GPU Acceleration of L-BFGS Optimizer

Nedim Džajić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ndzajic1@etf.unsa.ba

Emir Kapić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ekapic1@etf.unsa.ba

Emir Salkić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
esalkic1@etf.unsa.ba

Muaz Sikirić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
msikiric1@etf.unsa.ba

***Abstract*—This study investigates the acceleration of the L-BFGS optimization algorithm using GPUs. We implement the L-BFGS algorithm with CUDA technology to leverage the parallel processing capabilities of modern GPUs for large-scale optimization problems. Our implementation focuses on parallelizing computationally intensive components, including gradient evaluations, vector operations, and the two-loop recursion for Hessian approximation. We compare various line search strategies using Wolfe and Armijo conditions and analyzing their performance and parallelization potential. Experiments conducted on the challenging Rosenbrock function demonstrate that GPU acceleration becomes significant primarily for high-dimensional problems, with observable speedups emerging at dimensions of 10,000 and above. While Armijo-based methods initially outperform Wolfe-based approaches in sequential implementations, our parallelized Wolfe methods achieve substantially higher speedups (up to 3.55×), narrowing this performance gap. The results indicate that the inherently sequential nature of L-BFGS limits parallelization benefits for small-scale problems, but GPU implementation provides significant advantages for large-scale optimization tasks, particularly when combined with highly parallelizable line search methods.**

***Index Terms*—L-BFGS Algorithm, GPU Acceleration, Line Search Methods, Rosenbrock Function, High-Dimensional Optimization**

I. Introduction

The Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm is a widely used optimization technique for solving large-scale problems, particularly in machine learning and scientific computing. As a variant of the BFGS algorithm, L-BFGS belongs to the class of quasi-Newton methods, which iteratively minimize a function by approximating its Hessian matrix. The Hessian matrix, representing second derivatives, captures curvature information that is crucial for efficient optimization. Unlike the full-memory BFGS method, which explicitly stores the entire Hessian matrix, L-BFGS adopts a limited-memory approach by maintaining only a small set of vectors that approximate the inverse Hessian [1]. This characteristic significantly reduces memory and computational requirements, making L-BFGS well-suited for high-dimensional problems, such as neural network training and large-scale numerical simulations.

L-BFGS achieves a balance between rapid convergence—typically faster than simple gradient descent—and efficient memory usage. At each iteration, it updates parameters using the current gradient and the stored approximation vectors, continuing until it converges or satisfies a stopping criterion, such as a sufficiently small gradient magnitude [1]. Due to its robust convergence properties and ability to handle both convex and certain non-convex functions, L-BFGS is widely applied in various domains, including machine learning optimization and computational physics.

Despite its advantages, parallelizing the L-BFGS algorithm presents challenges due to its inherently sequential nature and the necessity of preserving curvature information across iterations. However, various strategies have been explored to optimize L-BFGS for parallel architectures, particularly Graphics Processing Units (GPUs) [2]. These efforts primarily focus on parallelizing computationally intensive components, such as gradient evaluations and matrix-vector operations, by distributing them across multiple cores or GPU threads. Although achieving efficient parallelism is non-trivial, GPU-accelerated implementations of L-BFGS have demonstrated significant speedups in large-scale optimization tasks where computational efficiency is paramount [3].

The rise of GPU computing has been transformative in accelerating optimization algorithms, making large-scale problems more tractable. GPUs are well-suited for highly parallelizable tasks due to their massive core count and high memory bandwidth. While the inherently sequential nature of L-BFGS poses challenges, advancements in hardware and algorithmic adaptations have made GPU acceleration increasingly viable. Researchers have explored hybrid approaches [4] that offload specific operations to the GPU while maintaining the integrity of the algorithm's iterative process.

This study examines GPU-based implementations of L-BFGS, analyzing their performance, scalability, and effectiveness in leveraging the parallel processing capabilities of mod-

ern GPUs. By exploring existing approaches and evaluating their impact on optimization efficiency, this work aims to contribute insights into the practical benefits and challenges of GPU acceleration for L-BFGS.

II. Algorithm Description

L-BFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) is a quasi-Newton optimization method designed for large-scale optimization problems, where storing and computing the full Hessian matrix is impractical. Instead of storing a dense $n \times n$ Hessian, L-BFGS reconstructs an approximation using a limited number of previous gradient and parameter updates [1]. Implementation of the algorithm in CUDA further accelerates this process by leveraging GPU parallelism for key operations, including gradient evaluation, vector operations, and the two-loop recursion for Hessian approximation [2].

The optimization process begins with computing the gradient of the objective function:

$$g_k = \nabla f(x_k) \tag{1}$$

This step is fully parallelized, with each GPU thread computing partial derivatives for different data points or model parameters. The results are then reduced efficiently using warp shuffle and shared memory optimizations to form the full gradient.

Once the gradient is obtained, the search direction is computed using the limited-memory inverse Hessian approximation. Instead of explicitly storing and inverting the Hessian matrix, the two-loop recursion is used to reconstruct the inverse Hessian-vector product:

$$p_k = -H_k^{-1} g_k \tag{2}$$

where $-H_k^{-1}$ is approximated using past updates of parameter differences $s_k = x_k - x_{k-1}$ and gradient differences $y_k = g_k - g_{k-1}$. The recursion follows the iterative update:

$$H_{k+1}^{-1} = H_k^{-1} - \frac{H_k^{-1} y_k y_k^T H_k^{-1}}{y_k^T s_k} + \frac{s_k s_k^T}{s_k^T y_k} \tag{3}$$

In the CUDA implementation, the two-loop recursion involves multiple dot products and vector operations, which are optimized using warp-synchronous programming and SIMD (Single Instruction, Multiple Data) parallelism. This ensures efficient computation across thousands of parallel GPU threads [2].

A line search is performed to determine an appropriate step size α_k, ensuring sufficient function decrease:

$$x_{k+1} = x_k + \alpha_k p_k \tag{4}$$

To improve efficiency, multiple step sizes can be evaluated in parallel using independent CUDA kernels, though selecting the final step size requires a reduction step.

The CUDA implementation can support several line search methods. The Wolfe condition-based line search method ensures that the step size satisfies the Armijo condition (sufficient function decrease) and the curvature condition (ensuring the gradient changes sufficiently). The step size α_k is chosen such that [5]:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k g_k^T p_k \tag{5}$$

$$g(x_k + \alpha_k p_k)^T p_k \geq c_2 g_k^T p_k \tag{6}$$

where $0 < c_1 < c_2 < 1$ are predefined parameters. The function and gradient evaluations required for these conditions can be parallelized, with different CUDA threads evaluating different step sizes simultaneously.

Backtracking line search is a simpler approach that starts with an initial step size α_0 and reduces it iteratively until it satisfies the Armijo condition [1]. This method follows:

$$\alpha \leftarrow \tau \alpha \tag{7}$$

where $0 < \tau < 1$ is a reduction factor. In the CUDA implementation, multiple candidate step sizes can be tested in parallel, and the first one that satisfies the condition is selected.

When function values at two different step sizes α_1 and α_2 are known, an interpolated step size can be computed to estimate a better step size. A common approach is quadratic interpolation, where the new step size is computed as:

$$\alpha_{\text{new}} = \frac{\alpha_1^2 g_2 - \alpha_2^2 g_1}{2(\alpha_1 g_2 - \alpha_2 g_1)} \tag{8}$$

where $g_1 = \nabla f(x_k + \alpha_1 p_k)^T p_k$ and $g_2 = \nabla f(x_k + \alpha_2 p_k)^T p_k$. This interpolation step helps reduce the number of function evaluations required for line search, improving efficiency. CUDA optimizations can be used to parallelize function evaluations at different step sizes and perform efficient reductions to compute interpolated values.

Strong Wolfe line search is a more robust variant of Wolfe conditions that provides better numerical stability. It ensures that the step size satisfies:

$$\left| g(x_k + \alpha_k p_k)^T p_k \right| \leq c_2 \left| g_k^T p_k \right| \tag{9}$$

This method requires bisection and interpolation steps, which can be partially parallelized using multi-kernel execution in CUDA.

Memory management is critical for performance, as L-BFGS only retains the most recent m updates to maintain its limited-memory advantage. These past updates are stored in global GPU memory and accessed efficiently to minimize latency. The curvature pair,

$$\rho_k = \frac{1}{y_k^T s_k} \tag{10}$$

is precomputed in parallel for each stored update, further accelerating the recursion step.

CUDA optimizations play a crucial role in speeding up L-BFGS. Coalesced memory access patterns ensure that memory reads and writes are aligned, reducing access latency. Shared memory is used to accelerate dot product operations, and warp-level primitives allow efficient summation of partial results without expensive global memory operations [3]. By distributing gradient evaluations, vector updates, and memory transfers

across thousands of GPU cores, the proposed implementation of the L-BFGS algorithm achieves substantial performance gains over traditional CPU-based L-BFGS.

III. GPU Implementation

The parallel implementation of the L-BFGS algorithm is designed to leverage CUDA technology, enabling the acceleration of key computational tasks by offloading them to the GPU. This approach is particularly advantageous for large-scale optimization problems, where the GPU's massive parallelism can significantly reduce computational time. The implementation focuses on parallelizing the most computationally intensive components of the algorithm, such as vector operations [6], matrix-vector products, and inner products, using CUDA kernels and the CUBLAS [8] library. The use of CUDA kernels and the CUBLAS library enables efficient computation of vector operations, while asynchronous data transfer using CUDA streams minimizes idle time. The implementation incorporates practical optimizations and considerations that are not explicitly described in the pseudocode in [3], making it well-suited for real-world applications. These differences highlight the importance of adapting theoretical algorithms to practical implementations, particularly in the context of high-performance computing.

A. GPU Memory Management

The implementation begins by allocating GPU memory for all necessary vectors and matrices, including the solution vector x, gradient vector g, search direction d, and the histories of vectors s and y. This ensures that all data is readily available on the GPU for parallel processing. Memory allocation is performed separately for each vector and matrix, with additional memory allocated for intermediate results such as d_x_new and d_g_new.

B. CUDA Kernels

Several CUDA kernels are employed to perform key operations in parallel. The updateVectors kernel computes the differences between successive solutions and gradients, storing the results in vectors s and y. The copyGradient kernel copies the gradient vector g to another vector q, which is used in the two-loop recursion. The negateVector kernel negates the elements of a vector r to compute the search direction d. The updateSolution kernel updates the solution vector x using the step size α and the search direction d. Finally, the scaleByRho kernel scales a vector by a scalar value ρ, which is used in the two-loop recursion to adjust intermediate results. These kernels enable efficient parallel computation of the L-BFGS algorithm on the GPU.

C. CUBLAS Library

The CUBLAS [8] library plays a critical role in the implementation, particularly for performing dot products (`ddot`) and vector updates (`daxpy`). These operations are essential for the two-loop recursion in L-BFGS, where inner products and vector additions are repeatedly executed. The library is also utilized to compute vector norms, which are necessary for checking convergence.

D. Asynchronous Operations

To optimize performance, the implementation employs CUDA streams to overlap computation and data transfer. Two streams are created: one for computation (`compute_stream`) and one for data transfer (`transfer_stream`). This allows the GPU to perform computations while simultaneously transferring data between the host and device, reducing idle time and improving overall efficiency.

E. Line Search

The line search is performed on the CPU, with the necessary vectors transferred back to the host asynchronously. The step size α_k is computed using one of several line search methods, such as backtracking or Wolfe conditions. Once the step size is determined, the updated solution is transferred back to the GPU for further iterations.

To optimize performance and reduce data transfer overhead, four additional implementation versions of the L-BFGS algorithm were developed. In each version, a specific line search method (backtracking, Armijo interpolation, Wolfe interpolation, or backtracking-Wolfe) is directly integrated into the L-BFGS function. This integration allows the line search [7] to directly access GPU variables without the need to transfer them back to the host. By eliminating the data transfer between the device and host, these versions significantly reduce latency and improve computational efficiency. Each integrated line search method is tailored to work seamlessly with the GPU-accelerated L-BFGS algorithm, ensuring that the step size α_k is computed efficiently while maintaining the accuracy of the optimization process. This approach is particularly beneficial for large-scale problems, where minimizing data transfer overhead is critical for achieving high performance.

F. Convergence Check

Convergence is checked by computing the norm of the gradient vector g using the CUBLAS `ddot` function. If the norm falls below a specified tolerance, the algorithm terminates and returns the current solution.

G. Differences Between Implementation and Pseudocode

The implementation differs from the pseudocode in several key aspects:

- The pseudocode does not explicitly mention the use of CUDA streams for overlapping computation and data transfer, which is a significant optimization in the implementation.
- While the pseudocode refers to a generic line search function `mcsrch`, the implementation explicitly offers a choice of 4 line search methods such as backtracking and interpolation using Wolfe or Armijo conditions.
- Hessian update is described in terms of a general matrix $H0_k$ in the pseudocode, whereas the implementation

assumes a diagonal Hessian and updates it using a simple scaling factor ys/yy.

- The describe convergence evaluation is by comparing the norm of the gradient to the norm of the function value, while the implementation compares the gradient norm to a fixed tolerance.
- The pseudocode describes the two-loop recursion in an abstract manner, whereas the implementation explicitly handles it using CUBLAS routines and additional checks for edge cases.

IV. Experiments

A. GPU Implementation

We begin by implementing a sequential version of the algorithm before extending it to a parallel implementation that offloads computationally intensive operations to the GPU. The initial parallel version retains sequential line searches, following previous works. To evaluate performance, we use the Rosenbrock function, a benchmark used in literature [2] and [3]. The Rosenbrock function is particularly challenging due to its narrow, curved valley leading to the global minimum, making it highly sensitive to optimization algorithms. This characteristic makes it an effective test for assessing both convergence behavior and numerical stability. For consistency, we conduct five independent runs and report the average performance, using the function domain $[10, 10]$. We implement the Armijo backtracking line search strategy. Although Jacobi preconditioning was attempted in the sequential version, it did not yield improvements.

All experiments are conducted on Google Cloud [9]. The hardware setup includes an NVIDIA Tesla T4 GPU with 16 GB of VRAM, an Intel Xeon processor, and 12 GB of RAM. The cloud-based environment ensures consistency and accessibility while allowing us to leverage GPU acceleration. Given the inherently sequential nature of the algorithm, the extent of GPU acceleration depends on the problem scale. While smaller-dimensional problems see limited benefits, large-scale optimization tasks demonstrate more significant improvements.

As shown in Fig. 1, a speedup of 1.2× is observed only when the problem dimension reaches 10,000, and a similar trend continues at 20,000 dimensions. This suggests that GPU acceleration becomes effective primarily for high-dimensional optimization tasks, where the computational cost justifies parallel execution. The results indicate that while GPU-based implementations may not always provide immediate benefits for small-scale problems, they become increasingly advantageous as the dimensionality of the problem grows.

B. Line Search Comparison

When comparing the performance of these line searches on the same optimization problem, we observe that Armijo-based methods achieve faster results in both sequential and parallel cases (Table 1). One reason for this is that Armijo only enforces a sufficient decrease condition, making it less

Fig. 1. Optimizing Rosenbrock, time averaged on 5 runs. In our experiments, 10000 dimensions were needed to achieve 1.2x speedup.

restrictive than Wolfe, which also checks for curvature conditions. These additional constraints slow down Wolfe-based methods, requiring more function and gradient evaluations, thereby increasing computational overhead. This explains why Armijo-based searches tend to be faster in practical implementations, despite their lower theoretical guarantees on step-size selection.

The Armijo condition is governed by the constant 1e-4, which ensures a sufficient decrease in the objective function, while the Wolfe condition introduces an additional curvature constant 0.7, which enforces a stricter requirement on the gradient's behavior. The initial step size for line search is set to 1.0 and backtracking parameters, such as the step reduction factor to 0.5 and the convergence threshold 1e-8 further influence the efficiency of the line search. For Wolfe interpolation, the search range is bounded by a minimum of 1e-10 and a maximum of 10.0, ensuring that the step size remains within practical limits.

Examining the convergence behavior over 5000 iterations, we find that Armijo-based methods not only achieve satisfactory results faster but also require fewer iterations to do so. This could be due to the more aggressive step-size reductions in Wolfe searches, which slow down progress when the curvature condition is difficult to satisfy. Armijo's simpler condition allows it to take larger steps when possible, leading to faster convergence in practice. This highlights the trade-off between theoretical guarantees and practical performance, where Wolfe conditions provide stronger convergence properties but may not always lead to the most efficient optimization in a given computational setting.

C. Line Search Parallelization

From a parallelization perspective, different line search strategies exhibit varying levels of efficiency. Backtracking Armijo and Backtracking Wolfe rely on iterative step-size reductions, making them inherently sequential and difficult to parallelize effectively. In contrast, interpolation-based methods, such as Armijo Interpolation and Wolfe Interpolation,

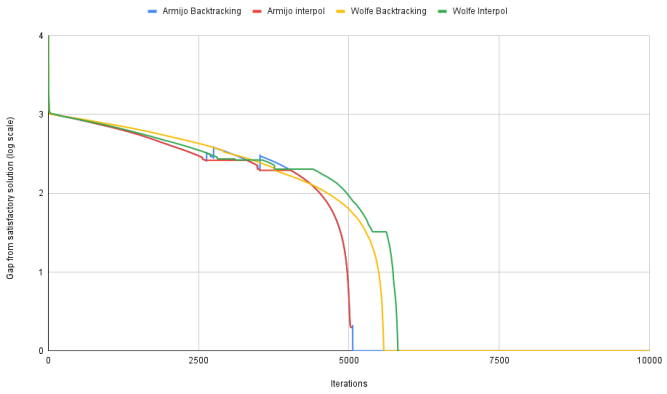

Fig. 2. Comparison of four line search strategies applied in our CUDA implementation. Armijo line search strategies show faster convergence.

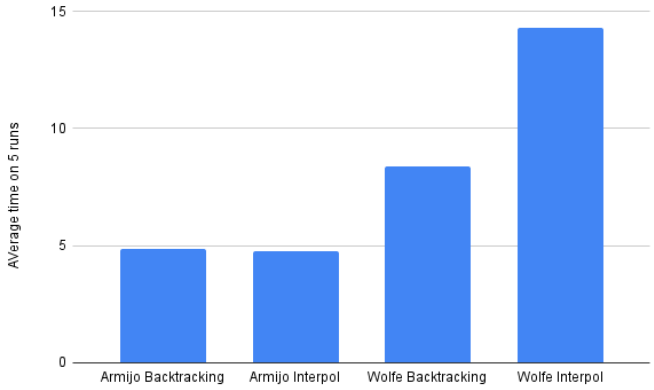

Fig. 3. Elapsed time comparison on four different line search strategies. Armijo line searches show faster execution time.

allow multiple function evaluations to be performed in parallel, improving efficiency. Among these, Wolfe Interpolation has the highest parallelization potential since it evaluates multiple candidate points before refining the step size, making it more suitable for GPU acceleration.

We extend our previous implementation by incorporating parallelized versions of line searches, allowing these operations to be executed more efficiently on the GPU. While the core optimization framework remains unchanged, the line search component is now adapted to leverage parallel computations, reducing the sequential bottlenecks present in our initial approach. To evaluate the effectiveness of this modification, we conduct experiments on the Rosenbrock function in 10,000 dimensions, using the standard domain [2.048,2.048], and compare different line search strategies under the new parallel framework.

The results (table I, figure 4) confirm our expectations regarding the parallelization capabilities of the different line search methods. While all approaches benefit from parallelization, the extent of improvement varies. Armijo-based methods, which are inherently less parallelizable due to their sequential step-size reduction process, exhibit only modest gains. Specifically, we observe speedups of 1.12 and 1.11 for Armijo Backtracking and Armijo Interpolation, respectively. In contrast, Wolfe-based methods, which have greater potential for parallel execution, show significantly higher speedups. Wolfe Backtracking achieves a speedup of 3.55, while Wolfe Interpolation reaches 1.79. These improvements highlight the advantage of parallelized interpolation methods, particularly for large-scale optimization problems.

TABLE I
SPEEDUP OF DIFFERENT LINE SEARCH STRATEGIES IN 10000D ROSENBROCK OPTIMIZATION

Line Search Strategy	Speedup
Armijo Backtracking	1.12
Armijo Interpolation	1.11
Wolfe Backtracking	3.55
Wolfe Interpolation	1.79

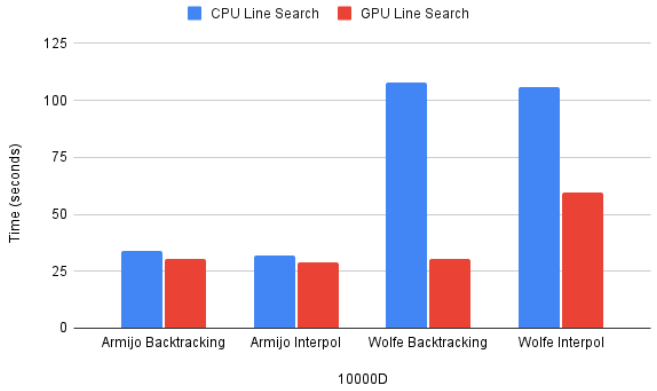

Fig. 4. Line search methods parallelized elapsed times. Slight speedups occur even on Armijo line searches, and Wolfe searches vastly benefit from GPU utilization.

Notably, the stronger speedups observed in Wolfe-based searches allow them to close the performance gap with Armijo-based methods. While Armijo was previously the faster option due to its simpler step-size condition, parallel execution now enables Wolfe methods to nearly match or at least come close to Armijo's overall efficiency. This suggests that, with sufficient computational resources, Wolfe-based searches may become more viable for large-scale optimization tasks, offering both improved convergence properties and competitive execution times.

Overall, our findings demonstrate that parallelization benefits all line search strategies, but to varying degrees. Armijo-based methods see limited improvements due to their sequential nature, whereas Wolfe-based methods leverage parallelization more effectively. This makes Wolfe Interpolation particularly promising, as it balances theoretical guarantees with practical efficiency in large-scale GPU-accelerated optimization.

V. Conclusion

In this work, we demonstrated that GPU acceleration is highly effective for large-scale optimization problems, particularly when dealing with high-dimensional datasets. By experimenting with up to 10,000 dimensions, we highlighted the significant improvements in computation time and efficiency achievable with GPU-powered methods. This showcases the potential of leveraging GPUs for handling complex problems that would otherwise be computationally prohibitive on standard CPUs.

We also explored the impact of different line search strategies on the performance of optimization algorithms. Our results indicate that Wolfe line searches provide a substantial speedup compared to other strategies, making them an excellent choice for large-scale optimization tasks. The empirical findings underscore the importance of selecting the appropriate line search method in order to maximize efficiency in optimization workflows.

Looking ahead, future work will focus on further optimizing GPU implementations and exploring additional line search strategies to enhance their performance. Additionally, we plan to investigate how these methods can be scaled to even higher-dimensional problems and integrated into real-world applications, particularly in areas such as machine learning and data science, where large-scale optimization is crucial.

References

[1] Richard Byrd et al. A limited memory algorithm for bound constrained optimization. SIAM Journal on Scientific Computing, 16(5), 1995.

[2] Jens Wetzl, Oliver Taubmann, Sven Haase, Thomas Köhler, Martin Kraus, and Joachim Hornegger. Gpu-accelerated time-of-flight super-resolution for image-guided surgery. In Hans-Peter Meinzer, Thomas Martin Deserno, Heinz Handels, and Thomas Tolxdorff, editors, Bildverarbeitung für die Medizin 2013, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] L. D'Amore, G. Laccetti, D. Romano, G. Scotti, and A. Murli. Towards a parallel component in a gpu–cuda environment: a case study with the l-bfgs harwell routine. International Journal of Computer Mathematics, 92(1):59–76, 2015.

[4] Huiyang Xiong, Bohang Xiong, Wenhao Wang, Jing Tian, Hao Zhu, and Zhongfeng Wang. Efficient fpga-based accelerator of the l-bfgs algorithm for iot applications. pages 1–5, 05 2023. doi: 10.1109/ISCAS46773.2023.10181544.

[5] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. Mathematical Programming, 45(1–3), 1989.

[6] Harris, Mark, et al. Optimizing parallel reduction in cuda.

[7] Huamin Wang and Yin Yang. Descent methods for elastic body simulation on the gpu. ACM Trans. Graph., 35(6), December 2016. ISSN 0730-0301. doi: 10.1145/2980179. 2980236. URL https://doi.org/10.1145/2980179.2980236.

[8] Nvidia. (2025.) cuBLAS. https://developer.nvidia.com/cublas

[9] Google. (2025). Google Colaboratory. https://colab.research.google.com/