KOMPUTASI PARALEL BERBASIS GPU UNTUK FLUIDA TAK-MAMPAT DI DALAM KOTAK DUA DIMENSI

SKRIPSI

Oleh :

ABD. MAJID HAMSI

0210930001-93

JURUSAN FISIKA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS BRAWIJAYA

MALANG

2009

KOMPUTASI PARALEL BERBASIS GPU UNTUK FLUIDA TAK-MAMPAT DI DALAM KOTAK DUA DIMENSI

SKRIPSI

Sebagai salah satu syarat untuk memperoleh gelar
Sarjana Sains dalam bidang Fisika

Oleh :
ABD. MAJID HAMSI
0210930001-93

JURUSAN FISIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS BRAWIJAYA
MALANG

2009

KOMPUTASI PARALEL BERBASIS GPU UNTUK FLUIDA TAK-MAMPAT DI DALAM KOTAK DUA DIMENSI

ABSTRAK

Pesamaan *Navier-Stokes* banyak digunakan dalam analisis dinamika fluida. Sifat non linear persamaan ini membuatnya sulit diselesaikan secara analitik kecuali untuk kasus-kasus tertentu. Dalam penelitian ini persamaan *Navier-Stokes* diselesaikan menggunakan metode fluida stabil (*stable fluid*). Diskritisasi menggunakan metode beda hingga dan penyelesaian sistem persamaan linear menggunakan metode iterasi Jacobi.

Dalam lima tahun terahir, penelitian dalam bidang penggunaan GPU untuk komputasi umum (dikenal sebagai GPGPU / *General Purpose Graphics Processing Unit*) berkembang pesat. Tahun 2003 NVIDIA mengeluarkan produk yang bernama CUDA (*Compute Unified Device Archsitecture*). CUDA membuat pemrograman paralel di GPU lebih mudah. Penetilian ini menggunakan GPU untuk meningkatkan proses komputasi.

Kata kunci: komputasi paralel, fluida tak-mampat, persamaan *Navier-Stokes*, fluida stabil, iterasi Jacobi, GPU , NVIDIA CUDA .

PARALLEL COMPUTING ON GPU FOR INCONPRESSIBLE FLUID IN TWO DIMENSIONS BOX

ABSTRACT

Navier-Stokes equation was frequently used in fluid dynamic. Analytic solution of this equation is hard to find because it's non-linearity, except in special case. This research uses stable fluid method in order to solve Navier-Stokes equation and finite difference method for discretization. Jacobi iteration is used to solve linear system equation.

Research uses GPU as general purpose computation, known as GPGPU (General Purpose Graphics Processing Unit) has become interesting research in the 5 years this time. In 2003 NVIDIA Copr. released CUDA (Compute Unified Device Architecture). CUDA makes parallel computation on GPU easer. This research uses GPU to increase rate of computation.

Keywords: parallel computing, incompressible fluid, stable fluid, Navier-Stokes equation, Jacobi iteration, GPU, NVIDIA CUDA.

KATA PENGANTAR

Dengan kesadaran yang nyata atas kebesaran Ilahi dan rasa suka cita yang dalam, penulis panjatkan puji dan syukur kehadirat Allah SWT yang telah melimpahkan berkah, rahmat, hidayah dan inayah-Nya. Sholawat dan salam semoga terlimpahkan atas Nabi Muhammad SAW dan kepada para pengikutnya.

Penyelesaian skripsi ini tentunya tidak lepas dari bantuan dan dorongan dari berbagai pihak. Ucapan terima kasih penulis ucapkan kepada:

1. Kedua orang tuaku tercinta yang selalu menjadi inspirasiku. Terima kasih atas segalanya. Aku takkan pernah mampu membalas kasih sayangmu.
2. Bapak Drs. Adi Susilo, M.Si., Ph.D selaku pembimbing II dan ketua jurusan Fisika Universitas Brawijaya Malang.
3. Bapak DR. Eng. Agus Naba,S.Si, M.Si, selaku Pembimbing I yang telah memberikan ide, bimbingan, arahan dan saran.
4. Bapak DR. Sugeng Rianto yang telah memberikan kritik dan sarannya kepada penulis.
5. Keempat kakakku terkasih, Jayadi, Hidayati, Nizammuddin, dan Qudratullah.
6. Semua keponakanku, Isbil, Kanzul, Dayat, Diva, Ayu, Nadia, Eca, serta seluruh keluargaku yang selalu memberiku dukungan dan dorongan tak henti-hentinya.
7. Batur-batur Asrama Rinjani 1 Malang.
8. Seluruh dosen, staf karyawan serta para laboran jurusan Fisika.
9. Sahabat-sahabatku Fisika angkatan 2002 yang telah memberikan motifasi dan dukungan kepada penulis, terutama Teguh, matur nuwun atas pinjaman printer-nya.
10. Semua pihak yang tidak dapat penulis sebutkan satu persatu.

Sebagai manusia biasa penulis sangat menyadari ada banyak kekurangan dan keterbatasan di dalam laporan ini. Dengan kerendahan hati penulis mengharapkan segala saran, dan kritik demi kesempurnaan laporan ini. Semoga laporan ini dapat bermanfaat bagi perkembangan ilmu pengetahuan di Indonesia tercinta.

Malang, Juli 2009

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR LAMPIRAN

BAB I
PENDAHULUAN

1.1 Latar Belakang

Sain klasik didasarkan pada observasi, teori, dan eksperimen. Observasi pada sebuah fenomena menuntun pada sebuah hipotesis, dari hipotesis ilmuan membangun sebuah teori untuk menjelaskan fenomena tersebut dan mendisain eksperimen untuk membuktikan teorinya. Sayangnya, eksperimen tidak selalu bisa dilakukan dengan alasan biaya yang mahal, memakan waktu yang lama atau eksperimen tersebut tidak mungkin dilakukan. Komputer berkecepatan tinggi memberikan kemudahan bagi ilmuan untuk membuktikan teori mereka dengan membuat simulasi terhadap fenomena fisis. Ilmuan membandingkan hasil simulasi dengan hasil observasi fenomena fisis untuk menguji teori mereka. Pada saat ini, observasi, teori, eksperimen, dan simulasi numerik merupakan dasar bagi ilmu modern. Akan tetapi banyak masalah saintifik yang terlalu komplek yang penyelesaian numeriknya membutuhkan komputer berkemampuan tinggi, salah satunya adalah persamaan *Navier-Stokes (Quinn, 1994)*.

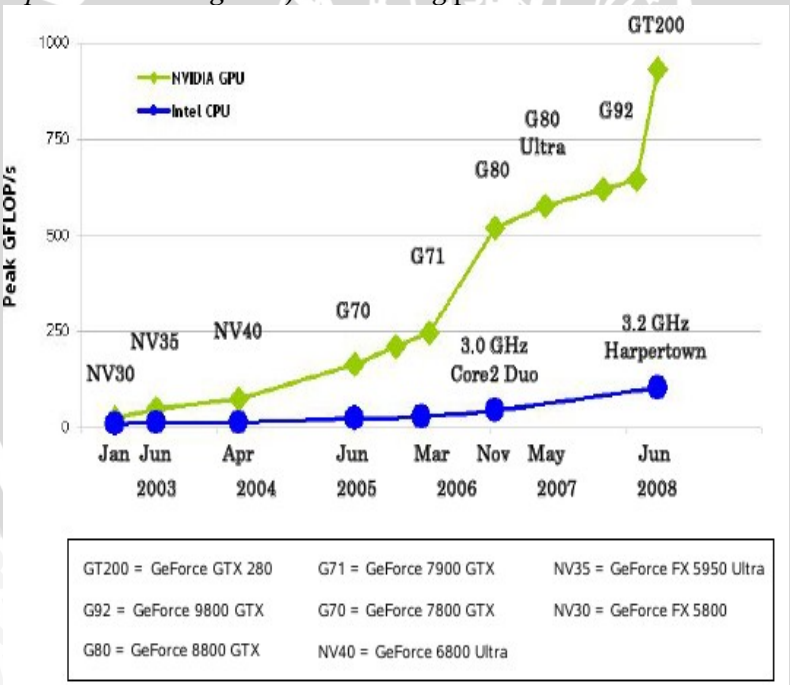
Persamaan *Navier-Stokes* merupakan salah satu persamaan yang banyak digunakan dalam menjelaskan aliran fluida. Persamaan ini memberikan informasi yang banyak mengenai aliran fluida. Akan tetapi sifat non linear persamaan ini membuatnya sulit diselesaikan secara analitik. Penyelesaian analitik persamaan ini hanya diperoleh utuk kasus-kasus tertentu dengan menggunakan asumsi-asumsi yang kasar. Analisis numerik memberikan solusi untuk persamaan – persamaan yang sulit mendapatkan solusi analitiknya seperti persamaan *Navier-Stokes*.

Griebel, dkk (1998) menggunakan metode beda hingga untuk menyelesaikan persamaan *Navier-Stokes* secara numerik. Metode beda hingga banyak digunakan untuk penyelesaian numerik persamaan diferensial parsial. Stam (1999) memperkenalkan metode fluida stabil (*stable fluid*) untuk menyelsaikan persamaan *Navier-Stokes*. Metode ini lebih stabil dan lebih mudah diimplementasikan.

Akan tetapi persamaan *Navier-Stokes* merupakan persamaan yang komplek sehingga solusi numeriknya pun komplek.

Kompleksitas solusi numerik ini merupakan masalah yang banyak dihadapi karena hal ini berkaitan langsung dengan lamanya waktu komputasi yang dibutuhkan. Akan tetapi masalah ini dapat diselesaikan menggunakan metode komputasi paralel. Perkembangan metode komputasi paralel didasari dari berkembangnya teknologi komputer, terutama teknologi komputer paralel.

Dalam beberapa dekade terahir, konsep paralelisme tidak hanya diterapkan dalam disain perangkat keras komputer akan tetapi juga diterapkan dalam disain GPU (*Graphics Processing Unit*) atau kartu grafik. GPU pada awalnya teknologi yang khusus didisain untuk pemerosesan grafis, akan tetapi kini telah banyak digunakan dalam komputasi paralel, sehingga kompuitasi paralel bisa dilakukan pada PC (*Personal Computer*) yang dilengkapi dengan GPU. Lebih dari lima tahun terahir, penelitian dalam bidang penggunaan GPU dalam komputasi umum (dikenal sebagai GPGPU /*General Purpose Graphics Processing Unit*) berkembang pesat.



Gambar 1.1. Perbandingan operasi floating point antara CPU dan GPU NVIDIA (Manual CUDA, 2008).

Gambar 1.1 menunjukkan perbandingan operasi *floating point* antara CPU dan GPU NVIDIA. Ukuran b*andwidth* dan operasi *floating-point* GPU meningkat dengan faktor 10 dibandingkan CPU.

Pada tahun 2003 NVIDIA mengeluarkan produk yang bernama CUDA (*Compute Unified Device Architecture*). CUDA merupakan model pemrograman paralel pada GPU, diekstensi dari bahasa C. CUDA memudahkan pemrograman paralel pada GPU.

1.2 Rumusan Masalah

1. Bagaimana membuat algoritma paralel yang sesuai dengan arsitektur GPU ?
2. Bagaimana perbandingan waktu komputasi antara CPU dan GPU ?

1.3 Batasan Masalah

1. Massa jenis fluida dianggap konstan.
2. Hanya membahas aliran dalam 2 dimensi.
3. Tidak membahas *error* komputasi.
4. Program yang dibuat hanya untuk GPU keluaran NVIDIA.
5. Hanya menggunakan iterasi Jacobi untuk menyelesaikan sistem persamaan linear.

1.4 Tujuan Penelitian

1. Membuat algoritma paralel yang dapat di terapkan pada GPU.
2. Mensimulasikan fluida di dalam kotak.
3. Membandingkan waktu komputasi antara CPU dan GPU.

1.5 Manfaat Penelitian

Dengan menggunakan metode komputasi paralel berbasi GPU, diharapkan komputasi akan lebih cepat dan lebih murah. Selain itu diharapkan penilitian ini dapat digunakan sebagai dasar komputasi paralel menggunakan GPU.

BAB II
TINJAUAN PUSTAKA

2.1 Dinamika Fluida Tak-Mampat

Menurut Kreith, dkk (1999), aliran fluida dapat dibedakan menjadi 5, yaitu:

1. Aliran tak-mampat
2. Aliran subsonik
3. Aliran transonik
4. Aliran supersonik
5. Aliran hipersonik

Pembagian jenis aliran ini didasari dari bilangan Mach (*Ma*), yang didefinikasn sebagai perbandingan kecepatan gas dengan kecepatan suara. Aliran subsonik sampai aliran hipersonik secara alami merupakan aliran mampat.

Aliran tak-mampat mempunyai bilangan $Ma \leq 0,3$. Semua jenis aliran fluida cair termasuk aliran tak-mampat karena kecepatan aliran fluida cair jauh lebih kecil dibandingkan dengan kecepatan suara. Adapun aliran fluida gas termasuk aliran tak-mampat jika kecepatan alirannya lebih kecil dari 100 m/s.

Menurut White (2000), bentuk persamaan *Navier-Stokes* (dalam dua dimensi) untuk fluida tak-mampat adalah :

$$\rho\left(\frac{\partial \vec{u}}{\partial t}+(\vec{u}\cdot\nabla)\vec{u}\right)=-\nabla p+\mu\nabla^2\vec{u}+\vec{f} \qquad (2.1)$$

dimana *u* dan *v* komponen vektor $\vec{u}$, ρ massa jenis fluida, μ koefesien viskositas, $\vec{u}$ vektor kecepatan, *p* tekanan, dan $\vec{f}$ gaya luar. Persamaan (2.1) dapat diterjemahkan dalam bentuk kata-kata : massa jenis x percepatan = gaya tekan persatuan volum + gaya viskos persatuan volum + gaya luar persatuan volum.

Suku kedua sebelah kiri persmaan (2.1) menunjukkan percepatan konveksi/adveksi. Percepatan konveksi merupakan percepatan yang diakibatkan oleh perubahan kecepatan terhadap posisi, misalnya kenaikan kecepatan fluida ketika fluida memasuki pipa lonjong. Suku pertama di sebelah kanan persamaan (2.1) menunjukkan pengaruh tekanan terhadap aliran fluida. Daerah

bertekanan lebih tinggi akan menekan ke arah daerah bertekanan rendah. Gradien negatif dari tekanan digunakan untuk mengetahui perngaruh perbedaan tekanan terhadap aliran fluida. Suku kedua persamaan (2.1) menunjukkan pengaruh viskositas terhadap aliran fluida. Gaya viskos fluida akan melawan deformasi fluida. Suku terahir sebelah kanan menujukkan pengaruh gaya luar terhadap aliran fluida, seperti gaya gravitas *(Bridson, dkk, 2007) dan (http://en.wikipedia.org/wiki/Navier-Stokes_equations)*.

Jika persamaan (2.1) diuraikan dalam bentuk kompoen-komponennya, maka diperoleh:

$$(\frac{\partial u}{\partial t}+(\vec{u}\cdot\nabla)u)=-\frac{\partial p}{\partial x}+\nabla^2 u+f_x \qquad (2.2a)$$

$$(\frac{\partial v}{\partial t}+(\vec{u}\cdot\nabla)v)=-\frac{\partial p}{\partial y}+\nabla^2 v+f_y \qquad (2.2b).$$

Persamaan (2.1) mempunyai tiga kuantitas yang tidak dikatahui : *u*, *v* dan *p* dan harus dikombinasikan dengan persamaan kontinuitas. Bentuk persamaan kontinuitas (hukum kekekalan massa) untuk fluida tak-mampat adalah :

$$\nabla\cdot\vec{u}=0 \qquad (2.3).$$

Persamaan (2.3) menunjukkan kondisi ketak-mampatan fluida, bagian dari persamaan *Navier-Stokes*. Medan vektor yang mengikuti persamaan (2.3) disebut medan bebas divergensi. Salah satu trik untuk mensimulasikan fluida tak-mampat adalah membuat medan vektor kecepatan tetap mengikuti persamaan kontinuitas *(White, 2000) dan (*B*ridson, dkk, 2007)*.

2.2 Diskritisasi

Diskritisasi digunakan dalam solusi numerik persamaan diferensial dengan mereduksi persamaan diferensial ke sistem persamaan aljabar. Ini menentukan nilai solusi hanya pada titik-titik tertentu pada suatu domain dimana solusi numerik akan dicari.

Rumus-rumus beda hingga yang paling banyak digunakan adalah aproksimasi tiga-titik beda-tengah dari turunan pertama dan kedua fungsi u, yaitu:

$$\left(\frac{\partial u}{\partial x}\right)_i = \frac{1}{2\Delta x}\left(u_{i+1} - u_{i-1}\right) + O\left(\Delta x^2\right) \qquad (2.4)$$

(2.4)

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i = \frac{1}{\Delta x^2}\left(u_{i+1} - 2u_{i-1} + u_{i-1}\right) + O\left(\Delta x^2\right) \qquad (2.5)$$

dengan $O(\Delta x^2)$ adalah *error*. Persamaan (2.4) dan (2.5) dapat dikembangkan untuk kasus 2 dimensi atau 3 dimensi. Tabel 2.1 menunjukkan diskritisasi operator ∇ (*nabla*) yang banyak digunakan dalam penelitian ini.

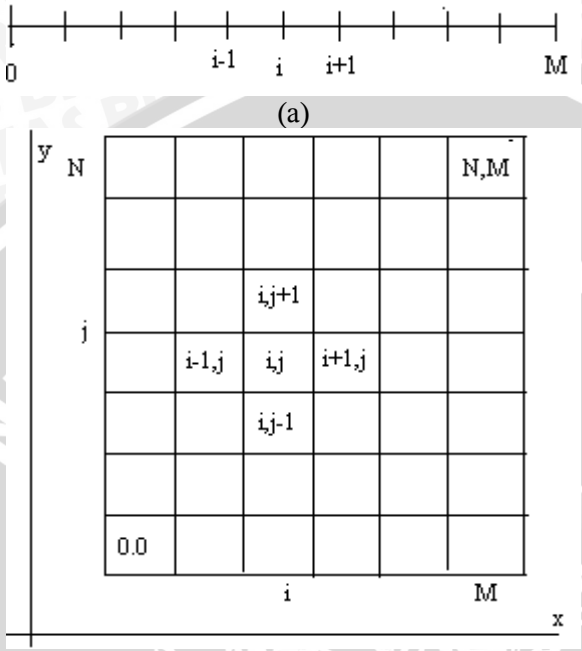
Tabel 2.1 Diskritisasi operator ∇ (del) (Sumber. Haris, 2007)

Operator	Definisi	Bentuk Diskrit (beda hingga)
Gradien	$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}\right)$	$\left(\frac{p_{i+1,j} - p_{i-1,j}}{2\Delta x}, \frac{p_{i,j+1} - p_{i,j-1}}{2\Delta y}\right)$
Divergensi	$\nabla \cdot \vec{u} = \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)$	$\frac{u_{i+1,j} - u_{i-j,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}$
Laplacian	$\nabla^2 u = \frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y}$	$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2}$

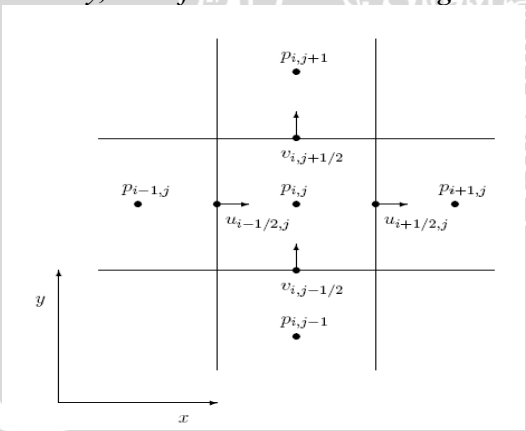
Subskrip i dan j yang digunakan dalam Tabel 2.1 menunjukkan lokasi titik-titik di dalam grid seperti yang ditunjukkan pada Gambar 2.1

S*taggered grid* banyak digunakan untuk mendiskritisasi persamaan *Navier-Stokes. V*ariabel-variabel yang tidak diketahui diletakkan pada lokasi yang berbeda-beda di dalam sel, seperti yang ditunjukka pada Gambar 2.2. Variabel tekanan p di dalam sel(i,j) diletakkan di tengah-tengah sel yang ditunjukkan oleh $p_{i,j}$. Komponen kecepatan arah horizontal diletakkan di tengah-tengah kompenen vertikal sel, yang ditunjukan oleh $u_{i+1/2,j}$, sedangkan komponen kecepatan arah vertikal diletakkan di tengah-tengah permukaan

sel(i,j) dalah arah horizontal seperti $v_{i,j+1/2}$ *(Bridson, dkk, 2007).*



Gambar 2. 1. (a) Diskritisasi dalam 1 dimensi. M dimensi grid arah x. (b) Diskritisasi dalam 2 dimensi. M, dan N dimensi grid arah x dan y, i dan j koordinat sel dalam grid



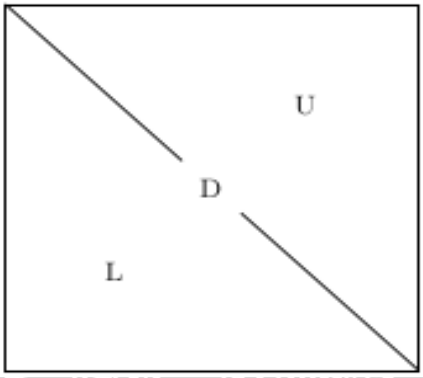
Gambar 2. 2 Staggered grid dalam 2-dimensi (Bridson, dkk, 2007).

2.3 Iterasi Jacobi

Dari diskritisasi persamaan diferensial akan diperoleh sebuah sistem persamaan linear

$$\mathbf{Ax} = \mathbf{b} \qquad (2.\ 6)$$

Matrik **A** mempunyai elemen yang sangat banyak dengan hanya sebagaian kecil elemennya yang tidak nol. Tentu saja mencari solusi terhadap sistem persamaan linear tersebut tidak efisien jika menggunakan metode langsung, seperti metode eliminasi Gauss. Metode iterasi memberikan solusi yang lebih efesien seperti metode iterasi Jacobi.



Gambar 2.3. Dekomposisi matrik **A**. **D** *matrik diagonal dari* **A**, **U** *matrik atas dari matrik* **A**, *dan* **L** *matrik bawah dari matrik* **A**.

Menurut Bahi, dkk (2008), metode iterasi Jacobi merupakan metode iterasi yang paling sederhana. Dimulai dengan mendekomposisi matrik **A** (lihat gambar 2. 2) menjadi :

$$\mathbf{A} = \mathbf{D} - (\mathbf{L} + \mathbf{U}) \qquad (2.\ 7)$$

dimana **D** matrik diagonal dari matrik **A**, **U** matrik atas, dan **L** matrik bawah. Diamsusikan semua elemen diagonal matrik **A** tidak ada yang nol. Jika persamaan (2.7) disubtitusikan ke parsamaan (2.6) maka diperoleh

$$\mathbf{D}\mathbf{x}^{(k+1)} - (\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)} = \mathbf{b}$$

atau

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\,\mathbf{x}^{(k)} + \mathbf{D}^{-1}\mathbf{b} \qquad (2.\ 8)$$

Dari persamaan (2.8) dapat disimpulkan bahwa pada tiap-tiap iterasi, masing-masing komponen vektor $\mathbf{x}^{(k+1)}$ menggunakan komponen vektor $\mathbf{x}^{(k)}$, sehingga semuah komponen vektor $\mathbf{x}^{(k)}$ harus disimpan dulu untuk bisa menghitung komponen vektor $\mathbf{x}^{(k+1)}$. Persamaan (2.8)

dapat ditulis dalam bentuk komponen-komponennya, yaitu :

$$x_i^{(k+1)} = (b_i - \sum_{j \neq i} A_{i,j} x_j^{(k)}) / A_{i,i} \qquad (2.9).$$

2.4 Komputasi Paralel

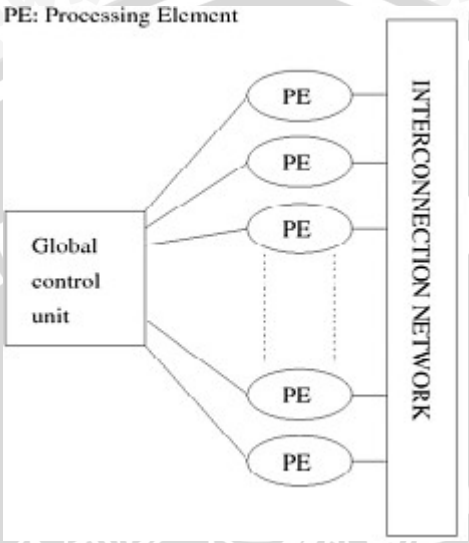
Menurut Knight (2008), komputasi paralel merupakan pengoperasian secara bersamaan terhadap tugas-tugas komputasi di dalam sistem komputer. Dari awal komputasi paralel telah menjadi bagian dari sistem komputer. Ada empat tingkatan berbeda dari komputasi paralel, yaitu :

1. Level tertinggi *job*, dimana sistem komputer beroperasi secara bersamaan terhadap tugas-tugas yang tak berhubungan.
2. Level kedua *program*, dimana sistem komputer beroperasi secara bersamaan pada bagian berbeda dari program yang sama (seperti operasi perulangan *do* pada banyak prosesor).
3. Level ketiga *intruksi*, dimana intruksi berbeda dieksekusi secara paralel.
4. Level keempat *aritmatic* dan *bit*, dimana paralelisme dicapai di dalam intruksi aritmatik tunggal atau intruksi bit.

Berdasarkan *data stream* (*stream* merupakan deretan intruksi dan atau data yang dieksekusi atau dioperasikan oleh prosesor), Knight (2008) membagi arsitektur komputer paralel menjadi 4 (pembagian ini disebut juga taksonomi Flynn), yaitu :

1. SISD (*Single Intruction Stream / Single Data Stream*), merupakan arsitektur dari komputer serial menggunakan data *stream* tunggal dan prosesor tunggal.
2. SIMD (*Single Intruction Stream / Multiple Data Stream*), merupakan jenis komputer paralel yang mempunyai banyak unit komputasi yang mampu mengeksekusi satu intruksi (seperti operasi penjumlahan) secara bersamaan pada data yang berbeda-beda.
3. MISD (*Multiple Intrucktion Stream / Single Data Stream*), merupakan jenis komputer paralel yang mampu mengeksekusi banyak intruksi dengan membagi unit-unit komputasi pada data yang sama.
4. MIMD (*Multiple Intruction Stream / Multiple Data Stream*). Arsitektur ini mempunyai banyak unit komputasi dimana

tiap-tiap unit komputasi memproses bagian data yang berbeda-beda.



Gambar 2.4. Arsitekture SIMD (Grama,dkk, 2003)

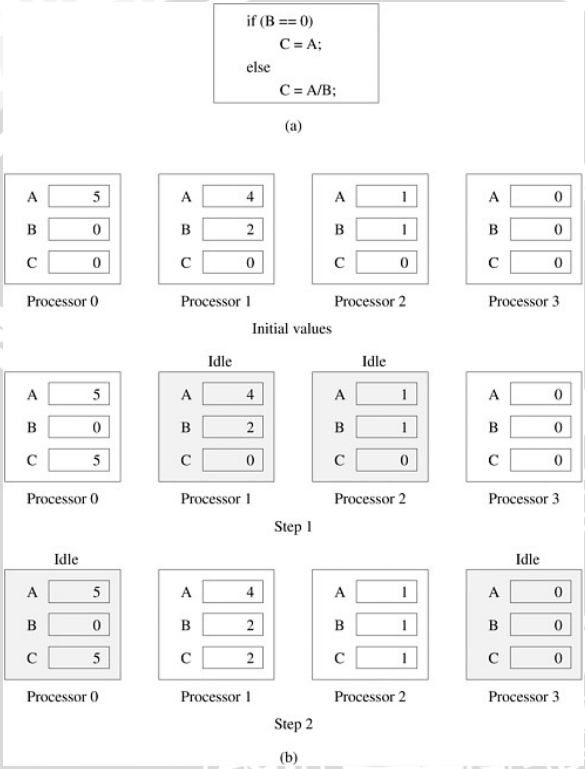
Kode program 2.1

```
1. for (i = 0; i < 1000; i++)
2.     c[i] = a[i] + b[i]; //dikerjakan dalam mode SIMD
```

Dalam komputer paralel berarsitektur SIMD, intruksi yang sama dieksekusi oleh semua unit pemrosesan secara sinkron. Pada kode program 2.1, intruksi *add* dikirimkan ke semua prosesor dan semua prosesor mengeksekusinya secara bersamaan *(Grama, dkk, 2003)*.

Konsep SIMD bekerja baik pada komputasi terstruktur dengan struktur data seperti *array*. Seringkali dibutuhkan menonaktifkan operasi pada data tertentu. Untuk alasan ini, sebagian besar paradigma pemrograman SIMD mengijinkan untuk sebuah "*activity mask*". Ini merupakan *mask* binari yang diasosiasikan dengan masing-masing item data dan sebuah operasi ditentukan apakah diikutkan dalam operasi atau tidak. Statemen kondisi di dalam pemrograman seperti statemen *if else* digunakan untuk

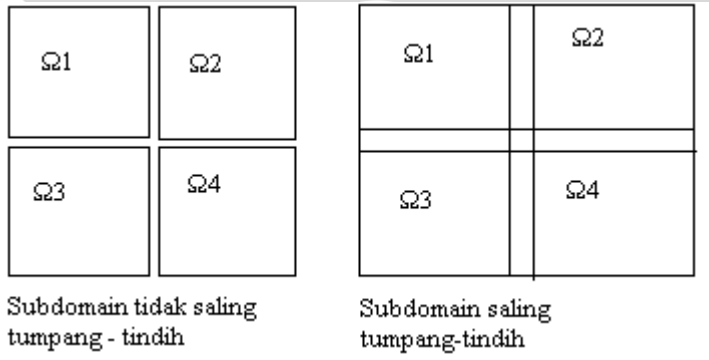
medukung eksekusi selektif. Eksekusi kondisional dapat merusak performa prosesor SIMD sehingga penggunaannya dalam pemrograman sebisa mungkin dihindari *(Grama, dkk, 2003)*.



Gambar 2.5. Eksekusi statemen kondisional pada komputer SIMD. (a) statemen kondisi, (b) eksekusi statement kondisi dalam dua langkah (Grama, dkk, 2003).

Gambar 2.5 menunjukkan eksekusi statemen kondisional pada arsitektur SIMD. Statemen kondisional pada Gambar 2.5 dieksekusi dalam dua langkah. Langkah pertama, semua prosesor yang mempunyai nilai B = 0 mengeksekusi intruksi C = A. Semua prosesor yang mempunyai B≠0 dalam keadaan *idle*. Langkah kedua, intruksi C=A/B dieksekusi. Prosesor yang dalam keadaan aktif pada langkah pertama sekarang dalam keadaan *idle* (nonaktif) *(Grama, 2003)*.

Salah satu pendekatan yang digunakan dalam memparalelisasi algoritma numerik untuk menyelesaikan persamaan diferensial adalah dengan membagi domain Ω menjadi subdomain Ω_1, Ω_2,. . . . Ω_N dan masing-masing prosesor menangani satu subdomain. Metode ini disebut *dekomposisi doman* (*domain decomposition*). Metode dekomposisi domain pertama kali diperkenalkan oleh *Amandus Schwarz* sehingga metode ini sering disebut juga metode *Schwarz*. Metode dekomposisi domain dibagi menjadi dua jenis yaitu tumpang-tindih (*overlapping*) dan tidak tumpang-tindih (*nonoverlapping*), bergantung pada apakah subdomain saling tumpang-tindih atau tidak, seperti yang ditunjukkan pada Gambar 2.6 *(Griebel, dkk, 1998)*.



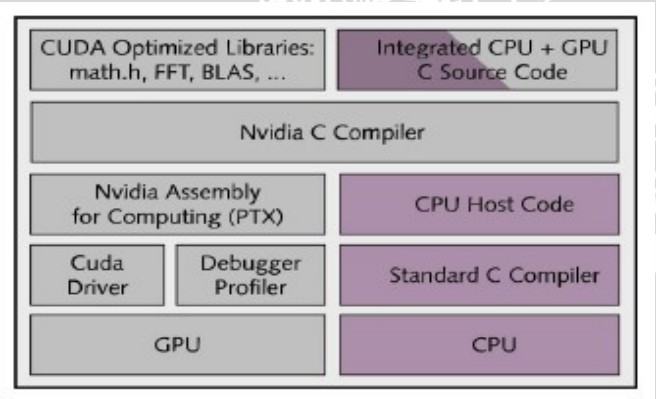
Gambar 2.6. Metode dekomposisi domain (Griebel,dkk, 1998).

2.5 Pemrograman CUDA Pada GPU

Pada awalnya pemgroraman GPU menggunakan *DirectX* atau *OpenGL*. DirectX dan openGL merupakan antar-muka aplikasi pemrograman (*Aplication programming Interface/API*) untuk menggambar 2 dimensi atau 3 dimensi yang diimplementasikan pada GPU. Ada dua kesulitan yang muncul ketika melakukan komputasi menggunakan DirectX atau openGL. Pertama, DirectX dan openGL mempunyai akses tingkat rendah pada GPU sehingga diperlukan pengetahuan yang cukup tentang perangkat keras GPU untuk bisa melakukan pemrograman GPU. Kedua, DirectX dan openGL didisain untuk menggambar 2 atau 3 dimensi bukan untuk komputasi yang umum, sehingga sulit untuk memetakan masalah-masalah komputasi yang umum pada DirectX atau openGL.

CUDA merupakan *compiler* keluaran NVIDIA untuk komputasi paralel pada GPU (*Graphic Processing Unit*). Dengan adanya CUDA pengembang *software* bisa menggunakan kemampuan GPU untuk mempercepat proses komputasi. CUDA di disain khusus hanya untuk kartu grafik (GPU) keluaran NVIDIA.

CUDA menyembunyikan detail dari perangkat keras GPU di bawah API. Penyembunyian detail perangkat keras mempunyai dua keuntungan. Pertama, menyederhanakan model perograman tingkat tinggi yang mengisolasi programer dari detail perangkat keras GPU. Kedua, kompatibilitas perangkat lunak, artinya perangkat lunak yang telah dibuat bisa berjalan pada berbagai macam artisetur GPU atau hanya mengalami perubahan sedikit untuk alasan optimasi jika pengembang perangkat keras (NVIDIA) merubah/mengembangkan arsitektur GPU-nya *(Halfhill, 2008)*.



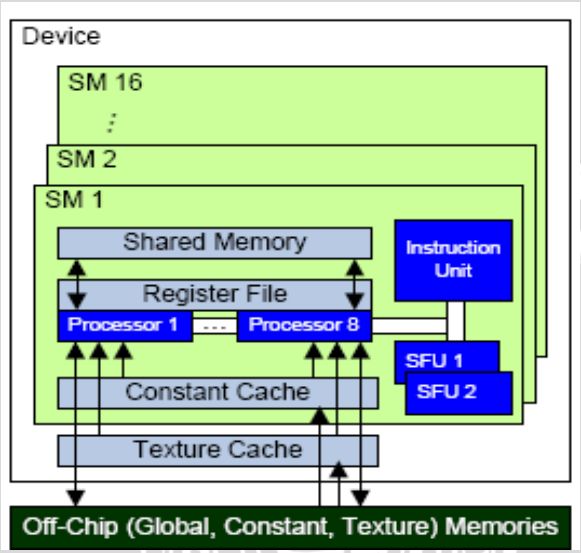
Gambar 2.7 Platform pemrogramann NVIDIA CUDA (Halfhill, 2008).

Seperti yang ditunjukkan paga Gambar 2.7, CUDA memasukkan *library* C/C++, *library* fungsi, dan mekanisme abstraksi perangkat keras (*driver* CUDA) yang menyembunyikan detail perangkat keras. CUDA mendukung makanisme pemrograman heterogen, artinya programer bisa menentukan kode yang akan dieksekusi di GPU atau CPU. Untuk sekarang, CUDA hanya mendukung operasi *single floating-point (Halfhill, 2008)*.

Level/tingkatan komputasi pada CUDA adalah paralel *thread.* Menurut Michel, dkk (2001), *thread* merupakan mekanisme

yang mengijinkan program melakukan lebih dari satu pekerjaan dalam satu waktu. Dalam konteks CUDA, fungsi *kernel* dapat dipandang sebagai program. Jumlah *thread* yang berjalankan pada GPU ditentukan pada saat fungsi kernel dijalankan (dieksekusi). Programer CUDA tidak perlu secara eksplisit mengatur *thread* yang berjalan pada GPU. CUDA secara otomatis mengatur *thread* sehingga memudahkan dalam pemrograman. *Thread* dalam CUDA bekerja dalam mode SIMD, ini artinya satu intruksi dijalankan oleh banyak *thread* secara bersamaan dengan masing-masing *thread* memproses data yang berbeda-beda.

2.5.1 Model *Hardware*



Gambar 2.8 Arsitektur NVIDIA G80 (Ryoo, 2008).

Gambar 2.8 menunjukkan arsitektur NVIDIA GeForce8800. NVIDIA GeForce8800 mempunyai 16 *Streaming Multiprocessors* (SM). Masing-masing SM terdiri dari 8 *Scalar Processor* (SP) atau *prosesor cores*, sehingga GeForce8800 mempunyai 16x8=128 SP. Tiap-tiap SP berjalan pada 1,35GHz *clock cycles.* Masing-masing SP mampu menangani maksimal 96 *thread,* sehingga jumlah maksimal *thread* yang mampu ditangani GeForce8800 adalah 96x128=12.288

thread. Masing-masing SP mengeksekusi intruksi *thread* dalam mode SIMD, dengan unit intruksi menyebarkan (*broadcasting*) intruksi yang dieksekusi sekarang ke semua SP. Tiap-tiap SP mempunyai registr 32-bit, single-presisi *floating-point*, dan unit aritmatik perkalian-penjumlahan yang mampu melakukan operasi aritmatik integer 32-bit. SM juga mempunyai dua unit SFU (*Special Function Unit*) yang betugas melakukan operasi-operasi floating-point lebih kompleks, seperti pembagian akar kuadrat,sinus, dan cosinus.

Untuk mengatur ratusan *thread* yang berjalan pada program yang berbeda, SM menerapkan arsitektur baru yang disebut SIMT (*Single Instruction Multiple Thread*). SM memetakan masing-masing *thread* ke satu SP. Masing-masing *thread* mempunyai alamat intruksi dan register, dan *thread* dieksekusi secara indipenden. Satu unit SIMT SM membuat, mengatur, menjadualkan dan mengeksekusi 32 kelompok *thread* secara paralel yang disebut *warp (Manual CUDA, 2008)*.

Berdasarkan model SIMD (*Single Intruction Multiple Data*), masing-masing SP di dalam satu SM harus mengeksekusi intruksi yang sama pada satu *clock cycle*, hanya data boleh berbeda. Pada satu intruksi, SIMT memilih warp yang siap dieksekusi. Sebuah *warp* mengeksekusi intruksi yang sama dalam satu *clock cycle*, sehingga efisiensi penuh akan tercapai jika *thread* dalam satu *warp* mengeksekusi alur intruksi yang sama. Jika *thread* dalam satu *warp* terpecah (*divergen*), misalnya melalui kondisi percabangan, maka *thread* di dalam *warp* akan dibagi lagi berdasarkan alur eksekusinya kemudian mengeksekusi kondisi alur percabangan secara serial, menonaktifkan *thread* yang tidak berada pada alur eksekusi yang sama *(Manual CUDA, 2008)*.

Masing-masing SM memunyai *on-chip* memori diantaranya :

1. Satu set register 32-bit per prosesor
2. Memori bersama (*Shared Memory*) yang dibagi untuk semua SP
3. Konstan *cache (Constant cache*)
4. Textur *cache* (*Texture cache*)

2.5.2 Model *Software*

Dalam konteks CUDA, GPU disebut *device* sedangkan CPU disebut *host.* Device hanya bisa mengakses memorinya sendiri.

Sebuah fungsi yang dieksekusi di GPU disebut fungsi *kernel. Kernel* dieksekusi dengan model SPMD (*Single Program Multiple Data*), artinya programer menentukan jumlah *thread* yang akan mengeksekusi fungsi kernel. Kode program 2.2 menunjukkan deklarasi fungsi kernel *(Breitbart, 2007)*.

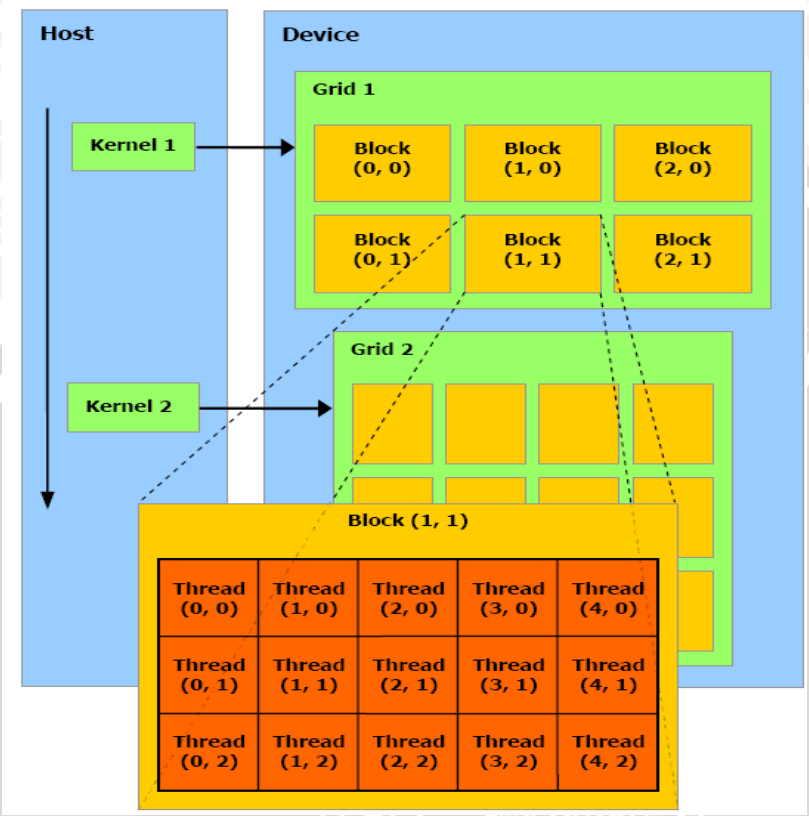
Kode rogram 2.2

```
__global__void kernel_func(arguman1, argumen2, . . . . . )
{
        //tubuh fungsi
}
```

Thread yang mengeksekusi *kernel* dikelompokkan sebagai sebuah grid dari blok-blok *thread* seperti yang ditunjukkan pada Gambar 2.9. Blok *thread* merupakan kumpulan *thread*. *Thread* dalam satu blok dapat berbagi data melalui memori yang disebut memori bersama (*shared memory*). *Thread* dalam satu blok dapat disinkronisasikan dengan memanggil fungsi *_syncthreads()*. Blok di dalam grid tidak bisa disinkronasikan. Jika sinkronisasi dibutuhkan diantara semua *thread*, maka pekerjaan harus dipecah menjadi dua *fungsi kernel* yang berbeda, dan dua *fungsi kernel* yang berbeda tidak dieksekusi secara paralel. Ukuran dimensi grid dan dimensi blok ditentukan pada saat menjalankan/memanggil fungsi *kernel.* Misalkan dimensi grid gridDim dan dimensi blok blockDim, maka bentuk pemanggilan fungsi kernel:

Kode program 2.3

```
kernel_func<<<gridDim, blockDim>>>(argumen1,
argumen2, . . . . , . . . .)
```



Gambar 2.9 Model software CUDA (Manual CUDA, 2008).

Masing-masing *thread* dalam satu blok diidentifikasi melalui indeks *thread*-nya yang merupakan nomor *thread* dalam blok tersebut. Untuk membantu dalam pengelamatan komplek yang menggunakan indeks *thread*, sebuah aplikas dapat menspesifikasikan sebuah blok dalam array 2 atau 3 dimensi dengan ukuran tertentu bergantung pada spesifikasi GPU. Untuk blok 2 dimensi dengan ukuran (D_x, D_y), id *thread* dari *thread* dengan index(x,y) adalah ($x + yD_x$) dan untuk blok berukuran 3 dimensi dengan ukuran dimensi (D_x, D_y, D_z), id *thread* dengan indeks (x, y, z) adalah ($x + yD_x + zD_xD_y$) *(Breitbart, 2007)*.

Pada saat fungsi kernel dipanggil, semua blok akan didistribusikan ke semua SM. Tiap-tiap blok dipetakan ke satu SM. Jumlah blok di dalam grid bergantung pada data yang akan diproses. Agar GPU berjalan efisien, jumlah blok minimal sama dengan jumlah MP *(Breitbart, 2007)*.

Sebuah grid dieksekusi di dalam GPU dengan menjadualkan blok *thread* ke dalam multiprosesor. Jadi, tiap-tiap blok dipetakan ke satu buah multiprosesor. Multiple blok *thread* dapat dipetakan pada multiprosesor yang sama dan kemudian dieksekusi secara besamaan. Jika multiple blok *thread* dieksekusi pada satu multiprosesor yang sama, maka sumberdaya-nya, seperti *register* dan *shared memory* akan dibagi ke semua blok *thread.* Ini membatasi jumlah blok yang bisa dipetakan pada multiprosesor yang sama. Sebuah blok akan tetap tinggal di dalam multiprosesor sampai eksekusi *kernel-nya* selesai *(Breitbart, 2007)*.

Blok *thread* dipecah menjadi kelompok SIMD (*Single Intruction Multiple Data*) yang disebut warp. Setiap *warp* berisi jumlah *thread* yang sama. Jumlah *thread* di dalam sebuah warp ditentukan oleh spesifikasi *hardware. Warp* dieksekusi dengan menjadualkannya pada prosesos didalam multiprosesor, sehingga *warp* dieksekusi dalam model SIMD. GPU *k*eluaran NVIDIA terbaru mempunyai ukuran warp 32*,* dimana tiap multiprosesor mempunyai 8 prosesor. Sehingga satu intruksi membutuhkan minimal 32/8=4 *clock cycles (Breitbart, 2007)*.

Salah satu aspek yang penting dalam pemograman CUDA adalah menejamen memori. Sebuah *thread* yang dieksekusi di *device* hanya mempunayi akses pada DRAM *device* dan *on-chip memory*. Masing-masing memori mempunyai kecepatan akses dan ukuran yang berbeda-beda seperti yang ditunjukkan berikut ini *(Svetlin,dkk, 2008)* :

- Baca/tulis per-*thread*, register (cepat, ukuran terbatas)
- Baca/tulis per-thread, memori lokal (lambat, tidak di-*cache*, ukuran terbatas)
- Baca/tulis per-blok, memori *shared* (cepat, ukuran terbatas)
- Baca/tulis per-grid, memori global (lambat, tidak di-*cache*, berukuran besar)
- Hanya baca per-grid, memori konstant (lambat, di-*cache*,

ukuran terbatas)

- Hanya baca per-grid, memori textur (lambat, di-*cache*, berukuran besar).

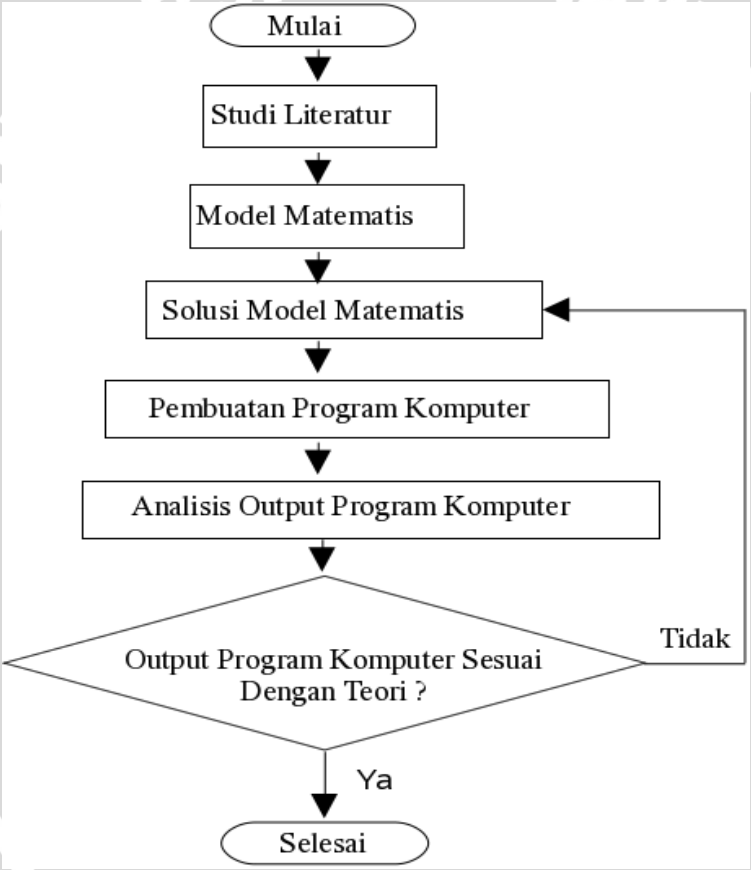
Pemilihan memori yang akan digunakan dalam kernel tergantung banyak faktor, seperti kecepatan akses, ukuran yang dibutuhkan, dan operasi yang akan dilakukan dalam menyimpan data. Keterbatasan yang penting adalah ukuran dari *shared memory*. Tidak seperti pemrograman CPU, programer perlu secara eksplisit meng-*copy* data dari memori global ke *shared memory*, dan sebaliknya. Akan tetapi, arsitektur terbaru mengijinkan operai *scather* dan *gather*. Operasi *gather* merupakan kemampuan dalam mengakses/membaca sembarang lokasi memori pada saat fungsi kernel berjalan. Operasi *scather* merupakan kemampuan mengakses/ menulis sembarang lokasi memori. Ketaktersediaan operasi *scather* merupakan malasah utama pada openGL ketika digunakan dalam pemrograman GPU. Dalam pemrograman CUDA, keseluruhan performa aplikasi bergantung pada manejemen memori *(Svetlin ,dkk, 2008)*.

BAB III
METODOLOGI PENELITIAN

3.1 Waktu Dan Tempat Penelitian

Penelitian ini dimulai pada tanggal 1 Desember 2008 sampai tanggal 30 Mei 2009. Penelitian dilakukan di Laboratorium Komputer, Jurusan Fisika, Universitas Brawijaya.

3.2 *Flowchart* Penelitian



3.3 Alat Dan Bahan

Berikut ini spesifikasi komputer dan GPU untuk menguji program komputer :

Spesifikasi Komputer	
Prosesor	Intel Pentium(R) Dual CPU E2160 1.80 GHz
Memori	2.0 GB
Sistem operasi	Linux openSUSE 11.1
Spesifikasi GPU	
Model	GeForce 8600 GT
Jumlah total memori global	536543232 bytes
Jumlah SM	4
Jumlah SP (*Scalar Processor*)	32
Jumlah total memori *constant*	65536 bytes
Jumlah maksimum *shared memory* per blok	16384 bytes
Jumlah total register per blok	8192
Ukuran *warp*	32
Jumlah Maksimum *thread* per blok	512
Ukuran maksimum dimensi blok	512 x 512 x 64
Ukuran maksimum dimensi grid	65535 x 65535 x 1
Kecepatan *clock*	1.19 GHz

3.4 Tahap Pengerjaan

3.4.1 Model Matetamtis Fluida Tak Mampat

Persamaan *Navier-Stokes* (persamaan (2. 1)) dan persamaan kontinuitas (persamaan(2.3)) akan digunakan sebagai model matematis, yang ditulis kembali menjadi:

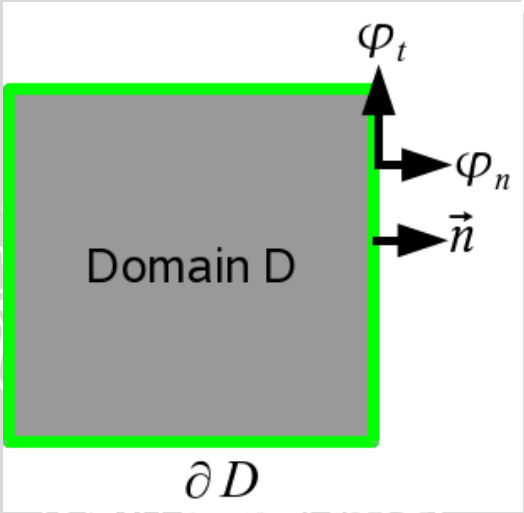
$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \nabla p + \sigma \nabla^2 \vec{u} + \vec{f} \quad (3.1)$$

$$\nabla \cdot \vec{u} = 0 \quad (3.2)$$

dimana σ (μ/ρ) viskositas kinematik. Viskositas kinematik menujukkan tingkat kekentalan fluida.

3.4.2 Solusi Model Matematis

Gambar 3.1 menunjukkan domain fisis yang digunakan dalam penelitian ini yaitu berupa kotak tertutup. Fluida dianggap terperangkap di dalam kotak dimana daerah batasnya dianggap berupa tembok padat.



Gambar 3.1. Domain fisis. Warna hijau menunjukkan daerah batas ∂D , φ_t dan φ_n komponen kecepatan pada daerah batas ∂D , $\vec{n}$ vektor normal pada daerah batas ∂D

Stam(1999) menggunakan teknik fluida stabil (*stable fluid*) yang lebih efisien untuk menyelesaikan persamaan *Naver-Stoke*. Bersasarkan teorema dekomposisi *Helmholtz-Hodge*, sembarang medan kecepatan $\vec{w}$ pada domain D (domain D dalam hal ini berupa kotak) dapat didekomposisi ke dalam bentuk :

$$\vec{w} = \vec{u} + \nabla p \quad (3.3)$$

dimana $\vec{u}$ mempunyai divergensi nol $(\nabla \cdot \vec{u} = 0)$ dan p merupakan medan skalar. Jadi medan vektor $\vec{w}$ merupakan jumlah dari medan konservasi massa dan medan gradien. Jika kedua ruas persamaan (3.3) dikenakan operator divergensi, maka akan diperoleh :

$$\nabla \cdot \vec{w} = \nabla^2 p \qquad (3.4)$$

Persamaan (3. 4) merupakan persamaan Poisson untuk medan skalar *p*. Solusi numerik persamaan (3. 5) diperoleh dengan mendiskritisasi operator *Laplacian* ∇^2 dan operator divergensi($\nabla \cdot$).

Misalkan mula-mula $\vec{w}$ sama dengan $\vec{u}$ dan nilai $\vec{w}$ diperoleh dengan menyelesaikan persamaan :

$$\frac{\partial \vec{w}}{\partial t} = -(\vec{w} \cdot \nabla)\vec{w} + \sigma \nabla^2 \vec{w} + \vec{f} \qquad (3.6)$$

Nilai medan kecepatan $\vec{u}$ diperoleh dengan mengurangi medan kecepatan $\vec{w}$ dengan gradien tekanan ∇p , yaitu :

$$\vec{u} = \vec{w} - \nabla p \qquad (3.6)$$

Persamaan (3. 6) merupakan solusi dari persamaan *Navier-Stokes*.

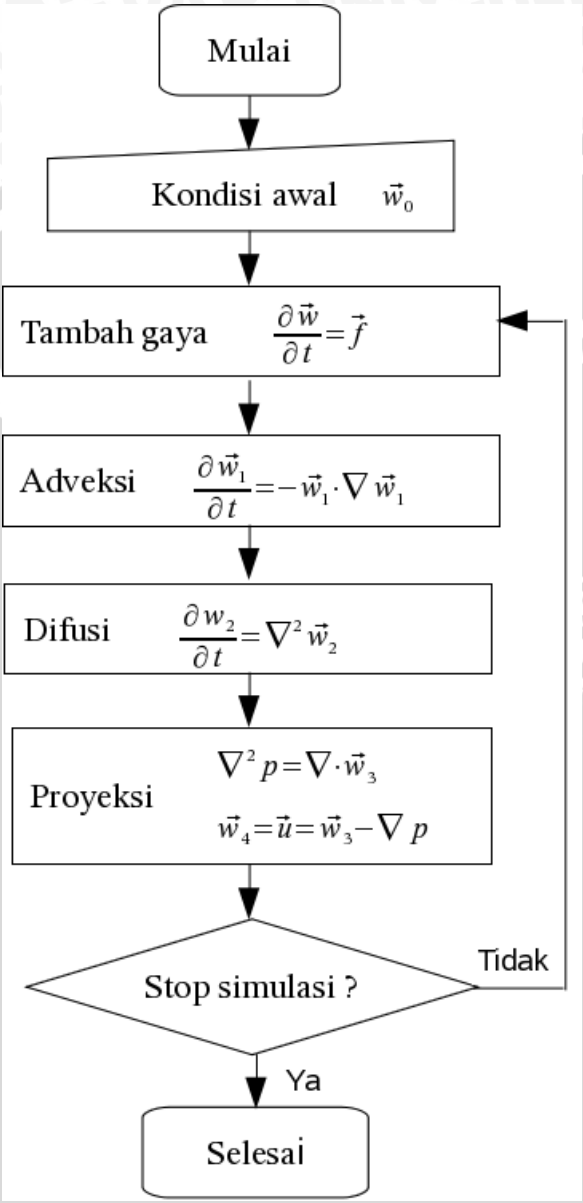
Solusi persamaan *Navier-Stokes* membutuhkan kondisi syarat batas. Untuk medan kecepatan digunakan kondisi syarat batas ***tak- tergelincir*** dimana fluida dianggap diam pada daerah batas ∂D . Dari sini diperoleh :

$$\varphi_n(x, y) = 0 \quad , \quad \varphi_t(x, y) = 0 \quad .$$

Untuk tekanan digunakan kondisi syarat batas ***Neumann*** yaitu :

$$\nabla p \cdot \vec{n} = 0$$

Dibutuhkan empat langkah berdasarkan pada persamaan (3. 4) dan persamaan (3. 6) untuk mengevaluasi medan kecepatan dengan kondisi awal $\vec{w}_0 = \vec{u}(\vec{x}, 0)$. Gambar 3. 2 menunjukkan *flowchart* untuk mendapatkan medan kecepatan $\vec{u}$ pada saat $t + \Delta t$.



Gambar 3. 2 . Flowchart metode fluida stabil

Berikut ini penjelasan *flowchart* Gambar 3.2

1. Penentuan kondisi awal

   Misalkan pada saat t = 0, $\vec{u}_0 = \vec{u}(\vec{x}, 0)$ yang merupakan kondisi awal medan kecepatan. Kondisi awal harus memenuhi persamaan kontinuitas. Misalkan pula solusi pada saat t adalah $w_0 = \vec{u}(\vec{x}, t)$.

2. Penambahan gaya luar

   Efek gaya luar pada aliran fluida, digambarkan persamaan

   $$\frac{\partial \vec{w}}{\partial t} = \vec{f}$$

   Jika gaya dianggap tidak berubah selama waktu Δt, maka diperoleh

   $$\vec{w}_1 = \vec{w}_0 + \Delta t \vec{f} \qquad (3.7)$$

3. Adveksi

   Adveksi dapat dipandang sebagai jenis "*transport* kecepatan", atau sebagai medan kecepatan men-transport dirinya sendiri. Suku adveksi yaitu :

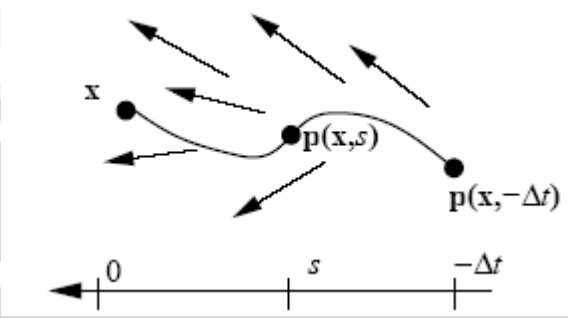
   $$\frac{\partial \vec{w}_1}{\partial t} = -\vec{w}_1 \cdot \nabla \vec{w}_1$$

   mengambarkan penjalaran ganguan pada medan kecepatan. Suku ini membuat persamaan *Navier-Stokes* menjadi non-linear, sehingga menimbulkan kompleksitas dan kesulitan dalam komputasi dinamika fluida. Suku adveksi dapat diselesaikan dengan menggunakna metode beda hingga dengan mendiksritisasi operator nabla ∇ , akan tetapi hal ini membutuhkan angkah waktu yang cukup kecil untuk menjaga kesetabilan penyelesaian numerik. Stam menggunakan teknik "*semi-langrangian*", yang didasarakan pada *metode karakteristik* yang biasa digunakan dalam penyelesaian persamaan diferensial *(Andersson, 2005)*.

   Pada setiap satu langkah waktu, semua partikel fluida digerakkan oleh kecepatan fluida itu sendiri. Sehingga untuk memperoleh kecepatan fluida pada posisi **x** pada saat $t + \Delta t$, titik **x** di-*backtrace* melalui medan $\mathbf{w}_1$ di sepanjang waktu Δt. Proses ini menghasilkan jejak **p** yang sama dengan

$\boldsymbol{x}$-$\boldsymbol{w}_1$ Δt, sehingga diperoleh :

$$\vec{w}_2 = \vec{w}_1\left(\vec{x} - \vec{w}_1\,\Delta t\right) \qquad (3.\ 8)$$



Gambar 3. 3. Untuk menyelesaikan suku adveksi, masing-masing titik di-trace ke waktu yang lalu. Kecepatan pada posisi $\boldsymbol{x}$ *merupakan kecepatan partikel fluida pada waktu* Δt *pada lokasi* $\boldsymbol{p(x, -\Delta t)}$ *(Stam, 1999).*

4. Difusi

Suku difusi mempunyai efek penghalusan pada medan kecepatan fluida. Bentuk standar persamaan difusi adalah :

$$\frac{\partial \vec{w}_2}{\partial t} = \sigma \nabla^2 \vec{w}_2$$

Penyeselesaian numerik yang biasa digunakan dalam menyelesaikan persasamaan difusi di atas adalah dengan mendeksritisasi operator ∇^2 dan kemudian menggunakan metode eksplisit untuk menyelesaikan turunan terhadap waktu. Akan tetapi metode ini tidak sstabil jika vikositas σ cukup besar. Stam (1999) menggunakan metode implisit untuk menyelesaikan persamaan difusi, yaitu :

$$\left(I - v\Delta t\,\nabla^2\right)\vec{w}_3 = \vec{w}_2 \qquad (3.\ 9)$$

dimana **I** merupakan matrik identitas. Jika operator ∇^2 didiskritisasi, maka akan dihasilkan sistem persmaan linear dalam bentuk **Ax = b.**

5. Proyeksi

Proses proyeksi digunakan untuk membuat medan kecepatan yang diperoleh pada langkah ke empat mengikuti hukum kontinuitas (persamaan (3.2)). Langkah proyeksi membutuhkan penyelesaian dan persamaan yaitu :

$$\nabla^2 p = \nabla \cdot \vec{w}_3 \qquad (3.\ 10)$$

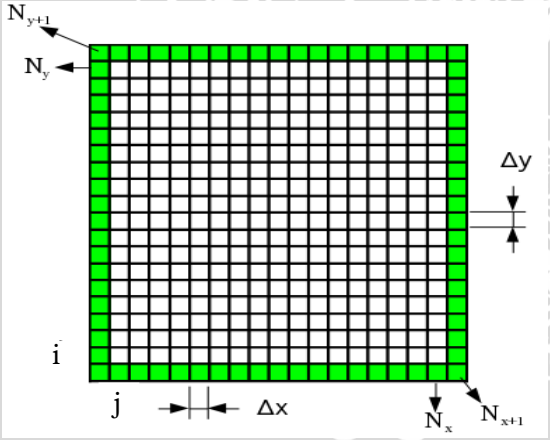
$$\vec{w}_4 = \vec{w}_3 - \nabla p \qquad (3.\ 11)$$

3.4.3 Diskritisasi

Gambar 3. 1 menunjukkan domain fisis yang digunakan dalam penelitian ini. Domain fisis dibagi menjadi Nx sel dalam arah x dan Ny sel dalam arah y, sehingga jarak antar grid adalah :

$$\Delta x = \frac{L}{\mathrm{Nx}} \quad , \quad \Delta y = \frac{T}{\mathrm{Ny}} \quad .$$

dimana *L* lebar kotak dan *T* tinggi kotak. Gambar 3. 4 menunjukkan hasil diskritisasi domain fisis. Jika *f*(x,y) adalah fungsi pada domain fisis maka f(iΔx, jΔy) adalah bentuk fungsi tersebut pada domain komputasi.



Gambar 3. 4. Domain komputasi (sel-sel berwarna hijau digunakan untuk menyimpan kondisi syarat batas)

Staggered grid merupakan metode beda hingga yang banyak digunakan dalam disktritisasi penyelesaian numerik persamaan *Navier-Stokes*. Konsekuensi penggunaan *staggered grid* adalah tidak

semua eksternal grid terletak pada batas domain. Misalnya batas kiri dan kanan tidak mempunyai komponen aray y dari medan kecepatan $\vec{u}$, demikian juga sebaliknya batas atas dan batas bawah tidak mempunyai komponen arah x dari medan kecepatan $\vec{u}$. Untuk itu diperlukan sel tambahan untuk menyimpan nilai dari kondisi syarat batas. Sel (0,j) dan sel(Nx+1, j) menyimpan kondisi syarat batas sebelah bawah dan atas, sedangkan sel(i,0) dan sel(Ny+1,0) menyimpan kondisi syarat batas sebelah kiri dan kanan.

Diskritisasi persamaan (3.9) menggunakan metode beda hingga menghasilkan :

$$u3_{i-1,j} - u3_{i+1,j} + \beta\, u3_{i,j} - u3_{i,j-1} - u3_{i,j+1} = \alpha\, u2_{i,j} \quad (3.\ 12a)$$

$$v3_{i-1,j} - v3_{i+1,j} + \beta\, v3_{i,j} - v3_{i,j-1} - v3_{i,j+1} = \alpha\, v2_{i,j} \quad (3.\ 12b)$$

dimana $\Delta x = \Delta y$, $\alpha = \dfrac{(\Delta x)^2}{\sigma \Delta t}$, $\beta = 4 + \alpha$, u3 dan v3 komponen vektor $\vec{w}_3$ dalam arah x dan y, u2 dan v2 komponen vektor $\vec{w}_2$ dalam arah x dan y

Diskritisasi persamaan (3. 10) menghasilkan

$$p_{i-1,j} + p_{i+1,j} - \beta\, p_{i,j} + p_{i,j-1} + p_{i,j+1} = \alpha(\nabla \cdot \vec{w}_3) \quad (3.\ 13)$$

dimana $\Delta x = \Delta y$, $\alpha = -(\Delta x)^2$, $\beta = 4$, dan

$$\nabla \cdot \vec{w}_3 = \frac{1}{2\Delta x}(u3_{i+1,j} - u3_{i-1,j} + v_{i,j+1} - v3_{i,j-1}) \ ,$$

dengan u3 dan v3 komponen medan kecepatan $\vec{w}_3$.

Sedangkan diskritisasi peramaan (3. 11) menghasilkan :

$$u4_{i,j} = u3_{i,j} - \frac{1}{\Delta x}(p_{i+1,j} - p_{i-1,j})$$
$$v4_{i,j} = v3_{i,j} - \frac{1}{\Delta y}(p_{i,j+1} - p_{i,j+1}) \quad (3.\ 14)$$

dimana u4 dan v4 komponen vektor $\vec{w}_4$ dalam arah x dan y. Persamaan (3. 12) dan (3. 13) dapat diubah ke dalam bentuk iterasi Jacobi, yaitu :

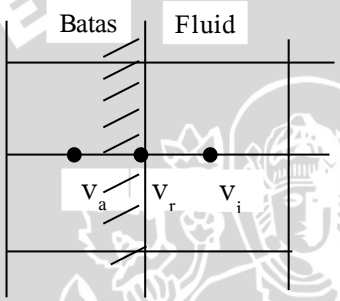
$$u_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} + \alpha b_{i,j}}{\beta} \quad (3.\ 15)$$

dengan *b* adalah suku disebelah kanan persamaan (3.12) dan (3.13).

Persamaan difusi dan persamaan divergensi membutuhkan kondisi syarat batas untuk medan kecepatan $\vec{u}$. Kondisi tak-tergelincir digunakan sebagai kondisi syarat batas. Kecepatan kontinus harus saling menghilangkan pada daerah batas untuk menyesuaikan dengan kondisi tak-tergelincir. Untuk nilai-nilai yang berada langsung pada daerah batas, diperoleh :

$u_{0,j} = 0,\ u_{Nx,j} = 0, \qquad j$=1....Nx

$v_{i,0} = 0,\ v_{i,Ny} = 0, \qquad i$=1....Ny



Karena daerah batas vertikal dan horizontal tidak mengandung nilai v dan u, maka nilai batas nol diterapkan pada kasus ini dengan merata-ratakan nilai di sebelah kiri dan sebelah kanan daerah batas

$v_r = (v_a + v_i)/2 = 0 \ \rightarrow\ v_a = - v_i$

Dari sini akan diperoleh dari keempat daerah batas, yaitu :

$u_{i,0} = -u_{i,1} \qquad u_{i,\ Ny+1} = -u_{Nx,j} \qquad i$=1....Nx

$v_{0,j} = -v_{1,j} \qquad v_{Nx+1,j} = -v_{i,Ny} \qquad j$=1....Ny

Kondisi syarat batas *Neumann* digunakan untuk menyelesaikan persamaan *Poisson* untuk tekanan, yaitu :

$$\frac{\partial p}{\partial \vec{n}} = 0 \quad \text{atau} \quad \nabla p \cdot \vec{n} = 0 \ .$$

Ini artinya, kecepatan perubahan tekanan yang tegak lurus terhadap daerah batas sama dengan nol. Untuk daerah batas sebelah kiri dengan $\vec{n}$ = $[-1,0]^T$, diperoleh :

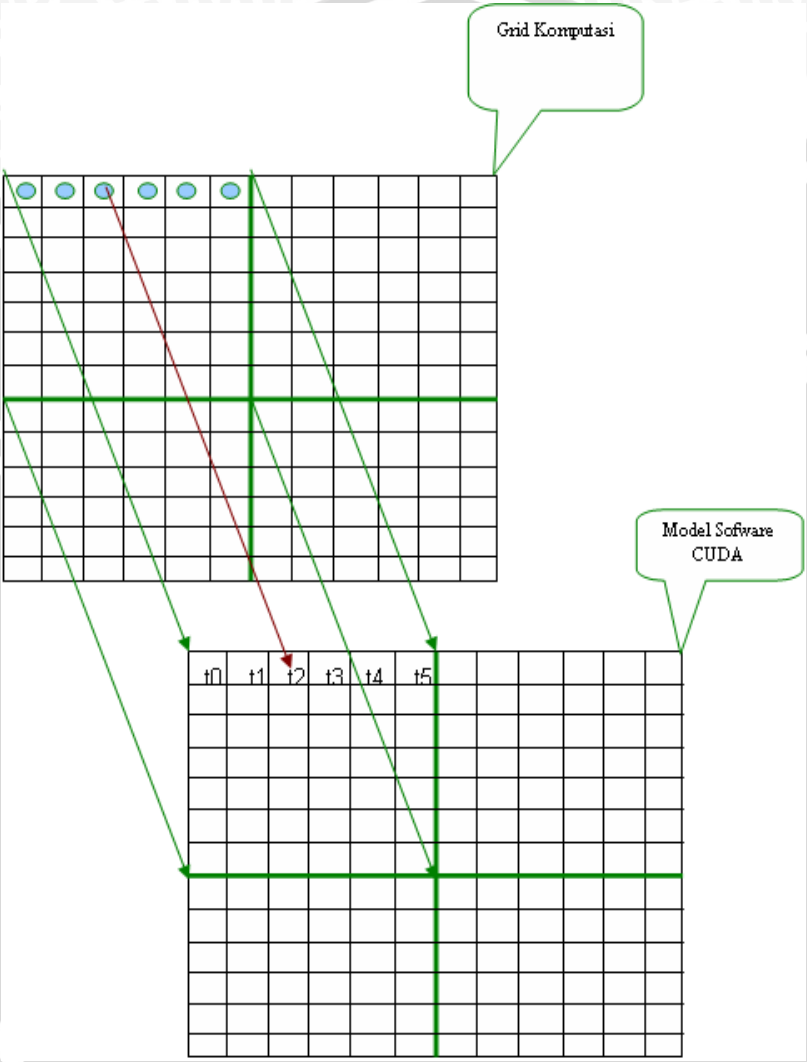
$$p_{0,j} - p_{1,j} = 0 \quad \text{atau} \quad p_{0,j} = p_{1,j} \qquad j = 1.....Ny$$

Untuk kondisi syarat batas di sebelah kanan ($\vec{n}$ = $[1,0]^T$), bawah ($\vec{n}$ = $[0,-1]^T$), dan atas ($\vec{n}$ = $[0,1]^T$) berturut-turut diperoleh :

$$p_{Nx+1,j} = p_{Nx,j} \qquad j = 1.....Ny$$

$$p_{i,0} = p_{i,1} \quad , \quad p_{i,Ny+1} = p_{i,Ny} \quad i = 1.....Nx$$

3.4.4 Implementasi

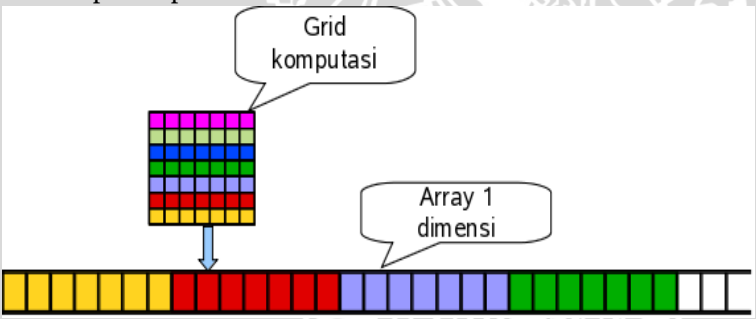


Gambar 3. 5. Pemetaan grid komputasi ke model software CUDA. Satu atau lebih subdomain dipetakan ke satu blok dan satu titik di dalam subdomain dipetakan ke satu thread di dalam blok CUDA

Paralelisasi menggunakan metode dekomposisi domain. Domain komputasi (Gambar 3. 4) dibagi menjadi subdomain-subdomain. Satu atau lebih subdomain akan ditangani oleh satu multiprosesor GPU. Sedangkan satu subdomain dipetakan ke satu blok dalam model *software* CUDA dan satu titik pada domain komputasi dipetakan ke satu *thread* di dalam sebuah blok. Gambar 3. 5 menunjukkan proses pemetaan domain komputasi ke model *software* CUDA. Berdasarkan skenario paralelisasi seperti ini, sebagian besar fungsi kernel menggunakan konfugurasi grid dan blok *threads* seperti berikut ini :

- Domain komputasi dibagi menjad $\left[\frac{N_x}{16} x \frac{N_y}{16}\right]$ yang dijadikan sebagai dimensi grid
- Ukuran dimensi blok sama dengan 16x16

Dalam perograman CUDA, pemahaman akan struktur data sangatlah penting. Ini akan membantu dalam hal meningkatkan kecepatan komputasi pada GPU.



Gambar 3. 6. Tranformasi data grid 2 dimensi ke array 1 dimensi

Gambar 3.6 menunjukkan skema penyimpanan data. Data grid 2 dimensi disimpan di dalam *array* 1 dimensi. Format penyimpanan data menggunakan format ***row major storage***, artinya penyimpanan data berdasarkan baris pada grid. Untuk mengakses data pada array 1 dimensi digunakan rumus :

$$indeksArray = i * Nx + j \qquad i = 1....Nx,\ j = 1.....Ny$$

dimana *i* dan *j* koordinat data dalam arah x dan y. Berdasarkan skema paralel yang digunakan, berarti indeks *i* dan *j* sama dengan indeks *thread* di dalam grid CUDA. Untuk mencari indeks *thread* di dalam grid CUDA digunakan rumus :

```
i = blockDim.x * blockIdx.x + threadIdx.x;
j = blockDim.y * blockIdx.x + threadIdx.y;
```

dimana blockDim.x, blockIdx.x, threadIdx.x, blockDim.y, blockIdx.y, dan threadIdx.y menunjukkan dimensi blok, indeks blok dan indeks *thread* dalam arah x dan y. Variabel-variabel ini merupakan variabel *build-in* artinya variabel-variabel ini otomatis tercipta pada saat fungsi kernel dijalankan, dan programer hanya mempunyai hak "baca" terhadap variabel-variabel ini.

Secara umum bentuk fungsi yang digunakan untuk menghitung proses adveksi, difusi dan proyeksi pada CPU adalah :

Kode program 3.1

```
void CPU_func (arg1, arg2,  ... , ..., ){
    for i = 1 to Nx do {
        for  j = 1 to Ny do {
            //proses adveksi, difusi, atau poyeksi

        }
    }
}
```

Jika fungsi pada Kode program 3.1 dikonversi ke dalam bentuk fungsi kernel GPU, maka diperoleh :

Kode program 3.2

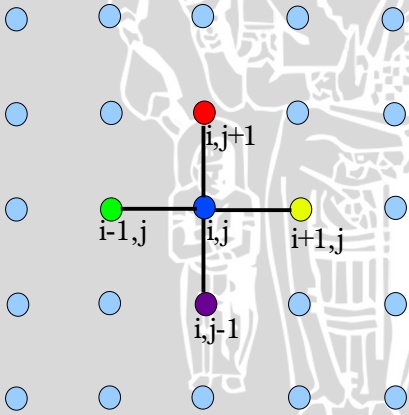
```
__global__ void GPU_func(arg1, arg2, ..., ...., ){
        i = blockDim.x * blockIdx.x + threadIdx.x + 1;
        j = blockDim.y * blockIdx.y + threadIdx.y + 1;
        // proses adveksi,difusi atau proyeksi
}
```

Penambahan nilai 1 pada indeks *i* dan *j* bertujuan agar masing-masing *thread* bekerja di dalam daerah syarat batas.

Iterasi Jacobi merupakan salah satu metode iterative yang banyak digunakan dalam menyeleseaikan sistem persamaan linear. Metode ini lebih mudah diimplementasikan dalam pemrograman GPU. Kekurangan metode ini adalah kecepatan konvergensi yang lambat nilai *error*-nya yang cukup besar. Penulis akan memfokuskan bagaimana memparalel iterasi Jacobi karena iterasi Jacobi membutuhkan waktu yang paling banyak dalam proses komputasi. Iterasi Jacobi digunakan untuk menyelesaikan persamaan difusi dan persamaan *Poisson*. Secara umum bentuk iterasi Jacobi yang digunakan dalam menyelesaikan persamaan difusi dan Poisson adalah :

$$w_{i,j}^{(k+1)} = \frac{w_{i-1,j}^{(k)} + w_{i+1,j}^{(k)} + w_{i,j-1}^{(k)} + w_{i,j+1}^{(k)} + \alpha b_{i,j}^{(k)}}{\beta} \quad 3.16$$

dengan b suku sebelah kanan hasil diskritisasi persamaan difusi (persamaan 3.12) dan Poisson (persamaan 3.13), dan k menunjukkan iterasi.



Gambar 3. 7. Hubungan antar data pada iterasi Jacobi (5-titik stensil)

Gambar 3. 7 menunjukkan hubungan antar data pada iterasi Jacobi. Pola data seperti ini banyak ditemukan pada solusi numerik persamaan direrensial. Pola seperti ini disebut juga *kode stensil* kare-

na untuk menghitung data pada sel(i,j) membutuhkan data tetangganya.

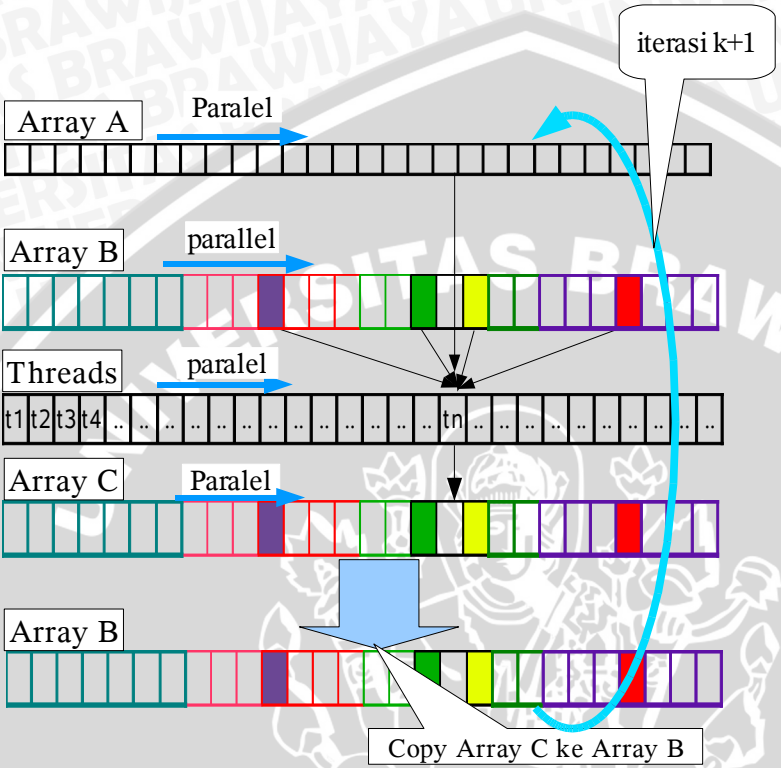
Persamaan (3.16) menunjukkan bahwa iterasi ke k+1 membutuhkan data pada iterasi ke k sehingga dibutuhkan proses sikronasi antara iterasi ke k+1 dan iterasi ke k. Sinkronasi yang paling mudah adalah menggunakan memori global GPU. Ini artinya proses iterasi dilakukan secara serial. Berdasarkan persamaan (3.16) maka dibutuhkan 3 buah array yaitu array A,array B dan array C. Array A digunakan untuk menyimpan data variabel b pada persamaan 3.16, array B digunakan untuk menyimpan data pada iterasi ke-k, dan array C digunakan untuk menyimpan data iterasi ke-k+1. Gambar 3. 8 menunjukkan proses iterasi Jacobi.

Berikut ini algoritma iterasi Jacobi :

1. Tentukan jumlah maksimum iterasi Jacobi.
2. Masing-masing *thread* men-copy data arary A dan array B dari global memori ke register.
3. Masing-masing *thread* menghitung data pada sel(i,j) berdasarkan persamaan 3.16
4. Masing-masing *thread* menyimpan data pada array C.
5. Copy data dari array C ke array B.
6. Kembali ke langkah 2 jika jumlah iterasi lebih kecil dari jumlah maksimum iterasi.

Langkah 2 sampai 4
diproses secara paralel

Gambar 3.8 menunjukkan proses iterasi Jacobi. Tanda panah ke arah kanan menunjukkan bahwa masing-masing *thread* menghitung iterasi ke k secara paralel. Setelah semua *thread* memproses data pada iterasi ke k maka proses dilanjutkan untuk iterasi ke k+1. Proses dari iterasi k ke iterasi k+1 dilakukan secara serial.



Gambar 3. 8. Proses iterasi Jacobi

BAB IV
PEMBAHASAN

Algoritma fluida stabil merupakan metode yang banyak digunakan untuk simulasi fluida di dalam GPU. Metode fluida stabil hanya merupakan metode pendekatan untuk menyelesaikan persamaan *Navier-Stokes*. Untuk keperluan dalam bidang teknik dan sain, penggunakan metode ini perlu dipertimbangkan. Kelebihan metode ini adalah lebih mudah diimplementsaikan dalam pemrograman GPU dan lebih stabil. Kesetabilan metode fluida stabil memungkinkan pemilihan satuan langkah waktu yang lebih besar dari pada metode konvensional (metode beda hingga atau metode elemen hingga) sehingga proses simulasi bisa berjalan lebih cepat.

Berdasarkan skenario paralel yang digunakan yaitu satu *thread* memproses satu data, jumlah maksimum data yang dapat diproses adalah :

*jumlah data maksimum = (gridDimMax_X * blockDim_X) * (gridDimMax_Y * blockDim_Y)*

gridDimMax_X dan gridDimMax_Y merupakan jumlah maksimum dimensi grid dalam arah x dan y. Nilai *gridDimMax_X dan gridDimMax_Y* bergantung pada spesifikasi GPU, sedangkan blockDim_X dan blockDim_Y merupakan dimensi blok yang digunakan, dalam hal ini dimensinya adalah 16x16. Untuk meningkatkan jumlah data yang dapat diproses maka skenario paralel dapat diubah dimana satu *thread* memproses banyak data.

4.1 Iterasi Jacobi

Penyelesaiaan numerik persamaan difusi (persamaan 3.9) dan Poisson (persamaan 3.4) menghasilkan sistem persamaan linear dengan bentuk **Ax = b**. Penyelesaian sistem persamaan linear ini menggunakan iterasi Jacobi. Metode iterasi Jacobi lebih mudah diimplementasikan dari pada metode yang lain, seperti metode iterasi Gauss Seidel, multigrid, atau *conjugate gradient*. Kekurangan metode iterasi Jacobi adalah kecepatan konvergensi lambat sehingga dibutuhkan iterasi yang banyak untuk mendapatkan hasil yang akurat. Untuk keperluan teknik dan sains metode iterasi Jacobi perlu dipertimbangan untuk digunakan. Akan tetapi penggunakan metode iterasi Jacobi dalam penelitian ini menghasilkan simulasi yang sudah realistis.

Proses iterasi Jacobi menggunakan memori global untuk mensinkronisasikan proses iterasi antara iterasi ke k dan iterasi ke k+1. Penggunaan memori global untuk sinkronisasi membuat proses iterasi dilakukan secara serial. Program 4.1 menunjukkan sample program iterasi Jacobi.

```
    KodeProgram  4.1
for iterasi=0 to maksIterasi do
{
    __global__ void iterasi_jacobi(float* arrayA, float* arrayB,
               float* arrayC, float* rhs);//jalankan fungsi kernel
    //copy arrayC ke arrayB
}
```

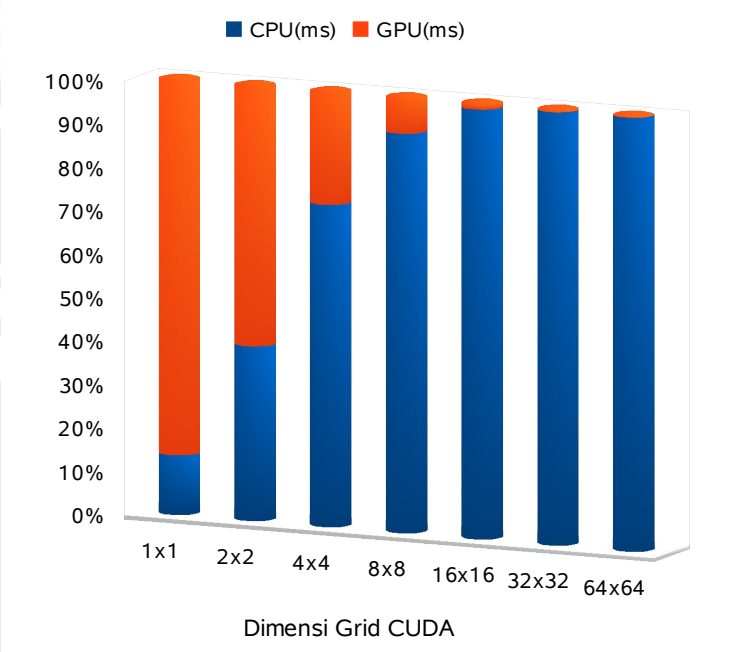
Kekurangan algoritma iterasi Jacobi ini adalah dalam hal penggunaan dan manejemen memori. Global memori mempunyai waktu akses yang lambat dan tidak di *cache*. Biasanya *shared memory* digunakan sebagai memori *cache* untuk memori global. Penggunaan shared memory sebagai *cache* untuk memori global dapat mempercepat akses ke memori global.

Proses iterasi Jacobi dapat juga diparalel dengan menggunakan *shared memory* sebagai tempat proses sinkronisasi. Penggunaan *shared memory* untuk proses sinkronisasi memerlukan sinkronisasi komunikasi antar blok kerena jenis dekomposisi domain yang digunakan adalah jenis dekomposisi domain tumpang- tindih (lihat Gambar 2.6).

Tabel 4.1 Perbandingan waktu eksekusi jumlah iterassi Jacobi antara CPU dan GPU

Jumlah iterasi	100			
Dimensi blok	16x16			
Dimensi Grid Komputasi	Dimensi Grid CUDA	CPU (ms)	GPU (ms)	Perbandingan kecepatan (CPU/GPU)
16x16	1x1	0,150	0,944	0,15
32x32	2x2	0,654	0,972	0,67
64x64	4x4	2,718	0,946	2,87
128x128	8x8	10,951	0,953	11,49

256x256	16x16	96,824	1,014	95,48
512x512	32x32	440,915	1,010	436,55
1024x1024	64x64	1824,17	1,003	1818,72



Gambar 4.1 Perbandingan waktu eksekusi iterasi Jacobi antara CPU dan GPU

Untuk mengetahui kemampuan komputasi paralel pada GPU maka diperlukan pengujian kemampuan komputasi GPU. Iterasi Jacobi digunakan untuk menguji kemampuan komputasi paralel pada GPU. Pemilihan iterasi Jacobi sebagai penguji dikarenakan iterasi Jacobi merupakan proses yang paling banyak memakan waktu. Tabel 4.1 dan Gambar 4.1 menunjukkan waktu eksekusi iterasi Jacobi pada CPU dan GPU.

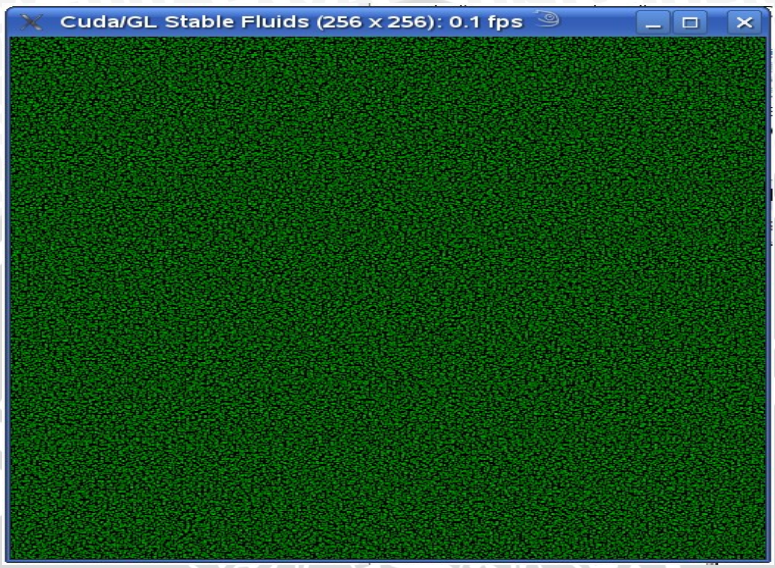
Tabel 4.1 dan Gambar 4.1 menunjukkan bahwa semakin besar dimensi grid, waktu proses komputasi iterasi Jacobi di CPU meningat drastis. Ini dapat dijelaskan dengan menghitung jumlah *looping* / perulangan pada proses iterasi Jacobi. Misalkan, pada dimensi grid 128x128 dengan iterasi 100 maka jumlah perulangannya adalah :

128x128x100 = 1.638.400

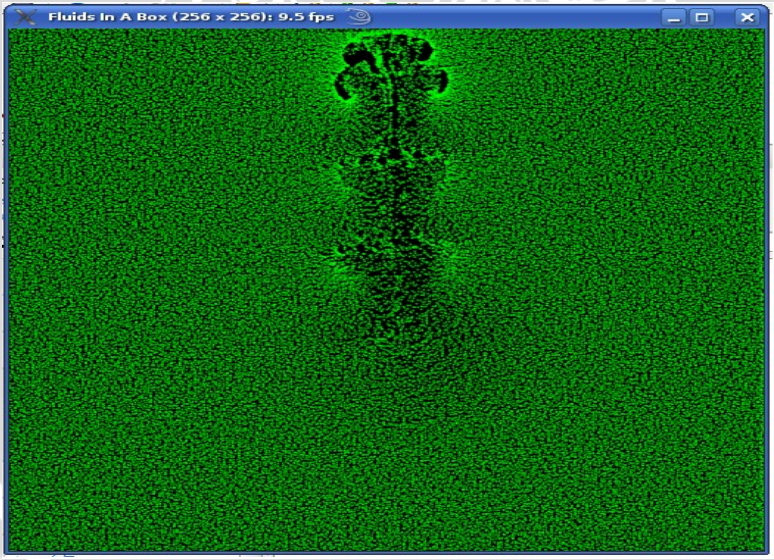
Sedangkan waktu komputasi iterasi Jacobi pada GPU relatif tidak berubah terhadap kenaikan besar dimensi grid. Hal ini menunjukkan kemampuan GPU dalam meningkatkan kecepatan komputasi. Kenaikan dimensi grid justru akan meningkatka performa komputasi pada GPU.

Tabel 4.1 juga menunjukkan perbandingan waktu komputasi iterasi Jacobi antara CPU dan GPU. Perbandingan kecepatan komputasi antara CPU dan GPU meningkat seiring dengan kenaikan dimensi grid. Pada dimensi grid 16x16 dan 32x32 kecepatan komputasi pada CPU lebih tinggi dari pada GPU, karena pada dimensi grid 16x16 jumlah blok sama dengan satu sehingga tidak semua SM GPU bekerja. Sedangkan pada dimensi grid 32x32 jumlah blok adalah 4 yang sama dengan jumlah SM. Ini menunjukkan bahwa optimasi akan dicapai jika jumlah blok lebih besar dari jumlah SM GPU sehingga semua SM dijaga tetap dalam keadaan bekerja. Sebagai catatan, program iterasi Jacobi ini hanya menggunakan register dan memori global GPU untuk menyimpan data komputasi.

4.2 Simulasi



a. Keadaan fluida mula-mula



b.Gerakan fluida setelah mouse digerakkan searah dengan tanda panah

Gambar 4.2. Proses simulasi

Proses visualisasi dan simulasi menggunakan metode *path line* atau *intergral curve*. Posisi partikel pada saat $t+\Delta t$ dapat dicari dengan persamaan $x(t+\Delta t)=x(t)+\int v(x(t))\,dt$. Karena nilai medan kecepatan hanya tedapat pada posisi tertentu di dalam grid, maka nilai medan kecepatan diinterpolasi menggunakan metode bilinear interpolasi untuk medapatkan nilai medan kecepatan pada sembarang posisi.

Gambar 4.2 menujukkan proses simulasi. Dimensi kotak adalah 1x1 satuan. Fluidanya adalah sejenis oli dengan viskositas mekanik sama dengan 0.000194 satuan. Proses simulasi dimulai dengan memberikan medan kecepatan mula-mula sama dengan nol. Ganguan diberikan dengan menggerakkan *mouse* komputer di permukaan fluida. Gambar 4.2.a menunjukkan keadaan mula-mula fluida. Tanda panah menunjukkan arah gerakan *mouse* komputer. Gambar 4.2.b menunjukkan gerakan fluida setelah *mouse* digerakkan (di-*drag*).

Tabel 4.2 Hubungan iterasi Jacobi dengan kecepatan simulasi

No	Iterasi Jacobi	Kecepatan Simulasi (*frames per second/ FPS*)
1	5	31.5
2	10	18.5
3	15	12.4
4	20	10.0
5	25	8.1
6	30	6.8
7	35	5.7
8	40	5.2
9	45	5.0
10	50	4.2
11	55	3.8
12	60	3.4

13	65	3.5
14	70	3.0
15	75	2.8
16	80	2.7

Tabel 4.2 menunjukkan hubungan antara iterasi Jacobi dengan kecepatan simulasi (dihitung dalam *frames per second/FPS*). Salah satu masalah dalam animasi aliran fluida adalah jumlah *frame* yang dibutuhkan untuk mendapatkan pandangan yang jelas terhadap pola aliran fluida. Menurut Frits dan Walsum (1993) , FPS yang dibutuhkan untuk mendapatkan pola aliran fluida yang jelas minimal 25 FPS. Dari Tabel 4.2 nilai FPS diatas 25 didapatkan pada saat iterasi sama dengan 5. Menurut Haris (2007), akurasi yang baik diperoleh jika jumlah iterasi antara 40 sampai 80. Iterasi dibawah 20 tidak direkomendasikan kerena nilai *error*-nya cuku besar. Sedangkan iterasi antara 30 sampai 80 nilai FPS yang diperoleh cukup kecil sehingga simulasi tidak berjalan mulus. Pilihan yang terbaik adalah menggunakan iterasi 20 atau 25. Dari percobaan simulasi, nilai iterasi iterasi 20 atau 25 sudah cukup memadai walaupun FPS-nya dibawah 25, hal ini mengakibatkan gerakan fluida tidak begitu mulus.

Dari percobaan simulasi ditemukan bahwa kecepatan fluida pada saat simulasi mula-mula tidak benar-benar sama dengan nol (walaupun kecepatan fluida sudah di-*seting* sama dengan nol), terutama pada daerah batas sebelah bawah. Hal ini desebabkan masih adanya *bug* pada program komputer. *Bug* ini berasal dari manejemen memori GPU yang masih kurang baik, mengingat manejemen memori merupakan aspek yang sangat penting pada pemrograman GPU. *Bug* ini juga berasal dari skenario paralel yang digunakan yaitu satu *thread* memproses satu data. Hal ini mengakibatkan kecepatan komputasi tidak terlalu optimal. Hal-hal yang bisa dilakukan untuk meningkatkan performa komputasi paralel pada GPU yaitu :

- C*oalesced memory accsess* yaitu metode untuk meningkatkan *bandwidth* akses memori global.
- Menggunakan *shared memory* sebagai tempat menyimpan data sementara.
- Jumlah blok *thread* harus lebih besar atau sama dengan

jumlah multiprosessor sehingga tidak ada multiprosesor dalam keadaan *idle* (menganggur). Adanya multiprosesor dalam keadaan *idle* bisa menurunkan performa komputasi secara drastis.

- Satu *thread* memproses lebih dari satu data sehinga jumlah data yang diproses tidak bergantung pada maksimum dimensi grid CUDA.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan pada hasil penelitian ini dapat ditarik beberapa kesimpulan sebagai berikut :

1. Penggunaan iterasi Jacobi dalam menyelesaikan persamaan linear yang dihasilkan dari diskritisasi persamaan difusi dan persamaan Poisson sudah cukup memadai dalam menghasilkan simulasi yang realistis.
2. Optimasi eksekusi di dalam GPU akan tercapai jika jumlah blok sama atau lebih besar dari jumlah multiprosesor GPU.
3. Kecepatan komputasi iterasi Jacobi pada GPU lebih cepat dari CPU, bergantung pada besar dimensi grid.

5.2 Saran

Ada beberapa rekomendasi atau saran yang dapat diajukan untuk penelelitian selanjutnya, yaitu :

1. Pengembangan penelitian ini untuk kasus tiga dimensi.
2. Komputasi paralel berbasis GPU tidak hanya digunakan dalam fluida akan tetapi dapat digunankan untuk bidang-bidang fisika yang lain seperti geofisika, biofisika atau mekanika kuantum.
3. Mengekploitasi kemampuan *shared memory* GPU sebagai *cache* untuk meningkatkan kecepatan proses komputasi.
4. Menggunakan metode yang lain untuk simulasi fluida seperti metode SPH (*Smooth Particles Hydronamic*) dan metode LBM (*Lattice Boltzmann Method*).
5. Masih ada *bug* dari program simulasi. Untuk itu, perlu adanya perbaikan untuk penelitian selanjutnya.

DAFTAR PUSTAKA

Andersson. L. 2005. Real-Time Fluid Dynamics for Virtual Surgery. Chalmers University of Technology. Goteborg

Anonimous. NVDIA CUDA Programming Guide. www.nvdia.com/object/cuda_home.html, diakses 8/21/2008

Anonimous. Navier-Stokes Equation. http://en.wikipedia.org/wiki/ Navier-Stokes_equations, diaskes 20/12/2008

Breitbart, J. 2007. A framework For easy CUDA Integration in C++ Applications. Kassel University. Kassel. German

Bridson, R. dan Muller-Fischer, M. 2007. Fluid Simulation. SIAGGRAP.

Frits H. P, dan van Walsum. T. 1993. FLUID FLOW VISUALIZATION. Focus on Scientific Visualization. Berlin

Griebel, M. Dornseifer, T. Neunhoeffer, T. 1998. Numerical Simulation In Fluid Dynamics: A Practical Introduction. Society For Industrial and Applied Mathematics. USA

Grama. A, Gupta. A, Karypis. G, dan Kumar. V . 2003. Introduction to Parallel Computing, Second Edition. Addison Wesley.

Halfhill. T. R. 2008. Parallel Processing With CUDA. w*ww.certext.com*

Haris. M. J . 2007. Fast Fluid Dynamics Simulation on the GPU(GPU Gems 3 Chapter 38). NVDIA. USA

Kreith. F, Berger. S. A,dkk. 1999. Fluid Mechanics, Mechanical Engineering Handbook. CRC Press LLC.

Knight. D. 2003. Parallel Computing In Computational Fluid Dynamics. w*ww.certext.com*

Mitchell. M, Oldham. J, dan Samuel.A . 2001. Advaced Linux Programming. New Rider Publishing. USA

Quinn J. M. 1994. Parallel Computing: Teory And Practice, 2 Ed. McGraw-Hill: USA

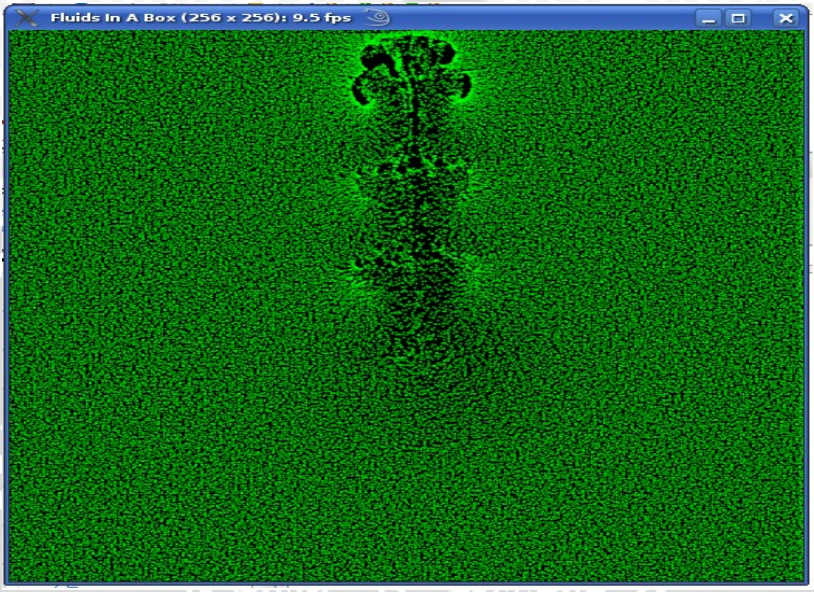
Ryoo. S, Rodrigues. C. I, Sara. S. B, Stone. S. M, Kirk. D. B, dan Wen-mei. W. H. 2008. Optimization Principles and Applica tion Performance Evaluation of a Multithreaded GPU Using CUDA. w*ww.certext.com*

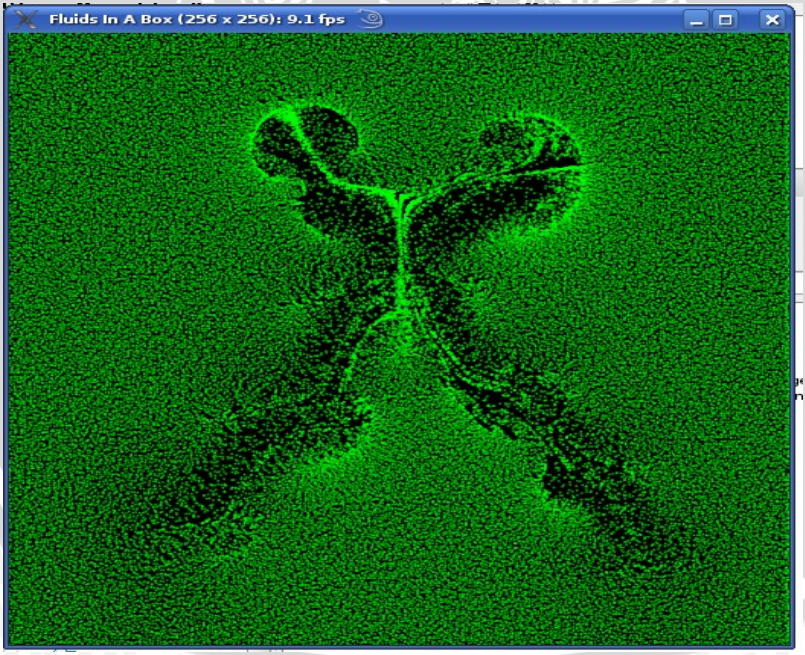
Stam.J. 1999. Stable Fluid. http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf, diakses 06/12/2008

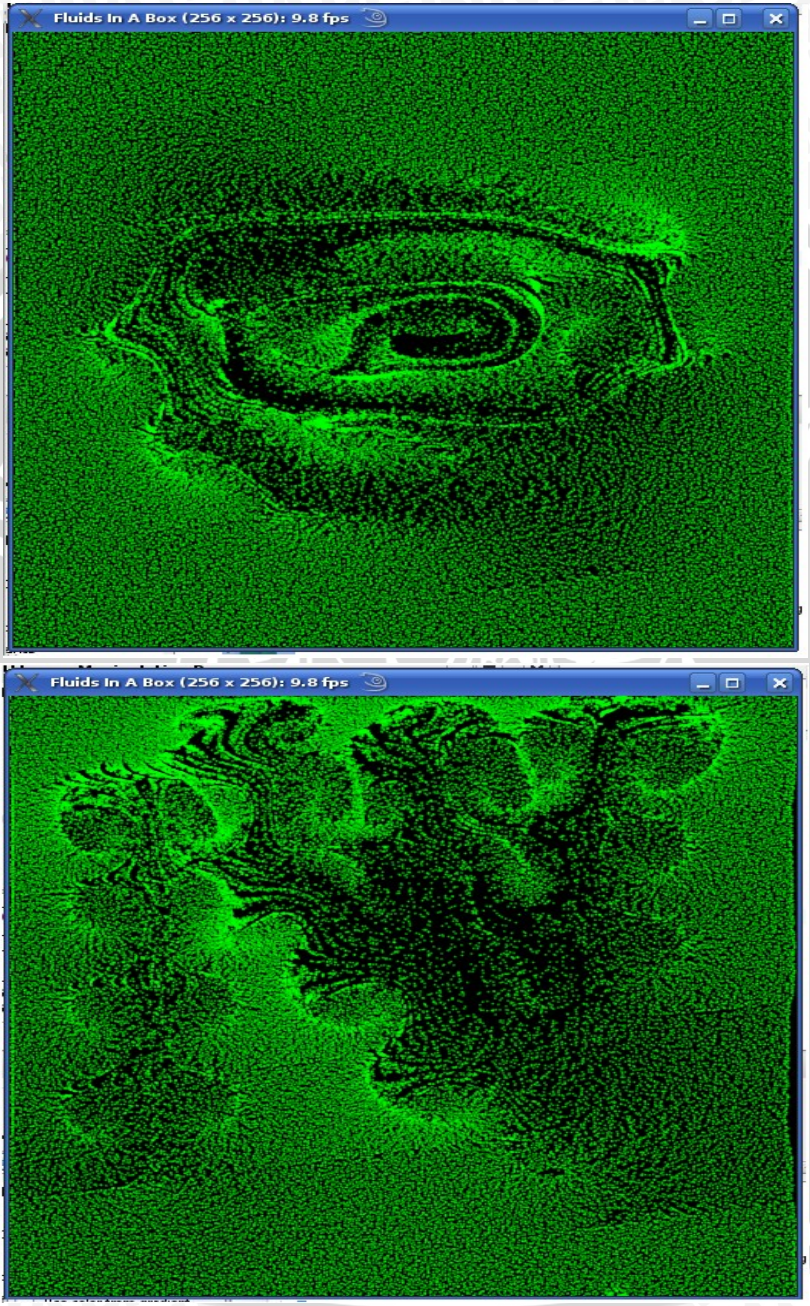
Svetlin A. M. dan Giorgio, V. 2008. CUDA Compatible GPU Cards As Efficient Hardware Accelerators For Smith-Waterman Squence Alignmen. http://www.biomedcentral.com/1471-2105/9/S2/S10, diakses 9/5/2008

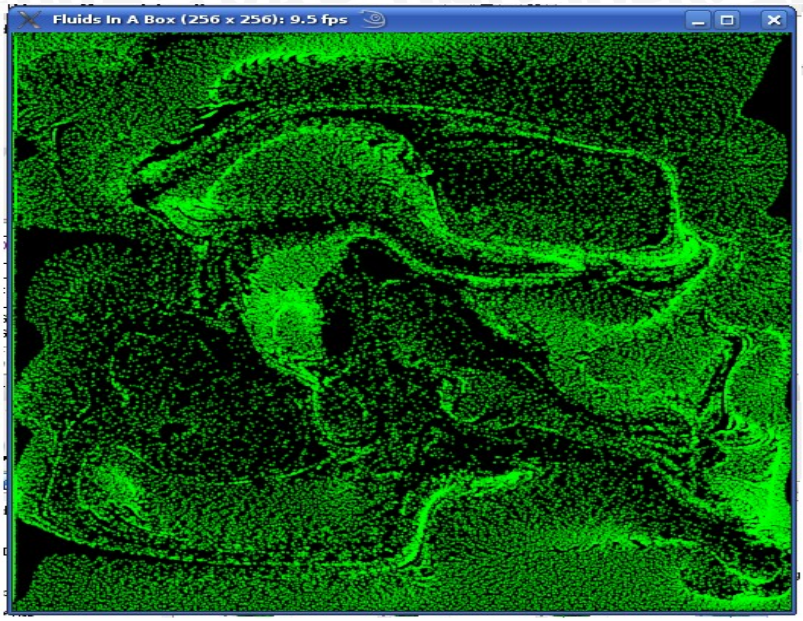
White. F. M. 2000. Fluid Mechanics, 4Ed. McGraw-Hill. USA

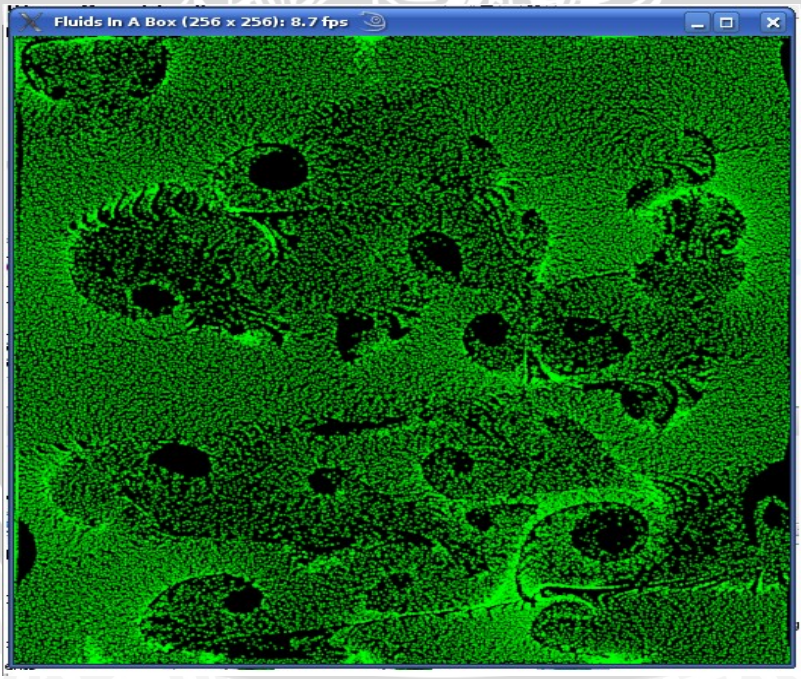
Lampiran 1 : Gambar Simulasi






Fluids In A Box (256 x 256): 9.8 fps
Fluids In A Box (256 x 256): 9.8 fps


Fluids In A Box (256 x 256): 9.5 fps


Fluids In A Box (256 x 256): 8.7 fps

UNIVERSITAS BRAWIJAYA

Lampiran 2 : Listing Program

```
/*
        fluid_kernel.cu
        Beberapa bagian dari program ini diambil dari contoh
        program bawaan CUDA SDK.
*/

// Referensi texture untuk pembacaan medan kecepatan
texture<float2, 2> texref;
static cudaArray *array = NULL;

void checkError( char *err ){
   CUT_CHECK_ERROR(err);
}

void checkMsg( char *msg){
   cutilCheckMsg( msg );
}

// fungsi untuk konfigurasi memori texture
// x dan y : dimensi texture
void setupTexture(int x, int y) {
   // Wrap mode appears to be the new default
   texref.filterMode = cudaFilterModeLinear;
   cudaChannelFormatDesc desc =
cudaCreateChannelDesc<float2>();

   cudaMallocArray(&array, &desc, y, x);
   checkError("cudaMalloc failed");
}

//fungsi untuk mem-bind/mengkaitkan memori texture ke array
CUDA
void bindTexture(void) {
   cudaBindTextureToArray(texref, array);
```

UNIVERSITAS BRAWIJAYA

```
    CUT_CHECK_ERROR("cudaBindTexture failed");
}

void unbindTexture(void) {
    cudaUnbindTexture(texref);
    checkError("cudaUnbindTexture failed");
}

//fungsi untuk mengupdate memori texture
void updateTexture(float2 *data, size_t wib, size_t h, size_t pitch) {
    cudaMemcpy2DToArray(array, 0, 0, data, pitch, wib, h,
cudaMemcpyDeviceToDevice);
    checkError("cudaMemcpy failed");
}

//fungsi untuk menghapus isi memori texture
void deleteTexture(void) {
    cudaFreeArray(array);
    checkError("cudaFreeArray failed");
}

//fungsi untuk menambah vektor gaya konstan ke medan kecepatan
// v : Medan kecepatan dalam arah x dan y.
//     Kecepatan dihitung dengan persamaan v(x,y,t+1) = v(x,y,t) + dt
// fx,fy : komponen vektor gaya
__global__ void
mouseForces_k(float2 *v, int spx, int spy, float fx, float fy, int r,
size_t pitch) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float2 *fj = (float2*)((char*)v + (ty + spy) * pitch) + tx + spx;

    float2 vterm = *fj;
    tx -= r; ty -= r;
    float s = 1.f / (1.f + tx*tx*tx*tx + ty*ty*ty*ty);
    vterm.x += s * fx;
    vterm.y += s * fy;
    *fj = vterm;
}
```

```
//Fungsi untuk menghitung proses adveksi
//u : Medan kecepatan
//dt : langkah waktu
//Nx, Ny : dimensi dalam arah x dan y
__global__ void
advect_k(float2 *u, float dt, int Nx, int Ny, size_t pitch){
        //mencari indeks threads di dalam grid
   int idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
   int idy = blockIdx.y * blockDim.y + threadIdx.y + 1;

   float2 vterm, ploc;

   //ambil data dari memori textur
   vterm = tex2D(texref, (float)idx, (float)idy);

    //cari posisi pada saat -dt
    ploc.x = (idx + 0.5) - (dt * vterm.x);
    ploc.y = (idy + 0.5) - (dt * vterm.y);

     //ambil data dari memori texture
     // berdasarkan lokasi plox.x dan ploc.y
     vterm = tex2D(texref, ploc.x, ploc.y);

     //set kecepatan saat ini sama dengan kecepatan pada saat -dt
     float2 *data = (float2*)((char*)u + idy * pitch) + idx;
     *data = vterm;

}

//Fungsi untuk menyelesaikan persamaan difusi.
//Penyelesaian ini menggunakan iterasi Jacobi.
__global__ void
diffusion_k(float2 *u, float2 *rhs, float alfa, float beta, int Nx, int Ny,
size_t pitch){
        //mencari indeks threads
   int idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
   int idy = blockIdx.y * blockDim.y + threadIdx.y + 1;
   //five stencils
```

```
    /*
            five stencils
                    south
                      |
             west---center--east
                      |
                    south
    */
    float2 west,east, south, north, center, crhs;

    //ambil data dari global memori, simpan di register
    west    = *(( float2* )(( char* )u + ( idy - 1 ) * pitch ) + idx );
    east    = *(( float2* )(( char* )u + ( idy + 1 ) * pitch ) + idx );
    south   = *(( float2* )(( char* )u + idy * pitch ) + idx - 1 );
    north   = *(( float2* )(( char* )u + idy * pitch ) + idx + 1 );
     crhs   = *(( float2* )(( char* )rhs + idy * pitch ) + idx );

      //tunggu sampai semua threads di dalam block
      //selesai mengambil data dari memori global
      __syncthreads();

      center.x = ( beta )*( west.x + east.x + south.x + north.x
                            + alfa * crhs.x );
      center.y = ( beta )*( west.y + east.y +  south.y + north.y
                           + alfa * crhs.y );

      //simpan hasil perhitungan di global memori
      float2 *vtmp = ( float2* )(( char* )u + idy * pitch ) + idx;
      *vtmp        = center;
}

//Fungsi untuk menghitung divergensi.
//Fungsi ini merupakan bagian dari proses proyeksi.
__global__ void
divergent_k(float *div, float2 *u, int Nx, int Ny, size_t pitch){
    //cari indeks thread di dalam grid
    int idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int idy = blockIdx.y * blockDim.y + threadIdx.y + 1;

    float2 uxr, uxl, vxr, vxl;
```

```
    //ambil data dari global memori,simpan di register
    uxl = *(( float2* )(( char* )u + idy * pitch ) + ( idx - 1 ));
    uxr = *(( float2* )(( char* )u + idy * pitch ) + ( idx + 1 ));
    vxl = *(( float2* )(( char* )u + ( idy - 1 ) * pitch ) + idx );
    vxr = *(( float2* )(( char* )u + ( idy + 1 ) * pitch ) + idx );


    //tunggu sampai semua threads di dalam block
    //selesai mengambil data dari memori global
    __syncthreads();

    //simpan data hasil perhitungan di global memory
    div[ idy * Nx + idx] = -0.5f * ( uxr.x-uxl.x+vxr.y-vxl.y ) /Nx;
}

//Fungsi untuk menghitung persamaan Poisson.
//Penyelesaiannya menggunakan iterasi Jacobi.
__global__ void
poisson_k( float *p, float *div, float alfa, float beta, int Nx, int Ny ){
   int idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
   int idy = blockIdx.y * blockDim.y + threadIdx.y + 1;
   //five points stencil
   float west, east, south, north, center;
   float divtmp;
   west  = p[ idy * Nx + ( idx - 1 ) ];
   east  = p[ idy * Nx + ( idx + 1 ) ];
   south = p[ ( idy - 1 ) * Nx + idx ];
   north = p[ ( idy + 1 ) * Nx + idx ];
   divtmp= div[ idy * Nx + idx ];
   center = (beta)*(west + east + south + north + alfa * divtmp );
   p[ idy * Nx + idx ] = center;
}



//Fungsi untuk mengurangkan nilai
//medan kecepatan dengan gradient dari tekanan
```

```
//Fungsi ini merupakan bagian dari proses proyeksi
__global__ void
subtract_k(float2 *u, float *p, int Nx, int Ny, size_t pitch){
   int idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
   int idy = blockIdx.y * blockDim.y + threadIdx.y + 1;
   float px, py;
   float2 uterm;
   px = p[ idy * Nx + ( idx + 1 ) ] - p[ idy * Nx + ( idx - 1 ) ];
   py = p[ ( idy + 1 ) * Nx + idx ] - p[ ( idy - 1 ) * Nx + idx ];
   float2 *utmp = ( float2* )(( char* )u + idy * pitch ) + idx;


    uterm = *utmp;
    uterm.x -= 0.5f * Nx * px;
    uterm.y -= 0.5f * Ny * py;
    *utmp = uterm;
}

//Fungsi untuk menghitung posisi partikel
//berdasarkan medan kecepatan yang sudah diperoleh.
__global__ void
advectParticles_k(float2 *part, float2 *u, float dt, int Nx, int Ny,
size_t pitch){

   int gtidx = blockIdx.x * blockDim.x + threadIdx.x + 1;
   int gtidy = blockIdx.y * blockDim.y + threadIdx.y + 1;

   // gtidx is the domain location in x for this thread
   float2 pterm, vterm;
   pterm = part[ gtidy * Nx + gtidx ];

   int xvi = (int)(pterm.x * Nx);
   int yvi = (int)(pterm.y * Ny);
   vterm = *((float2*)((char*)u + yvi * pitch) + xvi);

   pterm.x += dt * vterm.x;
   pterm.x = pterm.x - (int)pterm.x;
   pterm.x += 1.f;
   pterm.x = pterm.x - (int)pterm.x;
   pterm.y += dt * vterm.y;
```

```
    pterm.y = pterm.y - (int)pterm.y;
    pterm.y += 1.f;
    pterm.y = pterm.y - (int)pterm.y;
    part[ gtidy * Nx + gtidx ] = pterm;
}

//fungsi untuk menentukan kondisi sayarat batas medan kecepatan
__global__ void setVelBnd_k(float2 *v, int imax, int jmax){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int index = idx + 1;

    //kondisi syarat batas
   //kecepatan yang menyinggung daerah batas bawah dan atas
   v[IX(0,index)].x      = -v[IX(1,index)].x;//bawah
   v[IX(imax+1, index)].x =-v[IX(imax, index)].x;//atas

   v[IX(index, 0)].y      = -v[IX(index, 1)].y;//left
   v[IX(index, jmax +1)].y = -v[IX(index,jmax)].y;//right

    //u velocity
    //sudut kiri bawah
    v[IX(0  ,0  )].x     =0.5f*(v[IX(1,0  )].x     +v[IX(0  ,1)].x);
    //sudut kiri atas
    v[IX(0  ,imax+1)].x   =0.5f*(v[IX(1,jmax+1)].x   +v[IX(0
                          ,jmax)].x);
    //sudut kanan bawah
    v[IX(jmax+1,0  )].x   =0.5f*(v[IX(imax,0)].x
                                 +v[IX(imax+1,1)].x);
    //sudut kanan atas
    v[IX(imax+1,jmax+1)].x=0.5f*(v[IX(imax,jmax+1)].y+v[IX(imax
                                 +1,jmax)].x);



    //v velocity
     /sudut kiri bawah
    v[IX(0  ,0  )].y     =0.5f*(v[IX(1,0  )].y     +v[IX(0  ,1)].y);/
     //sudut kiri atas
    v[IX(0  ,imax+1)].y   =0.5f*(v[IX(1,jmax+1)].y   +v[IX(0
                          ,jmax)].y);
```

```
    //sudut kanan bawah
    v[IX(jmax+1,0  )].y   =0.5f*(v[IX(jmax,0  )].y
                                 +v[IX(imax+1,1)].y);
    //sudut kanan atas
    v[IX(imax+1,jmax+1)].y=0.5f*(v[IX(imax,jmax+1)].y+v[IX(imax
                                 +1,jmax)].y);
}

//fungsi untuk menseting kodisi syarat batas untuk tekanan
__global__ void
setPressBnd_k(float *p, int imax, int jmax){

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int index = idx + 1;

     p[IX(0       ,index)]  = p[IX(1,index)];//batas kiri
     p[IX(imax + 1, index)] = p[IX(imax,index)];//batas kanan
     p[IX(index ,0  )]      = p[IX(index,1)];//batas bawah
     p[IX(index ,jmax + 1)] = p[IX(index, jmax)];//batas atas

    //sudut kiri bawah
    p[IX(0  ,0  )]    = 0.5f*(p[IX(1,0  )]+p[IX(0  ,1)]);
    //sudut kiri atas
    p[IX(0  ,imax+1)] = 0.5f*(p[IX(1,jmax+1)]+p[IX(0  ,jmax)]);
    //sudut kanan bawah
    p[IX(jmax+1,0  )]    = 0.5f*(p[IX(imax,0  )]+p[IX(imax+1,1)]);
    //sudut kanan atas
    p[IX(imax+1,jmax+1)] = 0.5f*(p[IX(imax,jmax+1)
                                 +p[IX(imax+1,jmax)]);
}

/*
        fluid.cu
*/
#define DT 0.09f  //langkah waktu
#define VISC 0.00012f //viskositas fluida
#define NX 256 //dimensi grid arah x
#define NY 256 //dimensi grid arah y
#define DS (NX+2)*(NY+2) //jumlah total data
#define BLOCK_X 16 //dimensi blok arah x
```

```
#define BLOCK_Y 16 //dimensi blok arah y
#define ITERASI 20 //jumlah iterasi Jacobi
#define IX(i,j) ((i)*(NX + 2) + (j)) //makro untuk mengakses memori
global
#include "fluid_kernel.cu" //include fungsi kernel

//deklarasi variabel GPU
float2 *dv = NULL;
float2 *dv0 = NULL;
float  *dp = NULL;
float  *ddiv = NULL;

//deklarasi variabel CPU
float2 *hv = NULL;
float2 *particles = NULL;
size_t pitch = 0;

//Fungsi untuk mengalokasikan memori
void allocateMemory(){
   //Alokasi memori di CPU
   hv = (float2*)malloc(DS * sizeof(float2));
   particles = (float2*)malloc( DS * sizeof(float2));
   if(hv == NULL && particles == NULL) {
      printf("allocate memory in host failed !");
      exit(1);
   }
   //Alokasi memori di GPU
   cudaMallocPitch((void**)&dv, &pitch, (NX + 2) * sizeof(float2),
                        (NY + 2));
   cudaMallocPitch((void**)&dv0, &pitch, (NX + 2) *
                        sizeof(float2), (NY + 2));
   cudaMalloc((void**)&dp, DS * sizeof(float));
   cudaMalloc((void**)&ddiv, DS * sizeof(float));

   setupTexture((NX +2), (NY + 2));//seting memori texture
   bindTexture(); //kaitkan memori texture ke array CUDA
}

//Fungsi untuk men-nol-kan medan kecepatan di GPU
void resetVelocity(){
```

```
    cudaMemset(dv,0, sizeof(float2) * DS);
}

//Fungsi untuk membebaskan memori di CPU dan GPU
void freeMemory() {
  //Bebaskan memori di CPU
   if( hv != NULL ) free(hv);
   if( particles != NULL) free(particles);

   //Bebaskan memori di GPU
   unbindTexture();
   deleteTexture();
   cudaFree(dv);
   cudaFree(dv0);
   cudaFree(dp);
   cudaFree(ddiv);
}

//Fungsi untuk menambahkan gaya/ganguan
//pada medan kecepatan menggunakan mouse
void mouseForces(int dx, int dy, int spx
                      , int spy, float fx, float fy, int r) {
   dim3 tids(2*r+1, 2*r+1);
   //panggil fungsi kernel
   mouseForces_k<<<1, tids>>>(dv, dx, dy, spx, spy, fx, fy, r, pitch);
   cutilCheckMsg("addForces_k failed.");
}

//Fungsi untuk proses adveksi
void advectVelocity( float2 *u, float dt, int Nx, int Ny ) {

   //Konfigurasi dimensi grid dan dimensi blok
   dim3 gridDim((( Nx + 2 ) / BLOCK_X )
                    + (!(( Nx + 2 ) % BLOCK_X ) ? 0 : 1),
                   (( Ny + 2 ) / BLOCK_Y )
                  + (!(( Ny + 2 ) % BLOCK_Y ) ? 0 : 1), 1);

   dim3 blockDim(BLOCK_X, BLOCK_Y, 1);

   updateTexture(u, ( Nx + 2 ) * sizeof(float2), (Ny + 2), pitch);
```

```
    //Panggil fungsi kenel
    advect_k<<< gridDim, blockDim >>> ( u, dt, Nx, Ny, pitch );
    checkMsg( "advect_k failed." );

    dim3 gridDimB(((Nx + 2) / BLOCK_X)
                  + (!((Nx + 2) % BLOCK_X ) ? 0 : 1));
    dim3 blockDimB(BLOCK_X);

    //tentukan kondisi syarat batas untuk kecepatan
    setVelBnd_k<<<gridDimB, blockDimB>>>(u, Nx, Ny);
    checkMsg( "setVelBnd_k failed." );
}

//Fungsi untuk proses difusi
void diffusVelocity( float2 *u, float2 *rhs, int Nx, int Ny , int iterasi)
{

    //Kofigurasi dimensi grid dan dimensi blok
    dim3 gridDim((( Nx + 2 ) / BLOCK_X )
                 + (!(( Nx + 2 ) % BLOCK_X ) ? 0 : 1),
                 (( Ny + 2 ) / BLOCK_Y )
                 + (!(( Ny + 2 ) % BLOCK_Y ) ? 0 : 1), 1);

     dim3 blockDim(BLOCK_X, BLOCK_Y, 1);

    //Konfigurasi dimensi grid dan dimensi blok
    //untuk menjalankan fungsi kernel kondisi syarat batas
    dim3 gridDimB(((Nx+2) / BLOCK_X)
                  +(!((Nx+2) % BLOCK_X ) ? 0 : 1));
    dim3 blockDimB(BLOCK_X);

    float alfa = 1.0f / (VISC * DT * (Nx + 2 ) * (Ny + 2 ));
    float beta = 1.0f / (4.0f + alfa);
    cudaMemcpy( rhs, u, DS * sizeof(float2),
          cudaMemcpyDeviceToDevice);
    checkError( " copy failed." );
    cudaMemset(u, 0, DS * sizeof(float2));

    //iterasi Jacobi
```

```
    int i;
    for ( i = 0; i < iterasi; i++ ) {
       //Pangil fungsi kernel
       diffusion_k<<< gridDim, blockDim >>> ( u, rhs, alfa, beta
                                                , Nx, Ny, pitch );
       checkMsg( "diffusion_k failed." );

        //Tentukan kondisi syarat batas untuk kecepatan
       setVelBnd_k<<<gridDimB, blockDimB>>>(u, Nx, Ny);
       checkMsg( " setVelBnd_k failed." );
    }

}

//Fungsi untuk proses proyeksi
void project( float2 *u, float *p, float *div, int Nx, int Ny , int iterasi)
{
    //Konfigurasi dimensi grid dan blok
    dim3 gridDim((( Nx + 2 ) / BLOCK_X )
                    + (!(( Nx + 2 ) % BLOCK_X ) ? 0 : 1),
                     (( Ny + 2 ) / BLOCK_Y )
                     + (!(( Ny + 2 ) % BLOCK_Y ) ? 0 : 1), 1);
    dim3 blockDim(BLOCK_X, BLOCK_Y, 1);

    dim3 gridDimB(((Nx+2) / BLOCK_X)
                      +(!((Nx+2) % BLOCK_X ) ? 0 : 1));
    dim3 blockDimB(BLOCK_X);

    //Panggil fungsi kernel
    divergent_k<<< gridDim, blockDim >>> (div, u, Nx, Ny, pitch );
    checkMsg( "divergent_k failed." );

    //Tentukan kondisi syarat batas tekenan
    setPressBnd_k<<<gridDimB, blockDimB>>>(div, Nx, Ny);
    checkMsg( "setPressBnd_k failed." );

    float alfa = 1.0f;
    float beta = 1.0f/4.0f;

    //lakukan itetasi Jacobi
```

```
    cudaMemset(p, 0, DS * sizeof(float));
    int i;
    for( i = 0; i < iterasi; i++ ){
      //panggil fungsi kernel
       poisson_k<<< gridDim, blockDim >>> ( p, div, alfa,
                                            beta, Nx, Ny );
       checkMsg( "poisson_k failed." );

      //tentukan kondisi syarat batas
       setPressBnd_k<<<gridDimB, blockDimB>>>(p, Nx, Ny);
       checkMsg( "setPressBnd_k failed." );
    }

    subtract_k<<< gridDim, blockDim >>> ( u, p, Nx, Ny, pitch );
    checkMsg( "subtract_k failed." );

    setVelBnd_k<<<gridDimB, blockDimB>>>(u, Nx, Ny);
    checkMsg( " setVelBnd_k failed." );
}



//Fungsi untuk adveksi partikel.
//Fungsi ini digunakan dalam proses visualisasi
void advectParticles( GLuint vbo, float2 *u, float dt, int Nx, int Ny )
{
    //Konfigurasi dimensi grid dan blok
    dim3 gridDim((( Nx + 2 ) / BLOCK_X )
                 + (!(( Nx + 2 ) % BLOCK_X ) ? 0 : 1),
                  (( Ny + 2 ) / BLOCK_Y )
                 + (!(( Ny + 2 ) % BLOCK_Y ) ? 0 : 1), 1);
    dim3 blockDim(BLOCK_X, BLOCK_Y, 1);

    float2 *p;
    cudaGLMapBufferObject((void**)&p, vbo);
    checkMsg("cudaGLMapBufferObject failed");

    advectParticles_k<<< gridDim, blockDim >>> (p, u, dt, Nx,
                                                Ny,pitch);
    checkMsg( "advectParticles_k failed." );
```

```
    cudaGLUnmapBufferObject(vbo);
    checkMsg("cudaGLUnmapBufferObject failed");
}

//Inisialisasi partikel
void initParticles(float2 *p, int Nx, int Ny){
    int i, j;
    for (i = 0; i <= Ny; i++) {
        //p[IX(i,j)].x = ((j-0.5)/Nx);
        for (j = 0; j <= Nx; j++) {

            p[IX(i,j)].x = ((j-0.5)/Nx) +
                (rand() / (float)RAND_MAX - 0.5f) / Nx;
            p[IX(i,j)].y = ((i-0.5)/Ny) +
                 (rand() / (float)RAND_MAX - 0.5f) / Ny;

        }
    }
}
void initSimulate(){
    memset(particles,0, DS * sizeof(float2));
    initParticles(particles, NX, NY);
    memset(hv, 0.0f, DS * sizeof(float2));

    cudaMemcpy(dv,hv, DS * sizeof(float2),
          cudaMemcpyHostToDevice);
}
//Fungsi untuk proses simulasi
void simulate(GLuint vbo) {

    advectVelocity( dv, DT, NX, NY );//adveksi
    project( dv, ddiv, dp, NX, NY, ITERASI );//proyeksi
    diffusVelocity( dv, dv0, NX, NY, ITERASI );//adveksi
    project( dv, ddiv, dp, NX, NY, ITERASI );//proyeksi

    advectParticles( vbo, dv, DT, NX, NY ); //adveksi partikel
}
```

```
/*      visual.cu
        Kode program visualisasi.
        Kode program ini merupakan hasil modifikasi
        dari kode program bawaan CUDA SDK
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <GL/glew.h>
#include <cufft.h>
#include <cutil_inline.h>
#include <cuda_gl_interop.h>

#if defined(__APPLE__) || defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#include "fluid.cu"
#define FORCE (5.8f*NX) // Force scale factor
#define FR     4        // Force update radius
```

```
//forward declaration
void display(void);
void click(int button, int updown, int x, int y);
void motion(int x, int y);
void reshape(int x, int y);
void keyboard( unsigned char key, int x, int y);
void idle(void);
void cleanup(void);

GLuint vbo = 0;
static int wWidth = 512;
static int wHeight = 512;

static int clicked = 0;
static int lastx = 0, lasty = 0;
static int fpsCount = 0;
static int fpsLimit = 1;
unsigned int timer;

void display(void) {

  cutilCheckError(cutStartTimer(timer));
   simulate(vbo);

   glClear(GL_COLOR_BUFFER_BIT);
   glColor4f(0.0f,1.0f,0.0f,1.0f); glPointSize(1);
   glEnable(GL_POINT_SMOOTH);
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA,
         GL_ONE_MINUS_SRC_ALPHA);
   glEnableClientState(GL_VERTEX_ARRAY);
   glDisable(GL_DEPTH_TEST);
   glDisable(GL_CULL_FACE);

   glBindBufferARB(GL_ARRAY_BUFFER_ARB, vbo);
   glVertexPointer(2, GL_FLOAT, 0,0);

   glDrawArrays(GL_POINTS, 0,  DS);
   glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
   glDisableClientState(GL_VERTEX_ARRAY);
```

```
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisable(GL_TEXTURE_2D);

    cutilCheckError(cutStopTimer(timer));
    glutSwapBuffers();

  //Menghitung nilai FPS
   fpsCount++;
   if (fpsCount == fpsLimit) {
      char fps[256];
      float ifps = 1.f / (cutGetAverageTimerValue(timer) / 1000.f);
      sprintf(fps, "Fluids In A Box (%d x %d): %3.1f fps",
      NX, NY, ifps);
      glutSetWindowTitle(fps);
      fpsCount = 0;
      fpsLimit = (int)max(ifps, 1.f);
      cutilCheckError(cutResetTimer(timer));
    }
    glutPostRedisplay();
}


void idle(void) {
    glutPostRedisplay();
}

void keyboard( unsigned char key, int x, int y) {
    switch( key) {
        case 27:
            exit (0);
        case 'r':
            resetVelocity(); //nol kan kecepatan
            initSimulate(); //inisialisasi simulasi

            cudaGLUnregisterBufferObject(vbo);
            cutilCheckMsg("cudaGLUnregisterBufferObject failed");

            glBindBufferARB(GL_ARRAY_BUFFER_ARB, vbo);

            glBufferDataARB(GL_ARRAY_BUFFER_ARB,
```

```
                        sizeof(float2) * DS,
                        particles, GL_DYNAMIC_DRAW_ARB);
        glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
        cudaGLRegisterBufferObject(vbo);
        cutilCheckMsg("cudaGLRegisterBufferObject failed");
        break;
        default: break;
    }
}


void click(int button, int state, int x, int y) {
    lastx = x; lasty = y;
    clicked = !clicked;
}

void motion (int x, int y) {
    // Convert motion coordinates to domain
    float fx = (lastx / (float)wWidth);
    float fy = (lasty / (float)wHeight);
    int nx = (int)(fx * NX);
    int ny = (int)(fy * NY);

    if (clicked && nx < NX-FR && nx > FR-1
                    && ny < NY-FR && ny > FR-1) {
        int ddx = x - lastx;
        int ddy = y - lasty;
        fx = ddx / (float)wWidth;
        fy = ddy / (float)wHeight;
        int spy = ny-FR;
        int spx = nx-FR;
       //tambahkan gaya menggunakan mouse
        mouseForces(NX, NY, spx, spy, FORCE * DT * fx, FORCE *
DT * fy, FR);
        lastx = x; lasty = y;
    }
    glutPostRedisplay();
}
```

```
void reshape(int x, int y) {
    wWidth = x; wHeight = y;
    glViewport(0, 0, x, y);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 1, 1, 0, 0, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glMatrixMode (GL_MODELVIEW);
    glutPostRedisplay();}
void cleanup(void) {
    cudaGLUnregisterBufferObject(vbo);
    cutilCheckMsg("cudaGLUnregisterBufferObject failed");
    freeMemory();
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
    glDeleteBuffersARB(1, &vbo);
}

int run(int argc, char **argv){
    // use command-line specified CUDA device, otherwise use device
with highest Gflops/s
    if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") )
        cutilDeviceInit(argc, argv);
    else
        cudaSetDevice( cutGetMaxGflopsDeviceId() );

    // Allocate and initialize host data
    GLint bsize;
     cutilCheckError(cutCreateTimer(&timer));
    cutilCheckError(cutResetTimer(timer));

    allocateMemory();
    initSimulate();
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowSize(wWidth, wHeight);
    glutCreateWindow("Fluids In A Box");
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(click);
```

```
    glutMotionFunc(motion);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);

    glewInit();
    if (! glewIsSupported(
       "GL_ARB_vertex_buffer_object"
                    )) {
       fprintf( stderr, "ERROR: Support for necessary OpenGL
          extensions missing.");
       fflush( stderr);
       return CUTFalse;
    }

    glGenBuffersARB(1, &vbo);
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, vbo);

    glBufferDataARB(GL_ARRAY_BUFFER_ARB,
       sizeof(float2) * DS,  particles, GL_DYNAMIC_DRAW_ARB);

    glGetBufferParameterivARB(GL_ARRAY_BUFFER_ARB,
          GL_BUFFER_SIZE_ARB, &bsize);
    if (bsize != (sizeof(float2) * DS))
       goto EXTERR;
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);

    cudaGLRegisterBufferObject(vbo);
    cutilCheckMsg("cudaGLRegisterBufferObject failed");
    atexit(cleanup);
    glutMainLoop();
    cudaThreadExit();
    return 0;
EXTERR:
    printf("Failed to initialize GL extensions.\n");
    cudaThreadExit();
    return 1;
}
int main(int argc, char **argv){
    run(argc, argv);
}
```