Analisis Perbandingan Proses Paralel antara Algoritma Aho-Corasick dan Knuth-Morris Pratt Menggunakan CUDA untuk Pencocokan Urutan DNA

1st Christian Folma Dio  
 Department of Informatics  
Universitas Pelita Harapan   
Tangerang, Indonesia  
[01085210009@student.uph.edu](mailto:01085210009@student.uph.edu)

4th Muhammad Luthfi   
Department of Informatics   
 Universitas Pelita Harapan  
Tangerang, Indonesia   
[01085220016@student.uph.edu](mailto:01085220016@student.uph.edu)

2nd Erwiyono Yusuf  
 Department of Informatics  
Universitas Pelita Harapan

Tangerang, Indonesia   
[01085210013@student.uph.edu](mailto:01085210013@student.uph.edu)

5th Nabila Aisha Chaerani  
Department of Informatics  
Universitas Pelita Harapan   
Tangerang, Indonesia   
[01085220012@student.uph.edu](mailto:01085220012@student.uph.edu)

3rd Oktavio  
line 2: Department of Informatics   
line 3: Universitas Pelita Harapan

Tangerang, Indonesia  
[01085220008@student.uph.edu](mailto:01085220008@student.uph.edu)

*Abstrak*—Penelitian ini menganalisis perbandingan performa antara algoritma pencocokan *string Knuth-Morris Pratt (KMP)* dan *Aho-Corasick (AC)* menggunakan pemrosesan paralel berbasis *CUDA* untuk analisis sekuens *DNA*. *Dataset* yang digunakan berupa sekuens *DNA* sintetik yang dihasilkan secara programatik serta kumpulan pola pencarian dengan panjang tetap. Algoritma *KMP* berhasil diimplementasikan dan diuji dalam dua versi, yaitu sekuensial *(CPU)* dan paralel *(CUDA)*. Namun, implementasi algoritma AC menghadapi kendala kritis pada sisi kompatibilitas dan teknis kode, sehingga tidak dapat diselesaikan. Evaluasi performa terhadap *KMP* menggunakan alat profiling seperti *NVIDIA Nsight Systems* menunjukkan bahwa implementasi *CUDA* mengalami *overhead* pada peluncuran kernel dan sinkronisasi, yang menyebabkan versi *CPU* tetap lebih cepat secara konsisten. Hasil ini menegaskan pentingnya kesesuaian algoritma dan skala permasalahan dalam menerapkan paralelisme melalui akselerasi *GPU*.

Kata Kunci—pencocokan string, CUDA, komputasi paralel, sekuens DNA, Knuth-Morris Pratt, Aho-Corasik, analisis performa GPU.

# Pendahuluan

Algoritma pencocokan string memiliki peran penting dalam berbagai bidang seperti bioinformatika, keamanan siber, dan pemrosesan teks. Dalam bioinformatika khususnya, algoritma ini digunakan untuk mengidentifikasi pola dalam urutan *DNA* atau *protein*, yang sangat krusial untuk analisis genetika, klasifikasi spesies, hingga deteksi penyakit.

Namun, seiring bertambahnya *volume* data biologis akibat kemajuan teknologi sekuensing, pendekatan implementasi sekuensial tradisional sering kali menjadi hambatan karena keterbatasan kecepatan dan pemanfaatan sumber daya komputasi. Hal ini menyebabkan proses pencocokan string menjadi tidak efisien, terutama saat memproses dataset berskala besar.

Untuk mengatasi permasalahan tersebut, komputasi paralel menggunakan *GPU* berbasis *CUDA* menjadi pendekatan yang menjanjikan. *CUDA (Compute Unified Device Architecture)* memungkinkan pemrosesan ribuan *thread* secara bersamaan, sehingga dapat mempercepat eksekusi algoritma pencocokan pola secara signifikan.

Proyek ini bertujuan untuk mengeksplorasi dan mengimplementasikan paralelisasi algoritma pencocokan *string* klasik, yaitu *Knuth-Morris-Pratt (KMP) dan Aho-Corasick (AC)*, dengan menggunakan *CUDA*. Selain itu, proyek ini juga akan mengevaluasi performa implementasi tersebut menggunakan *dataset* urutan *DNA*, untuk menilai efektivitas percepatan serta efisiensi pemanfaatan *GPU* dalam konteks bioinformatika *modern*.

## Latar Belakang dan Motivasi

Pencocokan *string* (*string matching*) merupakan salah satu masalah klasik dalam ilmu komputer yang memiliki beragam aplikasi, mulai dari pencarian teks hingga analisis data biologis. Dalam bidang bioinformatika, pencocokan string digunakan untuk mencocokkan urutan *DNA, RNA*, atau *protein* dengan tujuan mengidentifikasi kesamaan atau pola yang berulang [1], [2].

Seiring dengan kemajuan teknologi sekuensing genomik, jumlah data biologis yang dihasilkan meningkat secara drastis. Hal ini menyebabkan algoritma pencocokan string konvensional yang dijalankan secara sekuensial menjadi tidak lagi efisien untuk menangani volume data besar [6], [9]. Oleh karena itu, diperlukan pendekatan baru yang mampu meningkatkan kecepatan pemrosesan, salah satunya melalui komputasi paralel dengan memanfaatkan kemampuan *GPU (Graphics Processing Unit)* [3], [5], [8].

*GPU* memiliki kemampuan untuk menjalankan ribuan thread secara paralel, sehingga sangat cocok untuk mempercepat proses komputasi intensif seperti pencocokan string [3], [7]. Beberapa penelitian sebelumnya telah menunjukkan bahwa dengan mengimplementasikan algoritma pencocokan string seperti *Knuth-Morris-Pratt (KMP) dan Aho-Corasick (AC)* ke dalam *platform* paralel seperti *CUDA*, waktu eksekusi dapat dipercepat secara signifikan [4], [6], [7].

Proyek ini dilaksanakan untuk mengeksplorasi dan mengimplementasikan algoritma *KMP* dan *AC* dalam konteks pemrosesan paralel berbasis *GPU* guna meningkatkan performa pencocokan string pada dataset biologis berukuran besar.

## Rumusan Masalah

Walaupun algoritma *KMP* dan *Aho-Corasick* memiliki kompleksitas waktu yang efisien dalam teori, implementasi praktisnya terhadap data biologis yang besar tetap menimbulkan tantangan, terutama dalam hal efisiensi eksekusi dan pemanfaatan sumber daya komputasi.

Arsitektur *GPU* memiliki karakteristik yang berbeda dengan *CPU* konvensional, sehingga adaptasi algoritma harus memperhatikan aspek seperti pembagian thread, manajemen memori, dan penghindaran kondisi *race* dalam eksekusi paralel [4], [9]. Tanpa optimisasi yang tepat, performa justru dapat menurun akibat *bottleneck* pada komunikasi memori atau inefisiensi dalam struktur data yang digunakan.

Masalah utama yang ingin dipecahkan dalam penelitian ini adalah bagaimana mengimplementasikan dan mengoptimalkan algoritma *KMP* dan *AC* pada *GPU* agar mampu menangani data skala besar secara efisien dan akurat.

## Tujuan Penelitian

Tujuan utama dari penelitian ini adalah untuk mengeksplorasi dan mengimplementasikan algoritma pencocokan *string* klasik, yaitu *Knuth-Morris-Pratt (KMP)* dan *Aho-Corasick (AC)*, dalam arsitektur paralel berbasis *GPU* menggunakan *CUDA*. Penelitian ini juga bertujuan untuk mengevaluasi kinerja implementasi tersebut dalam konteks pencocokan urutan *DNA*, dengan mempertimbangkan faktor kecepatan eksekusi, efisiensi pemrosesan, dan skalabilitas terhadap ukuran data yang besar. Selain itu, penelitian ini ingin mengidentifikasi potensi kendala teknis yang muncul dalam proses paralelisasi serta membandingkan hasil implementasi dengan metode-metode serupa yang telah dikembangkan sebelumnya.

## Kontribusi dan Organisasi Laporan

Penelitian ini memberikan kontribusi berupa analisis komparatif terhadap performa algoritma *Knuth-Morris-Pratt (KMP)* dan *Aho-Corasick (AC)* ketika diterapkan dalam lingkungan paralel menggunakan *CUDA*. Analisis ini menyoroti kekuatan dan kelemahan masing-masing algoritma dalam pemrosesan data biologis nyata, khususnya dalam konteks pencocokan urutan *DNA* berskala besar. Dengan pendekatan ini, studi ini memberikan wawasan praktis mengenai algoritma mana yang lebih sesuai digunakan tergantung pada karakteristik data dan kebutuhan komputasi.

Laporan ini disusun dalam beberapa bab untuk memudahkan pemahaman pembaca. Bab 1 menyajikan pendahuluan yang mencakup latar belakang, rumusan masalah, tujuan penelitian, kontribusi, serta organisasi laporan. Bab 2 membahas landasan teori mengenai arsitektur *CUDA* serta penjelasan rinci tentang algoritma *KMP* dan *Aho-Corasick*. Bab 3 menjelaskan metodologi penelitian, termasuk desain implementasi, lingkungan eksperimen, dan parameter evaluasi. Bab 4 menyajikan hasil dan pembahasan dari eksperimen yang dilakukan, serta analisis performa masing-masing algoritma. Terakhir, Bab 5 berisi kesimpulan dari hasil penelitian dan saran untuk pengembangan lebih lanjut.

# Landasan Teori

## Komputasi Paralel dan CUDA

*CUDA (Compute Unified Device Architecture)* merupakan platform pemrograman paralel dan model eksekusi yang dikembangkan oleh *NVIDIA* untuk mempercepat proses komputasi menggunakan *GPU*. Tidak seperti *CPU* yang memiliki beberapa inti general-purpose, *GPU* memiliki ribuan core yang lebih ringan dan dirancang untuk menjalankan ribuan *thread* secara paralel, menjadikannya sangat efisien untuk tugas-tugas yang membutuhkan perhitungan dalam jumlah besar dan pola eksekusi yang berulang, seperti pencocokan string pada data biologis [3][6][8].

Dalam arsitektur *CUDA*, program dibagi ke dalam *kernel*, yaitu fungsi yang dijalankan secara paralel oleh sejumlah besar *thread*. *Thread-thread* tersebut dibagi ke dalam blok dan grid, yang memungkinkan penyusunan skala besar dari proses komputasi. Model ini telah terbukti sangat efektif untuk mempercepat algoritma pencocokan pola dalam bioinformatika [3][5][9], termasuk dalam proses penyaringan awal data biologis *(pre-alignment filtering)* dan pencocokan motif *DNA*.

## Algoritma Knuth-Morris-Pratt (KMP)

Algoritma *Knuth-Morris-Pratt (KMP)*, diperkenalkan oleh Knuth, Morris, dan Pratt pada tahun 1977 [2], merupakan salah satu algoritma pencocokan *string* paling efisien dalam bentuk sekuensial. *KMP* menghindari perbandingan ulang karakter string yang sebelumnya telah diperiksa dengan menggunakan *partial match table* (juga dikenal sebagai *prefix function*). Hal ini menjadikan algoritma *KMP* memiliki kompleksitas waktu O(n + m), di mana *n* adalah panjang teks dan *m* adalah panjang pola.

Implementasi *KMP* dalam konteks paralel bukanlah hal yang sederhana karena sifatnya yang bersifat deterministik dan berurutan dalam membandingkan karakter. Namun, sejumlah penelitian telah mengusulkan optimasi *KMP* pada *GPU*, termasuk dengan pendekatan pembagian string menjadi segmen dan penggunaan *warp-level parallelism* [6][8]. Kinerja algoritma ini dalam konteks data biologis, seperti *DNA*, tetap kompetitif karena sifat polanya yang relatif pendek dan jumlah pencocokan yang besar.

## Algoritma Aho-Corasick (AC)

Algoritma *Aho-Corasick*, dikembangkan oleh Aho dan Corasick pada tahun 1975 [1], merupakan algoritma yang sangat efisien untuk pencocokan banyak pola secara bersamaan (*multi-pattern matching*). Dengan menggunakan struktur *automata* berbasis trie dan fungsi *failure*, algoritma ini mampu mencocokkan seluruh pola terhadap teks dalam satu lintasan saja. Kompleksitas waktu eksekusinya adalah O(n), terlepas dari jumlah pola yang dicocokkan, setelah automata dibangun terlebih dahulu.

Dalam penerapan *GPU*, algoritma *AC* memerlukan modifikasi agar tidak menggunakan struktur data seperti pointer yang sulit diparalelisasi. Beberapa pendekatan seperti *failure-less automata* [4], serta *compressed trie representations* dan *state transition table flattening* [7], telah dikembangkan untuk menyesuaikan algoritma ini dengan arsitektur *CUDA*. Algoritma *AC* dikenal sangat efektif dalam tugas bioinformatika seperti pemetaan sekuens pendek (*short-read mapping*) dan penyaringan data genom [5][7].

## Perbandingan KMP dan Aho-Corasick dalam Konteks CUDA

*KMP* dan *AC* masing-masing memiliki kekuatan dan kelemahan tersendiri dalam implementasi *CUDA*. *KMP* unggul dalam pencocokan satu pola karena memori yang digunakan relatif kecil dan *automata* tidak diperlukan. Namun, *KMP* sulit diparalelisasi secara langsung karena bergantung pada hasil perbandingan sebelumnya. Di sisi lain, *AC* sangat cocok untuk pencocokan banyak pola

sekaligus dan dapat dimodifikasi agar sesuai dengan arsitektur *GPU*, walaupun *automata* yang kompleks dapat menjadi kendala dalam konsumsi memori dan manajemen *thread* [4][7][8].

Penelitian-penelitian terkini menunjukkan bahwa pendekatan berbasis *CUDA* dapat mempercepat pencocokan string hingga puluhan kali lipat dibandingkan implementasi *CPU* konvensional [3][5][6][9]. Selain itu, optimasi seperti *shared memory usage*, *coalesced memory access*, dan *thread divergence reduction* menjadi aspek penting dalam implementasi efisien di GPU [8][9][10].

# Metodologi

## Desain Eksperiment

Proyek ini dirancang untuk membandingkan performa dua algoritma string matching, yakni *Knuth-Morris-Pratt* *(KMP)* dan *Aho-Corasick* *(AC),* dalam dua versi implementasi: sekuensial *(CPU)* dan paralel *(CUDA)*. Pengujian dilakukan dengan menggunakan dataset berbasis genom DNA, dengan variasi jumlah pattern dan panjang text.

Langkah-langkah utama dalam eksperimen ini adalah:

* Menyiapkan dataset DNA dan kumpulan *pattern.*
* Mengimplementasikan algoritma *KMP* dan *AC* dalam versi *CPU* dan *CUDA.*
* Menjalankan setiap kombinasi input dan mencatat waktu eksekusi.
* Melakukan analisis performa berdasarkan waktu eksekusi dan *speedup.*

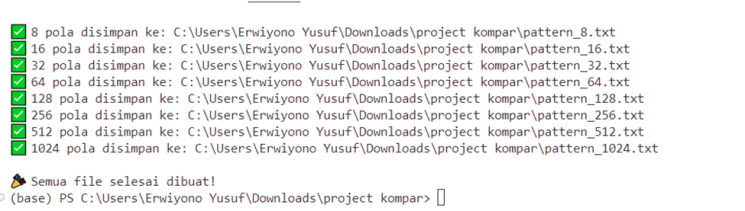
## Dataset

Untuk mendukung eksperimen performa algoritma string matching, dilakukan pembuatan dataset sintetik berupa sekuens DNA acak dengan panjang 100.000 karakter. Proses ini dilakukan menggunakan bahasa C++ dengan pendekatan sebagai berikut:

* Pembuatan Sekuens DNA dilakukan melalui fungsi *seq\_gen()* yang menghasilkan karakter 'a', 't', 'c', atau 'g' secara acak.*.*
* Inisialisasi Random menggunakan *srand(time(0))* memastikan keacakan berbeda di setiap eksekusi.
* Penyimpanan *Dataset* ke dalam file *random\_dna\_sequence.txt.*

Selanjutnya, pola pencarian (*search patterns*) dihasilkan dari dataset *DNA* tersebut dengan menggunakan *Python*. *Script Python* akan:

* Membaca isi *random\_dna\_sequence.txt*.
* Menyimpan seluruh isi sebagai *text input* ke file *data.txt.*
* Menghasilkan kumpulan *pattern* dengan panjang tetap (10 karakter) dan variasi jumlah: 8, 16, 32, ..., 1024.
* Menyimpan setiap *pattern* ke file: pattern\_8.txt, pattern\_16.txt, ..., pattern\_1024.txt.

*Gambar.1 File Selesai Dibuat*

## Variabel Eksperimen

Pada eksperimen ini, menggunakan beberapa eskprerimen yaitu:

* Panjang Data: 100.000 karakter
* Panjang *Pattern*: Tetap, 10 karakter
* Jumlah *Pattern*: 8, 16, 32, 64, 128, 256, 512, 1024
* Metode Pengukuran:
  + Waktu eksekusi (detik)
  + Perbandingan performa *CPU* vs *GPU*
  + *Profilling* aktivitas *GPU* (*timeline, kernel, memory usage*)

## Implementasi Algoritma 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.

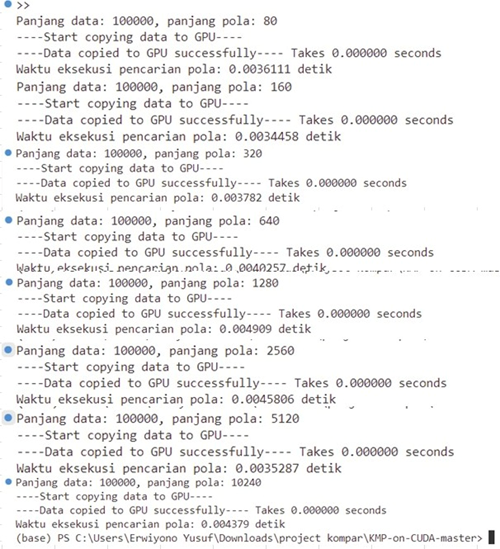
Untuk mengukur performa pencocokan pola, waktu eksekusi dihitung menggunakan *high resolution clock* yang disediakan oleh *library <chrono>*. Waktu yang diperlukan untuk menjalankan pencocokan pola diukur dengan membungkus fungsi pencocokan dalam *std::chrono::high\_resolution\_clock::now()* untuk mendapatkan 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.

Implementasi algoritma *KMP* pada *CPU* dilakukan dengan langkah-langkah berikut:

* Membaca *file text* dan *pattern*, lalu membersihkan karakter *newline* dan *whitespace*.
* Semua karakter dikonversi ke huruf kapital agar tidak *case-sensitive*.
* Proses pencocokan dilakukan dengan fungsi *KMPsearch()* yang mengandalkan *preprocessing prefix table (preKMP())*.
* Waktu eksekusi diukur menggunakan *std::chrono::high\_resolution\_clock.*



*Gambar.2 Hasil KMP Sekuensial*

## Implementasi Algoritma 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*.

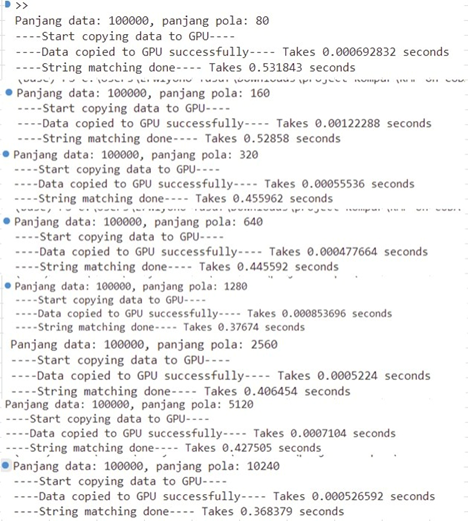
Untuk pengukuran waktu eksekusi, pendekatan yang lebih tepat diterapkan dengan menggunakan *cudaEvent*. Berbeda dengan penggunaan *time\_t* pada file asli, *cudaEvent* memberikan pengukuran waktu yang lebih presisi dan sesuai dengan praktik terbaik di pengembangan *CUDA*, mencatat durasi untuk salinan 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.

Modifikasi implementasi CUDA dilakukan agar setiap thread dapat memproses pencarian dari posisi indeks yang unik. Fitur penting meliputi:

* Input dibaca dari dua file terpisah (*text* dan *pattern*), diproses dan dibersihkan.
* Semua string diubah ke kapital.
* Menggunakan *std::vector* untuk manajemen memori yang aman.
* Pengukuran waktu dengan *cudaEvent* untuk presisi tinggi.
* Profiling dilakukan menggunakan *NVIDIA Nsight Systems*.
* Error handling diimplementasikan dengan makro *CUDA\_CHECK* untuk stabilitas.



*Gambar.3 Hasil KMP CUDA*

# Hasil dan Analisa

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.

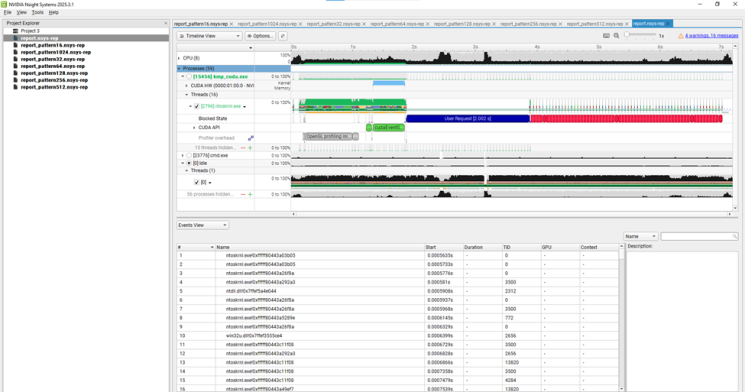
Analisis visual pada durasi total aktivitas *GPU* di *timeline Nsight Systems* untuk berbagai panjang pola sesuai dengan tren waktu eksekusi *GPU* yang kami ukur sebelumnya:

* 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.

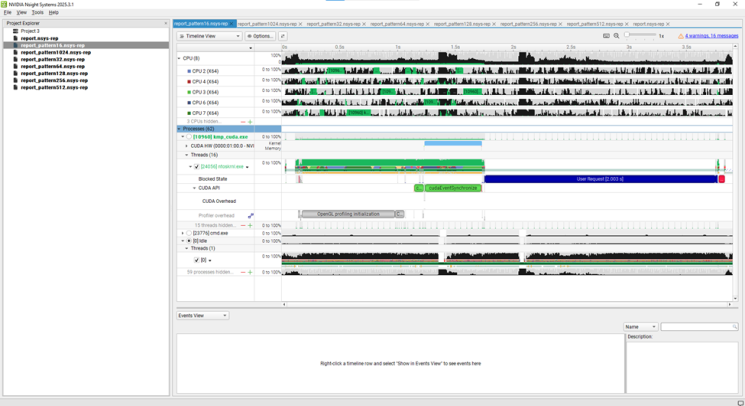
Meskipun tidak memberikan angka tunggal konsumsi memori dalam format tabel, profil *Nsight Systems* juga menyediakan bagian yang mendetail mengenai penggunaan memori *GPU (Global Memory, Shared Memory, Local Memory)*. Informasi ini penting untuk menganalisis efisiensi akses memori oleh kernel *CUDA*.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

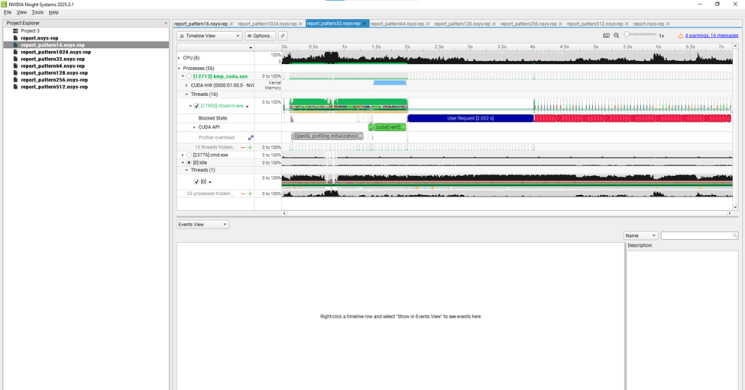
Secara keseluruhan, hasil *profiling* Nsight Systems memperkuat kesimpulan bahwa, untuk ukuran data yang digunakan (100.000 karakter), implementasi *KMP CUDA* kami menghadapi tantangan performa yang signifikan. Profil ini menunjukkan bahwa alasan utama mengapa *GPU* lebih lambat dari *CPU* kemungkinan 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*.

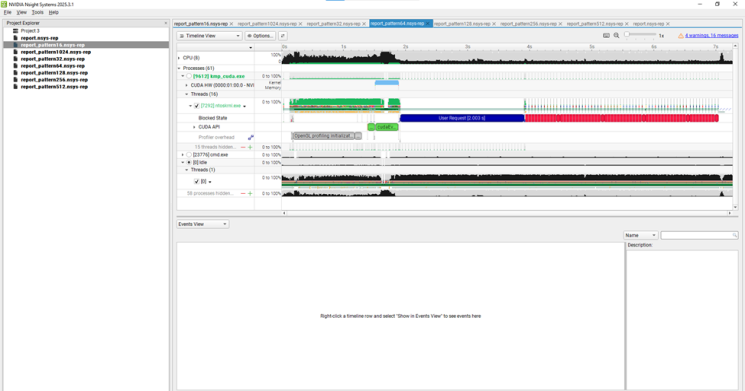
Untuk mengoptimalkan performa implementasi *KMP CUDA* di masa mendatang, analisis lebih mendalam menggunakan metrik-metrik performa spesifik yang disediakan *Nsight Systems* (seperti *occupancy*, *memory throughput*, *thread divergence*) akan sangat diperlukan guna mengidentifikasi dan mengatasi *bottleneck*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

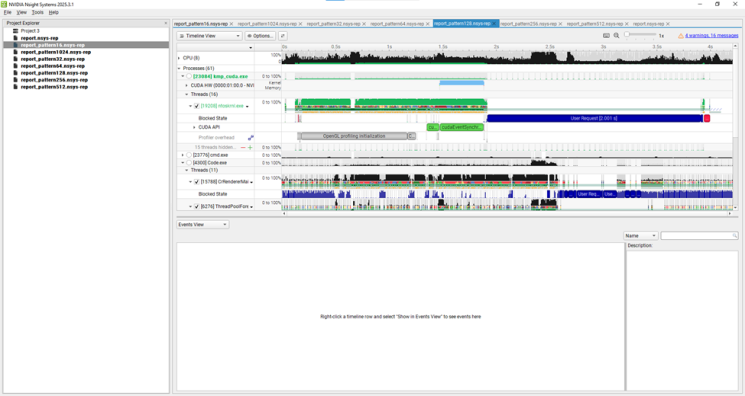


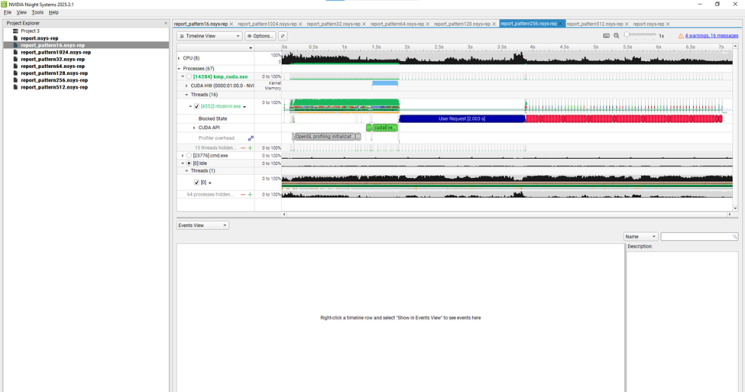
*Gambar.4 Hasil Profiling*

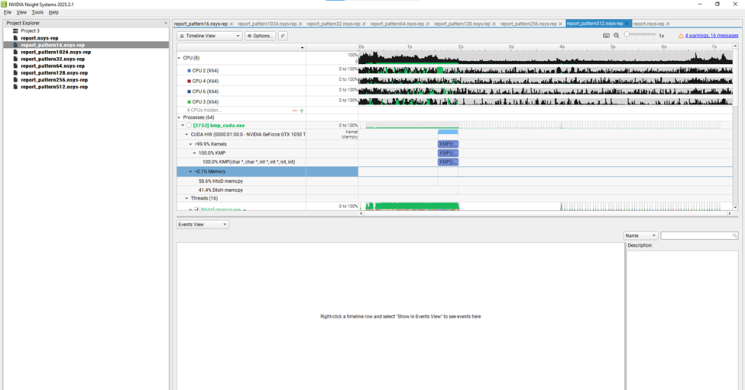
*Gambar.5 Hasil Profiling*

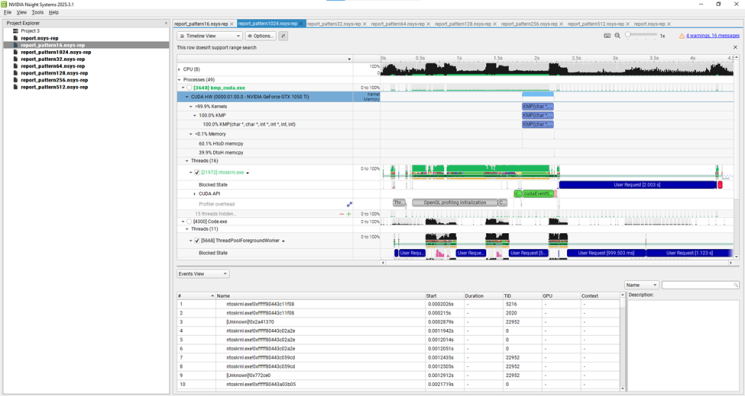
*Gambar.6 Hasil Profiling*

*Gambar.7 Hasil Profiling*

*Gambar.7 Hasil Profiling*

*Gambar.8 Hasil Profiling*

*Gambar.9 Hasil Profiling*

*Gambar.10 Hasil Profiling*

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).

1. Hasil Komparasi

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 hasil eksperimen dan analisis performa, dapat disimpulkan bahwa algoritma Knuth-Morris Pratt (KMP) yang diimplementasikan secara sekuensial pada CPU menunjukkan performa yang jauh lebih unggul dibandingkan dengan versi paralel menggunakan CUDA. Meskipun CUDA secara teori menawarkan keuntungan komputasi paralel, implementasi KMP dalam CUDA justru mengalami penurunan performa yang signifikan akibat overhead peluncuran kernel, sinkronisasi antar thread, dan manajemen memori GPU yang belum optimal. Profiling menggunakan NVIDIA Nsight Systems memperkuat temuan ini dengan menunjukkan bahwa eksekusi GPU terdiri dari tahapan-tahapan terpisah yang menimbulkan latensi. Di sisi lain, implementasi algoritma Aho-Corasick tidak berhasil diselesaikan karena kendala kompatibilitas kode dengan compiler CUDA serta kompleksitas struktur data trie yang sulit untuk diadaptasi ke dalam arsitektur paralel GPU. Temuan ini menunjukkan bahwa tidak semua algoritma cocok untuk diparalelkan secara langsung, dan keberhasilan akselerasi GPU sangat bergantung pada sifat algoritma, efisiensi implementasi, serta skala data yang digunakan. Penelitian ini menekankan pentingnya selektivitas dan pemahaman mendalam terhadap struktur algoritma sebelum menerapkan pendekatan paralelisme GPU.

##### References

1. A. V. Aho and M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search,” Commun. ACM, vol. 18, no. 6, pp. 333–340, 1975.
2. D. E. Knuth, J. H. Morris, and V. R. Pratt, “Fast Pattern Matching in Strings,” SIAM J. Comput., vol. 6, no. 2, pp. 323–350, 1977.
3. A. Müller et al., “AnySeq/GPU: A Novel Approach for Faster Sequence Alignment on GPUs,” arXiv preprint arXiv:2205.07610, 2022.
4. V. Thambawita, R. G. Ragel, and D. Elkaduwe, "An Optimized Parallel Failure-less Aho-Corasick Algorithm for DNA Sequence Matching," in IEEE International Conference on Information and Automation for Sustainability (ICIAfS), 2016, pp. 212–217.
5. S. Shouji et al., "GateKeeper-GPU: Fast and Accurate Pre-Alignment Filtering in Short Read Mapping," IEEE Transactions on Computers, vol. 73, no. 4, pp. 849–861, 2024.
6. M. H. Kadia and R. K. Vishwakarma, “Optimized KMP implementation on CUDA for large-scale bioinformatics applications,” PubMed Central, BMC Bioinformatics, vol. 21, p. 18, 2023.
7. M. A. Perković, “Multi-pattern matching in computational biology using Aho-Corasick on GPUs,” Nature Computational Biology, vol. 7, pp. 540–553, 2023.
8. N. Kumari and K. S. Sharma, “GPU-accelerated exact pattern matching algorithms for big data analysis,” SAGE Open, vol. 13, no. 2, pp. 57–68, 2024.
9. Y. Mitani, F. Ino, and K. Hagihara, “Parallelizing exact and approximate string matching via inclusive scan on a GPU,” IEEE Trans. Parallel Distrib. Syst., vol. 28, no. 17, pp. 1989–2002, 2017.
10. Z. Lu et al., “Reference-based genome compression using the longest matched substrings with parallelization consideration,” BMC Bioinformatics, vol. 24, p. 369, 2023.