BAB II

TINJAUAN PUSTAKA

II.1. Penelitian Terdahulu

Penelitian ini dilakukan tidak terlepas dari hasil penelitian-penelitian yang telah dilakukan sebelumnya, dimana penelitian-penelitian tersebut akan peneliti gunakan sebagai bahan perbandingan dan kajian. Dan hasil penelitian yang dijadikan perbandingan pada penelitian ini tidak terlepas dari topik yaitu Simulasi perambatan gelombang suara dengan menggunakan *Finite Difference Time Domain* (FDTD), *Staggered Grid* FDTD dan komputasi parallel dengan menggunakan arsitektur *Compute Unified Device Architecture* (CUDA).

Metode *Finite Difference* telah digunakan sebagai solusi numerik sejak tahun 1920, dan digunakan secara luas pada bebagai bidang. Beberapa bidang yang cukup banyak menggunakan metodologi ini contohnya adalah bidang yang mempelajari tentang gelombang elektromagnetik, seismologi, begitu juga dengan perambatan gelombang suara pada sebuah ruang akustik. Untuk kasus simulasi perambatan gelombang suara pada sebuah ruang akustik, beberapa penelitian yang telah dilakukan sebelumnya menggunakan 2 pendekatan. Pendekatan yang pertama adalah dengan pendekatan secara numerik menggunakan *Finite Difference Time Domain* (FDTD) dan *Digital Waveguide Mesh*. Kowalczyk dan Walstijn (2010) telah berhasil menunjukkan tingkat akurasi yang baik pada penggunaan skema FDTD untuk mensimulasikan perambatan gelombang suara, penelitian tersebut mengimplementasikan skema 3 dimensi yang berfokus pada

kondisi batas media dan pemodelan pada media.

Begitu juga dengan Southern dkk (2009) yang telah berhasil untuk menunjukan teknik auralisasi pada sebuah perambatan gelombang pada ruang akustik, selain itu pada penelitian tersebut ditunjukkan bahwa metode *wave based* adalah metode yang paling efisien untuk mensimulasikan perambatan gelombang suara. Webb dan Bilbao (2011) juga telah berhasil menerapkan metode FDTD untuk memodelkan perambatan dan pemantulan gelombang suara pada sebuah ruang akustik. Selain FDTD, pemodelan akustik juga dapat dilakukan dengan metode *Digital Signal Processing*. Savoija dkk (1996) telah berhasil menyajikan penggunaan struktur *digital waveguide mesh* dengan menghubungkan beberapa *node*. Sedangkan untuk FDTD sendiri dikenal ada 2 macam metode, yaitu *Unstaggered Grid* dan *Staggered Grid*. Gilles dkk (2000) telah melakukan penelitian untuk membandingkan antara *Unstaggered Grid* dan *Staggered Grid* untuk menggambarkan fenomena gelombang elektromagnetik, dan diperoleh hasil bahwa dengan *Staggered Grid* FDTD akan mendapatkan akurasi yang lebih baik bila dibandingkan dengan *Unstaggered Grid*.

Pendekatan yang kedua adalah dengan menggunakan *Raytracing*, pendekatan ini menggunakan teknologi *3D graphics rendering* untuk menggambarkan gelombang suara seperti bentuk sebuah cahaya. Rober dkk (2007) telah menggunakan metode *raytracing* untuk melakukan *render* auralisasi dari sebuah visualisasi perambatan gelombang suara pada sebuah ruang virtual. Dengan menggambarkan gelombang suara seperti halnya sebuah sinar cahaya, para peneliti tersebut kemudian dapat memperhitungkan pantulan gelombang

suara terhadap benda dan batas-batas tertentu pada sebuah ruang virtual. Drumm (2005) juga menggunakan pendekatan yang sama, yang dikenal sebagai *Adaptive Beam Tracing*. Akan tetapi, meskipun teknik ini dapat memodelkan pantulan gelombang suara secara akurat, metode ini tidak memungkinkan untuk dapat menampilkan efek difraksi gelombang suara.

Akan tetapi metode-metode yang tersebut diatas memiliki kebutuhan komputasi yang cukup berat, sehingga untuk menjalankan program tersebut akan diperlukan waktu yang cukup lama dan spesifikasi perangkat keras yang tinggi. Permasalahan tersebut dapat diselesaikan dengan menggunakan konsep komputasi paralel, akan tetapi dengan menyediakan beberapa komputer yang kemudian dirangkai secara seri akan membutuhkan biaya yang cukup tinggi. Dalam beberapa tahun terakhir perkembangan arsitektur GPU berkembang cukup pesat, dan GPU masa kini dapat diprogram untuk melakukan proses komputasi, bahkan memiliki kemampuan komputasi yang lebih baik dari CPU. Tsingos (2009) telah melakukan penelitian terhadap penggunaan *Programmable* GPU untuk melakukan proses komputasi pada sebuah kasus auralisasi pada sebuah gelombang suara. Dan berdasarkan pada penelitian tersebut diambil kesimpulan bahwa dengan memadukan CPU dan GPU maka akan diperoleh sebuah kinerja yang luar biasa. Savioja dkk (2010) memanfaatkan aplikasi GPU untuk membangun sebuah pemodelan dan auralisasi gelombang suara dari sebuah ruang akustik, simulasi tersebut dilakukan dengan metode berbasis *Wavebased* dan *Ray Based* dan hasilnya diperoleh percepatan proses komputasi yang cukup signifikan bila dibandingkan dengan komputasi tradisional menggunakan CPU, selain itu

faktor penyusunan algoritma juga sangat berpengaruh terhadap efektif atau tidaknya proses komputasi pada GPU. Selain pemanfaatan komputasi GPU pada kasus gelombang suara Stock dan Gharakhani (2008) mencoba untuk menguji penerapan algoritma yang tepat pada sebuah simulasi *N-body*, dan dengan pemilihan algoritma yang tepat ternyata proses dapat berjalan 17 kali lebih cepat. Kirk dan Hwu (2010) memperkenalkan konsep GPU sebagai komputer paralel, dimana dengan arsitektur GPU masa kini memungkinkan untuk memiliki banyak *core* dalam sebuah prosesor GPU, sehingga dapat dimanfaatkan untuk mempercepat proses komputasi pada berbagai bidang ilmu pengetahuan.

Arsitektur komputasi paralel pada GPU yang saat ini banyak digunakan adalah *Compute Unified Device Architecture* (CUDA). CUDA memiliki 3 komponen dasar untuk mengatur proses pengolahan data paralel, komponen tersebut adalah CUDA *driver*, CUDA *Application Programming Interface* (API), dan CUDA *mathematical libraries*. Ketiga komponen tersebut berjalan menggunakan *compiler* C (OČKAY dkk, 2008). Salah satu arsitektur CUDA yang cukup banyak digunakan adalah *Tesla* dimana pada arsitektur ini sebuah prosesor GPU memiliki 128 *multi-threaded processor* dan kecepatan akses memori hingga 76.8 Gb/s. Maciol dan Banas (2008) pernah melakukan pengujian terhadap arsitektur *Tesla* untuk aplikasi perkalian Vektor matriks, dan kemudian dari hasil pengujian tersebut ditarik kesimpulan bahwa untuk mencapai kecepatan yang maksimal diperlukan perpaduan perangkat keras yang saling mendukung untuk proses komputasi pada kernel. Beberapa pembatasan yang dimiliki oleh CUDA adalah bahwa *Threads* dan *threads blocks* dapat diciptakan dengan menggunakan

atau melibatkan kernel paralel (Nickolls dkk, 2008). Kemudian Stratton dkk (2008) memperkenalkan sebuah *framework* yang disebut dengan M-CUDA, yang memungkinkan program CUDA dapat dieksekusi secara efisien pada *shared memory*, dan *multi-core* CPUs. Dan pada *framework* M-CUDA terdiri atas beberapa transformasi *compiler* dan sistem waktu berjalan untuk eksekusi paralel.

Beberapa penelitian sebelumnya telah menerapkan metode FDTD pada GPU, baik pada skema 2 dimensi atau 3 dimensi. Seperti yang telah dilakukan oleh Valcarce dkk (2008) dan (2009) dimana berhasil menunjukkan percepatan sebesar 540 kali pada FDTD 2 dimensi untuk aplikasi prediksi cakupan area sinyal radio dengan menggunakan GPU dengan membandingkan implementasi pada Matlab. Adams dkk (2007) juga berhasil menunjukkan percepatan sebesar 429 kali pada aplikasi metode FDTD 3 dimensi untuk penyelesaian persamaan *Maxwell's*. Begitu juga dengan Micikevicius (2009) yang memaparkan tentang paralelisasi pada proses *3D Finite Difference* dengan menggunakan CUDA, dimana akses data digunakan sebagai perhitungan untuk menentukan implementasi yang optimal pada proses komputasi seperti diskretisasi persamaan gelombang yang pada waktu itu sangat diminati dalam komputasi seismik.

Beberapa peneliti juga telah menerapkan metode FDTD pada *multiple* GPU. Seperti yang telah dilakukan oleh Micikevicius (2009), dimana pada penelitian tersebut dapat ditunjukkan percepatan linear untuk metode FDTD 3 dimensi. Begitu juga dengan Michea dan Komatitsch (2010) yang menunjukkan akselerasi linier dari penggunaan 25 GPU pada sebuah aplikasi simulasi perambatan gelombang dengan menggunakan metode FDTD 3 dimensi, dimana

dengan *multiple* GPU dapat diperoleh percepatan sebesar 20-60 kali bila dibandingkan dengan komputasi serial. Beberapa peneliti sebelumnya juga menerapkan metode *grid* baik dalam skema 2 atau 3 dimensi menggunakan GPU. Seperti pada penelitian yang telah dilakukan oleh griebel dan zaspel (2010) ketika menerapkan *multiple* GPU pada persamaan Navier Stokes 3 dimensi dengan menggunakan 8 GPU. Chen dkk (2010) juga menunjukkan peningkatan yang signifikan dalam penggunaan algoritma FFT dalam skala besar.

Selain untuk menyelesaikan permasalahan numerik, CUDA juga dapat dimanfaatkan untuk menyelesaikan permasalahan selain komputasi numerik seperti yang dijelaskan pada penelitian LÍŠKA dan OČKAY (2007), Govindaraju dkk (2004) memanfaatkan GPU untuk proses komputasi basis data, dan pada penelitian tersebut peneliti tersebut telah berhasil mengembangkan algoritma untuk operasi basis data dengan memanfaatkan konsep paralel pada GPU. Begitu juga dengan Bakkum dan Skadron (2008) yang berhasil menunjukan bahwa dengan menggunakan GPU mereka dapat mempercepat proses *query*, dan peneliti tersebut memiliki keyakinan bahwa hal tersebut juga akan berlaku untuk *query SELECT* lainnya. Che dkk (2008) juga mempelajari mengenai penerapan GPU pada kegiatan komputasi secara umum seperti Simulasi Lalu Lintas, Simulasi Thermal, dan *K-Means*. Mereka berhasil menunjukkan bahwa aplikasi tersebut dapat dilakukan lebih cepat hampir 40 kali bila dibandingkan dengan implementasi pada CPU biasa.

Akan tetapi percepatan proses komputasi pada GPU selain dipengaruhi oleh spesifikasi dan kemampuan *hardware* GPU juga dipengaruhi oleh faktor

penyusunan algoritma program. Seperti yang ditunjukkan oleh OČKAY et al (2010), bahwa untuk mendapatkan performa yang baik itu tidak hanya sekedar dapat menerapkan suatu kasus pada GPU tetapi juga harus memperhatikan batasan kemampuan *hardware* dan alokasi memori yang mana hal tersebut ditentukan pada penentuan algoritma kernel.

II.2. *Finite Difference Time Domain*

Finite Difference dapat digunakan untuk melakukan pendekatan secara diskrit dengan *partial differential* untuk menggambarkan perambatan gelombang suara. Dalam kasus perambatan gelombang suara pada ruang akustik, pendekatan numerik dengan menggunakan FDTD dan arsitektur CUDA ditunjukkan oleh Kowalczyk dan Van Walstijn (2010), dengan menerapkan skema 3D yang berfokus pada kondisi batas dan pemodelan pada media. Webb dan Bilbao (2011) juga menggunakan FDTD untuk memodelkan perambatan gelombang suara dan *reverb* pada ruang akustik.

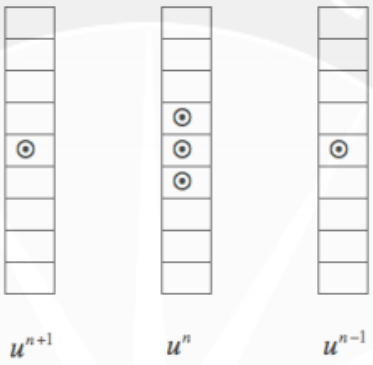
Akan tetapi pada penelitian ini peneliti tidak akan membahas mengenai persamaan model yang digunakan, tetapi lebih pada proses komputasi dari persamaan tersebut. Pada *Finite Difference* perambatan gelombang akan digambarkan dengan sebuah perpindahan nilai secara diskrit pada sebuah grid.

Pada skema 1 dimensi setiap *array* akan digambarkan dalam bentuk sebuah garis, dan besaran nilainya akan terlihat pada panjangnya garis (Gambar 2.1). Pada setiap langkah awal *state* perhitungan akan dinotasikan dengan U^{n+1} , dan pada notasi langkah sebelumnya U^{n} . Untuk 2 langkah sebelumnya

dinotasikan dengan U^{n-1} . Pada setiap langkah waktu akan menghitung nilai pada setiap titik pada *grid*. Persamaan untuk 1 dimensi :

$$u_l^{n+1} = Cu_l^n + D(u_{l-1}^n + u_{l+1}^n) - u_l^{n-1} \quad (1)$$

Nilai C dan D adalah konstan.

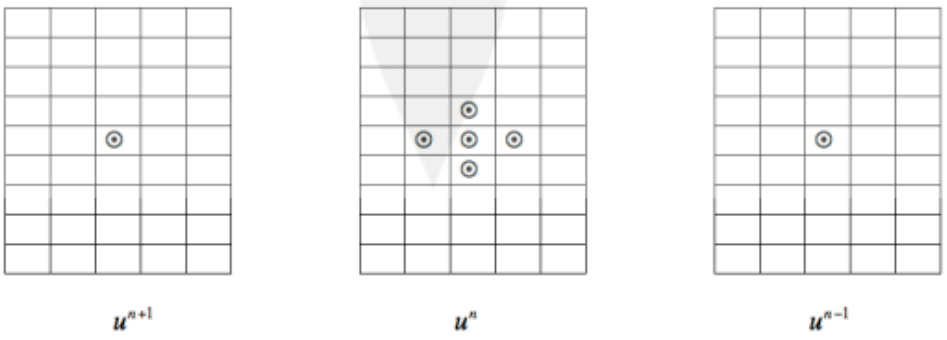


Gambar 2.1. Skema FDTD staggered Grid 1 dimensi

Dan untuk bentuk 2 dimensi akan menggunakan persamaan berikut :

$$u_{l.m}^{n+1} = Cu_{l.m}^n + D(u_{l-1,m}^n + u_{l,m-1}^n + u_{l,m+1}^n) - u_{l,m}^{n-1} \quad (2)$$

Sehingga pada skema 2 dimensi *array* akan digambarkan sebagai sebuah kotak seperti terlihat pada gambar 2.2.

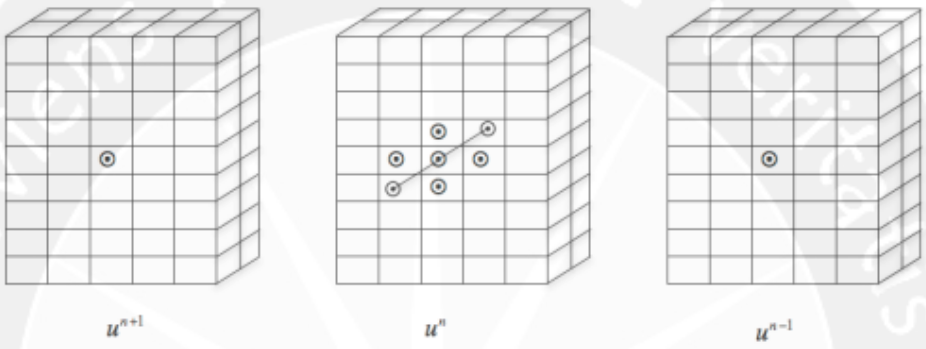


Gambar 2.2. Skema FDTD staggered Grid 2 dimensi

Dan untuk persamaan 3 dimensi :

$$u_{l.m,n}^{n+1} = Cu_{l.m,n}^{n} + D(u_{l-1,m,n}^{n} + u_{l+1,m,n}^{n} + u_{l,m-1,n}^{n} + u_{l,m+1,n}^{n} + u_{l,m,n-1}^{n} + u_{l,m,n+1}^{n}) - u_{l,m,n}^{n-1} \quad (3)$$

Untuk skema 3 dimensi *array* akan digambarkan dalam bentuk kubus seperti pada gambar 2.3 berikut :



Gambar 2.3**.** Skema FDTD staggered Grid 3 dimensi

Sehingga untuk *update* nilai diperlukan 2 elemen yang mendukung selama proses komputasi, yaitu yang pertama adalah mendefinisikan nilai awal dan input data, dan yang kedua adalah menentukan nilai batas kondisi atau *boundary*. Dan nilai batas kondisi tersebut akan menjadi ukuran grid yang digunakan selama proses komputasi.

Dan untuk algoritma untuk selama proses komputasi adalah sebagai berikut :

1.1. Mendefinisikan parameter dan inisialisasi memori untuk grid yang digunakan.

1.2. Menentukan kondisi awal atau *initial condition*.

1.3. *Looping* pada setiap satuan waktu.

1.3.1. Perbarui nilai u^{n+1} berdasarkan nilai u^{n} dan u^{n-1}, dengan memperhatikan nilai kondisi batas yang telah ditentukan.

1.3.2. Pindah *pointer* memori untuk menggeser *grid* pada langkah berikutnya.

II.3. *Staggered Grid Finite Difference Time Domain*

Metode FDTD yang digunakan pada penelitian ini menggunakan metode Yee (Yee, 1966) yang pernah digunakan untuk menyelesaikan persamaan Maxwell untuk perambatan gelombang elektromagnetik menggunakan skema leapfrog pada S*taggered Cartesian Grids*. Metode ini dapat digunakan pada kasus akustik dengan menggunakan orde pertama dimana kecepatan (*Velocity*) dan tekanan (*Pressure*) ditambah dengan persamaan differensial. Dimana *u* dan *v* adalah *x* dan *y* dari komponen kecepatan partikel, dan *p* adalah *pressure*, ρ adalah densitas, *c* kecepatan suara. Dari persamaan tersebut kemudian diturunkan menjadi persamaan berikut :

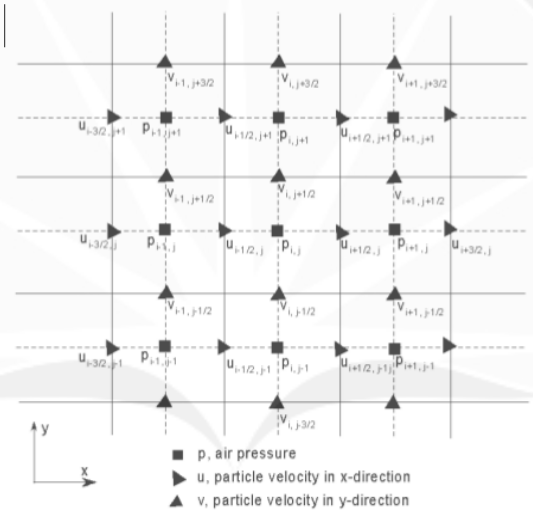
$$p_{i,j}^{n+1} = p_{i,j}^{n} - dt\left(\frac{u_{i+1,j} - u_{i,j}}{dx} + \frac{v_{i,j+1} - v_{i,j}}{dy}\right) \tag{4}$$

$$u_{i+1/2,j}^{n+1/2} = u_{i+1/2,j}^{n-1/2} - dt\left(\frac{p_{i+1/2,j}^{n} - p_{i,j}^{n}}{dx}\right) \tag{5}$$

$$v_{i,j+1/2}^{n+1/2} = u_{i,j+1/2}^{n-1/2} - dt\left(\frac{p_{i,j+1/2}^{n} - p_{i,j}^{n}}{dy}\right) \tag{6}$$

Dari Persamaan tersebut kemudian digunakan untuk mengembangkan

kode FDTD 2 dimensi, langkah yang pertama pada kode FDTD 2 dimensi tersebut adalah menghitung jumlah nilai-nilai awal pada setiap langkah waktu, kemudian nilai awal tersebut dilakukan *looping* terhadap setiap satuan waktu selama proses komputasi. Dan dalam *looping* ini, yang dihitung adalah partikel kecepatan, tekanan, dan batas kondisi nilai. Pada gambar 2.4 ditunjukkan bagaimana persamaan diatas digunakan untuk menghitung nilai-nilai dari *pressure* dan *velocity* dari setiap partikel dari nilai-nilai pada titik sebelumnya.



Gambar 2.4. Skema FDTD staggered Grid 2 dimensi pada sumbu x, y (Liu dan Albert, 2006).

Pada penelitian sebelumnya, *staggered grids* umumnya lebih banyak digunakan dibandingkan *unstaggered grids* untuk beberapa alasan. Seperti yang diungkapkan pada penelitian Gilles (2000), pada *unstaggered grids*, vektor medan elektrik dan magnetik terletak pada sumbu utama grid dan vektor medan dengan komponen terkolokasi. Pembedaan titik tengah pada *unstaggered grids* dapat menyebabkan osilasi numerik yang aneh pada proses *decoupling* (Press, 1992).

Terlebih lagi pada skema *unstaggered grids* menghasilkan kesalahan empat kali lebih besar untuk fase kecepatan numerik bila dibandingkan dengan skema *staggered grids*. Dengan mengesampingkan masalah-masalah tersebut, *unstaggered grids* memiliki beberapa keuntungan bila digunakan untuk pemodelan fenomena gelombang optik. Sebagai contoh, skema *unstaggered grids* memungkinkan untuk menggunakan frame jendela koordinat untuk pelacakan denyut pada perambatan optikal untuk jarak yang cukup jauh (Fidel dkk, 1997). Yang terpenting, penggunaan grid dengan komponen vektor medan terkolokasi efisien dan akurat untuk pemodelan perambatan gelombang optik non linier baik pada skema 2 dimensi ataupun 3 dimensi. Alasan dari fakta tersebut adalah bahwa pada pemodelan non linier membutuhkan perhitungan *high order* dari intensitas medan elektrik pada titik node di setiap satuan waktu. Jika komponen vektor medan elektrik tidak terkolokasi, seperti halnya dengan skema yee staggered, intensitas medan pada titik yang diberikan harus di interpolasi dari komponen medan yang berada di sekitarnya. Interpolasi ini cukup berat bila dilihat dari sisi efisiensi komputasional dan akurasi, dan dapat dihindari bila vektor medan elektrik terkolokasi.

II.4. Komputasi Parallel

Ada dua cara untuk dapat mengurangi waktu eksekusi program secara serial tanpa melakukan refactoring. Yang pertama adalah dengan meningkatkan jumlah operasi yang dilakukan setiap detik, dengan menggunakan prosesor yang lebih cepat. Sepanjang 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 dramatis. Teknologi prosesor dengan kecepatan 4GHz pun tidak mampu mengatasi masalah konsumsi energi dan suhu pada prosesor yang cukup tinggi , Kirk dan Hwu (2010).

Cara yang kedua adalah dengan melakukan komputasi dari beberapa program secara bersamaan dalam seri paralel. Dengan membagi threads ke beberapa prosesor, sehingga kita dapat meningkatkan kecepatan waktu proses tanpa harus meningkatkan kecepatan prosesor. Peningkatan kecepatan ini dijelaskan oleh hukum Amdahl's :

$$Speed - Up = \frac{1}{1 - P} \quad (7)$$

Jadi, jika 50% dari eksekusi program serial dapat dilakukan secara paralel, maka kecepatan maksimum hanya 2 kali. Jika 90% dilakukan secara paralel, maka kecepatan maksimum yang diperoleh adalah 10 kali. Pada 99%, kecepatan bias mencapai 100 kali, dengan memperikan coreyang cukup untuk memproses semua thread secara bersamaan.

Pada persamaan *Finite Difference* proses dilakukan secara bertahap dari waktu ke waktu. Sehingga *Finite Difference* adalah sebuah contoh kasus yang tepat untuk dapat menunjukan peningkatan kecepatan pada komputasi paralel, dimana masing proses akan dibagi ke setiap *core*.

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

a. Menggunakan beberapa CPU dalam satu komputer, seperti super komputer

CRAY dan BlueGene.

b. Menggunakan beberapa komputer dengan CPU tunggal yang secara fisik saling berhubungan melalui sebuah jaringan.

c. Menggunakan beberapa *core* pada satu CPU, seperti umumnya pada komputer desktop standar.

Cara pertama dan yang kedua biasa digunakan pada sebuah laboratorium dengan biaya yang cukup besar. Dan cara ketiga terbatas pada jumlah corepada CPU yang umum di pasaran, seperti 2 atau maksimal 4 *core* dalam satu CPU. Sehingga alternatif yang paling baik adalah dengan menggunakan GPU.

II.5. Perkembangan Komputasi GPU

Penelitian awal mengenai penggunaan prosesor pada GPU untuk proses komputasi dimulai pada akhir tahun 1970 (Harris dkk, 2002), dengan menggunakan mesin seperti Ikonas (England, 1978) dan *Pixel Machine* (Potmesil dan Hoffet, 1989). Dengan penemuan kartu grafis pada akhir tahun 1990, beberapa pengembang melakukan eksperimen untuk melakukan proses perhitungan-perhitungan numeris menggunakan GPU. Trendall dan Stewart (2000) melakukan penelitian mengenai pemanfaatan tersebut pada tahun 2000 yang kemudian dikenal sebagai *General Purpose Computing on GPU's* (GPGPU). Akan tetapi keterbatasan perangkat keras dan perangkat lunak menjadikan penggunaan teknik ini menjadi terbatas. Pengembangan aplikasi menjadi hal yang sulit dan melelahkan, sebagai contoh untuk mendapatkan hasil

proses komputasi berarti harus menuliskan hasil tersebut ke sebuah *pixel frame buffer*, dan kemudian menggunakan *pixel frame buffer* sebagai masukkan *texture map* untuk *pixel shader fragmen* pada tahap proses komputasi berikutnya, dan hal ini adalah yang sangat tidak efisien bila dipandang dari sudut pandang pemrograman.

Pada tahun 2007 Nvidia meluncurkan teknologi CUDA untuk pertama kalinya. Bersamaan dengan hal itu pada tahun 2008 Apple's meluncurkan OpenCL, hal ini memungkinkan bagi para ilmuan untuk memanfaatkan potensi dari Komputasi GPU. OpenCL digunakan pada pada sistem operasi terbaru Apple, seperti *Snow Leopard*, dan menggunakan pemrograman berbasis C, C++, Apple Inc (2009). CUDA memerlukan SDK sebagai ekstensi untuk pemrograman menggunakan bahasa pemrograman C. Ini merupakan keuntungan untuk dunia pemrograman yang ingin mengembangkan komputasi GPU.

II.6. *Compute Unified Device Architecture* (CUDA)

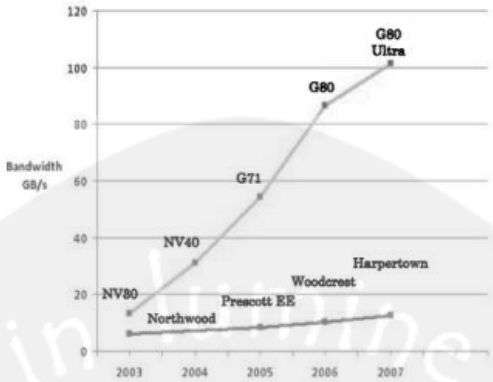
Ketika arsitektur prosesor *multicore* seperti quad coreIntel i7 berusaha untuk memaksimalkan kecepatan eksekusi program sekuensial, arsitektur *core* pada GPU juga mencoba untuk memaksimalkan kinerjanya untuk program komputasi paralel Kirk dan Hwu (2010). Kartu grafis seperti Nvidia Tesla C1060 dan GT200 berisi 240 *core*, dan masing-masing bersifat *multithreaded*. Hal ini memungkinkan untuk kinerja yang jauh lebih besar dibandingkan dengan CPU tradisional, seperti dapat dilihat pada gambar 2.5 berikut :

Gambar 2.5. Perbandingan kecepatan GPU dengan CPU (Webb, 2010).

Peningkatan kinerja ini disebabkan oleh desain *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 ditujukan untuk *throughput floating-point*. Untuk perbedaan arsitektur CPU dengan GPU dapat dilihat pada gambar 2.6 berikut :

Gambar 2.6. Arsitektur CPU dan GPU (Webb, 2010).

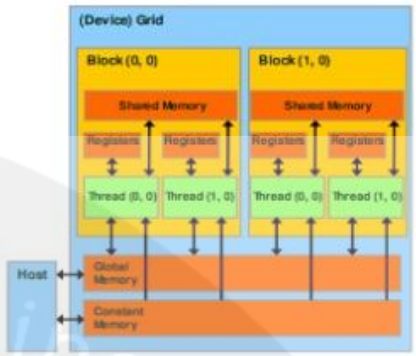
Kinerja juga meningkat karena dukungan *bandwith* memori yang cukup besar. Kartu grafis memiliki 10 kali lebih besar dibandingkan dengan CPU, seperti yang ditunjukkan pada gambar 2.7. Dengan model terkini GPU dapat meiliki kecepatan hingga 100Gb/s.



Gambar 2.7. Perbandingan *bandwith* memori CPU dan GPU (Webb, 2010).

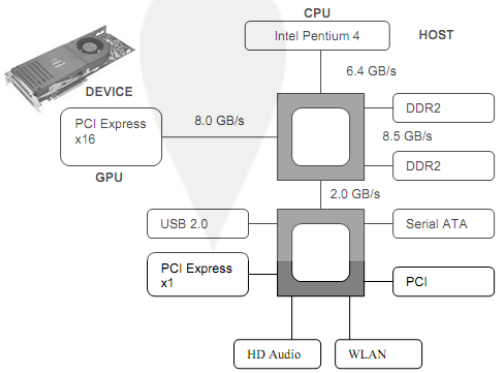
Perangkat keras yang memiliki kemampuan dan fitur CUDA GPU memiliki serangkaian *streaming multiprocessor* yang memiliki kemampuan proses yang cukup tinggi, dan setiap *streaming multiprocessor* memiliki *streaming processor* yang membagi *control logic* dan *instruction cache*. Pada Tesla C1060 terdapat 30 *streaming multiprocessor* dan masing-masing memiliki 8 *streaming processor*, dimana setiap *streaming processor* dapat melakukan proses dalam jumlah yang banyak, dan dapat melakukan ribuan proses dalam setiap aplikasi.

Ada beberapa tipe memori,seperti yang ditunjukkan pada gambar 2.8 setiap *core* memiliki *registers* dan *shared Memory* yang terdapat dalam satu *chip* dan memiliki akses yang sangat cepat. Dan ada juga *constant memory* yang juga cepat tetapi hanya *read only*, berguna untuk menyimpan parameter konstan. Dan yang terakhir adalah *Global Memory*, memiliki speed yang lambat tetapi ukurannya sangat besar. Bahkan Tesla C1060 memiliki *global memory* sebesar 4 gigabyte, *constant memory* sebesar 65 kilobyte, *register* dan *shared memory* sebesar 16 kilobyte pada setiap *streaming multiprocessor*.

Device code can : - R/W per-thread registers - R/W per-thread local memory - R/W per-block shared memory - R/W per-grid global memory - Read only per-grid constant memory **Host code can :** - R/W per-grid global and constant memory	

Gambar 2.8. Arsitektur CUDA GPU (Webb, 2010).

CUDA dapat diimplemetasikan pada berbagai macam aplikasi pada area teknologi Informasi saat ini, CUDA sendiri terdiri dari *software development kit* dan kompiler bahasa pemrograman C. Sistem ini kompatibel dengan mikroprosesor berbasis x86 dan x64 yang di produksi oleh Intel maupun AMD dengan menggunakan system operasi berbasis Linux ataupun Windows. Dan perangkat CUDA GPU terhubung dengan *host* melalui *port* PCI Expres x16 seperti yang ditunjukkan pada gambar 2.9, dimana melalui *port* tersebut akan dimungkinkan memiliki kecepatan interkoneksi sebesar 8 GB per detik (4GB/s *upstream* and 4GB/s *downstream*) (OČKAY dkk, 2008).



Gambar 2.9. Interkoneksi antara *Host* dengan CUDA GPU (OČKAY dkk, 2008).

Akan tetapi perbedaan memori *bandwith* GPU bila dibandingkan dengan CPU cukup tinggi, dimana memori *bandwith* GPU dapat mencapai 100Gb/s sehingga ini dapat menjadi sebuah masalah ketika harus mengeksekusi proses komputasi parallel. Seperti diketahui bahwa pada proses komputasi menggunakan CUDA memerlukan proses transfer data dari memori GPU ke memori CPU. Dan untuk mensiasati permasalahan tersebut dapat dilakukan dengan beberapa cara seperti yang ditunjukkan oleh Nvidia sebagai berikut :

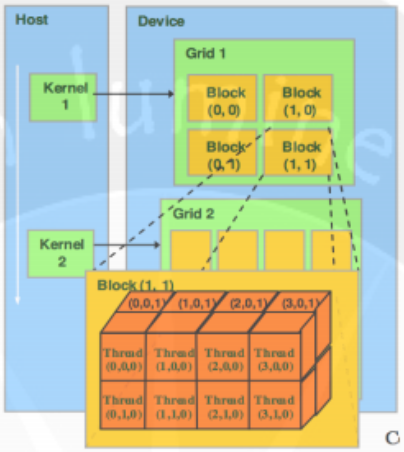
a. Meminimalisir proses transfer data antara *host* dengan GPU.
b. Memastikan bahwa akses *Global Memory* telah disatukan.
c. Meninimalisir penggunaan *Global Memory* dan leih mengutamakan penggunaan *Shared Memory*.

II.7. Model Komputasi CUDA

Dari perspektif pemrograman ada tiga operasi dasar yang dibutuhkan untuk menjalankan sebuah aplikasi pada GPU. Yang 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 sebanyak *n* kali dari total *threads* yang ada pada GPU. Dan terakhir ketika semua *threads* telah selesai, data dapat ditransfer kembali dari GPU ke *host*.

Threads berbetuk seperti sebuah blok, dan dapat juga digunakan pada skema 3 dimensi. *Threads* akan ditandai sebagai `threadIdx.x`, `threadIdx.y` dan `threadIdx.z` dan setiap blok terdiri dari 512 *threads*. Apabila membutuhkan lebih dari 512 *threads* akan akan diformulasikan dalam bentuk 2

dimensi dengan menyusun 2 buah blok atau yang dikenal sebagai *grid*, dan setiap blok akan ditandai sebagai `blockIdx.x` dan `blockIdx.y` seperti yang ditunjukkan pada gambar 2.10 berikut :



Gambar 2.10. *Thread blocks* 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 *grid*, dan `dimBlock` adalah *struct* dari tiga tipe data *integer* yang mendefinisikan ukuran blok.

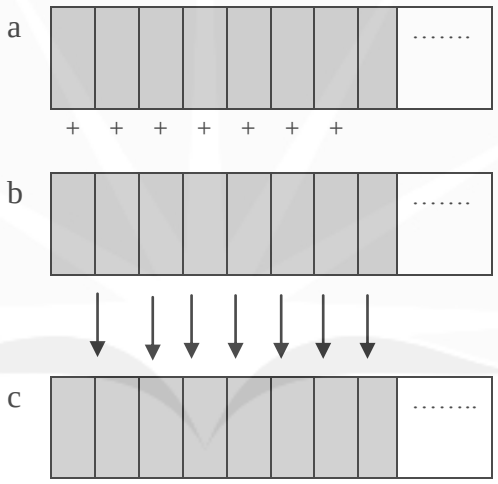
Sedangkan untuk mendefiniskan kernel menggunakan sintaks sebagai berikut :

```
__global__void Test_Kernel(parameters)
{
  kernel code
}
```

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

II.8. Implementasi Parallelisasi pada GPU

Disini akan dijelaskan secara teknis mengenai perbedaan proses komputasi pada CPU dengan proses komputasi pada GPU, sehingga dapat ditunjukkan mengenai perbedaan konsep pemrograman pada GPU dan konsep pemrograman pada GPU. Disini akan ditunjukkan implementasi dari pemrograman berbasis CPU yang kemudian dirubah menjadi pemrograman paralel berbasis GPU, proses yang akan dicontohkan adalah proses penjumlahan 2 vektor.



Gambar 2.11. Penjumlahan 2 Vektor.

Dari gambar 2.7 dapat dilihat bahwa proses penjumlahan vektor a dan b dilakukan secara *array*, dimana proses perhitungan dilakukan secara secara bergantian, proses berikutnya dapat dilaksanakan setelah proses sebelumnya selesai. Sehingga apabila *array* yang akan dieksekusi relatif banyak maka akan membutuhkan waktu komputasi yang semakin tinggi. Secara umum *code* program berbasis CPU yang digunakan untuk menyelesaikan penjumlahan 2 vektor tersebut adalah sebagai berikut (algoritma 2.1) :

```
#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;
}
```

Algoritma 2.1. *Code* penjumlahan 2 vektor dengan menggunakan CPU.

Dari *code* diatas dapat ditunjukan bahwa proses penjumlahan vektor tersebut dilakukan secara *looping* dari index `tid`=0 hingga `tid`=N-1, dengan melakukan penjumlahan elemen a[ ] dan b[ ] kemudian menempatkan hasil penjumlahan tersebut ke elemen c[ ]. Sehingga dapat kita asumsikan bila index terakhir adalah 4 dimana setiap *array* dilakukan pada setiap satuan waktu maka diperlukan 4 satuan waktu untuk menyelesaikan satu proses komputasi.

Sedangkan bila penjumlahan vektor tersebut dilakukan dengan menggunakan konsep paralel pada GPU implementasinya akan menjadi berbeda, dimana elemen-elemen tersebut 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 sebagai kernel.

Main( ) program *code* berbasis GPU agak sedikit berbeda bila dibandingkan dengan *main( )* program untuk *code* berbasis CPU, akan tetapi beberapa bagian dari *code* berbasis CPU masih akan tetap digunakan pada *code* berbasis GPU seperti yang ditunjukan pada algoritma 2.2 dan 2.3 berikut :

```
#include "../common/book.h"
#define N   10

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 ) );
```

Algoritma 2.2. *Main( ) code* penjumlahan 2 vektor dengan menggunakan GPU.

```
// 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
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Algoritma 2.3. *Main( ) code* penjumlahan 2 vektor dengan menggunakan GPU (Lanjutan Algoritma 2.2).

Beberapa perbedaan yang tampak adalah :

1. 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.
2. Dengan menggunakan `cudaMemcpy`( ), nilai data input di *copy* dari *host* ke GPU dengan menggunakan parameter `cudaMemcpyHostToDevice` dan kemudian meng-*copy* nilai hasil dari GPU ke *host* dengan menggunakan parameter `cudaMemcpyDeviceToHost`.
3. Setelah semua proses selesai kemudian dilakukan pengosongan memori pada GPU dengan menggunakan perintah `cudaFree`.

Untuk eksekusi program diperlukan sebuah kernel, fungsi dari kernel tersebut adalah untuk memerintahkan ekseskusi elemen-elemen a[ ], b[ ], dan 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

`blockIdx.x`, sehingga pembagian *block threads* para proses komputasi penjumlahan vektor akan tampak seperti pada gambar 2.11, dan *code* untuk kernel tersebut adalah sebagai berikut :

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

Algoritma 2.4. Kernel utama pada aplikasi penjumlahan 2 vektor.

Block 1

```
__global__ void
add( int *a, int *b, int *c ) {
  int tid = 0;
  if (tid < N)
  c[tid] = a[tid] + b[tid];
}
```

Block 2

```
__global__ void
add( int *a, int *b, int *c ) {
  int tid = 0;
  if (tid < N)
  c[tid] = a[tid] + b[tid];
}
```

Block 3

```
__global__ void
add( int *a, int *b, int *c ) {
  int tid = 0;
  if (tid < N)
  c[tid] = a[tid] + b[tid];
}
```

Block 4

```
__global__ void
add( int *a, int *b, int *c ) {
  int tid = 0;
  if (tid < N)
  c[tid] = a[tid] + b[tid];
}
```

Gambar 2.12. Alokasi *block threads* untuk penjumlahan vektor.

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

berbeda-beda.

II.9. *Finite Difference Time Domain* 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 paralel dengan menggunakan beberapa prosesor yang dimiliki GPU, semua prosesor di dalam GPU di desain untuk mengesekusi satu *code* yang sama sehingga GPU dapat melakukan iterasi secara efektif. Karena GPU tidak dapat mengakses langsung pada memori CPU maka kita harus mentransfer data antara memori CPU dengan memori GPU, oleh karena itu akan lebih baik bila dapat mengurangi proses transfer tersebut. *Device Memory* pada GPU memiliki kapasitas yang cukup besar akan tetapi memiliki kecepatan akses yang lambat. Akan tetapi ketiga memori yang dimiliki oleh GPU memiliki kecepatan yang tinggi dan dapat digunakan sebagai *cache memory* dari CPU. Pemrogramman GPU berbeda dengan pemrogramman CPU dimana harus mendeklarasikan jumlah *on-chip memory* yang akan digunakan oleh program. Penggunaan memori yang efektif dari beberapa memori yang berbeda, khususnya *Shared Memory* merupakan faktor kunci dalam mempercepat proses, karena *on-chip memory* memiliki kecepatan yang cukup cepat dan berukuran kecil maka kita harus dapat mengevaluasi seberapa besar data yang dapat diletakkan di dalam memori tersebut. Untuk dapat melakukan hal tersebut kita harus memahami arsitektur GPU dan menyusun *code* yang efektif untuk GPU.

Pada kasus *Finite Difference Time Domain* (FDTD) proses komputasi dilakukan dengan cara menghitung nilai pada setiap node pada sebuah grid, dimana semua node pada titik grid tersebut akan dihitung secara independen untuk satu sama lain pada setiap satuan waktu. Sehubungan dengan hal tersebut maka FDTD sangat memungkinkan untuk dilakukan parallelisasi dengan cara mengeksekusi semua proses perhitungan nilai node pada titik grid secara bersamaan pada setiap *block threads* yang didefinisikan pada setiap satuan waktu.