UNIVERSITAS INDONESIA

ALGORITMA PARALEL *REGULARIZED MARKOV CLUSTERING* PADA JARINGAN INTERAKSI PROTEIN-PROTEIN MENGGUNAKAN FORMAT DATA *SPARSE* ELLPACK-R

SKRIPSI

UMBU MARAMBA MESA
0806452324

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
PROGRAM STUDI SARJANA MATEMATIKA
DEPOK
JULI 2012

UNIVERSITAS INDONESIA

ALGORITMA PARALEL *REGULARIZED MARKOV CLUSTERING* PADA JARINGAN INTERAKSI PROTEIN-PROTEIN MENGGUNAKAN FORMAT DATA *SPARSE* ELLPACK-R

SKRIPSI

Diajukan sebagai salah satu syarat untuk memperoleh gelar sarjana sains

UMBU MARAMBA MESA
0806452324

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
PROGRAM STUDI SARJANA MATEMATIKA
DEPOK

JULI 2012

HALAMAN PERNYATAAN ORISINALITAS

Skripsi ini adalah hasil karya saya sendiri,

dan semua sumber baik yang dikutip maupun dirujuk

telah saya nyatakan dengan benar.

Nama	: Umbu Maramba Mesa
NPM	: 086452324
Tanda Tangan	: 
Tanggal	: 20 Juni 2012

HALAMAN PENGESAHAN

Skripsi ini diajukan oleh

Nama	: Umbu Maramba Mesa
NPM	: 0806452324
Program Studi	: Sarjana Matematika
Judul Skripsi	: Algoritma Paralel *Regularized Markov Clustering* pada Jaringan Interaksi Protein-Protein Menggunakan Format Data *Sparse* ELLPACK-R

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Sains pada Program Studi Matematika, Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Indonesia.

DEWAN PENGUJI

Pembimbing	: Alhadi Bustamam, Ph.D.	( )
Penguji I	: Bevina D. Handari, Ph.D.	( )
Penguji II	: Frederik Moses Poyk, M.Kom.	( )
Penguji III	: Gatot Fatwanto Hertono, Ph.D.	( )

Ditetapkan di : Depok
Tanggal : 20 Juni 2012

KATA PENGANTAR

Puji syukur penulis panjatkan kepada Yesus Kristus, karena atas kasih karunia-Nya, penulis dapat menyelesaikan skripsi ini. Penulisan skripsi ini dilakukan dalam rangka memenuhi salah satu syarat untuk mencapai gelar Sarjana Sains Jurusan Matematika pada Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Indonesia. Penulis menyadari bahwa, tanpa bantuan dan bimbingan dari berbagai pihak, sangatlah sulit bagi penulis untuk menyelesaikan skripsi ini. Karenanya, penulis mengucapkan terima kasih kepada:

(1) Mama, Patricia, Urdat, dan segenap keluarga besar di Sumba yang telah memberikan dukungan moril dan materil sejak lahir hingga saat ini, terutama saat penulisan skripsi. Terima kasih untuk doa yang dipanjatkan serta semangat yang tiada henti diberikan, *I love you*, *guys*.

(2) Bapak Alhadi Bustamam, Ph.D. selaku Pembimbing Tugas Akhir. Terima kasih banyak atas semua ilmu, nasihat, kesabaran, saran, dan motivasi yang telah diberikan selama penulisan skripsi ini. *Without you I won't be able to do such thing as completing this minithesis*.

(3) Ibu Dra. Sasky Mary Soemartojo, M.Si. selaku Pembimbing Akademis penulis selama penulis menempuh perkuliahan. *Thanks a ton for every good thing you've given to me, I'll never forget them*.

(4) Seluruh Dosen Departemen Matematika yang telah memberikan ilmunya kepada penulis selama 4 tahun ini. *You guys rock*! *Without you I won't be able to be like what I am now*.

(5) Seluruh karyawan Departemen Matematika khususnya kepada pihak Laboratorium Komputer Departemen Matematika yang telah banyak membantu penulis selama masa perkuliahan, penulis menghaturkan ribuan terima kasih. Semoga Tuhan membalas segala perbuatan baik kalian.

(6) KTB 38, ka Andy, Alvin, Anthony, Arkies, Edvan, Epin, Hendry, dan James, juga kepada ka Rontu, terima kasih atas kebersamaannya selama ini, penulis merasa sangat beruntung bisa bersahabat dengan kalian. Penulis

banyak mendapatkan makanan rohani dari kalian, penulis juga sangat senang bisa menjalani hari-hari yang cukup gila bersama kalian.

(7) Eci yang bawel dan ngeselin, Ansar si suara emas, Rogam si calon master TI, Fuad yang lemot, Ivan yang sering galau, Futdin yang heboh, Bio yang baik hati nan bijak, dan Joe yang cerewet kayak emak-emak, terima kasih atas bantuannya selama ini, dukungan moril dari kalian sangat berarti bagi penulis selama masa penulisan skripsi. *It's a great pleasure to know you, guys*.

(8) Teman-teman seperjuangan dalam menyusun skripsi: Ade, Citra, Awe, Laili, Dhila, Andy, Tuti, Risya, terima kasih atas hari-hari yang menyenangkan selama ini. Juga kepada Dian dan Ka Fauzan selaku rekan seperguruan di perguruan silat Bustamam, terima kasih untuk segala perbuatan baik kalian.

(9) Rekan-rekan seperjuangan Math UI 08: Yulial, Resti, Yulian, Cindy, Ines, Agnes, Ijut, Eka, May, Mei, Dheni, Oline, Adhi, Sita, Bowo, Ega, Agy, Dhea, Nita, Luthfa, Numa, Uchi, Ucil, Vika, serta teman-teman lain yang tidak mungkin disebutkan satu per satu. Terima kasih untuk kebersamaannya selama ini, *love you*, *guys*. *One Math, One Family*!

(10) Ka Ayat, Ka Hanif, Ka Nora, serta semua senior angkatan 2007 dan 2006, terima kasih untuk bimbingan dan nasehatnya selama ini, penulis sangat menghargai itu semua.

(11) Ninna, Sofi, Cepi, Yuza, Pino, Ganesha, Fikri, Bayu, dan seluruh rekan Math UI 2009 – 2011, *thank you guys for your smile, it helped me a lot through my rainy days*. Maaf kalau ada ucapan dan perbuatan kakak yang mungkin pernah membuat kalian kesal. Sukses ya.

Akhir kata, penulis berharap Tuhan yang Maha Pengasih lagi Penyayang berkenan membalas segala kebaikan semua pihak yang telah membantu. Semoga skripsi ini membawa manfaat bagi pengembangan ilmu.

Penulis

2012

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

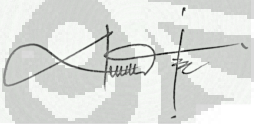
Nama	: Umbu Maramba Mesa
NPM	: 0806452324
Program Studi	: Sarjana Matematika
Departemen	: Matematika
Fakultas	: Matematika dan Ilmu Pengetahuan Alam
Jenis karya	: Skripsi

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia Hak Bebas Royalti Noneksklusif (*Non-exclusive Royalty Free Right*) atas karya ilmiah saya yang berjudul:
Algoritma Paralel *Regularized Markov Clustering* pada Jaringan Interaksi Protein-Protein Menggunakan Format Data *Sparse* ELLPACK-R.

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/format-kan, mengelola dalam bentuk pangkalan data (database), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/penciptadan sebagai pemilik Hak Cipta.

Demikian pernyataan ini saya buat dengan sebenarnya.

Dibuat di : Depok
Pada tanggal : 20 Juni 2012
Yang menyatakan



(Umbu Maramba Mesa)

ABSTRAK

Nama : Umbu Maramba Mesa
Program Studi : Matematika
Judul : Algoritma Paralel *Regularized Markov Clustering* pada Jaringan Interaksi Protein-Protein Menggunakan Format Data *Sparse* ELLPACK-R

Pada jaringan interaksi protein-protein terdapat beberapa protein yang menjadi pusat cluster, dimana protein-protein tersebut merupakan protein yang memegang peranan penting dalam sebuah fungsi seluler. Salah satu algoritma yang dewasa ini sering digunakan untuk melakukan pencarian pusat kluster adalah algoritma *Markov Clustering* (MCL). Algoritma *Regularized Markov Clustering* (R-MCL) merupakan algoritma modifikasi MCL yang bertujuan untuk mencari pusat kluster dengan mensimulasikan *random walk* dalam graf interaksi protein-protein dengan menggunakan operasi ekspansi namun tetap mempertahankan topologi awal dari graf. Komputasi paralel diperlukan dalam menyelesaikan proses klusterisasi ini sebab R-MCL melibatkan data yang berukuran besar dan mengandung proses yang memiliki kompleksitas waktu yang besar. Dalam skripsi ini akan dibahas mengenai konstruksi algoritma paralel R-MCL menggunakan bahasa pemrograman CUDA C pada GPU. Data disimpan dalam format yang lebih hemat memori yaitu format data *sparse* ELLPACK-R yang sesuai untuk komputasi pada GPU. Algoritma paralel ini akan diimplementasikan pada mesin *manycore* dengan menggunakan NVCC *compiler*.

Kata Kunci : CUDA, ELLPACK-R, GPU, *parallel computing, protein-protein interaction network, Regularized Markov Clustering*.
xiv+79 halaman ; 31 gambar; 16 tabel; 12 lampiran
Daftar Pustaka : 17 (2000-2012)

ABSTRACT

Name : Umbu Maramba Mesa
Program Study : Mathematics
Title : Balanced and Scalable Markov Clustering of Protein Interaction Networks using ELLPACK-R Sparse Data Format on GPU Computing Parallel Regularized Markov Clustering Algorithm on Protein-Protein Interaction Networks using ELLPACK-R Sparse Data Format

There are some proteins in protein-protein interaction network that act as the cluster centers because of the important roles they have related to cellular functions. One of the clustering algorithms that are often used in clustering is Markov Clustering Algorithm (MCL). Regularized Markov Clustering (R-MCL) algorithm is a modification of MCL in order to get better results by simulating random walk in the graph using expansion operation while maintaining the original topology of the graph. Parallel computation is needed to solve this clustering problem because R-MCL algorithm uses a big number of data and contains some operations with very big time complexities. The problem that will be discussed in this minithesis is the construction of parallel R-MCL algorithm using CUDA C on GPU. The PPI data will be converted into a more memory-friendly format, in this case in ELLPACK-R sparse data format that is suitable for GPU computation. This parallel algorithm will be implemented using a manycore machine with NVCC compiler installed on it.

Keywords : CUDA, ELLPACK-R, GPU, parallel computing, protein-protein interaction network, Regularized Markov Clustering.
xiv+79 pages ; 31 pictures; 16 tables; 12 attachments
Bibliography : 17 (2000-2012)

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR LAMPIRAN

BAB 1
PENDAHULUAN

1.1 Latar Belakang

Seiring berkembangnya teknologi di beberapa cabang ilmu biologi, berbagai macam proyek seperti proyek *sequencing*, studi *microarray*, proyek pemetaan *interactome*, studi mengenai fungsi gen, struktur genomik, serta proyek lainnya menghasilkan *database* berskala besar yang terus bertambah dengan kecepatan yang cukup signifikan. Dalam bioinformatika, *database* yang dihasilkan tersebut diproses sehingga diperoleh gambaran dari berbagai proses biologis dan kerja dari berbagai sub-sistem yang terdapat pada tingkat molekular, sel, jaringan, bahkan dalam suatu organisme utuh. Kunci dari proses ini adalah bahwa fungsi seluler tidak bergantung pada DNA, RNA, protein, atau molekul-molekul tunggal, namun pada interaksi yang melibatkan berbagai tipe molekul seperti protein-protein (PPI, *protein-protein interaction*), protein-DNA/RNA, protein-metabolit, maupun interaksi genetis lainnya yang membentuk *network* yang rumit (Satuluri & Parthasarathy, 2009). Untuk itu, diperlukan adanya suatu metode untuk menyederhanakan bentuk dari jaringan interaksi tersebut sehingga memudahkan interpretasi terhadap interaksi-interaksi yang terjadi di dalamnya.

Markov Clustering Algorithm (MCL) merupakan algoritma *clustering* (pengelompokan) yang dibangun untuk membantu menyelesaikan masalah *graph clustering*. Algoritma ini dikembangkan oleh Stijn van Dongen dan telah diimplementasikan di berbagai bidang ilmu, salah satunya di bidang bioinformatik seperti pada jaringan interaksi protein, analisa famili protein dan gen dari spesies tunggal maupun antarspesies, analisa ruang sekuens, *orthologous groups*, protein kinase, protein tersekretasi, protein mata, elemen genetis yang bersifat *mobile*, dan penentuan fungsi protein. Selain itu, MCL juga banyak dipakai di bidang *corpus linguistic*, pencarian gambar yang bersifat *content-based*, analisa *peer-to-peer network*, dan analisa jejaring sosial (Dongen, 2008).

Beberapa faktor yang mendorong banyaknya penggunaan algoritma MCL meliputi kemampuan algoritma ini untuk menghasilkan *clustering* nonhierarkis

yang setimbang (*well-balanced*), merupakan metode *bootstrapping*, memiliki parameter natural (*inflation*) yang dapat mempengaruhi glanuralitas dari *cluster* yang dihasilkan, dapat diimplementasikan dalam *sparse graph/matrix* yang berimplikasi pada skalabilitas yang baik, serta adanya hasil matematis yang dapat menjelaskan hubungan erat dari setiap iterasi proses MCL, interpretasi *cluster*, *inflation*, dan jumlah dari *cluster* yang dihasilkan (Dongen, 2008).

Dalam melakukan proses *clustering*, salah satu hal yang menjadi tujuan utama adalah meminimalisir jumlah *cluster* yang berukuran besar, sebab kompleks protein cenderung hanya memiliki sekitar 15-30 *nodes*. Serupa dengan itu, *output* berupa *singleton cluster* juga perlu diminimalisir jumlahnya, sebab *node* tunggal tidak mengandung cukup informasi yang bisa digunakan untuk mengidentifikasi interaksi dalam sebuah *network*. Banyak penelitian dan simulasi telah dilakukan untuk mengembangkan algoritma MCL untuk mencapai hasil yang lebih efisien. Dalam skripsi ini, algoritma yang dipakai untuk *clustering* jaringan interaksi dari protein-protein adalah algoritma *Regularized Markov Clustering Algorithm* (R-MCL) yang dikembangkan oleh Satuluri dan Parthasarathy (2009).

Di sisi lain, perkembangan teknologi GPU (*Graphics Processing Unit*) meningkat dengan sangat pesat. Berbeda dengan CPU (*Central Processing Unit*) yang masih berkutat dengan teknologi *multicore*, GPU yang selama ini dikenal sebagai perangkat *gaming* dan grafis telah beberapa langkah lebih maju dengan menerapkan teknologi *manycore*. Ratusan prosesor yang ditanam dalam sebuah GPU mengakibatkan peningkatan performa yang luar biasa dalam penggunaan GPU sebagai perangkat pemrograman paralel. Penggunaan GPU menjadi perangkat pemrograman yang lebih umum dikenal sebagai GPGPU (*General Purpose Computation on GPU*). Jumlah yang begitu besar dari prosesor ini memungkinkan adanya pengeksekusian program secara paralel berskala besar (*massively parallel programming*). Dengan memanfaatkan kelebihan dari GPU ini, berbagai proses yang membutuhkan waktu komputasi yang sangat besar bisa diselesaikan menggunakan GPU dalam waktu yang jauh lebih singkat. Pemrograman dengan GPU ini dilakukan dengan bantuan ekstensi minimal dari

bahasa pemrograman C/C++ yang disebut CUDA (*Compute Unified Device Architecture*) yang dikembangkan oleh NVIDIA.

Dengan memanfaatkan kelebihan ini, bisa dibangun suatu algoritma paralel dari *Markov Clustering Algorithm* mengunakan GPU dengan memperhatikan skalabilitas dan kesetimbangan dari *cluster* yang dihasilkan sehingga dapat digunakan dalam proses *clustering* untuk *protein-protein interaction networks*. Implementasi ini dilakukan dengan menggunakan format data ELLPACK-R yang merupakan pengembangan dari ELLPACK untuk optimisasi *sparse matrix vector multiplication* (SpMV) pada GPU. Konstruksi ELLPACK-R serupa dengan ELLPACK namun dengan menambahkan sebuah *array* yang berisi informasi mengenai banyak data yang tidak nol di setiap baris. Optimasi dari ELLPACK-R dilakukan untuk mencapai hasil yang optimal dalam pemetaan *thread* pada GPU, eksekusi yang *free-synchronization* (setiap *thread* mengeksekusi hasil dari baris yang berbeda), akses memori global, eksekusi yang *divergence-less*, serta pengunaan ulang data.

1.2 Perumusan Masalah

Bagamana membangun algoritma paralel yang *scalable* serta memiliki *speedup* yang baik dalam menyelesaikan masalah *clustering* pada *protein-protein interaction networks*?

1.3 Pembatasan Masalah

Pada skripsi ini, GPU yang digunakan memiliki maksimum 336 prosesor, maksimum jumlah *threads per block* adalah 1024, serta maksimum *block dimension* dan maksimum *grid dimension* adalah 65535. Algoritma diterapkan pada *protein-protein interaction* (PPI) *networks* dan disimulasikan pada komputer ber-OS Ubuntu dengan menggunakan NVIDIA CUDA *Compiler* dengan bantuan bahasa pemrograman CUDA C. Pembahasan hanya dibatasi pada konstruksi algoritma paralel untuk R-MCL dengan menggunakan *parallel reduction* tipe 7.

1.4 Metode Penelitian

Jenis penelitian yang digunakan adalah pengembangan algoritma dan simulasi.

1.5 Tujuan Penelitian

Tujuan dari penulisan skripsi ini adalah membangun algoritma paralel yang *scalable* dan memiliki *speedup* yang baik menggunakan GPU dengan bahasa pemrograman CUDA C untuk menyelesaikan masalah *clustering* pada *protein-protein interaction networks* dengan memanfaatkan format matriks *sparse* ELLPACK-R.

BAB 2
LANDASAN TEORI

Pada bab ini akan dibahas teori dasar mengenai biologi molekuler dan komputasi paralel berbasis CUDA yang diperlukan pada pembahasan bab-bab berikutnya, diantaranya biologi molekuler, *Markov Clustering Algorithm* (MCL) dan *Regularized Markov Clustering Algorithm* (R-MCL), komputasi paralel, GPU *Computing*, serta format penyimpanan data *sparse*.

2.1 Biologi Molekuler

Sel merupakan unit struktural dan fungsional terkecil penyusun tubuh makhluk hidup. Sel tersusun atas berbagai molekul yang saling bekerjasama menghasilkan fungsi-fungsi tertentu. Agar fungsi sel bisa berjalan dengan baik, molekul-molekul penyusun sel harus memenuhi dua kriteria. Pertama, molekul-molekul ini harus melakukan berbagai jenis reaksi kimia yang penting bagi berlangsungnya kehidupan. Agar fungsi ini bisa berjalan, sel membutuhkan struktur tiga dimensi yang beranekaragam dari molekul-molekul yang saling berinteraksi. Kedua, molekul-molekul ini harus menyampaikan informasi penting berupa instruksi mengenai cara pembuatan komponen-komponen konstituen kepada keturunannya. Cara paling efektif yang bisa dilakukan untuk memenuhi tujuan ini adalah dengan menggunakan media penyimpanan satu dimensi. Protein merupakan solusi untuk struktur tiga dimensi pada kriteria pertama, sedangkan DNA (*Deoxyribonucleic Acid)* merupakan solusi bagi penyimpanan informasi pada kriteria kedua. Molekul seluler yang lain yang disebut RNA (*Ribonucleic Acid*) berperan sebagai perantara bagi protein dan DNA serta melakukan beberapa fungsi dari kedua molekul tersebut (Tompa, 2009).

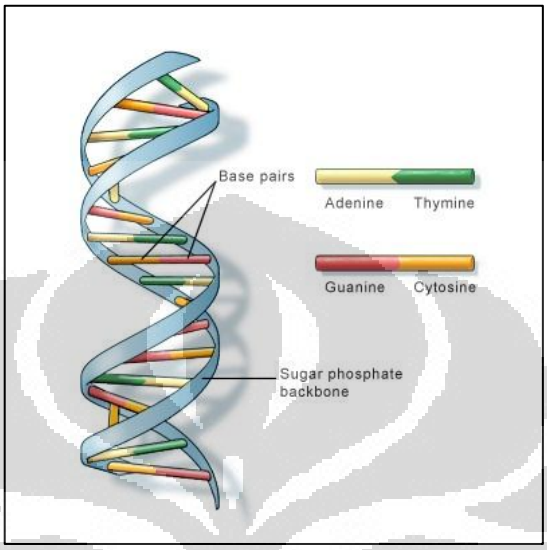
2.1.1 Protein

Protein memiliki berbagai macam fungsi yaitu sebagai pembawa sinyal dari dan ke luar sel serta di dalam sel, berperan sebagai enzim atau katalis biologis pada reaksi kimia, transportasi molekul-molekul kecil, pembentuk berbagai struktur seluler, serta sebagai pengatur proses-proses yang terjadi di dalam sel. Bentuk tiga dimensi dari protein dibentuk oleh komposisi satu dimensi dari protein itu sendiri dimana setiap protein merupakan rangkaian linier yang terdiri dari molekul-molekul konstituen yang lebih kecil. Molekul-molekul ini disebut asam amino. Bentuk tiga dimensi dari protein ditentukan oleh rangkaian linier asam amino dari N-*terminus* ke C-*terminus*. Protein mengandung sekitar 20 - 5000 asam amino, dengan rata-rata sekitar 350 asam amino. Setiap asam amino memiliki dua bagian yaitu bagian yang identik dengan 19 asam amino lainnya serta bagian yang berbeda disebut *side chain* atau R *group*. Bagian yang sama berfungsi sebagai pengubung antara asam amino yang satu dengan asam amino yang lain untuk membentuk rangkaian protein, sedangkan *side chain* menentukan sifat kimia dan fisis dari asam amino. Beberapa sifat kimia utama yang dimiliki oleh asam amino diantaranya adalah bermuatan positif (*basic*), negatif (*acidic*), polar (*hydrophilic*), dan nonpolar (*hydrophobic*) (Tompa, 2009).

2.1.2 DNA (*Deoxyribonucleic Acid*)

DNA mengandung instruksi-instruksi yang diperlukan oleh sel agar bisa melakukan fungsinya dengan baik. Informasi genetis lengkap yang terkandung di dalam DNA mendefinisikan struktur dan fungsi dari sebuah organisme. DNA terdiri dari dua jalinan pita yang membentuk *double helix*. Setiap sel manusia mengandung 23 pasangan kromosom yang masing-masing merupakan molekul DNA rantai-ganda yang panjang (Tompa, 2009). Setiap pita DNA dibangun oleh serangkaian molekul konstituen yang disebut nukleotida (dNTP, *deoxynucleoside triphosphates*). Setiap nukleotida terdiri dari tiga bagian yaitu grup fosfat, gula deoksiribosa, dan basa nitrogen. Fosfat dan deoksiribosa berfungsi sebagai pembentuk pita dari jalinan DNA. Basa nitrogen merupakan satu-satunya bagian yang membedakan nukleotida yang satu dengan nukleotida yang lain (Vierstraete,

2000). Terdapat empat macam basa pada DNA yaitu timin (T), sitosin (C), adenin (A), dan guanin (G), dengan adenin berpasangan dengan timin (A-T) serta guanin dengan sitosin (G-C) (Hughes, 2005).

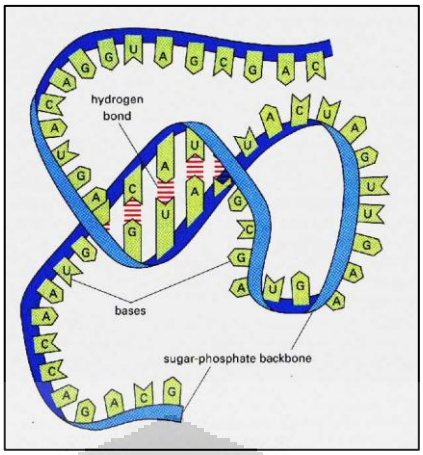


Gambar 2. 1 DNA

[Sumber: http://ghr.nlm.nih.gov]

2.1.3 RNA (*Ribonucleic Acid*)

Secara kimiawi, RNA sangat mirip dengan DNA. Namun, terdapat dua perbedaan utama dimana RNA menggunakan gula ribosa sebagai pengganti deoksiribosa dan basa urasil (U) sebagai pengganti timin. RNA memiliki dua sifat penting yaitu RNA cenderung membentuk rantai tunggal serta sering membentuk ikatan hidrogen intramolekuler yang secara parsial berhibridisasi dengan dirinya sendiri. Sama seperti protein, RNA bisa membentuk lipatan tiga dimensi yang kompleks. RNA memiliki beberapa sifat yang dimiliki oleh protein dan DNA. RNA bisa menyimpan informasi genetis layaknya DNA serta memiliki sifat enzimatis seperti protein (Tompa, 2009).



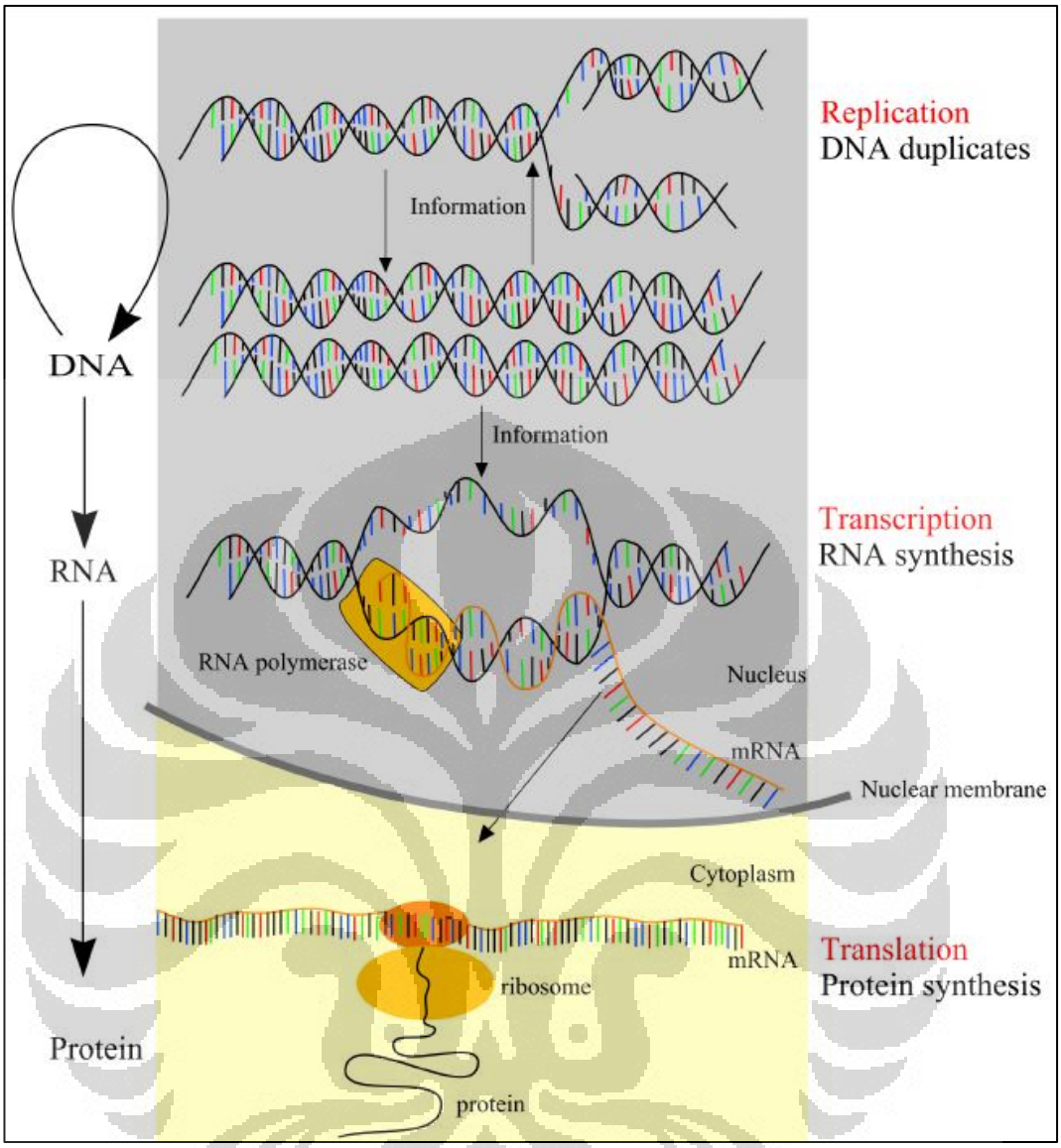
Gambar 2. 2 RNA

[Sumber: http://www.uic.edu]

2.1.4 Sintesis DNA, RNA, dan Protein

Menurut Dogma Sentral Biologi Molekuler (Hughes, 2005), molekul-molekul seluler dapat melakukan sintesis membentuk dirinya sendiri (duplikasi DNA) maupun membentuk molekul lain (sintesis RNA dan protein). Fungsi dari rantai-ganda DNA adalah sebagai media replikasi DNA yang akan diwariskan ke sel anak. Saat pembelahan sel terjadi, rantai ganda DNA akan terpisah menjadi dua rantai tunggal yang kemudian akan digunakan sebagai cetakan untuk sintesis rantai pasangan masing-masing sehingga terbentuk dua DNA baru yang identik (Tompa, 2009).

DNA mengandung informasi yang dibutuhkan oleh sel untuk membentuk RNA dan protein. Proses pembentukan RNA dengan menggunakan informasi pada DNA disebut transkripsi. Sebuah enzim yang disebut RNA *polymerase* secara sementara akan melepas rantai ganda DNA kemudian menggunakan salah satu rantai sebagai cetakan pembentuk rantai RNA. Transkripsi dimulai pada sebuah bagian pendek terpola pada DNA yang disebut *transcription start site*. Saat RNA *polymerase* mencapai suatu bagian pada DNA yang disebut *transcription stop site*, maka proses penyalinan rantai tunggal DNA akan berhenti sehingga terbentuklah sebuah mRNA (*messenger* RNA).



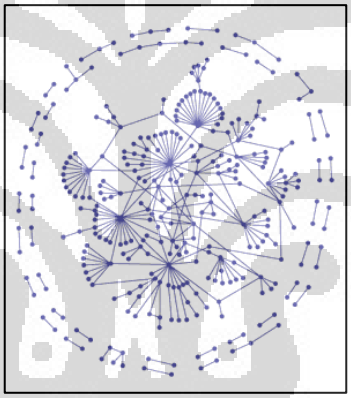
Gambar 2. 3 Dogma Sentral Biologi Molekuler

[Sumber: *Primary DNA Molecules Structure*, Andrew Hughes, 2005]

Proses sintesis protein dari mRNA disebut translasi. Asam amino dinyatakan dengan pasangan 3 nukleotida yang disebut kodon. Sebuah organel di dalam sel yang disebut ribosom berfungsi untuk membaca mRNA dan membentuk protein sesuai dengan kode genetis. Ribosom tersusun oleh protein dan rRNA (*ribosomal* RNA). Proses translasi mRNA menjadi protein menggunakan 61 buah tRNA (*transfer* RNA), masing-masing satu untuk setiap kodon non-terminasi (Tompa, 2009).

2.1.5 *Protein Interaction Network*

Agar fungsi sel bisa berjalan dengan baik, molekul-molekul penyusun sel harus melakukan berbagai jenis reaksi kimia yang penting bagi berlangsungnya kehidupan. Namun, untuk melakukan reaksi-reaksi tersebut, molekul-molekul di dalam sel tidak bisa bekerja sendiri. Molekul-molekul seperti protein, DNA, dan RNA bekerja sama untuk melakukan fungsi seluler sehingga terbentuk suatu jaringan interaksi yang rumit seperti interaksi protein-protein, protein-DNA/RNA, protein-metabolit, dan interaksi genetis lainnya (Satuluri *et al.*, 2010). Dalam setiap jaringan interaksi protein-protein yang terjadi, terdapat beberapa protein yang menjadi pusat interaksi. Protein yang menjadi pusat *cluster* biasanya merupakan protein yang memegang peranan penting dalam sebuah fungsi seluler.



Gambar 2. 4 Jaringan PPI dari 329 Protein pada Nukleus Sel *Yeast* [Sumber: bnl.gov]

2.2 *Graph Clustering*

Graph clustering merupakan proses pengelompokan verteks-verteks pada sebuah graf berdasarkan pada ada tidaknya hubungan antar-verteks dan seberapa banyak hubungan yang terjadi antara satu verteks dengan verteks yang lain. *Markov Clustering Algorithm* (MCL) merupakan algoritma yang dibangun untuk menyelesaikan masalah *graph clustering*. MCL telah diimplementasikan pada berbagai bidang, termasuk bioinformatik, dalam hal ini pada jaringan interaksi protein (PPI *networks*). Beberapa keuntungan MCL adalah kemampuan untuk

menghasilkan *cluster* yang setimbang dan *flat* (*nonhierarchic*), merupakan metode *bootstrapping*, memiliki parameter natural yang mempengaruhi glanularitas dari *cluster* yang dihasilkan, implementasinya sesuai untuk graf/matriks *sparse*, serta adanya hasil matematis yang dapat menjelaskan hubungan erat dari setiap iterasi proses MCL, interpretasi *cluster*, *inflation*, dan jumlah dari *cluster* yang dihasilkan (Dongen, 2000).

Pada skripsi ini, implementasi algoritma MCL dilakukan pada jaringan interaksi protein-protein. Jika graf G merupakan graf interaksi protein dengan protein dalam sebuah sel makhluk hidup, maka dapat dibentuk sebuah matriks *adjacency* M berukuran $n \times n$, dimana n adalah banyak protein dalam jaringan interaksi tersebut. Jika sebuah protein memiliki hubungan dengan sutau protein lain, maka entri yang bersesuaian pada matriks M akan diberi nilai 1, sedangkan jika tidak ada hubungan maka entri tersebut akan diberi nilai 0. Dengan menggunakan MCL, maka akan diperoleh sebuah matriks idempoten yang mengandung *cluster*-*cluster* dari jaringan interaksi ini. Pusat dari setiap *cluster* ditentukan dengan mengambil setiap elemen diagonal yang tak-nol, sebut $M_{i,i} \neq 0$, dengan $i = 1,2, \dots, n$, dan anggota *cluster* merupakan merupakan elemen-elemen tak-nol pada baris ke-i.

2.2.1 Algoritma *Markov Clustering* (MCL)

Berdasarkan Dongen (2000), terdapat tiga operasi utama yang menjadi jantung dari algoritma MCL yaitu ekspansi (*expand*), inflasi (*inflation*) dan pemotongan (*prune*). Berikut merupakan penjelasan dari setiap proses pada MCL.

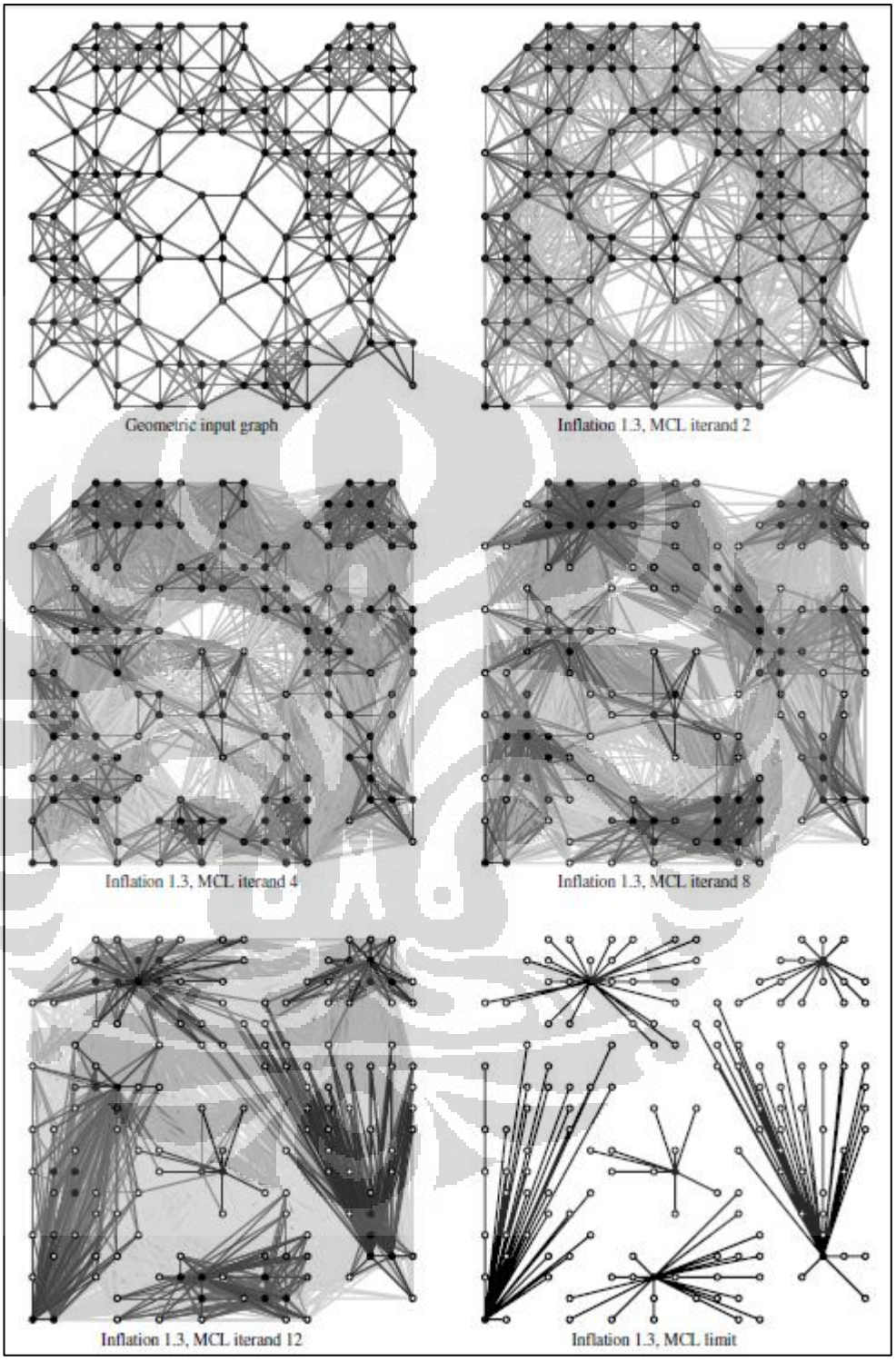
- ✓ *Expand*, mensimulasikan *random walk* (*flow*) pada graf G. Input berupa matriks Markov M yang sesuai dengan graf G dan output berupa matriks M_{exp} dimana $M_{exp} = Expand\ (M) = M^p$, $p \in \mathbb{Z}^+$ adalah parameter ekspansi. Karena M merupakan matriks *adjacency* dari graf G, maka bentuk M^p menyatakan probabilitas transisi dari sebuah *node* pada G ke *node* lainnya yang berjarak p dari *node* tersebut.

- ✓ *Inflate*, memperkuat *flow* yang kuat, memperlemah *flow* yang lemah. Diberikan matriks $M \in R^{mxn}$, $M \geq 0$, $r \in \mathbb{R}$, $r > 0$, maka operator inflasi Γ_r: $\mathbb{R}^{mxn} \rightarrow \mathbb{R}^{mxn}$ didefinisikan sebagai

  $$(\Gamma_r M)_{ij} = \frac{(M_{ij})^r}{\Sigma_{k=1}^{m}(M_{kj})^r}\ ,\ i = 1 \ldots m,\ j = 1 \ldots n \qquad (2.1)$$

  Nilai r antara 0 dan 1 akan meningkatkan kehomogenan (keseragaman) dari matriks M, sedangkan nilai r antara 1 dan ∞ akan meningkatkan ketakseragaman dari M, dimana entri dengan nilai yang kecil akan semakin diperkecil, sedangkan entri dengan nilai yang besar akan semakin diperbesar. Nilai r yang negatif tidak digunakan sebab akan mengubah urutan dari entri-entri pada M, dimana entri bernilai kecil akan menjadi besar, dan sebaliknya. Hal ini akan berakibat pada rusaknya topologi graf, dimana daerah-daerah yang seharusnya padat dan memiliki satu atau lebih pusat *cluster* akan diubah menjadi daerah dengan densitas yang rendah.
- ✓ *Prune*, penghapusan entri dari M dengan nilai yang dianggap cukup kecil pada setiap kolom, yaitu entri yang nilainya kurang dari *minval*, *default minval* adalah 10^{-5}. Dengan melakukan proses *prune*, hubungan antar-*node* dengan busur yang memiliki probabilitas yang rendah akan dipotong sehingga hanya tersisa *node-node* dengan tingkat ketertarikan (probabilitas) yang kuat.

Dengan melakukan operasi ekspansi dan inflasi secara bergantian, maka *edge* (busur, menyatakan probabilitas) yang menghubungkan dua buah *node* yang terletak pada *dense region* yang berbeda akan memiliki nilai probabilitas yang semakin kecil (Dongen, 2000). Hal ini sesuai dengan tujuan klusterisasi jaringan dimana daerah-daerah yang padat pada graf G akan terlepas satu sama lainnya, sehingga bisa meminimalisir terbentuknya pusat *cluster* yang menghubungkan dua daerah yang memiliki tingkat ketertarikan yang rendah. Proses tambahan yaitu *prune* digunakan untuk mempercepat konvergensi dari MCL, dimana entri dengan nilai yang sangat kecil (relatif terhadap $minval$ yang dipilih) akan dijadikan nol, sehingga klusterisasi yang dilakukan menjadi lebih fokus kepada entri dengan nilai yang lebih besar. Ilustrasi dari algoritma MCL diberikan pada Gambar 2.5.



Gambar 2. 5 *Graph Clustering* dengan MCL

[Sumber: *Graph Clustering via a Discrete Uncoupling Process*, Dongen, 2008]

Menurut Bustamam *et al.* (2010), tiap iterasi yang memuat proses ekspansi dan inflasi akan menghasilkan matriks idempoten, dimana iterasi akan berhenti saat dicapai kondisi idempoten dimana *global chaos* yang diperoleh kurang dari *threshold* minimum *e*, *default* $e = 10^{-3}$.

$$(chaos)_k = \max((\Gamma_r M)_{ik}, i = 1 \dots m) - \sum_{i=1}^{m} (\Gamma_r M)_{ik}^2 \quad (2.2)$$

$$glb\_chaos = \max((chaos)_k,\ k = 1 \dots n) \quad (2.3)$$

Global chaos menyatakan tingkat perubahan nilai dari M yang dihasilkan pada sebuah iterasi MCL dibandingkan dengan iterasi sebelumnya, sehingga iterasi MCL akan dihentikan jika tidak terdapat perubahan yang signifikan. Algoritma MCL diberikan pada Tabel 2.1.

Tabel 2. 1 Algoritma MCL

Input: matriks *adjacency A* dari graf G

Step 1: tambahkan *self-loop* pada graf, $A = A + I$

Step 2: bentuk matriks Markov M dengan menormalisasi kolom dari matriks A, yaitu $M = AD^{-1}$ dengan *D* adalah *diagonal degree matrix* dari A

Step 3: ulangi step 4 sampai step 6

Step 4: M = Ekspansi (M) := M^p dengan parameter ekspansi $p \in \mathbb{Z}$, default $p = 2$

Step 5: M = Inflasi (M, r)

Step 6: M = Prune (M), $M_{ij} = 0$ if $M_{ij} < minval$

hingga *global chaos* matriks M < *threshold e*

Output: Matriks *clustering M*

2.2.2 Algoritma *Regularized Markov Clustering* (R-MCL)

R-MCL merupakan algoritma modifikasi MCL yang dikembangkan oleh Satuluri dan Parthasarathy (SPU, 2009). Algoritma R-MCL serupa dengan MCL, hanya proses *expand* diganti dengan proses *regularize*, $Regularize(M) = M * M_G$, dimana M_G adalah matriks M awal. Dengan adanya proses ini, topologi

orisinal dari graf awal tetap mempengaruhi proses *clustering* untuk setiap iterasi, tidak hanya pada iterasi pertama.

Tabel 2. 2 Algoritma R-MCL

Input: matriks *adjacency* A dari graf G
Step 1: tambahkan *self-loop* pada graf, $A = A + I$
Step 2: bentuk matriks Markov M dan M_G dengan menormalisasi kolom dari matriks A, yaitu $M = M_G = AD^{-1}$ dengan D adalah *diagonal degree matrix* dari A
Step 3: ulangi step 4 sampai step 6
Step 4: M = *Regularize* (M) := $M * M_G$ dengan parameter ekspansi $p \in \mathbb{Z}$, default $p = 2$
Step 5: M = Inflasi (M, r)
Step 6: M = *Prune* (M), hapus entri dari M yang nilainya dianggap terlalu kecil, $M_{ij} = 0$ if $M_{ij} < minval$
hingga *global chaos* matriks M < *threshold e*
Output: Matriks *clustering M*

2.3 Komputasi Paralel

Komputer telah menjadi sahabat manusia sejak tahun 1940-an. Komputer membantu manusia dalam banyak hal, seperti di bidang komputasi sains, ekonomi, bisnis, hingga sektor hiburan. Pada mulanya para ahli berusaha untuk membuat komputer dengan ukuran yang lebih kecil, kemudian dengan kemampuan yang lebih bervariasi, hingga peningkatan *processing power*. Kebutuhan manusia di bidang informatika semakin lama semakin banyak dan semakin kompleks sehingga perlu dilakukan peningkatan performa dari prosesor komputer. Peningkatan kinerja ini awalnya dilakukan dengan melakukan peningkata frekuensi prosesor. Namun, ada batasan peningkatan yang tidak bisa dilewati. Hukum Amdahl menyatakan bahwa jika F merupakan bagian dari kalkulasi komputer yang berjalan secara sekuensial dan $(1 - F)$ merupakan bagian yang berjalan secara paralel, maka *speedup* maksimum yang bisa dicapai

dengan menggunakan N prosesor adalah $\frac{1}{F+\frac{1-F}{N}}$ (Mozdzynski, 2011). Berdasarkan hukum ini, akan tercapai suatu kondisi dimana peningkatan frekuensi prosesor tidak akan memberikan peningkatan kinerja yang signifikan. Akibatnya, dilakukan cara lain yaitu dengan menggandakan otak komputer sehingga berkembanglah teknik komputasi paralel. Implementasi komputasi paralel pertama kali dilakukan pada tahun 1960. Hingga saat ini, untuk komputer *mainstream* telah banyak beredar di pasaran prosesor dengan empat, enam, hingga delapan *processing cores*, sedangkan jumlah prosesor yang dipakai untuk super komputer telah mencapai 700 ribu inti (top500.org).

Klasifikasi arsitektur komputer paralel yang paling sering digunakan adalah taksonomi Flynn. Pengkategorian yang dilakukan oleh Michael Flynn didasarkan pada perbedaan jumlah *instruction stream* dan *data stream* (Zumkley, 2010). Pada Tabel 2.3 diberikan gambaran umum mengenai empat kategori komputasi paralel yang diajukan Flynn.

Tabel 2. 3 Taksonomi Flynn

SISD	*Single instruction stream, single data stream*	Serial, hanya terdapat 1 instruksi dan 1 data stream yang dieksekusi pada sekali *clock cycle*
SIMD	*Single instruction stream, multiple data stream*	Semua processing unit mengeksekusi sebuah instruksi yang sama namun dengan data yang berbeda
MISD	*Multiple instruction stream, single data stream*	Beberapa instruksi dilakukan pada sebuah data tunggal
MIMD	*Multiple instruction stream, multiple data stream*	Eksekusi beberapa instruksi yang berbeda pada *data stream* yang berbeda

2.4 MPI (*Message-Passing Interface*)

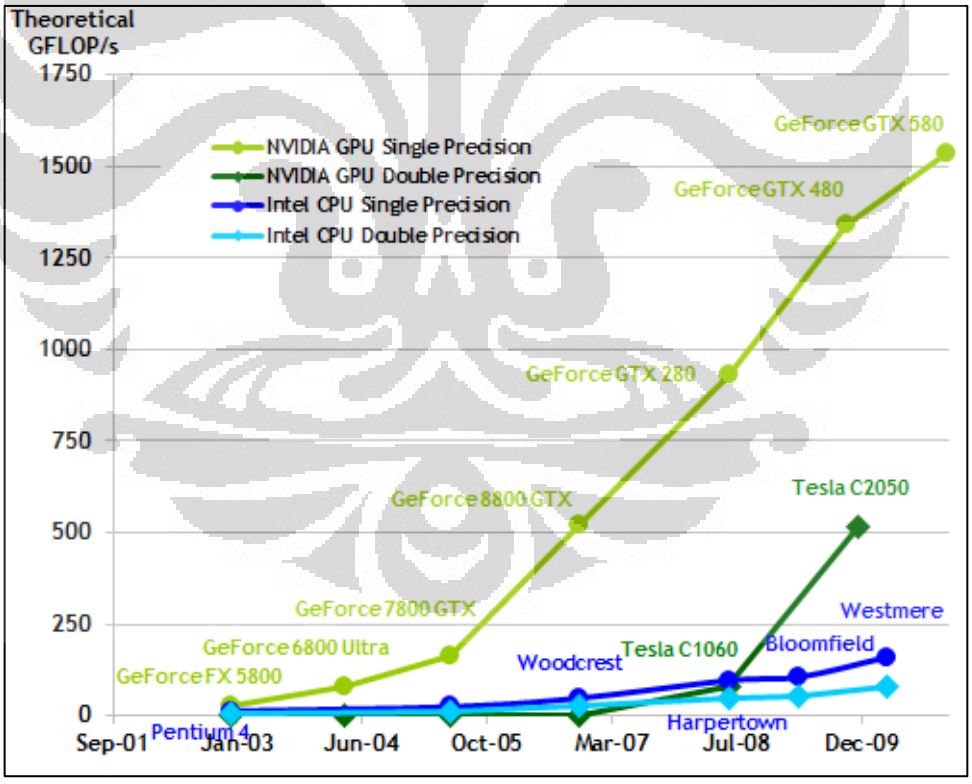
Peningkatan jumlah inti pada sebuah prosesor ternyata masih terhambat oleh suatu kendala lain. Berdasarkan hukum Moore, peningkatan jumlah inti prosesor menjadi dua kali lipat dari generasi sebelumnya cenderung membutuhkan waktu sekitar dua tahun (Intel, 2005). Hal ini sama sekali tidak berimbang dengan perkembangan permintaan pasar yang semakin kompleks dan beragam. Akibatnya , dilakukan metode lain yaitu dengan mengelompokkan sebuah komputer *master* dengan beberapa komputer *worker* ke dalam suatu jaringan atau lebih dikenal dengan sebutan *cluster computing*. *Interface* yang digunakan adalah MPI (*Message-Passing Interface*) yang merupakan standar untuk mengekspresikan paralelisme yang terdistribusi melalui *message passing* yang dapat dilakukan dengan menggunakan bahasa C, C++ dan Fortran (Gray *et al.*, 2007).

MPI terdiri dari *header file*, *library of routines* dan *runtime environment*. MPI merupakan API (*Application Programming Interface*) yang bertugas menentukan tampilan setiap *routine* dan bagaimana *routine* tersebut harus bertindak, namun tidak bertugas untuk menentukan bagaimana setiap *routine* harus diimplementasikan. MPI menggunakan sistem paralel SPMD (*Single Program Multiple Data*) dimana sebuah program MPI dieksekusi oleh semua proses yang terjadi dalam sekali *run*. Semua *worker* memulai eksekusi pada saat yang bersamaan pada bagian programnya masing-masing, dimana bagian-bagian tersebut saling independen satu sama lain. Namun, terdapat kendala yang cukup memberatkan pada koputasi jenis ini, yaitu implementasi MPI yang sulit, tidak fleksibel (tidak *scalable*), mobilitas yang rendah serta butuh *resource* besar (Gray *et al.*, 2007). Untuk itu, diperlukan suatu solusi baru yang bisa digunakan secara efektif dan efisien untuk menyelesaikan permasalahan komputasi yang semakin rumit.

2.5 GPU *Computing*

Di saat perkembangan teknologi *multicore* pada CPU terhambat hukum Moore serta adanya kendala pada implementasi MPI, di sisi lain perkembangan

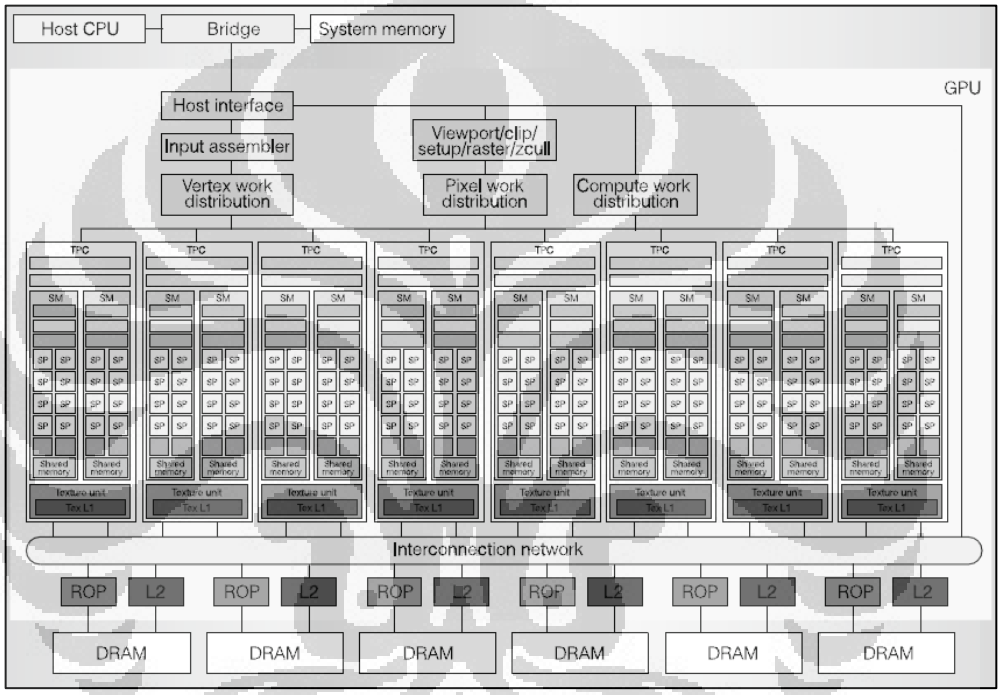
kartu grafis atau lebih dikenal sebagai GPU (*Graphics Processing Unit*) dengan teknologi *manycore* berkembang dengan sangat pesat. Sadar akan hal ini, NVIDIA sebagai salah satu pemimpin pasar kartu grafis termotivasi untuk mengembangkan sebuah *platform* yang bisa digunakan untuk memaksimalkan penggunaan GPU, bukan hanya dari sisi *grapics processing*, namun juga untuk kepentingan komputasi pada umumnya. *Platform* ini diberi nama GPGPU (*General Purpose Computation on* GPU). Namun, terdapat kendala API, sehingga pada awal tahun 2007 NVIDIA mengeluarkan sebuah model pemrograman paralel baru yang disebut CUDA (*Compute Unified Device Architecture*) yang memanfaatkan GPU sebagai perangkat komputasinya. Sejak saat itu, pemrograman paralel beralih ke era baru yang sangat menjanjikan, dimana *speedup* yang dihasilkan dari berbagai implementasi pada sejumlah algoritma mencapai puluhan hingga ratusan kali (NVIDIA CUDA *Zone*, 2012).



Gambar 2. 6 Perbandingan *Floating-Point Operations per Second* antara CPU dan GPU

[Sumber: CUDA C *Programming Guide* 4.0, NVIDIA]

CUDA merupakan ekstensi minimal dari bahasa pemrograman C dan C++. CUDA menyediakan tiga abstraksi utama, yaitu hierarki dari *thread*, *shared memory*, dan *barrier synchronization*. Tujuan dari pengembangan CUDA adalah untuk menuntun *programmer* membagi masalah ke dalam submasalah yang masih kasar yang bisa diselesaikan secara independen dan paralel, kemudian membaginya menjadi bagian-bagian yang lebih halus yang bisa diselesaikan secara kooperatif dalam ranah paralelisme (Lindholm *et al.*, 2008).

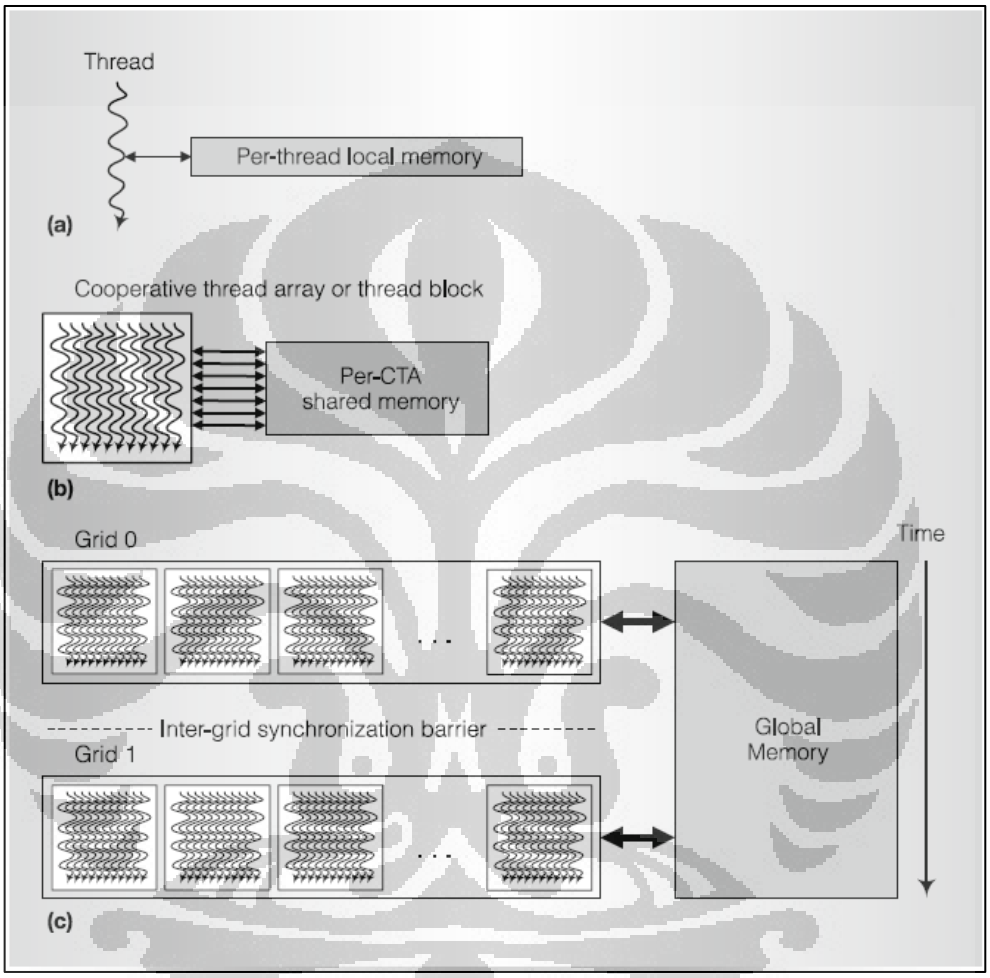


Gambar 2. 7 Arsitektur Tesla

[Sumber: *Tesla: A Unified Graphics and Computing Architecture,* Lindholm *et al.,* 2008]

Untuk menerapkan model pemrograman CUDA, diperlukan sebuah arsitektur komputasi yang bisa menjadikan prosesor-prosesor pada GPU menjadi satu *platform* yang efisien untuk mengolah grafis dan aplikasi-aplikasi komputasi paralel yang lebih umum. Sadar akan hal ini, NVIDIA mengembangkan arsitektur generasi pertama yang disebut Tesla. Arsitektur ini berupa *scalable array* dan *multithreaded* SMs (*streaming multiprocessors*). GPU saat ini memiliki sekitar 768 hingga 12.288 *thread* yang bisa dieksekusi secara bersamaan. Tiap TPC

(*Texture / Processor Cluster*) mengandung 1 *Geometry Controller*, 1 SM *Controller* (SMC), 2 SM dan 1 *texture unit*. Sebuah SM terdiri dari 8 *scalar* SP (*streaming processor*) *cores*, 2 SFU (*special function unit*), sebuah MT IU (*multithreaded instruction unit*) dan *shared memory* (16 kb).



Gambar 2. 8 Hierarki pada Arsitektur Tesla

[Sumber: *Tesla: A Unified Graphics and Computing Architecture*, Lindholm *et al.*, 2008]

SM membuat, mengatur, dan mengeksekusi hingga 768 *thread* secara bersamaan. *Geometry Controller* berfungsi mengatur *input* dan *output* dalam *chip* serta mendistribusikan data sesuai kebutuhan, SM *Controller* berfungsi mengatur penggunaan memori eksternal, *Texture Unit* berfungsi sebagai unit pengeksekusi ketiga, MT IU berperan dalam pengaturan pembagian kerja antar-*thread*, 24

SIMD dan *independent* MIMD, SP bertugas dalam mengeksekusi program, dan SFU berperan dalam mengatur fungsi transenden dan atribut interpolasi (Lindholm *et al.*, 2008).

Di dalam sebuah GPU terdapat berbagai jenis memori dengan ukuran dan waktu akses (*latency*) yang berbeda-beda. *Global memory* merupakan memori dengan ukuran paling besar dan waktu akses 200 - 600 *cycle* yang dipakai bersama oleh semua *thread* pada semua SM. *Constant memory* memiliki fungsi yang sama dengan *global memory* namun dengan ukuran yang lebih kecil dan kecepatan yang lebih besar. Setiap bagian dari *global memory* dapat dibentuk menjadi *texture memory* dengan fungsi yang lebih spesifik. *Shared memory* merupakan memori lokal yang dipakai bersama oleh semua *thread* yang terletak dalam *block* yang sama. *Constant cache* terdapat pada setiap SM, bersifat *read-only*, dipakai bersama oleh semua *thread* dalam sebuah *block*, dan bergantung pada *constant memory*. *Texture cache* mirip dengan *constant cache* namun dengan latensi yang lebih besar. *Register* merupakan memori yang dimiliki oleh masing-masing *thread*. Memori ini memiliki kecepatan yang sangat tinggi dengan latensi sekitar 1 *cycle*. *Local memory* merupakan *off-chip read/write memory* yang dimiliki oleh setiap *thread*. Memori ini terletak di *global memory* dan latensinya sama dengan latensi *global memory*. *Local memory* akan digunakan oleh *thread* jika *register* tidak mampu menampung struktur data yang ada (Wafai, 2009).

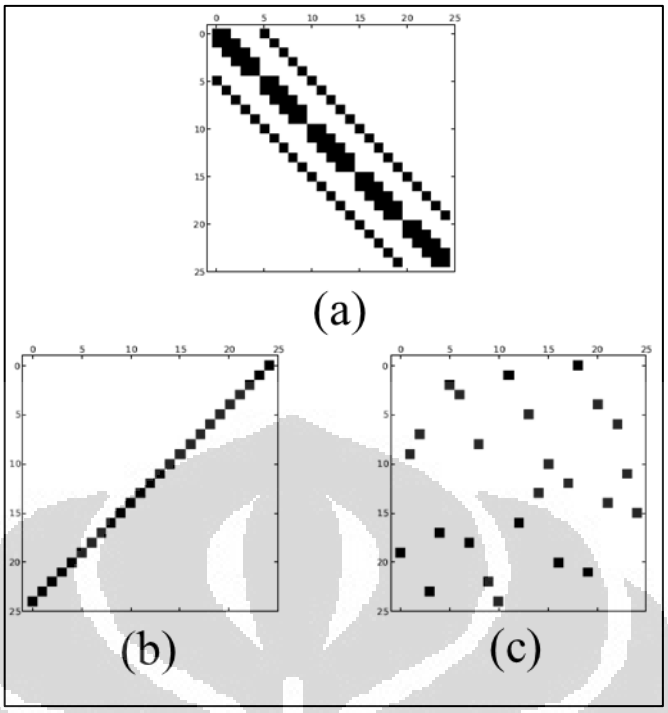
Berbeda dengan MPI yang menggunakan SPMD, untuk mengatur ratusan *thread* yang berjalan pada beberapa program yang berlainan, SM pada Tesla menggunakan arsitektur baru yang disebut SIMT (*Single-Intsruction Multiple-Thread*). SM memetakan setiap *thread* ke sebuah SP *scalar core*, dan setiap *scalar thread* berjalan secara independen sesuai dengan alamat instruksi dan *register state* masing-masing *thread*. Arsitektur SIMT mirip dengan SIMD *vector organization* dalam hal sebuah instruksi tunggal mengontrol banyak *processing elements*. Perbedaanya adalah pada SIMD *software* bisa mengetahui lebar dari SIMD (SIMD *width*), sedangkan pada SIMT hal ini tidak terjadi sebab instruksi-instruksi dari SM melakukan eksekusi dan penyebaran seolah-olah seperti pada *thread* tunggal. Selain itu, SIMD juga membutuhkan *software* untuk menggabungkan muatan-muatan ke dalam menjadi vektor dan mengatur

penyebarannya secara manual, sedangkan SIMT tidak demikian (Lindholm *et al.*, 2008).

2.6 Format Penyimpanan Data *Sparse*

Sparse matrix merupakan matriks dengan persentase elemen tak-nol yang rendah. Entri matriks *sparse* didominasi oleh nol. Matriks jenis ini sering muncul dalam berbagai masalah nyata seperti pada bidang *image processing*, pemodelan ekonomi, teknik elektro, dinamika fluida, pemodelan elemen hingga, desain sirkuit elektronik, fisika plasma dan masalah lainnya di bidang komputasi saintifik. Jika matriks *sparse* disimpan dalam memori komputer layaknya menyimpan matriks padat, akan terjadi redundansi memori yang disebabkan oleh banyaknya entri nol yang disimpan. Untuk itu, diperlukan tipe penyimpanan khusus untuk matriks *sparse* sehingga penggunaan memori bisa optimal. Terdapat beberapa format penyimpanan data berupa matriks *sparse*, diantaranya format diagonal, COO, CSR, CSC, JAD, *Hybrid*, ELLPACK, dan ELLPACK-R (Wafai, 2009).

Format diagonal dibentuk dengan menggunakan dua buah *array*, yaitu *data* dan *offsets*. *Array data* berisi nilai-nilai bukan nol, sedangkan *array offsets* berisi *offset* tiap diagonal yang dihitung dari diagonal utama dimana *offset* 0 diberikan untuk diagonal utama, *offset* > 0 untuk diagonal di atas diagonal utama, dan *offset* < 0 untuk diagonal dibawahnya. Keuntungan dari format data ini adalah indeks dari entri bukan nol dinyatakan secara implisit oleh posisinya pada kedua buah *array* serta semua akses memori ke *array* data berdekatan sehingga dapat meningkatkan efisiensi transaksi memori. Namun, format ini tidak efisien untuk menyelesaikan *matrix-vector multiplication* dengan pola *sparse* yang tersebar serta untuk matriks sparse dengan data yang terpusat di diagonal yang bersilangan dengan diagonal utama.



Gambar 2. 9 (a) Diagonal Format, Matriks *Sparse* dengan (b) Diagonal Terbalik, (c) Data Tersebar

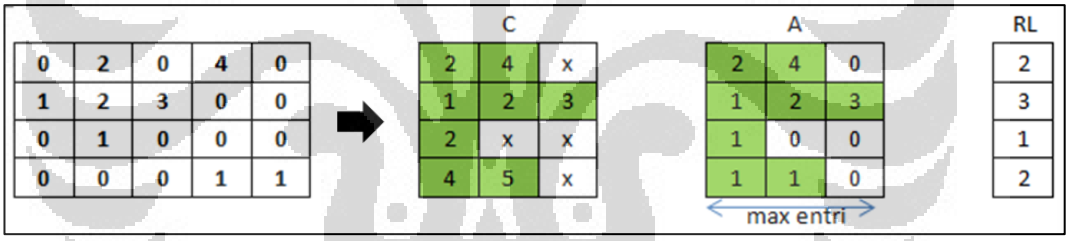
Coordinate Storage Format (COO) menggunakan tiga buah *array* untuk menyimpan data *sparse*. *Array* A[](*float*) berisi entri tak-nol dan *array* C[](*integer*) dan R[](*integer*) masing-masing berisi indeks kolom dan baris yang bersesuaian dengan data pada *array* A.

Format penyimpanan *Compressed Sparse Row* (CSR) merupakan salah satu format penyimpanan data *sparse* yang paling sering digunakan. Terdapat tiga buah *array* yang digunakan pada format CSR yaitu A[], C[] dan R_pt[]. Data yang disimpan di dalam *array* A dan C sama adalah entri tak-nol dan indeks kolom, sedangkan R_pt menyimpan *pointer* yang menunjukkan posisi elemen pertama pada setiap baris. Format penyimpanan *Compressed Sparse Column* (SCS) memiliki struktur yang serupa dengan CSR namun *array* C dan R_pt diganti dengan R dan C_pt.

Pada format penyimpanan *Jagged Diagonal* (JAD) terdapat dua tahap yang dilakukan. Tahap pertama sama dengan CSR, sedangkan pada tahap kedua dilakukan permutasi baris dimana baris-baris dengan elemen yang lebih banyak ditempatkan di bagian atas dari matriks.

Berbeda dengan format-format penyimpanan data *sparse* di atas, format penyimpanan ELLPACK merupakan format non-generik. Hal ini disebabkan oleh memori yang dipakai bervariasi bergantung pada variasi panjang baris maksimum dari entri tak-nol. Tahap awal konstruksi ELLPACK serupa dengan CSR dimana entri tak-nol ditumpuk di sebelah kiri matriks dan entri nol di sebelah kanan. Setelah itu, semua entri nol yang terletak setelah entri tak-nol paling kanan dipotong. Matriks ini kemudian diubah menjadi dua buah *array* A dan C.

ELLPACK-R merupakan format penyimpanan matriks *sparse* pengembangan dari format ELLPACK yang bertujuan untuk meningkatkan performa dari ELLPACK pada komputasi menggunakan GPU. Format ELLPACK-R terdiri dari dua buah matriks A[](*float*) dan C[](*integer*) berdimensi N x ME yang kemudian akan diubah menjadi *array*, serta tambahan sebuah *array integer* RL[] berdimensi N, dengan N = banyak baris dari matriks asal, ME = maksimum dari banyak entri tak-nol dari setiap baris.



Gambar 2. 10 ELLPACK-R

A berisi entri dari matriks awal, dimana seluruh entri tak-nol dipadatkan ke kiri, C berisi indeks kolom dari setiap entri yang bersesuaian di A, dan RL berisi banyak entri tak-nol pada setiap baris.

BAB 3
ALGORITMA PARALEL R-MCL

Pada bab ini akan dibahas algoritma paralel *Regularized Markov Clustering* pada GPU untuk klusterisasi jaringan interaksi protein-protein. Secara garis besar, algoritma ini terdiri dari tahap konversi matriks *sparse* menjadi ELLPACK-R, ekspansi, inflasi, *prune*, dan tahap pencarian *global chaos* sebagai *stop condition* bagi R-MCL.

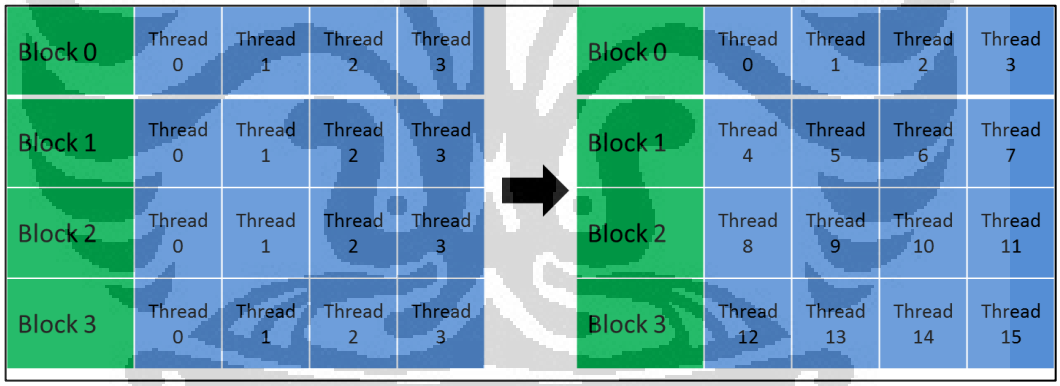
3.1 Pengindeksan Menggunakan Konstanta Baku pada GPU

Arsitektur GPU memberikan kemudahan dengan adanya hierarki *thread, block*, dan *grid*. Setiap *thread*, *block*, dan *grid* pada GPU memiliki beberapa konstanta *default* seperti $threadIdx$, $blockIdx$, $blockDim$, dan $gridDim$. Konstanta $threadIdx$ menyatakan index 3-dimensi dari sebuah thread di dalam *block* yang terdiri dari $threadIdx.x$, $threadIdx.y$, dan $threadIdx.z$. Dengan menggunakan konstanta-konstanta baku ini, dapat dilakukan berbagai pengindeksan manual terhadap setiap *thread* dan *block* sehingga diperoleh struktur data yang efisien dan efektif dalam proses komputasi. Selain memberikan keuntungan pada sisi fleksibilitas pembentukan struktur data, ketersediaan konstanta baku pada GPU juga berakibat pada skalabilitas yang baik pada komputasi paralel GPGPU. Skalabilitas disini berarti sebuah program paralel yang dieksekusi pada sebuah GPU dapat dengan mudah dijalankan pada GPU lain tanpa harus melakukan pengubahan program, struktur data, ukuran input, dan alokasi memori. Lain halnya dengan MPI, implementasi pada mesin MPI yang berbeda mengharuskan pengguna untuk mengubah program sesuai dengan arsitektur mesin yang akan digunakan.

Pada penelitian ini hanya digunakan pengindeksan 1-dimensi dengan pertimbangan bahwa pengindeksan tipe ini lebih cepat dibandingkan dengan tipe lain dengan dimensi yang lebih tinggi. Secara *default* pemberian indeks pada setiap *thread* dalam sebuah *block* dimulai dari 0 hingga $N - 1$, dengan N adalah

jumlah *thread* dalam satu *block*, N kelipatan 32, dimana 32 menyatakan jumlah *thread* yang bekerja sama secara *synchronous* dalam sebuah *warp*. *Synchronous* berarti seluruh *thread* dalam *warp* yang sama akan memulai pekerjaan secara serentak, lalu menunggu semua *thread* selesai bekerja untuk dapat mengeksekusi perintah baru. N disebut juga sebagai $blockDim$, dalam hal ini $blockDim.x$ untuk pengindeksan 1-dimensi. Sama halnya dengan *thread*, *block* juga memiliki pengindeksan *default* untuk setiap *block* yang terdapat dalam sebuah *grid* yang disebut $blockIdx$. Jumlah *block* dalam sebuah *grid* disebut $gridDim$, dimana jumlah *block* ini tidak harus merupakan kelipatan 32.

Salah satu pengindeksan yang paling sering digunakan adalah $i = blockDim.x * blockIdx.x + threadIdx.x$. Pengindeksan tersebut menghasilkan nilai i yang berturutan pada setiap *thread* pada suatu *block* dengan *block* berikutnya dengan *offset* (lompatan dalam pengindeksan) yang sama dengan banyak *thread* dalam satu *block* ($blockDim.x$).

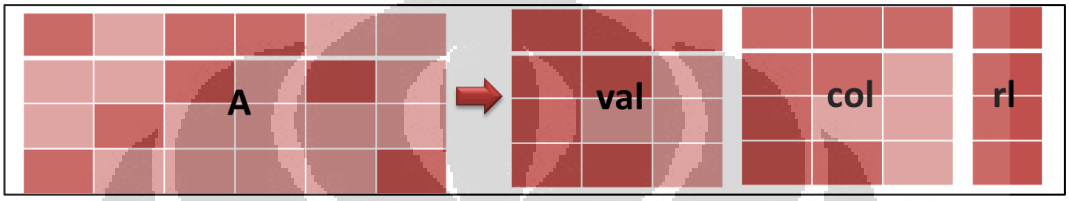


Gambar 3. 1 Pengindeksan $i = blockDim.x * blockIdx.x + threadIdx.x$

3.2 Algoritma Paralel untuk Konversi Matriks *Sparse* Menjadi Format ELLPACK-R

Untuk melakukan proses *clustering*, hal pertama yang harus dilakukan adalah mengubah input berupa matriks *adjacency* dari graf interaksi protein-protein menjadi format ELLPACK-R. Seperti telah dijelaskan pada bab sebelumnya, ELLPACK-R merupakan format yang sesuai untuk komputasi

menggunakan GPU. Misalkan A adalah matriks *adjacency* berukuran N dari graf jaringan interaksi protein-protein G, dengan $N = n \times n$, n adalah banyak protein dalam sebuah jaringan interaksi. Matriks *sparse* A terlebih dahulu ditambahakan dengan *self loop* yaitu dengan melakukan operasi $M = A + I$, dengan I adalah matriks identitas berukuran $n \times n$. Matriks ini kemudian akan diubah menjadi bentuk ELLPACK-R seperti pada Gmabar 3.2. Proses konversi M menjadi bentuk ELLPACK-R secara paralel diberikan pada Tabel 3.1.



Gambar 3. 2 Perubahan Bentuk Matriks *Sparse* Menjadi ELLPACK-R

Tabel 3. 1 Algoritma Paralel Konversi Mariks *Sparse* Menjadi ELLPACK-R

Input: matriks *sparse* $M_{n\times n}$ berupa *array* berukuran N

Step 1: inisialisasi *array* kosong $val[]$, $col[]$, dan $rl[]$

Step 2: set $count = 0$, $max\_col = 0$, $shift = 1$, $i = blockDim.x * blockIdx.x + threadIdx.x$, dan $tid = threadIdx.x$

Step 3: inisialisasi *array* pada *shared memory*, $sdata[blockDim.x] = 0$

Step 4: set $k = n * (blockIdx.x - (blockIdx.x/n) * n) + blockIdx.x/n * blockDim.x + tid$

Step 5: jika $M[k] \neq 0$, set $sdata[tid] = 1$

Step 6: lakukan *parallel reduction* tipe 7 untuk *array* $sdata[]$

Step 7: $rl[blockIdx.x] = sdata[0]$

Step 8: $max\_col = \max(rl)$

Step 9: jika $i < n$, lakukan step 10

Step 10: untuk $j = 0$ hingga $j = n - 1$, lakukan step 11

Step 11: jika $M[i * n + j] \neq 0$, lakukan step 12 hingga step 14

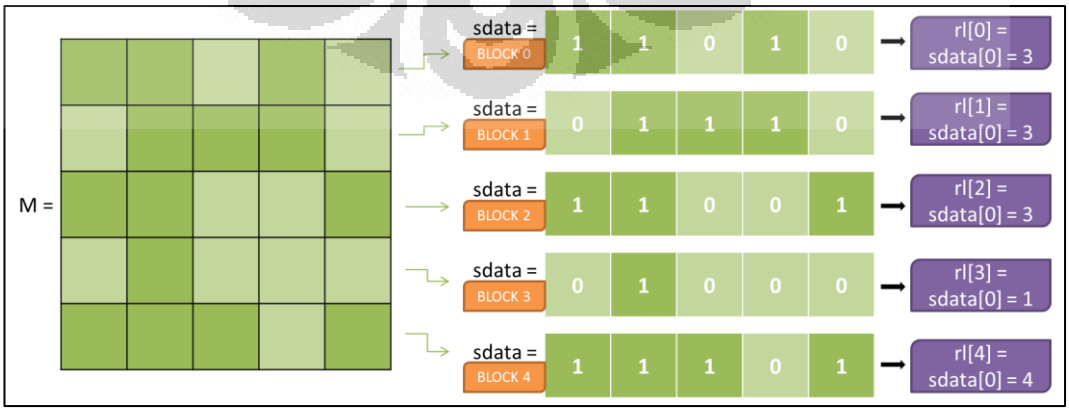
Step 12: $count += 1$

Step 13: $val[i * max\_col + count] = M[i * n + j]$

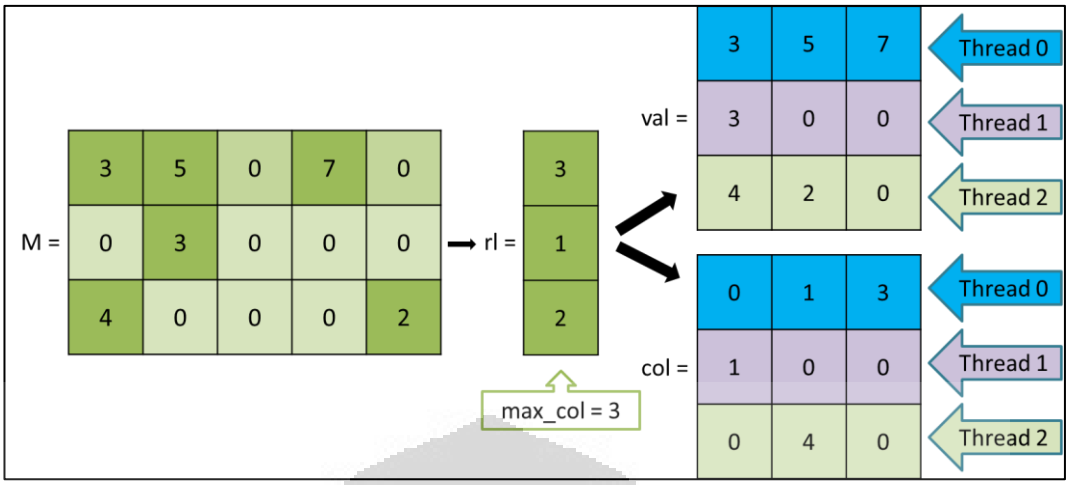
Step 14: $col[i * max\_col + count] = j$

Output: *array* $val[]$, $col[]$, dan $rl[]$ yang merupakan representasi matriks *sparse M*

Ide dari algoritma paralel ini adalah sebagai berikut. Pencarian panjang baris untuk *array* $rl[]$ pada *step* 3 saling bergantung dengan operasi pengisian entri pada *array* $val[]$ dan $col[]$ pada *step* 7, sehingga kedua operasi ini perlu dijalankan secara terpisah. Namun, pada masing-masing proses tidak terdapat *data dependency*. Akibatnya, kedua proses ini dapat dijalankan secara paralel sehingga hanya akan terdapat sebuah *looping*, yaitu untuk mengisi entri tak-nol ke *array* $val[]$ dan indeks kolom dari entri yang bersesuaian ke dalam *array* $col[]$. Untuk mengisi jumlah entri tak-nol per baris pada *array* $rl[]$, hal pertama yang perlu dilakukan adalah inisialisasi variabel $sdata[]$ pada *shared memory* (step 3) yang akan digunakan untuk menyimpan informasi mengenai banyaknya entri tak-nol pada setiap baris dimana setiap *block* akan mengeksekusi sebuah baris dari matriks M (step 4). Entri-entri dari $sdata[]$ yang semula bernilai nol akan diubah menjadi satu jika nilai dari entri yang bersesuaian pada matriks M merupakan entri tak-nol (step 5). Setelah proses ini selesai, maka akan dilakukan *parallel reduction* tipe 7 (Harris, 2008) dimana semua elemen pada setiap $sdata[]$ pada setiap *block* akan dijumlahkan sehingga setiap $sdata[0]$ akan menghasilkan nilai yang menyatakan banyaknya entri tak-nol dari baris yang bersesuaian pada M (step 6 & 7).



Gambar 3. 3 Tahap Pembentukan *Array* $rl[]$



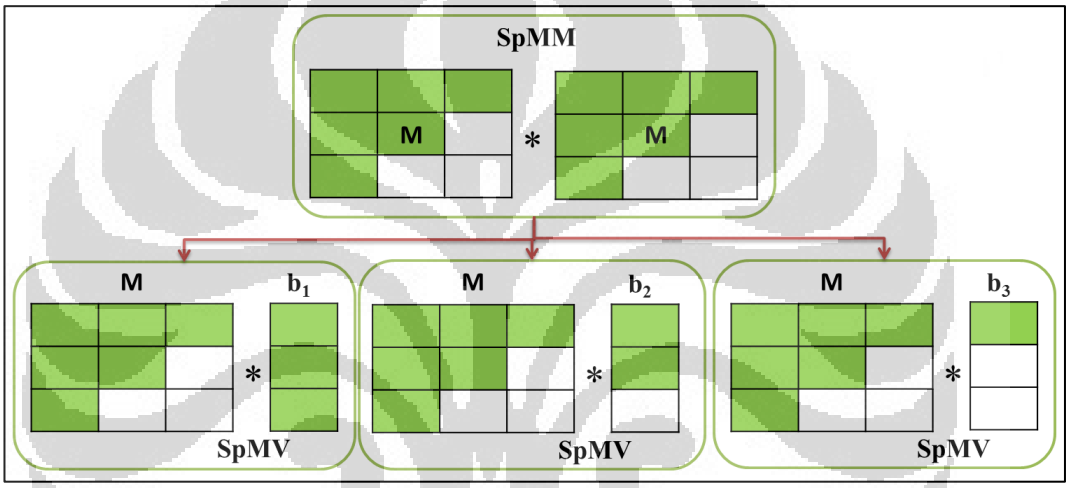
Gambar 3. 4 Tahap Pembentukan *Array* $\boldsymbol{val}$[] dan $\boldsymbol{col}$[]

Pada step 8 terdapat operasi $max\_col = \max(rl)$. Pencarian ini bisa dilakukan dengan memanfaatkan *shared memory*, dimana seluruh entri pada kolom yang sama akan dimasukkan ke dalam sebuah *block*, lalu pencarian nilai maksimum dari setiap *block* akan dilakukan dengan menggunakan *parallel reduction* yang telah dimodifikasi untuk menyelesaikan permasalahan yang menggunakan operator logika. Penjelasan yang lebih mendalam mengenai modifikasi *parallel reduction* terdapat dalam Subbab 3.6 mengenai pencarian *global chaos*.

Tahap berikutnya merupakan tahap pengisian entri tak-nol ke dalam *array* val[] dan indeks kolom dari entri-entri tersebut ke dalam col[]. Tahap ini hanya menggunakan n buah *thread*, yaitu *thread* 0 hingga *thread* $(n-1)$, dimana masing-masing *thread* akan melakukan sebuah *looping* untuk mencari entri tak-nol pada M (step 11). Ketika sebuah entri tak-nol ditemukan, maka entri tersebut akan dimasukkan ke dalam val[] (step 13), kemudian indeks kolom dari entri tersebut akan dimasukkan ke dalam col[] (step 14). Jika dijalankan secara sekuensial, tahap ini memiliki *time complexity* $O(n^2)$. Namun, pada algoritma paralel di atas, *time complexity* ini dapat dereduksi menjadi $O(n)$ karena tugas pengisian pada setiap baris saling independen sehingga dapat dilakukan secara paralel oleh masing-masing *thread*.

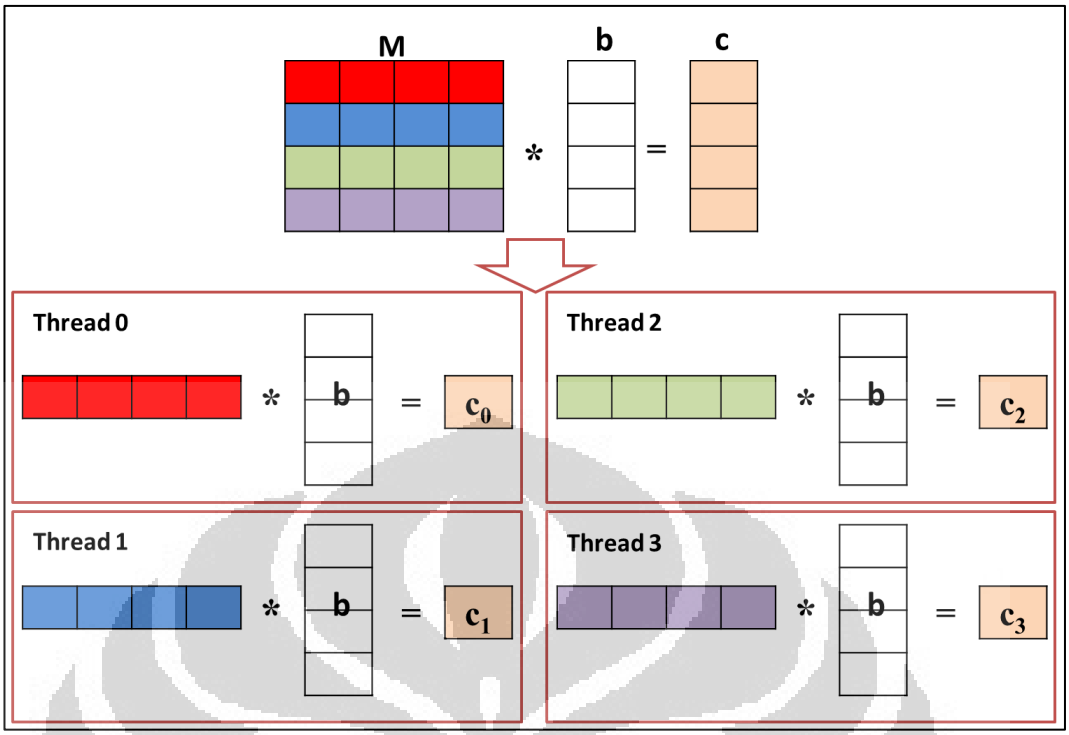
3.3 Algoritma Paralel untuk Proses Ekspansi

Ekspansi didefinisikan sebagai perkalian matriks M dengan dirinya sendiri sebanyak p kali, dengan p merupakan faktor inflasi, $p \in \mathbb{N}$. M merupakan matriks Markov berbentuk *sparse*, sehingga bentuk M^p merupakan bentuk perkalian matriks *sparse* dengan matriks *sparse*. *Sparse Matrix-Matrix Multiplication* (SpMM) merupakan inti dari tahap ekspansi. SpMM dapat dipecah menjadi n buah *Sparse Matrix-Vector Multiplication* (SpMV) yang saling independen sehingga setiap SpMV dapat dijalankan secara paralel.



Gambar 3. 5 Pemecahan SpMM Menjadi SpMV

Pada tahap sebelumnya, M telah diubah menjadi format ELLPACK-R, yaitu ke dalam tiga buah *array* $val[]$, $col[]$, dan $rl[]$. Pada tahap ekspansi, akan dilakukan n buah perkalian M dengan vektor b_i menghasilkan vektor c_i, dimana b_i adalah kolom ke-i dari matriks M, dengan $i = 0, 1, \dots, n-1$. Pada Gambar 3.5 diberikan contoh pemecahan sebuah SpMM untuk matriks berukuran 3×3 menjadi 3 buah SpMV. SpMV pertama adalah perkalian matriks M dengan vektor b_1 yang merupakan kolom pertama dari M, SpMV kedua merupakan perkalian M dengan b_2, dan SpMV ketiga merupakan perkalian M dengan b_3. Pada proses SpMV, setiap *thread* akan mengerjakan sebuah perkalian antara baris ke-i dari matriks M dengan vektor kolom b_i menghasilkan elemen ke-i dari vektor kolom c_i (Gambar 3.6).



Gambar 3. 6 Pembagian Tugas untuk Tiap *Thread* pada SpMV

Berikut merupakan algoritma paralel SpMV pada R-MCL.

Tabel 3. 2 Algoritma Paralel SpMV pada Proses Ekspansi

Input: *array* $val[N]$, $col[N]$, $rl[n]$, dan $b[n]$

Step 1: Inisialisasi *array* kosong $a[n]$

Step 2: set $j = blockIdx.x * blockDim.x + threadIdx.x$

Step 3: untuk $i = 0$ hingga $i = rl[j] - 1$, lakukan step 4 hingga 6

Step 4: int $val\_id = n * j + i$

Step 5: int $col\_b = col\ [val\_id]$

Step 6: $a[i] += val\ [val\_id] * b\ [col\_b]$

Output: *array* $a[n]$ hasil perkalian matriks M (dalam bentuk $val[]$, $col[]$, dan $rl[]$) dengan vektor b

Setiap *thread* akan memiliki konstanta tid masing-masing dan akan mengeksekusi *step* 4 hingga *step* 6 sebanyak $rl[tid]$ kali untuk memperoleh nilai dari a, dengan $rl[tid] \leq n$ menyatakan banyak elemen tak-nol pada baris yang

dieksekusi oleh sebuah *thread*. *Looping* tersebut tidak dilakukan sebanyak n kali sebab terdapat $n - rl[tid]$ entri nol pada yang notabene tidak akan mempengaruhi hasil perkalian baris ke-i dengan b. Hal ini akan berakibat pada semakin kecilnya *running time* dari proses perkalian SpMV.

SpMV akan dilakukan sebanyak n kali, dengan n adalah banyak kolom pada matriks $val[]$. Setiap $a[]$ yang diperoleh dari setiap SpMV kemudian digabung sehingga membentuk matriks baru yang merepresentasikan keseluruhan proses perkalian matriks *sparse* dengan matriks *sparse* (SpMM). Matriks baru hasil SpMM ini kemudian akan disimpan sebagai *array* $a[]$ yang baru untuk kemudian digunakan pada proses inflasi.

3.4 Algoritma Paralel untuk Proses Inflasi

Seperti telah dijelaskan pada bab sebelumnya, proses inflasi merupakan proses dimana *flow* yang kuat antara satu protein dengan protein lainnya akan semakin diperkuat sedangkan *flow* yang lemah akan semakin diperlemah. Hal ini dilakukan dengan memangkatkan setiap elemen dari M dengan faktor inflasi $r \in \mathbb{R}^+$, kemudian menormalisasi M dengan setiap elemen M dengan total elemen di kolom yang bersesuaian. Dengan melakukan normalisasi kolom, diperoleh matriks M yang baru dengan setiap elemen $M_{i,j}$ menyatakan probabilitas *node* ke-i tertarik ke arah *node* ke-j pada graf G, dengan $i, j = 1,2, \dots, n$. Jumlah setiap kolom j setelah dinormalisasi akan menjadi satu, dimana satu menyatakan probabilitas total setiap *node* tertarik ke arah *node* j. Algoritma paralel untuk proses inflasi pada R-MCL diberikan pada Tabel 3.3.

Tabel 3. 3 Algoritma Paralel Inflasi

Input: array $a[]$ yang diperoleh dari proses ekspansi
Step 1: inisialisasi *array* kosong $c[]$, $d[]$ dan $h[]$
Step 2: inisialisasi *array* kosong pada *shared memory* $sdata[blockDim.x]$
Step 3: set $j = blockIdx.x * blockDim.x + threadIdx.x$, $tid = threadIdx.x$

Step 4: lakukan operasi pemangkatan $c[j] = pow(a[j], r)$

Step 5: setiap elemen dari $c[]$ yang terletak pada kolom yang sama dimasukkan ke dalam *block* yang sama $sdata\ [tid] = c[n * (n\ /\ blockDim.x) * tid + blockIdx.x]$

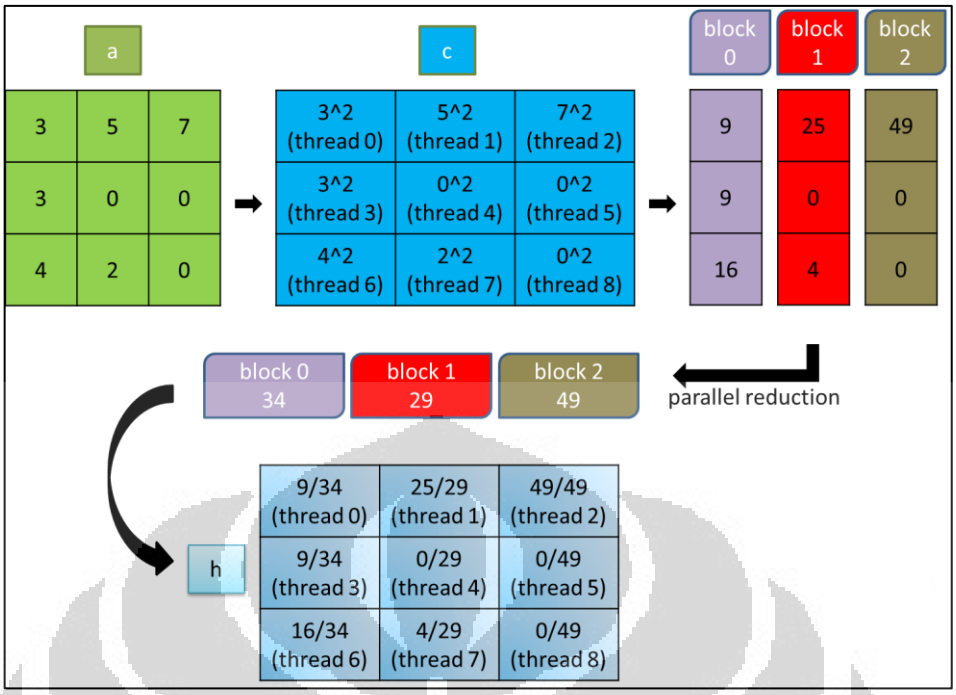
Step 6: lakukan *parallel reduction* tipe 7 untuk menjumlahkan setiap elemen pada *block* yang sama

Step 7: set $div = j\ mod\ n$, yaitu dengan mendefinisikan $div = j - (j/\ n) * n$

Step 8: bagi setiap elemen dengan total kolomnya $h[j] = c[j]\ /\ d[div]$

Output: $h[]$ yang merupakan representasi *array* dari matriks Markov yang baru

Ide dari algoritma paralel ini adalah sebagai berikut. Untuk proses pemangkatan, operasi pada setiap entri matriks M saling independen satu sama lain, sehingga operasi ini bisa dilakukan secara paralel oleh sejumlah *thread* yang tersedia. Pengindeksan manual yang perlu dilakukan pada tahap ini hanyalah pengindeksan standar, yaitu $j = blockIdx.x * blockDim.x + threadIdx.x$, sehingga setiap *thread* akan mengeksekusi pemangkatan satu buah elemen M yang bersesuaian dengan indeks j yang dimiliki oleh masing-masing *thread* (step 4). Tahap kedua yang perlu dilakukan adalah mencari total pada setiap kolom dari matriks M yang tiap elemennya telah dipangkatkan dengan r. Hal ini dapat dilakukan secara paralel dengan pertama-tama menyalin setiap elemen M yang berada pada kolom yang sama ke dalam satu *block* (step 5), kemudian melakukan *parallel reduction* untuk menjumlahkan elemen-elemen dalam *block* tersebut sehingga diperoleh sebuah nilai yang menyatakan jumlah kolom ke-k dari matriks M (step 6). Karena proses ini berlangsung di dalam *block* yang sama untuk setiap kolom, maka digunakan *shared memory* yang memiliki *latency* yang lebih kecil dibandingkan dengan *global memory*. Hal ini akan berakibat pada semakin kecilnya *running time* yang dibutuhkan setiap *block* untuk menyelesaikan operasi penjumlahan kolom.



Gambar 3. 7 Inflasi

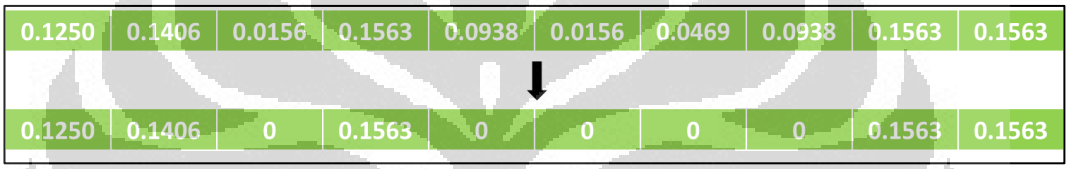
3.5 Algoritma Paralel untuk Proses *Prune*

Setelah dilakukan proses konversi ELLPACK-R hingga inflasi, tahap selanjutnya dari algoritma R-MCL adalah *prune*. Pada tahap ini akan dilakukan sebuah proses pemotongan dimana setiap elemen dari M yang nilainya kurang dari sebuah $minval$ akan diubah menjadi nol. Proses ini tidak melibatkan operasi-operasi khusus yang membutuhkan pengindeksan rumit dari *thread* serta penentuan dan distribusi data ke *shared memory*. Layaknya operasi pemangkatan pada tahap inflasi, proses pemotongan pada tahap ini bisa dilakukan secara paralel oleh sejumlah *thread* yang tersedia. Algoritma paralel untuk proses *prune* pada R-MCL diberikan pada Tabel 3.4.

Tabel 3. 4 Algoritma Paralel *Prune*

Input: *array* $h[N]$ yang diperoleh dari operasi inflasi
Step 1: set $j = blockIdx.x * blockDim.x + threadIdx.x$
Step 2: jika $j < N$ dan $h[j] < minval$, set $h[j] = 0$
Output: *array* $h[N]$ hasil pemotongan (*prune*)

Pada Algoritma 3.4, setiap entri $h[]$ akan dieksekusi oleh sebuah *thread*, sehingga hanya digunakan pengindeksan sederhana $j = blockIdx.x * blockDim.x + threadIdx.x$. Jika nilai dari sebuah entri pada *array* $h[]$ kurang dari $minval$, maka entri tersebut akan diubah menjadi nol (step 2). Bentuk $j < N$ menyatakan bahwa *thread* yang bekerja hanyalah *thread* dengan indeks kurang dari N. Hal ini akan berakibat pada penghematan penggunaan *thread*. Jika terjadi kasus dimana banyak elemen pada *array* yang dieksekusi lebih besar daripada banyak *thread* yang dipesan, maka dapat ditambahkan sebuah *while condition* untuk mengubah indeks j menjadi $j + blockDim.x * gridDim.x$, dimana $blockDim.x * gridDim.x$ menyatakan jumlah *thread* yang dipesan.



Gambar 3. 8 Prune dengan $\boldsymbol{minval = 0.1}$

3.6 Algoritma Paralel untuk Pencarian *Global Chaos*

Tiap iterasi dari proses ekspansi dan inflasi menghasilkan sebuah matriks idempoten dengan kluster-kluster di dalamnya. Matriks idempoten ini akan dipilih jika tidak lagi terdapat perubahan yang signifikan dari entri matriks yang dihasilkan dibandingkan dengan entri matriks pada iterasi sebelumnya. Kondisi numerik ini akan dicapai jika *global chaos* pada persamaan (2.3) memiliki nilai yang lebih kecil daripada *threshold* e, secara *default* $e = 10^{-3}$. Konstruksi algoritma paralel untuk pencarian *global chaos* mirip dengan inflasi sebab terdapat operasi pemangkatan setiap elemen matriks serta penjumlahan entri-entri setiap kolom matriks. Berikut merupakan algoritma paralel untuk pencarian *global chaos* pada R-MCL.

Tabel 3. 5 Algoritma Paralel untuk *Global Chaos*

Input: *array* $a[]$

Step 1: inisialisasi $glb\_chaos = 0$, array kosong $c[]$, $d[]$ dan $h[]$

Step 2: inisialisasi *array* kosong pada *shared memory* $sdata[blockDim.x]$

Step 3: set $j = blockIdx.x * blockDim.x + threadIdx.x$, $tid = threadIdx.x$

Step 4: lakukan operasi pemangkatan $c[j] = pow(a[j],2)$

Step 5: salin $c[]$ ke *shared memory*, $sdata\ [tid] = c[n * (n / blockDim.x) * tid + blockIdx.x]$

Step 6: lakukan *parallel reduction* tipe 7 untuk menjumlahkan setiap elemen pada *block* yang sama

Step 7: simpan hasil penjumlahan tiap *block*, $d[blockIdx.x] = sdata[0]$

Step 8: *reset* $sdata[] = 0$

Step 9: salin $a[]$ ke $sdata[]$, $sdata[tid] = a[n * (n / blockDim.x) * tid + blockIdx.x]$

Step 10: cari maksimum dari tiap *block* dengan *parallel reduction*, simpan ke dalam *array* $h[]$

Step 11: cari selisih $h[]$ dan $d[]$, $h[j] -= d[j]$, simpan nilainya kedalam $h[]$ (*overwrite*)

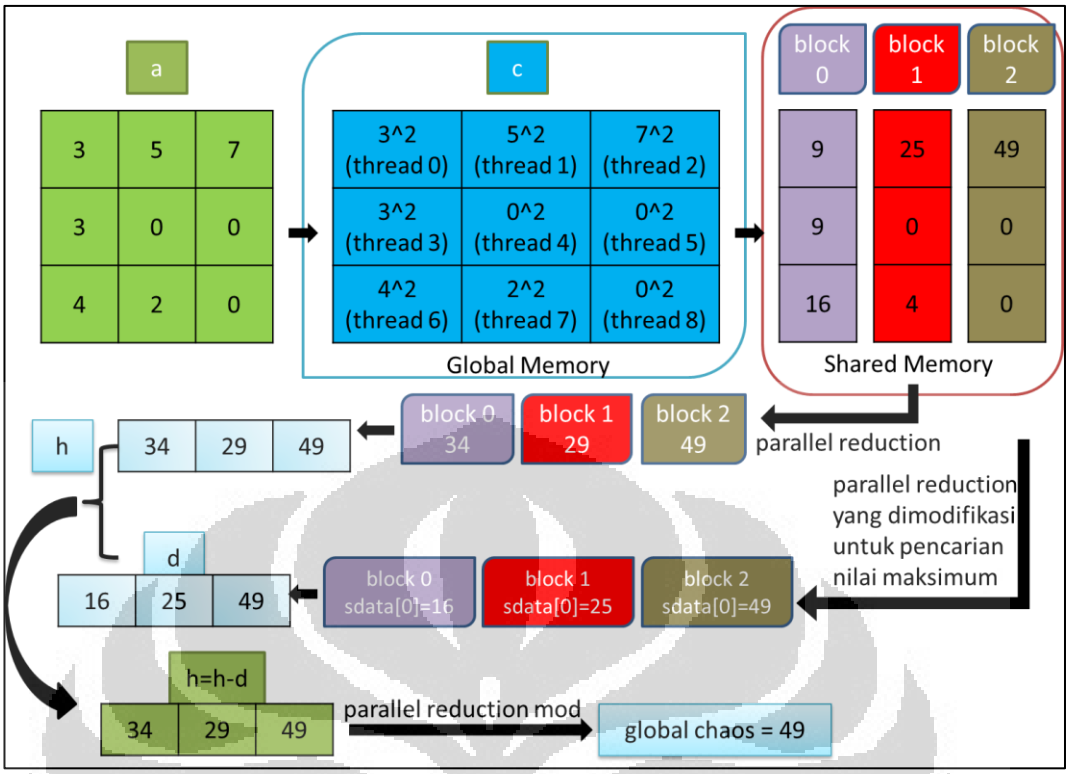
Step 12: cari maksimum $h[]$ dengan *parallel reduction*, $glb\_chaos = max\ (h)$

Output: glb_chaos yang kemudian akan digunakan sebagai *stop condition* dari algoritma R-MCL

Ide dari algoritma paralel ini adalah sebagai berikut. Operasi pertama pada algoritma paralel untuk pencarian *global chaos* merupakan operasi pemangkatan setiap elemen matriks (step 4). Seperti pada algoritma inflasi, operasi ini bisa dilakukan secara paralel oleh sejumlah *thread* yang tersedia dengan menggunakan pengindeksan $j = blockIdx.x * blockDim.x + threadIdx.x$, dimana setiap *thread* akan mengerjakan sebuah pemangkatan $pow(a[j], 2)$.

Tahap berikutnya yang harus dilakukan adalah mencari jumlah dari setiap kolom. *Array* $c[]$ disalin ke dalam *shared memory,* dimana satu *block* akan mengeksekusi satu kolom (step 5). Pada step 7 dilakukan *parallel reduction* untuk menjumlahkan elemen-elemen dari $c[]$ yang berada pada kolom yang sama. Jumlah dari setiap kolom disimpan di *thread* 0 dari setiap *block*, sehingga akan memudahkan proses penyalinan hasil ini ke *global memory* (step 7). Tahap selanjutnya adalah mencari maksimum dari setiap kolom pada $a[]$. Jika dijalankan secara sekuensial, proses pencarian nilai maksimum dapat dilakukan dengan membandingkan sebuah elemen dengan elemen lainnya sehingga diperoleh sebuah entri dengan nilai paling besar. Namun, dengan menggunakan GPU, hal ini bisa dicapai dengan menggunakan pendekatan yang mirip dengan *parallel reduction,* sehingga kompleksitas tahap ini bisa direduksi dari $O(N)$ menjadi $O(\log(N))$. Setiap kolom dari $a[]$ disalin ke sebuah *block* (step 9), kemudian dilakukan pendekatan *parallel reduction* untuk pencarian nilai maksimum dari masing-masing kolom. *Parallel reduction* yang semula bertujuan untuk menjumlahkan semua elemen pada *block* yang sama dapat dimodifikasi sehingga dapat digunakan untuk mencari nilai maksimum dari kolom yang bersesuaian dengan indeks dari setiap *block*. Modifikasi ini dilakukan dengan mengganti operasi aritmatika (dalam hal ini penjumlahan) dengan operasi logika. Misalkan terdapat m buah *thread* dalam sebuah *block*, maka *parallel reduction* dilakukan dengan membandingkan elemen matriks pada $\frac{m}{2}$ *thread* dengan $\frac{m}{2}$ *thread* berikutnya pada *block* yang sama, kemudian proses ini dilakukan secara rekursif hingga diperoleh masing-masing satu elemen terbesar pada setiap *block.*

Tahap berikutnya adalah mencari *chaos* untuk setiap kolom yaitu dengan mencari selisih dari elemen maksimum tiap kolom dengan jumlah kuadrat yang telah diperoleh pada proses sebelumnya (tahap 11). Tahap ini dilakukan di *global memory*. Tahap terakhir adalah mencari maksimum dari setiap *chaos* dengan menggunakan *parallel reduction* yang telah dimodifikasi (step 12). *Chaos* yang paling besar inilah yang akan digunakan sebagai *global chaos*, yaitu sebagai syarat berhentinya iterasi pada algoritma R-MCL.



Gambar 3. 9 Pencarian *Global Chaos*

Setelah satu iterasi dari R-MCL dijalankan, diperoleh sebuah matriks Markov baru yang masih berbentuk *sparse*. Jika *global chaos* masih lebih kecil daripada *threshold e*, maka iterasi berikutnya perlu dilakukan, sehingga matriks baru tersebut perlu kembali diubah menjadi bentuk ELLPACK-R. Hal ini tidak perlu dilakukan jika *threshold* telah dilampaui. Iterasi R-MCL akan terhenti sehingga matriks *sparse* hasil ekspansi, inflasi, dan *prune* bisa digunakan untuk penentuan kluster-kluster dari jaringan interaksi protein-protein.

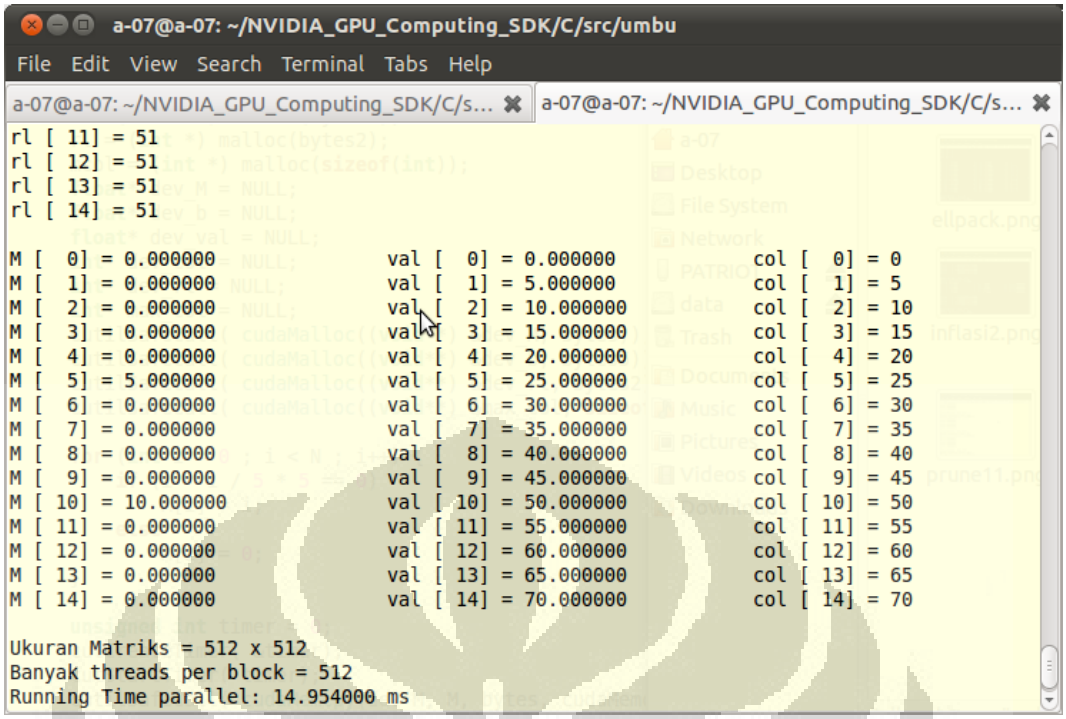
BAB 4
IMPLEMENTASI ALGORITMA PARALEL R-MCL PADA GPU

Pada Bab 3 telah dijelaskan mengenai algoritma paralel R-MCL untuk pemrograman GPGPU yang terdiri dari algoritma paralel untuk konversi matriks *sparse* menjadi ELLPACK-R, algoritma paralel untuk proses ekspansi, inflasi, *prune*, serta algoritma paralel untuk pencarian *global chaos*. Selanjutnya pada bab ini akan diberikan implementasi dan simulasi dari algoritma-algoritma tersebut dalam bentuk program. Program tersebut akan dijalankan pada komputer dengan spesifikasi sebagai berikut:

Prosesor : Intel® Dual Core™ E5500 @2.8 GHz dan Intel® Core™ i5 2410 @2.3 GHz
GPU : NVIDIA GTX 460 dengan 2 GB DRAM dan NVIDIA GT 540M dengan 1 GB DRAM
RAM : 2048MB
Sistem Operasi : Ubuntu 10.04 LTS 64-bit dan Ubuntu 12.04 LTS 32-bit
Compiler : NVIDIA CUDA Compiler (NVCC) v4.0 dan NVCC v.4.2

4.1 Implementasi Algoritma Paralel untuk Konversi Matriks *Sparse* Menjadi ELLPACK-R

Pada Subbab 3.2 telah dibangun sebuah algoritma paralel untuk konversi matriks *sparse* menjadi format ELLPACK-R. Program C pada Lampiran 1 dan kernel CUDA (bagian paralel yang dieksekusi secara *massive* pada *device*/GPU) pada Lampiran 2 merupakan implementasi dari Algoritma 3.1. Input dari program ini adalah sebuah *array* berukuran N yang merupakan representasi 1-dimensi dari matriks *sparse* M_{nxn}. Simulasi dilakukan dengan ukuran n dan banyak *threads per block* yang berbeda-beda, yaitu $n = 2^i$ dan $blockDim.x = 2^i$, $i = 5, 6, \dots, 10$. Gambar 4.1 menunjukkan contoh *screenshot* program konversi matriks *sparse* untuk $n = 512$ dan $blockDim.x = 512$.

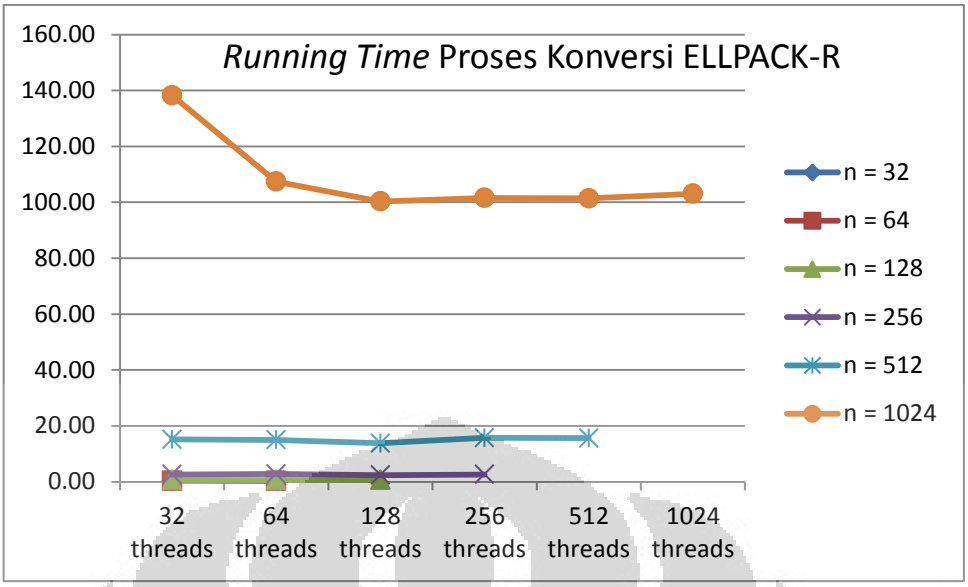


Gambar 4. 1 *Screenshot* ELLPACK-R untuk $n = 512$ dan $blockDim.x = 512$

Tabel 4.1 menunjukkan *running time* dari program konversi matriks *sparse* menjadi ELLPACK-R untuk n dan $blockDim.x$ yang berbeda-beda. Grafik *running time* program ini dapat dilihat pada Gambar 4.2.

Tabel 4. 1 *Running Time* Program Paralel ELLPACK-R

n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	0.282000					
64	0.377000	0.324000				
128	0.686000	0.651000	0.650000			
256	2.633000	2.757000	2.379000	2.602000		
512	15.177000	14.982000	13.789000	15.735001	15.636000	
1024	138.244995	107.412994	100.289001	101.546997	101.379997	103.063995

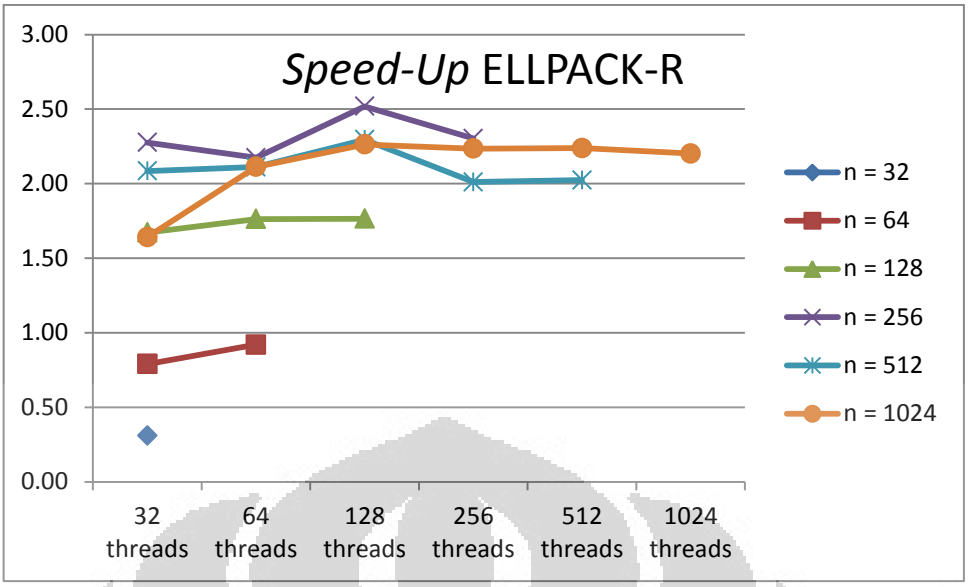


Gambar 4. 2 Grafik *Running Time* Algoritma Paralel untuk Konversi Matriks *Sparse* Menjadi Format ELLPACK-R

Speedup didefinisikan sebagai nilai perbandingan antara *running time* program serial (dengan menggunakan satu prosesor) dengan *running time* program paralel. Tabel 4.2 dan Gambar 4.3 berturut-turut merupakan *speedup* program paralel untuk konversi matriks *sparse* menjadi ELLPACK-R dan representasi grafisnya.

Tabel 4. 2 *Speedup* Program Paralel ELLPACK-R

n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	0.310273					
64	0.789962	0.919185				
128	1.672498	1.762417	1.765129			
256	2.275767	2.173411	2.518745	2.302880		
512	2.084861	2.111997	2.294723	2.010927	2.023659	
1024	1.641434	2.112594	2.262661	2.234630	2.238311	2.201739



Gambar 4. 3 Grafik *Speedup* ELLPACK-R

Dari hasil simulasi di atas, terlihat bahwa semakin kecil ukuran matriks, *running time* dari program ini cenderung semakin kecil. Waktu eksekusi paling singkat diperoleh untuk matriks berukuran 32×32, yaitu kurang dari 0.3 milisekon. Hal berikutnya yang dapat dilihat dari tabel dan grafik di atas adalah *running time* untuk banyak *threads per block* yang digunakan untuk eksekusi program. Terlihat bahwa untuk setiap nilai n, $blockDim.x = 128$ cenderung memberikan *running time* paling singkat. Jika dilihat dari sisi *speedup* yang dihasilkan, terlihat bahwa semakin besar ukuran matriks, *speedup* yang dihasilkan cenderung naik dan menuju titik kesetimbangan di 2.2 kali. *Speedup* terbaik diperoleh saat $blockDim.x = 256$. Layaknya *running time*, *speedup* program ini juga cenderung memberikan hasil yang baik pada penggunaan 128 *threads* untuk setiap *block*.

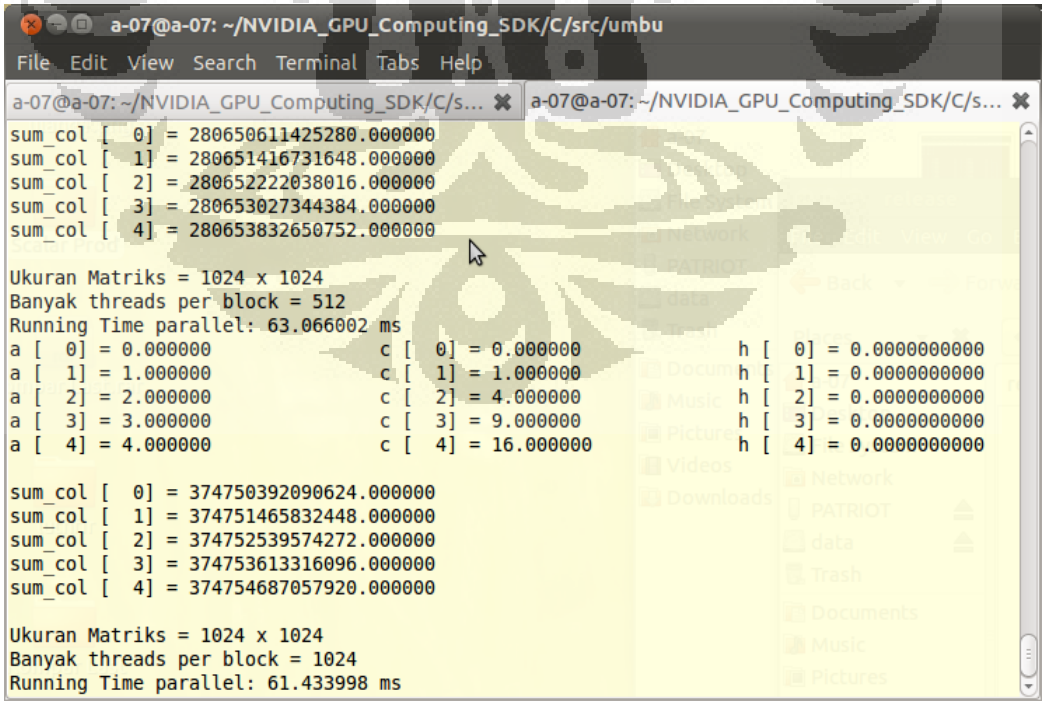
4.2 Implementasi Algoritma Paralel untuk Proses Ekspansi

Pada Subbab 3.3 telah dibangun algoritma paralel untuk proses ekspansi pada R-MCL yang dapat diimplementasikan pada pemrograman paralel menggunakan GPU. Lampiran 3 dan Lampiran 4 berturut-turut menyatakan program C dan kernel CUDA untuk Algoritma 3.2. Pada implementasi dengan

menggunakan mesin dengan spesifikasi yang telah disebut di atas, program ini tidak dapat berjalan dengan baik. Terdapat masalah alokasi memori yang mengakibatkan tidak disalinnya hasil ekspansi yang telah dihitung di GPU (dalam hal ini vektor *c* untuk setiap iterasi SpMV) ke *host* (CPU).

4.3 Implementasi Algoritma Paralel untuk Proses Inflasi

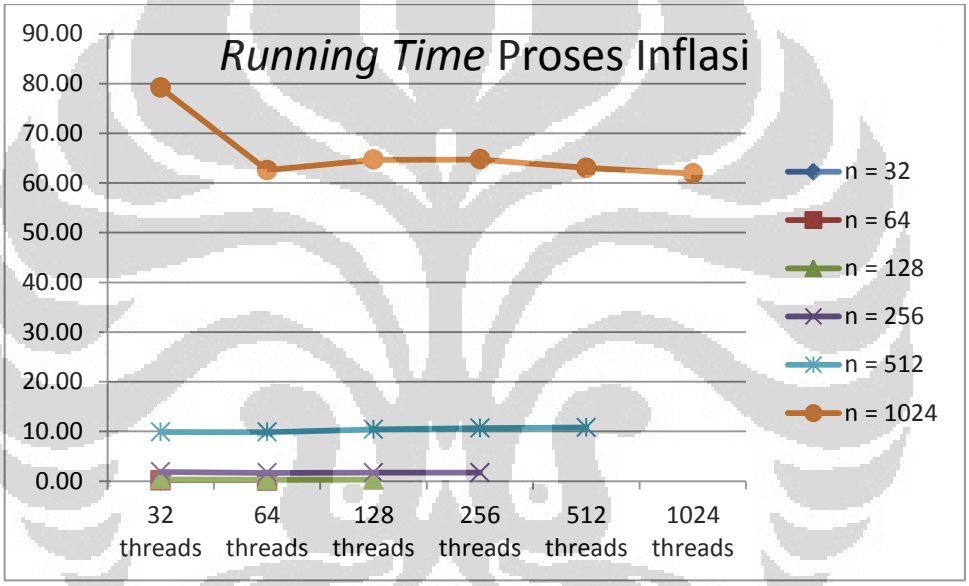
Pada Subbab 3.4 telah dikonstruksi sebuah algoritma paralel untuk menyelesaikan proses ketiga dari R-MCL, yaitu proses inflasi. Algoritma 3.3 kemudian diimplementasikan di GPU dengan menggunakan bahasa pemrograman CUDA C. Lampiran 5 merupakan program C untuk proses ekspansi, sedangkan Lampiran 6 berisi kernel CUDA untuk proses ini. Setelah kedua program ini dijalankan, diperoleh hasil seperti pada Gambar 4.4. *Running time* yang diperoleh untuk nilai n dan $blockDim.x$ yang bervariasi dapat dilihat pada Tabel 4.3, sedangkan representasi grafis dari *running time* program ini dapat dilihat pada Gambar 4.5, sedangkan *speedup* dari proses inflasi dan representasi grafisnya berturut-turut dapat dilihat pada Tabel 4.4 dan Gambar 4.6.



Gambar 4. 4 *Screenshot* Program Paralel Inflasi untuk $\boldsymbol{n = 1024}$ dan $\boldsymbol{blockDim.x = 1024}$

Tabel 4. 3 *Running Time* Program Paralel Inflasi

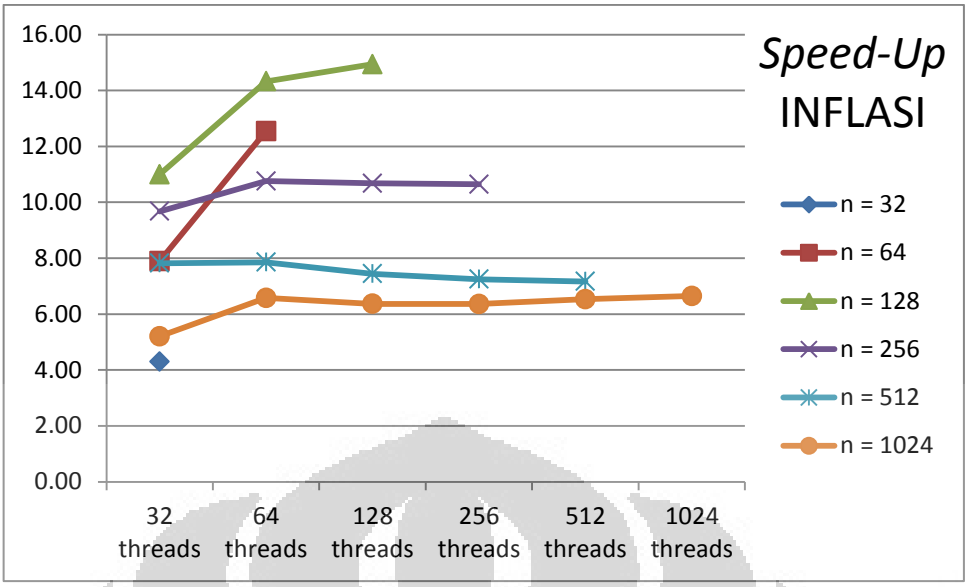
n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	0.086000					
64	0.143000	0.090000				
128	0.379000	0.291000	0.279000			
256	1.859000	1.672000	1.685000	1.690000		
512	9.871000	9.828000	10.377000	10.653000	10.773000	
1024	79.149002	62.556999	64.667000	64.721001	63.033001	61.910999



Gambar 4. 5 Grafik *Running Time* Algoritma Paralel untuk Proses Inflasi

Tabel 4. 4 *Speedup* Program Paralel Inflasi

n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	4.291860					
64	7.895035	12.544333				
128	10.997995	14.323849	14.939928			
256	9.677639	10.760006	10.676991	10.645402		
512	7.822079	7.856302	7.440661	7.247887	7.167153	
1024	5.200121	6.579350	6.364674	6.359364	6.529665	6.648001



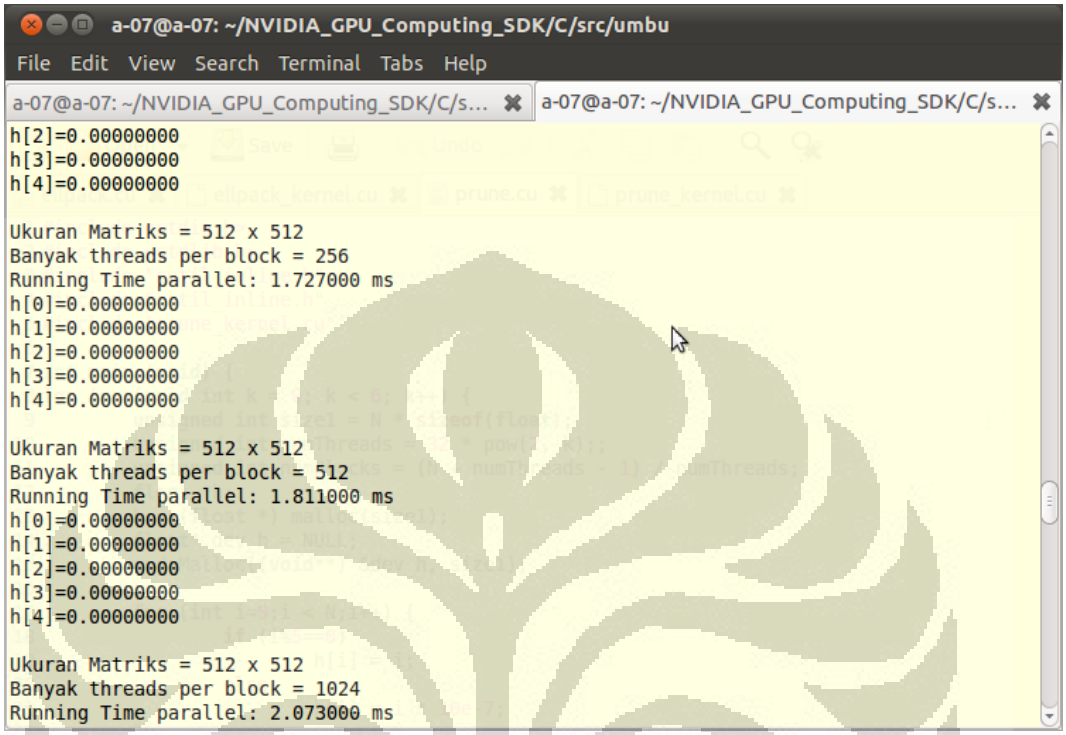
Gambar 4. 6 Grafik *Speedup* Proses Inflasi

Dari hasil simulasi di atas, terlihat bahwa semakin kecil ukuran matriks yang dieksekusi, *running time* yang diperoleh cenderung semakin kecil. Hal kedua yang bisa dievaluasi adalah dari sisi penggunaan banyak *threads per block*. Dari hasil simulasi di atas, terlihat bahwa penggunaan $blockDim.x = 64$ cenderung memberikan *running time* yang lebih kecil untuk proses ekspansi pada R-MCL. Jika diperhatikan dari sisi *speedup*, terlihat bahwa eksekusi matriks berukuran 128×128 memberikan *speedup* yang paling baik, hampir mencapai 15 kali kecepatan eksekusi program serial. Dari sisi penggunaan jumlah *thread* dalam setiap *block*, $blockDim.x = 64$ memberikan *speedup* yang cenderung lebih baik dibandingkan dengan $blockDim.x$ yang lain.

4.4 Implementasi Algoritma Paralel untuk Proses *Prune*

Pada Subbab 3.5 telah dibangun algoritma paralel untuk proses *prune* dari R-MCL. Implementasi dari Algoritma 3.4 terlampir pada Lampiran 7 dan Lampiran 8 yang berturut-turut berisi program C untuk *prune* dan kernel CUDA dari proses tersebut. Gambar 4.7 menunjukkan *screenshot* dari program ini ketika dieksekusi dengan menggunakan NVCC. Tabel 4.5 dan Gambar 4.8 berturut-turut

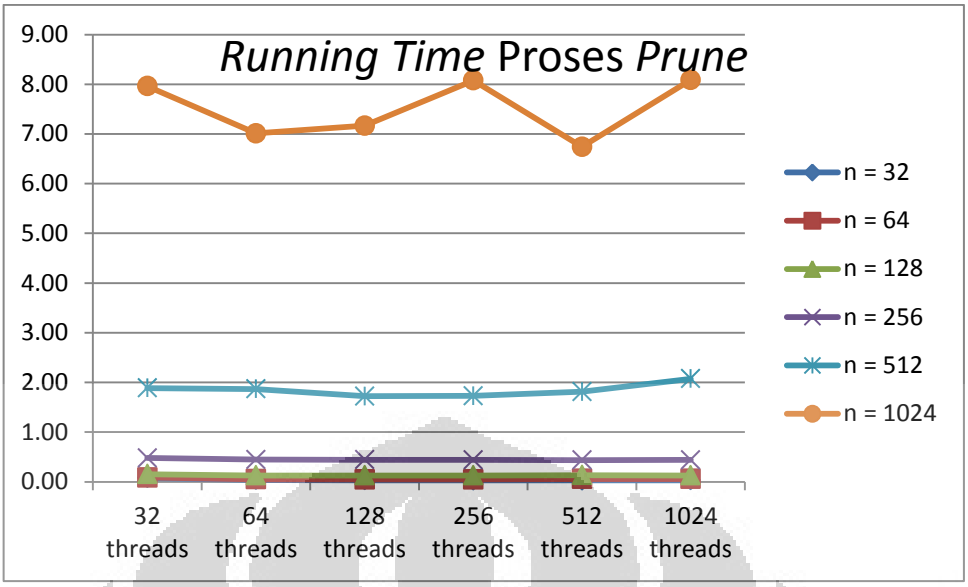
menunjukkan *running time* yang diperoleh dari simulasi proses *prune* dan representasi grafisnya.



Gambar 4. 7 *Screenshot* Program Paralel *Prune* untuk $n = 512$ dengan $blockDim.x = 256, 512, 1024$

Tabel 4. 5 *Running Time* Program Paralel *Prune*

n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	0.075000	0.038000	0.030000	0.026000	0.028000	0.032000
64	0.085000	0.049000	0.046000	0.046000	0.055000	0.056000
128	0.153000	0.128000	0.127000	0.126000	0.134000	0.126000
256	0.479000	0.447000	0.443000	0.439000	0.438000	0.439000
512	1.885000	1.868000	1.723000	1.727000	1.811000	2.073000
1024	7.961000	7.012000	7.165000	8.076000	6.741000	8.087000

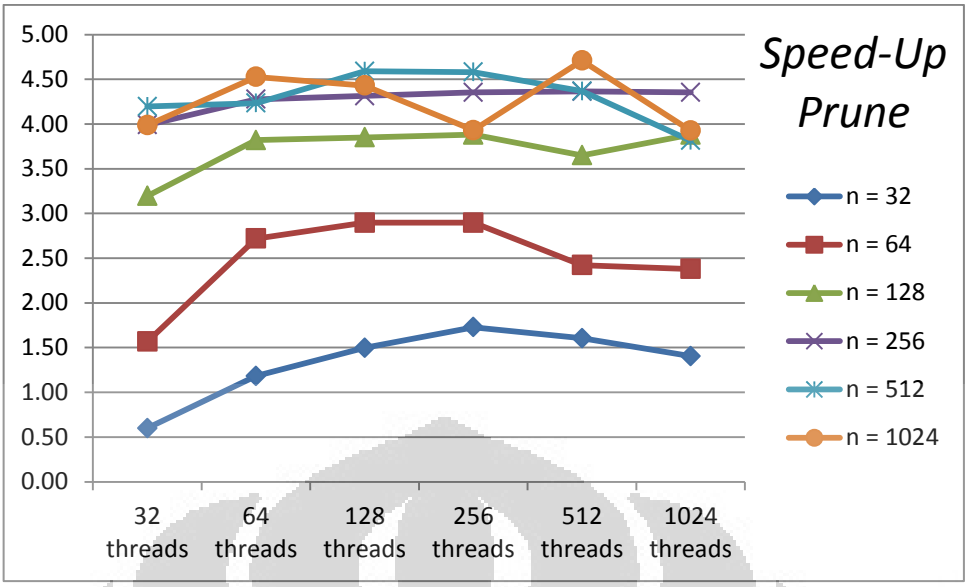


Gambar 4. 8 Grafik *Running Time* Algoritma Paralel untuk Proses *Prune*

Tabel 4.6 berisi *speedup* yang diperoleh setelah program paralel *prune* dieksekusi, sedangkan Gambar 4.9 merupakan representasi grafisnya.

Tabel 4. 6 *Speedup* Program Paralel *Prune*

n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	0.599067	1.182368	1.497667	1.728077	1.604643	1.404063
64	1.567529	2.719184	2.896522	2.896522	2.422545	2.379286
128	3.195948	3.820156	3.850236	3.880794	3.649104	3.880794
256	3.989729	4.275347	4.313950	4.353257	4.363196	4.353257
512	4.195761	4.233945	4.590255	4.579624	4.367206	3.815248
1024	3.987008	4.526607	4.429947	3.930234	4.708585	3.924888



Gambar 4. 9 Grafik *Speedup* Proses *Prune*

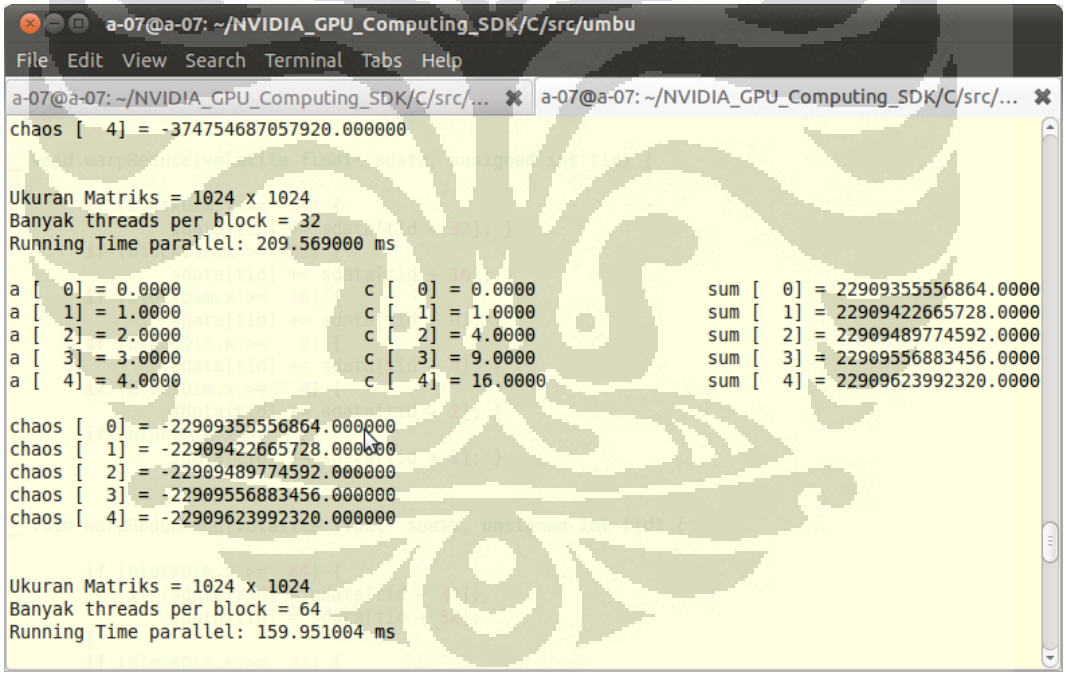
Evaluasi efektivitas dari program paralel *prune* dapat dilakukan dengan mengevaluasi *running time* berdasarkan ukuran matriks yang dieksekusi serta berdasarkan banyak *thread* yang digunakan dalam sebuah *block*. Dari simulasi di atas, semakin besar ukuran matriks, semakin besar pula waktu ekesekusi yang diperlukan. Jika dilihat dari sisi banyaknya *thread* yang digunakan, $blockDim.x = 256$ cenderung memberikan *running time* yang lebih kecil dibandingkan dengan penggunaan banyak *thread* lainnya. *Speedup* yang dihasilkan untuk ukuran matriks dengan $n = 256$, $n = 512$, dan $n = 1024$ memberikan hasil yang cenderung stabil di kisaran 4 kali, namun memberikan hasil yang kurang baik untuk matriks dengan n yang lebih kecil. Jika dilihat dari sisi banyak *thread* yang digunakan, penggunaan 256 *threads* untuk setiap *block* memberikan *speedup* yang lebih baik dibandingkan dengan $blockDim.x$ yang lain.

4.5 Implementasi Algoritma Paralel untuk Pencarian *Global Chaos*

Pada Subbab 3.6 telah dibangun sebuah algoritma paralel untuk mencari *global chaos* dari sebuah iterasi pada R-MCL. Algoritma ini kemudian diimplementasikan pada GPU dengan bahasa pemrograman CUDA C. Lampiran 9

merupakan *listing* program C untuk proses pencarian *global chaos*, sedangkan Lampiran 10 merupakan *listing* kernel CUDA yang digunakan sebagai program untuk memproses pencarian *global chaos* pada R-MCL.

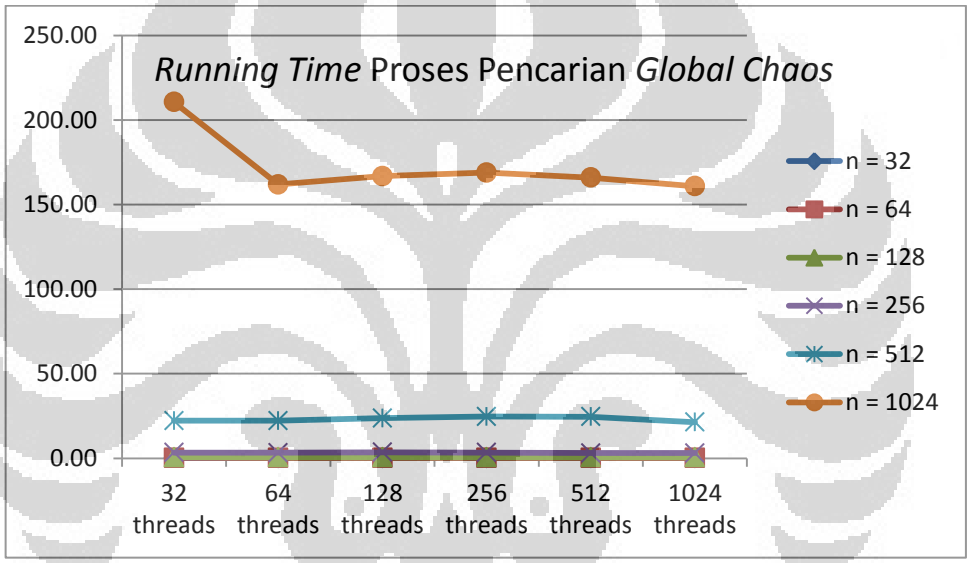
Simulasi dilakukan untuk nilai n dan $blockDim.x$ yang bervariasi. Gambar 4.10 merupakan contoh output yang dihasilkan saat program ini dijalankan untuk $n = 1024$ dengan banyak *threads per block* 32 dan 64. Tabel 4.7 menyatakan *running time* dari program paralel pencarian *global chaos* untuk $n = 32, 64, 128, \ldots, 1024$ dengan $blockDim.x = 32, 64, 128, \ldots, 1024$, sedangkan Gambar 4.11 merupakan representasi grafis dari *running time* yang diperoleh. Tabel 4.8 berisi informasi *speedup* yang dihasilkan pada proses pencarian *global chaos*, sedangkan Gambar 4.11 merupakan grafik *speedup* dari proses ini.



Gambar 4. 10 *Screenshot* Program Paralel Pencarian *Global Chaos* untuk $n = 1024$ dengan $blockDim.x = 32$ dan $blockDim.x = 64$

Tabel 4. 7 *Running Time* Program Paralel *Global Chaos*

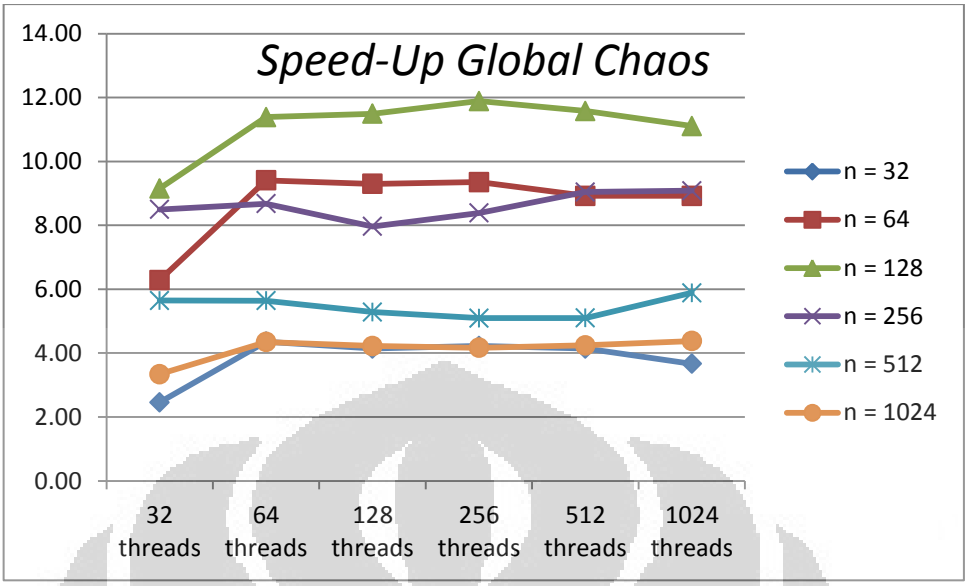
n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	0.170000	0.096000	0.101000	0.099000	0.101000	0.114000
64	0.247000	0.165000	0.167000	0.166000	0.174000	0.174000
128	0.683000	0.549000	0.544000	0.526000	0.540000	0.563000
256	3.350000	3.280000	3.575000	3.395000	3.149000	3.134000
512	22.250999	22.258999	23.764000	24.653999	24.634998	21.344000
1024	210.589996	161.876999	166.768997	168.860992	165.925995	160.871994



Gambar 4. 11 Grafik *Running Time* Algoritma Paralel untuk Pencarian *Global Chaos*

Tabel 4. 8 *Speedup* Program Paralel Pencarian *Global Chaos*

n	*Threads per Block*					
	32	**64**	**128**	**256**	**512**	**1024**
32	2.459412	4.355208	4.139604	4.223232	4.139604	3.667544
64	6.284980	9.408424	9.295749	9.351747	8.921782	8.921782
128	9.154143	11.388488	11.493162	11.886464	11.578296	11.105293
256	8.497301	8.678646	7.962506	8.384672	9.039682	9.082948
512	5.643900	5.641871	5.284565	5.093795	5.097724	5.883734
1024	3.342348	4.348147	4.220599	4.168310	4.242042	4.375311



Gambar 4. 12 Grafik *Speedup* Proses Pencarian *Global Chaos*

Dari simulasi di atas, terlihat bahwa semakin besar ukuran matriks input, semakin besar pula waktu eksekusi yang diperlukan. Penggunaan *threads per block* sebanyak 256 cenderung memberikan *running time* yang paling baik untuk proses pencarian *global chaos* matriks dengan $n = 32$, $n = 64$, dan $n = 128$, sedangkan untuk n yang lebih besar, *running time* terbaik diperoleh saat penggunaan 1024 *threads per block*. Untuk evaluasi *speedup*, hasil terbaik diperoleh saat ukuran matriks yang dieksekusi adalah 128×128. *Speedup* yang diperoleh berada pada kisaran 9 hingga 12 kali. Sejalan dengan *running time* yang dihasilkan, penggunaan 256 *threads per block* memberikan *speedup* yang cenderung lebih baik untuk $n \leq 128$, sedangkan penggunaan 1024 *threads per block* memberikan *speedup* yang lebih baik untuk $n \geq 256$.

BAB 5
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Pada skripsi ini telah dibangun algoritma paralel R-MCL untuk pemrograman GPGPU yang terdiri dari algoritma paralel untuk konversi matriks *sparse* menjadi ELLPACK-R, algoritma paralel untuk proses ekspansi, inflasi, *prune*, serta algoritma paralel untuk pencarian *global chaos*. Algoritma-algoritma paralel ini telah diimplementasikan dengan menggunakan *compiler* NVCC pada mesin dengan sistem operasi Ubuntu. Dari hasil konstruksi algoritma paralel dan simulasi yang dilakukan, diperoleh empat hasil berikut. Pertama, untuk setiap algoritma paralel yang dibangun, implementasi yang dilakukan memberikan *running time* yang berbanding lurus dengan ukuran matriks input. Kedua, untuk program paralel konversi matriks *sparse* menjadi ELLPACK-R, penggunaan 128 *threads per block* memberikan hasil paling baik, kemudian penggunaan 64 *threads per block* memberikan hasil paling baik untuk program paralel inflasi, serta 256 *threads per block* untuk program paralel *prune*. Untuk program paralel pencarian *global chaos*, *running time* yang optimal untuk ukuran matriks yang tidak lebih dari 128×128 diperoleh saat digunakan 256 *threads per block*, sedangkan *running time* yang paling baik untuk matriks dengan ukuran yang lebih besar diperoleh saat digunakan 1024 *threads per block*. Ketiga, untuk program ELLPACK-R, *speedup* terbaik diperoleh saat eksekusi matriks berukuran 256×256, sedangkan eksekusi matriks berukuran 128×128 memberikan *speedup* yang baik untuk program inflasi dan pencarian *global chaos*. Untuk program paralel *prune*, *speedup* yang diperoleh cenderung baik untuk matriks berukuran 256×256 hingga 1024×1024. *Speedup* terbaik dihasilkan oleh program paralel inflasi, dimana *speedup* yang dihasilkan bisa mencapai 15 kali. Terakhir, jika dilihat dari sisi penggunaan *threads per block*, *speedup* yang dihasilkan sejalan dengan *running time* yang diperoleh, yaitu 128 untuk

ELLPACK-R, 64 untuk inflasi, 256 untuk *prune*, serta 256 dan 1024 *threads per block* untuk pencarian *global chaos*.

5.2 Saran

Penulis menyadari bahwa masih banyak kekurangan dalam skripsi ini. Salah satunya adalah permasalahan alokasi memori pada saat eksekusi program paralel ekspansi. Selain itu, karena keterbatasan waktu, implementasi program paralel yang utuh untuk algoritma R-MCL, dari proses konversi matriks *sparse* hingga proses pencarian *global chaos*, belum bisa dilakukan. Kedua hal ini dapat menjadi topik untuk penelitian lebih lanjut, khususnya penelitian di Departemen Matematika FMIPA UI.

DAFTAR PUSTAKA

Bustamam, A., K. Burrage, & N. A. Hamilton. (2010). *Fast Parallel Markov Clustering using Massively Parallel Graphics Processing Unit Computing*. Second Intl. Workshop on High Performance Computational Systems Biology.

Dongen, S. V. (2000). *Graph Clustering by Flow Simulation*. Ph.D. thesis, University of Utrecht.

Dongen, S. V. (2008). *Graph Clustering via a Discrete Uncoupling Process*. SIAM J. Matrix Anal. Appl. vol. 30, no. 1, pp. 121-148.

Gray, Paul, Henry Neeman, & Charlie Peck. (2007). *Parallel & Cluster Computing: MPI Introduction*. University of Oklahoma.

Harris, Mark. (2008). *Optimizing Parallel Reduction in CUDA*. NVIDIA Developer Technology. Santa Clara: NVIDIA Press.

Lindholm, Erik, *et al*. (2008). *Tesla: A Unified Graphics and Computing Architecture*. IEEE Press.

Mozdzynski, George. (2011). *Concepts of Parallel Computing*.Use of High Performance Computing in Meteorology: Proceedings of the 12th ECMWF Workshop.

Nickolls, J., I. Buck, M. Garland, & K. Skadron. (2008). *Scalable Parallel Programming with CUDA*. ACM Queue vol. 6 no. 2, pp 40-53.

NVIDIA. (2012). *NVIDIA CUDA Zone*. Online. Available: http://www.nvidia.com/object/cuda_home.html.

NVIDIA. (2011). *CUDA C Programming Guide 4.0*. Santa Clara: NVIDIA Press.

Satuluri, V., & S. Parthasarathy. (2009). *Scalable Graph Clustering using Stochastic Flows: Application to Community Discovery*. New York: ACM.

Satuluri, V., S. Parthasarathy, & D. Ucar (2010). *Markov Clustering of Protein Interaction Networks with Improved Balanced and Scalability*. New York: ACM.

Tompa, Martin. (2009). *Basics of Moleculer Biology.* Seattle: Department of Genome Science University of Washington.

Vazquez, F., J. J. Fernandez, & E. M. Garzon. (2011). *A New Approach for Sparse Matrix Vector Product on NVIDIA GPUs*. Concurency and Computation: Practice and Experience 23:815-826.

Vierstraete, Andy. (2000). *The Central Dogma of Molecular Biology*. Belgium: Department of Biology University of Ghent.

Wafai, M. A. (2009). *Sparse Matrix-Vector Multiplications on Graphics Processors*. Universitat Stuttgart.

Zumkley, D. S., W. Wilhelms (2009). *Architectures of Parallel Computer*s. Universität Münster.

LAMPIRAN

Lampiran 1. *Listing* Program Paralel ELLPACK-R

```
#include <stdlib.h>
#include <stdio.h>
#include "cutil_inline.h"
#include "ellpack_kernel.cu"
#include <cuda_runtime.h>
int main (void) {
for (unsigned int k = 0; k <= log2(float(n / 32)); k++) {
      unsigned int bytes = N * sizeof(float);
      unsigned int bytes2 = n * sizeof(int);
      unsigned int numThreads = 32 * pow(2, k);
      unsigned int numBlocks = (N + numThreads - 1) / numThreads;
      unsigned int bytes5 = numBlocks * sizeof(float);
      float *M, *val, *b;
      int *col, *rl, *mcol;
      M = (float *) malloc(bytes);
      b = (float *) malloc(bytes5);
      rl = (int *) malloc(bytes2);
      mcol = (int *) malloc(sizeof(int));
      float* dev_M = NULL;
      float* dev_b = NULL;
      float* dev_val = NULL;
      int* dev_col = NULL;
      int* dev_rl = NULL;
      int* max_col = NULL;
      cutilSafeCall( cudaMalloc((void**) &dev_M, bytes));
      cutilSafeCall( cudaMalloc((void**) &dev_b, bytes5));
      cutilSafeCall( cudaMalloc((void**) &dev_rl, bytes2));
```

```
        cutilSafeCall( cudaMalloc((void**) &max_col, sizeof(int)));
        for (int i = 0 ; i < N ; i++) {
           if (i - i / 5 * 5 == 0)
              M[i] = i;
           else
              M[i] = 0;
        }
        unsigned int timer = 0;
        cutCreateTimer( &timer);
        cutStartTimer( timer);
        cutilSafeCall( cudaMemcpy(dev_M, M, bytes,
cudaMemcpyHostToDevice));
        unsigned int smemSize = (numThreads > 32 ? numThreads * sizeof(float) : 2
* numThreads * sizeof(float));
        maxcolumn<<<numBlocks, numThreads, smemSize>>>(dev_M, dev_rl,
max_col, dev_b);
        cutilSafeCall( cudaMemcpy(rl, dev_rl, bytes2,
cudaMemcpyDeviceToHost));
        cutilSafeCall( cudaMemcpy(mcol, max_col, sizeof(int),
cudaMemcpyDeviceToHost));
        cutStopTimer( timer);
           float gTime1=cutGetTimerValue( timer);
        for (int i = 0 ; i < 15 ; i++)
              printf("rl [%3d] = %-10d \n",i, rl[i]);
        printf("\n");
        unsigned int size_val = n * mcol[0];
        unsigned int bytes3 = size_val * sizeof(float);
        unsigned int bytes4 = size_val * sizeof(int);
        val = (float *) malloc(bytes3);
        col = (int *) malloc(bytes4);
int mmcol = mcol[0];
        cutResetTimer( timer);
```

```
        cutilSafeCall( cudaMalloc((void**) &dev_val, bytes3));
        cutilSafeCall( cudaMalloc((void**) &dev_col, bytes4));
        ellpack<<<numBlocks, numThreads, 1 * numThreads *
sizeof(float)>>>(dev_M, dev_val, dev_col, dev_rl, mmcol);
        cutilSafeCall( cudaMemcpy(val, dev_val, bytes3,
cudaMemcpyDeviceToHost));
        cutilSafeCall( cudaMemcpy(col, dev_col, bytes4,
cudaMemcpyDeviceToHost));
        cutStopTimer( timer);
           float gTime2=cutGetTimerValue( timer);
        for (int i = 0 ; i < 15 ; i++)
              printf("M [%3d] = %-15.6f \t val [%3d] = %-15.6f \t col [%3d] = %-
10d\n",i, M[i], i, val[i], i, col[i]);
        printf("\n");
        printf("Ukuran Matriks = %d x %d\n", n, n);
        printf("Banyak threads per block = %d\n", numThreads);
        printf ("Running Time parallel: %f ms\n\n", gTime1 + gTime2);
        cutDeleteTimer( timer);
        free(M);
        free(b);
        free(val);
        free(col);
        free(rl);
        free(mcol);
        cutilSafeCall( cudaFree(dev_M));
        cutilSafeCall( cudaFree(dev_b));
        cutilSafeCall( cudaFree(dev_val));
        cutilSafeCall( cudaFree(dev_col));
        cutilSafeCall( cudaFree(dev_rl));
        cutilSafeCall( cudaFree(max_col));
}
        return (0); }
```

Lampiran 2. *Listing* Kernel ELLPACK-R

```
#ifndef _ELLPACK_KERNEL_CU_
#define _ELLPACK_KERNEL_CU_
#define N 262144

__global__ void maxcolumn(float *M, int *rl, int *max_col, float *b) {
      extern __shared__ float sdata[];
      unsigned int i = blockIdx.x*blockDim.x+threadIdx.x;
      unsigned int tid = threadIdx.x;
      float result[n]={0.};
      unsigned int blokjlh = 1;
      if (n > blockDim.x)
         blokjlh = (n + blockDim.x -1) / blockDim.x;
      if (M[n * (blockIdx.x - (blockIdx.x / n) * n) + (blockIdx.x / n) * blockDim.x
+ tid] != 0)
         sdata[tid] = 1;
else
sdata[tid] = 0;
      __syncthreads();
      if (blockDim.x == 1024) {
            if (tid < 512)
                  sdata[tid] += sdata[tid + 512];
            __syncthreads(); }
      if (blockDim.x >= 512) {
            if (tid < 256)
                  sdata[tid] += sdata[tid + 256];
            __syncthreads(); }
      if (blockDim.x >= 256) {
            if (tid < 128)
                  sdata[tid] += sdata[tid + 128];
            __syncthreads(); }
```

```
if (blockDim.x >= 128) {
        if (tid < 64)
            sdata[tid] += sdata[tid + 64];
        __syncthreads(); }
if (tid < 32)
        warpReduce(sdata, tid);
if (tid == 0)
        b[blockIdx.x] = sdata[0];
if (i < n) {
    for (unsigned int k = 0; k < blokjlh; k++) {
        if (k * n + i < gridDim.x)
        result[i] += b [k * n + i]; }
    rl[i] = result[i]; }
__syncthreads();
sdata[tid] = (i < blockDim.x && blockIdx.x == 0 ? rl[i] : 0);
__syncthreads();
if (blockIdx.x == 0) {
if (blockDim.x == 1024) {
        if (tid < 512)
            if (sdata[tid] < sdata[tid+512])
                sdata[tid] = sdata[tid + 512];
        __syncthreads(); }
if (blockDim.x >= 512) {
        if (tid < 256)
            if (sdata[tid] < sdata[tid+256])
                sdata[tid] = sdata[tid + 256];
        __syncthreads(); }
if (blockDim.x >= 256) {
        if (tid < 128)
            if (sdata[tid] < sdata[tid+128])
                sdata[tid] = sdata[tid + 128];
        __syncthreads(); }
```

```
    if (blockDim.x >= 128) {
        if (tid < 64)
           if (sdata[tid] < sdata[tid+64])
              sdata[tid] = sdata[tid + 64];
        __syncthreads(); }
    if (tid < 32)
        warpReduceMod(sdata, tid);
    if (tid == 0)
        max_col[i] = sdata[0];}
}

__global__ void ellpack(float *M, float *val, int *col, int *rl, int max_col)  {
    unsigned int i = blockIdx.x*blockDim.x+threadIdx.x;
    unsigned int count = 0;
if (i < max_col) {
col[i] = 0;
val[i] = 0; }
__syncthreads();
    if (i < n) {
      for (int j = 0 ; j < n ; j++) {
        if (M[i * n + j] != 0) {
           count += 1;
           val[i * max_col + count] = M[i * n + j];
           col[i * max_col + count] = j;
           __syncthreads();
        }
      }
    }
}
#endif
```

Lampiran 3. *Listing* Program Paralel SpMV untuk Proses Ekspansi R-MCL

```
#include <stdio.h>
#include <stdlib.h>
#include "cutil_inline.h"
#include "spmv_kernel.cu"
#include <cuda_runtime.h>

int main (void) {
	unsigned int size1 = N * sizeof(float);
	unsigned int size2 = N * sizeof(int);
	unsigned int size3 = n * sizeof(float);
	  unsigned int size4 = n * sizeof(int);
	unsigned int numThreads = 32;
	unsigned int numBlocks = (N + numThreads - 1) / numThreads;
	float *val, *b, *c;
	int *col, *rl;
	val = (float *) malloc(size1);
	col = (int *) malloc(size2);
	rl = (int *) malloc(size4);
	b = (float *) malloc(size3);
	c = (float *) malloc(size3);
	float* dev_val = NULL, dev_b = NULL, dev_c = NULL;
	int* dev_col = NULL, dev_rl = NULL;
	   cudaMalloc((void**) &dev_val, size1);
	   cudaMalloc((void**) &dev_col, size2);
	   cudaMalloc((void**) &dev_rl, size4);
	   cudaMalloc((void**) &dev_b, size3);
	   cudaMalloc((void**) &dev_c, size3);
	   for (int i = 0; i < N; i++) {
			col[i] = i;
			if (i - (i / 5) * 5 == 0)
```

```
                val[i] = 2 * i;
            else
                val[i] = i;
            if (i < n) {
                b[i] = val[i];
                rl[i] = i / 2;
            }  }
        printf("\n");
    unsigned int timer = 0;
    cutCreateTimer( &timer);
    cutStartTimer( timer);
        cudaMemcpy(dev_val,val,size1,cudaMemcpyHostToDevice);
        cudaMemcpy(dev_col,col,size2,cudaMemcpyHostToDevice);
        cudaMemcpy(dev_rl,rl,size4,cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b,b,size3,cudaMemcpyHostToDevice);
        spmv<<<numBlocks, numThreads, 2 * numThreads *
sizeof(float)>>>(dev_val, dev_col, dev_rl, dev_b, dev_c);
        cudaMemcpy(c,dev_c,size3,cudaMemcpyDeviceToHost);
    cutStopTimer( timer);
        float gTime1=cutGetTimerValue( timer);
        for (int i = 0; i < 20; i++) {
            printf("val [%3d] = %-12.8f\t col[%3d] = %-12d \t rl[%3d] = %-12d
\n", i, val[i], i, col[i], i, rl[i]);  }
        for (int i = 0; i < 20; i++) {
            printf("b [%3d] = %-12.8f \t c[%3d] = %-12.8f \n", i, b[i], i, c[i]); }
        printf("\n");
    printf ("Running Time parallel: %f ms\n", gTime1);
    cutDeleteTimer( timer);
    free(val); free(col); free(rl); free(b); free(c);
        cudaFree(dev_val); cudaFree(dev_col); cudaFree(dev_rl);
        cudaFree(dev_b); cudaFree(dev_c);
        return (0);  }
```

Lampiran 4. *Listing* Kernel SpMV untuk Proses Ekspansi R-MCL

```
#ifndef _SPMV_KERNEL_CU_
#define _SPMV_KERNEL_CU_

__global__ void spmv2(float *val, int *col, int *rl, float *b, float *c) {
      extern __shared__ float sdata[];
      unsigned int tid = threadIdx.x, i = tid + __mul24(blockIdx.x, n);
      if (tid < rl[blockIdx.x]) {
         sdata [tid] = val[blockIdx.x * n + tid] * b[col[blockIdx.x * n + tid]];
         __syncthreads();  }
      if (blockDim.x == 1024) {
            if (tid < 512)
                  sdata[tid] += sdata[tid + 512];
            __syncthreads();  }
      if (blockDim.x >= 512) {
            if (tid < 256)
                  sdata[tid] += sdata[tid + 256];
            __syncthreads();  }
      if (blockDim.x >= 256) {
            if (tid < 128)
                  sdata[tid] += sdata[tid + 128];
            __syncthreads();  }
      if (blockDim.x >= 128) {
            if (tid < 64)
                  sdata[tid] += sdata[tid + 64];
            __syncthreads();  }
      if (tid < 32) {
            warpReduce(sdata, tid); }
      if (tid == 0)
            c[blockIdx.x] = sdata[0];  }
#endif
```

Lampiran 5. *Listing* Program Paralel untuk Proses Inflasi R-MCL

```
#include <stdio.h>
#include <stdlib.h>
#include "cutil_inline.h"
#include "inflate_kernel.cu"
#include <cuda_runtime.h>

int main (void) {
for (unsigned int k = 0; k < 6; k++) {
      unsigned int bytes = N * sizeof(float);
      unsigned int bytes2 = n * sizeof(float);
      unsigned int numThreads = 32 * pow(2, k);
      unsigned int numBlocks = (N + numThreads - 1) / numThreads;
      unsigned int bytes3 = numBlocks * sizeof(float);
      float *a, *b, *c, *d, *h;
      a = (float *) malloc(bytes);
      b = (float *) malloc(bytes3);
      c = (float *) malloc(bytes);
      d = (float *) malloc(bytes2);
      h = (float *) malloc(bytes);
      float* dev_a = NULL, dev_b = NULL, dev_c = NULL;
      float* dev_d = NULL, dev_h = NULL;
      cutilSafeCall( cudaMalloc((void**) &dev_a, bytes));
      cutilSafeCall( cudaMalloc((void**) &dev_b, bytes3));
      cutilSafeCall( cudaMalloc((void**) &dev_c, bytes));
      cutilSafeCall( cudaMalloc((void**) &dev_d, bytes2));
      cutilSafeCall( cudaMalloc((void**) &dev_h, bytes));
      for (int i = 0 ; i < N ; i++) {
            a[i] = i;  }
      unsigned int timer = 0;
      cutCreateTimer( &timer);
```

```
        cutStartTimer( timer);
        cudaMemcpy(dev_a, a, bytes, cudaMemcpyHostToDevice);
        unsigned int smemSize = (numThreads > 32 ? numThreads * sizeof(float) : 2
* numThreads * sizeof(float));
        inflate<<<numBlocks, numThreads, smemSize>>>(dev_a, dev_b, dev_c,
dev_d, dev_h);
        cudaMemcpy(c, dev_c, bytes, cudaMemcpyDeviceToHost);
        cudaMemcpy(d, dev_d, bytes2, cudaMemcpyDeviceToHost);
        cudaMemcpy(h, dev_h, bytes, cudaMemcpyDeviceToHost);
        cutStopTimer( timer);
           float gTime1=cutGetTimerValue( timer);
        for (int i = 0 ; i < 5 ; i++)
              printf("a [%3d] = %-15.6f \t c [%3d] = %-15.6f \t h [%3d] = %-
12.10f\n",i, a[i], i, c[i], i, h[i]);
        printf("\n");
        for (int i = 0 ; i < 5 ; i++)
              printf("sum_col [%3d] = %-16.6f \n",i, d[i]);
        printf("\n");
        printf("Ukuran Matriks = %d x %d\n", n, n);
        printf("Banyak threads per block = %d\n", numThreads);
        printf ("Running Time parallel: %f ms\n", gTime1);
        cutDeleteTimer( timer);
        free(a), free(b), free(c), free(d), free(h);
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);
        cudaFree(dev_d);
        cudaFree(dev_h);
}
        return (0);
}
```

Lampiran 6. *Listing* Kernel untuk Proses Inflasi R-MCL

```
#ifndef _INFLATE_KERNEL_CU_
#define _INFLATE_KERNEL_CU_

__global__ void inflate(float *a, float *b, float *c, float *d, float *h)  {
      extern __shared__ float sdata[];
      unsigned int j = blockIdx.x*blockDim.x+threadIdx.x;
      unsigned int tid = threadIdx.x;
      unsigned int blokjlh = n / blockDim.x;
      float result[n]={0.};
      if (j < N) {
         c[j] = pow(a[j],r);  }
      __syncthreads();
      sdata[tid] = (j < N ? c[n * (n / blockDim.x) * tid + blockIdx.x] : 0);
      __syncthreads();
      if (blockDim.x == 1024) {
            if (tid < 512)
                  sdata[tid] += sdata[tid + 512];
            __syncthreads(); }
      if (blockDim.x >= 512) {
            if (tid < 256)
                  sdata[tid] += sdata[tid + 256];
            __syncthreads(); }
      if (blockDim.x >= 256) {
            if (tid < 128)
                  sdata[tid] += sdata[tid + 128];
            __syncthreads(); }
      if (blockDim.x >= 128) {
            if (tid < 64)
                  sdata[tid] += sdata[tid + 64];
            __syncthreads(); }
```

```
    if (tid < 32) {
        warpReduce(sdata, tid);  }
    if (tid == 0)
        b[blockIdx.x] = sdata[0];
    if (j < n) {
      for (unsigned int k = 0; k < blokjlh; k++) {
        if (k * n + j < gridDim.x)
        result[j] += b [k * n + j];}
      d[j] = result[j];  }
    __syncthreads();
    if (j < N) {
      unsigned int div = j - (j/ n) * n;
      h[j] = c[j] / d[div];
    }
}
#endif
```

Lampiran 7. *Listing* Program Paralel *Prune*

```
#include <stdio.h>
#include <stdlib.h>
#include "cutil_inline.h"
#include "cutil_inline.h"
#include "prune_kernel.cu"

int main (void) {
for (unsigned int k = 0; k < 6; k++) {
	unsigned int size1 = N * sizeof(float);
	unsigned int numThreads = 32 * pow(2, k);
	unsigned int numBlocks = (N + numThreads - 1) / numThreads;
	float *h;
	h = (float *) malloc(size1);
	float* dev_h = NULL;
	   cudaMalloc((void**) &dev_h, size1);
	   for (int i=0;i < N;i++) {
		if (i%5==0)
			h[i] = i;
		else
			h[i] = i * 10e-7;
	   }
	unsigned int timer = 0;
	cutCreateTimer( &timer);
	cutStartTimer( timer);
	   cudaMemcpy(dev_h,h,size1,cudaMemcpyHostToDevice);
	unsigned int smemSize = (numThreads > 32 ? numThreads * sizeof(float) : 2
* numThreads * sizeof(float));
	   prune<<<numBlocks, numThreads, smemSize>>>(dev_h);
	   cudaMemcpy(h,dev_h,size1,cudaMemcpyDeviceToHost);
	cutStopTimer( timer);
```

```
    float gTime1=cutGetTimerValue( timer);
    for (int i = 0; i < 5; i++)
            printf("h[%d]=%.8f \n",i,h[i]);
    printf("\n");
  printf("Ukuran Matriks = %d x %d\n", n, n);
  printf("Banyak threads per block = %d\n", numThreads);
  printf ("Running Time parallel: %f ms\n", gTime1);
  cutDeleteTimer( timer);
  free(h);
  cudaFree(dev_h);
}
    return (0);
}
```

Lampiran 8. *Listing* Kernel *Prune*

```
#ifndef _PRUNE_KERNEL_CU_
#define _PRUNE_KERNEL_CU_

__global__ void prune(float *h) {
        int j = blockIdx.x*blockDim.x+threadIdx.x;
        float minval = 10e-4;
        if (j < N && h[j] < minval) {
        h[j] = 0.;
      }
}

#endif
```

Lampiran 9. *Listing* Program Paralel Pencarian *Global Chaos*

```
#include <stdio.h>
#include <stdlib.h>
#include "cutil_inline.h"
#include "gchaos_kernel.cu"
#include <cuda_runtime.h>

int main (void) {
for (unsigned int k = 0; k < 6; k++) {
      unsigned int bytes = N * sizeof(float);
      unsigned int bytes2 = n * sizeof(float);
      unsigned int numThreads = 32 * pow(2, k);;
      unsigned int numBlocks = (N + numThreads - 1) / numThreads;
      unsigned int bytes3 = numBlocks * sizeof(float);
      float *a, *b, *b2, *c, *d, *h;
      a = (float *) malloc(bytes);
      b = (float *) malloc(bytes3);
      b2 = (float *) malloc(bytes3);
      c = (float *) malloc(bytes);
      d = (float *) malloc(bytes2);
      h = (float *) malloc(bytes2);
      float *dev_a = NULL, *dev_b = NULL, *dev_b2 = NULL, *dev_c = NULL;
      float *dev_d = NULL, *dev_h = NULL;
      cutilSafeCall( cudaMalloc((void**) &dev_a, bytes));
      cutilSafeCall( cudaMalloc((void**) &dev_b, bytes3));
      cutilSafeCall( cudaMalloc((void**) &dev_b2, bytes3));
      cutilSafeCall( cudaMalloc((void**) &dev_c, bytes));
      cutilSafeCall( cudaMalloc((void**) &dev_d, bytes2));
      cutilSafeCall( cudaMalloc((void**) &dev_h, bytes2));
      for (int i = 0 ; i < N ; i++) {
            a[i] = i;  }
```

```
        unsigned int smemSize = (numThreads > 32 ? numThreads * sizeof(float) : 2
* numThreads * sizeof(float));
        unsigned int timer = 0;
        cutCreateTimer( &timer);
        cutStartTimer( timer);
        cutilSafeCall( cudaMemcpy((void *) dev_a,(void *)  a, bytes,
cudaMemcpyHostToDevice));
        gchaos1<<<numBlocks, numThreads, smemSize>>>(dev_a, dev_b, dev_c,
dev_d);
        cutilSafeCall( cudaMemcpy((void *) c,(void *) dev_c, bytes,
cudaMemcpyDeviceToHost));
        cutilSafeCall( cudaMemcpy((void *) d,(void *) dev_d, bytes2,
cudaMemcpyDeviceToHost));
        cutStopTimer( timer);
           float gTime1=cutGetTimerValue( timer);
        cutResetTimer( timer);
        gchaos2<<<numBlocks, numThreads, 2 * numThreads *
sizeof(float)>>>(dev_a, dev_b2, dev_h, dev_d);
        cutilSafeCall( cudaMemcpy((void *) h,(void *) dev_h, bytes2,
cudaMemcpyDeviceToHost));
cudaMemcpyDeviceToHost));
        cutStopTimer( timer);
           float gTime2=cutGetTimerValue( timer);
        for (int i = 0 ; i < 5 ; i++)
              printf("a [%3d] = %-15.6f \t c [%3d] = %-15.6f \t sum [%3d] = %-
12.10f\n",i, a[i], i, c[i], i, d[i]);
        printf("\n");
        for (int i = 0 ; i < 5 ; i++)
              printf("chaos [%3d] = %-16.6f \n",i, h[i]);
        printf("\n");
        printf("Ukuran Matriks = %d x %d\n", n, n);
        printf("Banyak threads per block = %d\n", numThreads);
```

```
        printf ("Running Time parallel: %f ms\n\n", gTime1 + gTime2);
        cutDeleteTimer( timer);
        free(a), free(b), free(b2), free(c), free(d), free(h);
        cutilSafeCall(cudaFree(dev_a));  cutilSafeCall(cudaFree(dev_b));
        cutilSafeCall(cudaFree(dev_b2));  cutilSafeCall(cudaFree(dev_c));
        cutilSafeCall(cudaFree(dev_d));  cutilSafeCall(cudaFree(dev_h));
}
        return (0);
}
```

Lampiran 10. *Listing* Kernel Pencarian *Global Chaos*

```
#ifndef _GCHAOS_KERNEL_CU_
#define _GCHAOS_KERNEL_CU_

__global__ void gchaos1(const float *a, float *b, float *c, float *d)  {
    extern __shared__ float sdata[];
    unsigned int j = blockIdx.x*blockDim.x+threadIdx.x;
    unsigned int tid = threadIdx.x;
    unsigned int blokjlh = (n + blockDim.x -1) / blockDim.x;
    float result[n] = {0.0f};
    c[j] = (j < N ? pow(a[j],2) : 0);
    __syncthreads();
    sdata[tid] = (j < N ? c[n * (n / blockDim.x) * tid + blockIdx.x] : 0.0f);
    __syncthreads();
    if (blockDim.x == 1024) {
        if (tid < 512)
            sdata[tid] += sdata[tid + 512];
        __syncthreads(); }
    if (blockDim.x >= 512) {
        if (tid < 256)
            sdata[tid] += sdata[tid + 256];
        __syncthreads(); }
    if (blockDim.x >= 256) {
        if (tid < 128)
            sdata[tid] += sdata[tid + 128];
        __syncthreads(); }
    if (blockDim.x >= 128) {
        if (tid < 64)
            sdata[tid] += sdata[tid + 64];
        __syncthreads(); }
    if (tid < 32) {
```

```
            warpReduce(sdata, tid);  }
        if (tid == 0)
            b[blockIdx.x] = sdata[0];
        if (j < n) {
           for (unsigned int k = 0; k < blokjlh; k++) {
            if (k * n + j < gridDim.x)
            result[j] += b [k * n + j];
            __syncthreads();  }
        d[j] = result[j];
        __syncthreads();  }  }

__global__ void gchaos2(const float *a, float *b2, float *h, float *d)
{
        extern __shared__ float shData[];
        unsigned int i = blockIdx.x*blockDim.x+threadIdx.x;
        unsigned int tid = threadIdx.x;
        unsigned int blokjlh = (n + blockDim.x -1) / blockDim.x;
        float result2[n] = {0.0f};
        shData[tid] = (i < N ? a[n * (n / blockDim.x) * tid + blockIdx.x] : 0.0f);
        __syncthreads();
        if (blockDim.x == 1024) {
            if (tid < 512)
               if (shData[tid] < shData[tid + 512])
                  shData[tid] = shData[tid + 512];
            __syncthreads(); }
        if (blockDim.x >= 512) {
            if (tid < 256)
               if (shData[tid] < shData[tid + 256])
                  shData[tid] = shData[tid + 256];
            __syncthreads(); }
        if (blockDim.x >= 256) {
            if (tid < 128)
```

```
            if (shData[tid] < shData[tid + 128])
                shData[tid] = shData[tid + 128];
        __syncthreads(); }
    if (blockDim.x >= 128) {
        if (tid < 64)
            if (shData[tid] < shData[tid + 64])
                shData[tid] = shData[tid + 64];
        __syncthreads(); }
    if (tid < 32)
        warpReduceMod(shData, tid);
    if (tid == 0)
        b2[blockIdx.x] = shData[0];
    if (i < n) {
       for (unsigned int k = 0; k < blokjlh; k++) {
        if (k * n + i < gridDim.x && result2[i] < b2[k * n + i])
        result2[i] = b2[k * n + i];
        __syncthreads();}
       h[i] = result2[i] - d[i];}
    __syncthreads();  }
#endif
```

Lampiran 11. *Listing* Kernel *Warp Reduction*

```
__device__ void warpReduce(volatile float* sdata, unsigned int tid) {
        if (blockDim.x >=  64)
            sdata[tid] += sdata[tid + 32];
        if (blockDim.x >=  32)
            sdata[tid] += sdata[tid + 16];
        if (blockDim.x >=  16)
            sdata[tid] += sdata[tid + 8];
        if (blockDim.x >=   8)
            sdata[tid] += sdata[tid + 4];
        if (blockDim.x >=   4)
            sdata[tid] += sdata[tid + 2];
        if (blockDim.x >=   2)
            sdata[tid] += sdata[tid + 1];
}
```

Lampiran 12. *Listing* Kernel Modifikasi *Warp Reduction* untuk Pencarian Nilai Maksimum

```
__device__ void warpReduceMod(volatile float* sdata, unsigned int tid) {

        if (blockDim.x >=  64) {
           if (sdata[tid] < sdata[tid+32])
              sdata[tid] = sdata[tid + 32];
        }
        if (blockDim.x >=  32) {
           if (sdata[tid] < sdata[tid+16])
              sdata[tid] = sdata[tid + 16];
        }
        if (blockDim.x >=  16) {
           if (sdata[tid] < sdata[tid+8])
              sdata[tid] = sdata[tid + 8];
        }
        if (blockDim.x >=   8) {
           if (sdata[tid] < sdata[tid+4])
              sdata[tid] = sdata[tid + 4];
        }
        if (blockDim.x >=   4) {
           if (sdata[tid] < sdata[tid+2])
              sdata[tid] = sdata[tid + 2];
        }
        if (blockDim.x >=   2) {
           if (sdata[tid] < sdata[tid+1])
              sdata[tid] = sdata[tid + 1];
        }
}
```