



UNIVERSITAS INDONESIA

PR 3 PEMROGRAMAN DI LINGKUNGAN GPU

LAPORAN TUGAS PEMROGRAMAN PARALEL

KELOMPOK III

Muhammad Fathurachman 1506706276

Otniel Yosi Viktorisa 1506706295

Yohanes Gultom 1506706345

**FAKULTAS ILMU KOMPUTER
PROGRAM STUDI MAGISTER ILMU KOMPUTER
DEPOK
MEI 2016**

DAFTAR ISI

Daftar Isi	ii
Daftar Gambar	iv
1 LINGKUNGAN PERCOBAAN	1
1.1 Lingkungan Pengembangan	1
1.2 Lingkungan Percobaan	1
1.2.1 Server GPU Fasilkom Universitas Indonesia (UI)	1
1.2.2 Personal Computer (Laptop)	2
1.2.3 Cluster Rocks University of California San Diego (UCSD)	2
2 PENGENALAN CUDA	4
2.1 Eksperimen	4
2.1.1 Eksperimen Perilaku Thread dan Block 1	4
2.1.1.1 Deskripsi Program	4
2.1.1.2 Hasil Eksperimen	5
2.1.2 Eksperimen Perilaku Thread dan Block 2	8
2.1.2.1 Deskripsi Program	8
2.1.2.2 Hasil Eksperimen	8
2.2 Kesimpulan	11
3 PERKALIAN MATRIKS-VEKTOR & MATRIKS BUJURSANGKAR	12
3.1 Eksperimen	12
3.1.1 Perbandingan Program Sekuensial dan CUDA	12
3.1.1.1 Deskripsi Program	12
3.1.1.2 Hasil Eksperimen	14
3.1.2 Program CUDA dengan Variasi ukuran <i>Grid/Block</i>	15
3.1.2.1 Deskripsi Program	15
3.1.2.2 Hasil Eksperimen	16
3.1.3 Program CUDA <i>Shared Memory</i> , CUBLAS dan MPI	17
3.1.3.1 Deskripsi Program	17
3.1.3.2 Hasil Eksperimen	19
3.2 Kesimpulan	23

4	CONJUGATE GRADIENT METHOD	25
4.1	Pendahuluan	25
4.2	Eksperimen	26
4.3	Kesimpulan	29
5	MOLECULAR DYNAMICS: AMBER	30
5.1	Pendahuluan	30
5.2	Eksperimen	30
5.3	Kesimpulan	32
6	DAFTAR KONTRIBUSI ANGGOTA	33

DAFTAR GAMBAR

2.1	Hasil eksperimen dengan variasi block 1	5
2.2	Hasil eksperimen dengan variasi block 2	6
2.3	Hasil eksperimen dengan variasi thread per block 1	7
2.4	Hasil eksperimen dengan variasi thread per block 2	7
2.5	Hasil eksperimen 1	9
2.6	Hasil eksperimen 2a	10
2.7	Hasil eksperimen 2b	10
3.1	Waktu eksekusi program matriks x vektor sekuensial dan CUDA . .	14
3.2	Waktu eksekusi program matriks x matriks sekuensial dan CUDA .	15
3.3	Waktu eksekusi program matriks x vektor CUDA dengan variasi ukuran <i>block</i> dan <i>grid</i>	16
3.4	Waktu eksekusi program matriks bujursangkar CUDA dengan vari- asi ukuran <i>block</i> dan <i>grid</i>	17
3.5	Waktu eksekusi program matriks x vektor <i>global memory</i> dan <i>shared memory</i> (940M)	19
3.6	Waktu eksekusi program matriks x vektor <i>global memory</i> dan <i>shared memory</i> (GTX 970)	20
3.7	Waktu eksekusi program matriks x vektor <i>global memory</i> dan <i>shared memory</i> (GTX 980)	20
3.8	Waktu eksekusi program perkalian matriks bujursangkar <i>global memory, shared memory, CUBLAS dan MPI</i> (940M)	21
3.9	Waktu eksekusi program perkalian matriks bujursangkar <i>shared memory dan CUBLAS</i> (940M)	21
3.10	Waktu eksekusi program perkalian matriks bujursangkar <i>global memory, shared memory, CUBLAS dan MPI</i> (GTX 970)	22
3.11	Waktu eksekusi program perkalian matriks bujursangkar <i>global memory, shared memory, CUBLAS dan MPI</i> (GTX 980)	22
3.12	Waktu eksekusi program perkalian matriks bujursangkar <i>shared memory dan CUBLAS</i> (970)	23
3.13	Waktu eksekusi program perkalian matriks bujursangkar <i>shared memory dan CUBLAS</i> (980)	23

4.1	Algoritma Conjugate Gradient Method.	25
4.2	Modifikasi nilai A	26
4.3	Modifikasi nilai b	26
4.4	Eksperimen 1 dengan ukuran array 4096	27
4.5	Eksperimen 2 dengan ukuran array 4096	27
4.6	Unjuk kerja Conjugate Gradient dengan CUDA	28
4.7	Unjuk kerja Conjugate Gradient dengan MPI pada cluster NBCR dengan variasi jumlah processor dan ukuran array	28
5.1	Eksperimen AMBER pada GPU (CUDA) dan CPU <i>cluster</i> (MPI)	31

BAB 1

LINGKUNGAN PERCOBAAN

1.1 Lingkungan Pengembangan

Program paralel yang digunakan di dalam eksperimen ini dibuat menggunakan bahasa C yang di-*compile* menggunakan pustaka OpenMPI pada sistem operasi Linux Ubuntu dan Mint.

Kode program-program dan laporan eksperimen ini kami simpan menggunakan layanan GitHub di alamat <https://github.com/yohanesgultom/parallel-programming-assignment>. Hal ini kami lakukan untuk mempermudah kolaborasi dalam pembuatan program dan laporan.

1.2 Lingkungan Percobaan

Dalam eksperimen yang dilakukan, kelompok kami menggunakan mesin-mesin dengan GPU NVIDIA dan *cluster* CPU Rocks University of California San Diego (UCSD).

1.2.1 Server GPU Fasikom Universitas Indonesia (UI)

Server dengan GPU NVIDIA merupakan lingkungan utama percobaan ini. Kelompok kami melakukan percobaan di 2 buah server Fasikom UI:

1. Server GTX 980 (152.118.31.27)
 - NVIDIA GTX 980 2048 CUDA Cores 4 GB GRAM
 - Intel(R) Core(TM) i7-3770 CPU 4 cores @ 3.40GHz
 - RAM 2x8 GB DDR3 1600 Mhz
 - SSD SAMSUNG MZ7TD128 128 GB
 - OS Debian 7 Wheezy x64
 - CUDA 7.0
 - Amber 14 dan Amber Tools 15
2. Server GTX 970 (152.118.31.34)

- NVIDIA GTX 970 1664 CUDA Cores 4 GB GRAM
- Intel(R) Core(TM) i7-3770 CPU 4 cores @ 3.40GHz
- RAM 2x8 GB DDR3 1600 Mhz
- SSD SAMSUNG MZ7TD128 128 GB
- OS Debian 7 Wheezy x64
- CUDA 7.0
- Amber 14 dan Amber Tools 15

Semua *server* ini dapat diakses dari jaringan Fasilkom menggunakan protokol *SSH* dan *credential Single Sign On* (SSO) UI. Sedangkan dari luar jaringan UI, semua *server* ini dapat diakses dengan masuk lebih dahulu ke kawung.cs.ui.ac.id menggunakan protokol *SSH* juga.

1.2.2 Personal Computer (Laptop)

Sebagai bahan perbandingan, kami juga menggunakan PC (*notebook*) yang juga menggunakan GPU NVIDIA yang mendukung CUDA:

- NVIDIA 940M 384 CUDA Cores 2 GB GRAM
- Intel(R) Core(TM) i7-5500U CPU 2 cores @ 3.40GHz
- RAM 8 GB DDR3 1600 Mhz
- SSD Crucial 250 GB
- OS Ubuntu 15.10 x64
- CUDA 7.5
- Amber 14 dan Amber Tools 15

1.2.3 Cluster Rocks University of California San Diego (UCSD)

*Cluster Rocks*¹ milik University of California San Diego (UCSD) ini dapat diakses pada alamat nbc-233.ucsd.edu menggunakan protokol *SSH* dari komputer yang telah didaftarkan *public key* nya. Berdasarkan informasi dari aplikasi *monitoring* Ganglia², cluster ini terdiri 10 *nodes* dengan total 80 prosesor.

¹www.rocksclusters.org

²<http://nbc-233.ucsd.edu/ganglia>

Pada *cluster* ini sudah terpasang pustaka komputasi paralel OpenMPI³ dan MPICH⁴ serta paket dinamika molekular AMBER. Program paralel MPI dan eksperimen AMBER dijalankan mekanisme antrian *batch-jobs* untuk menjamin ketersediaan sumberdaya komputasi (*computing nodes*) saat program dieksekusi. Pengaturan eksekusi program paralel ini ditangani oleh *Sun Grid Engine*⁵ yang juga tersedia dalam paket Rocks.

³<https://www.open-mpi.org/>

⁴<https://www.mpich.org/>

⁵<http://www.rocksclusters.org/roll-documentation/sge/5.4/>

BAB 2

PENGENALAN CUDA

2.1 Eksperimen

Pada bagian ini kami melakukan 2 eksperimen untuk mengamati perilaku *block* dan *thread* pada *kernel* dan mengamati pengaruh ukuran data terhadap eksekusi *kernel*.

2.1.1 Eksperimen Perilaku Thread dan Block 1

Bagian ini adalah eksperimen untuk mengamati perilaku *block* dan *thread* pada *kernel*.

2.1.1.1 Deskripsi Program

Program yang digunakan merupakan hasil modifikasi program dari *slide* Intro to Cuda: Program dimodifikasi agar dapat menerima argumen untuk menentukan jumlah *block* dan jumlah *thread* per *block*.

```
blockgrid [ukuran array] [thread/block] [block]
```

Program akan menjalankan 3 buah *kernel*. *Kernel* pertama akan mengisi array dengan angka konstan 9, *kernel* kedua akan mengisi array dengan *block id*, dan *kernel* ketiga akan mengisi array dengan *kernel id*.

```
__global__ void kernel1( int *a )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = 9;
}

__global__ void kernel2( int *a )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}

__global__ void kernel3( int *a )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

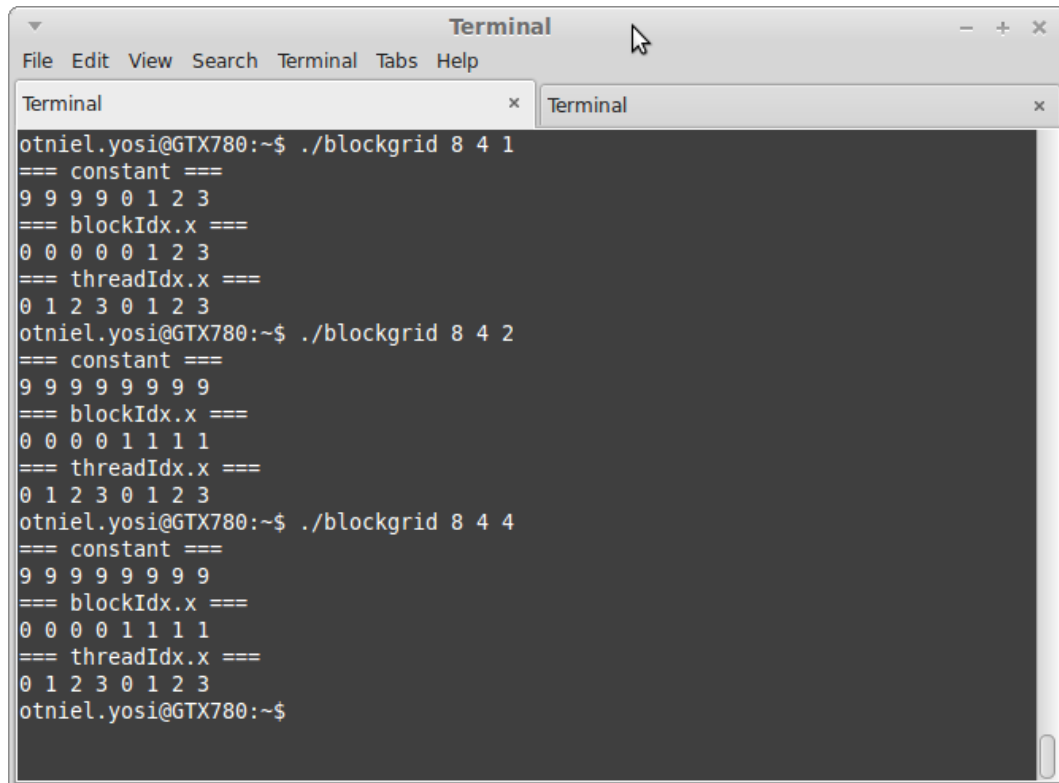
```

    a[idx] = threadIdx.x;
}

```

2.1.1.2 Hasil Eksperimen

Berikut adalah hasil eksperimen dengan variasi jumlah *block*.



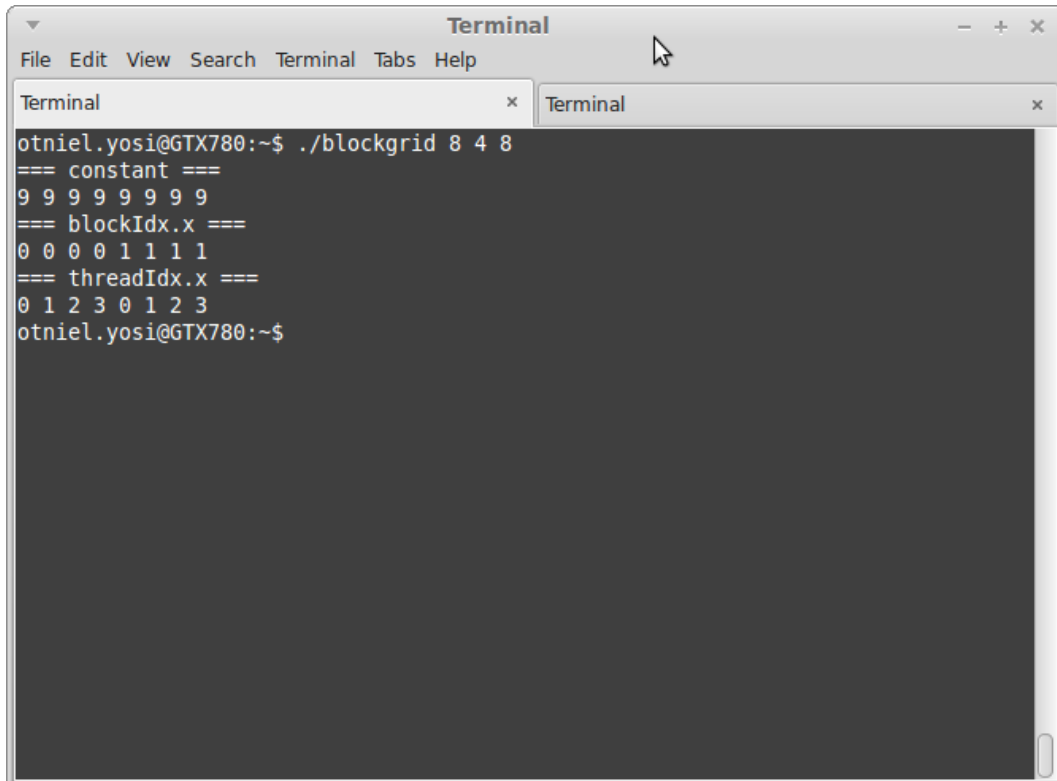
```

Terminal
File Edit View Search Terminal Tabs Help

Terminal
otniel.yosi@GTX780:~$ ./blockgrid 8 4 1
=== constant ===
9 9 9 9 0 1 2 3
=== blockIdx.x ===
0 0 0 0 0 1 2 3
=== threadIdx.x ===
0 1 2 3 0 1 2 3
otniel.yosi@GTX780:~$ ./blockgrid 8 4 2
=== constant ===
9 9 9 9 9 9 9 9
=== blockIdx.x ===
0 0 0 0 1 1 1 1
=== threadIdx.x ===
0 1 2 3 0 1 2 3
otniel.yosi@GTX780:~$ ./blockgrid 8 4 4
=== constant ===
9 9 9 9 9 9 9 9
=== blockIdx.x ===
0 0 0 0 1 1 1 1
=== threadIdx.x ===
0 1 2 3 0 1 2 3
otniel.yosi@GTX780:~$

```

Gambar 2.1: Hasil eksperimen dengan variasi block 1



```
otniel.yosi@GTX780:~$ ./blockgrid 8 4 8
=== constant ===
9 9 9 9 9 9 9 9
=== blockIdx.x ===
0 0 0 0 1 1 1 1
=== threadIdx.x ===
0 1 2 3 0 1 2 3
otniel.yosi@GTX780:~$
```

Gambar 2.2: Hasil eksperimen dengan variasi block 2

Berikut adalah hasil eksperimen dengan variasi jumlah *thread* per *block*.

```

otniel.yosi@GTX780:~$ ./blockgrid 8 1 4
=== constant ===
9 9 9 9 4 5 6 7
=== blockIdx.x ===
0 1 2 3 4 5 6 7
=== threadIdx.x ===
0 0 0 0 4 5 6 7
otniel.yosi@GTX780:~$ ./blockgrid 8 2 4
=== constant ===
9 9 9 9 9 9 9 9
=== blockIdx.x ===
0 0 1 1 2 2 3 3
=== threadIdx.x ===
0 1 0 1 0 1 0 1
otniel.yosi@GTX780:~$ ./blockgrid 8 4 4
=== constant ===
9 9 9 9 9 9 9 9
=== blockIdx.x ===
0 0 0 0 1 1 1 1
=== threadIdx.x ===
0 1 2 3 0 1 2 3
otniel.yosi@GTX780:~$

```

Gambar 2.3: Hasil eksperimen dengan variasi thread per block 1

```

otniel.yosi@GTX780:~$ ./blockgrid 8 8 4
=== constant ===
9 9 9 9 9 9 9 9
=== blockIdx.x ===
0 0 0 0 0 0 0 0
=== threadIdx.x ===
0 1 2 3 4 5 6 7
otniel.yosi@GTX780:~$

```

Gambar 2.4: Hasil eksperimen dengan variasi thread per block 2

Dari hasil eksperimen terlihat bahwa konfigurasi yang benar untuk menjalankan program adalah jumlah *thread* per *block* dikalikan jumlah *block* yang digunakan tidak melebihi jumlah data yang diproses. Saat program diberi parameter dengan jumlah data 8, jumlah *thread* per *block* 4, dan jumlah *block* 2 terlihat program dijalankan oleh 2 *block* dengan id 0 dan 1 dan pada tiap *block* terdapat *thread* dengan id 0 sampai 3. Namun saat jumlah *block* ditingkatkan dengan jumlah data dan *thread* per *block* yang sama, program tetap menampilkan 2 *block* dengan id 0 dan 1 karena 2 *block* tersebut sudah cukup untuk menangani jumlah data yang digunakan.

2.1.2 Eksperimen Perilaku Thread dan Block 2

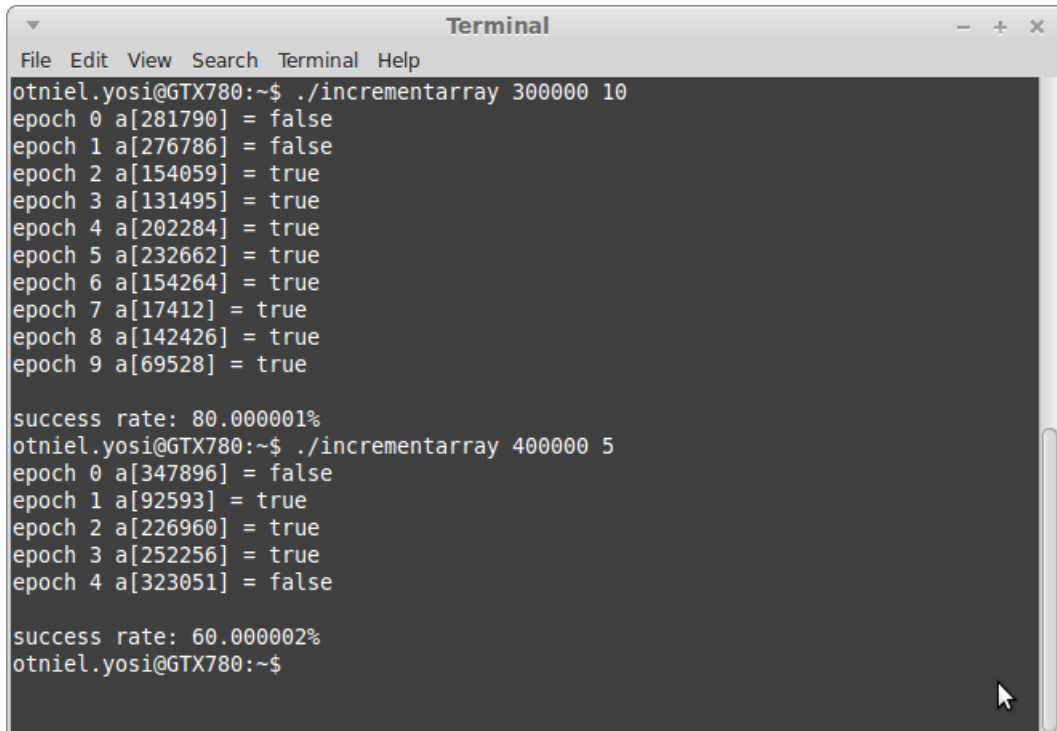
Bagian ini adalah eksperimen untuk mengamati pengaruh ukuran data terhadap eksekusi *kernel*.

2.1.2.1 Deskripsi Program

Program yang digunakan sesuai dengan lampiran pada soal tugas. Program dimodifikasi agar dapat menerima masukan berupa ukuran array dan banyaknya iterasi yang akan dilakukan.

2.1.2.2 Hasil Eksperimen

Eksperimen pertama dilakukan dengan memasukkan ukuran array maksimal dan banyaknya iterasi. Ukuran array yang digunakan di setiap iterasi akan diacak saat iterasi berlangsung. Banyaknya *block* yang digunakan adalah 4 dan banyaknya *thread* per *block* adalah ukuran array dibagi banyaknya *block*. Berikut adalah hasil eksperimennya.



```

Terminal
File Edit View Search Terminal Help
otniel.yosi@GTX780:~$ ./incrementarray 300000 10
epoch 0 a[281790] = false
epoch 1 a[276786] = false
epoch 2 a[154059] = true
epoch 3 a[131495] = true
epoch 4 a[202284] = true
epoch 5 a[232662] = true
epoch 6 a[154264] = true
epoch 7 a[17412] = true
epoch 8 a[142426] = true
epoch 9 a[69528] = true

success rate: 80.000001%
otniel.yosi@GTX780:~$ ./incrementarray 400000 5
epoch 0 a[347896] = false
epoch 1 a[92593] = true
epoch 2 a[226960] = true
epoch 3 a[252256] = true
epoch 4 a[323051] = false

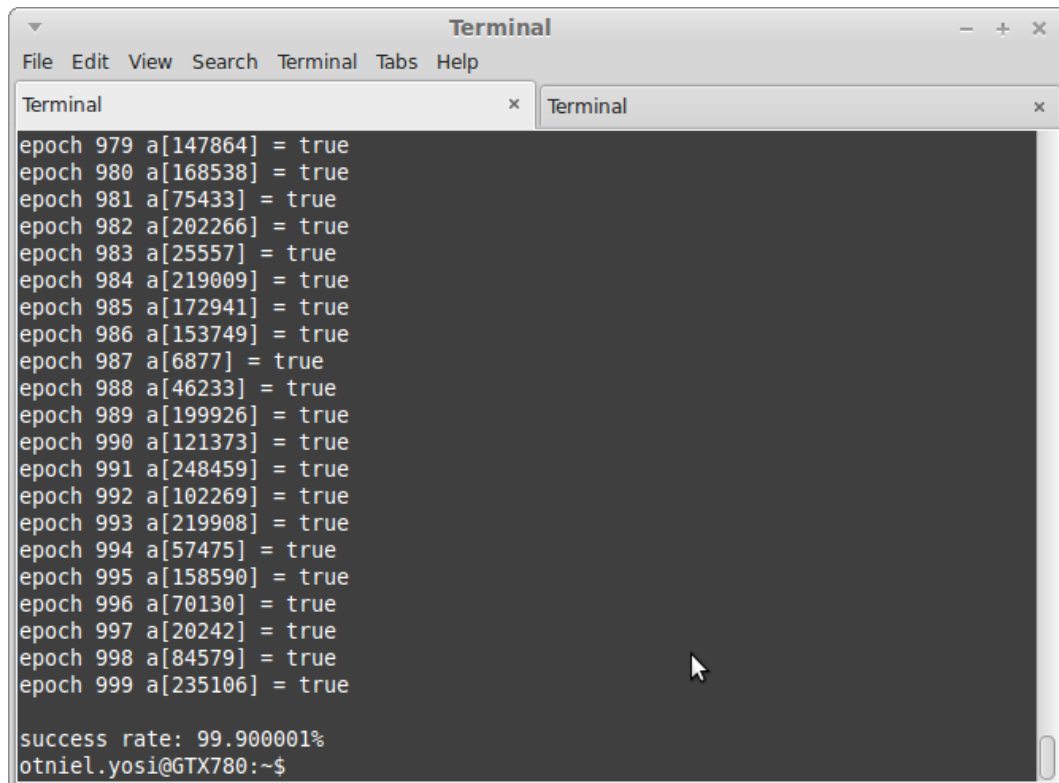
success rate: 60.000002%
otniel.yosi@GTX780:~$

```

Gambar 2.5: Hasil eksperimen 1

Terlihat program mengeluarkan nilai *false* pada ukuran array diatas 270000. Program akan mengeluarkan nilai *false* ketika ukuran array lebih dari atau sama dengan 262140 karena $262140/4$ adalah 65535 yang merupakan batas maksimum banyaknya *block* dalam grid 1D.

Eksperimen kedua dilakukan hampir sama dengan eksperimen pertama tetapi banyaknya *thread* per *block* disesuaikan dengan iterasi (iterasi ke-*i* maka banyaknya *thread* per *block* adalah *i*). Berikut adalah hasil eksperimennya.

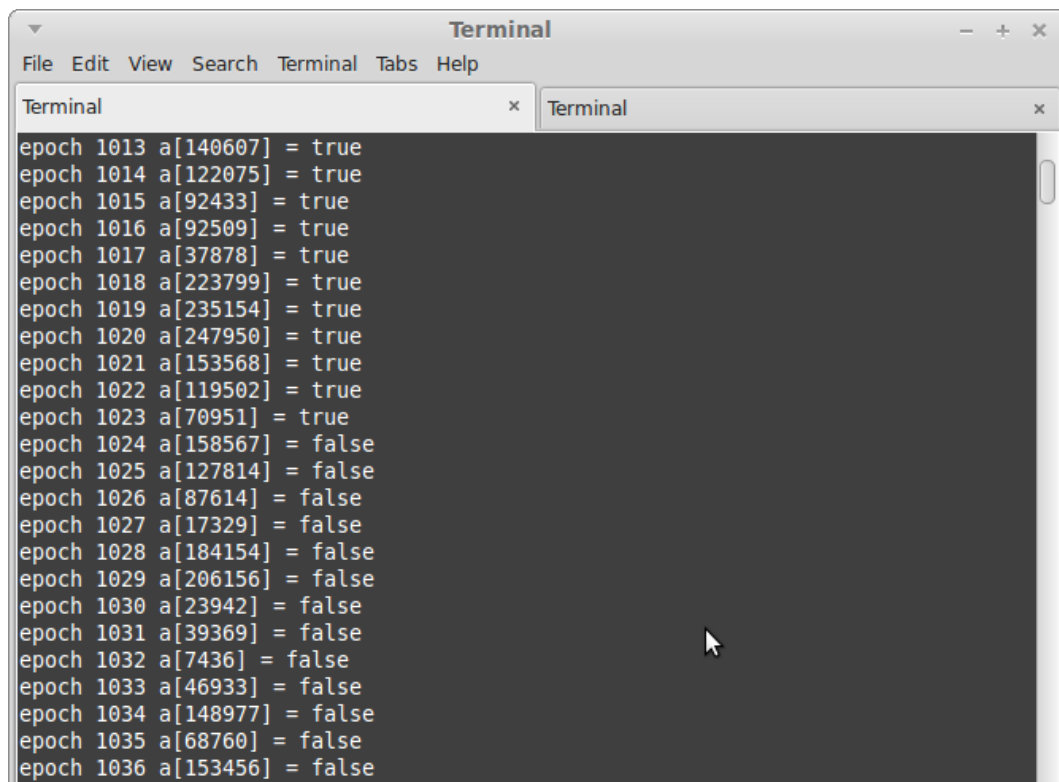


```
Terminal
File Edit View Search Terminal Tabs Help

epoch 979 a[147864] = true
epoch 980 a[168538] = true
epoch 981 a[75433] = true
epoch 982 a[202266] = true
epoch 983 a[25557] = true
epoch 984 a[219009] = true
epoch 985 a[172941] = true
epoch 986 a[153749] = true
epoch 987 a[6877] = true
epoch 988 a[46233] = true
epoch 989 a[199926] = true
epoch 990 a[121373] = true
epoch 991 a[248459] = true
epoch 992 a[102269] = true
epoch 993 a[219908] = true
epoch 994 a[57475] = true
epoch 995 a[158590] = true
epoch 996 a[70130] = true
epoch 997 a[20242] = true
epoch 998 a[84579] = true
epoch 999 a[235106] = true

success rate: 99.900001%
otniel.yosi@GTX780:~$
```

Gambar 2.6: Hasil eksperimen 2a



```
Terminal
File Edit View Search Terminal Tabs Help

epoch 1013 a[140607] = true
epoch 1014 a[122075] = true
epoch 1015 a[92433] = true
epoch 1016 a[92509] = true
epoch 1017 a[37878] = true
epoch 1018 a[223799] = true
epoch 1019 a[235154] = true
epoch 1020 a[247950] = true
epoch 1021 a[153568] = true
epoch 1022 a[119502] = true
epoch 1023 a[70951] = true
epoch 1024 a[158567] = false
epoch 1025 a[127814] = false
epoch 1026 a[87614] = false
epoch 1027 a[17329] = false
epoch 1028 a[184154] = false
epoch 1029 a[206156] = false
epoch 1030 a[23942] = false
epoch 1031 a[39369] = false
epoch 1032 a[7436] = false
epoch 1033 a[46933] = false
epoch 1034 a[148977] = false
epoch 1035 a[68760] = false
epoch 1036 a[153456] = false
```

Gambar 2.7: Hasil eksperimen 2b

Terlihat program mengeluarkan nilai *false* pada saat jumlah *block* lebih dari atau sama dengan 1024. Nilai tersebut adalah nilai maksimum banyaknya *thread* yang bisa digunakan dalam suatu *block*.

2.2 Kesimpulan

- Setiap *thread* dalam *block* dan setiap *block* dalam *grid* memiliki id yang unik.
- Dengan menggunakan kombinasi *thread id* dan *block id* kita bisa mendapatkan index yang unik di setiap *kernel*.
- *Device* memiliki batasan dalam hal jumlah memory, banyaknya *block*, dan banyaknya *thread* per *block* yang bisa digunakan.
- Konfigurasi yang melebihi batasan akan mengakibatkan kesalahan akses memori yang mengakibatkan kegagalan program.

BAB 3

PERKALIAN MATRIKS-VEKTOR & MATRIKS BUJURSANGKAR

3.1 Eksperimen

Pada bagian ini kami melakukan eksperimen perkalian matriks-vektor dan matriks bujursangkar pada CPU (sekuensial), CUDA, CUDA dengan *shared memory*, CUBLAS dan CPU (*cluster MPI*) untuk membandingkan kinerjanya.

3.1.1 Perbandingan Program Sekuensial dan CUDA

Pada percobaan ini kami membuat membandingkan waktu eksekusi program perkalian matriks-vektor dan perkalian matriks bujursangkar sekuensial (CPU) dengan paralel CUDA (GPU).

3.1.1.1 Deskripsi Program

1. Program perkalian matriks-vektor/matriks bujursangkar sekuensial pada CPU `mmul.c`¹ dengan cara penggunaan program:

```
$ mmul.c [baris/kolom matrix] [baris vektor] 1 [repetisi]
```

Argumen program:

- (a) Baris/kolom matriks: ukuran baris atau kolom matriks bujursangkar
- (b) Baris vektor: ukuran vektor
- (c) Repetisi: banyaknya perkalian diulang (untuk diambil rata-ratanya)

```
$ mmul.c [baris matrix A] [kolom matriks A/baris matriks B]  
[kolom matriks B] [repetisi]
```

Argumen program:

- (a) Baris matriks A: ukuran baris atau kolom matriks bujursangkar

¹<https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/PR3/problem2/mmul.c>

(b) Kolom matriks A/baris matriks B: ukuran kolom matriks A/baris matriks B

(c) Repetisi: banyaknya perkalian diulang (untuk diambil rata-ratanya)

2. Program perkalian matriks-vektor/matriks bujursangkar paralel dengan CUDA `mmul_cuda.cu`². Program ini diadaptasi dari satu sumber eksternal³. Cara penggunaannya:

```
$ mmul_cuda.cu [baris/kolom matrix] [baris vektor] 1 0 [
  repetisi]
```

Argumen program:

(a) Baris/kolom matriks: ukuran baris atau kolom matriks bujursangkar

(b) Baris vektor: ukuran vektor

(c) Repetisi: banyaknya perkalian diulang (untuk diambil rata-ratanya)

```
$ mmul_cuda.cu [baris matrix A] [kolom matriks A/baris
  matriks B] [kolom matriks B] [repetisi]
```

Argumen program:

(a) Baris matriks A: ukuran baris atau kolom matriks bujursangkar

(b) Kolom matriks A/baris matriks B: ukuran kolom matriks A/baris matriks B

(c) Repetisi: banyaknya perkalian diulang (untuk diambil rata-ratanya)

Program `mmul_cuda.cu` ini memiliki ukuran *threads/block* tetap yaitu `(TILE_WIDTH, TILE_WIDTH)` dengan nilai `TILE_WIDTH = 10`. Sedangkan ukuran *grid* akan bergantung pada ukuran matriks/vektor dengan rumus:

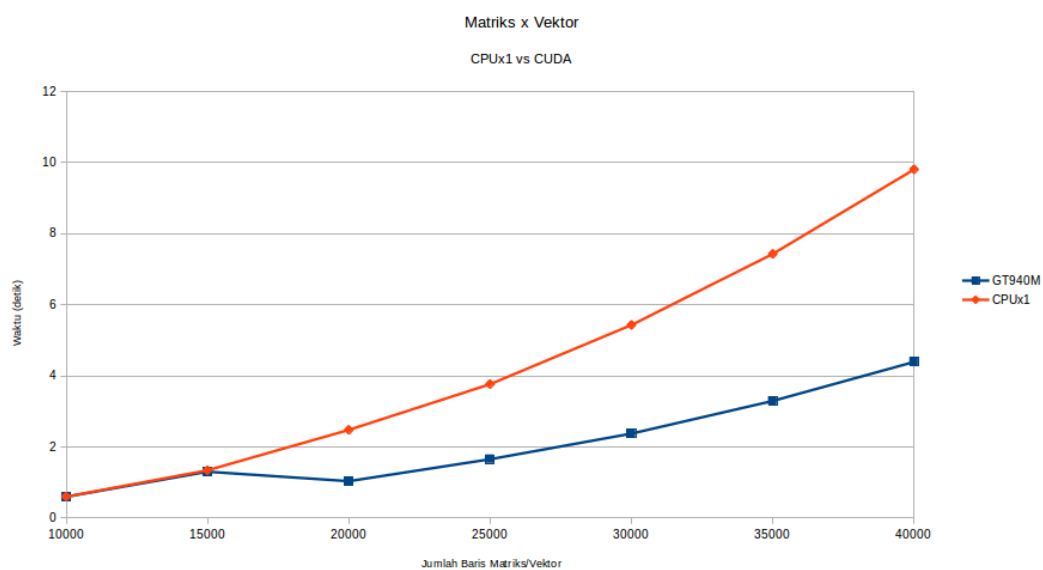
```
GRID SIZE = ((baris vektor-1) / TILE_WIDTH + 1, (baris
  matriks-1) / TILE_WIDTH + 1)
```

²https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/PR3/problem2/mmul_cuda.cu

³Sumber: <https://gist.github.com/wh5a/4313739>

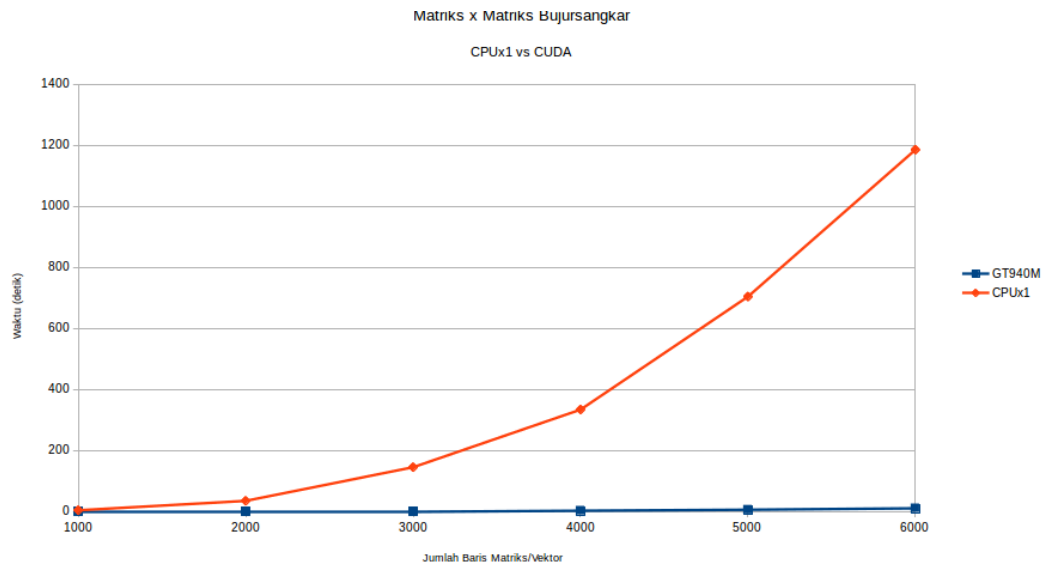
3.1.1.2 Hasil Eksperimen

Hasil eksekusi program perkalian matriks-vektor sekuensial dan CUDA (pada GPU 940M) dapat dilihat pada grafik gambar 3.1. Pada gambar tersebut terlihat bahwa waktu eksekusi program sekuensial dan CUDA masih sama dengan ukuran data 10.000 (matriks 10.000×10.000 dan vektor 10.000) hingga 15.000. Tapi pada ukuran data 20.000 hingga 40.000 terlihat bahwa program CUDA membutuhkan waktu yang lebih sedikit dibanding program sekuensial. Pada ukuran data terbesar di eksperimen ini, terlihat bahwa waktu yang dibutuhkan CUDA hanyalah $1/2$ dari program sekuensial.



Gambar 3.1: Waktu eksekusi program matriks x vektor sekuensial dan CUDA

Perbedaan waktu eksekusi terlihat semakin jelas pada hasil eksekusi program perkalian matriks bujursangkar pada grafik gambar 3.2. Waktu eksekusi program sekuensial dan CUDA hanya sama pada ukuran data 1.000 (perkalian dua matriks 1.000×1.000). Sedangkan untuk ukuran data 2.000 sampai 6.000, program CUDA membutuhkan waktu yang jauh lebih sedikit. Bahkan pada ukuran data terbesar di eksperimen ini, terlihat bahwa waktu yang dibutuhkan CUDA hanyalah $1/1000$ dari program sekuensial.



Gambar 3.2: Waktu eksekusi program matriks x matriks sekuensial dan CUDA

3.1.2 Program CUDA dengan Variasi ukuran *Grid/Block*

Tujuan dari eksperimen ini adalah membandingkan kinerja perkalian matriks-vektor/matriks bujursangkar CUDA dengan variasi ukuran *grid* dan *block*.

3.1.2.1 Deskripsi Program

Ada tujuh buah program⁴ yang digunakan pada eksperimen ini. Semua program ini diadaptasi dari satu sumber eksternal⁵. Ketujuh program ini hanya dibedakan oleh nilai `TILE_SIZE`-nya, yaitu sesuai dengan angka terakhir pada nama programnya (kecuali program `mmul_cuda.cu` yang memiliki `TILE_SIZE = 10`):

```
$ mul_cuda.cu [baris A] [kolom A/baris B] [kolom B] 0 [repetisi]
$ mul_cuda_20.cu [baris A] [kolom A/baris B] [kolom B] 0 [repetisi]
$ mul_cuda_30.cu [baris A] [kolom A/baris B] [kolom B] 0 [repetisi]
$ mul_cuda_40.cu [baris A] [kolom A/baris B] [kolom B] 0 [repetisi]
$ mul_cuda_50.cu [baris A] [kolom A/baris B] [kolom B] 0 [repetisi]
$ mul_cuda_60.cu [baris A] [kolom A/baris B] [kolom B] 0 [repetisi]
```

⁴<https://github.com/yohanesgultom/parallel-programming-assignment/tree/master/PR3/problem2>

⁵Sumber:<https://gist.github.com/wh5a/4313739>

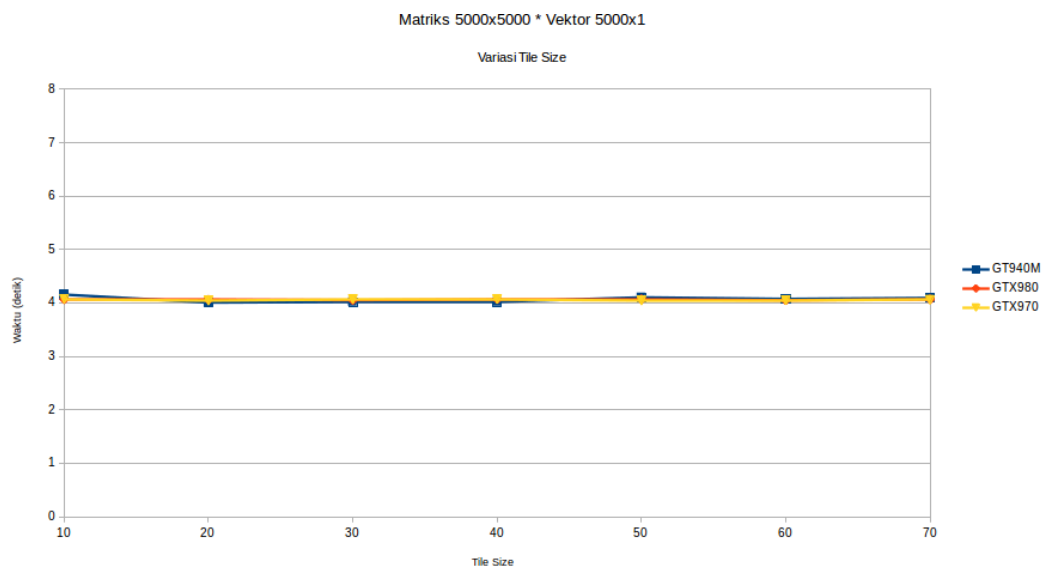
```
$ mul_cuda_70.cu [baris A] [kolom A/baris B] [kolom B] 0 [repetisi  
]
```

Seperti yang dijelaskan di eksperimen sebelumnya, program `mmul_cuda.cu` dan `mmul_cuda_XX.cu` ini memiliki ukuran *threads/block* tetap yaitu `(TILE_WIDTH, TILE_WIDTH)`. Sedangkan ukuran *grid* akan bergantung pada ukuran matriks/vektor dengan rumus:

```
GRID SIZE = ((baris vektor-1) / TILE_WIDTH + 1, (baris matriks-1)  
/ TILE_WIDTH + 1)
```

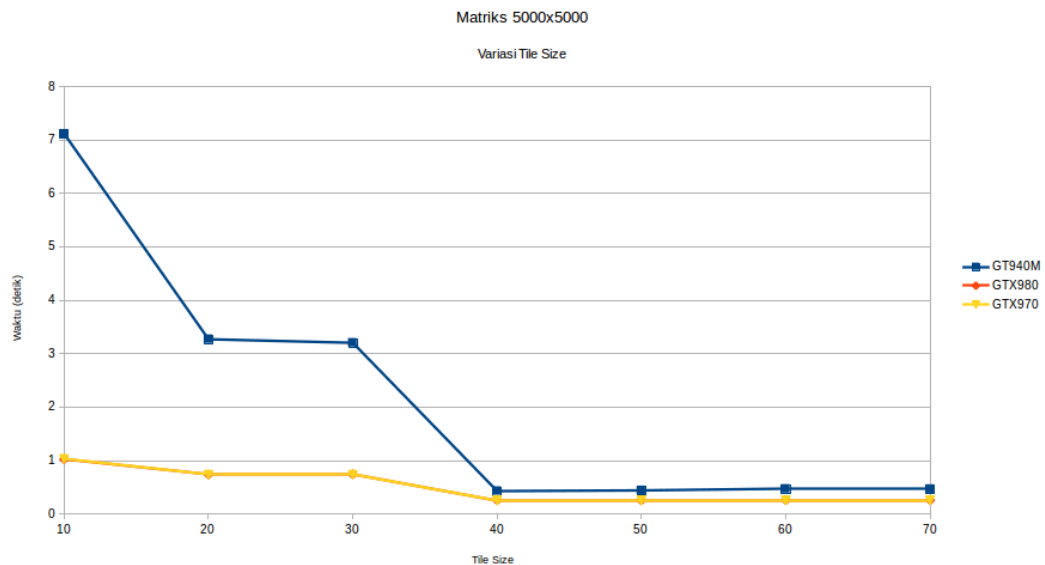
3.1.2.2 Hasil Eksperimen

Eksperimen perkalian matriks-vektor CUDA dengan ukuran data tetap (matriks 5.000×5.000 dan vektor 5.000) variasi ukuran *block* dan *grid* pada tiga buah GPU (GTX 980, GTX 970 dan 940M) memberikan hasil seperti pada grafik gambar 3.3. Hasil eksperimen menunjukkan bahwa tidak ada perbedaan signifikan ketika dilakukan variasi ukuran *block* dan *grid*. Hal ini terjadi secara konsisten di semua GPU yang digunakan.



Gambar 3.3: Waktu eksekusi program matriks x vektor CUDA dengan variasi ukuran *block* dan *grid*

Sedangkan pada perkalian matriks bujursangkar (dua buah matriks 5.000×5.000) yang hasilnya ditunjukkan pada gambar 3.4, terlihat adanya penurunan waktu eksekusi ketika ukuran `TILE_SIZE` diperbesar (ukuran *threads/block* semakin besar dan ukuran *grid* semakin kecil). Tetapi ketika ukuran `TILE_SIZE` dibuat lebih besar dari 40, tidak ada perubahan waktu eksekusi yang signifikan.



Gambar 3.4: Waktu eksekusi program matriks bujursangkar CUDA dengan variasi ukuran *block* dan *grid*

3.1.3 Program CUDA *Shared Memory*, CUBLAS dan MPI

Pada eksperimen ini akan diamati perbedaan waktu eksekusi program perkalian matriks-vektor CUDA yang hanya menggunakan *global memory* (normal) dan yang menggunakan *shared memory* (*optimized*). Selain itu untuk kasus perkalian matriks bujursangkar, akan dibandingkan waktu eksekusi program CUDA dengan *global memory* (normal), CUDA dengan *shared memory* (*optimized*), CUBLAS dan MPI (25 CPU pada *cluster UCSD*).

3.1.3.1 Deskripsi Program

1. Program perkalian matriks-vektor/matriks bujursangkar paralel dengan CUDA `mmul_cuda.cu`⁶. Program ini diadaptasi dari satu sumber eksternal⁷.

Cara penggunaannya:

```
$ mmul_cuda.cu [baris A] [kolom A/baris B] [kolom B] 0 [
  repetisi] [optimized]
```

Argumen program:

- (a) Baris matriks A: ukuran baris atau kolom matriks bujursangkar

⁶https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/PR3/problem2/mmul_cuda.cu

⁷Sumber: <https://gist.github.com/wh5a/4313739>

- (b) Kolom matriks A/baris matriks B: ukuran kolom matriks A/baris matriks B
- (c) Repetisi: banyaknya perkalian diulang (untuk diambil rata-ratanya)
- (d) *Optimized*: menggunakan *global memory* saja (0) atau optimasi dengan *shared memory* (1)

Program `mmul_cuda.cu` ini memiliki ukuran *threads/block* tetap yaitu (`TILE_WIDTH`, `TILE_WIDTH`) dengan nilai `TILE_WIDTH = 10`. Sedangkan ukuran *grid* akan bergantung pada ukuran matriks/vektor dengan rumus:

```
GRID SIZE = ((baris vektor-1) / TILE_WIDTH + 1, (baris
             matriks-1) / TILE_WIDTH + 1)
```

2. Program perkalian matriks-vektor/matriks bujursangkar paralel dengan CUBLAS `mmul_cublas.cu`⁸. Program ini diadaptasi dari satu sumber eksternal⁹. Cara penggunaannya:

```
$ mmul_cublas.cu [baris A] [kolom A/baris B] [kolom B] [
  repetisi]
```

Argumen program:

- (a) Baris matriks A: ukuran baris atau kolom matriks bujursangkar
- (b) Kolom matriks A/baris matriks B: ukuran kolom matriks A/baris matriks B
- (c) Repetisi: banyaknya perkalian diulang (untuk diambil rata-ratanya)

3. Program perkalian matriks-vektor/matriks bujursangkar paralel dengan MPI `mmul_rowwise.c`¹⁰. Cara penggunaannya:

```
$ mmul_rowwise.c [baris A] [kolom A/baris B] [kolom B]
```

Argumen program:

- (a) Baris matriks A: ukuran baris atau kolom matriks bujursangkar
- (b) Kolom matriks A/baris matriks B: ukuran kolom matriks A/baris matriks B

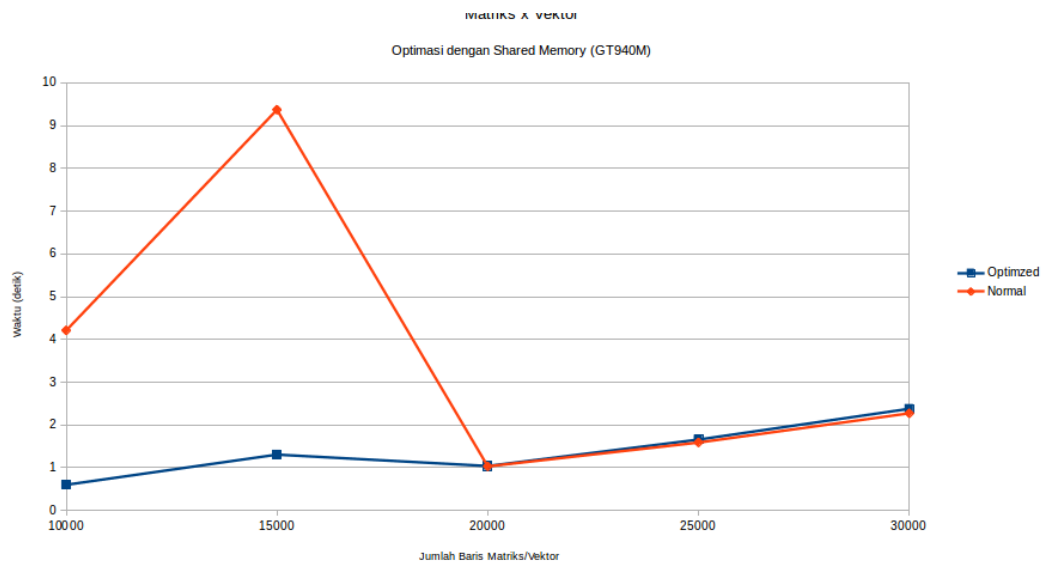
⁸https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/PR3/problem2/mmul_cublas.cu

⁹https://raw.githubusercontent.com/sol-prog/cuda_cublas_curand_thrust/master/mmul_1.cu

¹⁰https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/PR3/problem2/mmul_rowwise.c

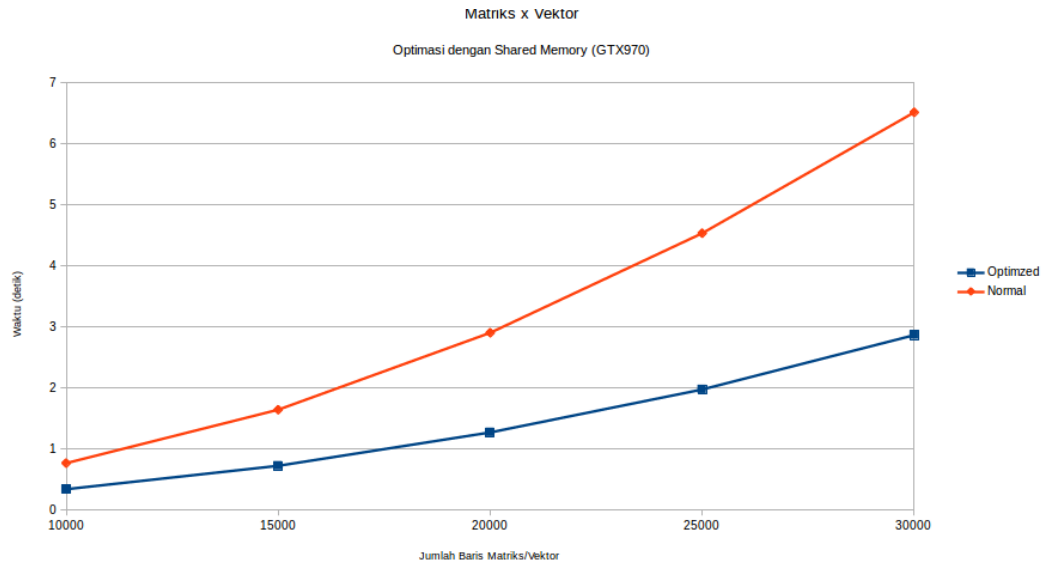
3.1.3.2 Hasil Eksperimen

Seperti yang terlihat pada gambar 3.5, hasil eksekusi perkalian matriks-vektor CUDA yang hanya menggunakan *global memory* dan yang dioptimasi dengan menggunakan *shared memory* pada GPU 940M menunjukkan bahwa optimasi dengan *shared memory* hanya mengurangi waktu eksekusi untuk ukuran data di bawah 20.000 (matriks 20.000×20.000 dan vektor 20.000). Sedangkan untuk ukuran yang lebih besar waktu eksekusinya sama.

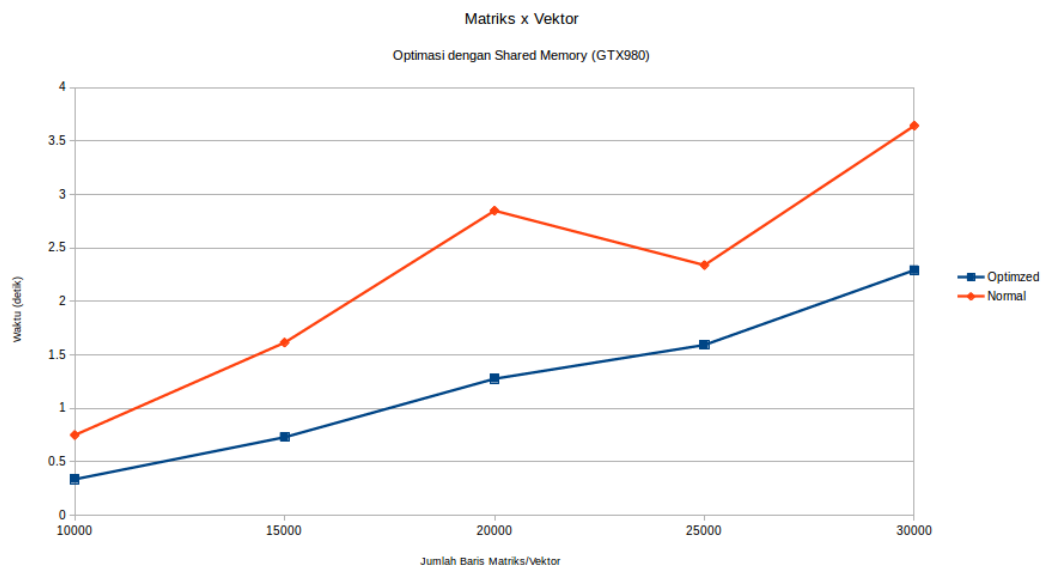


Gambar 3.5: Waktu eksekusi program matriks x vektor *global memory* dan *shared memory* (940M)

Hasil berbeda kami dapati pada GPU GTX 970 (gambar 3.6) dan GTX 980 (gambar 3.7) di mana program perkalian matriks-vektor dengan *shared memory* selalu lebih cepat (0,5-3 detik) dari program yang hanya menggunakan *global memory*. Menurut kami hal ini disebabkan karena sumber daya GPU 940M juga dipakai oleh OS mesinnya untuk menampilkan *Graphical User Interface* (GUI). Di mana untuk GPU lainnya, OS yang digunakan tidak menggunakan GUI sehingga sumber daya GPU hanya digunakan oleh program CUDA.



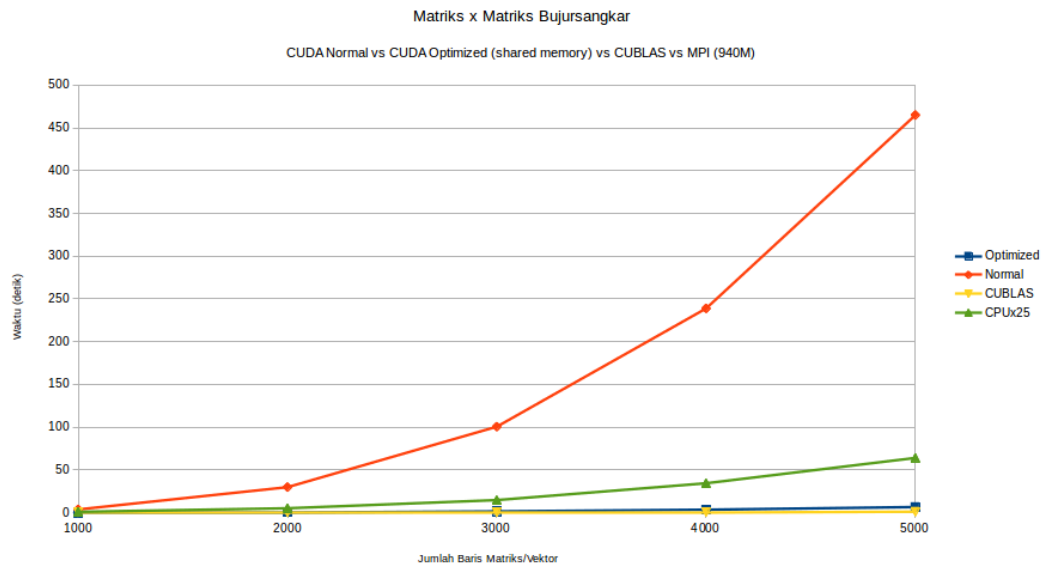
Gambar 3.6: Waktu eksekusi program matriks x vektor *global memory* dan *shared memory* (GTX 970)



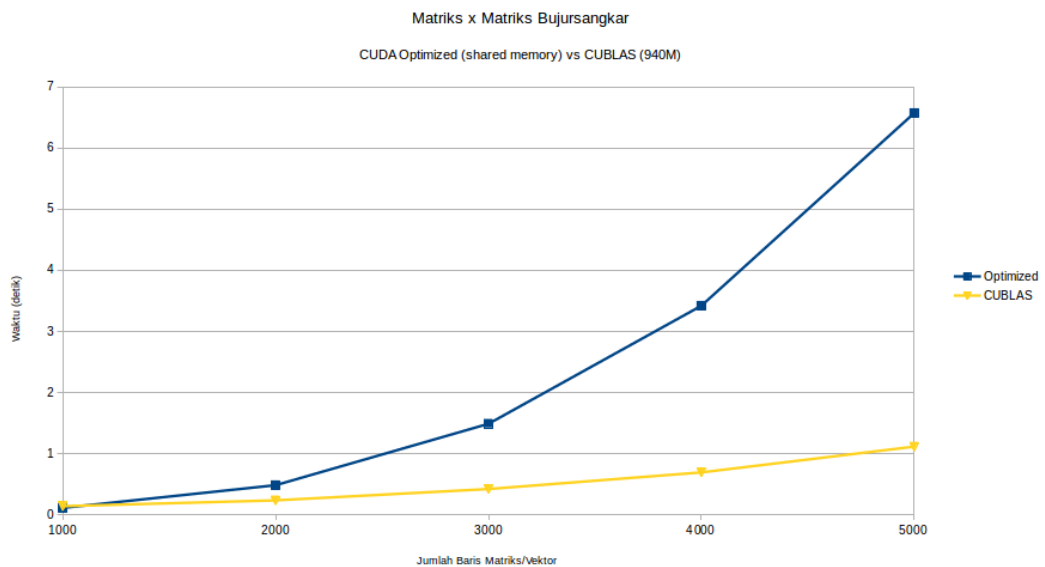
Gambar 3.7: Waktu eksekusi program matriks x vektor *global memory* dan *shared memory* (GTX 980)

Pada kasus perkalian matriks bujursangkar, perbandingan dilakukan sekaligus pada program CUDA dengan *global memory*, CUDA dengan *shared memory*, CUBLAS dan MPI. Pada GPU 940M (gambar 3.8) terlihat bahwa yang paling lambat adalah program CUDA dengan *global memory* saja. Urutan program dari yang paling lambat sampai paling cepat adalah program CUDA dengan *global memory* saja, MPI (*cluster 25 CPU*), CUDA dengan *shared memory* dan CUBLAS.

Jika dilihat lebih rinci (gambar 3.9), terlihat bahwa sebenarnya pada ukuran matriks di bawah 1.000, program CUDA dengan *shared memory* sama cepat dengan CUBLAS. Baru pada ukuran lebih besar dari itu CUBLAS lebih cepat daripada program CUDA dengan *shared memory*.



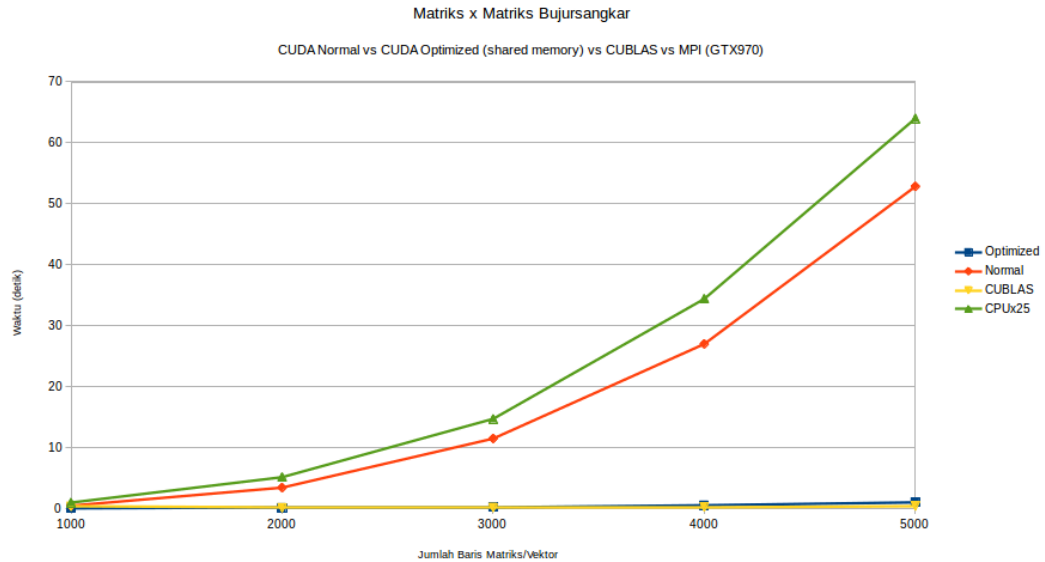
Gambar 3.8: Waktu eksekusi program perkalian matriks bujursangkar *global memory*, *shared memory*, CUBLAS dan MPI (940M)



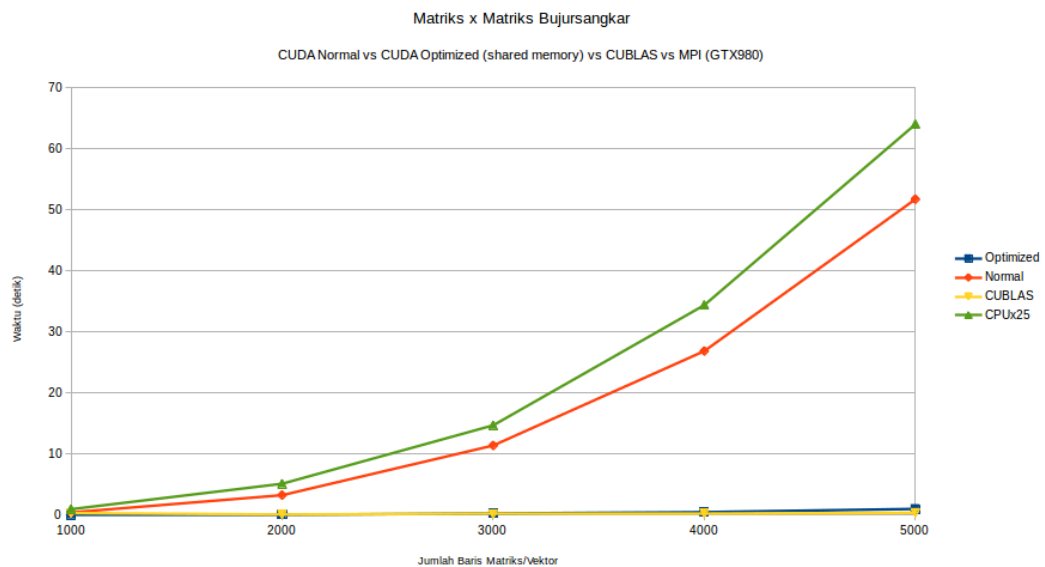
Gambar 3.9: Waktu eksekusi program perkalian matriks bujursangkar *shared memory* dan CUBLAS (940M)

Hasil yang berbeda ditunjukkan pada perkalian matriks bujursangkar pada GTX 970 dan GTX 980 (gambar 3.10 dan gambar 3.11), yaitu MPI (*cluster* 25 CPU)

adalah yang paling lambat dibanding yang lain. Setelah program MPI, urutan program yang terlambat hingga tercepat adalah CUDA dengan *global memory*, CUDA dengan *shared memory* dan CUBLAS.



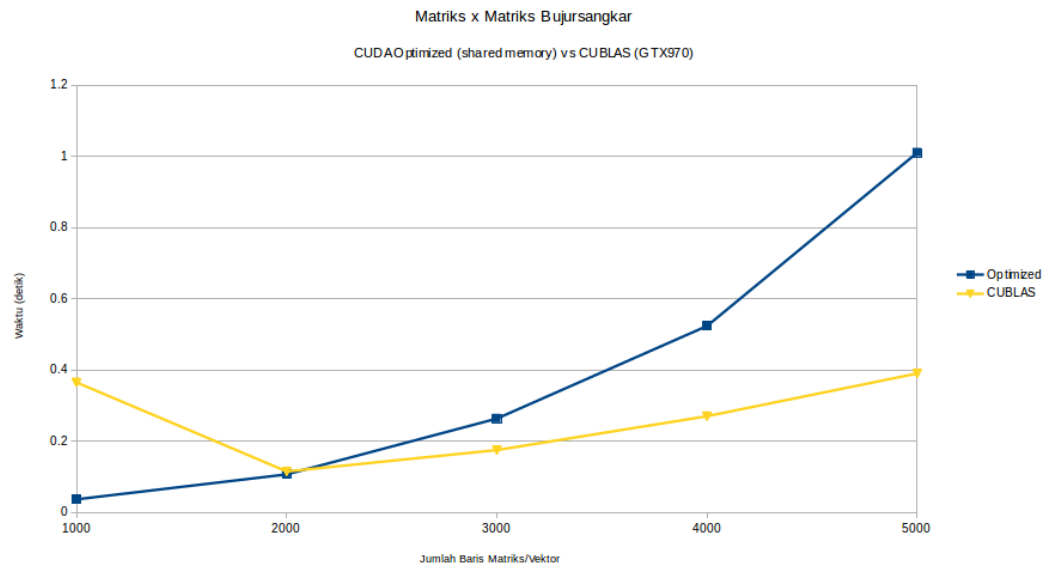
Gambar 3.10: Waktu eksekusi program perkalian matriks bujursangkar *global memory*, *shared memory*, CUBLAS dan MPI (GTX 970)



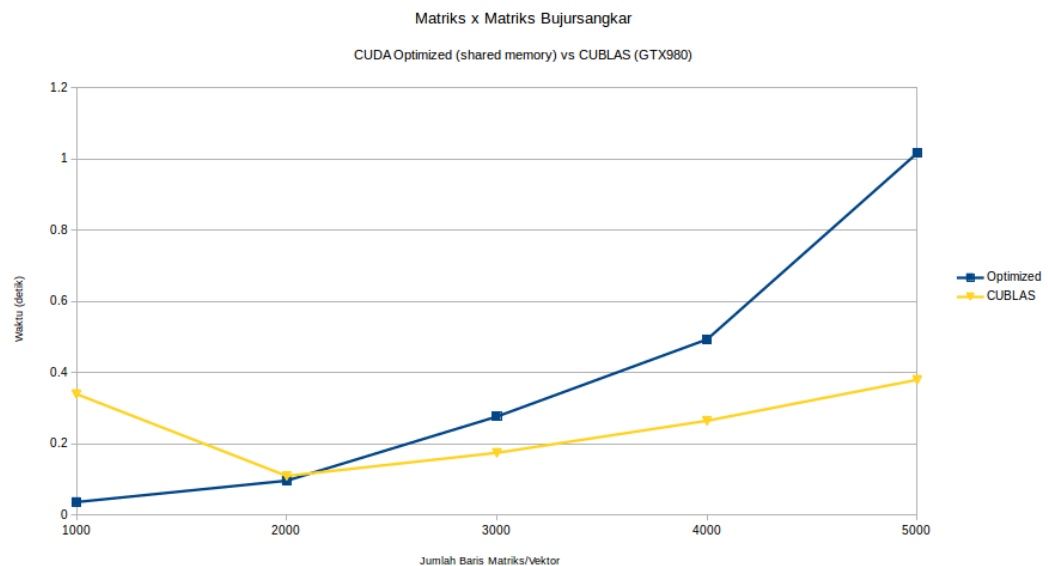
Gambar 3.11: Waktu eksekusi program perkalian matriks bujursangkar *global memory*, *shared memory*, CUBLAS dan MPI (GTX 980)

Hasil perbandingan waktu eksekusi perkalian matriks bujursangkar CUDA dengan *shared memory* dan CUBLAS pada GTX 970 dan GTX 980 (gambar 3.12 dan gambar 3.11) ternyata agak mirip dengan hasil pada 940M (gambar 3.9), yaitu

CUBLAS lebih cepat dari CUDA dengan *shared memory* pada ukuran matriks yang besar (dalam eksperimen ini lebih besar dari 2.000).



Gambar 3.12: Waktu eksekusi program perkalian matriks bujursangkar *shared memory* dan CUBLAS (970)



Gambar 3.13: Waktu eksekusi program perkalian matriks bujursangkar *shared memory* dan CUBLAS (980)

3.2 Kesimpulan

Berdasarkan hasil eksperimen di atas, kami menarik beberapa kesimpulan:

1. GPU (CUDA) lebih cepat dari CPU (baik sekuensial maupun paralel dengan MPI) untuk kasus perkalian matriks/vektor dengan banyak elemen (dan operasi)
2. Program CUDA dapat dioptimasi dengan menggunakan *shared memory* karena *shared memory* dapat diakses lebih cepat dari *global memory*
3. Dalam kasus dunia nyata, penggunaan library CUBLAS untuk melakukan operasi matriks/vektor lebih disarankan karena sudah dioptimasi khususnya untuk operasi terhadap data yang sangat besar
4. Jika menggunakan CUDA perlu diperhatikan konfigurasi ukuran *block* dan *grid* yang optimal untuk algoritma dan arsitektur GPU yang dipakai karena pada operasi perkalian matriks bujursangkar waktu proses akan lebih lambat jika konfigurasi tidak sesuai

BAB 4

CONJUGATE GRADIENT METHOD

4.1 Pendahuluan

Conjugate gradient method digunakan untuk menyelesaikan persamaan linear $Ax = b$ di mana matriks koefisiennya bersifat simetris dan definit positif. Matriks A $n \times n$ dikatakan simetris jika $a_{ij} = a_{ji}$ untuk $i, j = 1, \dots, n$. Matriks A dikatakan definit positif jika untuk setiap vektor x bukan nol, perkalian skalar $x \cdot Ax$ menghasilkan nilai lebih besar dari nol. Algoritma conjugate gradient method ditunjukkan pada Gambar 4.1. r_k merupakan sisa atau selisih antara b dengan Ax_k , sedangkan p_k merupakan *search direction*.

```

k = 0; x0 = 0; r0 = b
while (||rk||2 > tolerance) and (k < max_iter)
    k++
    if k = 1
        p1 = r0
    else
        βk =  $\frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}}$  // minimize ||pk - rk-1||
        pk = rk-1 + βkpk-1
    endif
    sk = Apk
    αk =  $\frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot s_k}$  // minimize q(xk-1 + αpk)
    xk = xk-1 + αkpk
    rk = rk-1 - αksk
endwhile
x = xk

```

Gambar 4.1: Algoritma Conjugate Gradient Method.

Program yang digunakan pada eksperimen ini merupakan program bawaan dari CUDA (NVIDIA). Sejak CUDA versi 7.0 NVIDIA menyertakan contoh program Conjugate Gradient yang menggunakan *library* CUBLAS dan CUSPARSE. Program dimodifikasi agar dapat menerima masukan berupa ukuran array yang digunakan dan menampilkan waktu yang diperlukan dalam 1 iterasi.

4.2 Eksperimen

Pada eksperimen 1, kode sumber hanya dimodifikasi agar dapat menerima argumen berupa ukuran array. Pada eksperimen 2 kode sumber dimodifikasi agar ukuran array mempengaruhi nilai A dan b. Berikut adalah potongan kode yang dimodifikasi.

```
void genTridiag(int *I, int *J, float *val, int N, int nz)
{
    I[0] = 0, J[0] = 0, J[1] = 1;
    val[0] = (float)rand()/RAND_MAX * N;
    val[1] = (float)rand()/RAND_MAX;
    int start;

    for (int i = 1; i < N; i++)
    {
        if (i > 1)
        {
            I[i] = I[i-1]+3;
        }
        else
        {
            I[1] = 2;
        }

        start = (i-1)*3 + 2;
        J[start] = i - 1;
        J[start+1] = i;

        if (i < N-1)
        {
            J[start+2] = i + 1;
        }

        val[start] = val[start-1];
        val[start+1] = (float)rand()/RAND_MAX * N;

        if (i < N-1)
        {
            val[start+2] = (float)rand()/RAND_MAX;
        }
    }

    I[N] = nz;
}
```

Gambar 4.2: Modifikasi nilai A

```
x = (float *)malloc(sizeof(float)*N);
rhs = (float *)malloc(sizeof(float)*N);

for (int i = 0; i < N; i++)
{
    rhs[i] = (float)rand()/RAND_MAX * N;
    x[i] = 0.0;
}
```

Gambar 4.3: Modifikasi nilai b

Eksperimen dilakukan dengan menggunakan variasi ukuran array 512, 1024, 2048, dan 4096. Gambar dibawah adalah contoh keluaran program.

```

Terminal
File Edit View Search Terminal Help
otniel.yosi@GTX780:~/samples/NVIDIA_CUDA-7.0_Samples/7_CUDALibraries/conjugateGr
adient$ ./conjugateGradient 4096
> GPU device has 13 Multi-Processors, SM 5.2 compute capabilities

version 82
create rhs & x
Get handle to the CUBLAS context
Get handle to the CUSPARSE context
CUSPARSE config
cuda malloc d_colcuda malloc d_rowcuda malloc d_valcuda malloc d_xcuda malloc d_
rcuda malloc d_pcuda malloc d_Axcuda memcpy d_col Jcuda memcpy d_row Icuda memcp
y d_val valcuda memcpy d_x xcuda memcpy d_r rhscuda cg preparationITER literatio
n = 1, residual = 2.789594e+00
ITER 2iteration = 2, residual = 2.090803e-01
ITER 3iteration = 3, residual = 1.748677e-02
ITER 4iteration = 4, residual = 1.564651e-03
ITER 5iteration = 5, residual = 1.420446e-04
ITER 6iteration = 6, residual = 1.380429e-05
ITER 7iteration = 7, residual = 1.290930e-06
Test Summary: Error amount = 0.000000
Time = 0.000000
otniel.yosi@GTX780:~/samples/NVIDIA_CUDA-7.0_Samples/7_CUDALibraries/conjugateGr
adient$

```

Gambar 4.4: Eksperimen 1 dengan ukuran array 4096

```

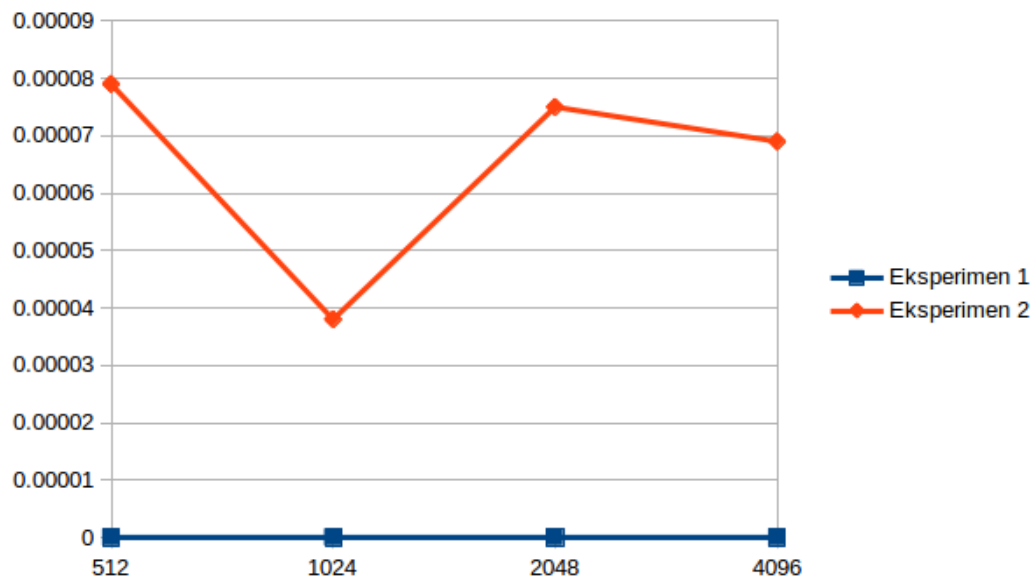
Terminal
File Edit View Search Terminal Help
ITER 559iteration = 559, residual = 5.459729e-05
ITER 560iteration = 560, residual = 5.063495e-05
ITER 561iteration = 561, residual = 4.847137e-05
ITER 562iteration = 562, residual = 4.555471e-05
ITER 563iteration = 563, residual = 4.237724e-05
ITER 564iteration = 564, residual = 3.929888e-05
ITER 565iteration = 565, residual = 3.580323e-05
ITER 566iteration = 566, residual = 3.247598e-05
ITER 567iteration = 567, residual = 2.970385e-05
ITER 568iteration = 568, residual = 2.787374e-05
ITER 569iteration = 569, residual = 2.550957e-05
ITER 570iteration = 570, residual = 2.246260e-05
ITER 571iteration = 571, residual = 2.001842e-05
ITER 572iteration = 572, residual = 1.799936e-05
ITER 573iteration = 573, residual = 1.657517e-05
ITER 574iteration = 574, residual = 1.520502e-05
ITER 575iteration = 575, residual = 1.378833e-05
ITER 576iteration = 576, residual = 1.223658e-05
ITER 577iteration = 577, residual = 1.058746e-05
ITER 578iteration = 578, residual = 9.330627e-06
Test Summary: Error amount = 0.008057
Time = 0.000069
otniel.yosi@GTX780:~/samples/NVIDIA_CUDA-7.0_Samples/7_CUDALibraries/conjugateGr
adient$

```

Gambar 4.5: Eksperimen 2 dengan ukuran array 4096

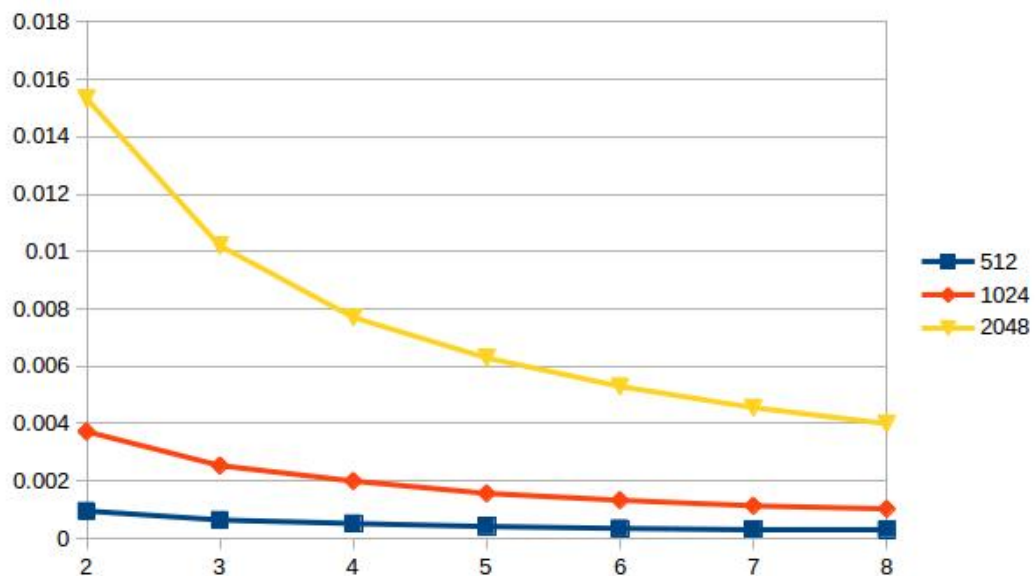
Gambar di bawah menampilkan grafik hasil eksperimen 1 dan 2. Terlihat bahwa nilai pada array mempengaruhi unjuk kerja. Namun demikian hasil keseluruhan

menunjukkan bahwa implementasi Conjugate Gradient yang digunakan sangatlah efisien.



Gambar 4.6: Unjuk kerja Conjugate Gradient dengan CUDA

Hasil eksperimen di atas jauh mengungguli unjuk kerja Conjugate Gradient dengan MPI. Hal ini menurut kami juga dipengaruhi oleh *library* CUBLAS dan CUSPARSE yang digunakan pada implementasi Conjugate Gradient dengan CUDA karena *library* tersebut merupakan *library* yang cukup matang untuk operasi matriks dan vektor.



Gambar 4.7: Unjuk kerja Conjugate Gradient dengan MPI pada cluster NBCR dengan variasi jumlah processor dan ukuran array

4.3 Kesimpulan

- GPU (CUDA) jauh lebih efisien dibanding cluster CPU (MPI) untuk menjalankan algoritma Conjugate Gradient Method.
- *Library* CUBLAS dan CUSPARSE sangat efektif untuk digunakan pada program yang membutuhkan operasi matriks dan vektor.

BAB 5

MOLECULAR DYNAMICS: AMBER

5.1 Pendahuluan

AMBER (*Assisted Model Building with Energy Refinement*) adalah paket program untuk menjalankan simulasi dinamika molekular (*molecular dynamics*) yang dikembangkan oleh *University of California, San Fransico*. Alamat *website* resmi dari AMBER adalah <http://ambermd.org>.

AMBER digunakan untuk berbagai eksperimen dinamika molekular seperti simulasi pergerakan fisik dari atom dan molekular, pemodelan protein dan eksperimen yang terkait dengan perancangan obat (*drug discovery*).

AMBER didistribusikan dalam dua bagian:

1. AMBER (berbayar, versi terakhir 14)
2. AMBERTool (*opensource* GPL, versi terakhir 15)

AMBER memiliki dua mode instalasi, yaitu berbasis CPU (dengan OpenMPI) dan berbasis GPU (dengan CUDA). Petunjuk instalasi dapat dilihat di <http://jswails.wikidot.com/installing-amber14-and-ambertools14>.

5.2 Eksperimen

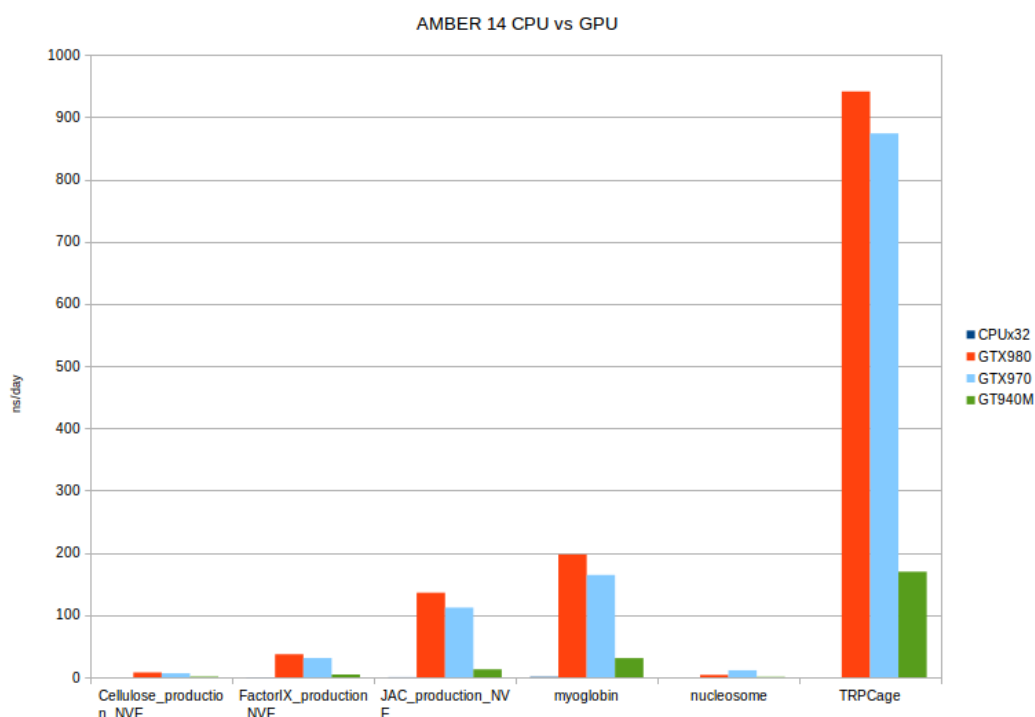
Percobaan dilakukan dengan menjalankan 6 buah eksperimen yang disediakan pada AMBER GPU *Benchmark Suite* yang diperoleh dari *website* resmi AMBER http://ambermd.org/Amber14_Benchmark_Suite.tar.bz2. Enam buah eksperimen yang kami jalankan adalah:

1. TRPCAGE Production (304 atoms, 1.000 nsteps)
2. Myoglobin Production (2,492 atoms, 25.000 nsteps)
3. JAC Production NVE (23,558 atoms, 1.000 nsteps)
4. Nucleosome Production (25,095 atoms, 200 nsteps)
5. Factor IX Production NVE (90,906 atoms, 10.000 nsteps)

6. Cellulose Production NVE (408,609 atoms, 10.000 nsteps)

Pada eksperimen ini kami mengamati kecepatan proses (ns/day) pada mesin yang berbeda, yaitu:

1. UCSD cluster 32 CPU + OpenMPI (nbc-233.ucsd.edu)
2. GTX 980 + CUDA 7.0 (Fasilkom UI)
3. GTX 970 + CUDA 7.0 (Fasilkom UI)
4. GT 940M + CUDA 7.5 (PC/notebook)



Gambar 5.1: Eksperimen AMBER pada GPU (CUDA) dan CPU *cluster* (MPI)

Hasil yang kami dapatkan adalah direpresentasikan oleh diagram pada gambar 5.1. Pada hasil eksperimen ini kinerja (ns/day) *cluster* CPU (MPI) sangatlah kecil jika dibandingkan dengan kinerja semua tipe GPU, yaitu selalu di bawah 1,0. Nilai kinerja GPU yang tercepat adalah GTX 980 diikuti GTX 970 yang sedikit di bawahnya. Kinerja GPU 940M cukup jauh di bawah kedua GPU lainnya tapi tetap masih 2-30 kali lebih cepat daripada *cluster* CPU.

Selain itu hasil eksperimen juga menunjukkan bahwa eksperimen dinamika molekular yang paling membutuhkan banyak sumber daya (ns/day terkecil) adalah Cellulose Production NVE dan Nucleosome Production yang memiliki rasio jumlah atom : nsteps terbesar.

5.3 Kesimpulan

Berdasarkan hasil eksperimen yang kami lakukan, kami mengambil kesimpulan sebagai berikut:

1. AMBER 14 berjalan jauh lebih efisien di GPU dibanding di CPU (MPI) bahkan sampai 200x lebih cepat (Myoglobin Production) bahkan ketika dibandingkan dengan 940M yang termasuk kelas *mainstream-mobile*.
2. AMBER 14 menggunakan kemampuan GPU dengan optimal sehingga bias terlihat jelas perbedaan kemampuan antar GPU. Dalam eksperimen ini 940M yang merupakan GPU kelas *mainstream-mobile* terlihat jelas kinerjanya jauh di bawah GTX 970 dan GTX 980 yang termasuk kelas *high-end-PC*.
3. Kebutuhan sumberdaya komputasi percobaan dinamika molekular tergantung pada banyaknya jumlah atom serta nsteps yang dilakukan. Semakin besar rasio jumlah atom : nsteps, maka semakin besar sumberdaya komputasi yang dibutuhkan dan semakin lama waktu proses yang dibutuhkan.

BAB 6

DAFTAR KONTRIBUSI ANGGOTA

Kontribusi tiap anggota kelompok pada tugas ini adalah sebagai berikut:

Otniel Yosi Viktorisa:

1. Eksperimen, presentasi & laporan topik 1 perilaku *block* dan *grid* pada CUDA
2. Eksperimen, presentasi & laporan topik 3 *Conjugate Gradient Method*

Yohanes Gultom:

1. Eksperimen, presentasi & laporan topik 2 perkalian matriks-vektor dan matriks bujursangkar dengan CPU (sekuensial), CUDA, CUDA dengan *shared memory*, CUBLAS dan CPU (*cluster MPI*)
2. Instalasi AMBER 14 dan AMBER Tools 15 pada server GTX 980 dan GTX 970 Fasilkom UI sebagai persiapan topik 4
3. Eksperimen, presentasi & laporan topik 4 AMBER dengan enam jenis *benchmarking set* dari *website* resmi AMBER pada 3 GPU dan 1 cluster (UCSD)

Muhammad Fathurachman:

1. Mengeksplorasi eksperimen topik 2 matriks-vektor dan matriks bujursangkar dengan CUDA