OPTIMISASI PUSTAKA UNTUK PERKALIAN MATRIKS MENGGUNAKAN ALGORITMA STRASSEN BERBASIS OPENCL

Arvin[1)], Sutrisno[2)], Pujianto Yugopuspito[3)]

[1),2),3)]Teknik Informatika, Universitas Pelita Harapan
UPH Tower, Lippo Vilage Karawaci, Tangerang 15811
e-mail: yugopuspito@uph.edu

Abstrak . *Optimisasi piranti lunak menjadi hal yang penting dan menjadi fokus dari penelitian ini. Optimisasi yang dilakukan dilakukan pada pustaka perkalian matriks Strassen menggunakan Open Computing Language (OpenCL). OpenCL adalah suatu framework terbuka untuk menulis piranti lunak yang dapat dieksekusi pada platform heterogen seperti central processing unit (CPU), graphics processing unit (GPU), digital signal processor (DSP) atau perangkat keras pemercepat komputasi. Pada penelitian ini berbagai macam pengujian telah dilakukan guna meningkatkan performa pustaka. Pengujian tersebut menggunakan matriks dengan ukuran yang berbeda-beda dan menggunakan parameter uji yang bervariasi. Parameter yang digunakan adalah local memory, local data store, pinned buffer, dan zero copy buffer. Analisa data dilakukan dengan cara membuat perbandingan dari data yang telah dikumpulkan. Hasil pengujian menunjukkan bahwa apabila parameter uji digunakan secara tepat maka peningkatan kecepatan dalam perhitungan perkalian matriks dapat dicapai. Perhitungan matriks berukuran lebih dari 128x128 menggunakan algoritma Strassen Optimal pada device GPU 2.2 kali lebih cepat. Pada ukuran matriks 8192x8192 pada device CPU dalam hal waktu komputasi perkalian matriks bisa 28,8 kali lebih cepat dibandingkan dengan algoritma Naif. Hasil ini sangat signifikan dalam pemrograman yang diujicobakan pada GPU dengan 1600 processing element.*

Kata kunci*: optimisasi, algoritma Strassen, perkalian matriks, OpenCL.*

1. Pendahuluan

Optimisasi program menjadi menjadi hal yang penting agar penggunaan piranti lunak dalam operasi bekerja secara maksimal pada perangkat keras yang menjalankan piranti lunak tersebut. Dalam paper ini, kasus pustaka perkalian matriks menggunakan algoritma Strassen untuk dioptimisasi agar dapat menghitung perkalian matriks berukuran besar maupun kecil secara efisien. Paralelisasi dilakukan pada pustaka perkalian matriks menggunakan Open Computing Language (OpenCL), kerangka kerja standar industri gratis dan terbuka untuk pemrograman paralel pada prosesor modern [1], untuk meningkatkan performa pustaka dalam menghitung perkalian matriks.

Tujuan dari penelitian ini adalah untuk membuat pustaka yang dapat menghitung perkalian matriks menggunakan algoritma Strassen secara paralel baik matriks berukuran kecil maupun besar dengan efisien menggunakan OpenCL. Selain menggunakan algoritma Strassen dalam membuat pustaka perkalian matriks, algoritma Naif digunakan sebagai basis perbandingan.

Dalam penelitian Hudi et al. [2] dan Yugopuspito et al. [3] membuat program perkalian matriks yang dapat bekerja secara paralel pada GPU menggunakan algoritma Strassen. Mereka menggunakan algoritma Naif sebagai pembanding dan melakukan pengujian pada NVIDIA menggunakan CUDA. Penelitian yang mereka lakukan berfokus pada penanggulangan yang dapat dilakukan untuk mengatasi masalah keterbatasan memori menggunakan algoritma Strassen, dengan simpulan algoritma Strassen dapat melakukan perkalian matriks lebih besar dibandingkan dengan algoritma Naif. Penelitian ini lebih berfokus pada optimisasi secara general yang dapat dilakukan pada pustaka perkalian matriks menggunakan algoritma Strassen.

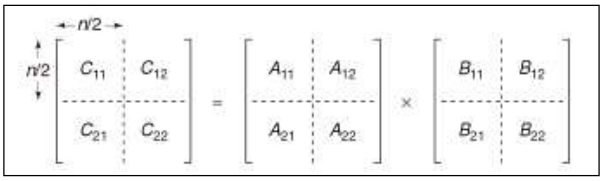
Sementara Shen et al. [4] melakukan berbagai macam pengujian pada device CPU menggunakan OpenCL. Mereka menggunakan prosesor Intel Xeon E5620 2.40 GHz *hyper-thread dual quad-core* (8 core, 16 thread) sebagai CPU. Selain itu, mereka juga menggunakan Intel OpenCL (OCL) SDK 1.5 dan AMD Accelerated Parallel Processing (APP) SDK 2.6. Penelitian yang mereka lakukan berfokus pada peningkatan performa OpenCL pada CPU. Penelitian ini mengelaborasinya menjadi CPU dan GPU dengan kasus khusus algoritma Strassen pada perkalian matriks dengan ukuran yang sama. Parameter uji yang akan digunakan antara lain adalah jenis algoritma perkalian matriks (Naif vs.

Strassen), jumlah *work item* dalam *work group*, jenis *device* OpenCL (CPU vs. GPU), penggunaan *local memory*, dan pemilihan metode transfer data (non)-zero copy.
Dalam perancangan pustaka perkalian matriks, algoritma Naif digunakan sebagai basis perbandingan terhadap algoritma Strassen. Algoritma Naif dapat digunakan untuk menghitung perkalian matriks berukuran n × n dengan kompleksitas waktu T(n^3). Perkalian matriks menggunakan algoritma Naif hanya dapat dilakukan apabila jumlah baris dalam matriks pertama sama dengan jumlah kolom pada matriks kedua sehingga perkalian matriks dengan algoritma Naif [5]. Pseudocode algoritma Naif untuk perkalian antara matriks A dengan matriks B dengan berukuran n×n seperti pada Gambar 1.

```
Naive (A [][], B [][])
{
 n = A.rows();
 Let C be a new n x n matrix;
 For i = 0 to n-1
   For j = 0 to n-1
      Cij = 0;
      For k = 0 to n-1
         Cij = Cij + Aik * Bkj;
 return C;
}
```

Gambar 1. *Pseudocode* algoritma Naif untuk perkalian matriks

Algoritma Stressen berbeda dengan algoritma Naif, kompleksitas waktu yang dibutuhkan dalam menghitung perkalian matriks menggunakan algoritma Strassen adalah ($T(n) = n^{2.81}$) [6]. Perkalian matriks menggunakan algoritma Strassen dilakukan dengan menerapkan penggunaan algoritma *divide and conquer*. Algoritma *divide and conquer* bekerja dengan cara membagi suatu masalah menjadi sub masalah, menyelesaikan sub masalah secara rekursif, dan menggabungkan seluruh solusi dari setiap sub masalah menjadi solusi untuk keseluruhan masalah [7]. Untuk menghitung perkalian antara matriks A dan matriks B yang hasilnya adalah matriks C maka yang pertama kali harus dilakukan adalah membagi matriks A dan B menjadi 4 submatriks yang lebih kecil, seperti yang terdapat pada Gambar 2. Setelah membagi masing-masing matriks A dan B menjadi 4 buah submatriks, maka akan dibuat tujuh buah matriks baru (M1, M2, M3, ... , M7) hanya dengan menggunakan tujuh operasi perkalian dengan pembentukan matriks C dari empat submatriks C_{11}, C_{12}, C_{21} dan C_{22} dari semua nilai M yang didapatkan.



Gambar 2. Pembagian submatriks pada algoritma Strassen[6]

Gambar 3 menunjukkan usulan *pseudocode* dari penerapan algoritma *divide and conquer* pada algoritma Strassen. Algoritma Strassen hanya dapat digunakan untuk menghitung perkalian matriks berukuran 2^n x 2^n sehingga apabila algoritma Strassen digunakan dalam menghitung matriks yang ukurannya bukan merupakan 2^n maka ukuran matriks harus diubah dalam bentuk 2^n dimana baris dan kolom yang kosong harus diisi dengan angka nol. Usulan ini telah mempertimbangkan suatu solusi dibutuhkan untuk meningkatkan kecepatan algoritma Strassen dalam menghitung matriks berukuran kecil yaitu dengan menggunakan algoritma Naif apabila ukuran submatriks telah mencapai nilai *threshold* tertentu, dan harus merupakan 2^n.

```
Strassen (A [][], B [][])
{
  n = A.rows ();
  If (n <= threshold) Then
     Naive (A, B);
  Else
  {
  m = n/2;
  Let A11, A12, A21, A22 be a new m x m matrix;
  Let B11, B12, B21, B22 be a new m x m matrix;
  Let M1, M2, M3, M4, M5, M6, M7 be a new m x m matrix;
  Partition A into four submatrices A11, A12, A21, A22;
  Partition B into four submatrices B11, B12, B21, B22;
  M1 = Strassen (A11 + A22, B11 + B22);
  M2 = Strassen (A21 + A22, B11);
  M3 = Strassen (A11, B12 - B22);
  M4 = Strassen (A22, B21 - B11);
  M5 = Strassen (A11 + A12, B22);
  M6 = Strassen (A21 - A11, B11 + B12);
  M7 = Strassen (A12 - A22, B21 + B22);
  C11 = M1 + M4 - M5 + M7;
  C12 = M3 + M5;
  C21 = M2 + M4;
  C22 = M1 - M2 + M3 + M6;
  Combine C11, C12, C21, and C22 into C;
  return C;
}
```

Gambar 3. *Pseudocode* algoritma Strassen yang telah dioptimasi.

Secara teoritis *device* CPU dapat menyelesaikan data berukuran kecil jauh lebih cepat dibandingkan GPU. Hal ini disebabkan karena waktu yang dibutuhkan untuk menjalankan kernel pada GPU jauh lebih lambat apabila dibandingkan dengan waktu yang dibutuhkan CPU dalam menjalankan kernel. Selain itu CPU juga memiliki *clock rate* yang lebih tinggi dibandingkan GPU sehingga dapat menghitung data berukuran kecil dengan sangat cepat. GPU dapat menghitung data yang berukuran sangat besar jauh lebih cepat dibandingkan CPU dikarenakan GPU memiliki *compute unit* dan *thread* yang jauh lebih banyak dibandingkan CPU. Misal GPU ATI Radeon HD 5870 yang digunakan memiliki 20 unit SIMD dimana setiap unit SIMD memiliki 16 *stream core* dan setiap *stream core* memiliki lima *processing element* yang berarti GPU ATI Radeon HD 5870 memiliki total (20 × 16 × 5) 1600 *processing element*. Jumlah ini jauh lebih banyak apabila dibandingkan dengan CPU yang setiap *compute unit* hanya memiliki satu *thread*. Dengan *thread* sebanyak itu yang bekerja secara paralel, GPU dapat melakukan perhitungan *kernel* pada data berukuran besar jauh lebih cepat.
Lingkungan pengujian pustaka perkalian matriks adalah laptop Sony Vaio VPCSA25GG dengan prosesor Intel Core i7-2620M 2.70 GHz *hyper-threading* (2 *core*, 4 *thread;)* dan *discrete* GPU AMD HD 6630M 485 MHz (6 *compute unit*, 96 *thread; 480 GFLOPS)*. Kernel OpenCL dikompilasi menggunakan AMD Accelerated Parallel Processing (APP) SDK 2.8. Hasil pengujian diperoleh dengan menggunakan cara mengambil *wall clock time* dari eksekusi kernel, dan menggunakan AMD APP Profiler 2.5. Besar ukuran matriks yang digunakan selama pengujian merupakan $2^n \times 2^n$ dan elemen pada data matriks bernilai dari 1 sampai 10. Pengujian dilakukan dengan memanggil pustaka perkalian matriks dari main program pengujian.

2. Pembahasan

Pengujian yang dilakukan pada pustaka perkalian matriks terdiri dari lima jenis pengujian yaitu pengujian optimisasi jumlah *work-item* dalam *work-group*, pengujian optimisasi pemilihan *device* OpenCL, pengujian optimisasi memori, pengujian waktu komputasi algoritma perkalian matriks, dan pengujian optimisasi proses transfer data. Dari pengujian optimisasi yang telah dikerjakan, yang paling penting adalah pengujian waktu komputasi algoritma perkalian matriks.
Pada pengujian waktu komputasi algoritma perkalian matriks, waktu komputasi dari algoritma Naif dan algoritma Strassen (dengan nilai *threshold* yang berbeda) akan dihitung dan dibandingkan satu

sama lain. Waktu komputasi merupakan waktu yang dibutuhkan program untuk melakukan proses perhitungan seperti perkalian, pembagian, pengurangan, dan penambahan. Itu berarti waktu yang dibutuhkan untuk membuat *buffer* matriks, mengisi *buffer* dengan data matriks, dan mengirim *buffer* menuju *device* bukan merupakan waktu komputasi.
Dalam algoritma Naif yang dimaksud dengan waktu komputasi merupakan waktu eksekusi kernel sedangkan pada algoritma Strassen waktu komputasi merupakan total waktu eksekusi dari semua kernel yang dijalankan dalam algoritma Strassen (kernel yang dijalankan pada algoritma Strassen lebih dari satu yaitu kernel untuk operasi penjumlahan, pengurangan, dan operasi perkalian matriks menggunakan algoritma Naif). Penggunaan banyak kernel pada algoritma Strassen dikarenakan tidak semua kode pustaka algoritma Strassen dapat diparalelisasi seutuhnya berbeda dengan algoritma Naif yang dapat diparalelisasi seutuhnya. Waktu komputasi diperoleh dengan menjumlahkan semua total waktu kernel yang dieksekusi pada algoritma Strassen sehingga waktu komputasi dari algoritma Strassen adalah:

$$T_{komputasi} = \sum T_{addMatrices} + \sum T_{naiveAlgoritm} + \sum T_{substracMatrices} + \sum T_{countC22} + \sum T_{countC11} \quad (1)$$

Pengujian waktu komputasi algoritma perkalian matriks akan dilakukan menggunakan *global memory* dan diuji pada *device* GPU dan CPU. Tabel 1 menunjukkan perbandingan waktu komputasi antara algoritma Naif dengan algoritma Strassen pada *device* CPU dan GPU dimana algoritma Strassen optimal lebih cepat dibandingkan algoritma Naif pada matriks berukuran 128×128. Tampak bahwa algoritma Naif jauh lebih cepat dibandingkan algoritma Strassen Murni di setiap ukuran matriks. Apabila algoritma Naif dibandingkan dengan algoritma Strassen Optimal maka dapat dilihat bahwa waktu komputasi algoritma Naif lebih cepat dibandingkan algoritma Strassen Optimal pada matriks berukuran 2×2 sampai dengan matriks berukuran 64×64 namun waktu komputasi algoritma Strassen optimal lebih cepat dibandingkan algoritma Naif dari matriks berukuran 128×128 sampai matriks berukuran 8192×8192.

Tabel 1. Hasil Perbandingan Algoritma Naif dan Algoritma Strassen pada CPU dan GPU

Ukuran Matrix	Waktu Eksekusi (ms) pada *device* CPU					Waktu Eksekusi (ms) pada *Device* GPU				
	Naif	Strassen Murni		Strassen Optimal		Naif	Strassen Murni		Strassen Optimal	
	T_N	T_{SM}	T_N / T_{SM}	T_{SO}	T_N / T_{So}	T_N	T_{SM}	T_{SM} / T_N	T_{SO}	T_{SO} / T_N
2 x 2	0.012	0.018	0.671	0.0185	0.671	0.071	0.424	0.168	0.424	0.168
4 x 4	0.015	0.084	0.177	0.0252	0.588	0.078	2.278	0.034	0.447	0.175
8 x 8	0.012	0.501	0.024	0.0233	0.524	0.081	14.641	0.006	0.467	0.174
16 x 16	0.018	3.36	0.005	0.0315	0.567	0.143	108.275	0.001	0.496	0.288
32 x 32	0.059	23.28	0.003	0.0763	0.769	0.241	762.128	0.000	0.656	0.367
64 x 64	0.163	163.90	0.001	0.2383	0.683	0.409	5365.372	0.000	0.866	0.472
128 x 128	1.39	1132.46	0.001	1.17	1.188	2.84	37158.800	0.000	1.28	2.212
256 x 256	11.06	7966.00	0.001	8.55	1.294	6.53	263341.00	0.000	4.33	1.507
512 x 512	116.54	56062.10	0.002	59.55	1.957	27.26	X	X	24.61	1.107
1024 x 1024	4324.69	384124.00	0.011	409.24	10.568	190.86	X	X	162.93	1.171
2048 x 2048	48004.61	X	X	2901.22	16.546	1510.27	X	X	1147.39	1.316
4096 x 4096	424453.90	X	X	21407.78	19.827	12094.81	X	X	8074.67	1.498
8192 x 8192	4136578.80	X	X	143613.0	28.804	97649.72	X	X	56638.40	1.724

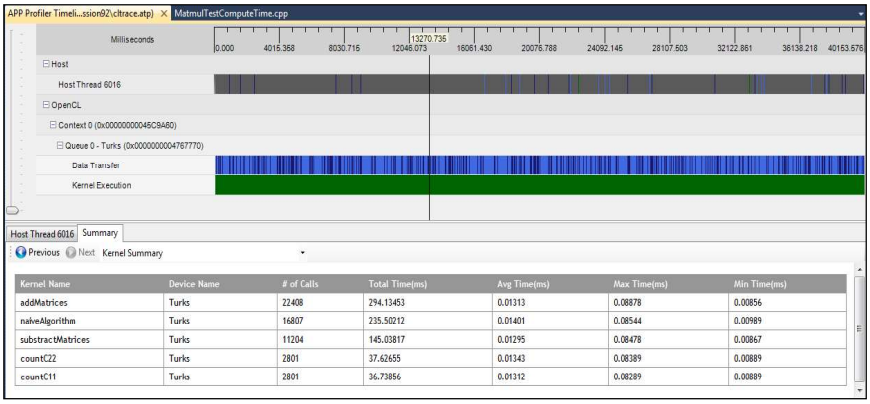
X = Tidak Dikerjakan (waktu pengujian lebih dari 10 jam)

Proses transfer data pada OpenCL umumnya dilakukan dengan cara membuat *buffer* pada *device* (CPU atau GPU) dan mengirim data *input* matriks dari *host* menuju *buffer* begitu juga sebaliknya. Kecepatan transfer data ini dapat ditingkatkan dengan cara membuat *buffer* yang disimpan pada *pinned host memory* (bagian dari *physical memory* (RAM) yang tidak pernah dipindahkan dalam

virtual memory). *Buffer* yang disimpan pada *pinned host memory* seringkali disebut dengan istilah *pinned buffer*. Penggunaan *pinned buffer* memungkinkan GPU agar dapat langsung menggunakan *direct memory access* (DMA) untuk mengkopi data dari atau menuju host (CPU) melalui PCI Express (PCIe). Penggunaan *pinned buffer* dapat meningkatkan kecepatan proses transfer data namun juga dapat menyebabkan RAM kehabisan memori lebih cepat dikarenakan *buffer* yang disimpan dalam RAM tidak pernah dipindahkan dalam *virtual memory*.

Tabel 2. Hasil Pengujian Proses Transfer Data pada GPU

Ukuran Matrix	Size	TSB (ms)	T2SB1ZCB (ms)	TSB / T2SB1ZCB	T3ZCB (ms)	TSB / T3ZCB
2 x 2	16 B	3.528	0.065	54.44	0.015	229.08
4 x 4	64 B	3.317	0.078	42.36	0.070	47.38
8 x 8	256 B	2.700	0.096	28.07	0.098	27.64
16 x 16	1024 B	3.336	0.143	23.37	0.156	21.44
32 x 32	4 KB	3.217	0.222	14.48	0.250	12.87
64 x 64	16 KB	3.456	0.392	8.82	0.353	9.79
128 x 128	64 KB	7.572	3.521	2.15	2.063	3.67
256 x 256	256 KB	11.331	6.487	1.75	8.132	1.39
512 x 512	1024 KB	35.788	28.410	1.26	33.932	1.05
1024 x 1024	4 MB	205.087	190.837	1.07	258.676	0.79
2048 x 2048	16 MB	1533.459	1507.668	1.02	1878.661	0.82
4096 x 4096	64 MB	12176.901	12096.985	1.01	17021.029	0.72
8192 x 8192	256 MB	99623.754	96954.820	1.03	136079.401	0.73



Gambar 4. Tipikal tampilan AMD APP *Profiler:* algoritma Strassen (*threshold* = 1) matriks 32×32

Zero copy buffer (ZCB) merupakan pengembangan dari konsep *pinned buffer*. Keduanya memiliki kesamaan yaitu *buffer* sama – sama disimpan pada *pinned host memory* namun yang membedakan adalah apabila dalam *pinned buffer* terjadi proses transfer data dari *host* (CPU) menuju *device* (GPU) maupun sebaliknya menggunakan DMA namun pada ZCB proses transfer data tidak terjadi sama sekali karena *device* dapat mengakses langsung *buffer* pada *pinned host memory* selama proses eksekusi kernel berlangsung. Penggunaan ZCB pada GPU dapat memberikan hasil yang berbeda. Pada *integrated* GPU, penggunaan ZCB akan selalu meningkatkan performa dikarenakan *physical memory* (RAM) selalu digunakan oleh CPU dan GPU bersama-sama sedangkan pada *discrete* GPU penggunaan ZCB akan meningkatkan performa selama *buffer* tidak dibaca dan dimodifikasi berkali-kali[8]. Apabila digunakan berulang kali maka penggunaan ZCB pada *discrete* GPU dapat memperlambat waktu eksekusi kernel dikarenakan lambatnya *device* GPU dalam mengakses CPU melalui PCIe. Hasil dari pengujian optimisasi proses transfer data menggunakan *standard buffer* (SB), dua SB dengan satu ZCB (2SB1ZCB), tiga ZCB (3ZCB) pada *device* GPU dapat dilihat secara

berurutan pada Tabel 2. Ini adalah hasil dari pengujian yang dilakukan pada matriks berukuran 2×2 sampai matriks berukuran 8192×8192 untuk *device* GPU dan matriks berukuran 2×2 sampai matriks berukuran 4096×4096 untuk *device* CPU dengan menggunakan AMD APP Profiler untuk memperoleh waktu dan kecepatan proses transfer data dari *host* menuju *device* maupun sebaliknya. Tipikal hasil AMD APP profile dapat dilihat pada Gambar 4.

3. Simpulan

Berdasarkan hasil penelitian yang telah dilakukan dengan mengunakan berbagai macam jenis pengujian maka dapat disimpulkan beberapa hal yaitu:

1. Penggunaan banyak *work item* dalam *work group* dapat meningkatkan performa *device* GPU dalam mengeksekusi kernel sedangkan pada *device* CPU tidak terjadi peningkatan yang berarti.
2. *Device* CPU jauh lebih cepat apabila digunakan untuk menghitung matriks berukuran kecil, sementara *device* GPU lebih cepat apabila digunakan untuk menghitung matriks berukuran besar, lebih dari 128x128.
3. Penggunaan *local memory* pada saat kernel dieksekusi telah terbukti meningkatkan kecepatan waktu eksekusi kernel pada *device* GPU. Sementara pada *device* CPU penggunaan *local memory* justru memperlambat waktu eksekusi kernel.
4. Algoritma Naif bekerja jauh lebih cepat apabila dibandingkan dengan algoritma Strassen murni dalam menghitung perkalian matriks, baik pada CPU dan GPU. Tetapi Algoritma Strassen yang telah dioptimasi juga bekerja jauh lebih cepat apabila dibandingkan dengan algoritma Strassen murni dalam menghitung perkalian matriks.
5. Penggunaan *zero copy buffer* pada *device* CPU lebih cepat apabila dibandingkan dengan penggunaan *buffer* biasa pada *device* CPU. Dan Pada *device discrete* GPU penggunaan *zero copy buffer* juga dapat menghapus biaya waktu yang dibutuhkan untuk melakukan proses transfer data namun juga dapat meningkatkan waktu eksekusi kernel.

Daftar Pustaka

[1]. Khronos Group., "OpenCL – The Open Standard for Parallel Programming of Heterogeneous System," 2013. http://www.khronos.org/opencl/.

[2]. Robertus Hudi. "Penanggulangan Masalah Keterbatasan Memori pada Proses Paralel GPU dengan Menggunakan Algoritma Strassen." S.Inf. Skripsi, Universitas Pelita Harapan, Indonesia, 2013.

[3]. Pujianto Yugopuspito, Sutrisno, dan Robertus Hudi, "Breaking Through Memory Limitation in GPU Parallel Processing Using Strassen Algorithm," in Proc. International Conference on Computer, Control, Informatics, and Its Application 2013.

[4]. Jin Shen, Jianbin Fang, Henk Sips, dan Ana Lucia Varbanescu, "Performance Traps in OpenCL for CPUs," in Proc. 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing 2013.

[5]. Sopandi Ahmad, 2013, Perkalian Matriks. http://matemakita.com/matriks/perkalian-matriks.php, diakses 10 Desember 2016.

[6]. Richard Neapolitan, 2003. *Foundations of Algorithms Using C++ Pseudocode*, 3rd ed. United States of America: Addison-Wesley Professional.

[7]. Robert Sedgewick, 1988. *Algorithms*, 3rd ed. United States of America: Addison-Wesley Publishing.

[8]. Jason Sanders dan Edward Kandrot, 2010. *CUDA by Example: An Introduction to General Purpose GPU Programming*. United States of America: Addison-Wesley Professional.