Kernel Fusion of Parallel Skeletons for GPU using Metaprogramming

João Antonio Soares[1], **André Rauber Du Bois**[1], **Gerson Geraldo H. Cavalheiro**[1]

[1] PPGC - Universidade Federal de Pelotas, Brazil - RS

{jansoares,dubois, gerson.cavalheiro}@inf.ufpel.br

Abstract. *Graphics Processing Units (GPUs) are the current generation of accelerators for high-performance computing (HPC). However, developing efficient applications fully utilizing the resources of the GPU is a time-consuming task. Algorithmic Skeletons help to simplify development by providing reusable patterns, but it often results in many independent kernel launches that are bounded by memory bandwidth, which otherwise could be fused to increase performance gains. In this work, we developed a fusion module that enables programmers to write composable GPU kernels that are efficiently merged. Our approach is applied to PolyHok, a domain-specific language (DSL) that exposes metaprogramming features enabling kernel fusion as high-level code transformations. The practical benefits of this solution are that it doesn't require modifying any existing compilers. We provide an open-source prototype implementation as well as an experimental evaluation of the feasibility of fusing skeletons.*

1. Introduction

To develop applications that efficiently leverage the full power of GPUs is considered a complex and error-prone task [Pérez et al. 2023]. Hence, algorithmic skeletons are higher-level abstractions for common parallel patterns that can hide low-level details allowing, developers to focus on how to express the problem [Cole 1989, Ernsting and Kuchen 2012].

To this end, PolyHok [Du Bois and Cavalheiro 2026] is a DSL implemented in Elixir and shares its metaprogramming features, enabling users to write *polymorphic high-order* GPU kernels using parallel skeletons. However, many applications require a composition of skeletons that, when are mapped to GPUs, are often split into many short operations. In such cases, only a small amount of computation is performed for each byte of data moved. Thus, the performance of a GPU application is limited by the bandwidth transfer rates of the system [Williams et al. 2009]. A common mitigation to this problem is to merge multiple kernels into a single monolithic one. This is referred to as kernel fusion, where consecutive stages of a computational pipeline are fused in a single kernel launch to minimize global memory traffic [Wahib and Maruyama 2014].

On the CUDA platform, libraries provide highly optimized fused kernels for specific domains. Nonetheless, developing and maintaining these kernels requires significant effort, which limits their availability to common use cases. In addition, using such libraries often requires large runtime dependencies, which may be excessive for simpler applications. Other previous works have studied reducing runtime dependencies, but the solution requires compiler modifications that are hard to maintain between CUDA versions [Fousek et al. 2011].

In this work, we propose the development of a prototype kernel fusion module for PolyHok [1]. Fusion is exposed as a *first-class* macro that can be enabled or disabled, allowing local composition and reuse as a building block for higher-level abstractions. Users can combine skeletons without dealing with architecture-level details, relying instead on their expertise to decide when fusion is valid and profitable. The main goal of this paper is to demonstrate the feasibility of performing fusion transformations without extending the language syntax or modifying the compiler.

2. Kernel Fusion Approach

Fusion of iterative parallel kernels is considered a complex task because it requires analyzing data dependencies and memory access patterns that are difficult to determine statically [Filipovič et al. 2015]. Usually, solutions targeting compilers adopt conversative constraints to select which kernel can be fused to avoid inconsistent memory access. Based on this observation, to be able to safely perform kernel fusion of GPU kernels, we delimit the scope of kernel fusion to algorithmic skeletons. We adopt the Bird-Meertens [Bird 1989] formalism common to functional programming to discuss parallel skeletons. The map skeleton is formally described as:

$$map(f)\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

which applies a high-order function to all elements of a given tensor. The result is a tensor of the same shape as the input applied to function f. The reduce skeleton:

$$reduce(\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

which apply a binary-associative function as operation to all elements of input tensor resulting in a low-order output tensor. These semantics allow kernel fusion to be expressed as algebraic program transformations. For example, two map skeletons can be composed into a single map. $(\text{map}\ f) \circ (\text{map}\ g) \equiv \text{map}\ (f \circ g)$. Internally, map and reduce skeletons are translated to grid-stride loops where each thread processes a tensor element.

Figure 1a exemplifies how a code block written in PolyHok can be optimized with kernel fusion. Users typically write a producer-consumer pipeline of map-reduce skeletons. Those skeletons share an intermediate tensor that can be eliminated by fusion. In Figure 1b, we express fusion by wrapping the pipeline of skeletons in a macro call. In Figure 1c, we demonstrate the resulting code after the expansion of the macro `withFusion`. In this example, the three map calls are fused into a single map call that receives an anonymous function that encapsulates the composition of f, g, and h.

The pipeline that performs fusion transformation is outlined in Figure 2. To start with, users write code using abstractions described in [Du Bois and Cavalheiro 2026]. The fusion process proceeds with a syntactic normalization by flattening the AST into a linear sequence of skeleton calls. This is essentially an additional step that desugars the AST received from PolyHok code to intermediate representation (IR) containing the skeleton operator tag (`map`, `map2`, `map3`, `reduce`), its explicit data arguments, and the body AST. Fusion itself is implemented as a bottom-up traversal of IR, building a new kernel declaration. Independent arguments passed to inner skeletons in the composition

[1] `https://github.com/JoaoNevesSoares/PolyHok`

```
require PolyHok.{Ske}

host_w = Ske.map(dev_x, f)
  |> Ske.map(g)
  |> Ske.map(h)
  |> PolyHok.get_gnx()
```

(a) Before fusion

```
require PolyHok.{Fusion, Ske}
host_w = PolyHok.withFusion(
  Ske.map(dev_x, f)
  |> Ske.map(g)
  |> Ske.map(h))
  |> PolyHok.get_gnx()
```

call to fusion macro

(b) Fusion

```
# Resulting kernel
Ske.map(
  dev_x, PolyHok.phok(
    fn x ->
      __fuse_in_1 = f(x)
      __fuse_in_2 = g(__fuse_in_1)
      return h(__fuse_in_2)
    end))
```

Renamed intermediates

(c) Fused

Figure 1. Example of fusion code transformation

are hoisted to the argument list of the resulting skeleton call. Then a new device function is constructed by inlining each function call of skeletons into a single anonymous function. If the skeleton receives a lambda function, variables are renamed to avoid naming conflicts. We then added explicit intermediate variables that act as carriers that bind the previous function output and become the first argument of the next function.

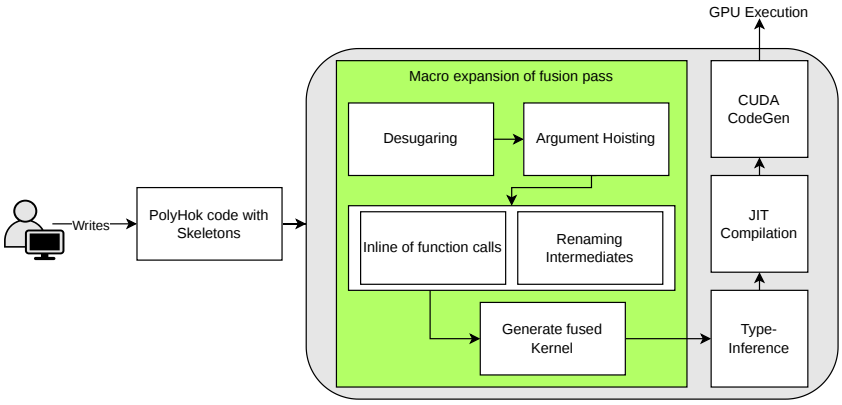

Figure 2. Polyhok architecture

Code generation depends on the last skeleton. If the chain contains only maps, we emit a single `Ske.map/map2/map3` call with the fused function. If a map chain ends with `reduce`, the compiler emits `Ske.mapReduce` (or `Ske.map2Reduce`). At runtime, the fused AST follows the standard PolyHok JIT path including type inference, lambda elimination, CUDA code generation and compilation with NVIDIA NVRTC.

3. Experimental Evaluation

To validate the proposed approach and obtain an initial indication of its performance, we conducted three experiments using a prototype implementation on a system equipped with a NVIDIA RTX 4060 8GB, using Linux Kernel 6.18, CUDA Toolkit 13.1 and Elixir 1.16. Table 1 shows the speedup of the fused kernel over the non-fused version. The

times presented for the execution of each instance of an application are the average of 30 executions. AXPY and Dot Product are characterized as level 1 BLAS operations over vector-to-vector, implemented using two maps and map and reduce, respectively. For numeric integration we used the Simpson's method with 1/3 rule to compute the definite integral of $\int_0^{\pi} \sin(x)\, dx$, consisting of three maps and one reduce skeleton.

Table 1. Experiment results

application (tensor size $n = 2^{22}$)	skeletons (before / after)	without fusion (ms)	with fusion (ms)	speedup ($\times$)
AXPY	2/1	926	436	2.12
Dot Product	2/1	968	524	1.84
Numerical Integration	5/2	2186	884	2.47

4. Conclusion

In this work, we presented the viability of kernel fusion using metaprogramming. We did so by extending a Skeleton DSL with fusion transformations at HIR. One advantage of this approach is that it doesn't require modifying the underlying language or the compiler. Future work will investigate the fusion of skeletons that takes *closure functions* as argument, i.e., functions that capture values from their lexical scope.

References

Bird, R. S. (1989). Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126.

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

Du Bois, A. R. and Cavalheiro, G. (2026). Polymorphic higher-order gpu kernels. In Nagel, W. E., Goehringer, D., and Diniz, P. C., editors, *Euro-Par 2025: Parallel Processing*, pages 100–113, Cham. Springer Nature Switzerland.

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.

Filipovič, J., Madzin, M., Fousek, J., and Matyska, L. (2015). Optimizing cuda code by kernel fusion: application on blas. *The Journal of Supercomputing*, 71(10):3934–3957.

Fousek, J., Filipovič, J., and Madzin, M. (2011). Automatic fusions of cuda-gpu kernels for parallel map. *SIGARCH Comput. Archit. News*, 39(4):98–99.

Pérez, V., Sommer, L., Lomüller, V., Narasimhan, K., and Goli, M. (2023). User-driven online kernel fusion for sycl. *ACM Trans. Archit. Code Optim.*, 20(2).

Wahib, M. and Maruyama, N. (2014). Scalable kernel fusion for memory-bound gpu applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202.

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.