

MaReCo: Configurable Map and Reduce Algorithmic Skeletons with Closure Support for GPUs in Elixir

Frederico Peixoto Antunes¹, André Rauber Du Bois¹,
Gerson Geraldo Homrich Cavaleiro¹

¹Centro de Desenvolvimento Tecnológico(CDTec),
Universidade Federal de Pelotas(UFPel)
Caixa Postal 354 – 96.010-610 – Pelotas – RS – Brazil

{fpantunes, dubois, gerson.cavaleiro}@inf.ufpel.edu.br

Abstract. *The rising demand for high-performance computing has boosted the adoption of abstractions that simplify parallel programming. One such abstraction is the use of algorithmic skeletons, which encapsulate recurring patterns such as Map and Reduce. This work presents MaReCo, a configurable Map and Reduce algorithmic skeleton library for GPUs in the Elixir language, providing skeletons whose behavior can be customized through parameters and whose computations can capture values from the surrounding Elixir environment through closures. This work seeks to reconcile expressiveness and efficiency by evaluating its performance against handwritten CUDA and CPU-based Elixir implementations across five benchmark applications. Results show that MaReCo achieves competitive performance compared with handwritten CUDA, with differences close to 1% for N-Body, while consistently outperforming the CPU-based Elixir implementations.*

1. Introduction

Since their commercial introduction in the late 1990s, GPUs have become ubiquitous across a wide range of computational domains, extending far beyond graphics processing to areas such as Machine Learning, scientific simulations, and large-scale data processing [Owens et al. 2008]. Despite their high computational capabilities, programming GPUs remains challenging, requiring programmers to explicitly manage memory transfers, execution hierarchies, and thread synchronization, as in programming models such as CUDA [Nickolls et al. 2008].

To reduce this complexity, several high-level abstractions have been proposed for parallel programming. Among them, algorithmic skeletons encapsulate recurring patterns of parallel computation as reusable higher-order functions, allowing programmers to express computations declaratively while delegating execution details to optimized implementations [Cole 1989]. Skeleton libraries have been proposed for several programming languages and execution platforms, including GPUs, demonstrating that high-level abstractions can simplify parallel programming while preserving efficient execution.

For the Elixir programming language [Valim and the Elixir Team 2026], PolyHok provides the underlying infrastructure required for GPU execution by supporting higher-order kernels generated through metaprogramming [Du Bois and Cavaleiro 2025]. While this enables user-defined functions to be incorporated into GPU kernels, it does

not provide reusable parallel programming abstractions such as configurable algorithmic skeletons supporting closures. As a result, programmers must still explicitly express common parallel patterns when developing GPU applications.

This paper presents MaReCo, a library that builds upon `PolyHok` to provide configurable *Map* and *Reduce* algorithmic skeletons for GPU programming in Elixir. Their behavior can be customized through configuration parameters, while MaReCo internally specializes each invocation according to its arguments and configuration. The implementation and evaluation presented in this paper focus on CUDA, the backend supported by the original `PolyHok` implementation. A more recent version of `PolyHok` also supports selecting an OpenCL backend, but MaReCo’s compatibility with that backend has not been evaluated.

In addition, MaReCo integrates *closures* into the configurable skeletons, allowing computations to capture values from the surrounding Elixir environment.

The main contributions of this work are:

- the design and implementation of configurable *Map* and *Reduce* algorithmic skeletons for GPU execution in Elixir;
- a specialization mechanism that combines Elixir’s pattern matching with dispatch to automatically select specialized GPU implementations generated through `PolyHok`;
- the integration of *closures* into configurable algorithmic skeletons;
- an experimental evaluation comparing MaReCo with handwritten CUDA and Elixir implementations using the `EXLA` backend across five benchmark applications.

This paper is organized as follows. Section 2 presents the background on algorithmic skeletons and `PolyHok`. Section 3 reviews related work. Section 4 describes the design and implementation of the MaReCo library. Section 5 presents the experimental evaluation and discusses the obtained results. Finally, Section 6 concludes the paper and outlines directions for future work.

2. Background

This section reviews the foundational concepts underlying this work. We first discuss the principles of algorithmic skeletons in parallel computing, followed by the execution patterns of *Map* and *Reduce*, and conclude with an overview of the `PolyHok` DSL.

2.1. Algorithmic Skeletons

Algorithmic skeletons were introduced by Murray Cole as a structured approach to parallel programming based on reusable patterns of computation [Cole 1989]. Rather than allowing arbitrary parallel programs, skeletons provide a restricted set of high-level abstractions for which efficient parallel implementations can be developed. This approach shifts the focus from explicitly coordinating parallel execution to selecting an appropriate computational pattern. Consequently, programmers describe *what* computation should be performed, while the skeleton determines *how* it is executed in parallel. This approach has since been explored extensively across different programming models and execution environments [Cardoso 2022, Herrmann 2024, Qummar et al. 2025].

Skeletons combine reusable algorithmic structures with higher-order functions, encapsulating concerns such as thread organization, synchronization, communication, and memory management. Beyond data-parallel patterns, the broader literature on algorithmic skeletons also encompasses task-parallel abstractions such as *Pipeline* and *Farm*, as well as spatial patterns like *Stencil* [Ciechanowicz et al. 2009]. By separating computational patterns from low-level execution details, this approach allows the same parallel structure to be reused across applications while relying on optimized implementations for the target architecture, making skeletons particularly attractive for GPU programming.

2.2. Map and Reduce

The *Map* and *Reduce* skeletons constitute fundamental building blocks of many data-parallel applications. The *Map* skeleton applies a user-defined function independently to each element of one or more input collections, producing a corresponding output collection. Because each function application is independent, *Map* can be executed efficiently on GPU architectures, where thousands of threads process different elements concurrently [Bird and Wadler 1988]. Conversely, the *Reduce* skeleton performs the complementary aggregation by combining collection elements into a single result through an associative function, starting from an initial value [Bird and Wadler 1988]. Associativity enables partial reductions to be computed independently and hierarchically merged without affecting the final result.

Practical applications often require execution variants, such as multi-dimensional indexing, coordinate awareness, in-place updates, or logical strides over contiguous buffers. Rather than introducing distinct skeleton abstractions for each pattern, MaReCo unifies these execution variants under a configurable parameter interface.

2.3. PolyHok

PolyHok [Du Bois and Cavalheiro 2025] is an embedded domain-specific language (DSL) for GPU programming in Elixir that provides the infrastructure required to define, generate, and execute CUDA kernels through Elixir’s metaprogramming facilities. A central feature is its support for higher-order kernels, which allows user-defined functions to be incorporated into generated GPU kernels. During code generation, the supplied function becomes part of the generated CUDA code, enabling specialized execution while preserving a high-level programming interface.

PolyHok also provides runtime polymorphic specialization, allowing generated kernels to adapt to different data types without manually written implementations, while kernel generation and compilation are performed transparently at runtime. Data exchanged between the host and GPU is represented through GNX structures, which encapsulate GPU-resident arrays and provide the interface for data transfers and kernel execution. MaReCo builds its configurable algorithmic skeletons on these abstractions, delegating kernel generation, specialization, and GPU execution to PolyHok.

3. Related Work

Algorithmic skeletons have been extensively explored as a high-level approach to parallel programming, motivating the development of libraries that encapsulate recurring parallel computation patterns while providing optimized implementations for different

hardware platforms. Among the most established GPU-oriented skeleton frameworks are `SkePU` [Enmyren and Kessler 2010, Ernstsson et al. 2018, Ernstsson et al. 2021, Qummar et al. 2025], `Muesli` [Ciechanowicz et al. 2009, Ernsting and Kuchen 2012, Ernsting and Kuchen 2017, Herrmann et al. 2024], and `Marrow` [Marques et al. 2013, Cardoso 2022]. These frameworks target multicore CPUs and GPUs through backends such as CUDA and OpenCL. Rather than exposing only the fundamental *Map* and *Reduce* skeletons, they typically provide multiple specialized variants, such as *MapReduce*, *MapOverlap*, and *MapPairs*, each designed for a particular execution pattern.

GPU programming has also been explored in functional programming environments. `Accelerate` [Chakravarty et al. 2011] provides algorithmic skeletons for expressing data-parallel GPU computations, while `Futhark` [Henriksen et al. 2017] offers similar high-level abstractions through second-order array combinators (SOACs), such as *map* and *reduce*. Although differing in terminology, SOACs play essentially the same role as algorithmic skeletons [Cardoso 2022]. While `Accelerate` is an embedded language for Haskell and `Futhark` is a dedicated data-parallel programming language, `MaReCo` instead extends the existing Elixir ecosystem as a library built on top of `PolyHok`, using Elixir’s metaprogramming facilities rather than introducing a separate compiler or programming language.

Another distinguishing aspect concerns support for *closures*. Although several functional GPU systems support anonymous functions or higher-order combinators, the treatment of captured variables differs considerably. `Futhark` supports *closures* internally through compiler transformations, whereas `Accelerate` embeds captured values into the generated array expressions before JIT compilation. In contrast, `MaReCo` integrates *closure* support into its configurable skeletons, allowing values from the surrounding Elixir scope to be captured transparently by GPU kernels without extending the skeleton interface.

GPU programming has also been explored through other high-level systems. JIT compilation frameworks such as `Numba` [Lam et al. 2015] and `Triton` [Tillet et al. 2019] generate GPU kernels from user-defined code, while `ÆminiumGPU` [Fonseca and Cabral 2013] provides high-level data-parallel operations for Java with automatic CPU/GPU execution selection. Unlike `MaReCo`, these systems do not provide configurable higher-order algorithmic skeletons: `Numba` and `Triton` focus on user-defined GPU kernels, while `ÆminiumGPU` provides predefined execution patterns. `MaReCo` instead exposes a small set of configurable algorithmic skeletons whose behavior is specialized through interface-level configuration while relying on higher-order kernel generation.

Within the Elixir ecosystem, `Nx` [Valim et al. 2021] provides multidimensional arrays and numerical tensor operations with CPU and GPU backends through `EXLA`. GPU computations are expressed through tensor operations rather than configurable algorithmic skeletons, and arbitrary user-defined functions cannot be incorporated into generated GPU kernels. `MaReCo` complements this ecosystem by enabling configurable *Map* and *Reduce* skeletons built on top of `PolyHok`.

The present work builds upon a line of research on GPU programming in Elixir. `GPotion` [Du Bois and Cavalheiro 2023] introduced a domain-

specific language (DSL) for GPU programming in Elixir, but did not support higher-order kernels. `Hok` [Du Bois et al. 2024] extended this approach by enabling higher-order kernels, although still relying on statically inferred types. `PolyHok` [Du Bois and Cavalheiro 2025] further advances this line of work by introducing runtime polymorphism and just-in-time compilation of higher-order GPU kernels, allowing higher-order kernel abstractions to operate on values of different types without requiring modifications to the Elixir compiler.

Unlike previous skeleton libraries, MaReCo adopts a configurable approach in which different execution behaviors are expressed through the parameterization of a small set of skeletons rather than through multiple specialized abstractions. This design allows a single *Map* interface to represent execution patterns that are commonly exposed as distinct skeletons in other frameworks, while preserving specialized implementations internally. MaReCo further integrates *closure* support into this configurable interface, allowing computations to capture values from the surrounding Elixir environment. To the best of our knowledge, MaReCo is the first GPU skeleton library for Elixir to use configuration parameters to express different execution patterns through a common skeleton interface.

4. MaReCo

MaReCo is a library of configurable algorithmic skeletons for GPU programming in Elixir built on top of `PolyHok`. Rather than exposing low-level GPU programming primitives, it provides high-level *Map* and *Reduce* skeletons, allowing execution behavior to be customized through configuration parameters. Internally, MaReCo relies on `PolyHok` to generate specialized CUDA kernels while exposing a uniform programming interface to the programmer.

Besides introducing configurable algorithmic skeletons, this work also integrates *closure* support into a GPU skeleton library, allowing user-defined computations to capture values from the surrounding Elixir environment without extending the skeleton interfaces.

4.1. Programming Model: Skeletons and Closures

MaReCo expresses GPU computations through operations over `GNX` structures, which represent arrays allocated in GPU memory. A computation consists of applying one of the available algorithmic skeletons to one or more `GNX` structures together with a user-defined function describing the computation to be performed.

The general form of the *Map* skeleton is

```
map(<gnx...>, <function> [,params...] [,options]) → gnX,
```

where the invocation receives one to three `GNX` structures, the function to be executed, an optional list containing up to three auxiliary parameters, and a list of configuration options. The auxiliary parameters are forwarded unchanged to every invocation of the user-defined function, while the configuration options determine characteristics such as the dimensionality of the computation, whether coordinates are provided to the function, and how the output is handled.

Additional values required by the computation can either be provided explicitly through the auxiliary parameters of the skeleton or implicitly through closures. Both approaches are supported by the same skeleton interface.

The *Reduce* skeleton performs an aggregation over a single GNX structure according to

$$\text{reduce}(\langle \text{gnx} \rangle, \langle \text{initial} \rangle, \langle \text{function} \rangle) \rightarrow \text{value},$$

where *initial* specifies the initial value of the reduction accumulator and the user-defined function combines partial results. Since the reduction is executed in parallel, this function must be associative to guarantee that different evaluation orders produce the same result.

The example in Code 1 illustrates the use of the *Map* skeleton together with a *closure*.

```
1  require PolyHok
2  use Ske
3
4  x = 10
5
6  arr = Nx.tensor([1,2,3,4])
7  fun = PolyHok.clo fn y -> x * y end
8
9  a = PolyHok.new_gnx(arr)
10
11 result = Ske.map(a, fun, [])
12      |> PolyHok.get_gnx()
```

Code 1. MaReCo’s Map skeleton using the closure capability.

The module *Ske* provides the programming interface of MaReCo, exposing the configurable *Map* and *Reduce* skeletons. The *new_gnx* and *get_gnx* primitives are responsible for transferring data between the host and GPU memory. User-defined GPU functions are normally created with the *PolyHok.phok* macro, while closures are created with *PolyHok.clo*, which additionally captures values from the surrounding Elixir scope. Both macros convert the supplied function into an intermediate representation that can later be incorporated into a generated CUDA kernel by *PolyHok*.

Closures are anonymous functions that capture values from their surrounding lexical scope. Unlike ordinary anonymous GPU functions, which operate only on their explicit parameters, closures retain access to external values defined in the host program. In *PolyHok*, captured values are incorporated directly as additional arguments to the generated GPU kernel, allowing skeleton computations to be specialized without modifying the skeleton interface. This improves the expressiveness of the configurable skeletons.

Another characteristic of MaReCo is the support for logical elements composed of multiple consecutive values through the *step* attribute associated with each GNX structure.

Although data is stored as a one-dimensional array in GPU memory, the *step* specifies how many contiguous positions correspond to each logical element. During execution, the skeleton advances through the array according to this value and provides the grouped elements as separate arguments to the user-defined function. This mechanism enables the representation of structured data, such as vectors or particles with multiple attributes, while preserving a contiguous memory layout suitable for GPU execution.

4.2. Configurable Map and Reduce

Unlike many skeleton libraries that introduce distinct abstractions for different execution patterns (e.g., *MapOverlap*, *MapPairs*, or in-place variants), MaReCo adopts a configurable design centered on a single *Map* abstraction. Different behaviors are selected through configuration parameters rather than through different skeleton interfaces. This design reduces interface verbosity while allowing a broad range of execution patterns to be expressed through the same programming abstraction.

The *Map* skeleton provides the following configuration options:

```
[coord:  true|false, return:  true|false, dim:  :one|:two]
```

Unless explicitly specified, omitted configuration options assume the underlined default values.

The `coord` option enables the coordinates of the processed element to be passed as additional arguments to the user-defined function, allowing position-aware computations. The `return` option specifies whether the user-defined function produces a value to be stored in a newly allocated GNX structure, or whether no output GNX is allocated and the function handles the result directly through the provided GNX structures. The `dim` option specifies whether the computation is interpreted over one- or two-dimensional data.

Besides these options, the *Map* skeleton accepts one to three input GNX structures together with up to three auxiliary parameters. These parameters are forwarded unchanged to each invocation of the user-defined function, enabling computations with different arities while preserving the same programming interface.

The combination of configuration options, the number of input structures, and the auxiliary parameters allows a single skeleton abstraction to express a variety of execution patterns that would traditionally require multiple specialized skeletons. The selection of the appropriate implementation is performed internally by MaReCo and is transparent to the programmer.

For the *Reduce* skeleton, execution follows the canonical pattern of parallel reduction over a single GNX structure. Rather than requiring interface-level flags, its specialization is driven directly by the reduction operator, the data type, and the associative combining function supplied by the programmer.

4.3. Implementation

Internally, MaReCo translates high-level skeleton invocations into specialized GPU executions generated through PolyHok. Rather than relying on a single generic implementation capable of handling every possible combination of arguments, the library defines multiple specialized function clauses corresponding to the supported invocation forms.

For the *Map* skeleton, specialization occurs in two stages. First, Elixir’s *pattern matching* mechanism selects the appropriate function clause according to the structure of the call. This selection is based on the function arity and on the organization of the supplied arguments, including the number of input GNX structures and auxiliary parameters. Consequently, no explicit inspection of the argument list is required within the implementation.

After the appropriate function clause has been selected, a dispatch mechanism determines the specialized implementation according to the configuration parameters supplied by the programmer. In particular, the combination of the `coord`, `return`, and `dim` options determines which specialized implementation is responsible for preparing and launching the corresponding GPU kernel.

The selected implementation prepares the execution by obtaining the information required from the input GNX structures, configuring the execution grid, allocating an output GNX when required, and invoking the corresponding CUDA kernel generated through PolyHok.

The generated kernels are implemented as higher-order kernels. Rather than embedding a fixed computation, they incorporate the user-defined function into the generated CUDA code during PolyHok’s code generation process. As a result, the same execution structure can be specialized for different computations while preserving a uniform high-level programming interface.

Closures extend this mechanism by allowing the user-defined function to retain access to values from its surrounding Elixir scope. This requires the closure to be transformed before it is incorporated into the generated kernel. The `PolyHok.clo` macro identifies the free variables in the closure’s AST, adds the corresponding arguments to the function representation, and generates a structure containing the transformed AST and the captured values. At runtime, when the kernel is compiled, these additional arguments are incorporated into the kernel and closure calls, allowing the captured values to be passed to the generated GPU function without extending the MaReCo skeleton interface.

Within the generated *Map* kernel, each GPU thread processes one logical element of the input data according to the selected dimensionality and the associated *step* size. The corresponding values are retrieved from the input GNX structures and passed to the user-defined function together with the auxiliary parameters and, when enabled, the coordinates of the processed element.

Complementing the transformation skeletons, the *Reduce* skeleton implements a unified parallel aggregation pattern. It receives one input GNX structure, an initial accumulator value, and an associative reduction function. The generated kernel performs a hierarchical parallel reduction in shared memory, progressively combining partial results until the final scalar value is obtained. As with *Map*, the user-defined reduction function is incorporated into the generated CUDA kernel by PolyHok.

This organization separates the programming interface from the execution mechanism. Programmers interact only with configurable skeleton abstractions, while pattern matching, specialization, kernel generation, and GPU execution remain entirely encapsulated within the MaReCo implementation.

5. Experimental Evaluation

The experimental evaluation has two objectives. First, it evaluates the performance gains obtained through GPU acceleration by comparing MaReCo with CPU-based Elixir implementations using the `Nx` library with the `EXLA` backend. Second, it quantifies the overhead introduced by the abstraction by comparing MaReCo with equivalent handwritten CUDA implementations.

The evaluation considers five benchmark applications that exercise different combinations of the proposed *Map* and *Reduce* skeletons. The experimental setup is presented first, followed by the benchmark applications and the analysis of the obtained results.

5.1. Experimental Setup

All experiments were performed on a system equipped with an AMD Ryzen 7 5700X processor (8 physical cores, 16 threads), 32 GB of DDR4 RAM, and an NVIDIA GeForce RTX 3060 GPU with 12 GB of GDDR6 memory. The software environment consisted of Ubuntu 22.04.4 LTS, Elixir 1.17.2 running on Erlang/OTP 27, the `Nx` numerical library configured with the `EXLA` backend using the `:host` client for CPU execution, and the CUDA Toolkit 11.5, with code compiled via `nvcc` and executed through the CUDA Driver API.

Two baselines were considered. The first consists of CPU-based Elixir implementations using `Nx` with the `EXLA` backend, allowing the performance gains of MaReCo’s GPU execution to be evaluated against CPU-based execution within the Elixir ecosystem. The second consists of handwritten CUDA implementations derived from the specialized code generated by PolyHok, with high-level skeleton abstractions removed while preserving the same computational structure and algorithmic steps. This comparison quantifies the performance difference between MaReCo’s high-level skeleton interface and specialized CUDA implementations under equivalent memory allocation and data transfer conditions.

For each benchmark, 30 independent executions were performed in fresh operating system processes to ensure statistical independence, and the arithmetic mean is reported. Statistical significance was confirmed using the Mann–Whitney U test [Mann and Whitney 1947] ($p < 0.001$). Reported execution times capture the complete workflow: GPU memory allocation, host-to-device transfers, runtime kernel generation and JIT compilation via NVRTC on the initial invocation, kernel execution, and device-to-host data retrieval. No warm-up phase was used, thus accounting for runtime compilation costs within the library’s lifecycle. Host-side data generation is excluded from the timings.

Input sizes were selected based on the limits of each execution platform. For the CPU-based `EXLA` baseline, problem sizes were constrained by host memory overhead, as intermediate tensor allocations within `EXLA` rapidly exhausted the 32 GB of system RAM on larger inputs. Conversely, GPU implementations maintain compact contiguous buffers without intermediate copying, allowing input sizes to be scaled up to the physical capacity of the 12 GB VRAM.

Table 1. Characteristics of the evaluated benchmark applications.

Benchmark	Map	Reduce	Configuration Used
Dot Product	X	X	Default (return: true, coord: false, dim: :one)
N-Body	X		return: false, step = 6
Ray Tracer	X		dim: :two, return: false, coord: true, step = 4
Nearest Neighbor	X	X	return: false
SAXPY	X		Default (return: true, coord: false, dim: :one)

5.2. Benchmarks

Five benchmark applications were selected to evaluate the execution patterns of the proposed skeletons: *Dot Product* and *SAXPY* follow the standard BLAS formulations [Dongarra et al. 1988]; *N-Body* simulates gravitational interactions among particles [NVIDIA 2023]; *Ray Tracer* renders three-dimensional spheres based on [Sanders and Kandrot 2010]; and *Nearest Neighbor* performs spatial distance queries, adapted from the Rodinia benchmark suite [Che et al. 2009].

Together, they exercise different usage patterns of the proposed skeletons, including one- and two-dimensional mappings, structured data layouts through the *step* mechanism, in-place computations, and compositions of *Map* and *Reduce*. The applications were chosen from representative workloads in Scientific Computing, Computer Graphics, and Data Processing. Table 1 summarizes the characteristics of each benchmark and the configuration parameters used.

5.3. Results

This section analyzes the performance of MaReCo across the evaluated benchmarks. Table 2 presents the comparison against CPU-based Elixir implementations using EXLA, while Table 3 compares MaReCo with handwritten CUDA implementations.

Table 2. Performance comparison with CPU-based Elixir implementations using EXLA.

	INPUT SIZE	MaReCo (ms)	EXLA CPU (ms)	Difference (%)
Dot Product	200M	593.00	638.00	7.59%
N-Body	5K	314.00	442.00	40.76%
Ray Tracer	2160	310.00	1748.00	463.87%
Nearest Neighbor	100M	375.00	587.00	56.53%
SAXPY	200M	461.00	529.00	14.75%

Table 2 shows that MaReCo consistently outperforms the CPU-based EXLA implementations across all evaluated applications, with speedups reaching up to 463.87% in the compute-intensive *Ray Tracer*. In memory-bound kernels such as *Dot Product*, the speedup is more modest (+7.59%), as execution time is dominated by memory traffic rather than raw arithmetic throughput.

Comparing MaReCo to handwritten CUDA (Table 3) reveals that high-level configurability introduces moderate overhead for most applications, typically between 23%

Table 3. Performance comparison with handwritten CUDA implementations.

	INPUT SIZE	CUDA (ms)	MaReCo (ms)	Difference (%)
Dot Product	900M	1054.25	1325.87	25.76%
N-Body	500K	82565.23	83549.93	1.19%
Ray Tracer	23170	3809.35	4710.53	23.66%
Nearest Neighbor	900M	683.78	929.43	35.93%
SAXPY	900M	2117.54	1188.47	-43.88%

and 36%. This difference encompasses skeleton dispatch, argument marshaling within the BEAM, and runtime JIT compilation via NVRTC.

This overhead corresponds to the cost of providing a configurable, high-level programming interface while dynamically dispatching specialized implementations. The results indicate that this overhead is primarily composed of fixed execution costs associated with host-side dispatch and runtime kernel generation rather than inefficiency in the generated GPU kernels. For short-running benchmarks (such as *Dot Product* or *Nearest Neighbor*, which complete in under one second), these fixed millisecond-scale costs represent a noticeable proportion of the total runtime.

In contrast, the *N-Body* benchmark runs for over 83 seconds, an execution time substantially longer than any other evaluated benchmark due to its intense arithmetic workload. In this scenario, the fixed dispatch overhead introduced by the skeleton abstraction is completely amortized by the long computation time, resulting in a performance difference of only 1.19% between handwritten CUDA (82565.23 ± 188.44 ms) and MaReCo (83549.93 ± 1117.97 ms). The low standard deviation across the 30 independent runs confirms the statistical stability of the measurements and demonstrates that abstraction costs become negligible for compute-intensive workloads.

Conversely, *SAXPY* presents an apparent anomaly where MaReCo outperforms handwritten CUDA by 43.88%. This result does not stem from superior GPU kernel execution, as both versions execute equivalent instructions on the device. Instead, it is explained by host memory management in the Erlang VM (BEAM). When the input vector is no longer referenced further in the application code following the skeleton call, the virtual machine can reuse the pre-allocated host memory buffer when retrieving results with `get_gnx`, rather than requesting a new memory allocation from the operating system [Rodrigues et al. 2026]. In contrast, the C++ CUDA implementation allocates a fresh host buffer for the result vector, incurring page allocation and mapping overheads across 900 million floating-point elements (approximately 3.6 GB).

Overall, the evaluation demonstrates that MaReCo provides an expressive, high-level abstraction in Elixir while delivering performance competitive with handwritten CUDA, achieving near-parity in compute-intensive workloads and offering substantial gains over CPU-based numerical execution.

6. Conclusion and Future Work

This paper presented MaReCo, a library of configurable algorithmic skeletons for GPU programming in Elixir built on top of the `PolyHok` DSL. By providing configurable *Map*

and *Reduce* abstractions whose behavior is customized through parameters, MaReCo automatically selects specialized GPU implementations, reducing the need for multiple specialized abstractions. The library also supports *closures*, allowing skeleton computations to capture values from the surrounding Elixir environment without extending skeleton interfaces.

Experimental results show that MaReCo achieves competitive performance while providing a higher-level programming interface. Compared with handwritten CUDA implementations, performance differences are moderate for most benchmarks, with *N-Body* exhibiting a difference close to 1%. Compared with CPU-based Elixir implementations using the EXLA backend, MaReCo consistently achieves substantial performance improvements through GPU execution. These results support the use of configurable algorithmic skeletons as a practical approach to GPU programming in Elixir.

Future work includes extending the specialization mechanism with additional execution patterns and configuration options for *Map* and *Reduce*, optimizing the *Reduce* implementation, and adding further algorithmic skeletons while preserving the configurable design.

Artifact Availability

The implementation of the proposed approach is available in the MaReCo repository¹

The artifact includes the complete implementation of the MaReCo library, the benchmark applications used in the experimental evaluation, and the scripts required to reproduce the reported results.

Detailed installation and execution instructions are provided in the repository documentation, including the required versions of Erlang, Elixir, and CUDA, as well as the complete workflow for compiling the framework and executing the benchmark suite.

Additional documentation about PolyHok and its programming model is available on the project's website²

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, and by FAPERGS (Grant No. 24/2551-0001372-5).

References

- Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice Hall International, Hertfordshire, UK.
- Cardoso, F. J. S. d. F. (2022). Extending, improving and optimizing Marrow. Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Lisboa, Portugal.

¹<https://github.com/hpfred/MaReCo>

²<https://lups.inf.ufpel.edu.br/polyhok>

- Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. (2011). Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14.
- Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee.
- Ciechanowicz, P., Poldner, M., and Kuchen, H. (2009). The Münster skeleton library Muesli: A comprehensive overview. In J. Becker, K. Backhaus, H. L. G., editor, *Working Papers, European Research Center for Information Systems*.
- Cole, M. I. (1989). *Algorithmic skeletons: structured management of parallel computation*. Pitman, London, UK.
- Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1988). An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1).
- Du Bois, A., Perlin, T., Antunes, F. P., and Cavalheiro, G. (2024). Hok: Higher-order GPU kernels in Elixir. In *Anais do XXVIII Simpósio Brasileiro de Linguagens de Programação*, pages 71–80, Porto Alegre, RS, Brasil. SBC.
- Du Bois, A. R. and Cavalheiro, G. (2023). GPotion: An embedded DSL for GPU programming in Elixir. In *Proceedings of the XXVII Brazilian Symposium on Programming Languages*, pages 1–8.
- Du Bois, A. R. and Cavalheiro, G. (2025). Polymorphic higher-order GPU kernels. In *European Conference on Parallel Processing*, pages 100–113. Springer.
- Enmyren, J. and Kessler, C. W. (2010). SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Fourth International Workshop on High-level parallel programming and applications*, pages 5–14.
- Ernsting, S. and Kuchen, H. (2012). Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138.
- Ernsting, S. and Kuchen, H. (2017). Data parallel algorithmic skeletons with accelerator support. *International Journal of Parallel Programming*, 45(2):283–299.
- Ernstsson, A., Ahlqvist, J., Zouzoula, S., and Kessler, C. (2021). SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *International Journal of Parallel Programming*, 49(6):846–866.
- Ernstsson, A., Li, L., and Kessler, C. (2018). SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80.
- Fonseca, A. and Cabral, B. (2013). *ÆminiumGPU: An Intelligent Framework for GPU Programming*, pages 96–107. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Henriksen, T., Serup, N. G., Elsmann, M., Henglein, F., and Oancea, C. E. (2017). Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571.

- Herrmann, N. (2024). *The Algorithmic Skeleton Library Revisited: The Münster Skeleton Library*. PhD thesis, Westfälische Wilhelms-Universität Münster, Münster, Germany. Dissertation, Deutsche Nationalbibliothek IDN 1355252458.
- Herrmann, N., Dieckmann, J., and Kuchen, H. (2024). Optimizing three-dimensional stencil-operations on heterogeneous computing environments. *International Journal of Parallel Programming*, 52(4):274–297.
- Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6.
- Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60.
- Marques, R., Paulino, H., Alexandre, F., and Medeiros, P. D. (2013). Algorithmic skeleton framework for the orchestration of GPU computations. In *Euro-Par 2013 Parallel Processing*, pages 874–885. Springer Berlin Heidelberg.
- Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53.
- NVIDIA (2023). Fundamentals of accelerated computing with CUDA C/C++. Online course, NVIDIA Deep Learning Institute. Accessed: 2026-06-01.
- Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899.
- Qummar, S., Ernstsson, A., Kessler, C., and Sysoev, O. (2025). SkePU-DNN: Algorithmic skeleton programming for deep learning on heterogeneous systems. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 423–432.
- Rodrigues, H. G., Du Bois, A. R., and Cavaleiro, G. G. H. (2026). Kernels de ordem superior para GPUs compatíveis com OpenCL utilizando metaprogramação em Elixir. In *Seminário Integrado de Software e Hardware (SEMISH)*, pages 422–433. SBC.
- Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, Boston, MA.
- Tillet, P., Kung, H.-T., and Cox, D. (2019). Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19.
- Valim, J., Moriarity, S., and contributors (2021). Nx: Multi-dimensional arrays (tensors) and numerical definitions for Elixir. <https://github.com/elixir-nx/nx>.
- Valim, J. and the Elixir Team (2026). Elixir programming language. Version 1.20.