2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10070935

eNODE: Energy-Efficient and Low-Latency Edge Inference and Training of Neural ODEs

Junkang Zhu[§]
jkzhu@umich.edu
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA

Yaoyu Tao[§]
taoyaoyu@umich.edu
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA

Zhengya Zhang
zhengya@umich.edu
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA

***Abstract*—Neural ordinary differential equations (NODEs) provide better modeling performance with smaller amount of model parameters in many tasks by embedding neural networks (NNs) in ordinary differential equations (ODEs). They have been shown to outperform in representing continuous-time data and learning dynamic systems, and are promising for on-device inference and training. However, an edge device is limited by area and energy budget, and real-time operations have a tight latency requirement. State-of-the-art NN accelerators are not optimized for the area- and power-hungry memory storage and access for NODE inference and training, and lack the flexibility to incorporate dynamic latency reduction techniques. We present eNODE by architecture-algorithm co-design to achieve efficient and fast inference and training of NODEs. eNODE adopts compact-size depth-first integration and depth-first training for higher energy efficiency. Through function reuse, packetized processing and a unified NN core design, the efficiency of eNODE's depth-first processing is further enhanced. We propose algorithm innovations, including slope-adaptive stepsize search and priority processing with early stop, to substantially shorten the latency. A hardware prototype is synthesized in a 28 nm CMOS technology for evaluation and benchmarking. eNODE demonstrates up to 6.59× better energy efficiency, 2.38× higher speed, and better area scalability over a SIMD ASIC baseline.**

I. Introduction

Residual neural networks (ResNets) [14], [15] and their variants [4], [16], [31], [32] have shown superior performance in computer vision (CV), natural language processing and many other tasks by enabling efficient training of very deep neural networks (DNNs) [2], [12], [19], [20], [27], [28]. However, state-of-the-art ResNets and networks with residual structures such as MobileNetV2 [26] feature a discrete layered structure and lack the modeling capability for continuous-time dynamic systems using moderate model size [5].

Continuous-time dynamic systems are often modeled mathematically by ordinary differential equations (ODEs). An ODE is usually formulated based on a physical model f, and solved by a numerical integrator over many time steps. With the emerging use of neural network (NN) as a universal function approximator, an ODE can be formulated based on an NN model f trained based on the data collected by the dynamic system. Similarly, the NN-based ODE can be solved by a numerical integrator.

[§]These authors contributed equally to this work.

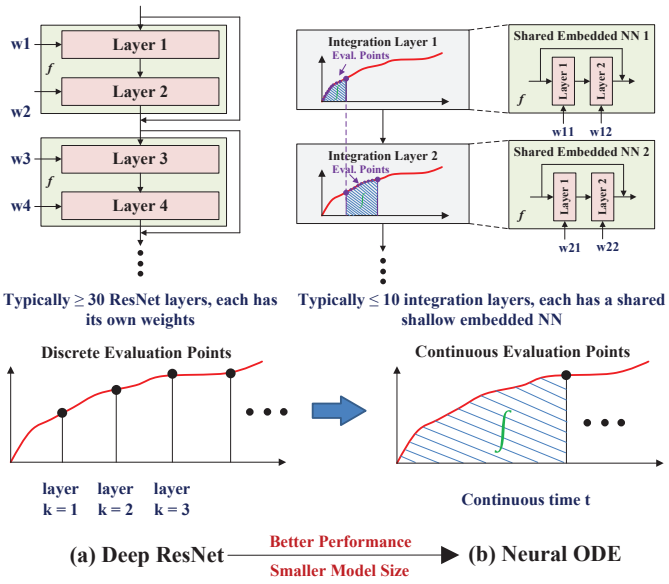


Fig. 1. (a) A deep residual network defines a discrete sequence of finite transformations. (b) A neural ODE network defines continuous transformations of the state.

Interestingly, a residual block in ResNet [14] can serve as a numerical integrator. A residual block consists of an embedded NN f with a skip connection that resembles a one-step forward Euler integrator as shown in Fig. 1(a): $h(t+\Delta t) = h(t)+f\Delta t$, where $h(t)$ and $h(t+\Delta t)$ represent the input and the output of the residual block, respectively, and Δt represents the time step. A residual block is however a first-order numerical integrator, and a ResNet has a fixed number of layers with a fixed time step, limiting its ability in accurately modeling continuous-time dynamic systems that require variable number of layers and adaptive time step.

The performance for such systems can be greatly enhanced by allowing more advanced numerical integrators, variable number of layers and adaptive time step in the form of neural ordinary differential equation (NODE) [5] as shown in Fig. 1(b). Many NODE variants emerged soon after NODE was invented [7], [9], [21], [30], [33], [34]. They are subsequently generalized as a class of continuous-in-depth NNs, or liquid NNs, [13], [24], where more complicated numerical integrators like Runge-Kutta (RK) [25] are used to achieve

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

stronger modeling smoothness and better performance.

A state-of-the-art NODE typically consists of a stack of integration layers, similar to the layers in ResNet, but each integration layer is subdivided into a set of evaluation points at a suitable Δt time step, as shown in Fig. 2(a). At each evaluation point, a high-order RK integrator is called to perform integration. To sum up, NODE introduces evaluation points per integration layer at variable time steps and adopts high-order integrators, providing better adaptivity and accuracy, but costing one or two orders of magnitude higher latency and complexity than conventional DNNs.

In continuous-time dynamic system modeling and control, new data is collected continuously, requiring low-latency inference and quick updates to adapt to runtime changes. A cloud-based solution is appropriate for handling the complexity of NODE inference and training, but the latency of transferring large amounts of data to the cloud can be significant, rendering the cloud approach impractical. Moving NODE inference and training to the edge, i.e., on robots, drones, probes, etc., provides low-latency data access, but an edge computing platform is constrained in size and energy, posing significant challenges to meet the latency requirement. In this work, we present eNODE, a solution based on architecture-algorithm co-design to bring energy-efficient and low-latency NODE inference and training to the edge. We summarize the contributions of this work as follows:

Depth-first integration and depth-first training: the depth-first integration computes on the fly and extends the pipeline of a high-order integrator, minimizing the local memory size and external memory access in every elementary step of NODE inference and training. The depth-first training further extends pipelining to reduce layer activation and gradient memory usage. The aggressive memory reduction contributes to a smaller size and a better energy efficiency.

Function reuse and packetized processing: a ring architecture is designed to reuse function f in a high-order integrator for energy-efficient processing in a compact size. Forward and backward passes are supported by looping in opposite directions along the ring. The depth-first integration is packetized to ensure fully pipelining without stalling.

Unified NN core: a spatial array architecture is designed to support convolution in both forward and backward directions used by NODE inference and training. The design adopts a processing element (PE) grouping to allow reuse of the hardware during both forward and backward passes.

Expedited stepsize adjustments: two algorithmic techniques are proposed to expedite the search of optimal stepsizes: 1) slope-adaptive stepsize search uses slope history as basis to predict the optimal integration stepsize; and 2) priority processing and early stop prioritizes the processing of sections of an input during a stepsize search to shorten the latency.

To the best of our knowledge, eNODE is the first hardware architecture that supports all NODE features. An eNODE prototype is implemented in RTL and synthesized in a 28nm CMOS technology. The prototype demonstrates up to 2.38× improvement in speed and 6.59× improvement in energy efficiency over a SIMD ASIC baseline. Compared to an Nvidia A100 deep learning GPU, eNODE provides 55× better energy efficiency in training.

II. Background

NODE models a dynamic system using a series of first-order ODEs in the form (1).

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = f(t, h(t), \theta),\ h(0) = x,\ t \in [0, T], \tag{1}$$

where $h(t)$ is the system state at time t, and f is a shallow NN called **embedded NN** with trainable parameters θ, i.e., weights in convolution layers. In the following, we will omit θ and simply use $f(t, h(t))$ in describing inference.

A. NODE Inference

NODE inference involves solving initial value problems (IVPs) to a series of ODEs like (1). For each ODE, given an initial system state $h(0)$, we compute $h(T)$, the system state at time T as in (2). This is called one **integration layer**.

$$h(T) = h(0) + \int_0^T f(t, h(t))dt. \tag{2}$$

Numerical integration can be used to solve (2) over many **evaluation points** spread over $t \in [0, T]$ with a stepsize Δt. A numerical integrator is called at each evaluation point, as illustrated in Fig. 2(a). The simplest integrator is an Euler integrator (3), shown in Fig. 2(b).

$$h(t + \Delta t) = h(t) + f(t, h(t))\Delta t. \tag{3}$$

A higher-order numerical integrator, such as the midpoint integrator or the RK23 integrator illustrated in Fig. 2(b) can be used to reduce the numerical integration error. Taking RK23 as an example, it establishes a series of intermediate **integral states** between the time interval $[t, t + \Delta t]$. These are named k_1, k_2, k_3, k_4 in Fig. 2(b) and (c). The finer subdivision into intermediate states enables better approximations of the slope in each time step, but also increases the complexity as obtaining each intermediate state requires an inference of the embedded NN f. Readers can refer to [8] for other commonly used RK integrators.

In short, NODE inference is done over a series of integration layers, one per ODE that covers a certain time period. Each integration layer is divided into evaluation points, where a high-order integrator is called to perform fine-grained integration. A high-order integration requires multiple inferences of the embedded NN f.

B. Stepsize Search for Numerical Integration

Other than the integration method and its order, the **time stepsize** Δt between evaluation points determines the accuracy of a numerical integrator. An adaptive integrator uses an iterative stepsize search to find the optimal stepsize to achieve the target accuracy while keeping the latency in check. An estimated integration error e is computed after each search to guide the adjustment.

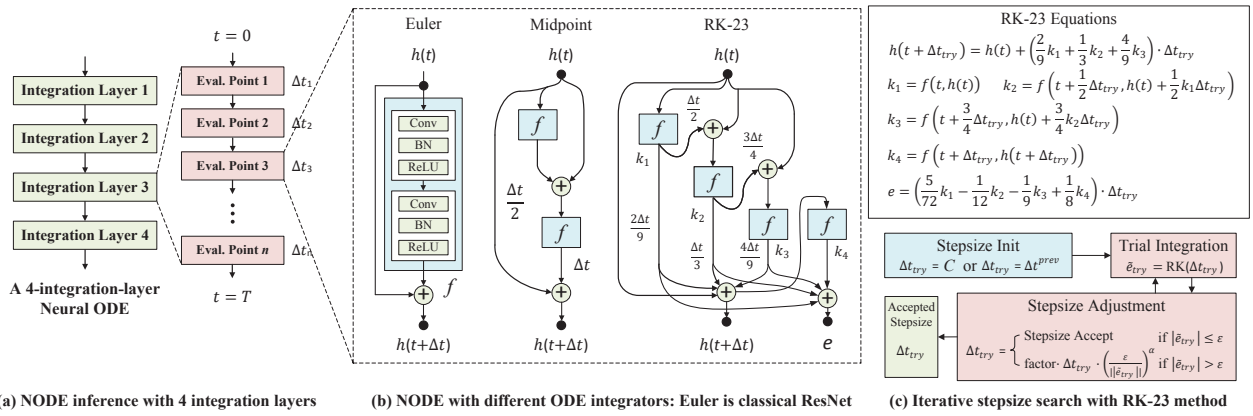


Fig. 2. NODE inference with RK23 integrator and iterative stepsize search.

We use the RK23 integrator as an example to describe how an adaptive integrator works. Its description is shown in Fig. 2(c). Using an iterative stepsize search algorithm [23], we perform a trial integration with an initial stepsize Δt_{try} based on a pre-defined value C or the stepsize used in the previous evaluation point Δt^{prev}; compute the truncation error $\tilde{e}_{try}$; and decide whether to accept or adjust Δt_{try}. We repeat the above steps until we find an acceptable stepsize to keep the truncation error within a given tolerance ϵ.

A NODE inference pipeline, also called a forward pass, is illustrated in Fig. 3, for one integration layer in an N-integration-layer NODE (N is variable). A forward pass of an integration layer consists of n_{eval} evaluation points. At each evaluation point, the stepsize search is done over n_{try} times. Each time it requires an integration trial using a high-order RK integrator that evaluates f for s times to produce s integral states. Therefore the compute complexity of a NODE inference or a forward pass is $O(N \times n_{eval} \times n_{try} \times s)$, compared to $O(N)$ for a conventional DNN, posing challenges in energy consumption and latency.

C. NODE Training

NODE training includes a forward pass and a backward pass [5], [9], [33] as shown in Fig. 3. The forward pass is described above. In the backward pass, a function called **adjoint** $a(t)$ is first computed that represents the loss gradients with respect to the hidden state, or $a(t) = \frac{\partial L}{\partial h(t)}$, where L is the loss function. For each layer, $a(t)$ is solved as an IVP to the ODE (4) [5] by numerical integration backward in time from T to 0 for each iteration layer.

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(h(t), t, \theta)}{\partial h(t)}, \; a(T) = \frac{\partial L}{\partial h(T)}. \quad (4)$$

The parameter gradients are then calculated using the adjoint by an integral described by (5) [5], and finally the model parameters are updated.

$$\frac{dL}{d\theta} = -\int_T^0 a(t)^T \frac{\partial f(h(t), t, \theta)}{\partial \theta} dt. \quad (5)$$

A backward pass requires large memory to store intermediate **training states** including the evaluation points and the integral states at each evaluation point. The adaptive-checkpoint-adjoint (ACA) method [33] reduces the memory size while retaining the training accuracy. The ACA method stores only evaluation points as checkpoints in each layer and uses local forward steps to compute the intermediate states. In particular, a backward pass repeats the following set of steps:

1) *Local forward step*: A forward integration is done from a checkpoint t_i to the next checkpoint t_{i+1} to recover the intermediate states.
2) *Adjoint calculation*: With all the intermediate states from t_i to t_{i+1}, the adjoint $a(t)$ is computed backward from t_{i+1} to t_i via numerical integration to solve (4).
3) *Parameter gradients calculation*: The parameter gradients $dL/d\theta$ is calculated from t_{i+1} to t_i via numerical integration following (5).

An ACA backward pass pipeline is illustrated in Fig. 3 for one integration layer. The compute complexity of an ACA backward pass is $O(N \times n_{eval} \times s)$ compared to $O(N)$ for back propagation in a conventional DNN. Note that a backward pass adopts the stepsizes obtained in the preceding forward pass, without needing stepsize search.

D. Performance and Memory Usage Profiling

In this study, we use a 4-integration-layer NODE with a RK23 integrator (similar to the setup in [33], [34]) as shown in Fig. 2(a). The NODE was trained with the classic CIFAR-10 dataset [18] using the ACA method [33]. The numerical integration error tolerance is set to $\epsilon = 10^{-6}$.

The NODE forward pass (inference) and backward pass are profiled using an Nvidia A100 GPU on AWS. The average latency breakdown for a training iteration is shown in Fig. 4(a). We can see that the forward pass taking a large proportion of the total latency. This is attributed to the long latency in the iterative stepsize search. For applications that require a high integration accuracy, the iterative stepsize search in the forward pass is the major performance bottleneck, accounting for 87% of the total latency as shown in Fig. 4(a).

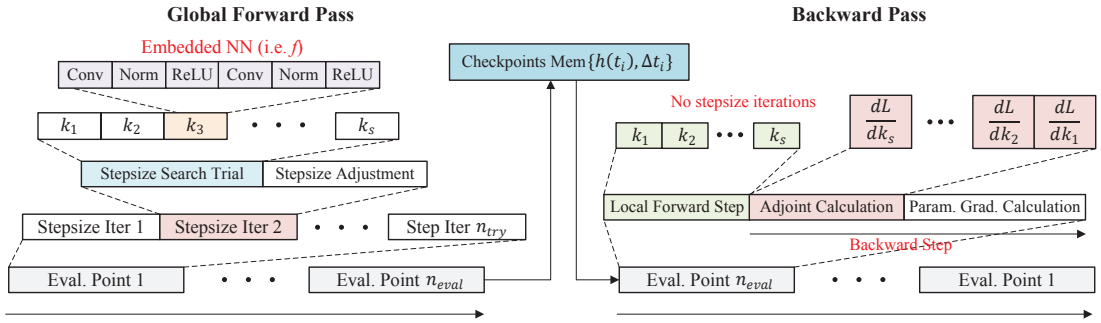


Fig. 3. Theoretical runtime analysis for a NODE integration layer.

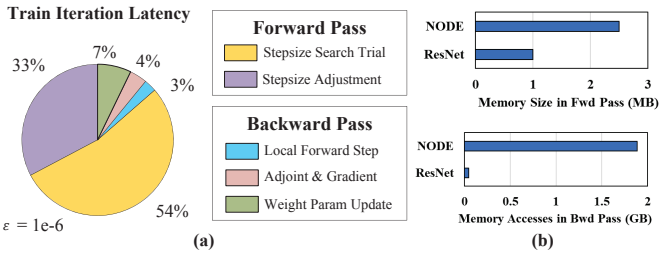


Fig. 4. (a) Runtime breakdown of training a 4-integration-layer NODE on CIFAR-10 dataset on an Nvidia A100 GPU. (b) Memory profile of ResNet-100 and a 4-integration-layer NODE on CIFAR-10 dataset.

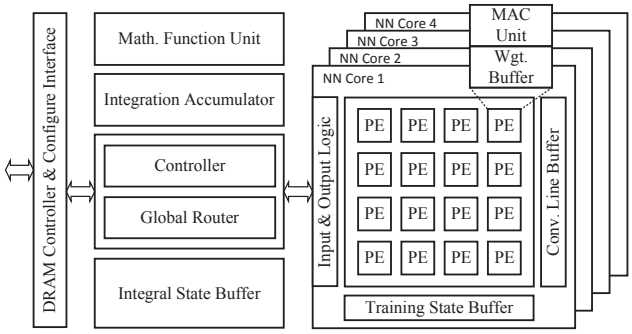


Fig. 5. eNODE high-level architecture.

The memory usage profile is shown in Fig. 4(b) for NODE inference (forward pass) and training (forward and backward pass). The usage is compared to ResNet-100. Both NODE inference and training cost more memory than ResNet, but the difference in memory usage is much more pronounced in training: NODE inference costs 2.5× more memory size than ResNet, while NODE training costs 41.5× more memory access than ResNet. The large difference is attributed to the large increase of intermediate states that need to be stored in and accessed from memory.

E. Design Goals and Applicability of NN Accelerators

The profiling above highlights the importance of designing a NODE accelerator to minimize the high latency and high memory usage. State-of-the-art NN accelerators can be used for both NODE inference and training, but the inference energy and latency are expected to be at least two to three orders of magnitude higher than NN inference, and the memory usage and external memory access for training are expected to be at least one order of magnitude larger. Given the higher compute complexity and memory access, it becomes impractical to deploy an NN accelerator for NODE operations in real time. In the following, we present our architecture-algorithm co-design approach to address the energy and latency challenge for NODE inference and training.

III. eNODE High-Level Architecture

A diagram of the eNODE high-level architecture is illustrated in Fig. 5. The architecture consists of instances of NN cores. Each NN core contains a PE array and local buffers. The cores share a global buffer, as well as an external DRAM and auxiliary function units through a controller and a global router. The whole system can be programmed by a configure interface. The weight mapping, dataflow configuration and memory allocation are determined at compile time based on the hardware and the algorithm specification of the target NODE. The system is fully configured before execution.

The following sections elaborate on each part of the eNODE architecture. In Section IV, we present depth-first integration, which forms the basis of eNODE. Section V describes an efficient mapping of forward and backward passes on the eNODE architecture. Section VI details an unified NN core design to support both forward and backward passes. In Section VII, we present algorithmic innovations to further improve the system latency and energy efficiency. Finally, we present benchmark evaluations in Section VIII.

IV. Depth-First Integration

NODE inference and training require a large amount of memory due to the explosion of intermediate integral states and training states. These intermediate states are typically stored in a two-level memory hierarchy: an external DRAM and a smaller on-chip SRAM cache. Reducing access to the external DRAM while keeping a compact on-chip SRAM is essential for reducing the energy consumption and making a compact design.

NODE is made of integration layers, which are processed by high-order numerical integrators. Inspired by the depth-first NN processing [1], [10], [11], we propose transforming a high-order integrator's compute graph to reduce external DRAM access and minimize the on-chip SRAM size.

A. Formulation of Depth-First Integration

Using the RK23 integrator as an example: assume the integration is done in a conventional layer-by-layer mechanism, the data dependency requires buffering the initial state $h(t)$ and all integral states k_1 to k_4. These buffered states need to be kept until we finish computing the next state $h(t+\Delta t)$ and the error state e. This translates to a larger on-chip memory or more external memory access. To tackle this problem, we propose depth-first integrator following two criteria: 1) the processing steps are fully pipelined so that the outputs from one step are streamed to the next step for *immediate* processing to reduce buffer size, and 2) the processing is ordered such that once an input is available, all the downstream processing steps using this input are done *in parallel* to reduce the buffering time window. The approach allows us to minimize the size of on-chip memory and off-chip memory access.

The data dependency graph (DDG) of a depth-first RK23 integrator is illustrated in Fig. 6(a). The nodes of the DDG are the states, including the initial state $h(t)$, the final state $h(t+\Delta t)$, the error state e and the integral states k_1 to k_4. We factor out the partial states in computing the integral states, namely $p_{i,j}$, $i \in \{2,3,4\}$, $j < i$, and the partial states in computing the error state, namely e_i, $i \in \{1,2,3\}$. The integration is scheduled by connecting these nodes following the order shown in Fig. 6(a): 1) the first-order integral state k_1 is computed by applying f on $h(t)$; 2) k_1 is scaled by RK coefficients and is accumulated with $h(t)$ to obtain the low-order partial states $p_{i,1}$, $i \in \{2,3,4\}$; 3) the low-order integral states k_j are scaled and are accumulated with the low-order partial states $p_{i,j-1}$, $j < i$, to obtain the higher-order partial states $p_{i,j}$; 4) the integral states k_2, k_3 and k_4 are computed by applying f to partial states $p_{2,1}$, $p_{3,2}$ and $p_{4,3}$; 5) the partial error states e_i are computed when the associated integral states are available.

The partial states factoring and the ordering of the depth-first integration allow an input to trigger all dependent processing in parallel to consume that input. The data dependency does not cross one stage in the DDG, indicating that the buffered data can be purged or overwritten right after consumption. Fig. 6(b) explains the operation of the depth-first integration. To simplify the illustration, we assume f is made of one 3×3 convolution layer (recall that f is usually made of a shallow set of NN layers). Referring to Fig. 6(b), when a new input pixel ① in $h(t)$ arrives, it is immediately convolved with the 3×3 kernel to produce a partial sum (psum) patch to update the patch ② in k_1. After the update, the upper-left pixel ③ in the patch is finalized as the input to trigger all downstream processing that requires the pixel. After proper scaling and accumulation with corresponding pixel ④ in $h(t)$, the downstream pixels ⑤ in $p_{2,1}$, ⑥ in $p_{3,1}$, ⑦ in $p_{4,1}$, and ⑧ in e_1 are computed. In the following, ③ and ④ are no longer needed and retired from the buffer. The newly generated ⑤, ⑥, ⑦ and ⑧ will move forward in computing new pixels in k_2, $p_{3,2}$, $p_{4,2}$ and e_2, respectively. The processing is ordered such that an input is consumed by all downstream processing that requires the input, so that the input can be quickly retired from the buffer. The processing is fully pipelined and stalling is eliminated by a packetized processing architecture to be presented in Section V.B.

The depth-first integrator cuts the buffer usage significantly. The buffer usage for each state and partial state depends on the lifetime of the data, defined as the period from when it is first produced to when it is last used. For instance, the integral states, k_1, k_2, k_3 and k_4, in the form of partial states, require one row of buffer for each partial state as seen in Fig. 6(b), due to a one-line lag between the production and consumption of partial state pixels; the convolution layers each requires two rows of buffer around the convolution window, which depends on the size of the convolution kernel (3×3); and the buffering for the rest is minimal. Altogether, the buffer size is reduced from 5 full feature maps to store $h(t)$, k_1, k_2, k_3 and k_4 (320 rows for 64×64 feature maps) to only 15 rows.

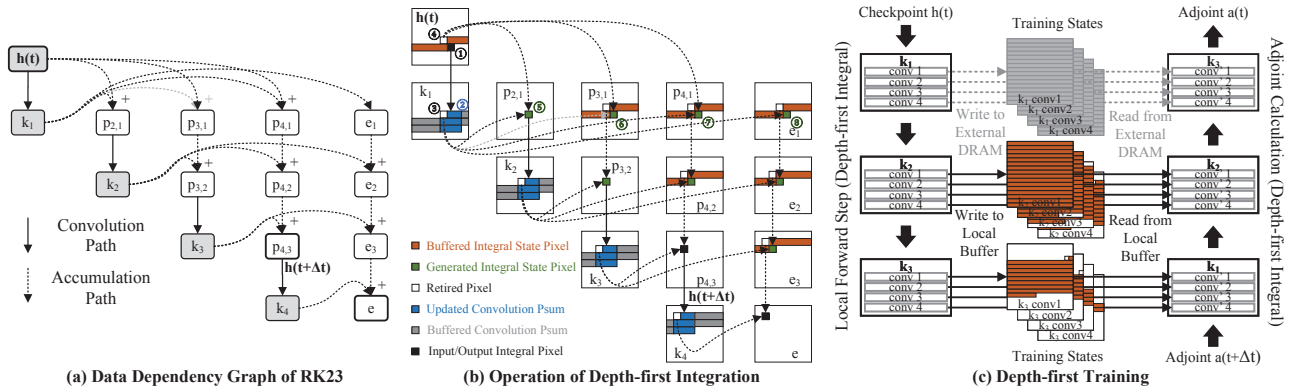
B. Depth-First Training

Building upon the depth-first integrator, we propose depth-first training to further reduce the memory cost of the intermediate training states in the backward pass. An example backward pass using a RK23 depth-first integrator is illustrated in Fig. 6(c). We assume f consists of 4 convolution layers. The backward pass only computes the integral states k_1, k_2 and k_3. Starting from a checkpoint $h(t)$, it uses a local forward step to compute all the training states including the integral states k_1, k_2 and k_3 and the internal convolution layer states. Then the adjoint $a(t)$ and parameter gradients are computed backwards using the training states. Since a depth-first integrator is used, we can start computing the adjoint when the last training state (k_3's conv4 state) has enough inputs. The adjoint calculation also proceeds in the depth-first manner, consuming inputs along the backward pass. Only the early produced training states of longer lifetime need to be stored to an external DRAM. Later produced training states with shorter lifetime can be stored on chip for immediate use by the adjoint calculation, as illustrated in Fig. 6(c). These later produced training states can be quickly retired after use to save on-chip memory.

In this example, given a 4-layer f, the memory size is reduced by 4.85 times for layer size of 64×64. The reduction leads to power savings, because SRAM and DRAM dominate the training power consumption.

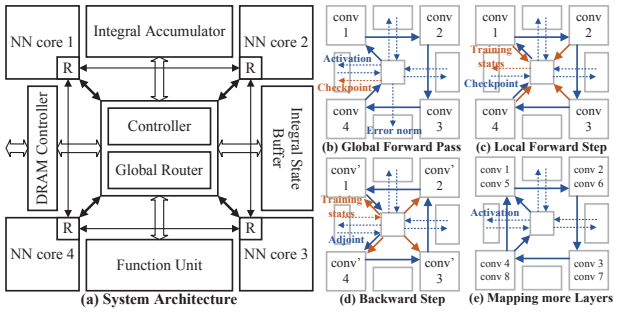
V. Function Reuse and Packet Processing

A high-order integrator requires multiple evaluations of an identical f function with identical weights. For example, an RK23 integrator evaluates the same f function four times. Therefore, a compact architecture can be made for one f function and the function along with the loaded weights can be



(a) Data Dependency Graph of RK23 (b) Operation of Depth-first Integration (c) Depth-first Training

Fig. 6. Depth-first integration.



(a) System Architecture (b) Global Forward Pass (c) Local Forward Step (d) Backward Step (e) Mapping more Layers

Fig. 7. eNODE architecture and computational dataflow.

reused to implement one integrator to achieve high efficiency. We design eNODE architecture to facilitate function reuse.

Function reuse requires folding the entire integrator onto one set of hardware, which creates a problem for the implementation of the depth-first integration as it typically requires unfolding the compute graph as illustrated in Fig. 6(a). To enable depth-first processing in a folded architecture, We propose to packetize depth-first processing.

A. eNODE Architecture for Function Reuse

As a prototype, we construct a 4-core system that consists of 4 depth-first NN cores, and each maps a convolution layer to implement a 4-layer f function, as illustrated in Fig. 7(a). A NoC connects the 4 NN cores in a ring. A loop around the ring completes one f evaluation. A high-order integration requires looping through the ring multiple times. A forward pass loops in the clockwise direction while a backward pass loops in the counter-clockwise direction. The NN cores and their stored weights are reused between loops. The function reuse leads to efficient weight reuse.

The 4 NN cores are connected to a central hub that includes a controller and a global router that can be programmed to manage the data and control flow of different types of integrators. Through the global router, the central hub accesses four peripherals: an integral accumulator that performs scaling and accumulation in an integration step; an integral state buffer that stores the integral states; a function unit that computes loss function and truncation error norm for stepsize search; and a DRAM controller that accesses external data that cannot fit in on-chip memory. The global router moves data between the NN cores and the peripherals.

The dataflow of a forward pass is illustrated in Fig. 7(b). The central hub routes the initial state from an external DRAM to NN core 1 to start the computation. One loop around all NN cores in the clockwise direction completes one f evaluation. During the evaluation, each NN core keeps the buffered lines of partial sums (psums) in convolution layers locally. The integral accumulator then computes integral partial states which are then stored to the integral state buffer. Due to the reduction in buffering of integral states, the integral state buffer can be kept small and implemented entirely on chip. The function unit computes error and decides whether to accept the current stepsize. If accepted, the final state of the current step is stored to the external DRAM as one checkpoint.

A backward pass repeats sets of a local forward step and a backward step. The local forward step reads a checkpoint from the external DRAM and computes training states as shown in Fig. 7(c). The training states that will be reused soon are buffered on chip. The training states that will be reused later are stored to the external DRAM. In a backward step, adjoint and weight gradients are calculated with the training states in the counter-clockwise direction around the NN cores as shown in Fig. 7(d). The weights are updated locally.

The eNODE architecture can be extended to support a deeper f and each NN core can map multiple layers as shown in Fig. 7(e), but a shallow f is the most common for NODE. Layers can also be split and mapped on multiple NN cores.

B. Packetized Processing

In depth-first processing, data continues to move forward. After the early data completes the first loop, it continues to the second loop. An NN core in a ring needs to handle multiple streams concurrently. In a conventional design, an NN core blocks the second stream until the first stream completes,

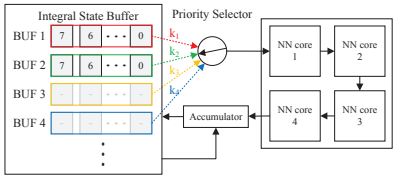


Fig. 8. Priority selection for depth-first control.

which defeats the purpose of the depth-first processing. We propose to packetize the processing to unblock later streams.

The control mechanism of the packetized processing is illustrated in Fig. 8 for computing the RK23 integrator. The controller retains four state buffers to store the inputs for the four f evaluations to compute k_1, k_2, k_3 and k_4, as required by the RK23 integrator. A priority selector monitors the availability of inputs in the state buffers and dispatches the inputs for different streams. Inputs are packetized, each with a tag to identify the stream it belongs to and its index (in our prototype, an input packet size of $R \times S \times C = 1 \times 1 \times 8$ is used). A later stream is given a higher priority to continue the data consumption and free up buffer space.

At the start, BUF 1 input is dispatched to NN core 1 to start the first stream to compute k_1. After the first input is written to BUF 2, the priority selector switches to BUF 2 and dispatches input to NN core 1 to start the second stream to compute k_2. Similarly, when the first output of the second stream becomes available, the third stream takes over, and so on. After the later streams consume the outputs, the earlier streams resume. In this way, the buffer usage is minimized.

In the example illustrated, eNODE reuses the NN cores in packetized processing. It supports various types of integrators and different orders that can be used for NODE. The hardware utilization depends on the balance between an NN core's compute capacity and the bandwidth of the ring that links the controller and the NN cores. In particular, the link bandwidth needs to be sufficiently high to maintain a high utilization of the NN cores. In the 4-core, 4-layer f example above, the NN core is designed for a 576 GFLOPS compute capacity, and it requires 1 GB/s link bandwidth for full utilization.

VI. Unified NN Core Design

Each NN core in eNODE is a depth-first NN engine that supports both forward pass and backward pass used by NODE inference and training. The design of an NN core is shown in Fig. 9(a). The NN core includes a channel collector, a PE array, a line buffer, a pre-/post-processing unit and a training state buffer. The channel collector packetizes inputs and distributes them to the PE array. The PE array is a MAC array with an adder tree to perform convolution. The line buffer caches lines of psums for depth-first convolution. The pre-/post-processing unit computes Norm and ReLU layers. The training state buffer caches training states.

In our prototype design, the PE array contains 64 PEs to support the processing of 8 input channels and 8 output channels. A convolution layer with more than 8 input channels

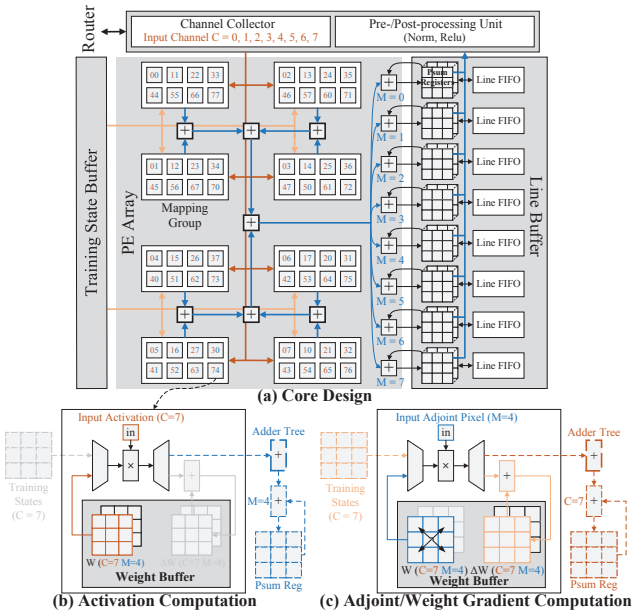


Fig. 9. Unified NN core.

and output channels can be mapped to the PE array by time-multiplexing. Each PE caches a 3×3 kernel for input channel C and output channel M and is labeled PE_{CM} (simply as CM in Fig. 9(a)). The PEs are organized in groups to facilitate the reuse of the adder tree in both forward and backward pass. An example is shown in Fig. 9(a), where the 64 PEs are divided into 8 groups. Group 0 contains $\text{PE}_{i,i}$, $i \in \{0, 1, ..., 8\}$; Group 1 contains $\text{PE}_{i,(i+1)\%8}$; Group 2 contains $\text{PE}_{i,(i+2)\%8}$; and so on, where $\%$ is the modulo operator.

The grouping unifies the processing of convolution in both forward pass and backward pass. In a forward pass, an input packet of 8 input channels are broadcast to the PE array. Within each group, the 8 PEs each take one input channel from the packet and perform multiplications to compute 9 psums for input channel C and output channel M. For example, PE_{74} in Group 5 multiplies an input with its cached 3×3 kernel to obtain a set of 9 psums at $C = 7$ and $M = 4$. This set of psums are accumulated with 7 other sets of psums ($C = \{0, 1, 2, 3, 4, 5, 6\}, M = 4$), one set from each of the remaining groups to compute the summation as illustrated in Fig. 9(b). The adder tree contains 8 parallel lanes to perform parallel summing of psums from 8 output channels.

In a backward pass, the adjoint is computed by convolution in the backward direction. In the convolution, the input and output channels switch their roles, but the processing pipeline is completely reused with flipped weight kernel shown in Fig. 9(c). For example, PE_{74} in Group 5 multiplies an adjoint input with the flipped 3×3 weight kernel to obtain a set of 9 psums at $M = 4$ and $C = 7$. This set of psums are accumulated with 7 other sets of psums ($M = \{0, 1, 2, 3, 5, 6, 7\}, C = 7$), one set from each of the remaining groups to compute the summation using the same adder tree. Weight gradients (ΔW)

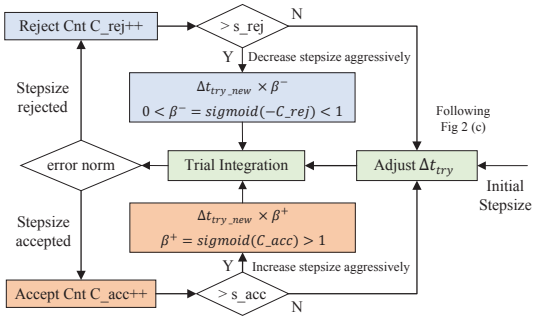


Fig. 10. Slope-adaptive stepsize search.

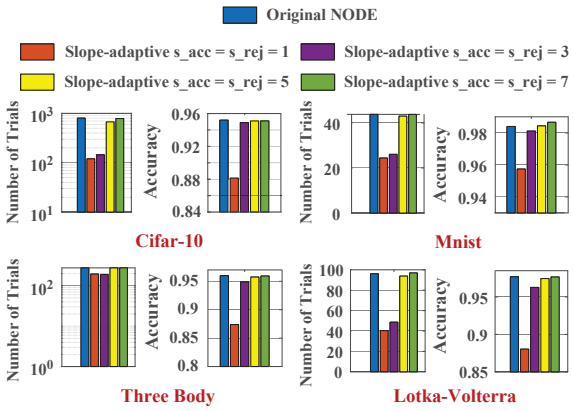


Fig. 11. Number of trials per integration layer and training accuracy using slope-adaptive stepsize search across different datasets with $\epsilon = 10^{-6}$.

are computed using the same PEs. The unified forward and backward convolution pipeline allows the reuse of the PEs, the cached weights and the adder tree.

VII. Expedited Stepsize Adjustments

As seen from the profiling in Section II.C, the forward pass in NODE inference and training dominates the overall latency. The key contributor to the long latency is the iterative stepsize search at each evaluation point. We propose slope-adaptive and priority processing techniques to shorten the latency of the iterative stepsize search.

A. Slope-Adaptive Stepsize Search

The stepsize is the unique and the most important knob in NODE. The optimal size of a step is the largest size for the step that meets the error tolerance. In a conventional stepsize search, a fixed starting stepsize is used. Then through each search trial, if the stepsize results in an unacceptable error, the stepsize is scaled down. The stepsize choice affects the latency: smaller stepsizes result in more evaluation points and longer latency, while larger stepsizes result in larger errors and require more search trials and thus longer latency to meet a given error tolerance.

The problem with the conventional stepsize search method is that it uses a nearly fixed scaling factor and ignores how fast the state changes. If the slope is fast-varying, the stepsize needs to be adjusted more aggressively. We present an improved stepsize search method based on the recent history of the evaluation points and adapt the scaling factor to reach more optimal stepsizes using fewer search trials.

Our method is illustrated in Fig. 10. The stepsize Δt_{try} is initialized to either a constant stepsize C or Δt from the previous evaluation point. We use a counter $\mathcal{C}_{acc}$ to track the number of consecutive evaluation points that accept the initial stepsize before the current evaluation point. An increase in $\mathcal{C}_{acc}$ indicates that current stepsize Δt_{try} is small and the error tolerance is always met. It could also indicate that the slope is increasing. We set a threshold s_{acc}, such that if $\mathcal{C}_{acc} \geq s_{acc}$, the scaling factor is set to $\beta^+ = \text{sigmoid}(C_{acc})$, $\beta^+ > 1$, to opportunistically increase the stepsize to reduce the number of evaluation points. Similarly, we use another counter $\mathcal{C}_{rej}$ to track the number of consecutive evaluation points that reject the initial stepsize. An increase in $\mathcal{C}_{rej}$ indicates that the current stepsize Δt_i is too large and/or the slope is changing quickly. We set a threshold s_{rej}, such that if $\mathcal{C}_{rej} \geq s_{rej}$, the scaling factor is set to $\beta^- = \text{sigmoid}(-C_{rej})$, $0 < \beta^- < 1$, to decrease the stepsize to reduce the number of search trials.

Experimented in solving classic image classification problems (CIFAR-10 [18] and MNIST [6]) and modeling two representative dynamic systems (Three-Body equations [3] and Lotka-Volterra equations [29]), the slope-adaptive stepsize search can effectively reduce the latency by up to 6.7× (for CIFAR-10 dataset) as shown in Fig. 11. However, as an approximation algorithm, it may impact the accuracy. Using a threshold of $s_{acc} = s_{rej} = 3$, the accuracy degradation across different datasets can be kept within 1%, while still achieving similar trial reduction as $s_{acc} = s_{rej} = 1$. Further increasing the thresholds improves the training accuracy, but diminishes the trial reduction as indicated in Fig. 11.

B. Priority processing and early stop

At each evaluation point, search trials are performed to adjust the stepsize until the L^2 norm of the truncation error $\|e\|_2$ falls within the error tolerance ϵ. Each trial traverses the entire input feature map to compute the intermediate integral states k's and estimate the truncation error norm $\|e\|_2$ as shown in Fig. 12(a), representing a significant latency bottleneck. We observe that the truncation error norm is often dominated by only a portion of the output feature maps, namely, the high error region. Therefore, we propose priority processing and early stop as explained by Fig. 12(b), where the high error region is identified and prioritized for processing, aiming to stop the search trial early for latency and energy saving.

The depth-first integrator computes e incrementally, if a partially computed $\|e\|_2$ exceeds ϵ, a search trial can be terminated early to cut the latency. To make it even more effective, the processing needs to be priority-ordered as shown in Fig. 12(b). At each evaluation point, we use the first search trial as initialization, where we compute the entire trial integration and identify the region of input that contributes high error. This high error region is outlined by a number of

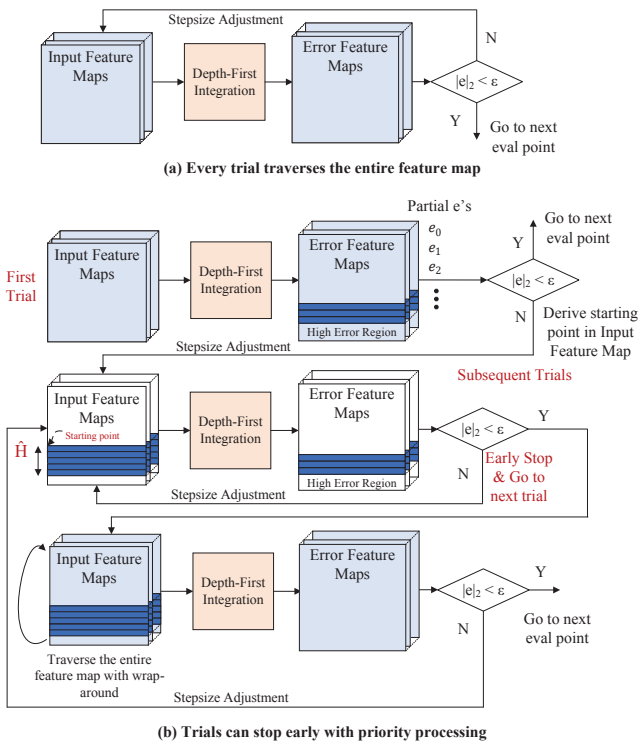


Fig. 12. Priority processing and early stop.

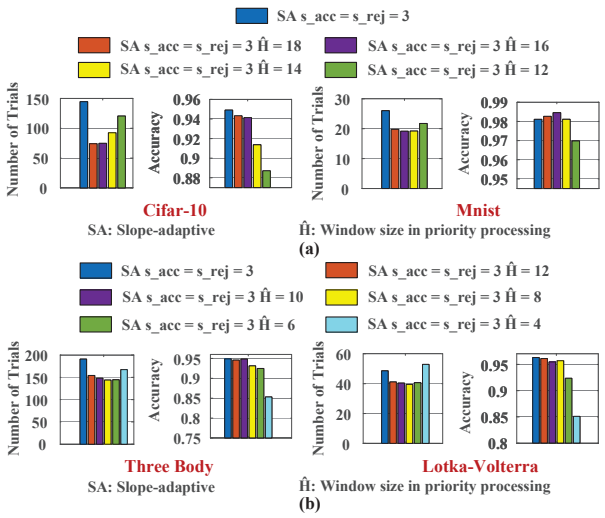


Fig. 13. Number of trials per integration layer and training accuracy using priority processing and early stop in (a) image classification workloads and (b) dynamic system workloads with $\epsilon = 10^{-6}$.

consecutive rows in the truncation error map (e) that have the largest $\|e\|_2$ and corresponds to a priority processing window of height $\hat{H}$ at the input feature map. Then, in subsequent search trials, the high error region is processed first by the depth-first integrator. If partial $\|e\|_2$ of the high error region is still less than ϵ, the depth-first integrator continues to process the remaining rows; otherwise the trial can be stopped early.

The trial reduction and impact on accuracy by priority processing and early stop are illustrated in Fig. 13(a) and (b) for different choices of window size $\hat{H}$. In general, applying the approach helps reduce the number of trials, but a smaller window size tends to degrade the accuracy. For the image classification workloads (CIFAR-10 and MNIST), keeping the accuracy degradation within 3% requires a window size $\hat{H} \geq 16$. For the dynamic system workloads (Three-Body problem and Lotka-Volterra equations), a window size of $\hat{H} \geq 8$ ensures less than 3% accuracy drop. One can choose a proper combination of s_{acc}, s_{rej} and $\hat{H}$ to meet the target accuracy for a given workload.

VIII. Evaluations and Benchmarking

We developed a parameterized cycle-accurate model of eNODE for performance analysis and memory access profiling. A prototype of eNODE was also implemented in RTL and synthesized in a 28 nm technology for performance, area and power evaluation. We used PrimeTime to estimate the power consumption based on activities of running complete integration steps of the benchmarks. The power consumption of DRAM accesses was estimated by the Ramulator DRAM simulator [17]. For comparison, a cycle-accurate model and a RTL implementation of an ASIC baseline were also developed. The ASIC baseline was designed in a weight-stationary SIMD architecture with local psum accumulation [22]. It processes NODE layer by layer. The baseline contains the same number of MAC units as the eNODE prototype for a fair comparison. All designs use FP16 precision to support ODE applications.

Designs were evaluated by running training and inference on two image classification workloads: CIFAR-10 [18] and MNIST [6], and two dynamic system workloads: Three-body equation [3] and Lotka-Volterra equations [29]. These are the most common benchmarks used by the NODE algorithm community.

The Three-Body equation describes the trajectories of planets and interactions in space. The equation is given in (6), where G is the gravitation constant; r_i is the location of planet i, each of dimension 3; $\ddot{r}_i$ is the second derivative with respect to time; and m_i is the mass of planet i.

$$\ddot{r}_i = -\sigma_{j \neq i} G m_j \frac{r_i - r_j}{|r_i - r_j|^3}. \tag{6}$$

Lotka-Volterra equations describe the dynamics of a biological system in which two species, predator and prey, interact. The equations are given in (7), where x is the number of preys; y is the number of predators; $\dot{x}$ and $\dot{y}$ are the first derivatives with respect to time t; others are the parameters of a biological system.

$$\dot{x} = \alpha x - \beta xy, \dot{y} = \delta xy - \eta y. \tag{7}$$

A. Memory Size and Area

The baseline design for NODE needs to buffer the entire initial state $h(t)$ and all integral states k_i. eNODE's depth-first integration reduces memory usage for integral states, but requires additional line buffers to store lines of psums for convolution layers. However, as stated previously, shallow

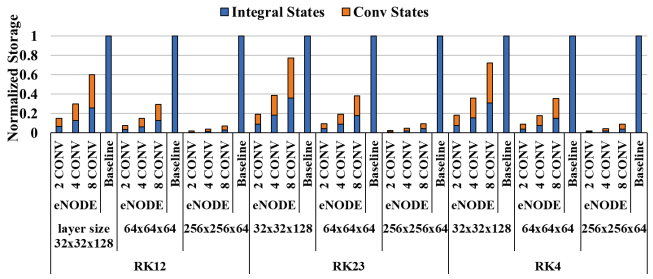


Fig. 14. Normalized integral states storage size for different integrators, layer sizes, and number of conv layers in f.

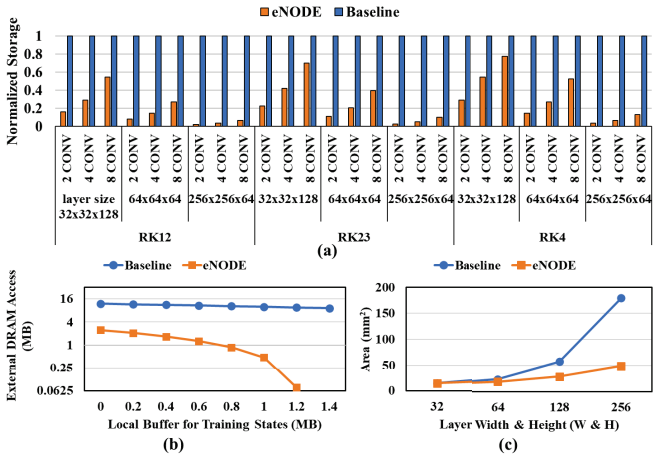


Fig. 15. (a) Normalized training states storage size for different integrators, layer sizes, and number of layers in f. (b) The effectiveness of a local buffer to eliminate external DRAM access for training states. The example uses RK23 integrator with 4 conv layers in f. (c) Area scalability comparison of eNODE with the ASIC baseline.

NNs, e.g., 4-layer f, are the most common for NODE, limiting the line buffer size. Fig. 14 summaries the reduction with different choices of integrators, layer sizes and number of convolution layers in f. The memory size reduction depends on the layer size and more reduction is possible for large layer sizes. This is because for a layer of size $H \times W \times C$, eNODE's memory size is $O((W+1) \times C)$ while the baseline's memory size is $O(H \times W \times C)$. For example, for a layer size of $64 \times 64 \times 64$, eNODE's memory size is 60% smaller than the baseline; and for a layer size of $256 \times 256 \times 64$, eNODE's memory size is 90% smaller than the baseline.

eNODE's depth-first training reduces storage for training states as summarized in Fig. 15(a). The reduction is also dependent on the layer size and the depth of f. For a 4-layer f, the storage size is reduced by more than 45%. With the sizable reduction, we can opt for a smaller on-chip memory to eliminate the external DRAM access for training states. As shown in Fig. 15(b), a 1 MB on-chip memory reduces the external DRAM access for training states in eNODE to 0.48 MB, a 21× reduction compared to the baseline; and a 1.25 MB on-chip memory in eNODE fully eliminates the DRAM access for training states. In comparison, a 6 MB on-chip memory is required by the baseline to remove the DRAM access for training states.

TABLE I
MEMORY AND AREA BREAKDOWN OF BASELINE AND ENODE

	Baseline		eNODE	
	MB	mm^2	MB	mm^2
Configuration A	For Layer Size: 64×64×64			
Core & Control	-	3.53	-	3.66
Weight Buffer	2.25	5.34	2.25	5.34
Integral State Buffer	2	9.24	0.44	2.03
Line Buffer	-	-	0.5	2.31
Training State Buffer	1.25	5.78	1.25	5.78
Total	**5.5**	**23.89**	**4.44**	**19.12**
Configuration B	For Layer Size: 256×256×64			
Core & Control	-	3.53	-	3.66
Weight Buffer	2.25	5.34	2.25	5.34
Integral State Buffer	32	147.84	1.76	8.13
Line Buffer	-	-	2	9.24
Training State Buffer	4.9	22.64	4.9	22.64
Total	**39.15**	**179.35**	**10.91**	**49.01**

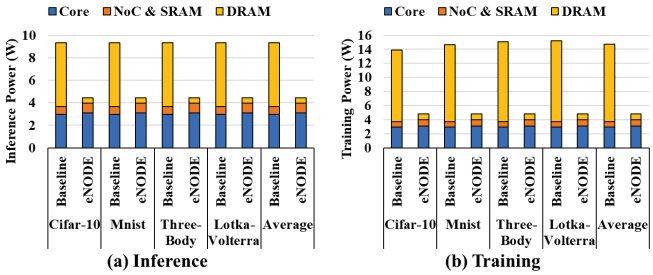


Fig. 16. Inference and training power consumption of baseline and eNODE.

eNODE demonstrates favorable area scalability. Table I lists the memory and area breakdown of ASIC baselines and eNODE prototypes targeting two layer sizes. In Configuration A for the layer size of $64 \times 64 \times 64$, the eNODE prototype saves 1.06 MB in on-chip SRAM and 20% in total area over the ASIC baseline; and in Configuration B for the layer size of $256 \times 256 \times 64$ the eNODE prototype saves 28.24 MB in on-chip SRAM and 72.7% in total area over the ASIC baseline. Fig. 15(c) summarizes the area scalability for different layer sizes. The eNODE design scales nearly linearly while the ASIC baseline scales quadratically.

B. Power Consumption

eNODE reduces power consumption by reducing and even eliminating DRAM accesses through depth-first processing. In inference, depth-first integration keeps intermediate activations in the dataflow pipeline and on-chip buffers throughout the integration steps. In comparison, the ASIC baseline transfers intermediate activations of every NN layer between the cores and the DRAM. Fig. 16(a) summaries the inference power for all benchmarks using Configuration A in Table I. On average, eNODE reduces the DRAM power from 5.65 W to 0.48 W and reduces the total power from 9.32 W to 4.43 W, a 2.1× reduction over the baseline.

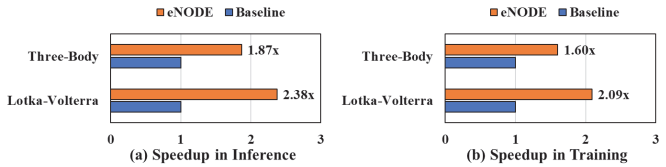


Fig. 17. Speedup by eNODE over baseline in inference and training.

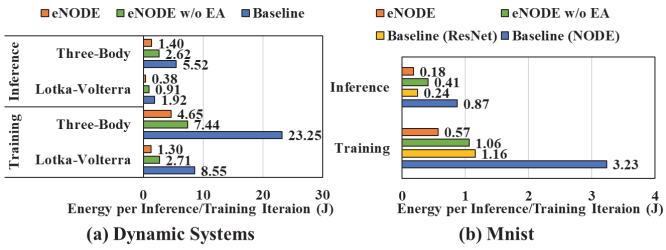


Fig. 18. Energy efficiency of eNODE and baseline inference and training. ResNet-200 is used in (b).

In training, the backward pass requires training states to be stored to DRAM for later computation, which costs more power. In eNODE, depth-first training eliminates most of the DRAM access for training states, thus reducing the average DRAM power from 11.03 W to 0.85 W for Configuration A, as shown in Fig. 16(b). The average training power is reduced by 3.05× over the baseline, from 14.72 W to 4.82 W.

C. Speedup

With the expedited stepsize adjustments, eNODE achieves speedup in both inference and training. Fig. 17(a) and (b) summarize the speedup in inference and training for the Three-Body and the Lotka-Volterra benchmarks over the ASIC baseline for Configuration A in Table I. In the experiments, ϵ is set to 10^{-6}, s_{acc} and s_{rej} are both set to 3 for the slope-adaptive stepsize search, and $\hat{H}$ is set to 10 for the priority processing and early stop. The parameter settings ensure less than 2% drop in accuracy across all benchmarks. In inference, eNODE achieves 1.87× and 2.38× speedup over the baseline on Three-Body and Lotka-Volterra, respectively. In training, eNODE achieves 1.6× and 2.09× speedup over the baseline on Three-Body and Lotka-Volterra, respectively.

D. Energy Efficiency

We evaluate the energy efficiency of eNODE in terms of energy per inference (J/inference) and energy per training iteration (J/training iteration) using Configuration A in Table I. Fig. 18(a) summarizes the energy of the two dynamic system benchmarks: Three-Body and Lotka-Volterra. Both the depth-first integration and the expedited algorithms (labeled "EA" in Fig. 18) play a role in improving eNODE's energy: the depth-first integration reduces the DRAM power; and the expedited stepsize adjustments reduce the computation complexity, shorten the latency and improve the throughput.

Without the expedited algorithms, eNODE's depth-first architecture alone provides 3.12× and 3.16× lower energy in training over the baseline on Three-Body and Lotka-Volterra, respectively; and 2.1× lower energy in inference on the two benchmarks. Putting together expedited algorithms and depth-first integration, eNODE provides 5× and 6.59× lower energy in training over the baseline on Three-Body and Lotka-Volterra, respectively; and 3.94× and 5× lower energy in inference on the two benchmarks, respectively.

eNODE compares favorably to conventional NNs in terms of energy in performing same tasks of comparable accuracy. For example, ResNet-200 was proven to achieve a comparable accuracy with NODE in certain tasks [24], but eNODE outperforms ResNet-200 (mapped on the ASIC baseline) in energy, e.g., in running the MNIST image classification benchmark [6] as shown in Fig. 18(b). Even without the expedited algorithms, eNODE still outperforms ResNet-200 in training.

Compared to an Nvidia A100 deep learning GPU on AWS, eNODE reduces the CIFAR-10 training energy by 55×. Note that unlike eNODE, A100 does not target edge compute.

IX. Conclusions

NODE is an emerging paradigm for modeling dynamic systems. NODE's complexity and memory usage significantly exceed those of conventional NNs. It is impractical to deploy an NN accelerator designed for real-time NN operations to perform NODE operations in real time.

We present eNODE, an architecture-algorithm co-designed NODE accelerator. To reduce memory and memory-associated power consumption, eNODE makes use of depth-first integration at each elementary step of NODE processing. The high-order integrator's compute graph is factored and the steps are ordered such that whenever an input becomes available, all dependent processing is triggered in parallel, allowing the input to be quickly consumed and retired from the buffer to save memory. Based on depth-first integration, eNODE employs depth-first training to further reduce the memory of intermediate training states in the backward pass.

To improve efficiency, eNODE implements function reuse on the same hardware. The eNODE architecture is based on a ring of NN cores with cached weights. A loop around the ring implements a function evaluation. The function and weights are reused by looping multiple times. The processing is packetized to allow multiple concurrent loops to support depth-first integration. The NN core also supports both forward pass and backward pass using the same hardware.

To improve latency, eNODE adopts two algorithmic approaches to alleviate the bottleneck in the iterative stepsize search trials: a slope-adaptive stepsize search to find the optimal stepsize based on recent slope history, and priority processing and early stop to process high-priority windows and terminate a search trial early to save latency.

An eNODE prototype is implemented in RTL and synthesized in a 28nm technology. The prototype uses 60% less memory in inference and costly DRAM access can be eliminated with proper local buffer size choice. The eNODE prototype demonstrates up to 6.59× lower energy and 2.38× higher speed over a SIMD ASIC baseline.

REFERENCES

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[2] D. Balduzzi, M. Frean, L. Leary, J. Lewis, K. W.-D. Ma, and B. McWilliams, "The shattered gradients problem: If resnets are the answer, then what is the question?" in *International Conference on Machine Learning*. PMLR, 2017, pp. 342–350.

[3] J. Barrow-Green, *Poincaré and the three body problem*. American Mathematical Soc., 1997, no. 11.

[4] B. Chang, L. Meng, E. Haber, L. Ruthotto, D. Begert, and E. Holtham, "Reversible architectures for arbitrarily deep residual neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[5] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/69386f6bb1dfed68692a24c8686939b9-Paper.pdf

[6] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[7] E. Dupont, A. Doucet, and Y. W. Teh, "Augmented neural odes," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/21be9a4bd4f81549a9d1d241981cec3c-Paper.pdf

[8] E. Fehlberg, *Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems*. National aeronautics and space administration, 1969, vol. 315.

[9] A. Gholami, K. Keutzer, and G. Biros, "Anode: Unconditionally accurate memory-efficient gradients for neural odes," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, ser. IJCAI'19. AAAI Press, 2019, p. 730–736.

[10] K. Goetschalckx and M. Verhelst, "Breaking high-resolution cnn bandwidth barriers with enhanced depth-first execution," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 323–331, 2019.

[11] K. Goetschalckx and M. Verhelst, "Depfin: A 12nm, 3.8tops depth-first cnn processor for high res. image processing," in *2021 Symposium on VLSI Circuits*, 2021, pp. 1–2.

[12] M. Hardt and T. Ma, "Identity matters in deep learning," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=ryxB0Rtxx

[13] R. Hasani, M. Lechner, A. Amini, D. Rus, and R. Grosu, "Liquid time-constant networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 9, pp. 7657–7666, May 2021. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16936

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*. Springer, 2016, pp. 630–645.

[16] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[17] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[18] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 and cifar-100 datasets," *URl: https://www.cs.toronto.edu/kriz/cifar. html*, vol. 6, no. 1, p. 1, 2009.

[19] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf

[20] Y. Li and Y. Yuan, "Convergence analysis of two-layer neural networks with relu activation," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/a96b65a721e561e1e3de768ac819ffbb-Paper.pdf

[21] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama, "Dissecting neural odes," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 3952–3963. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/293835c2cc75b585649498ee74b395f5-Paper.pdf

[22] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 246–247.

[23] W. H. Press and S. A. Teukolsky, "Adaptive stepsize runge-kutta integration," *Computers in Physics*, vol. 6, no. 2, pp. 188–191, 1992.

[24] A. F. Queiruga, N. B. Erichson, D. Taylor, and M. W. Mahoney, "Continuous-in-depth neural networks," *arXiv preprint arXiv:2008.02389*, 2020.

[25] C. Runge, "Uber die numerische auflosung von differentialgleichungen," *Mathematische Annalen*, vol. 46, no. 2, pp. 167–178, 1895.

[26] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[27] Y. Tao and Z. Zhang, "Hima: A fast and scalable history-based memory access engine for differentiable neural computer," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021, p. 845–856.

[28] A. Veit, M. J. Wilber, and S. Belongie, "Residual networks behave like ensembles of relatively shallow networks," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper/2016/file/37bc2f75bf1bcfe8450a1a41c200364c-Paper.pdf

[29] V. Volterra, "Variations and fluctuations of the number of individuals in animal species living together," *Animal ecology*, pp. 409–448, 1926.

[30] H. Yan, J. Du, V. Tan, and J. Feng, "On robustness of neural ordinary differential equations," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=B1e9Y2NYvS

[31] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[32] X. Zhang, Z. Li, C. Change Loy, and D. Lin, "Polynet: A pursuit of structural diversity in very deep networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 718–726.

[33] J. Zhuang, N. Dvornek, X. Li, S. Tatikonda, X. Papademetris, and J. Duncan, "Adaptive checkpoint adjoint method for gradient estimation in neural ode," in *International Conference on Machine Learning*. PMLR, 2020, pp. 11 639–11 649.

[34] J. Zhuang, N. C. Dvornek, sekhar tatikonda, and J. s Duncan, "Mali: A memory efficient and reverse accurate integrator for neural odes," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=blfSjHeFM_e