# BAB II

# TINJAUAN PUSTAKA

## 2.1. Penelitian Terdahulu

### 2.1.1. Penelitian Tsunami

Tsunami ini merupakan kejadian alam yang dipengaruhi adanya aktifitas yang terjadi di dasar laut, aktifitas ini dapat berupa gempa laut, gunung berapi meletus, atau hantaman meteor di laut, tanah longsor di dasar laut, patahan. Terjadinya aktifitas tersebut dapat membuat tsunami yang menghancurkan seperti di pantai utara New Guinea pada 1998 (Ward, 2000; Geist, 2000; Tappin, et. al., 1999). Pergeseran di bawah laut ini memang menyebabkan perpindahan air laut, bahkan kecepatannya bisa mencapai 800 km/jam, parahnya ketika mencapai daratan, kecepatan tsunami mengalami penurunan tapi ketinggiannya bertambah disebabkan oleh penumpukan massa air (Ramya dan palaniappan, 2011). Pada laut lepas misal terjadi gelombang pasang sebesar 8 m tetapi begitu memasuki daerah pelabuhan yang menyempit tinggi gelombang pasang menjadi 30 m (Nur, 2010) Bahkan akibat dari tsunami dapat merubah ekosistem laut, seperti pada tsunami aceh 26 desember 2004 (Huda dkk, 2009).

### 2.1.2. Metode Lattice Boltzmann Untuk Air Dangkal

Perairan dangkal merupakan perairan yang mempunyai kedalaman < 60 Km.

Teori perairan dangkal biasanya yang digunakan untuk pemodelan tsunami secara numeric. Persamaan air dangkal biasanya diguakan untuk mensimulasikan gelombang yang panjang gelombangnya mirip dengan ketinggian air secara keseluruhan (Thurey, dkk. 2006). Dalam hal ini kecepatan propagasi gelombang untuk amplitude adalah konstan. Simulasi air dangkal juga dapat dibentuk dengan menggunakan persamaan lattice Boltzmann. Tidak hanya mempertimbangkan tekanan fluida tetapi nilai ketinggian dihutung untuk setiap sel.

Penelitian yang memanfaatkan metode lattice Boltzmann untuk memodelkan tsunami juga pernah dilakukan yaitu mengenai Model 2D visualisasi tsunami aceh dengan metode lattice Boltzmann, Nazarudin menggunakan CPU (Nazarudin, 2013), karena menggunakan CPU prosesnya pemodelannya berjalan lambat, sehingga dapat dikembangkan lebih lanjut dengan menggunakan teknologi GPU.

### 2.1.3. Metode Lattice Boltzmann Dengan GPU

Metode lattice Boltzmann (Thurey, 2003), sesuai dengan namanya, bekerja dalam area *lattice*. Ada beragam jenis *lattice* yang dapat digunakan, tergantung pada lingkungan pengaplikasiannya. Penamaannya pun disesuaikan menurut aturan DXQY, di mana X adalah jumlah dimensi, misalnya 3, dan Y menunjukkan banyaknya arah kecepatan *lattice*. Metode lattice Boltzmann merupakan salah satu jenis *cellular automata*, yang berarti fluida terbentuk dari banyak sel sejenis. Semua sel diperbaharui disetiap langkah waktu dengan aturan sederhana, dengan ikut memperhitungkan sel-sel disekitarnya. Metode lattice

Boltzmann memodelkan fluida yang tak mampu-mampat (*incompressible*) dimana partikel fluida hanya dapat bergerak searah dengan vektor kecepatan *lattice*.

Keuntungan dari metode ini adalah kemudahan dalam komputasi paralel karena lokalitas interaksi partikel dan transportasi informasi partikel, fleksibilitas dalam geometri karena implementasi yang relatif mudah dengan menentukan kondisi batas yang kompleks dan sifat kompleks dari sistem cairan. Dengan menggunakan metode lattice Boltzmann dapat mengoptimalkan proses pemodelannya (Revell, 2013). Metode ini juga sangat baik untuk aliran yang kompleks, dan yang bisa diparalelkan, metode ini juga mudah untuk diimplementasikan (Januszewski, 2012).

### 2.1.4. NVidia CUDA Untuk Metode Lattice Boltzmann

Nvidia Cuda merupakan tools pemrograman parallel yang sangat baik digunakan dalam penerapan metode lattice Boltzmann, lebih baik dari menggunakan teknologi GPU yang lain. (Bernaschi, dkk, 2009). Penggunaan CUDA memberikan kenyaman dalam pemrograman parallel karena CUDA menyediakan akses ke level arsitektur komputasi (Gohari dan Ghadyani, 2012). Karena pengguanaan NVidia CUDA lebih baik dan optimal mengakibatkan penelitian ini menggunakan NVidia CUDA sebagai *tools*nya.

## 2.2. Tsunami

Tsunami (dalam bahasa Jepang) secara arafiah berarti "**Ombak**" besar (***nami***) di pelabuhan (***tsu***), adalah sebuah ombak yang terjadi setelah gempa bumi, gempa laut, gunung berapi meletus, atau hantaman meteor di laut tanah longsor di dasar laut. Gerakan vertikal pada kerak bumi, dapat mengakibatkan dasar laut atau turun secara tiba – tiba, yang mengakibatkan gangguan kesetimbangan air yang berada di atasnya. Hal ini mengakibatkan terjadinya aliran energi air laut yang ketika sampai di pantai menjadi gelombang besar yang mengakibatkan terjadinya tsunami.

Gerakan vertikal ini dapat terjadi pada patahan bumi. Lempeng samudera yang lebih rapat menelusup ke bawah lempeng benua dalam status proses yang disebut subduksi, dan gempa bumi subduksi sangat efektif meghasilkan tsunami. Kecepatan penjalaran gelombang tsunami berkisar antara 50 km sampai 1.000 km per jam. Pada saat mendekati pantai, kecepatannya semakin berkurang, karena adanya gesekan dasar laut, tetapi tinggi gelombangnya justru akan bertambah besar pada saat mendekati pantai (mencapai ketinggian maksimum pada pantai berbentuk landai dan berbentuk seperti teluk dan muara sungai). Peristiwa ini bisa menyebabkan kerusakan erosi pada kawasan pesisir pantai dan kepulauan (Syamsul Arifin, 2005).

## 2.3. Metode Lattice Boltzmann

Ludwig Eduard Boltzmann (1844-1906), adalah seorang fisikawan asal

Autria yang banyak memberikan sumbangsi dalam penelitian tentang mekanika statistik, yang menjelaskan dan memprediksi bagaimana sifat-sifat atom molekul (sifat mikroskopis) menentukan sifat fenomenologis (makroskopik) seperti viskositas, konduktivitas termal, dan koefisien difusi. Fungsi distribusi (probabiltas untuk menentukan partikel dalam jarak tertentu dari kecepatan tertentu dari berbagai lokasi pada waktu tertentu) menggantikan penendaan setiap partikel, seperti pada simulasi dinamis molekul. Dalam dunia komputasi metode ini digunakan karena dapat menghemat sumber daya komputer secara drastis (Mohamad, 2011).

Metode numerik yang digunakan untuk simulasi fluida menghitung variabel makroskopik yaitu kecepatan dan kepadatan (density). Metode lattice Boltzmann didasarkan pada persamaan kinetic mikroskopik untuk menghitung fungsi distribusi partikel fluida (Zhang, 2011).

### 2.3.1. Aliran Fasa Tunggal

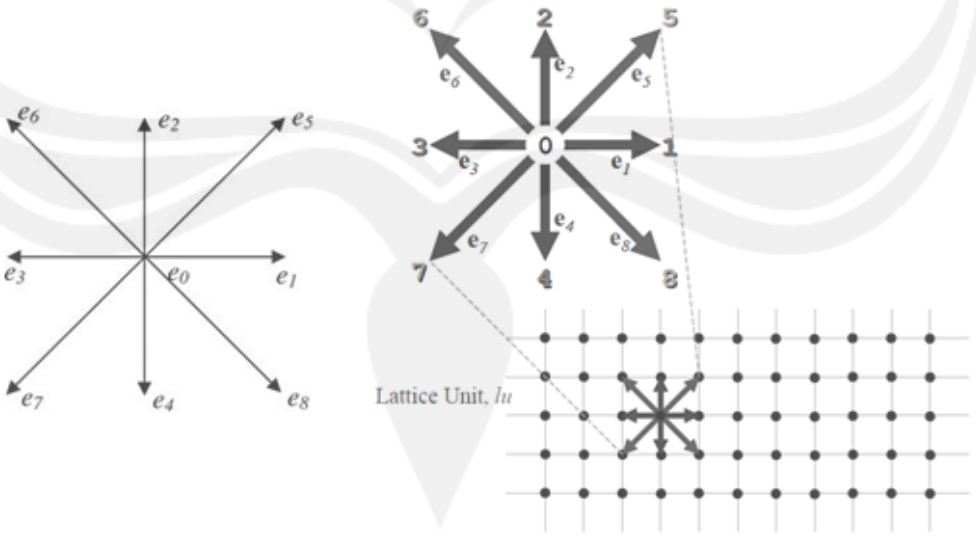
Aliran fasa tunggal hanya mewakili satu gerak fluida saja, yaitu gas atau cair. Dalam penelitian ini yang digunakan hanya fasa cair.

Pemodelan yang dihasilkan dalam penelitian ini adalah pemodelan aliran fasa tunggal dua dimensi, maka model yang digunakan yaitu model D2Q9 dari sel *lattice* yang dianggap mudah dilihat dari sisi akurasi dan kemampuan komputasi dari simulasi yang ingin dihasilkan.

### 2.3.2. Model D2Q2 Lattice Boltzmann

Model dalam metode lattice Boltzmann biasa dilambangkan dengan DnQm, dimana n menyatakan jumlah dimensi yang digunakan, dan m menyatakan jumlah arah lattice yang digunakan. Metode lattice Boltzmann menyediakan model-model lattice untuk digunakan 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 digunakan D2Q9, yaitu berdimensi 2 dengan 9 arah lattice. Bisa dilihat pada gambar 3.1, yang menunjukkan kartesian lattice dan kecepatan $e_a$ dimana a = 0,1,…,8 adalah indeks arah dan $e_0 = 0$ yang menunjukan partikel saat diam. Setiap sisi dari se memiliki panjang 1. Unit lattice (lu) adalah ukuran panjang dalam metode lattice Boltzmann dan selisih waktu (ts) adalah unit waktu.




**Gambar 2. 1 Model D2Q9 arah dan kecepatan**

### 2.3.3. Persamaan Lattice Boltzmann

Metode lattice Boltzmann adalah model yang sangat sederhana di dalam konseptual Boltzmann dengan mengurangi jumlah partikel special dan momentum mikroskopis. Metode ini bekerja pada daerah lattice, dimana posisi partikel terbatas pada node-node. Metode lattice Boltzmann merupakan salah satu jenis cellular automata, yang berarti fluida terbentuk dari banyak sel sejenis. Setiap sel diperbaharui disetiap langkah dan waktu dengan aturan sederhana, dengan ikut memperhitungkan sel-sel disekitarnya.

Persamaan diskrit lattice Boltzmann kinetik untuk dua dimensi dapat disusun sebagai berikut:

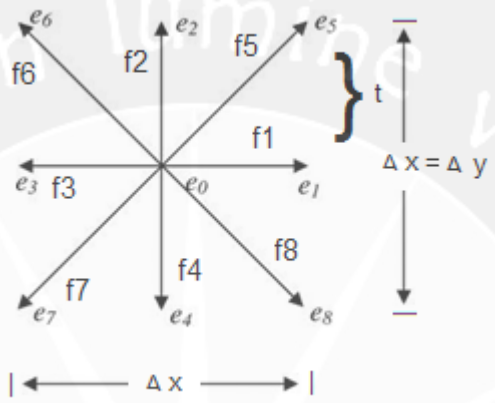
$$\frac{\partial f}{\partial t} + e_i . \nabla f_i = J_i + F_i \quad i = 1,2, \ldots . i - 1 \tag{2.1}$$

Dimana $f = (\bar{x}, \bar{v}, t)$ adalah densitas partikel, e adalah kecepatan mikroskopik pada area lattice, J merupakan operator tumbukan, F adalah pengaruh gaya luar. Di dalam pendekatan BGK (*Bhatnagar, Gross, Krook*) persamaan diskrit Boltzmann dapat disusun,

$$\frac{\partial f_i}{\partial t} + e_i . \nabla f_i = -\frac{1}{r}(f_i - f_i^{eq}) + F_i \tag{2.2}$$

Dimana $r$ adalah waktu relasi dan $f_i^{eq}$ adalah fungsi distribusi.

Di dalam proses simulasi, semua sel harus menyimpan informasi partikel yang bergerak menurut arah masing-masing fektor kecepatan dan fungsi distribusi partikel. Fungsi ini dinotasikan dengan $f_i$ dimana $i$ menunjukkan nomor fektor lattice.




**Gambar 2. 2 Model D2Q9 fungsi distribusi nilai fi**

Pada gambar 2.2 di atas terdapat susunan fungsi $f_i$ yaitu $f_0$, $f_1$, $f_2$, $f_3$, $f_4$, $f_5$, $f_6$, $f_7$, $f_8$. Vector dengan nomor 0 mempunyai panjang 0 dan menyimpan jumlah partikel yang berhenti di sel berikutnya. Partikel ini tidak akan bergerak kemana-mana dilangkah waktu berikutnya, tetapi beberapa diantaranya kemungkinan akan dipercepat (bergerak) karena tumbukan dengan pertikel lain, jadi jumlah partikel yang diam bisa saja berubah.

Dari gambar 2.2 di atas, kita dapat mendefinisikan sembilan kecepatan $e_i$ dalam model D2Q9 yang didefinisikan sebagai berikut:

$e_o = (0,0).c, e_1 = (1,0).c, e_2 = (0,1).c, e_3 = (-1,0).c, e_4 = (0,-1).$

$c, e_5 = (1,1).c, e_6 = (-1,1).c, e_7 = (-1,-1).c, e_8 = (1,-1).c$

Atau bisa juga didefinisikan sebagai berikut:

$e_0 = (0,0).c, e_{1,3} = (\pm 1,0).c, e_{2,4} = (0,\pm 1).c, e_{5,6,7,8} = (\pm 1,\pm 1).c$

Diamana c=$\frac{\Delta x}{\Delta t} = \frac{\Delta y}{\Delta t}$. Disini, $\Delta t$ digunakan untuk melihat selisih waktu (ts) untuk menghitung jarak gerak antar partikel. Setiap arah memiliki bobot, bobot arah tersebut adalah $w_0 = \frac{4}{9}, w_{1,2,3,4} = \frac{1}{9}, w_{5,6,7,8} = \frac{1}{36}$. Dalam bentuk persamaan dapat ditulis sebagai berikut:

$$w_1 \begin{cases} \frac{4}{9}, i = 0 \\ \frac{1}{9}, i = 1,2,3,4 \\ \frac{1}{36}, i = 5,6,7,8 \end{cases} \quad (2.3)$$

Dapat juga ditulis dengan persamaan:

$$\sum_{i=0}^{\beta-1} w_i = 1 \quad (2.4)$$

Di dalam fungsi distribusi, ada dua nilai penting yang dihasilkan dengan cara

menilai semua (9) fungsi distribusi, didapat kepadatan (massa/volume) dari sel, dengan asumsi semua partikel mempunyai massa yang sama yaitu 1. Hasil yang penting lainnya yaitu untuk semua sel didapat kecepatan dan arah kemana partikel akan cenderung bergerak dari setiap sel. Kepadatan momentum perlu dihitung, yaitu jumlah dari semua fungsi distribusi partikel, tetapi setiap distribusi harus dikalikan dengan vector lattice. Sehingga, fungsi distribusi partikel 0 akan dikalikan dengan vector (0,0) yang selalu menghasilkan 0. Fungsi distribusi $f_1$ dikalikan dengan vector (1,0) dan ditambah fungsi distribusi $f_3$ dikalikan dengan vector (-1,0) dan begitu seterusnya. Dari proses perhitungan di atas didapatkan vector dua dimensi yang panjangnya ditentukan oleh kepadatan volume. Cukup dengan membagi momentum kepadatan dengan kepadatan sehingga didapat vector kecepatan untuk satu sel. Untuk awal simulasi, kepadatan diberi nilai 1. Karena metode lattice Boltzmann digunakan untuk fluida yang tidak dapat dipadatkan, artinya nilai kepadatan disetiap fluida adalah konstan dan keterikatan ini merenggang selama proses simulasi. Dalam simulasi biasanya akan dijumpai perbedaan kepadatan, tetapi secara keseluruhan akan tetap membentuk suatu fluida tak mampu mampat.

Di dalam transportasi metode lattice Boltzmann dapat diatur oleh fungsi distribusi yang mewakili partikel di lokasi *r*(x, y) pada waktu *t,* dan partikel akan digantikan oleh (dx, dy) dalam waktu dt dengan dipengaruhi oleh gaya F pada molekul cairan. Persamaan yang mengatur fungsi distribusi f(r,c,t) memiliki dua

istilah, langka aliran (streaming) dan tumbukan (collusion) panjang disini, x dan y adalah koordinat special, t adalah waktu, c adalah kecepatan diskrit lattice.

Variabel makroskopik didefinisikan sebagai fungsi dari fungsi distribusi partikel. Persamaanya dapat dilihat sebagai berikut:

Makroskopik densitas fluida ($\rho$):

$$\rho = \sum_{-=0}^{\beta-1} f_i \tag{2.5}$$

Makroskopic kecepatan:

$$\vec{u} = \frac{1}{\rho} \sum_{-=0}^{\beta-1} f_i \overrightarrow{e_i} \tag{2.6}$$

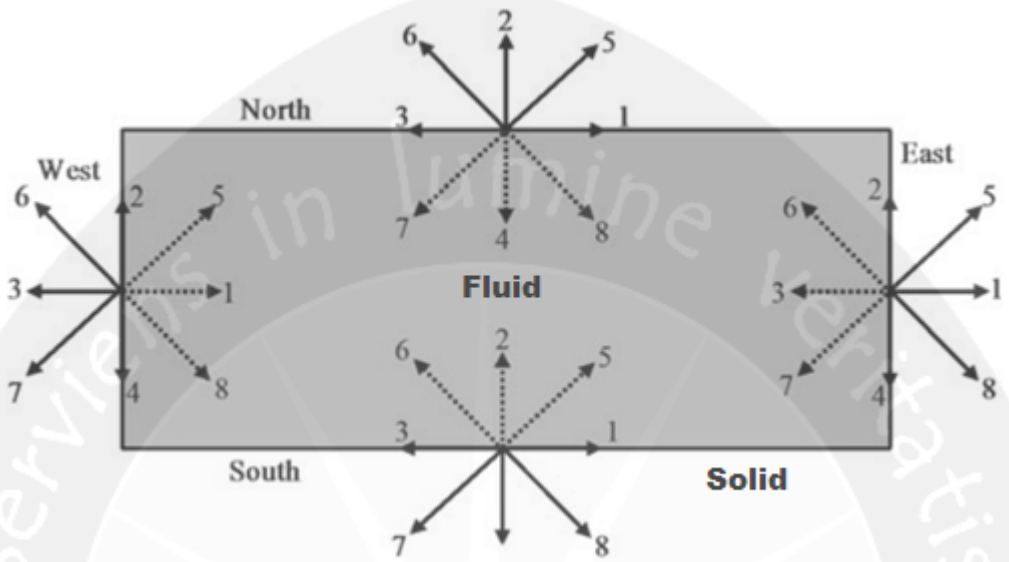
### 2.3.4. Kondisi Batas (Boundary Condition)

Dalam simulasi fluida, menentukan kondisi batas merupakan hal yang penting di dalam mengimplementasikan metode metode lattice Boltzmann. Dalam penelitian ini kondisi batas yang digunakan adalah bounce-back dengan ekstrapolasi.

Proses aliran fluida yang dilakukan tidak periodic, artinya aliran yang melewati batas dinding tidak dapat muncul kembali dari arah berlawanan, tetapi aliran fluida yang digunakan bersifat memantul kembali (bounce-back) pada gambar 2.3 kita asumsikan saja empat batas dinding sebagai empat arah mata angin yang terdiri dari bagian permukaan padat (solid) dan fluida.

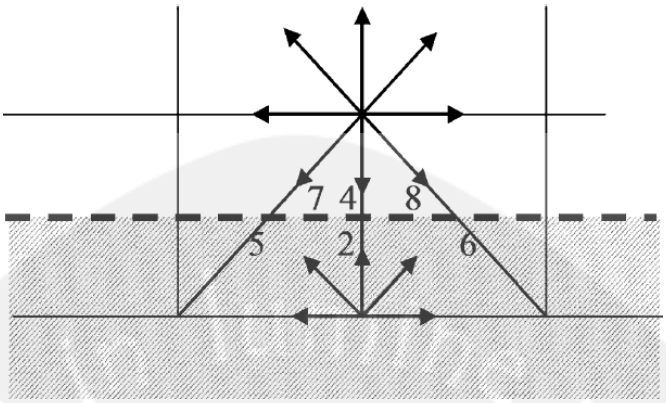
Arah aliran datang: $f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8$.

Arah aliran memantul: $f_0, f_3, f_4, f_1, f_2, f_7, f_8, f_5, f_6$.




**Gambar 2. 3 Arah kecepatan aliran pada bidang fluida dan solid**

Kondisi batas bounce-back digunakan untuk memodelkan kondisi batas pada permukaan yang diam atau yang bergerak atau aliran yang menggunakan hambatan. Partikel yang menuju permukaan yang padat memantul kembali menuju arah aliran datang.




**Gambar 2. 4 Langkah kondisi bounce-back**

Dari gambar 2.4 diatas dapat dituliskan untuk langkah kondisi bounce-back di dalam metode lattice Boltzmann untuk fungsi arah terhadap waktu:

$$f_7(x,1) = f_5(x,t+1)$$

$$f_4(x,1) = f_2(x,t+1)$$

$$f_8(x,1) = f_6(x,t+1)$$

Kondisi bounce-back dengan ekstra polasi adalah menentukan kondisi batas pada fluida pada titik lattice menggunakan perkiraan titik tetangganya pada fluida bagian luar. Untuk mendapatkan nilai titik posisi lattice berada, maka digunakan titik lattice tetangganya yang memiliki arah lattice yang sama.

### 2.3.5. Tumbukan Dan Aliran (Collision and Streaming)

Ada dua tahap dalam proses simulasi yang diulang setiap langkah waktu. Pertama adalah tahap aliran dimana setiap perpindahan sebenarnya dari partikel melalui grid dilakukan. Kedua adalah menghitung tumbukan terjadi selama pergerakan sehingga dinamakan tahap tumbukan.

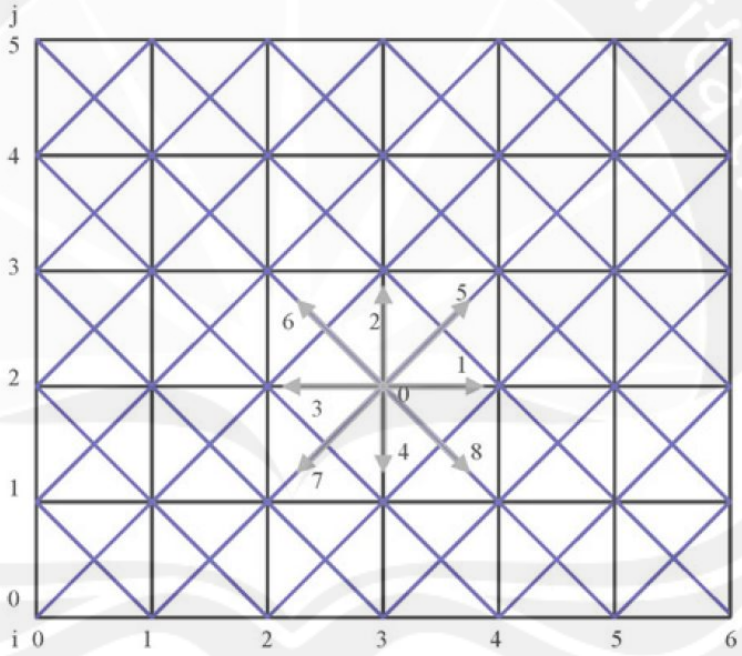
a) Aliran

Untuk tahap aliran hanya terdiri dari operasi salin. Untuk setiap sel, semua fungsi distribusi disalin kesemua sel tetangga yang searah dengan vector lattice. Sedangkan fungsi distribusi sel dengan koordinat (i,j) untuk vector yang menunjuk keatas disalin kefungsi distibusi yang mengarah ke atas dari sel (i,j+1). Fungsi distribusi untuk vector 0 tidak berubah dalam tahap aliran karena vector tersebut tidak menunjuk kemana-mana. Hasilnya adalah fungsi distribusi yang bergerak pada grid. Kecepatan juga kepadatan sel akan berubah, tanpa interaksi lebih lanjut.

Di dalam peroses aliran, sebagai contoh $f_i(i,j)$ bergerak menuju $f_1(i+1,j)$, $f_2(i,j)$ bergerak menuju $f_2(i,j+1)$, $f_3(i,j)$, bergerak menuju $f_3(i-1,j)$, $f_4(i,j)$ bergerak menuju $f_4(i,j-1)$, $f_5(i,j)$ bergerak menuju $f_5(i+1,j+1)$, $f_6(i,j)$ bergerak menuju $f_6(i-1,j+1)$, $f_7(i,j)$ bergerak menuju $f_7(i-1,j-1)$, $f_8(i,j)$ bergerak menuju $f_8(i+1,j-1)$. Di dalam

pemrograman, perlu diperhatikan bahwa data pada hasil perubahan pada fungsi distribusi masih diperlukan untuk tahap aliran sel lain. Secara numeric metode lattice Boltzmann dapat dituliskan dengan persamaan aliran dalam waktu t:

$$f_i'(x,t)=f_i(x+e_i,t+1) \quad (2.7)$$



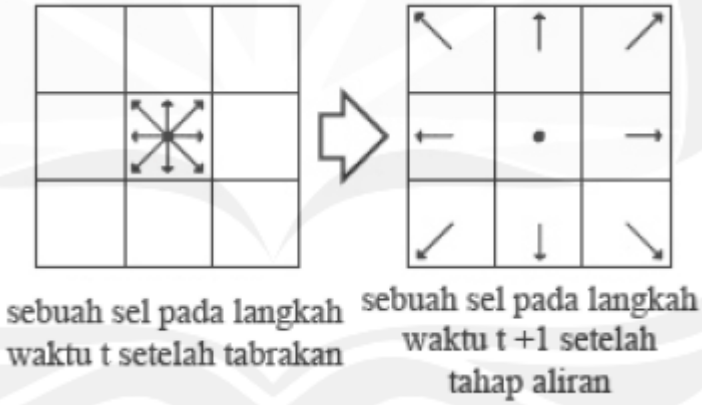

**Gambar 2. 5 Pengaturan *lattice* untuk model D2Q9 (sumber: Mohammad,2011)**

b) Tumbukan

Tumbukan terjadi ketika partikel merambat pada sel yang sama pada waktu yang sama, memancarkan partikel kearah yang

berlainan. Proses tumbukan dilakukan untuk mendapat nilai keseimbangan ($f_i^{eq}$). Tahap tumbukan tidak merubah kepadatan atau kecepatan dari sel, tetapi hanya merubah distibusi partikel.

Model tumbukan di dalam metode lattice Boltzmann meupakan interaksi tumbukan antara partikel dalam fluida selama terjadi gerakan, proses tumbukan ini terjadi dengan adanya relaksasi fungsi distribusi yang dapat dihitung untuk setiap sel dengan kepadatan dan kecepatan dari persamaan variabel makroskopik. Proses tumbukan partikel fluida dapat dilihat pada gambar dibwah ini:




**Gambar 2. 6 Ilustrasi tumbukan sel tunggal pada D2Q9**

Secara numeric persamaan metode lattice Boltzmann dapat ditulis dengan menggunakan persamaan (2.2) dan (2.7):

$$f_i(x + e_i, t + 1) - f_i(x, t) = -\frac{1}{r}(f_i(x, t) - f_i^{eq}(x, t)) \qquad (2.8)$$

Dimana $= \frac{1}{r}$, koefisien *w* dinamakan frekuensi tumbukan dan $r$ dunamakan factor relaksasi. Fungsi kesetimbangan distribusi local dilambangkan dengan ($f^{eq}$) yang merupakan fungsi distribusi Maxwell-boltzmann.

Kepadatan sel dilambangkan dengan rho dan vector kecepatan dengan $\vec{u} = (u_1, u_2)$. Vector kecepatan dari lattice adalah vector $\vec{e}$ 0..8, masing –masing mempunyai bobot $w_i$. Untuk tahap tumbukan nilai kesetimbangan fungsi distribusi dapat dihitung dari kepadatan dan kecepatan.

Ketiga scalar dari vector kecepatan dan vector lattice pada persamaan (2.3) dengan mudah dapat dihitung. Ketiganya perlu diskala dengan sesuai dan kemudian dijumlah dengan bobot dan kepadatan. Nilai waktu relaksasi *w* akan menentukan fluida dapat mencapai titik keseimbangan lebih cepat atau lebih lambat, fungsi partikel distribusi yang baru ($f_i'$) dapat dihitung menurut persamaan:

$$f_i' = (1-w)f_i + wf_i^{eq} \qquad (2.9)$$

## 2.4. Metode Lattice Boltzmann Untuk Persamaan Air Dangkal

Perairan dangkal adalah perairan yang mempunyai *surface* (batas permukaan)

dan *bottom* (batas dasar). Teori perairan dangkal digunakan dalam pemodelan tsunami secara numerik.

Persamaan *incompressible Navier-Stokes* untuk 2 dimensi yaitu:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = o, \tag{2.10}$$

Persamaan air dangkal dalam konteks dua dimensi memiliki dua komponen penting yaitu persamaan kekekalan massa dan kekekalan momentum x dan y.

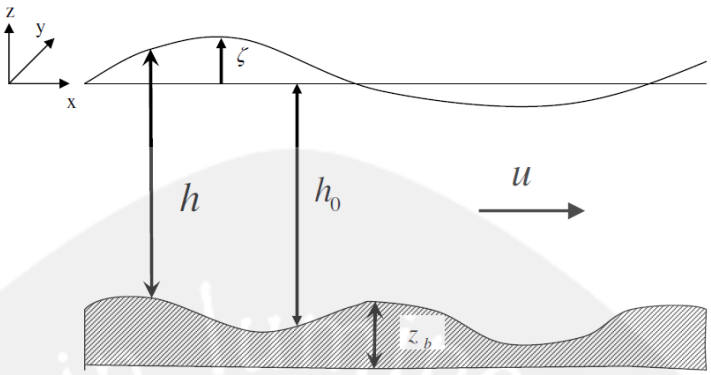
Bentuk persamaan differensial parsial untuk persamaan air dangkal yaitu:

$$\frac{\partial h}{\partial t} + \frac{\partial (uh)}{\partial x} + \frac{\partial (vh)}{\partial y} = o \tag{2.11a}$$

$$\frac{\partial (uh)}{\partial t} + \frac{\partial (u^2 h + \frac{1}{2} g h^2)}{\partial x} + \frac{\partial (uvh)}{\partial y} = o \tag{2.11b}$$

$$\frac{\partial vh}{\partial t} + \frac{\partial (uvh)}{\partial x} + \frac{\partial (v^2 h + \frac{1}{2} g h^2)}{\partial y} = o \tag{2.11c}$$

Dimana h adalah ketinggian air, u dan v adalah kecepatan arah x dan y, t untuk waktu dan g adalah gravitasi. Bentuk $\partial x, \partial y, \partial t$ adalah persamaan diferensial parsial untuk hiperbolik tunggal




**Gambar 2. 7 Gelombang air dangkal**

Untuk model metode lattice Boltzmann D2Q9 menggunakan persamaan air dangkal, bobot arah lattice yang digunakan sama dengan (2.3), tetapi menggunakan fungsi $f^{eq}$ yang berbeda. Untuk persamaan air dangkal fungsi kesetimbangan tergantung pada kedalaman air (h) dan velocity u=(x,t), sehingga dapat ditulis sebagai berikut :

Menghitung ketinggian gelombang:

$$h = \sum_{i=0}^{N-1} f_i(x,t) \quad (2.12a)$$

Menghitung kecepatan:

$$h.u = \sum_{i=0}^{N-1} e_{i.} f_i(x,t) \text{ atau } u = \frac{1}{h}\sum_{i=0}^{N-1} e_{i.} f_i(x,t) \quad (2.12b)$$

Dimana h adalah ketinggian gelombang, N adalah jumlah lattice, dan $e_i$

adalah arah kecepatan lattice.

Fungsi kesetimbangan yang digunakan ($f^{eq}$) untuk model D2Q9 yang juga digunakan oleh salmon (1999) dapat dituliskan sebagai berikut:

$$f_i^{eq}(h,u) = \begin{cases} h - f_0 * h\left(\frac{15gh}{2} - \frac{3}{2}u^2\right), & i = 0 \\ f_i * h\left(\frac{3gh}{2} + 3e_i.u + \frac{9(e_i u_i)^2}{2} + \frac{3u^2}{2}\right) & i = 1,\dots,8 \end{cases} \quad (2.13)$$

Fungsi kesetimbangan yang digunakan ($f^{eq}$) untuk model D2Q9 yang juga digunakan oleh geveler (2010) untuk asumsi air dangkal, dapat dituliskan sebagai berikut:

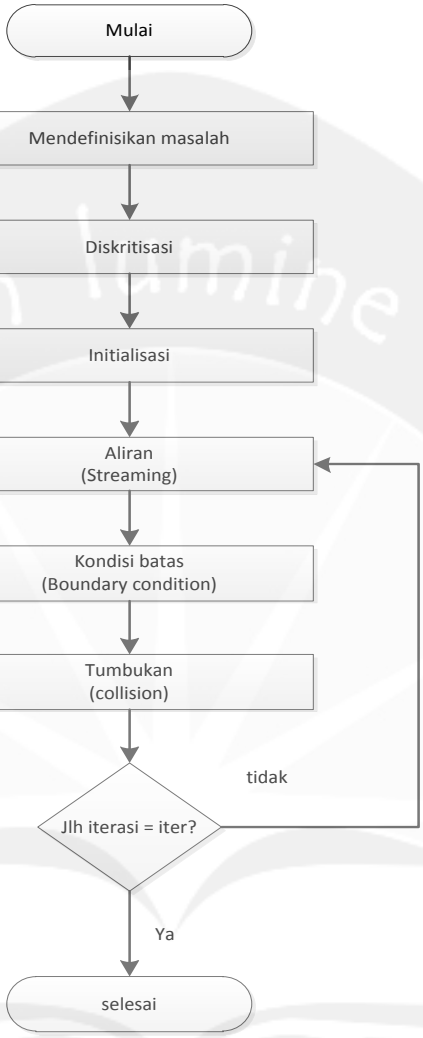
$$f_i^{eq} = \begin{cases} h * (1 - \frac{5gh}{6e^2} - \frac{2}{3e^2}u.u) & i = 0 \\ h * \left(\frac{gh}{6e^2} + \frac{e.u}{3e^2} + \frac{e.u}{2e^4} + \frac{u.u}{6e^2}\right) & i = 1,2,3,4 \\ h * \left(\frac{gh}{24e^2} + \frac{e.u}{12e^2} + \frac{e.u}{8e^4} + \frac{u.u}{24e^2}\right) & i = 5,6,7,8, \end{cases} \quad (2.14)$$

Fungsi keseimbangan persamaan (2.14) merupakan fungsi keseimbangan yang terbaru digunakan untuk asumsi air dangkal maka persamaan ini digunakan untuk simulasi fluida perambatan gelombang tsunami.

## 2.5. Algoritma Metode Lattice Boltzmann

Untuk memudahkan penyusunan metode lattice Boltzmann maka digunakan

bagan alir seperti berikut:




**Gambar 2. 8 Bagan alir untuk metode lattice Boltzmann**

Dari bagan alir diatas dapat kita uraikan sebagai berikut:

a. Mendefinisikan masalah untuk metode lattice Boltzmann dengan menentukan ruang dimensi yang akan digunakan.

b. Diskritasi: dalam ruang dimensi, akan dibentuk mesh dalam bentuk grid seragam dalam koordinat kartesian sesuai dengan geometrid dan

domain komputasi.

c. Inisialisasi: untuk menginisialisasi fungsi distribusi, bukan hanya struktur lattice yang harus ditentukan, kecepatan awal dan tekanan awal juga diperlukan, waktu relaksasi dan multi grid metode lattice Boltzmann.

d. Aliran: menentukan perpindahan partikel dari satu titik lattice ke titik lattice lainnya.

e. Kondisi batas: menentukan kondisi batas dan membangun fungsi distribusi sesuai dengan kebutuhan.

f. Tumbukan: menentukan dan menghitung fungsi setiap tumbukan partikel unutk setiap node di dalam wilayah.

g. Apakah jumlah iterasi sesuai dengan yang ditentukan, jika tidak kembali keproses no 4. Jika ya maka proses selesai.

## 2.6. Komputasi Paralel

Waktu eksekusi program secara serial dapat dikurangi dengan dua cara tanpa melakukan *refactoring*. Pertama dengan meningkatkan jumlah operasi yang dilakukan tiap detik dan menggunakan prosesor yang lebih cepat. Pada periode tahun 1980-an dan 90-an, terjadi sebuah tren dimana perkembangan kecepatan CPU clock meningkat setiap tahun. Namun, sejak tahun 2003 perkembangan ini menurun secara drastic. Teknologi prosesor dengan kecepatan 4GHz pun tidak mampu

mengatasi masalah konsumsi energy dan suhu pada prosesor yang cukup tinggi (Kirk dan Hwu, 2010).

Cara kedua adalah dengan melakukan komputasi secara parallel. Dengan membagi *threads* kebeberapa prosesor, sehingga kita dapat meningkatkan waktu kecepatan proses tanpa meningkatkan kemampuan prosesor. Peningkatan kecepatan ini dijelaskan oleh hukum Amdahl's:

$$Speed - Up = \frac{1}{1-P} \tag{2.15}$$

Jadi, jika 50% dari eksekusi program serial dapat dilakukan secara parallel, maka kecepatan maksimum hanya 2 kali, jika 90% kecepatan bisa mencapai 100 kali, dengan memberikan *core* yang cukup unutk memproes semua *thread* secara bersamaan.

Dari sudut pandang perangkat keras, kita dapat melakukan komputasi parallel dengan beberapa cara, yaitu:

a. Mengguanakan beberapa CPU dalam satu computer, seperti super computer CRAY dan BlueGene.
b. Menggunakan beberapa computer CPU tunggal secara fisik, dan dihubungkan dalam sebuah jaringan.
c. Menggunakan beberapa *core* pada CPU, seperti pada computer desktop standar.

Cara pertama dan kedua biasanya dilakukan di laboratorium, dengan biaya yang cukup besar, sedangkan cara ketiga terbatas pada jumlah *core* dalam satu CPU. Sehingga alternative yang paling baik adalah dengan menggunakan GPU.

## 2.7. CUDA (Compute Unified Device Architecture)

CUDA (Compute Unified Device Architecture) merupakan arsitektur GPU dari Nvidia yang memungkinkan untuk menjalankan program pada GPU. Hal ini membuat GPU bukan hanya digunakan untuk melakukan kalkulasi grafis saja, tetapi juga untuk melakukan komputasi yang umum (general purpose computing) seperti pada CPU. Arsitektur CUDA mulai diperkenalkan pada GPU Nvidia seri G80 pada tahun 2007.

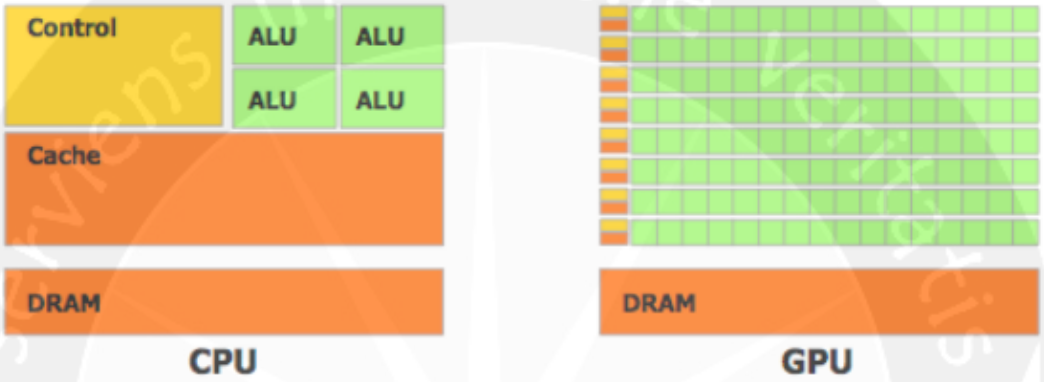
Model pemrograman CUDA adalah dengan membagi pekerjaan yang akan dilakukan kebanyak unit pemroses paling kecil, yakni thread. Setiap thread memiliki memory sendiri dan akan mengerjakan unit pekerjaan yang kecil dan akan berjalan secara bersamaan dengan thread lain, sehingga waktu yang dibutuhkan untuk mengerjakan pekerjaan seluruhnya akan menjadi lebih singkat. Thread-thread tersebut dikelompokkan menjadi block, yaitu kumpulan thread yang memiliki satu memory yang dapat digunakan oleh setiap thread dalam block tersebut secara bersama-sama untuk media komunikasi antar thread tersebut yang dinamakan dengan shared memory. Setiap block tersebut akan dikelompokkan lagi menjadi sebuah grid yang merupakan kumpulan dari semua block yang digunakan

dalam suatu komputasi. (Balevic, Ana, 2009). Arsitekture *core* pada GPU juga mencoba memaksimalkan kinerja untuk program komputasi parallel (Kirk dan Hwu, 2010). Banyaknya *core* dalam sebuah kartu grafis dan masing-masing bersifat *multithreaded.* Hal ini memungkinkan untuk kinerja yang jauh lebih besar dibandingkan dengan CPU tradisional, seperti dilahat pada gamba berikut:


**Gambar 2. 9 Perbandingan kecepatan GPu dengan CPU**

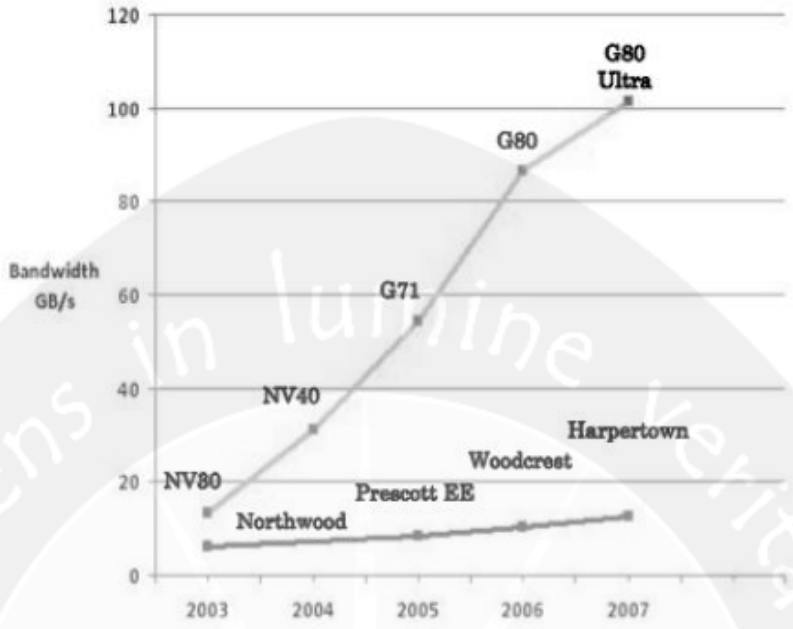
**(http://csgillespie.files.wordpress.com/2011/01/gpu4.png).**

Peningkatkan kinerja disebabkan oleh desai *hardware* GPU yang berbeda dengan CPU, dimana CPU *multicore* menyediakan *cache* besar dan melaksanakan instruksi x86 secara penuh pada setiap *core*, sedangkan GPU *core* yang lebih kecil yang ditunjukan untuk *throughput floating-point.* Untuk perbedaan arsitektur CPU dengan GPU dapat dilihat gambar dibawah ini:



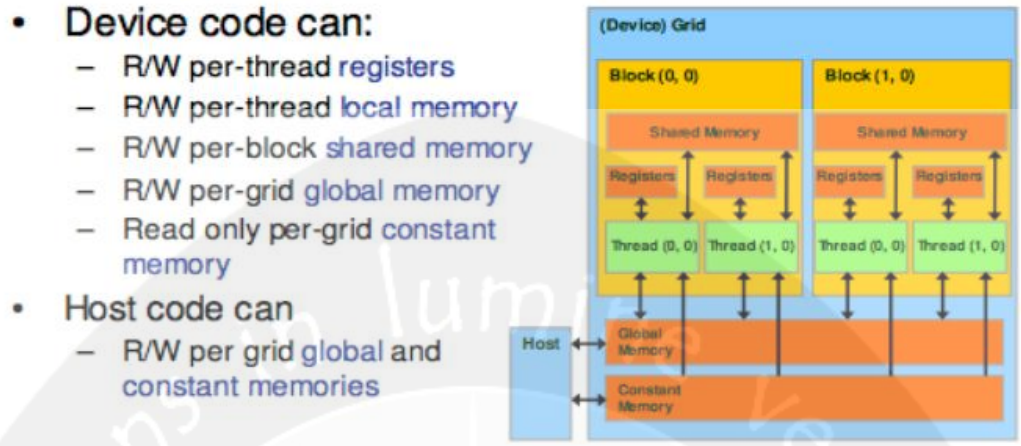

**Gambar 2. 10 Arsitektur CPU dan GPU (Webb,2010)**

Kinerja juga meningkat karena dukungan *bandwith* memori yang cukup besar. Kartu grafis memiliki 10 kali lebih besar dibandingkan CPU, seperti yang ditunjuk pada gambar 2.11, dengan model GPU dapat memiliki kecepatan hingga 100Gb/s.




**Gambar 2. 11 Perbandingan *bandwith* memori CPU dan GPU**

Perangkat keras yang memiliki kemampuan dan fitur CUDA GPU memiliki serangkaian *streaming multiprocessor* yang memiliki kemampuan proses yang cukup tinggi, dan setiap *streaming multiprocessor* memiliki *streaming processor* yang membagi *control logic* dan *instruction cache*. Ada beberapa tipe memori, yang seperti gambar 2.12 setiap *core* memiliki *registers* dan *shared memory* yang terdapat dalam satu *chip* dan memiliki akses yang sangat cepat, *constant memory* yang cepat tetapi bersifat *read only*, berguna untuk menyimpan parameter konstan dan *global memory* memiliki kecepatan yang lambat tetapi ukuran yang sangat besar.




**Gambar 2. 12 Arsitektur CUDA GPU**

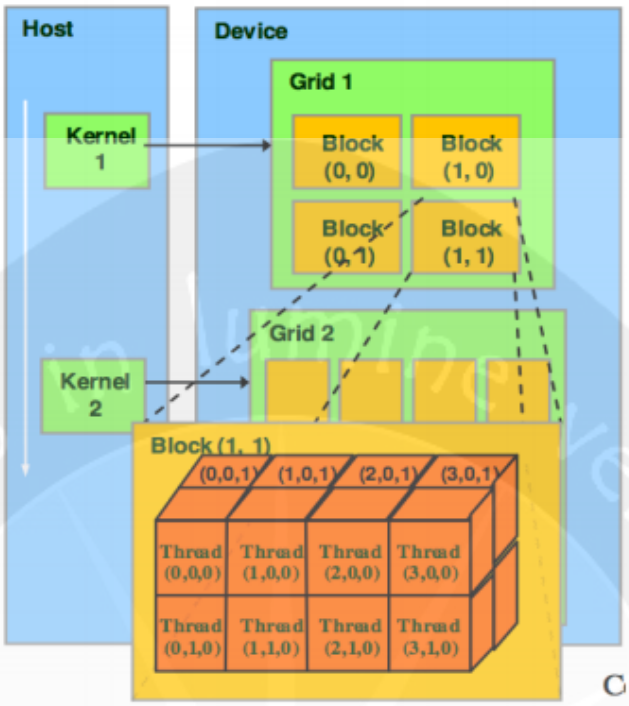
CUDA dapat implementasi pada berbagai macam aplikasi pada area teknologi informasi saat ini, CUDA sendiri dari *software development kit* dan compiler bahasa pemrograman C. seperti diketahui bahwa proses komputasi menggunakan CUDA memerlukan transfer data dari memori GPU ke memori CPU, dan untuk mensiasati permasalahan tersebut dapat dilakukan dengan beberapa cara seperti yang ditunjukkan oleh NVdia sebagai berikut:

a. Memininalisir proses transfer data antara host dengan GPU.
b. Memastikan bahwa akses *global memory* telah disatukan.
c. Meminimalisir penggunaan *global memory* dan lebih mengutamakan *shared memory.*

## 2.8. Model Komputai CUDA

Dari perspektif pemrograman ada tiga operasi dasar yang dibutuhkan untuk menjalankan sebuah aplikasi pada GPU. Pertama adalah melakukan inisialisasi dan melakukan transfer data dari memori *host* ke memori GPU (*global memory*), kedua dengan data yang sudah berada pada GPU, kernel akan dieksekusi sebannyak *n* kali dari total *threads* yang ada pada GPU dan terakhir ketika semua *threads* telah selesai, data dapat kembali ditransfer kembali dari GPU ke *host.*

*Threads* berbentuk seperti sebuah blok, dan dapat juga digunakan pada skema 3 dimensi. Threads akan ditandai sebagai *threadIdx.x, threadIdy.y, threadIdz.z,* dan setiap blok terdiri dari 512 *threads.* Apabila membutuhkan lebih dari 512 *threads* akan akan diformulasikan dalam bentuk 2 dimensi dengan menyusun 2 blok atau yang dikenal sebagai *grid*, dan setiap blok akan ditandai sebagai *blockIdx.y* seperti gambar dibawah ini:




**Gambar 2. 13 Threads block dan grid**

Untuk meluncurkan kernel menggunakan *code* berikut:

```
Test_Kernel <<<dimGrid, dimBlock>>>(Parameters…)
```

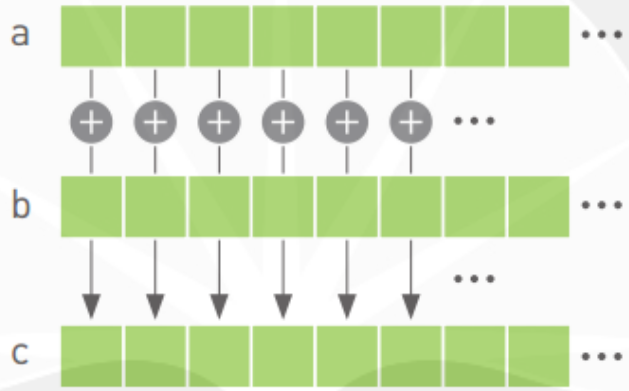
Dimana dimGrid adalah *struct* dari dua tipe data *integer* yang mendefinisikan ukuran blok. Sedangkan untuk mendefinisikan kernel menggunakan sintaks sebagai berikut:

```
__global__ void Test_Kernel (parameters)
{
kernel code
}
```

Dimana kernel tersebut akan dieksekusi pada setiap *threads* selama proses komputasi berlangsung.

## 2.9. Implementasi Parallelisasi pada GPU

Pada bagian ini akan dijelaskan secara teknis proses komputasi pada CPU dengan proses komputasi pada GPU, sehingga dapat ditunjukkan perbedaan konsep pemrograman pada GPU dan konsep pemrograman pada CPU. Disini akan ditunjukkan implementasi dari pemrograman parallel berbasis GPU, proses yang akan dicontohkan adalah proses penjumlahan 2 vektor (Sanders & Kandrot, 2011).




**Gambar 2. 14 Penjumlahan 2 vektor**

Dari gambar diatas dapat dilihat bahwa proses penjumlahan vektor a dan b, dilakukan secara *array*, dimana proses perhitungan dilakukan secara bergantian, proses berikutnya dapat dilaksanakan setelah proses berikutnya selesai. Sehingga apabila *array* yang akan dieksekusi relative banyak maka akan membutuhkan waktu komputasi yang tinggi. Secara umum kode program yang akan digunakan untuk menyelesaikan penjumlahan 2 vektor tersebut adalah sebagai berikut:

```
#include "../common/book.h"

#define N   10

void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

**Gambar 2. 15 *code* CPU penjumlahan 2 vektor**

Dari *code* di atas dapat dijelaskan bahwa proses penjumlahan tersebut dilakukan secara *looping* dari index `tid=0` sampai `tid = N-1`, dengan menggunakan penjumlahan elemen a[ ] dan b[ ] kemudian menempatkan hasil penjumlahan tersebut keelemen c[ ]. Bila kita asumsikan index terakhir adalah 4 dimana setiap *array* dilakukan pada setiap satuan waktu maka diperlukan 4 satuan waktu untuk menyelesaikan satu proses komputasi.

Sedangkan apabila penjumlahan vector tersebut dilakukan dengan menggunakan konsep parallel pada GPU implementasinya akan menjadi berbeda, dimana elemen-elemen tidak akan dieksekusi secara serial seperti sebuah *array* pada pemrograman CPu, tetapi perhitungan elemen-elemen tersebut akan dilakukan pada masing-masing bagian *block threads* pada GPU, dan perintah untuk melakukan hal tersebut disebut kernel.

*Main ()* kode program berbasis GPU agak sedikit berbeda bila dibandingkan dengan *main ()* berbasis CPU, tetapi beberapa bagian kode program berbasi CPU masih digunakan pada GPU, seperti berikut ini:

```
#include "../common/book.h"

#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // this thread handles the data at its
thread id
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
```

```
                                cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                                cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                                cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

**Gambar 2. 16 *code* GPU penjumlahan 2 vektor**

Beberapa perbedaan yang tampak adalah:

a. Alokasi *array* pada perangkat dilakukan dengan menggunakan perintah `cudaMalloc` dimana 2 *array* `dev_a` dan `dev_b` sebagai simpanan nilai input dan satu *array* `dev_c` sebagai simpanan nilai hasil.

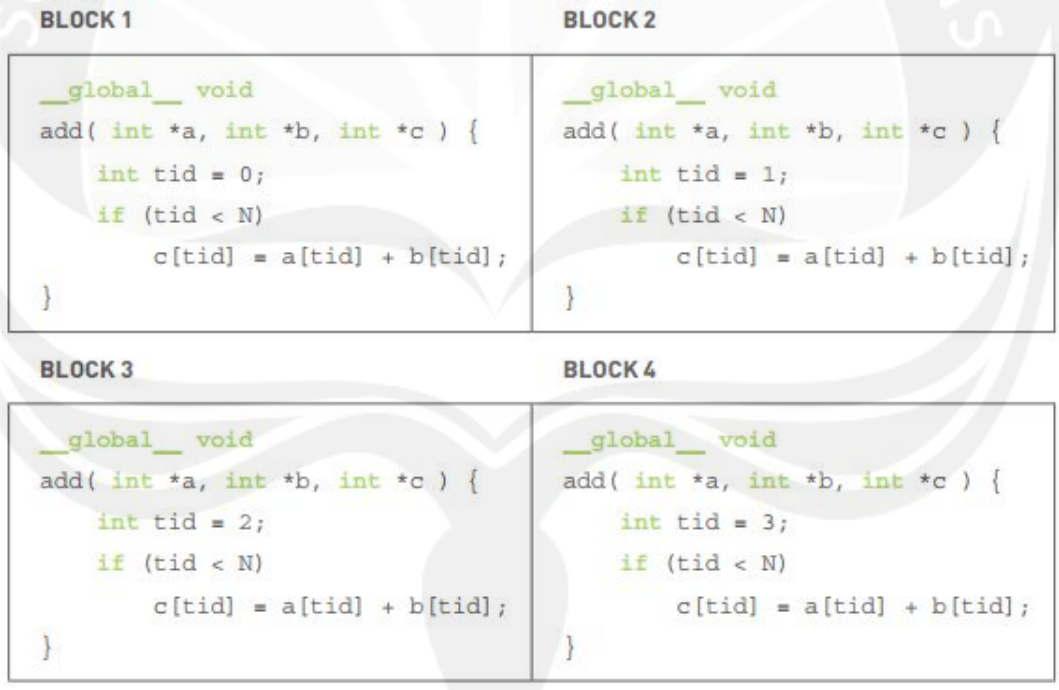
b. Dengan menggunakan `cudaMemcpy()`, nilai data dari *host* di *copy* ke GPU dengan menggunakan para meter `cudaMemcpyHostToDevice` dan kemudian meng-*copy* nilai hasil dari GPU ke *host* dengan menggunakan parameter `cudaMemcpyDeviceToHost`.

c. Setelah semua proses kemudian digunakan pengosongan memori pada GPU dengan perintah `cudaFree`.

Untuk eksekusi program dibutuhkan sebuah kernel, fungsi dari kernel tersebut adalah untuk memerintahkan eksekusi elamen-elemen a[ ], b[ ], c[ ] mulai dari index `tid=0` hingga `tid=N-1` agar dilakukan pada masing-masing *block threads* dimana index pada masing-masing *block* akan ditandai sebagai `blockId.x`, sehingga pembagian *block threads* pada proses komputasi penjumlahan vector akan seperti kode program berikut ini:

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // this thread handles the data at its
thread id
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

**Gambar 2. 17 *Code* pembagian *block threads***




**Gambar 2. 18 Alokasi *block threads* untuk penjumlahan vector**

Kernel tersebut akan dieksekusi dari *host* dengan menggunakan *main () code* di atas, ketika kernel dieksekusi, nilai N akan didefinisikan sebagai banyaknya *block* parallel dan sekumpulan *block* parallel disebut sebagai grid. Setiap *threads* memiliki nilai yang bervariasi, dimana `tid=0` adalah indeks yang pertama dan `tid=N-1` adalah index terakhir. Diasumsikan kita memiliki 4 *blocks* yang bekerja dalam satu waktu yang bersamaan tetapi masing-masing *blocks* memiliki `blockId.x` yang berbeda-beda.

## 2.10. Tsunami pada GPU

Implementasi *hardware* pada GPU berbeda dengan CPU, sebuah GPU memiliki beberapa prosesor dan memiliki beberapa memori yang berbeda dalam satu *hardware*, yaitu *shared memory, constant memory* dan *registers*, dimana masing-masing memiliki ukuran dan *bandwith* yang berbeda. Kita dapat mengimplementasikan komputasi parallel dengan menggunakan beberapa prosesor yang dimiliki GPU, semua prosesor di dalam GPU didesai untuk mengeksekusi satu *code* yang sama sehingga GPU dapat melakukan iterasi secara efektif. Karena GPU tidak dapat mengakses langsung pada memori GPU, maka kita harus mentransfer data antara memori GPU dan memori CPU.

Pada kasus simulasi perambatan gelombang tsunami, nilai lattice akan ditampung salam array sehingga dapat dipertakan kedalam sebuah grid, dimana semua node pada titik grid tersebut akan dihitung secara independen untuk setiap

satuan waktu. Sehubungan dengan hal tersebut maka simulasi perambatan gelombang tsunami ini dapat dilakukan secara parallelisasi dengan cara mengeksekusi semua proses perhitungan nilai node pada titik grid secara bersamaan pada setiap *block threads* yang didefinisikan pada setiap satuan waktu.

Pada proses simulasi perambatan gelombang tsunami ini, OpenGL *driver*s mengalokasikan memori pada GPU, mengoptimalkan pengaturan perangkat keras sehingga dapat mempercepat proses komputasinya dan OpenGL memiliki kelebihan untuk melakukan *render*, dimana kernel mengolah data *image* dan kemudian data *image* tersebut diolah oleh OpenGL.