**Synthetic DNA Sequence Generation**

Untuk keperluan pengujian performa algoritma string matching, digunakan dataset berupa sekuens DNA acak yang dibangkitkan secara sintetik. Dataset ini disusun menggunakan program C++ yang memanfaatkan fungsi rand() untuk menghasilkan susunan karakter acak dari empat basa nitrogen DNA, yaitu Adenin (A), Timin (T), Sitosin (C), dan Guanin (G). Karakter ditulis dalam huruf kecil ('a', 't', 'c', 'g') untuk konsistensi pengolahan data pada tahap selanjutnya.

Proses pembuatan sekuens dilakukan dengan langkah-langkah berikut:

1. **Inisialisasi Generator Angka Acak:**  
   Fungsi srand() digunakan dengan seed dari waktu sistem (time(0)) agar hasil random berbeda setiap eksekusi.
2. **Pembuatan Sekuens:**  
   Fungsi seq\_gen() menerima parameter panjang sekuens (n) dan array karakter (seq[]) yang akan diisi. Setiap posisi diisi secara acak dengan salah satu dari empat huruf DNA menggunakan rand() % 4.
3. **Penyimpanan ke File:**  
   Sekuens yang telah dibangkitkan disimpan dalam file random\_dna\_sequence.txt menggunakan ofstream.

Kode ini menghasilkan sekuens dengan panjang 100.000 karakter secara default, namun nilai ini dapat diubah sesuai kebutuhan eksperimen. File keluaran digunakan sebagai dataset utama dalam pengujian performa algoritma pencocokan string baik pada versi serial maupun paralel (CUDA/OpenACC).

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <ctime>

void seq\_gen(int n, char seq[]) {

    for (int i = 0; i < n; i++) {

        int base = rand() % 4;

        switch (base) {

            case 0: seq[i] = 'a'; break;

            case 1: seq[i] = 't'; break;

            case 2: seq[i] = 'c'; break;

            case 3: seq[i] = 'g'; break;

        }

    }

}

int main() {

    srand(static\_cast<unsigned int>(time(0))); // Seed random number generator

    int sequence\_length = 100000; // Ubah sesuai kebutuhan

    char\* sequence = new char[sequence\_length + 1];

    sequence[sequence\_length] = '\0'; // Null-terminate string

    seq\_gen(sequence\_length, sequence);

    // Simpan ke file

    std::ofstream outfile("random\_dna\_sequence.txt");

    if (!outfile) {

        std::cerr << "Error membuka file untuk menulis." << std::endl;

        delete[] sequence;

        return 1;

    }

    outfile << sequence;

    outfile.close();

    std::cout << "Sekuens DNA acak dengan panjang " << sequence\_length << " telah disimpan ke random\_dna\_sequence.txt" << std::endl;

    delete[] sequence;

    return 0;

}

**Generation of DNA Search Patterns**

Setelah dataset utama berupa sekuens DNA acak berhasil dibangkitkan dan disimpan ke dalam file random\_dna\_sequence.txt, langkah selanjutnya adalah menghasilkan pola pencarian (*search patterns*) yang akan digunakan sebagai input pada algoritma pencocokan string.

Proses ini dilakukan menggunakan skrip Python yang terdiri dari beberapa fungsi utama:

1. **Pembacaan Sekuens dari File:**  
   Fungsi read\_sequence\_from\_file() membaca seluruh isi file DNA, menghapus karakter newline (\n), dan memastikan seluruh huruf dikonversi menjadi huruf kecil untuk konsistensi.
2. **Penyimpanan Sekuens Utama:**  
   Sekuens hasil pembacaan kemudian disimpan kembali ke dalam file data.txt sebagai representasi input teks yang akan dicocokkan.
3. **Pembuatan Pola Pencarian:**  
   Fungsi generate\_patterns() digunakan untuk mengekstrak pola-pola pencarian dari sekuens DNA. Setiap pola memiliki panjang tetap (pattern\_length, dalam hal ini 10 karakter), dan jumlah pola yang dihasilkan bervariasi tergantung pada parameter num\_patterns (misalnya 8, 16, 32, hingga 1024). Pola dipilih dari sekuens utama menggunakan *sliding window* dengan langkah yang merata untuk memastikan distribusi yang seragam.
4. **Penyimpanan Pola:**  
   Setiap set pola disimpan dalam file terpisah dengan nama pattern\_X.txt, di mana X menyatakan jumlah pola dalam file tersebut.

Langkah ini memastikan bahwa pengujian performa algoritma pencocokan string dapat dilakukan dengan berbagai skenario beban pencarian, dari yang ringan (sedikit pola) hingga yang kompleks (banyak pola). Seluruh proses dilakukan secara otomatis dan terorganisasi untuk mendukung eksperimen dengan konsistensi data yang tinggi.

A screenshot of a computer

AI-generated content may be incorrect.

import os

def read\_sequence\_from\_file(file\_path):

    """Membaca sekuens DNA dari file teks biasa (tanpa header FASTA)."""

    print(f"📥 Membaca file dari: {file\_path}")

    with open(file\_path, 'r') as f:

        sequence = f.read().replace('\n', '').lower()

    print(f"✅ Panjang sekuens: {len(sequence)} karakter\n")

    return sequence

def generate\_patterns(sequence, pattern\_length, num\_patterns):

    """Menghasilkan sejumlah pola dari sekuens dengan panjang dan jumlah tertentu."""

    patterns = []

    step = max(1, (len(sequence) - pattern\_length) // num\_patterns)

    for i in range(0, step \* num\_patterns, step):

        pattern = sequence[i:i + pattern\_length]

        if len(pattern) == pattern\_length:

            patterns.append(pattern)

    return patterns

def save\_to\_file(filename, lines):

    """Menyimpan list string ke file, setiap item satu baris."""

    with open(filename, 'w') as f:

        for line in lines:

            f.write(line + '\n')

def main():

    # Lokasi file input/output (KEMBALI KE FOLDER SEBELUMNYA)

    base\_dir = r'C:\Users\Erwiyono Yusuf\Downloads\project kompar'

    input\_file = os.path.join(base\_dir, 'random\_dna\_sequence.txt')

    output\_data = os.path.join(base\_dir, 'data.txt')

    # Baca sekuens DNA

    sequence = read\_sequence\_from\_file(input\_file)

    # Simpan sekuens ke data.txt

    save\_to\_file(output\_data, [sequence])

    print(f"✅ Disimpan: {output\_data}\n")

    # Daftar jumlah pola yang dihasilkan

    pattern\_sizes = [8, 16, 32, 64, 128, 256, 512, 1024]

    for size in pattern\_sizes:

        patterns = generate\_patterns(sequence, pattern\_length=10, num\_patterns=size)

        output\_patterns = os.path.join(base\_dir, f'pattern\_{size}.txt')

        save\_to\_file(output\_patterns, patterns)

        print(f"✅ {size} pola disimpan ke: {output\_patterns}")

    print("\n🎉 Semua file selesai dibuat!")

if \_\_name\_\_ == '\_\_main\_\_':

    main()

KMP CUDA

Dalam implementasi algoritma **Knuth-Morris-Pratt (KMP)** berbasis CUDA yang telah dimodifikasi, terdapat beberapa perubahan signifikan dari file asli. Pertama, pengelolaan file input mengalami perbaikan. Pada versi asli, file input dibaca sebagai satu entitas, dengan pola dan target dibaca dalam satu file menggunakan f1 >> tar >> pat;. Namun, pada versi modifikasi, dua file terpisah digunakan, masing-masing untuk data target dan pola. File-file ini kemudian dibaca sepenuhnya menggunakan istreambuf\_iterator<char> dan diproses setelah dihapus spasi di awal dan akhir serta newline yang ada. Fungsi trim() dan removeNewlines() ditambahkan untuk memastikan data bersih dan siap untuk diproses lebih lanjut. Selain itu, seluruh data diubah menjadi **uppercase** menggunakan transform(), yang memungkinkan pencocokan pola yang tidak **case-sensitive**.

Dalam hal pemrosesan pola menggunakan algoritma KMP, terdapat perubahan penting dalam cara kernel CUDA beroperasi. Di versi asli, setiap thread memproses rentang indeks berdasarkan perhitungan berbasis jumlah thread. Namun, pada implementasi modifikasi, setiap thread memulai pencarian dari posisi indeksnya masing-masing, yang meningkatkan kontrol terhadap proses pencocokan dan mencegah kemungkinan terjadinya **out-of-bounds access** pada memori target. Perubahan ini memastikan bahwa pencocokan dilakukan secara lebih aman dan efisien.

Di sisi alokasi memori, perubahan besar terjadi dengan mengganti penggunaan char\* dan new[] di sisi host dengan std::vector<char> dan std::vector<int>. Penggunaan vector mengurangi risiko kebocoran memori dan lebih mudah dikelola karena sifatnya yang otomatis mengelola memori (RAII). Sebagai tambahan, fungsi untuk mengecek error pada setiap pemanggilan API CUDA ditambahkan dengan makro CUDA\_CHECK, yang memungkinkan program untuk secara langsung menangani kesalahan saat bekerja dengan CUDA, memberikan stabilitas dan kemudahan dalam **debugging**.

Untuk pengukuran waktu eksekusi, pendekatan yang lebih tepat diterapkan dengan menggunakan cudaEvent. Berbeda dengan penggunaan time\_t pada file asli, cudaEvent memberikan pengukuran waktu yang lebih presisi dan sesuai dengan praktik terbaik di pengembangan CUDA, mencatat durasi untuk salinan data ke GPU serta eksekusi kernel. Hasil pencocokan pola, termasuk posisi dan panjang pola yang cocok, juga dicetak secara lebih informatif, memberikan kejelasan tentang posisi dan ukuran match yang ditemukan dalam data target.

Selain itu, integrasi dengan **CUDA Profiler** melalui cudaProfilerStart() dan cudaProfilerStop() memberikan kemampuan untuk memantau performa program pada level yang lebih mendalam, yang berguna dalam pengoptimalan dan analisis lebih lanjut mengenai kinerja eksekusi kernel CUDA.

Secara keseluruhan, modifikasi yang dilakukan pada file ini bertujuan untuk meningkatkan fleksibilitas, efisiensi, dan **keamanan** program. Penggunaan teknik-teknik modern seperti std::vector, pengukuran waktu berbasis CUDA, dan pengecekan error memperkuat kualitas kode, menjadikannya lebih mudah dipelihara dan lebih siap untuk menangani berbagai kondisi input yang beragam.

A black text on a white background

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A black text on a white background

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A close up of numbers

AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

A close up of numbers

AI-generated content may be incorrect.

A close up of a number

AI-generated content may be incorrect.

berisi hasil ujicoba algoritma KMP yang diimplementasikan **menggunakan CUDA (berjalan di GPU)**. Waktu yang tercatat di sana adalah total waktu eksekusi di GPU (termasuk copy data dan string matching).

KMP SEKUENSIAL

Dalam implementasi modifikasi algoritma **Knuth-Morris-Pratt (KMP)** untuk pencocokan pola berbasis **CPU**, beberapa langkah penting dilakukan untuk mempersiapkan dan menjalankan pencocokan pola antara data dan pola yang diberikan. Pertama-tama, file input untuk **data** dan **pola** dibaca dan diproses dengan memanfaatkan fungsi trim() dan removeNewlines() untuk menghilangkan spasi kosong, tab, dan karakter newline yang tidak diinginkan dari kedua string. Selanjutnya, seluruh data dan pola diubah menjadi huruf kapital dengan menggunakan std::transform(), yang memungkinkan pencocokan yang **tidak sensitif terhadap huruf besar/kecil**.

Pada bagian utama implementasi, digunakan algoritma KMP untuk mencari kecocokan pola dalam data yang telah dibaca. Proses pencocokan dimulai dengan membangun fungsi **failure** (fungsi Gagal) menggunakan fungsi preKMP(), yang kemudian digunakan dalam proses pencarian pola di dalam data dengan fungsi KMPsearch(). Fungsi ini melibatkan pencocokan karakter dalam **text** dan **pattern** berdasarkan aturan algoritma KMP, yang menghindari pencocokan yang berulang, meningkatkan efisiensi pencarian pola.

Untuk mengukur performa pencocokan pola, waktu eksekusi dihitung menggunakan **high resolution clock** yang disediakan oleh library <chrono>. Waktu yang diperlukan untuk menjalankan pencocokan pola diukur dengan membungkus fungsi pencocokan dalam std::chrono::high\_resolution\_clock::now() untuk mendapatkan waktu awal dan waktu akhir eksekusi. Selisih antara keduanya memberikan durasi dalam satuan detik, yang menunjukkan waktu yang dibutuhkan CPU untuk menyelesaikan pencarian pola dalam data.

Setelah pencocokan selesai, hasilnya akan menunjukkan posisi-posisi di mana pola ditemukan dalam data. Hasil ini dicetak untuk memberikan informasi yang jelas mengenai hasil pencocokan pola. Jika pola tidak ditemukan dalam data, program akan memberikan indikasi bahwa pencocokan tidak berhasil.

Pengujian dengan menggunakan **CPU** ini berfokus pada pengukuran waktu eksekusi untuk algoritma KMP pada platform CPU, yang memberikan dasar perbandingan yang baik dengan implementasi menggunakan CUDA yang akan dilakukan kemudian. Hal ini memungkinkan analisis lebih lanjut tentang kelebihan dan kekurangan dari kedua platform (CPU dan GPU) dalam hal kinerja pencocokan pola.

A close up of a number

AI-generated content may be incorrect.

A close up of numbers

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

A close up of numbers

AI-generated content may be incorrect.

A close up of a number

AI-generated content may be incorrect.

A close up of numbers

AI-generated content may be incorrect.

A close up of numbers

AI-generated content may be incorrect.

A close up of a number

AI-generated content may be incorrect.

berisi hasil ujicoba algoritma KMP yang diimplementasikan **secara sekuensial (berjalan di CPU tanpa CUDA)**.

**Analisis Performa Menggunakan NVIDIA Nsight Systems**

Untuk mendapatkan pemahaman yang lebih mendalam mengenai karakteristik performa implementasi algoritma KMP dengan CUDA, kami menggunakan alat *profiling* (pembuatan profil) NVIDIA Nsight Systems. Nsight Systems memungkinkan visualisasi aktivitas pada CPU dan GPU secara *timeline* (garis waktu), serta pengumpulan metrik performa yang detail. Hasil *profiling* ini sangat krusial untuk mengidentifikasi *bottleneck* (kemacetan) dalam eksekusi paralel dan memahami di mana waktu komputasi dihabiskan.

Gambar-gambar hasil *profiling* yang dilampirkan menunjukkan tampilan utama Nsight Systems. Pada tampilan ini, kita dapat mengamati:

* **Timeline CPU:** Menampilkan aktivitas thread-thread CPU, termasuk kapan CPU melakukan panggilan ke fungsi-fungsi CUDA API (seperti alokasi memori, transfer data, dan peluncuran kernel).
* **Timeline GPU:** Menampilkan aktivitas eksekusi kernel di GPU dari waktu ke waktu. Segmen-segmen berwarna pada timeline GPU mengindikasikan periode di mana GPU sedang sibuk menjalankan tugas komputasi paralel.
* **Events View:** (Jika terlihat jelas) Menampilkan daftar peristiwa (panggilan CUDA API atau eksekusi kernel) beserta detail seperti waktu mulai, durasi, dan perangkat GPU yang terlibat.

Dari hasil *profiling* untuk berbagai panjang pola (mulai dari 8 hingga 1024), pola eksekusi pada GPU secara konsisten terlihat terdiri dari **beberapa blok eksekusi kernel yang terpisah-pisah**. Ini mengindikasikan bahwa implementasi KMP CUDA kami memecah proses pencarian pola menjadi beberapa tahapan komputasi yang diluncurkan secara terpisah ke GPU, dengan intervensi dari CPU di antaranya (misalnya, untuk peluncuran kernel berikutnya atau sinkronisasi). Pola eksekusi multi-tahap semacam ini seringkali menimbulkan *overhead* antar tahapan (seperti latensi peluncuran kernel dan sinkronisasi) yang dapat mempengaruhi performa total.

Analisis visual pada durasi total aktivitas GPU di timeline Nsight Systems untuk berbagai panjang pola **sesuai dengan tren waktu eksekusi GPU yang kami ukur sebelumnya**:

* Durasi total aktivitas GPU terlihat *berkurang* saat panjang pola meningkat dari 8 ke 128. Ini mendukung temuan bahwa performa GPU membaik dalam rentang pola ini.
* Durasi total aktivitas GPU terlihat *sedikit memanjang kembali* saat panjang pola meningkat dari 128 ke 256 dan 512. Ini konsisten dengan kenaikan waktu eksekusi GPU yang terukur setelah pola 128.

Meskipun tidak memberikan angka tunggal konsumsi memori dalam format tabel, profil Nsight Systems juga menyediakan bagian yang mendetail mengenai penggunaan memori GPU (Global Memory, Shared Memory, Local Memory). Informasi ini penting untuk menganalisis efisiensi akses memori oleh kernel CUDA. Pola akses memori yang tidak optimal dapat menjadi *bottleneck* signifikan dalam komputasi GPU.

Secara keseluruhan, hasil *profiling* Nsight Systems memperkuat kesimpulan bahwa, untuk ukuran data yang digunakan (100.000 karakter), implementasi KMP CUDA kami menghadapi tantangan performa yang signifikan. Profil ini menunjukkan bahwa **alasan utama mengapa GPU lebih lambat dari CPU** kemungkinan adalah kombinasi dari **overhead terkait eksekusi multi-tahap di GPU (peluncuran kernel, sinkronisasi)** dan **efisiensi eksekusi kernel paralel itu sendiri** yang bervariasi dengan panjang pola. Meskipun GPU mampu melakukan komputasi paralel, *overhead* yang terlibat dalam mengelola paralelisme tersebut pada skala masalah ini lebih besar daripada waktu eksekusi sekuensial KMP yang sudah sangat efisien di CPU.

Untuk mengoptimalkan performa implementasi KMP CUDA di masa mendatang, analisis lebih mendalam menggunakan metrik-metrik performa spesifik yang disediakan Nsight Systems (seperti *occupancy*, *memory throughput*, *thread divergence*) akan sangat diperlukan guna mengidentifikasi dan mengatasi *bottleneck* pada tingkat kernel.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

Semua gambar Nsight Systems (untuk pola 8, 16, 32, 64, 128, 256, 512, dan 1024) adalah hasil *profiling* dari implementasi KMP yang **menggunakan CUDA (berjalan di GPU)**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Panjang Data | Panjang Pola | Waktu Eksekusi CPU Sekuensial (detik) | Waktu Eksekusi GPU CUDA (detik) | Keterangan Perbandingan |
| 100.000 | 80 | 3.611 | 51.843 | CPU Jauh Lebih Cepat |
| 100.000 | 160 | 34.458 | 52.858 | CPU Jauh Lebih Cepat |
| 100.000 | 320 | 3.782 | 455.962 | CPU Jauh Lebih Cepat |
| 100.000 | 640 | 40.257 | 445.592 | CPU Jauh Lebih Cepat |
| 100.000 | 1280 | 4.909 | 37.674 | CPU Jauh Lebih Cepat |
| 100.000 | 2560 | 45.806 | 406.454 | CPU Jauh Lebih Cepat |
| 100.000 | 5120 | 35.287 | 427.505 | CPU Jauh Lebih Cepat |
| 100.000 | 1024 | 1.1 | 2.003 | CPU sangat cepat |

**Kesimpulan:**

Berdasarkan seluruh data ujicoba dan hasil *profiling*, dapat disimpulkan bahwa:

1. **Implementasi KMP Sekuensial di CPU Jauh Lebih Unggul:** Untuk tugas pencocokan string KMP pada data berukuran 100.000 karakter dan pola dalam rentang yang diuji, implementasi sekuensial di CPU memberikan performa yang jauh lebih cepat dibandingkan dengan implementasi paralel di GPU menggunakan CUDA.
2. **Overhead GPU Melebihi Manfaat Paralelisasi:** Kesenjangan performa yang besar menunjukkan bahwa *overhead* terkait penggunaan GPU (seperti waktu peluncuran dan manajemen kernel, sinkronisasi antar tahapan, dan potensi *idle time* atau inefisiensi dalam eksekusi kernel paralel pada skala ini) jauh lebih besar daripada penghematan waktu komputasi yang didapat dari eksekusi paralel, jika dibandingkan dengan kecepatan tinggi dari algoritma KMP sekuensial yang efisien di CPU modern.
3. **Efisiensi Implementasi GPU Bergantung pada Panjang Pola:** Performasi implementasi KMP CUDA Anda bervariasi dengan panjang pola. Pola 128 memberikan performa terbaik di GPU dalam rentang yang diuji, menunjukkan bahwa pada panjang pola tersebut, keseimbangan antara kerja komputasi paralel dan *overhead* relatif paling baik. Peningkatan atau penurunan performa di luar titik ini mengindikasikan bahwa cara implementasi CUDA menangani pola yang lebih pendek atau lebih panjang kurang optimal dibandingkan pada pola 128.
4. **Analisis Profil Nsight Penting:** Profil Nsight Systems sangat membantu dalam memahami mengapa GPU lebih lambat. Ia mengungkapkan bahwa eksekusi GPU tidak mulus dalam satu blok komputasi, melainkan terbagi dalam beberapa tahap, yang menambah *overhead*. Analisis lebih mendalam pada metrik spesifik di Nsight (misalnya, *thread utilization, memory access patterns, kernel launch latencies*) diperlukan untuk mengidentifikasi akar penyebab inefisiensi di dalam kernel itu sendiri.

Secara umum, hasil ini menunjukkan bahwa memindahkan algoritma ke GPU tidak secara otomatis menghasilkan percepatan. Ini sangat bergantung pada sifat algoritma (seberapa baik ia dapat diparalelkan secara masif tanpa banyak ketergantungan data), kualitas implementasi paralel, dan skala masalah. Untuk tugas yang sudah sangat cepat di CPU sekuensial (seperti KMP pada data berukuran sedang), *overhead* penggunaan GPU bisa menjadi penghalang utama performa.

Eksperimen ini memberikan data yang berharga tentang performa implementasi spesifik Anda dan menyoroti pentingnya *profiling* untuk memahami *bottleneck* dalam komputasi paralel. Untuk melihat potensi keuntungan GPU untuk KMP, pengujian pada skala data yang *jauh lebih besar* mungkin diperlukan, bersamaan dengan optimasi mendalam pada kode kernel CUDA berdasarkan wawasan dari Nsight Systems.

Aho-Corasick

Rencana awal penelitian kami meliputi komparasi performa antara algoritma Knuth-Morris-Pratt (KMP) dan algoritma Aho-Corasick. Namun, kami menghadapi kendala teknis yang signifikan dalam proses implementasi algoritma Aho-Corasick, termasuk kesulitan pada tahap *coding* serta keterbatasan atau isu kompatibilitas dengan *compiler* yang kami gunakan. Oleh karena itu, kami tidak dapat menyelesaikan ujicoba performa untuk algoritma Aho-Corasick, dan analisis perbandingan dalam laporan ini hanya berfokus pada algoritma KMP.