BAB II
TINJAUAN PUSTAKA

2.1. Tinjauan Pustaka

Visualisasi merupakan salah satu cara untuk merepresentasikan data (Hansen & Johnson, 2005) sebuah data bisa di dapatkan dari bermacam metode yang ada seperti simulasi (Weiskopf, 2006). Selain data untuk membuat sebuah visualisasi di butuhkan beberapa *tools* pendukung seperti perangkat lunak dan perangkat keras, perangkat lunak yang yang pasti di butuhkan adalah video *accelerator* atau yang sering kita kenal dengan VGA (*Video Graphics Accelerator*) yang berfungsi untuk menerjemahkan data ke dalam *output* layar monitor. OpenGL, paraview, blender, matlab, openCV dan lain lain merupakan *tools* aplikasi yang banyak di gunakan untuk menampilkan hasil visualisasi, seperti beberapa peneliti ini yang membuat visualisasi penelitiannya dengan beberapa *tools* tersebut (Thurey, 2007; Schreiber, 2010; Zhang et al., 2008). Ada beberapa metode numerik untuk membuat sebuah simulasi fluida di antarnya. *Smoothed particle hydrodynamics* (SPH), Level-set and NS (nevier-stokes), dan Lattice Boltzmann (LB), merupakan metode metode yang sering di gunakan untuk membuat simulasi fluida seperti Thurey (2007) dan Munoz (2010) yang menggunakan metode LB, Janicke & Scheuermann (2010) yang menggunakan metode SPH.

Keuntungan metode LB adalah kodenya lebih simpel di bandingkan dengan metode lainnya, eksekusi waktunya lebih cepat, dan akurasi yang di dapat tidak kalah dengan metode yang lain. Metode ini banyak di gunakan peneliti untuk mesimulasikan fluida cair, tidak hanya air namun semua yang memiliki

sifat *lattice* volume, seperti asap, minyak, udara (Thurey, 2007). Selain aliran satu fase Metode LB bisa juga di terapkan untuk simulasi fluida multifase (Mohamad, 2011; Sukop & Thorne, Jr., 2007),

Aliran multi fase dan multi komponen terjadi sekitar kita, di alam atau pun di industri. Multi fase merupakan kondisi di mana terdapat dua atau lebih jenis fluida dalam satu aliran, komponen yang di maksud adalah mengacu pada unsur kimia sehingga untuk *multiphase* satu komponen misal H_2O terdiri dari air dan uap air (Sukop & Thorne, Jr., 2007). Sebaliknya multi komponen terdiri dari unsur kimia yang terpisah seperti minyak dan air. Pembuatan simulasi satu cukup mudah karena hanya menerapkan satu model *lattice* yang sama. Namun untuk simulasi dua fase atau multi fase memiliki kesulitan tersendiri di sebabkan harus menghitung nilai kecepatan rata-rata yang berbeda pada tiap fasenya. Model yang yang banyak di gunakan adalah model yang di usulkan oleh Shan & Chen (1993), dan banyak peneliti yang menggunakan model tersebut di antaranya (Sukop & Thorne, Jr., 2007) (Huang et al., 2008) (Schreiber, 2010) (Huang et al., 2010). Model shan and chen relatif mudah di terapkan dalam pemodelan dua fase. Dalam penelitian ini menyimulasikan satu fase dan dua fase yang merupakan bagian dari multi fase.

Beban komputasi menjadi masalah tersendiri saat melakukan komputasi simulasi fluida. Masalah ini yang di alami oleh Fadlila Rohmadani (2010) dan Dharmawan dkk (2011) yang membuat simulasi gerakan air 3 Dimensi dengan metode Lattice Boltzmann, hasil yang di dapat memuaskan namun peneliti belum menerapkan komputasi paralel hingga membuat beban komputasinya besar.

Semakin besar beban komputasinya maka akan semakin lama waktu simulasinya, untuk mengatasinya adalah dengan menerapkan komputasi paralel pada programnya. Komputasi paralel merupakan teknik untuk membagi proses perhitungan yang di bebankan pada CPU kepada bagian pemroses lain. Tidak perlu menggunakan prosesor berkecepatan tinggi dengan membagi proses pada banyak prosesor dengan kecepatan rendah maka bisa mengurangi waktu proses simulasinya (Wehner, 2009).

Pada saat ini yang cukup populer dalam penggunaan komputasi paralel adalah penggunaan GPU CUDA (*Compute Unified Device Architecture*) CUDA sendiri merupakan *tools* GPU untuk keperluan umum (*non-grafis*). Arsitektur CUDA memungkinkan proses komputasi perangkat lunak berjalan di GPU buatan Nvidia, menurut Strzodka (2005) GPU merupakan kekuatan proses paralel pada akhir tahun ini. Telah banyak penelitian yang menggunakan teknik komputasi paralel seperti yang di lakukan oleh Schreiber (2010) Brodtkorb dkk. (2011), yang membuat simulasi dan visualisasi fluida GPU CUDA sebagai komputasi paralel.

GPU di rancang khusus untuk melakukan operasi secara paralel, karena transistor yang lebih banyak dari pada CPU di khususkan hanya untuk pengolahan data saja (Gebaly, 2011). Pemrograman CUDA sama seperti membuat program C biasa. Saat kompilasi, syntax C biasa akan di proses oleh compiler C, sedangkan *syntax* dengan *keyword* CUDA akan di proses oleh compiler CUDA (nvcc) (nvidia, 2012). Dengan itu proses yang di jalankan bisa lebih cepat.

Selain penggunaan perangkat keras atau *hardware* dalam visualisasi bahasa pemrograman juga berpengaruh. Dalam hal ini penggunaan OpenGL

sangat membantu dalam membuat visualisasi dari data hasil simulasi. openGL merupakan *software interface to graphics* (Schreiber, 2010; Gebaly, 2011).

Dari kajian pustaka di atas maka pada penelitian ini di usulkan pengembangan simulasi untuk membuat penelitian tentang metode Lattice Bolztmann dengan penerapan visualisasi fluida satu fase dan dua fase menggunakan pemrograman GPU CUDA dan mengkaji perbedaan waktu komputasi yang di butuhkan untuk menyelesaikan simulasi fluida satu dan dua fase.

2.2. Landasan Teori

2.2.1. Visualisasi

Bidang visualisasi di fokuskan pada penciptaan gambar yang menyampaikan informasi penting tentang data yang mendasari (Hansen & Johnson, 2005). Membuat visualisasi di perlukan sebuah data, sumber data bisa di peroleh dari, simulasi numerik, pengukuran data fisik atau database. (Weiskopf, 2006) dalam pembuatan visualisasi data mentah akan di ubah oleh tahap penyaringan *pipeline* menjadi data abstrak visualisasi, operasi penyaringan yang umum adalah denoising oleh konvolusi dengan filter *kernel* atau perbaikan data dengan segmentasi, tahap berikutnya adalah pemetaan visualisasi, yang membangun pencitraan dari data visualisasi, representasi pencitraan memiliki bagian di ruang dan waktu, dan berisi atribut yang bisa terdiri dari geometri warna, transparansi, refleksi, dan tekstur permukaan.

Visualisasi bisa di klasifikasikan menjadi 3 tipe (Myers, 1990) yang di kutip dalam buku visualization scientific parallel programs (1994).

1. *Scientific Visualization* mengacu pada penerapan aplikasi grafis yang berorientasi pada data seperti yang di hasilkan oleh simulasi super komputer dan alat alat ukur yang di gunakan dalam bidang astronomi, meotorologi, dan medis.
2. Visualisasi Program. Terdiri dari beberapa teknik yang di tentukan dengan cara yang konvensional dan representatif bergambar di gunakan untuk ilustrasi aspek yang berbeda dari program. Misalnya dalam menjalankannya terhadap perhitungan waktu. Visualisasi program awalnya hanya di gunakan untuk debugging dan pengajaran yang bertujuan untuk presentasi. Dalam konteks *advance computer system*, program *visualization* di gunakan sebagai salah satu untuk Monitoring dan tuning.
3. Visual Program adalah sebuah sistem yang di gunakan oleh pengguna untuk pemrograman satu atau dua (atau yang lebih tinggi) dimensi grafis.

2.2.2. Fluida

Fluida yang memiliki jarak molekul yang tidak terlalu rapat seperti pada benda padat dan gaya antar molekul yang lemah atau kita lebih mengenalnya dengan sebutan zat cair, zat cair (air, minyak, dan lain-lain) (Munson et al., 2004). Dengan kondisi molekul tersebut maka zat cair lebih mudah di deformasi (tapi tidak mudah di mampatkan). Karena hal tersebut fluida mempunyai sifat bisa mengalir dan memberikan sedikit hambatan terhadap perubahan bentuk (kanginan, 2007) temperatur. Tekanan di lambangkan dengan *P* (*pressure*). Dan T untuk temperatur.

2.2.3. Aliran fluida

a. Aliran satu fase (single phase)

Aliran fase tunggal hanya mewakili satu gerak fluida saja, yaitu gas atau cair. Dalam tesis ini, gerak fluida yang di gunakan adalah satu fase cair.

b. Aliran Multi fase (multiphase)

Aliran dalam banyak fase atau atau multi fase sering di jumpai dalam kehidupan sehari hari, dalam hal ini dua fase yang bagian dari multi fase, bisa berupa gas-cair, dan gas cair. Kriteria dari dua fase yaitu memiliki perbedaan kerapatan dan kerapatan pada tiap fase fluidanya. Dalam tesis ini kasus dua fase adalah mewakili uap air dan air.

2.2.4. Lattice Boltzmann Method (LBM)

Metode LB yang berawal dari sebuah persamaan Boltzman, Di namakan sesuai dengan penemunya ilmuwan asal Austria, Ludwig Boltzmann. Persamaan ini merupakan bagian fisika statistik klasik dan menggambarkan perilaku gas dalam sekala mikroskopis.

LBM merupakan pengembangan *cellular automata*, yang berarti fluida terbentuk dari banyak sel sejenis. Semua sel di perbaharui di setiap langkah waktu dengan aturan sederhana, dengan ikut memperhitungkan sel-sel di sekitarnya. LBM memodelkan fluida yang tak mampu-mampat (*incompressible*) di mana partikel fluida hanya dapat bergerak searah dengan *collousionr* kecepatan *lattice*. LB menjelaskan Model perilaku makroskopik cairan dalam fisika statistik, dan menjelaskan Model perilaku makroskopik cairan dalam fisika statistik.

$$\partial_t f + \vec{v}.\nabla f = \Omega(f) \quad (2.1)$$

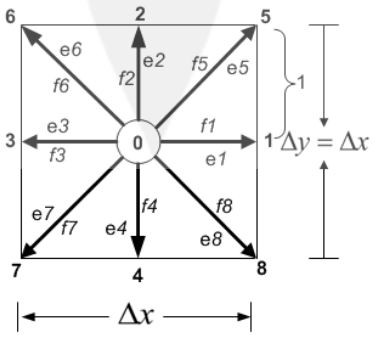
f adalah probabilitas fungsi distribusi

$\vec{v}$ kecepatan partikel

Ω ( ) suku perubahan tumbukan.

Model dalam metode Lattice Boltzmann biasa di lambangkan dengan D*n*Q*m*, di mana *n* menyatakan jumlah dimensi yang di gunakan dan *m* menyatakan jumlah arah *lattice* yang di gunakan. Metode Lattice Boltzmann menyediakan model-model *lattice* untuk di gunakan sesuai dengan ruang dimensi dan kebutuhan seperti model D1Q2 dan D1Q3, D1Q5, D2Q5 dan D2Q4, D2Q9, D3Q15, dan D3Q19 (Mohamad:2010).

Dalam penelitian ini, akan di bangun model simulasi perambatan gelombang menggunakan model Lattice Boltzmann D2Q9, yaitu berdimensi 2 dengan 9 arah *lattice*. Bisa di lihat pada Gambar 2.1, yang menunjukkan kartesian *lattice* dan kecepatan e_a di mana a = 0, 1,...,8 adalah indeks arah dan $e_0 = 0$ yang menunjukkan partikel saat diam. Setiap sisi dari sel memiliki panjang 1. Unit *lattice (lu)* adalah ukuran panjang dalam Metode Lattice Boltzmann dan selisih waktu (*ts)* adalah unit waktu.



Gambar 2.1 Visualisasi arah gerakan *lattice* D2Q9

Pada Gambar 2.1 di atas terdapat susunan fungsi f_i yaitu f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7,f_8. Vektor dengan nomor 0 mempunyai panjang 0 dan menyimpan jumlah partikel yang berhenti di sel berikutnya. Partikel ini tidak akan bergerak ke mana-mana di langkah waktu berikutnya, tetapi beberapa di antaranya mungkin akan di percepat (bergerak) karena tumbukan dengan partikel lain, jadi jumlah partikel yang diam bisa saja berubah.

Dari Gambar 2.1 di atas, kita juga dapat mendefinisikan sembilan kecepatan *ei* dalam model D2Q9 yang di definisikan sebagai berikut :

e0 = (0,0).c, e1 = (1,0).c, e2 = (0,1).c, e3 = (-1,0).c, e4 = (0,-1).c,

e5 = (1,1).c, e6 = (-1,1).c, e7 = (-1,-1).c, e8 = (1,-1).c,

atau bisa juga di defenisikan sebagai berikut :

e0 = (0,0).c, e1,3 = (±1,0).c, e2,4 = (0,±1).c, e5,6,7,8 = (±1, ±1).c,

di mana $c = \Delta x/\Delta t = \Delta y/\Delta t$. Di sini, Δt di gunakan untuk melihat selisih waktu (*ts*) untuk menghitung jarak gerak antar partikel. Setiap arah memiliki bobot, bobot arah tersebut adalah $_0\,\omega = 4/9$, $_{1,2,3,4}\,\omega = 1/9$, $_{5,6,7,8}\,\omega = 1/36$. Dalam bentuk persamaan dapat di tulis sebagai berikut :

$$\omega = \begin{cases} {}^4/_9, i = 0 \\ {}^1/_9, i = 1,2,3,4, \\ {}^1/_{36}, i = 5,6,7,8 \end{cases} \qquad (2.2)$$

dapat juga di tulis dengan persamaan :

$$\sum_{i=0}^{\beta-1} \omega_i = 1 \qquad (2.3)$$

Fungsi distribusi, ada dua nilai penting yang di hasilkan dengan cara menjumlah semua (9) fungsi distribusi, di dapatkan kepadatan (massa/volume) dari sel, dengan asumsi semua partikel mempunyai massa yang sama yaitu 1. Hasil yang penting lainnya adalah untuk setiap sel di dapat kecepatan dan arah ke mana partikel akan cenderung bergerak dari setiap sel. Kepadatan momentum perlu di hitung, yaitu jumlah dari semua fungsi distribusi partikel, tetapi setiap distribusi harus di kalikan dengan vektor *lattice*. Sehingga, fungsi distribusi partikel 0 akan di kalikan dengan vektor (0,0) yang selalu menghasilkan 0, fungsi distribusi f_1 di kalikan dengan (1,0) dan di tambahkan dengan fungsi distribusi f_3 di kalikan dengan vektor (-1,0) dan begitu seterusnya. Dari proses perhitungan di atas di dapatkan hasil dalam vektor 2 dimensi yang panjangnya di tentukan oleh kepadatan volume. Cukup dengan membagi momentum kepadatan dengan kepadatan sehingga di dapatkan vektor kecepatan untuk satu sel. Untuk awal simulasi, kepadatan di beri nilai 1. Karena LBM di gunakan untuk melakukan simulasi fluida yang tidak dapat di padatkan, artinya nilai kepadatan di setiap bagian dari fluida adalah konstan, keterikatan ini merenggang selama proses simulasi. Dalam simulasi biasanya akan di jumpai perbedaan kepadatan, tetapi secara keseluruhan masih akan membentuk satu fluida tak mampu mampat.

Di dalam transportasi Lattice Boltzmann dapat di atur oleh fungsi distribusi yang mewakili partikel di lokasi r (x, y) pada waktu t, dan partikel akan di gantikan oleh (dx, dy) dalam waktu dt dengan di pengaruhi oleh gaya F pada molekul aliran.

Persamaan yang mengatur fungsi distribusi *f* (*r*, *c*, *t*) memiliki dua istilah, langkah aliran (*streaming*) dan tumbukan (*collision*) panjang. Di sini, *x* dan *y* adalah koordinat spasial, *t* adalah waktu, dan *c* adalah kecepatan *discrete lattice* beristirahat, dari nilai tersebut nilai-nilai mikroskopik untuk kepadatan (*density*) (2.4) dan kecepatan (*velocity*) (2.5) bisa di hitung :

$$\rho = \sum_{i=1}^{19} f_i \tag{2.4}$$

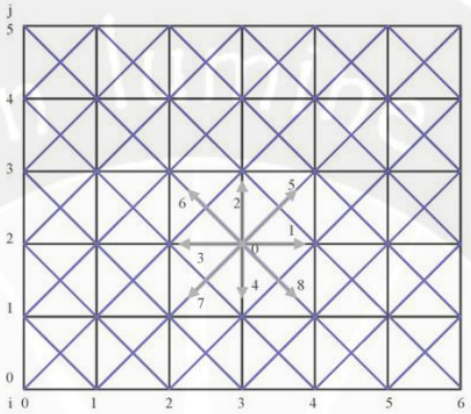
$$u = \frac{1}{\rho}\sum_{i=1}^{19} f_i e_i \tag{2.5}$$

Proses simulasi LBM terdiri dari dua tahap yang di ulang setiap langkah waktu. Yang pertama adalah tahap aliran (*streaming*) (2.6) di mana perpindahan sebenarnya dari partikel melalui grid di lakukan. Tahap berikutnya menghitung tabrakan yang terjadi selama pergerakan itu, sehingga di namakan tahap tabrakan (collision)(2.7) di tambahkan F sebagai gaya luar misalnya gravitasi (2.8).

$$f_i(\bar{x}, t) = f_i'(\bar{x} - \bar{e_i}, t + 1) \tag{2.6}$$
$$f_i'(\bar{x}, t) = f_i(\bar{x}, t) - \frac{1}{\tau}\left(f_i(\bar{x}, t) - f_i^{eq}(\bar{x}, t)\right) + F_i \tag{2.7}$$
$$f_i(\bar{x} + \bar{e}_i\Delta t, t + \Delta t) - f_i'(\bar{x}, t) = -\frac{1}{\tau}\left(f_i - f_i^{eq}\right) + F_i \tag{2.8}$$

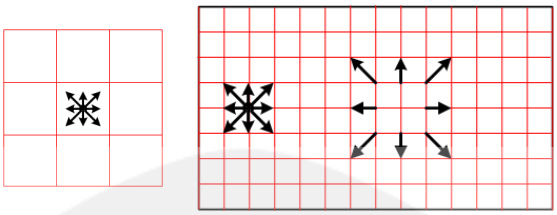
Proses aliran (streaming), sebagai contoh f1(i,j) bergerak menuju f1(i+1, j), f2(i,j) bergerak menuju f2(i, j+1), f3(i,j) bergerak menuju f3(i-1, j), f4(i,j) bergerak menuju f4(i, j-1), f5(i,j) bergerak menuju f5(i+1, j+1), f6(i,j) bergerak menuju f6(i-1, j+1), f7(i,j) bergerak menuju f7(i-1, j-1), f8(i,j) bergerak menuju f8(i+1, j-1).Di dalam pemrograman, perlu di perhatikan bahwa data hasil

perubahan pada fungsi distribusi masih di perlukan untuk tahap aliran (*streaming*) sel lain. Secara numerik LBM dapat di tuliskan dalam persamaan aliran (*streaming*) dalam waktu (*t*) :



Gambar 2.2 Pengaturan *lattice* untuk model D2Q9. (Mohamad, 2011, p.63)

Proses tumbukan di lakukan untuk mendapatkan nilai keseimbangan (f_i^{eq}). Tahap tumbukan (*collision*) tidak mengubah kepadatan atau kecepatan dari sel, tetapi hanya mengubah distribusi partikel untuk semua fungsi distribusi partikel, Model tumbukan di dalam Lattice Boltzmann merupakan interaksi tumbukan antara partikel dalam fluida selama terjadi gerakan, proses tumbukan ini terjadi dengan adanya relaksasi fungsi distribusi yang dapat di hitung untuk setiap sel dengan kepadatan dan kecepatan dari persamaan variabel makroskopik. Proses tumbukan partikel fluida dapat di lihat pada Gambar 2.3 di bawah ini:



(a) (b)

Gambar 2.3 (a) Ilustrasi tumbukan sel tunggal pada *Lattice* D2Q9
(b) Proses aliran setelah proses tumbukan

Secara numerik persamaan Lattice Boltzmann dapat di tuliskan dengan menggunakan persamaan (2.9). pada $\omega = \frac{1}{\tau}$, koefisien ω di namakan frekuensi tumbukan dan τ di namakan faktor relaksasi. Fungsi kesetimbangan distribusi lokal di lambangkan dengan (f^{eq}) yang merupakan fungsi distribusi Maxwell-Boltzmann.

$$\frac{\partial f_i}{\partial t} + e_i . \nabla f_i = -{}^{1}/_{\tau}\left(f_i - f_i^{eq}\right) + F_i \qquad (2.9)$$

$$f^{eq} = \rho\omega_i\left[1 + 3(e_i . u) + {}^{9}/_{2}\,(e_i u)^2 + {}^{3}/_{2}\,u^2\right] \qquad (2.10)$$

Kepadatan dari sel di lambangkan dengan rho dan vektor kecepatan di lambangkan dengan $u = (u_1, u_2)$. Vektor kecepatan dari *lattice* adalah vektor $e_{0..8}$, masing-masing mempunyai bobot $_i\omega$. Untuk tahap tumbukan nilai kesetimbangan fungsi distribusi perlu di hitung dari kepadatan dan kecepatan. Ketiga skalar dari vektor kecepatan dan vektor *lattice* pada persamaan (2.3) dengan mudah dapat di hitung. Ketiganya perlu di skala dengan sesuai dan kemudian di jumlahkan menurut bobot dan kepadatan. Nilai waktu relaksasi ω akan menentukan fluida

dapat mencapai titik kesetimbangan lebih cepat atau lebih lambat. Fungsi distribusi partikel yang baru (f) dapat di hitung menurut persamaan :

$$f_i' = (1 - \omega) f_i + \omega f_i^{eq} \quad (2.11)$$

2.2.5. Lattice Boltzmann untuk multi fase

Pada aliran multiphase berubah karena adanya nilai tambahan yaitu gaya tekanan pada setiap *lattice*nya. Persamaan yang di gunakan adalah persamaan yang di gunakan Shan & Chen (1993).Persamaan ini sering di gunakan di sebabkan saat ini adalah metode yang cukup stabil dalam hal menghitung gaya luar

$$F(x,t) = -G\psi(x,t) \sum_{a=1}^{8} w_a \psi(X + e_a \Delta t, t) e_a \quad \text{(Shan \& Chen, 1993)} \quad (2.12)$$

G adalah kekuatan interaksi, $w_a$adalah 1/9 a ={1,2,3,4}, di mana *1/36* for $a = \psi$ adalah potensi interaksi :

$$\psi(\rho) = \psi_0 exp\left(^{-\rho_0}/_{\rho}\right) \quad (2.13)$$

ψ_0 dan ρ_0 adalah nilai konstan sembarang.

2.2.6. Komputasi paralel

Komputasi paralel merupakan salah satu teknik melakukan komputasi secara bersamaan dengan memanfaatkan beberapa komputer juga secara bersamaan. Komputasi paralel di butuhkan saat kapasitas yang di perlukan sangat besar untuk memproses komputasi. Di samping itu pemakai harus membuat

pemrograman paralel untuk dapat merealisasikan komputasi. Pemrograman paralel memiliki tujuan utama yaitu untuk meningkatkan performa komputasi. Oleh karena itu semakin banyak hal yang bisa di lakukan secara bersamaan dalam waktu yang sama, semakin banyak pekerjaan yang bisa di selesaikan.

Ada dua cara untuk mengurangi waktu komputasi, yang pertama dengan meningkatkan jumlah operasi yang di lakukan setiap detik, dengan menggunakan prosesor yang lebih cepat, tapi cara ini telah menurun bersamaan dengan berhentinya perkembangan kecepatan prosesor pada kecepatan 4Ghz di sebabkan belum mampu mengatasi masalah konsumsi daya dan suhu yang tinggi pada prosesor (Kirk & Hwu, 2010), cara kedua adalah melakukan komputasi dari beberapa program secara bersamaan dalam seri paralel. dengan membagi *threads* ke beberapa prosesor, sehingga kita dapat mengurangi waktu yang di butuhkan untuk proses tanpa harus meningkatkan kecepatan prosesor.

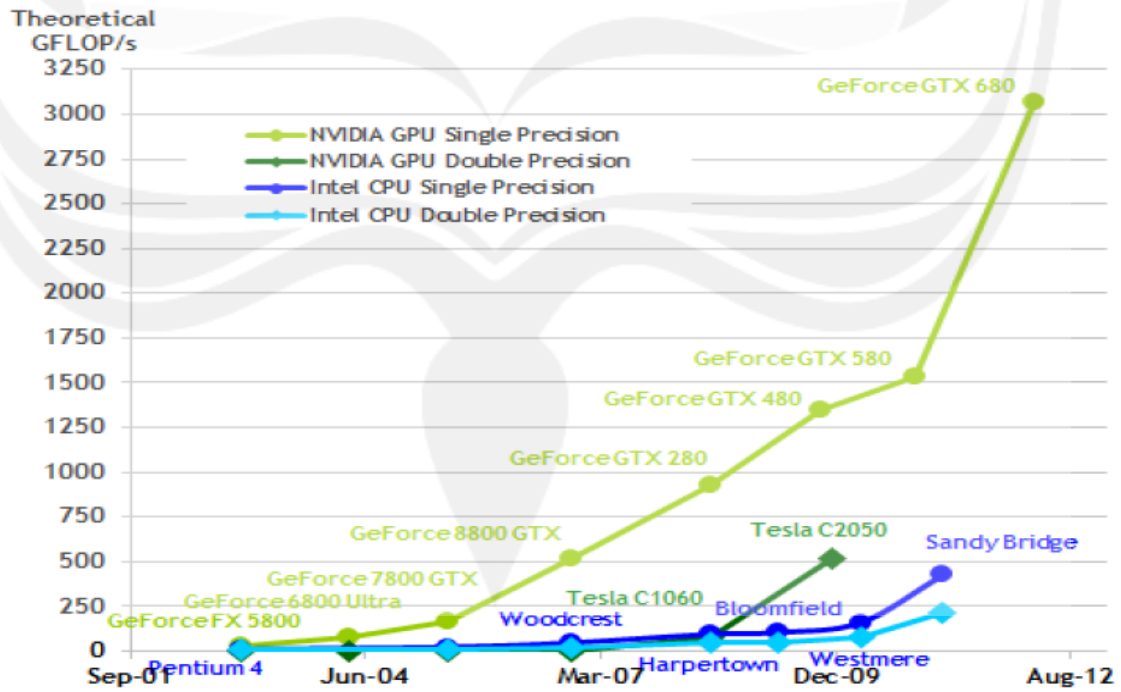
Kita dapat melakukan komputasi paralel dengan menggunakan beberapa perangkat keras sekaligus. Misalnya :

a. Menggunakan banyak CPU dalam satu komputer, contohnya supercomputer yang memiliki lebih dari satu prosesor (Kirk & Hwu, 2010)
b. Menggunakan beberapa komputer dengan CPu tunggal yang secara fisik di hubungkan dengan jaringan.
c. Menggunakan beberapa core pada satu CPU.

Pada cara pertama dan kedua bisa kita terapkan pada sebuah laboratorium dengan biaya yang cukup besar. Dan cara ketiga terbatas pada jumlah core yang

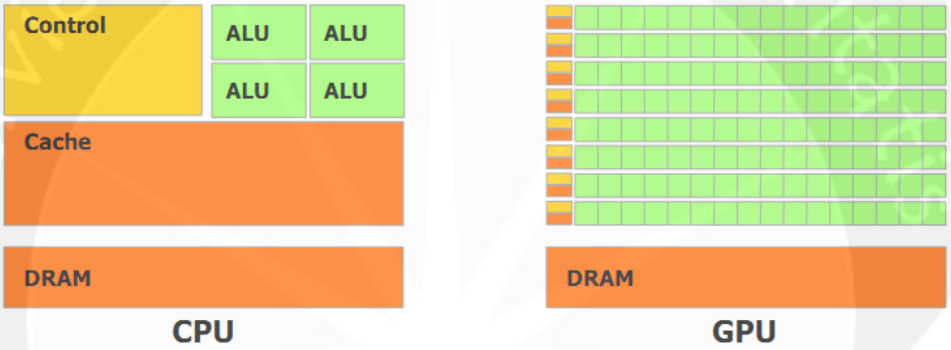
ada pada CPU umumnya di pasaran, maksimal 4 core dalam 1 CPU, sehingga pilihan lain adalah menggunakan paralel dengan membagi *thread* pada beberapa prosesor lain adalah hal ini kita bisa menggunakan prosesor pada GPU.

Pengembangan arsitektur mikroprosesor dari tahun 2003 yaitu, *multicore* dan *many-core* (Kirk & Hwu, 2010). Pada saat ini arsitektur *multicore* dari hanya berinti dua telah meningkat menjadi delapan inti pada satu mikroprosesor. CPU (*central proceesing unit*) banyak menggunakan arsitektur *multicore* tersebut. sedangkan *many-core* berfokus untuk memaksimalkan kinerja program untuk bekerja secara paralel. Arsitektur *many-core* yang banyak di gunakan GPU memiliki lebih banyak unit pemroses daripada arsitektur *multicore*. Dalam Gambar 2.4 terlihat perbandingan kecepatan antara CPU dan GPU.



Gambar 2.4 Perbandingan performa GPU dan CPU (Nvidia, 2012)

Perbedaan kecepatan antara GPU dan CPU di sebabkan karena GPU di khususkan untuk perhitungan intensif dan perhitungan paralel, dan untuk di rancang sedemikian rupa sehingga lebih banyak transistor yang di khususkan untuk pengolahan data bukan untuk *cache* atau *flow-control* (Nvidia, 2012) tidak seperti pada CPU yang memiliki *cache* besar seperti terlihat pada Gambar 2.5 di bawah ini.



Gambar 2.5 GPU lebih banyak transistor ke pengolahan data. (nvidia, 2012)

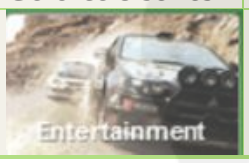
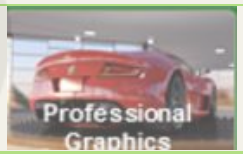
GPU tidak bisa menggantikan CPU sepenuhnya karena GPU di desain untuk melakukan kalkulasi numerik dan untuk memproses secara paralel saja, bukan untuk perhitungan sekuensial.

2.2.7. CUDA (*Compute Unified Device Architecture*)

CUDA ™ adalah platform komputasi paralel dan model pemrograman yang memungkinkan peningkatan dramatis dalam kinerja komputasi dengan memanfaatkan kekuatan dari *graphics processing unit* (GPU) (Nvidia, 2012). Sejak di perkenalkan pada 2006, CUDA telah banyak di sebarkan melalui ribuan aplikasi dan di terbitkan pada makalah penelitian, dan di dukung

oleh dasar terinstal lebih dari 300 juta CUDA-enabled GPU di notebook, workstation, menghitung cluster dan super-computer. (Nvidia, 2012).

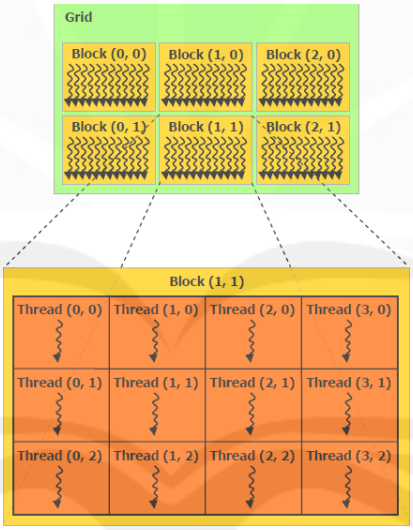
Cuda juga di lengkapi dengan lingkungan perangkat lunak yang memungkinkan pengembang menggunakan bahasa C sebagai bahasa pemrogramannya (nvidia, 2012). Pada Gambar 2.6 di ilustrasikan bahasa pemrograman lain yang bisa di gunakan dalam CUDA dan implementasi pemrograman interface.

<table>
<tr><td colspan="8">GPU Computing Application</td></tr>
<tr><td rowspan="2"></td><td colspan="7">Libraries and Middleware</td></tr>
<tr><td>cuFFT
cuBLAS
cuRAND
cuSPASE</td><td>LAPACK
CULA
MAGMA</td><td>Thrust
NPP
cuDPP</td><td>VSIPL
SVM
OpencCurrent</td><td>PhysX
OptiX</td><td>Iray
RealityServer</td><td>MATALAB
Mathematica</td></tr>
<tr><td>C</td><td>C++</td><td>Fotran</td><td>Java
Python
Wrappers</td><td>Di rect
Compute</td><td>OpenCL^{tm}</td><td colspan="2">Di rectives
(e.g.
OpenACC)</td></tr>
<tr><td colspan="8">NVIDIA GPU
Whit the CUDA Parallel Computing Architecture</td></tr>
<tr><td colspan="2">Fermi Architecture</td><td>Geforce 400
series</td><td colspan="2">Quadro Fermi Series</td><td colspan="2">Tesla 20 Series</td></tr>
<tr><td colspan="2">Tesla Architecture</td><td>Geforce 200
series
Geforce 9 series
Geforce 8 series</td><td colspan="2">Quadro FX series
Quadro Plex series
Quadro NVX Series</td><td colspan="2">Tesla 10 series</td></tr>
<tr><td colspan="2"></td><td> 
</td><td colspan="2"> 
</td><td colspan="2"> 
</td></tr>
</table>

Gambar 2.6 CUDA untuk bahasa pemrograman dan aplikasi program interface. (nvidia, 2012)

Model pemrograman CUDA adalah membagi pekerjaan yang akan di lakukan ke banyak unit pemroses paling kecil, yakni *thread.* Setiap *thread*

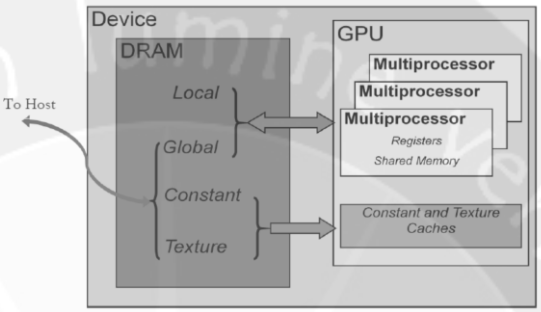
memiliki memori tersendiri dan mengerjakan inti pekerjaan yang kecil dan akan berjalan bersamaan dengan *thread* lain sehingga waktu yang di butuhkan untuk mengerjakan pekerjaan seluruhnya menjadi lebih singkat. Kumpulan dari *thread* tersebut di sebut sebagai blok dalam blok memiliki satu memori tersendiri yang bisa di gunakan *thread* dalam blok tersebut. Media untuk saling berkomunikasi antar *thread* di namakan dengan Share memori. Setiap blok akan di kelompokkan lagi menjadi sebuah grid yang menjadi sekumpulan *block* yang di gunakan dalam komputasi. Ilustrasi seperti terlihat pada Gambar 2.7.



Gambar 2.7 Struktur unit pemroses pada CUDA (nvidia, 2012)

CPU dalam CUDA di sebut dengan istilah *Host*, dan GPU di sebut dengan istilah *Device*. Ada 2 bagian pada pemrograman CUDA yaitu fungsi yang di jalankan pada CPU dan fungsi yang di jalankan pada GPU, *kernel* adalah sebutan untuk program CUDA yang berjalan pada GPU. Pada saat fungsi ini berjalan, pemrogram harus memberikan informasi pada GPU untuk memesan berapa banyak *block*, *thread* yang akan di gunakan dalam masing-masing blok.

Perangkat CUDA menggunakan spasi memori yang memiliki karakteristik yang berbeda yang menunjukkan yang berbeda dalam aplikasi Cuda. Ruang ruang memori pada CUDA seperti terlihat pada Gambar 2.8



Gambar 2.8. Model memori perangkat CUDA

Jumlah memori yang tersedia di setiap ruang memori pada setiap tingkat kemampuan menghitung *Global*, *Constant* dan *memory texture* memiliki *latency* yang terbesar, di ikuti oleh memori konstan, register dan shared memori, ciri tiap memori bisa terlihat pada Tabel 2.1 di bawah.

Tabel 2.1 Fitur *memory device*

Memory	**Location on/off chip**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	On	-	R/W	1 *thread*	*Thread*
Local	Off	+	R/W	1 *Thread*	*Thread*
Shared	On	n/a	R/W	All *thread* in block	Block
Global	Off	+	R/W	All *thread* + host	Host location
Constant	Off	Yes	R	All *thread* + host	Host location
Texture	Off	Yes	R	All *thread* + host	Host location

Tabel 2.2 Beberapa perintah untuk pemrograman CUDA.

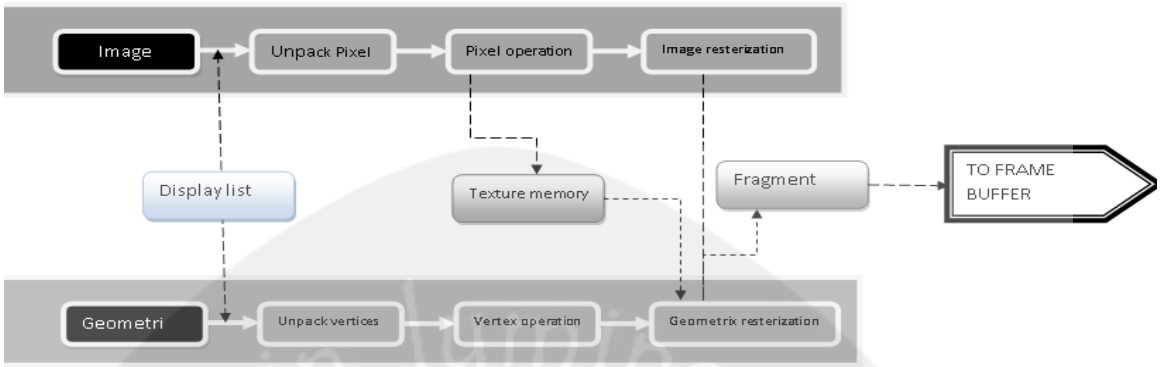
***Syntax*s**	**Write in program**	**Fungsi**	**Keterangan**
cudaGetDeviceCount().	cudaGetDeviceCount().	*Device* management	
cudaGetDeviceProperties().	cudaGetDeviceProperties().	*Device* management	
cudaMalloc()	cudaMalloc()	*Device* memory management	
cudaFree ()	cudaFree ()		
cudaMemcpy ()	cudaMemcpy ()		
cudaBindTexture ()	cudaBindTexture ()	Texture management	
cudaBindTextureTo*Array* ()	cudaBindTextureTo*Array* ()		
cudaGLMapBufferObject ()	cudaGLMapBufferObject ()	Graphics interoperability	
cudaD3D9MapVertexBuffer ()	cudaD3D9MapVertexBuffer ()		
__global__	Void MyKernel(){}	Function qualifiers	Cal from host, execute on GPU
__device__	Float MyDeviceFunc() {}	Function qualifiers	Cal from GPU, execute on GPU
__host__	Int HostFunc() {}	Function qualifiers	Cal from host, execute on host
__device__	Float MyGPU*Array*[32];	Variable qualifiers	In GPU memory space
__constant__	Float MyConst*Array*[32];	Variable qualifiers	Write by host; read by GPU
__shared__	Float MyShared*Array*[32];	Variable qualifiers	Shared within *thread* block
Int1, int2… **Float1, float2…** **Double, double2..etc**		Built-in vector types	
<<< >>>	<<< Grid, Block >>>	Excution configuration	Launch kernel
gridDdi m	Di m3 gridDdi m;	Variable and function valid in *device* code	Grid dimension
blockDi m	Di m3 blockDi m	Variable and function valid in *device* code	Block dimension
blockIdx	Di m3 blockIdx	Variable and function valid in *device* code	Block index
***Threadi* dx**	Di m3 *Threadi* dx	Variable and function valid in *device* code	*Thread* index
__syc*thread*s()	Void __syc*thread*s()	Variable and function valid in *device* code	*Thread* synchronization

Bahasa pemrograman yang di gunakan oleh CUDA adalah bahasa C/C++ yang di beri ekstensi fungsi-fungsi dan *syntax*s yang di gunakan khusus untuk CUDA untuk manajemen perangkat CUDA. *Syntax*s khusus pada pemrograman CUDA memiliki fungsi fungsi yang berbeda dan tujuan yang berbeda tiap *syntax*snya. Seperti terlihat pada Tabel 2.2 di atas.

2.2.8. OpenGL

OpenGL terdiri dari tiga perpustakaan : Graphics Library (GL) Graphics Library Utilitas (Glu), dan Graphics Library Utilities Toolkit (GLUT). Nama fungsi di mulai dengan gl, glu, atau Glut, tergantung pada perpustakaan mereka miliki.

OpenGL (Open Graphics Library) adalah spesifikasi standar yang mendefinisikan sebuah lintas-bahasa, lintas platform API untuk mengembangkan aplikasi yang menghasilkan grafis komputer 2D maupun 3D. Antarmuka terdiri dari lebih dari 250 panggilan fungsi yang berbeda yang dapat di gunakan untuk menggambar tiga dimensi yang adegan-adegan kompleks dari bentuk-bentuk primitif sederhana. OpenGL di kembangkan oleh Silicon Graphics Inc (SGI) pada tahun 1992 (Shreiner, 2010) dan secara luas di gunakan dalam CAD, realitas maya, visualisasi ilmiah, visualisasi informasi, dan simulasi penerbangan. Hal ini juga di gunakan dalam video game, di mana bersaing dengan Direct3D on Microsoft Windows platform. Singkatnya OpenGL adalah *source* program yang di gunakan untuk menampilkan *output* dari *input* data setelah pengolahan grafis (Strzodka et al., 2005) Pada Gambar 2.9 adalah alur kerja dari OpenGL :



Gambar 2.9. Alur kerja OpenGL

Pada dasarnya openGL adalah pemrograman dengan menggambarkan objek objek dasar atau sering di sebut objek primitif, yaitu, titik, garis, dan lingkaran, setiap fungsi untuk penggambaran lain hanya menggunakan kombinasi dari fungsi penggambaran primitif, seperti halnya untuk membuat sebuah elips maka hanya menggunakan algoritma untuk membangun dari fungsi lingkaran dengan sedikit modifikasi pada jari-jarinya.

Perintah dan fungsi di openGL mempunyai format yang sama ini memudahkan ahli program untuk membaca bagaimana jalannya fungsi ini, lihat Tabel 2.3, dan pada Tabel 2.4 di bawah menunjukkan beberapa contoh perintah openGL.

Tabel 2.3 Format fungsi openGL

Suffix	**Tipe Data**	**C++ atau C**	**OpenGL**
b	Integer 8-bit	Signed char	GLbyte
s	Integer 16-bit	Short	GLshort
i	Integer 32-bit	Int atau long	GLint, GLsizei
f	Float 32-bit	Float	GLfloat
d	Float 64-bit	Double	GLdouble`

Tabel 2.4 Contoh perintah openGL

Perintah	**Arti**	**Keterangan**
glVertex2i(x, y);	Lokasi titik berada (x, y)	Tipe argumennya adalah integer dan 2D (x, y)
glVertex2f(x, y);	Lokasi titik berada (x, y)	Tipe argumennya adalah float dan 2D (x, y)
glVertex3i(x, y, z);	Lokasi titik berada (x, y, z)	Tipe argumennya adalah integer dan 3D (x, y, z)
glVertex3f(x, y, z);	Lokasi titik berada (x, y, z)	Tipe argumennya adalah float dan 3D (x, y, z)
glClearColour(R, G, B,);	Warna latar belakang	Empat komponen warna RGBA
glColor3f(R, G, B)	Warna latar muka	Tiga komponen warna (RGB)
glColor4f(R, G, B,)	Warna latar muka	Empat komponen warna (RGBA)
glBegin(GL_POINTS);	Titik	Objek primitif
glBegin(GL_LINES);	Garis	Objek primitif
glBegin(GL_LINE_STRIP);	Poligaris	Objek primitif
glBegin(GL_LINE_LOOP);	Poligaris tertutup	Objek primitif
glBegin(GL_TRIANGLES);	Segitiga	Objek primitif
glBegin(GL_TRIANGLE_STRIP);	Segitiga	Objek primitif
glBegin(GL_TRIANGLE_FAN);	Segitiga	Objek primitif
glBegin(GL_QUADS);	Segiempat	Objek primitif
glBegin(GL_QUADS_STRIP);	Segiempat	Objek primitif
glBegin(GL_LINE_STIPPLE);	Garis putus-putus	Objek primitif
glBegin(GL_POLY_STIPPLE);	Poligon dengan pola tertentu	Objek primitif