P-ISSN: 1411-3635
E-ISSN: 2962-5645

Efisiensi Pendataran Memori pada Komputasi Matriks Padat: Studi Komparatif Rust dan Go

[1]Hamdan Yuwafi Mastu Wijaya, [2]Ani Dijah Rahajoe, [3]Retno Mumpuni
[1,2,3]Informatika, Universitas Pembangunan Nasional "Veteran" Jawa Timur, Surabaya

E-mail: [1]19081010092@student.upnjatim.ac.id, [2]anidijah.if@upnjatim.ac.id,
[3]retnomumpuni.if@upnjatim.ac.id

ABSTRAK

Tuntutan komputasi berkinerja tinggi (HPC) modern menghadapi hambatan *Memory Wall*, di mana efisiensi tata letak memori (*memory layout*) kini lebih krusial dalam menentukan performa dibandingkan penambahan inti prosesor. Struktur array dinamis bersarang secara inheren memicu fenomena pointer chasing yang merusak lokalitas data tingkat perangkat keras. Penelitian ini menyajikan studi komparatif arsitektural antara model konkurensi bahasa Go (penjadwalan M:N) dan Rust (utas kernel 1:1) melalui evaluasi komputasi matriks intensif pada ordo 500×500 hingga 2000×2000. Eksperimen mengontraskan struktur memori bersarang konvensional melawan penataan linear datar (1D *Contiguous Memory Layout*). Hasil empiris dari pengujian terisolasi menunjukkan bahwa implementasi bawaan Go mendominasi efisiensi pada skala menengah berkat rendahnya beban inisiasi Goroutine. Namun, pada arsitektur Rust, struktur bersarang memicu lonjakan latensi eksponensial akibat tingginya metrik kegagalan tembolok (*cache miss*). Pendataran memori pada Rust terbukti mengeliminasi pointer chasing, menekan latensi komputasi hingga 32,3% pada ordo tertinggi, dan memicu aktivasi instruksi vektorisasi otomatis (SIMD) oleh kompilator LLVM. Sebaliknya, aplikasi transformasi indeks linear manual pada Go justru menciptakan degradasi performa masif sebesar 79% akibat akumulasi beban kalkulasi aritmatika yang gagal dioptimasi oleh runtime. Penelitian ini membuktikan bahwa tidak ada superioritas bahasa yang absolut; pemilihan arsitektur harus didasarkan pada keselarasan *Data-Oriented Design* dengan kapabilitas kompilator, memberikan parameter rekomendasi konkret bagi perancangan sistem perangkat lunak skala industri.

Kata kunci: *Data-Oriented Design*, Go, Komputasi Paralel, Konkurensi, Pointer Chasing, Rust, Vektorisasi SIMD.

ABSTRACT

Modern high-performance computing (HPC) demands face the Memory Wall, where memory layout efficiency is now more crucial in determining performance than the number of processor cores. Nested dynamic array structures inherently induce pointer chasing, which breaks hardware-level data locality. This paper presents an architectural comparative study between the concurrency models of Go (M:N scheduling) and Rust (1:1 kernel threads) through the evaluation of intensive matrix computations on the order of 500x500 to 2000x2000. Experiments contrast conventional nested memory structures against a flat linear layout (1D Contiguous Memory Layout). Empirical results from isolated tests show that Go's native implementation dominates in efficiency at medium scales due to its low Goroutine initiation overhead. However, in Rust, nested structures induce exponential latency spikes due to high cache miss metrics. Memory flattening in Rust has been shown

to eliminate pointer chasing, reduce computational latency by up to 32.3% at the highest order, and trigger the activation of automatic vectorization (SIMD) instructions by the LLVM compiler. In contrast, manual application of linear index transformations in Go actually creates a massive performance degradation of 79% due to the accumulated arithmetic calculation load that the runtime fails to optimize. This research demonstrates that there is no absolute language superiority; architecture selection should be based on the alignment of Data-Oriented Design with compiler capabilities, providing concrete recommendation parameters for designing industrial-scale software systems.

Keywords: Concurrency, Data-Oriented Design, Go, Parallel Computing, Pointer Chasing, Rust, SIMD Vectorization.

1. PENDAHULUAN

Perkembangan sistem komputasi berkinerja tinggi (HPC) modern yang menargetkan beban kerja berpusat data telah menggeser fokus optimasi secara fundamental. Peningkatan *throughput* sistem tidak lagi dapat dicapai secara linear hanya dengan mengandalkan ekspansi jumlah inti prosesor. Pada kelas algoritma yang intensif secara numerik, hambatan utama bergeser dari kapasitas kalkulasi mentah prosesor menuju efisiensi pasokan dan keteraturan aliran data ke register CPU. Keputusan desain struktur data pada level bahasa pemrograman akhirnya berdampak langsung terhadap kemampuan kompilator dan perangkat keras dalam meminimalkan latensi akses memori.

Sejumlah penelitian terdahulu telah berupaya mengidentifikasi akar permasalahan efisiensi memori pada komputasi modern ini. Kajian sebelumnya membuktikan bahwa struktur data yang bersifat *pointer-chasing* secara tak terhindarkan menghasilkan pola akses memori yang tidak dapat diprediksi, sehingga menggagalkan optimasi perangkat keras secara masif (Choe et al., 2022). Di sisi lain, literatur juga menggarisbawahi adanya kesenjangan abstraksi (*abstraction gap*) antara sintaksis kode tingkat tinggi yang memanjakan pemrogram dengan kerja berat (*heavy lifting*) yang harus dipikul oleh prosesor dan kompilator di belakang layar untuk memetakan struktur tersebut (Hagedorn et al., 2023). Dalam konteks arsitektur yang lebih modern, studi terkini mulai menyoroti adanya biaya performa struktural (*safety cost overhead*) yang harus dibayar ketika pengembang terlalu kaku mengikuti aturan validasi kompilator statis (Patil, 2026). Meskipun berbagai literatur tersebut telah meletakkan landasan mengenai dampak lokalitas data, kajian-kajian tersebut belum menyentuh evaluasi empiris mengenai bagaimana batasan sintaksis sebuah bahasa pemrograman memengaruhi kapabilitas perangkat keras secara langsung.

Di tengah tuntutan optimalisasi lokalitas memori tersebut, bahasa pemrograman Rust memunculkan paradigma baru dengan menjanjikan kinerja sekelas bahasa C/C++ sekaligus menjamin keamanan memori absolut tanpa intervensi pengumpul sampah (*Garbage Collector*) (Gadkari, 2025; Klabnik & Nichols, 2018). Namun, paradigma ini membawa kerumitan tersendiri. Sistem *Ownership* (Kepemilikan) dan *Borrow Checker* (Pengecek Peminjaman) pada Rust memberlakukan aturan validasi referensi silang yang sangat restriktif, terutama ketika berhadapan dengan mutasi data pada blok memori linier secara serentak (Jung et al., 2020; Zeeb, 2022). Untuk menghindari konflik kompilasi (*compiler errors*) terkait aturan *lifetime* dan pinjam-pakai data, pengembang perangkat lunak

P-ISSN: 1411-3635
E-ISSN: 2962-5645

sering kali memilih jalur kompromi pragmatis dengan mengalokasikan data ke dalam struktur memori terpisah menggunakan array dinamis bersarang seperti Vec<Vec<T>> (Patil, 2026). Sayangnya, pendekatan yang murni ditujukan untuk mencapai kepatuhan sintaksis kompilator (*compiler compliance*) ini justru menciptakan degradasi performa arsitektural. Hingga saat ini, masih sangat jarang ditemukan literatur yang secara empiris menginvestigasi dampak penalti performa tingkat perangkat keras—khususnya terkait inaktivasi fungsi *auto-vectorization* SIMD dan lonjakan *cache miss*—akibat kompromi pengembang dalam menghadapi mekanika *Borrow Checker* Rust pada perancangan algoritma intensif matematis (Patil, 2026).

Sebagai solusi arsitektural atas degradasi performa akibat fragmentasi memori tersebut, penelitian ini mengusulkan penerapan teknik pendataran memori (*memory flattening*) melalui alokasi array linier satu dimensi (*1D Contiguous Memory Layout*) dalam memodelkan komputasi matriks intensif (Nyberg, 2021). Pada ekosistem Rust, arsitektur memori berdekatan (*contiguous*) ini memungkinkan pemisahan ruang memori bebas tumpang-tindih (*non-overlapping memory slicing*) secara aman, sehingga mematuhi seluruh kaidah *Borrow Checker* tanpa mengorbankan struktur lokalitas data (Cui et al., 2024; Lakhdar et al., 2019). Kepatuhan pada tata letak linier ini secara fundamental membuka jalan bagi *Hardware Prefetcher* untuk mengeliminasi lonjakan *cache miss*, sekaligus memicu kompilator LLVM untuk mengaktifkan instruksi vektorisasi otomatis (*auto-vectorization*) SIMD (*Single Instruction, Multiple Data*) pada prosesor modern (Zheng et al., 2025). Sebagai tolok ukur (*baseline*) yang representatif terhadap bahasa modern dengan manajemen memori berbasis *Garbage Collector*, pendekatan ini juga dikomparasikan pada lingkungan *runtime* bahasa Go. Evaluasi empiris difokuskan pada pengukuran metrik interaksi tingkat prosesor—termasuk utilisasi L1/L2 tembolok dan kalkulasi waktu eksekusi komputasi (*execution time*)—guna mengukur komparasi beban biaya operasional (*overhead cost*) antara manajemen memori statis milik Rust melawan mekanisme alokasi berbasis *Garbage Collector* milik Go (Viitanen, 2020).

Berangkat dari kesenjangan komparasi arsitektural tersebut, penelitian ini bertujuan untuk mengevaluasi dan membuktikan secara empiris perbedaan tingkat efisiensi alokasi memori berdekatan (*1D Contiguous Memory Layout*) pada bahasa Rust dibandingkan dengan penataan struktur dinamis pada bahasa Go dalam menangani beban komputasi matriks densitas tinggi. Kontribusi utama dari penelitian ini adalah menyediakan parameter evaluasi arsitektural berbasis Desain Berorientasi Data (*Data-Oriented Design*) bagi pengembangampun perangkat lunak, guna mensiasati galat validasi kompilator secara elegan tanpa harus mengorbankan lokalisasi tembolok tingkat prosesor.

2. LANDASAN TEORI

2.1 Hierarki Memori dan Hambatan *Memory Wall*

Selama dekade terakhir, peningkatan *throughput* sistem secara konvensional dicapai melalui arsitektur *multicore* yang memperluas paralelisme eksekusi instruksi (Hennessy & Patterson, 2019). Kendati demikian, model ekspansi ini terbentur oleh fenomena *Memory Wall*, yaitu kesenjangan performa yang semakin melebar antara laju pemrosesan inti CPU yang sangat cepat dan latensi tinggi dari kecepatan akses memori utama (RAM) (Govindasamy & Dömer, 2023). Hambatan ini mengakibatkan penambahan inti prosesor tidak lagi berbanding lurus dengan efisiensi waktu

eksekusi secara linear pada beban kerja intensif. Guna memitigasi dampak *Memory Wall*, minimalisasi jeda transfer data sangat bertumpu pada optimalisasi lokalitas spasial (*spatial locality*) serta efisiensi retensi data di dalam hierarki tembolok tingkat rendah, yang mencakup komponen *cache* L1, L2, dan L3 (Choe et al., 2022; Nguyen et al., 2017).

2.2 Abstraksi Struktur Data Dinamis dan Fenomena *Pointer Chasing*

Penerapan abstraksi struktur data tingkat tinggi, seperti array bersarang (*array of arrays*) atau *nested vectors*, banyak dipilih dalam rekayasa perangkat lunak modern karena menawarkan kemudahan sintaksis dalam penulisan logika komputasi (Hagedorn et al., 2023). Namun, abstraksi tingkat tinggi tersebut cenderung menyembunyikan realitas penataan blok memori fisik yang sebenarnya dikelola oleh perangkat keras (Henriksen et al., 2019). Pada mayoritas implementasi memori dinamis bahasa pemrograman, setiap baris dari struktur larik bersarang dialokasikan secara terpisah di area *heap* dan hanya terikat melalui penyimpanan alamat penunjuk (*pointer*). Konfigurasi memori yang terfragmentasi secara acak ini memaksa prosesor untuk mengeksekusi operasi penelusuran penunjuk berantai (*pointer chasing*) pada setiap tahapan iterasi gelung (Choe et al., 2022). Karakteristik pola akses yang tidak teratur ini menggagalkan fungsionalitas unit *hardware prefetcher* untuk memprediksi dan memuat baris data berikutnya secara proaktif. Akibat kegagalan tersebut, rasio kegagalan tembolok (*cache miss rate*) pada tingkat L1 dan L2 melonjak tajam, memaksa *pipeline* instruksi eksekusi CPU mengalami kondisi mandek (*stall*) hanya untuk menunggu suplai data laten dari RAM utama (Hennessy, John L.; Patterson, 2019).

2.3 Abstraksi Struktur Data Dinamis dan Fenomena *Pointer Chasing*

Arsitektur kompilator modern dan infrastruktur optimalisasi kode seperti LLVM membutuhkan keteraturan struktur data fisis yang rapat untuk mengaktifkan fitur optimasi tingkat rendah secara agresif (Hagedorn et al., 2023). Secara umum di ranah manipulasi data, transformasi struktur matriks multidimensi atau data berskala besar menjadi bentuk linear searah, yang dikenal sebagai proses vektorisasi, telah terbukti secara empiris mampu menyederhanakan representasi data, mempercepat waktu pemrosesan, serta meningkatkan akurasi performansi sistem secara signifikan (Rahajoe et al., 2023).

Tata letak memori yang berdekatan (*contiguous memory layout*) bertindak sebagai prasyarat utama yang memampukan kompilator membuktikan ketiadaan risiko tumpang-tindih referensi (*memory aliasing*) secara statis (Jung et al., 2020). Keberhasilan pembuktian kondisi aman tersebut memicu kompilator modern untuk menerapkan teknik *loop unrolling* secara optimal serta mengeksekusi otomatisasi instruksi vektor berbasis SIMD pada register perangkat keras (Henriksen et al., 2019; Zheng et al., 2025).

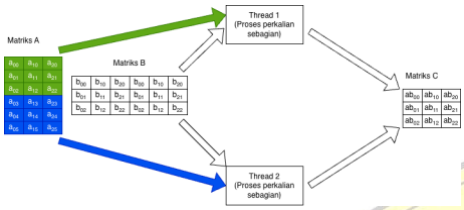
3. METODOLOGI

3.1 Dekomposisi Paralel Berbasis Baris (Row-based Concurrent Spawning)

Strategi komputasi paralel dalam mengeksekusi kalkulasi matriks pada penelitian ini menerapkan arsitektur dekomposisi data berbasis baris (*row-wise block striped partitioning*) (Grama et al., 2003). Dalam arsitektur ini, operasi dasar perkalian matriks $C=A \times B$ didistribusikan sedemikian rupa sehingga setiap utas (*thread*) pekerja diinstruksikan untuk menghitung sekumpulan baris independen

P-ISSN: 1411-3635
E-ISSN: 2962-5645

dari matriks hasil *C*, sebagaimana diilustrasikan pada Gambar 1.



Gambar 1 Pemetaan dekomposisi berbasis baris pada utas konkuren.

Implementasi pembagian beban kerja ini dieksekusi mutlak menggunakan primitif bawaan tingkat bahasa (*language-native primitives*), untuk mencegah sisipan optimasi asinkron tersembunyi. Pada bahasa Go, peluncuran utas memanfaatkan entitas *Goroutine* yang dikendalikan melalui mekanisme penghalang sinkronisasi *sync.WaitGroup* (Donovan & Kernighan, 2015). Sementara itu, implementasi pada bahasa Rust diatur secara eksplisit menggunakan pemanggilan *std::thread::spawn* untuk menciptakan batas utas sistem operasi asli (*Native OS Threads* 1:1), yang secara simultan disinkronkan tanpa melanggar validasi keamanan mutasi dari *Borrow Checker* (Klabnik & Nichols, 2018).

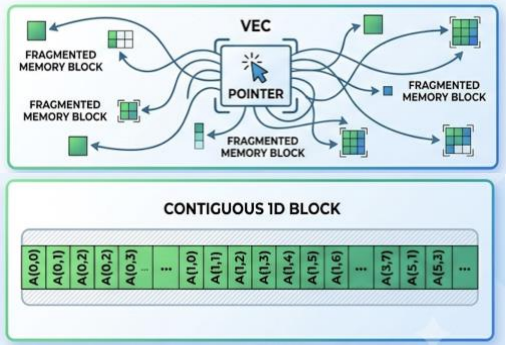
3.2 Konfigurasi Struktur Data

Eksperimen ini mengontraskan dua model alokasi dinamis untuk mengevaluasi struktur data terhadap memori. Konfigurasi pertama adalah struktur array bersarang (*Nested Structures*), yang diimplementasikan sebagai *[][]float64* pada Go dan *Vec<Vec<f64>>* pada Rust. Pendekatan konvensional ini secara inheren menciptakan fragmentasi memori, di mana setiap vektor baris tidak dialokasikan secara bersebelahan di dalam *heap*, sehingga memaksa unit prosesor untuk mengeksekusi penelusuran penunjuk tak terprediksi (*pointer-chasing*) (Choe et al., 2022).

Sebagai resolusi arsitektural, konfigurasi kedua menerapkan penataan memori linear datar (*1D Contiguous Memory Layout*). Pada model ini, matriks dua dimensi berukuran $M \times N$ diproyeksikan dalam struktur array satu dimensi *Vec<T>* dengan panjang ekuivalen $N \times M$. Transformasi ruang alamat memori ini mengeliminasi kompleksitas *pointer*, dengan kalkulasi koordinat elemen (*i, j*) yang disubstitusi menjadi pemetaan indeks linear melalui Persamaan 1.

$$Index_{1D} = (i \times N) + j \quad (1)$$

Perbandingan visual tata letak fisik di dalam memori antara arsitektur bersarang dan linear datar ini dapat dilihat pada Gambar 2.



Gambar 2 Komparasi alokasi memori terfragmentasi (atas) dan blok linier rapat (bawah).

Pendataran struktur ini memastikan seluruh elemen data tersusun secara rapat (*contiguous*) dalam blok memori fisik, sebuah prasyarat krusial untuk memaksimalkan kapabilitas *Hardware Prefetcher* dalam menurunkan rasio *cache miss* serta memungkinkan aktivasi instruksi vektorisasi SIMD secara otomatis oleh kompilator (Nyberg, 2021).

3.3 Lingkungan Eksperimen dan Spesifikasi Perangkat

Isolasi pengujian performa dieksekusi pada lingkungan perangkat keras tunggal yang dikonfigurasi secara identik lintas seluruh skema pemrograman. Eksperimen dijalankan pada arsitektur komputer berbasis prosesor SoC Apple M3 (arsitektur ARM64) yang dilengkapi dengan unit L1 Cache, L2 Cache, dan Last-Level Cache (LLC) terpadu. Dari sisi perangkat lunak, ekosistem Go berjalan di atas *runtime* versi Go 1.26.3 darwin/arm64, sedangkan ekosistem Rust dikompilasi menggunakan versi kompilator rustc 1.97.0 dengan target LLVM *backend* diaktifkan lewat flag optimasi rilis penuh (*cargo build --release*).

3.4 Lingkungan Eksperimen dan Spesifikasi Perangkat

Guna mengeliminasi bias yang ditimbulkan oleh keliaran peluncuran utas dinamis (*thread trashing*) yang dapat mendistorsi keabsahan perbandingan, penelitian ini menerapkan pembatasan ketat pada jumlah pekerja aktif (*Bounded Worker Pool*) sebagai variabel kontrol utama. Baik pada *runtime* Go maupun Rust, jumlah utas pekerja paralel dikunci secara konstan sebanyak 8 *workers*, disesuaikan dengan jumlah inti fisik (*physical cores*) pada prosesor. Variabel bebas dalam pengujian ini dipisahkan menjadi dua dimensi: dimensi tata letak memori (*go:normal, go:flat, rust:normal, rust:green2d, rust:green1d*) serta dimensi skala beban kerja yang direpresentasikan melalui eskalasi ordo matriks persegi ($N \times N$) pada tingkatan 500, 1000, 1500, dan 2000. Sementara itu, variabel terikat yang diukur secara mutlak adalah durasi waktu eksekusi murni (*raw execution time*) dalam satuan milidetik (ms) menggunakan fungsi penanda waktu bawaan masing-masing bahasa tanpa intervensi pustaka pencatatan luar.

3.5 Lingkungan Eksperimen dan Spesifikasi Perangkat

Alur eksekusi pengujian diatur secara terotomasi menggunakan skrip orkestrasi terpusat untuk menjamin keabsahan data statistik serta menghindari manipulasi latar belakang oleh mekanisme internal bahasa. Sebelum pencatatan waktu untuk setiap ordo dimulai, perintah pembersihan menyeluruh berupa *go clean -cache -testcache* dan *cargo clean* wajib dieksekusi secara berurutan. Langkah ini krusial untuk memastikan data yang diperoleh berasal dari proses pembangunan segar (*fresh build*) dan bersih dari bias penyimpanan sementara (*incremental cache*). Setiap skenario ukuran ordo matriks diuji secara terisolasi sebanyak 10 kali replikasi perintah *run* independen. Hasil pencatatan dari 10 kali replikasi tersebut kemudian dihitung nilai rata-rata agregatnya (*mean*) untuk mereduksi faktor anomali gangguan dari proses latar belakang sistem operasi (*OS background noise*). Seluruh sebaran data mentah dari variasi waktu eksekusi aktual didokumentasikan penuh secara transparan dalam lampiran sebagai bukti otentik transparansi riset (*reproducibility*).

4. HASIL DAN PEMBAHASAN

4.1 Analisis Kegagalan Tembolok (*Cache Miss*) dan Fenomena Pointer Chasing pada Implementasi Rust

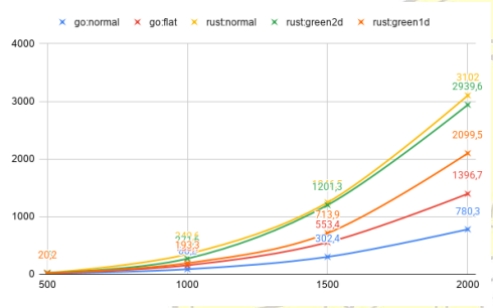
Data agregat rata-rata waktu eksekusi dari 10 replikasi independen seperti yang terdapat pada Tabel 1 dan grafik rerata waktu pada Gambar 3, menunjukkan bahwa konfigurasi *rust:normal* dengan array dinamis bersarang (*Vec<Vec<T>>*) menghasilkan performa paling lambat. Durasi komputasi meningkat eksponensial seiring pembesaran skala matriks: 25,1 ms (500×500), 349,6 ms (1000×1000),

P-ISSN: 1411-3635
E-ISSN: 2962-5645

1246,5 ms (1500×1500), hingga 3102,0 ms (2000×2000).

Tabel 1. Tabel Agregat Waktu Eksekusi (*ms*)

Ordo	*Go*		*Rust*		
	normal	*flat*	*normal*	*green2d*	*green1d*
500	13	17,2	25,1	26	20
1000	88,2	156,1	349,6	271,6	193,3
1500	302,4	553,4	1246,5	1201,3	713,9
2000	780,3	1396,7	3102	2939,6	2099,3



Gambar 3 Grafik garis tren multi-skala waktu eksekusi perkalian matriks paralel Go dan Rust.

Degradasi performa masif ini mengonfirmasi terjadinya fenomena *pointer chasing* berantai yang merusak lokalitas data spasial. Dalam arsitektur *Vec<Vec<T>>*, setiap elemen vektor induk merupakan penunjuk (*pointer*) ke alamat *heap* terpisah tempat elemen baris berada. Fragmentasi memori ini menyebabkan pola akses data selama iterasi bersifat acak dan ireguler, sehingga menggagalkan prediksi *hardware prefetcher* dan memicu tingginya *cache miss* L1/L2. Akibatnya, terjadi penundaan eksekusi (*pipeline stall*) karena CPU harus menunggu pengambilan data dari RAM utama yang memiliki latensi lebih tinggi.

4.2 Mekanisme Auto-Vectorization Kompilator LLVM dan Aktivasi Instruksi Register SIMD pada Rust

Optimasi alokasi memori linear datar pada *rust:green1d* memotong penalti performa secara signifikan dengan catatan waktu konstan di bawah varian Rust lainnya: 20,2 ms (500×500), 193,3 ms (1000×1000), 713,9 ms (1500×1500), dan 2099,5 ms (2000×2000). Pada ordo ekstrem 2000×2000, teknik pendataran memori 1D ini memangkas waktu eksekusi dari 3102,0 ms menjadi 2099,5 ms (efisiensi arsitektural 32,3%), melampaui varian berbasis blok 2D (*rust:green2d*) yang tertahan di angka 2939,6 ms.

Keunggulan *rust:green1d* berakar pada keselarasan struktur data linear kontigu dengan mekanika *auto-vectorization pass* pada kompilator backend LLVM (Hagedorn et al., 2023). Struktur memori yang rapat memungkinkan kompilator membuktikan tiadanya tumpang-tindih referensi (*memory aliasing*) secara statis (Henriksen et al., 2019). Kondisi aman ini memicu LLVM untuk menerapkan *loop unrolling* agresif dan memetakan operasi skalar ke dalam instruksi register SIMD pada prosesor Apple M3 (Zheng et al., 2025). Paralelisme tingkat instruksi ini mengeksekusi beberapa data *floating-point* 64-bit secara simultan, mengeliminasi biaya *looping boundary check*, dan memaksimalkan *throughput* register CPU.

4.3 Mekanisme Auto-Vectorization Kompilator LLVM dan Aktivasi Instruksi Register SIMD pada Rust

Temuan anomali terdokumentasi pada bahasa Go, di mana *go:normal* (*[][]float64*) konsisten mendominasi pengujian dengan waktu tercepat: 13,0 ms (500×500), 88,2 ms (1000×1000), 302,4 ms (1500×1500), dan 780,3 ms (2000×2000). Sebaliknya, skema memori linear datar (*go:flat*) justru mengalami degradasi performa masif hingga 79% (1396,7 ms pada ordo 2000×2000) dibandingkan bentuk normalnya.

Anomali ini menyingkap adanya biaya operasional tersembunyi (*hidden*

overhead) akibat kalkulasi aritmatika indeks manual pada runtime Go yang tidak didesain untuk vektorisasi tingkat rendah. Berbeda dengan *go:normal* yang dioptimalkan secara internal melalui *slice header*, skema *go:flat* memaksa transformasi koordinat 2D ke 1D secara manual pada perulangan terdalam sebanyak N^3 kali menggunakan persamaan 2.

$$Index_{1D} = (i \times N) + j \qquad (2)$$

Karena kompilator Go tidak melakukan penguraian instruksi SIMD seagresif LLVM untuk penataan memori datar, akumulasi operasi perkalian dan penjumlahan integer ini menjadi *bottleneck* baru pada unit ALU. Beban komputasi aritmatika indeks manual tersebut terbukti jauh lebih mahal daripada biaya dereferensi penunjuk *slice header* bawaan runtime Go.

5. KESIMPULAN

5.1 Kesimpulan

Berdasarkan hasil analisis data eksperimental dan pembahasan arsitektural tingkat rendah (*low-level architectural analysis*) yang telah dipaparkan, penelitian ini menyimpulkan tiga temuan fundamental terkait *trade-off* antara manajemen memori statis dan mekanika *runtime* pada komputasi numerik paralel:

1. Efisiensi Utas Ringan Go pada Skala Menengah: Model konkurensi M:N pada runtime Go (*go:normal*) menghasilkan waktu eksekusi yang lebih superior dan konsisten dibanding model utas OS 1:1 bawaan Rust. Penjadwalan kooperatif *user-space* Go secara radikal mampu menekan *spawning overhead* dan biaya *context switching* pada beban kerja *CPU-bound* skala moderat.

2. Lokalitas Data Spasial dan Vektorisasi SIMD: Struktur *array* bersarang pada Rust (*rust:normal*) memicu penalti performa eksponensial akibat *pointer chasing* di area *heap*, yang melumpuhkan *Hardware Prefetcher* dan meningkatkan *cache miss* L1/L2. Sebaliknya, teknik memori linear 1D (*rust:green1d*) meningkatkan performa hingga 32,3% pada ordo matriks 2000×2000. Alokasi kontigu ini mengeliminasi *memory aliasing*, sehingga memicu kompilator LLVM mengaktifkan instruksi SIMD dan *loop unrolling* secara maksimal.

3. Biaya Aritmatika Indeks Tersembunyi: Penerapan memori linear pada Go (*go:flat*) justru menurunkan performa hingga 79%. Hal ini membuktikan bahwa pada *runtime* Go, biaya kalkulasi aritmatika indeks manual *index = (i×N) + j* yang dieksekusi N^3 kali jauh lebih membebani prosesor dibandingkan biaya dereferensi penunjuk *slice header* bawaannya.

Sebagai implikasi praktis rekayasa perangkat lunak (*software engineering implications*), *software architect* disarankan memilih bahasa Go untuk infrastruktur hibrida yang mengutamakan kecepatan siklus pengembangan (*agility*) serta manajemen jaringan yang dinamis. Sementara itu, bahasa Rust menawarkan superioritas mutlak untuk beban kerja komputasi numerik murni tingkat rendah (*bare-metal*), dengan catatan wajib mengimplementasikan struktur data linear datar sejak awal perancangan guna menghindari jebakan degradasi lokalitas memori akibat restriksi ketat *Borrow Checker*.

5.2 Saran

Berlandaskan keterbatasan eksperimen dalam makalah ini, arah pengembangan riset ke depan direkomendasikan untuk menitikberatkan pada dua pilar perluasan:

P-ISSN: 1411-3635
E-ISSN: 2962-5645

- Eksplorasi Lingkungan Terdistribusi: Menguji implementasi penataan memori linear datar (*1D Flattened Vector*) dan struktur data bebas penguncian (*Lock-Free structures*) pada kluster komputasi awan skala produksi (*production-scale distributed cloud deployment*) guna mengevaluasi dampak latensi jaringan hibrida.

- Analisis Kedalaman Manajemen Memori: Mengintegrasikan instrumen *profiling* perangkat keras tingkat lanjut (seperti *pprof*, *Valgrind*, atau *heaptrack*) untuk merekam visualisasi fluktuasi konsumsi RAM (*heap allocation spikes*) secara kuantitatif saat mesin pengumpul sampah *Tricolor Mark-and-Sweep* milik Go sedang aktif bekerja menandingi stabilnya garis alokasi statis Rust.

6. UCAPAN TERIMA KASIH

Penulis menyampaikan penghargaan yang setinggi-tingginya kepada Fakultas Ilmu Komputer, Universitas Pembangunan Nasional "Veteran" Jawa Timur atas dukungan fasilitas akademik yang memungkinkan terlaksananya eksperimen komputasi ini. Terima kasih yang mendalam juga ditujukan kepada Dosen Pembimbing yang telah memberikan arahan kritis terkait metodologi evaluasi performa, serta kepada seluruh pihak yang secara langsung maupun tidak langsung telah berkontribusi dalam validasi instrumen arsitektur tingkat rendah pada penelitian ini.

DAFTAR PUSTAKA

Choe, J., Crotty, A., Moreshet, T., Herlihy, M., & Bahar, R. I. (2022). HybriDS. *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, *1*(1), 321–332. https://doi.org/10.1145/3490148.3538591

Cui, M., Mao, P., Sun, S., Zhou, Y., & Xu, H. (2024). *Fearless Unsafe. A More User-friendly Document for Unsafe Rust Programming Base on Refined Safety Properties*. 1–22. http://arxiv.org/abs/2412.06251

Donovan, A. A. A., & Kernighan, B. W. (2015). *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

Gadkari, B. R. (2025). Sarcouncil Journal of Engineering and Computer Sciences RUST: A Memory Leakage-Proof Way of Building Applications. *Sarcouncil Journal of Engineering and Computer Sciences*, *04*(09), 10–15. https://doi.org/https://doi.org/10.5281/zenodo.17020357

Govindasamy, V., & Dömer, R. (2023). Instruction-Level Modeling and Evaluation of a Cache-Less Grid of Processing Cells. *2023 Forum on Specification & Design Languages (FDL)*, *2023-Septe*(2), 1–8. https://doi.org/10.1109/FDL59689.2023.10272195

Grama, A., Karypis, G., Kumar, V., & Gupta, A. (2003). *An Introduction to Parallel Computing: Design and Analysis of Algorithms* (2nd ed.). Pearson Education Limited.

Hagedorn, B., Fan, B., Chen, H., Cecka, C., Garland, M., & Grover, V. (2023). Graphene: An IR for Optimized Tensor Computations on GPUs. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, *3*, 302–313. https://doi.org/10.1145/3582016.3582018

Hennessy, John L.; Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (S. Merken, N. McFadden, S. Viswanathan, & C. J. Bilbow, Eds.; 6th ed.). Morgan Kaufmann Publishers.

Henriksen, T., Thorøe, F., Elsman, M., & Oancea, C. (2019). Incremental

flattening for nested data parallelism. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, (i), 53–67. https://doi.org/10.1145/3293883.3295707

Jung, R., Dang, H.-H., Kang, J., & Dreyer, D. (2020). Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages*, *4*(POPL), 1–32. https://doi.org/10.1145/3371109

Klabnik, S., & Nichols, C. (2018). *The Rust Programming Language* (2nd ed.). No Starch Press, Inc.

Lakhdar, R. S., Charles, H.-P., & Kooli, M. (2019). Toward Modeling Cache-Miss Ratio for Dense-Data-Access-Based Optimization. *Proceedings of the 30th International Workshop on Rapid System Prototyping (RSP'19)*, 64–70. https://doi.org/10.1145/3339985.3358498

Nguyen, T. T., Dahl, V. A., & Bærentzen, J. A. (2017). Cache-mesh, a Dynamics Data Structure for Performance Optimization. *Procedia Engineering*, *203*, 193–205. https://doi.org/10.1016/j.proeng.2017.09.807

Nyberg, F. (2021). *Investigating the Effect of Implementing Data-Oriented Design Principles on Performance and Cache Utilization* [Umeå University]. https://www.diva-portal.org/smash/record.jsf?pid=diva2:1578616&dswid=-9728

Patil, M. K. (2026). The Rust Tax: Measuring the Cost of Memory Safety and Safely Recovering What You Can. *International Journal Of Scientific Research In Engineering & Technology*, *6*(1), 40. https://doi.org/10.59256/ijsreat.20260601007

Rahajoe, A. D., Winarko, E., & Guritno, S. (2023). Vectorization Method Based on High Correlation for Multivariate Time Series Hybrid Filter Wrapper Feature Selection. *ICIC Express Letters*, *17*(4), 471–478. https://doi.org/10.24507/icicel.17.04.471

Viitanen, R. (2020). *Evaluating Memory Models for Graph-Like Data Structures in the Rust Programming Language: Performance and Usability Utvärdering av Minnesmodeller för Graf-Liknande Datastruk-turer i Programmeringsspråket Rust: Användbarhet och Pre-standa* [Linköping University]. www.liu.se

Zeeb, C. (2022). *Memory Management in Rust: Principles and Comparison with C and C++* [Ludwig-Maximilians-University of Munich]. https://www.en.pms.ifi.lmu.de/publications/projektarbeiten/Claudio.Zeeb/BA_Claudio.Zeeb.pdf

Zheng, Z., Wu, K., Cheng, L., Li, L., Rocha, R. C. O., Liu, T., Wei, W., Zeng, J., Zhang, X., & Gao, Y. (2025). *VecTrans: Enhancing Compiler Auto-Vectorization through LLM-Assisted Code Transformations*. 1–12. http://arxiv.org/abs/2503.19449