Matrix multiplication

- Normal Multiplication
- Pure Thread
- Cache Optimization
- SIMD

Jacky 6580244

Sareena 6481197

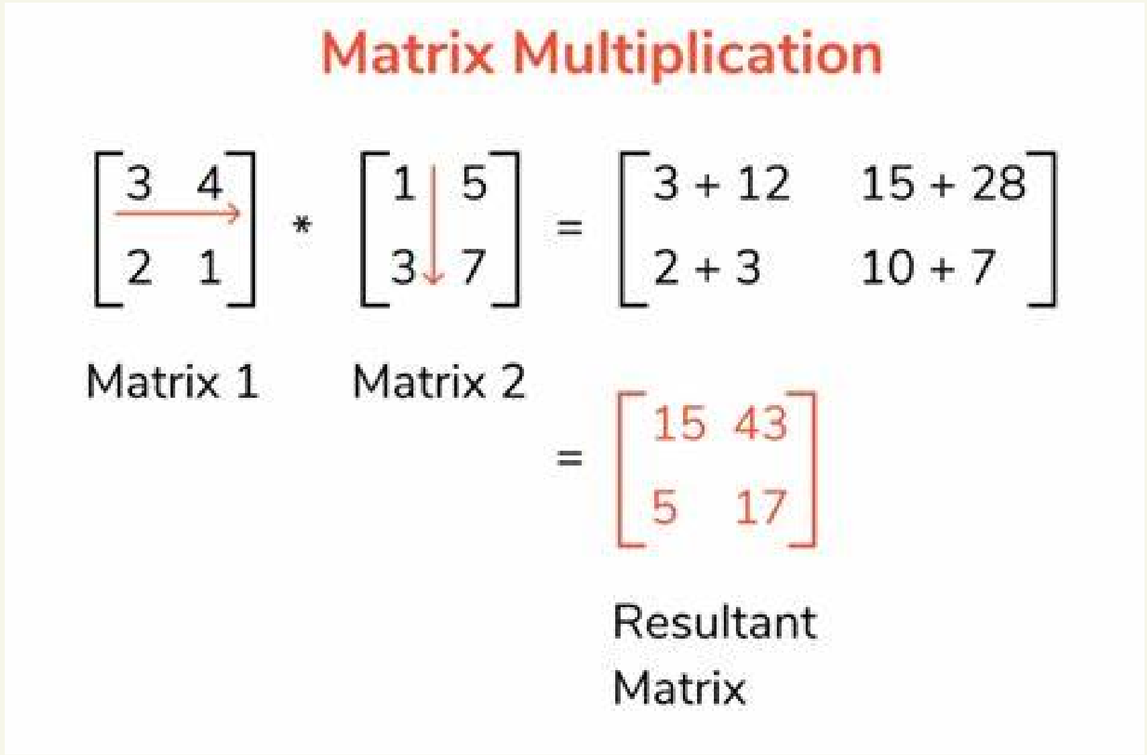
Introduction

In this project, our goal is to explore and optimize various matrix multiplication methods in Rust, emphasizing performance and efficiency. By implementing normal multiplication, pure thread, cache optimized, and SIMD.

How we represent a matrix

```rust
pub type Matrix = Vec<Vec<i32>>;
```

How to compute the matrix multiplication:



Normal matrix multiplication implementation:

```rust
pub type Matrix = Vec<Vec<i32>>;

pub fn matrix(a: &Matrix, b: &Matrix) -> Option<Matrix> {
    let a_row: usize = a.len();
    let b_col: usize = b[0].len();
    let a_col: usize = a[0].len();
    let mut result: Vec<Vec<i32>> = vec![vec![0; b_col]; a_row];

    if a_col != b.len() {
        return None;
    }

    for i: usize in 0..a_row {
        for j: usize in 0..b_col {
            result[i][j] = (0..a_col).map(|k: usize| a[i][k] * b[k][j]).sum();


        }
    }
    Some(result)
}
```

dot product calculations here

Threaded implementation:

```rust
pub fn matrix(a: &Matrix, b: &Matrix) -> Option<Matrix> {
    let a_row: usize = a.len();
    let b_col: usize = b[0].len();
    let a_col: usize = a[0].len();
    let mut result: Vec<Vec<i32>> = vec![vec![0; b_col]; a_row];
    if a_col != b.len() {
        return None;
    }
    let mut handles: Vec<JoinHandle<Vec<i32>>> = vec![];
    for i: usize in 0..a_row {
        let a: Vec<Vec<i32>> = a.clone();
        let b: Vec<Vec<i32>> = b.clone();
        let mut result_row: Vec<i32> = result[i].clone();
        let handle: JoinHandle<Vec<i32>> = thread :: spawn(move || {

            for j: usize in 0..b_col {
                result_row[j] = (0..a_col).map(|k: usize| a[i][k] * b[k][j]).sum();
            }
            result_row
        });
        handles.push(handle);
    }
    for (i: usize, handle: JoinHandle<Vec<i32>>) in handles.into_iter().enumerate() {
        result[i] = handle.join().unwrap();
    }
```

each thread will be responsible for one dot product

Work & Span:

Work: O(n^3)

Span: O(n)

cache optimization implementation:

```rust
pub fn matrix(a: &Matrix, b: &Matrix, block_size: usize) -> Option<Matrix> {
    let a_row: usize = a.len();
    let b_col: usize = b[0].len();
    let a_col: usize = a[0].len();
    let mut result: Vec<Vec<i32>> = vec![vec![0; b_col]; a_row];
    //the resulting size of matrix multi is a's row, b's col
    if a_col != b.len() {
        //the condi for multi together
        return None;
    }
    for ii: usize in (0..a_row).step_by(step: block_size) {
        for jj: usize in (0..b_col).step_by(step: block_size) {
            for kk: usize in (0..a_col).step_by(step: block_size) {
                for i: usize in ii..(ii + block_size).min(a_row) {
                    for j: usize in jj..(jj + block_size).min(b_col) {
                        for k: usize in kk..(kk + block_size).min(a_col) {
                            result[i][j] += a[i][k] * b[k][j];
                        }
                    }
                }
            }
        }
    }

    Some(result)
} fn matrix
```

where we seperate the matrix into smaller block, and do dot product within each block

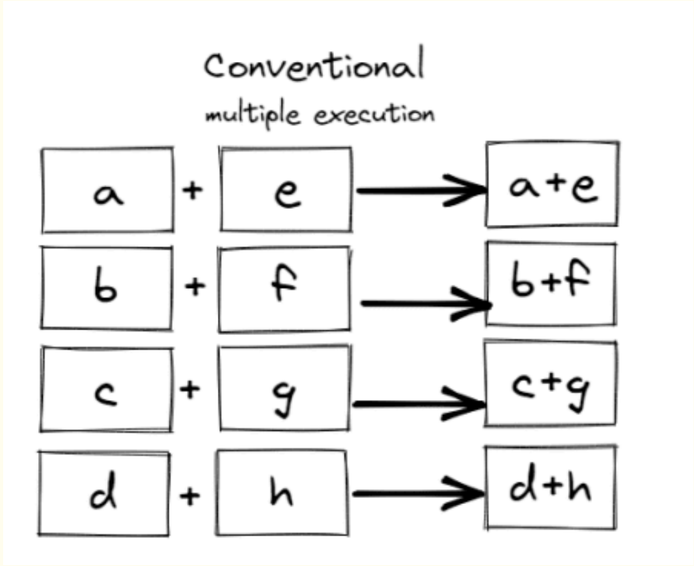
SIMD implementation:

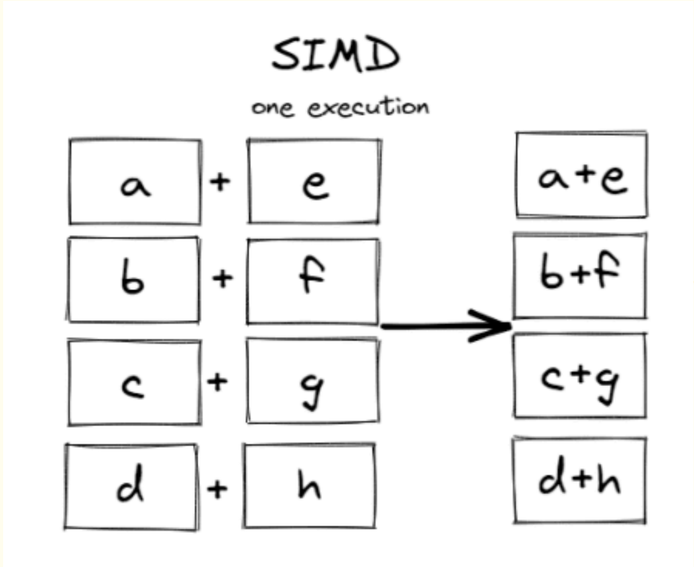
```
unsafe {
    // Allocate memory for matrix_a and matrix_b
    let layout: Layout = Layout::from_size_align(size: a_rows * a_cols * std::mem::size_of::<i32>()
                                                 , align: std::mem::align_of::<i32>()).unwrap();
    let matrix_a: *mut i32 = alloc(layout) as *mut i32;
    let matrix_b: *mut i32 = alloc(layout) as *mut i32;

    // Initialize matrix_a and transpose matrix_b
    for i: usize in 0..a_rows {
        for j: usize in 0..a_cols {
            *matrix_a.add(count: i * a_cols + j) = a[i][j];
            *matrix_b.add(count: j * b_cols + i) = b[i][j]; // Transpose matrix b for better memory access
        }
// Perform matrix multiplication with SIMD
for i: usize in 0..a_rows {
    for j: usize in 0..b_cols {
        let mut sum: __m128i = _mm_setzero_si128();
        for k: usize in (0..a_cols).step_by(step: 4) {
            let va: __m128i = _mm_loadu_si128(mem_addr: matrix_a.add(count: i * a_cols + k) as *const __m128i);
            let vb: __m128i = _mm_loadu_si128(mem_addr: matrix_b.add(count: j * a_cols + k) as *const __m128i);

            let product: __m128i = _mm_mullo_epi32(a: va, b: vb);
            sum = _mm_add_epi32(a: sum, b: product);
        }

        // Horizontally add the elements of sum
        let mut temp: i32 = _mm_extract_epi32(sum, 0) + _mm_extract_epi32(sum, 1) + _mm_extract_epi32(sum, 2) + _mm_extract_epi32(sum, 3);
        temp = temp + (temp << 16);
        temp = temp + (temp << 8);
        result[i][j] = temp >> 24;
    }
}
```

- Tranpose b is done to optimize memory access patterns when do multiplication
- Create a register to accumulate the result by make a use of 128-bit SIMD
- Loading chunk of matrix data into accumulators (SIMD)
- After we accumulated the result, we horizontally add it in matrix the form.

Key things

- Parallelism - using SIMD enables to processing matrix elements parallel.
- SIMD instructions can operate on multiple elements of the matrices at once.(_mm_mullo_epi32, _mm_add_epi32, etc.)
- Tranposing can help to minimize cache misses and maximize data locality. Effectively, SIMD contiguous memory locations, CPU can fetch data quickly without waiting for slow memory accesses.





Success measurement:

```rust
let n: usize = 2000; // Size of the matrix
let a: Vec<Vec<i32>> = generate_random_matrix(n);
let b: Vec<Vec<i32>> = generate_random_matrix(n);
```

```rust
// Normal multiplication
let start: Instant = Instant::now();
let _ = normal_mult::matrix(&a, &b);
let duration: Duration = start.elapsed();
println!("Normal multiplication took: {:?}", duration);
```

```
PS E:\Rust\matrix_multiplication_project> cargo run --release
    Finished `release` profile [optimized] target(s) in 0.05s
     Running `target\release\matrix_multiplication_project.exe`
Normal multiplication took: 21.9256392s
Cache-optimized multiplication took: 12.4735637s
Pure threads took: 10.0986357s
SIMD multiplication took: 2.8324188s
PS E:\Rust\matrix_multiplication_project>
```

- we use 2000*2000 size of matrix to enlarge the performance showing
- we perform the same timed process to each method, and make sure they are calculating the same matrix multiplication

what do we learn from this project?

- Understanding SIMD (Single Instruction, Multiple Data). Usage and benefits
- Memory management optimization -using unsafe
- learn how to handle the low-level memory management safely
- using transpose to reduce cache misses (improve cache efficiency) or narrow down the size of matrix into small block size
- Parallelism - learning how to parallize the effectively using SIMD, and getting into more in parallel concepts, which requires for high-performance computing besides matrix.

THANK YOU