Esqueletos de Paralelismo Map e Reduce Customizáveis com PolyHok

Frederico Peixoto Antunes, André Rauber Du Bois[1]

[1]Centro de Desenvolvimento Tecnológico(CDTec)
Universidade Federal de Pelotas(UFPel) – Pelotas – RS – Brazil

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

Resumo. *A crescente demanda por alto desempenho tem impulsionado o uso de abstrações que simplifiquem a programação paralela. Entre elas, os esqueletos de paralelismo encapsulam padrões recorrentes como Map e Reduce, que estruturam a aplicação de funções sobre conjuntos de dados segundo estratégias paralelas pré-definidas. Este trabalho expande o suporte destes esqueletos na DSL PolyHok, explorando otimizações e características configuráveis dos mesmos. A proposta busca conciliar expressividade e eficiência, comparando seu desempenho com implementações em CUDA puro.*

1. Introdução

Desde sua introdução comercial no final da década de 1990, as GPUs tornaram-se ubíquas em diferentes domínios computacionais, estando presentes não apenas em aplicações gráficas, mas também aplicações de propósito geral, como aprendizado de máquina, simulações científicas e processamento de grandes volumes de dados. Apesar do seu alto poder de processamento paralelo, a programação de GPUs ainda apresenta uma barreira significativa de entrada, exigindo conhecimento detalhado sobre gerenciamento de memória, hierarquia de execução e sincronização de threads, como observado em modelos de programação como CUDA.

Com o objetivo de reduzir essa complexidade, diversas abstrações foram propostas para facilitar o desenvolvimento de aplicações paralelas. Entre elas estando os esqueletos de paralelismo[Cole 1989, Marques et al. 2013, Du Bois and Cavalheiro 2025], funções de ordem superior que encapsulam padrões recorrentes de paralelismo. Esses esqueletos fornecem implementações previamente otimizadas de estruturas paralelas comuns, permitindo ao programador concentrar-se apenas na definição da função que será aplicada aos dados.

Nesse contexto, surge a PolyHok[Du Bois and Cavalheiro 2025], uma DSL desenvolvida para a linguagem Elixir que possibilita a definição de kernels de ordem superior para execução em GPU. A PolyHok permite que funções sejam passadas como parâmetros para kernels, ampliando significativamente o nível de abstração disponível para o programador.

Este trabalho apresenta a SkeLib, uma biblioteca de esqueletos de paralelismo para GPUs desenvolvida em PolyHok. A SkeLib define esqueletos baseados nas funções map e reduce, amplamente utilizadas na programação funcional. As funções map e reduce permitem a modelagem de uma ampla gama de problemas, incluindo aqueles definidos por indução estrutural.

2. PolyHok

A PolyHok introduz suporte a kernels de ordem superior na linguagem Elixir. No contexto de programação em GPU, um kernel corresponde a uma função executada no dispositivo. Tradicionalmente, linguagens como CUDA não permitem que kernels recebam funções como parâmetros de forma direta.

A PolyHok supera essa limitação ao permitir a definição de kernels que recebem funções como argumentos e que podem especializar seu comportamento com base nessas funções. Essa capacidade viabiliza a implementação de abstrações de mais alto nível, como esqueletos de paralelismo, mantendo a geração de código eficiente para execução na GPU.

Ao fornecer suporte a tipagem polimórfica e especialização dinâmica, a PolyHok cria o ambiente necessário para a construção de bibliotecas que encapsulam padrões paralelos reutilizáveis, como é o caso da SkeLib apresentada neste trabalho.

3. Skelib

A SkeLib é uma biblioteca desenvolvida sobre a PolyHok que disponibiliza esqueletos de paralelismo para execução em GPU. Seu objetivo é oferecer ao programador uma interface simples para definição de computações paralelas, abstraindo detalhes relacionados à geração e configuração de kernels CUDA.

A SkeLib disponibiliza atualmente os esqueletos Map e Reduce.

3.1. Map e Reduce Configuráveis

A chamada ao esqueleto Map segue a notação:

$$Ske.map(< gnx... >, < function > [, params...][, options]) \rightarrow gnx$$

Nessa notação, *gnx* representa uma ou mais matrizes alocadas na memória da GPU; *function* corresponde a uma função definida pelo usuário ou a uma função anônima; *params* é uma lista opcional contendo de um a três parâmetros adicionais a serem utilizados pela função; e *options* é uma lista opcional de opções que controlam o comportamento do esqueleto durante a execução.

A lista de opções admite os seguintes parâmetros:

$$[coord : [true|\underline{false}], return : [\underline{true}|false], dim : [\underline{: one}| : two]]$$

O parâmetro coord indica se as coordenadas do elemento processado devem ser passadas como argumento para a função. O parâmetro return define se o valor retornado pela função deve ser armazenado na estrutura de saída. Já dim especifica a dimensionalidade dos dados processados, permitindo distinguir entre coleções unidimensionais e bidimensionais.

O número de estruturas gnx fornecidas determina automaticamente a aridade da computação, permitindo a aplicação do Map sobre múltiplas coleções de dados.

A chamada do esqueleto Reduce, em comparação, pode ser representada como:

$$Ske.reduce(< gnx >, < function >) \rightarrow valor$$

onde ***function*** deve ser uma função associativa definida pelo usuário. O Reduce combina progressivamente os elementos da coleção até produzir um único valor resultante.

3.2. Implementação

Internamente, a SkeLib é responsável por traduzir as chamadas de alto nível para definições especializadas de kernels CUDA gerados por meio da PolyHok. A partir da função fornecida pelo usuário e das opções de configuração, a biblioteca realiza a especialização do kernel correspondente, definindo sua assinatura, parâmetros auxiliares e comportamento de retorno.

No caso do Map, a implementação gera um kernel no qual cada thread é responsável pelo processamento de um elemento (ou posição) da estrutura de dados, respeitando a dimensionalidade especificada. Já o Reduce utiliza uma estratégia clássica de redução paralela em GPU, baseada em etapas sucessivas de combinação parcial e sincronização entre threads até a obtenção do resultado final.

Dessa forma, a SkeLib mantém uma separação clara entre interface e implementação: o usuário interage apenas com abstrações funcionais de alto nível, enquanto a geração e otimização do código CUDA são tratadas automaticamente pela biblioteca.

4. Resultados

O conjunto de benchmarks usado consistente de dois programas: Dot Product e Nbodies. Ambos possuindo tanto uma versão implementada usando CUDA puro, quanto uma versão que utiliza os esqueletos Map e Reduce disponibilizados pela biblioteca SkeLib.

Tabela 1. Média do tempo de execução de 30 amostras da suíte de benchmark

	Entrada	CUDA (ms)	SkeLib (ms)	Slowdown
Dot Product	900000000	1040.306	1440.166	38.43%
Nbodies	7000	84171.783	89135.333	5.89%

Para validação do sistema os resultados apresentados Tabela 1 consistem da média calculada entres os tempos de execução resultantes das 30 execuções de cada programa. Ela também apresenta o percentual de slowdown observado com o uso da Skelib. Estes resultados foram obtidos a partir da execução em uma CPU AMD Ryzen 7 5700X 8-Core, GPU NVIDIA GeForce RTX 3060 12GB.

Os resultados indicam que a utilização da SkeLib introduz um overhead variável dependendo do perfil da aplicação. Em benchmarks de menor duração, como Dot Product, observa-se um slowdown médio em torno de 40%. Já no caso do Nbodies, cuja execução é significativamente mais longa, o slowdown reduz-se para aproximadamente 6%. Esse comportamento sugere que o custo adicional introduzido pela abstração tende a ser amortizado em aplicações com maior carga computacional, tornando-se relativamente pouco significativo em cenários de maior escala.

5. Trabalhos Relacionados

Diversas bibliotecas oferecem suporte a esqueletos de paralelismo para GPUs. Entre elas destacam-se: ***SkePU***[Enmyren and Kessler 2010], ***Marrow***[Marques et al. 2013] e ***Muesli***[Ciechanowicz et al. 2009, Ernsting and Kuchen 2012].

Essas bibliotecas disponibilizam implementações de esqueletos como Map, Reduce e variações relacionadas. Em geral, são desenvolvidas em C++ e baseadas em modelos como CUDA ou OpenCL, diferenciando-as da Skelib desenvolvida em Elixir.

A principal diferença da SkeLib em relação a essas abordagens está no ambiente de desenvolvimento e no nível de abstração oferecido. Enquanto as bibliotecas citadas operam diretamente em linguagens tradicionalmente voltadas à programação de sistemas, a SkeLib é construída sobre a PolyHok na linguagem Elixir, explorando suporte nativo a funções de ordem superior e polimorfismo. Além disso, a interface parametrizável da SkeLib permite customização explícita do comportamento dos esqueletos sem exigir alterações na sua implementação interna.

6. Conclusão

Os resultados obtidos demonstram que a SkeLib oferece um nível elevado de abstração para programação em GPU, simplificando a definição de padrões paralelos sem exigir conhecimento detalhado da infraestrutura CUDA. Embora a biblioteca introduza overhead perceptível em aplicações de curta duração, os experimentos indicam que esse custo tende a ser amortizado em aplicações com maior carga computacional, tornando-se relativamente reduzido em cenários de maior escala.

Como trabalhos futuros, pretende-se expandir o conjunto de benchmarks, avaliar diferentes tamanhos de entrada e estender a interface do esqueleto Reduce com novas opções de configuração, ampliando sua flexibilidade e potencial de otimização.

Referências

Ciechanowicz, P., Poldner, M., and Kuchen, H. (2009). The münster skeleton library muesli: A comprehensive overview. In J. Becker, K. Backhaus, H. L. G., editor, *Working Papers, European Research Center for Information Systems*.

Cole, M. I. (1989). *Algorithmic skeletons: structured management of parallel computation*. Pitman London.

Du Bois, A. R. and Cavalheiro, G. (2025). Polymorphic higher-order gpu kernels. In *European Conference on Parallel Processing*, pages 100–113. Springer.

Enmyren, J. and Kessler, C. W. (2010). Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Fourth International Workshop on High-level parallel programming and applications*, pages 5–14.

Ernsting, S. and Kuchen, H. (2012). Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138.

Marques, R., Paulino, H., Alexandre, F., and Medeiros, P. D. (2013). Algorithmic skeleton framework for the orchestration of gpu computations. In *Euro-Par 2013 Parallel Processing*, pages 874–885. Springer Berlin Heidelberg.