ANALISIS PENYELESAIAN TRAVELING SALESMAN PROBLEM DENGAN METODE *BRUTE FORCE* MENGGUNAKAN *GRAPHIC PROCESSING UNIT*

Andrew Wilson, Yuliant Sibaroni, Izzatul Ummah

Prodi Ilmu Komputasi, Fakultas Informatika Telkom University, Bandung
saragiandrewilson@yahoo.com, yuliant2000@yahoo.com, izza.hpc@telkomuniversity.ac.id

Abstrak

Komputasi paralel sangat dibutuhkan dalam masalah komputasi yang memiliki kompleksitas tinggi sehingga dapat dikerjakan dalam waktu yang cepat. Komputasi paralel membutuhkan *hardware* yang memiliki kinerja tinggi dan *software* yang memadai untuk mengeksekusi algoritma secara parallel. Salah satu pendekatan komputasi paralel adalah *Graphic Processing Unit* (GPU) *Computing*, dimana dalam sebuah GPU terdapat banyak *thread* yang mampu ditugaskan secara paralel. *Brute Force* adalah teknik pemecahan masalah yang sangat umum dan dapat digunakan untuk menyelesaikan masalah komputasi untuk menemukan jalur terbaik. *Brute Force* bekerja dengan meng-enumerasi semua kandidat kemungkinan yang ada, sehingga menghasilkan solusi yang terbaik. Pada penelitian ini akan diimpelementasikan *Brute Force* dengan teknik *Exhaustive Search* pada GPU, dan menganalisis jumlah *threadProcess* pada setiap *thread* dan pengaruh *block* dan *thread* pada GPU. Setelah dilakukan penelitian dan uji statistik, didapatkan bahwa Perubahan *threadProcess* yang semakin besar akan mengakibatkan persentase penurunan waktu semakin besar dan juga semakin banyak *threadProcess* maka kecepatan komputasi GPU akan menuju satu titik, karena kecepatan komputasi pada device GPU telah mencapai titik maksimal. Pada percobaan yang dilakukan, GPU akan mengungguli kinerja CPU ketika jumlah *maxCity*>10. Kemudian terdapat perbedaan waktu yang signifikan antara *threadProcess* 1 dan *threadProcess* 2 sebesar 41.35%, hal ini disebabkan karena device GPU tersebut tidak dipakai secara maksimal. Untuk presentase penurunan waktu terbesar ketika menggunakan metode *parallel* adalah ketika *user* mampu menemukan kombinasi *thread* dan *block* yang pas, karena penambahan jumlah *thread* dan *block* tidak selalu menjamin penurunan kecepatan waktu pencarian, dimana waktu pencarian akan mencapai titik konvergen.

Kata kunci : ***TSP, Brute Force, GPU, CUDA.***

Abstract

Parallel computing is needed in computing problems that have high complexity so that it can be done in a short time. Parallel computing requires high-performance hardware and software that is sufficient to execute the algorithm in parallel. One approach to parallel computing is the Graphic Processing Unit (GPU) computing, which in a GPU capable there are many threads in parallel assigned. Brute Force is a problem-solving technique that is very common and can be used to solve computational problems to find the best path. Brute Force works by clicking on enumerating all possibilities existing candidates, resulting in the best solution. In this study, Brute Force with techniques Exhaustive Search will be implemented on the GPU, and the amount threadProcess on every thread and the effect block and threads on the GPU will be analyzed as well. After some research and statistical tests, it was found that the greater threadProcess changes will result in a decrease in the percentage of time, and adding more threadProcess will cause GPU computing speeds toward a point, because the speed of computation on the GPU device has reached the maximum point. In the experiments conducted, the GPU will surpass the performance of the CPU when the number maxCity> 10. Then there is a significant time difference between threadProcess threadProcess 1 and 2 at 41.35%, this is because the GPU device is not used optimally. For the largest percentage decrease in time when using the parallel method is when a user is able to find a combination of thread and block fitting, because increasing the number of threads and blocks do not always guarantee a decrease in the speed of search time, where the search time will reach the converging point.

1. Pendahuluan

Perkembangan Penelitian dalam permasalahan transportasi masih terus dilakukan untuk mencari algoritma baru untuk menyelesaikan sebuah permasalahan yang menghasilkan solusi yang lebih optimal dari penelitian sebelumnya. Salah satu permasalahan transportasi yang paling sering dibahas adalah *Traveling salesman problem* (TSP). TSP merupakan permasalahan dimana seorang s*alesman* harus mengunjungi setiap kota, s*alesman* tersebut berangkat dari kota awal dan berakhir pada kota yang sama, serta setiap kota harus dikunjungi tidak lebih dari satu kali. Fungsi tujuan dari TSP adalah untuk mencari rute terpendek yang bisa dilalui oleh *salesman* sehingga dihasilkan total jarak terpendek [6].

Masalah *traveling salesman problem* masih sering kali ditemukan dalam kehidupan sehari-hari. Masalah nyata yang sering terjadi misalnya pada logistik perusahaan, dimana sebuah perusahaan harus dapat menentukan berapa banyak jumlah yang dibutuhkan untuk mengantarkan suatu barang atau jasa dari supplier kepada pelanggan sehingga biaya yang dibutuhkan seminimum mungkin. Selain itu ada juga masalah transportasi untuk menentukan jalur angkutan umum yang efisien dan pemasangan jaringan telekomunikasi[5]. Manfaat menyelesaikan kasus TSP juga akan menghasilkan prospek yang bagus kedepannya, karena penyelesaian kasus TSP ini juga dapat digunakan untuk navigasi seperti GPS yang sangat populer saat ini.

Tugas akhir ini membahas penerapan Algoritma *Brute force* pada *Graph*ic Processing Unit (GPU) dalam menyelesaikan permasalahan pada *Traveling salesman problem*. Penelitian dilakukan dengan membandingkan performa antara GPU yang bekerja secara parallel dengan performa CPU. GPU yang digunakan dalam penelitian ini adalah NVIDIA CUDA, karena CUDA menggunakan bahasa "C" standar, dengan beberapa ekstensi yang simpel. Arsitektur CUDA memungkinkan GPU menjadi arsitektur terbuka seperti layaknya CPU. Namun, tidak seperti CPU, GPU memiliki arsitektur banyak-inti yang pararel. Setiap inti memiliki kemampuan untuk menjalankan ribuan *thread* secara simultan. Jika aplikasi yang dijalankan sesuai dengan arsitektur ini, GPU dapat menyediakan keuntungan yang lebih besar dari segi performa proses aplikasi tersebut [9].

Berbagai metode telah dilakukan untuk menyelesaikan permasalahan *Traveling salesman problem*, di antaranya *Tabu Search, Genetic Algorithm, Ant Colony Optimization, Neural Network*, dan beberapa metoda lain. Dalam penelitian ini digunakan pendekatan metode *Brute force.* Metode *Brute force* merupakan metode pencarian dengan kelebihan solusi yang dihasilkan akan selalu bernilai tepat. Sejauh ini sudah ada banyak penelitian yang meneliti tentang TSP dengan menggunakan *Brute force*. Hasil dari penelitian tersebut menyimpulkan beberapa kendala dan juga kendala yang sama seperti masih membutuhkan waktu yang besar. Karena itu melihat dari beberapa penelitian yang sudah ada, penulis mencoba melakukan paralelisasi dengan platform lain yaitu CUDA oleh NVIDIA. Paralelisasi akan menggunakan Metode *Brute force* pada kasus TSP dilakukan dengan Teknik *Exhaustive Search*.

2. Dasar Teori

2.1. *Traveling Salesman Problem*

Traveling salesman problem merupakan salah satu masalah yang paling populer dalam ilmu komputer [12]. Dimana *traveling salesman problem* merupakan suatu masalah dalam dunia optimasi, dimana ada sebuah kota awal dan sejumlah *n* kota untuk dikunjungi dan seorang salesman dituntut untuk memulai perjalanan dari kota awal ke seluruh kota dengan mengunjungi setiap kota tepat satu kali [6]. Tujuan dari permasalahan ini adalah untuk menemukan rute yang paling singkat. Permasalahan dapat dideskripsikan dengan menyatakan sebuah kota menjadi sebuah simpul graf, dan jalan yang menghubungkan kota sebagai sisi. Bobot pada sisi menyatakan jumlah antara dua buah kota. Dengan kata lain, *Traveling salesman problem* dapat dimodelkan sebagai sebuah graf.

Graf yang digunakan untuk merepresentasikan kasus tersebut yaitu dengan graf lengkap, yaitu sirkuit Hamilton. Sirkuit Hamilton merupakan siklus grafik (loop tertutup) melalui grafik yang mengunjungi setiap node tepat satu kali [13]. Sebuah siklus Hamiltonian grafik dapat diidentifikasi dengan menggunakan Hamiltonian *Cycle*.

Masalah ini akan sulit diselesaikan apabila terdapat jumlah kota yang banyak. Tantangan dalam penyelesaian masaah ini bukan hanya tentang rute yang baik, tetapi harus menjamin bahwa hasil yang didapat merupakan rute terpendek [2]. Penyelesian masalah TSP mengharuskan untuk melakukan perhitungan terhadap semua rute yang diperoleh, baru kemudian memilih salah satu rute terpendek. Untuk itu jika terdapat n kota yang harus dikunjungi, maka diperlukan proses pencarian sebanyak *n/2* rute dengan cara ini waktu komputasi yang diperlukan akan jauh meningkat seiring dengan bertambahnya jumlah kota yang harus dikunjungi [6].

2.2. Metode *Brute Force*

Metode *Brute force* merupakan pendekatan yang langsung dalam memecahkan suatu masalah, yang didasarkan pada pernyataan masalah serta mendefinisikan terhadap konsep yang dilibatkan secara langsung. Pemecahan masalah menggunakan

Metode *Brute force* sangat sederhana, langsung dan jelas. [13]. Karakteristik pada Metode *Brute force* :

1. Jumlah langkah yang di perlukan besar.
2. Dipakai sebagai dasar dalam menemukan suatu solusi yang lebih efisien atau kreatif.
3. Hampir semua masalah dapat diselesaikan dengan metode ini.
4. Digunakan sebagai dasar dalam perbandingan kualitas suatu algoritma

2.3. Algoritma *Brute Force* Dengan Teknik *Exhaustive Search* Pada *Traveling Salesman Problem*

Exhaustive search adalah teknik pencarian solusi secara solusi *brute force* untuk masalah yang melibatkan pencarian elemen dengan sifat khusus. Biasanya di antara objek-objek kombinatorik seperti permutasi, kombinasi, atau himpunan bagian dari sebuah himpunan [6]. Penggunaan *Brute force* pada permasalahan TSP dapat di ilustrasikan sebagai berikut:

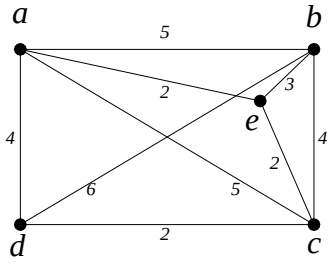
Diberikan *n* buah kota serta diketahui jarak antara setiap kota satu sama lain. Temukan perjalanan (*tour*) terpendek yang melalui setiap kota lainnya hanya sekali dan kembali lagi ke kota asal keberangkatan. Permasalahan *TSP* ini tidak lain adalah menemukan sirkuit Hamilton dengan

bobot minimum. Langkah-langkah metode

exhaustive search:

1. Buat list (Enumerasi) setiap solusi yang mungkin dengan cara yang sistematis.
2. Evaluasi setiap kemungkinan solusi satu per satu, terlebih solusi yang *mirror* , bisa di

   keluarkan. Kemudian simpan solusi terbaik yang ada sampai saat ini.
3. Setelah pencarian berakhir, umumkan solusi terbaik.
4. Sumber daya yang dibutuhkan dalam pencarian solusi menggunakan *Exhaustive Search* sangat besar, meskipun algoritma *exhaustive* secara teoritis menghasilkan solusi.



Gambar 1. *Graph* ABCD

Pada Gambar 1, *graph* memiliki (4-1)! = 6 sirkuit Hamilton yaitu:

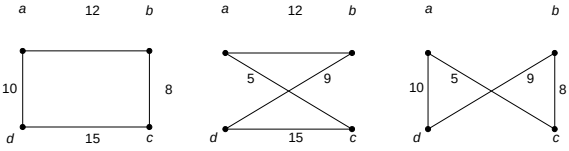
L1 = (A,B,C,D,A) = 10 + 12 + 8 + 15 = 45
L2 = (A,B,D,C,A) = 12 + 5 + 9 + 15 = 41
L3 = (A,C,B,D,A) = 10 + 5 + 9+ 8 = 32
L4 = (A,C,D,B,A) = 12 + 5 + 9+ 15 = 41
L5 = (A,D,B,C,A) = 10 + 5 + 9+ 8 = 32
L6 = (A,D,C,B,A) = 10 + 12 + 8+ 15 = 45

Jika contoh diselesaikan dengan metode *exhaustive search*, maka kita harus mengenumerasi sebanyak (*n* – 1)! buah sirkuit Hamilton, menghitung setiap bobotnya, dan memilih sirkuit Hamilton dengan bobot terkecil.

Dari algoritma *exhaustive search* di atas dapat dilakukan perbaikan mengenai banyaknya pencarian rute, yakni dengan diketahui bahwa setengah dari rute perjalanan adalah hasil pencerminan dari setengah rute yang lain, yaitu dengan mengubah arah rute perjalanan, rute tersebut adalah :

1. L1 dan L6
2. L2 dan L4
3. L3 dan L5

Maka dapat dihilangkan setengah dari jumlah permutasi (dari 6 menjadi 3). Ketiga buah sirkuit Hamilton yang dihasilkan adalah seperti gambar 2.3 [6]:



Gambar 2. Sirkuit Hamilton

Untuk graf dengan *n* buah simpul, hanya perlu mengevaluasi sirkuit Hamilton sebanyak (*n* – 1)!/2 buah.

2.4. *Graphics Processing Unit*

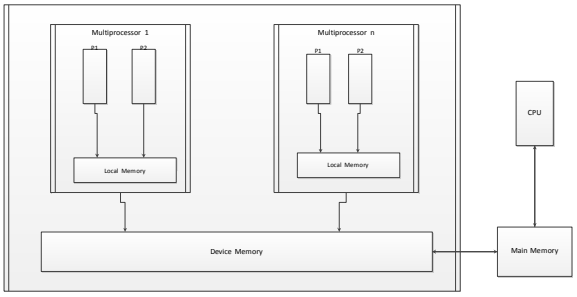
Graphics Processing Unit (GPU) adalah sebuah sirkuit khusus yang dirancang untuk mempercepat *output* gambar dalam *frame buffer* untuk *output* ke layar. GPU sangat efisien dalam memanipulasi grafis computer dan umumnya lebih efektif daripada CPU [1]. GPU bisa berada pada *Video Card* khusus (*VGA Card*) atau terintegrasi dalam *Motherboard* berupa *Integrated* GPU. GPU berfungsi untuk mengolah dan memanipulasi grafis pada CPU (*Central Processing Unit*), untuk nantinya ditampilkan dalam bentuk *Visual Graphics* pada monitor (*output*) [9].

GPU memiliki arsitektur tertentu, hal ini disebabkan karena GPU prosesor *multithread* yang mampu mendukung jutaan pemrosesan data pada satu waktu. Setiap *thread* prosesor terdiri dari beberapa precision FPU (*Fragment Processing Unit*). *Device memory* akan menjadi tempat

pemrosesan data sementara selama proses parallel. Pada pemrosesan data, GPU menggunakan metode *shared memory multiprocessor*. Kelebihan shared memory ini dibandingkan dengan jenis paralel computer yang lain adalah lebih cepat dan efisien karena kecepatan transfer data antar unit komputasi tidak mengalami degradasi.

Secara khusus GPU ditugaskan hanya untuk mengolah tampilan *graphics*. Pada *graphics card add-on*, yang dimaksud dengan GPU adalah *chip graphics*-nya yang kita kenal dengan nama *Geforce*, Radeon dan lainnya. Sedangkan pada solusi *integrated graphics*, GPUnya biasanya tidak berupa *chip* mandiri karena sudah diintegrasikan ke dalam *chipset motherboard*. Istilah GPU sendiri dipopulerkan oleh *chip graph*ics buatan NVIDIA

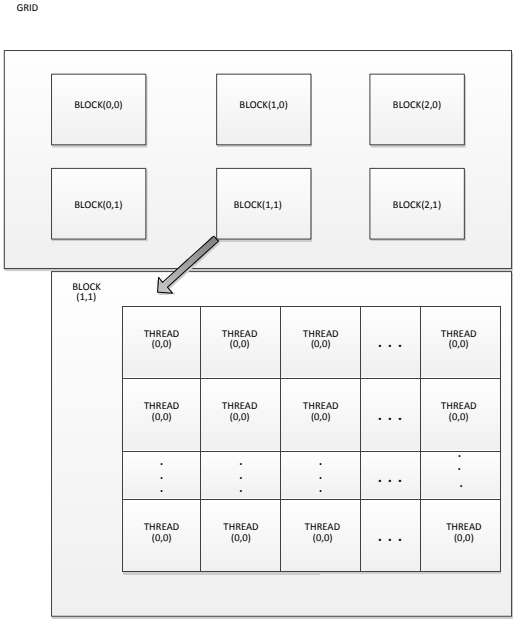
yaitu *Geforce* 256. Hal ini tidak lain karena trik *marketing* yang dilakukan NVIDIA. Pada saat peluncurannya tahun 1999, *Geforce* 256 dikenalkan pada dunia sebagai "*The World's First GPU*". Klaim ini sendiri tidak sepenuhnya benar apabila kita mengkategorikan GPU adalah semua *chip* yang bertugas untuk mengolah tampilan *graphics*. Ada banyak *chip* untuk *graphics* yang pernah hadir sebelum *Geforce* 256 lahir. Namun pada saat itu *chip* tersebut dikenal dengan sebutan *video card*, 2D *accelerator* atau 3D *accelerator* [8]. Sederhananya, GPU merupakan otak dari *graphics card*.



Gambar 3. Arsitektur GPU

2.5. *General Purpose Parallel Computing*

Selain untuk mengolah tampilan *graphics*, GPU dapat digunakan untuk komputasi, khususnya secara paralel. Contoh aplikasi perhitungan komputasi GPU seperti simulasi ilmiah, pemodelan statistika, *image processing*, *life sciences* dan *medical imaging* [15].Salah satu pelopor dari GPU *computing* adalah NVIDIA, dimana NVIDIA memperkenalkan *general purpose parallel computing platform*, yaitu *Computer Unified Device Architecture* (CUDA). CUDA merupakan suatu pembaharuan model *parallel programing* yang mengatur dan mendistibusikan *threads* pada GPU[7].



Gambar 4. *Illustration of a Grid of Thread Blocks*

General-Purpose Parallel Computing pada GPU (GPGPU) adalah sebuah teknik dengan menggunakan GPU untuk melakukan pengolahan data pada aplkasi non-grafis [4]. Cara kerja GPGPU adalah menyalin data pemrosesan dari memori utama ke memori GPU, kemudian GPU akan di beri instruksi oleh CPU untuk memproses data secara parallel, dan langkah terakhir adalah menyalin data hasil pemrosesan dari memori GPU ke memori utama [7].

2.6. CUDA

Computer Unified Device Architecture (CUDA) adalah sebuah teknologi berupa platform komputasi parallel dan model pemrograman yang dikembangkan oleh NVIDIA untuk mempermudah utilitasi GPU untuk keperluan umum (non-grafis) [10]. Hal ini memungkinkan peningkatan dramatis dalam kinerja komputasi dengan memanfaatkan kekuatan dari *graphics processing unit*. CUDA memberikan kemampuan dua *hardware* untuk melakukan komputasi secara nersamaan dan bersatu yaitu pada bagian CPU dan GPU. Hal inilah yang menyebabkan kemampuan komputer menjadi luar biasa. Sebagai contoh sebuah proses

perhitungan yang dilakukan oleh CPU bisa diselesaikan dalam beberapa detik tapi dengan bantuan GPU hal ini dipersingkat hanya menjadi *millisecond* dan CUDA yang menjembatani proses tersebut [4]. Arsitektur CUDA memungkinkan GPU menjadi arsitektur terbuka seperti layaknya *Central Processing Unit* (CPU). Hanya tidak seperti CPU, GPU memiliki banyak inti arsitektur yang parallel. Setiap inti memiliki kemampuan untuk menjalankan ribuan *thread* secara simultan. Apabila aplikasi yang dijalankan sesuai dengan arsitektur tersebut, GPU dapat menyediakan keuntungan yang lebih besar dari segi performa proses aplikasi tersebut [9].

Dewasa ini, para developer dapat memanfaatkan kemampuan *processing* GPU untuk mengakselerasi komputasi dari program mereka dengan jauh lebih mudah. Namun tidak tertutup kemungkinan untuk pengguna biasa dapat menikmati manfaat dari teknologi CUDA. Saat ini sudah mulai banyak yang mendukung akeselerasi dengan CUDA. Misalnya MATLAB dan beberapa plugin dari Adobe Photoshop CS. Bila perangkat lunak ini mendeteksi adanya *hardware* yang kompatibel dengan CUDA, maka beberapa proses komputasinya akan dilaksanakan oleh GPU dan program akan dapat dieksekusi dengan lebih cepat. CUDA juga memungkinkan *programmer* untuk mengeksekusi program pada GPU dimana program tersebut dapat dipecah menjadi banyak eksekusi parallel [3].

Pemrograman CUDA sama seperti membuat program C biasa. Saat kompilasi, *syntax* C biasa akan diproses oleh *compiler* C, sedangkan *syntax* dengan *keyword* CUDA akan diproses oleh *compiler* CUDA (nvcc) [11].

Untuk dapat bekerja dengan teknologi CUDA ada tiga komponen yang harus tersedia pada perangkat PC atau notebook. Komponen tersebut antara lain [14]:

1. CUDA *driver*, merupakan *driver* yang harus sudah terinstal pada *device* yang akan digunakan untuk menjalankan program CUDA.

   Device tersebut harus menggunakan kartu grafis dari NVIDIA.
2. CUDA *toolkit*, merupakan ruang lingkup pengembangan CUDA dari bahasa C sehingga akan menghasilkan CUDA-*enabled* GPU.

CUDA SDK, berisi contoh-contoh program CUDA dan *header* yang dapat dijalankan oleh program CUDA.

2.7. *ThreadProcess*

ThreadProcess atau *t*(*p*) menyatakan berapa banyak proses dalam sebuah *thread*. Pada awalnya penggunaan *thread* pada GPU hanyalah satu *thread* untuk satu tugas, apabila *threadProcess* diinsialisasi menjadi 2, maka setiap *thread* akan melakukan tugas sebanyak dua kali ketika jumla *thread* yang ada diinisialisasikan telah masing-masing menyelesaikan tugasnya, ketimbang meminta GPU untuk menginisialisasi *thread* tambahan.

2.8. Uji t

Dalam menyangkut uji dua rataan keadaan yang lebih umum berlaku ialah keadaan dengan variansi tidak diketahui. Bila si peneliti bersedia menganggap bahwa kedua distribusi normal dan bahwa $a_1 = a_2 = a$, maka uji t gabungan (sering disebut uji-t dua-sampel) dapat digunakan. Uji statistik tersebut berbentuk:

$$t = \frac{(\overline{x_1} - \overline{x_2}) - d_0}{s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \qquad (1)$$

Untuk

$$S_p = \frac{S_1^2(n_1 - 1) + S_1^2(n_2 - 1)}{n_1 + n_2 - 2} \qquad (2)$$

Distribusi -t digunakan disini bila hipotesisnya ekasisi maka hipotesis tidak ditolak bila, seperti dugaan tandingan ekasisi menimbulkan daerah kritis ekasisi. Sebagai contoh:

$H_1 : \mu_1 - \mu_2 > d_0$ tolak $H_1 : \mu_1 - \mu_2 = d_0$
bila $t > t_{a,n_1+n_1^{-2}}$ (3)

Untuk α (Alpha) secara pragmatis didefinisikan sebagai resiko salah dalam penarikan kesimpulan penelitian. Dengan memperkecil α, berarti memperkecil resiko salah dalam penarikan kesimpulan. Hal ini dapat ditempuh dengan jalan memperkecil kesalahan (ketidakpastian) dalam setiap langkah rangkaian analisis data inferensial.

Untuk mernahami lebih lanjut mengenai α dapat diperhatikan ilustrasi di bawah ini [16].Berikut prosedur yang dilakukan dalam mengunakan *uji-t*:

1. Menentukan formulasi hipotesis

   Contoh:
   $H_0 : \mu_1 - \mu_2 = 0$
   $H_1 : \mu_1 - \mu_2 > 0$
2. Menentukan taraf nyata (α)

   $\alpha = P(H_0 \text{ ditolak} / H_0 \text{ benar})$
3. Mencari derajat kebebasan (*dv*)

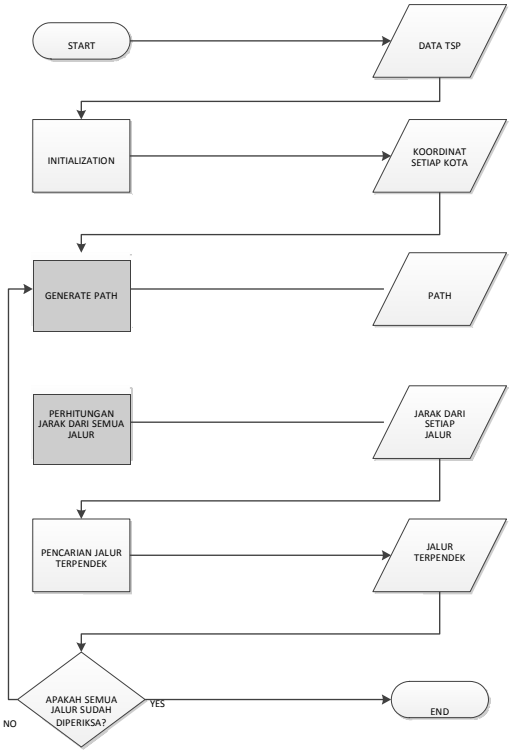
   $n_1 + n_2 - 2 = d_v$
4. Mencari daerah kritis (*t*) pada tabel nilai kritis distribusi t
5. Melakukan perhitungan *t*
6. Menentukan kriteria keputusan

   Contoh:
   $H_1 : \mu_1 - \mu_2 > d_0$ tolak $H_1 : \mu_1 - \mu_2 = d_0$
   bila $t > t_{a,n_1+n_1^{-2}}$
7. Membuat kesimpulan

   Menyimpulkan H_0 diterima atau ditolak.

3. Perancangan Sistem

Dalam mencari jalur TSP terpendek menggunakan algoritma *Brute Force*, dibutuhkan jarak antar kota. Data yang digunakan didapat dari men-*generate* secara random. Jalur – jalur TSP akan di cari secara *Brute force* sehingga semua kemungkinan jalur dapat dimunculkan.



Gambar 5. Gambaran Umum Penyelesaian TSP Menggunakan *Brute Force*

Pada awalnya sistem akan membaca data , menerima koordinat dari kota-kota yang ada. Dari koordinat yang ada, ditentukan jarak antar kota yang akan digunakan untuk menghitung jarak yang di tempuh jalur. Selanjutnya akan dilakukan pembagian jalur yang akan di proses, dimana jalur-jalur yang akan di perhitungkan akan di munculkan pada proses *Generate Path* dalam GPU dengan menggunakan *Library* "*nextPermutation"* pada platform C++. Berbeda dengan cara seri, proses men-*generate path* dengan cara paralel menggunakan GPU, dimana setiap *thread* melakukan tugas *generate path* secara bersamaan.

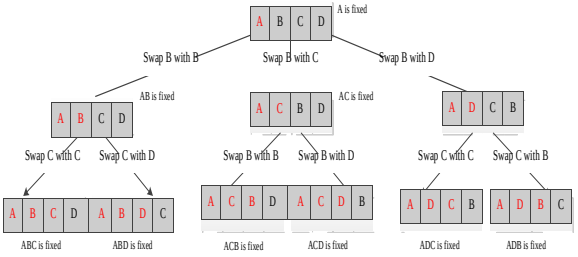
Setelah seluruh jalur dimunculkan, sistem akan membandingkan jarak total yang ditempuh antar jalur , sampai didapat jalur dengan bobot jarak terkecil, pada proses Pencarian Jalur Terpendek dalam sistem.

3.1.1 *Initialization*

Pada tahap ini dilakukan proses *generate* titik secara random, penentuan jumlah maksimum *thread* dan *block* pada GPU, dan juga penentuan jumlah maksimum jalur yang nantinya akan digunakan pada proses selanjutnya.

3.1.2 *Generate Path*

Proses ini bertujuan untuk memunculkan jalur dari data TSP sebanyak maksimum jalur yang telah di inisialisasi. Jumlah jalur yang dapat dimunculkan dapat berubah apabila $mod\left(\frac{(n-1)!}{Max.Jalur}\right) \neq 0$, sehingga jumlah jalur pada *t(p)* terakhir menjadi $m = mod\left(\frac{(n-1)!}{Max.Jalur}\right)$. Kemudian, rumus untuk mencari jumlah jalur yang ada adalah : $n = (n-1)!$



Gambar 6. Ilustrasi proses *Generate Path*

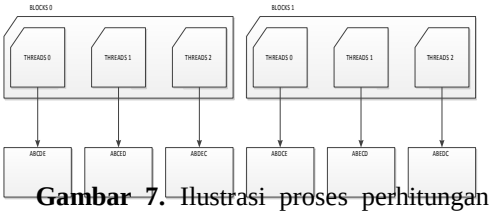
Untuk men-generate jalur-jalur yang ada , dilakukan dengan menggunakan *Library* "*nextPermutation"* dalam platform C++. *Library* ini akan menghasilkan urutan-urutan jalur yang merupakan permutasi dari jumlah kota(*n*) yang ada pada data. Cara kerja *Library* ini adalah dengan membentuk jalur berdasarkan kota-kota yang ada. Pertama-tama kota akan diurutkan , kemudian men-*swap* kota(node) satu dengan yang lain sehingga menjadi jalur baru , begitu seterusnya hingga seluruh jalur yang memungkinkan ditemukan. Kemudian dari urutan-urutan kota baru yang dihasilkan akan di hitung jumlah jarak yang di tempuh dari setiap urutan tersebut.

3.1.3 Perhitungan Jarak Dari Semua Jalur

a.Secara Paralel

Proses pencarian jalur terpendek secara paralel dilakukan secara bertahap dan terbagi-bagi untuk mengurangi pemakaian memori RAM. Berapa banyak jumlah pembagian yang akan dilakukan untuk mencari jalur terpendek dari seluruh *path* yang sudah di-*generate* nantinya berdasarkan jumlah *thread*,

block dan *t(p)* setiap *thread* pada GPU. Selanjutnya perhitungan jarak tempuh total dari setiap jalur yang sudah di generate ke dalam setiap *thread* pada masing-masing block. Perancangan paralel dilakukan dengan membagi-bagi perhitungan jarak tempuh total.



Gambar 7. Ilustrasi proses perhitungan jarak dari setiap jalur dalam *thread* dan *block*.

Perhitungan jarak antar kota pada setiap jalur dihitung dengan menggunakan rumus jarak antar dua titik sebagai berikut :

Jarak = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Kemudian, dihitung jarak tempuh total pada setiap jalurnya.

b. Secara Seri

Perhitungan jarak dari semua jalur secara seri tidak berbeda jauh dibandingkan dengan perhitungan secara paralel, yang membedakan perhitungan secara seri adalah tidak membagi perhitungan jarak setiap jalurnya ke dalam *thread* dan *block* atau dapat diartikan bahwa perhitungan jarak setiap jalurnya dilakukan satu per-satu dalam CPU.

3.1.4 Pencarian Jalur Terpendek

Tahap ini bertujuan untuk mencari jalur dengan jarak terpendek dari seluruh jalur yang telah di-*generate* pada setiap *t(p)*, kemudian akan menghasilkan data jalur(*path*) dengan jarak terpendek, beserta jarak jalur tersebut.

4. Analisis Hasil Pengujian

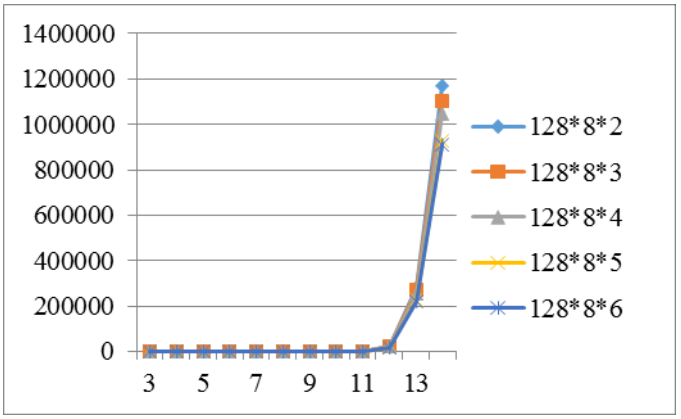
4.1. Konfigurasi *Hardware*

Perangkat keras yang digunakan dalam tugas akhir ini adalah sebagai berikut :

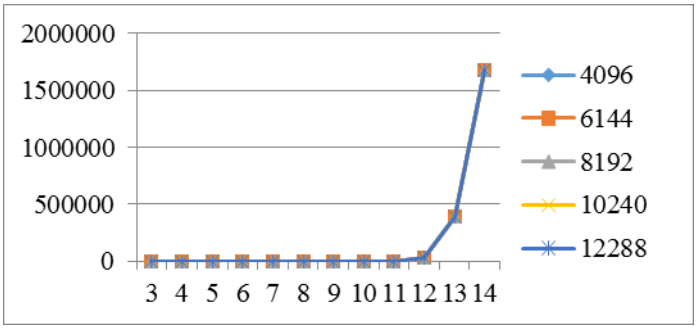
- Processor : AMD Phenom XII
- GPU : NVIDIA GTX 750 2GB
- RAM : 8GB

4.2. Skenario Pertama

Skenario ini membandingkan performa GPU dan CPU dalam kecepatan mencari jalur dengan jarak tempuh terpendek, dan menganalisis pengaruh variable-variabel yang ada. Percobaan dengan GPU dilakukan dengan menginisialisasi jumlah *thread* dan *block* yang sama, yaitu 8 buah *thread* dan 128 *block*, pada 12 buah *maxCity* yang berbeda (*maxCity* : 3 sampai *maxCity* : 14), namun dengan jumlah *t(p)* yang berbeda-beda. Untuk percobaan dengan menggunakan CPU, MAX_PATH disesuaikan dengan jumlah perkalian antara *thread* x *block* x *t(p)*, agar setara.



Gambar 8. Perubahan Waktu(*ms*) terhadap *maxCity* dengan *thread*=8, *block=128* dan *t(p)*=2, 3, 4, 5 dan 6 pada GPU



Gambar 9. Perubahan Waktu(*ms*) terhadap *maxCity* dengan MAX_PATH= 4096, 6144, 8192, 10240 dan 12288

Dari hasil percobaan di atas, terlihat GPU mengungguli CPU pada saat terjadi perubahan *t(p)* menjadi 2, dan begitu juga selanjutnya, ketika *t(p)>1*, performa GPU selalu mengungguli CPU. Namun, jika *maxCity* yang diuji berjumlah sedikit, tetap saja penggunaan GPU menjadi tidak efisien dan CPU tetap unggul dalam hal tersebut. Dari hasil percobaan yang telah dilakukan, GPU memerlukan *maxCity*=11 untuk menunjukkan keunggulannya dibandingkan CPU.

Tabel 1. Persentase rata-rata penurunan waktu pada setiap *t(p)*

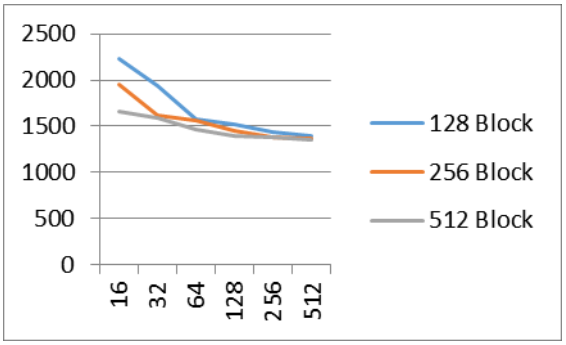
t(p) / Kota	t(p) 1-2	t(p)2-3	t(p) 3-4	t(p) 4-5	t(p)5-6
11	36.87%	1.74%	10.94%	8.30%	3.81%
12	35.61%	4.50%	7.59%	11.09%	4.66%
13	41.35%	3.91%	7.50%	10.02%	3.65%
14	38.39%	5.64%	4.98%	11.26%	1.75%
Rata - rata	38.06%	3.95%	7.75%	10.17%	3.47%

Dapat dilihat dari Tabel 1 bahwa rata-rata penurunan waktu pada setiap *t(p)* yang paling tinggi terdapat pada *t(p)*1-2 sebesar 41.35%, dan rata-rata penurunan waktu pemrosesan pada *t(p)*1-2 dengan kota 11 sampai 14 adalah 38.06%.

Jadi dapat disimpulkan bahwa semakin banyak *t(p)* maka kecepatan komputasi GPU akan menuju satu titik, dengan kata lain disebut konvergen. Hal ini disebabkan oleh karena kecepatan komputasi pada device GPU telah mencapai titik maksimal. Dalam permasalahan ini, perbedaan waktu *t(p)*=1 dan *t(p)*=2 sangat signifikan, hal ini dikarenakan device GPU tersebut tidak dipakai secara maksimal.

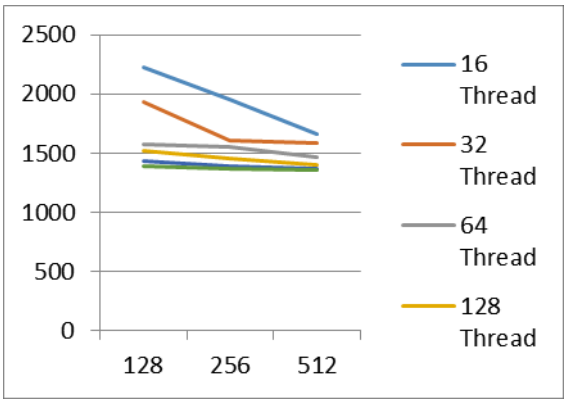
4.3. Skenario Kedua

Dari skenario pertama dapat dilihat bahwa *t(p)* yang mempengaruhi kecepatan GPU secara signifikan adalah *t(p)*=2 dan pada *maxCity*=11, sehingga dalam skenario kedua ini akan dilakukan pengujian terhadap kedua variable tersebut terhadap *thread* dan *block*. Tujuan utama skenario kedua ini adalah untuk mengetahui pengaruh *thread* dan *block* dengan menggunakan *threadProcess* terhadap waktu pemrosesan, berikut ini adalah salah satu hasil pengujian skenario kedua.



Gambar 10. Pengaruh perubahan *thread* terhadap waktu

Berdasarkan Gambar 4.7 jumlah block sebesar 128 akan cenderung turun apabila jumlah *thread* bertambah besar, begitu pula dengan 256 block dan 512 block. Jadi perubahan *thread* dari 16 hingga 512 akan cenderung turun, dan konvergen atau menuju ke satu nilai.



Gambar 11. Pengaruh perubahan *block* terhadap waktu

Terlihat pada Gambar 4.8, ketika menggunakan 16, 32, dan 64 *Thread*, waktu akan cenderung menurun seiring dengan bertambahnya jumlah block. Sedangkan ketika menggunakan 128, 256 dan 512 *Thread*, penurunan waktu tidak se-signifikan bila dibandingkan dengan menggunakan 16, 32 atau 64 *Thread*. Berdasarkan Gambar 4.8, pengaruh *block* dapat disimpulkan, bahwa semakin banyak jumlah *Thread* yang diinisialisasi, maka jumlah *Block* tidak terlalu berpengaruh terhadap penurunan waktu pemrosesan. Jadi, penggunaan *Thread* dan *Block* dengan jumlah yang banyak tidak menjanjikan waktu pemrosesan akan selalu turun, karena waktu pemrosesan justru akan konvergen. Kemudian untuk meneliti lebih lanjut, dilakukan uji statistik dengan menggunakan Uji *t*. Berikut beberapa langkah yang dilakukan yaitu:

1. Menentukan formulasi hipotesis
   a. H_o (Hipotesis nol) menentukan pernyataan tidak adanya perbedaan, hubungan atau pengaruh antara pengujian suatu populasi atau sampel disini berupa tidak ada perbedaan yang signifikan dari rata-rata waktu percobaan masing-masing *Block* pada GPU.
      $H_o : \mu_1 = \mu_2$ atau $\mu_1 - \mu_2 = 0$
      μ_1 = rata-rata waktu menggunakan *block* percobaan pertama pada GPU.
      μ_2 = rata-rata waktu menggunakan *block* percobaan kedua pada GPU.
   b. H_1 (Hipotesis tandingan) menyatakan lawan dari hipotesis nol bahwa adanya suatu perbedaaan yang signifikan dari rata-rata waktu percobaan masing-masing antara pengujian suatu populasi atau sampel *Block* pada GPU.
      $H_1 : \mu_1 - \mu_2 > 0$
      μ_1 = rata-rata waktu menggunakan *block* percobaan pertama pada GPU.
      μ_2 = rata-rata waktu menggunakan *block* percobaan kedua pada GPU.
2. Menentukan taraf nyata (α)
   $\alpha = P$ (H_0 ditolak atau H_0 diterima)
   α = 0.01 dan 0.05

Alpha didefinisikan sebagai resiko salah dalam penarikan kesimpulan penelitian. Pada pengujian ini diambil alpha sebesar 1% atau 0.01 dan 5% atau 0.05

3. Melihat *t* tabel
   t tabel ditentukan dengan derajat kebebasan (v)
   $dv = n_1+n_2-2$
   $dv = 10 + 10 - 2$
   $dv = 18$
4. Mencari daerah kritis (*t*) pada tabel nilai kritis distibusi *t*
   t = 2.55237963 dengan α = 1%
   t = 1.734063607 dengan α = 5%
5. Melakukan perhitungan *t*
6. Menentukan kriteria pengujian
   $H_1 : \mu_1 - \mu_2 > 0$
   Tolak H_o : $\mu_1 - \mu_2 = 0$, bila $t > t_{\alpha,n_1+n_2-2}$
7. Membuat kesimpulan
   Menyimpulkan H_0 diterima atau tidak dengan membandingkan antara langkah keempat dengan kriteria pengujian pada langkah keenam.

Tabel 2. Hasil Uji T

Percobaan 1	**Percobaan 2**	**Hasil Uji** *t*	Keputusan α = 1%	Keputusan α = 5%
16*T*, 128*B*	16*T*, 256*B*	14.812	Tolak H_0	Tolak H_0
16*T*, 128*B*	16*T*, 512*B*	46.431	Tolak H_0	Tolak H_0
16*T*, 256*B*	16*T*, 512*B*	26.886	Tolak H_0	Tolak H_0
32*T*, 128*B*	32*T*, 256*B*	22.184	Tolak H_0	Tolak H_0
32*T*, 128*B*	32*T*, 512*B*	24.412	Tolak H_0	Tolak H_0
32*T*, 256*B*	32*T*, 512*B*	6.220	Tolak H^0	Tolak H^0
64*T*, 128*B*	64*T*, 256*B*	2.439	Terima H^0	Tolak H^0
64*T*, 128*B*	64*T*, 512*B*	16.326	Tolak H_0	Tolak H_0
64*T*, 256*B*	64*T*, 512*B*	16.927	Tolak H_0	Tolak H_0
128*T*, 128*B*	128*T*, 256*B*	7.804	Tolak H_0	Tolak H_0
128*T*, 128*B*	128*T*, 512*B*	14.565	Tolak H_0	Tolak H_0
128*T*, 256*B*	128*T*, 512*B*	10.951	Tolak H_0	Tolak H_0
256*T*, 128*B*	256*T*, 256*B*	8.362	Tolak H_0	Tolak H_0
256*T*, 128*B*	256*T*, 512*B*	7.548	Tolak H^0	Tolak H^0
256*T*, 256*B*	256*T*, 512*B*	3.286	Tolak H_0	Tolak H_0
512*T*, 128*B*	512*T*, 256*B*	5.044	Tolak H_0	Tolak H_0
512*T*, 128*B*	512*T*, 512*B*	7.493	Tolak H_0	Tolak H_0
512*T*, 256*B*	512*T*, 512*B*	3.032	Tolak H_0	Tolak H_0

Kesimpulan yang bisa diambil dari pengujian statistika (uji *t*) tersebut terdapat perbedaan yang signifikan dari rata-rata waktu percobaan masing-masing *Thread* dan *Block* pada GPU, yaitu penambahan *block* dan *thread* yang besar akan berpengaruh secara signifikan, kecuali pada percobaan 128 *Block* dan 256 *Block* dengan 64 *thread* hasil percobaan menunjukkan tidak ada pengaruh yang signifikan.

5. Kesimpulan

Berdasarkan percobaan dan analisis yang telah dibahas pada penelitian Tugas Akhir ini, diperoleh beberapa kesimpulan sebagai berikut:

GPU akan mengungguli CPU pada saat menggunakan *threadProcess*>1 dan menggunakan *maxCity*≥11. Apabila menggunakan *threadProcess*=1 dan *maxCity*<11, performa CPU akan tetap lebih baik dibandingkan GPU.

Perubahan *t(p)* yang semakin besar akan mengakibatkan persentase penurunan waktu semakin besar pula, seiring dengan kecepatan komputasi GPU.

Semakin banyak *t(p)* maka kecepatan komputasi GPU akan menuju satu titik, dengan kata lain disebut konvergen. Hal ini disebabkan oleh karena kecepatan komputasi pada device GPU telah mencapai titik maksimal.

Penambahan *block* dan *thread* yang besar akan berpengaruh secara signifikan, kecuali pada percobaan 128 *Block* dan 256 *Block* dengan 64 *thread* hasil percobaan menunjukkan tidak ada pengaruh yang signifikan.

6. Daftar Pustaka

[1] Anthony Lippert – "*NVidia GPU Architecture for General Purpose Computing*"

[2] Cook, a professor at the Georgia Institute of Technology, is author of *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation* (Princeton University Press, 2012).

[3] David Kirk/NVidia and Wen-mei Hwu, 2006-2008 – "*CUDA Threads*"

[4] Ghorpade, Jayshre., Parande, Jitendra., Kulkarni, Madhura. *GPGPU Processing in CUDA Architecture*. Advanced Computing: An International Journal (ACIJ), Vol.3, 2012

[5] Munir, Rinaldi. 2005. *Matematika Diskrit*. Bandung. Penerbit Informatika

[6] Munir, Rinaldi. Diktat Kuliah IF2251.*Strategi Algoritmik*
[7] NVIDA Corporation, 2010. *CUDA Programming Guide 3.0*
[8] https://www-304.ibm.com/ diakses 1 Januari 2013
[9] https://developer.nvidia.com/ diakses 1 Januari 2013
[10] www.nvidia.com/cuda/ diakses diakses 1 Januari 2013
[11] http://nvidiapartnerbp.blogspot.com/ diakses 1 Januari 2013
[12] www.travellingsalesmanproblem.com/ diakses 1 Januari 2013
[13] http://mathworld.wolfram.com/HamiltonianCycle.html diakses 27 Juni 2013
[14] J. Sanders, E. Kandrot, CUDA by example, Addison Wesley, 2011.
[15] W.W. Hwu, GPU Computing Gem Emerald Edition, Morgan Kaufman, 2011.
[16] Walpole, Ronald E. 1995. Ilmu Peluang dan Statistika untuk Insinyur dan Ilmuwan. Bandung: Penerbit ITB.
[17] Riduwan. 2010. Dasar-dasar Statistika. Bandung: Alfabeta.
[18] Walpole, Ronald E. 1995. Pengantar Statistika. Jakarta: Penerbit PT Gramedia Pustaka Utama, 1990.
[19] Tim Penelitian dan Pengembangan Komputer. 2001. Pengolahan Data Statistik dengan SPSS. PT Salemba Infotek, Jakarta.
[20] Munir, Rinaldi, 2006; Metode Numerik; INFORMATIKA Bandung.