Library Algoritma Genetik dan Whale Optimization berbasis GPU Programming

Bernard Niklas Satrijo[1], Hendrawan Armanto[1], Lukman Zaman PCSW[1], and C. Pickerling[1]
[1]Departemen Teknik Informatika, Fakultas Sains dan Teknologi, Institut Sains dan Teknologi Terpadu Surabaya, Surabaya, Indonesia

Corresponding author: Bernard Niklas Satrijo (e-mail: satrijoniklas@yahoo.com).

ABSTRACT A method to search for the best solution to a problem continues to be developed from time to time. Various studies have been carried out in various fields to solve these problems. One of the methods available is evolutionary algorithms. One of the well-known evolutionary algorithms is the genetic algorithm (GA). In addition to the genetic algorithm, another new algorithm has emerged and developed, namely the whale optimization (WOA) algorithm. However, the completion of evolutionary algorithms generally takes a long time. GPU Programming can be used in the evolutionary algorithms created to overcome this. The nature of the library created is a general-purpose library. In this library, the user only needs to create one derived class from the class, set parameters, and implement abstract methods. All processes that occur in the library are transparent. When completed, the user can retrieve the best fitness value, the best solution, the best fitness value per generation, and the average fitness value per generation. With the creation of this library, it is expected that users can implement genetic algorithms and whale optimization in solving a problem efficiently and in a faster time than other libraries.

KEYWORDS Evolutionary Algorithm, Genetic Algorithm, Whale Optimization Algorithm, GPU Programming, Cuda C++, Library.

ABSTRAK Pencarian solusi terbaik pada suatu permasalahan menjadi suatu hal yang terus dikembangkan dari waktu ke waktu. Berbagai penelitian dilakukan dalam berbagai bidang untuk dapat menyelesaikan masalah-masalah tersebut. Salah satu cara yang dapat menjadi solusi adalah digunakannya algoritma evolusioner. Salah satu algoritma evolusioner yang terkenal adalah algoritma genetik (GA). Selain algoritma genetik, terdapat algoritma baru lain yang telah muncul dan dikembangkan, yaitu algoritma whale optimization (WOA). Meskipun begitu, penyelesaian algoritma evolusioner pada umumnya membutuhkan waktu yang lama. Hal tersebut dapat ditanggulangi dengan penggunaan GPU Programming pada algoritma evolusioner yang dibuat. Sifat dari library yang dibuat adalah general purpose library. Pada library ini, user hanya perlu membuat satu class turunan dari class dalam library, mengatur parameter, dan mengimplentasikan abstract method. Seluruh proses yang terjadi dalam library bersifat transparan, dan apabila selesai, user dapat mengambil nilai fitness terbaik, solusi terbaik, nilai fitness terbaik per generasi, dan nilai rata-rata fitness per generasi. Dengan dibuatnya library ini, diharapkan user dapat mengimplementasikan algoritma genetik dan whale optimization dalam menyelesaikan suatu permasalahan dengan mudah dan dalam waktu yang lebih cepat dibandingkan dengan library lainnya.

KATA KUNCI Evolutionary Algorithm, Genetic Algorithm, Whale Optimization Algorithm, GPU Programming, Cuda C++, Library.

I. PENDAHULUAN

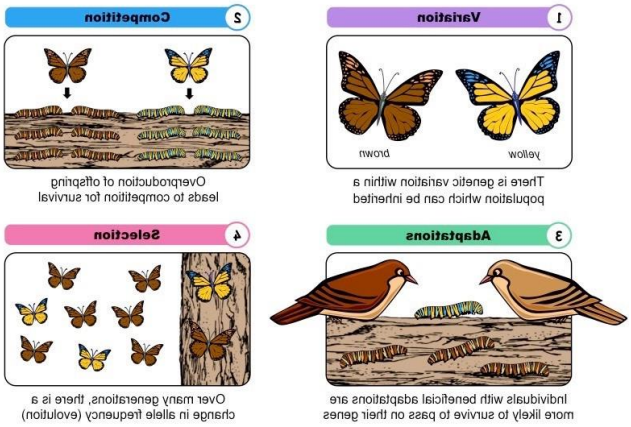
Penggunaan algoritma evolusioner dalam menyelesaikan permasalahan kompleks telah diakui dan dikembangkan oleh berbagai ilmuwan. Namun, performa yang dihasilkan dari program algoritma evolusioner dapat lebih ditingkatkan lagi. Penggunaan iterasi yang sangat banyak dan proses yang tidak jauh berbeda di setiap iterasinya adalah kendala yang menyebabkan performa algoritma evolusioner tidak lebih tinggi dari performa komputasi aritmatik pada umumnya. Salah satu metode yang dapat digunakan untuk meningkatkan

performa algoritma evolusioner adalah penggunaan GPU Programming. GPU Programming memungkinkan adanya peningkatan performa program secara signifikan pada algoritma yang dibuat..

GPU Programming memungkinkan adanya penggunaan GPU dalam menjalankan kode program. Penggunaan GPU Programming ini tidak selalu menjamin bertambahnya kecepatan program dibandingkan pemrograman pada CPU. Pemilihan algoritma dan cara kerja program untuk mencapai suatu tujuan menggunakan thread GPU juga menjadi tantangan yang cukup besar bagi pembuat program untuk dapat mengoptimalisasi kinerja program. Maka dari itu, pada akan dibuat library untuk algoritma genetik (GA) dan Whale Optimization (WOA) untuk menunjang pembuatan algoritma evolusioner yang lebih mudah dengan basis GPGPU (General-Purpose computing on Graphics Processing Units) sehingga performa dari algoritma evolusioner yang dihasilkan dapat lebih meningkat dibandingkan dengan program algoritma evolusioner sejenis dengan penggunaan CPU dalam prosesnya. Dalam perkembangannya, telah terdapat berbagai macam platform yang mendukung prinsip GPU Programming. Platform yang digunakan pada pembuatan library algoritma evolusioner adalah NVIDIA CUDA.

II. ALGORITMA GENETIK [1]–[3]

Algoritma genetik atau yang biasa disebut GA mulai diperkenalkan pada tahun 1975 oleh John Holland di New York, Amerika Serikat. Algoritma genetik adalah pencarian dengan metode metaheuristik yang terinspirasi dari teori Charles Darwin mengenai seleksi alam yang termasuk dalam algoritma evolusioner. Algoritma evolusioner sendiri banyak digunakan untuk menyelesaikan masalah [4], [5].

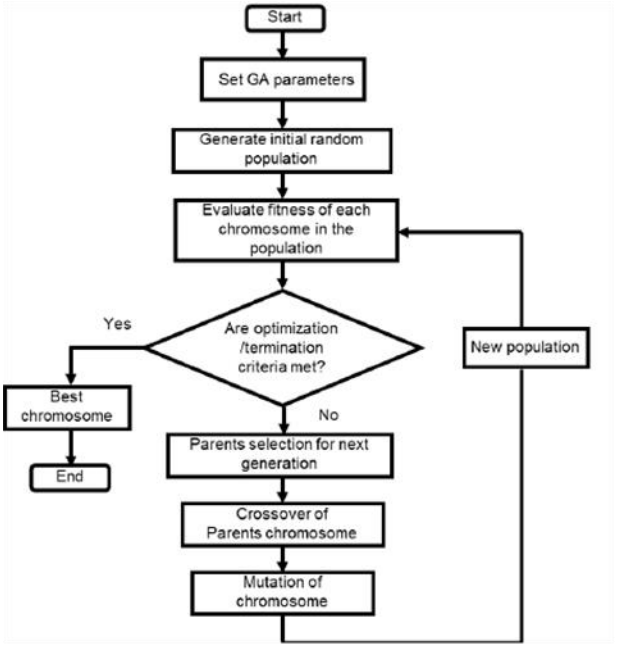


Gambar 1. Seleksi Alam Menurut Prinsip Charles Darwin

Charles Darwin menyebutkan seleksi alam sebagai "Prinsip dimana setiap variasi kecil (dari suatu sifat), jika berguna, dipertahankan". Selain itu, Charles Darwin menyebutkan pula bahwa prinsip adaptasi makhluk hidup didasarkan pada *survival of the fittest*. Individu yang dapat beradaptasi dengan lingkungan mereka, lebih besar kemungkinannya untuk dapat bertahan hidup dan bereproduksi. Selama terdapat variasi di antara setiap individu dan variasi tersebut dapat diwariskan, tidak dapat dihindari bahwa akan terdapat seleksi individu dengan variasi yang paling menguntungkan. Jika variasi tersebut diwariskan, maka keberhasilan reproduksi dari individu dengan variasi yang berbeda akan mengarah kepada evolusi progresif suatu populasi tertentu dari suatu spesies, dan populasi yang berevolusi menjadi cukup berbeda sehingga akhirnya memunculkan spesies yang berbeda pula.

Dalam pengaplikasian seleksi alam pada algoritma genetik, terdapat beberapa proses pada operator application yang membedakan algoritma genetik dengan algoritma evolusioner lainnya. Proses tersebut adalah selection, crossover, dan mutation. Proses inilah yang akan menciptakan individu yang diharapkan dapat memiliki nilai fitness yang semakin baik di setiap generasinya.

Dalam algoritma genetik, suatu solusi dalam populasi (disebut kromosom) untuk masalah optimisasi berevolusi menjadi solusi yang lebih baik. Setiap solusi memiliki seperangkat properti (disebut gen) yang dapat bermutasi dan diubah. Gen ini direpresentasikan dengan suatu jenis data (pada umumnya bit/integer). Proses dasar pada algoritma genetik dapat dilihat pada gambar 2.



Gambar 2. Struktur Dasar Algoritma Genetik

Proses algoritma genetik dimulai dari pengaturan parameter. Parameter yang ada dalam algoritma genetik adalah: banyak generasi/batas fitness (digunakan dalam Stopping Criteria), banyak individu dalam populasi, crossover rate (antara 0-1), mutation rate (antara 0-1), jenis selection, jenis crossover, dan jenis mutation. Kemudian dilanjutkan dengan populasi individu yang dihasilkan secara acak. Tidak seluruh gen dalam individu harus dihasilkan

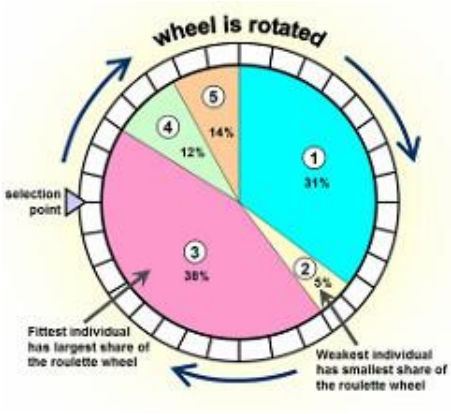
secara random, bergantung pada permasalahan yang akan diselesaikan (hal ini akan lebih dijelaskan pada subbab selanjutnya).

Proses dilanjutkan dengan proses iterasi setiap generasi pada proses selanjutnya sampai Stopping Criteria terpenuhi. Secara umum, Stopping Criteria dari algoritma genetik adalah: jumlah generasi pada parameter telah tercapai, terdapat individu dengan nilai fitness sama dengan atau melebihi batas fitness pada parameter, dan nilai fitness dari individu terbaik dalam beberapa generasi terakhir yang tidak membaik.

Dalam setiap generasi, individu dalam populasi akan dievaluasi. Dari evaluasi tersebut, dihasilkan nilai fitness untuk setiap individu. Kemudian akan dicek apakah algoritma genetik yang dijalankan telah memenuhi Stopping Criteria. Apabila belum terpenuhi, akan dilanjutkan dengan proses selection, yaitu pemilihan solusi yang akan menjadi induk untuk proses selanjutnya. Individu dalam populasi akan dipilih secara stochastic, di mana proses ini bergantung pada metode selection yang dipilih. Proses Selection digunakan untuk pemilihan parent yang digunakan dalam Crossover dan Mutation. Berikut ini adalah beberapa jenis Selection yang diberikan dalam library algoritma genetik ini:

A. Roulette Wheel Selection

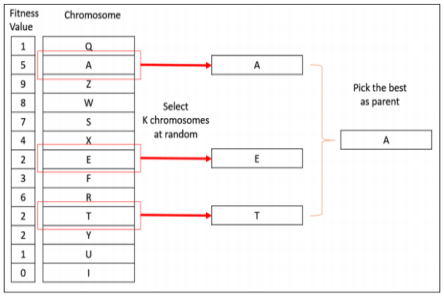
Roulette Wheel Selection menggunakan nilai fitness sebagai acuan untuk memilih individu sebagai parent. Semakin besar fitness yang dimiliki oleh individu, semakin besar kemungkinan individu tersebut dipilih.



Gambar 3. Roulette Wheel Selection

B. Tournament Selection

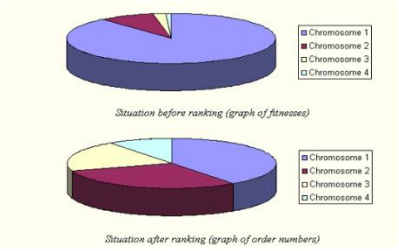
Berbeda dengan Roulette Wheel Selection, pada Tournament Selection akan dirandom sebanyak n individu untuk dapat dipertandingkan. Setelah ditemukan n individu, akan dipilih parent dengan fitness terbaik dari pilihan tersebut



Gambar 4. Tournament Selection

C. Rank Selection

Pada Rank Selection, fitness juga menjadi salah fitur penting dalam menentukan individu yang akan dijadikan parent. Perbedaannya dengan Roulette Wheel Selection adalah pada Rank Selection, setiap individu dengan rank yang lebih tinggi mempunyai suatu proporsi yang tetap berapapun perbedaan fitnessnya.

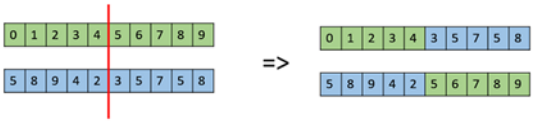


Gambar 5. Rank Selection

Secara umum, individu dengan nilai fitness yang lebih baik berkesempatan lebih besar untuk dapat dipilih dalam proses ini. Individu yang telah dipilih, akan dikawinkan dengan individu lain dalam proses crossover. Sama dengan prinsip perkawinan makhluk hidup, akan dihasilkan individu baru dengan gen yang sesuai dengan gen induknya. Proses crossover [6]–[8] ini juga bergantung pada metode crossover yang dipilih. Berikut ini adalah beberapa jenis Crossover yang diberikan dalam library algoritma genetik ini:

1) One Point Crossover

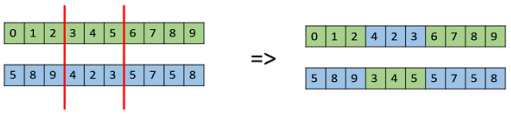
Pada One Point Crossover, akan dirandom sebuah angka dengan maksimal jumlah gennya. Kemudian gen dengan indeks yang lebih kecil atau lebih besar dari angka random tersebut akan ditukar dengan individu parent lain.



Gambar 6. One Point Crossover

2) Multi Point Crossover

Multi Point Crossover adalah modifikasi dari One Point Crossover, di mana akan dirandom satu lagi angka untuk menentukan batas akhir indeks untuk proses crossover.



Gambar 7. Multi Point Crossover

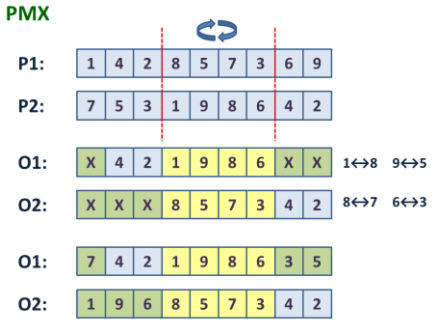
3) Uniform Crossover

Di dalam Uniform Crossover, kromosom tidak dipisahkan berdasarkan segmen tertentu, tetapi setiap gen akan diperlakukan secara terpisah. Pada dasarnya, setiap gen memiliki kesempatan masing-masing 50% untuk mendapatkan gen dari induk 1 atau induk 2.

Gambar 8. Uniform Crossover

4) PMX Crossover

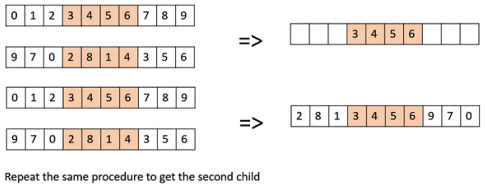
Dalam merepresentasikan solusi menjadi satu kromosom, terkadang terdapat beberapa permasalahan di mana seluruh gen di dalam kromosom harus memiliki nilai yang berbeda. Salah satu contoh permasalahan tersebut adalah permasalahan TSP. PMX Crossover bekerja dengan mempertahankan sebanyak mungkin gen dari induk lain. Pada awalnya, akan dirandom dua buah angka sebagai batas indeks seperti pada Multi Point Crossover. Kemudian, akan dilakukan operasi swap untuk angka yang berada dalam batas indeks ke dalam solusi baru. Lalu akan dilakukan proses mapping untuk memetakan nilai gen yang sudah terdapat di antara kedua batas indeks. Gen yang berada di luar indeks, tetapi terdapat dalam map dari hasil mapping, akan ditukar dengan nilai sesuai pada map, agar menghindari nilai yang kembar.



Gambar 9. PMXCrossover

5) Davis Order Crossover/Order1 Crossover (OX1)

Sama dengan PMX Crossover, Order1 Crossover digunakan untuk permasalahan yang setiap gennya harus memiliki nilai yang berbeda. Proses dimulai dengan menghasilkan 2 angka random sebagai batas indeks dan untuk swapping operation, sama seperti PMX Crossover. Perbedaannya adalah pada Order1 Crossover, gen dalam solusi baru di luar batas indeks tidak akan banyak dipertahankan posisinya. Hal yang paling penting adalah solusi baru yang dihasilkan tidak melanggar constraint di mana solusi mengandung nilai yang sama dalam 2 gen. Dimulai dari gen pada indeks setelah batas indeks akhir, salin seluruh nilai gen yang belum ada dalam solusi baru, secara urut.



Gambar 10. Order1 Crossover

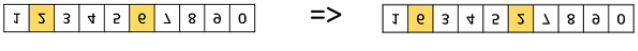
Seluruh individu yang dihasilkan dari proses sebelumnya, berkesempatan untuk mengalami proses mutation [6]. Untuk setiap individu, akan dirandom suatu angka dari 0-1, dan jika angka tersebut memasuki crossover rate, maka mutasi akan terjadi. Proses ini mengubah gen yang ada pada kromosom sesuai dengan metode mutation yang dipilih. Proses ini dilakukan berdasarkan prinsip bahwa kromosom baru dapat mengalami anomali pada beberapa gennya. Akan terdapat nilai mutation rate yang memberikan kemungkinan untuk suatu individu mengalami mutasi. Berikut ini adalah beberapa jenis Mutation yang diberikan dalam library algoritma genetik ini:

1) Random Resetting

Mutasi dengan metode Random Resetting hanya dapat dilakukan untuk individu dengan nilai integer/float. Proses yang dilakukan adalah merandom satu indeks gen pada kromosom, kemudian nilai gen tersebut diubah dengan angka yang dirandom ulang.

2) Swap Mutation

Pada Swap Mutation, akan dirandom dua indeks gen yang kemudian kedua gen ini akan ditukar nilainya. Proses ini hanya dilakukan sekali. Jenis mutasi ini dapat digunakan untuk permasalahan dengan encoding permutasi (seperti TSP).

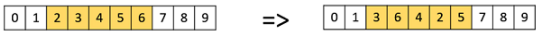


Gambar 11. Swap Mutation

3) Scramble Mutation

Pada Scramble Mutation, akan dirandom dua indeks gen yang menjadi batas indeks. Kemudian setiap gen

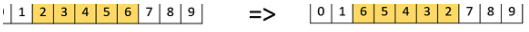
di dalam batas indeks akan diacak kembali posisinya. Jenis mutasi ini juga dapat digunakan untuk permasalahan dengan encoding permutasi.



Gambar 12. Scramble Mutation

4) Inversion Mutation

Pada Inversion Mutation, akan dirandom dua indeks gen yang menjadi batas indeks. Kemudian akan dilakukan invert untuk seluruh gen dalam batas indeks. Jenis mutasi ini juga dapat digunakan untuk permasalahan dengan encoding permutasi.



Gambar 13. Inversion Mutation

Individu baru yang dihasilkan dari seluruh operator ini adalah sejumlah nilai chromosome per generation, sehingga di setiap iterasi, populasi akan bertambah sebanyak dua kali dari jumlah kromosom yang ada pada awalnya. Individu baru yang dihasilkan akan masuk ke dalam populasi dan dievaluasi kembali, menghasilkan proses eliminasi pada ½ dari populasi yang ada, sesuai dengan umur individu atau nilai fitness yang dimilikinya. Proses ini akan terus berulang, sampai Stopping Criteria terpenuhi.

III. ALGORITMA WHALE OPTIMIZATION [9]–[11]

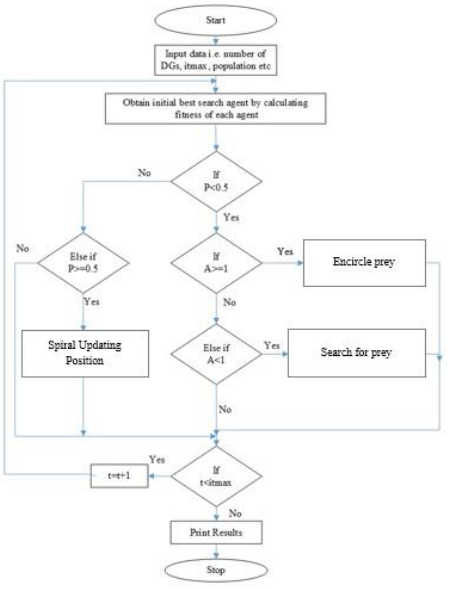
Algoritma Whale Optimization (WOA) merupakan algoritma optimasi yang baru diusulkan pada tahun 2016. WOA memiliki kemampuan untuk dapat menghindari local optima dan mendapatkan solusi global optima. Keunggulan ini menyebabkan WOA menjadi algoritma yang tepat untuk memecahkan masalah optimasi yang berbeda atau tidak terbatas untuk aplikasi praktis tanpa reformasi struktural dalam algoritma. Sejauh ini, belum banyak pengembangan yang dilakukan pada WOA, sehingga hal ini menjadi salah satu alasan akan dipilihnya WOA dalam pembuatan library Algoritma Whale Optimization didasarkan pada kecerdasan paus mencari mangsa secara berkelompok.

Gambar 14. Paus Bungkuk (Humpback Whale) mencari mangsa

Paus merupakan hewan mamalia terbesar dari seluruh mamalia yang ada. Seekor paus dewasa dapat tumbuh hingga panjang 30 meter dan berat 180 ton. Mereka tidak pernah tidur karena mereka harus bernapas dari permukaan lautan. Bahkan, separuh otak dari otak paus hanya berguna untuk tidur. Paus banyak dianggap sebagai predator.

Hal yang paling menarik tentang paus bungkuk adalah metode berburu khusus mereka. Perilaku mencari makan secara berkelompok ini disebut metode bubble-net attacking method. Dari hasil pengamatan, pencarian makan oleh paus bungkuk dilakukan secara berkelompok dengan membuat gelembung khusus di sepanjang jalur lingkaran atau jalur berbentuk angka 9 di sekitar mangsanya. Dua manuver terkait dengan gelembung yang dilakukan oleh paus bungkuk dinamakan upward-spirals dan double-loop. Dalam manuver pertama, sekumpulan paus bungkuk menyelam sekitar 12 meter ke bawah dan kemudian mulai membuat gelembung dalam bentuk spiral di sekitar mangsa dan berenang ke permukaan. Manuver selanjutnya mencakup tiga tahapan yang berbeda, yaitu coral loop, lobtail, dan capture loop. Manuver ini tidak dipergunakan dalam dasar WOA.

Sama halnya dengan algoritma genetik, suatu solusi dalam permasalahan perlu direpresentasikan agar dapat dilakukan proses komputasi. Perbedaannya adalah pada WOA, solusi harus direpresentasikan dengan array of float. Proses dasar pada algoritma Whale Optimization dapat dilihat pada gambar 13.



Gambar 15. Struktur Dasar Algoritma Whale Optimization

Proses algoritma WOA dimulai dengan pengaturan parameter generasi dan agen per generasi. Pada WOA, hanya akan digunakan jumlah generasi sebagai batas looping yang dilakukan. Kemudian dilanjutkan dengan populasi agen pencari mangsa yang dihasilkan secara acak. Tidak seluruh gen dalam individu harus dihasilkan secara random, bergantung pada permasalahan yang akan diselesaikan. Lalu akan dilakukan iterasi setiap generasi pada proses selanjutnya sampai jumlah generasi pada parameter terpenuhi.

Dalam setiap generasi, agen dalam populasi akan dievaluasi. Apabila iterasi belum mencapai jumlah generasi yang telah ditentukan, akan dilanjutkan dengan inisialisasi variabel yang diperlukan, yaitu r1, r2, a, A, C, dan p di mana

nilai r1 dan r2 merupakan angka random antara {0,1}, nilai a secara linear menurun dari 2 sampai 0 sepanjang iterasi, dan nilai p merupakan angka random dari {0,1}. Berikut adalah rumus untuk variabel A dan C:

$$A = 2a * r1 - a \quad \text{.....(1)}$$
$$C = 2 * r2 \quad \text{.....(2)}$$

Kemudian akan dilakukan operasi pada WOA yang terdiri dari Search for Prey, Encircling Prey, dan Spiral Updating Position. Pada operasi Search for Prey, setiap agen akan mencari posisi mangsa secara acak. Operasi ini akan dilakukan apabila nilai P < 0.5 dan nilai |A| >= 1. Mekanisme ini 1 ini menekankan eksplorasi dan memungkinkan algoritma WOA untuk melakukan pencarian global. Model matematisnya adalah sebagai berikut:

$$D = \left| C \cdot X_{rand}(t) - X(t) \right| \quad \text{.....(3)}$$
$$X(t+1) = X_{rand}(t) - A \cdot D \quad \text{.....(4)}$$

Variabel Xrand merupakan posisi agen yang dipilih secara acak. Perhitungan variabel A dan C dapat dilihat pada rumus (1) dan (2). Diharapkan dengan operasi ini, solusi yang ditemukan tidak terjebak pada local optima.

Pada operasi Encircling Prey, setiap agen akan mendekati posisi mangsa dan mengelilinginya. Sejak solusi optimal dalam permasalahan belum diketahui, maka WOA akan mengasumsikan solusi terbaik pada generasi tersebut sebagai mangsa, atau posisi yang mendekati mangsa. Setelah agen terbaik didefinisikan, agen lainnya kemudian akan mencoba untuk memperbarui posisi untuk mendekati agen terbaik. Tahap ini akan dilakukan apabila nilai P < 0.5 dan nilai |A|< 1. Hal ini diwakili oleh persamaan berikut:

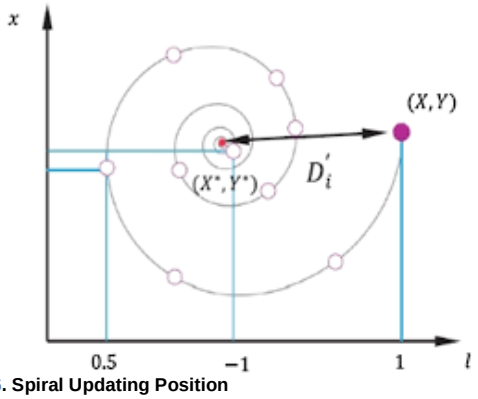
$$D = \left| C \cdot X_p(t) - X(t) \right| \quad \text{.....(5)}$$
$$X(t+1) = X_p(t) - A \cdot D \quad \text{.....(6)}$$

Variabel t menunjukkan iterasi saat ini, A dan C adalah variabel koefisien yang telah dihitung pada proses sebelumnya, Xp adalah posisi mangsa (dalam kata lain agen dengan fitness terbaik), dan X adalah posisi agen pencari mangsa. Diharapkan dengan operasi ini, agen pencarian dapat semakin mendekati solusi optimal.

Pada operasi terakhir yaitu Spiral Updating Position, setiap agen akan mendekati mangsa (solusi terbaik dianggap sebagai mangsa) dengan pergerakan berbentuk helix/spiral. Pergerakan ini dilakukan dengan melihat posisi agen sendiri dan posisi mangsa. Tahap ini akan dilakukan apabila nilai P >= 0.5. Operasi ini dimulai dengan perhitungan jarak antara agen pencari mangsa dan posisi mangsa (agen terbaik). Persamaan spiral kemudian dibuat dengan menggunakan posisi paus dan mangsa untuk menirukan gerakan paus bungkuk berbentuk heliks sesuai rumus berikut:

$$D' = \left| X_p(t) - X(t) \right| \quad \text{.....(2.7)}$$
$$X(t+1) = D' e^{bt} \cos(2\pi t) + X_p(t) \quad \text{.....(2.8)}$$

Variabel D digunakan untuk mengindikasikan jarak antara agen pencari mangsa dan mangsa, b merupakan konstanta untuk mendefinisikan bentuk spiral logaritmik (umumnya bernilai 1), dan t merupakan angka random dari {-1,1}.



Gambar 16. Spiral Updating Position

IV. CUDA [12]–[14]

GPU pada masa modern ini memberikan kekuatan pemrosesan yang sangat besar, bandwidth memori, dan efisiensi jauh di atas CPU. Telah dibuktikan bahwa kecepatan GPU 50-100 kali lebih cepat dalam tugas yang membutuhkan beberapa proses paralel, seperti pada bidang machine learning dan analisis big data. Akan tetapi, tidak seluruh permasalahan komputasi dapat diselesaikan dengan menggunakan GPU. Pada dasarnya CPU memiliki kecepatan yang lebih tinggi apabila hal yang dikerjakan berupa single task. Selama bertahun-tahun, GPU telah memberikan kemampuan render image dan pergerakan (motion) pada layar komputer, tetapi secara teknis, GPU mampu melakukan lebih banyak hal. Beberapa contoh tugas yang lebih unggul dikerjakan dalam GPU dibandingkan CPU adalah pada bidang *games*, visualisasi 3D, dan pengolahan citra.

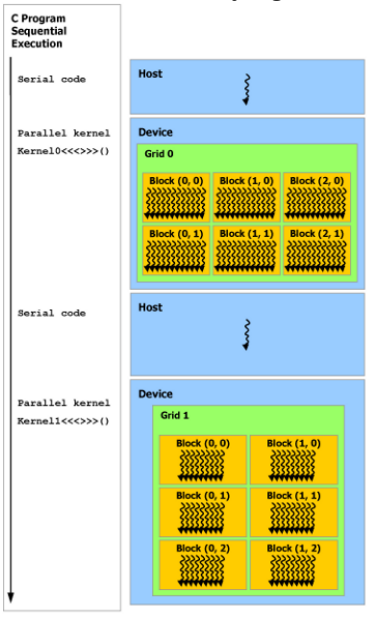
Meskipun memiliki banyak kelebihan untuk pemrosesan data secara paralel, GPU bukanlah pengganti dari arsitektur CPU. Sebaliknya, GPU adalah akselerator yang kuat untuk infrastruktur yang telah ada. Komputasi pada GPU menurunkan beban pada bagian-bagian komputasi yang intensif, sementara sisa komputasi lainnya masih berjalan pada CPU. Dari perspektif pengguna, aplikasi hanya berjalan lebih cepat. Walaupun komputasi dengan tujuan umum masih merupakan domain CPU, GPU adalah inti dari hampir semua aplikasi yang membutuhkan komputasi secara intensif.

Sebelum pada abad 20, penggunaan GPU dikhususkan untuk tujuan pengolahan citra dan susah untuk diprogram. Penggunaan GPU untuk permasalahan komputasi umum mulai dapat dipraktekkan pada tahun 2001, dengan diciptakannya shadder programming dan disediakannya

support untuk floating point pada prosesor grafis. Khususnya, masalah yang melibatkan matriks dan/atau vektor - terutama vektor dengan dua, tiga, atau empat-dimensi – cukup mudah diterjemahkan ke GPU yang bertindak dengan kecepatan native. Permasalahan multiplikasi matriks menjadi permasalahan pertama dalam penyelesaian permasalahan umum di luar render citra menggunakan GPU.

Penggunaan GPU pada permasalahan umum di luar permasalahan grafis disebut dengan GPGPU (General Purpose computing on Graphics Processing Units).. Proses pada GPGPU Programming didukung dengan adanya Heterogenous Programming, di mana pemrograman untuk permasalahan umum tidak hanya dikerjakan oleh GPU, tetapi juga CPU turut bekerja dalam menjalankan program dan menyelesaikan beberapa tugas. Kerangka GPU-CPU ini memberikan keuntungan yang lebih besar dibandingkan arsitektur beberapa CPU disebabkan oleh spesialisasi dalam setiap chip. Pada GPGPU Programming, CPU akan berfokus pada beberapa tugas yang tidak membutuhkan proses paralel dan membutuhkan komputasi yang lebih kompleks, sementara GPU akan berfokus pada tugas yang membutuhkan proses paralel yang besar dengan komputasi yang tidak terlalu rumit. Dalam pengaplikasian GPGPU Programming, CUDA merupakan salah satu tools yang dapat digunakan untuk dapat memanfaatkan CPU dan GPU secara bersamaan.

CUDA adalah platform komputasi paralel dan model pemrograman yang dikembangkan oleh NVIDIA untuk komputasi umum pada GPU (Graphics Processing Unit). Dengan CUDA, pembuat program dapat mempercepat aplikasi komputasi secara dramatis dengan memanfaatkan kekuatan GPU. Ketika menggunakan CUDA, pembuat program bekerja dalam bahasa populer seperti C, C ++, Fortran, Python, dan MATLAB, dan memanfaatkan teknik pararel melalui ekstensi yang disediakan.



Gambar 17. Heterogenous Programming pada CUDA

Prinsip kerja CUDA dimulai dengan pemanggilan kernel (method yang akan dijalankan pada GPU) melalui kode pada CPU. Jumlah thread untuk menjalankan kernel yang dipanggil dari CPU dapat didefinisikan terlebih dahulu oleh pengguna CUDA. Dalam pemrosesan kernel, CPU dan GPU akan bekerja secara asynchronous. Memori yang berasal dari CPU dan diperlukan oleh GPU, memerlukan suatu proses transfer yang dapat didefinisikan dengan sintaks yang akan dijelaskan pada subbab berikutnya. Hal ini juga berlaku apabila memori dari GPU telah diproses dan ingin dikembalikan menuju CPU kembali.

Hal yang perlu diketahui adalah pada CUDA, permasalahan yang harus diperhatikan adalah adanya bottleneck pada pentransferan memori dari CPU menuju GPU atau sebaliknya, optimalisasi dengan memaksimalkan kerja paralel GPU, dan permasalahan thread divergence yang perlu dihindari. CUDA menggunakan arsitektur SIMT (Single Instruction Multiple Threads) dan arsitektur SIMD (Single Instruction Multiple Data), di mana perbedaan kode yang dijalankan pada thread GPU (terjadi pada branching atau looping dengan jumlah loop yang tidak sama, disebut dengan thread divergence) akan menyebabkan proses saling menunggu di antara beberapa thread yang seharusnya bekerja dalam waktu yang bersamaan. Permasalahan ini menjadi tantangan bagi pembuat program dengan CUDA untuk dapat mengoptimasi kerja dari program yang dibuat.

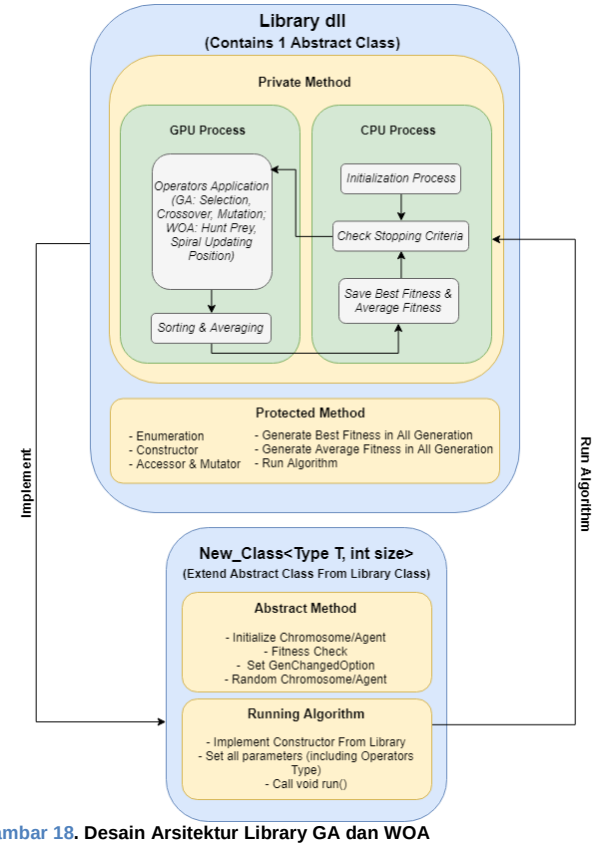
V. DESAIN LIBRARY [15], [16]

Untuk kedua algoritma yang dipakai dalam penelitian ini, kedua library berada pada file terpisah. Library yang dibuat hanya menyediakan 1 macam class di mana class ini harus diturunkan ke dalam class buatan user. Class ini telah berisi protected dan private method bawaan, di mana private method terbagi lagi menjadi 2 macam, yaitu method yang akan dijalankan pada GPU dan method yang akan dijalankan pada CPU. User sendiri tidak perlu mengatur bagian private method dikarenakan seluruh proses ini telah diatur oleh library. Untuk protected method sendiri terbagi menjadi 2 macam, yaitu abstract method dan normal method.

Desain arsitektur library dapat dilihat pada gambar 17. Algoritma pada library akan dijalankan secara implisit oleh library. Terdapat 4 macam abstract method yang harus/bisa diturunkan oleh pengguna library yang meliputi inisialisasi kromosom/agen, pengecekan fitness, pengaturan variabel GenChangedOption, dan random kromosom/agen. Inisialisasi kromosom/agen digunakan untuk menginisialisasikan nilai seluruh kromosom/agen dalam populasi, pengecekan fitness digunakan untuk mengecek fitness di setiap iterasi, pengaturan variabel GenChangedOption digunakan untuk mendefinisikan peraturan apakah gen tertentu pada kromosom/agen dapat diubah/tidak, dan random kromosom/agen digunakan untuk mendefinisikan nilai random untuk suatu kromosom/agen. Method setGenChangedOption secara default menunjukkan bahwa seluruh gen dapat diubah, dan method ini dapat diturunkan dalam class yang dibuat untuk mendefinisikan

peraturan pengubahan gen sesuai keinginan pengguna library.

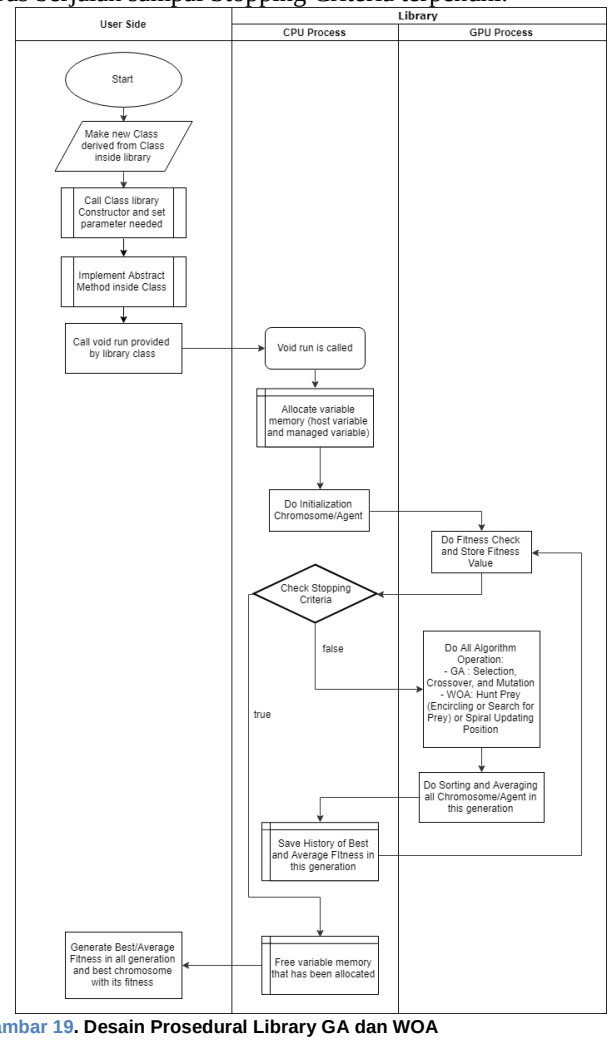
Untuk dapat menjalankan algoritma pada library, pengguna library hanya perlu mengimplementasikan konstruktor, mengatur parameter yang diperlukan (hal ini sudah termasuk dalam konstruktor dan dapat diubah secara manual sebelum library dijalankan), dan memanggil void run untuk menjalankan library.



Gambar 18. Desain Arsitektur Library GA dan WOA

Desain prosedural library dapat dilihat pada gambar 18. Dalam alur sistem dari sisi user, pada awalnya, pengguna library harus membuat suatu class yang merupakan turunan (extend) dari abstract class pada library. Pembuatan class ini otomatis diikuti dengan pemberian tipe dari solusi dan ukuran dari suatu solusi. Kemudian pengguna library perlu untuk memanggil konstruktor library dan mengatur parameter yang dibutuhkan. Jika hal ini tidak dilakukan, maka nilai yang diberikan pada parameter adalah nilai default. Selain pengaturan parameter, pengguna library perlu untuk melakukan implement pada abstract method yang ada. Penjelasan mengenai abstract method yang ada akan dijelaskan pada subbab Uji Coba. Setelah pengisian abstract method dilakukan, pengguna library dapat menjalankan void run yang telah disediakan library dan menunggu hingga proses selesai.

Pemanggilan void run akan menjalankan seluruh proses yang terjadi secara transparan. Pada awalnya akan dialokasikan variabel sesuai dengan memori yang dibutuhkan. Pada library yang dibuat, diimplementasikan sistem Unified Memory Programming, sehingga variabel yang dibutuhkan pada proses GPU akan dialokasikan secara Unified menggunakan sintaks cudaMallocManaged. Terdapat pula alokasi untuk host memori, yang dilakukan dengan fungsi bawaan CUDA cudaMallocHost yang menempatkan memory pada host dan disimpan dalam RAM untuk mempercepat pengaksesan memori. Kemudian akan dilakukan inisialisasi, yang telah diimplementasikan oleh user pada abstract method. Lalu akan dilakukan iterasi yang dimulai dari pengecekan nilai fitness setiap kromosom/agen, dan penyimpanan nilai fitness yang telah dihitung. Pengecekan nilai fitness ini juga telah didefinisikan oleh pengguna library pada abstract method. Proses iterasi akan terus berjalan sampai Stopping Criteria terpenuhi.



Gambar 19. Desain Prosedural Library GA dan WOA

Proses selanjutnya adalah dilakukannya operasi sesuai dengan algoritma. Apabila library yang dipakai adalah library GA, maka akan dilakukan selection, crossover, dan mutation, sementara apabila library yang dipakai adalah library WOA, maka akan dilakukan Hunt Prey (Encircling Prey atau Search for Prey) atau Spiral Updating Position. Jika proses operasi telah selesai, dilakukan proses Sorting and Averaging. Averaging digunakan untuk menyimpan

data rata-rata dari fitness untuk dapat disimpan dalam variabel AverageFitnessPerGeneration.

Hasil dari fitness terbaik dan fitness rata-rata akan disimpan pada proses Save History. Apabila seluruh proses telah selesai dan memenuhi Stopping Criteria, maka proses library diakhiri dengan pembebasan variabel yang telah dialokasikan. Dari sisi pengguna library, seluruh proses iterasi ini tidak terlihat dan pengguna library langsung dapat mengambil nilai fitness terbaik per generasi, nilai rata-rata fitness per generasi, nilai fitness terbaik di akhir generasi, dan nilai kromosom/agen terbaik di akhir generasi.

VI. UJI COBA

Proses uji coba ini dilakukan oleh pembuat library, yang dimulai dengan uji coba untuk seluruh method pada library, uji coba untuk permasalahan TSP, uji coba untuk permasalahan Sudoku, dan kemudian uji coba kecepatan dengan menggunakan kode CPU sebagai perbandingan. Uji coba dilakukan dengan menggunakan pada perangkat NVIDIA GeForce 840M dengan Compute Capability 5.0 dan 384 Cores.

Pada tahap awal setelah dilakukannya implementasi pada library, dilakukan proses uji coba pada seluruh modul library. Uji coba ini bertujuan untuk memastikan agar seluruh modul yang telah diimplementasikan berjalan dengan baik. Modul yang diujicobakan tidak hanya modul yang wajib diimplementasikan oleh user, tetapi juga modul yang secara opsional dapat diimplementasikan oleh user. Hasil uji coba akan diletakkan dalam sebuah tabel sebagai hasil dari uji coba terhadap modul. Pada tabel hasil uji coba modul library, akan digunakan kolom hasil yang menandakan apakah modul telah berhasil atau belum. Tanda O, berarti modul telah sesuai harapan, dan tanda X, berarti modul masih memunculkan beberapa keanehan. Uji coba modul pada library dapat dilihat pada tabel I

TABEL I
DAFTAR HASIL UJICOBA PADA MODUL LIBRARY GA

No.	Modul	Hasil
1	Constructor pada Class	O
2	Method run()	O
3	Inisialisasi proses	O
4	Inisialisasi per iterasi	O
5	Pengecekan *Stopping Criteria*	O
6	Pengecekan Nilai Fitness pada CPU dan GPU	X
7	Method doGPUOperation()	O
8	*Rank Based Selection*	O
9	*Roulette Wheel Selection*	O
10	*Tournament Selection*	O
11	*One Point Crossover*	O
12	*Multi Point Crossover*	O
13	*Uniform Crossover*	O
14	*PMX Crossover*	X
15	*Order1 Crossover*	X
16	*Random Resetting Mutation*	O
17	*Swap Mutation*	O
18	*Scramble Mutation*	O
19	*Inversion Mutation*	O
20	GPU Method generateChild	O
21	*Sorting* dan *Averaging*	O
22	Penyimpanan History	O
23	Pembebasan Memory	O
24	Pencetakan Hasil	O
25	Random Utility	O

Pada tabel I, dapat dilihat hasil uji coba library GA yang dilakukan untuk setiap method, bahwa seluruh method telah berjalan dengan baik kecuali pada pengecekan nilai fitness dan 2 jenis crossover. Pengecekan nilai fitness pada GPU telah berjalan dengan baik apabila tidak ada kebutuhan tambahan seperti pada TSP yang membutuhkan variabel koordinat. Apabila hal ini terjadi, maka pemanggilan fitness harus dilakukan pada CPU. Selain itu pada PMX Crossover dan Order1 Crossover, permasalahan yang terjadi adalah kecepatan yang menurun dengan sangat pesat dikarenakan dua crossover ini membutuhkan banyak iterasi dan sinkronisasi thread.

TABEL II
DAFTAR HASIL UJI COBA PADA MODUL LIBRARY WOA

No.	Modul	Hasil
1	Constructor pada Class	O
2	Method run()	O
3	Inisialisasi proses	O
4	Inisialisasi per iterasi	O
5	Pengecekan *Stopping Criteria*	O
6	Pengecekan Nilai Fitness pada CPU dan GPU	X
7	Method doGPUOperation()	O
8	*Encircling Prey/Search for Prey*	O
9	*Spiral Updating Position*	O
10	GPU Method generateChild	O
11	*Sorting* dan *Averaging*	O

12	Penyimpanan History	O
13	Pembebasan Memory	O
14	Pencetakan Hasil	O
15	Random Utility	O

Pada tabel II, untuk uji coba modul pada library WOA, dapat dilihat bahwa seluruh modul telah berjalan dengan baik kecuali pada pengecekan nilai fitness. Hal ini sama dengan library GA dikarenakan prinsip CUDA yang tidak dapat mempassingkan abstract function. Pengembangan pada library dapat dilakukan lebih lanjut untuk memperbaiki modul ini.

Setelah keseluruhan modul dievalueasi dan dipastikan dapat diimplementasikan pada library, maka proses selanjutnya adalah uji coba pada permasalahan TSP. TSP (Traveling Salesman Problem) adalah salah satu permasalahan umum yang dapat diselesaikan pada algoritma evolusioner. Pada permasalahan ini, diberikan satu area yang terdiri dari beberapa kota pada koordinat tertentu. Tujuannya adalah pencarian rute yang dilakukan dengan jarak seminimal mungkin, untuk mengunjungi masing-masing kota minimal 1 kali. Dalam implementasinya, terdapat 2 jenis permasalahan TSP yang akan diujicobakan dalam penelitian ini, yaitu TSP Fully Connected dan TSP Semi Connected.

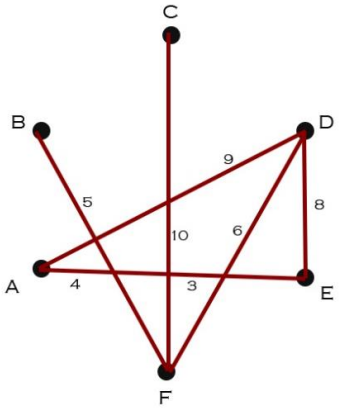
Permasalahan TSP Fully Connected menyatakan bahwa seluruh kota dapat terhubung dengan satu sama lain. Seluruh kota akan memiliki titik koordinat tersendiri yang dapat terhubung dengan kota yang terdapat dalam satu area TSP. Solusi berupa satu set urutan kota yang bersifat distinct, yaitu tidak ada kota yang sama dalam gen solusi. Secara tidak langsung, permasalahan ini termasuk dalam permasalahan permutasi. Permasalahan TSP ini sering kali tidak dapat diimplementasikan dalam keadaan riil yang di mana seluruh kotanya tidak pasti terhubung dan memiliki bobot jarak yang berbeda. Namun dalam implementasi, pencarian jarak kota minimal pada TSP Fully Connected dapat menjadi contoh permasalahan yang baik untuk diselesaikan.

Untuk uji coba pada TSP Fully Connected yang dilakukan, jumlah kota yang digunakan adalah sebanyak 38 kota. Data yang terdapat pada dataset ini adalah koordinat yang dimiliki setiap kota, di mana perhitungan jarak koordinat pada proses selanjutnya akan menggunakan rumus Eucledian Distance. Implementasi dilakukan dengan menggunakan library GA, khususnya karena algoritma genetik dapat menyelesaikan permasalahan tipe permutasi dengan lebih mudah. Representasi solusi pada permasalahan TSP Fully Connected ini berupa integer dengan ukuran solusi sebesar 38. Nilai setiap integer ini menyatakan indeks kota dimulai dari indeks 0, di mana setiap gen pada solusi bersifat distinct dan berbeda satu sama lain. Pada pengimplementasian TSP Fully Connected, input parameter pada library GA terdiri dari jumlah generasi sebanyak 10.000, jumlah kromosom per generasi sebanyak 150, nilai crossoverRate 0,15 dan nilai mutationRate 0.35, seleksi dengan tipe RankSelection, crossover dengan tipe PMXCrossover dan mutasi dengan tipe InversionMutation. Hasil output pada TSP Fully Connected dapat dilihat pada gambar 20.

```
Operation Finished..
Total Execution Time: 703.492981 s
Operation finished in generation 10000...
Best Fitness in last generation: 975563.500000
City Index: 26,5,2,17,21,10,8,20,28,9,34,31,27,37,29,3,12
6,7,19,24,15,22,14,18,23,11,30,36,4,32,1,13,16,0,35,33,25
```

Gambar 20. Hasil Output TSP Fully Connected

Permasalahan TSP Semi Connected menyatakan bahwa tidak seluruh kota akan terhubung dengan satu sama lain. Hal ini menyebabkan solusi pada kromosom tidak bersifat distinct, karena satu kota dapat dilewati lebih dari satu kali.



Gambar 21. Dataset TSP

Pada permasalahan TSP Semi Connected, seluruh kota tidak akan memiliki posisi khusus. Seluruh kota akan memiliki daftar indeks untuk kota yang dapat dijelajahi, beserta dengan bobotnya (jarak kedua kota). Solusi berupa satu set urutan kota yang tidak bersifat distinct, berbeda dengan permasalahan TSP Fully Connected. Permasalahan TSP ini lebih dapat diimplementasikan dalam keadaan riil yang di mana seluruh kotanya tidak pasti terhubung dan memiliki bobot jarak yang berbeda.

Untuk uji coba yang dilakukan pada TSP Semi Connected, representasi dataset yang digunakan terdapat pada gambar 15. Dataset ini berupa 6 kota yang tidak terhubung seluruhnya. Perhitungan nilai fitness untuk permasalahan ini dilakukan dengan melihat bobot yang ada, dengan penalti besar yang diberikan apabila tidak terdapat bobot antara kedua kota. Pada permasalahan TSP Semi Connected, implementasi dilakukan dengan menggunakan library WOA. Representasi solusi pada permasalahan TSP Semi Connected ini berupa integer dengan ukuran solusi sebesar 12. Ukuran solusi tersebut bernilai 12 dikarenakan satu kota dapat dikunjungi lebih dari 1 kali, untuk dapat mencapai kota lain pada permasalahan TSP Semi Connected ini. Nilai setiap integer ini menyatakan indeks kota dimulai dari indeks 0. Selain itu, library WOA juga cocok untuk penyelesaian permasalahan dengan angka sebagai representasinya. Pada pengimplementasian TSP Semi Connected ini, input

parameter hanya berupa jumlah generasi sebanyak 100 dan jumlah kromosom per generasi sebanyak 20. Hasil output pada TSP Semi Connected dapat dilihat pada gambar 22.

```
Operation Finished..
Total Execution Time: 0.362000 s
Operation finished in generation 100...
Best Fitness in last generation: 560.000000
City Index: -4.498109,-222.436325,-69.565346,-338.626038,-92.100510,-162.764526,
-57.863350,-43.891415,-78.754044,-58.722092,-159.527786,-160.901764
```

Gambar 22. Hasil Output TSP Fully Connected

Uji coba yang dilakukan setelah TSP adalah uji coba pada permasalahan sudoku. Sudoku adalah suatu permainan berupa teka teki angka, di mana pada awalnya disediakan kotak berukuran 9x9. Kotak ini harus diisikan dengan angka 1-9, dengan peraturan tidak boleh ada angka yang sama dalam satu baris, satu kolom, dan satu kotak berukuran 3x3. Contoh sudoku beserta penyelesaiannya dapat dilihat pada gambar 23.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Gambar 23. Contoh Sudoku dan Penyelesaiannya

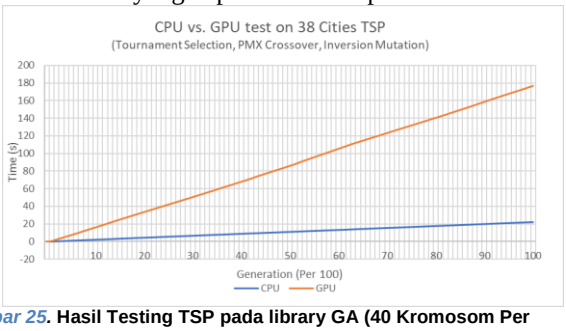
Pada persiapan uji coba pada Sudoku, akan dijelaskan mengenai dataset TSP, representasi solusi untuk kode pada library, dan input parameter GA. Untuk uji coba yang dilakukan pada Sudoku, disediakan soal sudoku tingkat medium yang dapat dilihat pada gambar 25. Dataset ini berupa kotak 9x9 dengan 21 angka yang terisi. Representasi solusi pada permasalahan Sudoku ini berupa integer dengan ukuran solusi sebesar 81. Nilai setiap integer ini menyatakan angka pada kotak sudoku yang mempunyai range dari 1-9. Pada permasalahan Sudoku, implementasi dilakukan dengan menggunakan library GA, dengan crossover dan mutasi yang tidak diperuntukkan permasalahan dengan tipe permutasi. Pada pengimplementasian Sudoku ini, input parameter pada library GA terdiri dari jumlah generasi sebanyak 10.000, jumlah kromosom per generasi sebanyak 150, crossoverRate 0.15 dan mutationRate 0.35, seleksi Tournament Selection, crossover Multi Point Crossover, mutasi Random Resetting. Hasil output pada Sudoku dapat dilihat pada gambar 24.

```
Operation Finished..
Total Execution Time: 698.310974 s
Operation finished in generation 10000...
Best Fitness in last generation: 3.000000
Best Sudoku:

5 3 4 5 4 7 2 6 8
6 8 9 2 1 2 3 6 7
7 3 7 9 6 3 4 9 8
1 3 1 6 5 8 4 2 7
8 2 3 4 1 9 5 7 6
4 4 5 2 3 6 7 1 3
1 9 5 7 8 4 2 3 6
9 7 8 6 3 2 4 1 5
1 3 4 1 2 4 6 5 7
```

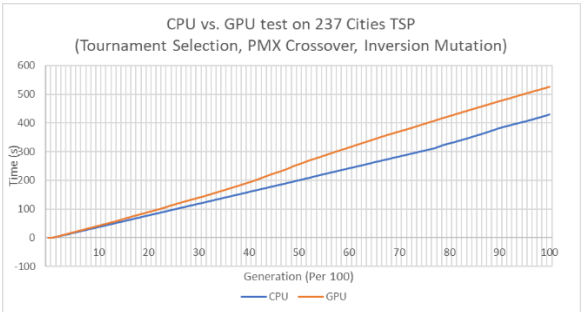
Gambar 24. Hasil Output Sudoku

Uji coba terakhir yang dilakukan adalah perbandingan kecepatan CPU dan GPU. Untuk tahap perbandingan kecepatan pada GPU dan CPU, dilakukan testing dengan berbagai parameter untuk library GA dan WOA. Uji coba pada library GA digunakan dalam permasalahan TSP Fully Connected dengan menggunakan Rank Selection, PMX Crossover, dan Inversion Mutation, sementara testing pada library WOA digunakan dalam permasalahan Sudoku. Perbandingan dilakukan dengan kode CPU yang telah dibuat oleh penulis, dengan perbandingan kode seminimal mungkin. Untuk test pada TSP, . Dataset yang digunakan adalah berupa 38 kota dan 237 kota. Ukuran solusi adalah 38 dan 237, jumlah solusi per generasi adalah 40 dan 100, dan jumlah generasi yang digunakan pada testing adalah berjumlah sampai 10.000 generasi. Maksimum dari jumlah solusi per generasi yang dapat dilakukan adalah 1.024, mengingat angka tersebut adalah angka maksimum dari threadPerBlocks yang dapat dilakukan pada GPU.



Gambar 25. Hasil Testing TSP pada library GA (40 Kromosom Per Generasi)

Pada gambar 25, hasil testing TSP menunjukkan hasil yang kurang memuaskan. Hal ini dikarenakan proses PMX Crossover yang membutuhkan banyak iterasi, meskipun implementasi dilakukan pada GPU. Hal ini menjadi suatu tantangan apabila hasil program dikembangkan lebih jauh lagi, untuk dapat berpikir secara paralel pada PMX Crossover. Kecepatan GPU juga menjadi lebih lambat karena implementasi hanya dilakukan pada 38 kota dan 40 kromosom per generasi, sehingga terjadi bottleneck pada transfer memori.



Gambar 26. Hasil Testing TSP pada library GA (100 Kromosom Per Generasi)

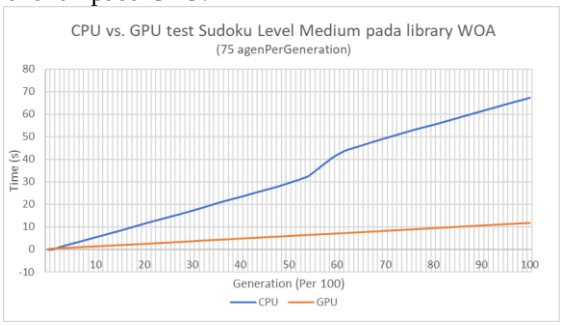
Pada gambar 26, hasil testing TSP menunjukkan hasil yang tetap kurang memuaskan, tetapi lebih baik dari hasil testing problem TSP sebelumnya. Peningkatan kecepatan terjadi karena jumlah kromosom pada algoritma lebih besar dari sebelumnya, yaitu sebanyak 8 kali lebih banyak dari

jumlah kota sebelumnya. Meskipun begitu, kecepatan yang dihasilkan masih kurang memuaskan dan dapat dioptimalkan lebih lanjut lagi

			5	4				8
6					2	3		
		7			3		9	
	3	1		5			2	
	4			3		7	1	
	9		7			2		
		8	6					5
1				2	4			

Gambar 27. Sudoku Dataset

Untuk dataset sudoku level medium yang dapat dilihat pada gambar 27 pada subbab Uji Coba Sudoku. Percobaan dilakukan dengan library WOA. Ukuran solusi secara otomatis bernilai 81, jumlah solusi per generasi bervariasi dari 75 sampai 1.000, dan jumlah generasi yang digunakan pada testing adalah berjumlah sampai 10.000 generasi. Maksimum dari jumlah solusi per generasi yang dapat dilakukan adalah 1.024, mengingat angka tersebut adalah angka maksimum dari threadPerBlocks yang dapat dilakukan pada GPU.



Gambar 28. **Hasil Testing Sudoku pada library WOA (75 Agen Per Generasi)**

Pada gambar 28, hasil testing Sudoku menunjukkan penaikan kecepatan dengan menggunakan GPU sebesar kira-kira 6 kali lebih cepat saat generasi mencapai angka 10.000. Hal ini merupakan hasil rata-rata dari lima percobaan yang dilakukan untuk masing-masing CPU dan GPU. Penaikan kecepatan pada proses GPU dapat dilihat dimulai dari 100 generasi karena peristiwa bottleneck pada pentransferan memori pada GPU Programming. Selain itu, masih terdapat anomali yang dapat dilihat dimulai dari generasi 55 sampai 62. Masih belum diketahui penyebab anomali pada generasi tersebut. Selanjutnya, dilakukan testing untuk angka agen per generasi yang lebih besar, yaitu 500 dan 1000. Hasil uji coba untuk permasalahan Sudoku level medium dengan 500 dan 1.000 agen per generasi dapat dilihat pada gambar 29.

Gambar 29. **Hasil Testing Sudoku pada library WOA (75 Agen Per Generasi)**

Pada percobaan testing dengan 500 dan 1.000 agen per generasi pada gambar 5.11, dihasilkan kecepatan yang cukup tinggi jika dibandingkan dengan kecepatan pada CPU. Hal ini dapat dilihat pada perbedaan kecepatan yang menonjol dimulai dari generasi ke-200. Peningkatan kecepatan dengan banyak agen per generasi ini mencapai 9 sampai 10 kali lipat lebih cepat dari implementasi CPU pada generasi 10.000. Hal ini menunjukkan bahwa dalam permasalahan kecepatan, library WOA yang dibuat telah melebihi kecepatan CPU dalam penyelesaiannya.

VII. KESIMPULAN

Dari hasil uji coba yang dilakukan, hasil output dari library yang dihasilkan sudah cukup sesuai dengan konteks permasalahan. Dalam segi kecepatan, library yang dihasilkan mampu meningkatkan kecepatan, apabila jumlah generasi dan solusi per generasi sangatlah besar. Namun, kecepatan ini tidak terjadi pada seluruh bagian library, seperti pada PMX Crossover di library GA yang memberikan kecepatan yang lebih lambat dari implementasi CPU. Pengembangan library ini dapat dilakukan lebih jauh untuk meningkatkan fungsionalitas dan kecepatan pada library.

PERAN PENULIS

Setiap penulis memiliki kontribusi yang sama dalam Analisis Formal, Investigasi, Administrasi Proyek, Sumber Daya, Perangkat Lunak, Validasi, Visualisasi, Penulisan Penyusunan Draf Asli, Penulisan Review & Penyuntingan.

COPYRIGHT

DAFTAR PUSTAKA

[1] L. Haldurai, T. Madhubala, and R. Rajalakshmi, "A study on genetic algorithm and its applications," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 10, p. 139, 2016.
[2] A. Thengade and R. Dondal, "Genetic algorithm--survey paper," in *MPGI national multi conference*, 2012, pp. 7–8.
[3] A. Lambora, K. Gupta, and K. Chopra, "Genetic algorithm-A literature review," in *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, 2019, pp. 380–384.
[4] P. Pickerling, H. Armanto, and S. K. Bastari, "Multilevel Image Thresholding Memanfaatkan Firefly Algorithm, Improved Bat Algorithm, dan Symbiotic Organisms Search," *J. Intell. Syst. Comput.*, vol. 1, no. 1, 2019, doi: 10.52985/insyst.v1i1.24.
[5] H. Armanto, R. Kevin, and P. Pickerling, "Perencanaan Perjalanan Wisata Multi Kota dan Negara dengan Algoritma Cuttlefish," *J. Intell. Syst. Comput.*, vol. 1, no. 2, 2019, doi: 10.52985/insyst.v1i2.91.
[6] O. Quan and H. Xu, "The study of comparisons of three crossover operators in genetic algorithm for solving single machine scheduling problem," in *2015 6th International Conference on Manufacturing Science and Engineering*, 2015, pp. 293–297.
[7] K. Puljić and R. Manger, "Comparison of eight evolutionary crossover operators for the vehicle routing problem," *Math. Commun.*, vol. 18, no. 2, pp. 359–375, 2013.
[8] P. H. P. Rosa, H. Sriwindono, R. A. Nugroho, A. M. Polina, and K. Pinaryanto, "Comparison of crossover and mutation operators to solve teachers placement problem by using genetic algorithm," in *Journal of Physics: Conference Series*, 2020, vol. 1566, no. 1, p. 12021.
[9] S. Mirjalili and A. Lewis, "The Whale Optimization Algorithm," *Adv. Eng. Softw.*, vol. 95, 2016, doi: 10.1016/j.advengsoft.2016.01.008.
[10] B. D. Shivahare, M. Singh, A. Gupta, S. Ranjan, D. Pareta, and B. M. Sahu, "Survey Paper: Whale optimization algorithm and its variant applications," in *2021 International Conference on Innovative Practices in Technology and Management (ICIPTM)*, 2021, pp. 77–82.
[11] F. S. Gharehchopogh and H. Gholizadeh, "A comprehensive survey: Whale Optimization Algorithm and its applications," *Swarm Evol. Comput.*, vol. 48, pp. 1–24, 2019.
[12] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "GPGPU processing in CUDA architecture," *arXiv Prepr. arXiv1202.4347*, 2012.
[13] D. Kirk, B. S. Center, and others, "NVIDIA CUDA software and GPU parallel computing architecture," 2008.
[14] R. S. Dehal, C. Munjal, A. A. Ansari, and A. S. Kushwaha, "GPU Computing Revolution: CUDA," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, 2018, pp. 197–201.
[15] R. S. Sinha, S. Singh, S. Singh, and V. K. Banga, "Accelerating genetic algorithm using general purpose GPU and CUDA," *Int. J. Comput. Graph.*, vol. 7, no. 1, pp. 17–30, 2016.
[16] M. Oiso, Y. Matsumura, T. Yasuda, and K. Ohkura, "Implementation Method of Genetic Algorithms to the CUDA Environment using Data Parallelization," *J. Japan Soc. Fuzzy Theory Intell. Informatics*, vol. 23, no. 1, pp. 18–28, 2011.