Hok: Higher-Order GPU kernels in Elixir

André Rauber Du Bois
PPGC - UFPel - Pelotas, RS - Brazil
dubois@inf.ufpel.edu.br

Tiago Perlin
PPGC - UFPel - Pelotas, RS - Brazil
tiago.perlin@inf.ufpel.edu.br

Frederico Peixoto Antunes
PPGC - UFPel - Pelotas, RS - Brazil
fpantunes@inf.ufpel.edu.br

Gerson Cavalheiro
PPGC - UFPel - Pelotas, RS - Brazil
gerson.cavalheiro@inf.ufpel.edu.br

Abstract

GPUs (Graphics Processing Units) are usually programmed using low-level languages like CUDA or OpenCL. Although these languages allow the implementation of very optimized software, they are difficult to program due to their low-level nature, where programmers have to mix coordination code, i.e., how tasks are created and distributed, with the actual computation code. In this paper we present Hok, an extension to the Elixir functional language that allows the implementation of higher-order GPU kernels, granting programmers the ability to clearly separate coordination from computation. The Hok system provides a DSL (Domain-Specific Language) for writing low-level GPU kernels that can be parameterized with the computation code. Hok allows device functions, including anonymous functions, to be created and referenced in the host code so that they can configure a kernel before it is launched. We demonstrate that Hok can be used to implement high-level abstractions such as algorithmic skeletons and array comprehensions. We also present experiments that demonstrate the usability of the current implementation of Hok, and show that high speedups can be obtained in comparison to pure Elixir, specially in computationally intensive programs with large inputs.

CCS Concepts: • **Computing methodologies → Parallel programming languages**.

Keywords: parallel programming, gpu, Elixir

1 Introduction

GPUs are mainly programmed in low-level languages like CUDA and OpenCL. Programming in such languages is difficult due to their low-level nature and lack of abstraction. When writing GPU programs in these languages, programmers have to mix *coordination* code, i.e., code relative to creation and coordination of tasks, together with *computation*, code, i.e., the code that does the real computation.

In simple programs, coordination is usually boilerplate code, but in more complex systems, the efficient coordination of tasks can become a complex endeavor, and should, as much as possible, be left to specialized libraries designed by expert programmers.

This paper presents Hok, an extension to the functional programming language Elixir that allows the implementation of higher-order GPU kernels. Hok allows device functions, including anonymous device functions, to be referenced in host code, and be passed as arguments to kernels at launch time. This allows the programmer to separate coordination code, that is encapsulated in kernels and Elixir functions, from computation code, that can be left in device functions.

The contributions of this paper are as follows:

- We present Hok, an Elixir extension for GPU programming. Hok extends the GPotion DSL [11] with Higher-Order Kernels, which are GPU kernels that can take functions as arguments, hence allowing programmers to separate coordination code from computation code. Furthermore, Hok allows the programmer to define and reference at host code, device functions that can be used to configure kernels before they are executed. Device functions and anonymous functions are values in the language, so that they can be passed as arguments to kernels or to other device functions. We discuss the current implementation of Hok, which is implemented using the meta-programming features of Elixir, with no modifications to the Elixir compiler.
- A clean separation between coordination and computation allows the development of high-level abstractions for GPU programming. We demonstrate the usability of Hok through a series of examples, including high-level abstractions like algorithmic skeletons and array comprehensions
- We present an experiment that demonstrates the usability of the architecture proposed for the implementation of Hok. In the experiments, we compare five programs implemented in Hok with programs that use the same abstractions but are written in pure Elixir. The experiments demonstrate that higher speed-ups can be obtained in larger instances of problems that are more computationally intensive.

The source code for Hok and the benchmarks used in the experiments is available as free software in a GitHub repository. [1]

This paper is organized as follows: we start by covering some background on Elixir, Actors model and GPU programming (Section 2). Next, we introduce Hok, the main contribution of this paper, by first describing how basic GPU kernels

[1] https://github.com/ardubois/hok

are implemented in the GPotion DSL (Section 3.1), and then how higher-order kernels can be expressed (Section 3.2). The following Section demonstrates how Hok can be used to implement high-level abstractions like algorithmic skeletons and array comprehensions. Section 4 describes how the current prototype of Hok was implemented. In Section 5, we present experiments comparing the same abstractions both in Hok and pure Elixir. Finally, related work and conclusions are discussed.

2 Background

2.1 Elixir and Actors

The Actors model, was presented by Hewitt in his work [20] as framework for handling concurrency, where computations are structured around Actors. These Actors, are autonomous entities that engage in asynchronous message passing to communicate. Presently, the Actors model is adopted across various programming languages and frameworks. Notable among these implementations are the Akka framework[2] [37] for Java and Scala, as well as Microsoft Orleans [31] for .NET. Additionally, languages like Erlang [3] and Elixir[3] inherently support the Actors model. Recently, the Actors model has found application in federated machine learning [38, 41], underscoring its suitability for leveraging GPU programming within this domain.

Elixir, is a dynamic functional programming language designed for software development using the Actors model. It executes on the Erlang virtual machine (BEAM) [25], inheriting its features such as scalability, distribution, and fault tolerance, while offering a more modern language. Due to its functional nature, Elixir ensures concurrency without relying on shared state, with actors exclusively interacting through asynchronous message passing.

2.2 GPU programming

Originally, GPU programming predominantly involved writing low-level C code using either CUDA, supported by NVIDIA, or OpenCL, an open standard for cross-platform parallel programming. These languages remain the primary options for general-purpose parallel programming on GPUs [23].

Various systems aim to simplify GPU programming in C by offering higher-level abstractions through pragmas, such as OpenMP, OpenACC, hiCUDA [17], or other higher-level annotations like SPar [34].

Exploring GPU programming beyond C can be approached through different languages, including libraries like CuPy [32] and JCuda [42], just-in-time (JIT) compilation techniques [24, 26, 29], or embedded domain-specific languages (DSLs) such as Accelerate [7], Nikola [27], and Obsidian [40]. However, supporting GPU programming in languages other than C often necessitates significant modifications to the compiler and its internal representation [4]. Some endeavors involve defining entirely new languages with high-level abstractions tailored for GPU programming, as seen in works like Lime [12], Chesnut [39], HIPA [30], and GPUActors [18].

In many GPU programming paradigms, like CUDA and OpenCL, a function executable on the GPU is termed a *kernel*. Typically, upon launching a kernel, it is executed by numerous threads. In CUDA and similar GPU libraries, threads are hierarchically organized into three-dimensional blocks, and a kernel is always launched within a grid, which is an array of blocks. By combining its thread and block IDs, a thread can compute a unique identifier used to associate it with data elements or tasks. During kernel execution, blocks are mapped to the GPU's stream multiprocessors, where threads run concurrently. Since GPUs have separate memory from the CPU, the typical program flow involves initially loading data onto the CPU, then transferring it to GPU memory, followed by kernel/grid launch, and finally, moving the results back from GPU memory to the CPU.

3 Hok

The main contribution of this paper is Hok, an Elixir extension that allows the implementation of higher-order GPU kernels. In Hok, GPU kernels are written using the GPotion DSL, and we review it in Section 3.1. Hok extends GPotion with higher-order kernels, that are described in Section 3.2.

3.1 GPotion kernels

GPotion [11] extends Elixir with two abstractions: the GMatrex and GPotion kernels. A GMatrex is an array that resides in the GPU memory and can only be manipulated by GPotion kernels. The GPotion language provides compatibility with the Matrex library [4], which is an Elixir library for fast operations on numerical arrays. Matrex provides fast saving and loading of arrays, and compatibility with NumPy arrays. The Matrex library offers many common matrix operations implemented in native C code and also offers a back-end that uses the CBLAS library.

A new GMatrex can be created from an existing Matrex, or by providing a number of rows and columns:

```

gmat1 = Hok.new_gmatrex(Matrex.random(1,10000))
gmat2 = Hok.new_gmatrex(1,10000)

```

A GMatrex is just a reference to a Matrex that resides in the GPU memory and is of no use in the regular Elixir world. A GMatrex can only be operated inside kernels that are implemented using the GPotion DSL. A GMatrex can be transformed back into a Matrex using `get_gmatrex`:

```
new_matrex = Hok.get_gmatrex(gmat1)
```

[2]https://akka.io/
[3]https://elixir-lang.org/
[4]https://hexdocs.pm/matrex/Matrex.html

```
1  require Hok
2
3  Hok.defmodule Saxpy do
4
5  defk saxpy_kernel(a,b,c,n) do
6     index = blockIdx.x*blockDim.x+threadIdx.x
7     stride = blockDim.x * gridDim.x
8
9    for i in range(index,n,stride) do
10       c[i] = 2 * a[i] + b[i]
11    end
12  end
13 end
```

Figure 1. A GPU kernel that computes Saxpy

The `get_gmatrex` function takes a GMatrex as an argument and transfers it into the host memory returning a Matrex that can be manipulated in the Elixir world using the regular Matrex operations. A GMatrex created in the host does not need to be freed as it is automatically garbage collected by the Elixir/Erlang virtual machine.

GPU kernels are written using the GPotion DSL, as can be seen in the *saxpy* example of Figure 1. Hok kernels are written inside Hok modules, which are just like regular Elixir modules extended with the ability to also implement kernels and GPU functions. As can be seen in the example, a kernel is defined using the `defk` keyword. The example also illustrates the main differences between the GPotion DSL and regular Elixir code: GPotion exports CUDA runtime constants, as for example those related to blocks and grids (lines 6 and 7), GPotion allows update in-place of memory locations (line 10), and, as Elixir does not provide loops, GPotion extends Elixir with python-like fors (lines 9 to 11) and whiles. The reader should notice that in-place update is restricted to GPU kernels and does not affect regular Elixir code. The purely functional part of Elixir can not offload a GMatrex while it is being mutated by kernels, as a call to `get_gmatrex` blocks while the respective GMatrex is used on the GPU. A GPotion kernel can be seen as a unique isolated process that does not communicate. Although GPotion kernels can not send and receive messages, they can easily be executed inside other processes that communicate and execute GPotion kernels accordingly.

The example in Figure 2, illustrates how a GPotion kernel can be launched. First, the Hok module where the kernel was defined is imported (line 1). Next, in lines 4 and 5, two new CPU arrays containing random numbers are created. Lines 7 and 9 show how GPU arrays can be generated, either from an exiting Matrex (lines 7 and 8) or by providing the number of rows and columns of the new empty array to be created (line 9). A kernel is launched using the `Hok.spawn` primitive (line 14), which takes as arguments two tuples configuring grid and blocks, and a list with the kernel arguments. Finally,

```
1  Hok.include [Saxpy]
2
3  n = 10000000
4  mat1 = Matrex.random(1,n)
5  mat2 = Matrex.random(1,n)
6
7  gm1  = Hok.new_gmatrex(mat1)
8  gm2  = Hok.new_gmatrex(mat2)
9  gmr  = Hok.new_gmatrex(1,n)
10
11 threads = 128;
12 n_blocks = div(n + threads_pb - 1, threads_pb)
13
14 Hok.spawn(&Saxpy.saxpy_kernel/4,{n_blocks,1,1},
15                                 {threads,1,1},
16                                 [gm1,gm2,gmr,n])
17 result = Hok.get_gmatrex(gmr)
```

Figure 2. Spawning a kernel

the result of the computation can be brought back to the CPU memory using `get_gmatrex` (line 17).

3.2 Hok: Higher-Order kernels

The main novelty of Hok is to allow kernels and device functions to be higher-order, meaning that device functions, including kernels, can take functions as their arguments. Furthermore, device functions, including device anonymous functions, can be created and referenced in host code so that they can be passed to kernels when they are launched.

Higher-order kernels allow the separation between *coordination* and *computation* code, as can be seen in the example of Figure 3. The `apply_k` kernel takes as arguments two arrays, the input array and the result array of the same size, the size of the arrays, and a function, which is applied to each element of the input array generating the result array. To simplify the code presented in this text, we always assume that one grid can hold all the elements of the arrays used in the examples. The `apply_k` kernel encapsulates the coordinating aspects of the general idea of applying a transformation into the elements of an array which in turn computes a new array.

Computation code can be encapsulated in Hok functions, which are device functions that are implemented using the `defh` keyword, as the function `square` in Figure 3. Hok functions can be written in pure Elixir or also using the GPotion extensions. As Hok functions are compiled into CUDA (see Section 4), they can not call other Elixir functions, but can call other Hok functions and also C CUDA functions.

The `apply_k` kernel could be spawned with the following command:

```
Hok.spawn(&Ex1.apply_k/4, {n_blocks,1,1},
        {threads_pb,1,1},
        [gm1,gmr,n, &Ex1.square/1])
```

```
require Hok
Hok.defmodule Ex1 do
  defk apply_k(a,r,size,f) do
   id = blockIdx.x * blockDim.x + threadIdx.x
   if(id < size) do
      r[id] = f(a[id])
    end
  end
  defh square(x) do
     x*x
  end
end
```

Figure 3. Defining Hok kernels and functions

The kernel receives as arguments an input array gm1, the result array gmr, the size n of the arrays, and the function square (from Figure 3). Hence, this kernel launch will square all the elements of the input array, generating a new array.

Hok kernels also accept *anonymous* functions. For example, the same previous program could be written by substituting the square function by an anonymous function with the same semantics:

```
Hok.spawn(&Ex1.apply_k/4, {n_blocks,1,1},
          {threads_pb,1,1},
          [vet1_gpu,resp_gpu, n,
          Hok.hok fn x -> x*x end]
```

Device anonymous functions can be created with the Hok.hok primitive. As with device functions, Hok anonymous functions can be written using pure Elixir or GPotion. As Elixir values are not valid in the GPU, we do not support variable capture. Also, no Elixir functions can be called on Hok anonymous functions, only Hok or CUDA functions can be invoked.

In the next Section, we discuss how higher-order kernels can be used to implement high-level abstractions for GPU computing such as composable algorithmic skeletons and array comprehensions.

3.3 High-level abstractions

3.3.1 Composable Algorithmic Skeletons. *Algorithm skeletons* [7, 10, 13, 14, 16, 28] are a high-level programming paradigm for parallel and distributed computing. Skeletons are functions that encapsulate common patterns of parallel and distributed computing. Programmers use skeletons by configuring these functions with aspects specific to the program being constructed. Skeletons hide the complex and platform specific details of task coordination, giving to programmers a simple interface where they have to focus only in the computation details of the algorithm.

Languages that provide higher-order functions are the perfect playground to experiment with algorithmic skeletons as the skeleton functions can be configured dynamically with computations to be executed on parallel and distributed hardware. The apply_k kernel from the previous Section could be used to implement a skeleton commonly known as map, which applies a function to all elements of an array generating a new array. In Figure 4, we present the implementation of a binary map, a variant of the map skeleton (taken from [13, 16]), which takes two input arrays and a function with two arguments, applying the function to the elements of the arrays in pairs. The skeleton is implemented using a kernel (map2_kernel), which is similar to apply_k, and an Elixir function map2, which is responsable for launching the kernel.

```
Hok.defmodule Ske do
  defk map2_kernel(a1,a2,a3,size,f) do
    id =blockIdx.x * blockDim.x + threadIdx.x
    if(id < size) do
      a3[id] = f(a1[id],a2[id])
    end
  end
  def map2(t1,t2,func) do
   {l,size} = Hok.gmatrex_size(t1)
   result_gpu =Hok.new_gmatrex(l,size)
   t_block = 128;
   n_blocks = div(size + t_block - 1, t_block)
   Hok.spawn(&DP.map2_kernel/5,
               {n_blocks,1,1},{t_block,1,1},
               [t1,t2,result_gpu,size,func])
      result_gpu
  end
end
```

Figure 4. The map2 skeleton

It is important that skeletons receive and return GMatrexes so that skeletons can be *composed* together to generate more complex computations on the GPU. For example, *reduce* is another common pattern of computation in functional languages where the reduce function (also called foldr in some languages) takes a list, an accumulator and a binary function, and combines the elements of the list using the binary function. We have implemented a reduce kernel in Hok that executes on the GPU, and is launched by the skeleton, implemented as an Elixir function, defined in Figure 5. The reduce skeleton first creates a new GMatrex with only one position containing the accumulator, then it calculates the number of blocks and threads to be used, and finally it launches the reduce_kernel kernel, passing as parameters the input array, the accumulator, the size of the input, and the binary operation.

By making the map and reduce skeletons receive and return GMatrexes, it is possible to use Elixir's pipe operator (|>) to compose operations executed on the GPU. For example, the dot product of two arrays can be implemented by composing the map2 and reduce skeletons defined previously:

```
def reduce(input, acc, f) do
  {_l,size} = Hok.gmatrex_size(input)

  result =Hok.new_gmatrex(Matrex.new([[acc]]))
  thPerBlock = 128
  bPerGrid = div(size + thPerBlock - 1,
                          thPerBlock)
  Hok.spawn(&DP.reduce_kernel/4,{bPerGrid,1,1},
           {thPerBlock,1,1},
           [input, result, size,f])
  result
  end
```

Figure 5. The reduce skeleton

```
def dot_product(arr1,arr2) do
 arr1
  |> Ske.map2(arr2, Hok.hok fn (a,b)->a * b end)
  |> Ske.reduce(Hok.hok fn (a,b)->a + b end)
end
```

Map and reduce skeletons appear in the literature with a series of variations [7, 10, 13, 14, 16, 28], i.e., type of argument function they take (unary, binary, or ternary), the shape of the input (vectors or matrices) and how many input arrays are available. We have implemented a library with different map and reduce skeletons and have used it to implement *array comprehensions* (see Section3.3.2) and a collection of benchmarks presented in Section 5. In applications that store structured data in arrays, each data element occupies more then one contiguous memory space. As Hok only support numerical arrays, we also provide a version of map that *slices* the array, so that the function argument is mapped not to single elements but to slices of the array.

3.3.2 Array Comprehensions.

List comprehensions is a programming abstraction, usually available in functional programming languages, where programmers can describe new lists, based on exiting lists, using a notation similar to the one used in set theory.

For example, in Elixir one can describe a new list `ln` containing the squared elements of an existing list `le` using a comprehension of the form:

```
le = [1,2,3,4,5]

ln = for x <- le, do: x*x
```

A list comprehension usually contains at least one *generator* and a computation to be executed for each generated element.

We can simulate a comprehension, similar to the Elixir example presented before, but that runs on the GPU, by using the `comp` function from Figure 6, which int turn uses a `map` skeleton implemented using the `apply_k` kernel, to execute the body of the comprehension on each element of the input array.

Hence, the previous comprehension could be written as:

```
def comp(array,func) do
    {l,size} = Matrex.size(array)

    array_gpu = Hok.new_gmatrex(array)
    result_gpu =Hok.new_gmatrex(l,size)

    Ske.map(array_gpu, result_gpu, size,func)

    r_gpu = Hok.get_gmatrex(result_gpu)
    r_gpu
  end
```

Figure 6. Executing a Comprehension

```
defmacro gpufor({:<-,_,[var,tensor]},do: b) do
 quote do:
   Comp.comp(unquote(tensor),
   Hok.hok (fn (unquote(var))->(unquote b) end))
 end
```

Figure 7. Macro for an array comprehension with a single generator

```
le = Matrex.new(...)

ln = comp(le, Hok.hok fn x -> do x*x end)
```

Furthermore, it is possible to give a more high-level syntax to the GPU comprehensions by using the meta-programming features of Elixir. The macro on Figure 7, translates a list comprehension with a single generator, into a call to the `comp` function. Using this macro, the previous GPU comprehension can now be expressed as:

```
le = Matrex.new(...)

ln = Hok.gpufor x<- le,  do: x * x
```

We have implemented a library for GPU array comprehensions over the Matrex library supporting different kinds of generators (up to two), and have used it to implement some of the programs of Section 5. For example, we can sum two arrays on the GPU using a generator based on intervals:

```
ln = Hok.gpufor i <- 0..size, a1, a2 do
  a1[i] + a2[i]
end
```

4 Implementation

4.1 GMatrex

An Elixir Matrex is internally represented as a contiguous memory space, where its first two positions contain the number of rows and columns. Hence, when a GMatex is created, we allocated memory on the GPU with `cudaMalloc`, using the number of rows and collums to compute its size, and, if the new GMatrex is based on an existing Matrex, copy its data using `cudaMemcpy`. Similarly, the `get_gmatrex` function first allocates memory in the host for the new Matrex, and then copies the existing GMatrex, using `CUDAMemcpy`, from the device memory back into the host memory.

A GMatrex is represented internally as a new resource in the Erlang/Elixir VM, and once a resource is not referenced anymore, a destructor is automatically called for it. For a GMatrex, the descructur calls `cudaFree` to liberate the memory allocated for it.

4.2 Hok Modules

The `Hok.defmodule` primitive is implemented as a *macro* in Elixir. Macros are special functions used for metaprogramming, which are executed only at compile time. Elixir's compiler first generates the Abstract Syntax Tree (AST) of a program and then calls all defined macros which process the AST possibly generating a new one. The `Hok.defmodule` macro receives the AST of the Hok module being defined and sends it to the Hok compiler. The first step in the Hok compiler is performing type inference (see Section 4.3) for untyped programs or type checking for typed ones. The compiler then sends the module's AST and type information to the CUDA back-end, which in turn compiles the module into a CUDA C shared library. Kernels and device functions are directly translated by traversing the AST and emitting equivalent CUDA constructs.

For each kernel, the compiler also generates an *access function*, which is responsible for converting values from the Elixir world to the C world, and also making the real kernel launch with the proper arguments. For each device function in the module, the compiler also generates a global pointer. Higher-order kernels and higher-order device functions are compiled into CUDA C functions that take function pointers as arguments. When a higher-order kernel is executed, we copy the function pointers of its function arguments from the device memory to the host memory, and launch the kernel with these pointers as arguments. Hence, at runtime, a kernel receives a valid pointer to the device's memory where the function resides. This pointer can be directly used as a function or forwarded to, and used by, other device functions.

The Hok compiler also substitutes a Hok module by a regular Elixir module that contains all the regular Elixir functions defined by it plus Elixir versions of its kernels and device functions. These versions are just regular Elixir functions that represent kernels and device functions in the Elixir world, so that they can be used as first class values. These functions will generate an exception if called outside of spawn.

All interaction between C and Elixir is implemented using the NIFs library [1].

4.3 Type inference

For kernels and device functions, type inference can most of the time detect types correctly by using control flow analysis. As Matrex numerical arrays are internally represented as arrays of floats, type inference is greatly simplified. Type inference starts from source expressions to which types are known, e.g, array index, array update, CUDA constants, literals etc., and then the type inference mechanism follows the control flow, departing from these expressions and propagating their types to their parent and subexpressions. Type inference executes in passes, propagating type information, and it stops when the types of all variables are known, or when a pass does not change the mapping of variables to types.

For example, the type inference mechanism can not detect the type of `x` in the following anonymous function:

```

f = Hok.hok fn x -> x end

```

In such a case, the compiler stops and reports the error. Programmers can always insert type annotations in anonymous functions as in the following example:

```
f = Hok.hok fn x -> type x integer
                    x
                    end
```

For unknown numerical types, type inference always assumes the same type of the array content (float). When a kernel is launched, it is typed checked at runtime against its provided arguments to check that type inference has correctly detected types. If the types are not correct, an exception is raised so the programmer can insert type information in the code.

We also provide a typed version of Hok, in which types are declared in a similar way to the typed Elixir proposal [5].

4.4 Anonymous Functions

For each program, the Hok compiler generates a global CUDA C module. This module contains the other Hok modules imported into the program (using `Hok.include`) plus all the anonymous functions defined in the program. The `Hok.hok` primitive is also implemented as a macro. This macro compiles the anonymous function into a device function and its pointer and ads it to the global module. The compiler leaves in the place of the anonymous function a runtime representation of it that contains its name and type. The name is used to access its global pointer, and the type to type check it against the kernel type at kernel launch.

5 Experiments

The objective of this Section is to give some insight on the performance of the architecture proposed to support higher-order kernels in Elixir. We have implemented 5 programs in Hok using skeletons and comprehensions: array sum (SM) from Section 3.3.2, matrix multiplication (MM), dot product (DP) from Section 3.3.1, Julia (JL), that generates images based on the Julia set (ported from [36]), and Nearest Neighbor (NN) taken from the Rodinia Benchmark Suite [9].

As a base for comparison, we have also implemented pure Elixir versions of these programs using map, reduce and list comprehensions. We used lists in the Elixir versions of the programs for two reasons: first, we wanted to use similar abstractions to the ones used in the Hok programs, and second, due to immutability of data, it is usually faster to construct a result list then to individually update a result array, as each update to a position of the array results in a new array. The times presented for the execution of each instance of an application are the average of 30 executions. The times for Hok include the time to transfer data to/from the device. The experiments were executed on a machine with a 12th Gen Intel(R) Core(TM) i7-12700K CPU @ 3.60GHz with 16 GB of RAM, NVIDIA GeForce RTX 3070, Ubuntu 22.04.3 LTS, Elixir 1.16.1, Erlang OTP 26, and nvcc 12.3.107. Table 1 summarizes the results of the experiments.

Table 1. Results of the experiments

	Elixir (ms)	Hok (ms)	Speedup
SA 50M	776.8	313.9	2.5
SA 100M	1,998.4	380.2	5.3
SA 200M	13,922.4	531.3	26.2
MM 7K x 7K	100,333.0	854.3	117.4
MM 9K x 9K	215,192.1	1,471.5	146.2
MM 11K x 11K	397,310.6	2,464.5	161.2
DP 50M	2,322.1	315.9	7.4
DP 100M	7,073.2	386.9	18.3
DP 200M	13,211.2	527.1	25.1
JL 2048x2048	65,589.6	241.8	271.3
JL 4096x4096	288,827.7	376.1	767.9
JL 8192x8192	1,235,025.1	883.7	1,397.6
NN 50M	4,261.5	315.6	13.5
NN 100M	9,877.5	526.7	18.8
NN 200M	58,038.2	773.9	75.0

The sum arrays (SA) application is a micro-benchmark that uses the array comprehension presented in Section 3.3.2 to sum two arrays. The Elixir version of the program uses a list comprehension to sum two lists. Constructing a list by summing the elements of other two lists is computed fast in Elixir, and Hok achieved the lowest speedup in the 50M instance. With smaller instances, the time to transfer data to/from the device impacts the performance, but, as the size of the input increases, also the difference between Hok and Elixir increases, and Hok achieves a speedup of 26.2 with the 200M instance.

The matrix multiplication (MM) in Hok was implemented as an array comprehension using two generators, and it is translated into a 2D map. For the sequential version, we are using an efficient implementation of matrix multiplication that comes with the standard Matrex installation, and is implemented in pure C. This implementation avoids the problem of immutability by allocating a single resulting matrix that is updated in the C world, before being passed back to Elixir. Matrix Multiplication is computationally intensive as for each position of the result matrix a dot product must be computed. In Hok, each position of the resulting matrix is computed by a different thread resulting in speedups from 117.4 to 161.2.

The dot product (DP) application is implemented in both Hok and Elixir as a composition of map and reduce, as can be seen in the example of Section 3.3.1. DP presented similar results to SA with a low speedup of 7.4 in the smaller instance, increasing to 25.1 in the largest instance of the problem.

The Julia (JL) benchmark was implemented as a map where the configuring function takes as input a coordinate x and y and computes a pixel of the resulting image. Julia is more computationally expensive on the Elixir side, as it uses an iterative equation to decide if a coordinate is in the Julia set or not. This explains the slower sequential times and the high speedup obtained.

Nearest Neighbor (NN), ported from the Rodinia benchmark [9], is implemented as a composition of map and reduce. The map, calculates the Euclidean distance from each record, in a randomly generated data set, to a target position. The reduce stage, computes the shortest distance from all the computed Euclidean distances. The smallest speedup obtained was 13.5 with 50M, and it increases to 75 with the 200M instance.

In general, for small instances of problems, where the Elixir version runs at most in a few seconds, Hok obtained the smallest speedups. For more computationally expensive programs, the speedups were larger, and it increases when the instance of problems also increases.

6 Related Work

In CUDA, one can implement higher-order kernels using pointers to functions. A kernel can receive pointers to functions but the programmer can not directly get the address of a device function in host code. The workaround is to define a global device pointer to the function and then copy the content of this pointer to the host memory, before launching the kernel. This is the approach taken by the Hok compiler under the hood. For each device function and lambda abstraction, the Hok compiler generates a global pointer, and at runtime, the content of this pointer is automatically copied to the host when a device function or anonymous function is referenced for the first time. The approach taken by Hok is more high-level, as it hides the complexities of accessing device functions, while CUDA makes them explicit.

There are many works on extending the Python programming language for GPU programming, e.,g, [6, 26, 35, 42]. Closest to our approach are Parakeeet [35] and Copperhead [6], which are high-level data-parallel languages embedded in Python. In both DSLs, programmers write array computations that are made parallel by the use of skeletons like map and reduce. Both languages allow users to define functions that are automatically compiled for the GPU, but the user is restricted to generate parallelism with the built-in skeletons.

Significant research has been conducted on embedding domain-specific languages (DSLs) targeting GPUs within the Haskell functional programming language, as seen in works like [7, 27, 28, 40]. Nikola [27], for instance, is a DSL designed for array processing in Haskell. It leverages type class overloading to construct representations of computations described by the DSL, which are then processed and compiled into CUDA C. Similarly, Accelerate [7, 28, 29] is another embedded DSL in Haskell, aimed at array processing on GPUs. Accelerate allows programmers to use higher-order functions for array manipulation within a high-level language framework. The array computations expressed in Accelerate are compiled into skeleton calls implemented in CUDA. Subsequently, new back-ends for both CPU and GPU were developed for Accelerate by compiling it into the LLVM architecture [29]. Besides Haskell, many other functional language provide high-level skeletons for GPU programming, as for example Futhark [19]. Although these languages provide composable abstractions that can be configured by user defined functions, they are very high-level, hence the programmer can use and compose the provided abstractions, but can not implement new ones.

In Besard et al. [4], a new infrastructure for extendable compilers is proposed, and as a case study, the Julia functional language is extended for NVIDIA GPU programming. Julia provides higher-order abstractions for GPU programming, such as map and reduce, that can be configured using user defined functions but regular kernels are not higher-order.

Numerous works investigated skeletons in different parallel architectures, including GPUs. One state-of-art example is the SkePU C++ framework, that was released in 2010 [13] and is in continuous development, e.g, [13–16, 33]. The SkePU framework provides a collection of skeletons targeting different parallel architectures, GPUs included. In the first version of SKePU, user functions had to be defined using special C++ macros, that were expanded to implementations on different architectures. Users of the system were restricted by the collection of macros provided since, user function signatures need to match one of the available macros. SkePU 2 [16] and SkePU 3 [14] substitute the use of macros by a source-to-source translator (precompiler) implemented using libraries from the Clang project. Our objective in this paper was not to implement an efficient skeleton library, but rather to demonstrate that the idea of skeleton programming can be expressed by using the abstractions provided by Hok.

GPU programming within the Actors model has been previously explored in studies such as [18, 21, 22]. In [21] and [22], an extension to the C++ Actor Framework (CAF) [8] is introduced which shares similarities with our approach, as it enables programmers to embed low-level OpenCL kernels within Actors. The implementation of this system is facilitated by CAF being a C++ library, a language compatible with OpenCL. Additionally, [18] describes the Ensemble Actors language which allows the incorporation of OpenCL code within actors through an OpenCL dialect specific to Ensemble. Unlike CAF, Ensemble is a completely new language and thus requires a dedicated compiler.

7 Conclusions and Future Work

This paper presented Hok, an extension of the functional programming language Elixir for developing GPU programs. Hok provides an abstraction of GPU arrays and a DSL for implementing kernels. The main abstraction provided by Hok are higher-order kernel, i.e., kernels that can take functions as arguments. Hok allows device functions and anonymous device functions to be referenced at host code, and used to configure kernels when they are launched. Device functions are first-class values, so they can be passed to kernels or other device functions. We have demonstrated that Hok can be used to implement high-level abstractions for GPU programming, including algorithmic skeletons and array comprehensions. The source code for Hok and the benchmarks used in the experiments is available as free software in a GitHub repository. [5]

There are a number of lines for future work. In the near future, we plan to investigate other high-level abstractions using Hok, including abstractions for multi GPU programming. We also want to investigate how to implement other algorithmic skeletons available in the literature. Furthermore, the Matrex library represents its numerical arrays internally as arrays of floats, this simplifies type inference, and allows the type system to be monomorphic. This also allows the compiler to generate device code during compilation and not at runtime. In future versions of the system, we plan to provide compatibility with the Nx [2] library, and support other array types. Type checking/type inference could be performed in a similar way for different types of arrays. Mixing arrays of different types is more challenging for type inference, and in that case code generation for kernels could happen at kernel launch, where we already know the types of the arrays in use, and type specialization [35] can be performed. Furthermore, we would like to compare Hok with Nx, pure CUDA, and other GPU frameworks in the future.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

[5]https://github.com/ardubois/hok

References

[1] 2024. The NIFs library. WWW page, https://www.erlang.org/doc/man/erl_nif.html.

[2] 2024. THe Nx library. WWW page, https://github.com/elixir-nx/nx.

[3] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph. D. Dissertation. Royal Institute of Technology, Stockholm, Sweden.

[4] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2019. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 827–841. https://doi.org/10.1109/TPDS.2018.2872064

[5] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2 (Oct. 2023). https://doi.org/10.22152/programming-journal.org/2024/8/4

[6] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an Embedded Data Parallel Language. *SIGPLAN Not.* 46, 8 (feb 2011), 47–56. https://doi.org/10.1145/2038037.1941562

[7] 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 DAMP 2011* (Austin, Texas, USA). ACM, New York, NY, USA, 3–14.

[8] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2014. CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents Decentralized Control* (Portland, Oregon, USA) *(AGERE! '14)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2687357.2687363

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[10] Murray I Cole. 1989. *Algorithmic skeletons: structured management of parallel computation.* Pitman London.

[11] Andre Rauber Du Bois and Gerson Cavalheiro. 2023. GPotion: An embedded DSL for GPU programming in Elixir. In *Proceedings of the XXVII Brazilian Symposium on Programming Languages* (, Campo Grande, MS, Brazil,) *(SBLP '23)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3624309.3624314

[12] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. 2012. Compiling a High-Level Language for GPUs: (Via Language Support for Architectures and Compilers). 47, 6 (2012). https://doi.org/10.1145/2345156.2254066

[13] Johan Enmyren and Christoph W Kessler. 2010. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*. 5–14.

[14] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. 2021. SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters. *Int. J. Parallel Program.* 49, 6 (dec 2021), 846–866. https://doi.org/10.1007/s10766-021-00704-3

[15] August Ernstsson, Dalvan Griebler, and Christoph Kessler. 2023. Assessing Application Efficiency and Performance Portability in Single-Source Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming* 51, 1 (2023), 61–82.

[16] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming* 46 (2018), 62–80.

[17] Tianyi David Han and Tarek S. Abdelrahman. 2011. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2011), 78–90. https://doi.org/10.1109/TPDS.2010.62

[18] Paul Harvey, Kristian Hentschel, and Joseph Sventek. 2015. Parallel Programming in Actor-Based Applications via OpenCL. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada) *(Middleware '15)*. ACM, New York, NY, USA, 162–172.

[19] 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.* 52, 6 (jun 2017), 556–571. https://doi.org/10.1145/3140587.3062354

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[21] Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. 2015. Manyfold Actors: Extending the C++ Actor Framework to Heterogeneous Many-Core Machines Using OpenCL. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Pittsburgh, PA, USA) *(AGERE! 2015)*. ACM, New York, NY, USA, 45–56.

[22] Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. 2018. OpenCL Actors – Adding Data Parallelism to Actor-Based Programming with CAF. In *LNCS*. Springer International Publishing, 59–93.

[23] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11, Article 239 (mar 2023), 81 pages. https://doi.org/10.1145/3570638

[24] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. 2013. GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 315–324. https://doi.org/10.1109/IPDPSW.2013.173

[25] John Högberg. 2020. A brief introduction to BEAM. WWW page, https://www.erlang.org/blog/a-brief-beam-primer/.

[26] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. ACM, New York, NY, USA, Article 7, 6 pages.

[27] Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (Baltimore, Maryland, USA) *(Haskell '10)*. ACM, New York, NY, USA, 67–78.

[28] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. *SIGPLAN Not.* 48, 9 (sep 2013), 49–60. https://doi.org/10.1145/2544174.2500595

[29] Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. 2015. Type-Safe Runtime Code Generation: Accelerate to LLVM. *SIGPLAN Not.* 50, 12 (aug 2015), 201–212. https://doi.org/10.1145/2887747.2804313

[30] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2016. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 210–224. https://doi.org/10.1109/TPDS.2015.2394802

[31] Thomas Nelson. 2022. Introducing Microsoft Orleans. In *Introducing Microsoft Orleans: Implementing Cloud-Native Services with a Virtual Actor Framework*. Springer, 17–27.

[32] ROYUD Nishino and Shohei Hido Crissman Loomis. 2017. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems* 151, 7 (2017).

[33] Tomas Öhberg, August Ernstsson, and Christoph Kessler. 2020. Hybrid CPU–GPU execution support in the skeleton programming framework SkePU. *The Journal of Supercomputing* 76, 7 (2020), 5038–5056.

[34] Dinei A. Rockenbach, Júnior Löff, Gabriell Araujo, Dalvan Griebler, and Luiz Gustavo Fernandes. 2022. High-Level Stream and Data Parallelism in C++ for GPUs. In *Proceedings of the XXVI Brazilian Symposium on Programming Languages* (Virtual Event, Brazil) *(SBLP '22)*. ACM, 41–49.

[35] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. 2012. Parakeet: A Just-in-Time Parallel Accelerator for Python. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism* (Berkeley, CA) *(HotPar'12)*. USENIX Association, USA, 14.

[36] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[37] Satish Narayana Srirama, Freddy Marcelo Surriabre Dick, and Mainak Adhikari. 2021. Akka framework based on the Actor model for executing distributed Fog Computing applications. *Future Generation Computer Systems* 117 (2021), 439–452. https://doi.org/10.1016/j.future.2020.12.011

[38] Satish Narayana Srirama and Deepika Vemuri. 2023. CANTO: An actor model-based distributed fog framework supporting neural networks training in IoT applications. *Computer Communications* 199 (2023), 1–9.

[39] Andrew Stromme, Ryan Carlson, and Tia Newhall. 2012. Chestnut: A GPU Programming Language for Non-Experts. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores* (New Orleans, Louisiana) *(PMAM '12)*. Association for Computing Machinery, New York, NY, USA, 156–167. https://doi.org/10.1145/2141702.2141720

[40] Joel Svensson, Koen Claessen, and Mary Sheeran. 2010. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science* 1, 1 (2010), 2065–2074. https://doi.org/10.1016/j.procs.2010.04.231 ICCS 2010.

[41] Ruomeng (Cocoa) Xu, Anna Lito Michala, and Phil Trinder. 2022. CAEFL: Composable and Environment Aware Federated Learning Models. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang* (Ljubljana, Slovenia) *(Erlang 2022)*. ACM, New York, NY, USA, 9–20.

[42] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par 2009 Parallel Processing*, Henk Sips, Dick Epema, and Hai-Xiang Lin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 887–899.