Jurnal Teknologi Informasi dan Ilmu Komputer (JTIIK) DOI: 10.25126/jtiik.201744488
Vol. 4, No. 4, Desember 2017, hlm. 221-226 p-ISSN: 2355-7699
Akreditasi KEMENRISTEKDIKTI, No. 51/E/KPT/2017 e-ISSN: 2528-6579

OPTIMASI RENDERING GAME 2D ASTEROIDS MENGGUNAKAN PEMROGRAMAN CUDA

Fathony Teguh Irawan[1], Muhammad Rizal Ma'rufi[2], Imam Cholissodin[3]

[1,2,3]Fakultas Ilmu Komputer Universitas Brawijaya
Email: [1]fathony.teguh@gmail.com, [2]mrmakroefi@gmail.com, [3]imamcs@ub.ac.id

(Naskah masuk: 19 Oktober 2017, diterima untuk diterbitkan: 25 Desember 2017)

Abstrak

Sumber untuk mendapatkan hiburan sangatlah banyak, salah satunya melalui media *video game*. Minatnya masyarakat terhadap *video game* dibuktikan dengan besarnya angka pengguna *video game*. Oleh karena itu, performa *video game* sangatlah diperhitungkan agar dapat memperluas pasar. Salah satu cara untuk meningkatkan performa dari *video game* adalah dengan memanfaatkan GPU. Cara untuk membuktikan bahwa performa GPU lebih baik daripada CPU dalam pemrosesan secara parallel adalah dengan cara membandingkan hasil dari proses CPU dibandingkan dengan hasil proses GPU. Paper ini memaparkan perbedaan performa sebuah *video game* yang diimplementasikan menggunakan CPU yang dibandingkan dengan implementasi GPU.

Kata kunci: *games, video game, game development, CPU, GPU, CUDA, optimasi, analisis*

Abstract

There are many sources for having fun, one of them is through a video game. Public interest in a video game is proven by a large number of video game user. Therefore, the performance of video game is considered to expand the market. One of many ways to improve performance is using GPU processing. The way to prove that GPU processing is faster than CPU processing on the parallel process is by comparing the result of GPU processing and CPU processing. This paper describes the differences in performance of video game that is implemented using GPU approach and CPU approach.

Keywords: *games, video game, game development, CPU, GPU, CUDA, optimization, analysis*

1. PENDAHULUAN

Era modern ini banyak sekali hiburan yang bisa dilakukan disaat senggang untuk melatih konsentrasi, daya ingat, terapi untuk edukasi, ataupun menghilangkan penat. Salah satu cara yang paling praktis, populer, dan hampir bisa dilakukan dimana saja serta tersedia dihampir semua platform seperti *mobile, desktop*, *laptop* dan *console*, adalah *video game*. *Video game* sangat diminati oleh masyarakat, hal ini disebabkan karena game dapat menyuguhkan sebuah cerita unik, baik visual maupun audio, dan *user* dapat berinteraksi langsung di dalamnya. Tidak sedikit penggemar *video game* yang bermain hanya untuk hiburan atau menghabiskan waktu luang, hal ini dibuktikan dengan meningkatnya kebutuhan *video game* dari tahun ke tahun dengan jumlah pemain *video game* yang mencapai pada angka 1,3 miliar (De Prato, Feijoo, & Simon, 2014).

Dipihak produsen game sendiri, ini merupakan perilaku yang menguntungkan bagi mereka, yaitu dengan memperluas pasar penjualan *video game*. Namun, kondisi ini tentunya harus diimbangi dengan peningkatan kualitas produksi *video game* oleh pihak industri, salah satunya ialah dengan cara memanfaatkan kemampuan perangkat keras yang tersedia sehingga performa dari *video game* yang dimainkan dapat dijalankan oleh pemain dengan selancar mungkin. Dengan semakin optimalnya sebuah *video game*, maka dapat dipastikan luas pasar yang dapat dijangkau semakin besar. Meskipun sebelumnya, pasar utama dari *Graphical Processing Unit* (GPU) untuk pemrosesan grafis, namun GPU juga dapat digunakan dalam pemrosesan non-grafis dan menghasilkan performa yang sangat bagus. (Ryoo, et al., 2008). Salah satu cara untuk meningkatkan kualitas hasil dari *video game* sehingga optimal adalah dengan menggunakan *parallel processing*. *Parallel processing* digunakan agar dapat melakukan komputasi secara paralel sehingga dapat meningkatkan jumlah *frame* yang diproses dalam satu detik, sehingga meningkatkan performa *video game* secara signifikan.

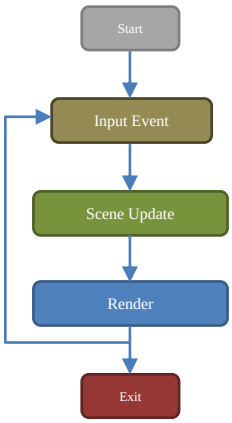
Pemrograman GPU mencakup konsep dasar GPU *programming*, pembahasan mendalam teknik komputasi parallel berbasis GPU, implementasi teknologi terkini, yang meliputi *deep learning*, *self-driving cars*, *virtual reality*, *game development*, *accelerated computing*, *design* & *visualization*, *automous machines*, dengan melakukan komputasi secara cepat dan tepat, serta mengintegrasikan teknik paralel terhadap kebutuhan teknologi masa depan dengan mengambil potensi dari pengolahan data yang cepat untuk meningkatkan hasil yang lebih akurat.

Tujuan diadakan penelitian *video game* Asteroids sebagai objek adalah untuk meningkatkan performa secara umum. Proses peningkatan performa dari *video game* adalah dengan memanfaatkan *interoperability* dari CUDA dan OpenGL. Hasil dari penelitian ini adalah membandingkan perbedaan performa antara *video game* yang memanfaatkan *interoperability* CUDA dan OpenGL dibandingkan dengan *video game* tanpa memanfaatkan *interoperability* dari CUDA dan OpenGL. Batasan dari penelitian ini adalah *video game* ***Asteroids*** dengan memanfaatkan freeglut sebagai *windowing library* dan OpenGL sebagai *rendering library*.

2. DASAR TEORI

2.1 Game Loop

Game loop merupakan *central point* dari sebuah game. Pada tahap ini, program akan menerima *input* dari *user*, yang kemudian akan diproses sebagai *input* untuk mengubah *environment* game dan perubahan visual akan ditampilkan sebagai *output*. Sehingga terjadi interaksi dari *player*. Proses ini akan dilakukan berulang dan terus menerus sampai program dihentikan. Pada Gambar 1 merupakan alur dari game loop berupa diagram.

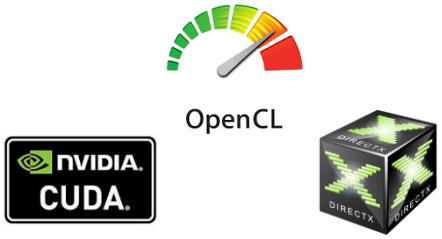


Gambar 1. Game Loop dari game Asteroids

2.2 Pemograman GPU

Graphics Processing Unit (GPU) merupakan sebuah *hardware* yang dimaksudkan untuk memproses gambar pada komputer. Pada tahun 2001, GPU mulai dimanfaatkan untuk keperluan komputing umum atau non-grafis. Kelebihan utama dari GPU dibandingkan dengan CPU adalah *core* GPU yang berjumlah sampai ribuan buah. Hal ini membuat waktu proses menjadi lebih singkat dan efisien dengan memanfaatkan banyaknya *core* untuk membagi dan memproses data secara bersamaan dalam satu waktu.

Kemampuan dari GPU dapat dimanfaatkan harus ada jembatan untuk menghubungkan program dengan *hardware* GPU. Berdasarkan pada Gambar 2, setidaknya ada 3 API untuk memanfaatkan GPU, yakni CUDA dari Nvidia, OpenCL dari Khronos, dan DirectCompute dari Microsoft. Dalam paper ini, API yang digunakan adalah Nvidia CUDA.



Gambar 2. Nvidia CUDA, OpenCL dan DirectX

2.3 CUDA

Compute Unified Device Architecture (CUDA) merupakan *platform* dan *programming* untuk komputasi paralel yang dikembangkan Nvidia. CUDA difokuskan untuk mengambil keuntungan sepenuhnya dari GPU Nvidia. Berbeda dengan OpenCL yang mampu memanfaatkan GPU non-Nvidia (AMD, Intel, Mali, dsb). CUDA memungkinkan untuk melakukan paralel komputing menggunakan GPU untuk keperluan non-grafis.

CUDA sendiri didesain untuk diprogram pada bahasa pemrograman C, C++, dan Fortan. Selain itu, CUDA juga mendukung API seperti OpenCL, hal ini memungkinkan kita untuk menggunakan API CUDA jika pada platform yang menggunakan GPU Nvidia, dan menggunakan API OpenCL pada platform non Nvidia seperti AMD, Intel, Mali, dan PowerVR.

2.4 Kode Kernel dan Host

Kode Kernel adalah kode yang akan dijalankan pada device atau GPU. Kode ini dijalankan pada *thread* di *device*. Salah satu contoh Kode Kernel yang ada pada CUDA dan kode host pada CPU ada pada Kode Program 1 berikut.

```
1  // kode kernel
2  __global__ void fnMatrixPlusGPU(float
3  *A, float *B, float *Hasil){
4      int kolom = threadIdx.x;
5      int baris = threadIdx.y;
6      int tid = (baris * N) + kolom;
7      Hasil[tid] = A[tid] + B[tid];
8  }
9  // kode host
10 void fnMatrixPlusCPU(float *A, float
11 *B, float *Hasil, int jumlahData){
12   for (int i = 0; i < jumlahData;
13   i+=1){
14      Hasil[i] = A[i] + B[i];
15   }
16 }
```

Kode Program 1. Penjumlahan Matrik N x N

Kode Host adalah kode yang dijalankan pada host atau CPU. Inisialisasi memori yang akan diproses oleh *device* akan dilakukan pada Kode Host. Pada Kode Program 1, `N` menyatakan ukuran dari

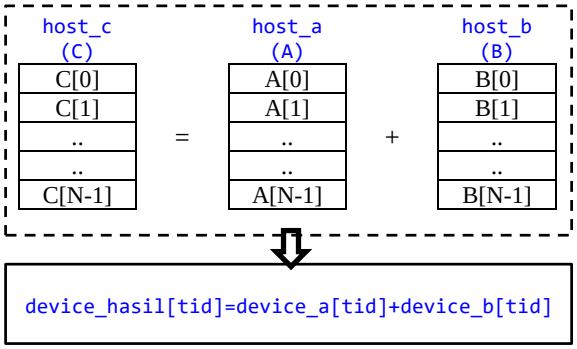
baris atau kolom dari matrik, sedangkan `jumlahData` merupakan hasil dari `N x N`.

2.5 Grid, Block, dan Thread

Ilustrasi penjelasan sebagai contoh yang lebih sederhana untuk memahami dengan mudah terkait konsep *grid*, *block* dan *threads* adalah dengan melihat Kode Program 2 dan 3 GPU untuk vectorAddition dengan blockIdx.x dibandingkan dengan Kode Program 4 dan 5 GPU untuk vectorAddition dengan threadIdx.x berikut.

```
1  __global__ void FnVectAdd(float
2  *device_hasil,float *device_a, float
3  *device_b){
4     // thread index (tid) yang diambil
5     // dari blockIdx.x
6     int tid = blockIdx.x;
7     // int tid2 = threadIdx.x;
8     // printf("cek threadIdx.x = %d \n",
9     // tid2);
10    // printf("cek blockIdx.x = %d \n",
11    // tid);
12    if (tid < N){
13       device_hasil[tid] = device_a[tid]
14       + device_b[tid];
15    }
16 }
```

Kode Program 2. vectorAdd dengan blockIdx.x



Gambar 3. Ilustrasi Proses Paralel pada vectorAdd

```
1  ////////////////////////////////
2  // set ukuran grid dan block  //
3  ////////////////////////////////
4  dim3 grid(N, 1, 1);
5  dim3 block(1, 1, 1);
6
7  // memanggil fungsi pada GPU
8  FnVectAdd << < grid, block >>
9  >(device_hasil, device_a, device_b);
10 // atau dengan FnVectAdd << < N, 1 >>
11 // >(device_hasil, device_a, device_b);
```

Kode Program 3. Set grid, block dan Fungsi ke-1

Pada vectorAddition memiliki ukuran 1 x N, atau seperti matriks dengan ordo 1 x N atau sebaliknya, yaitu dengan ukuran N x 1 seperti matriks dengan ordo N x 1 seperti pada Gambar 3. Dari dim3 grid(N, 1, 1); dapat dijabarkan berikut:

- gridDim.x = N, gridDim.y = 1, gridDim.z = 1
- banyakBlok = gridDim.x*gridDim.y*gridDim.z = N
- blockIdx.x = {0,1,...,gridDim.x-1} = {0,1,...,N-1} yang diambilkan dari grid(N,..,..), blockIdx.x, memiliki anggota sebanyak N
- blockIdx.y = {0} yang diambilkan dari grid(..,1,..), blockIdx.y, memiliki anggota sebanyak 1
- blockIdx.z = {0} yang diambilkan dari grid(..,..,1), blockIdx.z, memiliki anggota sebanyak 1

Kemudian dari dim3 block(1, 1, 1); dapat dijabarkan berikut:

- blockDim.x = 1, blockDim.y = 1, blockDim.z = 1
- banyakThread = blockDim.x*blockDim.y*blockDim.z = 1
- threadIdx.x = {0} yang diambilkan dari block(1,..,..), threadIdx.x, memiliki anggota sebanyak 1
- threadIdx.y = {0} yang diambilkan dari block(..,1,..), threadIdx.y, memiliki anggota sebanyak 1
- threadIdx.z = {0} yang diambilkan dari block(..,..,1), threadIdx.z, memiliki anggota sebanyak 1

```
1  __global__ void FnVectAdd(float
2  *device_hasil, float *device_a, float
3  *device_b){
4     // thread index (tid) yang diambil
5     // dari threadIdx.x
6     int tid = threadIdx.x;
7     // int tid2 = blockIdx.x;
8     // printf("cek blockIdx.x = %d \n",
9     // tid2);
10    // printf("cek threadIdx.x = %d \n",
11    // tid);
12    if (tid < N){
13       device_hasil[tid] = device_a[tid]
14       + device_b[tid];
15    }
16 }
```

Kode Program 4. vectorAdd dengan threadIdx.x

```
1  dim3 grid(1, 1, 1);
2  dim3 block(N, 1, 1);
3
4  // FnVectAdd << < grid, block >>
5  // >(device_hasil, device_a, device_b);
6  //atau dengan
7  FnVectAdd << < 1, N >> >(device_hasil,
8  device_a, device_b);
```

Kode Program 5. Set grid, block dan Fungsi ke-2

Dari dim3 grid(1, 1, 1); dapat dijabarkan berikut:

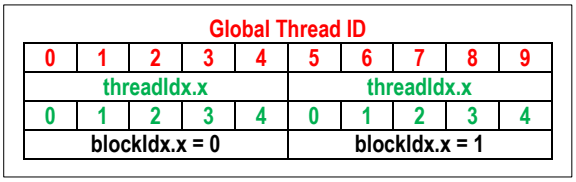
- gridDim.x = 1, gridDim.y = 1, gridDim.z = 1
- banyakBlok = gridDim.x*gridDim.y*gridDim.z = 1
- blockIdx.x = {0} yang diambilkan dari grid(1,..,..), blockIdx.x, memiliki anggota sebanyak 1

- blockIdx.y = {0} yang diambilkan dari grid(..,1,..), blockIdx.y, memiliki anggota sebanyak 1
- blockIdx.z = {0} yang diambilkan dari grid(..,..,1), blockIdx.z, memiliki anggota sebanyak 1

Kemudian dari dim3 block(N, 1, 1); dapat dijabarkan berikut:

- blockDim.x = N, blockDim.y = 1, blockDim.z = 1
- banyakThread = blockDim.x*blockDim.y*blockDim.z = N
- threadIdx.x = {0,1,...,blockDim.x-1} yang diambilkan dari block(N,..,..), threadIdx.x, memiliki anggota sebanyak N
- threadIdx.y = {0} yang diambilkan dari block(..,1,..), threadIdx.y, memiliki anggota sebanyak 1
- threadIdx.z = {0} yang diambilkan dari block(..,..,1), threadIdx.z, memiliki anggota sebanyak 1

Global Thread ID (x-*direction*), pada komputasi paralel CUDA yang melibatkan blockIdx.x, blockDim.x, dan threadIdx.x adalah berikut.



Gambar 4. Ilustrasi Global Thread ID (tid)

Pada Gambar 4, diketahui gridDim.x = 2, blockDim.x = 5, jumlah block = 2, dan jumlah thread (T) = 5. Jika dituliskan dalam bentuk set ukuran grid dan block-nya menjadi seperti "dim3 grid(2, 1, 1);" dan “dim3 block(5, 1, 1);”, serta berikut kernel yang memungkinkan untuk digunakan:

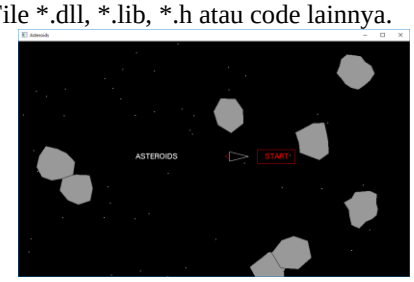
- FnvectAdd<<<1, N>>>
- FnvectAdd<<<(int)ceil(N/T),T>>>, terbaik untuk digunakan
- FnvectAdd<<<N, T>>>, atau lainnya

Global Thread ID (tid) dapat dihitung dengan “tid = blockIdx.x * blockDim.x + threadIdx.x”. Jika Global Thread ID = 7, maka blockIdx.x = 1, blockDim.x = 5, dan threadIdx.x = 2. Dan jika Global Thread ID = 4, maka blockIdx.x = 0, blockDim.x = 5, dan threadIdx.x = 4.

3. IMPLEMENTASI

Kode yang diadopsi bersumber dari link web https://github.com/RyGuyM/Asteroids dan kode tersebut telah dioptimasi dengan menggunakan pemrograman CUDA, seperti yang terlihat pada Kode Program 7. Spesifikasi *hardware*, *software*, *dependency file* yang digunakan pada saat melakukan proses implementasi:

- Core i7, RAM 8 GB.
- NVIDIA GeForce GTX 1050Ti.
- Windows 10.
- Visual Studio Community 2015.
- CUDA Toolkit 8.0.
- File *.dll, *.lib, *.h atau code lainnya.



Gambar 5. Tampilan Game Asteroids

Pada Gambar 5, game 2D Asteroids akan dilakukan proses *improvement*, dengan menggunakan GPU. Di dalam sebuah game pasti memiliki main loop. Pada main loop tersebut, sebuah proses pasti dilakukan secara berulang, dan hal inilah yang menjadi peluang untuk dilakukan pengoptimalan dengan Pemrograman CUDA, agar proses yang dilakukan dapat berjalan lebih cepat.

```
1  for (int i = 0; i < MAX_ASTEROIDS; i++)
2  {
3      if (asteroids[i].active == 1){
4         asteroids[i].x = asteroids[i].x
5         + (asteroids[i].dx);
6         asteroids[i].y = asteroids[i].y
7         + (asteroids[i].dy);
8         asteroids[i].phi =
9         asteroids[i].phi
10        + asteroids[i].dphi;
11        if (asteroids[i].x < 0){
12           asteroids[i].x = xMax;
13        } else if (asteroids[i].x >
14        xMax){
15           asteroids[i].x = 0;
16        } else if (asteroids[i].y < 0){
17           asteroids[i].y = yMax;
18        } else if (asteroids[i].y >
19        yMax){
20           asteroids[i].y = 0;
21        }
22     }
23 }
```

Kode Program 6. Kode CPU (Game Asteroids)

Pada game loop yang terdapat pada game Asteroids di atas, terdapat beberapa method untuk melakukan *update* objek. Salah satunya yaitu melakukan *update* pada array asteroids. *Update* yang dilakukan yaitu berupa *update* posisi dan rotasi dari asteroids. Kode for-loop yang menggunakan CPU untuk melakukan *update* rotasi dan posisi asteroids ada pada file “Asteroids.c”. Penjelasan Kode Program 6 adalah sebagai berikut:

1. Baris 1 for-loop untuk mengakses seluruh asteroid.
2. Baris 3 seleksi untuk merubah asteroids yang memiliki status aktif adalah *true*.
3. Baris 4 dan baris 6 untuk melakukan *update* posisi asteroids berdasarkan kecepatan masing-masing asteroids.

4. Baris 8 untuk melakukan *update* rotasi dari asteroids.
5. Barus 12-22 melakukan seleksi apabila asteroids keluar dari *border*, maka akan dipindahkan diseberangnya.

Kode di atas diubah agar dapat menggunakan GPU untuk melakukan *update* dari asteroids yang terdapat pada game. Dalam kasus ini for-loop akan dihilangkan dan akan diproses oleh masing-masing *thread* yang terdapat pada GPU. Hasil perubahan dari kode CPU di atas terdapat pada Kode Program 7.

```
1  __global__ void updateAsteroidsGPU
2  (Asteroid * asteroids, int xMax, int
3  yMax){
4      int i = threadIdx.x;
5      if (asteroids[i].active == 1){
6         asteroids[i].x = asteroids[i].x
7         + (asteroids[i].dx);
8         asteroids[i].y = asteroids[i].y
9         + (asteroids[i].dy);
10        asteroids[i].phi =
11        asteroids[i].phi +
12        asteroids[i].dphi;
13        if (asteroids[i].x < 0){
14           asteroids[i].x = xMax;
15        } else if (asteroids[i].x >
16        xMax){
17           asteroids[i].x = 0;
18        } else if (asteroids[i].y < 0){
19           asteroids[i].y = yMax;
20        } else if (asteroids[i].y >
21        yMax){
22           asteroids[i].y = 0;
23        }
24     }
25 }
```

Kode Program 7. Kode CUDA (Game Asteroids)

Penjelasan detail dari kode program CUDA:

1. Baris 1 mendeklarasikan method kernel updateAsteroidsGPU.
2. Baris 4 mendapatkan threadIdx.x kemudian dimasukkan ke variable *i*.
3. Baris 5-24 merupakan kode dari Kode Program 6 dari kode host.

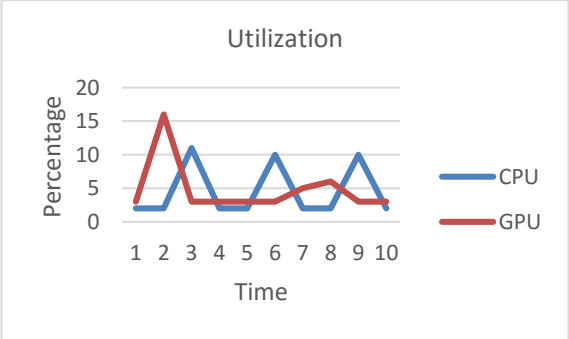
Hasil proses *running* ketika dijalankan pada Visual Studio dengan menggunakan GPU dapat dilihat keluarannya pada Gambar 6.

Gambar 6. Hasil Running dengan GPU

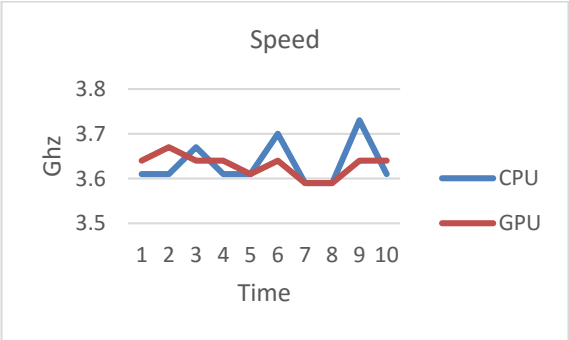
4. PENGUJIAN DAN ANALISIS

Berdasarkan hasil pengujian CPU dan dengan menggunakan GPU, dapat dilihat hasilnya secara visualisasi. Berdasarkan grafik yang ditunjukkan oleh Gambar 7-11, dapat dianalisis bahwa hasil nilai-nilai *utilization*, *speed*, *processes*, *thread*, dan *handles* yang menggunakan GPU relatif lebih kecil daripada kode yang di eksekusi menggunakan CPU.

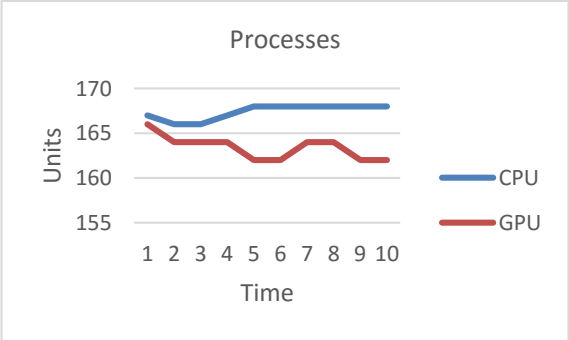
Berdasarkan data yang telah didapatkan dari pengujian, hasil percobaan menunjukkan bahwa implementasi dengan menggunakan pemrograman GPU dapat mengurangi *utilization* pada CPU. Dan pada percobaan menggunakan hanya CPU, terkadang terjadi *spiking*. Sedangkan ketika dilakukan implementasi dengan GPU, *utilization* dari CPU relatif stabil dan tidak terlihat adanya peningkatan kinerja yang signifikan. Hasil yang hampir sama juga terjadi pada *speed*, lalu pada *processes*, *threads*, dan *handles* juga, bahwa dengan GPU itu relatif lebih rendah dibandingkan CPU tanpa GPU.



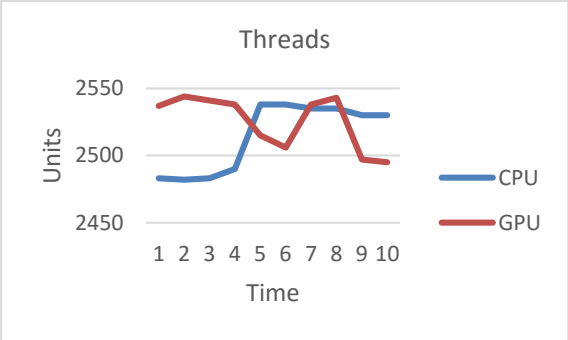
Gambar 7. Grafik perbandingan *utilization*



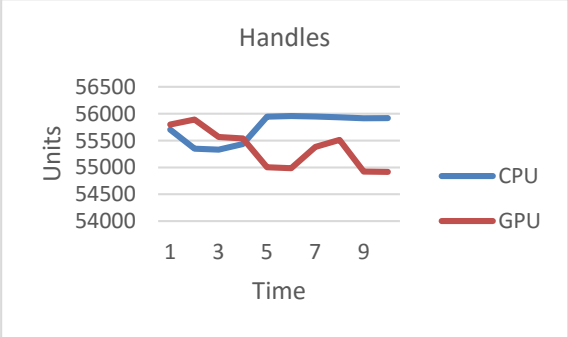
Gambar 8. Grafik perbandingan *speed*



Gambar 9. Grafik perbandingan *processes*



Gambar 10. Grafik perbandingan *threads*



Gambar 11. Grafik perbandingan *handles*

5. KESIMPULAN DAN SARAN

Percobaan untuk melakukan peningkatan performa pada permainan Asteroids berhasil dengan memanfaatkan CUDA sebagai GPU *processing*. Pengujian dilakukan dengan merubah proses *update* dari objek yang di-*render* pada layar permainan. Hasil perubahan menggunakan GPU berdampak pada penggunaan sumber daya yang lebih ringan apabila dibandingkan dengan implementasi asli.

Hasil percobaan dapat disimpulkan bahwa implementasi GPU dapat mengurangi *utilization* pada CPU. Pada percobaan CPU tanpa GPU, penggunaan *CPU* kadang terjadi *spiking*. Sedangkan pada implementasi GPU, *utilization* dari CPU relatif stabil. Hal yang sama juga terjadi pada *speed.* Kemudian pada bagian *processes*, *threads*, dan *handles* penggunaan dengan GPU relatif lebih rendah dibandingkan penggunaan CPU tanpa GPU. Kemudian untuk hasil yang didapatkan dari GPU adalah tidak adanya aktifitas yang signifikan pada CPU, karena proses yang awalnya berjalan pada CPU dipindahkan ke GPU. Akibat implementasi tersebut maka hasil pengoptimalan dapat tercapai. Sehingga dapat disimpulkan bahwa penggunaan GPU dapat meningkatkan performa dibandingkan dengan implementasi CPU, meskipun tidak terlihat peningkatan performa secara signifikan, terutama dari segi tampilan grafis 2D. Dan saran untuk penelitian selanjutnya adalah meningkatkan performa dari *video game* dengan mengimplementasikan seluruh *update* yang dilakukan oleh CPU dengan GPU, seperti *collision detection*, *particle*, dan objek-objek lain yang ada pada *video game*.

6. DAFTAR PUSTAKA

BAKHODA, A., YUAN, G. L., FUNG, W. L., WONG, H., AAMODT, T. M., 2009, Analyzing CUDA Workloads Using a Detailed GPU Simulator. *University of British Columbia*.

CHE, B., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., SKADRON., K., 2008, A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA, *Parallel and Distributed Computing*

KECKLER, S. W., DALLY, W.J., KHAILANY, B., GARLAND, M., GLASCO. D., 2011, GPUs and The Future of Parallel Computing. *IEEE Computer Society*, 7-17.

LEE, J. H., CLARKE, R. I., KARLOVA, N., THORNTON, K., & PERTI, A. 2014. Facet Analysis of Video Game Genres. *iConference 2014*, 131.

MIVULE, K., HARVEY, B., COBB, C., SAYED, H. E., 2014, A Review of CUDA, MapReduce, and Pthreads Parallel Computing Models. *Bowie State University*.

NICKOLLS, J., DALLY, W. J., 2010, The GPU Computing Era. *IEEE Computer Society*, 56-69.

PRATO, G. D., FEIJOO, C., & SIMON, J.-P. 2014. Innovations in the Video Game Industry: Changing Global Markets. *Digiworld Economic Journal*, 17-18.

RYOO, SHANE, RODRIGUES, CHRISTOPHER I., BAGHSORKHI, SARA S., STONE, SAM S., KIRK, DAVID B., HWU, WEN-MEI W., Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA, *PPoPP '08 Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 73-82.

WERKHOVEN, B. V., MAASSEN, J., SEINSTRA, F. J., 2011, Optimizing Convolution Operations in CUDA with Adaptive Tiling. *VU University*.