **3** 2 (*Preprint* gr-qc/0003101)
[19] Chang P and Etienne Z 2020 *Mon. Not. Roy. Astron. Soc.* **496** 206–214 (*Preprint* 2002.09613)
[20] Kalinani J V *et al.* 2025 *Class. Quant. Grav.* **42** 025016 (*Preprint* 2406.11669)
[21] Zhu H, Fields J, Zappa F, Radice D, Stone J, Rashti A, Cook W, Bernuzzi S and Daszuta B 2024 (*Preprint* 2409.10383)
[22] Liebling S L, Palenzuela C and Lehner L 2020 *Class. Quant. Grav.* **37** 135006 (*Preprint* 2002.07554)
[23] Dieselhorst T, Cook W, Bernuzzi S and Radice D 2021 *Symmetry* **13** 2157 ISSN 2073-8994 (*Preprint* 2109.02679)
[24] Ansel J, Yang E, He H, Gimelshein N, Jain A, Voznesensky M *et al.* 2024 Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)* (ACM) URL https://pytorch.org/assets/pytorch2-2.pdf
[25] NVIDIA Corporation 2024 *NVIDIA TensorRT* high-performance deep learning inference engine URL https://developer.nvidia.com/tensorrt
[26] Kastaun W, Kalinani J V and Ciolfi R 2021 *Physical Review D* **103** 023018 ISSN 2470-0010, 2470-0029 (*Preprint* 2005.01821)
[27] Banyuls F, Font J A, Ib´anez J M, Mart´ı J M and Miralles J A 1997 *The Astrophysical Journal* **476** 221 URL https://dx.doi.org/10.1086/303604
[28] Mart´ı J M and M¨uller E 2003 *Living Reviews in Relativity* **6** 7 ISSN 1433-8351
[29] Font J A 2008 *Living Reviews in Relativity* **11** 7 ISSN 2367-3613, 1433-8351
[30] Janka H T, Zwerger T and Moenchmeyer R 1993 *Astronomy and Astrophysics (ISSN 0004-6361), vol. 268, no. 1, p. 360-368.* **268** 360–368
[31] Read J S, Lackey B D, Owen B J and Friedman J L 2009 *Physical Review D* **79** 124032 ISSN 1550-7998, 1550-2368
[32] Schneider A S, Roberts L F and Ott C D 2017 *Physical Review C* **96** 065802 ISSN 2469-9985, 2469-9993
[33] Wouters T 2024 *Machine learning algorithms for the conservative-to-primitive conversion in relativistic hydrodynamics* M.S. thesis KU Leuven
[34] Bernuzzi S, Breschi M, Daszuta B, Endrizzi A, Logoteta D, Nedora V, Perego A, Schianchi F, Radice D, Zappa F, Bombaci I and Ortiz N 2020 *Monthly Notices of the Royal Astronomical Society* **497** 1488–1507 ISSN 0035-8711, 1365-2966 (*Preprint* 2003.06015)
[35] Semih Kacmaz 2024 C2PNets:Machine Learning for Conservative-to-Primitive Variable Recovery in Numerical Relativity accessed: 2024-12-09 URL https://github.com/semihkacmaz/C2PNets
[36] Boerner T J, Deems S, Furlani T R, Knuth S L and Towns J 2023 Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services & support *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good* PEARC '23 (New York, NY, USA: Association for Computing Machinery) p 173–176 ISBN 9781450399852 URL https://doi.org/10.1145/3569951.3597559
[37] Hunter J D 2007 *Computing in Science & Engineering* **9** 90–95
[38] Waskom M L 2021 *Journal of Open Source Software* **6** 3021 URL https://doi.org/10.21105/joss.03021

Code availability

The code needed to reproduce the results presented in this manuscript is available at the following GitHub repository [35].