Analisis Implementasi CSR-*Adaptive* pada Perkalian Matriks Jarang dengan Vektor Menggunakan GPU - CUDA

Iksandi Lojaya[1] Fitriyani[2] Izzatul Ummah[3]

[1,2,3]Prodi Ilmu Komputasi Telkom University, Bandung
[1]iksandi@lojaya.com [2]fitriyani.y@gmail.com [3]izzatul.ummah@gmail.com

Abstrak

Perkalian antara matriks jarang dengan vektor merupakan *kernel* paling penting dalam pemrosesan matriks jarang. Sebelum diproses, matriks jarang umumnya akan disimpan menggunakan skema penyimpanan khusus, salah satunya adalah *Compressed Sparse Row* (CSR). Kemampuannya CSR untuk merepresentasikan segala jenis matriks jarang secara efisien membuat skema ini menjadi skema paling populer. Sayangnya, implementasi CSR menggunakan GPU menghasilkan kinerja yang kurang memuaskan dikarenakan akses memori tak berkesinambungan (*uncoalesced*) dan minimnya pemrosesan secara paralel. CSR-*Adaptive* adalah metode yang mampu menjadi jawaban atas kekurangan implementasi CSR terdahulu. Selagi dapat mengakses memori secara berkesinambungan, CSR-*Adaptive* juga dapat memaksimalkan pemrosesan secara paralel. Implementasinya memberikan rata-rata peningkatan kecepatan hingga 23.7 kali lipat dibandingkan dengan implementasi CSR terdahulu. Dalam penelitian ini akan dianalisis kinerja dari implementasi CSR-*Adaptive* yang memanfaatkan teknologi CUDA pada GPU.

Kata kunci : CSR-*Adaptive*, Matriks Jarang, CUDA, *coalesced*

Abstract

Sparse matrix-vector multiplication is the most important kernel in sparse matrix processing. Before processed, normally sparse matrix will be stored in a special storage scheme and one of them is Compressed Sparse Row. This storage sceme is able to represents a wide variety of sparse matrices efficiently that it has become the most popular storage scheme. Unfortunately, CSR implementation gives a poor performance on GPU because of uncoalesced memory accesses and lack of paralelism. CSR-Adaptive is a method that could be an answer to the disadvantage of the previous CSR implementations. Whilst able to perform a coalesced memory accesses, CSR-Adaptive also able to maximize the parallel processing. The implementation achieves average speedup of 23.7x over the previous CSR implementations. This research will analyze the performance of CUDA-based CSR-Adaptive implementation on GPU.

Keywords: CSR-*Adaptive*, *Sparse Matrix*, CUDA, *coalesced*

1. Pendahuluan

Operasi yang sering melibatkan matriks jarang sebagai komponen utamanya adalah operasi perkalian antara matriks jarang dengan vektor (SpMV / *sparse matrix-vector multiplication*) [7]. Sebelum diproses, matriks jarang secara lazim disimpan menggunakan suatu skema penyimpanan khusus. Skema penyimpanan ini memanfaatkan keunikan karakteristik matriks dengan cara tidak mengikutsertakan elemen bernilai nol. Salah satu skema penyimpanan tersebut diantaranya adalah CSR (*Compressed Sparse Row*).

Pada penelitian sebelumnya, CSR disimpulkan sebagai sebuah skema penyimpanan matriks jarang yang dapat secara baik merepresentasikan semua bentuk matriks jarang [5].

Pada penelitian berikutnya, telah dilakukan implementasi operasi SpMV pada berbagai macam skema penyimpanan menggunakan GPU (*Graphics Processing Unit*) berteknologi CUDA (*Compute Unified Device Architecture*). Penelitian tersebut memberikan dua teknik operasi SpMV yang memanfaatkan skema penyimpanan CSR, yaitu CSR-*Scalar* dan CSR-*Vector* [7].

CSR-*Scalar* yang merupakan implementasi naïve secara umum memberikan performa buruk disebabkan oleh akses memori yang tidak berkesinambungan (*uncoalesced*). Di lain pihak, CSR-*Vector*, walaupun berhasil memperbaiki kekurangan CSR-*Scalar*, tidak dapat memanfaatkan kelebihan GPU ketika berhadapan dengan matriks yang memiliki sangat sedikit elemen tak nol setiap barisnya [7].

Baru-baru ini muncul sebuah teknik baru untuk mengoperasikan SpMV yang menggunakan skema penyimpanan CSR bernama CSR-*Adaptive*. Teknik ini memberikan rata-rata peningkatan kecepatan hingga 14.7x dibandingkan dengan teknik berbasis CSR sebelumnya [4]. Penelitian ini akan melakukan analisis terhadap implementasi CSR-*Adaptive* pada operasi SpMV dengan memanfaatkan teknologi CUDA pada GPU NVIDIA.

2. Tinjauan Pustaka

2.1 Matriks

Matriks merupakan *array* dua dimensi yang terdiri atas objek-objek matematika yang dapat berupa angka, simbol, atau ekspresi. Dalam matriks,

objek-objek ini disebut sebagai elemen dan diatur oleh baris dan kolom. Matriks umumnya dinotasikan menggunakan huruf besar tebal, sementara itu untuk setiap elemen didalamnya dinotasikan menggunakan huruf kecil yang memiliki dua buah *subscript* (contoh: a_{12} atau $a_{1,2}$) diletakan setelahnya sebagai penanda posisi elemen pada matriks. *Subscript* pertama menandakan posisi elemen pada baris, sementara *subscript* kedua menandakan posisi kolom. Matriks secara konvensional mendahulukan baris daripada kolom, sehingga apabila diketahui ada matriks sebesar $m\ x\ n$, berarti matriks tersebut memiliki n jumlah baris dan n jumlah kolom. [8]. Misalkan diberikan matriks sebagai berikut:

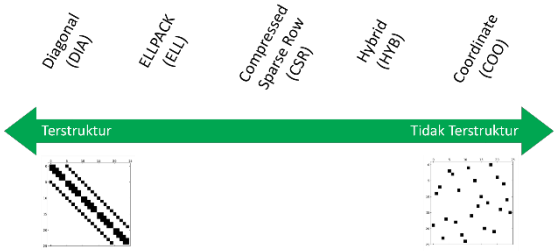
$$\boldsymbol{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

Matriks **A** merupakan matriks yang terdiri dari empat baris dan empat kolom dengan total 16 elemen.

2.2 Matriks Jarang

Secara umum, definisi dari matriks jarang adalah matriks yang mayoritas elemennya memuat nilai nol. Matriks jarang secara luas dipakai dalam berbagai bidang matematika seperti aljabar linear, data mining, dan analisis graf [2]. Acap kali peneliti menggunakan matriks jarang berukuran sangat besar untuk menyelesaikan suatu permasalahan, seperti menyelesaikan persamaan diferensial parsial [9].

Sebelum matriks jarang diproses oleh komputer, biasanya matriks ini akan disimpan menggunakan skema penyimpanan khusus. Skema penyimpanan ini memanfaatkan fakta bahwa matriks jarang memiliki banyak elemen bernilai nol, sehingga ukuran matriks dapat dikurangi dengan cara tidak mengikutsertakan elemen tersebut. Selain itu skema penyimpanan ini dapat menghindari pemborosan memori akibat pengolahan nilai nol.



Gambar 2.1. Komparasi skema penyimpanan

Setiap skema penyimpanan dapat secara baik merepresentasikan matriks jarang berdasarkan distribusi atau struktur elemen bernilai tak nol pada matriks jarang tersebut. Menurut Bell (2009) pola tersebut dapat dibagi menjadi tiga, yaitu: matriks diagonal, matriks dengan jumlah nilai tak nol (NNZ) per baris yang kurang lebih seragam, dan matriks dengan NNZ per baris yang tidak seragam [7].

2.2.1 *Compressed Sparse Row*

Skema penyimpanan *compressed sparse row* (CSR) merupakan skema paling populer dan sering digunakan untuk merepresentasikan matriks jarang. Skema CSR tersusun oleh tiga buah *array*.

Dua *array* pertama adalah *array data* dan *indices* yang sama seperti *array data* dan *col* pada skema COO. *Array* ketiga merupakan *ptr*, yaitu *array* penunjuk baris (*offset*) sebesar $M + 1$ untuk matriks jarang sebesar $M\ x\ N$. Elemen pertama pada *array ptr* akan selalu bernilai 0 dan elemen terakhir merupakan NNZ total pada matriks jarang [7].

Berikut adalah contoh representasi matriks **A** pada sub bab 2.1. dengan menggunakan skema CSR:

$$\begin{aligned} \textbf{data} &= [1\ \ 7\ \ 2\ \ 8\ \ 5\ \ 3\ \ 9\ \ 6\ \ 4] \\ \textbf{indices} &= [0\ \ 1\ \ 1\ \ 2\ \ 0\ \ 2\ \ 3\ \ 1\ \ 3] \\ \textbf{ptr} &= [0\ \ 2\ \ 4\ \ 7\ \ 9] \end{aligned}$$

Skema CSR dapat dikatakan sebagai bentuk yang lebih ringkas daripada skema COO, sehingga selagi skema CSR memberikan keuntungan yang sama seperti skema COO, skema ini juga dapat merepresentasikan matriks jarang berpola secara efisien.

2.3 Perkalian Matriks Jarang dengan Vektor (SpMV)

SpMV merupakan *kernel* yang paling sering digunakan dalam mengolah data yang berkaitan dengan matriks jarang. Formula SpMV dapat dituliskan sebagai $y = Ax$. Input A merupakan matriks jarang berukuran $d\ x\ d$, sedangkan input x dan output y merupakan vektor kolom padat (dense) berukuran $d\ x\ 1$.

Berikut ini adalah algoritma operasi SpMV secara umum [6]:

```
input:   A {matriks jarang berukuran d x d}
         x {vektor padat berukuran d x 1}
output:  y ← A.x
1  for r = 0 to d − 1 do
2      y[r] ← 0
3      foreach entri tidak nol pada (baris r,
       kolom c) do
4          y[r] = y[r] + A[r][c] * x[c]
5      endfor
6  endfor
```

Algoritma SpMV secara umum

2.4 GPU

GPU (*Graphics Processing Unit*) merupakan salah satu komponen perangkat komputer yang secara lazim digunakan untuk keperluan pengolahan grafis. Dalam dekade terakhir ini, GPU telah berevolusi dari yang awalnya hanya digunakan untuk keperluan grafis, menjadi alat yang juga dapat dimanfaatkan untuk mengakselerasi proses komputasi. Salah satu pioner dalam pengembangan GPU adalah NVIDIA,

merekalah yang pertama kali memperkenalkan CUDA sebagai platform untuk mengakses GPU yang dapat dimanfaatkan sebagai alat untuk mempercepat proses komputasi [1].

2.4.1 CUDA

CUDA (*Compute Unified Device Architecture*) merupakan sebuah *platform* komputasi paralel dan *application programming interface* (API) yang dikembangkan oleh NVIDIA. CUDA didesain untuk dapat bekerja di berbagai bahasa pemrograman seperti C, C++ dan Fortran. CUDA memungkinkan pengembang perangkat lunak untuk memanfaatkan GPU berteknologi CUDA agar dapat mengakselerasi proses komputasi [11].

Cara kerja CUDA dapat dibilang cukup sederhana. Data dari memori utama akan disalin ke memori GPU (*global memory*), kemudian instruksi yang harus dilaksanakan oleh GPU dikerjakan secara paralel. Setelah GPU selesai memproses instruksi, data hasil akan disalin kembali dari *global memory* ke memori utama.

Eksekusi program CUDA mengharuskan pengguna untuk meng-input jumlah *block* dan *thread*. Setiap *block* berisi sejumlah *thread*, banyaknya *thread* sesuai dengan apa yang pengguna input. Selanjutnya, GPU NVIDIA akan membagi *block* secara merata kepada setiap *streaming multiprocessors* (SM). SM kemudian akan menjadwalkan dan mengeksekusi sekelompok *thread* (*warp*) pada *block-block* tersebut.

2.4.2 *Bandwidth*

Bandwidth merupakan lebar jalur memori yang dapat dipakai oleh pemroses untuk melakukan transfer data. Dalam pemrograman GPU besarnya penggunaan *bandwidth* adalah parameter yang menentukan seberapa baik dan optimal suatu *thread* melakukan transfer data. Formulasi umum dalam menghitung penggunaan *bandwidth* pada suatu *kernel* GPU adalah sebagai berikut:

$$BW = \frac{\left(\text{readwrite x sizeof(tipedata)}\right) \text{ x } 10^{-9}}{t \text{ x } 10^{-3}} \ldots (1)$$

Penggunaan *bandwidth* dihitung dari seberapa banyak suatu *kernel* melakukan operasi *read*/*write* pada *global memory*. Jumlah tersebut kemudian dikalikan dengan besarnya ukuran tipe data dalam *byte*. *Bandwidth* umumnya memakai satuan GB/s, sehingga hasil perkalian sebelumnya dikalikan dengan 10^{-9} untuk mendapatkan ukuran total data yang dibaca/ditulis pada *global memory* dalam gigabyte.

Ukuran total data tersebut kemudian dibagi dengan waktu yang diperlukan bagi *kernel* untuk menyelesaikan operasi. Satuan waktu yng dipakai oleh GPU adalah *milisecond*, sehingga harus dikalikan terlebih dahulu dengan 10^{-3} untuk mengubahnya menjadi *second*.

2.4.3 FLOPS

FLOPS (*Floating-Point Operation Per Second*) merupakan ukuran kualitas kinerja komputer untuk menghitung berapa banyak operasi floating-point yang dapat dilakukan per detiknya. Saat ini, standar pengukuran untuk GPU merupakan GFLOPS yang berarti satu miliar operasi *floating-point* per detik [10].

2.5 CSR-Adaptive

Selagi menutupi kekurangan CSR-*Vector*, Algoritma CSR-*Adaptive* juga tetap mempertahankan pola akses memori yang berkesinambungan. Secara garis besar, CSR-*Adaptive* menambah satu komponen lagi bernama array `row_blocks` pada CSR. Komponen ini berisi *offset* blok kumpulan baris yang dibatasi oleh sejumlah NNZ atau `LOCAL_SIZE`.

```
input:    totalRows, ptr[]
output:   row_blocks[]
1   row_blocks[0] ← 0; last_i ← 0; ctr ← 0;
2   for i = 1 to totalRows − 1 do
        //hitung NNZ pada row i
3       sum += ptr[i] − ptr[i − 1]
        //jika NNZ = LOCAL_SIZE
4       if sum == LOCAL_SIZE then
5           last_i ← i; row_blocks[ctr++] ← i; sum ← 0;
6       else if
7           if i − last_i > 1 then
8               row_blocks[ctr++] ← (i − 1); i−−;
                //Jika baris ekstra tidak muat
9           else if i − last_i == 1 then
                row_blocks[ctr++] ← i;
10              //jika NNZ pada baris terlalu
                besar
11          end
12          last_i ← i; sum ← 0;
13      end
14  end
15  row_blocks[ctr++] ←totalRows;
```

Algoritma untuk membangung array `row_blocks`

Algoritma CSR-*Adaptive* bekerja dengan cara menempatkan *thread* ke tiap blok yang berisi sejumlah baris dengan total jumlah NNZ kurang lebih sebesar `LOCAL_SIZE`. Nilai `LOCAL_SIZE` dapat diisi berapapun sesuai keinginan pemrogram. Berdasarkan penelitian sebelumnya, algoritma ini memberikan kinerja terbaik untuk variabel `LOCAL_SIZE` bernilai 1024 [4].

Algoritma CSR-*Adaptive* memiliki dua kondisi, yaitu kondisi dimana satu blok baris berisi lebih dari satu baris dan kondisi dimana satu blok baris hanya berisi satu baris akibat NNZ-nya lebih besar dari `LOCAL_SIZE`. Kondisi pertama akan diproses menggunakan algoritma CSR-*Stream*, sedangkan kondisi kedua akan diproses menggunakan algoritma CSR-*Vector* [4].

2.5.1 CSR-*Stream*

Algoritma CSR-*Stream* merupakan bagian dari algoritma CSR-*Adaptive*. Algoritma ini dijalankan apabila terdapat dua atau lebih baris dalam satu blok baris. Pada implementasinya, CSR-*Stream* terlebih dahulu akan menyalin dan memproses nilai tak nol yang terdapat pada blok baris ke dalam *shared memory* secara berkesinambungan.

Selanjutnya, akan dilakukan reduksi terhadap nilai-nilai tersebut menggunakan satu *thread* per baris seperti CSR-*Scalar*. Pada tahap reduksi tersebut, walaupun pola akses memori tidak berkesinambungan, namun akses data pada memori akan lebih cepat karena data diakses dari *shared memory*, bukan dari *global memory*.

```
1  startRows ← row_blocks[workgroupID];
2  nextStartRows ← row_blocks[workgroupID +
   1];
3  NNZ ← ptr[nextStartRows] -
   ptr[startRows];
4  {stream NNZ values into shared memory}
5  num_rows ← nextStartRows - startRows;
6  thread_start_point ← ptr[startRows +
   localTID];
7  thread_end_point ← ptr[startRows +
   localTID + 1];
8  {perform reduction on num_rows, rather
   than static number}
9  while rows_done < num_rows do
       {scalar reduction out of the shared
10     memory for values between
       thread_end_point and
       thread_start_point}
11     rows_done += wordgroupSize;
12 end
13 output[startRow + localTID] ← temp;
```

Algoritma CSR-*Stream*

Teknik ini memaksimalkan pemrosesan secara paralel selagi menjaga agar pola akses memori tetap berkesinambungan. Pada kondisi pertama, terkadang terdapat situasi dimana blok baris berisi lebih dari satu baris namun hanya sedikit, misal hanya berisi 4-6 baris. Untuk situasi tersebut, reduksi secara serial tidak cocok digunakan karena akan terdapat banyak *thread* berstatus *idle*.

Maka dari itu di dalam CSR-*Stream* harus diberikan lagi 2 kondisi untuk menentukan apakah blok baris tersebut sebaiknya direduksi secara serial atau paralel. Kondisi pertama dilakukan apabila baris yang terdapat pada blok baris berjumlah banyak, sebaliknya apabila baris pada blok baris sedikit, kondisi kedua akan dieksekusi.

Dalam penelitian tugas akhir ini, algoritma CSR-*Stream* akan dimodifikasi untuk mengisi kelemahan seperti yang telah dijelaskan pada paragraf sebelumnya. parameter yang digunakan untuk menentukan apakah suatu blok akan direduksi secara serial atau paralel adalah jumlah *thread* pada *warp*. Lebih lengkapnya, algortima CSR-*Adaptive* yang akan digunakan adalah sebagai berikut:

```
1  if(nnz_per_row < warp_size)
2   | serial reduction per row
3  elseif(num_rows > 1)
4   | paralel reduction per row
5  else
6   | paralel reduction
7  end
```

Algoritma CSR-*Adaptive*

3. Perancangan Sistem

Dalam penelitian ini akan dibuat sebuah sistem yang memanfaatkan teknologi CUDA pada GPU NVIDIA untuk melakukan operasi SpMV berbasis CSR. Sistem yang dibuat merupakan implementasi algoritma yang dapat diterapkan secara paralel pada GPU menggunakan teknologi CUDA agar didapat peningkatan kecepatan waktu eksekusi dalam memproses data.

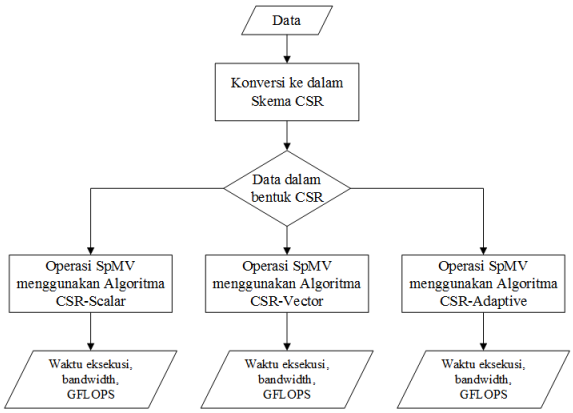
Sistem juga akan dibandingkan terhadap implementasi berbasis CSR sebelumnya untuk memeriksa apakah ada peningkatan performa atau tidak.

Hasil dari sistem ini adalah informasi berupa waktu eksekusi, penggunaan *bandwidth*, dan jumlah GLOPS.

3.1 Rancangan Sistem

Sistem yang dibangun terdiri atas dua tahapan. Tahap pertama adalah melakukan konversi data matriks jarang ke dalam bentuk skema CSR. Tahap kedua adalah melakukan operasi SpMV terhadap data matriks jarang yang telah dikonversi ke dalam bentuk CSR.

Operasi dilakukan menggunakan algoritma CSR-*Scalar*, CSR-*Vector*, dan CSR-*Adaptive* secara paralel menggunakan GPU NVIDIA. Berikut ini merupakan alur kerja dari sistem yang akan dibangun.



Gambar 3.1. Komparasi skema penyimpanan

Pada tahapan pertama, sistem akan menkonversi data matriks jarang ke dalam bentuk CSR. Konversi data akan dilakukan di dalam lingkup CPU dan hasilnya akan disimpan ke dalam memori utama. Selanjutnya, sistem akan menyalin data pada memori utama ke dalam memori GPU.

Kemudian sistem akan mengeksekusi operasi SpMV sesuai masing-masing algoritma lalu menyimpan informasi berupa waktu eksekusi, penggunaan *bandwidth*, dan jumlah GLOPS. Informasi ini kemudian akan ditampilkan ke layar komputer dan disimpan untuk dianalisis.

3.2 Data

Data matriks jarang diperoleh dari koleksi matriks jarang *University of Florida* [3]. Data yang diambil merupakan data berformat *matrix market* yang merupakan representasi skema COO untuk matriks jarang terkait.

Akan diambil 19 data matriks jarang dengan tingkat distribusi nilai tak nol dan dimensi yang beragam untuk diproses menggunakan sistem yang akan dibangun.

Nama	Ukuran	NNZ	NNZ/Baris
Dense2	2K * 2K	4 ,000,000	2,000
Protein	36K * 36K	4,344,765	119
FEM/Sphere	83K * 83K	6,010,480	72
FEM/Cantilever	62K * 62K	4,007,383	64
Wind Tunnel	218K * 218K	11,634,424	53
FEM/Harbor	47K * 47K	2,374,001	51
FEM/Ship	141K * 141K	7,813,404	55
Economics	207K * 207K	1,273,389	6
Epidemiology	526K * 526K	2,100,225	4
FEM/Accelerator	121K * 121K	2,624,331	22
Circuit	171K * 171K	958,936	6
Webbase	1M * 1M	3,105,536	3
LP	4K * 1097K	11,284,032	2634
cicrcuit5M	5.5M * 5.5M	59,524,291	11
eu-2005	863K * 863K	19,235,140	22
Ga41As41H72	268K * 268K	18,488,476	69
in-2004	1.3M * 1.3M	16,917,053	12
mip1	66K * 66K	10,352,819	156
Si41Ge41H72	186K * 186K	15,011,265	81

Tabel 3.1. Dataset matriks yang akan digunakan

3.3 Skenario Pengujian Sistem

Skenario yang digunakan untuk pengujian sistem adalah:

a) Melakukan input data matriks jarang yang akan diproses beserta jumlah *block*, *thread*, dan iterasi.
b) Operasi SpMV akan dilakukan terhadap data matriks yang telah diinput dengan matriks sebesar mx 1 bernilai 1 menggunakan masing-masing algoritma CSR yang diuji.
c) Akan dihitung waktu eksekusi, penggunaan *bandwidth*, dan jumlah GFLOPS untuk tiap algoritma.

3.4 Perangkat Keras yang Digunakan

Perangkat keras yang digunakan untuk menjalankan implementasi ini adalah komputer dengan spesifikasi sebagai berikut:

- CPU Intel Core i7-3770 (4x2 CPU) @ 3.40GHz
- 7 GB RAM
- 120 GB Hard Disk

Komputer dilengkapi GPU NVIDIA GeForce GTX 770 dengan spesifikasi sebagai berikut:

Streaming Multiprocessors (SM)	8
CUDA Processors	1536
Core Clock Rate	1046 MHz
Memory Size	2048 MB
Memory Clock Rate	1752 MHz
Peak Memory Bandwidth	224.26 GB/s
Shared Memory per SM	48 KB
Single Precision Peak Performance	3213 GFLOPS
Double Precision Peak Performance	134 GFLOPS

Tabel 3.2. Spesifikasi NVIDIA GeForce GTX 770

Sistem akan dijalankan pada sistem operasi CentOS 6.5, di-*compile* menggunakan NVCC versi 7.5.17.

4. Pengujian dan Analisis

4.1 Implementasi Sitem

Algoritma akan dieksekusi menggunakan kombinasi jumlah *block* dan *thread* sebanyak 10 kali. Algoritma dieksekusi sebanyak 10 kali untuk mendapatkan rata-rata waktu eksekusi tiap algoritma terkait.

Jumlah *block* yang digunakan berkisar antara 512 – 16384 dan jumah *thread* yang digunakan berkisar antara 128 – 1024 untuk setiap konfigurasi *block*. Nilai GFLOPS hasil eksekusi akan dirata-ratakan untuk mendapatkan konfigurasi jumlah *block* dan *thread* terbaik.

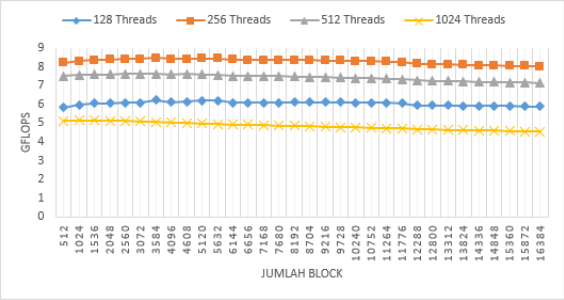
Setelah konfigurasi jumlah *block* dan *thread* terbaik diperoleh, algoritma pembanding (CSR-*Scalar* dan CSR-*Vector*) akan dieksekusi menggunakan konfigurasi tersebut. Hasil eksekusi tersebut, bersama dengan hasil eksekusi algoritma CSR-*Adaptive* sebelumnya, akan dibandingkan dan dianalisis kinerjanya.

4.1.1 Menentukan Jumlah *Block* dan *Thread* Terbaik

Jumlah *block* yang akan digunakan merupakan kelipatan dari 512 dengan nilai maksimum 16384, hal tersebut menghasilkan 32 konfigurasi jumlah *block*. Di lain pihak, jumlah *thread* yang akan digunakan adalah 128, 256, 512, dan 1024.

Total akan diperoleh 128 output untuk setiap data matriks. Selanjutnya, data GFLOPS pada setiap output masing-masing matriks tersebut akan dijumlahkan dan dicari rata-ratanya, sesuai konfigurasi jumlah *block* dan *thread* yang digunakan.

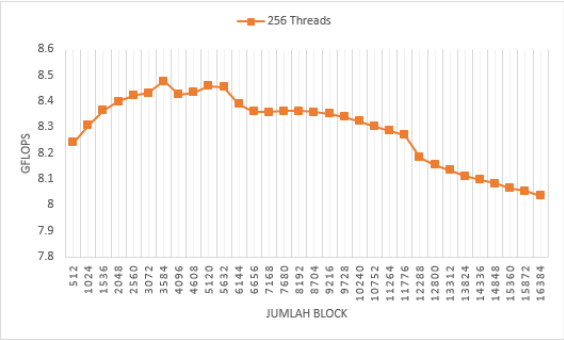
Langkah berikutnya, berdasarkan data yang telah diproses akan dipilih konfigurasi jumlah *thread* terbaik. Dibawah ini adalah grafik yang menampilkan kinerja tiap jumlah *thread*.



Gambar 4.1. Grafik perbandingan kinerja tiap konfigurasi thread

Pada grafik tersebut, dapat disimpulkan bahwa dari sisi jumlah GFLOPS, *thread* sejumlah 256 memiliki kinerja yang sangat baik dibanding konfigurasi jumlah *thread* lainnya. Maka dari itu, nilai 256 akan dipilih sebagai konfigurasi jumlah *thread* terbaik.

Selanjutnya akan dicari konfigurasi jumlah *block* terbaik. Berikut ini adalah grafik perbandingan kinerja tiap *block* pada konfigurasi jumlah *thread* sebanyak 256:



Gambar 4.2. Grafik perbandingan kinerja tiap konfigurasi block untuk thread sebanyak 256

Dari grafik diatas, dapat dilihat dari sisi jumlah GFLOPS yang dihasilkan bahwa *block* dengan jumlah sekitar 3584 memiliki kinerja paling baik dibandingkan kinerja konfigurasi jumlah *block* lainnya. Jumlah *block* yang lebih besar dari 3584 membuat kinerja kian menurun, hal tersebut disebabkan terjadinya latensi akibat proses yang mubazir pada GPU.

Di lain pihak, jumlah *block* yang lebih kecil dari 3584 tidak memaksimalkan paralelisasi sehingga waktu yang diperlukan bagi GPU untuk menyelesaikan eksekusi program menjadi lama, akibatnya jumlah GFLOPS yang dihasilkan menjadi tidak maksimal.

Dari eksperimen diatas, diperoleh bahwa CSR-*Adaptive* memberikan performa terbaik untuk konfigurasi jumlah *block* sebanyak 3584 dan jumlah *thread* sebanyak 256.

4.2 Hasil Analisis

Setiap algoritma akan diuji menggunakan konfigurasi jumlah *thread* sebanyak 256, jumlah *block* sebanyak 3584, dan iterasi sebanyak 10 kali terhadap 19 data matriks jarang. Pengujian tersebut akan memberikan output berupa waktu eksekusi, penggunaan *bandwidth*, dan jumlah GFLOPS.

Berikut adalah tabel yang memuat hasil eksekusi setiap algoritma pada tiap data matriks jarang:

Matrix	GFLOP/s		
	Scalar	Vector	Adaptive
Dense 4K	1.46	4.46	4.83
Protein	1.09	9.53	10.91
FEM/Spheres	1.09	9.48	9.49
FEM/Cantilever	1.13	9.23	9.43
Wind Tunnel	1.15	9.65	8.92
FEM/Harbor	1.12	8.92	8.83
FEM/Ship	1.11	8.97	8.90
Economy	2.47	7.41	9.37
Epidemiology	4.94	7.10	14.05
FEM/Accelerator	1.11	7.63	6.27
Circuit	2.05	7.29	9.77
Webbase	1.11	6.92	9.47
LP	0.38	1.92	2.84
circuit5M	0.15	4.90	7.81
eu-2005	1.19	8.28	7.79
Ga41As41H72	1.03	6.14	7.35
in-2004	1.27	8.29	8.76
mip1	0.37	7.02	8.61
Si41Ge41H72	1.03	6.19	7.54
Rata-rata	1.33	7.33	8.47

Tabel 4.1. Jumlah GFLOPS masing-masing algoritma

Matrix	Time		
	Scalar	Vector	Adaptive
Dense 4K	0.0055	0.0009	0.0009
Protein	0.0080	0.0006	0.0008
FEM/Spheres	0.0111	0.0009	0.0013
FEM/Cantilever	0.0071	0.0006	0.0008
Wind Tunnel	0.0202	0.0019	0.0026
FEM/Harbor	0.0042	0.0004	0.0005
FEM/Ship	0.0141	0.0014	0.0017
Economy	0.0010	0.0010	0.0003
Epidemiology	0.0009	0.0026	0.0003
FEM/Accelerator	0.0047	0.0008	0.0008
Circuit	0.0009	0.0009	0.0002
Webbase	0.0056	0.0049	0.0006
LP	0.0593	0.0059	0.0044
circuit5M	0.8203	0.0473	0.0131
eu-2005	0.0323	0.0056	0.0050
Ga41As41H72	0.0359	0.0044	0.0050
in-2004	0.0266	0.0072	0.0039
mip1	0.0555	0.0018	0.0019
Si41Ge41H72	0.0291	0.0034	0.0039
Rata-rata	0.0601	0.0049	0.0025

Tabel 4.2. Durasi waktu eksekusi masing-masing algoritma

Matrix	Bandwidth (GB/s)		
	Scalar	Vector	Adaptive
Dense 4K	29.16	105.43	102.84
Protein	21.91	181.59	155.50
FEM/Spheres	21.93	159.11	146.76
FEM/Cantilever	22.86	149.40	152.50
Wind Tunnel	23.30	146.52	151.95
FEM/Harbor	22.72	132.87	144.12
FEM/Ship	22.47	138.06	149.93
Economy	54.24	29.49	130.81
Epidemiology	113.64	19.46	211.03
FEM/Accelerator	22.74	75.29	91.83
Circuit	45.29	26.82	138.30
Webbase	26.42	15.12	156.05
LP	7.61	45.55	62.42

circuit5M	3.07	30.22	132.49
eu-2005	24.50	83.11	114.25
Ga41As41H72	20.80	101.64	113.45
in-2004	26.65	56.30	121.94
mip1	7.49	140.57	136.92
Si41Ge41H72	20.81	107.47	112.93
Rata-rata	28.30	91.79	132.95

Tabel 4.3. Penggunaan *bandwidth* masing-masing algoritma

Dari tabel diatas, dapat dilihat bahwa algoritma CSR-*Adaptive* memiliki rata-rata kinerja yang lebih baik dibanding implementasi algoritma CSR lainnya. CSR-*Adaptive* mampu memberikan rata-rata peningkatan jumlah GFLOPS hingga 6.4x lipat, rata-rata peningkatan penggunaan *bandwidth* hingga 4.7x lipat, dan rata-rata peningkatan kecepatan hingga 23.7x lipat dari semua dataset yang diuji dibandingkan dengan implementasi CSR terdahulu (CSR-*Scalar*).

Untuk beberapa dataset, terlihat bahwa kinerja CSR-*Vector* sedikit lebih baik daripada CSR-*Adaptive*. Hal tersebut dikarenakan terdapatnya nilai ekstrim pada nilai NNZ per baris pada blok baris tertentu sehingga kinerja CSR-*Stream* merosot akibat kurangnya pemrosesan secara paralel.

Di lain pihak, untuk beberapa dataset, kinerja CSR-*Adaptive* melambung tinggi mengalahkan kinerja CSR lainnya. Hal tersebut diakibatkan variansi nilai NNZ per baris pada setiap blok baris relatif kecil, sehingga GPU dapat memaksimalkan paralelisasi.

Matrix	GFLOP/s	
	Original	Modifikasi
Dense 4K	4.89	4.83
Protein	7.49	10.91
FEM/Spheres	10.05	9.49
FEM/Cantilever	9.96	9.43
Wind Tunnel	10.62	8.92
FEM/Harbor	8.94	8.83
FEM/Ship	9.17	8.90
Economy	9.66	9.37
Epidemiology	14.28	14.05
FEM/Accelerator	9.43	6.27
Circuit	9.96	9.77
Webbase	9.27	9.47
LP	2.86	2.84
circuit5M	7.79	7.81
eu-2005	8.91	7.79
Ga41As41H72	5.02	7.35
in-2004	7.78	8.76
mip1	7.30	8.61
Si41Ge41H72	4.74	7.54
Rata-rata	8.32	8.47

Tabel 4.4. Jumlah GFLOPS algoritma CSR-Adaptive original dan modifikasi

Matrix	Time	
	Original	Modifikasi
Dense 4K	0.0009	0.0009
Protein	0.0012	0.0008
FEM/Spheres	0.0012	0.0013
FEM/Cantilever	0.0008	0.0008
Wind Tunnel	0.0022	0.0026
FEM/Harbor	0.0005	0.0005
FEM/Ship	0.0017	0.0017
Economy	0.0003	0.0003
Epidemiology	0.0003	0.0003
FEM/Accelerator	0.0006	0.0008
Circuit	0.0002	0.0002
Webbase	0.0007	0.0006
LP	0.0043	0.0044
circuit5M	0.0132	0.0131
eu-2005	0.0043	0.0050
Ga41As41H72	0.0073	0.0050
in-2004	0.0043	0.0039
mip1	0.0023	0.0019
Si41Ge41H72	0.0062	0.0039
Rata-rata	0.0028	0.0025

Tabel 4.5. Durasi waktu eksekusi algoritma CSR-Adaptive original dan modifikasi

Matrix	Bandwidth	
	Original	Modifikasi
Dense 4K	104.07	102.84
Protein	106.75	155.50
FEM/Spheres	155.42	146.76
FEM/Cantilever	161.09	152.50
Wind Tunnel	180.87	151.95
FEM/Harbor	145.79	144.12
FEM/Ship	154.49	149.93
Economy	134.91	130.81
Epidemiology	214.48	211.03
FEM/Accelerator	138.07	91.83
Circuit	140.99	138.30
Webbase	152.70	156.05
LP	62.68	62.42
circuit5M	132.14	132.49
eu-2005	130.71	114.25
Ga41As41H72	77.54	113.45
in-2004	108.34	121.94
mip1	116.05	136.92
Si41Ge41H72	71.01	112.93
Rata-rata	130.95	132.95

Tabel 4.6. Penggunaan *bandwidth* algoritma CSR-Adaptive original dan modifikasi

Pada penelitian ini, turut diukur pula kinerja dari algoritma CSR-*Adaptive original* dari penelitian sebelumnya [6]. Seperti yang telah dijelaskan pada sub bab 2.5.1., algoritma CSR-*Stream* pada CSR-*Adaptive original* tidak memiliki solusi untuk `row_blocks` yang memiliki sedikit baris.

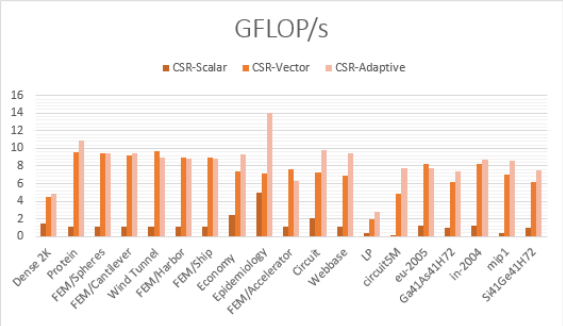
Penulis mencoba untuk mengatasi hal tersebut dengan cara memodifikasi algoritma CSR-*Stream* sehingga terdapat kondisi yang dapat menentukan apakah nilai tak nol setiap baris pada `row_blocks` akan direduksi secara serial atau secara paralel.

Dari tabel 4.2., terlihat bahwa algoritma modifikasi memberikan rata-rata kinerja sedikit lebih baik daripada algoritma *original* untuk semua dataset yang dipilih.

Pada beberapa dataset, algoritma original menghasilkan kinerja yang lebih baik, hal tersebut dikarenakan adanya variansi nilai NNZ per baris yang cukup besar pada setiap blok baris sehingga baris pada tiap blok baris yang seharusnya direduksi secara serial malah diproses secara paralel atau sebaliknya.

4.2.1 Analisis Perbandingan GFLOPS

Algoritma CSR-*Adaptive* mampu menghasilkan rata-rata peningkatan jumlah GFLOPS hingga 6.4x lipat dibandingkan dengan CSR-*Scalar*. CSR-*Adaptive* memberikan performansi terbaik pada data matriks jarang `economy`, `epidemiology`, `circuit`, `webbase`, dan `circuit5M`. Hal itu dikarenakan data matriks jarang tersebut memiliki rata-rata NNZ yang kecil tiap barisnya, sehingga CSR-*Adaptive* melakukan reduksi data secara serial dengan cepat tanpa terjadinya akses memori yang tidak berkesinambungan. Perhatikan pula kinerja pada matriks `dense2` dan `LP`, meskipun rata-rata NNZ per barisnya 2634 namun kinerjanya mengalahkan algoritma CSR lainnya. Hal tersebut disebabkan NNZ per barisnya melebihi `LOCAL_SIZE` sehingga CSR-Adaptive melakukan operasi SpMV berbasis *vector* dan melakukan reduksi data secara paralel dengan cepat.



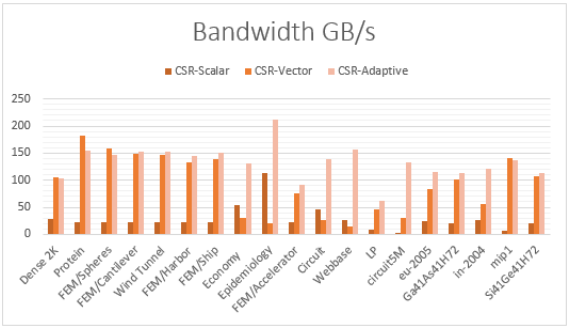
Gambar 4.3. Grafik perbandingan jumlah GFLOPS

Di lain pihak, ada beberapa data matriks yang menghasilkan performansi lebih tinggi untuk CSR-*Vector* dibandingkan CSR-*Adaptive*. Hal tersebut disebabkan adanya nilai ekstrim pada jumlah NNZ per baris pada blok baris tertentu yang mengakibatkan kinerja CSR-*Stream* menjadi terganggu. Perlu ditambahkan pula bahwa algoritma CSR-*Vector* melakukan operasi *floating-point* lebih banyak dibandingkan dengan algoritma CSR-*Adaptive*. Terlebih lagi, implementasi algoritma CSR-*Adaptive* tidak sederhana seperti CSR-*Vector*, karena sebelum proses utama dimulai, ada beberapa *preprocessing* yang dapat mempengaruhi total waktu eksekusi.

4.2.2 Analisis Perbandingan *Bandwidth*

Penggunaan *bandwidth* oleh algoritma CSR-*Adaptive* memberikan rata-rata peningkatan penggunaan *bandwidth* hingga 4.7x lipat dibandingkan dengan implementasi algoritma CSR lainnya. Performansi yang sangat baik terdapat pada matriks `epidemiology` yang mampu melakukan utilisasi *bandwidth* hingga mencapai 211.03 GB/s atau sekitar 94% dari *peak memory bandwidth* GTX 770 (224.26 GB/s).

Performansi penggunaan *bandwidth* sangat bergantung terhadap aktifitas *thread*, semakin sedikit jumlah *thread* yang berada dalam keadaan *idle*, semakin besar pula *bandwidth* yang diutilisasi. Penggunaan *bandwidth* juga bergantung pada akses memori yang dilakukan oleh *thread*. Apabila *thread* melakukan akses memori yang tidak berkesinambungan (*uncoalesced*), maka hal tersebut akan mempengaruhi waktu yang diperlukan bagi *thread* untuk menstransfer data, sehingga waktu yang dibutuhkan untuk menyelesaikan proses pun menjadi lebih lama.



Gambar 4.4. Grafik perbandingan penggunaan bandwidth

Pada grafik terlihat bahwa CSR-*Scalar* mengalahkan penggunaan *bandwidth* CSR-*Vector* untuk data matriks jarang `epidemiology`. Hal tersebut terjadi karena data matriks `epidemiology` memiliki rata-rata NNZ per baris yang kecil, sehingga setiap thread pada algoritma CSR-*Scalar* dapat melakukan akses memori yang lumayan berdekatan satu sama lain. Akibatnya waktu yang diperlukan bagi *thread* untuk mentransfer data menjadi lebih cepat. Sedangkan pada CSR-*Vector*, terdapat banyak *thread* yang berada dalam posisi *idle* dikarenakan data matriks `epidemiology` memiliki rata-rata NNZ per baris yang kecil.

4.2.3 Analisis Perbandingan waktu

Pada dasarnya, informasi waktu eksekusi hanya diperlukan untuk keperluan menghitung GFLOPS dan penggunaan *bandwidth*. CSR-*Adaptive* sendiri memberikan rata-rata peningkatan kecepatan hingga 23.7x lipat dibandingkan dengan implementasi algoritma CSR lainnya. Waktu eksekusi pada algoritma CSR-*Adaptive* sangat dipengaruhi oleh NNZ per baris pada `row_blocks` yang berisi lebih dari satu. Apabila ada nilai ekstrim pada NNZ per baris, maka proses CSR-*Stream* pada CSR-*Adaptive* akan menjadi lambat.

5. Penutup

5.1 Kesimpulan

Algoritma akan dieksekusi Berdasarkan hasil implementasi dan analisis yang dilakukan pada algoritma CSR-*Adaptive*, maka dapat ditarik kesimpulan sebagai berikut:

1. Implementasi CSR-*Adaptive* dapat dilakukan menggunakan teknologi CUDA pada GPU NVIDIA dan dapat dijadikan solusi implementasi kernel SpMV berbasis CSR yang cepat tanpa

harus mengubah skema penyimpanan yang telah diimplementasikan sebelumnya.
2. CSR-*Adaptive* mampu memberikan rata-rata peningkatan jumlah GFLOPS hingga 6.4x lipat, rata-rata peningkatan penggunaan *bandwidth* hingga 4.7x lipat, dan rata-rata peningkatan kecepatan hingga 23.7x lipat dibandingkan dengan implementasi algoritma CSR lainnya.
3. Dalam penelitian ini, didapat konfigurasi jumlah *block* dan *thread* terbaik untuk menjalankan algoritma CSR-*Adaptive* yaitu sebesar 3584 untuk *block* dan 256 untuk *thread*.
4. Pada CSR-*Adaptive*, terdapat 2 kondisi untuk CSR-*Stream*, kondisi tersebut hanya bekerja dengan baik apabila distribusi NNZ tiap baris pada blok baris seragam. Apabila terdapat nilai ekstrim pada salah satu baris, kinerja algoritma akan menjadi buruk.
5. Meskipun untuk beberapa matriks jarang kinerja CSR-*Adaptive* lebih rendah dibandingkan CSR-Vector, namun perbandingan performa kedua algoritma untuk matriks tersebut tidak begitu jauh.
6. Kinerja CSR-*Adaptive* pada data matriks jarang dengan rata-rata NNZ per baris yang kecil memberikan hasil yang sangat memuaskan.

5.2 Saran

1. Memperbaiki algoritma CSR-*Stream* sehingga apabila terdapat nilai ekstrim pada salah satu baris, kinerja algoritma dapat tetap dipertahankan kualitasnya.
2. Melakukan perbandingan terhadap implementasi algoritma SpMV lainnya seperti COO, HYB, ELLPACK, dan DIA.

Daftar Pustaka

[1] NVIDIA. “What is GPU Computing? | High Performance Computing | NVIDIA | NVIDIA”. http://www.nvidia.com/object/what-is-gpu-computing.html.

[2] J. R. Gilbert, S. Reinhardt, and V. B. Shah, “Highperformance Graph Algorithms from Parallel Sparse Matrices,” in Proc. of the Int’l Workshop on Applied Parallel Computing, 2006.

[3] T. A. Davis; Y. Hu. “Tim Davis: University of Florida Sparse Matrix Collection: sparse matrices from a wide range of applications”. http://www.cise.ufl.edu/research/sparse/matrices/

[4] Joseph L. Greathouse, Mayank Daga. "Efficient Sparse Matrix-Vector Multiplication on GPUs using the CSR Storage Format". SC, 2014.

[5] R. W. Vuduc, “Automatic Performance Tuning of Sparse Matrix Kernels,” Ph.D. dissertation, University of California, Berkeley, 2003.

[6] Hoang-Vu Dang, Bertil Schmidt. "The Sliced COO format for Sparse Matrix-Vector Multiplication on CUDA-enabled GPUs". ICCS, 2012. pp 57-66

[7] N. Bell, M. Garland, Implementing Sparse Matrix–Vector Multiplication on throughput-oriented processors, in: SC, ACM, 2009, pp. 1–11

[8] Arnold, Vladimir I.; Cooke, Roger (1992), Ordinary differential equations, Berlin, DE; New York, NY: Springer-Verlag.

[9] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. "Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression". ACM. 2008.

[10] Wikipedia. “FLOPS – Wikipedia, the free encyclopedia”. https://en.wikipedia.org/wiki/FLOPS.

[11] NVIDIA. “Parallel Programming and Computing Platform | CUDA | NVIDIA | NVIDIA”. http://www.nvidia.com/object/cuda_home_new.html.