BAB II

TELAAH LITERATUR

2.1 Rekayasa Piranti Lunak

Menurut Fritz Bauer di dalam buku yang ditulis oleh K. K. Aggarwal (2005, p5), rekayasa piranti lunak didefinisikan sebagai *"The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines"*. Stephen Schach di dalam buku yang ditulis oleh K. K. Aggarwal (2005, p5) mendefinisikannya sebagai *"A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements"*.

Kedua definisi di atas populer dan diterima oleh kebanyakan orang. Walaupun demikian, karena peningkatan biaya untuk memelihara piranti lunak, tujuannya bergeser menjadi menghasilkan piranti lunak berkualitas yang dapat dipelihara, diimplementasikan tepat waktu, sesuai budget, dan juga memenuhi kebutuhan-kebutuhannya (K. K. Aggarwal, 2005, p5).

Sedangkan Janner Simarmata (2010, p10), mendefinisikan rekayasa piranti lunak sebagai satu bidang ilmu yang mendalami cara-cara pengembangan piranti lunak termasuk pembuatan, pemeliharaan, manajemen organisasi pengembangan piranti lunak, dan sebagainya.

UNIVERSITAS MULTIMEDIA NUSANTARA

Janner Simarmata (2010, p11) menjelaskan bahwa istilah rekayasa piranti lunak secara umum digunakan dalam tiga arti, yaitu:

1. Sebagai istilah umum untuk berbagai kegiatan yang dulunya bernama pemrograman atau analisis sistem
2. Sebagai istilah yang luas untuk analisis teknis dari semua aspek-aspek praktis yang bertentangan dengan teori pemrograman komputer
3. Sebagai istilah yang mewujudkan advokasi suatu pendekatan spesifik ke pemrograman komputer, satu hal yang mendesak yang diperlakukan sebagai profesi rekayasa daripada sebuah seni atau kerajinan, dan advokasi dari kodifikasi praktis yang disarankan dalam bentuk metodologi rekayasa piranti lunak

Rekayasa piranti lunak adalah disiplin rekayasa dengan piranti lunak yang dikembangkan. Biasanya proses melibatkan penemuan pada keinginan klien, menyusunnya di dalam daftar kebutuhan, merancang arsitektur yang mampu mendukung semua kebutuhan, perancangan, pengodean, pengujian, dan pengintegrasian bagian yang terpisah, menguji keseluruhan, penyebaran dan pemeliharaan piranti lunak. Pemrograman hanya menjadi bagian kecil dari rekayasa piranti lunak (Janner Simarmata, 2010, p11).

Disiplin masih berada dalam pertumbuhannya (tahap awal perkembangan/pengembangan) sebagai suatu disiplin rekayasa. Tidak akan pernah ada pengalaman yang cukup, maupun kumpulan data empiris yang cukup untuk secara sistematis memahami dan meramalkan siklus hidup proyek piranti lunak (Janner Simarmata, 2010, p11).

UNIVERSITAS MULTIMEDIA NUSANTARA

Alain Abram, dkk (2004, p2) dalam SWEBOK membagi rekayasa piranti lunak ke dalam 10 area pengetahuan, yaitu:

1. Kebutuhan piranti lunak
2. Perancangan piranti lunak
3. Konstruksi piranti lunak
4. Pengujian piranti lunak
5. Pemeliharaan piranti lunak
6. Manajemen konfigurasi piranti lunak
7. Manajemen piranti lunak
8. Proses piranti lunak
9. Metode dan tool piranti lunak
10. Kualitas piranti lunak

2.2 Bahasa Pemrograman C

Hanif Al Fatta (2006, p1) menjelaskan bahwa pemrograman bahasa C dikembangkan di Bell lab pada awal tahun 1970-an. Bahasa itu diturunkan dari bahasa sebelumnya, yaitu B yang diturunkan dari bahasa sebelumnya, yaitu BCL. Awalnya, bahasa tersebut dirancang sebagai bahasa pemrograman yang dijalankan pada sistem operasi UNIX. Pada perkembangannya, versi ANSI (American National Standard Institute) C menjadi versi yang dominan. Meskipun versi tersebut sekarang jarang dipakai dalam pengembangan-pengembangan baru, tetapi versi itu masih banyak dipakai dalam beberapa pengembangan sistem dan jaringan maupun untuk sistem embedded.

UNIVERSITAS MULTIMEDIA NUSANTARA

2.3 Citra Digital *Grayscale*

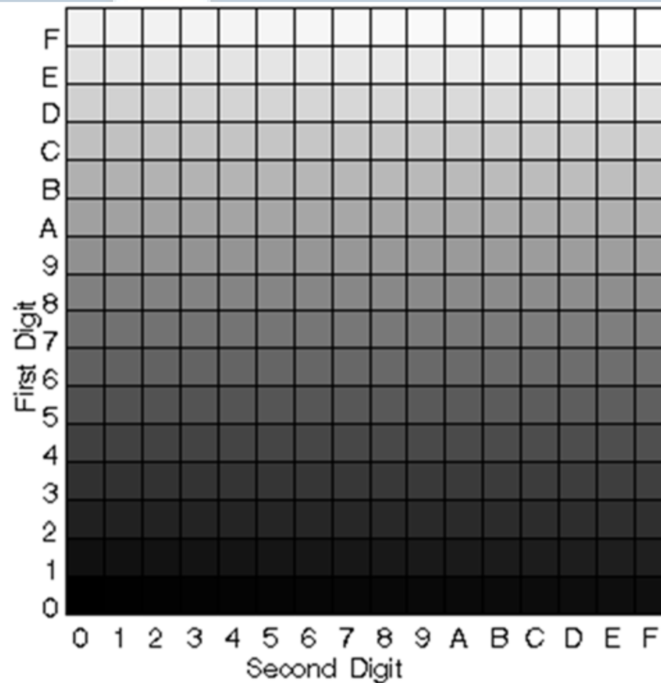
Jayaraman (2009, p21) menjelaskan bahwa citra digital *grayscale* hanya memiliki informasi *brightness*. Setiap nilai *pixel* pada citra digital *grayscale* mengacu kepada jumlah atau kuantitas dari cahaya. Tingkat dari kecerahan dapat dibedakan pada citra digital *grayscale*. Pada citra digital *grayscale*, setiap *pixel* direpresentasikan oleh satu *byte* atau *word*, yaitu nilai yang mereprensentasikan tingkat kepekatan cahaya pada satu buah titik pada citra. Sebuah citra digital delapan bit akan memiliki variasi kecerahan dari 0 sampai dengan 255, dimana 0 merepresentasikan hitam dan 255 merepresentasikan putih. Sebuah citra grayscale hanya merepresentasikan kepekatan cahaya. Setiap *pixel* merupakan skalar proporsional dari kecerahan.

Gonzales dan Richard E. W. (2002, p1) menjelaskan bahwa citra digital berdimensi M x N *pixel* dapat direpresentasikan oleh sebuah matriks berukuran M x N. Elemen (i, j) pada matriks merupakan *pixel* (i, j) pada citra. Nilai elemen matriks a_{ij} adalah nilai intensitas *pixel* (i, j) pada citra.

Gonzales dan Richard E. W. (2002, p52) menjelaskan jumlah bit (b) yang dibutuhkan untuk menyimpan citra digital berdimensi M x N *pixel* dengan 2^k skala keabuan adalah

$$B = M \times N \times k$$

Sebagai contoh: sebuah citra 8 bit (2^8 = 256 skala keabuan) berdimensi 256 x 256 *pixel* membutuhkan memori penyimpanan sebesar 256 x 256 x 8 = 524.288 bit atau 65.536 byte (1 byte = 8 bit).



Gambar 2.1 Intensitas *pixel* dalam hexadesimal
Sumber: http://support.sas.com/techsup/technote/ts688/ts688.html, 21 Juni 2011

2.4 Kompresi Citra

Menurut Rabbani *et al.* (1955, p3) kompresi citra adalah seni atau ilmu untuk melakukan *coding* yang efisien dari data citra, dengan harapan memanfaatkan redudansi pada citra digital untuk mengurangi jumlah bit yang dibutuhkan untuk merepresentasikan citra. Hal ini dapat menyebabkan memori yang dibutuhkan untuk menyimpan berkurang secara signifikan.

2.5 Rasio Kompresi

Menurut Savitri (1999, p45), rasio kompresi yang dicapai, dapat diukur dengan persamaan berikut

$$rasio = 100\% - \left(\frac{ukuran\ citra\ hasil\ kompresi}{ukuran\ citra\ asli}\right) \times 100\%$$

Ukuran citra menyatakan besar penggunaan memori untuk menyimpan citra. Semakin tinggi rasio yang dicapai, maka metode kompresi semakin baik.

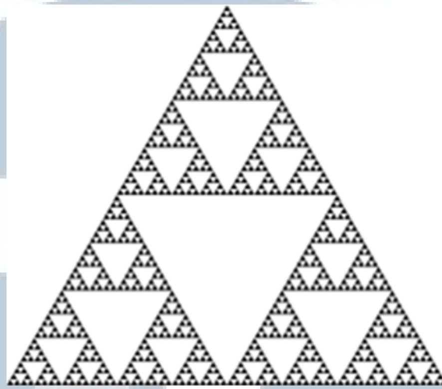
2.6 Metode Fraktal

2.6.1 Citra Fraktal

Menurut Ning Lu (1997, p11), citra fraktal merupakan citra yang bersifat *self-similarity*, yaitu bagian-bagian citra sama dengan citra secara keseluruhan. Contoh citra fraktal adalah segitiga *Sierpinski*.

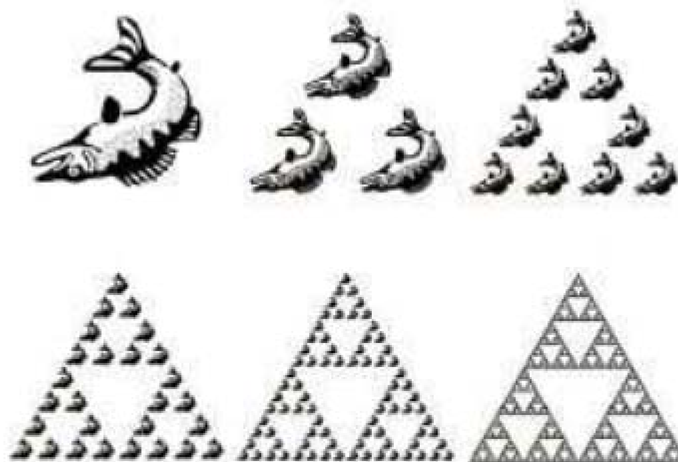
Segitiga *Sierpinski* dibentuk oleh tiga bagian. Ketiga bagian tersebut terlihat sama dengan segitiga *Sierpinski* secara keseluruhan. Ketiga bagian tersebut juga dibentuk oleh tiga bagian yang lebih kecil, yang juga terlihat sama dengan segitiga *Sierpinski* secara keseluruhan (Stephen T. Welstead, 1999, p34).

UNIVERSITAS MULTIMEDIA NUSANTARA



Gambar 2.2 Segitiga *Sierpinski*
Sumber: http://en.wikipedia.org/wiki/Sierpinski_triangle, 24 Juni 2011

Michael Barnsley (*V-variable fractals and superfractals,* p3) menjelaskan bahwa proses ini tidak bergantung pada bentuk permulaan dari segitiga tersebut, penggunaan bentuk segitiga hanya untuk mempermudah. Barnsley menggunakan citra awal ikan untuk mengilustrasikan hal ini.



Gambar 2.3 Segitiga *Sierpinski* dari citra awal ikan
Sumber: http://maths.anu.edu.au/~barnsley/pdfs/V-var_super_fractals.pdf, 24 Juni 2011

2.6.2 Kompresi Fraktal

Menurut Fisher (1994, p2) kompresi fraktal dapat digambarkan sebagai mesin fotokopi spesial yang mereduksi citra yang disalin menjadi setengah ukuran asli dan mereproduksi citra tersebut tiga kali pada hasilnya. Jika citra hasil dari

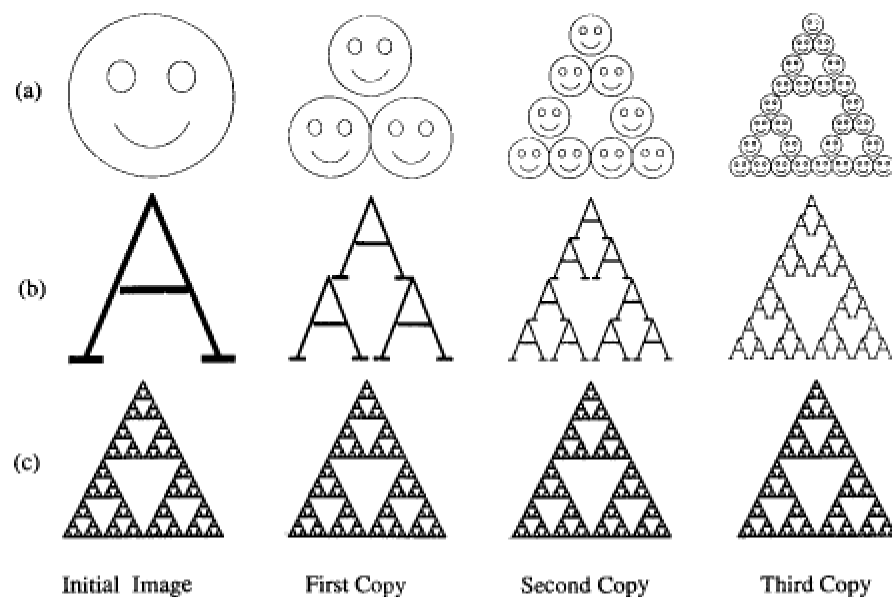
UNIVERSITAS MULTIMEDIA NUSANTARA

mesin ini dimasukkan kembali ke mesin sebagai masukan dan proses ini diulang berkali-kali, maka hasil dari salinan akan menyatu ke citra akhir yang sama, seperti pada Gambar 2.4. Karena citra akhir ini dibentuk dari tiga salinan tereduksi dari dirinya sendiri, maka citra ini mempunyai detail pada setiap skala, hal ini disebut sebagai fraktal. Citra ini akan disebut sebagai *attractor* dari mesin fotokopi ini. Karena mesin ini mereduksi citra masukan, hasil salin dari citra awal akan tereduksi ke suatu titik. Karena hasil salinannya kembali dijadikan sebagai citra masukan, akan ada lebih dan lebih salinan, tetapi setiap salinan akan berukuran semakin kecil. Oleh karena itu, citra awal tidak mempengaruhi hasil citra akhir, hanya posisi dan orientasi dari salinan yang menentukan hasil akhirnya.

Karena hasil akhir dari proses mesin fotokopi yang berulang-ulang tersebut hanya ditentukan dari cara citra masukan ditransformasi, hanya transformasi tersebut yang perlu disimpan. Tranformasi yang berbeda akan menghasilkan *attractor* yang berbeda, dengan keterbatasan teknik bahwa transformasi tersebut harus bersifat kontraktif, yaitu transformasi dari dua titik pada citra masukan akan mendekatkan kedua titik tersebut pada citra keluaran. Keterbatasan teknis ini sangat natural karena jika kedua titik tersebut malah menyebar, hasil dari *attractor* akan berukuran tak hingga. Mengecualikan kondisi tersebut, transformasi dapat berbentuk apa saja. Pada kenyataan, memilih transformasi dalam bentuk

$$w_i \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}$$

sudahlah cukup untuk membuat hasil *attractor* yang bervariasi. Transformasi tersebut disebut transformasi *affine*, dimana masing-masing dapat mencondongkan, menarik, merotasi, mengalikan dengan skala, dan mentranslasikan citra masukan (Fisher, 1994, p2).



Gambar 2.4 Tiga hasil salinan pertama
Sumber: Fractal Image Compression: Theory and Application

Menurut Ning Lu (1997, p33) transformasi *affine* berikut cukup untuk digunakan dalam kompresi fraktal:

Tabel 2.1 Transformasi *affine*

Matriks	Deskripsi
$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	Identitas
$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	Pencerminan terhadap sumbu y
$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	Pencerminan terhadap sumbu x
$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	Pencerminan terhadap garis y = x

$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$	Pencerminan terhadap garis y = -x
$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	Rotasi 90°
$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	Rotasi 180°
$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$	Rotasi 270°

M. Barnsley di dalam buku yang ditulis oleh Yuval Fisher (1994, p3) memberikan ide bahwa mungkin menyimpan citra sebagai sekumpulan transformasi (mesin fotokopi) akan menghasilkan metode kompresi citra. Pakis pada Gambar 2.5 mungkin terlihat kompleks, tetapi citra tersebut hanya dibuat dari empat transformasi affine. Setiap transformasi w_i didefinisikan sebagai enam angka, a_i, b_i, c_i, d_i, e_i dan f_i, tidak membutuhkan banyak memori untuk disimpan dalam komputer (mereka dapat disimpan dalam 4 transformasi x 6 angka per transformasi x 32 bit per angkat = 768 bit). Menyimpan citra pakis tersebut sebagai sekumpulan pixel ternyata membutuhkan memori jauh lebih banyak (setidaknya 65.536 bit untuk resolusi pada Gambar 2.5). Maka proses penyimpanan citra pakis tersebut, hanya transformasi-transformasi *affine* saja yang perlu disimpan dan citra pakis dapat dibuat dari transformasi-transformasi tersebut.

UNIVERSITAS MULTIMEDIA NUSANTARA

Gambar 2.5 *Barnsley's Fern*
Sumber: http://www.home.aone.net.au/~byzantium/ferns/fractal.html, 25 Juni 2011

2.6.3 *Iterated Function System*

Mesin fotokopi pada sub bab 2.6 adalah metafora untuk model matematika yang disebut sebagai *iterated function system* (IFS). Fisher (1994, p6) menjelaskan bahwa sebuah iterated function system terdiri dari sekumpulan transformasi kontraktif $\{w_i\colon \mathbb{R}^2 \rightarrow \mathbb{R}^2 | i = 1, \ldots, n\}$ yang memetakan bidang $\mathbb{R}^2$ ke dirinya sendiri. Sekumpulan transformasi ini mendefinisikan sebuah peta

$$W(\cdot) = \bigcup_{i=1}^{n} w_i(\cdot)$$

Peta W tidak diaplikasikan ke bidang, melainkan diaplikasikan ke sekumpulan titik di dalam bidang. Diberikan sebuah himpunan masukan S, $w_i(S)$ dapat dihitung untuk setiap i (membuat salinan tereduksi dari citra), mengambil himpunan gabungan dari himpunan tersebut (menyatukan salinan tereduksi), dan

UNIVERSITAS MULTIMEDIA NUSANTARA

mendapatkan himpunan baru W(S) (keluaran dari IFS). Maka W adalah sebuah peta dari ruang himpunan bagian dari bidang tersebut (Fisher, 1994, p6).

Menurut Fisher (1994, p7), terdapat dua buah fakta penting:

- Jika w_i kontraktif pada bidang, maka W bersifat kontraktif di ruang himpunan bagian dari bidang tersebut.
- Jika didapatkan sebuah peta W dari ruang citra, maka terdapat citra spesial yang disebut *attractor* dan dinotasikan dengan x_W, dengan sifat-sifat:
  1. Jika IFS diaplikasikan pada *attractor*, keluarannya akan sama dengan masukan, citra tersebut bersifat tetap, dan *attractor* x_W disebut sebagai titik tetap dari W:

     $$W(x_w) = x_w = w_1(x_w) \cup w_2(x_w) \cup \ldots \cup w_n(x_w)$$

  2. Diberikan sebuah citra masukan S_0, IFS dapat dijalankan sekali untuk mendapatkan $S_1 = W(S_0)$, dua kali untuk mendapatkan $S_2 = W(S_1) = W(W(S_0)) \equiv W^{on}(S_0)$, dan seterusnya. *Superscript* "o" menandakan iterasi, bukan exponen: W^{o2} adalah keluaran dari iterasi kedua. Hasil dari keluaran IFS dijadikan sebagai masukan kembali dan dilakukan berulang-ulang (*attractor*), adalah

     $$x_w \equiv S_\infty = \lim_{n \to \infty} W^{on}(S_0)$$

  3. x_W adalah unik. Jika didapatkan himpunan S apapun dan sebuah transformasi citra W yang memenuhi W(S) = S, maka S adalah *attractor* dari W, dimana S = x_W. Hal ini berarti hanya satu himpunan yang memenuhi persamaan titik tetap di sifat pada poin 1 di atas.

2.6.4 Metrik Pada Citra

Menurut Fisher (1994, p8), metrik adalah sebuah fungsi yang mengukur jarak di antara dua benda. Benda yang dimaksud misalnya dua titik pada sebuah garis, dan metrik untuk hal tersebut dapat dianggap nilai absolut dari perbedaan titik-titik tersebut. Alasan mengapa dipilih kata "metrik" daripada kata "perbedaan" atau "jarak" adalah karena konsep dari metrik lebih luas dan umum. Terdapat metrik yang mengukur jarak di antara dua citra, jarak di antara dua titik, atau jarak di antara dua himpunan, dan sebagainya.

Menurut Fisher (1994, p8), terdapat banyak metrik yang dapat dipilih, tetapi salah satu yang cukup sederhana untuk digunakan adalah metrik *supremum*

$$d_{sup}(f,g) = \underset{(x,y) \in I^2}{sup} |f(x,y) - g(x,y)|$$

dan metrik rms (*root mean square*)

$$d_{rms}(f,g) = \sqrt{\int_{I^2} (f(x,y) - g(x,y))dxdy}$$

Metrik supremum mencari posisi (x, y) dimana dua citra f dan g memiliki perbedaan yang paling besar dan menggunakan nilai ini sebagai jarak antara f dan g (Fisher, 1994, p9).

2.6.5 *Self-Similarity* Pada Citra

Fisher (1994, p9) menjelaskan bahwa misalnya pada citra wajah, tidak memiliki *self-similarity* seperti yang ditemukan pada segitiga *Sierpinski*. Citra

wajah tidak memiliki transformasi *affine* dari dirinya sendiri, tetapi citra wajah memiliki sejenis *self-similarity* yang berbeda. Gambar 2.6 menunjukkan contoh bagian dari citra Lenna yang mirip pada skala yang berbeda: bagian dari bahunya tumpang tindih dengan bagian kecil yang hampir identik, dan bagian pada pantulan topi di cermin yang mirip dengan bagian kecil dari topinya setelah ditransformasi. Perbedaan dengan segitiga *Sierpinski* adalah bahwa segitiga *Sierpinski* terbentuk dari salinan dirinya secara keseluruhan dengan transformasi yang sesuai, sedangkan gambar Lenna di bawah akan dibentuk dari bagian dari dirinya sendiri dengan transformasi yang sesuai. Bagian yang sudah ditransformasi tersebut tidak persis sama dengan citra aslinya, maka akan terdapat kesalahan di representasi citra sebagai himpunan dari *self-transformation*. Hal ini berarti bahwa citra dapat disandikan sebagai sekumpulan transformasi yang bukan merupakan salinan identik, melainkan aproksimasi.

Gambar 2.6 Bagian yang *self-similar* pada citra Lenna

Hasil eksperimen menunjukkan bahwa hampir seluruh citra yang terbentuk secara natural dapat dikompresi dengan memanfaatkan sifat *self-similarity* ini. Hal

UNIVERSITAS MULTIMEDIA NUSANTARA

ini adalah redudansi yang dicoba untuk dieliminasi dengan kompresi fraktal (Fisher, 1994, p10).

2.6.6 *Partitioned Iterated Function System*

Mesin fotokopi pada sub bab 2.6 dapat diubah menjadi mesin yang dapat menyalin hanya sebagian dari citra dan dapat mengubah faktor *brightness* dan kontras. Mesin ini adalah metafora dari partitioned iterated function system (PIFS). Menurut Fisher (1994, p11) PIFS tidak bergantung kepada jenis dari transformasi, tetapi pada penelitian ini yang akan digunakan adalah transformasi *affine*. Terdapat dua dimensi ruang dan tingkat dari keabuan menambahkan dimensi ketiga, maka transformasi w_i menjadi bentuk

$$w_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix}$$

dimana s_i mengontrol kontras dan o_i mengontrol brightness dari transformasi. Untuk mempermudah, didefinisikan bagian ruang v_i dari transformasi di atas sebagai

$$v_i(x, y) = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}$$

Apabila citra dibuat menjadi model matematika sebagai fungsi f(x,y) dimana f(x,y) adalah intensitas keabuan dari citra pada koordinat (x,y), kita dapat mengaplikasikan w_i ke citra f dengan $w_i(f) \equiv w_i(x, y, f(x, y))$. Lalu v_i akan menentukan bagaimana partisi domain dari citra asli dipetakan ke salinan, dimana s_i dan o_i menentukan transformasi *brightness* dan kontras. Secara implisit D_i dan

R_i berada di atas bidang, dan perlu diingat bahwa setiap wi terbatas pada D_i x I, ruang vertikal di atas D_i. Hal ini berarti bahwa $v_i(D_i) = R_i$ (Fisher, 1994, p11).

Karena W(f) harus menghasilkan citra, maka $\cup R_i = I^2$ dan $R_i \cap R_j = \emptyset$ pada saat $i \neq j$. Hal ini berarti pada saat kita mengaplikasikan W ke sebuah citra, kita mendapatkan fungsi bernilai tunggal di atas titik pada kotak I^2. Pada metafora mesin fotokopi di atas, hal ini berarti bahwa salinan dari seluruh halaman, dan mereka dapat bersebelahan tetapi tidak tumpang tindih (Fisher, 1994, p11).

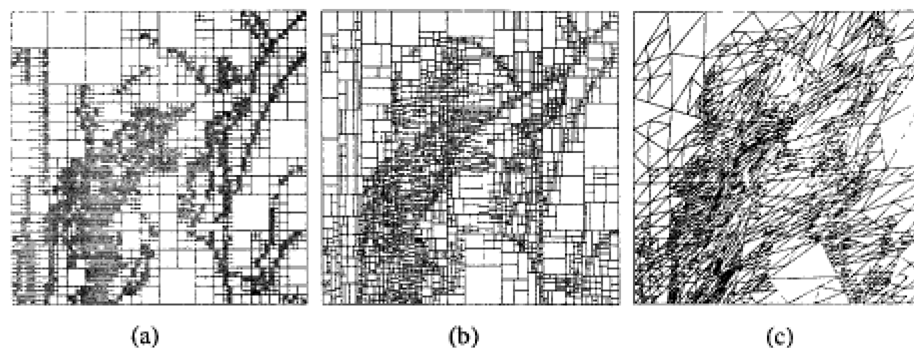
Menjalankan mesin fotokopi di dalam perulangan berarti mengiterasikan peta W. Kita memulai dengan citra asli f_0 dan melakukan iterasi $f_1 = w(f_0), f_2 = W(f_1) = (W(W(f_0))$, dan seterusnya. Iterasi ke-n akan dinotasikan sebagai $f_n = W^{on}(f_0)$ (Fisher, 1994, p11).

2.6.7 Partisi Citra

Fisher (1994, p13) menjelaskan salah satu partisi citra yang paling mudah adalah dengan menggunakan partisi berukuran tetap. Misalnya adalah untuk citra berukuran 256 x 256 pixel, maka dibentuk R_1, R_2, …, R_{1024} sebagai 8 x 8 pixel bagian dari citra yang tidak saling tumpang tindih, dan D sebagai 16 x 16 pixel bagian dari citra yang saling tumpang tindih. D berisi 241 x 241 = 58.081 kotak. Untuk setiap R_i, dicari dari seluruh D untuk menemukan $D_i \in D$ yang memiliki metrik terkecil, yaitu menemukan bagian dari citra yang paling mirip dengan R_i.

Menurut Fisher (1994, p16), setelah menemukan R_i dan D_i, matriks transformasi pada sub bab 2.10 yang berisi s_i, o_i, a_i, b_i, c_i, d_i, e_i dan f_i juga dapat

ditentukan. Maka transformasi $W = \cup w_i$ didapatkan dan dapat menyandikan aproksimasi dari citra awal. Selain partisi berukuran tetap, terdapat beberapa partisi lain misalnya:



Gambar 2.7 (a) Partisi quadtree (5008 kotak), (b) partisi HV (2910 kotak), dan (c) partisi segitiga (2954 kotak)
Sumber: Fractal Image Compression: Theory and Application

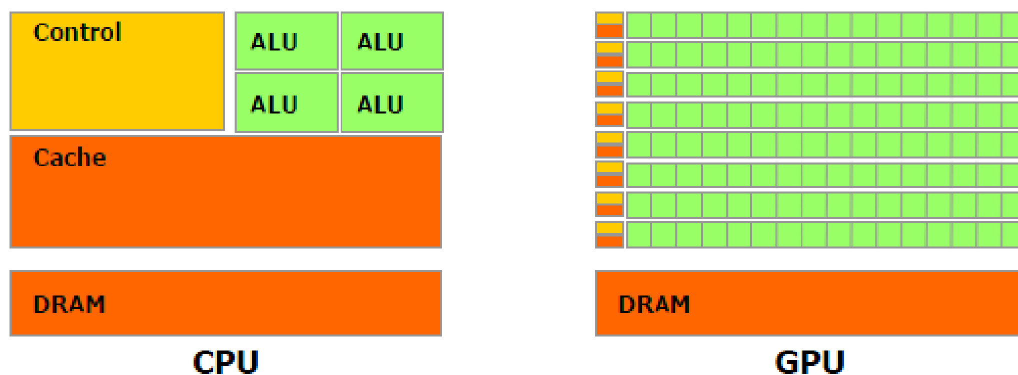
2.7 Komputasi Paralel

2.7.1 Pemrosesan Grafis Untuk Komputasi Paralel Umum

Menurut CUDA C Programming Guide (2011, p1), didorong oleh permintaan pasar untuk grafik 3D yang *realtime* dan *high-definition*, *Graphic Processor Unit* (GPU) yang dapat diprogram telah berevolusi menjadi prosesor yang sangat paralel, *multithreaded*, dan *manycore* dengan kemampuan komputasi yang sangat kuat dan *bandwith* memori yang sangat tinggi.

Jason Sanders (2010, p. xiii) mengatakan "*GPUs from NVIDIA have outpaced standard CPUs in floating-point performance in recent years. Furthermore, they have arguably become as easy, if not easier, to program than multicore CPUs*".

Alasan dibalik perbedaan kemampuan memproses *floating-point* di antara CPU dan GPU adalah bahwa GPU telah dispesialisasikan untuk komputasi yang *compute-intensive*, dan sangat paralel (sangat diperlukan untuk *rendering* grafis). Oleh karena itu GPU didesain dengan transistor yang lebih banyak untuk pemrosesan data daripada untuk kontrol aliran dan *caching* data (CUDA C Programming Guide, 2011, p3).



Gambar 2.8 GPU memiliki lebih banyak transistor untuk pemrosesan data
Sumber: CUDA C Programming Guide

Secara lebih spesifik, GPU dibuat secara khusus untuk menangani masalah yang dapat dikatakan sebagai komputasi data paralel dengan intensitas aritmatika yang tinggi dibanding operasi memori. Karena program yang sama dieksekusi untuk setiap elemen data, terdapat kebutuhan yang lebih rendah untuk kontrol aliran yang rumit dan karena dieksekusi untuk elemen data yang banyak dan intensitas aritmatika yang tinggi, *latency* akses memori dapat disebunyikan dengan perhitungan daripada *cache* data yang besar (CUDA C Programming Guide, 2011, p3).

Pemrosesan data paralel memetakan elemen data ke *thread* pemroses paralel. Banyak aplikasi yang memproses sekumpulan data berukuran besar dapat menggunakan model programming data paralel untuk mempercepat komputasi. Pada 3D *rendering*, sekumpulan besar *pixel* dan *vertice* dipetakan ke *thread* paralel. Dengan cara yang sama, aplikasi untuk pemrosesan citra dan media seperti *post-processing* dari citra yang telah dirender, *encoding* dan *decoding* video, *scaling* citra, *stereo vision*, dan pengenalan pola dapat memetakan blok citra dan *pixel* ke thread pemroses paralel. Faktanya, banyak algoritma diluar bidang *rendering* dan pemrosesan citra yang dapat dipercepat dengan pemrosesan data paralel, mulai dari pemrosesan *signal* secara umum atau simulasi fisika sampai dengan komputasi keuangan atau komputasi biologi (CUDA C Programming Guide, 2011, p3).

2.7.2 CUDA

Pada November 2006, NVIDIA memperkenalkan CUDA, yaitu sebuah arsitektur komputasi paralel untuk tujuan umum dengan arsitektur model pemrograman paralel baru dan kumpulan instruksi yang baru. Hal ini memungkinkan mesin komputasi paralel di NVIDIA GPU untuk menyelesaikan banyak masalah komputasi kompleks dengan cara yang lebih efisien daripada menggunakan CPU (CUDA C Programming Guide, 2011, p3).

CUDA diluncurkan dengan lingkungan piranti lunak yang memungkinkan pengembang untuk menggunakan C sebagai bahasa pemrograman tingkat tinggi. Bahasa pemrograman atau API yang lain juga didukung, seperti CUDA

UNIVERSITAS MULTIMEDIA NUSANTARA

FORTRAN, OpenCL, dan DirecCompute (CUDA C Programming Guide, 2011, p4).

2.7.3 Model Pemrograman *Scalable*

Menurut CUDA C Programming Guide (2011, p4), munculnya CPU *multicore* dan GPU *manycore* berarti bahwa chip prosesor yang utama sekarang adalah sistem paralel. Lebih lanjut lagi, paralelisme ini berlanjut berkembang sesuai dengan hukum Moore. Tantangannya adalah untuk mengembangkan piranti lunak yang secara transparan menyesuaikan dengan paralelisme untuk memanfaatkan jumlah *core* prosesor yang terus bertambah, seperti piranti lunak grafik 3D yang secara transparan menyesuaikan paralelisme mereka dengan GPU *manycore* yang memiliki *core* dengan jumlah beragam.

Model pemrograman paralel CUDA didesain untuk menanggulangi tantangan ini sambil tetap mempertahankan kurva pembelajaran yang rendah untuk programmer yang sudah familiar dengan bahasa pemrograman standar seperti C (CUDA C Programming Guide, 2011, p4).

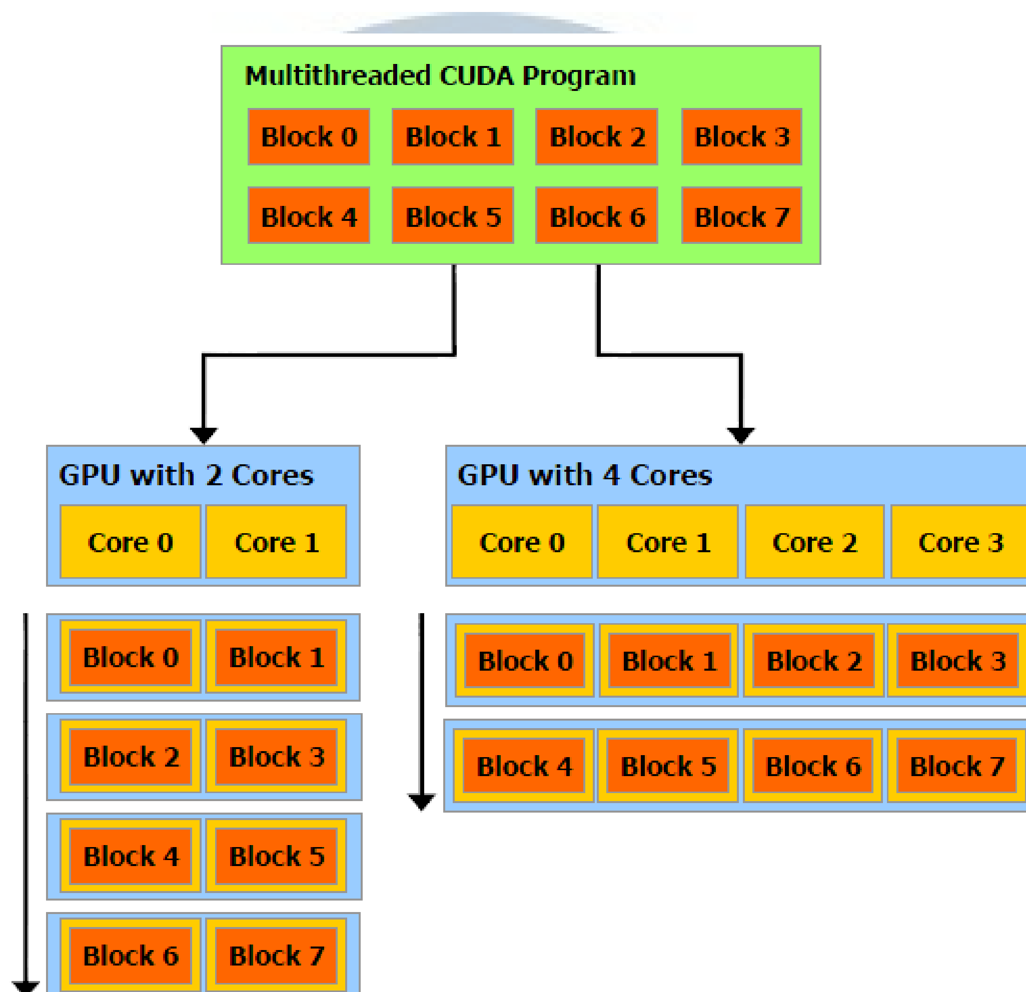
Pada intinya terdapat tiga kunci abstraksi yaitu: hierarki dari grup *thread*, *shared memory*, dan sinkronisasi pembatas. Tiga hal ini diekspos kepada programmer sebagai sekumpulan perluasan dari bahasa (CUDA C Programming Guide, 2011, p4).

Abstraksi ini menyediakan paralelisme data dan *thread* detail yang bersarang pada data dan *thread* paralelisme yang lebih luas. Abstraksi ini menuntun programmer untuk membagi masalah menjadi sub masalah yang lebih

Implementasi kompresi citra..., Ambrosius Alvin Gunawan, FTI UMN, 2012

luas yang dapat diselesaikan secara independen di blok paralel yang tersusun dari *thread*, dan setiap sub problem menjadi kepingan yang lebih detail yang dapat diselesaikan secara lebih kooperatif di *thread* paralel di dalam blok (CUDA C Programming Guide, 2011, p4).

Dekomposisi ini mempertahankan kemampuan bahasa untuk berekspresi dengan memungkinkan *thread* untuk berkooperasi satu sama lain pada saat menyelesaikan sub masalah, dan pada saat yang bersamaan memungkinkan skalabilitas yang otomatis. Setiap blok *thread* dapat dijadwalkan untuk berjalan pada *core* prosesor yang ada, dalam urutan apapun, secara bersamaan maupun sekuensial, sehingga program CUDA dapat berjalan pada berapapun jumlah *core* prosesor. Hanya sistem *runtime* yang membutuhkan jumlah fisik dari prosesor (CUDA C Programming Guide, 2011, p5).



Gambar 2.9 Skalabilitas secara otomatis
Sumber: CUDA C Programming Guide

2.7.4 *Kernel*

Menurut CUDA C Programming Guide (2011, p7) , CUDA C memperluas bahasa C dengan memungkinkan programmer untuk mendefinisikan fungsi C yang disebut *kernel*, yang pada saat dieksekusi, akan dieksekusi sebanyak N kali secara paralel oleh N *thread* CUDA yang berbeda, dimana pada fungsi C biasa hanya dieksekusi sekali.

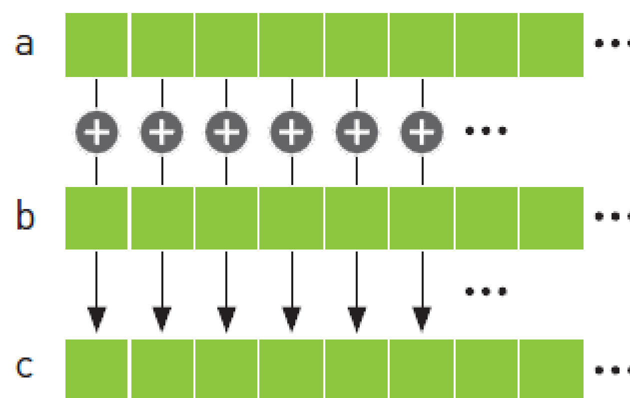
Sebuah *kernel* didefinisikan menggunakan kata kunci __global__ dan jumlah *thread* CUDA yang akan menjalankan *kernel* tersebut untuk sebuah

pemanggilan *kernel* ditentukan menggunakan sintaks konfigurasi eksekusi <<<…>>>. Setiap thread yang menjalankan kernel diberikan sebuah ID pengenal unik *thread* yang dapat diakses dari dalam kernel melalui variable *built-in* **threadIdx** (CUDA C Programming Guide, 2011, p7).

Sebagai ilustrasi, contoh kode berikut ini menambahkan dua buah vektor A dan B dengan ukuran N dan menyimpan hasilnya di dalam vektor C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
	int i = threadIdx.x;
	C[i] = A[i] + B[i];
}

int main()
{
	...
	// Kernel invocation with N threads
	VecAdd<<<1, N>>>(A, B, C);
}
```

Masing-masing dari N *thread* akan mengeksekusi VecAdd() yang menghitung sepasang operasi penjumlahan.



Gambar 2.10 Menjumlahkan dua buah vektor
Sumber: CUDA by Example

2.7.5 Hierarki *Thread*

Menurut CUDA C Programming Guide (2011, p8) , untuk mempermudah pengembang dibuatlah **threadIdx,** yaitu vektor dengan tiga komponen sehingga thread dapat diidentifikasi dengan menggunakan indeks *thread* satu dimensi, dua dimensi, atau tiga dimensi, membentuk blok *thread* satu dimensi, dua dimensi, atau tiga dimensi. Hal ini menyediakan cara yang natural untuk membangkitkan perhitungan ke elemen-elemen di suatu domain seperti vektor, matriks, atau volume.

Indeks dari suatu *thread* dan ID *thread* berelasi satu sama lain. Untuk blok satu dimensi, mereka sama. Untuk blok dua dimensi dengan ukuran (D_x, D_y), ID *thread* dengan indeks (x, y) adalah (x + y D_x). Untuk blok tiga dimensi dengan ukuran (D_x, D_y, D_z), ID *thread* dengan indeks (x, y, z) adalah (x + y D_x + z D_x D_y) (CUDA C Programming Guide, 2011, p8).

Sebagai contoh, kode berikut menambahkan elemen-elemen dari matriks A dan B dengan ukuran N x N dan menyimpan hasilnya di dalam matriks C:

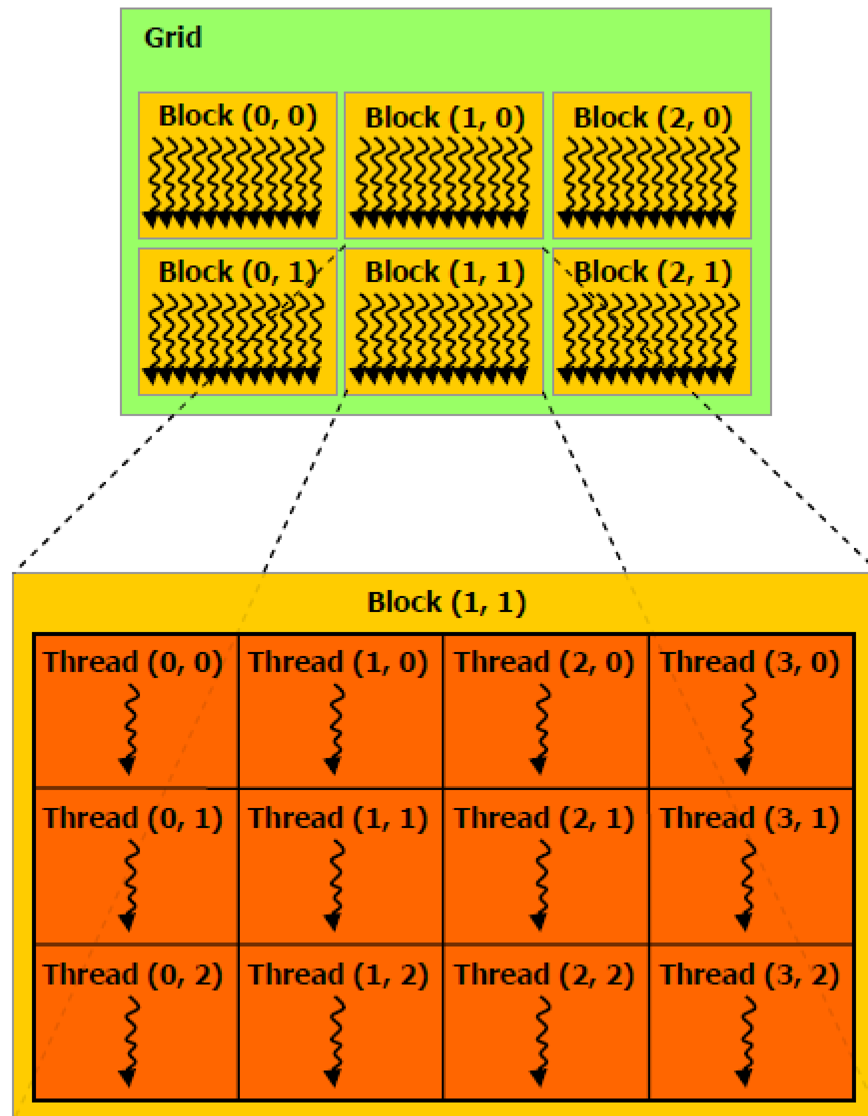
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
      int i = threadIdx.x;
      int j = threadIdx.y;
      C[i][j] = A[i][j] + B[i][j];
}

int main()
{
      ...
      // Kernel invocation with one block of N * N * 1 threads
      int numBlocks = 1;
      dim3 threadsPerBlock(N, N);
      MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Terdapat batasan teknis mengenai jumlah *thread* per blok, karena setiap *thread* di dalam suatu blok berada di dalam *core* prosesor yang sama dan harus berbagi sumber daya memori yang terbatas di dalam *core* tersebut. Pada kebanyakan GPU yang ada sekarang, suatu blok *thread* memungkinkan sampai dengan 1024 *thread* (CUDA C Programming Guide, 2011, p8).

Walaupun demikian, sebuah *kernel* dapat dieksekusi oleh banyak blok *thread* dengan ukuran yang sama, sehingga jumlah total *thread* sama dengan jumlah *thread* per blok dikalikan dengan jumlah blok (CUDA C Programming Guide, 2011, p8).

Blok dikelompokkan ke sebuah *grid* dari *thread* blok dengan satu dimensi, dua dimensi, atau tiga dimensi. Banyaknya blok *thread* di dalam suatu *grid* biasanya ditentukan dari ukuran data yang akan diproses atau jumlah prosesor di dalam sistem (CUDA C Programming Guide, 2011, p8).

UNIVERSITAS
MULTIMEDIA
NUSANTARA



Gambar 2.11 *Grid* dari sebuah blok *thread*
Sumber: CUDA C Programming Guide

Banyaknya *thread* per blok dan jumlah blok per *grid* ditentukan di dalam sintaks <<<…>>> dan dapat bertipe data int atau dim3. Blok atau *grid* dua dimensi dapat dispesifikasikan seperti contoh kode di atas (CUDA C Programming Guide, 2011, p9).

UNIVERSITAS MULTIMEDIA NUSANTARA

Setiap blok di dalam *grid* dapat diidentifikasi dengan indeks satu dimensi, dua dimensi, atau tiga dimensi yang dapat diakses dari dalam *kernel* melalui variabel *built-in* **blockIdx**. Dimensi dari blok *thread* dapat diakses dari dalam *kernel* melalui variabel *built-in* **blockDim** (CUDA C Programming Guide, 2011, p9).

Dengan mengembangkan contoh MatAdd() di atas untuk menangani banyak blok, kodenya menjadi seperti berikut.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	int j = blockIdx.y * blockDim.y + threadIdx.y;
	if (i < N && j < N)
		C[i][j] = A[i][j] + B[i][j];
}

int main()
{
	...
	// Kernel invocation
	dim3 threadsPerBlock(16, 16);
	dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
	MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

Sebuah blok thread dengan ukuran 16 x 16 (256 thread), walaupun angka tersebut seperti diambil secara acak, adalah salah satu pilihan yang umum. Grid dibuat dengan jumlah blok yang cukup untuk memiliki satu thread per elemen matriks seperti contoh di atas. Untuk alasan penyederhanaan, contoh di atas mengasumsikan bahwa jumlah *thread* per *grid* di setiap dimensi dapat dibagi dengan bulat oleh jumlah *thread* per blok di dimensi tersebut, walaupun kenyataannya tidak selalu seperti itu (CUDA C Programming Guide, 2011, p10).

Blok *thread* dibuat untuk dieksekusi secara independen satu sama lain. Blok *thread* harus dibuat dengan memikirkan kemungkinan bahwa mereka dieksekusi dengan urutan acak, secara paralel, atau secara berurutan. Hal ini memungkinkan blok *thread* untuk dijadwalkan dalam urutan apapun di dalam jumlah *core* berapapun, sehingga memungkinkan programmer untuk membuat kode yang menyesuaikan dengan jumlah *core* (CUDA C Programming Guide, 2011, p10).

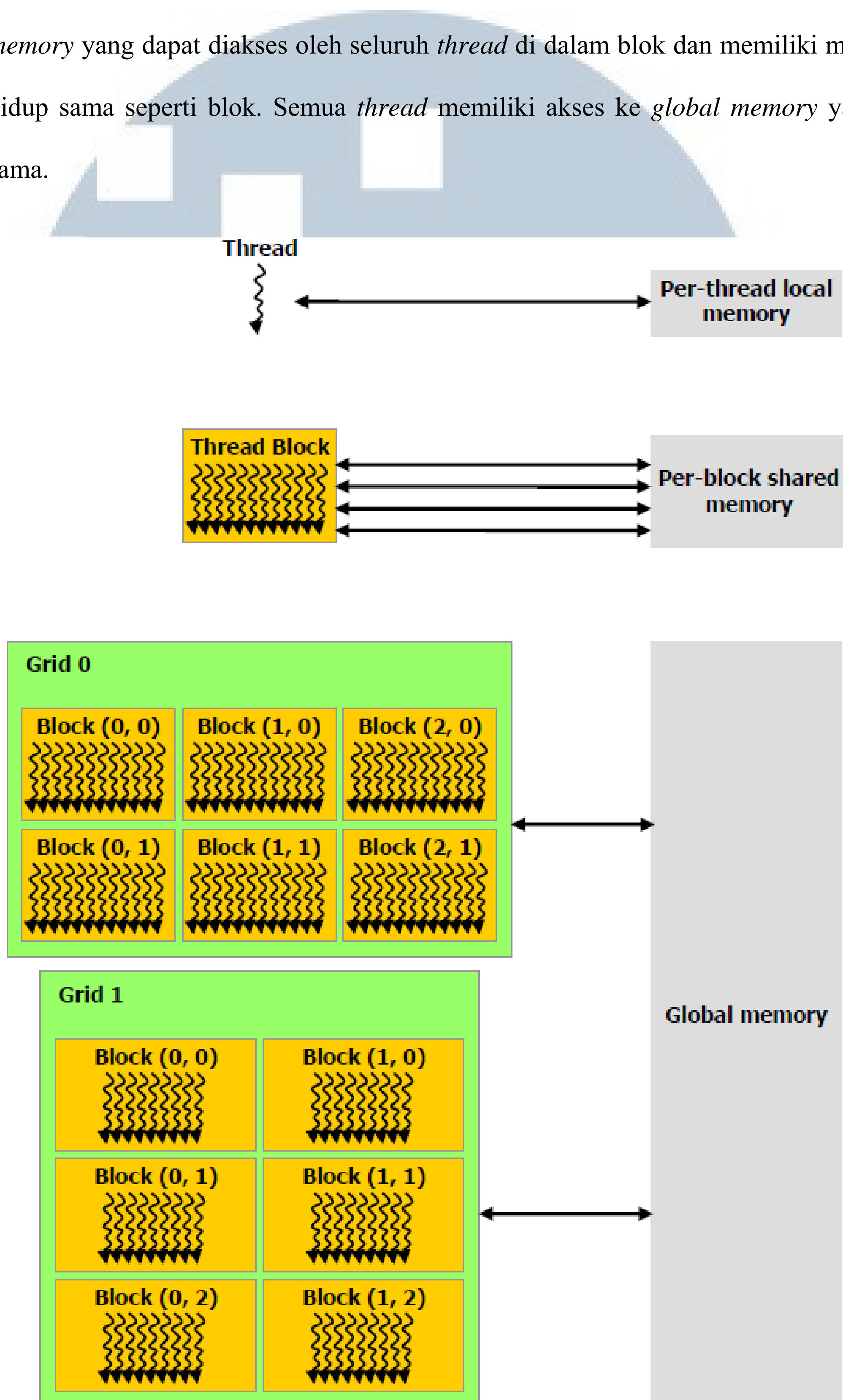
Thread di dalam blok dapat saling berkooperasi dengan berbagi data satu sama lain melalui *shared memory* dan dengan melakukan sinkronisasi eksekusi mereka untuk mengkoordinasikan akses memori. Programmer dapat menentukan titik sinkronisasi di dalam *kernel* dengan memanggil fungsi __syncthreads(), fungsi ini berlaku sebagai tameng dimana seluruh *thread* di dalam blok harus menunggu sebelum bisa melanjutkan eksekusi mereka (CUDA C Programming Guide, 2011, p10).

Untuk kooperasi yang efisien, shared memory dibuat sebagai memori dengan *low-latency* di setiap *core* prosesor (seperti *L1 cache*) dan __syncthreads() dibuat sebagai fungsi yang ringan dan tidak memakan banyak sumber daya (CUDA C Programming Guide, 2011, p10).

2.7.6 Hierarki Memori

Menurut CUDA C Programming Guide (2011, p10), *thread* CUDA dapat mengakses data dari berbagai ruang memori pada saat eksekusi. Setiap *thread* memiliki memori lokal pribadi. Setiap *thread* di dalam blok memiliki *shared*

memory yang dapat diakses oleh seluruh *thread* di dalam blok dan memiliki masa hidup sama seperti blok. Semua *thread* memiliki akses ke *global memory* yang sama.



Gambar 2.12 Hierarki memori
Sumber: CUDA C Programming Guide

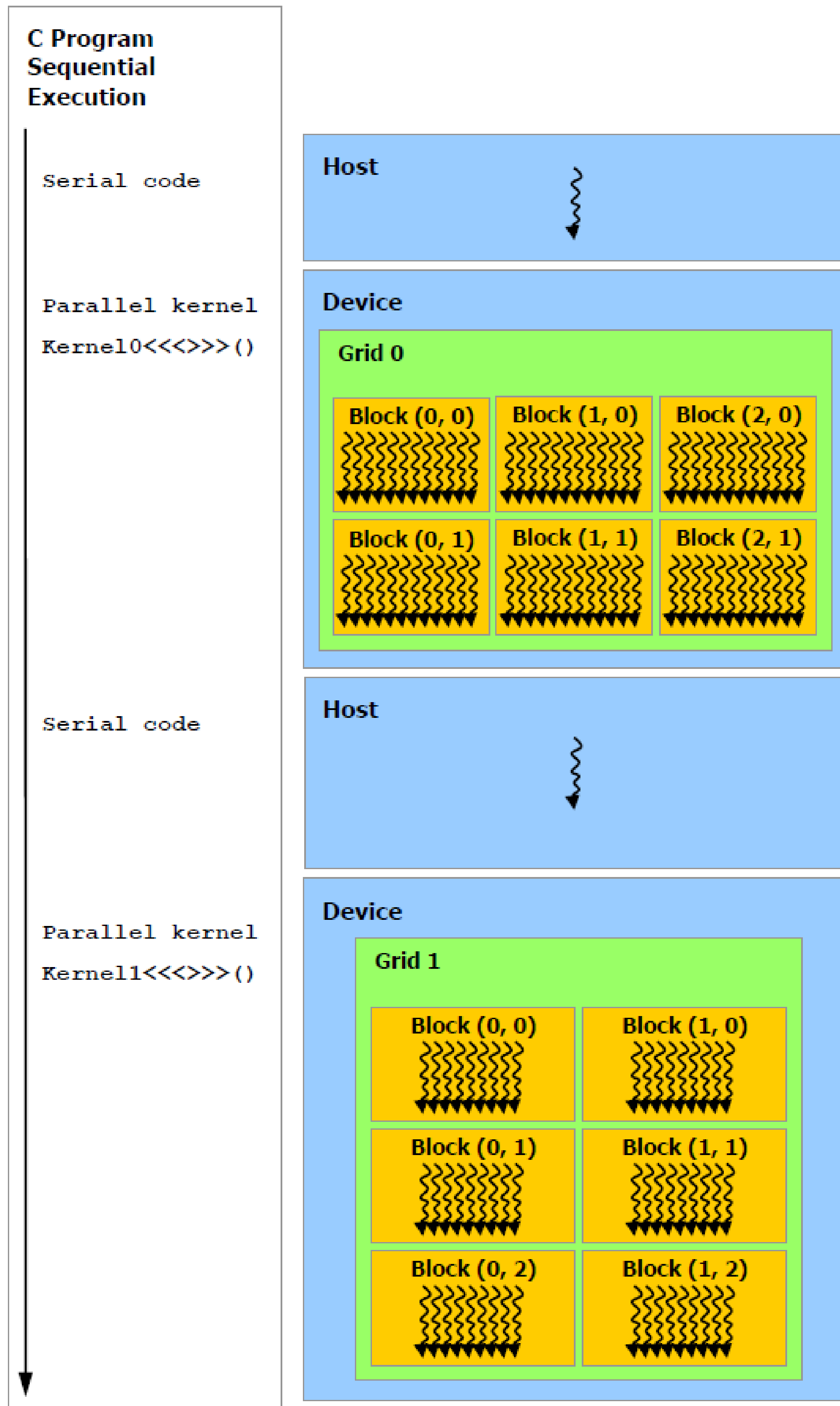
Selain itu, juga terdapat dua ruang memori *read-only* tambahan yang dapat diakses oleh seluruh *thread*: ruang memori konstan dan *texture*. Ruang memori *global*, konstan, dan *texture* dioptimisasi untuk penggunaan memori yang berbeda. Memori *texture* juga menyediakan mode pengalamatan yang berbeda dan penyaringan data, untuk format data tertentu (CUDA C Programming Guide, 2011, p10).

Ruang memori global, konstan, dan *texture* bersifat tetap selama eksekusi kernel di dalam aplikasi yang sama (CUDA C Programming Guide, 2011, p10).

2.7.7 Heterogeneous Programming

Menurut CUDA C Programming Guide (2011, p11), model pemrograman CUDA mengasumsikan bahwa *thread* CUDA berjalan pada perangkat yang terpisah secara fisik yang beroperasi sebagai *coprocessor* dari induk yang menjalankan program C. Sebagai contoh, pada saat *kernel* berjalan pada CPU, sisa dari program berjalan pada CPU.

Model pemrograman CUDA juga mengasumsikan bahwa induk dan perangkat mengatur sendiri ruang memori mereka di DRAM, yang dirujuk sebagai *host memory* dan *device memory*. Oleh karena itu, program mengatur ruang memori global, konstan, dan *texture* yang dapat digunakan oleh *kernel* melalui CUDA *runtime*, termasuk alokasi dan dealokasi *device memory* dan juga transfer data antara *host memory* dan *device memory* (CUDA C Programming Guide, 2011, p12).



Gambar 2.13 Heterogeneous programming
Sumber: CUDA C Programming Guide