OCL-PolyHok: DSL para GPGPU com abstrações de alto nível baseada em OpenCL

Henrique Gabriel Rodrigues[1], **André Rauber Du Bois**[1]

[1]Centro de Desenvolvimento Tecnológico
Universidade Federal de Pelotas (UFPel) – Pelotas, RS – Brasil

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

Resumo. *A programação de GPUs para propósitos gerais (GPGPU) oferece alto desempenho, mas impõe barreiras devido à complexidade de APIs de baixo nível, como CUDA e OpenCL. Este trabalho apresenta a OCL-PolyHok, uma versão da DSL PolyHok que substitui seu backend e biblioteca de runtime, antes baseados em CUDA, por uma implementação baseada no padrão aberto OpenCL. O objetivo é prover portabilidade para a linguagem de programação de GPUs em diferentes arquiteturas de hardware, mantendo sua expressividade funcional e recursos de alto nível. Testes experimentais com três benchmarks demonstraram que a OCL-PolyHok cumpriu seu objetivo de portabilidade e superou o desempenho da versão original em todos os cenários avaliados.*

1. Introdução

O avanço das arquiteturas de hardware heterogêneas e da computação GPGPU permitiu a execução massivamente paralela de algoritmos complexos. Entretanto, as APIs tradicionais utilizadas para programar estes dispositivos (como CUDA e OpenCL) são ferramentas de baixo nível que impõem desafios devido à sintaxe verbosa e gerenciamento manual de memória e sincronização, como apontado por [Kolesnichenko et al. 2015]. Como alternativa, as Linguagens de Domínio Específico (DSLs) surgem para elevar o nível de abstração sem comprometer o desempenho.

Nesse cenário, foi proposta a PolyHok [Du Bois and Cavalheiro 2025]. Desenvolvida pelo Laboratório de Sistemas Ubíquos e Paralelos da Universidade Federal de Pelotas (LUPS/UFPel), a PolyHok é uma DSL para GPGPU embutida na linguagem de programação funcional Elixir, que oferece recursos de alto nível como polimorfismo dinâmico, kernels de alta ordem e gerência automática de memória. No entanto, a implementação original da PolyHok é restrita ao ecossistema proprietário CUDA, o que limita sua portabilidade e gera um aprisionamento tecnológico (*vendor lock-in*).

Este trabalho apresenta a OCL-PolyHok, uma versão da DSL PolyHok com um backend e biblioteca de tempo de execução (*runtime*) baseados no padrão aberto OpenCL. O objetivo principal é conferir agnosticismo de hardware à ferramenta, permitindo sua execução em dispositivos de diferentes fabricantes (como NVIDIA, AMD e Intel) enquanto preserva a expressividade original da linguagem.

A solução foi validada por meio de uma avaliação quantitativa da execução de três benchmarks, comparando o desempenho da OCL-PolyHok com sua versão original, além de implementações puras em OpenCL e CUDA. Os resultados demonstram que a transição para o padrão aberto garantiu a portabilidade almejada para a DSL e reduziu o tempo de execução em comparação à versão anterior.

2. Implementação

A implementação da OCL-PolyHok envolveu a modificação de dois componentes centrais do sistema original: o módulo gerador de código em Elixir (o *backend*) e a biblioteca de tempo de execução `gpu_nifs`. As subseções a seguir descrevem as modificações realizadas em cada componente. O código-fonte completo da OCL-PolyHok está disponível publicamente em um repositório no GitHub[1].

2.1. Geração de Código OpenCL C

Embora CUDA e OpenCL compartilhem semelhanças de sintaxe por serem baseadas em C, existem divergências nas abstrações de hardware, implicando em diferentes comandos nas APIs para realizar tarefas semelhantes. Para manter compatibilidade com código já existente para a PolyHok original, optou-se por preservar a sintaxe baseada em CUDA da PolyHok na OCL-PolyHok. Dessa forma, o novo backend atua como uma camada de tradução. Expressões intrínsecas do CUDA (como `threadIdx`, `blockIdx` e `blockDim`) são mapeadas automaticamente para as funções equivalentes do OpenCL (`get_local_id`, `get_group_id` e `get_local_size`).

Além disso, o backend também foi modificado para utilizar os qualificadores de memória corretos do OpenCL, mapeando o `__shared__` do CUDA para o `__local` do OpenCL e convertendo o comando de sincronização `__syncthreads()` para o equivalente `barrier(CLK_LOCAL_MEM_FENCE)` em OpenCL C.

Essas modificações foram implementadas no módulo gerador de código da DSL em Elixir utilizando técnicas de *pattern-matching* na Árvore Sintática Abstrata (AST) dos kernels e funções *device* implementados pelo usuário.

2.2. Biblioteca de Interface Nativa

A biblioteca `gpu_nifs` é uma biblioteca de tempo de execução (*runtime*) responsável pela comunicação entre a máquina virtual do Elixir (a BEAM) e a GPU. O código da biblioteca – originalmente escrito em C utilizando a API CUDA – foi integralmente reimplementada utilizando C++ versão 11 e a API para C++ do OpenCL. Para melhorar a organização e futuras manutenções do código, foi desenvolvida a classe `OCLInterface`, que encapsula toda a comunicação com a API do OpenCL. Dessa forma, o código da biblioteca `gpu_nifs` concentra-se na interface Elixir-GPU, enquanto a comunicação com a API do OpenCL é realizada internamente pela nova classe. Isso promove uma separação de responsabilidades que não existia anteriormente, facilitando a manutenção do código e a adição de novas funcionalidades na DSL.

2.3. Compatibilidade com Ponto Flutuante de 64 Bits

Diferente do ecossistema CUDA, onde o suporte a ponto flutuante de 64 bits (o tipo *double*) é obrigatório em todas as GPUs dessa arquitetura, no OpenCL este recurso é uma extensão opcional. Isso significa que toda GPU CUDA-compatível suporta operações com o tipo *double*, mas nem toda GPU compatível com OpenCL suporta esse tipo de dado — como é o caso de muitas GPUs de entrada e APUs (*Accelerated Processing Units*).

A OCL-PolyHok introduz uma verificação em tempo de execução que detecta se o *hardware* do usuário suporta *double*. Caso um kernel tente utilizar este tipo de dado

[1]Disponível em `https://github.com/Equiel-1703/ocl-polyhok`.

em um *hardware* incompatível, a DSL lança uma exceção com texto descritivo em Elixir explicando a limitação. Optou-se por realizar esta abordagem para evitar erros durante a compilação do kernel pelo compilador JIT (*Just-in-Time*) do OpenCL, que pode produzir mensagens de erro com uma terminologia mais complexa e dificultar a identificação da causa real do problema pelo programador.

3. Avaliação Experimental

Os experimentos foram conduzidos em uma máquina do laboratório LUPS/UFPel, equipada com um processador Intel(R) Core(TM) i7-12700K @ 3.60GHz, 16 GB de memória RAM e uma GPU NVIDIA GeForce RTX 3060 com 8 GB de VRAM. Utilizou-se o sistema operacional Ubuntu 22.04.3 LTS, com Elixir 1.16.1 e Erlang/OTP 26.

Foram selecionados três benchmarks para os testes: Produto Escalar (DP), Conjunto de Julia (JL) e Multiplicação de Matrizes (MM). Estes foram três dos benchmarks utilizados por [Du Bois and Cavalheiro 2025] na avaliação de desempenho da PolyHok original, garantindo comparabilidade dos resultados obtidos.

Os experimentos compararam a OCL-PolyHok com a PolyHok original baseada em CUDA e com versões nativas dos benchmarks, implementados diretamente em C++ utilizando as APIs OpenCL e CUDA. Todas as versões dos benchmarks utilizam as mesmas configurações de paralelismo (*grid*/*work-groups*/*threads*), para garantir uma comparação justa entre as diferentes plataformas.

Os resultados, apresentados na Tabela 1, demonstram um desempenho consistentemente superior da OCL-PolyHok em relação à PolyHok. Os tempos correspondem à média de 30 execuções consecutivas de cada programa em cada plataforma, e incluem o tempo de movimentação de memória (cópia de *arrays* entre GPU-CPU e vice-versa) e de geração e compilação dos kernels (no caso das DSLs).

Table 1. Resultados dos benchmarks. Tempos em milissegundos (ms).

Benchmark	OCL-PolyHok	PolyHok	Ganho (%)	OpenCL	CUDA
DP 400M	647.2	742.7	12,86%	576.4	514.6
DP 500M	788.4	868.3	9,20%	713.9	638.8
DP 600M	929.3	994.0	6,51%	861.2	764.5
JL 7.168×7.168	494.9	632.9	21,80%	492.1	503.9
JL 9.216×9.216	814.3	955.9	14,81%	812.6	831.9
JL 11.264×11.264	1.213.5	1.363.2	10,98%	1.213.3	1.241.7
MM 5k	254.2	404.2	37,11%	231.8	254.0
MM 7k	607.9	767.2	20,76%	606.6	665.6
MM 9k	1.285.2	1.403.6	8,44%	1.323.0	1.481.0

3.1. Análise dos Resultados

A superioridade da OCL-PolyHok em relação à versão CUDA decorre, majoritariamente, da eficiência do modelo de compilação Just-In-Time (JIT) do OpenCL, que apresenta menor latência do que a compilação JIT realizada pela biblioteca NVRTC (*NVIDIA Runtime Compiler*) utilizada originalmente na versão baseada em CUDA.

Um fenômeno relevante pode ser observado nos resultados do benchmark de Multiplicação de Matrizes (MM), onde as DSLs foram mais rápidas que suas versões nativas. Isso ocorre devido ao gerenciamento inteligente de memória realizado pela máquina virtual BEAM. A BEAM utiliza contagem de referências para reutilizar *buffers* de memória já instanciados, evitando o custo de alocação (*cold start*) enfrentado pelas implementações nativas – que alocam um novo *array* na memória para armazenar o resultado.

4. Conclusão

Este trabalho apresentou a OCL-PolyHok, uma versão da DSL PolyHok que utiliza o padrão OpenCL como plataforma de execução. A migração permitiu que a linguagem superasse a limitação de portabilidade da DSL original, tornando-a compatível em hardware de diferentes fabricantes sem exigir modificações em código PolyHok já existente. Os resultados experimentais demonstraram que a OCL-PolyHok não só atingiu seu objetivo de portabilidade, mas também apresentou desempenho superior à sua versão original baseada em CUDA em todos os benchmarks avaliados.

Como trabalhos futuros, pretende-se expandir as capacidades da DSL implementando recursos de distribuição de carga em sistemas com múltiplas GPUs, explorando a capacidade do OpenCL de gerenciar contextos com múltiplos dispositivos e o modelo de concorrência do Elixir.

References

Du Bois, A. R. and Cavalheiro, G. (2025). Polymorphic higher-order GPU kernels. In *Proceedings of the 31st International European Conference on Parallel and Distributed Computing*. Springer Nature.

Kolesnichenko, A., Poskitt, C. M., Nanz, S., and Meyer, B. (2015). Contract-based general-purpose gpu programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE'15, page 75–84. ACM.