HPS

Hoang Nam Nguyen

Rust Programming for HPC Application

A first look into CUDA in Rust

Table of contents

Motivation

Figure: Digital Reality Data Center in Ashburn, Virginia, U.S.

Image source: https://www.reuters.com/technology/big-techs-data-center-boom-poses-new-risk-us-grid-operators-2025-03-19/

High Performance Computing

- Software on the HPC needs to be:
  - Scalable
  - Efficient
  - Guarantee correctness
  - Reliability
- Current programming languages for HPC often include C, C++ and Fortran
  - Good performance
  - Known for lack of memory safety

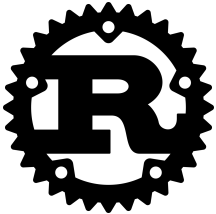
Outline

Rust in the Linux Kernel

Figure: Rust added to Linux kernel languages

Image source: https://lwn.net/Articles/1049831/(accessed 17.01.26)

Rust



- In development since 2006, release 2015
- Since 2022 experimental and 2025 official development language for the Linux kernel
- Used by software companies such as Mozilla, Google, Microsoft, Amazon
- Main goals: Performance, Reliability, Productivity

Rust Features

- Memory safety
- Efficiency
- Multi threading support
- Interfaces for existing C code bases
- (Similar syntax to C, C++)

Rust Ownership

Ownership: Set of rules how Rust manages memory

- Each value has an owner
- There can only be one owner
- When the owner goes out of scope, the value will be dropped

Advantages:

- Ownership is enforced by the compiler
- Automatically frees memory
- No pointers without owner

Rust Ownership Example

```rust
let a = String::from("Hello");
let b = a;

//Error: a no longer owns the value
// println!("{}", a);
println!("{}", b); // Ok: b now owns the value
```

Code from:https://www.w3schools.com/rust/rust_ownership.php

Rust Borrowing and References

Rust Borrowing Example

```rust
let a = String::from("Hello");
let b = &a;

println!("a = {}", a);
println!("b = {}", b);
```

Rust Mutable References Example

```rust
let mut name = String::from("John");
let name_ref = &mut name;
name_ref.push_str(" Doe");

println!("{}", name_ref); // John Doe
```

- & as a pointer that *borrows* the value of the owner
- both borrowing and referencing need to be done consciously
- &mut allows the value of a reference to change
- only one mutable reference at a time is allowed!

Code from:https://www.w3schools.com/rust/rust_borrowing_references.php

Outline

Graphics Processing Unit

- GPUs contain many more cores than CPUs
- For simpler tasks that can be parallelized much faster than CPUs
- Example: NVIDIA RTX PRO 6000: 24,064 CUDA cores, 752 Tensor cores, 188 RT cores

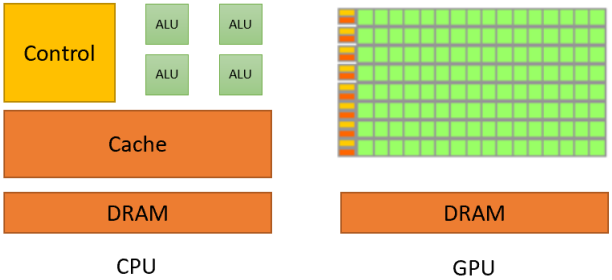


Image source: https://developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

CUDA

- Compute Unified Device Architecture
- Release 2007
- Parallel Computing platform and API for NVIDIA GPU programming
- Almost universally used for GPU programming applications
- Commonly written in C, C++, Fortran

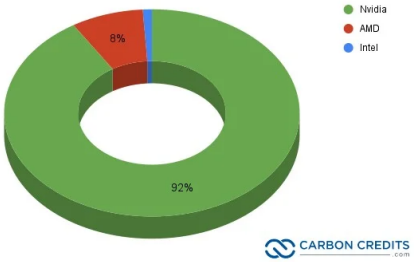


Figure: Discrete GPU market share for the year 2025

;

Image source: https://carboncredits.com/nvidia-controls-92-of-the-gpu-market-in-2025-and-reveals-next-gen-ai-supercomputer/

CUDA Compile Pipeline

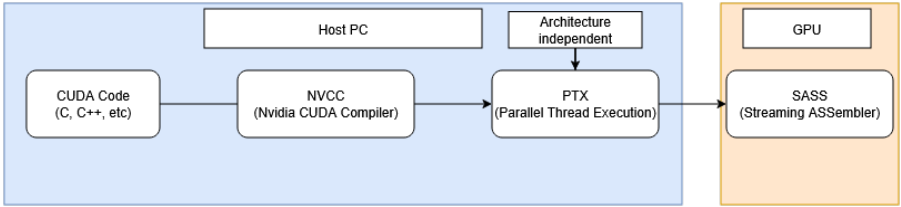


Figure: CUDA compiler pipeline visualized

- CUDA Code compiled with NVCC to PTX (Assembly like)
- PTX is then converted to SASS for the specific GPU

Rust CUDA

Main goals: Compiling Rust to PTX code, crates for using existing CUDA libraries, create an efficient and secure alternative to C/C++/Fortran in GPU programming

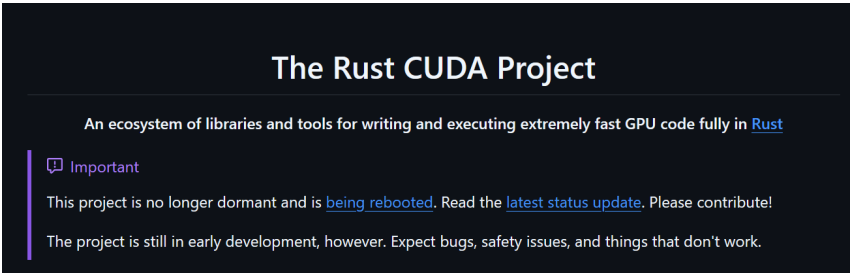


Figure: Rust CUDA Readme

Image source: https://github.com/Rust-GPU/rust-cuda

Rust CUDA Crates

Main crates:

- `rustc_codegen_nvvm`: generates PTX code out of Rust code
- `cuda_std`: GPU-side functions and utilities
- `cust`: CPU-side CUDA features
- `gpu_rand`: GPU-friendly random number generator

Additional crates for specific GPU uses:

- `cudnn`: GPU-accelerated primitives for deep neural networks
- `optix`: Hardware raytracing

Outline

Experimental Setup

Main goals:

- Comparison between equivalent implementations in Rust and C++
- Evaluate performance, memory usage, and PTX code

Experimental Setup

Main goals:

- Comparison between equivalent implementations in Rust and C++
- Evaluate performance, memory usage, and PTX code

Execution:

- Developed locally and then executed on the SCC (Nvidia A100)
- Same Docker/Apptainer image
  - Ubuntu 24.04 LTS
  - CUDA Toolkit 12.8.1

Naive General Matrix Multiplication

- Matrix A and B of a specific size.
- Uses BLAS format: [`C = alpha * (A*B) + beta*C`] with alpha=1, beta=0
- For simplicity: Calculating Matrix: `C = A * B`
- Computes each element of output matrix `C` independently
- Matrix multiplication is performed 20 times and the performance is averaged throughout
- Using matrices of size 32, 64, 128, ..., 8192

Naive GEMM Kernel Implementations in C++

C++

```cpp

__global__ void mat_mul_kernel_blas(float* A, float* B, float* C, int N1, int N2, int N3,
                                    float alpha, float beta)
{
    // Each thread computes one element C[row,col]
    int col = blockDim.x*blockIdx.x + threadIdx.x;
    int row = blockDim.y*blockIdx.y + threadIdx.y;


    // Parallel mat mul
    if (row < N1 && col < N3)
    {
        float value = 0;
        // Dot product of row from A with column from B
        for (int k = 0; k < N2; k++)
        {
            value += A[row*N2+k] * B[k*N3+col];
        }

        C[row*N3+col] = alpha * value + beta * C[row*N3+col];
    }
}
```

Naive GEMM Kernel Implementation in Rust

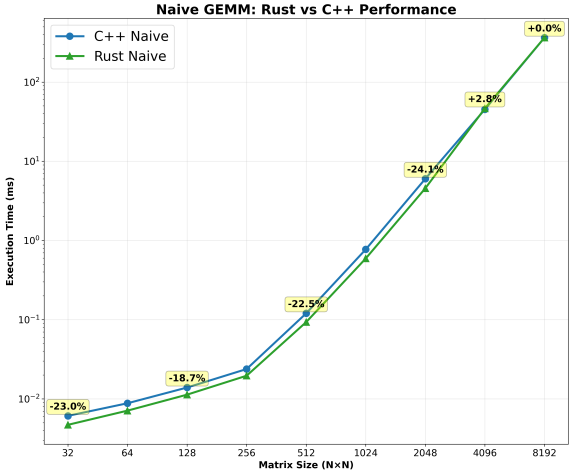
Rust

```rust
pub unsafe fn GEMM_naive(mat_a: *const f32, mat_b: *const f32, mat_c: *mut f32,
    m: usize, n: usize, k: usize, alpha: f32, beta: f32
) {
    // Each thread computes one element C[row,col]
    let col = (thread::block_dim_x() * thread::block_idx_x() + thread::thread_idx_x()) as usize;
    let row = (thread::block_dim_y() * thread::block_idx_y() + thread::thread_idx_y()) as usize;

    // Parallel mat mul
    if row < m && col < n {
        let mut sum = 0.0f32;
        // Dot product of row from A with column from B
        for i in 0..k {
            let a_val = *mat_a.add(row * k + i);
            let b_val = *mat_b.add(i * n + col);
            sum += a_val * b_val;
        }
        let elem = mat_c.add(row * n + col);
        *elem = alpha * sum + beta * *elem;
    }
}
```

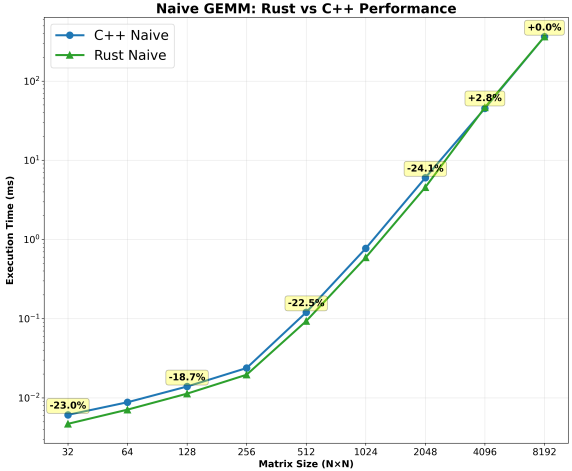
Results Naive GEMM Implementations

- Rust performance ca. 20% behind C++ until size 4096



Results Naive GEMM Implementations

- Rust performance ca. 20% behind C++ until size 4096
- At larger sizes near parity in performance



Tiled GEMM Implementations

Naive approach:

- Global Memory access is slow
- Same data is accessed frequently

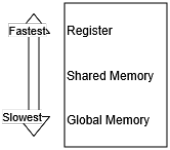


Figure: Memory hierarchy in GPUs

Tiled GEMM Implementations

Naive approach:

- Global Memory access is slow
- Same data is accessed frequently

Tiled approach:

- Load tiles into fast Shared Memory (once)
- All threads in block reuse data from Shared Memory
- Dramatic decrease in Global Memory access

=> This should significantly speed up processing time

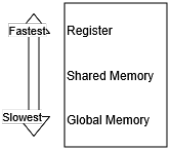
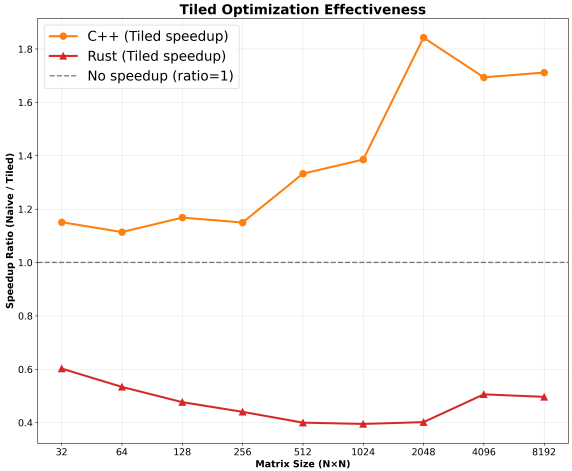


Figure: Memory hierarchy in GPUs

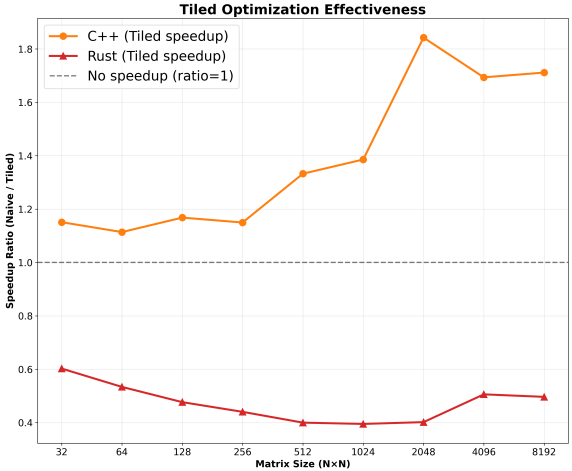
Results Tiled GEMM Implementations

- Improvement of 15% to over 82% for C++



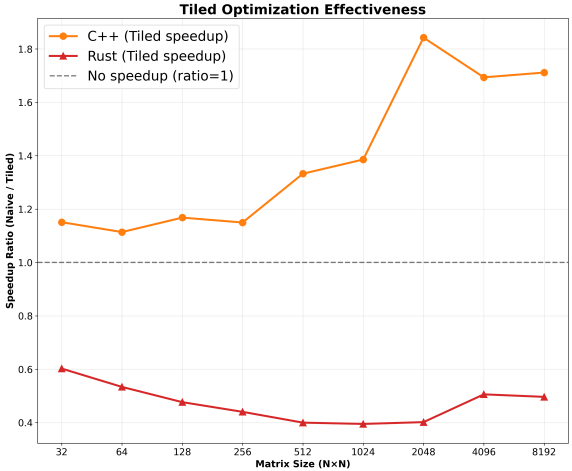
Results Tiled GEMM Implementations

- Improvement of 15% to over 82% for C++
- Performance reduction to only 40% in some cases for Rust implementation



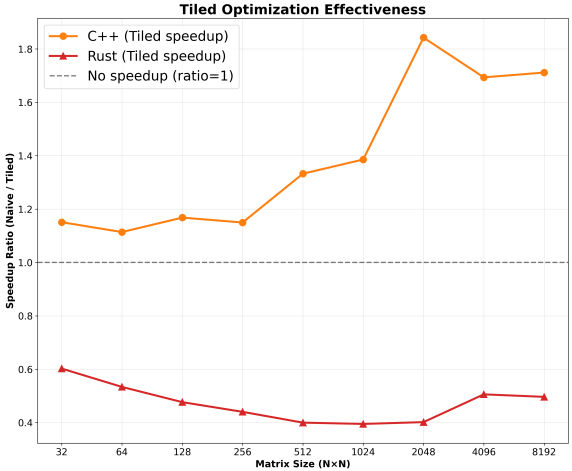
Results Tiled GEMM Implementations

- Improvement of 15% to over 82% for C++
- Performance reduction to only 40% in some cases for Rust implementation
- But why exactly does performance decrease?



Results Tiled GEMM Implementations

- Improvement of 15% to over 82% for C++
- Performance reduction to only 40% in some cases for Rust implementation
- But why exactly does performance decrease?
- $\Rightarrow$ **analyze the PTX Code**



PTX Analysis Rust

- Tiled Rust PTX creates branches that "trap" (terminate thread)

```
$L__BB1_29:
    trap;

$L__BB1_31:
    trap;

$L__BB1_20:
    trap;

$L__BB1_17:
    trap;
```

PTX Analysis Rust

- Tiled Rust PTX creates branches that "trap" (terminate thread)
- Main loop of the calculation consists of 3 branches

```
$L__BB1_28:
    add.s64     %rd96, %rd8, %rd110;
    setp.lt.u64     %p18, %rd96, 1024;
    @%p18 bra   $L__BB1_30;
    bra.uni     $L__BB1_29;

$L__BB1_30:
    add.s64     %rd110, %rd110, 1;
    add.s64     %rd108, %rd108, 32;
    setp.lt.u64     %p19, %rd108, 1024;
    @%p19 bra   $L__BB1_32;
    bra.uni     $L__BB1_31;

$L__BB1_32:
    add.s64     %rd41, %rd36, 128;
    add.s64     %rd42, %rd38, 4;
    ld.shared.f32   %f17, [%rd36];
    ld.shared.f32   %f18, [%rd38];
    fma.rn.f32  %f6, %f18, %f17, %f6;
    setp.lt.u64     %p20, %rd110, %rd32;
    mov.u64     %rd36, %rd41;
    mov.u64     %rd38, %rd42;
    @%p20 bra   $L__BB1_28;
```

PTX Analysis Rust

- Tiled Rust PTX creates branches that "trap" (terminate thread)
- Main loop of the calculation consists of 3 branches
- Checking for bounds occurs 2 times in the first two branches

```
$L__BB1_28:
    add.s64     %rd96, %rd8, %rd110;
    setp.lt.u64     %p18, %rd96, 1024;  Bounds check
    @%p18 bra   $L__BB1_30;             if in Bounds continue
    bra.uni     $L__BB1_29;             else jump to trap

$L__BB1_30:
    add.s64     %rd110, %rd110, 1;
    add.s64     %rd108, %rd108, 32;
    setp.lt.u64     %p19, %rd108, 1024; Bounds check
    @%p19 bra   $L__BB1_32;             if in Bounds continue
    bra.uni     $L__BB1_31;             else jump to trap

$L__BB1_32:
    add.s64     %rd41, %rd36, 128;
    add.s64     %rd42, %rd38, 4;
    ld.shared.f32   %f17, [%rd36];
    ld.shared.f32   %f18, [%rd38];
    fma.rn.f32  %f6, %f18, %f17, %f6;
    setp.lt.u64     %p20, %rd110, %rd32;
    mov.u64     %rd36, %rd41;
    mov.u64     %rd38, %rd42;
    @%p20 bra   $L__BB1_28;
```

PTX Analysis Rust

- Tiled Rust PTX creates branches that "trap" (terminate thread)
- Main loop of the calculation consists of 3 branches
- Checking for bounds occurs 2 times in the first two branches
- Final calculation and repeat of the loop if still needed

```
$L__BB1_28:
    add.s64     %rd96, %rd8, %rd110;
    setp.lt.u64     %p18, %rd96, 1024;  Bounds check
    @%p18 bra   $L__BB1_30;             if in Bounds continue
    bra.uni     $L__BB1_29;             else jump to trap

$L__BB1_30:
    add.s64     %rd110, %rd110, 1;
    add.s64     %rd108, %rd108, 32;
    setp.lt.u64     %p19, %rd108, 1024; Bounds check
    @%p19 bra   $L__BB1_32;             if in Bounds continue
    bra.uni     $L__BB1_31;             else jump to trap

$L__BB1_32:
    add.s64     %rd41, %rd36, 128;
    add.s64     %rd42, %rd38, 4;
    ld.shared.f32   %f17, [%rd36];      Loading from Shared Memory
    ld.shared.f32   %f18, [%rd38];      Loading from Shared Memory
    fma.rn.f32  %f6, %f18, %f17, %f6;   Addition on Register
    setp.lt.u64     %p20, %rd110, %rd32; Check if loop is done
    mov.u64     %rd36, %rd41;           Moving pointers
    mov.u64     %rd38, %rd42;           Moving pointers
    @%p20 bra   $L__BB1_28;             Return to start of loop
```

PTX Analysis Rust

- Tiled Rust PTX creates branches that "trap" (terminate thread)
- Main loop of the calculation consists of 3 branches
- Checking for bounds occurs 2 times in the first two branches
- Final calculation and repeat of the loop if still needed
- Bounds checks 2 times per iteration likely hinders performance

```
$L__BB1_28:
    add.s64     %rd96, %rd8, %rd110;
    setp.lt.u64     %p18, %rd96, 1024;  Bounds check
    @%p18 bra   $L__BB1_30;             if in Bounds continue
    bra.uni     $L__BB1_29;             else jump to trap

$L__BB1_30:
    add.s64     %rd110, %rd110, 1;
    add.s64     %rd108, %rd108, 32;
    setp.lt.u64     %p19, %rd108, 1024; Bounds check
    @%p19 bra   $L__BB1_32;             if in Bounds continue
    bra.uni     $L__BB1_31;             else jump to trap

$L__BB1_32:
    add.s64     %rd41, %rd36, 128;
    add.s64     %rd42, %rd38, 4;
    ld.shared.f32   %f17, [%rd36];      Loading from Shared Memory
    ld.shared.f32   %f18, [%rd38];      Loading from Shared Memory
    fma.rn.f32  %f6, %f18, %f17, %f6;   Addition on Register
    setp.lt.u64     %p20, %rd110, %rd32; Check if loop is done
    mov.u64     %rd36, %rd41;           Moving pointers
    mov.u64     %rd38, %rd42;           Moving pointers
    @%p20 bra   $L__BB1_28;             Return to start of loop
```

PTX Analysis C++

- Tiled C++ PTX code repeatedly adds from Shared Memory

```
$L__BB1_10:
    st.shared.f32   [%r7], %f118;
    bar.sync    0;
    ld.shared.f32   %f17, [%r9];
    ld.shared.f32   %f18, [%r8];
    fma.rn.f32  %f19, %f18, %f17, %f119;
    ld.shared.f32   %f20, [%r9+128];
    ld.shared.f32   %f21, [%r8+4];
    fma.rn.f32  %f22, %f21, %f20, %f19;
    ld.shared.f32   %f23, [%r9+256];
    ld.shared.f32   %f24, [%r8+8];
    fma.rn.f32  %f25, %f24, %f23, %f22;
    ld.shared.f32   %f26, [%r9+384];
    ld.shared.f32   %f27, [%r8+12];
    fma.rn.f32  %f28, %f27, %f26, %f25;
```

PTX Analysis C++

- Tiled C++ PTX code repeatedly adds from Shared Memory
- NVCC recognizes loop and performs all 32 calculations in one block

```
$L__BB1_10:
    st.shared.f32   [%r7], %f118;
    bar.sync    0;
    ld.shared.f32   %f17, [%r9];
    ld.shared.f32   %f18, [%r8];
    fma.rn.f32  %f19, %f18, %f17, %f119;
    ld.shared.f32   %f20, [%r9+128];
    ld.shared.f32   %f21, [%r8+4];
    fma.rn.f32  %f22, %f21, %f20, %f19;
    ld.shared.f32   %f23, [%r9+256];
    ld.shared.f32   %f24, [%r8+8];
    fma.rn.f32  %f25, %f24, %f23, %f22;
    ld.shared.f32   %f26, [%r9+384];
    ld.shared.f32   %f27, [%r8+12];
    fma.rn.f32  %f28, %f27, %f26, %f25;
```

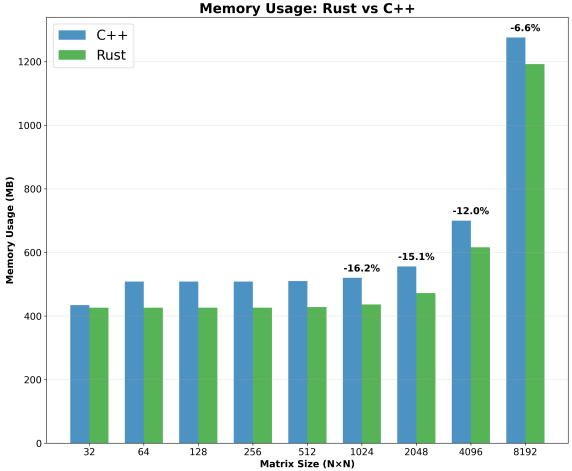
PTX Analysis C++

- Tiled C++ PTX code repeatedly adds from Shared Memory
- NVCC recognizes loop and performs all 32 calculations in one block
- Less jumping likely leads to speedup

```
$L__BB1_10:
    st.shared.f32   [%r7], %f118;
    bar.sync    0;
    ld.shared.f32   %f17, [%r9];
    ld.shared.f32   %f18, [%r8];
    fma.rn.f32  %f19, %f18, %f17, %f119;
    ld.shared.f32   %f20, [%r9+128];
    ld.shared.f32   %f21, [%r8+4];
    fma.rn.f32  %f22, %f21, %f20, %f19;
    ld.shared.f32   %f23, [%r9+256];
    ld.shared.f32   %f24, [%r8+8];
    fma.rn.f32  %f25, %f24, %f23, %f22;
    ld.shared.f32   %f26, [%r9+384];
    ld.shared.f32   %f27, [%r8+12];
    fma.rn.f32  %f28, %f27, %f26, %f25;
```

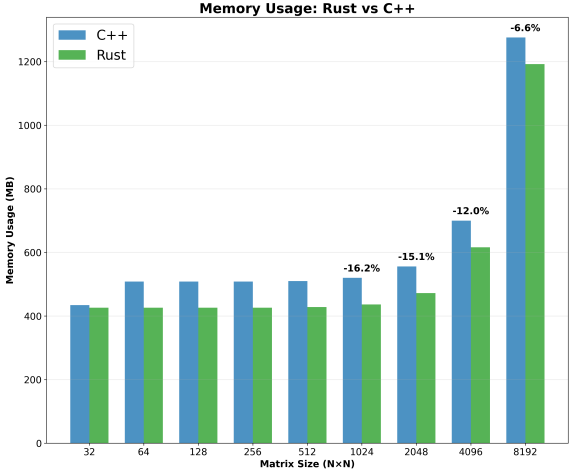
Memory Usage Analysis

- Rust consistently uses less memory than C++



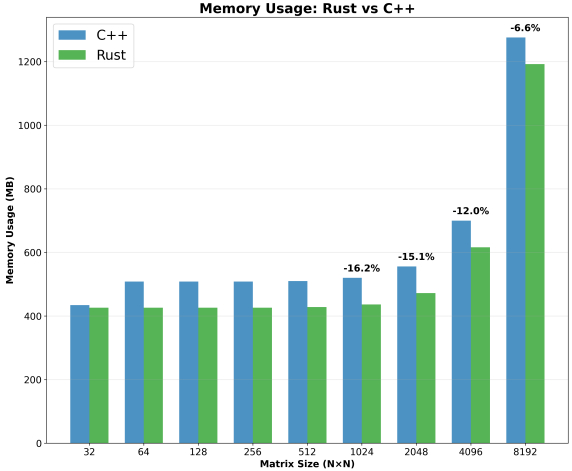
Memory Usage Analysis

- Rust consistently uses less memory than C++
- At 8192×8192 Rust uses 1,192 MB vs C++ 1,276 MB (-6.6%)



Memory Usage Analysis

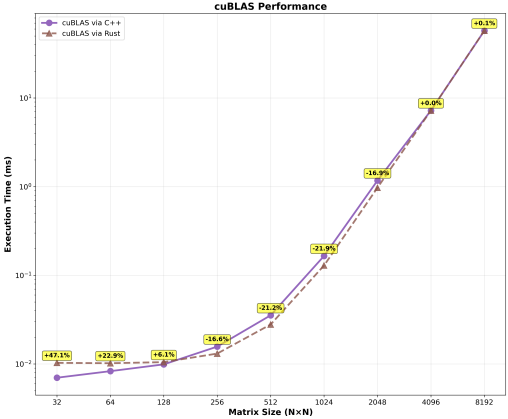
- Rust consistently uses less memory than C++
- At 8192×8192 Rust uses 1,192 MB vs C++ 1,276 MB (-6.6%)
- Possible memory overhead in the C++ code



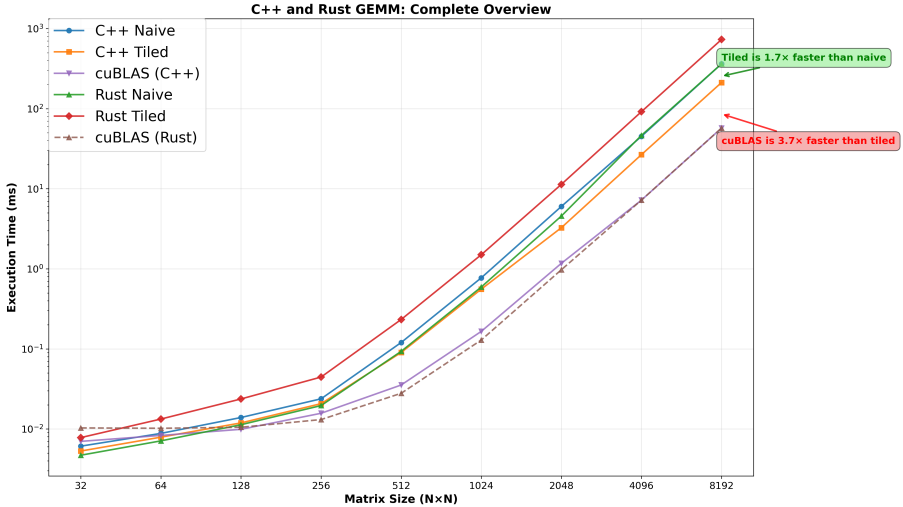
cuBLAS Performance Comparison

cuBLAS: Highly optimized library for linear algebra by Nvidia

- At size 8192 both Rust and C++ achieve 57ms
- cuBLAS integration in Rust seems to be good since they perform near identical



Final GEMM Performance Comparison



Outline

Limitations

- Rust CUDA is still in **early** development
- C++ CUDA is much more matured
- Rust CUDA currently **requires** nightly build (unstable)
- Some CUDA specific libraries are missing
- Rust CUDA will always lag behind in new features being added
- Rust CUDA currently does not work on Windows

Challenges

- Almost no resources available for Rust CUDA
- Especially not much on optimization
- CUDA has many high quality resources available
- C++ CUDA ran without any initial problems on HPC
- Rust encountered issues with accessing the nightly build on HPC

Outline

Conclusion

Rust has ...

- ... better memory safety with little or no performance penalty
- ... better memory efficiency
- ... a growing ecosystem and tooling
- ... for now little resources to learn
- ... not quite matured as much as C++ CUDA yet