User-Directed GPU Kernel Fusion of Parallel Skeletons Using Metaprogramming

João Antonio Soares
PPGC/CDTec
Universidade Federal de Pelotas
Pelotas, Rio Grande do Sul, Brazil
jansoares@inf.ufpel.edu.br

André Rauber Du Bois
PPGC/CDTec
Universidade Federal de Pelotas
Pelotas, Rio Grande do Sul, Brazil
ardubois@inf.ufpel.edu.br

Gerson Geraldo H. Cavalheiro
PPGC/CDTec
Universidade Federal de Pelotas
Pelotas, Rio Grande do Sul, Brazil
gerson.cavalheiro@inf.ufpel.edu.br

Abstract

Kernel fusion is an important optimization for GPU programs, as it reduces intermediate memory traffic and kernel launch overhead in producer-consumer computations. However, automatic fusion is difficult to integrate into high-level GPU frameworks, especially when kernels are generated dynamically. This paper presents a user-directed runtime kernel fusion mechanism for PolyHok, an Elixir-embedded DSL for GPU programming based on data-parallel skeletons. Programmers explicitly delimit candidate skeleton pipelines, while an Elixir macro performs source-to-source transformations that compose the corresponding computations and generate fused GPU kernels. The implementation does not require changes to the GPU compiler or to the Elixir runtime. An experimental evaluation with synthetic and application-level benchmarks indicates that the proposed mechanism can reduce intermediate memory traffic and improve performance in memory-bound workloads.

Keywords

Kernel Fusion, Metaprogramming, GPU Programming

1 Introduction

High-level Graphics Processing Unit (GPU) programming frameworks often expose reusable data-parallel patterns to reduce the complexity of GPU programming [1, 2, 7, 19, 25]. These abstractions let programmers build applications by composing higher-level primitives, rather than writing low-level GPU kernels directly. In practice, however, compositions of such patterns are commonly lowered into sequences of independent kernels. In producer-consumer pipelines, this execution model tends to introduce intermediate tensor allocations, additional global-memory traffic, and repeated kernel launches. As a result, application performance can become limited by memory transfer rates rather than computational throughput, creating a costly bottleneck in memory-bound workloads.

A common optimization strategy for GPU applications is kernel fusion [17], in which multiple kernels are merged into a single execution unit to reduce global memory traffic and kernel launch overhead. However, many GPU programming frameworks still lack effective mechanisms for performing kernel fusion. One challenge faced by these systems is that compilers often lack sufficient information to safely and effectively fuse independently generated kernels. In practice, GPU kernels are frequently compiled separately through static compilation pipelines, which introduces additional challenges such as pointer aliasing, indirect function calls, and limited visibility of higher-level program structure.

In this paper, we investigate the feasibility of a user-directed runtime kernel fusion mechanism. Our approach leverages Elixir metaprogramming capabilities to operate over the program's Abstract Syntax Tree (AST), enabling the composition of skeletons into fused GPU kernels. Consequently, the approach avoids the engineering and maintenance costs associated with compiler-dependent optimizations across toolchain versions. Furthermore, the proposed fusion mechanism does not introduce additional runtime dependencies beyond the existing PolyHok infrastructure, making the solution suitable for integration into other Elixir-based DSLs and applications. In summary, this work presents the following contributions:

- The development of a new fusion module capable of composing parallel skeletons into fused GPU kernels. The proposed solution targets memory-bound applications by reducing redundant global-memory accesses and intermediate memory allocations.
- A new language keyword (`Fusion.compose`) to delimit skeleton pipelines that should be considered in the source-to-source translation phase. The construct enables selective application of the transformation without requiring users to manually rewrite the underlying kernels.
- An experimental evaluation of the proposed solution using four benchmarks. We evaluate the performance gains of fusing producer-consumer kernel pipelines compared with non-fused execution. Experimental results demonstrate that runtime kernel fusion can be achieved with performance improvements for memory-bound GPU applications.

The remainder of this work is organized as follows. In Section 2, we present the fundamental concepts on which this work is built. Section 3 then presents the design of the proposed extension along with implementation details. In Section 4, we present the experiments comparing the performance of fused and non-fused benchmarks. Related work is discussed in Section 5, and conclusions are drawn in Section 6.

2 Background

In this section, we present some preliminary concepts on which the proposed solution is built.

2.1 GPU Execution Model

GPUs are SIMT (Single Instruction, Multiple Threads) processors in which programs are expressed through kernels, which are functions executed concurrently by many threads. The execution hierarchy is organized into threads, thread blocks, and grids. Threads within the same block can cooperate through shared memory and synchronization primitives, while multiple blocks compose a grid representing

the complete kernel launch. In CUDA-like programming models [20], each thread identifies the data it processes through built-in indexing variables such as `threadIdx`, `blockIdx`, and `blockDim`.

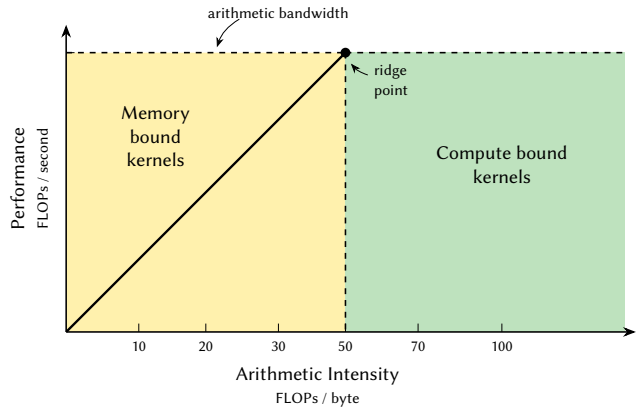


Figure 1: An example of a roofline model chart for a commercial RTX 4060 GPU. The yellow area indicates that kernel execution time is influenced by memory bandwidth, while the green area indicates that the kernel is limited by compute throughput. The ridge point indicates the minimum amount of computation required to avoid the memory wall (adapted from [27]).

GPUs have distributed memory with an explicit hierarchy that should be controlled by the programmer. At the fastest level, each thread has private registers that store temporary values during kernel execution. Registers are considered a critical resource and can directly impact performance if a kernel consumes too many of them. Shared memory is part of the SM's L1 cache and should also be explicitly managed by programmers. Shared memory can be used to communicate data between threads within the same thread block. Finally, global memory can be accessed by all threads from all thread blocks, although it is considerably slower than registers and shared memory.

One common model used to analyze the behavior of a GPU program is the Roofline Model [27], shown in Figure 1. The Roofline Model relates the achieved performance of a kernel to its arithmetic intensity, defined as the ratio between floating-point operations and bytes transferred from memory.

These different levels of GPU memory should be considered when implementing kernel fusion. In fused kernels, registers and shared memory are employed as temporary storage for intermediate computations, allowing data reuse and reducing unnecessary accesses to global memory.

2.2 PolyHok

In previous work [6, 10, 11], we presented *PolyHok*, a DSL embedded in the Elixir ecosystem. It allows Elixir developers to write low-level, imperative code that runs on GPU hardware. For this purpose, PolyHok introduces the idea of higher-order kernels. These are polymorphic procedures that run entirely on the GPU. The kernels written in PolyHok, such as the one presented in Listing 1, have

```
1  require PolyHok
2
3  PolyHok.defmodule Ske do
4      defk map_ker(a1, a2, size, f) do
5          tid = blockIdx.x * blockDim.x + threadIdx.x
6          str = blockDim.x * gridDim.x
7          for i in range(tid, size, str) do
8              a2[i] = f(a1[i])
9          end
10     end
11 end
```

Listing 1: A GPU kernel for a `map` skeleton in PolyHok

their types determined at runtime through on-the-fly type inference. They are also just-in-time (JIT) compiled into back-end code and are highly configurable through launch parameters such as the number of threads and blocks.

Additionally, PolyHok adopts several functional programming features to provide high expressiveness and ease of use, such as dynamic typing and automatic memory management. Due to its strong integration with Elixir, PolyHok can take advantage of the host language's macro capabilities to implement its core features; for example, the existing language constructs are defined as Elixir macros that operate directly on the program's Abstract Syntax Tree, avoiding a separate parser usually required by a standalone DSL.

$e ::= v$	(constant)
$\mid$ id	(identifier)
$\mid$ id$[e_1, \ldots, e_n]$	(array indexing)
$\mid$ id$(e_1, \ldots, e_n)$	(function call)
$\mid e_1$ op e_2	(binary operation)
$s ::=$ id $= e$	(assignment)
$\mid$ id$[e] = e$	(array update)
$\mid e$	(expression statement)
$b ::= s_1, \ldots, s_m$	(statement sequence)
$kern ::=$ `defk` id$(\text{id}_1, \ldots, \text{id}_n)$ `do` b `end`	(kernel declaration)
$fun ::=$ `phok(fn` $\text{id}_1, \ldots, \text{id}_n$ `->` b `end)`	(anonymous device function)
$\mid$ `defd` id$(\text{id}_1, \ldots, \text{id}_n)$ `do` b `end`	(named device function)
$d ::= kern \mid fun$	(PolyHok declaration)
$m ::=$ `defmodule` id `do` $d_1, \ldots, d_n$ `end`	(module declaration)

Figure 2: Core PolyHok syntax relevant to this work.

PolyHok adopts a CUDA-like imperative style for kernel code. It also provides specific constructs for declaring kernels and device functions. The `defk` construct defines a GPU kernel, whereas `defd` and `phok` define named and anonymous device functions, respectively.

Data that resides in the host system must be transferred to device memory before a GPU kernel can access it. For this purpose, PolyHok provides the `new_gnx` function, which converts an `Nx` tensor into a GPU-resident `GNx` value that can be passed to kernels. The data associated with each allocation is managed automatically and released when the GNx value is garbage-collected. We point out that the current implementation supports the element types `float32`, `float64`, and `i32`.

2.3 GPU Skeletons

We delimit the scope of kernel fusion to data parallel skeletons that can be composed to express GPU programs. All skeletons have monomorphic types, meaning that we assume their arguments have the same type. The current implementation supports the Parallel Map and the Parallel Reduce skeletons.

2.3.1 ***Parallel*** `Map`. The `map` pattern is a higher-order parallel pattern that applies a function independently to all elements of an input tensor. We adopt the Bird-Meertens formalism [5], where `map` can be formally defined as follows:

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

where f is a side-effect-free function applied element-wise to the input domain. Since each iteration is independent, `map` skeletons naturally expose data parallelism that can be explored by the underlying GPU device runtime.

Additionally, higher-arity variants such as `map2`, `map3`, and `mapN` are internally represented as zip-based skeletons operating over multiple tensors simultaneously. For example, `map2` can be formally described as follows:

$$\texttt{map2}(f, [a_1, \ldots, a_n], [b_1, \ldots, b_n]) = [f(a_1, b_1), \ldots, f(a_n, b_n)]$$

where corresponding elements from multiple tensors are combined pointwise through the user-defined function f. More generally, `mapN` extends these semantics to N-ary pointwise operations over tensors with compatible shapes.

PolyHok additionally extends zip semantics to support scalar broadcasting for literal scalar values. In this model, operands passed to `mapN` may be scalar constants instead of tensors. Scalars are implicitly broadcast across the iteration domain without requiring explicit tensor materialization by the programmer. For example:

$$\texttt{map2}(f, [a_1, \ldots, a_n], c) = [f(a_1, c), \ldots, f(a_n, c)]$$

where c is treated as an invariant scalar reused across all iterations. This design avoids allocating replicated scalar tensors in global memory and eliminates unnecessary memory operations associated with loading broadcast values from tensor storage. Instead, literal scalar values can be directly propagated as invariant kernel parameters or compile-time constants during code generation and fusion. Similar behavior is found in other programming languages [15]. In `Futhark`, scalar expansion is represented through syntactic sugar such as `map (\x -> x + c) xs`, where the compiler avoids materializing replicated tensors and instead fuses `map` operations with replicate-like semantics during optimization passes.

2.3.2 ***Parallel*** `Reduce`. The parallel `reduce` behaves similarly to the parallel `map` but performs a reduction operation:

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

This operation applies a binary associative function to all elements of an input tensor, producing a lower-order output tensor. In the current implementation, the parallel `reduce` skeleton uses a tree-based reduction in shared memory [14].

Users can provide the reduction operator as a device function, that is, a function compiled for execution on the GPU and invoked from a GPU kernel. PolyHok accepts three forms of device functions: named functions, anonymous functions, and closures that capture values from their lexical scope.

2.3.3 ***Representing Other Parallel Patterns As Compositions Of Map And Reduce.*** The combination of these parallel patterns provides a rich set of abstractions that users can employ to describe parallel computations. Their well-defined semantics also enable algebraic program transformations, which can be exploited for optimizations such as kernel fusion. For instance, two `map` skeletons can be composed into a single `map`:

$$(\texttt{map}\ f) \circ (\texttt{map}\ g) \equiv \texttt{map}\ (f \circ g).$$

`map` and `reduce` skeletons translate into grid-stride loops, where each thread processes one or more tensor elements.

2.4 Kernel Fusion

PolyHok follows the skeleton programming model, in which each skeleton is lowered into a GPU kernel. Kernels are generated dynamically at runtime and compiled just in time. Consequently, each skeleton invocation within a composition is translated into an independent GPU kernel, fragmenting pipelines of parallel patterns across multiple kernel launches and compilation units.

Kernel fusion is a widely used optimization technique for GPU applications that addresses this fragmentation problem by merging multiple kernels into a single execution unit. The main advantage of kernel fusion in our context is the reduction of fragmented execution caused by producer-consumer compositions of skeletons. Combining multiple kernels into a single generated kernel can reduce the runtime overhead associated with launching successive kernels while also increasing opportunities for instruction-level parallelism and improving intra-kernel warp scheduling.

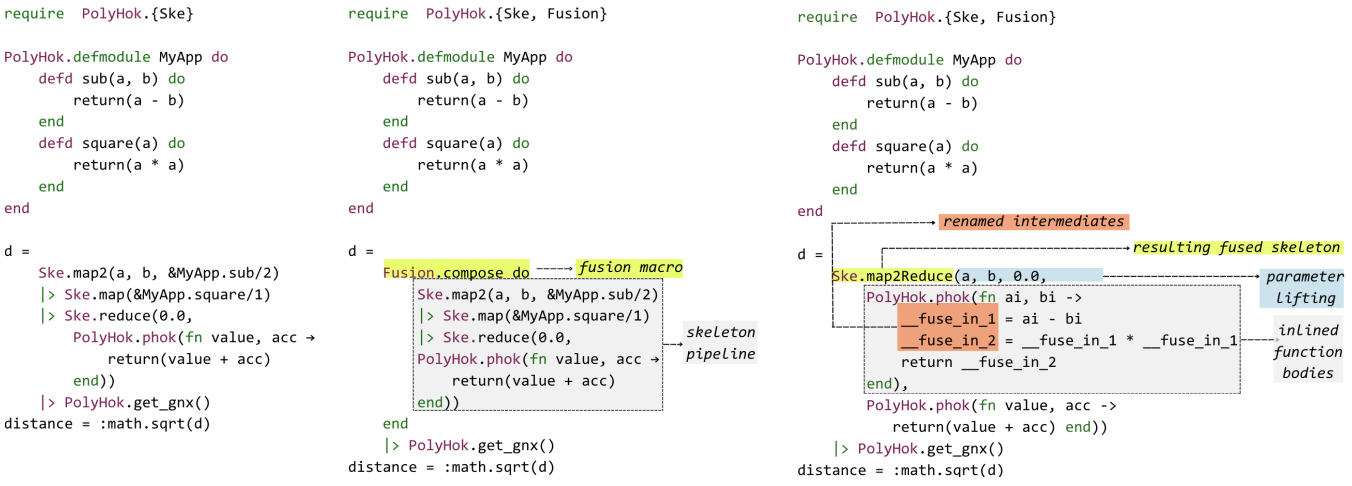
Another important benefit of kernel fusion is the elimination of intermediate memory accesses between producer-consumer stages. In many data-parallel applications, consecutive kernels repeatedly perform a read-compute-write pattern over the same tensor elements. This optimization is commonly referred to as vertical kernel fusion, in which the intermediate values produced by one kernel are directly consumed by the next stage without requiring additional memory transfers. In PolyHok, this pattern emerges naturally in compositions of skeletons, making kernel fusion a suitable optimization for improving the execution of these pipelines. The next section discusses the design considerations of the proposed mechanism.

3 Fusion: A Transformation Over Skeleton Composition

This section presents the proposed fusion mechanism and discusses the main stages involved in the transformation process.

3.1 A motivating Example For Fusing Skeletons

A typical vector operation in many statistical and machine learning applications is the computation of the Euclidean distance in a vector space, expressed as $d(A, B) = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}$, where A and B are two one-dimensional vectors of floating-point elements. Using the skeletons available in PolyHok, a user may break down this computation into three individual skeletons, as shown in Figure 3a. This naive implementation makes the code very easy to read and



(a) Before fusion **(b) Fusion request** **(c) Fused result**

Figure 3: Transformation of a skeleton composition into a fused `map2Reduce` skeleton.

understand and also allows some of the operations to be reused in other computations, increasing the modularity of the implementation. However, without fusion, the vectors propagated between stages must be synchronized through global memory. Since this application executes only one or two floating-point operations per data element read, its performance will be limited by the overhead of memory movement.

Instead of rewriting the computation as a more specialized skeleton or writing a specific kernel, users can call a keyword macro that composes the participating skeletons into a single optimized kernel. This transformation reduces memory accesses and improves overall efficiency without sacrificing expressiveness or modularity. In the following sections, we explain the steps taken by the mechanism to capture the skeleton pipeline from the user's program, parse and validate the code, perform the metaprogramming transformation, and generate the resulting code.

3.2 Capture of Skeleton Chains

The proposed mechanism is exposed through the `Fusion.compose` macro. Programmers use this construct to explicitly delimit a sequence of skeleton invocations that should be considered for fusion. As illustrated in Figure 3b, an existing skeleton pipeline can usually be adapted by enclosing it in a `Fusion.compose` block, without modifying its skeleton definitions or device functions.

Inside the fusion region, producer-consumer dependencies are expressed through the Elixir pipe operator (`|>`). The operator forwards the result of one skeleton invocation as the first input of the next invocation. It therefore describes an ordered chain in which each stage directly consumes the value produced by its predecessor. The first skeleton receives all its inputs explicitly, whereas subsequent skeletons omit the input supplied through the pipe.

This explicit representation makes the producer-consumer dependencies available directly in the program syntax. Consequently, the macro can capture the fusion region as a linear sequence of skeleton calls, without reconstructing dependencies from arbitrary variable definitions and uses or constructing a general dataflow graph. Programmers remain responsible for selecting candidate regions, while the fusion module performs the subsequent validation, transformation, and code generation steps.

3.3 Parsing And Structural Validation Of Skeleton Chains

The next step starts when the Elixir compiler expands the `compose` macro. At this point, the fusion front-end receives an AST containing the call graph in tuple format; for example, a node in that AST has the following structure:

```
{:|>, [],
  [
    {:|>, [],
      [
        call(Ske.map2, [a, b, f]),
        call(Ske.map, [g])
      ]},
    call(Ske.reduce, [0.0, h])
  ]}
```

The parser recursively traverses these nodes and converts them into an ordered sequence of skeleton stages,

$$C = [S_1, S_2, \ldots, S_n].$$

This sequence preserves the order of the original pipeline. The first stage receives all its inputs explicitly, whereas each subsequent stage receives the value produced by its predecessor through the pipe operator. Since this producer-consumer dependency is encoded in the syntax, the front-end can recover the dataflow as a linear sequence without reconstructing dependencies from arbitrary variable definitions and uses or building a dataflow graph.

Each stage in C is represented by an `AstCall` structure that stores the skeleton operator, the AST of its device function, its explicit inputs, and metadata required by the subsequent fusion passes. A simplified version of this representation is shown below:

```
1 defstruct [
2   :ske,
3   :kernel_ast,
4   explicit_inputs: [],
```

```
  initial_ast: nil,
  piped_input?: false,
  stage_kind: :map,
  terminal?: false,
  output_kind: :tensor
]
```

Listing 2: Internal representation of a skeleton stage.

This representation distinguishes inputs written explicitly in a skeleton call from the input supplied by the preceding pipe stage. It also records reduction initial values and whether a stage produces a tensor or a scalar. The parser currently recognizes `map`, `map2`, `map3`, `map4`, and `reduce`, and it converts their different standalone and piped call forms into the same stage-level representation.

After parsing, the front-end performs structural validation over the ordered sequence. The first stage must use a standalone call form, subsequent stages must use their piped forms, and the number of explicit inputs must match the arity expected by each skeleton. The validation also rejects unsupported skeletons, malformed calls, multiple reductions, and reductions that do not occur at the end of the chain. This phase does not analyze tensor dimensions, aliasing, side effects, or algebraic properties of reduction operators.

3.4 Fusion Of Skeleton Pipelines

We model fusion as a sequence of local transformations over an ordered producer-consumer chain. Let $A_1, \ldots, A_k$ be tensors of equal length n. A k-ary map skeleton is defined as

$$\mathrm{map}_k(f; A_1, \ldots, A_k) = \left[f(a_{1,i}, \ldots, a_{k,i}) \mid 1 \le i \le n \right].$$

The simplest fusion case consists of two unary maps. Given an input tensor A, the first stage applies f to each element and produces an intermediate tensor that is subsequently consumed by $\mathrm{map}(g)$. Since both operations are pointwise, the intermediate tensor can be eliminated by composing their functions:

$$\mathrm{fuse}\left(\mathrm{map}_1(g) \circ \mathrm{map}_1(f)\right)(A) = \mathrm{map}_1(g \circ f; A),$$

where

$$(g \circ f)(x) = g(f(x)).$$

For an input $A = [a_1, \ldots, a_n]$, both sides produce

$$[g(f(a_1)), \ldots, g(f(a_n))].$$

The equality therefore follows from the pointwise semantics of `map`. The unfused composition materializes $[f(a_1), \ldots, f(a_n)]$ between the two stages, whereas the fused form computes $g(f(a_i))$ directly for each element.

The same transformation applies when a consumer receives additional tensor or scalar inputs. Let $\vec{A}$ denote the inputs of a producer $\mathrm{map}_p(f)$, and let $\vec{B}$ denote the additional explicit inputs of a consumer $\mathrm{map}_{q+1}(g)$. The producer output occupies the first logical input of the consumer. The local fusion rule is

$$\mathrm{map}_{q+1}\left(g; \mathrm{map}_p(f; \vec{A}), \vec{B}\right) \Rightarrow_F \mathrm{map}_{p+q}\left(h; \vec{A}, \vec{B}\right),$$

where the fused function h is defined by

$$h(\vec{a}, \vec{b}) = g(f(\vec{a}), \vec{b}).$$

For example,

$$\mathrm{map}_2\left(g; \mathrm{map}_1(f; A), B\right) \Rightarrow_F \mathrm{map}_2(h; A, B),$$

with

$$h(a, b) = g(f(a), b).$$

This rule corresponds directly to a pipeline such as

```
Ske.map(A, f)
|> Ske.map2(B, g)
```

because the value produced by the first stage becomes the first logical input of the second stage. A chain of map-like skeletons is fused by repeatedly applying this local rule. Consider a pipeline

$$C = S_1 \mid > S_2 \mid > \cdots \mid > S_m,$$

where every S_j is a map-like skeleton. Let $\vec{X}_1$ denote all explicit inputs of the first stage and let $\vec{X}_j$, for $j > 1$, denote the additional explicit inputs of stage j. The composed pointwise function is defined recursively as

$$F_1(\vec{x}_1) = f_1(\vec{x}_1),$$

and

$$F_j(\vec{x}_1, \ldots, \vec{x}_j) = f_j\left(F_{j-1}(\vec{x}_1, \ldots, \vec{x}_{j-1}), \vec{x}_j\right), \qquad 2 \le j \le m.$$

Fusion then rewrites the complete chain as

$$\mathrm{fuse}(C) = \mathrm{map}_r\left(F_m; \vec{X}_1, \ldots, \vec{X}_m\right),$$

where r is the number of external inputs referenced by the original pipeline. Repeated external inputs may be represented by the same parameter in the generated skeleton. This recursive definition makes the transformation reproducible: each stage function receives the expression generated by its predecessor as its first logical argument, together with its remaining explicit inputs.

3.4.1 Composition Of Reduce Skeletons. Pipelines ending in a reduction are handled as a terminal case. Let $\oplus$ be an associative binary operator with identity element e. The reduction of a tensor $A = [a_1, \ldots, a_n]$ is defined as

$$\mathrm{reduce}^e_{\oplus}(A) = e \oplus a_1 \oplus \cdots \oplus a_n.$$

After the preceding map stages have been composed into a function F, a map-reduce pipeline is transformed according to

$$\mathrm{reduce}^e_{\oplus}\left(\mathrm{map}_r(F; \vec{A})\right) \Rightarrow_F \mathrm{mapReduce}_r\left(F, \oplus, e; \vec{A}\right).$$

The semantics of the resulting skeleton is

$$\mathrm{mapReduce}_r\left(F, \oplus, e; A_1, \ldots, A_r\right) = e \oplus \bigoplus_{i=1}^{n} F(a_{1,i}, \ldots, a_{r,i}).$$

Unlike the functions of consecutive map stages, the reduction function is not incorporated into the pointwise mapping function. Instead, the composed function F produces values consumed by the reduction stage, without materializing the mapped tensor in global memory. A reduction is therefore terminal, and a fusible chain may contain at most one reduction as its final stage.

3.4.2 Discussion. The transformation can also be interpreted as a restricted producer-consumer graph reduction. Each skeleton is a node, and each piped value defines an edge from a producer to its consumer. Since the accepted pipelines are linear, every intermediate result has a single consumer. Fusion contracts these edges by replacing the consumer's piped parameter with the producer expression. The process terminates when the chain has been reduced to one `mapN` skeleton or one terminal `mapNReduce` skeleton.

These equivalences require tensors with compatible iteration domains and pointwise device functions without observable side effects. Map-reduce fusion additionally requires an associative reduction operator and a valid identity element. The current model does not prove tensor compatibility, absence of aliasing, device-function purity, or fusion profitability. These properties are assumed or handled by the existing PolyHok runtime.

3.5 Code Generation

Once a valid fusion pattern has been identified, the back-end constructs a fused skeleton through a source-to-source transformation. The transformation traverses the sequence of skeleton stages in producer-consumer order and incrementally builds a new device function corresponding to the composition of the original stage functions. Figure 3c illustrates the resulting fused code for the Euclidean-distance pipeline.

The generation process applies four main transformations:

- **Function composition:** the result produced by each stage is propagated as the piped input of the subsequent stage.
- **Parameter lifting:** external tensor and scalar inputs referenced by the stages are added to the parameter list of the resulting skeleton.
- **Function inlining:** the bodies of the original device functions are inserted into the generated function in producer-consumer order.
- **Intermediate renaming:** stage-specific names are assigned to intermediate and local values, providing SSA-like name uniqueness without constructing SSA form or a control-flow graph.

Because type inference and back-end compilation operate on the expanded AST, some errors may reference generated names. The current prototype reports structural errors on the original pipeline, but full source-level remapping of later diagnostics remains future work.

3.6 Fusion Constraints and Extension Opportunities

Our current implementation operates only on elements of one-dimensional tensors. Support for matrix operations and other skeletons could be added by extending the fusion module, especially its ad hoc intermediate representation and validation phase. The representation could record the computational class of each skeleton, such as pointwise, stencil, reduction, or matrix-tiling, together with the properties required by that class. Matrix computations could use specialized skeletons that iterate over two dimensions, or future work could investigate nested parallelism, allowing one skeleton to invoke another.

To support skeletons with different computation patterns, validation could be organized as a collection of rewrite rules, with each rule specifying the conditions under which a transformation is valid and how the corresponding skeletons should be composed.

Another limitation not addressed in this paper is the restriction to linear producer-consumer chains expressed through `|>`. A future extension could accept restricted blocks with skeleton calls and variable bindings, allowing intermediate results to be reused by non-adjacent stages. Such computations could be represented as dataflow graphs composed of single-output skeleton nodes and analyzed using T2-style graph-reduction techniques [16].

The current implementation does not estimate profitability from register or memory pressure. Register use is difficult to predict from the AST because it depends on CUDA back-end optimizations and the target GPU. A future version could use simple front-end heuristics, then inspect compiled-kernel attributes to estimate resource usage and occupancy.

4 Experimental Evaluation

This section presents an experimental evaluation using one synthetic benchmark and four application-level benchmarks. The synthetic benchmark isolates the effect of vertical fusion in a controlled producer-consumer pipeline, while the application benchmarks assess its behavior in more realistic workloads with different computational and memory access patterns.

4.1 Experimental Methodology

The experiments were conducted on a dedicated system equipped with an *NVIDIA* `RTX 4060 8GB` GPU and running the `Linux` kernel 6.18, `CUDA Toolkit` 13.1, and `Elixir` 1.16. During the experiments, the machine was reserved exclusively for benchmarking, and unnecessary background services were disabled to reduce measurement interference.

For each experiment, we performed 30 independent runs, alternating between the versions being compared. For the `map`-based experiments, we report performance in terms of GFLOPs and memory bandwidth (GB/s). For the application benchmarks, we report the average execution time in milliseconds, including data transfers between host and device as well as kernel execution time. To support further analysis, we instrumented the PolyHok back-end to emit the generated `CUDA` kernels as text files, enabling inspection of the generated code and profiling with `CUDA` tools. Our primary comparison is between PolyHok applications implemented with fusion and those implemented without fusion. For the application benchmarks, we evaluated three input sizes for each application. Experimental artifacts, including benchmark scripts, raw measurements, and generated `CUDA` files, are available in a public repository, as described in the Artifact Availability section.

4.2 Experimental Results

The evaluation first isolates the effects of fusion in synthetic experiments and then assesses its behavior in application-level benchmarks.

4.2.1 ***Number Of Vertically Fused Parallel Map Operations.*** As a first experiment, we evaluated the impact of increasing the

Table 1: Results from increasing the number of vertically fused `map` skeletons

# Maps	unfused		fused		Speedup
	GFLOP/s	GB/s	GFLOP/s	GB/s	
2	0.28	2.07	0.48	1.93	1.82x
3	0.26	2.04	0.67	1.81	2.60x
5	0.27	2.16	1.10	1.76	4.09x
10	0.29	2.34	2.20	1.76	7.52x
15	0.30	2.35	3.27	1.74	11.23x
20	0.32	2.54	4.19	1.68	14.19x

number of fused `map` stages. We constructed synthetic producer-consumer pipelines containing between two and twenty consecutive `map` skeletons. The input size was the same for every run: 100 million floating-point elements. Each stage performs a simple arithmetic operation, allowing the experiment to isolate the effects of intermediate tensor elimination from application-specific behavior. The objective is to evaluate how the benefits of fusion scale as longer producer-consumer chains are combined into a single kernel.

Table 1 summarizes the results. The speedup increases consistently with the number of fused stages, ranging from 1.82× for two `map` operations to 14.19× for twenty `map` operations. This trend is expected because each additional fused stage eliminates an extra intermediate tensor and the corresponding global-memory reads and writes. Consequently, the amount of memory traffic avoided by fusion grows proportionally with the length of the producer-consumer chain. The generated `CUDA` code was inspected to confirm that the fused versions do not allocate intermediate tensors between producer-consumer stages and that intermediate values are propagated as local expressions inside the generated kernel.

To better understand these results, we analyzed the generated `CUDA` kernels using *NVIDIA* `Nsight Compute`. Contrary to a common concern regarding kernel fusion, the larger fused kernels did not increase register pressure. In fact, the number of registers per thread decreased from 32 to 29, while the achieved occupancy increased from 93.10% to 97.78%. These results indicate that the fusion process did not introduce resource constraints capable of reducing GPU residency. One possible explanation is that the `CUDA` compiler was able to perform additional optimizations over the larger composed functions, such as replacing arithmetic sequences with fused multiply-add instructions (`FMADD`), reducing the number of temporary values required during execution.

The profiling results also confirm the expected shift from a memory-bound execution profile toward a more compute-oriented one. The unfused version reaches 16.47% of peak DRAM throughput, whereas the fused kernel requires only 2.56%. At the same time, SM throughput increases from 28.0% to 57.32%. These observations are consistent with the Roofline discussion presented in Section 2: by eliminating intermediate memory transfers, fusion reduces pressure on the memory subsystem and allows a larger fraction of execution time to be spent performing useful computation.

Table 2: Results of Applying Kernel Fusion (Values are reported in milliseconds)

Benchmark	Input Size	Unfused	Fused	Speedup
N-Body	2.5×10^5	1829.0	1760.0	1.02
	5.0×10^5	6802	6746.0	1.01
	1.0×10^6	27192.5	26952.2	1.00
Nearest Neighbor	1.0×10^8	283.3	250.1	1.13
	2.0×10^8	437.7	398.5	1.09
	4.0×10^8	751.3	730.4	1.03
Portfolio Pricing	1.0×10^7	39.1	23.2	1.71
	2.0×10^7	45.7	25.3	1.81
	4.0×10^7	63.4	34.5	1.84
Numeric Integration	1.0×10^6	92.0	40.0	2.32
	2.0×10^6	107.4	50.1	2.14
	4.0×10^6	90.5	42.2	2.13

4.2.2 ***Impact Of Data Types On Fusion Benefits.*** This experiment evaluates whether arrays with different element types can affect the effectiveness of kernel fusion. We evaluated a synthetic pipeline composed of three consecutive `map` operations, `map(input, mult) |> map(divf) |> map(sub)`, using an input tensor containing 100 million elements. Three different data types were considered: `f32` (single-precision floating-point), `f64` (double-precision floating-point), and `s32` (32-bit integer).

The experimental results showed that the integer version achieved a speedup of approximately 1.30× after fusion, while the `f32` version achieved a similar speedup of 1.28×. In contrast, the `f64` version achieved a smaller speedup of only 1.17×. One possible explanation is that the `s32` and `f32` versions are predominantly memory-bound, meaning that a significant fraction of execution time is spent transferring intermediate arrays through global memory. Since kernel fusion eliminates these intermediate reads and writes, the impact on performance is more pronounced. For the `f64` version, however, the computational cost of arithmetic instructions becomes more relevant due to the lower double-precision throughput typically available on consumer GPUs. As a result, the workload becomes relatively less memory-bound and more compute-bound, reducing the relative benefit obtained from eliminating global-memory traffic through fusion. These results indicate that the benefit of fusion is more pronounced when intermediate memory traffic dominates execution time.

4.2.3 ***Application-Level Evaluation.*** We implemented four applications and optimized them using the proposed fusion module. These benchmarks exercise different fusion scenarios, including `map-map` pipelines, `map-reduce` compositions, and applications with limited fusible regions. Table 2 summarizes the main statistics obtained from the experiments.

N-Body simulation: This benchmark computes the Newtonian gravitational interaction among a set of bodies in three-dimensional space. The implementation consists of three kernels. The first kernel computes the force exerted on each body by all other bodies, while the remaining two kernels update acceleration, velocity, and

position through pointwise operations implemented as `map2` skeletons. Fusion was applied only to the latter two stages, resulting in a fused `map3` skeleton.

As shown in Table 2, fusion yielded only marginal improvements, with speedups ranging from 1.00× to 1.02×. Profiling revealed that approximately 80% of the execution time was spent in the force-computation kernel, which cannot be fused under the current model. Consequently, the optimizable portion of the application represented only a small fraction of the execution time. This result highlights an important limitation of kernel fusion: applications dominated by compute-bound kernels with little intermediate memory traffic offer limited opportunities for performance improvement.

Nearest Neighbor: This benchmark was adapted from the *Rodinia* benchmark suite [9]. The implementation consists of a producer consumer composition between a `map` skeleton, which computes Euclidean distances from each record to a target position, and a `reduce` skeleton, which selects the minimum distance.

Fusion yielded speedups between 1.03× and 1.13×, with larger gains observed for smaller inputs. Although the application is memory-bound, the resulting fused kernel inherits conflicting execution requirements from the `map` and reduction stages. The `map` operation benefits from a large number of thread blocks to maximize parallelism, whereas the reduction stage becomes increasingly sensitive to synchronization and atomic update overheads as the number of blocks increases. Consequently, a single fused configuration cannot simultaneously optimize both stages, reducing the overall gains obtained through fusion.

Portfolio Pricing: This benchmark was adapted from [4] and implements a portfolio valuation workflow based on the Black-Scholes model [22]. The application is composed of three consecutive `map` skeletons that compute the fair value of each option, apply portfolio weights, and compute the contribution of each asset.

Fusion combined the three stages into a single kernel, achieving speedups ranging from 1.71× to 1.84×. Since all stages perform relatively simple arithmetic operations over large tensors, the unfused implementation spent a considerable fraction of its execution time transferring intermediate results through global memory. Eliminating two intermediate tensors significantly reduced memory traffic, explaining the substantial performance improvements observed across all input sizes.

Numeric Integration: This benchmark was adapted from [23] and computes definite integrals using Simpson's 1/3 rule. Following the structure described in the original work, the implementation consists of five kernels organized as a producer-consumer pipeline.

Fusion reduced the pipeline from five kernels to two, eliminating three intermediate tensors. As shown in Table 2, this benchmark achieved the highest speedups, between 2.13× and 2.32×. The result is consistent with the synthetic experiments presented previously: longer producer-consumer chains provide more opportunities to eliminate intermediate memory transfers. Consequently, the benefits of fusion become more pronounced as the number of removable intermediate stages increases.

5 Related Work

In the literature, several works have investigated the benefits of kernel fusion for GPU applications. Early studies such as [12] and [13] explored fusion in the `CUDA` programming model, demonstrating performance improvements by reducing intermediate accesses to global memory and lowering kernel launch overhead. Similarly, [26] evaluated the impact of kernel fusion on the energy consumption of GPU applications. While these studies demonstrate the benefits of fusion, they primarily focus on manually optimized `CUDA` kernels rather than high-level programming abstractions.

Kernel fusion is also adopted by many domain-specific languages and parallel programming frameworks. `Accelerate` [8], `Futhark` [15], and `Triton` [25] provide mechanisms for combining computations and eliminating intermediate arrays. In particular, `Futhark` employs uniqueness types and a rich fusion algebra to enable aggressive compiler-driven optimizations over compositions of parallel patterns. These systems typically rely on static analyses and compiler transformations to identify fusion opportunities automatically. In contrast, our work exposes fusion as an explicit programming abstraction and performs transformations through metaprogramming at the application level, without requiring modifications to the compiler infrastructure.

Several compiler-oriented approaches have also investigated automatic fusion strategies for heterogeneous systems. Works such as [21, 24, 28] use static program analyses to identify fusion opportunities and generate optimized kernels. These approaches assume that sufficient information is available during compilation and that the compiler has visibility over the program structure. Our work addresses a different setting, namely a dynamically generated GPU programming environment in which kernel compositions are available only at runtime and are transformed through metaprogramming.

The work of Amoros et al. [3] is related to ours because it also exploits metaprogramming techniques to construct fused GPU kernels. However, their approach focuses on composing iterative GPU library operations using C++ template metaprogramming, whereas our work exploits the semantics of parallel skeletons to compose producer-consumer pipelines expressed in a high-level DSL. Additionally, [18] investigates horizontal kernel fusion, while our work focuses exclusively on vertical fusion over skeleton compositions.

6 Conclusion And Future Work

In this work, we studied the feasibility of fusing kernels written in the PolyHok DSL using metaprogramming techniques. We presented a user-directed fusion mechanism capable of composing parallel skeletons and transforming them into fused GPU kernels through source-to-source rewriting.

A key characteristic of the proposed approach is its low-intrusion design. Fusion is exposed through a reusable macro abstraction that can be selectively applied by programmers with virtually no language changes. Therefore, it can be incrementally introduced into existing applications while preserving the original programming model offered by the DSL.

Experimental results demonstrated that the proposed transformation can effectively reduce memory traffic and improve performance in the benchmarks tested. The largest gains were observed

in applications composed of long producer-consumer pipelines, in which the elimination of intermediate tensors resulted in substantial speedups.

Future work will explore several extensions of the current prototype. First, we intend to investigate horizontal kernel fusion techniques capable of combining independent kernels to improve resource utilization and occupancy. We also plan to evaluate `CUDA Graphs` as an alternative mechanism for reducing kernel launch overhead and compare their effectiveness with the proposed fusion strategy. Another promising direction is the development of performance prediction models capable of estimating the profitability of fusion before applying the transformation. Finally, we intend to extend the fusion algebra with additional skeletons and evaluate the proposed approach on larger GPU applications.

Artifact Availability

The research artifacts, including the source code of the implemented module and the scripts used in the experiments, are publicly available at https://github.com/JoaoNevesSoares/PolyHok.

Acknowledgements

This study was financed in part by CAPES - Finance Code 001, and by FAPERGS - 24/2551-0001372-5.

The authors declare that large language models (LLMs) assisted in preparing the manuscript. `Google Gemini` assisted in structuring arguments and refining the narrative flow, while `LanguageTool` was used for paraphrasing and grammatical revision. The authors rigorously reviewed all AI-generated suggestions and assume full responsibility for the paper's final content and scientific integrity.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467 [cs.DC]

[2] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 8, 1 pages. doi:10.1145/3388333.3388658

[3] Oscar Amoros, Albert Andaluz, Johnny Nunez, and Antonio J. Pena. 2025. The Fused Kernel Library: A C++ API to Develop Highly-Efficient GPU Libraries. arXiv:2508.07071 [cs.DC] https://arxiv.org/abs/2508.07071

[4] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (June 2016), 27 pages. doi:10.1145/2898354

[5] Richard S. Bird. 1989. Algebraic identities for program calculation. *Comput. J.* 32, 2 (1989), 122–126.

[6] André Du Bois, Tiago Perlin, Frederico Antunes, and Gerson Cavalheiro. 2024. Hok: Higher-Order GPU kernels in Elixir. In *Anais do XXVIII Simpósio Brasileiro de Linguagens de Programação* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 71–80. doi:10.5753/sblp.2024.3690

[7] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, et al. 2021. JAX: Autograd and XLA. arXiv:ascl:2111.002

[8] 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 the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1926354.1926358

[9] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10) (IISWC '10)*. IEEE Computer Society, USA, 1–11. doi:10.1109/IISWC.2010.5650274

[10] André Rauber Du Bois and Gerson Cavalheiro. 2026. Polymorphic Higher-Order GPU Kernels. In *Euro-Par 2025: Parallel Processing*, Wolfgang E. Nagel, Diana Goehringer, and Pedro C. Diniz (Eds.). Springer Nature Switzerland, Cham, 100–113.

[11] André Rauber Du Bois and Gerson Geraldo H. Cavalheiro. 2025. GPotion: Embedding GPU programming in Elixir. *Journal of Computer Languages* 83 (2025), 101323. doi:10.1016/j.cola.2025.101323

[12] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[13] Jan Fousek, Jiří Filipovič, and Matuš Madzin. 2011. Automatic fusions of CUDA-GPU kernels for parallel map. *SIGARCH Comput. Archit. News* 39, 4 (Dec. 2011), 98–99. doi:10.1145/2082156.2082183

[14] Mark Harris et al. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 70.

[15] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/2502323.2502328

[16] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/2502323.2502328

[17] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for GPU programming. *Comput. Surveys* 55, 11 (2023), 1–81.

[18] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2020. Automatic Horizontal Fusion for GPU Kernels. arXiv:2007.01277 [cs] doi:10.48550/arXiv.2007.01277

[19] 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 (Jan 2016), 210–224. doi:10.1109/TPDS.2015.2394802

[20] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (2008), 40–53.

[21] Víctor Pérez, Lukas Sommer, Victor Lomüller, Kumudha Narasimhan, and Mehdi Goli. 2023. User-driven Online Kernel Fusion for SYCL. *ACM Trans. Archit. Code Optim.* 20, 2, Article 21 (March 2023), 25 pages. doi:10.1145/3571284

[22] Victor Podlozhnyuk. 2007. *Black-Scholes Option Pricing.* Technical Report. NVIDIA Corporation. https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/BlackScholes/doc/BlackScholes.pdf NVIDIA CUDA SDK Documentation.

[23] Shigeyuki Sato and Hideya Iwasaki. 2009. A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems* (Seoul, Korea) *(APLAS '09)*. Springer-Verlag, Berlin, Heidelberg, 79–94. doi:10.1007/978-3-642-10672-9_8

[24] Savvas Sioutas, Sander Stuijk, Twan Basten, Henk Corporaal, and Lou Somers. 2020. Schedule synthesis for halide pipelines on gpus. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 3 (2020), 1–25.

[25] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[26] Guibin Wang, YiSong Lin, and Wei Yi. 2010. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing (GREENCOM-CPSCOM '10)*. IEEE Computer Society, USA, 344–350. doi:10.1109/GreenCom-CPSCom.2010.102

[27] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[28] Chenchen Zhang, Hao Luo, and Chao Yang. 2025. *Leonid: Exploring Automated Kernel Fusion in Performance-Portable Programming Models for Scientific Computation.* Technical Report. Peking University.