PEMROGRAMAN DASAR DAN ANALISIS KINERJA APLIKASI DALAM KOMPUTASI MENGGUNAKAN GPU

Enrico Budianto, Hary Prabowo, Hafiz, M. Nanda Kurniawan, Prayoga Dahirsa, dan Tirmidzi Faisal

Fakultas Ilmu Komputer, Universitas Indonesia, Kampus Baru UI Depok, Jawa Barat, 16424, Indonesia

E-mail: enrico.budianto@ui.ac.id

Abstrak

GPU atau singkatan dari *Graphical Processing Unit* merupakan mikroprosesor khusus yang berfungsi memercepat proses *rendering* grafik 2 dimensi atau 3 dimensi. GPU telah digunakan di beberapa perangkat seperti sistem yang telah ditanam, telepon genggam, komputer, *workstation*, dan *game console*. Penggunaan GPU sangat membantu efisiensi penggunaan waktu dalam proses perhitungan. Struktur paralel yang dimilikinya membuat efektivitas GPU lebih baik dibandingkan *Control Processing Unit* (CPU). Saat ini penggunaan GPU tidak hanya di bidang ilmu komputer saja, kemampuan yang dimiliki GPU dalam proses perhitungan yang rumit dan berulang-ulang menyebabkan penggunaanya telah dimanfaatkan di berbagai bidang. Dalam bidang kedokteran, GPU dimanfaatkan dalam mendiagnosis sebuah penyakit, sementara pada bidang akuntansi GPU digunakan dalam perhitungan data yang sangat banyak. Pada penelitian ini akan dijelaskan mengenai efektivitas penggunaan GPU dalam menjalankan sebuah program dan aplikasi serta perbedaannya dengan penggunaan CPU biasa.

Kata Kunci: *CPU, graphical processing unit, kinerja aplikasi*

Abstract

GPU stands for Graphical Processing Unit is a specialized microprocessor that serves to accelerate the process of rendering two-dimensional charts or three dimensions. GPUs have been used in several devices such as in embedded systems, mobile phones, computers, workstations, and game console. GPU usage helps the efficiency use of time in the calculation process. Its parallel structure, makes the effectiveness of GPU better than the CPU. Today the use of GPU not only in the field of computer science, the capabilities of the GPU in the complex and repetitive calculations process causes it has been utilized in various fields. In the medical field, a GPU used in diagnosing the disease, while in the field of accounting GPU used in the calculation of very much data. In this research, researcher will explain the effectiveness of using GPU in running a program and applications as well as the differences with ordinary CPU usage.

Keywords: *application performance, CPU, graphical processing unit*

1. Pendahuluan

Pemrograman berbasis GPU membantu *programmer* dalam menyelesaikan permasalahan komputasional yang rumit dibandingkan dengan cara konvensional menggunakan CPU. Salah satu pemrograman berbasis GPU adalah pemrograman yang berbasis arsitektur CUDATM yang dikembangkan oleh perusahaan NVIDIA. Arsitektur ini menggunakan bahasa pemrograman C. Pemrograman paralel yang berbasis CUDA terdiri dari tiga *core* atau inti, yaitu hirarki dari *thread*, *shared memory*, dan sinkronisasi. Pemanfaatan *threads* pada GPU dapat memercepat perhitungan atau komputasi dengan membagi-bagi tugas perhitungan dan menyebarkannya ke *thread-thread* untuk kemudian dilakukan komputasi oleh masing-masing thread tersebut. *Shared memory* di setiap blok berfungsi untuk memudahkan penggunaan *resource* oleh *threads* yang akan melakukan komputasi sehingga tidak perlu keluar blok untuk mencari *resource* utama yang berada di luar blok. Hal ini akan memercepat proses pengiriman dan penggunaan *resource* itu sendiri sehingga akhirnya akan mengurangi waktu komputasi. Sinkronisasi antar *thread* juga harus diimplementasikan pada proses komputasi, sebab perhitungan yang dilakukan oleh *threads* nantinya akan digabungkan ke dalam satu hasil perhitungan.

2. Metodologi

Pada bagian ini akan dijelaskan mengenai pemrograman umum berbasis GPU dan penggunaan fitur-fitur utama yang ada di GPU serta perilaku *output* dari program tersebut seperti yang terlihat pada gambar 1.

```
#include <cuda.h> //CUDA library
#include <stdio.h> //Standard input-output
#include <time.h> //Time library to
manipulate dates and times

//a kernel function that can be called from
host function to
//generate a grid of threads on a device
__global__ void vectorFill(float *E, int
size)
{

//index bertipe integer diinisialisasi
nilai dari thread yang //dijumlahkan dengan
ukuran block dan gridnya
        int index = blockIdx.y * blockDim.y
+ threadIdx.y;

//memasukkan nilai pada array dengan index
yang sudah //diinisialisasi sebelumnya
dikalikan ukuran vektor
        E[index] = (float)index * size;
}

//a CUDA host function
__host__ void vectorFill_naive(float *E,
int size)
{
        int i = 0;
        //looping biasa tanpa menggunakan
thread
        for(i = 0; i < size; i++)
        {
                E[i] = (float)i * size;
        }
}

//Fungsi untuk menjalankan program pada
host //maupun device
void run(int M) {
        long size = M * sizeof(float);
float* E = (float*) malloc(size);
        float* Ed;
        clock_t start, end; //tipe clock_t
dari time.h
        double diff;
        printf("%12d\t\t ", M);
//an API's function to allocate a piece of
global memory for //an object
        cudaMalloc((void**)&Ed, size);
        //transfer data dari memory host ke
device
        cudaMemcpy(Ed, E, size,
cudaMemcpyHostToDevice);
        //describing configuration blocks
        dim3 dimBlock(16, 16);
        //describes the configuration of
grid
        dim3 dimGrid(2,2);
        //at device
        start = clock();
        vectorFill<<<dimGrid,
dimBlock>>>(Ed, M);

//agar penghitungan dalam  thread selesai
semuanya
cudaThreadSynchronize();
cudaMemcpy(E, Ed, size,
cudaMemcpyDeviceToHost); //meng-copy
kembali nilai dari memory device ke host
//mengosongkan isi memory yang sudah
dialokasikan sebelumnya
cudaFree(Ed);
        end = clock();
        diff = ((double) end -
start)/(double)CLOCKS_PER_SEC;
        diff *= 1000;
        printf(" %10.7f \t\t ", diff);
        //at host
        float* G = (float*) malloc(size);
        start = clock();
        vectorFill_naive(G, M);
        free(G);
        end = clock();
        diff = ((double) end -
start)/(double)CLOCKS_PER_SEC;
        diff *= 1000;
        printf(" %10.7f \n", diff);
}
int main() //fungsi main dalam program ini
{
        int i = 0;
        int M = 0;
        int N = 0;
        printf("Masukkan maksimum vector
size: ");
        scanf("%i", &M);
        printf("Masukkan interval (dikali)
ukuran vector size: ");
        scanf("%i", &N);
printf("Vector Length \t Device function
time(ms) \t Host function time(ms)\n");
        for(i = 1; i < M; i *= N)
        {
                run(i);
        }
        printf("\n");
        return 0;
}
```

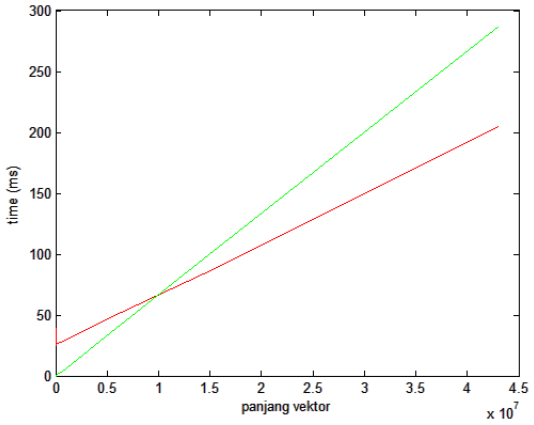
Gambar 1. Kode program berbasis GPU.

Proses untuk melihat hasil *output* dari program dilakukan dengan dua cara, yaitu dengan menggunakan *device emulator*, yang disediakan pada Cuda Toolkit SDK 3.0, serta menggunakan komputer yang mempunyai GPU dari NVIDIA, dengan tipe GeForce GTS 250. Perintah untuk melakukan kompilasi terhadap program di *device emulator* adalah nvcc –deviceemu –g [Nama_File].cu, sementara itu, perintah untuk melakukan kompilasi program di komputer yang mempunyai GPU adalah nvcc [Nama_File].cu –o [Nama_Exec_File].

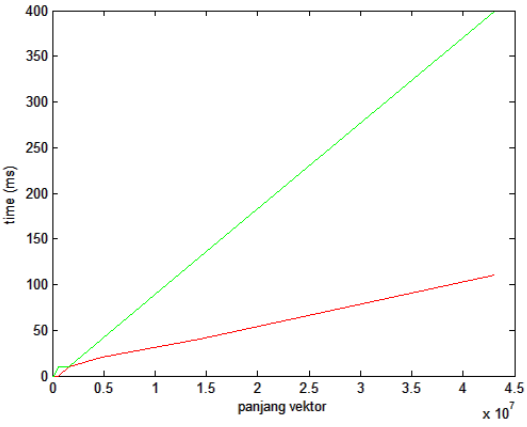
Gambar 2 merupakan tampilan grafik yang memberikan keterhubungan antara panjang vektor dengan waktu komputasi, di mana garis merah merupakan waktu komputasi pada saat dijalankan

pada *device* dan garis hijau merupakan waktu pada saat dijalankan pada *host*. Dari grafik di bawah ini terlihat bahwa titik balik dari waktu komputasi program terjadi pada saat panjang vektor mencapai 107. Setelah titik tersebut, terlihat bahwa pertumbuhan fungsi dari proses di CPU meningkat lebih cepat daripada proses yang dilakukan di GPU.

Program di atas dijalankan dalam sebuah komputer yang memiliki prosesor CPU, tanpa adanya bantuan dari GPU, sedangkan jika dilakukan langsung terhadap komputer yang mempunyai GPU yang berada di lantai 5 gedung B Fasilkom UI, terlihat hasil yang akan dijelaskan pada bagian bawah dari makalah ini.



Gambar 2. Grafik keterhubungan panjang vektor dan waktu.



Gambar 3. Grafik keterhubungan panjang vektor dan waktu pada GPU.

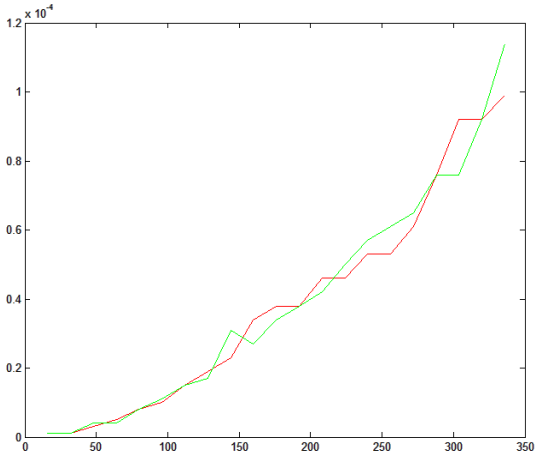
Program selanjutnya dijalankan pada komputer yang mempunyai GPU, dengan jenis GeForce GTS 250, dan berjalan di atas sistem operasi Ubuntu Linux. Gambar 3 merupakan tampilan grafik di mana garis merah merupakan waktu pada saat dijalankan pada *device* dan garis hijau merupakan waktu pada saat dijalankan di *host*.

Dari hasil ini dapat dibuktikan bahwa dengan menjalankan program pengisian vektor ini dapat dilihat bagaimana proses yang berjalan pada *device* (GPU) dapat berlangsung lebih cepat dibandingkan di *host* (CPU) biasa. Hal ini dapat diketahui dari perkembangan fungsi tersebut, di mana pertumbuhan dari fungsi *compute time* untuk *device* jauh lebih lambat daripada *compute time* dari program yang dijalankan di *host*.

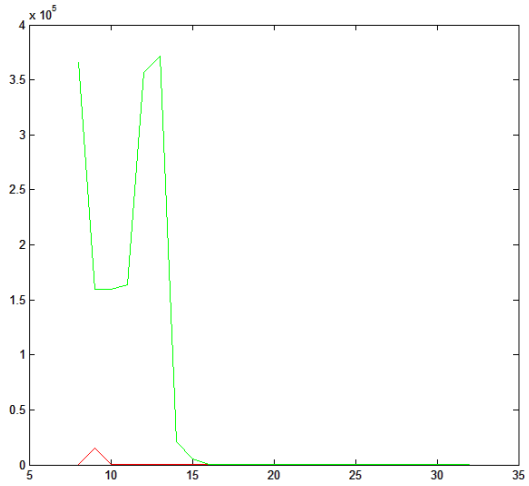
3. Hasil dan Pembahasan

Pada bagian ini peneliti mencoba melakukan analisa terhadap program perkalian matriks sebagai kasus. Percobaan dilakukan dengan variasi besar ukuran matriks sehingga mencapai ukuran maksimum. Variasi ukuran berupa variasi dimensi *grid*, *block*, dan *thread*. Pada percobaan ini peneliti juga memanfaatkan *shared memory* di GPU. Selain melihat akibat dari variasi dimensi *grid*, *block*, dan *thread*, peneliti juga akan melihat perbandingan waktu komputasi dan *error* diantara setiap jenis prosesor, yaitu CPU, GPU *non-shared*, dan GPU dengan *shared memory*.

Untuk melihat dampak dari variasi ukuran digunakan program perkalian matriks yang menghasilkan *output* berupa waktu yang dibutuhkan oleh GPU dengan *shared-memory*, GPU dengan *non-shared memory*, dan CPU biasa dalam mengalikan matriks dengan ukuran yang berbeda-beda. Selain itu program ini juga memberikan *output* berupa *error* yang terjadi pada penghitungan matriks menggunakan GPU, baik dengan *shared memory* maupun dengan *non-shared memory*. Program akan meminta *input* dari *user* terlebih dahulu (*input* 1 untuk menghitung waktu eksekusi matriks, dan *input* 2 untuk mengecek selisih *error*, dan 3 adalah untuk melihat nilai variasi dari perubahan ukuran *thread* dan *block*).



Gambar 4. Grafik *error* pada skenario *device* yang digunakan.



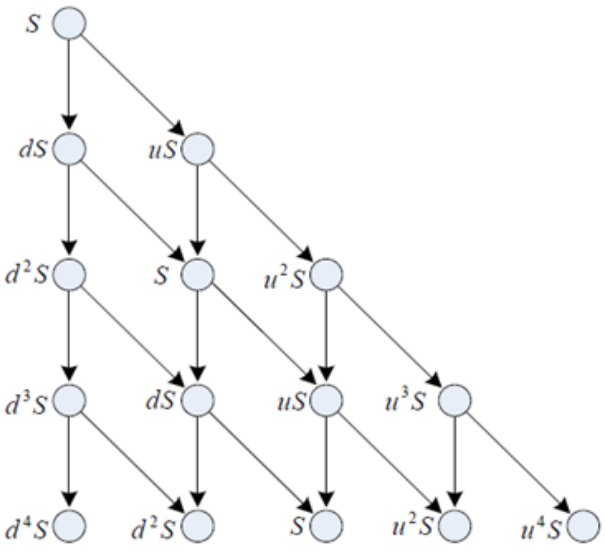
Gambar 5. Grafik *error* pada skenario *device* yang digunakan dengan variasi *thread.*

Di bawah ini adalah grafik hasil *output* program, garis merah menandakan *error* pada penggunaan *non-shared memory* dan garis hijau menandakan *error* pada penggunaan *shared memory*.

Berdasarkan grafik pada gambar 4, *error* yang terjadi pada perhitungan menggunakan GPU, baik menggunakan *shared memory* maupun *non shared*, tidak terlalu signifikan. Pertumbuhan fungsi *error* juga dapat ditoleransi sebab bersifat linier. Hal ini membuktikan bahwa akurasi hasil perhitungan menggunakan GPU cukup baik dengan tingkat *error* yang rendah serta pertumbuhan fungsi yang linear.

Selain melakukan perhitugan matriks dengan GPU dan CPU peneliti juga melakukan uji coba dalam variasi jumlah *thread* yang digunakan pada penghitungan perkalian matriks. Gambar 5 menunjukkan tampilan grafik pada *output* di atas, garis merah menandakan *error* yang terjadi pada eksekusi yang menggunakan *non-shared memory* dan garis hijau menandakan *error* yang terjadi pada eksekusi yang menggunakan *shared memory*.

Salah satu aplikasi akuntansi yang memanfaatkan GPU yaitu *Binomial Option Tree Model*. Akan dijelaskan mengenai sejarah dan penggunaan algoritma serta GPU pada aplikasi ini. Sejak dibuatnya *trading* opsi yang terorganisasi pada 1973, proses *pricing* (pemberian harga) pada opsi menjadi sangat penting. Opsi adalah sebuah persetujuan antara dua pihak, yaitu pembeli opsi dan penjual opsi. Opsi sendiri merupakan suatu *underlying assets*, misalnya, saham, yang merupakan aset tak terlihat. Pembeli opsi adalah seseorang yang memiliki hak untuk melakukan *trading* ini sedangkan pembeli opsi adalah orang yang melakukan persetujuan dengan sang pembeli, dengan melalui suatu penentuan nilai opsi pada waktu tertentu, hingga masuk waktu jatuh tempo. *Call option*, memberikan hak pada pemegang opsi untuk membeli, sedangkan *Put option* memberikan hak pada pemegang opsi untuk menjual.



Gambar 6. BOTM setelah 4 langkah waktu dT.

TABEL I
HASIL EKSPERIMEN PADA CPU DAN GPU

NUM_STEPS	OPT_N_MAX	CPU	GPU
32	16	0	0.13539
64	16	0	0.16189
128	16	0	0.15453
256	16	0	0.17891
521	16	20,000	0.27571
1,024	16	60,000	0.61382
2,048	16	260,000	1.88323
4,096	16	4,050,000	6.78163
8,192	16	35,620,000	25.9656
16,384	16	219,800,000	102.12378
32,768	16	∞	404.40869
65,536	16	∞	1,609.73853
131,072	16	∞	6,426.15432

Tipe dari transaksi opsi sendiri banyak, ada *European-type*, yang penilaian opsi dilakukan pada waktu jatuh tempo, ada juga *American-type* yang penilaiannya bisa dilakukan secara fleksibel pada waktu-waktu sebelum jatuh tempo, serta waktu jatuh tempo itu sendiri. Melihat pentingnya penilaian harga opsi tersebut, peneliti mencoba melakukan uji coba pada algoritma opsi itu sendiri. Di antara banyaknya algoritma yang ada, peneliti memilih salah satu algoritma yang cukup terkenal, yaitu *Binomial Option Tree Model* (selanjutnya dibaca BOTM), kali ini peneliti mencoba pada *pricing* opsi yang bertipe *European-type*. BOTM sendiri adalah suatu metode *iterative* yang melihat suatu nilai perubahan opsi secara menyeluruh, sejak dilakukan *strike price* pertama kali, hingga jatuh tempo. Pada tiap kurun waktu yang ditentukan, akan dienumerasi suatu nilai baru untuk opsi dengan asumsi bahwa nilai opsi bisa naik atau turun dengan probabilitas tertentu, misal, Pu, untuk probabilitas naik, dan Pd untuk probabilitas turun. Suatu nilai opsi S pada kurun waktu tertentu akan berubah menjadi u.S (nilai naik) atau d.S (nilai turun) pada waktu dT. Perhatikan contoh BOTM pada gambar 6. Hasil eksperimen dengan membandingkan implementasi BOMT pada CPU dan GPU dapat dilihat pada tabel I.

Dari data di atas tampak bahwa *running time* GPU jauh lebih cepat dibandingkan dengan *running time* CPU. Hal ini terjadi karena iterasi kedua, iterasi untuk mencari nilai *callValue* pada *node* sampai *root*, dikerjakan secara paralel dengan memanfaatkan fitur-fitur seperti *shared memory* yang pengaksesan datanya lebih cepat dibandingkan dengan global memori. *Shared memory* ini juga dimanfaatkan untuk melakukan reduksi pada proses penghitungan *callValue* pada setiap *node* sehingga mengurangi jumlah memori yang dipakai. Reduksi digunakan pada banyak algoritma paralel untuk meningkatkan paralelisme program. Pada program ini, jika reduksi tidak dilakukan maka kemungkinan besar penghitungan *callValue* akan dilakukan pada CPU. Hal ini tidak efisien karena iterasi yang paling banyak dilakukan ada pada bagian yang harus direduksi ini. Oleh sebab itu, dengan memparalelkan iterasi kedua maka program akan berjalan jauh lebih cepat.

Untuk mencari kompleksitas algoritma ini, perlu dicari kompleksitas iterasi yang paling dominan. Iterasi yang paling dominan terdapat pada iterasi kedua yang melakukan iterasi dari baris NUM_STEPS-1 sampai *root*. Iterasi ini berupa *nested loop* dengan jumlah *inner loop* sebanyak 1.

Secara umum jumlah iterasi dapat diperkirakan dengan cara seperti persamaan 1 berikut:

$$1 + 2 + 3 + ... + N \leq N + N + N + ... + N = N \times N = N^2 \approx O(N^2) \quad (1)$$

Sehingga kompleksitas algoritma yang diperoleh adalah $O(N^2)$.

Dari data di atas dapat dilihat bahwa setiap *input* bernilai dua kali *input* sebelumnya. Selain itu, perbandingan *output* CPU dengan *output* CPU sebelumnya rata-rata bernilai 4. Maka,

$$T(N) = c\,N^a = 60000 \approx 6000$$
$$T(2N) = c\,2^a N^a = 26000 \approx 24000$$
$$2^a = 4$$

Selanjutnya diperoleh a = 2 sehingga kompleksitas menjadi $O(N^2)$. Kompleksitas yang diperoleh ini cocok dengan kompleksitas yang diperkirakan sebelumnya. Jadi, secara umum kompleksitas algoritma *Binomial Tree Option* adalah $O(N^2)$.

5. Kesimpulan

Penggunaan GPU dapat meningkatkan efisiensi perhitungan pada suatu permasalahan yang melibatkan kalkulasi secara berulang-ulang.

Hal ini dikarenakan tingkat homogenitas pada data tersebut cukup tinggi sehingga perhitungan ini sangat cocok dilakukan oleh GPU yang dapat mengeksekusi data secara paralel pada waktu yang bersamaan.

Referensi

[1] J.C. Cox, S.A. Ross, & Mark Rubinstein, "Option Pricing: A Simplified Approach," *Journal of Financial Economics*, vol. 7, pp. 229-263, 1979.

[2] NVDIA Corporation, NVDIA CUDA C Programming Guide: Version 3.2. NVDIA Corp., 2010.

[3] V. Podlozhnyuk, Binomial Option Pricing Model. California: NVDIA Corporation, 2007.

[4] W.W. Hwu & D.B. Kirk, *Programming Massively Parallel Processors: A hands-on Approach*, Elsevier Inc., Burlington, pp. 39-57, 2010.