Ferrum: uma extensão embutida de Rust para programação em GPUs

Lucas de Souza Barbosa[1], **André Rauber Du Bois**[1], **Gerson Geraldo Homrich Cavalheiro**[1]

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

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

Abstract. *This paper presents Ferrum, an embedded extension of the Rust programming language for GPU programming. Using Rust's metaprogramming facilities, Ferrum allows programmers to write CUDA-like kernels directly in Rust, which are statically validated and translated to CUDA C at compile time, without modifying the Rust compiler. Device memory is managed by Rust's ownership rules, and the mutability declared in a kernel signature is checked and carried through to the generated CUDA code. Experiments with three classic benchmarks show performance comparable to NVIDIA's experimental cuda-oxide compiler, and large speedups over sequential Rust code.*

Resumo. *Este artigo apresenta a Ferrum, uma extensão embutida da linguagem Rust para programação em GPUs. Usando os mecanismos de metaprogramação de Rust, a Ferrum permite escrever kernels no estilo CUDA diretamente em Rust, que são validados estaticamente e traduzidos para CUDA C em tempo de compilação, sem modificar o compilador da linguagem. A memória do device é gerenciada pelas regras de ownership de Rust, e a mutabilidade declarada na assinatura do kernel é verificada e propagada até o código CUDA gerado. Experimentos com três benchmarks clássicos mostram desempenho comparável ao do compilador experimental cuda-oxide, da NVIDIA, e ganhos expressivos sobre o código sequencial em Rust.*

1. Introdução

Nos últimos anos, as GPUs (*Graphics Processing Units*) deixaram de ser dispositivos exclusivos para processamento gráfico e se tornaram a principal plataforma para computação paralela de propósito geral (GPGPU), sendo usadas em áreas como Inteligência Artificial, criptografia e computação científica [Owens et al. 2007]. Plataformas como CUDA [NVIDIA 2026] e OpenCL [Khronos Group 2026] permitem escrever funções chamadas *kernels* que executam no *device* (a GPU) a partir do código *host* (a CPU) [Nickolls et al. 2008].

Apesar de poderosas, essas plataformas expõem abstrações de baixo nível baseadas em C/C++: o programador é responsável por alocar e liberar memória explicitamente, coordenar transferências entre host e device e sincronizar milhares de *threads*. Esse gerenciamento manual favorece o surgimento de erros como *data races*, acessos fora dos limites de vetores e vazamentos de memória, que são notoriamente difíceis de depurar em programas massivamente paralelos [Eizenberg et al. 2017]. Boa parte das

ferramentas que atacam esse problema, como CURD [Peng et al. 2018] e BARRACUDA [Eizenberg et al. 2017], atua depois que o código foi escrito, detectando falhas em tempo de execução, em vez de preveni-las por construção.

Nesse cenário, a linguagem Rust [The Rust Project Developers 2026] se destaca por oferecer segurança de memória sem *garbage collector*: seu *borrow checker* valida, em tempo de compilação, um conjunto de regras baseado nos conceitos de *ownership* e *borrowing*, eliminando por construção erros como *data races* na CPU, uso de memória já liberada e dupla desalocação [Qin et al. 2020]. A pergunta natural é: por que não levar essas garantias, que já funcionam tão bem no host, para o código que executa na GPU?

Este artigo apresenta a Ferrum[1], uma extensão embutida de Rust para programação em GPUs. Com ela, o programador escreve kernels no estilo CUDA diretamente em Rust, dentro do próprio projeto Cargo, e a extensão se encarrega de validar, traduzir e compilar esse código para a GPU, tudo em tempo de compilação e sem nenhuma modificação no compilador de Rust. A abordagem segue a linha de DSLs embutidas via metaprogramação, como a GPotion [Du Bois and Cavalheiro 2023] em Elixir, mas com um diferencial importante: por ser estaticamente tipada, Rust permite validar o kernel em tempo de compilação, e as regras de *ownership* da própria linguagem passam a governar também os dados que vivem na GPU. O protótipo atual leva para a GPU o gerenciamento da memória e as verificações de tipos e de mutabilidade; estender as regras de empréstimo ao modelo de execução do device é trabalho em andamento (Seção 3.5).

As contribuições deste artigo são:

- Apresentamos a Ferrum, uma extensão embutida de Rust para a escrita de kernels GPU, construída inteiramente com as macros procedurais da linguagem (Seção 3);
- Descrevemos a implementação da extensão: um pipeline que, em tempo de compilação, analisa a AST do kernel, realiza verificação de tipos com diagnósticos integrados ao compilador de Rust, gera código CUDA C equivalente e o compila com o `nvcc` (Seção 4);
- Avaliamos o desempenho da extensão com três benchmarks clássicos, comparando com Rust sequencial e com o cuda-oxide [NVlabs 2026], um compilador experimental de Rust para CUDA desenvolvido pela NVIDIA (Seção 5).

O restante do artigo está organizado da seguinte forma: a Seção 2 revisa os fundamentos de Rust e do modelo de execução CUDA. A Seção 3 descreve a extensão do ponto de vista do programador, e a Seção 4 explica como ela é implementada. A Seção 5 apresenta os experimentos, a Seção 6 discute os trabalhos relacionados e a Seção 7 conclui e aponta os próximos passos.

2. Fundamentos

2.1. Rust, *ownership* e o *borrow checker*

Rust é uma linguagem multiparadigma voltada para programação de sistemas, que combina desempenho comparável ao de C/C++ com garantias fortes de segurança de memória. O ponto central da linguagem é o modelo de *ownership*: cada valor

[1]O código-fonte da extensão e dos benchmarks está disponível em `https://github.com/sblucas97/ferrum`.

tem um único dono (*owner*) e, quando o escopo do dono termina, o valor é desalocado automaticamente, sem *garbage collector* e sem chamadas manuais de liberação [The Rust Project Developers 2026].

Sobre essa base, Rust define as regras de *borrowing*: um valor pode ser emprestado por referência, e o compilador garante que, a qualquer momento, existe *ou* uma única referência mutável (`&mut T`) *ou* qualquer número de referências imutáveis (`&T`), nunca as duas coisas ao mesmo tempo. Quem faz essa validação é o *borrow checker*, um componente do compilador que analisa estaticamente os tempos de vida (*lifetimes*) de cada referência. O resultado prático é que programas com *data races*, ponteiros pendentes ou dupla desalocação simplesmente não compilam [Qin et al. 2020]. O trabalho Oxide [Weiss et al. 2019] formaliza a essência dessas regras, e serve de base teórica para levar essas garantias a outros modelos de execução.

Outro recurso importante para este trabalho é a metaprogramação de Rust. Macros procedurais (*proc macros*) são funções que executam durante a compilação, recebendo a árvore sintática (AST) do código anotado e podendo analisá-la, transformá-la ou gerar código novo a partir dela. É esse mecanismo que permite embutir uma linguagem de kernels dentro de Rust sem tocar no compilador.

2.2. Programação de GPUs com CUDA

No modelo CUDA [Nickolls et al. 2008], uma função que executa na GPU é chamada de *kernel*. Ao ser lançado, um kernel executa em um *grid* de blocos, e cada bloco contém um conjunto de *threads*. Combinando os índices de thread e de bloco (`threadIdx`, `blockIdx`, `blockDim`, `gridDim`), cada thread calcula um identificador único, geralmente usado para decidir quais elementos dos dados ela processa.

A GPU possui memória própria, separada da memória principal. O fluxo típico de um programa CUDA é: alocar memória no device (`cudaMalloc`), copiar os dados de entrada para lá (`cudaMemcpy`), lançar o kernel sobre um grid, e copiar os resultados de volta para o host. Cada uma dessas etapas é manual e propensa a erros: esquecer um `cudaFree` causa vazamento, errar um tamanho de cópia corrompe memória, e duas threads escrevendo na mesma posição sem sincronização geram um *data race* silencioso.

3. A extensão Ferrum

Esta seção apresenta a Ferrum do ponto de vista de quem programa com ela. A extensão adiciona a Rust duas abstrações principais: o `CudaVec`, um vetor que vive na memória da GPU, e os kernels, funções Rust anotadas que executam no device.

3.1. CudaVec: vetores na GPU

O `CudaVec<T>` é a ponte entre o mundo do host e o do device. Ao criar um `CudaVec` a partir de um vetor Rust comum, a extensão aloca a memória correspondente na GPU e copia os dados para lá:

```
1 let a: CudaVec<f32> = CudaVec::new(vec![1.0; n]);
```

A partir daí, as regras de *ownership* de Rust valem também para a memória do device. Quando o `CudaVec` sai de escopo, sua implementação de `Drop` libera automaticamente a memória da GPU. É o equivalente ao `cudaFree`, mas emitido pelo compilador

no ponto certo, o que elimina uma das formas mais comuns de vazamento em programas CUDA, a liberação esquecida. A cópia dos resultados de volta para o host é explícita, via `copy_from_device()`, deixando claro no código o ponto em que a transferência acontece.

Nem todo tipo pode ser colocado em um `CudaVec`: os elementos devem implementar o *trait* selado `DeviceScalar`, implementado apenas para os tipos numéricos primitivos (`f32`, `u64` etc.). Isso impede, em tempo de compilação, que estruturas contendo ponteiros do heap do host sejam copiadas por engano para a GPU, um erro que, em CUDA C, compila normalmente e falha só em execução.

3.2. Escrevendo kernels

Os kernels são escritos dentro de um módulo Rust anotado com `#[cuda_module]`, e cada kernel é uma função anotada com `#[kernel]`. A Figura 1 mostra um kernel que soma dois vetores, no mesmo espírito do exemplo clássico de CUDA.

```
#[cuda_module]
pub mod vector_sum_kernel {
    use runtime::CudaVec;
    use macros::kernel;

    #[kernel]
    pub fn add_vectors(a: &CudaVec<f32>, b: &CudaVec<f32>,
                       result: &mut CudaVec<f32>, n: u64) {
        let idx: u64 = blockIdx.x * blockDim.x + threadIdx.x;
        let stride: u64 = blockDim.x * gridDim.x;

        for i in (idx, n).step_by(stride) {
            result[i] = a[i] + b[i];
        }
    }
}
```

Figura 1. Um kernel Ferrum que soma dois vetores

Visualmente, o código é sintaticamente Rust válido: a sintaxe é aceita pelo *parser* da linguagem e editores e ferramentas do ecossistema (`rust-analyzer`, `cargo`) continuam funcionando. Semanticamente, porém, o corpo do kernel segue o modelo CUDA: as constantes `threadIdx`, `blockIdx`, `blockDim` e `gridDim` estão disponíveis, e o laço `for` com `(início, fim).step_by(passo)` expressa o padrão *grid-stride loop*, comum em kernels CUDA.

Um detalhe importante da assinatura do kernel: os parâmetros de entrada são recebidos como referências imutáveis (`&CudaVec<f32>`) e o de saída como referência mutável (`&mut CudaVec<f32>`). Essa distinção não é só documentação: escrever através de um parâmetro declarado como `&CudaVec<T>` é rejeitado em tempo de compilação, e a mutabilidade é propagada até o código CUDA gerado, como veremos na Seção 4. Dentro do kernel, o programador também pode usar um conjunto de funções matemáticas do device (`sqrtf`, `cosf`, `sinf` e `floorf`), traduzidas para as funções correspondentes do CUDA.

3.3. Lançando kernels

Para executar um kernel, usa-se a macro `spawn!`, que recebe o kernel, as dimensões do grid e dos blocos, e os argumentos (Figura 2).

```
pub fn run(n: usize) {
    let a: CudaVec<f32> = CudaVec::new((1..=n).map(|i| i as f32).collect
        ());
    let b: CudaVec<f32> = CudaVec::new((1..=n).map(|i| i as f32).collect
        ());
    let mut result: CudaVec<f32> = CudaVec::new(vec![0.0f32; n]);

    let threads_per_block: u32 = 128;
    let num_blocks: u32 =
       (n as u32 + threads_per_block - 1) / threads_per_block;

    spawn!(vector_sum_kernel::add_vectors,
         (num_blocks, 1, 1), (threads_per_block, 1, 1),
         a, b, result, n as u64);

    result.copy_from_device();
}
```

Figura 2. Lançando um kernel Ferrum a partir do código host

O `spawn!` é síncrono: ele lança o kernel e espera a GPU terminar, o que simplifica o raciocínio sobre o programa. Depois do lançamento, o host recupera os resultados com `copy_from_device()` e segue usando o vetor como um vetor Rust qualquer.

3.4. O que o compilador rejeita

Como os kernels são sintaticamente funções Rust válidas, os erros são reportados pelo próprio `cargo build`, no mesmo formato dos demais erros de compilação. A extensão realiza sua própria verificação de tipos sobre o corpo do kernel (Seção 4) e rejeita, por exemplo:

- atribuições a variáveis não declaradas como `mut` (`cannot assign: 'x' is not mutable`) e usos de variáveis não declaradas (`unknown variable 'x'`);
- escritas em um `CudaVec` recebido por referência imutável, que é o que faz a assinatura do kernel valer como contrato;
- indexação com um índice que não seja de tipo inteiro, ou indexação de algo que não seja um `CudaVec`;
- incompatibilidades de tipos em expressões (`type mismatch: expected 'F32 * F32', found 'F32 * F64'`), condições de `if` que não sejam booleanas e *casts* inválidos (`invalid cast from 'U64' to 'Bool'`);
- chamadas a funções fora do conjunto suportado no device (`unknown function 'scale'`) ou com número errado de argumentos (`'sqrtf' expects 1 argument(s), found 2`).

3.5. O que a Ferrum garante hoje

Como a segurança é uma motivação central deste trabalho, convém delimitar o alcance dessas garantias no protótipo atual, separando quatro camadas: os valores `CudaVec` no host, o corpo do kernel, o lançamento e a execução no device.

Vale distinguir, primeiro, de onde vem cada verificação, porque o host e o kernel estão em situações opostas. Do lado do host, um `CudaVec` é um valor Rust comum, então as regras usuais de *ownership* valem sem nenhum esforço adicional: o compilador impede usar um vetor depois de movido e libera a memória do device no fim do escopo. Nessa camada, a extensão acrescenta apenas dois mecanismos ao que a linguagem já faz, a implementação de `Drop` e o *trait* selado `DeviceScalar`, que restringe os elementos a escalares numéricos. Dentro do kernel acontece o contrário: a macro `#[kernel]` entrega ao compilador de Rust apenas a assinatura da função, e o corpo é consumido pela extensão. Nenhuma verificação da linguagem alcança, portanto, o código do kernel; tudo o que vale lá dentro, os tipos e as conversões, a mutabilidade das variáveis locais e a escrita apenas através de `&mut CudaVec<T>`, é o que o verificador da extensão implementa (Seção 4).

Essa segunda camada se sustenta também pelo que o subconjunto aceito deixa de fora. Dentro de um kernel não existe `unsafe`, não há como formar uma referência ou um ponteiro, não há chamada de método e não há laço fora do padrão *grid-stride*, de modo que a única operação de memória que a extensão expõe no device é a indexação de um `CudaVec`. Isso delimita com clareza o que resta por garantir: os erros de memória possíveis no código gerado se reduzem a índices fora dos limites da alocação e a escritas concorrentes na mesma posição. Nenhum dos dois é verificado hoje.

A primeira delas merece ser explicitada, porque o host e o device se comportam de maneiras opostas. No host, indexar um `CudaVec` é indexar o vetor Rust que ele mantém, com a verificação de limites usual da linguagem. No device, o mesmo colchete é traduzido em acesso direto por ponteiro, sem verificação alguma: a extensão não sintetiza um teste de faixa nem o injeta no prólogo do kernel. A razão é que o comprimento da alocação não atravessa a fronteira do lançamento, já que um `CudaVec` é passado apenas como ponteiro, sem um parâmetro de tamanho que o acompanhe. Todo índice calculado dentro do kernel é, portanto, contrato do programador, e não apenas o uso do parâmetro `n` que vários kernels recebem: os testes de faixa presentes no código gerado, como o limite do laço da Figura 1, vêm do que foi escrito no kernel Ferrum, não da tradução.

A mutabilidade declarada na assinatura é verificada no corpo do kernel e propagada até o código gerado (Seção 4), e vale ser preciso sobre o que isso significa: nenhuma escrita através de um parâmetro declarado `&CudaVec<T>` é possível, o que acaba sendo garantido duas vezes, pela extensão e depois pelo próprio `nvcc`. O que a anotação não expressa é a disjunção dos acessos entre threads, porque `&mut` é uma permissão de escrita do parâmetro, não de cada thread. O kernel do *nbodies* (Seção 5) ilustra bem o ponto: ele recebe um único `&mut CudaVec<f32>` que todas as threads leem e escrevem, e é correto apenas porque cada corpo ocupa seis posições consecutivas do vetor, das quais a thread lê as três primeiras (a posição) e escreve as três últimas (a velocidade). Essa partição não aparece na assinatura e não é verificada em lugar algum; na versão sequencial do mesmo algoritmo, escrita em Rust puro, o mesmo padrão de acesso é rejeitado pelo *borrow checker* e exige uma cópia do vetor a cada passo da simulação.

O lançamento é a camada em que o protótipo ainda não oferece garantias. A macro `spawn!` usa o caminho do kernel apenas para derivar o nome do arquivo PTX e converte cada argumento em um parâmetro de lançamento sem confrontá-lo com a assinatura declarada, de modo que escapam dessa conferência tanto a exclusividade dos empréstimos, já que nada impede passar o mesmo `CudaVec` como entrada e como saída, quanto a aridade, a ordem e os tipos dos argumentos. Resta a última camada, a da execução no device, em que o *aliasing* entre threads, a ausência de *data races* e os acessos dentro dos limites das alocações seguem sendo contrato do programador. Fazer a verificação alcançar o ponto de lançamento, e não apenas o corpo do kernel, é o primeiro passo do trabalho em andamento (Seção 7).

4. Implementação

Assim como a GPotion faz com as macros de Elixir [Du Bois and Cavalheiro 2023], a Ferrum é implementada inteiramente com os recursos de metaprogramação de Rust, sem modificar o compilador da linguagem. A Figura 3 resume o processo de compilação de um kernel.

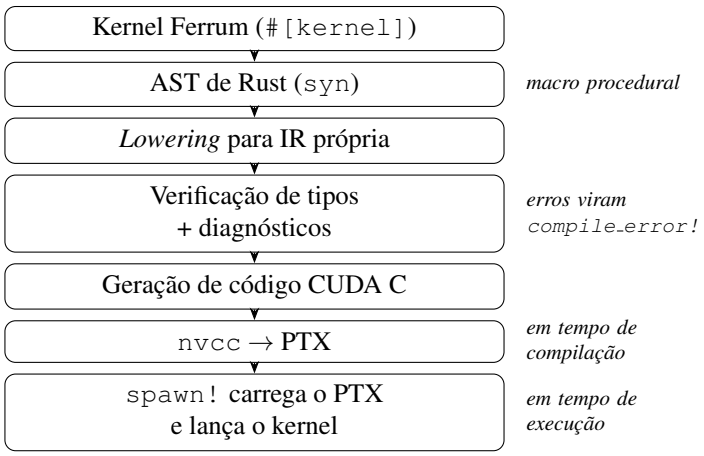

Figura 3. Pipeline de compilação de um kernel Ferrum

4.1. Das macros à IR

Quando o `cargo build` encontra um módulo anotado com `#[cuda_module]`, a macro procedural correspondente é executada pelo compilador de Rust. Ela recebe a AST do módulo (por meio da biblioteca `syn`) e, para cada função anotada com `#[kernel]`, dispara o pipeline de compilação da extensão.

O primeiro passo é o *lowering*: a AST de Rust, que é grande e cheia de detalhes irrelevantes para o nosso caso, é convertida em uma Representação Interna (IR) própria, bem mais enxuta, contendo apenas os construtos aceitos dentro de kernels (declarações, atribuições, expressões aritméticas, indexação de vetores, condicionais, laços no padrão *grid-stride* e chamadas às funções matemáticas do device). Construções fora desse subconjunto são rejeitadas já nessa fase, com uma mensagem indicando o que não é suportado.

4.2. Verificação de tipos e diagnósticos

A verificação de tipos da extensão é executada sobre a IR. Ela mantém um contexto com o tipo e a mutabilidade de cada variável e percorre o corpo do kernel validando cada declaração, expressão e atribuição, incluindo as regras específicas do domínio, como "o índice de um `CudaVec` precisa ser um inteiro", "só se escreve em um `CudaVec` recebido por `&mut`" e o conjunto de funções matemáticas disponíveis no device. As conversões implícitas são deliberadamente restritas: um inteiro pode ser alargado dentro do mesmo sinal (de `u32` para `u64`, por exemplo), mas conversões entre inteiro e ponto flutuante, ou entre com e sem sinal, exigem um `as` explícito, como em Rust.

Um aspecto que consideramos importante é a forma de reportar erros: quando alguma validação falha, a macro não interrompe a compilação com um *panic*. Em vez disso, ela emite um `compile_error!` com a mensagem do diagnóstico, que o compilador de Rust apresenta ao usuário como um erro de compilação comum, apontando o kernel problemático. Na prática, errar dentro de um kernel Ferrum tem a mesma experiência de errar em Rust puro.

4.3. Geração de código e execução

Passadas as validações, o back-end percorre a IR e emite um kernel CUDA C equivalente. A tradução é direta, já que cada construto da IR tem um mapeamento óbvio para C, e é nesse ponto que as anotações de mutabilidade se refletem no código gerado: parâmetros recebidos como `&CudaVec<f32>` viram ponteiros `const float*` no código gerado, enquanto `&mut` vira um ponteiro mutável comum. A Figura 4 mostra o código gerado para o kernel da Figura 1.

```
extern "C" __global__ void add_vectors(const float *a,
      const float *b, float *result, uint64_t n) {
   uint64_t idx = blockIdx.x * blockDim.x + threadIdx.x;
   uint64_t stride = blockDim.x * gridDim.x;
   for (uint64_t i = idx; i < n; i += stride) {
      result[i] = a[i] + b[i];
   }
}
```

Figura 4. Código CUDA C gerado para o kernel da Figura 1

Ainda durante a compilação do projeto Rust, a macro invoca o `nvcc` para compilar o código gerado para PTX, a representação intermediária das GPUs NVIDIA. Ou seja: se o `cargo build` terminou, o kernel já está validado, traduzido e compilado. Não existe uma etapa separada de build para a parte GPU do programa.

Em tempo de compilação, a macro `spawn!` se expande para o código que carrega o PTX correspondente ao kernel, monta a configuração de lançamento (grid, blocos) e os argumentos, extraindo automaticamente os ponteiros de device dos `CudaVec`. Esse código gerado é executado em tempo de execução, lançando o kernel por meio da biblioteca `cudarc` [Lowman 2026], que encapsula a API de driver do CUDA. O runtime sincroniza o stream ao final, de modo que o `spawn!` retorna apenas quando o kernel terminou.

4.4. Cobertura das verificações

As verificações descritas acima são exercitadas por uma suíte de regressão que não exige GPU nem o toolkit CUDA instalado, já que tudo o que ela observa acontece em tempo de compilação. São três partes. A primeira é um conjunto de sete kernels malformados, um para cada classe de erro listada na Seção 3, compilados com o `trybuild`: cada caso fixa o diagnóstico exato que o `cargo build` deve produzir, de modo que uma mudança no veredicto ou na redação do verificador aparece como diferença em vez de passar despercebida. A segunda espelha o corpo de todos os kernels dos benchmarks da Seção 5 e exige que cada um passe pelo *lowering* e pela verificação de tipos. A terceira fixa o código CUDA C gerado para cada tipo escalar suportado, incluindo os parâmetros `const`, para que a propagação da mutabilidade não se perca em uma mudança do gerador. Ampliar essa suíte às camadas de lançamento e de device, assim que elas passarem a ser verificadas, é parte do trabalho previsto.

5. Experimentos

O objetivo desta seção é comparar o desempenho de duas abordagens distintas de levar Rust à GPU e situar ambas em relação a uma execução sequencial na CPU. Não se trata de isolar o custo da abstração da Ferrum: os dois caminhos de compilação são diferentes, de modo que as diferenças observadas dizem respeito ao caminho inteiro, e não apenas à tradução que a extensão realiza. Comparamos três implementações de cada benchmark:

- **Rust sequencial**: implementação single-thread em Rust puro, executada na CPU, como linha de base;
- **Ferrum**: a extensão apresentada neste artigo;
- **cuda-oxide** [NVlabs 2026]: compilador experimental da NVIDIA que traduz Rust idiomático diretamente para PTX por meio de um back-end de codegen próprio do `rustc`.

O cuda-oxide é um comparativo interessante por dois motivos: ele persegue o mesmo objetivo geral (escrever kernels CUDA em Rust), mas com uma estratégia de implementação oposta: um back-end de compilador dedicado, preso a uma versão *nightly* específica do `rustc`, contra nossa abordagem de macros, que funciona com as distribuições estáveis do Rust, sem exigir um *toolchain* especial.

Utilizamos três benchmarks clássicos de GPGPU: *julia*, que gera imagens do conjunto de Julia; *nbodies*, uma simulação de N corpos sob influência gravitacional; e *raytracer*, um traçador de raios sobre uma cena com 20 esferas. Julia e raytracer são adaptados dos exemplos de [Sanders and Kandrot 2010], os mesmos usados na avaliação da GPotion [Du Bois and Cavalheiro 2023]; nos dois casos, o tamanho do problema é a dimensão da imagem gerada (uma imagem $n \times n$), e no nbodies é o número de corpos simulados.

Para viabilizar a comparação, as versões em Rust sequencial e em cuda-oxide foram portadas a partir das implementações Ferrum com o auxílio de IA generativa (Claude [Anthropic 2026]), seguida de revisão manual e validação dos resultados produzidos: as três versões de cada benchmark geram saídas equivalentes sobre as mesmas entradas. Consideramos essa informação relevante tanto pela transparência metodológica quanto por ilustrar um uso prático de IA na construção de artefatos experimentais.

Os experimentos foram executados em uma máquina com CPU AMD Ryzen 7 5700G (16 threads), 16 GB de RAM e uma GPU NVIDIA GeForce RTX 3070 (8 GB), usando Linux, CUDA 12.0 e rustc 1.95.0. Cada configuração foi executada 30 vezes, em ordem blocada entre as implementações, e reportamos a média. Os tempos medidos cobrem a porção GPU do programa: alocação dos vetores no device, transferências entre host e device e a execução dos kernels. Os desvios padrão ficaram abaixo de 3% da média em todas as configurações. A Tabela 1 resume os resultados.

Convém registrar que a memória disponível nessa máquina foi o que determinou os tamanhos de entrada. No julia e no raytracer, cada imagem RGBA em ponto flutuante ocupa 16 bytes por pixel na GPU, e o `CudaVec` mantém uma cópia equivalente no host, de modo que os 8 GB do device e os 16 GB de RAM se esgotam por volta de uma imagem de 20.000×20.000. É por isso que os maiores tamanhos testados param onde param. No nbodies o limite é de outra natureza: o kernel é de pares, com custo quadrático, então o teto prático vem do tempo de execução e não da memória. Uma consequência disso é que a comparação com o Rust sequencial só pôde ser feita no menor tamanho de cada benchmark, já que as versões de CPU levam dezenas de segundos por execução e são repetidas 30 vezes.

Tabela 1. Resultados dos experimentos (tempos médios de 30 execuções)

	Tamanho	Rust seq. (ms)	Ferrum (ms)	cuda-oxide (ms)	Diferença
Julia	10000	50.853	2.226	2.394	−7,0%
Julia	12000	–	3.159	3.498	−9,7%
Julia	16000	–	5.574	5.989	−6,9%
nBodies	80000	50.747	240	231	+3,9%
nBodies	500000	–	3.740	3.531	+5,9%
nBodies	800000	–	8.906	8.747	+1,8%
Raytracer	10000	2.971	1.428	1.063	+34,3%
Raytracer	15000	–	3.029	2.214	+36,8%
Raytracer	20000	–	5.297	3.876	+36,6%

A coluna “Diferença” é o tempo da Ferrum em relação ao cuda-oxide (valores negativos indicam que a Ferrum foi mais rápida).

5.1. Discussão

O primeiro resultado esperado se confirma: mover a computação para a GPU compensa muito. Os ganhos abaixo valem apenas para o menor tamanho de cada benchmark, o único em que a versão sequencial foi medida: no julia, a Ferrum foi cerca de $23\times$ mais rápida que o Rust sequencial; no nbodies, o ganho passa de $200\times$; no raytracer, cuja versão sequencial é mais competitiva, ainda assim a execução na GPU alcança o dobro do desempenho.

A comparação com o cuda-oxide é mais interessante. No nbodies, a diferença fica entre 1,8% e 5,9%, na prática um empate técnico, o que indica que o caminho “macro $\rightarrow$ CUDA C $\rightarrow$ `nvcc`” não impõe um custo relevante nesse tipo de kernel. No julia, a Ferrum foi consistentemente mais rápida (7% a 10%). Como os tempos medidos incluem

alocação e transferências, além da execução do kernel, e nenhuma das versões foi instrumentada, atribuir essa diferença à qualidade do código gerado por cada back-end seria uma hipótese, não uma conclusão sustentada pelos dados; decompor o tempo por etapa é trabalho futuro.

O raytracer mostra o outro lado: a Ferrum ficou entre 34% e 37% mais lenta. É o kernel mais pesado dos três, com um laço sobre as 20 esferas da cena contendo um desvio condicional que diverge entre threads do mesmo warp, várias variáveis locais em ponto flutuante vivas ao mesmo tempo e leituras indexadas com passo sete sobre o vetor de esferas. Não chegamos a instrumentar o kernel, então não temos como afirmar qual desses fatores domina, e a diferença pode estar tanto no código que emitimos quanto na forma como cada back-end trata a divergência. Perfilar esse caso com as ferramentas da NVIDIA, para então decidir o que otimizar na geração de código, é um dos próximos passos naturais do trabalho.

No conjunto, os resultados sugerem que, nas cargas avaliadas, a abordagem embutida via metaprogramação é competitiva: em dois dos três benchmarks, ela fica no mesmo patamar (ou à frente) de um compilador dedicado mantido pela própria NVIDIA. Convém não estender essa leitura além do que foi medido: são três benchmarks, e os padrões que eles exercitam já são limitados pelo protótipo, que não oferece memória compartilhada nem primitivas de sincronização, de modo que os kernels avaliados são todos de threads independentes sobre a memória global, com lançamento síncrono.

6. Trabalhos relacionados

O trabalho mais próximo em espírito é a GPotion [Du Bois and Cavalheiro 2023], uma DSL embutida em Elixir para programação de GPUs, implementada com as macros da linguagem e sem modificar seu compilador, exatamente a estratégia que adotamos. A Polyhok [Du Bois and Cavalheiro 2025] estende essa linha com kernels polimórficos e de alta ordem. A diferença central da Ferrum está na linguagem hospedeira: por Rust ser estaticamente tipada e ter as regras de *ownership* e *borrowing* nativas, conseguimos validar os kernels em tempo de compilação e colocar a memória do device sob o mesmo regime de *ownership* do host, algo que uma linguagem dinâmica como Elixir não oferece.

Na direção de linguagens seguras para GPU, o Descend [Köpcke et al. 2024] é uma linguagem de sistemas inspirada em Rust que leva *ownership* e *borrowing* ao modelo massivamente paralelo das GPUs, com um sistema de tipos capaz de rejeitar *data races* em tempo de compilação. O Descend se apoia na formalização do Oxide [Weiss et al. 2019], e é uma referência importante para as regras que queremos garantir. A diferença é de abordagem: o Descend é uma linguagem e um compilador novos, enquanto a Ferrum estende o próprio Rust, preservando o ecossistema (Cargo, editores, bibliotecas) que os desenvolvedores já usam. Já o Futhark [Henriksen et al. 2017] alcança segurança por meio do paradigma funcional puro, mas restringe o estilo de programação, afastando-se do controle de baixo nível que kernels CUDA oferecem.

No ecossistema Rust, RustaCUDA [Heisler 2026] e cudarc [Lowman 2026] encapsulam a API CUDA no host, mas os kernels continuam escritos em CUDA C ou PTX externo, que é justamente a lacuna que a Ferrum preenche. O projeto Rust GPU [Rust GPU Project 2026] compila Rust para SPIR-V com foco em *shaders* gráficos, e Holk et al. [Holk et al. 2013] exploraram, ainda nos primórdios de Rust, abstrações de

alto nível para GPU na linguagem. O trabalho mais próximo tecnicamente é o cuda-oxide [NVlabs 2026], da NVIDIA, que compila Rust idiomático para PTX por meio de um back-end dedicado do `rustc`. Vale registrar a linha do tempo: os primeiros commits da Ferrum são de julho de 2025, anteriores à criação do repositório público do cuda-oxide (abril de 2026). As duas iniciativas chegaram de forma independente à mesma ideia, o que reforça a relevância de trazer Rust para a programação de GPUs. As abordagens, porém, são complementares: o cuda-oxide exige um *toolchain nightly* específico e um back-end de codegen próprio, enquanto a Ferrum funciona como uma biblioteca de macros sobre o Rust estável.

Por fim, ferramentas de detecção dinâmica de *data races* em CUDA, como CURD [Peng et al. 2018] e BARRACUDA [Eizenberg et al. 2017], atacam o mesmo problema de segurança por outro ângulo: encontrar erros em execução. Nosso objetivo de longo prazo é o inverso, ou seja, impedir que esses programas compilem, e essas ferramentas são úteis justamente para validar a corretude dos códigos gerados pela extensão.

7. Conclusão e trabalhos futuros

Este artigo apresentou a Ferrum, uma extensão embutida de Rust para programação de GPUs. Com ela, kernels no estilo CUDA são escritos em uma sintaxe Rust válida, com semântica própria, validados estaticamente e traduzidos para CUDA C em tempo de compilação, usando apenas as macros procedurais da linguagem, sem compilador próprio e sem *toolchain* especial. A memória do device passa a ser governada pelo *ownership* de Rust, com alocação e liberação automáticas via `Drop`, a mutabilidade declarada na assinatura do kernel é verificada e propagada até o código gerado, e os erros chegam ao programador como diagnósticos comuns do `cargo`.

Os experimentos indicam que a abordagem é viável em termos de desempenho nas cargas avaliadas: nos benchmarks julia e nbodies, a Ferrum ficou no mesmo patamar (ou à frente) do cuda-oxide, o compilador experimental de Rust para CUDA da NVIDIA, e o caso desfavorável (raytracer) aponta uma direção clara de otimização do código gerado.

O próximo passo do trabalho é o seu objetivo mais ambicioso: estender o *borrow checker* ao modelo de execução do device. A arquitetura da extensão já foi desenhada com isso em mente, e a IR sobre a qual é executada a verificação de tipos é o ponto natural para aplicar regras de *ownership* e *borrowing* adaptadas à hierarquia de threads do CUDA, seguindo a formalização do Oxide [Weiss et al. 2019] e as regras exploradas pelo Descend [Köpcke et al. 2024], de modo a rejeitar *data races* e erros de sincronização em tempo de compilação. O primeiro passo concreto nessa direção é fazer a verificação alcançar o ponto de lançamento, e não apenas o corpo do kernel, de modo que o `spawn!` tome os empréstimos correspondentes à assinatura declarada.

O protótipo também tem limitações que pretendemos remover. O subconjunto da linguagem aceito dentro de kernels ainda é pequeno, e funções auxiliares de device, embora já sejam traduzidas pelo gerador de código, ainda não são reconhecidas pela verificação de tipos, o que na prática restringe os kernels às funções matemáticas embutidas. Além disso, não há suporte a memória compartilhada nem a sincronização entre threads, o que também restringe os padrões de programação que a avaliação da Seção 5 pôde exercitar.

Por fim, planejamos ampliar o conjunto de benchmarks e validar os códigos ge-

rados com detectores de races como o CURD [Peng et al. 2018]. Para isso, pretendemos repetir os experimentos em uma infraestrutura mais robusta, com mais memória de host e de device do que a máquina usada aqui. Isso permitiria explorar tamanhos de entrada bem maiores e verificar se as diferenças observadas entre as implementações se mantêm nessa escala, algo que os limites de memória relatados na Seção 5 não deixaram avaliar.

O presente trabalho foi realizado com apoio da CAPES - Código de Financiamento 001 - e da FAPERGS - 24/2551-0001372-5.

Referências

Anthropic (2026). Claude code. Disponível em: `https://claude.com/claude-code`. Acesso em: ago. 2026.

Du Bois, A. R. and Cavalheiro, G. G. H. (2023). GPotion: An embedded DSL for GPU programming in Elixir. In *Proceedings of the XXVII Brazilian Symposium on Programming Languages (SBLP 2023)*, pages 1–8, Campo Grande, MS, Brazil. ACM.

Du Bois, A. R. and Cavalheiro, G. G. H. (2025). Polymorphic higher-order GPU kernels. In *Euro-Par 2025: Parallel Processing – 31st European Conference on Parallel and Distributed Processing*, pages 100–113, Dresden, Germany.

Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., and Devietti, J. (2017). BARRACUDA: Binary-level analysis of runtime races in CUDA programs. *SIGPLAN Notices (PLDI 2017)*, 52(6):126–140.

Heisler, B. (2026). RustaCUDA: Rusty wrapper for the CUDA driver API. Disponível em: `https://github.com/bheisler/RustaCUDA`. Acesso em: ago. 2026.

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. *SIGPLAN Notices (PLDI 2017)*, 52(6):556–571.

Holk, E., Pathirage, M., Chauhan, A., Lumsdaine, A., and Matsakis, N. D. (2013). GPU programming in Rust: Implementing high-level abstractions in a systems-level language. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*, pages 315–324.

Khronos Group (2026). OpenCL: Open standard for parallel programming of heterogeneous systems. Disponível em: `https://www.khronos.org/opencl/`. Acesso em: ago. 2026.

Köpcke, B., Gorlatch, S., and Steuwer, M. (2024). Descend: A safe GPU systems programming language. *Proceedings of the ACM on Programming Languages*, 8(PLDI):181:1–181:24.

Lowman, C. (2026). cudarc: Safe Rust wrapper around the CUDA toolkit. Disponível em: `https://github.com/coreylowman/cudarc`. Acesso em: ago. 2026.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2):40–53.

NVIDIA (2026). CUDA toolkit documentation. Disponível em: `https://developer.nvidia.com/cuda`. Acesso em: ago. 2026.

NVlabs (2026). cuda-oxide: an experimental Rust-to-CUDA compiler. Disponível em: `https://github.com/NVlabs/cuda-oxide`. Acesso em: ago. 2026.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

Peng, Y., Grover, V., and Devietti, J. (2018). CURD: A dynamic CUDA race detector. *SIGPLAN Notices (PLDI 2018)*, 53(4):390–403.

Qin, B., Chen, Y., Yu, Z., Song, L., and Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 763–779, London, UK.

Rust GPU Project (2026). Rust GPU: Making Rust a first-class language and ecosystem for GPU shaders. Disponível em: `https://github.com/Rust-GPU/rust-gpu`. Acesso em: ago. 2026.

Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional.

The Rust Project Developers (2026). The Rust programming language. Disponível em: `https://doc.rust-lang.org/`. Acesso em: ago. 2026.

Weiss, A., Patterson, D., Matsakis, N. D., and Ahmed, A. (2019). Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982*.