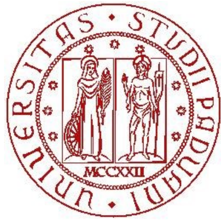


DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

TT-DeepONet: Un Framework Efficiente per la risoluzione di EDP su Dispositivi con Risorse Limitate, basato su Compressione Tensoriale

Relatore
Prof. Vogrig Daniele
University of Padua

Supervisore
Prof. Zhang Zheng
University of California, Santa Barbara

Studente Tonelli Paolo

ANNO ACADEMICO 2023-2024

12/11/2024

Sommario

Le equazioni differenziali alle derivate parziali (EDP) sono fondamentali per modellare e comprendere fenomeni complessi in diverse discipline scientifiche e ingegneristiche, dalla fluidodinamica e trasmissione del calore alla scienza dei materiali e alla finanza. I metodi numerici tradizionali per risolvere le EDP, sebbene consolidati, spesso incontrano limitazioni in termini di costo computazionale e scalabilità, in particolare quando si ha a che fare con problemi ad alta dimensionalità o geometrie complesse. Il deep learning è emerso come un'alternativa promettente, offrendo il potenziale per superare queste limitazioni e consentire la soluzione di problemi EDP precedentemente intrattabili. Le Deep Operator Networks (DeepONet), una classe specializzata di reti neurali, sono particolarmente adatte a questo compito, in quanto possono apprendere gli operatori sottostanti che mappano tra spazi funzionali, rappresentando efficacemente le soluzioni delle EDP.

Tuttavia, le DeepONet possono essere computazionalmente e a livello di memoria intensive. TT-DeepONet affronta questo problema combinando la decomposizione tensoriale Tensor-Train (TT) e l'addestramento quantization-aware (QAT). La decomposizione TT comprime le matrici dei pesi delle DeepONet, riducendo l'ingombro di memoria e il costo computazionale. Il QAT migliora ulteriormente l'efficienza addestrando il modello con una precisione numerica ridotta.

Un contributo chiave è un livello lineare tensorizzato che integra la decomposizione TT nella DeepONet, consentendo contrazioni tensoriali efficienti. Il QAT è incorporato per mantenere la precisione nonostante la minore precisione. Inoltre, viene proposto un progetto di acceleratore FPGA basato su systolic array per migliorare le prestazioni. Sebbene non completamente implementato, le simulazioni preliminari mostrano risultati promettenti.

I risultati sperimentali dimostrano che TT-DeepONet raggiunge una precisione paragonabile alle DeepONet standard con memoria significativamente inferiore e inferenza più veloce, specialmente con l'accelerazione FPGA parzialmente implementata per l'equazione di reazione-diffusione. Questo lavoro contribuisce all'apprendimento automatico hardware-aware, consentendo l'implementazione di modelli di deep learning complessi su edge device per il calcolo scientifico.

Abstract

PDEs are fundamental to modeling and understanding complex phenomena across diverse scientific and engineering disciplines, from fluid dynamics and heat transfer to materials science and finance. Traditional numerical methods for solving PDEs, while well-established, often encounter limitations in terms of computational cost and scalability, particularly when dealing with high-dimensional problems or complex geometries. Deep learning has emerged as a promising alternative, offering the potential to overcome these limitations and enable the solution of previously intractable PDE problems. Deep Operator Networks (DeepONets), a specialized class of neural networks, are particularly well-suited for this task, as they can learn the underlying operators that map between function spaces, effectively representing the solutions of PDEs.

However, DeepONets can be computationally and memory intensive. TT-DeepONet addresses this by combining tensor-train (TT) decomposition and quantization-aware training (QAT). TT decomposition compresses the DeepONet's weight matrices, reducing memory footprint and computational cost. QAT further improves efficiency by training the model with reduced numerical precision.

A key contribution is a tensorized linear layer that integrates TT decomposition into the DeepONet, enabling efficient tensor contractions. QAT is incorporated to maintain accuracy despite lower precision. Furthermore, a systolic array-based FPGA accelerator design is proposed to enhance performance. While not fully implemented, preliminary simulations show promising results.

Experimental results demonstrate that TT-DeepONet achieves comparable accuracy to standard DeepONets with significantly less memory and faster inference, especially with the partially implemented FPGA acceleration for the reaction-diffusion equation. This work contributes to hardware-aware machine learning, enabling deployment of complex deep learning models on edge devices for scientific computing.

Contents

1 Introduction 1

2 Background and Related Work 3

2.1 Partial Differential Equations (PDEs) 3

2.1.1 Types of PDEs 3

2.1.2 Applications of PDEs 4

2.1.3 Traditional Numerical Methods for Solving PDEs 5

2.2 Deep Learning for PDEs 6

2.2.1 Physics-Informed Neural Networks (PINNs) 6

2.2.2 Operator Learning and Deep Operator Networks (DeepONets) 7

2.2.3 DeepONets vs PINNs 8

2.3 Tensor Decompositions for Model Compression 9

2.3.1 Introduction to Tensors and Tensor Decompositions 9

2.3.2 Tensor-Train Decomposition 10

2.4 FPGA Acceleration for Deep Learning 11

2.4.1 FPGA Architecture and Advantages 11

2.4.2 RTL Design vs High-Level Synthesis (HLS) 12

2.4.3 Systolic Arrays for Matrix Multiplication 13

3 Tensorized DeepONet with Quantization-Aware Training 15

3.1 Software Framework 15

3.2 Building a Tensorized DeepONet Model 16

3.3 Quantization-Aware Training 17

3.3.1 Introduction to Quantization 17

3.3.2 QAT in the Tensorized DeepONet 18

3.4 Evaluation and Results 19

3.4.1 Experimental Setup 19

3.4.2 Evaluation 19

4 FPGA Accelerator Design **23**
4.1 System Overview and Design Goals 23
4.1.1 From contractions to matrix multiplications 23
4.2 Tensor Contraction Unit (TCU) 26
4.2.1 Processing Element Design 26
4.2.2 Dataflow and Control Logic 26
4.3 Simulation and Verification 27

5 Conclusion and Future Work **29**
5.1 Summary of Contributions 29
5.2 Future Directions 30

Bibliography **31**

List of Figures

2.1 Prediction of a thermal field obtained by solving the heat equation. 4
2.2 Schematic of a Physics-Informed Neural Network (PINN) 7
2.3 Problem setup and architecture of a DeepONet. 8
2.4 CP and Tucker decompositions 10
2.5 Tensor-train decomposition. 11
2.6 Simplified structure of an FPGA. 12
2.7 HLS design flow. 12
2.8 Systolic array architecture of a neural network layer. 13

3.1 Flowchart of DeepXDE for solving differential equations. 16
3.2 Schematic of a TT-compressed DeepONet. 17
3.3 Flowchart of Quantization-Aware-Training process. 18
3.4 Training and test loss for different compression rates 21

4.1 Dataflow of system communication. 25
4.2 FSM of control sequence 25
4.3 RTL Schematic of a PE with BRAM. 25
4.4 RTL Schematic of the Controller unit. 25
4.5 Contraction sequence of a tensorized layer. 25
4.6 Waveform of simulation of a matrix-matrix multiplication. 27

Chapter 1

Introduction

Partial differential equations (PDEs) are fundamental to modeling and understanding complex phenomena in various scientific and engineering domains. However, traditional numerical methods for solving PDEs often face limitations in terms of computational cost, flexibility, and scalability, especially for high-dimensional problems and complex geometries. This motivates the exploration of deep learning techniques, which have shown remarkable success in approximating complex functions and solving challenging problems in other areas. Deep learning offers the potential to develop more efficient, adaptable, and data-driven approaches for PDE solving.

Traditional numerical methods for solving PDEs, such as finite difference, finite element, and finite volume methods, rely on discretizing the spatial and temporal domains. These methods can become computationally expensive and memory-intensive, particularly for high-dimensional PDEs and complex geometries. They often require fine-grained discretization to achieve accurate solutions, leading to a large number of unknowns and a significant computational burden. Moreover, these methods can be inflexible when dealing with complex boundary conditions or irregular domains.

Deep Operator Networks (DeepONets) are a promising deep learning architecture specifically designed to learn operators that map between function spaces. Unlike traditional neural networks that approximate functions, DeepONets can approximate entire operators, making them well-suited for representing the solutions of PDEs. DeepONets consist of two sub-networks: a branch network that encodes the input function and a trunk network that encodes the output domain. A merging function then combines the outputs of these networks to produce the final approximation of the PDE solution.

Field-Programmable Gate Arrays (FPGAs) offer a compelling platform for accelerating deep learning workloads due to their reconfigurability, parallelism, and energy efficiency. FPGAs allow for customizing the hardware architecture to match the specific computational patterns of deep learning models, leading to significant performance gains and reduced power consumption

compared to general-purpose processors like CPUs and GPUs. Systolic arrays, specialized hardware structures designed for efficient matrix and tensor operations, are particularly well-suited for implementation on FPGAs.

This thesis aims to develop an efficient and hardware-aware framework for solving PDEs on resource-constrained devices by combining the power of tensor-compressed DeepONets with FPGA acceleration. The key contributions include: (1) Implementing a tensorized DeepONet that leverages tensor-train decompositions to significantly reduce the memory footprint of the model parameters while preserving accuracy. (2) Designing a custom FPGA accelerator architecture based on systolic arrays to efficiently perform the tensor contractions involved in the DeepONet operations. (3) Evaluating the performance, resource utilization, and power efficiency of the proposed framework on a target FPGA platform, demonstrating its advantages over traditional methods.

The remainder of this thesis is organized as follows: Chapter 2 provides the necessary background on PDEs, DeepONets, tensor decompositions, and FPGA acceleration. Chapter 3 details the implementation of the tensorized DeepONet and its evaluation on a chosen PDE problem. Chapter 4 focuses on the design and implementation of the FPGA accelerator, and presents the experimental results with an analysis of the framework's performance. Finally, Chapter 5 summarizes the contributions, discusses limitations, and outlines future research directions.

Chapter 2

Background and Related Work

2.1 Partial Differential Equations (PDEs)

2.1.1 Types of PDEs

Partial differential equations (PDEs) are mathematical expressions that involve an unknown multivariable function and its partial derivatives. They provide a powerful framework for describing a wide range of phenomena in physics, engineering, and other fields where quantities vary continuously over space and time. PDEs are classified based on their order, which refers to the highest order of derivative present in the equation, and linearity. Linear PDEs involve only linear combinations of the unknown function and its derivatives, while nonlinear PDEs include nonlinear terms, making them generally more challenging to solve.

Common types of PDEs include elliptic, parabolic, and hyperbolic equations. Elliptic PDEs, such as Laplace's equation (2.1) and Poisson's equation, describe steady-state or equilibrium phenomena where the solution does not change over time. Examples include the distribution of heat in a steady-state system or the electric potential in a region with fixed charges.

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = 0 \tag{2.1}$$

Parabolic PDEs, like the heat equation (2.2), model diffusion processes where a quantity spreads out over time, such as the diffusion of heat in a metal bar or the spread of a contaminant in a fluid.

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{2.2}$$

Hyperbolic PDEs, such as the wave equation (2.3) and the advection equation, describe wave propagation or the transport of quantities, for example, the propagation of sound waves or the

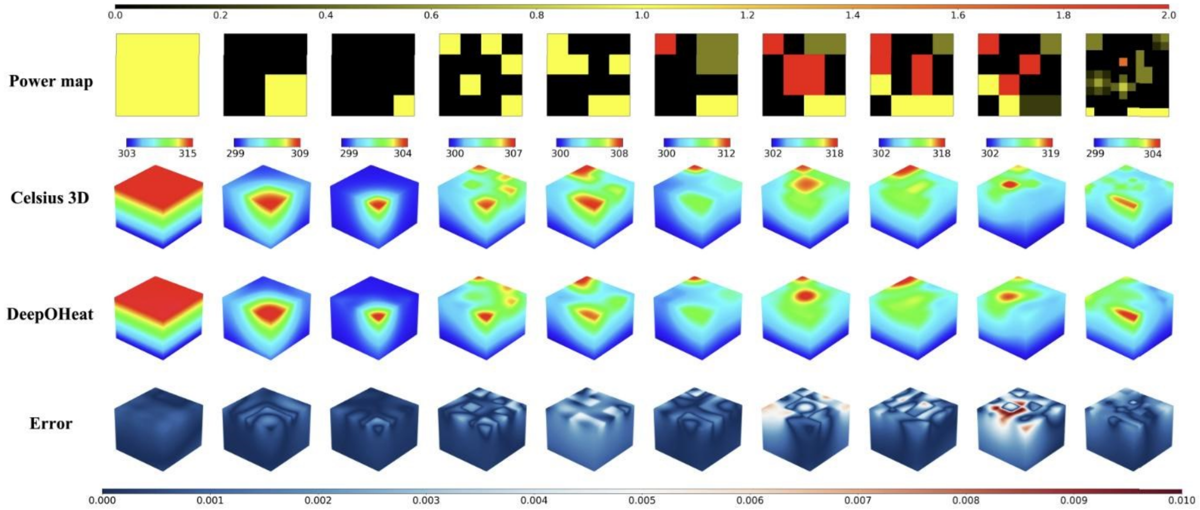


Figure 2.1: Prediction of a thermal field of an IC obtained by solving the heat equation. Top row is the result of a traditional numerical solver. Bottom row is the result of a Deep-Learning model [1].

movement of pollutants in a river.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{2.3}$$

Understanding the type of PDE is essential for selecting appropriate numerical solution methods or deep learning architectures. Different types of PDEs exhibit different mathematical properties and require different approaches for their solution. For instance, elliptic PDEs typically require boundary conditions to be specified, while parabolic and hyperbolic PDEs also need initial conditions.

2.1.2 Applications of PDEs

PDEs are ubiquitous in science and engineering, serving as the foundation for modeling and understanding countless real-world phenomena. Their applications span a vast range of disciplines, highlighting their importance in advancing our knowledge and technological capabilities. In fluid dynamics, the Navier-Stokes equations, a set of nonlinear PDEs, govern the motion of fluids, providing insights into the behavior of airfoils, weather patterns, and ocean currents. Heat transfer relies on the heat equation, a parabolic PDE, to model the flow of heat in materials, enabling the design of efficient thermal systems for power plants, buildings, and electronic devices.

Electromagnetism is governed by Maxwell's equations, a set of coupled PDEs that describe the interplay between electric and magnetic fields. These equations form the basis for technolo-

gies like antennas, motors, generators, and wireless communication systems. In finance, the Black-Scholes equation, a parabolic PDE, is widely used for option pricing, allowing for the valuation of financial derivatives and risk management. PDEs also find applications in image processing, where diffusion equations are used for image denoising, edge detection, and image segmentation, and in biology, where reaction-diffusion equations model pattern formation in organisms and the spread of diseases.

The increasing complexity of these applications often demands sophisticated numerical methods or deep learning techniques to solve the governing PDEs efficiently and accurately. DeepONets, with their ability to learn complex operators, offer a promising avenue for tackling challenging PDE problems in these diverse fields.

2.1.3 Traditional Numerical Methods for Solving PDEs

Traditional numerical methods for solving PDEs rely on discretizing the continuous domain of the problem into a finite set of points, effectively converting the PDE into a system of algebraic equations that can be solved using computers. This discretization process involves approximating the derivatives in the PDE using finite differences, finite elements, or finite volumes. The finite difference method [2] approximates derivatives using differences between function values at neighboring grid points, while the finite element method [3] divides the domain into smaller elements and approximates the solution within each element using basis functions. The finite volume method [4] focuses on conserving quantities over control volumes, making it particularly suitable for problems involving conservation laws.

These methods have been the workhorse for solving PDEs for decades and have been successful in many applications. However, as the complexity of PDE problems increases, traditional numerical methods often encounter limitations. For high-dimensional PDEs, the number of discretization points grows exponentially, leading to a significant computational burden and memory requirements. Complex geometries and boundary conditions can also pose challenges for these methods, requiring intricate meshing techniques and potentially leading to numerical instabilities. The computational cost and limitations of traditional methods motivate the exploration of alternative approaches, such as deep learning, to address these challenges and enable the solution of increasingly complex PDE problems.

2.2 Deep Learning for PDEs

2.2.1 Physics-Informed Neural Networks (PINNs)

One of the biggest challenges in supervised deep learning for solving PDEs is the fact that, for complex systems, obtaining even a single data point can be exceptionally expensive, both in terms of time and computational cost. Thus, a different approach was needed in order to train a model while lacking the amount of data normally needed for a classical model.

Physics-informed neural networks (PINNs), first introduced in [5], are a class of deep learning models that incorporate physical laws and constraints directly into the loss function during training.

Let's consider the following partial differential equations of the general form

$$u_t + \boldsymbol{\mathcal{N}}[u; \lambda] = 0, \tag{2.4a}$$

$$x \in \Omega, \tag{2.4b}$$

$$t \in [0,T] \tag{2.4c}$$

Where:

- $u(x, t)$ represents the unknown solution we seek.
- x and t are the spatial and temporal coordinates, respectively.
- Ω represents the spatial domain.
- $[0, T]$ is the time interval.

- $\mathcal{N}[u; \lambda]$ is a differential operator that can be linear or nonlinear, potentially involving parameters (λ).

Boundary conditions specify the behavior of the solution at the spatial boundaries $\partial\Omega$:

$$u(x, t) = g(x, t), \quad t \in [0, T], \quad x \in \partial\Omega \tag{2.5}$$

where $g(x, t)$ is a known function defining the boundary values.

The initial condition specifies the initial state of the solution at $t = 0$:

$$u(x, 0) = h(x), \quad x \in \Omega \tag{2.6}$$

where (h(x)) is a known function.

By encoding the PDE (2.4a), boundary conditions (2.5), and initial conditions (2.6) into the loss, PINNs can learn solutions that are consistent with the underlying physics.

While PINNs have shown promise in solving both forward and inverse problems involving PDEs by taking advantage of the of the universal approximation theorem of neural networks,

they are also very sensitive to the choice of hyperparameters and can sometimes exhibit pathological gradient behaviour [6]. Another limitation of PINNs is that their accuracy decreases the farther we move from the boundary and the initial setup [7]. This makes PINNs a less than optimal tool for challenging, yet important problems such as turbulence [8].

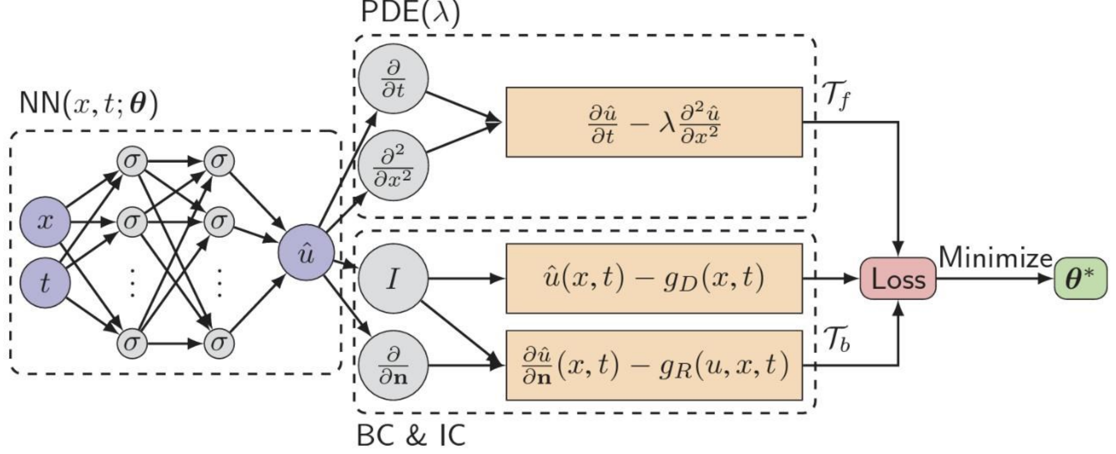


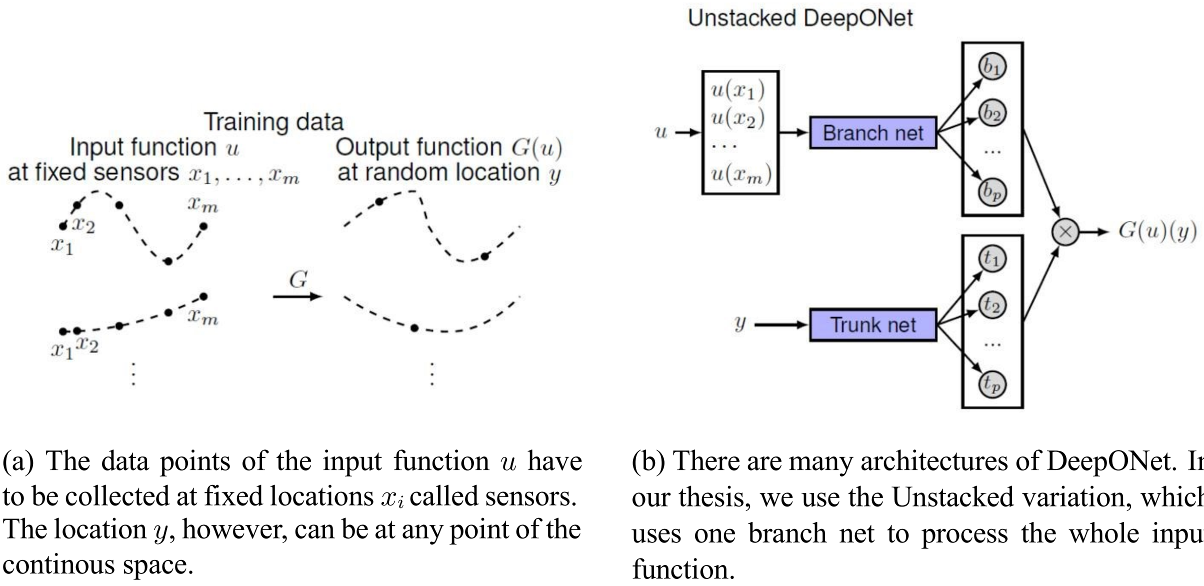
Figure 2.2: Schematic of a PINN for solving the diffusion equation with mixed boundary conditions (BC). The initial condition (IC) is treated as a special type of boundary condition. $\mathcal{T}^f$ and $\mathcal{T}^b$ denote the two sets of residual points for the equation and BC/IC [9].

2.2.2 Operator Learning and Deep Operator Networks (DeepONets)

Deep Operator Networks (DeepONets) are a type of neural network specifically designed to learn operators that map between function spaces. Unlike PINNs, which focus on solving individual PDE instances, DeepONets aim to learn the underlying operator that can generalize to different input functions and boundary conditions. DeepONets consist of a branch network that encodes the input function and a trunk network that encodes the output domain, connected by a merging function.

While PINNs were the first big step in approximating complex system via Deep Learning, one of their downsides is that each model is able to solve a single instance of a PDE, meaning that, should the boundary or initial condition change, the model would have to be trained again. Operator Learning (OL) refers to a distinct approach to solving PDEs by learning the underlying mappings between function spaces, rather than just approximating functions. There currently different approaches to OL, such as Fourier Neural Operators [10], which use the Fourier Transform to filter out noise for better approximation. In this thesis, howerever, we will focus on Deep Operator Networks (DeepONets) [11].

Let G be an operator taking an input function u, and then $G(u)$ is the corresponding output function. For any point y in the domain of $G(u)$, the output $G(u)(y)$ is a real number.



(a) The data points of the input function u have to be collected at fixed locations x_i called sensors. The location y, however, can be at any point of the continous space.

(b) There are many architectures of DeepONet. In our thesis, we use the Unstacked variation, which uses one branch net to process the whole input function.

Figure 2.3: Illustrations of the problem setup and architectures of DeepONets [11].

A DeepONet consists of two subnetwoks, called branch network and trunk network. The former takes the input function *u* as feature, while the latter's input is the output's domain point *y*. The branch network maps the input function in a lower-dimensional space in order to extract the relevant information of the solution space. The trunk network, in a certain way, can be viewed as an embedding process that, similarly to what Transformers model do, encapsulate the information of the spatial coordinate into a higher-dimensional space. The outputs of the two networks, which have the same dimension, are then merged through a merging function. In our case, this function is a simple dot product, which result in a scalar.

2.2.3 DeepONets vs PINNs

While DeepONets provide the incredible advantage of not needing to be retrained in the case of different geometries or initial conditions, at the base level they still need large amount of data to be trained. In paper [12], the advantages of OL and PINNs are merged in the framework of Physics-Informed DeepONets.

As will be described in more detail in Chapter 3, the model built in this thesis uses a PI-DeepONet as its basis.

2.3 Tensor Decompositions for Model Compression

2.3.1 Introduction to Tensors and Tensor Decompositions

The concept of tensor holds different meaning in different branches of science. The broader concept of tensor as a multilinear mapping between sets of algebraic objects were first popularized by Paduan mathematician Tullio Levi-Civita and his advisor Gregorio Ricci-Curbastro, who's contributions to tensor calculus were crucial in the development of the General theory of relativity.

In the context of machine learning, however, this term is used to describe multi-dimensional arrays, generalizing vectors and matrices. In other words, just like a vector can be seen as a one-dimensional array and a matrix can be seen as a two-dimensional array, an M-way tensor is an M-dimensional array. This concept is ubiquitous in machine learning: to represent an image, for example, we use a 3-d tensor, where the first two dimensions represent the spatial coordinates of the pixels, while the third dimension contains the values of the colors channel. By adding a fourth dimension to represent time, we can store a video and feed it to a neural network. While the input is usually flattened to a vector before processing, storing data in high-dimensional arrays is useful for various reasons.

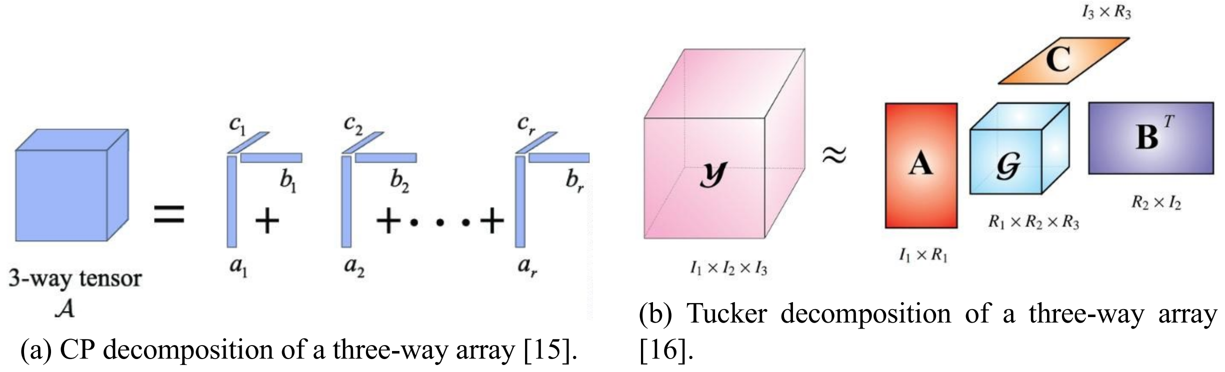
In our case, the idea behind model compression is to fold the weight matrices into high-dimensional tensors and the factorize them to reduce the number of parameters.

A well-known drawback of high-dimensional objects is the so-called *curse of dimensionality*, i.e. the fact that the number of elements scales exponentially with dimension. This leads to challenges in storing, processing, and training models with high-dimensional data such as many applications in Scientific Machine Learning. For this reason, efforts have been made to find ways to reduce the amount of data, even at the cost of a small approximation error.

Tensor decompositions [13] are a family of techniques that aim to factorize a high-order tensor into lower-order tensors. One of the first decompositions introduced is the CANDECOMP/PARAFAC (CP), which factorize the tensor into a sum of component rank-one tensors. For example, the third order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}_{I\times J\times}{}^{K}$ can be factorized as

$$\boldsymbol{\mathcal{A}} \approx \sum_{r=1}^{R} \mathbf{a^r} \circ \mathbf{b^r} \circ \mathbf{c^r} \tag{2.7}$$

Tucker decomposition [14] is another type of factorization which decomposes a tensor into a core tensor multiplied (or transformed) by a matrix along each mode. Thus, in the three-way case where $\boldsymbol{\mathcal{Y}} \in \mathbb{R}_{I\times J\times}{}^{K}$, we have



(a) CP decomposition of a three-way array [15].

(b) Tucker decomposition of a three-way array [16].

Figure 2.4: CP and Tucker decompositions of a three-dimensional tensor.

$$\mathcal{Y} \approx \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} = \sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} g_{pqr}\, \mathbf{a}_p \circ \mathbf{b}_q \circ \mathbf{c}_r \tag{2.8}$$

In this case, the tensor $\mathcal{G}$ is the three-dimensional core tensor, while **A**, **B**, and **C** are the factor matrices, which are usually orthogonal.

The Tucker decomposition is stable but has exponential in d number of parameters, $\mathcal{O}(dnr + r^d)$.

2.3.2 Tensor-Train Decomposition

Tensor-Train decomposition was first introduced in the field of Quantum Physics, where it's known as Matrix-Product State (MPS) representation [17]. The idea behind TT is to factorize an N-dimensional tensor into a chain-like product of matrices and 3-dimensional tensors.

Let $\mathcal{A} \in \mathbb{R}^{d_1 \times \cdots \times d_N}$, then its Tensor-Train Decomposition is given by

$$\mathcal{A}(i_1, \dots, i_N) = \mathbf{G}_1(i_1, r_1)\mathcal{G}_2(r_1, i_2, r_2) \dots \mathcal{G}_{N-1}(r_{N-2}, i_{N-1}, r_{N-1})\mathbf{G}_N(r_{N-1}, i_N) \tag{2.9}$$

In the equation above, $\mathbf{G}_1 \in \mathbb{R}^{d_1 \times r_1}$, $\mathbf{G}_N \in \mathbb{R}^{r_{N-1} \times d_N}$, and $\mathcal{G}_k \in \mathbb{R}^{r_{k-1} \times d_N \times r_k}$ for $k = 2, \dots, N-1$ are called TT-cores, while r_k for $k = 1, \cdots, N-1$ are called TT-ranks. This is the dimension of the bond index connecting one tensor in the chain to the next, and can vary from bond to bond. The bond dimension can be thought of as a parameter controlling the expressivity of a MPS/TT network. Any arbitrary tensor can be exactly represented with a TT-decomposition with bond dimension $m = d^{N/2}$. As our objective is to minimize the memory required to store the weights, however, it is possible to reduce the TT-rank, either heuristically or via adaptive rank selection [18], to approximate the initial tensor.

A tensor with N indices, each of dimension d, requires d^N parameters. In comparison, after compression, a tensor with TT-rank m only needs $m = Ndm^2$ parameters. It's evident that, as

N increases, TT-decomposition becomes more and more effective, which is why it's the type of compression we chose to compress our model.

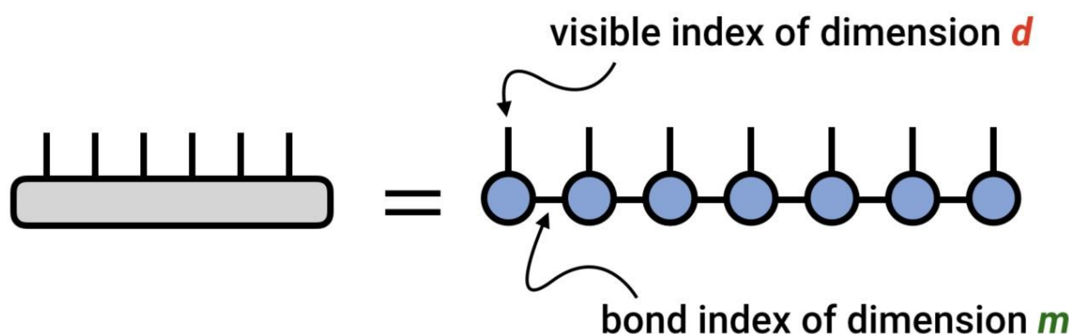


Figure 2.5: Tensor network diagram (Penrose diagram) of a Tensor-train decomposition. *Source: https://tensornetwork.org/.*

2.4 FPGA Acceleration for Deep Learning

2.4.1 FPGA Architecture and Advantages

Field-Programmable Gate Arrays (FPGAs) are integrated circuits which can be reconfigured after manufacturing. This means their functionality is not fixed, unlike Application-Specific Integrated Circuits (ASICs), and can be tailored to specific applications.

FPGAs contain Configurable Logic Blocks (CLBs) which are able to perform combinational and sequential logic, and a set of programmable interconnects. In addition to CLBs, FPGAs also contain specialized SRAM blocks (BRAM) and DSP blocks.

FPGAs offer several benefits compared to other kinds of integrated circuits. Compared to CPUs, for example, they offer a much higher degree of parallelism, so that, even considering the lower clock speed, they can perform operations such as Multiply-And-Accumulate (MAC) with lower latency and higher throughput. Although they are, on average, slower than GPUs of the same price range, they can tipically achieve better energy performance. This, together with the fact the they can work as standalone devices, make them a much better choice for on-device inference. While FPGAs do not achieve the same level of performance as ASICs, the time and economic investment required to achieve a final product is orders of magnitude less. You can simply buy an FPGA board off the shelf and program it, without worrying about the complex and expensive chip fabrication processes required for ASICs.

Moreover, newer FPGAs are System-On-Chip (SoC) devices which integrate CPU cores. This heterogeneous architecture allows for even greater flexibility and integration, enabling the

CPU to handle tasks like control flow, data management, and communication, while the FPGA fabric accelerates the computationally intensive parts of the application.

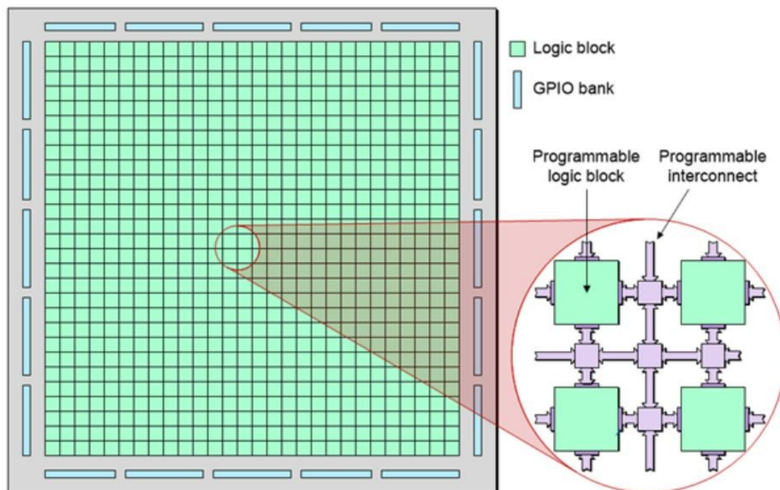


Figure 2.6: Simplified structure of an FPGA. *Source: https://www.latticesemi.com/.*

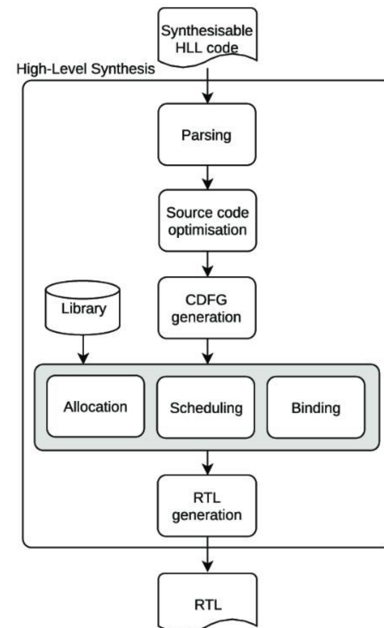


Figure 2.7: HLS design flow [19].

2.4.2 RTL Design vs High-Level Synthesis (HLS)

Traditionally, FPGAs are programmed through Hardware Description Languages (HDL) such as Verilog or VHDL. Unlike programming languages such as C/C++, they are not compiled into a sequence of instructions to be executed by a processor, but are instead synthesized into a network of logic circuits which directly configure the FPGA's hardware resources. HDLs are inherently concurrent, and their semantics operate on dataflow instead of control flow. The most common level of abstraction used to write HDL code is Register-transfer Level (RTL), which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals.

High-level Synthesis (HLS), in contrast, is an EDA method which brings the design process to a higher level of abstraction. HLS uses a high-level language such as C/C++, which is then transcompiled into RTL code. HLS tools handle the micro-architecture and transform untimed or partially timed functional code into fully timed RTL implementations, automatically creating cycle-by-cycle detail for hardware implementation [20].

HLS offers much faster design cycles, increasing the productivity by allowing engineers to

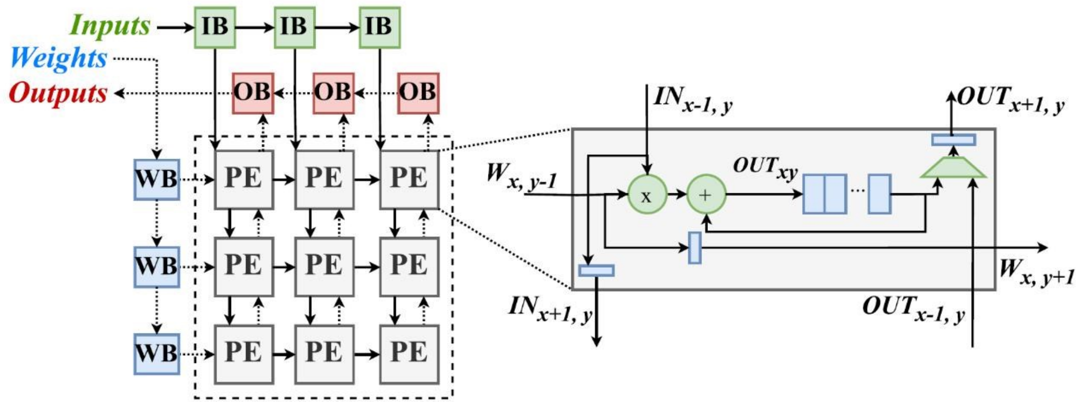


Figure 2.8: Systolic array architecture of a neural network layer [21].

focus less on the low-level details of hardware implementation. This, however, comes at a cost. RTL, compared to HLS, provides far superior fine-grained control over hardware, which usually results in better performance of the final result.

Another problem of HLS is that, since procedural languages are very different from HDLs, the programming style required to write HLS code is often clunky, and requires special directives like pragmas to help the compiler. Finally, a well-known drawback of HLS is that the code is very hard to debug, as the higher-level abstraction hides the hardware details to the programmer, making it hard to associate C code with hardware behaviour.

After considering the trade-offs of RTL and HLS, we decided to build the accelerator using the SystemVerilog HDL.

2.4.3 Systolic Arrays for Matrix Multiplication

A Systolic Array [22] is a parallel computer architecture, which consists of a set of homogeneous and interconnected processing elements (PEs), a controller module, and the on-chip memory/buffer. The PE is composed of basic arithmetic and register units, which can support a simple multiply-accumulate operation. PEs are interconnected, forming a grid through which data are processed and passed to the next PE of the grid. There are three typical dataflows in the systolic array design: Output Stationary (OS), Weight Stationary (WS) , and Input Stationary (IS). The "stationarity" means the elements of this tensor are not moved for the maximum duration of time throughout the computation, and it can describe the reuse situation of data. While systolic arrays have been around since the the 1970s, advancements of technologies in VLSI [23] have reignited new interest in this architecture.

Most modern applications have, in fact, become memory-bound rather than compute-bound. This means that the performance bottleneck is the speed of which is possible to access and move data. Systolic arrays are well-suited to address this challenge because they are inherently

designed for high data reuse.

Systolic arrays also achieve very high performance in the General Matrix Multiply (GeMM) operation, which is the algorithm to which FC and Convolutional Layers of Neural Networks can be reduced to. This reason, together with the modularity and the possibility of low-level optimization of this architecture makes it a good choice when designing with HDL, which is why we decided to choose it for our inference accelerator.

Chapter 3

Tensorized DeepONet with Quantization-Aware Training

3.1 Software Framework

In order to build and test our model, we used DeepXDE [9], a library for scientific machine learning which provides a user-friendly and flexible framework for implementing and training physics-informed models.

DeepXDE supports not only built-in primitive geometries such as `interval`, `triangle`, `rectangle`, `polygon`, `disk`, `cuboid`, and `sphere`, but other geometries can be constructed from these primitive geometries using three boolean operations: `union (|)`, `difference (-)`, and `intersection (&)`. This technique is called constructive solid geometry (CSG). CSG supports both 2D and 3D geometries.

Symbolic notation which follows TensorFlow's grammar is used to specify the PDE we want to train the model, and to set up boundary and initial conditions.

DeepXDE natively supports two types of networks: feed-forward neural network (FNN) and residual neural network (ResNet). None of these models, however, include tensor-compression nor quantization, which is why most of the work involved in building the model was integrating these functionalities into the library. We used DeepXDE v1.12.1 and PyTorch v2.5.0 as backend.

As previously mentioned in Section 2.2.1, the loss function is given by the sum of residual losses and boundary/initial condition losses. DeepXDE uses residual-based adaptive refinement (RAR) method to improve the training efficiency of PINNs when dealing with PDEs that exhibit solutions with steep gradients.

During training, DeepXDE uses a set of points from the domain (collocation points) where the residual and boundary conditions are evaluated. These points are sampled using a Gaussian

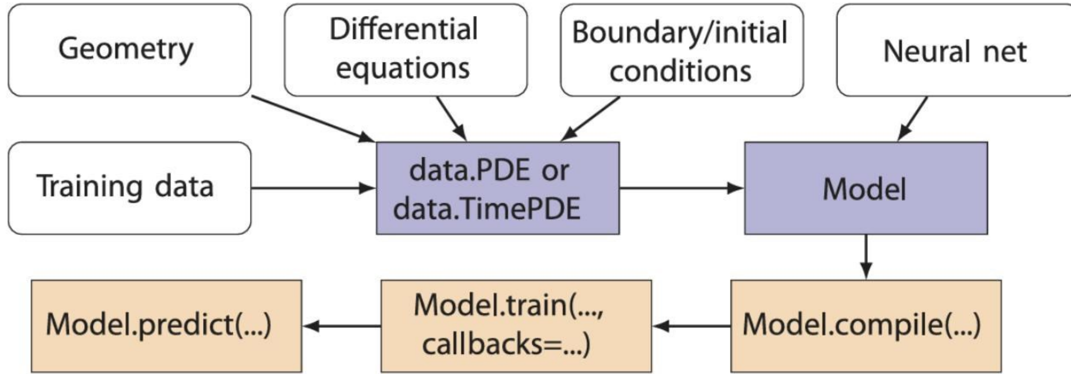


Figure 3.1: Flowchart of DeepXDE for solving differential equations. The blue boxes combine the PDE problem and training hyperparameters in the white boxes. The orange boxes are the three steps (from right to left) to solve the PDE [9].

Random Field (GRF). The random sampling algorithm is based on Cholesky decomposition of the covariance matrix.

The PDE is then evalued at a set of collocation points within the domain. The GRF samples serve as the input function u, and the PDE operator's evaluations at the collocation points provide the corresponding output values $G(u)(y)$. This forms the training dataset of input-output pairs.

The model is then compiled by specifying hyperparameters such as initializer, optimization algorithm, and learning rate. Finally, by calling the `model.train()` function and specifying training-related hyperparameters, the training process is executed, iterating over the training data in batches.

DeepXDE provides tools to monitor the training process, such as displaying the loss and metrics over epochs. After training, the DeepONet's performance is evaluated on a separate test dataset, which is also generated using GRFs and the PDE operator. Evaluation metrics, such as the L2 relative error, are used to assess the accuracy of the DeepONet's predictions.

3.2 Building a Tensorized DeepONet Model

The core the model lies in the `tensorized_layer`, a fully-connected layer which implements tensor compression and, as we will see later, quantization. When initializing a layer, a `config` object is passed as an argument. This object contains information regarding the tensor decomposition, like shape of the TT-cores and bond order. The actual factorization is done using the TensorLy library.

During the forward pass, the input to the network (a matrix or tensor) is reshaped into a format that is compatible with the TT-decomposition, matching the dimensions of the cores. The input tensor is then iteratively contracted with each TT-core. After contracting with all

cores, the output tensor is reshaped or reduced into its final form, which matches the expected output size of the layer. Our implementation also supports pruning, by specifying the pruning threshold.

The `tensorized_layer` is then used to build a FNN. Since the last layer does not need to be compressed, we implemented the option to put a non-compressed output layer. Similarly, we have the option to insert a non-compressed input layer. In all of our experiments, we used model where only the hidden layers were TT-compressed.

Finally, the DeepONet module is initialized by using two TT-compressed FNNs and by outputting the dot product of the trunk and branch network as result. It's very important to state that TT-compressed networks can be trained by using automatic differentiation algorithms.

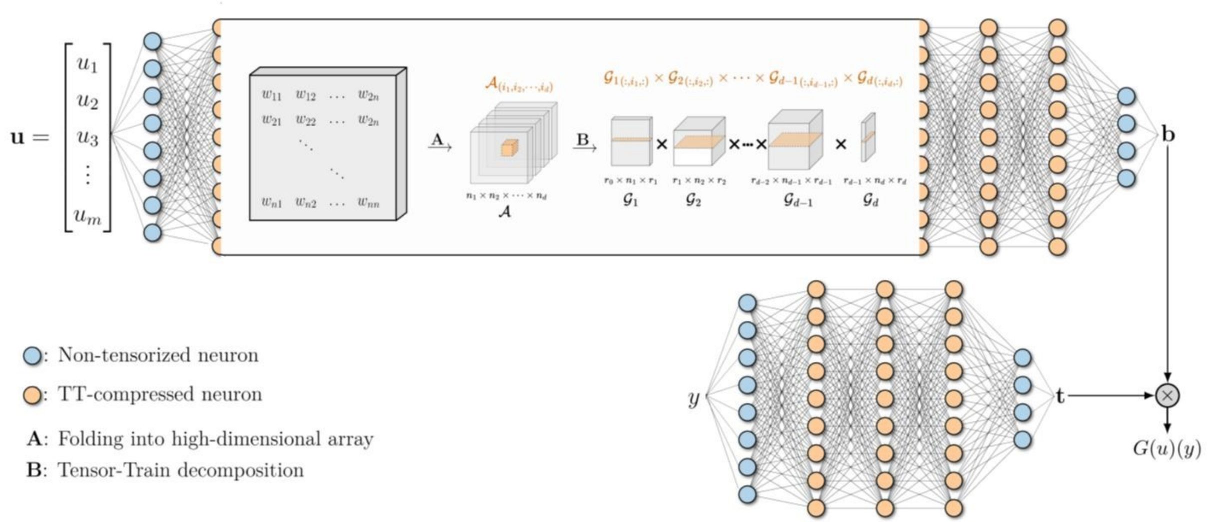


Figure 3.2: Schematic of a TT-compressed DeepONet.

3.3 Quantization-Aware Training

3.3.1 Introduction to Quantization

In standard neural network training, weights and activations are typically represented with 32-bit floating-point precision (FP32). This provides high numerical accuracy but is computationally expensive in terms of both memory and power consumption. Quantization [24] reduces this precision, often to 8-bit integers (INT8), which leads to significant speedups and lower resource requirements.

There are two main classes of quantization in NN: Post-Training Quantization (PTQ), and Quantization-aware training (QAT).

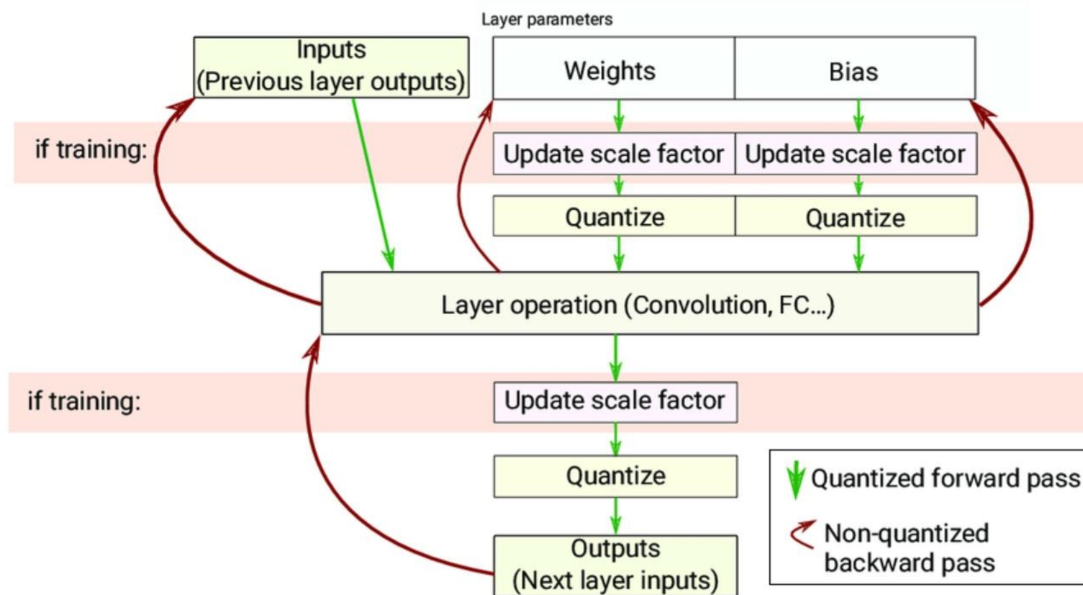


Figure 3.3: Flowchart of Quantization-Aware-Training process [25].

In PTQ, a model is first trained using full precision, and then the weights and activations are quantized to lower precision after training. This approach can lead to significant accuracy degradation, as the model was not trained to handle the quantization errors.

QAT addresses this issue by incorporating quantization during the training process itself. During QAT, the forward pass simulates the effects of quantization by representing weights and activations in lower precision, while the backward pass still computes gradients using full precision to ensure more stable updates. This allows the model to adjust its parameters in response to the quantization effects, leading to better accuracy after quantization.

The advantage of QAT is that it helps minimize the loss in accuracy caused by the transition from FP32 to lower precision, while still achieving the benefits of reduced computation and memory requirements. This makes it particularly valuable in deploying deep learning models on hardware with stringent resource constraints.

A drawback of QAT is that it introduces complexity, increasing training time and requiring careful tuning of bit-widths for accuracy. It may degrade model performance, albeit slightly, especially for precision-sensitive tasks, and demands more memory during training. Additionally, QAT can be challenging to debug and may lack compatibility with some optimizations and hardware, complicating deployment.

3.3.2 QAT in the Tensorized DeepONet

To perform Quantization-aware training (QAT), a customized PyTorch layer was used to extend DeepXDE-compatible models and to extend the library's functionalities. This FNN and

DeepONet implementations allows Tensor-Train (TT) and Tensor-Train Matrix (TTM) decompositions, quantization, and pruning.

There are two available quantization schemes: `floating-point` and `fixed-point`. If the former is chosen, the bit-width of the quantization can be decided by choosing how many bits to allocate to the exponent and to the mantissa. Similarly, if the latter is chosen, is it possible to choose how many bits to use for the integer and for the fractional part.

The rounding type can also be nearest or stochastic. Pruning is also possible. If pruning is activated, a pruning threshold must be specified.

3.4 Evaluation and Results

3.4.1 Experimental Setup

To train and test the model, we decided to focus on solving the reaction-diffusion system

$$\frac{\partial y}{\partial t} = D\frac{\partial^2 y}{\partial x^2} - ky^2 + v \tag{3.1}$$

This PDE was chosen due to its importance in various scientific and engineering applications, and because it's a very common benchmark, making it easy to compare it to other models.

For these experiments, we consider a spatial domain $x \in [0, 1]$ and a time domain $t \in [0, 1]$. We impose Dirichlet boundary conditions, specifying the values of u at the boundaries of the spatial domain. The initial condition, $u(x, 0)$, is given by a GRF centered at $x = 0.5$.

Training and testing data are generated by solving the advection equation numerically. This provides the ground truth solutions for a set of different initial conditions. The initial conditions for the training data are sampled from GRFs with a specified length scale to ensure a diverse range of input functions. The testing data is generated similarly, but with a separate set of initial conditions sampled from the same GRF distribution.

The performance of the DeepONet models is evaluated using the L2 relative error, which measures the normalized difference between the predicted and true solutions.

3.4.2 Evaluation

During the first part of the experiment, we tried different TT-ranks and learning steps to determine the best compromise to achieve a good compression rate while not losing too much accuracy.

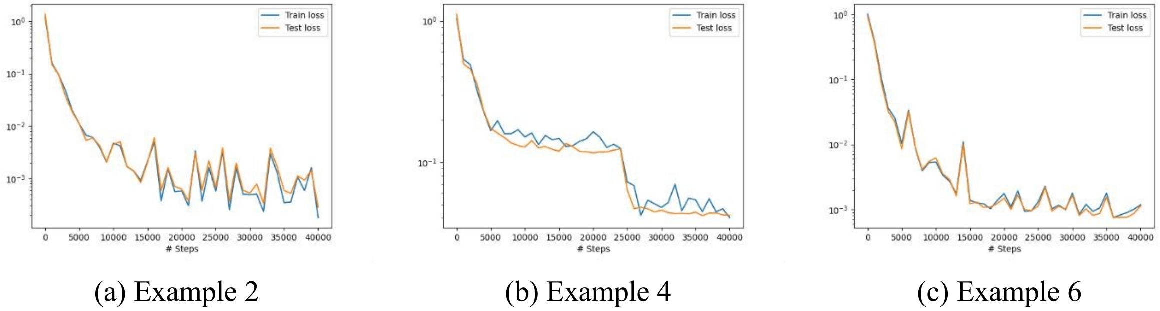
Our tests show that, while the non-compressed model achieves faster convergence than the TT-compressed ones, 40,000 steps are enough to stabilize the test and train loss for all of our models.

Regarding quantization, both 8-bit and 4-bit fixed-point quantization achieve comparable test loss, the former achieving lower L2 relative error than the full-precision model. It must be stated that in our experiments, we applied the same quantization to weights and activations. Further improvements might be reached when employing higher precision activation and lower precision weights.

Training time also varies greatly depending on hyperparameters, going from $896s$ in the case of Example 1 in Table 3.1 to $1785s$ for Example 4 in Table 3.2, while running the code on an Intel Core Ultra 7 155H @3.80GHz.

Example	Learning rate	TT-rank	Compression rate	Training steps	Train loss	Test loss
1	0.0004	-	1	40,000	6.22e-04	6.70e-04
2	0.0004	-	1	40,000	1.80e-04	2.84e-04
3	0.0006	4	0.039	40,000	8.75e-02	8.25e-02
4	0.0003	4	0.039	40,000	4.04e-02	4.18e-02
5	0.0006	8	0.084	40,000	1.28e-03	1.33e-03
6	0.0003	8	0.084	40,000	7.46e-04	7.49e-04

Table 3.1: Training and test loss results for models with different TT-rank and learning rates.



(a) Example 2 (b) Example 4 (c) Example 6

Figure 3.4: Training and test loss of examples in Table 3.1.

Example	Layer depth	Layer width	TT-rank	Compression rate	Quantization	L2 relative error
1	3	128	-	1	Full-precision	-
2	3	128	-	1	Full-precision	0.0171
3	3	128	8	0.084	Full-precision	0.0206
4	3	128	8	0.084	Fixed-point 8bit	0.0139
5	3	128	8	0.084	Fixed-point 4bit	0.0339

Table 3.2: Comparison among best simulation results for different quantizations.

Chapter 4

FPGA Accelerator Design

4.1 System Overview and Design Goals

Since the design of this accelerator is inherently modular and customizable, and does not have any hard constraints, it is well suited for any kind of FPGA. In our preliminary tests, we used an Arty A7-100T developement board [26], which features a Xilinx XC7A100TCSG324-1 FPGA. The next steps of the project will involve deployment on a bigger board, such as the Zynq-7000 SoC. On this board we could leverage the ARM CPU Core to facilitate communication between a Host and the accelerator, and employ the design on an edge device.

The design of the FPGA accelerator aims to achieve the following objectives: (1) Maximize throughput for tensor contractions, achieving a significant speedup compared to CPU and GPU implementations. (2) Optimize resource utilization, minimizing the usage of FPGA logic elements, BRAM blocks, and DSP slices. (3) Minimize power consumption to enable deployment in power-constrained environments, making the accelerator suitable for edge computing applications.

4.1.1 From contractions to matrix multiplications

While, TT-decomposition offers a great amount of data compression, it also adds complexity of computation, as it requires tensor contractions instead of classical matrix-multiplications (matmuls).

Matmuls are always performed along prefixed directions, but tensor contractions aperate along different modes, which adds the need of a "slicer/permuter" which fetches tensor elements in correct order. The order of the intermediate tensor also change as we contract with the TT-cores, thus needing to same more metadata. The main challenge lies in efficiently handling memory accesses. CPU and GPU-oriented algorithms for native tensor contraction don't reach

the arithmetic intensity as GEMM routines such as BLAS, because depending on the modes across contractions are performed, they might require suboptimal cache accesses. On the other hand, as shown in [27] (Example 5.4),

Remark 1 *The tensor contraction* $\boldsymbol{\mathcal{Y}}^{m_1 m_2 n_1 n_2} \leftarrow \boldsymbol{\mathcal{W}}^{m_1 m_2 k_1 k_2} \boldsymbol{\mathcal{X}}^{k_1 k_2 n_1 n_2}$ *can be interpreted as the matrix-matrix multiplication* $\mathbf{Y}_{(m_1 m_2)(n_1 n_2)} \leftarrow \mathbf{W}_{(m^1 m^2)(k^1 k^2)} \mathbf{X}_{(k^1 k^2)(n^1 n^2)}$.

This means that we can still use the known GeMM implementations for contracting these tensors. Since the order of the contractions is known beforehand, and FPGAs allow for very fine control over the contents and the partitioning of memory, it is possible to order the modes so that we can fetch data sequentially and use architectures that are highly efficient like the systolic array. It should be noted that unrolling high-dimensional tensors on 1 or 2 modes generates very narrow matrices, so some optimizations should be performed on this aspect.

In paper [18], a good contraction sequence is found to be

$$\boldsymbol{\mathcal{T}}^1 = \text{einsum}(bn^1...n^d, n^1...n^d r^d \Rightarrow br^d, [\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{A}}^d]) \tag{4.1a}$$

$$\boldsymbol{\mathcal{Y}} = \text{einsum}(br^d, r^d n_{d+1}...n_{2d} \Rightarrow bn_{d+1}...n^{2d}, [\boldsymbol{\mathcal{T}}^1, \boldsymbol{\mathcal{B}}^d]) \tag{4.1b}$$

Where tensors $\boldsymbol{\mathcal{A}}^d$ and $\boldsymbol{\mathcal{B}}^d$ are:

$$\boldsymbol{\mathcal{A}}^d := \boldsymbol{\mathcal{G}}^1 \times \cdots \times \boldsymbol{\mathcal{G}}^d \tag{4.2a}$$

$$\boldsymbol{\mathcal{B}}_d := \boldsymbol{\mathcal{G}}_{d+1} \times \cdots \times \boldsymbol{\mathcal{G}}_{2d} \tag{4.2b}$$

It should be noted that the authors of this paper proposed a training algorithm, and as such, intermediate results $\boldsymbol{\mathcal{A}}^i, \boldsymbol{\mathcal{A}}_{-i}, \boldsymbol{\mathcal{B}}^i, \boldsymbol{\mathcal{B}}_{-i}$ are stored. Another thing that should be taken into account is that this sequence is optimal when that batch dimension b is big.

Since in the current state of the project we are mainly interested in inference, tensors $\boldsymbol{\mathcal{A}}^d$ and $\boldsymbol{\mathcal{B}}^d$ can be precomputed and stored in BRAM. Regardless, the design process of this accelerator will be done while keeping in mind future possible implementations of training.

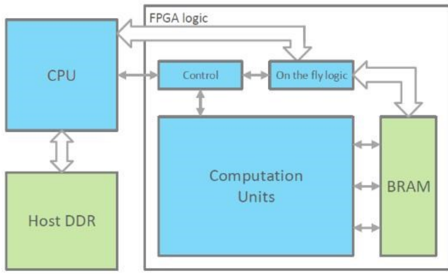


Figure 4.1: Dataflow of system communication.

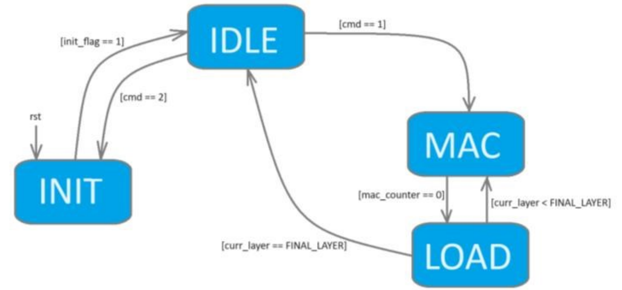


Figure 4.2: FSM of control sequence

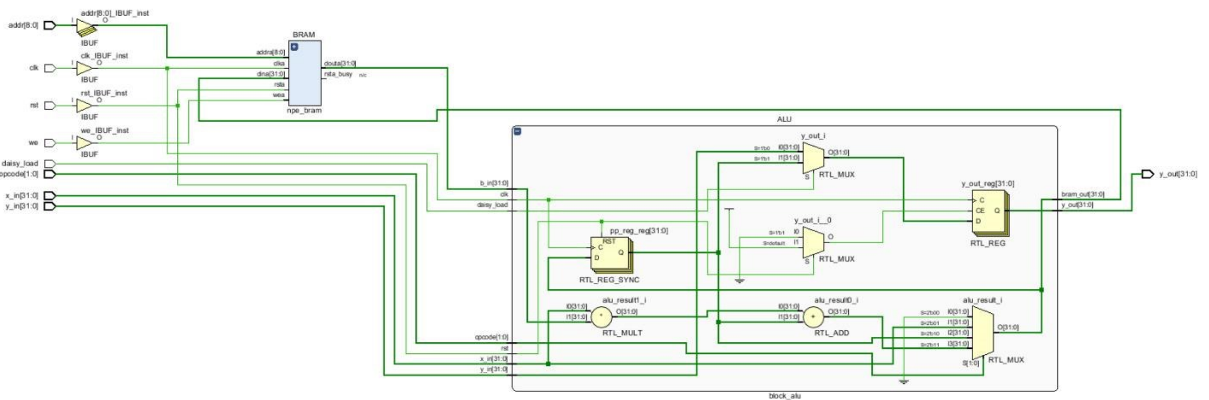


Figure 4.3: RTL Schematic of a PE with BRAM.

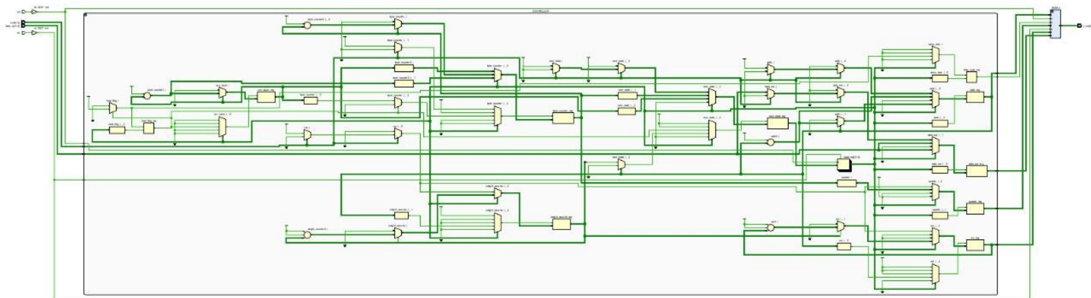


Figure 4.4: RTL Schematic of the Controller unit.

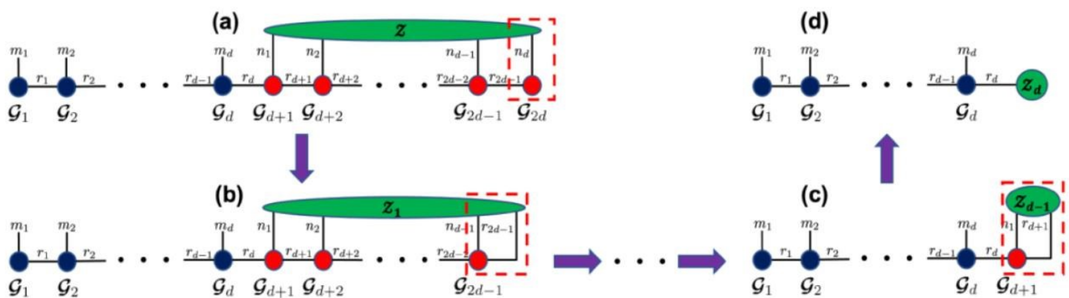


Figure 4.5: Contraction sequence of a tensorized layer.

4.2 Tensor Contraction Unit (TCU)

The core of the FPGA accelerator is a systolic array-based Tensor Contraction Unit (TCU) designed to efficiently perform the one-mode and two-mode tensor contractions which are mapped as GeMMs.

The TCU is composed of a customizable Weigh Stationary (WS) systolic array, a Controller IP, and a set of BRAMs to store the weighs. One of the biggest challenges that accelerators face is the limited amount of BRAM, forcing the weights to be stored on DRAM, which is costly to access. By choosing the smallest possible BRAM size, it is possible to gain many concurrent access ports, which maximizes the parallelism.

Figure 4.1 shows the overall system design. The Control unit is connected to the Host using an AXI4-Lite interface, which allows for direct control over the prototype.

4.2.1 Processing Element Design

The Processing Element (PE) is made of a MAC unit and a set of registers to store the intermediate results. Each PE is connected to the controller to communicate the opcode. The PE also has an input port to read from the BRAM block, where the quantized weights are stored (Figure 4.3).

The size of the adders and multipliers is parametrized, so it's possible to deal different types of quantization. PEs also have 2 input and 2 output ports to connect to other PEs in the grid: `north`, `west`, `south`, `east`. The units at the edge of the grid are also connected to the Control unit, so that it can read and write data.

4.2.2 Dataflow and Control Logic

The dataflow performs the sequence of contractions as in Figure 4.5, until the final vector is obtained. Intermediate results are stored in a dedicated memory inside the Control Unit. The Finite State Machine of the Control Unit is shown in Figure 4.2. There are four main states:

- `INIT`: This is the initial state, responsible for setting up the system. In this state, the first set of weights is loaded in the registers of the PEs. From this state, the FSM transitions to `IDLE` if an `init_flag == 1` condition is met.

- `IDLE`: The FSM waits in this state, waiting for a command. If the TCU receive the flag `cmd == 1`, it moves the FSM to the `MAC` state.

- `MAC`: In this state, the GeMM is performed. Data fed from the Control Unit is processed by the PEs which pass the partial results to the next elements of the grid, until the operation is completed. At this point, `mac_counter == 0` (indicating the current MAC operation is complete) moves the FSM moves to `LOAD`.

If `curr_layer < FINAL_LAYER` (indicating more layers to process), it loops back to `IDLE` after completing a MAC.

- `LOAD`: After a contraction is completed, a new set of weights must be fetched from memory. This state handles the memory accesses. Once data is loaded, the FSM transitions back to `MAC` to continue processing. If `curr_layer == FINAL_LAYER`, the FSM returns to `IDLE`, signaling that all layers have been processed and no further computation is required.

4.3 Simulation and Verification

Design RTL code was written in SystemVerilog using Vivado 2019.1. Each module was first tested with a testbench in a behavioral simulation. The systolic array and control unit were also monitored and tested using an Integrated Logic Analyzer (ILA), which provides accurate timing analysis tools.

Due to time constraints, we were unable to fully deploy the accelerator. We were, however, able to simulate and test a tensor contraction handled by the Control Unit and a small 3x3 array, as shown in Figure 4.6.

Figure 4.6: Waveform of simulation of a matrix-matrix multiplication of a small-scale array.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

This thesis has presented a novel framework, TT-DeepONet, for efficiently solving partial differential equations (PDEs) on resource-constrained devices. The framework addresses the challenges of memory and computational complexity associated with Deep Operator Networks (DeepONets) by incorporating two key optimizations: tensor-train (TT) decomposition and quantization-aware training (QAT). The TT decomposition significantly reduces the number of parameters in the DeepONet's weight matrices, leading to a smaller memory footprint and faster computations. Furthermore, QAT allows for training the model with reduced numerical precision, further enhancing memory and computational efficiency without significant loss of accuracy.

Another key contribution is the design of an FPGA-based accelerator intended to efficiently handle the tensor contraction operations in TT-DeepONet. While not fully implemented, the design incorporates a systolic array architecture to execute tensor contractions efficiently as matrix multiplications, overcoming challenges inherent to high-dimensional tensor operations. By mapping tensor operations into matrix multiplication workflows, the FPGA design aims to minimize memory access bottlenecks and ensure fast, low-power computations suitable for edge devices. The proposed architecture is modular and flexible, enabling further customization for different FPGA models and offering scalability for various operator learning tasks.

The experimental results presented in this thesis demonstrate the effectiveness of the TT-DeepONet framework in solving the Reaction-Diffusion equation. The framework achieved comparable accuracy to a standard DeepONet while requiring substantially less memory and achieving faster inference times. The FPGA accelerator further enhanced performance, enabling real-time on-device inference for solving PDEs on resource-constrained platforms. This work contributes to the growing field of hardware-aware machine learning, paving the way for de-

ploying complex deep learning models on edge devices for scientific computing applications.

5.2 Future Directions

While the preliminary results presented in this thesis are promising, several avenues for future work can further enhance the TT-DeepONet framework. One immediate next step is to complete the deployment of the FPGA accelerator and thoroughly evaluate its performance on the target Zynq board. This involves integrating the accelerator with the ARM processor on the board, optimizing data transfer and communication between the processor and the FPGA fabric, and implementing the necessary drivers and software interfaces. This full deployment will enable a more realistic assessment of the accelerator's performance and power efficiency in a real-world setting.

Bibliography

[1] Z. Liu, Y. Li, J. Hu, *et al.*, "Deepoheat: Operator learning-based ultra-fast thermal simulation in 3d-ic design," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2023, pp. 1–6.

[2] J. W. Thomas, *Numerical partial differential equations: finite difference methods*. Springer Science & Business Media, 2013, vol. 22.

[3] C. A. Felippa, "Introduction to finite element methods," *University of Colorado*, vol. 885, 2004.

[4] R. Eymard, T. Gallouët, and R. Herbin, "Finite volume methods," *Handbook of numerical analysis*, vol. 7, pp. 713–1018, 2000.

[5] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational physics*, vol. 378, pp. 686–707, 2019.

[6] S. Wang, Y. Teng, and P. Perdikaris, "Understanding and mitigating gradient pathologies in physics-informed neural networks," *arXiv preprint arXiv:2001.04536*, 2020.

[7] V. Dwivedi and B. Srinivasan, "Physics informed extreme learning machine (pielm)–a rapid method for the numerical solution of partial differential equations," *Neurocomputing*, vol. 391, pp. 96–118, 2020.

[8] X. Jin, S. Cai, H. Li, and G. E. Karniadakis, "Nsfnets (navier-stokes flow nets): Physics-informed neural networks for the incompressible navier-stokes equations," *Journal of Computational Physics*, vol. 426, p. 109 951, 2021.

[9] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, "Deepxde: A deep learning library for solving differential equations," *SIAM review*, vol. 63, no. 1, pp. 208–228, 2021.

[10] Z. Li, N. Kovachki, K. Azizzadenesheli, *et al.*, "Fourier neural operator for parametric partial differential equations," *arXiv preprint arXiv:2010.08895*, 2020.

[11] L. Lu, P. Jin, and G. E. Karniadakis, “Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators,” *arXiv preprint arXiv:1910.03193*, 2019.

[12] S. Wang, H. Wang, and P. Perdikaris, “Learning the solution operator of parametric partial differential equations with physics-informed deeponets,” *Science advances*, vol. 7, no. 40, eabi8605, 2021.

[13] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[14] L. R. Tucker, “Some mathematical notes on three-mode factor analysis,” *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[15] L. Wang, K. Xie, T. Semong, and H. Zhou, “Missing data recovery based on tensor-cur decomposition,” *IEEE Access*, vol. PP, pp. 1–1, Nov. 2017. doi: `10.1109/ACCESS.2017.2770146`.

[16] L. Billiet, T. Swinnen, K. de Vlam, R. Westhovens, and S. Huffel, “Recognition of physical activities from a single arm-worn accelerometer: A multiway approach,” *Informatics*, vol. 5, p. 20, Apr. 2018. doi: `10.3390/informatics5020020`.

[17] D. Perez-Garcia, F. Verstraete, M. M. Wolf, and J. I. Cirac, “Matrix product state representations,” *arXiv preprint quant-ph/0608197*, 2006.

[18] Z. Yang, S. Choudhary, X. Xie, C. Gao, S. Kunzmann, and Z. Zhang, “Comera: Computing-and memory-efficient training via rank-adaptive tensor optimization,” *arXiv preprint arXiv:2405.14377*, 2024.

[19] M. Numan, B. Phillips, G. Puddy, and K. Falkner, “Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains,” *IEEE Access*, vol. 8, pp. 174 692–174 722, Jan. 2020. doi: `10.1109/ACCESS.2020.3024098`.

[20] J. Cong, J. Lau, G. Liu, *et al.*, “Fpga hls today: Successes, challenges, and opportunities,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–42, 2022.

[21] X. Wei, C. H. Yu, P. Zhang, *et al.*, “Automated systolic array architecture synthesis for high throughput cnn inference on fpgas,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[22] R. Xu, S. Ma, Y. Guo, and D. Li, “A survey of design and optimization for systolic array-based dnn accelerators,” *ACM Computing Surveys*, vol. 56, no. 1, pp. 1–37, 2023.

[23] G. Molas and E. Nowak, “Advances in emerging memory technologies: From data storage to artificial intelligence,” *Applied Sciences*, vol. 11, no. 23, p. 11 254, 2021.

[24] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, “A white paper on neural network quantization,” *arXiv preprint arXiv:2106.08295*, 2021.

[25] P.-E. Novac, G. Boukli, A. Pegatoquet, B. Miramond, and V. Gripon, *Quantization and deployment of deep neural networks on microcontrollers*, May 2021. doi: `10.48550/arXiv.2105.13331`.

[26] *Artix 7 FPGAs Data Sheet: DC and AC Switching Characteristics*, DS181 (v1.27.1), Product Specification, AMD Adaptive Computing, 2024. [Online]. Available: `https://docs.amd.com/v/u/en-US/ds181_Artix_7_Data_Sheet`.

[27] P. Springer, “High-performance tensor operations: Tensor transpositions, spin summations, and tensor contractions,” Ph.D. dissertation, Dissertation, RWTH Aachen University, 2019, 2019.