Orchestra: CPU-GPU Cooperative Programming in Elixir

Henrique Gabriel Rodrigues
PPGC, Universidade Federal de Pelotas
Pelotas, RS, Brazil
henrique.grdr@inf.ufpel.edu.br

Eduardo Beloni Mailan
CDTec, Universidade Federal de Pelotas
Pelotas, RS, Brazil
ebmailan@inf.ufpel.edu.br

Gerson Geraldo H. Cavalheiro
PPGC, Universidade Federal de Pelotas
Pelotas, RS, Brazil
gerson.cavalheiro@inf.ufpel.edu.br

André Rauber Du Bois
PPGC, Universidade Federal de Pelotas
Pelotas, RS, Brazil
dubois@inf.ufpel.edu.br

Abstract

This paper presents Orchestra, a set of abstractions embedded in the Elixir language for defining parallel kernels that can be easily executed on CPUs and GPUs. This execution model enables heterogeneous cooperative programming, where different processors (the CPU and GPU) work together to solve a computational problem. Building upon the PolyHok DSL, Orchestra's kernels are polymorphic and higher-order, enabling the construction of parallel abstractions such as algorithmic skeletons that can execute across different devices. Orchestra is compatible with the Elixir Nx tensor library and combines low-level control over computation with high-level facilities, such as dynamic typing, garbage-collected CPU and GPU memory, and simplified host-device data transfers.

Keywords

DSL, Elixir, Parallel Programming, Heterogeneous Computing

1 Introduction

In recent years, Graphics Processing Units (GPUs) have been widely adopted to accelerate compute-intensive applications due to their massively parallel architectures, facilitated by GPGPU (General-Purpose computing on Graphics Processing Units) APIs [7, 28]. However, most heterogeneous applications rely exclusively on the GPU for acceleration, while the Central Processing Unit (CPU) either remains idle or executes tasks unrelated to the core computation [21]. This approach leads to suboptimal resource utilization, ignoring potential performance gains that could be achieved by exploiting the architectural strengths of each processor in a cooperative way.

Unfortunately, developing applications that leverage cooperative heterogeneous parallelism is not a trivial task. While cross-platform frameworks like OpenCL exist, the landscape is heavily fragmented by vendor-specific or architecture-specific tools, such as CUDA for NVIDIA GPUs, HIP for AMD GPUs, and Pthreads or OpenMP for CPUs [25]. Consequently, programmers often resort to using multiple APIs to achieve CPU-GPU cooperation [13]. This introduces severe maintainability challenges, as these low-level C/C++ based APIs require manual memory management, explicit inter-device data transfers, intricate thread synchronization, and substantial boilerplate code. Mixing different APIs with distinct programming mechanisms and paradigms further exacerbates this software engineering complexity.

Functional programming offers a natural foundation for reusable parallel abstractions, since computations can be expressed through higher-order operators such as `map`, `reduce`, and `scan` [12]. However, in heterogeneous programming, these abstractions are often lost when programmers must descend to low-level APIs and device-specific kernels [5]. As a result, the benefits of functional composition and code reuse rarely carry over to cooperative CPU-GPU programming.

The lack of higher-level abstractions, the inherent complexity of low-level systems programming, and the tight coupling between *coordination* code (responsible for device initialization, memory allocation, and kernel dispatching) and *computation* code (the actual parallel logic) makes CPU-GPU cooperation notoriously difficult. To address these challenges, this paper presents Orchestra, a set of abstractions embedded in the Elixir functional language for the design and coordination of heterogeneous computations. Orchestra abstracts the underlying API complexities while providing high-level features inherited from the PolyHok DSL [10], such as higher-order kernels, garbage-collected device memory, and dynamic typing.

In Orchestra, programmers can define device functions or anonymous functions to be passed as arguments to a kernel. This allows the creation of reusable algorithmic skeletons (such as `map`, `reduce`, and `scan`) that can be dispatched to either the CPU or GPU at runtime without code duplication. Additionally, Orchestra also allows for dynamic workload distribution, enabling the execution context to switch seamlessly between processors based on real-time application load, as demonstrated in Section 5.2.

This paper presents the following contributions:

- We present Orchestra, an Elixir-embedded set of abstractions for heterogeneous parallel orchestration. Orchestra extends the PolyHok DSL [10] – that provides the foundation for GNx device memory and kernel syntax – with orchestration mechanisms that allow programmers to seamlessly launch kernels on different execution targets. Because kernels defined in Orchestra are device-independent, they allow the creation of universal algorithmic abstractions, such as skeletons. The current implementation and architecture of Orchestra are detailed in Section 4.
- We evaluate the effectiveness of Orchestra's CPU parallel execution through a set of linear algebra benchmarks implemented using composable algorithmic skeletons. By comparing Orchestra against the Numerical Elixir (Nx) tensor library, we demonstrate that Orchestra's higher-order kernels and zero-copy tensor infrastructure can substantially

reduce runtime overheads associated with immutable tensor representations, achieving significant speedups on compute-intensive workloads.
- We present a case study of dynamic heterogeneous orchestration using a Breadth-First Search (BFS) benchmark adapted from the Chai benchmark suite [14]. The study evaluates Orchestra's ability to transparently coordinate CPU and GPU execution based on runtime workload characteristics. The results show that cooperative execution can yield statistically significant performance improvements over GPU-only execution when communication costs are low.

The source code for Orchestra and all experiments presented in this paper are publicly available in a GitHub repository[1]. The remainder of this paper is organized as follows: Section 2 provides the necessary background concepts; Sections 3 and 4 detail Orchestra's programming abstractions and implementation; Section 5 presents the experimental evaluation; Section 6 discusses related work; and Section 7 outlines conclusions and future work.

2 Background

2.1 Elixir and Metaprogramming

Elixir is a dynamic functional programming language compiled and executed on the Erlang Virtual Machine (BEAM), an environment known for low-latency, highly concurrent, and fault-tolerant systems [11]. The most relevant feature of Elixir for Orchestra is its metaprogramming support through *macros*. Macros receive as argument the Abstract Syntax Tree (AST) generated by the Elixir parser at compile time and output a newly transformed AST. This capability is heavily exploited by Orchestra and other Elixir-embedded DSLs to implement custom syntax and high-level abstractions without modifying the compiler [8–10].

Another critical feature of Elixir is its ability to interface with external Native Implemented Functions (NIFs). These are functions implemented in lower-level compiled languages (such as C, C++, or Rust) that utilize Erlang's development headers. NIFs allow Elixir to bypass the virtual machine for highly specific, performance-critical, or hardware-level tasks. Orchestra heavily relies on this interoperability to manage device memory and dispatch execution kernels, as detailed in Section 4.

2.2 PolyHok

PolyHok is a DSL embedded in the Elixir language designed to facilitate GPGPU computing [10]. PolyHok enables the implementation of polymorphic higher-order GPU kernels, allowing them to accept device functions – including anonymous device functions – as arguments when they are launched. It also brings high-level facilities, such as dynamic typed kernels, garbage-collected memory and integration with the Nx (Numerical Elixir) library.

By treating device functions as first-class values, PolyHok enables programmers to implement high-level abstractions typically associated with functional programming, such as composable algorithmic skeletons (e.g., `map` and `reduce`) and also array comprehensions. Listing 1 illustrates a `map` kernel implemented using the PolyHok DSL.

[1] https://github.com/Equiel-1703/orchestra

Listing 1: Map kernel implementation using PolyHok

```
require PolyHok

PolyHok.defmodule PMap do
    defk map_ker(a1,a2,size,f) do
        index = get_global_id(0)
        stride = get_global_size(0)

        for i in range(index,size,stride) do
            a2[i] = f(a1[i])
        end
    end
end
```

As seen in Listing 1, the `PMap` module defines a GPU kernel named `map_ker` using the `defk` keyword. The kernel computes the global thread index and the total grid stride using the standard built-in OpenCL functions `get_global_id` and `get_global_size`. A `for` loop is used to iterate over the array using a grid-stride loop. This ensures all elements are processed even if the array size exceeds the total number of launched threads. Inside the loop, the higher-order capability of the kernels is demonstrated: the kernel applies the function `f` (which was passed as an argument), to each element of the input array `a1`, storing the result in-place at the same index in array `a2`.

PolyHok was primarily designed for GPGPU programming. Therefore, its kernels can only be dispatched to the GPU. Orchestra extends this foundation by adding CPU execution capabilities and orchestration abstractions, transforming PolyHok's GPU-only paradigm into a fully cooperative heterogeneous environment.

2.3 Cooperative Heterogeneous Programming

Cooperative heterogeneous systems are computing environments that combine multiple distinct processor architectures (most commonly, CPUs and GPUs) to solve a single computational problem [17]. Each architecture is optimized for fundamentally different workloads and tasks. For instance, CPUs excel at sequential tasks and pointer chasing, while GPUs are engineered for high-throughput, massively parallel computations [26]. Cooperative heterogeneous programming seeks to leverage both processors simultaneously, dynamically routing computational tasks to the architecture best suited to execute them.

However, achieving true cooperation introduces significant challenges. On the software side, orchestration is notoriously complex because it usually requires dealing with different APIs and programming models [3, 18]. A developer might have to write and synchronize OpenMP threads for the CPU alongside OpenCL or CUDA kernels for the GPU. This forces the programmer to manually manage workload partitioning, inter-device synchronization, and memory consistency across entirely different programming paradigms. The lack of unified, high-level abstractions for heterogeneous cooperation is the primary barrier preventing the widespread adoption of this technique.

3 Orchestra

In this section, we present the main abstractions and features provided by Orchestra. We start by introducing the *with* primitive and execution contexts in Section 3.1. Then, we present device-agnostic

algorithmic skeletons, a new abstraction enabled by Orchestra, in Section 3.2.

3.1 Kernel Orchestrations

The main contribution of this paper is the orchestration abstractions introduced by Orchestra. To allow the programmer to seamlessly invoke kernels on the GPU or CPU, Orchestra offers the *with* primitive. This primitive defines an *execution context* for all Orchestra functions enclosed within its scope, automatically routing instructions to the device specified by the context[2]. To exemplify the usage of this primitive, Listing 2 demonstrates how the `map` skeleton presented in Listing 1 can be spawned on the GPU using the `Orchestra.gpu()` context.

Listing 2: Spawning the map kernel in the GPU

```
1  # Creates an array in the CPU memory with 1024 integers
2  arr_int = Orchestra.tensor(Enum.to_list(1..1024), :s32)
3
4  cpu_res =
5      # Create an Orchestra GPU context
6      Orchestra.with Orchestra.gpu() do
7          # Copy the integer array to the GPU memory
8          arr_int_gnx = Orchestra.new_gnx(arr_int)
9          # Creates an empty GPU array to hold the result
10         arr_result_gnx = Orchestra.new_gnx({1024}, :s32)
11
12         # Spawn kernel
13         Orchestra.spawn(
14             &PMap.map_ker/4,
15             {8,1,1},
16             {128,1,1},
17             [
18                 arr_int_gnx,
19                 arr_result_gnx,
20                 1024,
21                 Orchestra.hok fn x -> x + 1 end
22             ])
23
24         # Get result back to the CPU memory
25         arr_result_cpu = Orchestra.get_gnx(
26             arr_result_gnx
27         )
28     end
```

As seen in Listing 2, the execution starts by creating a host-side tensor with 1,024 integers using the `Orchestra.tensor` function (line 2). This function was introduced by Orchestra and returns a tensor fully compatible with the Numerical Elixir (Nx) library. The architectural reason CPU tensors are allocated via `Orchestra.tensor` rather than the traditional `Nx.tensor` routine involves the use of Shared Virtual Memory (SVM), which is detailed later in Section 4.2. Inside the GPU context (defined in line 6), this tensor data is transferred to the device using `Orchestra.new_gnx` (line 8), which allocates a GNx (GPU Nx) tensor in the GPU memory. The GNx abstraction is directly inherited from PolyHok, and simply represents an Nx tensor living in the GPU memory. In line 10, an empty GNx is created to store the `map` result.

To invoke the `map` kernel itself, the `Orchestra.spawn` function is utilized (line 13). The user must provide the target kernel reference, a tuple defining the grid dimensions, another tuple for the block dimensions, and a list containing the arguments to be passed to the kernel in the exact order of their declaration. Notably, the final argument passed is an anonymous device function created with the `Orchestra.hok` primitive (line 21). This primitive, also inherited from PolyHok, dynamically compiles the Elixir anonymous function provided into device code at runtime. In this example, it defines the scalar operation (`x + 1`) to be applied by the map skeleton to every element. Finally, in line 25, the result from the `map` kernel execution is transferred back to the CPU as an Nx tensor using `Orchestra.get_gnx`.

If the user wishes to execute the kernel on the CPU instead, the execution context can be simply changed to `Orchestra.cpu()`. This illustrates the symmetric design of Orchestra's API: the same `map` kernel can be dispatched to either the GPU or the CPU by changing the execution context and allocating the input and output tensors in the appropriate memory space, as shown in Listing 3.

Listing 3: Spawning the map kernel in the CPU

```
1  # Creates an array in the CPU memory with 1024 integers
2  arr_int = Orchestra.tensor(Enum.to_list(1..1024), :s32)
3  # Creates an empty tensor to hold the result
4  cpu_res = Orchestra.tensor({1024}, :s32)
5
6  # Create an Orchestra CPU context
7  Orchestra.with Orchestra.cpu() do
8      # Spawn kernel
9      Orchestra.spawn(
10         &PMap.map_ker/4,
11         {1024},      # Total number of work-items
12         {0},         # OpenCL-decided block size
13         [
14             arr_int,
15             cpu_res,
16             1024,
17             Orchestra.hok fn x -> x + 1 end
18         ])
19 end
```

Listing 3 highlights the device-agnostic nature of the higher-order kernels defined in Orchestra. The exact same `map_ker` definition is reused without any modifications. The primary differences lie in the execution context (`Orchestra.cpu()`) and the memory arguments: the CPU tensors (`arr_int` and `cpu_res`) can be passed directly to the kernel as arguments, without the need for `new_gnx` allocations. This is possible because the host CPU can directly access the underlying memory of the tensors, making the GNx abstraction unnecessary for CPU execution.

Furthermore, the launch dimensions provided to `spawn` in the CPU dispatch are explicitly adapted to the target hardware architecture. While the GPU implementation utilizes a smaller grid with 128 threads per block, the CPU implementation dispatches a configuration invoking 1,024 global work-items (the `{1024}` tuple), allowing the underlying OpenCL runtime to automatically determine the optimal work-group size (denoted by the special `{0}` tuple).

3.2 Device-Agnostic Algorithmic Skeletons

The execution context abstraction provided by Orchestra enables programmers to build reusable, device-agnostic higher-order algorithmic skeletons. Because the execution context is specified independently of the skeleton implementation, these abstractions

[2]In Orchestra, a *context* simply represents the target device on which kernels will be executed. Currently, the available execution contexts are `Orchestra.gpu()` and `Orchestra.cpu()`.

can be freely composed and orchestrated across heterogeneous devices. Listing 4 demonstrates this capability with a Dot Product implementation in which the *map* skeleton is executed on the GPU while the *reduce* skeleton in the CPU.

Listing 4: Composing device-agnostic skeletons in Orchestra

```
1  arr_1 = Orchestra.tensor({1024}, :s32, fn _ -> 1 end)
2  arr_2 = Orchestra.tensor({1024}, :s32, fn _ -> 2 end)
3
4  result =
5    Skeletons.map_2(
6      arr_1,
7      arr_2,
8      Orchestra.hok(fn x, y -> x * y end),
9      Orchestra.gpu())
10   |> Skeletons.reduce(
11     0,
12     Orchestra.hok(fn x, y -> x + y end),
13     Orchestra.cpu())
```

As shown in Listing 4, the developer can leverage Elixir's native pipe operator (`|>`) to construct a heterogeneous computational pipeline. First, the `map_2` skeleton performs an element-wise multiplication of the two input tensors (lines 5–9). By receiving `Orchestra.gpu()` as its execution context, the skeleton automatically allocates the required GNx structures, dispatches the computation to the GPU, and transfers the resulting tensor back to host memory. The resulting tensor is then piped as the first argument to the subsequent `reduce` skeleton, which executes on the CPU (`Orchestra.cpu()`) and accumulates the sum of the products (lines 10–13).

Internally, both `map_2` and `reduce` rely on Elixir's native pattern matching against the execution context to select the appropriate execution strategy. When executing within a CPU context, the skeleton passes the tensors directly to the kernel. In contrast, when in a GPU context, they transparently allocates the corresponding GNx structures, transfers data to device memory, launches the kernel, retrieves the result, and returns it as a standard Nx tensor. Consequently, heterogeneous memory management remains entirely encapsulated within the skeleton implementation, allowing programmers to compose device-agnostic operations through a unified functional API.

From the programmer's perspective, the execution context behaves as just another parameter of the skeleton, specifying the target execution device without affecting the algorithm itself. This design decouples algorithmic composition from device-specific execution, enabling the same functional skeletons to be reused across heterogeneous architectures without exposing memory-management details to the programmer.

4 Implementation

Orchestra is built upon the OpenCL implementation of PolyHok (known as OCL-PolyHok [24]). The primary advantage of this approach is that Orchestra achieves wide hardware portability across both CPUs and GPUs using a single unified API. This section details the internal memory management structures, the orchestration macros, and the compilation optimizations that enable cooperative heterogeneous execution.

4.1 GNx: GPU Nx Tensors

Inherited from PolyHok, Orchestra utilizes the Numerical Elixir[3] (Nx) library as its standard way of representing multi-dimensional numerical tensors of various data types in memory (host and device). Orchestra also uses PolyHok's GNx (GPU Nx) data structure. A GNx is essentially an Nx tensor residing in GPU memory. More specifically, a GNx acts as a host-side reference to an OpenCL memory buffer (`cl::Buffer`) allocated on the target device, typically the GPU's VRAM.

Because GNx tensors reside in isolated device memory, they cannot be directly manipulated by standard Elixir Nx functions and must be operated on exclusively within kernels. A critical architectural advantage of the GNx abstraction is its integration with the Erlang Virtual Machine (BEAM) Garbage Collector (GC). GNx structures are implemented as Erlang resources, and when a GNx tensor falls out of scope in the host application, the BEAM GC automatically invokes a custom C++ destructor. This destructor safely calls the OpenCL API to release the underlying memory object, thereby preventing memory leaks without requiring explicit manual memory management by the programmer.

A new GNx can be created using the `new_gnx` function and retrieved back to host memory as a new Nx tensor using the `get_gnx` function. A new performance optimization introduced by Orchestra is the ability to read a GNx and write its data directly into an already allocated Nx tensor on the CPU. This avoids the latency of allocating entirely new arrays in RAM to hold results, bypassing the lazy-allocation overhead of modern operating systems (as pointed in [24]). Listing 5 demonstrates this feature.

Another feature introduced in Orchestra to help mitigate PCIe bandwidth bottlenecks is the ability to specify the exact number of elements to transfer from the GPU, avoiding the overhead of copying the entire array when only a partial read is required. If the user doesn't provide this number, the `get_gnx` function will copy the entire array by default.

Listing 5: Reading GNx into existing CPU tensor

```
array_cpu = Orchestra.tensor(Enum.to_list(1..1024), :s32)

result_cpu =
    Orchestra.with Orchestra.gpu() do
        gpu_array_input = Orchestra.new_gnx(array_cpu)

        Orchestra.spawn(...)

        # Using the already allocated memory of array_cpu
      to save the result
        Orchestra.get_gnx(gpu_array_input, array_cpu)
    end
```

4.2 Orchestra Zero-Copy CPU Tensors

To enable efficient CPU parallelism via OpenCL (the underlying execution API used by Orchestra), the arrays provided to the kernels must be strictly aligned to the byte boundaries of the data types the kernel will use to access them [19]. An aligned array is one whose starting physical or virtual memory address is an exact multiple of its data type byte size. OpenCL imposes strict memory alignment

[3] https://nx.hexdocs.pm/Nx.html

rules for two primary reasons: architectural portability and performance. Unaligned memory access behavior varies drastically across devices. While some processors can handle it with a latency penalty, other architectures will trigger hardware faults or segmentation violations. Enforcing alignment ensures cross-platform stability.

From a performance perspective, modern CPUs heavily rely on Single Instruction Multiple Data (SIMD) vector instructions to process data in broad chunks (e.g., `float4` or `long16`). If the underlying memory is unaligned, the CPU memory controller requires multiple cache line fetches to populate a single vector register, severely degrading memory bandwidth and overall computational throughput.

These strict alignment requirements prevents standard Nx tensors from being passed directly to OpenCL CPU kernels. The binary data underlying an Nx tensor is allocated by the BEAM Virtual Machine's memory allocator, which does not guarantee the alignment constraints required by OpenCL . As a result, although Nx tensors reside in host memory, their layout may be unsuitable for safe and efficient kernel execution.

A naive solution to this problem would be to allocate an aligned `cl::Buffer`, copy the contents of the Nx tensor into this buffer before kernel execution, and copy the results back afterwards. However, this approach introduces two significant drawbacks. First, it causes redundant memory allocation, doubling the RAM usage by maintaining identical data in both the BEAM heap and the OpenCL buffer. Second, it introduces a severe memory copy overhead. The latency of constantly transferring data between unaligned and aligned buffers for every kernel launch can substantially reduce or even eliminate the performance gains obtained from CPU parallelization.

To circumvent these limitations, Orchestra implements a robust zero-copy memory model using OpenCL 3.0 Shared Virtual Memory (SVM). With this approach, CPU tensors are allocated through the `Orchestra.tensor` function. When invoked, Orchestra's C++ runtime requests an SVM buffer from the OpenCL driver, enforcing the strict byte alignments required by the target CPU architecture. The NIF then wraps this aligned memory pointer inside an Erlang Resource Binary and returns it to the host. This allows the BEAM Garbage Collector to seamlessly manage the buffer's lifecycle, while Elixir can wrap the binary into a standard Nx tensor, achieving total compatibility with the Nx library.

This design enables both the OpenCL runtime and the Elixir application to operate on the same physical memory without intermediate copies. Consequently, Orchestra eliminates redundant allocations and memory transfers while preserving the usability of Nx tensors. The result is a zero-copy execution model that enables CPU kernels to achieve near-native performance while maintaining seamless integration with Elixir's environment.

4.3 The *with* Macro

The orchestration abstraction introduced by Orchestra fundamentally relies on Elixir's robust macro system, specifically through the implementation of the `Orchestra.with` block. This macro is responsible for defining the hardware execution context (e.g., `Orchestra.gpu()` or `Orchestra.cpu()`) for a designated block of code.

At compile time, when the `with` macro expands, it intercepts the Abstract Syntax Tree (AST) of the enclosed expressions. The macro traverses this AST and injects the specified execution context into every Orchestra module command within its scope. Listing 6 demonstrates this metaprogramming transformation.

Listing 6: With macro expansion demonstration

```
Orchestra.with Orchestra.gpu() do
    gpu_array = Orchestra.new_gnx(cpu_array)

    Orchestra.spawn(
        &Example.ker/1,
        {1, 1, 1},
        {128, 1, 1},
        [gpu_array])

    Orchestra.get_gnx(gpu_array)
end

# ========== After Macro Expansion ==========

gpu_array = Orchestra.new_gnx(Orchestra.gpu(), cpu_array)

Orchestra.spawn(
    Orchestra.gpu(),
    &Example.ker/1,
    {1, 1, 1},
    {128, 1, 1},
    [gpu_array])

Orchestra.get_gnx(Orchestra.gpu(), gpu_array)
```

As illustrated in Listing 6, the `with` block modifies the abstract syntax tree by prepending the provided context (in this instance, the `Orchestra.gpu()`) as the implicit first argument to every relevant function call, such as `new_gnx`, `spawn`, and `get_gnx`.

This design ensures that every call to an Orchestra function (specially `Orchestra.spawn`) inherently knows which OpenCL device and command queue to target. It also decouples the kernel definition from its device target, allowing the exact same polymorphic kernel to be seamlessly routed to different processors dynamically.

4.4 Kernel Caching

Because Orchestra relies on dynamic typing, kernels must be Just-In-Time (JIT) compiled at runtime once the specific types of the incoming arguments are available. While OpenCL's JIT compilation is efficient, repeatedly parsing, AST-transforming, and JIT-compiling the OpenCL C code for every kernel launch would introduce a big latency, especially in iterative algorithms like graph traversals.

To reduce this overhead, Orchestra implements an internal kernel caching mechanism using a dedicated Elixir process known as the `module_server` to store it. Created during compile time, this server stores all kernel ASTs, types maps and compiled kernel resources. When the `Orchestra.spawn` function is invoked, Elixir dynamically generates a unique signature for the kernel dispatch. First, it maps the formal parameters of the kernel to the inferred types of the actual arguments provided by the user (e.g., mapping an array `a` to an integer pointer `:tint`). Second, it creates a map resolving any higher-order function arguments to their actual OpenCL string names. For a named device function, this might look like `f => "inc"`, whereas an anonymous function utilizes its auto-generated name, such as `f => "anon_2do4g0jge0"`.

These mappings are combined into a single keyword list and paired with the kernel's internal name and the target execution context to form a unique 3-tuple caching key. For example, a map kernel targeting the GPU would generate a key resembling `{:map_ker, [a: :tint, f: "inc"], :gpu}`. Orchestra sends a message containing this key to the `module_server` to request the corresponding compiled kernel.

If the kernel was previously compiled under this *exact* configuration, the server immediately returns the compiled kernel as an Erlang Term, completely bypassing the compilation pipeline. Conversely, if it results in a cache miss (i.e., the server returns `nil`), Orchestra proceeds to infer the types, generate the OpenCL C code, and compile it via the C++ runtime NIFs. The newly compiled kernel is then executed and subsequently stored in the `module_server` using the generated key. This architecture guarantees that the heavy JIT compilation overhead is paid exclusively on the first invocation of a specific kernel signature and device target, ensuring maximum performance for subsequent iterative dispatches.

5 Experiments

To evaluate both the runtime performance and heterogeneous orchestration capabilities of Orchestra, we conducted two sets of experiments. The first evaluates the performance gains of Orchestra's parallel CPU execution model and algorithmic skeletons over idiomatic Elixir tensor programming provided by the Nx library. The second is a case study based on a Breadth-First Search (BFS) benchmark adapted from the Chai benchmark suite [14], designed to assess Orchestra's support for cooperative heterogeneous execution between CPUs and GPUs.

All benchmarks were executed on a heterogeneous workstation equipped with an AMD Ryzen 7 9700X 8-Core processor (16 threads), 128 GB of RAM, and two NVIDIA GeForce RTX 4070 GPUs. The system runs Ubuntu 24.04.4 LTS and uses Elixir 1.17.3 with Erlang/OTP 27. The source code for all benchmarks is publicly available in Orchestra's GitHub repository.[4]

5.1 CPU Parallel Skeleton Evaluation

In this first experiment, we evaluate Orchestra as an alternative execution model for Elixir tensor programs, executing in parallel across the CPU cores. We implemented two fundamental linear algebra benchmarks: Vector Dot Product (DP) and Matrix Multiplication (MM).

To establish a baseline representing standard, idiomatic tensor operations in pure Elixir, the reference implementations were written using Numerical Elixir (`Nx`, version 0.12.1) with its default sequential `BinaryBackend` (`Nx.dot`). This comparison intentionally captures the architectural overhead imposed by the Erlang Virtual Machine (BEAM) during tensor computations in native Elixir. In contrast, Orchestra implementations express DP and MM using composable algorithmic skeletons executed within an `Orchestra.cpu()` context. In this context, Orchestra dispatches multi-threaded OpenCL CPU kernels across the available host CPU cores, executing the computational kernels outside the BEAM virtual machine. The goal of this experiment is therefore not to compare Orchestra against highly optimized numerical libraries or execution backends (e.g., cuBLAS or XLA), but rather to quantify the performance benefits available to developers when replacing idiomatic Elixir tensor implementations with Orchestra.

Both DP and MM in Orchestra were implemented as a parallel `map` skeleton followed by a `reduce` skeleton. Each benchmark was executed 30 times for three input sizes. The average times and standard deviations are reported in Table 1.

Table 1: Linear algebra benchmarks results. Times represent the average of 30 runs.

	Nx Baseline (ms)	Orchestra (ms)	Speedup
DP 10M	3,275.97 ± 30.61	401.03 ± 2.54	**8.17x**
DP 15M	4,706.73 ± 41.14	402.53 ± 1.63	**11.69x**
DP 20M	13,962.63 ± 89.26	405.53 ± 2.10	**34.43x**
MM 128 × 128	173.50 ± 1.11	16.47 ± 0.51	**10.53x**
MM 256 × 256	2,330.73 ± 50.89	17.47 ± 0.51	**133.41x**
MM 512 × 512	37,342.20 ± 314.23	29.13 ± 0.57	**1,281.92x**

The results in Table 1 demonstrate substantial speedups when transitioning from BEAM-managed binaries to Orchestra's multi-threaded CPU execution. For DP, Orchestra achieved speedups from 8.17x to 34.43x. For Matrix Multiplication (MM), speedups scaled from 10.53x (128×128) to 1,281.92x (512×512). The low standard deviations observed across all benchmarks (below 3 ms) also indicate that Orchestra provides a more stable runtime behavior.

These dramatic gains suggest that VM-level memory overheads (inherent to pure Elixir tensor backends) dominate the execution time of these workloads, rather than differences in the underlying algorithms. The default `Nx.BinaryBackend` treats tensors as immutable Erlang binaries, forcing the VM to repeatedly allocate and copy binary structures during iterations, which saturates memory bandwidth and triggers frequent Garbage Collection (GC) cycles. For larger tensors (as in MM 512×512), the GC overhead completely dominates execution time. Orchestra circumvents these limitations by allocating strictly aligned Shared Virtual Memory (SVM) tensors outside the BEAM heap (as detailed in Section 4.2), allowing kernels executing in an `Orchestra.cpu()` context to update them in place while preserving a functional programming interface.

5.2 Case Study: GPU-CPU Cooperation with BFS Benchmark

Unlike the previous experiment, which evaluates Orchestra's CPU execution model, this case study evaluates Orchestra's ability to orchestrate the execution of a single application across different processors. For this, we ported the Breadth-First Search (BFS) graph traversal benchmark from the Chai Collaborative Heterogeneous Applications suite [14]. Graph traversals like BFS represent challenging, irregular, memory-bound workloads that exhibit severe dynamic load imbalances. To optimize device execution, our implementation utilizes a highly optimized Warp-Centric GPU kernel. Unlike a standard thread-centric approach where each thread processes one node, a Warp-Centric kernel assigns an entire warp (32 synchronized threads) to cooperatively process the adjacency list of a single node, ensuring coalesced memory accesses and minimizing

[4] https://github.com/Equiel-1703/orchestra

branch divergence. For the CPU kernel, we implemented a multi-threaded thread-centric kernel utilizing the Test-and-Test-and-Set (TTAS) idiom across all 16 available CPU execution threads (8 physical cores with SMT) to minimize atomic contention in shared memory.

Since BFS explores a graph level-by-level, the algorithm is inherently iterative. In Orchestra, this is implemented natively as a recursive Elixir function that dispatches the appropriate kernel during every step of the traversal. This iterative nature highlights the importance of the Kernel Caching mechanism described in Section 4. Because the kernel signature and data types remain constant across levels, the JIT compilation overhead is paid exclusively on the first iteration, allowing all subsequent kernel launches to execute with near-zero latency.

Furthermore, the BFS benchmark showcases Orchestra's dynamic load-balancing capabilities. In social network graphs, the frontier size begins small, grows exponentially in the middle levels, and shrinks rapidly at the end. Rather than assigning the entire BFS traversal to a single processor, Orchestra allows that both processors participate in the execution of a single traversal. If the frontier size exceeds a defined threshold, the workload is dispatched to the `Orchestra.gpu()` context to leverage massive parallelism. If the frontier falls below the threshold, the workload remains in the `Orchestra.cpu()` context, where the CPU's superior branch prediction and lower latency can process small arrays faster than the GPU's kernel launch overhead. This collaboration pattern is known as *Coarse-grained Task Partitioning* [14]. The threshold is a user-configurable parameter that determines when each BFS iteration will be executed on the CPU or the GPU according to the frontier size.

To evaluate this cooperative approach, we traversed two datasets obtained from the Stanford Network Analysis Project (SNAP) repository [20]: the *Social Circles from Google+* graph (107K nodes, 13.6M edges) and the massive *LiveJournal* social network (4.8M nodes, 68.9M edges). To measure the impact of heterogeneous cooperation, we compared three execution modes: CPU-only, GPU-only, and Cooperative, all using the same source code. The routing was controlled entirely by adjusting the frontier threshold variable. Setting it to 50M forced a CPU-only execution (as the frontier never reaches this value in either graph), setting it to 0 forced a GPU-only execution, and setting an empirically determined optimal threshold of $1,024$ enabled full CPU-GPU cooperation. To ensure statistical significance, the traversal was repeated 30 times for each configuration on the same machine used in the previous experiment. Table 2 reports the average execution time and 95% confidence intervals across all executions. Our measurements intentionally include the first execution, which incurs the kernel cache initialization (*warm-up*) cost. This approach ensures a fair comparison among all three execution modes, as the Cooperative configuration requires caching both the CPU and GPU kernels, whereas the CPU-only and GPU-only configurations cache only a single kernel.

The execution times presented in Table 2 illustrate a massive performance disparity between the CPU and the GPU, as expected for highly parallel graph traversals. However, the success of the Cooperative mode depends heavily on the dataset and the physical hardware interconnection. In Table 3, we present a statistical analysis of the GPU-Only and Cooperative results using the Mann-Whitney U Test and Cohen's d to evaluate the statistical significance of the execution time differences.

Table 2: BFS traversal results. Graph load time from disk is not measured.

Dataset	CPU-Only (ms)	GPU-Only (ms)	Cooperative (ms)
Google+	20.60 ± 0.25	6.17 ± 0.58	**3.67 ± 1.09**
LiveJournal	110.53 ± 0.66	12.47 ± 0.90	**12.17 ± 1.21**

Table 3: Statistical comparison of the GPU-Only and Cooperative execution results using the Mann-Whitney U Test and Cohen's *d*.

Dataset	p-value	Significance	Cohen's d
Google+	1.79×10^{-11}	Significant ($p < 0.05$)	1.07 (Large)
LiveJournal	0.284	Not Significant ($p \geq 0.05$)	0.17 (Small)

As shown in Table 3, for the Google+ dataset, the Cooperative mode achieved a statistically significant speedup over the GPU-only execution (dropping from 6.17ms to 3.67ms with a large effect size of $d = 1.07$). Because this graph contains only 107K nodes, the `visited` state array (a 32-bit integer array used to track whether a node has been visited) is exceptionally small ($\sim$ 430 KB). Transferring 430 KB across the PCIe bus during a context switch takes mere microseconds. Therefore, Orchestra's ability to dynamically route the small initial and final frontiers to the CPU completely bypassed the GPU's kernel launch overhead without incurring a heavy data-transfer penalty.

Conversely, for the LiveJournal dataset, the Cooperative execution time (12.17ms) and the GPU-only execution time (12.47ms) were statistically indistinguishable ($p = 0.284$). Because LiveJournal contains 4.8 million nodes, the `visited` array grows to $\sim$ 19.3 MB. Every time Orchestra switches execution contexts between the CPU and the discrete GPU, it must synchronize this 19.3 MB array across the PCIe bus. The physical latency of transferring this data back and forth completely obfuscates the computation time saved by the CPU on the small frontiers.

Ultimately, these results suggest that the dominant overhead in the LiveJournal workload is the PCIe communication cost and not the orchestration abstraction itself. The lack of a statistically significant speedup in the Cooperative configuration for LiveJournal is therefore attributed primarily to the latency of synchronizing the 19.3 MB `visited` array between host and device memory through the PCIe bus. This phenomenon could theoretically be mitigated by using a smaller datatype for the `visited` array or by using a GPU SVM that the CPU could access directly without having to explicitly transfer data. However, while SVMs have been shown to improve performance on integrated heterogeneous architectures by eliminating explicit copies and enabling direct DRAM sharing between CPUs and GPUs [14], these benefits are substantially reduced on discrete GPU systems, where host and device memories remain physically separated and communication must still traverse the PCIe interconnect [23].

6 Related Work

Orchestra is product of a research trajectory investigating high-performance computing within the Elixir ecosystem. This lineage began with GPotion [9], which introduced the foundational mapping of Elixir syntax to CUDA semantics and established garbage collected device memory. Hok [8] elevated this abstraction by enabling higher-order kernels via function pointers. More recently, PolyHok [10, 24] substituted static typing and pointers with dynamic type inference, AST manipulation, and Just-In-Time (JIT) compilation. While Orchestra inherits PolyHok's dynamic typing, GNx abstraction and JIT architecture, these predecessors were designed to discrete GPU-only execution. Orchestra extends this ecosystem by introducing orchestration macros, enabling cooperative heterogeneous execution across both CPUs and GPUs.

In the broader context of heterogeneous coordination, C++ frameworks and runtime systems have historically dominated. EngineCL [7], StarPU [1], and HCE [28] abstract task scheduling and load balancing across disparate processors, but employ distinct mechanisms to do so. EngineCL conceals hardware complexities behind a single virtual device layer while offers different load balancing algorithms for the user to choose. HCE utilizes software pipelining to optimize and overlap inter-device communications, whereas StarPU dynamically schedules tasks and automates data transfers based on history-based performance models. Merge [21] similarly targets heterogeneous multi-core systems by employing predicate dispatch to dynamically route map-reduce workloads to pre-compiled C++ function variants. To further abstract structural code, algorithmic skeletons have been widely adopted. SkePU [12] provides highly optimized C++ templates for data-parallel skeletons, utilizing smart containers to transparently manage data transfers and memory consistency across devices. Similarly, SPar [15] utilizes C++ attributes to generate stream-parallel pipelines. While highly performant, these frameworks require programmers to manage the complexities of the statically-typed C++ ecosystem and rely heavily on Ahead-Of-Time (AOT) compilation. Orchestra elevates this abstraction to a dynamically-typed, garbage-collected functional language.

Within managed and functional programming paradigms, earlier systems such as Eden [22] explored how functional languages can support parallel programming through high-level abstractions and skeletons. However, these systems were not designed for modern accelerator-based heterogeneous execution. More recent frameworks often rely on intermediate representations (IR) or code translation to target accelerators. Delite [4] leverages Scala and Lightweight Modular Staging (LMS) to construct a domain-specific IR, applying static parallel optimizations before generating heterogeneous execution graphs. SparkCL [25] integrates OpenCL into the Apache Spark big data ecosystem, utilizing Aparapi to execute a Java code base on accelerator devices. Futhark [16] offers highly optimized array computations relying on an Ahead-Of-Time (AOT) compilation pipeline. In contrast, Accelerate [6] and dynamic environments like Julia [2] embrace dynamic execution. Accelerate embeds array computations in Haskell using an online code generator at application runtime, while Julia serves as a Just-In-Time (JIT) compiler for GPUs, allowing developers to write native-like kernels that are dynamically specialized. Modern hyper-heterogeneous AI frameworks like H2 [27] tackle hardware diversity differently, employing a unified PyTorch-compatible interface (DiTorch) that bridges various vendor-specific operator libraries rather than JIT-compiling custom kernels. Orchestra aligns with the dynamic JIT philosophy and Garbage Collection (GC) integration seen in environments like Julia, but distinguishes itself by avoiding complex IR graphs entirely. Instead, it dynamically extracts and compiles Elixir ASTs directly to OpenCL C, caching them to avoid overhead, and focuses on explicit cooperative orchestration via its context-based `with` macro.

Finally, the use of Abstract Syntax Tree (AST) manipulation to hide boilerplate code is a growing trend across modern languages. Recent work by Faé and Griebler [13] demonstrates how Rust procedural macros can automatically generate OpenCL pipeline code from annotated Rust functions. While Faé and Griebler utilize Rust's procedural macros at compile time to directly generate OpenCL code from strictly constrained Rust functions, Orchestra leverages Elixir macros to generate and JIT compile kernels at runtime, to accommodate dynamic typing. Both approaches confirm that AST metaprogramming is an effective technique for abstracting the severe complexities of heterogeneous APIs.

7 Conclusions and Future Work

This paper presented Orchestra, a set of abstractions embedded in Elixir designed to simplify the development of heterogeneous applications. By decoupling the logical definition of kernels from their physical execution targets, Orchestra eliminates much of the boilerplate code traditionally required to coordinate workloads across CPUs and GPUs while preserving a unified programming model. Furthermore, Orchestra enables the construction of reusable algorithmic abstractions, such as skeletons, through device-independent kernels.

The experimental evaluation demonstrated that Orchestra provides substantial runtime performance improvements over the default Numerical Elixir (Nx) `BinaryBackend` for dense linear algebra workloads executing on multi-core CPUs. The BFS case study further showed that Orchestra's orchestration abstractions can support coarse-grained heterogeneous cooperation by dynamically selecting the CPU or GPU according to the workload characteristics. For workloads with low communication overhead, this strategy improved execution time compared to GPU-only execution, whereas for larger datasets the benefits were limited by PCIe communication costs, and not by the abstraction layer itself.

Future work will focus on extending Orchestra with abstractions for multi-GPU execution and asynchronous execution streams. We also plan to evaluate Orchestra using a broader collection of heterogeneous applications and orchestration patterns, including additional cooperative benchmarks.

Artifact Availability

The source code for Orchestra and all experiments presented in this paper are publicly available on Zenodo (DOI 10.5281/zenodo.21799503) under the MIT License.

Acknowledgments

This study was financed in part by CAPES - Finance Code 001, and by FAPERGS - 24/2551-0001372-5.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* (Feb. 2011), 187–198. doi:10.1002/cpe.1631

[2] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2019. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (April 2019). doi:10.1109/TPDS.2018.2872064

[3] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. 2016. Have abstraction and eat performance, too: optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 194–205. doi:10.1145/2854038.2854042

[4] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society. doi:10.1109/PACT.2011.15

[5] Nicola Capodieci, Roberto Cavicchioli, and Andrea Marongiu. 2022. A Taxonomy of Modern GPGPU Programming Methods: On the Benefits of a Unified Specification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2022), 1649–1662. doi:10.1109/TCAD.2021.3082863

[6] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. Association for Computing Machinery. doi:10.1145/1926354.1926358

[7] María Angélica Dávila Guzmán, Raúl Nozal, Rubén Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, and Jose Luis Bosque. 2019. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *J. Supercomput.* (March 2019). doi:10.1007/s11227-019-02768-y

[8] André Du Bois, Tiago Perlin, Frederico Antunes, and Gerson Cavalheiro. 2024. Hok: Higher-Order GPU kernels in Elixir. In *Anais SBLP 2024*. SBC, Porto Alegre, RS, Brasil, 71–80. doi:10.5753/sblp.2024.3690

[9] Andre Rauber Du Bois and Gerson Cavalheiro. 2023. GPotion: An embedded DSL for GPU programming in Elixir. In *Proc. SBLP 2023* (Campo Grande, MS, Brazil). ACM, New York, USA, 1–8. doi:10.1145/3624309.3624314

[10] André Rauber Du Bois and Gerson Cavalheiro. 2025. Polymorphic Higher-Order GPU Kernels. In *Euro-Par 2025: 31st European Conference on Parallel and Distributed Processing, Proceedings, Part I* (Dresden, Germany). Springer-Verlag, Berlin, Heidelberg, 100–113. doi:10.1007/978-3-031-99854-6_7

[11] Elixir Team. 2026. *Elixir Lang. Doc.* The Elixir Team. https://elixir-lang.org/ Disponível em: https://elixir-lang.org/docs.html.

[12] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. 2021. SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters. *International Journal of Parallel Programming* (Dec. 2021). doi:10.1007/s10766-021-00704-3

[13] Leonardo Faé and Dalvan Griebler. 2025. Towards GPU Parallelism Abstractions in Rust: A Case Study with Linear Pipelines. In *Anais do XXIX Simpósio Brasileiro de Linguagens de Programação*. SBC. doi:10.5753/sblp.2025.13152

[14] Juan Gómez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. 2017. Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE.

[15] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters* (2017). doi:10.1142/S0129626417400059

[16] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *SIGPLAN Not.* (June 2017). doi:10.1145/3140587.3062354

[17] Intel. 2023. Heterogeneous vs. Homogeneous Computing Environments. https://www.intel.com/content/www/us/en/docs/sycl/introduction/latest/01-homogeneous-vs-heterogeneous.html. [Accessed 29-05-2026].

[18] Laxmikant V. Kalé and David M. Kunzman. 2011. Programming Heterogeneous Systems . In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE Computer Society, Los Alamitos, CA, USA, 2061–2064. doi:10.1109/IPDPS.2011.377

[19] Khronos OpenCL Working Group. 2026. The OpenCL™ Specification. https://registry.khronos.org/OpenCL/specs/unified/html/OpenCL_API.html. Version v3.1.1, [Accessed 30-05-2026].

[20] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[21] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. 2008. Merge: a programming model for heterogeneous multi-core systems. *SIGOPS Oper. Syst. Rev.* (March 2008), 287–296. doi:10.1145/1353535.1346318

[22] Rita Loogen, Ortega-Mallén Yolanda, and Ricardo Peña. 2005. Parallel functional programming in Eden. *J. Funct. Program.* 15 (05 2005), 431–475. doi:10.1017/S0956796805005526

[23] Nurlan Nazaraliyev, Elaheh Sadredini, and Nael Abu-Ghazaleh. 2025. DREAM: Device-Driven Efficient Access to Virtual Memory. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 1190–1205. doi:10.1145/3721145.3725748

[24] Henrique Gabriel Rodrigues, André Rauber Du Bois, and Gerson Geraldo H. Cavalheiro. 2026. Kernels de Ordem Superior para GPUs Compatíveis com OpenCL Utilizando Metaprogramação em Elixir. In *Anais do Seminário Integrado de Software e Hardware (SEMISH)*.

[25] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. 2015. *SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters*. arXiv:1505.01120 [cs.DC] https://arxiv.org/abs/1505.01120

[26] Amar Shan. 2006. Heterogeneous Processing: a Strategy for Augmenting Moore's Law. https://www.linuxjournal.com/article/8368. [Accessed 29-05-2026].

[27] Ding Tang, Jiecheng Zhou, Jiakai Hu, Shengwei Li, Huihuang Zheng, Zhilin Pei, Hui Wang, and Xingcheng Zhang. 2025. *H2:Towards Efficient Large-Scale LLM Training on Hyper-Heterogeneous Cluster over 1,000 Chips*. arXiv:2505.17548 [cs.DC] https://arxiv.org/abs/2505.17548

[28] Lanjun Wan, Weihua Zheng, and Xinpan Yuan. 2021. HCE: A Runtime System for Efficiently Supporting Heterogeneous Cooperative Execution. *IEEE Access* (2021). doi:10.1109/ACCESS.2021.3124856