UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor Thesis

Shatin Nguyen, Theis Baasch and Oliver Raaschou

Empirically comparing high level and low level AD

Advisor: Troels Henriksen

Handed in: June 10, 2025

Abstract

Automatic Differentiation (AD) is a critical technology for gradient-based optimization in machine learning and scientific computing. AD systems can be broadly categorized as high-level, which operate on source code and can leverage semantic information, or low-level, which operate on generic intermediate representations (IRs) like LLVM IR. While both paradigms have theoretical advantages, their empirical performance trade-offs for are not well understood. This thesis addresses that gap by empirically comparing the runtime performance of Futhark's high-level, native AD system against a low-level approach using Enzyme on Futhark-generated LLVM IR. We conduct a systematic comparison across a curated suite of benchmarks. Our results reveal a distinct performance trade-off dictated by the workload's computational pattern. We found that while low-level AD excels on computationally simple programs, high-level AD provides a significant performance advantage — up to 10x — for memory-bound workloads dominated by large matrix operations. We conclude that a high-level AD system's ability to apply semantic transformations offers a reliable and significant performance advantage for a wide range of programs.

Contents

Introduction

Automatic Differentiation (AD) has in the recent years gained more attraction across numerous scientific and engineering disciplines. Its most prominent usage is in modern machine learning, where it can be used in training models through efficiently computing gradients. AD techniques offer a systematic way to compute the derivatives of functions expressed as computer programs, providing an alternative to more error prone manually implementing derivations and less precise numerical approximation.

Motivation and Problem statement

The methodologies for implementing AD are diverse, often distinguished by the level of program representation at which the differentiation transformation is applied. High-level AD tools can operate directly on, or close to, the source code allowing them to take advantage of informative semantic and domain-specific knowledge to enhance in order to optimize the gradient function. Futhark is a high-level functional language which implement such AD tools and can effectively exploit and transform programs structures to produce efficient code. Conversely, low-level AD systems for example Enzyme AD operate on low-level intermediate language (in our case LLVM IR).

While both methods present distinct theoretical advantages, a clear understanding of their empirical performance trade-offs is crucial. The choice of AD strategy can influence the efficiency of the generated derivative code, especially concerning how the original program is preserved, transformed, or newly exploited in the differentiated version. This thesis addresses the core problem of empirically comparing the runtime performance and compilation overhead of high-level AD, as implemented in Futhark's native system, versus low-level AD, using Enzyme applied to Futhark-generated LLVM IR, for a suite of Futhark programs. Furthermore, it seeks to investigate the underlying factors contributing to any observed performance differences.

Hypothesis

Our hypothesis is that it is advantageous to perform AD at a higher level of abstraction, since high level languages expose semantic and structural information that can lead to the generation of more efficient derivatives, whereas low-level representations often obscure such information.

Empirical Benchmarking and Scope

To test our hypothesis, this thesis uses empirical benchmarking. This approach involves a systematic execution of selected programs suitable for differentiation using Futharks's AD implementation versus Enzyme AD. The performance of these differentiated versions is then quantitatively measured and compared. The analysis subsequently focuses on interpreting these performance results, aiming to identify the reasons for observed differences by relating them to the characteristics of the AD tools and the benchmarks themselves.

The scope of this project:

1. An empirical performance comparison between Futhark's native AD system and the Enzyme AD tool.
2. A selection of Futhark benchmarks.
3. The measurement and analysis of runtime execution performance.
4. An investigation into the reasons for performance differences, with particular attention to the handling and transformation of program structures.
5. Documentation of the practical challenges encountered in setting up and utilizing AD in Futhark and using Enzyme.

This project does not aim to:

1. Conduct an exhaustive survey or comparison of all available AD tools or techniques.
2. Develop novel AD algorithms or new compiler optimizations for either Futhark or Enzyme.
3. Perform a deep formal verification of the AD transformations, beyond what is necessary to ensure correctness for the purpose of performance comparison.

Key Findings and Contributions

Our empirical investigation reveals a distinct performance trade-off dictated by the workload's computational pattern. We find that while low-level AD excels on computationally simple programs, Futhark's high-level AD provides a significant performance advantage (up to 10x) for memory-bound computations. Furthermore, we demonstrate that high-level AD's ability to perform structural transformations provides noticeable results when compared to low-level AD at certain problem scales.

Background

Automatic Differentiation (AD)

Automatic Differentiation (AD) is a method for computing the derivatives of functions written as computer programs. Unlike symbolic differentiation, which creates a new mathematical formula, AD computes the derivative's exact numerical value during the program's execution. This makes it a practical and precise alternative to both symbolic and numerical methods. AD has two main approaches: forward mode and reverse mode. The choice between them is critical for performance. Forward mode is typically more efficient for functions with fewer inputs compared to outputs, while reverse mode is better suited for functions with more inputs compared to outputs [1].

Forward mode

Forward mode in AD will find the derivative of a function f with respect to x_1 by first giving each intermediate values v_i a derivative $v_i' = \frac{\partial v_i}{\partial x_1}$. Then applying chain rule on each elementary operations (often called forward primal trace) gives the tangent v_i'. Then evaluate v_i and the tangent v_i' gives the derivative $v_j = \frac{\partial v_i}{\partial v_1}$. Forward mode will require computing the Jacobian of a function $\mathbb{R}^n \to \mathbb{R}^m$ where n is the inputs of x_i and m is the outputs of y_i. Each evaluation generate one column in the Jacobian and the full Jacobian can be computed in n evaluations. Forward mode is more preferable when $n \ll m$ [1].

Reverse mode

In reverse mode the derivative is calculated in two phases. First phase is running the original program and set the intermediate values v_i while keep track of dependencies. The second phase calculate the derivative by propagating the adjoint in reverse. Reverse mode is preferable when $m \ll n$[1].

Futhark

Futhark is a high-level, purely functional, data-parallel programming language. It is specifically designed to facilitate the creation of high-performance parallel programs that can be efficiently executed on contemporary hardware, particularly Graphics Processing Units (GPUs). Its programming model is rooted in functional principles and often emphasizes nested parallelism, allowing for the expression of parallel computations within other parallel computations. The language is intentionally kept small to enable a compiler that can apply numerous aggressive optimizations. Futhark has built-in functionality for AD both for forward mode and reverse mode. Futhark has a built-in backend for generating C code as well as GPU code, corresponding to code written in Futhark [8].

LLVM

LLVM is a modern compiler infrastructure supporting a variety of programming languages. It provides an optimizer that attempts to improve the runtime of code at the cost of compile time. This project primary focuses on using the LLVM optimizer on C and C++ programs. LLVM can also represent code in form of LLVM intermediate representation (IR) providing a human readable instruction set. [9].

Enzyme

Enzyme AD is an AD tool which is implemented as a plugin for the LLVM compiler. Enzyme performs AD by directly transforming the LLVM IR of a program. Here it analyzes the LLVM IR of the given function and then generates new LLVM IR which computes the functions derivative, here supporting both forward- and reverse mode. The design goal of Enzyme is to operate on optimized LLVM code which can make simple and optimized gradients.

Enzyme is relevant to this study since it operate on low-level LLVM IR and differentiates code after compiler optimizations have been performed and when the program is in LLVM IR. Even though this allows Enzyme to operate on compiler optimized code, it also means that the code Enzyme have limited knowledge about the original high-level code and sematic information. In practice a trade-off is that some functions cannot be re-computed and if it that function is needed for calculating the derivative, Enzyme needs to handle these, and possibly introduce overhead instructions to do so [10].

Methodology and Implementation

Experimental Setup

The experiments were conducted with these specifications:

Hardware	
CPU:	Apple M1
GPU:	Apple M1
RAM:	16 GB

Software	
Operation System:	MacOS Sequia 15.5
Futhark Compiler:	0.25.31. Compiled with GHC 9.12.2.
LLVM Toolchain:	Version 19.1.7
Enzyme:	Version 0.0.180
Key Compiler flags:	-O2 -fno-vectorize -fno-slp-vectorize -fno-unroll-loops

Benchmark selection

The benchmarks used in this study were from gradBench [4] and Futhark's AdBench [[3]] to provide a robust basis for comparison. These algorithms provde a wide range of algorithms with different applications of AD. They include scenarios where both forward mod and reverse mode are used in different ways.

The specific benchmarks are described below:

1. Dot Product: Computes the dot product of two vectors. While simple, it serves as a fundamental building block and a basic test case for AD.
2. Matrix multiplication: A fundamental linear algebra operation computing the multiplication of two matrices. It is a classic example of a highly data-parallel computation and is commonly used in many scientific and machine learning fields.
3. Gaussian Mixture Model (GMM): This benchmark implements a Gaussian Mixture Model, a probabilistic model used in machine learning [14].
4. Bundle Adjustment (BA): Is an algorithm in used in computer vision for 3D reconstruction [14].
5. Hand Tracking (HT): HT is used to track a real hand by observing the depth and fitting the model to depth information [14].
6. K-Means Clustering (KMeans): An iterative algorithm that partitions a dataset into K distinct, non-overlapping clusters. Its main computational steps, assigning points to clusters and recalculating cluster centroids [12].

7. Long Short-Term Memory (LSTM): LSTM is a type of recurrent neural network architecture [5]

8. Linear least squares (LLSQ): This is a benchmark taken from [2]

The implementations can be found on GitHub [1].

Application of Futhark's AD

We utilized Futhark's built-in AD capabilities to obtain the high-level differentiated versions of the selected benchmarks. This process involved using the AD constructs provided directly by Futhark's language and compiler.

For each of the benchmarks, the primary function targeted for differentiation was transformed using Futhark's AD syntax. This involved applying either jvp (Jacobian-vector product for forward mode AD) or vjp (vector-Jacobian product for reverse mode AD). We used reverse mode (vjp): Dot Product, Matrix Multiplication, GMM, BA, LLSQ, and LSTM, and forward mode (jvp) for Hand Tracking (HT). Kmeans uses both jvp and vjp to compute the derivative. Our primary role in this phase was therefore not to implement the AD transformations in Futhark ourselves, but rather to correctly apply the existing AD constructs and ensure the resulting programs could be compiled and executed within our experimental environment.

Application of Enzyme to Futhark Programs

To perform low-level AD we applied Enzyme to the LLVM Intermediate Representation (IR) derived from the Futhark benchmark programs. This required a multi-step pipeline that presented the primary programming and interfacing challenges of this project. The process began by compiling the original Futhark programs into C code using the Futhark compiler.

A crucial phase then involved manually modifying this Futhark-generated C code to be able to interface with Enzyme. We identified the functions to be differentiated and integrated calls to Enzyme's C API, such as __enzyme_autodiff. Invoking this API requires specifying how Enzyme should treat each of the primal function's arguments during differentiation. This is achieved by passing special attributes alongside each argument, for example:

1. Active and Duplicated (e.g., using enzyme_dup): Signifies that this value is active and required for computing the gradient. The argument's original (primal) value must be preserved for the gradient computation.

2. Active, Duplicated and Primal Not Needed for Gradient (e.g., using enzyme_dupnoneed): Marks an active argument as not needed. This is useful when the user wants value of the gradient and not the primal value.

[1]https://github.com/ABBAASCH/ADBenchmark-Futhark-vs-Enzyme

3. Constant (e.g., using enzyme_const): Indicates the argument is not active in the differentiation and its derivative is not computed. This is used for inputs that do not require gradients, such as fixed parameters.

Once the C code has been modified to use the Enzyme API and attributes, it was then compiled into LLVM IR using Clang. The Enzyme AD transformation itself occurred in the LLVM IR file by loading Enzyme as a compiler plugin within LLVM's opt tool. This step generated a new LLVM IR file containing the differentiated functions, which was finally compiled into an executable. The practical application of this workflow involved overcoming several technical hurdles, particularly in ensuring a correct interface between the Futhark-generated code and Enzyme's API. These challenges, along with a concrete example, are detailed in the next section.

Challenges in Applying Enzyme to Futhark-Generated Code

One challenge we encountered in using Enzyme was custom allocation functions. We used some allocation functions which were generated by Futhark, which the Enzyme plugin could not handle. The issue was in the use of a free list. We ended up removing this free list from the allocation function, which resulted in a simple malloc function being invoked, whenever we called the custom allocation function. This was not ideal, however, it is important to note that this compromise affected both Futhark and Enzyme. It is therefore a fair compromise for our benchmarks since they are performed under the same terms.

Illustrative Walkthrough of a Basic Benchmark

As an example we have the following Futhark function, $f(x) = x^2$ that we want to differentiate. Consider the following function definition:

```
def f (x: f64): f64 = x**2
```

Compiling this program would give us the following C code (Simplified version):

```
int futrts_f(double *out, double x) {
    int err = 0;
    double prim_out;
    double f_res = fpow64(x, 2.0);

    prim_out = f_res;
    *out = prim_out;

    return err;
}
```

This is a simple primal function. However, we still need to add other function to our program to perform the benchmark. We can extend our initial Futhark with the following:

```
def f (x: f64): f64 = x**2

entry primal (x : f64): f64 = #[noinline] f x

entry diff_enzyme (x: f64) : f64 = ???

entry diff_futhark (x: f64) : f64 = vjp f x 1.0
```

Here we see our primal function, which we have explained earlier, with three different entry points. The entry points give us the ability to benchmark these functions separately with given datasets. We have also used the noinline flag in the futhark entry point. This is done so the primal function is not in-lined so we can call the primal function. Notice what our enzyme entry points is defined with '???', this is a pattern we use in multiple benchmarks. The reason we do this is just to generate the function with no meaningful content, since we are going to replace its content later anyways. With all of the functions defined, we can now compile the Futhark program to C code. Here we can see the Enzyme function that was generated with the '???' which has generated a Hole warning:

```c
int diff_enzzyme(double *out, double x) {
    int err = 0;
    double prim_out;

    if (!0) {
        set_error(...)
    }

    prim_out = 0.0;
    *out = prim_out;

    return err;
}
```

We replace the content of this function with the relevant enzyme attributes. This is where we perform the AD with Enzyme:

```c
int diff_enzzyme(double *out, double x) {
    ...
    double seed, d_seed = 1.0;  // seed for output
    double d_x = 0.0;           // dervative adjoint

    __enzyme_autodiff((void*) f,
        enzyme_const, ctx,
        enzyme_dupnoneed, &out, &d_out,
        enzyme_dup,       &x,   &d_x
    );
    ...
}
```

This C function, diff_enzyme, now demonstrates how we interface with Enzyme to perform Automatic Differentiation. The core of this is the call to __enzyme_autodiff. The first argument, (void*)f passes a pointer to our primal C function (which originated from the Futhark f function). Subsequent arguments pair Enzyme-specific attributes like enzyme_const, enzyme_dupnoneed, and enzyme_dup with the primal function's arguments (e.g., ctx, output, x) and their corresponding derivative variables (e.g., d_output, d_x). These attributes, conceptually introduced in the previous section, instruct Enzyme on how to treat each variable during differentiation for instance, enzyme_const marks ctx as non-differentiable, while enzyme_dup indicates that x is an active input whose derivative d_x is required and whose primal value is needed. This instrumented C code is then compiled to LLVM IR, which Enzyme processes to generate the differentiated function, before final compilation to an executable. This walkthrough illustrates the fundamental steps applied across all benchmarks to enable differentiation via Enzyme.

Performance Measurement and Metrics

To empirically compare the two AD approaches we collected runtime using Futhark's build-in benchmark tool. Futhark bench is a standard tool for benchmarking Futhark programs; it performs multiple timed runs, including a warm-up run, and reports statistical summaries with arithmetic mean runtime and 95% confidence interval[7]. We used the mean runtime from these summaries for our analysis. Runtime in Futhark bench is defined as the wall-clock duration for when the program start to finish [6].

All benchmarks were executed on the identical hardware and software environment using the same input datasets for both AD approaches. The functional correctness of the derivatives generated by both methods was verified for each benchmark. This verification involved comparing the program's output on its first dataset against a reference validation output. These reference outputs were either provided as part of the existing ADBench datasets or were generated by us from the output of the original Futhark AD implementation. If the differentiated program's output matched this reference for the first dataset, the implementation was assumed to be correct for the purpose of performance benchmarking. The datasets used for GMM, KMeans, HT, BA and LSTM are taken from Gradbench [4]. These dataset are large and minimize noise during the benchmark tests.

Evaluation and Results

In this section we will present our empirical findings form our comparative study of Futhark's AD and Enzyme's AD. We will begin by presenting the results from the various benchmarks, this being primarily through graphical representations of speedup. Subsequently we dive into a detailed analysis of those results. Here we aim to clarify the the factors that contributed to the observed performance differences and to evaluate our hypothesis regarding advantages of high-level AD.

Presentation of Benchmark Results

The core of the performance metric used for this comparison is the speedup achieved by Enzyme relative to Futhark, calculated as speedup $= \frac{\text{Futhark AD execution time}}{\text{Enzyme AD execution time}}$. If we have a speedup greater than 1.0 then it indicates that Enzyme is faster, while a value less than 1.0 indicates that Futhark is faster.

How we analyzed our results

As an example of how we analyze the benchmark results we consider the result from the benchmark with dot product. This is a smaller benchmark, but serves its purpose in explaining how we analyze and conclude our analysis of the benchmarks. Our first step is to look at the graph of our benchmark results.

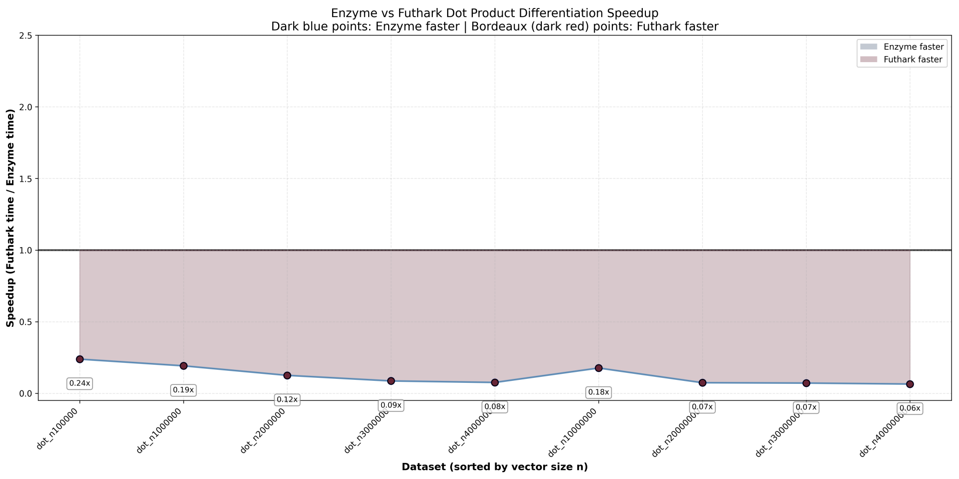

Figure 1: Speedup graph for dot product

As we can see in figure 1, Futhark gets increasingly faster as the input for dot product increases. Specifically, we see a speedup range from 0.24x to 0.06x. This means for the smallest dataset with Futhark is approximately 4 times faster and on the largest dataset approximately 16x faster. As this is our smallest benchmark, it is viable for us to look at the LLVM IR generated from this benchmark. This will give insight into how the code structure is optimized and will explain the trend we see in the graph.

The purpose is to identifying loops and potential instructions that impact runtime the most. We will be analyzing the potential issues using perf which is a Linux tool that analyze performance by counting CPU events. The LLVM IR we analyze after optimizations have been performed with the '-O2' flag as well as before and after AD.

Our method involved tracing the program's execution paths. LLVM IR uses the `br` instruction to manage control flow and jump between labels. A loop is formed by a `br` instruction that can return to the same label. When we identified a loop structure we examined the controlling branch condition to determine the criteria for exiting the loop. Concurrently, we analyzed the instructions within the loop body to identify sources of potential performance overhead.

Our analysis of the LLVM IR generated by Enzyme for a dot product function reveals that it allocates memory for each input argument. For a dot product between two vectors of length n, this means Enzyme inserts two separate calls to malloc, each allocating a buffer of size n.

This memory allocation is a fundamental caching mechanism in Enzyme. It is required to preserve the original input values for the reverse pass of automatic differentiation. This step becomes critical when these original values cannot be easily recomputed but are necessary for the gradient calculation [11]. The example of the generated LLVM IR shows that there is two different `malloc` calls followed by `llvm.memcpy` which copies the values of the input parameters, A and B into the cache. After the caching phase Enzyme then accesses the cached values in reverse order. Consequentially creating two loops indexing from 0 to n and afterward a loop backward from n down to 0.

Both the Enzyme-generated and Futhark-compiled code compute the dot product gradient using a single loop over the n elements of the input vectors. However, the core difference between them lies in the compiler's high-level interpretation of the gradient computation itself.

Futhark performs a high-level optimization, recognizing that the gradient of a dot product with respect to one input vector is simply the other input vector. As a result, Futhark avoids allocating intermediate "adjoint" arrays for the gradients. The resulting instruction sequence is significantly simpler and more efficient, as it only needs to copy the final gradient values. In contrast, Enzyme adheres strictly to the structure of the primal function. As discussed previously, it allocates two memory buffers of size n to cache the primal inputs. It then allocates two additional buffers for the output adjoints, leading to a total of four large memory allocations. Enzyme uses an accumulation loop, where each gradient element is updated with a load, add, and store sequence. Futhark's optimization, by contrast, would only require a load and a store.

This difference in memory strategy and instruction complexity is directly reflected in performance metrics. On a large test vector, perf reports the following:

1. Futhark: 78,205 page faults and 9.63% cache misses.
2. Enzyme: 97,736 page faults and 10.09% cache misses.

While the difference in cache misses is minor and could be attributed to runtime variance, the disparity in page faults is substantial. This indicates significantly higher memory pressure from Enzyme's approach. This would explain the difference in runtime, that we see in figure 1.

Matrix multiplication

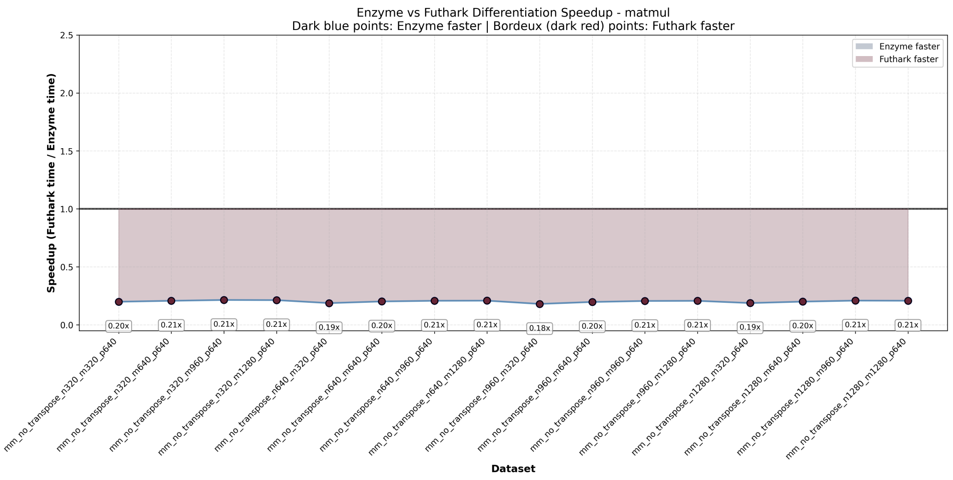

Figure 2: Speedup graph for matrix multiplication

In the figure 2 we see the results from the benchmark for matrix multiplication. We notice that Futhark is consistently faster with only a small range of changes in the speedup. Here we see that Futhark has a speedup which ranges from 0.18x to 0.21x.

Futhark's superior performance in matrix multiplication AD is due to high-level compiler optimizations, as detailed in [13]. Even though the paper focuses on rewrite rules and optimizations on GPU code. It is still relevant to notice the optimizations in section V-A [13]. This optimization focuses on eliminating the bad temporal locality we would see in the triple nested for loop.

Instead of differentiating the primal's triple-nested loop directly, Futhark restructures the entire reverse pass. It transforms the gradient computation into two distinct, cache-friendly matrix multiplication accumulators for each input's adjoint. This strategy, confirmed by the code snippet for our benchmark below, significantly improves spatial and temporal locality. The fundamental difference in these looping strategies can be illustrated with the following pseudocode in figure 3:

In contrast, Enzyme operates on the lower-level LLVM IR, applying its AD rules to the existing loop structure without performing a semantic transformation. Futhark's ability to restructure the computation at a higher level of abstraction is therefore the key driver of its performance advantage.

The LLVM IR generated by Enzyme for matrix multiplication differentiation is similar to the dot product process but instead uses two triple-nested loops. Like in dot product it allocates memory and copies. This time however, we allocate two matrices of size $n \times m \times p$ for A and B into new buffers.

The reason Enzyme chooses to cache larger memory blocks for A and B is

```
primal(A, B, n, m, p)
  for i=0; i<n; i++
    for j=0; j<p; j++
      for k=0; k<m; k++
        A[i*m*k]
        B[k*p*j]
```

```
diff_Futhark(A, B, n, m, p)
  for i=0; i<n; i++
    for j=0; j<p; j++
      for k=0; k<m; k++
        dA[j*m+k] = dA[j*m+k] + A[i*m+k]

    for j=0; j<p; j++
      for k=0; k<m; k++
        dB[k] = dB[k] + B[k*p+j]
```

```
diff_Enzyme(A, B, n, m, p)
  for i=0; i<n; i++
    for j=0; j<m; j++
      for k=0; k<0
        // Enzyme caching arguments

  for i=n; i=0; i--
    for j=n; j=0; j--
      for k=n; k=0; k--
        // Enzyme calculate dA and dB
```

Figure 3: Matrix multiplication loops with Primal (top), Futhark (bottom left), Enzyme (bottom right).

because it needs space for all the instructions from the forward pass and replay them in the reverse pass using the tape data structure [10].

Enzyme's reverse pass differentiates the primal code directly, failing to optimize for temporal locality in the way Futhark's high-level restructuring does. This architectural difference leads to a dramatic gap in memory performance. Measurements from perf on a sample benchmark show that Enzyme suffers from 13% cache misses and 129,486 page faults. In contrast, Futhark maintains only 7% cache misses and 1,585 page faults. While the cache miss rate is notable, the nearly 100-fold increase in page faults for Enzyme points to a severe performance penalty from its less efficient memory access patterns and initialization overhead.

GMM

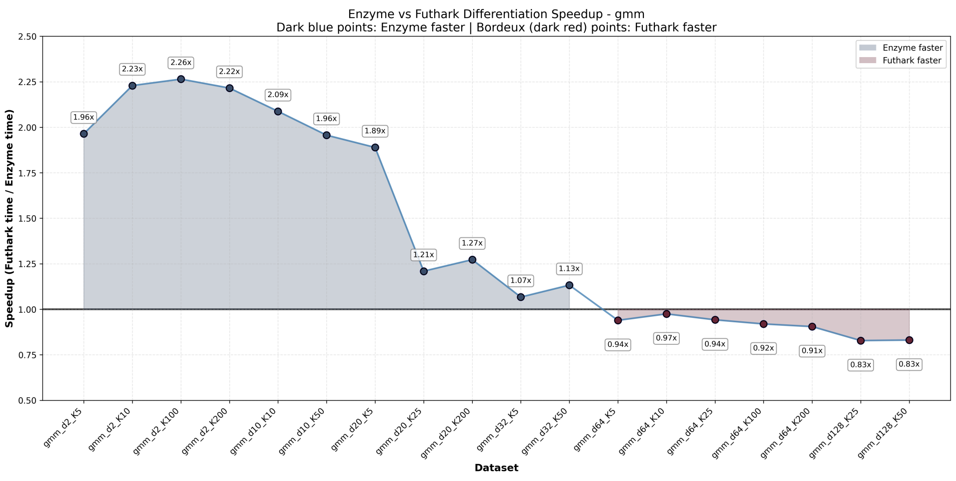

Figure 4: Speedup graph for GMM

The results for the Gaussian Mixture Model (GMM) benchmark, presented in Figure 4, reveal a performance that is strongly dependent on the data dimensionality, d.

For configurations with low dimensionality (d=2, 10, and 20), Enzyme consistently and significantly outperforms Futhark's AD. The speedup for Enzyme in this regime is substantial, ranging from 1.21x to a peak of 2.26x on the d=2 and K=100 dataset. This indicates that when the computation is less intensive, the overhead of Futhark's high-level transformation is more pronounced, and Enzyme's ability to leverage low-level scalar and basic loop optimizations provides a clear performance advantage.

However, this performance dynamic shifts as dimensionality increases. The shift occurs around d=32, where Enzyme's advantage diminishes to 1.07x. For all higher-dimensional datasets (d=64 and d=128), Futhark's AD becomes the superior approach. Futhark's advantage grows with dimensionality, delivering a 1.20x speedup (a ratio of 0.83x) over Enzyme on the largest d=128 K=50 configuration.

This trend provides strong empirical support for our central hypothesis. An analysis of the gmmObjective source code confirms that its runtime is dominated by a matrix-vector multiplication involving a $d \times d$ matrix derived from the inverse covariance factor. The cost of this operation scales quadratically $O(d^2)$ with the dimension d. As this memory-intensive operation becomes the bottleneck at higher dimensions, the benefits of Futhark's high-level AD transformation become decisive. As described in our previous section about the matrix multiplication analysis, , we can apply the same logic. Futhark's AD can perform semantic, structural transformations, such as accumulator optimizations, that are specifically designed to restructure gradient updates and improve cache locality. Enzyme, operating at the lower LLVM IR level, does not perform this semantics restructuring.

Therefore, the GMM benchmark clearly demonstrates a critical trade-off: while low-level AD is highly effective for computing simple computations, a high-level AD implementation offers a significant performance advantage for memory-bound and is dictated by the structure of large matrix operations.

KMEANS

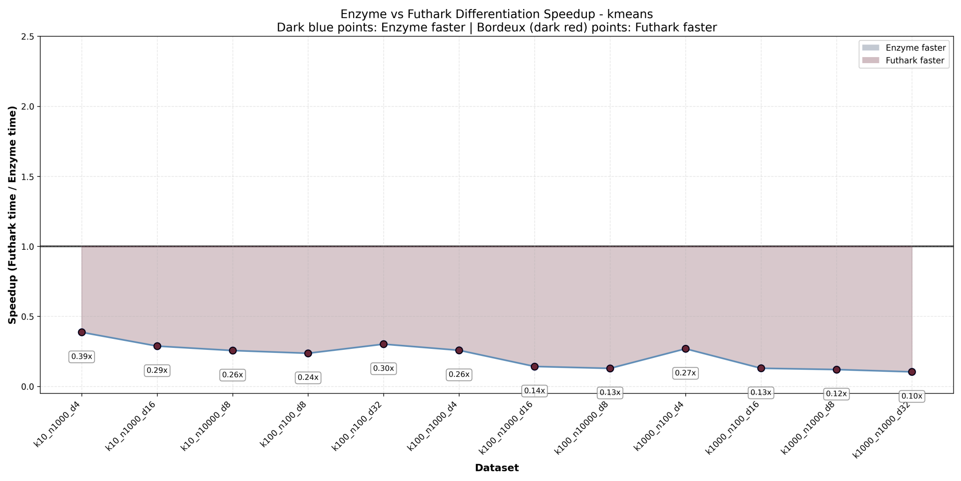

Figure 5: Speedup graph for kmeans

The K-means benchmark, with results shown in Figure 5, show strong empirical evidence in support of our hypothesis. Across every tested configuration, Futhark's AD outperforms Enzyme, achieving a speedup of between 2.5x and 10x (corresponding to speedup ratios of 0.39x to 0.10x).

The results demonstrate a clear trend where Futhark's performance advantage generally increases with the scale of the dataset, which is a function of the number of points n, clusters k, and dimensions d. While the trend is consistent, minor spikes in the speedup ratio, such as at k=1000, n=100 and d=4, correlate with lower dimensionality. This suggests that while Futhark's superiority is universal for this benchmark, its relative advantage is less pronounced for computationally simpler, low-dimension problems a finding consistent with the GMM analysis.

HT

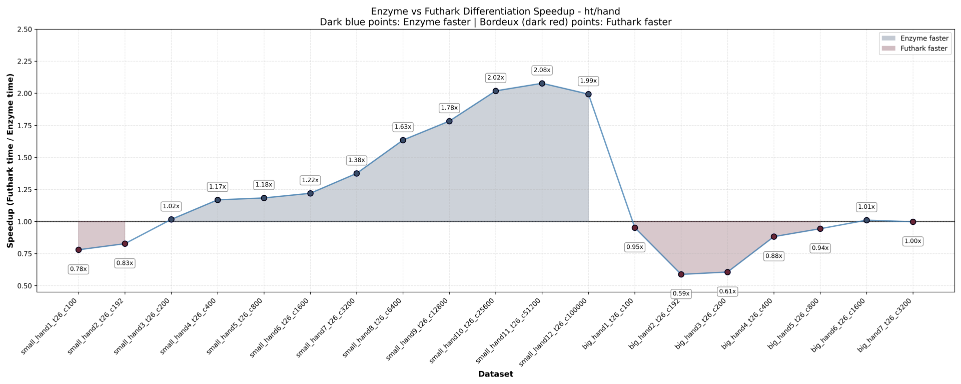

Figure 6: Speedup graph for ht

The ht/hand benchmark, which uniquely utilizes forward-mode AD, presents a complex performance narrative with three distinct phases, as shown in Figure 6. These results highlight a nuanced trade-off between low-level and high-level AD approaches that is dependent on problem scale and structure.

Initially, on the two smallest datasets hand1: t=26, c100 and hand2: t=26, c=192, Futhark's AD is faster, suggesting that for minimal computations, the overhead of Enzyme may outweigh its optimization benefits.

However, this trend reverses for the remaining small datasets. In this second phase, Enzyme becomes increasingly dominant as the problem size grows, peaking at a significant 2.11x speedup. This behavior is characteristic of forward-mode AD, which generates derivative code that mirrors the primal program's control flow [10].

The performance dynamic shifts again for the big_hand datasets. The most significant result in this benchmark is the dramatic performance reversal at big hand3: t=26 c=200, where Futhark becomes 1.69x faster than Enzyme (a 0.59x speedup). Following this anomaly, performance remains more contested, with Futhark often holding a slight advantage. This "performance cliff" for Enzyme strongly suggests that while its low-level optimizations are powerful, they are applied to a fixed code structure. At a certain scale or data layout, this structure can lead to severe bottlenecks, likely related to cache inefficiency or memory access patterns. Futhark's high-level AD, by being able to perform semantic transformations on the array primitives themselves, can generate derivative code with a different structure that avoids this specific bottleneck.

The similar structure of the code when using forward mode AD could explain the similar runtimes for both AD systems, seen in the biggest dataset big hand6: t=26, c=1600 and big hand7: t=26, c=3200. Here we see that Enzyme is not bound by the same memory problems or by the high-level data structures, which could be ineffective compared to the high-level transformations utilized by Futhark. Considering this, it would be reasonable to see this equal performance when using forward mode AD.

In conclusion, the ht/hand benchmark demonstrates that while a low-level, forward-mode approach can be highly effective for a broad range of problem sizes, it is also susceptible to performance cliffs. However, with the biggest datasets in this benchmark, we see a similar performance, which is reasonable when we think about the characteristics of forward mode AD.

BA

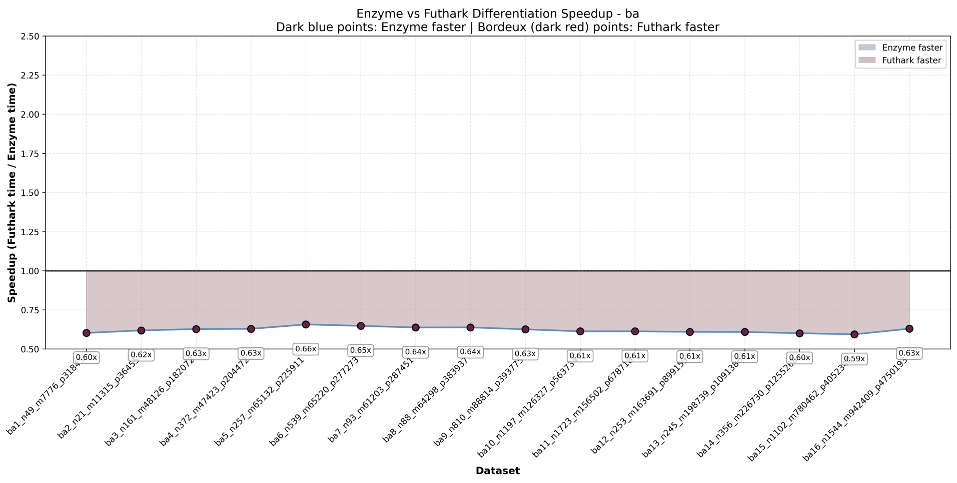

Figure 7: Speedup graph for BA

The Bundle Adjustment (BA) benchmark, as shown in Figure 7, reveals a different performance dynamic. Futhark's AD is consistently faster than Enzyme across all 16 datasets, with the speedup ratio remaining stable in a narrow band between 0.59x and 0.66x. This corresponds to a reliable performance advantage for Futhark of approximately 1.5x to 1.7x.

An analysis of the source code shows that the BA workload is characterized by applying reverse-mode AD to a large number of small, independent computations one for each observation point. These inner computations are dominated by scalar arithmetic and operations on small, fixed-size vectors, not the large, memory-intensive array operations seen in previous benchmarks.

This ”many-small-tasks” structure explains the observed performance. The function that needs to be differentiated is small and does not contain any loop. Enzyme require more instructions to compute the same output. This emphasizes our hypothesis that Enzyme has more overhead.

Therefore, the BA benchmark refines our hypothesis. Based on these results we see that Futhark not only benefits from large matrix operations but also on consistently small workloads.

LSTM

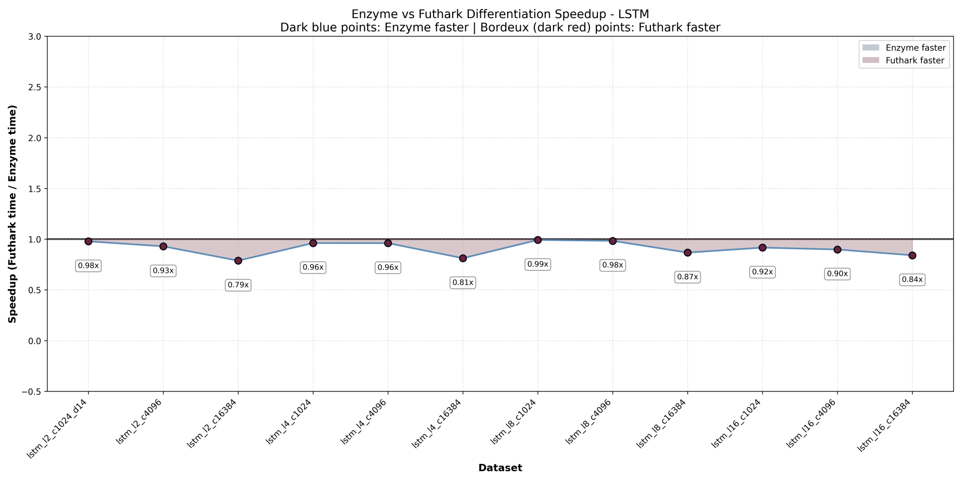

Figure 8: Visual presentation of speedup with LSTM

The Long Short-Term Memory (LSTM) benchmark, with results presented in Figure 8, further reinforces the central hypothesis of this thesis. Futhark's native AD demonstrates a consistent performance advantage over Enzyme across all tested configurations, achieving a speedup of up to 0.79x.

The most significant trend revealed by the data is the relationship between performance and the number of hidden units, or cells (c), in the LSTM layers. Within each configuration of a fixed number of layers, Futhark's relative speedup increases as the cell count grows. This indicates that the high-level AD approach becomes more advantageous as the underlying array operations become larger and more memory-intensive.

This trend is directly explained by the computational structure of the LSTM algorithm. The core of the lstmModel function consists of several element-wise vector operations involving the weight and hidden state vectors. While the cost of these operations scales linearly $O(c)$ with the number of cells, they become the dominant factor in overall runtime as c grows, making performance highly sensitive to memory access patterns. As argued in previous sections, Futhark's ability to perform high-level, structural optimizations on these dense array computations results in more efficient C code. Enzyme, operating at the lower LLVM IR level, is does not utilize semantics-aware and does not do restructuring.

In contrast to the Bundle Adjustment benchmark, which was characterized by many small, independent tasks, the LSTM benchmark's performance is dictated by large, memory-bound array operations, similar to the GMM benchmark. The consistent results across both GMM and LSTM strongly suggest that for algorithms dominated by such patterns, a high-level AD system that can semantically restructure the derivative computation offers a performance advantage.

LLSQ

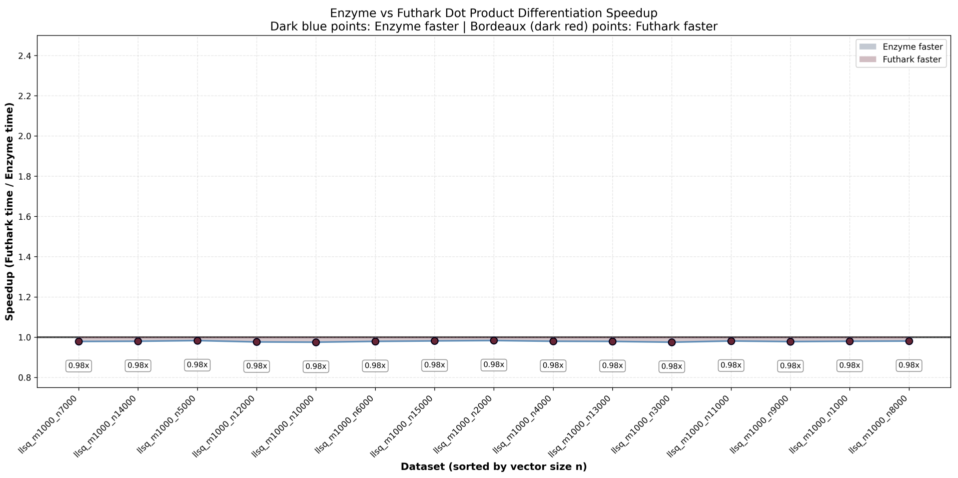

Figure 9: Speedup graph for llsq

The Linear Least Squares (LLSQ) benchmark presents a notable contrast to the previous results. As shown in Figure 9, the performance of Futhark's AD and Enzyme is nearly identical across all datasets. Futhark maintains a marginal but consistent advantage, with the speedup ratio remaining stable at approximately 0.98x, indicating a 2% performance benefit.

The key finding from this benchmark is the absence of a significant performance gap or a trend related to problem size. This result is directly explained by the computational structure of the LLSQ algorithm. An analysis of the source code reveals that the workload is a nested loop dominated by scalar arithmetic. It lacks large-scale memory requirement like GMM and LSTM, and the "many-small-tasks" structure of the BA benchmark.

There are no obvious opportunities for Futhark's high-level AD to apply the semantic transformations as in other benchmarks. Both the high-level and low-level AD systems are likely differentiating a simple loop structure in a similar way. Since both the Futhark C backend and the LLVM optimizer are optimizing such loops, they generate derivative code with nearly equivalent performance.

Therefore, the LLSQ benchmark serves as a good control case. It does not contradict our hypothesis, but rather refines it by identifying a class of workload where the specific advantages of high-level AD is not present. This demonstrates that the performance benefits of a high-level approach are tied to the presence of specific, transformable computational patterns, and in their absence, the performance of high-level and low-level AD systems can be expected to converge.

Overall Trends and Discussion

The individual benchmark results when analyzed collectively reveal that the performance trade-off between high-level and low-level AD is not monolithic but is instead dictated by the underlying computational pattern of the workload. Our findings identify three distinct categories of behavior that, taken together, provide a nuanced understanding of when each AD paradigm excels.

This is most evident in memory-bound benchmarks like in GMM and LSTM. In these cases, Futhark's AD becomes increasingly dominant as data dimensionality and hidden state sizes grow. This advantage comes from its high-level understanding of array operations, which enables structural optimizations that improve memory locality. In contrast, the Bundle Adjustment (BA) benchmark, characterized by thousands of small, independent scalar computations highlights a different advantage. Futhark's AD transformation use less computation compared to the low-level Enzyme AD tool, that possibly introduce additional overhead to achieve the same result.

Furthermore, the forward-mode ht/hand benchmark demonstrates that while a low-level approach can be highly effective, its reliance on a fixed code structure makes it vulnerable to severe "performance cliffs" at specific problem scales a bottleneck Futhark's structural transformations can avoid.

Our investigation into GPU execution, while preliminary, reinforces the practical advantages of a high-level, integrated system. The significant technical hurdles encountered when attempting to apply Enzyme to hand-written CUDA performance - compromising workarounds for memory management - highlight the complexity of manual GPU differentiation. Futhark's ability to automatically generate efficient derivative code for its primary hardware target underscores the value of a co-designed system where the AD transformations are aware of the underlying parallelism model.

Collectively, these findings provide nuanced support for our hypothesis. The primary advantage of high-level AD lies in its ability to apply semantics-aware transformations that are tailored to the specific structure, whether to optimize memory access, reduce invocation overhead, or provide performance resilience against architectural bottlenecks.

Conclusion and Future Work

This thesis conducted an empirical comparison of performing AD on high-level and low-level using Futhark and Enzyme AD. We have applied AD to various programs with different computational patterns and systematically investigated the performance trade-offs between differentiating at the source-code level versus the LLVM IR level. Our findings provide an answer to our central hypothesis, revealing that it is optimal for runtime to perform AD at a higher level to benefit from optimizations and more efficient workload structure.

Summary of Findings

Our empirical investigation yielded three distinct categories of performance behavior. First, for memory-bound benchmarks dominated by large, dense array operations, such as Gaussian Mixture Models (GMM) and Long Short-Term Memory (LSTM), high-level AD becomes increasingly advantageous as data dimensionality grows. Second, for workloads composed of many small, independent tasks, such as Bundle Adjustment (BA), high-level AD provides a consistent and moderate performance advantage by avoiding the cumulative overhead of the AD tool. Finally, the forward-mode Hand Tracking (HT) benchmark demonstrated that while a low-level approach can be highly effective it is also vulnerable to severe ”performance cliffs” at specific problem scales, a bottleneck that Futhark's structural transformations can avoid.

Conclusion regarding Hypothesis

Our findings provide strong, nuanced support for our central hypothesis: that performing AD at a higher level of abstraction offers distinct performance advantages. This advantage is primarily attributed to the AD system's ability to leverage semantic and structural information present in high level representations compared to low-level code.

The evidence confirms that there is not a single advantage. The benefits of high-level AD depends on the computational pattern. For memory-bound workloads, it excels by performing structural optimizations that improve data locality. Its advantage lies in transformation that minimizes overhead as well as providing performance resilience by generating structurally different derivative code that can avoid the bottlenecks that affect low-level approaches. Therefore, we conclude that while low-level AD is a powerful and general tool, a high-level, language AD system offers a critical advantage leveraging semantic information to generate fundamentally more efficient derivative code.

Limitations of the Study

We acknowledge the following limitations to our study. First, all experiments were conducted on a single CPU architecture. Performance dynamics on GPUs, which have vastly different memory hierarchies and parallelism models, may differ significantly and were not explored. Second, while our chosen benchmarks cover a wide range of diverse computational patterns, they do not represent the entire space of program structures; workloads with more complex control flow or irregular memory access might yield different results. Finally, our methodology involved applying Enzyme to Futhark-generated C code. A native implementation in a language like C++ might allow for different low-level optimizations, potentially altering the performance comparison.

Future Work

Based on our findings and limitations, several promising areas for future research emerge. The most critical next step is to extend this empirical comparison to a full suite of benchmarks on GPU hardware. Our preliminary results indicate significant practical challenges in using low-level tools for GPU differentiation, making a systematic comparison a valuable contribution. A second investigation would be a deeper analysis of the ”performance cliff” observed in the ht/hand benchmark to precisely identify its architectural cause. Finally, our results suggest that neither AD paradigm is universally superior, motivating long-term research into hybrid AD systems that could selectively apply high-level structural transformations to memory-bound array kernels while using low-level AD for scalar-heavy or control-flow-intensive sections of a program.

Bibliography

[1] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: `1502.05767 [cs.SC]`. URL: `https://arxiv.org/abs/1502.05767`.

[2] cmpad. *llsq_obj*. URL: `https://cmpad.readthedocs.io/llsq_obj.html`. (accessed: 01.04.2025).

[3] futhark-ad. *futhark-ad*. URL: `https://github.com/diku-dk/futhark-ad`. (accessed: 06.05.2025).

[4] GradBench. *GradBench*. URL: `https://github.com/gradbench/gradbench`. (accessed: 06.05.2025).

[5] gradbench. *Long Short-Term Memory (LSTM)*. URL: `https://github.com/gradbench/gradbench/tree/main/evals/lstm`. (accessed: 01.09.2025).

[6] Troels Henriksen. *Futhark book*. URL: `https://futhark-book.readthedocs.io/en/latest/practical-matters.html`. (accessed: 06.09.2025).

[7] Troels Henriksen. *Futhark-bench - Futhark 0.26.0 Documentation*. URL: `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`. (accessed: 06.09.2025).

[8] Troels Henriksen. *Why Futhark*. URL: `https://futhark-lang.org/`. (accessed: 01.04.2025).

[9] Chris Lattner. *The Architecture of Open Source Applications (Volume 1) LLVM*. URL: `https://aosabook.org/en/v1/llvm.html`. (accessed: 04.05.2025).

[10] William S. Moses and Valentin Churavy. “Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS ’20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[11] William S. Moses et al. “Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’22. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445.

[12] Simon Rogers. and Mark Girolami. *A First Course in Machine Learning*. Chapman Hall/CRC, 2011. ISBN: 9781498738484.

[13] Robert Schenck et al. *AD for an Array Language with Nested Parallelism*. 2022. arXiv: `2202.10297 [cs.PL]`. URL: `https://arxiv.org/abs/2202.10297`.

[14] Filip Šrajer, Zuzana Kukelova, and Andrew Fitzgibbon. *A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning*. 2018. arXiv: `1807.10129 [cs.MS]`. URL: `https://arxiv.org/abs/1807.10129`.