Leveraging Julia programming language for efficient Automatic Differentiation

Kacper Chabros
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
kacper.chabros.stud@pw.edu.pl

Stanisław Horodecki
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
stanislaw.horodecki.stud@pw.edu.pl

Wiktor Łodyga
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
wiktor.lodyga@pw.edu.pl

***Abstract*—Automatic differentiation is widely used in machine learning and scientific computing, making its efficient implementation important for practical applications. This study presents different layer-wise optimization methods involving metaprogramming and loop unrolling implemented in a custom library written in Julia. These techniques managed to reduce the execution time of im2col-based convolution by 25% and reduce memory overhead. The proposed library was evaluated on a convolutional neural network and compared with PyTorch and Flux, and achieved nearly threefold speedup over both reference frameworks, while maintaining low memory overhead. These outcomes showcase how leveraging Julia's features can result in efficient, well-optimized code.**

***Index Terms*—Machine Learning, Optimization, Automatic Differentiation, Julia Language, Metaprogramming**

I. Introduction

Automatic Differentiation (AD) [1] is a core technology in modern machine learning and scientific computing. Unlike numerical differentiation based on finite differences, automatic differentiation evaluates derivatives without truncation error, subject only to floating-point rounding errors. Machine learning algorithms depend on AD to compute gradients via backpropagation to train neural networks by updating their weights. Apart from artificial intelligence, AD is used to solve complicated optimization problems where calculating gradients quickly and accurately is important.

Julia programming language provides a suitable framework for developing advanced AD packages. It is conceptually similar to Python, but with the efficiency of C, which allows complex implementations to be written quickly and without sacrificing the performance. One important feature of Julia is multiple dispatch, which selects a method based on the types of all function arguments. However, when developing efficient programs in Julia, programmers need to pay attention to avoid dynamic dispatch. It happens when the Julia compiler cannot determine the types of parameters and forces the runtime system to resolve them dynamically. In heavy iterative processes, this can cause significant performance drops due to constant type checking. The solution to this problem involves strict type stability and usage of parametric types enabling well optimized machine code generation.

This paper presents a custom reverse-mode automatic differentiation library implemented in Julia. We investigate the impact of pre-allocation, in-place operations, SIMD vectorization, and loop unrolling on the performance of neural-network layers. The resulting implementation is evaluated on a convolutional neural network. It is compared with both Flux.jl and PyTorch in terms of execution time, and with Flux.jl in terms of memory allocation.

II. Related Work

Automatic differentiation and its use in machine learning are surveyed by Baydin et al. [1], who distinguish AD from symbolic and finite-difference differentiation and discuss forward- and reverse-mode implementations. Reverse-mode AD underlies gradient computation in widely used machine-learning frameworks, including PyTorch [2] and the Flux.jl ecosystem [3]. General optimization techniques relevant to the implementation include instruction- and data-level parallelism. Hennessy and Patterson [4] describe loop unrolling as a means of reducing branch overhead and pipeline stalls, while Flynn's SIMD taxonomy [5] provides the basis for applying one instruction across multiple data elements. Convolution is a major computational bottleneck in neural networks. The im2col transformation [6] maps sliding-window convolution to optimized matrix multiplication at the cost of additional memory. More recent work includes the memory-efficient im2win method [7], FFT-based convolution [8], optimized direct convolution on ARM processors [9], and per-layer tuning of direct convolution on multicore processors [10].

Julia was introduced as a high-level language for technical computing that combines expressive programming with performance [11]. Its multiple dispatch and dataflow type inference allow type-stable for loops to be compiled efficiently, avoiding the mandatory vectorization and two-language patterns discussed by Bezanson et al. [12].

III. Methodology and Implementation

A. Approach

The library implements reverse-mode automatic differentiation, which is well suited to computing gradients of a scalar loss function with respect to many model parameters. The forward pass evaluates the graph nodes in topological order, while the reverse pass initializes the loss gradient to 1 and