Kernels de ordem superior para GPUs compatíveis com OpenCL utilizando metaprogramação em Elixir*

Henrique Gabriel Rodrigues[1], **André Rauber Du Bois**[1], **Gerson Geraldo H. Cavalheiro**[1]

[1]Programa de Pós-Graduação em Computação (PPGC)
Universidade Federal de Pelotas (UFPel) – Pelotas, RS – Brasil

{henrique.grdr,dubois,gerson.cavalheiro}@inf.ufpel.edu.br

Abstract. *GPU programming requires the use of low-level APIs, such as CUDA and OpenCL, which hinder the creation of abstractions like higher-order functions. Domain-Specific Languages (DSLs) mitigate this issue but often face limitations in portability or expressiveness. This paper presents OCL-PolyHok, an embedded DSL in Elixir that enables the execution of higher-order kernels on OpenCL, overcoming the lack of function pointer support through code generation and runtime specialization. Experiments with six benchmarks show, with 95% statistical confidence, that OCL-PolyHok outperforms its CUDA-based version, achieving lower JIT compilation overhead while preserving portability and expressiveness.*

Resumo. *A programação de GPUs exige o uso de APIs de baixo nível, como CUDA e OpenCL, que dificultam a criação de abstrações como funções de ordem superior. Linguagens de Domínio Específico (DSLs) atenuam esse problema, mas apresentam limitações de portabilidade ou expressividade. Este artigo apresenta a OCL-PolyHok, uma DSL embutida em Elixir que permite a execução de kernels de ordem superior em OpenCL, contornando a ausência de suporte a ponteiros de função por meio de geração de código e especialização em tempo de execução. Experimentos com seis benchmarks indicam, com 95% de confiança estatística, desempenho superior à versão baseada em CUDA, com menor overhead de compilação JIT, mantendo portabilidade e expressividade.*

1. Introdução

A Programação de Propósito Geral em GPUs (GPGPU) permite que o hardware de Unidades de Processamento Gráfico, altamente paralelo, seja aproveitado para aplicações de propósito geral, antes restrito apenas a CPUs. Essa técnica é extremamente eficiente para explorar o paralelismo sobre grandes volumes de dados, como evidenciado em padrões de computação paralela recorrentes (e.g., `map`), multiplicação de matrizes, simulações de partículas (*nBodies*) e busca de vizinhos mais próximos. No entanto, a técnica de GPGPU requer o uso de APIs de baixo nível, como CUDA e OpenCL, que possuem sintaxe verbosa e exigem o gerenciamento manual de memória, paralelismo e sincronização [Kolesnichenko et al. 2015]. Nesse contexto, as Linguagens de Domínio Específico (DSLs) consolidaram-se como uma es-

*Este estudo foi financiado em parte pela CAPES – Código de Financiamento 001, e pela FAPERGS – Processo nº 24/2551-0001372-5.

tratégia eficaz para elevar o nível de abstração, permitindo que algoritmos sejam expressos de forma mais intuitiva e produtiva, sem comprometer o desempenho final [Du Bois and Cavalheiro 2025, Du Bois et al. 2024, Henriksen et al. 2017].

Uma abstração que tem sido investigada para facilitar a programação de GPUs são os *algorithmic skeletons* (esqueletos algorítmicos) [Metzger 2021, Ernstsson et al. 2021], os quais encapsulam padrões de computação paralela recorrentes (como `map`, `reduce` e `scan`), permitindo que o desenvolvedor se concentre na lógica da aplicação. Contudo, para que os *algorithmic skeletons* sejam implementados e executados de forma eficiente, é fundamental que a plataforma de execução ofereça suporte a funções de ordem superior (*higher-order functions*). Funções de ordem superior são aquelas capazes de receber outra função como argumento ou retorná-la como resultado. Esse recurso não possui um suporte nativo ideal nas principais APIs de computação heterogênea. No ecossistema CUDA, embora seja permitido ponteiros de função, a prática prejudica o desempenho uma vez que os compiladores atuais não conseguem aplicar diversas otimizações quando ponteiros de função estão presentes [Du Bois and Cavalheiro 2025]; já o padrão OpenCL proíbe explicitamente o uso de ponteiros de função, impossibilitando nativamente os *kernels* de ordem superior.

Existem diversas DSLs e *frameworks* que buscam oferecer *algorithmic skeletons* e/ou *kernels* de ordem superior para simplificar a programação GPGPU, como a DSL Futhark [Henriksen et al. 2017], a DSL Accelerate [Chakravarty et al. 2011] e o *framework* SkePU [Ernstsson et al. 2021]. Contudo, essas soluções frequentemente possuem algum tipo de limitação: ou carecem de portabilidade por serem fortemente acopladas à API CUDA (limitando-se ao *hardware* NVIDIA), ou restringem a expressividade do programador oferecendo um conjunto pré-definido de *skeletons* e, tipicamente, um único modelo de paralelismo suportado. Nesse contexto, a DSL PolyHok [Du Bois and Cavalheiro 2025] introduziu a abstração de *kernels* de ordem superior na linguagem Elixir, com alta granularidade de controle da computação no dispositivo e sem restrições quanto ao modelo de paralelismo. Todavia, a sua implementação original carece de portabilidade por ser fortemente acoplada à API CUDA.

Este trabalho apresenta a OCL-PolyHok, um novo compilador e ambiente de execução para a linguagem PolyHok que permite o uso de *kernels* de ordem superior em qualquer GPU compatível com o padrão aberto OpenCL. Experimentos com seis *benchmarks* diferentes demonstram que a OCL-PolyHok supera, em todos os cenários avaliados, o antigo ambiente de execução baseado em CUDA da PolyHok, possuindo um menor *overhead* de compilação JIT. Como contribuição à ciência aberta, a ferramenta é disponibilizada como *software* livre. O código-fonte, os experimentos e a documentação completa do ecossistema PolyHok encontram-se no site oficial do projeto[1] e em seu repositório no GitHub[2].

O restante deste artigo está organizado como segue: a Seção 2 apresenta a fundamentação teórica necessária para a compreensão da plataforma; a Seção 3 aborda a DSL PolyHok, linguagem que serviu de base para a OCL-PolyHok; a Seção 4 detalha a arquitetura do novo compilador e do ambiente de execução da OCL-PolyHok; a Seção 5 apresenta e discute os resultados dos experimentos realizados; a Seção 6 discute os

[1] `https://lups.inf.ufpel.edu.br/polyhok`
[2] `https://github.com/Equiel-1703/ocl-polyhok`

trabalhos relacionados; e, por fim, a Seção 7 apresenta as considerações finais.

2. Fundamentação Teórica

Nesta seção serão apresentados os conceitos essenciais para a compreensão da plataforma proposta. Inicialmente, aborda-se o padrão OpenCL e suas restrições arquiteturais. Em seguida, a linguagem Elixir e seu mecanismo de metaprogramação.

2.1. O Padrão OpenCL

O OpenCL (Open Computing Language) é um padrão aberto e multiplataforma mantido pelo Khronos Group para o desenvolvimento de aplicações paralelas em ambientes heterogêneos, incluindo CPUs, GPUs e outros aceleradores [Munshi et al. 2011]. Seu modelo de execução divide o sistema em um *Host*, responsável pelo controle da execução, e um ou mais dispositivos de computação, responsáveis pela execução de *kernels*.

A execução de *kernels* ocorre sobre um espaço N-dimensional (*NDRange*), composto por múltiplos *work-items* organizados em *work-groups*. Apesar de sua portabilidade, a linguagem de programação de *kernels* do OpenCL (o OpenCL C) impõe restrições severas visando garantir segurança e compatibilidade entre dispositivos. Dentre essas restrições, destaca-se a proibição explícita ao suporte de ponteiros de função [Khronos OpenCL Working Group 2016], o que impede o uso direto de funções de ordem superior e, consequentemente, dificultando a criação de abstrações de alto nível como os *algorithmic skeletons*.

2.2. Elixir e Metaprogramação

O Elixir é uma linguagem de programação funcional de propósito geral executada sobre a máquina virtual Erlang (BEAM), oferecendo suporte nativo à concorrência e um robusto sistema de metaprogramação baseado em macros [Elixir Team 2026]. Em Elixir, as macros permitem que o código seja tratado como uma estrutura de dados, provendo acesso direto sobre a Árvore Sintática Abstrata (AST – *Abstract Syntax Tree*). Em vez de receber valores avaliados em tempo de execução, as macros recebem a representação em AST dos argumentos, permitindo manipular essas estruturas e injetar comportamentos/outras estruturas customizadas em tempo de compilação. Este mecanismo de interceptação e manipulação da AST foi crucial para a implementação do compilador da OCL-PolyHok (Seção 4.2).

3. A DSL PolyHok

A PolyHok [Du Bois and Cavalheiro 2025] é uma Linguagem de Domínio Específico (DSL) embutida em Elixir projetada para facilitar a programação GPGPU com abstrações de alto nível, como *kernels* de ordem superior e polimorfismo dinâmico. A linguagem também oferece gerência automática de memória do dispositivo, tipagem dinâmica e funções anônimas na GPU. A Listagem 1 ilustra a sintaxe da DSL.

O exemplo implementa e utiliza um algoritmo de `map` na lingugagem. Um `map` é um esqueleto algorítmico que aplica uma dada função a cada elemento de uma coleção de dados, gerando uma nova coleção com os resultados. Por meio da macro `defk` (linha 4), define-se o *kernel* `map_ker`, que recebe a função `f` como um de seus argumentos – evidenciando o suporte da DSL a *kernels* de ordem superior. No corpo do *kernel* (linhas 5

e 6), observa-se a presença de variáveis intrínsecas da API CUDA (como `blockIdx.x` e `threadIdx.x`) para o cálculo da grade de paralelismo, demonstrando o forte acoplamento da DSL original com o ecossistema CUDA. A aplicação da função `f` a cada elemento ocorre dentro do laço de repetição na linha 9.

Listagem 1. Exemplo de implementação de um Map na DSL PolyHok

```
1  require PolyHok
2
3  PolyHok.defmodule PMap do
4      defk map_ker(a1,a2,size,f) do
5          index = blockIdx.x * blockDim.x + threadIdx.x
6          stride = blockDim.x * gridDim.x
7
8          for i in range(index,size,stride) do
9              a2[i] = f(a1[i])
10         end
11     end
12
13     defd inc(x) do
14       x+1
15     end
16     def map(input, f) do
17         shape = PolyHok.get_shape(input)
18         result_gpu = PolyHok.new_gnx(shape, PolyHok.get_type(input))
19
20         size = Tuple.product(shape)
21         threadsPerBlock = 128;
22         numberOfBlocks = div(size + threadsPerBlock - 1,
23                     threadsPerBlock)
24
25         PolyHok.spawn(&PMap.map_ker/4,
26                     {numberOfBlocks,1,1},
27                     {threadsPerBlock,1,1},
28                     [input,result_gpu,size, f])
29         result_gpu
30     end
31 end
32
33 # Criando um array com mil numeros inteiros
34 arr_int = Nx.tensor(Enum.to_list(1..1000),type: {:s, 32})
35
36 # Invocando skeleton no array
37 host_res1 = arr_int |> PolyHok.new_gnx
38                     |> PMap.map(&PMap.inc/1)
39                     |> PolyHok.get_gnx
```

Em seguida, a macro `defd` é utilizada (linhas 13 a 15) para declarar a função *device* `inc`, que simplesmente incrementa o valor recebido. Esta função será posteriormente repassada como argumento ao *kernel*. A função hospedeira `map` (linhas 16 a 30) encapsula a alocação de memória na GPU (linha 18, via `PolyHok.new_gnx`) e o cálculo das dimensões de bloco e *threads* (linhas 20 a 23), utilizando a diretiva `PolyHok.spawn` (linhas 25 a 28) para despachar a execução. Por fim, o fluxo de execução principal (linhas 34 a 39) ilustra o alto nível de abstração herdado do paradigma funcional, empregando o operador *pipe* (`|>`) do Elixir para encadear de forma fluida a transferência do tensor `Nx` para a GPU, a invocação do esqueleto recebendo a função por referência na linha 38 (`&PMap.inc/1`), e a imediata recuperação dos resultados para a memória do *host* (através da chamada `PolyHok.get_gnx` na linha 39). Além de funções *device* definidas com `defd`, os *kernels* implementados na PolyHok também aceitam funções anônimas como argumento.

4. Implementação

Esta seção apresenta a implementação da DSL OCL-PolyHok, incluindo a abstração GNx, o suporte a *kernels* de ordem superior com compilação JIT, e as extensões de sintaxe relacionadas à retrocompatibilidade.

4.1. GNx: Abstração de Arrays na GPU

Em uma aplicação típica de GPGPU é necessário transferir dados alocados na memória do *Host* (normalmente a RAM) para a GPU e, após a computação, transferir de volta. No ecossistema Elixir, a biblioteca Nx (*Numerical Elixir*) oferece a capacidade de criar tensores numéricos multidimensionais de diferentes tipos de dados. Assim como a PolyHok original, a OCL-PolyHok utiliza a abstração GNx (*GPU Nx*) para representar *arrays* que residem na GPU [Du Bois and Cavalheiro 2025], portando essa abstração para um ambiente OpenCL.

Um novo GNx pode ser criado utilizando função `new_gnx`, que recebe um tensor Nx e o aloca no dispositivo OpenCL, retornando uma referência para um objeto GNx no *Host*. Inversamente, a função `get_gnx` copia os dados de um GNx alocado na GPU de volta para a memória principal, reconstituindo o tensor Nx original que pode ser manipulado utilizando as funções da biblioteca Nx no Elixir puro.

Um GNx, em termos simples, nada mais é do que uma referência para um Buffer OpenCL (`cl::Buffer`) que reside na GPU. Seus dados não podem ser acessados diretamente no Elixir puro, mas somente em *kernels* e funções *device* da OCL-PolyHok.

Um dos diferenciais da arquitetura da OCL-PolyHok é a gerência automática de memória da GPU pela máquina virtual BEAM. Internamente, *arrays* GNx são representados como recursos controlados pelo *Garbage Collector* da BEAM. Quando um GNx deixa de ser referenciado no escopo do programa hospedeiro, a máquina virtual invoca automaticamente um destrutor que executa comandos da API OpenCL para liberar a memória do dispositivo. Esse mecanismo previne vazamentos de memória (*memory leaks*) sem exigir que o programador gerencie alocações manualmente.

4.2. Compilação JIT e Kernels de Ordem Superior

Para suportar funções de ordem superior em um ambiente como o OpenCL, que proíbe explicitamente ponteiros de função, a OCL-PolyHok baseia-se fortemente nos recursos de metaprogramação do Elixir. Em Elixir, a metaprogramação é realizada principalmente por meio de macros, que são funções especiais executadas em tempo de compilação. Elas recebem a Árvore Sintática Abstrata (AST – *Abstract Syntax Tree*) do código, podem manipulá-la, e retornar uma nova.

Durante a compilação do código Elixir, a macro `PolyHok.defmodule` coleta as ASTs de todos os *kernels* e funções *device* definidos no módulo. Ela também computa uma lista de funções chamadas dentro de cada *kernel*/função *device* (um *call graph*) e envia essas informações para um processo Elixir (o *module_server*), responsável por armazenar esse dicionário estrutural. No caso de funções anônimas (definidas através da macro `PolyHok.phok`), estas são transformadas ainda em tempo de compilação em uma estrutura contendo um nome gerado aleatoriamente e a sua respectiva AST.

O passo principal, no entanto, ocorre em tempo de execução, quando o comando `spawn` é invocado. Para contornar a ausência de ponteiros de função, o processo de despacho e compilação do *kernel* ocorre em três etapas principais:

1. **Inferência e Verificação de Tipos:** Utilizando as ASTs previamente armazenadas e os tipos dos argumentos fornecidos ao *kernel*, executa-se um algoritmo de inferência de tipos que determina os tipos das variáveis internas e das funções em seu *call graph*. Esse algoritmo é essencialmente o mesmo da PolyHok original [Du Bois and Cavalheiro 2025], com a adição de um mapeamento para funções especiais do OpenCL. Assim, ao encontrar chamadas como `get_local_id`, o algoritmo já conhece suas assinaturas, evitando classificá-las incorretamente como funções *device* definidas pelo usuário.
2. **Geração de Código e *Inlining* Estático:** Com a AST e os tipos inferidos em mãos, o *backend* em Elixir inicia a geração do código-fonte OpenCL C. É nesta etapa que os *kernels* de ordem superior são efetivamente resolvidos. Todas as invocações das funções que foram passadas como argumento no corpo do *kernel* são interceptadas e substituídas por chamadas estáticas e diretas ao nome real das funções *device*. Consequentemente, os parâmetros de função são omitidos da assinatura final do *kernel* gerado. Se uma função anônima foi passada como argumento, a estrutura contendo sua AST e seu nome (atribuído pelo processo de geração) é anexada à árvore de chamadas do *kernel* e seu código é gerado no programa OpenCL resultante como uma função *device*.
3. **Compilação JIT:** O resultado da etapa anterior é um código-fonte estático, puramente de primeira ordem e válido no padrão OpenCL. Essa *string* de código é submetida, então, ao *driver* OpenCL do dispositivo. O *driver* realiza a compilação *Just-In-Time* (JIT), traduzindo o código para a linguagem de máquina específica do *hardware* subjacente e iniciando a execução paralela do *kernel* na GPU.

4.3. Adições de Sintaxe e Retrocompatibilidade

O padrão OpenCL utiliza um modelo de indexação sintaticamente diferente do utilizado pelo CUDA. Na API CUDA existem estruturas (*structs*) especiais que o programador pode utilizar para acessar índices de *threads* ou dimensões de blocos, como `threadIdx.x` e `blockDim.x`. O OpenCL, por outro lado, oferece funções nativas, como `get_local_id(0)` e `get_local_size(0)`, que são semanticamente equivalentes às estruturas do CUDA. Adicionalmente, o OpenCL ainda oferece funções especiais, como a `get_global_id`, que retorna o índice global do elemento corrente. Essa função dispensa o cálculo manual necessário em CUDA para obter o mesmo valor.

Com o objetivo de oferecer total expressividade ao programador e, ao mesmo tempo, prover retrocompatibilidade com códigos previamente desenvolvidos para a PolyHok original baseada em CUDA, decidiu-se por suportar na OCL-PolyHok ambas as sintaxes. Agora, o desenvolvedor pode escolher escrever *kernels* utilizando as funções nativas do OpenCL ou manter a sintaxe tradicional de estruturas CUDA.

5. Experimentos

Para avaliar o impacto e a eficiência do novo ambiente de execução da OCL-PolyHok, foram conduzidos experimentos medindo o tempo total de execução de seis *benchmarks*

paralelos distintos, que utilizam esqueletos algorítmicos implementados nas DSLs para realizar a computação. Os algoritmos avaliados foram: *Dot Product* (DP – utiliza `map` e `reduce`), *Julia Set* (JL – utiliza `map`), *Matrix Multiplication* (MM – utiliza `map`), *nBodies* (NB – utiliza `map`), *Nearest Neighbor* (NN – utiliza `map` e `reduce`) e *Ray Tracer* (RT – utiliza `map`). O código-fonte dos *benchmarks*, bem como o da OCL-PolyHok, está disponível em um repositório público no GitHub[3].

A avaliação ocorreu em quatro cenários distintos: utilizando a OCL-PolyHok (apresentada neste trabalho), a PolyHok original baseada em CUDA, e versões nativas implementadas diretamente em OpenCL e CUDA puros. Para garantir uma comparação justa e exata, os códigos das versões em OpenCL e CUDA puros foram extraídos diretamente dos códigos gerados pelos próprios compiladores da PolyHok e OCL-PolyHok. Dessa forma, as implementações nativas são estritamente equivalentes aos códigos executados pelas DSLs na GPU. Os testes foram executados em um ambiente equipado com um processador Intel(R) Core(TM) i7-12700K @ 3.60GHz, 16 GB de RAM e uma GPU NVIDIA GeForce RTX 3060 com 8 GB de VRAM. O sistema operacional utilizado foi o Ubuntu 22.04.3 LTS, com Elixir 1.16.1 e Erlang/OTP 26.

Para garantir uma comparação justa entre os *benchmarks*, todas as versões utilizam as mesmas configurações de paralelismo (*grid*/*work-groups*/*threads*). Nas DSLs, o tempo medido engloba a inferência dinâmica de tipos, geração de código, compilação JIT e movimentação de memória entre *Host* e *Device*, enquanto nos códigos nativos (CUDA e OpenCL puros) corresponde apenas à execução na GPU e à movimentação de memória. Cada *benchmark* foi executado 30 vezes em cada cenário, e intervalos de confiança de 95% foram estimados utilizando o método *bootstrap* com 10.000 iterações para as médias das DSLs. A Tabela 1 apresenta os resultados obtidos, com o tempo médio de execução, seus respectivos intervalos de confiança e o percentual de ganho de desempenho da OCL-PolyHok em relação à PolyHok original.

5.1. Análise Estatística do Desempenho

A análise inicial das amostras por meio do teste de Shapiro-Wilk revelou a ausência de normalidade na distribuição dos dados. Dessa forma, adotou-se o teste não-paramétrico de Mann-Whitney (*U-test*) para comparar as distribuições da OCL-PolyHok com a PolyHok original. Adicionalmente, calculou-se o tamanho da diferença por meio do coeficiente d de Cohen. A Tabela 2 apresenta os resultados dos testes.

Em todos os cenários, o p-valor obtido foi substancialmente inferior ao limiar de significância adotado ($\alpha = 0,05$), permitindo afirmar, com 95% de confiança estatística, que a OCL-PolyHok apresenta um desempenho superior à PolyHok original. Observa-se que a estatística U resultou em zero na grande maioria dos experimentos, o que demonstra a ausência de sobreposição nos dados e evidencia a clara vantagem da OCL-PolyHok. Nos poucos casos em que $U \neq 0$ (exclusivos do *nBodies*), o p-valor e o coeficiente d asseguram que a ferramenta manteve a superioridade. Valores de $|d|$ do coeficiente de Cohen superiores a 0,8 são tipicamente considerados diferenças grandes (o sinal negativo na Tabela 2 apenas indica a direção da diferença). Nos resultados obtidos, a grande maioria ultrapassa 8, chegando ao expressivo valor de $|54,99|$ no caso do JL 9.216.

[3]`https://github.com/Equiel-1703/ocl-polyhok`

Tabela 1. Resultados dos *benchmarks*. Tempos em milissegundos (ms).

Benchmark	OCL-PolyHok	PolyHok	Ganho (%)	OpenCL	CUDA
DP 400M	647,2 [643,5; 653,5]	742,7 [741,5; 744,1]	12,86%	576,4	514,6
DP 500M	788,4 [787,6; 789,2]	868,3 [867,3; 869,3]	9,20%	713,9	638,8
DP 600M	929,3 [928,6; 930,1]	994,0 [992,8; 995,2]	6,51%	861,2	764,5
JL 7.168×7.168	494,9 [494,0; 496,0]	632,9 [628,8; 640,0]	21,80%	492,1	503,9
JL 9.216×9.216	814,3 [813,7; 815,1]	955,9 [954,9; 957,0]	14,81%	812,6	831,9
JL 11.264×11.264	1.213,5 [1.212,9; 1.214,2]	1.363,2 [1.361,7; 1.365,0]	10,98%	1.213,3	1.241,7
MM 5k	254,2 [252,1; 256,7]	404,2 [400,3; 408,7]	37,11%	231,8	254,0
MM 7k	607,9 [605,9; 610,8]	767,2 [760,9; 773,2]	20,76%	606,6	665,6
MM 9k	1.285,2 [1.279,9; 1.292,2]	1.403,6 [1.402,3; 1.404,8]	8,44%	1.323,0	1.481,0
NB 100k	2.042,9 [2.041,4; 2.044,5]	2.180,6 [2.179,8; 2.181,4]	6,31%	2.040,7	2.041,6
NB 200k	8.186,8 [8.167,0; 8.211,2]	8.300,3 [8.291,2; 8.313,5]	1,37%	8.184,3	8.165,0
NB 400k	32.780,2 [32.686,1; 32.904,9]	32.863,7 [32.785,5; 32.956,0]	0,25%	32.790,7	32.736,6
NN 100M	462,6 [460,6; 464,6]	665,1 [663,2; 666,9]	30,45%	433,5	432,9
NN 200M	874,5 [873,2; 875,9]	1.078,4 [1.076,9; 1.080,1]	18,91%	849,2	848,3
NN 300M	1.292,0 [1.291,6; 1.292,4]	1.538,9 [1.498,0; 1.619,3]	16,04%	1.273,1	1.267,9
RT 7.168×7.168	235,0 [233,7; 236,6]	365,4 [364,0; 367,0]	35,69%	231,2	231,0
RT 9.216×9.216	383,3 [382,6; 384,2]	514,9 [513,7; 516,2]	25,56%	380,9	381,0
RT 11.264×11.264	570,6 [569,2; 572,4]	711,1 [701,2; 726,2]	19,76%	570,0	569,5

Os ganhos de desempenho observados sugerem que o *overhead* de compilação JIT (*Just-In-Time*) do OpenCL é inferior e atende de forma mais eficiente o modelo de geração de código em tempo de execução de uma DSL do que o compilador de tempo de execução oferecido pela NVIDIA: o *NVIDIA Runtime Compiler* (NVRTC), utilizado na versão anterior da linguagem. Uma evidência forte que sustenta esta hipótese pode ser observada nos tempos do *benchmark* DP. A implementação pura em OpenCL foi mais lenta do que a versão em CUDA em todas as entradas. Ainda assim, a OCL-PolyHok superou a PolyHok, evidenciando um *overhead* significativamente menor.

5.2. Otimização de Memória via Máquina Virtual BEAM

No *benchmark* MM, a OCL-PolyHok apresentou um tempo total de execução menor do que as versões em CUDA puro e OpenCL puro para a maior carga de dados (9k), con-

Tabela 2. Análise estatística comparando OCL-PolyHok e PolyHok.

Benchmark	p-valor	U	Cohen's d
DP 400M	$2,69 \times 10^{-11}$	0,0	-8,26
DP 500M	$2,66 \times 10^{-11}$	0,0	-31,10
DP 600M	$2,22 \times 10^{-11}$	0,0	-22,67
JL 7.168 $\times$ 7.168	$2,46 \times 10^{-11}$	0,0	-10,77
JL 9.216 $\times$ 9.216	$2,43 \times 10^{-11}$	0,0	-54,99
JL 11.264 $\times$ 11.264	$2,46 \times 10^{-11}$	0,0	-42,02
MM 5k	$2,65 \times 10^{-11}$	0,0	-15,64
MM 7k	$2,89 \times 10^{-11}$	0,0	-11,76
MM 9k	$2,90 \times 10^{-11}$	0,0	-9,32
NB 100k	$2,59 \times 10^{-11}$	0,0	-39,68
NB 200k	$6,10 \times 10^{-8}$	84,0	-2,31
NB 400k	$1,16 \times 10^{-5}$	153,0	-0,30
NN 100M	$2,89 \times 10^{-11}$	0,0	-37,10
NN 200M	$2,76 \times 10^{-11}$	0,0	-48,75
NN 300M	$1,97 \times 10^{-11}$	0,0	-1,60
RT 7.168 $\times$ 7.168	$2,14 \times 10^{-11}$	0,0	-31,32
RT 9.216 $\times$ 9.216	$2,59 \times 10^{-11}$	0,0	-44,18
RT 11.264 $\times$ 11.264	$2,56 \times 10^{-11}$	0,0	-5,35

forme evidencia a Tabela 3. Tal resultado desafia a premissa de que abstrações de alto nível sempre carregam uma penalidade de desempenho inerente, resultando em *overhead* sistemático.

Tabela 3. Comparação de desempenho no *benchmark* MM.

Benchmark	OCL-PolyHok (ms)	OpenCL (ms)	CUDA (ms)	Diferença (%) vs OpenCL	Diferença (%) vs CUDA
MM 5k	254,2	231,8	254,0	+9,65%	+0,05%
MM 7k	607,9	606,6	665,6	+0,21%	-8,67%
MM 9k	1.285,2	1.323,0	1.481,0	-2,86%	-13,22%

Após testes e investigações no ciclo de vida da memória, descobriu-se que a vantagem da DSL neste cenário não reside no tempo de execução do *kernel*, mas sim no gerenciamento eficiente da memória do *Host* realizado pela máquina virtual BEAM. A máquina BEAM gerencia grandes estruturas binárias por meio de contadores de referência (*Reference-Counted Binary*). Após o lançamento do *kernel*, os vetores de entrada da matriz perdem suas referências no escopo de execução, permitindo que a VM reutilize imediatamente essas regiões de memória no *Host* para armazenar o tensor de resultado vindo da GPU. Essa reciclagem evita o custo de alocar um novo espaço de memória pelo sistema operacional (que utiliza *lazy allocation*), algo que não foi feito nas versões puras em C/C++, mas que a BEAM faz automaticamente.

Experimentos forçando uma nova alocação de memória em Elixir por meio de dependências artificiais e ajustes na *flag* `:min_bin_vheap_size` confirmaram que a estratégia adotada pelo *Garbage Collector* da BEAM foi a responsável pela supressão do

custo de alocação. Especificamente, ao inibir manualmente a reutilização de *buffers* na VM, a OCL-PolyHok sofreu uma penalidade de desempenho de 6,57% no tempo total de execução para a entrada de 5k, e de 4,78% na carga máxima de 9k (conforme avaliações complementares). Inversamente, quando a mesma técnica de reaproveitamento de memória foi replicada manualmente na implementação pura, a versão nativa obteve reduções de tempo de 8,37% (MM 5k) e 5,33% (MM 9k), validando empiricamente o impacto da latência de alocação tardia (*lazy allocation*) do sistema operacional sobre os tempos.

6. Trabalhos Relacionados

A OCL-PolyHok é fruto de uma linha de pesquisa que investiga a integração da linguagem Elixir com a programação de alto desempenho em GPUs. Esta trajetória iniciou com a GPotion [Du Bois and Cavalheiro 2023], que introduziu o gerenciamento automático de memória na GPU via *Garbage Collector* da BEAM. A Hok [Du Bois et al. 2024] avançou ao suportar *kernels* de ordem superior, mas apresentou limitações de desempenho devido ao uso de ponteiros de função e restrição ao tipo *float*. A PolyHok [Du Bois and Cavalheiro 2025] unificou essas abordagens ao introduzir polimorfismo dinâmico e compilação *Just-In-Time* (JIT) baseada em manipulação de ASTs. A OCL-PolyHok herda essas abstrações, mas substitui o *backend* CUDA pelo padrão OpenCL, garantindo portabilidade entre diferentes aceleradores.

Na literatura acadêmica, a simplificação da programação heterogênea frequentemente ocorre via bibliotecas em linguagens de baixo nível. *Frameworks* baseados em C++, como SkePU [Ernstsson et al. 2021] e SkelCL [Steuwer et al. 2011], oferecem *algorithmic skeletons* com suporte a múltiplos *backends* (incluindo OpenCL). Entretanto, essas soluções exigem que o desenvolvedor permaneça atrelado à sintaxe e ao gerenciamento manual do ecossistema C/C++. A OCL-PolyHok, por sua vez, eleva a abstração para o paradigma funcional dinâmico do Elixir, delegando a orquestração e a coleta de lixo (*Garbage Collection*) dos dispositivos para a máquina virtual.

No espectro das linguagens funcionais, o Accelerate [Chakravarty et al. 2011] é uma DSL embutida em Haskell que compila computações sobre *arrays* para CUDA, utilizando otimizações como fusão de *arrays*. A OCL-PolyHok se alinha com o Accelerate na validação do paradigma funcional para programação de GPUs, mas diverge em aspectos cruciais: o Accelerate é estaticamente tipado e historicamente vinculado ao ecossistema CUDA, já a OCL-PolyHok adota a tipagem dinâmica com compilação JIT e utiliza OpenCL para assegurar independência de hardware.

A Futhark [Henriksen et al. 2017] é uma linguagem puramente funcional voltada ao processamento paralelo de *arrays*. Por ser uma linguagem *standalone* e utilizar um compilador *ahead-of-time* focado em otimizações agressivas, ela exige integração manual com a aplicação principal via uma linguagem hospedeira (como C, C++ ou Python). Em contraste, a OCL-PolyHok é uma DSL embutida em Elixir, unificando a lógica de negócio, a orquestração e os *kernels* em um único código-fonte. A Futhark também restringe a expressividade do programador ao suportar apenas paralelismo aninhado regular, expresso por meio de *Second-Order Array Combinators* (SOACs) – operadores de ordem superior análogos a esqueletos algorítmicos. A OCL-PolyHok, por outro lado, não exige regularidade topológica no paralelismo, e confere ao desenvolvedor um controle granular sobre a computação realizada no dispositivo.

No contexto de linguagens dinâmicas, diversos trabalhos exploram a programação GPGPU. No ecossistema Python, destacam-se Numba [Lam et al. 2015], Parakeet [Rubinsteyn et al. 2012] e Copperhead [Catanzaro et al. 2011], que utilizam compilação JIT para gerar código nativo otimizado para GPU (como PTX e CUDA C) a partir de construções de alto nível. Bibliotecas como CuPy [Okuta et al. 2017] e TensorFlow [Abadi et al. 2016] também oferecem aceleração por GPU, porém com abstrações voltadas principalmente para computação numérica e aprendizado de máquina. Em Java, o framework Tornado [Clarkson et al. 2018] permite execução transparente em GPUs via anotações e compilação JIT com Graal utilizando OpenCL, enquanto [Ishizaki et al. 2015] estendem o IBM Java 8 para compilar *lambdas* e *parallel streams* para GPU, ferramentas como jCUDA [Yan et al. 2009] fornecem uma interface amigável que automatiza a geração dos *bindings* (JNIs) e transferências de dados para chamadas CUDA a partir de Java.

7. Conclusão

Este artigo apresentou a OCL-PolyHok, um novo ambiente de execução para a linguagem PolyHok que viabiliza o uso de *kernels* de ordem superior em plataformas heterogêneas compatíveis com o padrão OpenCL, eliminando a dependência de *hardware* NVIDIA e garantindo portabilidade sem comprometer a expressividade. Por meio de metaprogramação em Elixir e manipulação da Árvore Sintática Abstrata (AST) em tempo de execução, a OCL-PolyHok contorna a ausência de ponteiros de função no OpenCL, permitindo a geração dinâmica de código estático altamente otimizável e preservando o suporte a funções *device* como argumentos.

A avaliação experimental demonstrou, com alto grau de confiança estatística, que a OCL-PolyHok supera a sua versão original baseada em CUDA em todos os cenários, apresentando menor *overhead* de compilação *Just-In-Time*. Além disso, a integração com a máquina virtual BEAM revelou uma vantagem arquitetural relevante: no *benchmark* de Multiplicação de Matrizes, a reutilização automática de memória eliminou custos de alocação, permitindo desempenho superior até mesmo a implementações equivalentes em CUDA e OpenCL puros.

Como trabalhos futuros, pretende-se expandir a OCL-PolyHok para suportar múltiplas GPUs e explorar também o paralelismo heterogêneo em múltiplos *cores* de CPU.

Referências

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). TensorFlow: a system for large-scale machine learning. In *Proc. USENIX OSDI 2016*, pages 265–283.

Catanzaro, B., Garland, M., and Keutzer, K. (2011). Copperhead: compiling an embedded data parallel language. *SIGPLAN Not.*, 46(8):47–56.

Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. (2011). Accelerating Haskell array codes with multicore GPUs. In *Proc. 6th DAMP Workshop*, page 3–14, New York, USA. ACM.

Clarkson, J., Fumero, J., Papadimitriou, M., Zakkak, F. S., Xekalaki, M., Kotselidis, C., and Luján, M. (2018). Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal. In *Proc. ManLang 2018*, pages 4:1–4:13.

Du Bois, A., Perlin, T., Antunes, F., and Cavalheiro, G. (2024). Hok: Higher-Order GPU kernels in Elixir. In *Anais SBLP 2024*, pages 71–80, Porto Alegre, RS, Brasil. SBC.

Du Bois, A. R. and Cavalheiro, G. (2023). GPotion: An embedded DSL for GPU programming in Elixir. In *Proc. SBLP 2023*, page 1–8, New York, USA. ACM.

Du Bois, A. R. and Cavalheiro, G. (2025). Polymorphic Higher-Order GPU Kernels. In *Proc. Euro-Par 2025*. Springer Nature.

Elixir Team (2026). *Elixir Lang. Doc.* The Elixir Team. Disponível em: `https://elixir-lang.org/docs.html`.

Ernstsson, A., Ahlqvist, J., Zouzoula, S., and Kessler, C. (2021). SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters. *Int. J. Parallel Program.*, 49(6):846–866.

Henriksen, T., Serup, N. G. W., Elsman, M., Henglein, F., and Oancea, C. E. (2017). Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proc. ACM SIGPLAN PLDI 2017*, pages 556–571, New York, USA. ACM.

Ishizaki, K., Hayashi, A., Koblents, G., and Sarkar, V. (2015). Compiling and Optimizing Java 8 Programs for GPU Execution. In *Proc. IEEE PACT 2015*, pages 419–431.

Khronos OpenCL Working Group (2016). *OpenCL C Spec. v2.0*. The Khronos Group Inc. Disponível em: `https://registry.khronos.org/OpenCL/specs/opencl-2.0-openclc.pdf`.

Kolesnichenko, A., Poskitt, C. M., Nanz, S., and Meyer, B. (2015). Contract-based general-purpose GPU programming. In *Proc. ACM SIGPLAN GPCE 2015*, page 75–84. ACM.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: a LLVM-based Python JIT compiler. In *Proc. LLVM-HPC 2015*, page 7.

Metzger, P. (2021). *Programmer-transparent efficient parallelism with skeletons*. PhD thesis, Univ. Edinburgh.

Munshi, A., Gaster, B., Mattson, T. G., Fung, J., and Ginsburg, D. (2011). *OpenCL Programming Guide*. Addison-Wesley Educational, Boston, MA.

Okuta, R., Unno, Y., Nishino, D., Hido, S., and Loomis, C. (2017). CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proc. LearningSys (NIPS) 2017*.

Rubinsteyn, A., Hielscher, E., Weinman, N., and Shasha, D. (2012). Parakeet: a just-in-time parallel accelerator for python. In *Proc. USENIX HotPar 2012*, page 14.

Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *Proc. IEEE IPDPSW 2011*, pages 1176–1182.

Yan, Y., Grossman, M., and Sarkar, V. (2009). JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Proc. Euro-Par 2009*, pages 887–899.