Poređenje implementacija i performansi algoritama množenja matrica u programskim jezicima Julia i Rust

Nikola Kušlaković E2 121/2021

Uvod

Množenje matrica predstavlja fundamentalnu operaciju i aritmetičko jezgro brojnih naučnih i inženjerskih disciplina, uključujući mašinsko učenje (posebno u treningu dubokih neuronskih mreža), numeričke simulacije kompleksnih sistema, obradu signala i računarsku grafiku. Efikasna implementacija ove operacije je od kritične važnosti za performanse celokupnih sistema, jer množenje matrica često predstavlja dominantni faktor u vremenu izvršavanja. Zbog svoje centralne uloge, aktivno se istražuju i primenjuju različiti algoritmi i optimizacione tehnike. Poseban fokus je stavljen na prilagođavanje algoritama specifičnim tipovima matrica, kao što su guste i retke matrice. Stoga, izbor i optimizacija algoritma direktno utiču na mogućnost rešavanja problema velikih dimenzija.

U poslednje vreme, programski jezici Julia i Rust privlače značajnu pažnju u domenu računarstva visokih performansi. Julia, kao dinamički jezik sa JIT (Just-In-Time) programskim prevodiocem, dizajnirana je za primene u načnim istraživanjima i proračunima i teži da pruži "brzinu C-a uz lakoću pisanja koda kao u Python-u". S druge strane, Rust je statički tipiziran jezik čije su glavne karakteristike dobre performanse, sigurnost pri upravljanju memorijom i dobar dizajn samog programskog prevodioca koji sprečava greške već u fazi prevođenja.

Ovaj rad ima za cilj da uporedi implementacije algoritama za množenje matrica u jezicima Julia i Rust, kao i da analizira njihove performanse kroz merenja nad različitim veličinama matrica. Fokus implementacije i analize biće na iterativnom (klasičnom) algoritmu, metodi zavadi-pa-vladaji (eng. *Divide and Conquer*), kao i na Štrasenovom algoritmu. Kroz analizu, uporedićemo kako karakteristike oba jezika utiču na brzinu i efikasnost operacije množenja matrica.

Teorijske osnove

U ovom poglavlju analizirana je vremenska složenost primenjenih algoritama. Za analizu algoritama zavadi-pa-vladaj i Štrasenovog algoritma koji se u praksi najčešće implementiraju rekurzivno, korišćene su rekurentne relacije, čija su rešenja dobijena primenom master teoreme [1].

U nastavku, $T(n)$ označava funkciju vremenske složenosti, odnosno broj elementarnih operacija potrebnih za izvršavanje algoritma nad kvadratnom matricom dimenzije $n \times n$

Iterativni algoritam

Iterativni (klasični) algoritam podrazumeva direktnu implementaciju množenja matrica kroz tri ugnježdene petlje. Kako svaka od tri petlje vrši iteraciju od 1 do n ukupan broj operacija proporcionalan je trećem stepenu dimenzije matrice. Stoga je asimptotska složenost ovog pristupa $O(n^3)$.

Algoritam zavadi-pa-vladaj

Ovaj pristup se zasniva na rekurzivnoj podeli matrica dimenzije $n \times n$ na četiri podmatrice dimenzije $n \times n$. Algoritam zahteva 8 rekurzivnih množenja tih podmatrica i njihovo naknadno sabiranje. Vreme potrebno za sabiranje matrica je kvadratno, odnosno $O(n^2)$.

Rekurentna relacija koja opisuje ovaj algoritam glasi:

$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

Primenom master teoreme na ovu relaciju, dobija se da dominantni faktor ostaje broj rekurzivnih poziva, te je konačna složenost ista kao i kod iterativnog pristupa:

$$T(n) = O(n^3)$$

Štrasenov algoritam

Štrasenov algoritam [2] optimizuje prethodni pristup smanjenjem broja skupih operacija množenja. Korišćenjem specifičnih linearnih kombinacija podmatrica, ovaj algoritam uspeva da izračuna rezultat koristeći samo 7 rekurzivnih množenja, umesto 8. Iako je broj operacija sabiranja i oduzimanja povećan, on i dalje ostaje u okviru $O(n^2)$.

Rekurentna relacija za Štrasenov algoritam je:

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

Na osnovu master teoreme, smanjenje faktora grananja rekurzije dovodi do manje asimptotske složenosti:

$$T(n) = O(n^{log_2 7}) \approx O(n^{2.81})$$

Implementacije u Juliji

Programski jezik Julia N-dimenzionalni niz implemetira kroz tip podatka `Array{T, N}`, gde je `T` neki osnovni tip (`Float64`, `Int64`, `String` itd.), a `N` je dimenzija datog niza. Pošto se u praksi jako često koriste 1D i 2D niz, u Juliji su oni imenovani kao `Vector{T}` i `Matrix{T}` što je samo drugi naziv za `Array{T, 1}` i `Array{T, 2}`. Ovo je značajna prednost u odnosu na Rust, gde ne postoji nativni tip za matricu. Jako bitna karakterstika Julije jeste da se matrice u memoriji skladište kontinualno po kolonama, a ne po redovima. Takođe, ovi tipovi imaju neke korisne funkcije i implementirane operatore za indeksiranje, sabiranje, oduzimanje, deljenje i sl. [3].

Iterativni algoritma

Na isečku koda 1 vidimo implementaciju iterativnog algoritma, gde je redosled petlji prilagođen načinu skladištenja matrice u memoriji. Funckija prima tip podatka `AbstractMatrix`, a `Matrix` tip je podtip tog tipa. Ovo je urađeno kako bi ova funkcija mogla da prima i sve tipove matrica, kao i poglede (eng. *slice*) neke određene matrice.

```julia
function iterative_matmul(A::AbstractMatrix{T}, B::AbstractMatrix{T})
where T
  m, n = size(A)
  q, p = size(B)

  if n != q
    throw(DimensionMismatch("Matrix dimensions must agree: A is
$(m)x$(n), B is $(q)x$(p)"))
  end

  C = zeros(T, m, p)

  # Column major computation (for better cache performance)
  # Julia uses column-major order by default for storing matrices
  for j in 1:p      # column of C
    for k in 1:n    # accumulation
      for i in 1:m  # row of C
        C[i, j] += A[i, k] * B[k, j]
      end
    end
  end

  return C
end
```

Isečak koda 1: Implementacija iterativnog algoritma za množenje matrica (Julia)

Zavadi-pa-vladaj i Štrasenov algoritam

Na isečku koda 2 vidimo rekurzivnu implementaciju zavadi-pa-vladaj algoritma. Uz pomoć `@view` makroa kreira se pogled (isečak) matrice bez kopiranja matrice. Paralelizacija je izvršena uz pomoć `Base.Threads` ugrađene biblioteke [4]. Biblioteka u pozadini prvo alocira N sistemskih niti. Uz pomoć `@spawn` makroa kreiraju se zadaci (virtuelne niti) koje planer niti dodeljuje sistemski nitima na izvršavanje. Poziv funkcije `fetch` blokira izvršavanje dok se zadatak ne izvrši. Zadataka može biti znatno više od sistemskih niti. Ovaj princip rada je znatno pogodan za pralelizaciju algoritama koji su implementirani kroz rekurziju. Parametar `threshold` je granica na osnovu koje se

odlučuje kada je problem dovoljno mali (dimenzija matrica), da bi se prestalo sa deljenjem i tada pozvao standardni iterativni algoritam. Ukoliko je `threshold=1` množenje se svodi na množenje dva skalara. Ukoliko matrice imaju dimenzije koje nisu stepen dvojke, urađeno je proširivanje matrice na dimenziju koja je najbliža nekom stepenu dvojke. Ovo je urađeno za obe matrice pre poziva rekurzivne funkcije. Ovaj pristup prilagođavanja algoritma na sve dimenzije matrica je izabran zato što je najjednostavniji i ne komplikuje implementaciju samog algoritma.

```
function _divide_conquer_recursive(A::AbstractMatrix{T},
B::AbstractMatrix{T}, threshold::Int) where T
  m, n = size(A)
  q, p = size(B)

  # Base case: use our own iterative implementation for fair
comparison
  if m <= threshold || n <= threshold || p <= threshold
    return iterative_matmul(A, B)
  end

  # Ensure matrices can be divided evenly (pad if necessary)
  m_half = m ÷ 2
  n_half = n ÷ 2
  p_half = p ÷ 2

  # Divide A into quadrants
  A11 = @view A[1:m_half, 1:n_half]
  A12 = @view A[1:m_half, n_half+1:n]
  A21 = @view A[m_half+1:m, 1:n_half]
  A22 = @view A[m_half+1:m, n_half+1:n]

  # Divide B into quadrants
  B11 = @view B[1:n_half, 1:p_half]
  B12 = @view B[1:n_half, p_half+1:p]
  B21 = @view B[n_half+1:n, 1:p_half]
  B22 = @view B[n_half+1:n, p_half+1:p]

  # Parallel computation using threads
  # Spawn 8 tasks for the 8 required products
  t1 = Threads.@spawn _divide_conquer_recursive(A11, B11, threshold)
```

```
  t2 = Threads.@spawn _divide_conquer_recursive(A12, B21, threshold)
  t3 = Threads.@spawn _divide_conquer_recursive(A11, B12, threshold)
  t4 = Threads.@spawn _divide_conquer_recursive(A12, B22, threshold)
  t5 = Threads.@spawn _divide_conquer_recursive(A21, B11, threshold)
  t6 = Threads.@spawn _divide_conquer_recursive(A22, B21, threshold)
  t7 = Threads.@spawn _divide_conquer_recursive(A21, B12, threshold)
  t8 = Threads.@spawn _divide_conquer_recursive(A22, B22, threshold)

  # Fetch results and combine
  C11 = fetch(t1) + fetch(t2)
  C12 = fetch(t3) + fetch(t4)
  C21 = fetch(t5) + fetch(t6)
  C22 = fetch(t7) + fetch(t8)

  # Combine quadrants
  C = [C11 C12; C21 C22]

  return C
end
```

Isečak koda 2: Implementacija zavadi-pa-vladaj algoritma (Julia)

Štrasenov algoritam implementiran je rekurzivno po sličnom principu kao i zavadi-pa-vladaj algoritam, gde su korišćeni isti primitivi za paralelizaciju.

Izvorno množenje matrica u Juliji

Bitno je istaći da je množenje matrica izvorno podržana u programskom jeziku Julia. kroz operator “*”. Julia tada poziva BLAS (Basic Linear Algebra Subprograms) [5] biblioteke. BLAS predstavlja standard rutina koje su decenijama optimizovane u C-u i Fortranu kako bi maksimalno iskoristile hardverske resurse, uključujući vektorizaciju (SIMD), efikasno korišćenje procesorske keš memorije i izvršavanje u vidu više niti. Julia ovime postiže performanse identične onima u jezicima kao što su C++ ili Fortran.

Dodatnu efikasnost pruža u Julijin sistem odlučivanja pri množenju koji na osnovu dimenzija i specifičnog tipa matrica (poput gustih, retkih ili simetričnih) inteligentno bira optimalnu rutinu za dati zadatak. Na primer, za veoma male matrice Julia može koristiti specijalizovane algoritme unutar samog jezika kako bi izbegla troškove pozivanja

spoljnih biblioteka, dok za ogromne retke matrice koristi algoritme koji preskaču množenje nulama, čime se postižu maksimalne performanse u svim scenarijima [6].

Implementacije u Rustu

Programski jezik Rust u svojoj osnovi nema podržan tip podataka za reprezentaciju matrice, niti često korišćene funkcije i operatore koji se koriste sa matricama (sabiranje, oduzimanje, deljenje itd.). Zbog toga bilo je prvo potrebno implementirati ovakav tip podatka. Na isečku koda 3 vidimo strukturu koja predstavlja matricu kao i potipise funkcija koje se koriste uz ovaj tip podatka. Struktura `Matrix` sadrži niz svih elemenata i dimenzije matrice kao svoja polja. Zbog konzistentnosti sa Julijom, elementi matrice se skladište kontinualno kao jednodimenzioni niz po kolonama. Osim funkcija, implementirani su operatori za indeksiranje i izmenu elemenata matrice. Metode `submatrix i combine_quadrants` kreiraju nove matrice i vraćaju ih kao povratne vrednosti. Metode `add i sub` rade sabiranje i oduzimanje dve matrice modifikujući matricu bez kopiranja (in-place modifikacija). Obe metode vraćaju mutabilnu reference matrice kako bi mogle da se uvezuju jedna za drugom. Ovo je implementirano zarad jednostavnosti implementacije algoritama množenja matrica. Kao i u Juliji, ukoliko matrica nije imala dimenzije nekog stepena dvojke, bilo je urađeno proširivanje te matrice pre poziva rekurzivne funkcije.

```
#[derive(Clone, Debug)]
pub struct Matrix {
      pub data: Vec<f64>,
      pub rows: usize,
      pub cols: usize,
}

impl Matrix {
      /// Create a new matrix with given dimensions, initialized to
zero
      pub fn new(rows: usize, cols: usize) -> Self { ... }

      /// Create a random matrix with values in [0, 1)
      pub fn random(rows: usize, cols: usize) -> Self { ... }

      /// Extract a submatrix (creates a copy)
      pub fn submatrix(
```

```
            &self,
            row_start: usize,
            row_end: usize,
            col_start: usize,
            col_end: usize,
      ) -> Matrix { ... }

      /// Add two matrices element-wise
      pub fn add(&mut self, other: &Matrix) -> &mut Self
      { ... }

      /// Subtract two matrices element-wise
      pub fn sub(&mut self, other: &Matrix) -> &mut Self
      { ... }

      /// Pad matrix with zeros to new dimensions
      pub fn pad(&self, new_rows: usize, new_cols: usize) -> Matrix
      { ... }

      /// Combine four quadrant matrices into a single matrix
      pub fn combine_quadrants(c11: &Matrix, c12: &Matrix, c21:
&Matrix, c22: &Matrix) -> Matrix { ... }

// Implement indexing: matrix[(row, col)]
impl Index<(usize, usize)> for Matrix {
      type Output = f64;

      #[inline]
      fn index(&self, (row, col): (usize, usize)) -> &Self::Output {
            &self.data[col * self.rows + row]
      }
}

// Implement mutable indexing: matrix[(row, col)] = value
impl IndexMut<(usize, usize)> for Matrix {
      #[inline]
      fn index_mut(&mut self, (row, col): (usize, usize)) -> &mut
Self::Output {
            &mut self.data[col * self.rows + row]
```

```
    }
}
```

Isečak koda 3: Implementacija strukture matrica, funkcija i operatora te strukture (Rust)

Iterativni algoritam

Iterativni algoritam u Rustu je imao isti broj linija koda kao i algoritam u Juliji, gde su čitljivosti koda pomogle unapred definisane metode iz strukture `Matrix`.

Zavadi-pa-vladaj i Štrasenov algoritam

Na isečku koda 4 vidimo algoritam zavadi-pa-vladaj koji je implementiran rekurzivno, a za paralelizaciju je korišćena bublioteka `Rayon` [7] koja radi po sličnom principu kao i `Base.Threads` u Juliji. Broj sistemskih niti je unapred definisan tokom izvršavanja, a sama biblioteka se bavi kreiranjem i upravljanjem virtuelnih niti i njihovim dodeljivanjem sistemskim nitima.

```
fn divide_conquer_recursive(a: &Matrix, b: &Matrix, threshold: usize)
-> Matrix {
    let m = a.rows;
    let n = a.cols;
    let p = b.cols;

    // Base case: use iterative implementation for fair comparison
    if m <= threshold || n <= threshold || p <= threshold {
        return iterative_matmul(a, b);
    }

    // Divide matrices into quadrants
    let m_half = m / 2;
    let n_half = n / 2;
    let p_half = p / 2;
```

```
    // Divide A into quadrants
    let a11 = a.submatrix(0, m_half, 0, n_half);
    let a12 = a.submatrix(0, m_half, n_half, n);
    let a21 = a.submatrix(m_half, m, 0, n_half);
    let a22 = a.submatrix(m_half, m, n_half, n);

    // Divide B into quadrants
    let b11 = b.submatrix(0, n_half, 0, p_half);
    let b12 = b.submatrix(0, n_half, p_half, p);
    let b21 = b.submatrix(n_half, n, 0, p_half);
    let b22 = b.submatrix(n_half, n, p_half, p);

    // Parallel computation using Rayon
    let results: Vec<Matrix> = vec![
        (&a11, &b11),
        (&a12, &b21),
        (&a11, &b12),
        (&a12, &b22),
        (&a21, &b11),
        (&a22, &b21),
        (&a21, &b12),
        (&a22, &b22),
    ]
    .into_par_iter()
    .map(|(a_sub, b_sub)| divide_conquer_recursive(a_sub, b_sub,
threshold))
    .collect();

    let [mut c11,
        r1,
        mut c12,
        r3,
        mut c21,
        r5,
        mut c22,
        r7]: [Matrix; 8] =
        results.try_into().unwrap();

    c11.add(&r1);
```

```
        c12.add(&r3);
        c21.add(&r5);
        c22.add(&r7);

        // Combine quadrants
        Matrix::combine_quadrants(&c11, &c12, &c21, &c22)
}
```

Isečak koda 4: Implementacija strukture matrica, funkcija i operatora te strukture (Rust)

Štrasenov algoritam implementiran je rekurzivno po sličnom principu kao i zavadi-pa-vladaj algoritam, gde je korišćena ista biblioteka za paralelizaciju. Bilo je potrebno optimizovati redosled instrukcija kako bi se smanjilo kreiranje međurezultata.

Korišćenje `std::thread` umesto `Rayon` biblioteke

Korišćenje `std::thread` za rekurzivne algoritme poput Štrasenovog je inicijalno neefikasno zbog direktnog mapiranja na niti operativnog sistema, što pri dubokoj rekurziji neizbežno dovodi do alociranje velikog broja niti, iscrpljivanja memorijskih resursa i drastičnog pada performansi usled preteranog menjanja konteksta. Za razliku od Rayon-a koji automatski balansira opterećenje, sistemske niti nemaju pametan mehanizam dodele zadataka, pa je za njihovu efikasnu primenu neophodno ručno implementirati neki hibridni pristup: ograničiti dubinu paralelizacije kako bi se nakon nekoliko nivoa prešlo na sekvencijalni rad i obavezno koristiti `std::thread::scope` kako bi se omogućilo bezbedno deljenje referenci matrica bez skupog kopiranja podataka između niti. Takođe, algoritme bismo mogli implementirati iterativno umesto rekurzivno koristeći eksplicitni stek ili strukturu za praćenje zavisnosti, čime bismo sistemsku rekurziju zamenili nizom diskretnih zadataka. Ipak, ovakav pristup bi drastično zakomplikovao implementaciju jer zahteva ručno upravljanje stanjem i sinhronizaciju spajanja rezultata, što se u rekurzivnom modelu dešava automatski.

Metod merenja

Svaki algoritam u oba programska jezika bio je testiran za različite veličine kvadratnih matrica. Svaki algoritam je bio pokrenut više puta za istu veličinu matrica i izračunata je srednja vrenost svih ponovljenih merenja. Srednje vrednosti su uzete kao konačne i predstavljene su u poglavlju za rezultate i diskusiju.

Merenje u Julia programskom jeziku bilo je urađeno uz pomoć biblioteke `BenchmarkTools` [8]. Isečak koda 5 pokazuje kako se merenje algoritma vršilo. U Juliji funkcije mogu da se prosleđuju kao parametar drugim funkcijama. Ključna reč `@benchmark` je makro koji poziva funkciju za n uzoraka (`samples`) i za svaki uzorak vrši k poziv funkcije (`evals`) pre nego što zabeleži vrednost merenja. Konačno vreme i ukupna zauzeta memorija su izračunati kao srednja vrednost svih uzoraka. Memorija koju ovaj biblioteka meri je zazueće heap memorije od strane procesa sakupljača otpada (eng. *Garbage collector*). Bitno je napomentu da zbog JIT prevođenja, ovaj makro pre svih merenja vrši jedno dodatno pokretanje funkcije (*warmup* faza) u kojoj se funkcija prvi put prevodi.

```
function benchmark_algorithm(algorithm_fn, name::String,
A::Matrix{T}, B::Matrix{T}; kwargs...) where T
  println("  Benchmarking $name...")

  trial = @benchmark $algorithm_fn($A, $B; $(kwargs)...) samples = 10
evals = 1

  mean_time = mean(trial.times) / 1e6  # Convert to milliseconds
  mean_memory = mean(trial.memory) / 1e6  # Convert to megabytes

  return (mean_time, mean_memory)
end
```

Isečak koda 5: Funkcije za merenje algoritama (Julia)

U Rustu je korišćen sličan princip samo što je za merenje alocirane heap memorije bila korišćena biblioteka `stats_alloc` [9]. Ukratko, ova biblioteka radi po principu instrumentacije kod, gde se svaki poziv koji alocira memorije presreće i beleži se koliko memorije je ukupno alocirano. Vreme izvršavanja je bilo mereno uz pomoć ugrađene `std::time` biblioteke.

```
fn benchmark_algorithm<F>(algorithm_fn: F, name: &str, a: &Matrix, b:
&Matrix) -> (f64, f64)
where
      F: Fn(&Matrix, &Matrix) -> Matrix,
{
      println!("  Benchmarking {}...", name);
```

```
    // Benchmark runs
    let samples = 10;
    let mut times = Vec::with_capacity(samples);
    let mut memory_allocations = Vec::with_capacity(samples);

    for _ in 0..samples {
        let reg: Region<'_, std::alloc::System> =
Region::new(&INSTRUMENTED_SYSTEM);
        let start = Instant::now();
        let _ = algorithm_fn(a, b);
        let duration = start.elapsed();
        let stats = reg.change();

        times.push(duration.as_secs_f64() * 1000.0); // Convert to
milliseconds
        memory_allocations.push(stats.bytes_allocated as f64 /
1e6); // Convert to megabytes
    }

    let mean_time = calculate_mean(&times);
    let mean_memory = calculate_mean(&memory_allocations);

    (mean_time, mean_memory)
}
```

Isečak koda 6: Funkcije za merenje algoritama (Rust)

Specifikacija hardver i softvera korišćenog za izvršavanje programa i merenje bio je sledeći:

- OS: Windows 11 sa okruženjem WSL 2.6.3 - Ubuntu 22.04.5 LTS
- Procesor: AMD Ryzen 7 7700 3.8 GHz (x86-64 arhitektura)
- RAM: 32 GB

- Julia 1.12.4
- Rust 1.91.1

Rezultati i diskusija

Za potrebe merenja performansi, oba programa, i u Juliji i u Rustu, prevedeni su uz upaljene sve optimizacije koje programski prevodioci ova dva jezika pružaju (`-O3` optimizacija). Takođe, broj sistemskih niti za paralelno izvršavanje (korišćen u zavadi-pa-vladaj i Štrasenovom algoritmu) bio je unapred podešen na 8 niti. Algoritmi su bili testirani nad kvadratnim matricama čija je veličina stepen dvojke (64, 128, 256 …). U algoritmima zavadi-pa-vladaj i Štrasenov algoritam, granična vrednost (`threshold`) bila je podešena na 32, što znači da je rekurzija išla do matrica veličina 32, nakon čega je bio pozivan iterativni algoritam. Svako merenje algoritma je bilo izvršeno 10 puta za iste ulazne parametre, nakon čega je bila izračunata srednja vrednost za svako merenje po svakom algoritmu.

Na tabela 1, tabeli 2 i tabeli 3 predstavljeno je vreme izvršavanja i zauzeće memorije iterativnog, zavadi-pa-vladaj i Štrasenovog algoritma tim redom. Vidimo da je za iterativni algoritam Rust za sve veličine matrica bio u proseku 2 puta brži. Kod Štrasenovog i zavadi-pa-vladaj algoritma Rust je bio brži od Julije kada su matrice bile veće od 256. Što se tiče memorijskog zauzeća, Julia je u odnosu na Rust, trošila nešto manje memorije kod Štrasenovog i zavadi-pa-vladaj algoritma, a ovo se dešava zato što Julia podržava poglede na delove matrica bez kopiranja. Ovo u Rustu nije bilo implementirano.

Iterative Algorithm - Performance Comparison

Matrix size	Julia Time (ms)	Rust Time (ms)	Speedup (J/R)	Julia Memory (MB)	Rust Memory (MB)	Memory Ratio (J/R)
64	0.13	0.06	2.25	0.03	0.03	1.0
128	1.09	0.33	3.28	0.13	0.13	1.0
256	7.77	2.42	3.22	0.52	0.52	1.0
512	58.47	19.68	2.97	2.1	2.1	1.0
1024	452.44	148.36	3.05	8.39	8.39	1.0
2048	4377.95	2625.01	1.67	33.56	33.55	1.0

Tabela 1: Poređenje vremena izvršavanja i zauzeća memorije za iterativni algoritam za oba jezika

Divide-Conquer Algorithm - Performance Comparison

Matrix size	Julia Time (ms)	Rust Time (ms)	Speedup (J/R)	Julia Memory (MB)	Rust Memory (MB)	Memory Ratio (J/R)
64	0.16	0.53	0.3	0.17	0.26	0.65
128	0.67	0.97	0.7	1.49	2.1	0.71
256	3.7	3.65	1.01	12.47	16.81	0.74
512	34.51	20.17	1.71	101.91	134.48	0.76
1024	263.68	116.01	2.27	823.11	1075.84	0.77
2048	1963.28	884.62	2.22	6619.86	8606.71	0.77

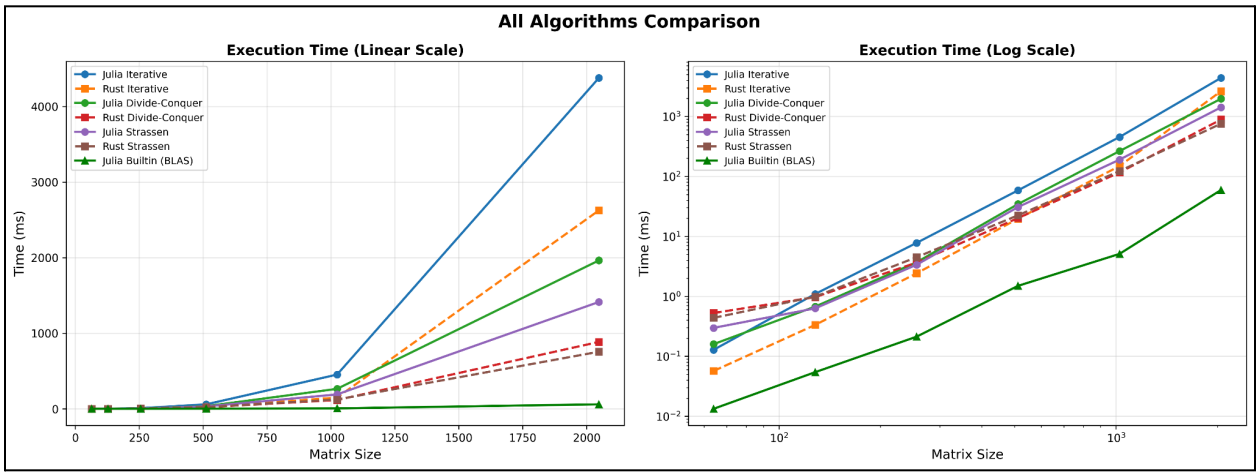
Tabela 2: Poređenje vremena izvršavanja i zauzeća memorije za zavadi-pa-vladaj algoritam za oba jezika

Strassen Algorithm - Performance Comparison

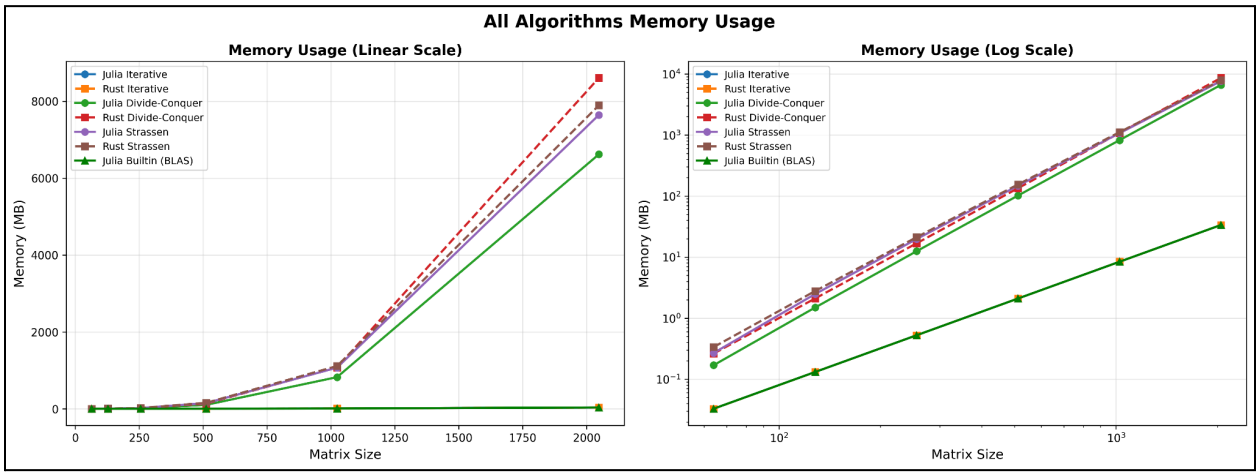
Matrix size	Julia Time (ms)	Rust Time (ms)	Speedup (J/R)	Julia Memory (MB)	Rust Memory (MB)	Memory Ratio (J/R)
64	0.3	0.44	0.68	0.27	0.34	0.8
128	0.63	0.99	0.64	2.48	2.78	0.89
256	3.35	4.48	0.75	19.73	21.17	0.93
512	30.76	22.43	1.37	147.6	154.98	0.95
1024	189.88	122.55	1.55	1070.14	1112.11	0.96
2048	1413.37	754.8	1.87	7644.75	7893.79	0.97

Tabela 3: Poređenje vremena izvršavanja i zauzeća memorije za Štrasenov algoritam za oba jezika

Na kraju, na slikama 1 i 2 poredimo međusobno performanse svih implementacija u oba programska jezika. Ovde je uključeno i izvorno množenje matrica u Juliji koje koristi BLAS rutine. Ono što se jasno vidi je da je izvorno množenje matrica u Juliji najbrže i troši najmanje memorije u odnosu na sve ostale algoritme predstavljene u ovom radu.



Slika 1: Poređenje vremena izvršavanja za sve algoritma za oba jezika



Slika 2: Poređenje zauzeća memorije za sve algoritma za oba jezika

Zaključak

Implementacija u Juliji se pokazala mnogo jednostavnija i praktičnija u poređenju sa Rustom, jer je tip podataka matrice zajedno sa ostalim operatorima i metodama već podržan u samom jeziku što znatno olakšava debagovanje i razvoj algoritama. Vreme izvršavanja manje je u Rustu, dok je u Juliji manje zauzeće memorije zbog manjeg kopiranja podataka. U praksi se preporučuje da se koristi izvorno množenje matrica u Juliji zbog bolje optimizacije i značajno boljih performansi.

Reference i prilozi

1. Master teorema: https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)
2. *Strassen, V. (1969). Gaussian elimination is not optimal. Numerische Mathematik, 13(4), 354–356.*
3. Više dimenzinalni nizovi u Juliji: https://docs.julialang.org/en/v1/manual/arrays/#man-multi-dim-arrays
4. Multi-threading Julia: https://docs.julialang.org/en/v1/base/multi-threading/
5. BLAS: https://www.netlib.org/blas/
6. LinearAlgebra Julia: https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/
7. Rayon Rust: https://crates.io/crates/rayon
8. BenchmarkTools Julia: https://juliaci.github.io/BenchmarkTools.jl/stable/
9. Stats_alloc Rust: https://docs.rs/stats_alloc/latest/stats_alloc/
10. GitHub repozitorijum projekta: https://github.com/nkusla/matmul-bench