BAB V
KESIMPULAN DAN SARAN

A. Kesimpulan

Berdasarkan hasil pengujian dapat disimpulkan:

1. Telah berhasil dikembangkan program untuk melakukan proses *inpainting* citra digital menggunakan PDP orde keempat berbasis CPU dan GPU dengan bahasa pemrograman C++. Hal ini dikarenakan kecocokan pemrosesan PDP dengan pemrograman secara paralel.
2. Pemrosesan secara paralel dengan GPU untuk kerusakan yang lebar menunjukan ada peningkatan yang lebih baik dibandingkan dengan CPU. Namun pada kasus yang *inpainting* yang terdapat kelengkungan, pemrosesan paralel kurang memberikan kualitas yang lebih baik dibandingkan dengan CPU.
3. Berdasarkan beberapa pengujian dari citra uji dapat disimpulkan bahwa, menggunakan GPU NVDIA dapat mempercepat proses komputasi secara signifikan sampai dengan 48x dibandingkan dengan CPU.

B. Saran

Program solusi *inpainting* dapat dikembangkan lebih lanjut dengan peningkatan seperti:

1. Penanganan *inpainting* secara paralel untuk kasus kelengkungan garis.
2. Penggunaan texture dan share memori yang lebih efisien
3. Penggunaan *inpainting* tidak hanya untuk kasus gambar foto saja, namun dapat ditingkatkan untuk proses *inpainting* video.

DAFTAR PUSTAKA

Bertalmio, M., Sapiro, G., Caselles, V. & Ballester, C., 2000. Image inpainting. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp.417-24.

Bertozzi, A.L., Esedoglu, S. & Gillette, A., 2007. Inpainting of binary images using the Cahn-Hilliard equation. *IEEE Transactions on image processing*, 16(1), pp.285-91.

Blake, G., Dreslinski, R.G. & Mudge, T., 2009. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6), pp.26-37.

Bosch, J., Kay, D., Stoll, M. & Wathen, A.J., 2014. Fast Solvers for Cahn--Hilliard Inpainting. *SIAM Journal on Imaging Sciences*, 7(1), pp.67-97.

Burger, M., He, L. & Schönlieb, C.-B., 2009. Cahn-Hilliard inpainting and a generalization for grayvalue images. *SIAM Journal on Imaging Sciences*, 2(4), pp.1129-67.

Cahn, J.W. & Hilliard, J.E., 1958. Free energy of a nonuniform system. I. Interfacial free energy. *The Journal of chemical physics*, 28(2), pp.258-67.

Chang, L. & Chongxiu, Y., 2011. New interpolation algorithm for image inpainting. *Physics Procedia*, 22, pp.107-11.

Chan, T.F., Kang, S.H. & Shen, J., 2002. Euler's elastica and curvature-based image inpainting. *SIAM Journal on Applied Mathematics*, 63(2), pp.564-92.

Chan, T. & Shen, J., 2001. Mathematical models for local nontexture inpaintings. *SIAM Journal on Applied Mathematics*, 62(3), pp.1019–43.

Chapra, S.C. & Canale, R.P., 1998. *Numerical methods for engineers*. 3rd ed. McGraw-Hill.

Chapra, S.C. & Canale, R.P., 2013. *Numerical methods for engineers*. 7th ed. McGraw-Hill.

Diaz, J., Mun˜oz-Caro, C. & Nin˜o, A., 2012. A Survey of Parallel Programming Models and Tools in The Multi and Many Core Era. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 23(8), pp.1369-86.

Du, P. et al., 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8), pp.391-407.

Ebrahimi, M., Holst, M. & Lunasin, E., 2012. The Navier–Stokes–Voight model for image inpainting. *IMA Journal of Applied Mathematics*, 78, pp.869–94.

Emile-Male, G., 1976. *The Restorer's Handbook of Easel Painting*. New York: Van Nostrand Reinhold.

Esedoglu, S. & Shen, J., 2002. Digital inpainting based on the Mumford–Shah–Euler image model. *European Journal of Applied Mathematics*, 13(4), pp.353-70.

Evans, L.C., 2002. Partial differential equations, vol. 19 of Graduate Studies in Mathematics. *Stud. Math., AMS, Providence*, 19.

GaryBradski & Kaebler, A., 2008. *Computer vision with the OpenCV library*. O'Reilly.

Gelsinger, P., 2006. Moore's Law–The Genius Lives On. *Solid-State Circuits Society Newsletter, IEEE*, 11(5), pp.18-20.

Geman, S. & Geman, D., 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6), pp.721-41.

Gillette, A., 2006. *Image inpainting using a modified Cahn-Hilliard equation*. Doctoral dissertation. University of California Los Angeles.

Gonzalez, R.C. & Woods, R.E., 2008. *Digital Image Processing*. 3rd ed. Pearson Education, Inc.

Greer, J.B., Bertozzi, A.L. & Sapiro, G., 2006. Fourth order partial differential equations on general geometries. *Journal of Computational Physics*, 216(1), pp.216-46.

Hore, A. & Ziou, D., 2010. Image Quality Metrics: PSNR vs. SSIM. *ICPR*, 34, pp.2366-69.

Ismail Avcıbas, B.S.K.S., 2002. Statistical evaluation of image quality measures. *Journal of Electronic Imaging*, 11(2), pp.206-23.

Kasim, H., March, V., Zhang, R. & See, S., 2008. Survey on parallel programming model. *Network and Parallel Computing*, pp.266-75.

Kruger, J., Kipfer, P., Konclratieva, P. & Westermann, R., 2005. A particle system for interactive visualization of 3D flows. *Visualization and Computer Graphics, IEEE Transactions on*, 11(6), pp.744-56.

Li, F., Shen, C., Liu, R. & Fan, J., 2011. A fast implementation algorithm of TV inpainting model based on operator splitting method. *Computers & Electrical Engineering*, 37(5), pp.782-88.

Mumford & Shah, 1989. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on pure and applied mathematics*, 42(5), pp.577-685.

Nvidia, 2014. *CUDA C Programming Guide version 6.5*. Nvdia Corporation.

Owens, J.D. et al., 2008. GPU computing. *Proceedings of the IEEE*, 96(5), pp.879-99.

Owens, J.D. et al., 2007. A Survey of general-purpose computation on graphics hardware. *Computer graphics forum*, 26(1), pp.80-113.

Perona, P. & Malik, J., 1990. Scale-space and edge detection using anisotropic diffusion. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(7), pp.629-39.

Ravi, S., Pasupathi, P., Muthukumar, S. & Krishnan, N., 2013. Image in-painting techniques-A survey and analysis. In *Innovations in Information Technology (IIT), 2013 9th International Conference on*. Abu Dhabi, 2013. IEEE.

Scho□nlieb, C.-B., 2009. *Modern pde techniques for image inpainting*. Doctoral dissertation. University of Cambridge.

Schönlieb, C.-b. & Bertozzi, A., 2011. Unconditionally stable schemes for higher order inpainting. *Communications in Mathematical Sciences*, 9(2), pp.413-57.

Walden, S., 1985. *The Ravished Image*. New York: St. Martin's Press.

Zhang, X. & Chan, T.F., 2010. Wavelet inpainting by nonlocal total variation. *Inverse problems and Imaging*, 4(1), pp.191-210.

Zhang, N., Chen, Y.-s. & Wang, J., 2010. Image parallel processing based on GPU. In *Advanced Computer Control (ICACC), 2010 2nd International Conference on*., 2010. IEEE.

LAMPIRAN

Lampiran 1. Pemrograman Secara Umum CPU

Sebelum dapat melakukan pemrograman untuk kasus ini terlebih dahulu dibutuhkan koneksi terhadap *library* OpenCV untuk pemrograman berbasis CPU. Untuk melakukan hal ini maka dibutuhkan kode sebagai berikut:

```
#include "opencv2\core\core.hpp"
#include "opencv2\highgui\highgui.hpp"
```

Kode Library OpenCV

Dalam kode tersebut digunakan komponen core dan highgui pada *library* openCV. Digunakan komponen *core.hpp* dari untuk melakukan akses ke kelas dasar, contohnya yaitu kelas Mat yang akan digunakan sebagai penampung matriks citra digital. Komponen core ini juga membantu memangil fungsi DFT yang bertujuan untuk transformasi dari ruang 2D ke ruang frekuensi. Selain itu membutuhkan highgui.hpp untuk melakukan operasi input output seperti *imread, imwrite* dan *imshow*. Fungsi *imread* bertujuan untuk membaca citra digital sebagai matriks yang siap diolah, *imwrite* bertujuan untuk merubah matriks menjadi citra digital dan *imshow* bertujuan untuk menampilkan matriks ke layar monitor.

```
#include <iostream>
#include <iomanip>
#include <chrono>
```

Kode Library C++

Kode juga menggunakan komponen dari C++ pada VS2013 yaitu *iostream, iomanip* dan *chrono*. Komponen iostream digunakan untuk melakukan operasi input dan output, pada kode ini digunakan untuk membaca perintah cout pada bahasa pemrograman C++. Sedangkan chrono dan iomanip digunakan sebagai komponen untuk menghitung waktu ekseskusi dan menampilkan waktu sesuai dengan format yang diinginkan. Komponen *chrono*

hanya terdapat pada VS2013 dan tidak terdapat pada versi sebelumnya. Jika menggunakan visual studio versi sebelumnya maka harus melakuan penambahan sendiri *library chrono*.

Kelas Mat dapat mempunyai kanal lebih dari satu. Kanal tersebut biasanya digunakan menyimpan 3 kanal warna citra digital yaitu matriks pada kanal warna merah, warna hijau dan warna biru (RGB). kelas Mat juga dapat menyimpan satu kanal saja, yaitu pada saat menyimpan citra *grayscale* atau hitam putih.

Kelebihan kelas Mat yang dapat mempuyai lebih dari satu kanal tersebut juga dapat digunakan untuk menyimpan bilangan kompleks. Pada bilangan kompleks digunakan kelas Mat dengan dua kanal, hal tersebut unuk menyimpan nilai riil dan nilai imajiner. Perkalian bilangan kompleks membutuhkan perlakuan khusus untuk itu dibutuhkan kode untuk menangani perhitungan bilangan kompleks.

```
split(curvbar, complexLapCurvbar);
complexLapCurvbar[0] = complexLapCurvbar[0].mul(Denominator);
complexLapCurvbar[1] = complexLapCurvbar[1].mul(Denominator);
merge(complexLapCurvbar, 2, LapCurvbar);
```

Kode Pemisahan Bilangan Kompleks pada CPU

Pada fungsi split bertujuan untuk memisahkan matriks dalam beberapa kanal. Kanal matriks dapat diakses seperti mengakses sebuah array, namun jika array yang diakses langsung pada nilai data, pada kode ini akan mengakses matriks suatu kanal. Pemisahan ini bertujuan untuk memisahkan matriks bilangan riil dan matriks bilangan imajiner. Setelah operasi perkalian dilakukan maka kembali disatukan dengan perintah merge.

Pemrograman kasus ini akan menggunakan banyak bilangan kompleks untuk melakukan perhitungan, hal ini dikarenakan adanya transformasi dari dimensi ruang ke dimensi frekuensi oleh fungsi DFT (*discret fourier transform*). Dengan transformasi ke ruang frekuensi maka proses konvolusi yang membutuhkan banyak perhitungan pada dimensi ruang dapat dikerjakan hanya seperti perkalian biasa. Hal ini akan dapat menghemat waktu perhitungan.

Lampiran 2. Pemrograman Secara Umum GPU

Untuk kasus pemrograman secara paralel pada GPU dibutuhkan koneksi terhadap *library* CUDA dan OpenCV. *Library* OpenCV digunakan untuk menampung data matriks yang nantinya akan diproses oleh GPU setelah selesai akan kembali ditampilkan dengan bantuan *library* tersebut. Untuk melakukan hal ini maka dibutuhkan kode sebagai berikut:

```
#include "opencv2\opencv.hpp"
#include "opencv2\core\core.hpp"
#include "opencv2\highgui\highgui.hpp"
#include "cuda_runtime.h"
#include "cufft.h
```

Kode Library OpenCV dan CUDA

Dalam kode tersebut digunakan komponen *opencv.hpp, core.hpp* dan *highgui.hpp* pada *library* openCV. Komponen *core* dan *highgui* mempunyai penjelasan yang sama pada bab sebelumnya. Untuk kasus ini menggunakan *opencv.hpp* untuk mengaktifkan fungsi `cvtColor` yang mempunyai efek untuk merubah struktur *pointer* dari matriks pada kelas Mat sehingga selaras dengan pointer pemrosesan pada *library* CUDA. Komponen cuda_runtime bertujuan mengaktifkan fungsi umum pada cuda dan cufft berfungsi mengaktifkan fungsi cuda untuk perhitungan pada Transformasi Fourier.

```
Mat src = imread("original.png", CV_LOAD_IMAGE_COLOR);
cvtColor(src, src, CV_BGR2GRAY);
unsigned char * h_src = (unsigned char*)src.data;
// *h_src siap diproses dengan library CUDA
```

Kode Pemanggilan Citra GPU

Untuk mencocokan tipe data antar *library* tentu membutuhkan waktu yang lama. Bahkan untuk itu kadang dibutuhkan suatu fungsi tambahan untuk melakukan transfer data. Perpindahan data dari *library* OpenCV dan CUDA juga menjadi suatu permasalahan tersendiri. Namun fungsi *cvtColor* mempunyai efek merubah matriks menjadi susunan yang

bisa digunakan oleh CUDA dengan baik. Oleh karena itu kita dapat memanfaatkan fungsi tersebut untuk kemudahan dan kesederhanaan kode.

Pada pemrograman paralel dikenal CPU sebagai *host* dan GPU sebagai *device*. Data yang akan diproses harus di transfer dari *host* ke *device* untuk diproses. Untuk melakukan itu dibutuhkan perintah *cudaMemcpy(output, input, size, cudaMemcpyHostToDevice)*. Variabel *output* adalah variabel yang ada pada *device*. Sebelumnya variabel ini harus terlebih dahulu diinisialisasi melalui perintah *cudaMalloc(output, size)*. Variabel *size* merupakan jumlah alokasi yang dipesan pada suatu lokasi memori. Setelah pemrosessan selesai, data akan dipindahkan dari *device* ke *host* kembali. Data yang akan diproses harus di transfer dari *host* ke *device* untuk diproses. Untuk melakukan itu dibutuhkan perintah *cudaMemcpy(output, input, size, cudaMemcpyDeviceToHost)*. Perbedaan dari perintah sebelumnya adalah arah yang terjadi, yaitu sebelumnya dari *host* ke *device* (*cudaMemcpyHostToDevice*) berubah menjadi dari *device* ke *host*(cudaMemcpyDeviceToHost).

```
int sizedouble = sizeof(cufftDoubleReal)*sizedata;
checkCudaErrors(cudaMalloc(&d_u, sizedouble));
checkCudaErrors(cudaMemcpy(d_u, h_src, sizedouble, cudaMemcpyHostToDevice));
...
... // Pemrosesan data
...
checkCudaErrors(cudaMemcpy(h_output, d_u, sizedouble, cudaMemcpyDeviceToHost));
```

Kode Transfer data GPU

Setelah data yang akan diolah di pindah pada *device* (GPU), akan dilakukan proses perhitungan pada GPU. Untuk itu GPU sebelumnya membuat kernel dengan susunan sebagai berikut:

```
__global__
void computeLU(double *u, double* lambda, cufftDoubleComplex* lubar, int numRows,
int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
```

```
        if ((row >= numRows) || (col >= numCols)){
                return;
        }
        double value = 0;
        int index = col + numCols * row;          // current pixel index
        value = u[index] * lambda[index];
        lubar[index].x= value;
        lubar[index].y = 0;
}
```

Kode Contoh Kernel GPU

Variabel __global__ merupakan penanda akan membuat sebuah matriks pada kernel GPU. Setiap fungsi yang melibatkan proses secara paralel pada GPU biasanya akan diawali dengan variabel ini. Didalam fungsi ini akan dibentuk variabel *row* dan *col* untuk sebagai penanda proses paralel yang terjadi. Jika pemrosesan sudah dilakukan melebihi jumlah baris dan kolom maka tread yang sudah berjalan secara paralel akan di berhentikan. Sedangkan jika pemrosesan ada dalam radius pemrosesan maka data akan diproses. Perhitungan indek akan menjadi sedikit berbeda, karena proses paralel tidak mengenal tipe matriks 2 dimensi. Untuk itu semua data yang masuk akan dianggap sebagai data serial. Oleh karena itu pengaksesan indek menjadi sedikit berbeda yaitu: nomor pemrosesan kolom ditambah dengan nomor pemrosesan baris dikali dengan jumlah kolom *(col + numCols*row).*

Kernel tersebut harus dipanggil menggunakan cara yang tepat untuk melakukan pemrosesan secara paralel yaitu:

```
const dim3 blockSize(32, 32, 1);
const dim3 gridSize(100,100 , 1);
computeLU << <gridSize, blockSize >> >(d_u, d_lambda, cLU, numRows, numCols);
```

Kode Pemanggilan kernel GPU

Dengan menggunakan cara tersebut akan membuat pemrosesan secara paralel sebanyak 32x32x100x100 yaitu 10.240.000 *thread*. Sehingga kernel computeLU akan dijalankan sebanyak 10.240.000 dalam waktu yang hampir bersamaan.

```
const dim3 blockSize(32, 32, 1);
const dim3 gridSize((numCols / 32) + 1, (numRows / 32) + 1, 1);
computeLU << <gridSize, blockSize >> >(d_u, d_lambda, cLU, numRows, numCols);
```

Kode Pemanggilan kernel GPU efisien

Untuk melakukan efisiensi maka biasanya dilakukan cara pembagian dimensi citra dengan *blockzise* yang kemudian ditambahkan dengan nilai 1. Penambahan nilai 1 dilakukan untuk melakukan pembulatan ke atas, sehingga tidak terjadi data citra belum diproses. Dengan cara ini jumlah *thread* yang tidak melakukan pekerjaan akan menjadi *minimum*, sehingga pemrosesan secara paralel akan menjadi lebih efisien.

Pemrosesan dari dimensi ruang ke dimensi frekuensi menggunakan *library* cufft. Hal ini kemudian dipanggil dengan memangil *handle* untuk perubahan tersebut. Handle akan digunakan untuk membuat plan pemrosesan 2d (*cufftPlan2d*). untuk pemrosesan 1 dimensi maka menggunakan *cufftPlan1d*. Suatu plan hanya dapat dibunakan untuk melakukan suatu proses saja, maka dibutuhkan 2 plan untuk melakukan proses *Fast Fourier Transform (FFT*) dan *inversnya*(IFFT). Pada kode diberi nama *pFFT* dan *pIFFT*. Pada plan yang dibuat terdapat digunakan parameter *CUFFT_Z2Z*. Hal ini menandakan bahwa akan dilakukan perubahan nilai dari bilangan *double* komplek dari dimensi ruang ke bilangan *double* komplek pada dimensi frekuensi. Parameter ini dapat diganti menjadi *CUFFT_C2C* untuk presisi tipe data *float*. Pemilihan parameter ini menggunakan komplek ke komplek dengan tipe data *double* dikarenakan dengan penggunaan parameter ini jumlah data inputan dan output akan tetap. Sedangkan untuk tipe parameter *CUFFT_R2C* (riil ke komplek) atau *CUFFT_C2R* (komplek ke riil) akan merubah jumlah data sehingga membingungkan dalam operasi citra digital.

Setelah plan sudah dibuat maka dapat dilakukan eksekusi transfer dari dimensi ruang ke dimensi frekuensi dengan perintah *cufftExecZ2Z* yang menggunakan parameter

CUFFT_FORWARD. Untuk melakukan inverse dengan perintah yang sama namun dengan menggunakan paramenter *CUFFT_INVERSE*.

```
cufftHandle pFFT;
cufftHandle pIFFT;
if (cufftPlan2d(&pFFT,numRows,numCols, CUFFT_Z2Z) != CUFFT_SUCCESS){
       fprintf(stderr, "CUFFT error:01.FFT Plan creation Real 2 Complex failed");
       return;
       }

if (cufftPlan2d(&pIFFT,numRows,numCols, CUFFT_Z2Z) != CUFFT_SUCCESS){
       fprintf(stderr, "CUFFT error:02.IFFT Plan creation Complex 2 Real failed");
       return;
       }
if (cufftExecZ2Z(pFFT, cLU, cLUbar, CUFFT_FORWARD) != CUFFT_SUCCESS){
       fprintf(stderr, "CUFFT error: Exec FFT lubar to Complex lubar failed");
       return;
       }
if (cufftExecZ2Z(pIFFT, cresult2, cresult2, CUFFT_INVERSE) != CUFFT_SUCCESS){
       fprintf(stderr, "CUFFT error: Exec IFFT u to Complex result failed");
       return;
       };
```

Kode Pembuatan plan FFT GPU

Pada ruang frekuensi bilangan riil berubah menjadi bilangan komplek. Untuk melakukan pemrosesan bilangan komplek *library* CUDA belum menyediakan. Oleh karena itu untuk mengatasi permasalahan ini dibutuhkan fungsi baru untuk menangani operator penjumlahan bilangan kompleks yaitu penambahan, pengurangan, perkalian dan pembagian. Fungsi ini hanya akan dipanggil dalam kernel GPU saja. Untuk membuat kernel ini mampu dipanggil dalam kernel GPU maka dibutuhkan awalan satic __device__ __host __ inline.

```
static __device__ __host__ inline cufftDoubleComplex
ComplexScale(cufftDoubleComplex a, double s)
{
       cufftDoubleComplex c;
       c.x = s * a.x;
       c.y = s * a.y;
       return c;
}
static __device__ __host__ inline cufftDoubleComplex
ComplexSub(cufftDoubleComplex a, cufftDoubleComplex b)
{
       cufftDoubleComplex c;
       c.x = a.x - b.x;
       c.y = a.y - b.y;
       return c;
}
static __device__ __host__ inline cufftDoubleComplex
ComplexAdd(cufftDoubleComplex a, cufftDoubleComplex b)
```

```
{
        cufftDoubleComplex c;
        c.x = a.x + b.x;
        c.y = a.y + b.y;
        return c;
}
static __device__ __host__ inline cufftDoubleComplex
ComplexScaleDiv(cufftDoubleComplex a, double s)
{
        cufftDoubleComplex c;
        c.x = a.x/s;
        c.y = a.y/s;
        return c;
}
```

Kode Fungsi perhitungan bilangan kompleks GPU

Fungsi tersebut akan membantu dalam melakukan pemrosesan bilangan kompleks yaitu perkalian, pembagian, penjumlahan dan pengurangan.

Lampiran 3. Pemrograman *Inpainting* berbasis CPU

1) Inisialisasi Gambar dan Mask

```
int vartype = CV_64FC1;
Mat src = imread("input.png", CV_LOAD_IMAGE_GRAYSCALE);
Mat mask = imread("input-mask_i.png", CV_LOAD_IMAGE_GRAYSCALE);
src.convertTo(src, vartype);
mask.convertTo(mask, vartype);
```

Kode Load Gambar CPU

Membaca *file* inputan *file* yang mengalami kerusakan dan *file* mask dari kerusakan. Kemudian dilakukan perubahan tipe data dari tipe data awal menjadi double dengan satu kanal. Hal ini bisa diketahui dari tipe variabel yang digunakan untuk melakukan konversi yaitu *CV_64FC1*. *CV* merupakan penanda menggunakan variabel dari opencv, sedangkan nilai *64F* merupakan penanda tipe data double, sedangkan *C1* merupakan penanda menggunakan satu kanal.

2) Inisialisasi parameter perhitngan

```
int dt = 100;
double epsilon = 0.005;
double ep2 = pow(epsilon , 2);
int lpower = 2;
double lambda0 = 100;
lambda = lambda.mul(lambda0);
double c1 = 200;
double c2 = 100;
int Itermax = 2000;
```

Kode Inisialisasi parameter CPU

Melakukan inisialisasi parameter untuk melakukan perhitungan *inpainting*, antara lain: *Itermax* untuk jumlah iterasi, *dt* untuk delta t, *epsilon* untuk epsilon, *ep2* untuk epsilon kuadrat, *lambda0* untuk nilai penguatan, konstanta *C1* dan *C2*. Nilai *lambda0* digunakan untuk menguatkan gambar dari mask yang pada kode di beri nama *lambda*.

3) Menghitung Denominator

```
Mat Lambda1 = Mat::zeros(rows, rows, vartype);
Mat Lambda2 = Mat::zeros(cols, cols, vartype);
```

```
for (int j = 0; j<rows; j++){
	Lambda1.at<double>(j, j) = 2 * (cos(2 * j * PI / rows) - 1);
}

for (int j = 0; j<cols; j++){
	Lambda2.at<double>(j, j) = 2 * (cos(2 * j * PI / cols) - 1);
}
int h1 = 1;
int h2 = 1;
Mat Denominator = 1 / (h1*h1)*(Lambda1*Mat::ones(rows, cols, vartype)) + 1 /
(h2*h2)*(Mat::ones(rows, cols, vartype)*Lambda2);
```

Kode Menghitung denominator CPU

Fungsi menghitung denominator yang berasal dari fungsi cos. Denominator ini nantinya akan berguna untuk mencari nilai laplasian dari matriks *curve*.

4) Transformasi gambar dan mask pada ruang frekuensi

```
Mat ubar, uabar, LUbar, LU0bar, LapCurvbar,curvbar;
Mat curv,alpha,LU,tesfft,ivtes;
Mat complexLapCurvbar[] = { Mat::zeros(u.size(), vartype), Mat::zeros(u.size(),
vartype) };
Mat complexubar[] = { Mat::zeros(u.size(), vartype), Mat::zeros(u.size(),
vartype) };

dft(u, ubar, DFT_COMPLEX_OUTPUT);
dft(u.mul(lambda), LU0bar, DFT_COMPLEX_OUTPUT);
LUbar = LU0bar.clone();
```

Kode Transformasi FFT CPU

Matriks gambar asal akan ditransformasi ke dalam ruang frekuensi dengan fungsi *FFT* kemudian akan diberi nama *ubar*. Matriks mask sebelum ditrasformasi ke dalam ruang frekuensi akan di kuatkan terlebih dahulu dengan dikalikan dengan 100. Setelah itu matriks mask akan dikenai perkalian per elemen dengan matriks gambar asal, hasil dari perkalian tersebut kemudian akan ditransformasi dalam ruang frekuensi dengan *FFT* kemudian akan diberi nama *lu0bar*. Yang merupakan penanda bahwa merupakan mask awal.

5) Menghitung *Curve*

```
Mat ComputeCurve(Mat src, int rows, int cols, int vartype)
```

```
{
	Mat result = Mat::zeros(rows, cols, vartype);
	Mat ux = Mat::zeros(rows, cols, vartype);

	double epsilon = 0.005;
	double ep2 = pow(epsilon, 2);
	int w, n, s, e = 0;
	for (int i = 0; i < rows; i++)
	{
		for (int j = 0; j < cols; j++)
		{
			e = j + 1;
			w = j - 1;

			n = i - 1;
			s = i + 1;

			w <= 0 ? w = 0 : w = w;
			e>cols - 1 ? e = cols - 1 : e = e;
			n <= 0 ? n = 0 : n = n;
			s > rows - 1 ? s = rows - 1 : s = s;

			double x = (src.at<double>(i, e) - src.at<double>(i, w)) / 2;
			double y = (src.at<double>(s, j) - src.at<double>(n, j)) / 2;
			double xx = src.at<double>(i, w) + src.at<double>(i, e) - 2 *
			 src.at<double>(i, j);
			double yy = src.at<double>(s, j) + src.at<double>(n, j) - 2 *
			 src.at<double>(i, j);
			double dp = src.at<double>(n, w) + src.at<double>(s, e);
			double dm = src.at<double>(n, e) + src.at<double>(s, w);
			double xy = (dp - dm) / 4;
			double num = xx*(ep2 + y*y) - 2 * x*y*xy + yy*(ep2 + x*x);;
			double den = pow((ep2 + x*x + y*y), 1.5);
			double curve = num / den;
			result.at<double>(i, j) =curve;
		}
	}
	return result;
}
```

Kode Menghitung Curve CPU

Fungsi ini menghitung nilai *curve* dari suatu matriks. Nilai ini terdiri dari hasil selisih dari nilai pada matriks di sekitarnya.

6) Transformasi nilai *Curve* ke ruang frekuesi

```
curv = ComputeCurve(u, src.rows, src.cols, vartype);
dft(curv, curvbar,DFT_COMPLEX_OUTPUT);
```

Kode Transformasi FFT Curve CPU

Hasil dari *curve* kemudian ditransformasi ke ruangan frekuensi untuk memudahkan perhitungan. Kemudian diberi nama curvebar.

7) Menghitung nilai Laplasian dan curvebar.

```
split(curvbar, complexLapCurvbar);
complexLapCurvbar[0] = complexLapCurvbar[0].mul(Denominator);
complexLapCurvbar[1] = complexLapCurvbar[1].mul(Denominator);
merge(complexLapCurvbar, 2, LapCurvbar);
```

Kode Perhitungan bilangan Kompleks

Nilai *curvebar* kemudian akan dikali dengan Denominator untuk menghasilkan nilai Laplasian dari *curvebar* yang akan disebut *Lapcurvbar*.

8) Update ubar

```
split(ubar, complexubar);
complexubar[0] = complexubar[0].mul(alpha);
complexubar[1] = complexubar[1].mul(alpha);
merge(complexubar, 2, ubar);

ubar = ubar - dt*LapCurvbar + dt*(LU0bar - LUbar);
tesfft = dt*LapCurvbar;
split(ubar, complexubar);
divide( complexubar[0],alpha, complexubar[0]);
divide(complexubar[1], alpha, complexubar[1]);
merge(complexubar, 2, ubar);
```

Kode Perhitungan Ubar

Update nilai ubar dari perhitungan dengan menggunakan komponen *lapcurvar, lubar, lu0bar* sesuai dengan Persamaan 19.

9) Kembalikan nilai dari dimensi frekuensi ke dimensi ruang 2D

```
idft(ubar, u, DFT_SCALE | DFT_REAL_OUTPUT); // Applying IDFT
```

Kode Tranformasi IFFT CPU

Melakukan proses *inverse* dari dimensi frekuensi ke dimensi ruang 2d. Sehingga data akan berubah menjadi matriks yang siap untuk ditampilkan.

10) Tampilkan dan simpan hasil

```
u.convertTo(u, CV_8UC1, 255.0);
imshow("Cahn Hilliard high order Inpainting", u);
imwrite("output.png", u);
waitKey(0);
```

Kode Penyimpanan data CPU

Melakukan konversi ke tipe data sesuai dengan kebutuhan. Untuk tipe data citra biasanya merupakan tipe data U (*unsigned char*) pada OpenCV dan menggunakan satu kanal. Setelah itu data matriks disimpan menjadi *file* output.png dengan perintah *imwrite*. Perintah *waitkey(0)* merupakan perintah untuk menunggu sehingga program tidak langsung menutup ketika aplikasi selesai dieksekusi.

Lampiran 4. Pemrograman Inpainting berbasis GPU

1) *Inisialisasi Gambar dan Mask*

```
Mat src = imread("original.png", CV_LOAD_IMAGE_COLOR);
Mat mask = imread("mask.png", CV_LOAD_IMAGE_COLOR);
```

Kode Load Gambar pada GPU

Pada langkah awal dilakukan proses pembacaan *file* citra yang akan dikenai proses *inpainting* dan *file* citra mask perbaikan proses *inpainting*. *File* citra tersebut dengan bantuan *library* OpenCV akan dibaca oleh komputer menjadi matriks citra *inpainting* dan matriks mask *inpainting* yang siap diolah. Proses ini dilakukan pada CPU.

2) Inisialisasi denominator

```
double *h_denominator = (double*)Denominator.data;
```

Kode Inisialisasi denominator GPU

Melakukan inisialisasi parameter untuk melakukan perhitungan *inpainting* denominator. Hal ini diset pada pada CPU untuk kemudian siap ditranfer ke GPU untuk melakukan perhitungan data.

3) Merubah tipe data

```
gpu_image2double(h_src, h_lamda, h_doubleSrc, h_doubleLamda, rows, cols);
```

Kode Fungsi merubah variabel GPU

Fungsi akan memanggil kernel fungsi pada GPU untuk melakukan perubahan tipe data untuk kemudian siap diproses pada fungsi *inpainting* berikutnya. Fungsi ini akan memanggil kernel GPU bernama gpu_convert2double dan inverse2double. Kernel tersebut merubah tipe data menjadi double yaitu nilai pada radius antara 1 sampai dengan $1x10^{-14}$.

```
void gpu_image2double(unsigned char *h_src, unsigned char *h_lamda, double*
h_outsrc, double *h_outlamda, int numRows, int numCols)
{
	double *d_outlamda, *d_outsrc;
	unsigned char *d_src, *d_lamda;
	int size = numRows*numCols;

	const dim3 blockSize(32, 32, 1);
	const dim3 gridSize((numCols / 32) + 1, (numRows / 32) + 1, 1);

	checkCudaErrors(cudaMalloc(&d_src, sizeof(unsigned char)* size));
	checkCudaErrors(cudaMalloc(&d_lamda, sizeof(unsigned char)* size));
	checkCudaErrors(cudaMalloc(&d_outlamda, sizeof(double)* size));
	checkCudaErrors(cudaMalloc(&d_outsrc, sizeof(double)* size));

	checkCudaErrors(cudaMemcpy(d_src, h_src, sizeof(unsigned char)*size,
	cudaMemcpyHostToDevice));
	checkCudaErrors(cudaMemcpy(d_lamda, h_lamda, sizeof(unsigned char)*size,
	cudaMemcpyHostToDevice));

	checkCudaErrors(cudaMemset(d_outlamda, 0, sizeof(double)*size));
	checkCudaErrors(cudaMemset(d_outsrc, 0, sizeof(double)*size));

	Convert2double << <gridSize, blockSize >> >(d_src, d_outsrc, numRows,
	numCols);
	Inverse2double << <gridSize, blockSize >> >(d_lamda, d_outlamda, numRows,
	numCols);

	checkCudaErrors(cudaMemcpy(h_outsrc, d_outsrc, sizeof(double)* size,
	cudaMemcpyDeviceToHost));
	checkCudaErrors(cudaMemcpy(h_outlamda, d_outlamda, sizeof(double)* size,
	cudaMemcpyDeviceToHost));

	cudaFree(d_src);
	cudaFree(d_lamda);
	cudaFree(d_outlamda);
	cudaFree(d_outsrc);
}
```

Kode Isi Fungsi gpu_image2double GPU

Fungsi ini melakukan inisialisasi ukuran *blok* dan *grid*. Setelah itu melakukan alokasi memori pada GPU dan melakukan transfer data dari CPU ke GPU. Data kemudian akan diproses menggunakan kernel Convert2double dan Inverse2double. Setelah data selesai diproses maka data akan ditrasfer kembali ke CPU. Proses fungsi ini akan berakhir ketika fungsi melakukan pembebasan memori pada GPU.

```
__global__
void Convert2double(unsigned char* const inputChannel, double* const
outputChannel, int numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;	// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;	// current column
	int idx = col + numCols * row;		// current pixel index
```

```
        if ((row >= numRows) || (col >= numCols))
        {
                return;
        }
        double intensity = inputChannel[idx];
        outputChannel[idx] = intensity / 255;
}

__global__
void Inverse2double(unsigned char* const inputChannel, double* const
outputChannel, int numRows, int numCols)
{
        int row = blockIdx.y * blockDim.y + threadIdx.y;        // current row
        int col = blockIdx.x * blockDim.x + threadIdx.x;        // current column
        int idx = col + numCols * row;                  // current pixel index

        if ((row >= numRows) || (col >= numCols))
        {
                return;
        }
        double intensity = inputChannel[idx];
        outputChannel[idx] = (1 - intensity / 255) * lambda0;
}
```

Kode Kernel untuk merubah tipe data GPU

Untuk mengubah nilai menjadi double maka cukup dilakukan pembagian dengan 255. Hal ini dikarenakan 255 merupakan nilai tertinggi pada matriks citra. Sehingga nilai tertingi dari nilai menjadi 1. Sedangkan untuk inverse cukup dilakukan pengurangan hasil pembagian dengan bilangan 1. Sehingga akan menghasilkan kebalikan dari citra asal. Setelah itu citra inverse dikuatkan nilainya dengan perkalian dengan *lambda0*.

4) Memanggil fungsi untuk proses *inpainting*

```
gpu_inpaintinghighorder(h_doubleSrc, h_doubleLamda, h_denominator, h_out,h_curve,
2000, elapsedtime, rows,cols);
```

Kode Fungsi Inpainting orde keempat GPU

Semua parameter sudah diproses, kini siap untuk dilakukan pemrosesan perbaikan citra. Pemrosesan menggunakan iterasi sebanyak 2000 iterasi. Fungsi ini menggunakan *pointer array* dari citra asli, citra mask dan denominator. Fungsi ini akan memberikan nilai keluaran berupa *pointer array* hasil pemrosesan dan waktu eksekusi program.

Dalam fungsi tersebut akan melakukan alokasi block dan grid yang tepat. Alokasi memory menggunakan lebar maksimum dari hardware GPU yaitu 32 x 32 thread, yaitu 1024 thread per block. Setelah itu dibutuhkan alokasi pada variabel yang disimpan pada GPU, untuk melakukan hal tersebut digunakan perintah cudaMalloc. Variabel menggunakan tipe *cufftDoubleReal* dan *cufftDoubleComplex* sebagai tipe data untuk pemrosesan pada fast forier transform. Tipe data ini akan dialokasikan sesuai dengan dimensi citra yaitu panjang citra dikali lebar citra.

```
const dim3 blockSize(32, 32, 1);
const dim3 gridSize((numCols / 32) + 1, (numRows / 32) + 1, 1);

int sizedata = numRows*numCols;
int sizedouble = sizeof(cufftDoubleReal)*sizedata;
int sizecomplex = sizeof(cufftDoubleComplex)*sizedata;

cufftDoubleReal * d_u, *d_lambda, *d_denominator, *d_output;
cufftDoubleReal * ux, *uy, *uxx, *uyy, *uxy, *dp, *dm;
cufftDoubleReal *LUbar, *curve, *result;
cufftDoubleComplex  *cu, *cubar, *cLU0bar, *cLU, *cLUbar, *ccurve,
*ccurvebar, *lapcurvbar, *cresult, *cresult2;

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

checkCudaErrors(cudaMalloc(&d_u, sizedouble));
checkCudaErrors(cudaMalloc(&result, sizedouble));
checkCudaErrors(cudaMalloc(&d_output, sizedouble));

checkCudaErrors(cudaMalloc(&d_lambda, sizedouble));
checkCudaErrors(cudaMalloc(&d_denominator, sizedouble));

checkCudaErrors(cudaMalloc(&ux, sizedouble));
checkCudaErrors(cudaMalloc(&uy, sizedouble));
checkCudaErrors(cudaMalloc(&uxx, sizedouble));
checkCudaErrors(cudaMalloc(&uyy, sizedouble));
checkCudaErrors(cudaMalloc(&dp, sizedouble));
checkCudaErrors(cudaMalloc(&dm, sizedouble));
checkCudaErrors(cudaMalloc(&uxy, sizedouble));

checkCudaErrors(cudaMalloc(&curve, sizedouble));
checkCudaErrors(cudaMalloc(&LUbar, sizedouble));

checkCudaErrors(cudaMalloc(&cubar, sizecomplex));
checkCudaErrors(cudaMalloc(&cu, sizecomplex));
checkCudaErrors(cudaMalloc(&cLU0bar, sizecomplex));
checkCudaErrors(cudaMalloc(&cLU, sizecomplex));
checkCudaErrors(cudaMalloc(&cLUbar, sizecomplex));
checkCudaErrors(cudaMalloc(&ccurve, sizecomplex));
checkCudaErrors(cudaMalloc(&ccurvebar, sizecomplex));
checkCudaErrors(cudaMalloc(&lapcurvbar, sizecomplex));
checkCudaErrors(cudaMalloc(&cresult, sizecomplex));
checkCudaErrors(cudaMalloc(&cresult2, sizecomplex));
```

```
        checkCudaErrors(cudaMemcpy(d_u, h_src, sizedouble,
cudaMemcpyHostToDevice));
        checkCudaErrors(cudaMemcpy(d_lambda, h_lamda, sizedouble,
        cudaMemcpyHostToDevice));
        checkCudaErrors(cudaMemcpy(d_denominator, h_denominator, sizedouble,
        cudaMemcpyHostToDevice));
```

Kode Alokasi variabel pada GPU

Setelah variabel pada GPU dibuat, maka data akan ditransfer dari CPU (*host*) ke GPU(*device*) dengan menggunakan perintah cudaMemcpy dengan parameter *cudaMemcpyHostToDevice*. Variabel yang dipindah ke GPU adalah pointer array dari citra asli, citra mask dan denominator.

```
cufftHandle pFFT;
cufftHandle pIFFT;

if (cufftPlan2d(&pFFT,numRows,numCols, CUFFT_Z2Z) != CUFFT_SUCCESS){
        fprintf(stderr, "CUFFT error:01.FFT Plan creation Real 2 Complex failed");
        return;
}

if (cufftPlan2d(&pIFFT,numRows,numCols, CUFFT_Z2Z) != CUFFT_SUCCESS){
        fprintf(stderr, "CUFFT error:02.IFFT Plan creation Complex 2 Real failed");
        return;
}
```

Kode Pembuatan Plan FFT pada untuk Inpainting GPU

Fungsi tersebut akan membuat plan. Plan akan dibuat 2 macam yaitu untuk tranformasi dari dimensi ruang ke dimensi frekuensi dan tranformasi dari dimensi frekuensi ke dimensi ruang (*inverse*). Plan tersebut mempunyai tipe double kompleks yang akan ditransformasi ke double kompleks (*CUFFT_Z2Z*).

```
computeLU << <gridSize, blockSize >> >(d_u, d_lambda, cLU, numRows, numCols);
if (cufftExecZ2Z(pFFT, cLU, cLUbar, CUFFT_FORWARD) != CUFFT_SUCCESS){
        fprintf(stderr, "CUFFT error: Exec FFT lubar to Complex lubar failed");
                return;
}
```

Kode Eksekusi FFT pada GPU

Setelah pembuatan plan, maka akan memulai perhitungan untuk citra asli, dikalikan dengan citra mask kemudian dilakukan transformasi ke dimensi frekuensi hal ini tampak dalam parameter *CUFFT_FORWARD*. Transformasi tersebut menggunakan perintah *cufftExecZ2Z*. Namun sebelumnya perlu dilakukan perhitungan citra asli dengan mask pada kernel computeLU. Isi kernel tersebut adalah sebagai berikut:

```
__global__
void computeLU(double *u, double* lambda, cufftDoubleComplex* lubar, int numRows,
int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	double value = 0;
	int index = col + numCols * row;		// current pixel index
	value = u[index] * lambda[index];
	lubar[index].x= value;
	lubar[index].y = 0;
}
```

Kode Isi Kernel GPU Compute LU

Kernel ini tidak hanya melakukan perkalian antara citra asli dan citra mask, namun juga melakukan transformasi tipe data menjadi format bilangan kompleks. Bilangan komplek mempunyai tempat penyimpanan tipe riil dan imajiner. Data merupakan bilangan yang riil namum menempati tempat bilangan kompleks, maka pada penyimpanan bilangan imajiner di set nilainya dengan 0(lubar[index].y=0). Sehingga nantinya eksekusi perintah tranformasi dapat memenuhi persyaratan double kompleks ke double komplek.

```
copyData << <gridSize, blockSize >> >(cLUbar, cLU0bar, numRows, numCols);
```

Kode Pemanggilan Kernel GPU CopyData

Kernel ini akan melakukan copy data dari variabel cLUbar untuk inisialisasi data cLU0bar. Data cLU0bar akan menjadi penanda data pertama. Hal ini nantinya akan

digunakan pada iterasi perhitungan. Sedangkan data LUbar akan terus berubah sesuai dengan iterasi.

```
real2complex << <gridSize, blockSize >> >(d_u, cu, numRows, numCols);
if (cufftExecZ2Z(pFFT, cu, cubar, CUFFT_FORWARD) != CUFFT_SUCCESS){
	fprintf(stderr, "CUFFT error: Exec FFT u to Complex ubar failed");
	return;
}
```

Kode Eksekusi FFT pada GPU ke 2

Kernel real2complex ini bertujuan untuk merubah tipe data dan set nilai menjadi sesuai format yang dibutuhkan oleh cufftExecZ2Z. Setelah selesai maka akan dilakukan tranformasi data ke dimensi frekuensi oleh Transformasi Fourier.

```
__global__	void real2complex(double * in, cufftDoubleComplex  *out, int
numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column

	if ((row >= numRows) || (col >= numCols))
	{
		return;
	}

	int index = col + numCols * row;
	out[index].x = in[index];
	out[index].y = 0.0f;
}
```

Kode Isi kernel GPU real2complex

Proses perhitungan akan melakukan iterasi sebanyak inputan dari variabel iteration. Pertama kali maka akan melakukan perhitungan curve. Setelah perhitungan selesai dilakukan maka akan ditransformasi ke dimensi frekuensi dengan Transformasi Fourier. Setelah itu akan mencari nilai lapasian dari *curve* pada dimensi frekuensi. Hal tersebut dapat dilakukan dengan melakukan perkalian dengan denominator. Nilai ini akan disimpan pada variabel lapcurvbar. Setelah itu akan melakukan perhitungan secara paralel dengan kernel inpaintinghighordercomplex. Pada kernel ini akan menerapkan Persamaan 19. Hasil perhitungan tersebut merupakan nilai untuk inputan iterasi berikutnya. Oleh karena itu maka

data harus ditransformasi dari dimensi frekuensi ke dimensi ruang. Setelah transformasi selesai, maka akan dihitung nilai lubar yang baru dan nilai lubar akan ditransformasi ke dimensi frekuensi untuk inputan perhitungan iterasi berikutnya.

```
for (int i = 0; i < iteration; i++)
{
	ComputeCurve << < gridSize, blockSize >> >(d_u, ccurve,curve, sizedata,
	numRows, numCols);
	if (cufftExecZ2Z(pFFT, ccurve, ccurvebar, CUFFT_FORWARD) != CUFFT_SUCCESS){
		fprintf(stderr, "CUFFT error: Exec FFT curv to Complex failed");
		return;
	}
	computelapcurvbar << <gridSize, blockSize >> >(ccurvebar, d_denominator,
	lapcurvbar, numRows, numCols);  //lapcurvbar = Denominator.*fft2(curv);

	//lapruvbar agak beda sedikit, karena nilai 0nya.
	InpaintingHighOrderComplex << <gridSize, blockSize >> >(cubar,
	d_denominator, lapcurvbar, cLU0bar, cLUbar, cresult, numRows, numCols);
	if (cufftExecZ2Z(pIFFT, cresult, cu, CUFFT_INVERSE) != CUFFT_SUCCESS)
	{
		fprintf(stderr, "CUFFT error: Exec IFFT curv to Complex failed");
		return;
	}
	copyData << <gridSize, blockSize >> >(cresult, cubar, numRows, numCols);
	//lu0bar copy
	complex2real << <gridSize, blockSize >> >(cu, d_u, numRows, numCols);
	computeLU << <gridSize, blockSize >> >(d_u, d_lambda, cLU, numRows,
	numCols);
	if (cufftExecZ2Z(pFFT, cLU, cLUbar, CUFFT_FORWARD) != CUFFT_SUCCESS)
	{
		fprintf(stderr, "CUFFT error: Exec IFFT lubar to Complex failed");
		return;
	}
}
```

Kode Proses Iterasi perhitungan pada GPU

Perhitungan *inpainting* sangat bergantung pada perhitungan nilai *curve*. Nilai ini akan diperoleh dari perhitungan nilai disekitarnya. Pada perhitungan *curve* ini digunakan Persamaan 19. Hal yang penting menjadi perhatian adalah lokasi tepi dari tiap matriks. Perhitungan harus memperhatikan transformasi matriks 2D menjadi pointer array.

```
__global__
void ComputeCurve(double *u, cufftDoubleComplex *curv, double *cur,  int size,
int numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column

	if ((row >= numRows) || (col >= numCols)){
		return;
	}
```

```
        int index = col + numCols * row;

        int idown = index + numCols;
        int iup = index - numCols;
        int iright = index + 1;
        int ileft = index - 1;
        ////for diagonal
        int iupleft = iup - 1;
        int iupright = iup + 1;
        int idownleft = idown - 1;
        int idownright = idown + 1;

        //menangani pojok
        iup<0 ? iup = index : iup = iup;
        ileft < 0 ? ileft = index : ileft = ileft;
        idown >= size ? idown = index : idown = idown;
        iright >= size ? iright = index : iright = iright;

        ////menangani pojok
        if (row == 0){
              iupleft = ileft;
              iupright = iright;
              ////idownleft = idown;
        }
        else if (row == numRows - 1){
              idownleft = ileft;
              idownright = iright;
        }
        ////supaya 0,0 mengikuti colom
        if (col == 0){
              ileft = index;  //menangani selisih kanan kiri
              iupleft = iup; //menangani diagonal pojok
              idownleft = idown;
        }
        else if (col == numCols - 1){
              iright = index;
              iupright = iup;
              idownright = idown;
        }

        double x = (u[ileft] - u[iright])/2;
        double y = (u[idown] - u[iup])/2;
        double xx = u[ileft] + u[iright] - 2 * u[index];
        double yy = u[iup] + u[idown] - 2 * u[index];
        double Dp = u[iupleft] + u[idownright];
        double Dm = u[iupright] + u[idownleft];
        double xy = (Dp - Dm) / 4;
        double Num = xx * (ep2 + y * y) - 2 * x * y * xy + yy * (ep2 + x * x);
        double Den = powf((ep2 + x * x + y * y), 1.5);
        double Curv = Num / Den;

        //Curv complex format
        curv[index].x = Curv;
        curv[index].y = 0;
        cur[index] = Curv;
        __syncthreads();
}
```

Kode Isi kernel GPU computeCurve

Melakukan proses *inpainting* dengan perhitungan pada Persamaan 26. Variabel akan dibaca dari register memory pada kernel, sehingga lebih cepat. Proses perhitungan ini menggunakan bilangan kompleks.

```
__global__
void InpaintingHighOrderComplex(cufftDoubleComplex * ubar, double * Denominator,
cufftDoubleComplex * lapcurvbar, cufftDoubleComplex * lu0bar, cufftDoubleComplex
* lubar, cufftDoubleComplex *newubar, int numRows, int numCols)
{
	double dt = 100;
	double c1 = 200;
	double c2 = 100;

	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}

	int index = col + numCols * row;
	double alpha = 1 + c2*dt + dt*c1*(Denominator[index] * Denominator[index]);
	cufftDoubleComplex result = ComplexSub(ComplexScale(ubar[index], alpha),
	ComplexAdd(ComplexScale(lapcurvbar[index], dt),
	ComplexScale(ComplexSub(lubar[index], lu0bar[index] ), dt)));
	result = ComplexScaleDiv(result, alpha);
	newubar[index] = result;
}
```

Kode Isi kernel GPU InpaintingHighOrderComplex

Hasil dari perhitungan akan dikembalikan ke fungsi utama. Sehingga dapat diproses pada iterasi berikutnya.

5) Transfer data dari *device*(GPU) ke *host*(CPU)

```
checkCudaErrors(cudaMemcpy(h_output, d_u, sizedouble, cudaMemcpyDeviceToHost));
```

Kode Transfer data dari GPU ke CPU

Setelah data selesai diproses data akan ditrasfer ke CPU dengan perintah cudaMemcpy dan parameter cudaMemcpyDeviceToHost.

6) Dealokasi Memori pada GPU

```
cufftDestroy(pFFT);
cufftDestroy(pIFFT);

cudaFree(d_u);
cudaFree(d_lambda);
cudaFree(d_denominator);
cudaFree(d_output);
cudaFree(LUbar);
```

```
cudaFree(curve);
cudaFree(cu);
cudaFree(cubar);
cudaFree(cLU0bar);
cudaFree(cLU);
cudaFree(cLUbar);
cudaFree(ccurve);
cudaFree(ccurvebar);
cudaFree(lapcurvbar);
cudaFree(cresult);
```

Kode Dealokasi variabel pada GPU

Membersihkan semua memori yang tertinggal dalam proses *inpainting*.

7) Tampilkan Gambar

Untuk menampilkan data matriks dibutuhkan bantuan dari *library* OpenCV. Data pointer akan diinisialisasi ke kelas Mat sehingga menjadi objek yang sesuai. Setelah itu objek akan ditampilkan ke layar dengan perintah imshow. Interuksi waitkey bertujuan untuk memberikan jeda waktu sehingga proses tidak akan terminate ketika selesai.

```
Mat outputsrc(src.rows, src.cols, vartype, h_doubleSrc);
Mat outpuinpainting(src.rows, src.cols, vartype, h_out);
imshow("soruce", outputsrc);
imshow("inpainting", outpuinpainting);
waitKey();
```

Kode Tampilkan data

8) Simpan Gambar

```
outpuinpainting.convertTo(outpuinpainting, CV_8UC1, 255.0);
imwrite("outputGPU.png", outpuinpainting);
```

Kode Menyimpan gambar

Untuk menyimpan gambar hasil *inpainting* sesuai dengan format yang benar, maka data akan dikonversi terlebih dahulu ke tipe CV_8UC1. Setelah itu diberikan parameter 255, hal ini bertujuan sebagai penanda nilai maksimum dari matriks yang dibuat. Setelah format tercukupi, maka data disimpan dengan bantuan perintah imwrite.

Lampiran 5. Kode Lengkap Inpainting Berbasis CPU

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include <math.h>

#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/features2d/features2d.hpp"

const double PI = 3.1415;

using namespace std;
using namespace cv;

Mat src; Mat u; Mat mask;
char window_name[] = "Cahn Hilliard Inpainting ";
Mat ComputeCurve(Mat src, int rows, int cols, int vartype);
double psnr(Mat & img_src, Mat & img_compressed);
double mse(Mat & img1, Mat & img2);
double ssim(Mat & img_src, Mat & img_compressed, int block_size, bool show_progress =
false);
double sigma(Mat & m, int i, int j, int block_size);
double cov(Mat & m1, Mat & m2, int i, int j, int block_size);


int main(void)
{
	double maxs = 0, mins = 0, maxm = 0, minm = 0, maxg = 0, ming = 0;
	Mat lambda, g;
	int vartype = CV_64FC1;
	src = imread("input.png", CV_LOAD_IMAGE_GRAYSCALE);
	mask = imread("input-mask_i.png", CV_LOAD_IMAGE_GRAYSCALE);
	src.convertTo(src, vartype);
	mask.convertTo(mask, vartype);

	minMaxLoc(src, &mins, &maxs);
	minMaxLoc(mask, &minm, &maxm);
	src = src / maxs;

	mask = 1-mask / maxm;

	lambda = mask.clone();
	g = src.clone();
	minMaxLoc(g, &ming, &maxg);
	divide(g, maxg, g);


	//parameter
	int dt = 100;
	double epsilon = 0.005;
	double ep2 = pow(epsilon , 2);
	int lpower = 2;
	double lambda0 = 100;
	lambda = lambda.mul(lambda0);
	double c1 = 200;
	double c2 = 100;
	int Itermax = 2000;

	int rows = g.rows;
	int cols = g.cols;
```

```
	u = g;
	imshow("Lambda", lambda);
	imshow("Original Image", g);
	waitKey(1);

	Mat Lambda1 = Mat::zeros(rows, rows, vartype);
	Mat Lambda2 = Mat::zeros(cols, cols, vartype);

	for (int j = 0; j<rows; j++){
		Lambda1.at<double>(j, j) = 2 * (cos(2 * j * PI / rows) - 1);
	}

	for (int j = 0; j<cols; j++){
		Lambda2.at<double>(j, j) = 2 * (cos(2 * j * PI / cols) - 1);
	}
	int h1 = 1;
	int h2 = 1;
	Mat Denominator = 1 / (h1*h1)*(Lambda1*Mat::ones(rows, cols, vartype)) + 1 /
(h2*h2)*(Mat::ones(rows, cols, vartype)*Lambda2);


	Mat ubar, uabar, LUbar, LU0bar, LapCurvbar,curvbar;
	Mat curv,alpha,LU,tesfft,ivtes;
	Mat complexLapCurvbar[] = { Mat::zeros(u.size(), vartype), Mat::zeros(u.size(),
vartype) };
	Mat complexubar[] = { Mat::zeros(u.size(), vartype), Mat::zeros(u.size(),
vartype) };

	dft(u, ubar, DFT_COMPLEX_OUTPUT);
	dft(u.mul(lambda), LU0bar, DFT_COMPLEX_OUTPUT);
	LUbar = LU0bar.clone();

	//LUbar = LU0bar;
	std::chrono::time_point<std::chrono::system_clock> start, end;
	start = std::chrono::system_clock::now();
	for (int i = 0; i<200; i++)
	{
		curv = ComputeCurve(u, src.rows, src.cols, vartype);
		dft(curv, curvbar,DFT_COMPLEX_OUTPUT);

		split(curvbar, complexLapCurvbar);
		complexLapCurvbar[0] = complexLapCurvbar[0].mul(Denominator);
		complexLapCurvbar[1] = complexLapCurvbar[1].mul(Denominator);
		merge(complexLapCurvbar, 2, LapCurvbar);
		alpha = 1 + c2*dt + dt*c1*Denominator.mul(Denominator);

		split(ubar, complexubar);
		complexubar[0] = complexubar[0].mul(alpha);
		complexubar[1] = complexubar[1].mul(alpha);
		merge(complexubar, 2, uabar);

		ubar = uabar - dt*LapCurvbar + dt*(LU0bar - LUbar);
		tesfft = dt*LapCurvbar;
		split(ubar, complexubar);
		divide( complexubar[0],alpha, complexubar[0]);
		divide(complexubar[1], alpha, complexubar[1]);
		merge(complexubar, 2, ubar);

		idft(ubar, u, DFT_SCALE | DFT_REAL_OUTPUT); // Applying IDFT
		LU = u.mul(lambda);
		dft(LU, LUbar,DFT_COMPLEX_OUTPUT);
```

```
            //imshow("Cahn Hilliard high order Inpainting", u);
            //waitKey(1);
            cout << "Iterasi - " << i + 1 << endl;
        }
        end = std::chrono::system_clock::now();

        u.convertTo(u, CV_8UC1, 255.0);
        imshow("Cahn Hilliard high order Inpainting", u);
        imwrite("output.png", u);
        std::chrono::duration<double> elapsed_seconds = end - start;
        time_t end_time = std::chrono::system_clock::to_time_t(end);
        struct tm timeinfo;
        localtime_s(&timeinfo, &end_time);
        cout << "Finished computation in " << put_time(&timeinfo, " %d-%m-%Y at
%H:%M:%S") << endl;
        cout << "Elapsed time: " << elapsed_seconds.count() << "s\n";
        u.convertTo(u, CV_64FC1);
        cout << "\nQuality Measurement" << endl;
        cout << "MSE : " << mse(g, u) << endl;
        cout << "PSNR: " << psnr(g, u) << endl;
        cout << "SSIM: " << ssim(g, u, 8, false) << endl;

        waitKey(0);

        return 0;
}
Mat ComputeCurve(Mat src, int rows, int cols, int vartype)
{
        Mat result = Mat::zeros(rows, cols, vartype);
        Mat ux = Mat::zeros(rows, cols, vartype);

        double epsilon = 0.005;
        double ep2 = pow(epsilon, 2);
        int w, n, s, e = 0;
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                e = j + 1;
                w = j - 1;

                n = i - 1;
                s = i + 1;

                w <= 0 ? w = 0 : w = w;
                e>cols - 1 ? e = cols - 1 : e = e;
                n <= 0 ? n = 0 : n = n;
                s > rows - 1 ? s = rows - 1 : s = s;

                double x = (src.at<double>(i, e) - src.at<double>(i, w)) / 2;
                double y = (src.at<double>(s, j) - src.at<double>(n, j)) / 2;
                double xx = src.at<double>(i, w) + src.at<double>(i, e) - 2 *
src.at<double>(i, j);
                double yy = src.at<double>(s, j) + src.at<double>(n, j) - 2 *
src.at<double>(i, j);
                double dp = src.at<double>(n, w) + src.at<double>(s, e);
                double dm = src.at<double>(n, e) + src.at<double>(s, w);
                double xy = (dp - dm) / 4;
                double num = xx*(ep2 + y*y) - 2 * x*y*xy + yy*(ep2 + x*x);;
                double den = pow((ep2 + x*x + y*y), 1.5);
                double curve = num / den;
```

```
                        //ux.at<double>(i, j) = curve;
                        result.at<double>(i, j) =curve;
                }

        }
        return result;
}

double psnr(Mat & img_src, Mat & img_compressed)
{
        int D = 255;
        return (10 * log10((D*D) / mse(img_src, img_compressed)));
}

double mse(Mat & img1, Mat & img2)
{
        int i, j;
        double mse = 0;
        int height = img1.rows;
        int width = img1.cols;

        for (i = 0; i < height; i++)
        for (j = 0; j < width; j++)
                mse += (img1.at<double>(i, j) - img2.at<double>(i, j)) *
(img1.at<double>(i, j) - img2.at<double>(i, j));

        mse /= height * width;

        return mse;
}

double ssim(Mat & img_src, Mat & img_compressed, int block_size, bool show_progress)
{
        // typicaly block 8
        double ssim = 0;
        float C1 = (float)(0.01 * 255 * 0.01 * 255);
        float C2 = (float)(0.03 * 255 * 0.03 * 255);
        int nbBlockPerHeight = img_src.rows / block_size;
        int nbBlockPerWidth = img_src.cols / block_size;

        for (int k = 0; k < nbBlockPerHeight; k++)
        {
                for (int l = 0; l < nbBlockPerWidth; l++)
                {
                        int m = k * block_size;
                        int n = l * block_size;

                        double avg_o = mean(img_src(Range(k, k + block_size), Range(l, l
+ block_size)))[0];
                        double avg_r = mean(img_compressed(Range(k, k + block_size),
Range(l, l + block_size)))[0];
                        double sigma_o = sigma(img_src, m, n, block_size);
                        double sigma_r = sigma(img_compressed, m, n, block_size);
                        double sigma_ro = cov(img_src, img_compressed, m, n, block_size);

                        ssim += ((2 * avg_o * avg_r + C1) * (2 * sigma_ro + C2)) /
((avg_o * avg_o + avg_r * avg_r + C1) * (sigma_o * sigma_o + sigma_r * sigma_r + C2));

                }
                // Progress
                if (show_progress)
```

```
                        cout << "\r>>SSIM [" << (int)((((double)k) / nbBlockPerHeight) *
100) << "%]";
        }
        ssim /= nbBlockPerHeight * nbBlockPerWidth;

        if (show_progress)
        {
                cout << "\r>>SSIM [100%]" << endl;
                cout << "SSIM : " << ssim << endl;
        }

        return ssim;
}

double sigma(Mat & m, int i, int j, int block_size)
{
        double sd = 0;

        Mat m_tmp = m(Range(i, i + block_size), Range(j, j + block_size));
        Mat m_squared(block_size, block_size, CV_64F);

        multiply(m_tmp, m_tmp, m_squared);

        // E(x)
        double avg = mean(m_tmp)[0];
        // E(x²)
        double avg_2 = mean(m_squared)[0];


        sd = sqrt(avg_2 - avg * avg);

        return sd;
}

double cov(Mat & m1, Mat & m2, int i, int j, int block_size)
{
        Mat m3 = Mat::zeros(block_size, block_size, m1.depth());
        Mat m1_tmp = m1(Range(i, i + block_size), Range(j, j + block_size));
        Mat m2_tmp = m2(Range(i, i + block_size), Range(j, j + block_size));


        multiply(m1_tmp, m2_tmp, m3);

        double avg_ro = mean(m3)[0]; // E(XY)
        double avg_r = mean(m1_tmp)[0]; // E(X)
        double avg_o = mean(m2_tmp)[0]; // E(Y)


        double sd_ro = avg_ro - avg_o * avg_r; // E(XY) - E(X)E(Y)

        return sd_ro;
    }
```

Lampiran 6. Kode Lengkap Inpainting Pemrosesan Paralel untuk CPU

```
#include "opencv2\core\core.hpp"
#include "opencv2\opencv.hpp"
#include "opencv2\highgui\highgui.hpp"
#include "cuda_runtime.h"

#include <stdio.h>
#include <math.h>

const double PI = 3.1415;
using namespace std;
using namespace cv;

void gpu_image2double(unsigned char *h_src, unsigned char *h_lamda, double* h_outsrc,
double *h_outlamda, int cols, int rows);
void gpu_createvariabel(double *h_src, double * h_out, int cols, int rows);
void gpu_inpaintinghighorder(double *h_src, double *h_lamda, double *denominator,
double* h_output, double * h_curve, int iteration, float *elapsedtime, int numRows,
int numCols);

void main()
{
	printf(" CUDA Inpainting High Order Started \n");
	int vartype = CV_64FC1;
	//Read Data
	Mat src = imread("original.png", CV_LOAD_IMAGE_COLOR);
	Mat mask = imread("mask.png", CV_LOAD_IMAGE_COLOR);
	Mat lambda, u, g;
	int rows = src.rows;
	int cols = src.cols;

	//Init Parameter
	int Itermax = 1000;

	Mat Lambda1 = Mat::zeros(rows, rows, vartype);
	Mat Lambda2 = Mat::zeros(cols, cols, vartype);

	for (int j = 0; j<rows; j++){
		Lambda1.at<double>(j, j) = 2 * (cos(2 *j * PI / rows) - 1);
	}

	for (int j = 0; j<cols; j++){
		Lambda2.at<double>(j, j) = 2 * (cos(2 *j * PI / cols) - 1);
	}
	int h1 = 1;
	int h2 = 1;
	Mat Denominator = 1 / (h1*h1)*(Lambda1*Mat::ones(rows,cols, vartype)) + 1 /
(h2*h2)*(Mat::ones(rows, cols, vartype)*Lambda2);
	cvtColor(src, src, CV_BGR2GRAY);
	cvtColor(mask, mask, CV_BGR2GRAY);


	//GPU JADI
	unsigned char * h_src = (unsigned char*)src.data;
	unsigned char * h_lamda = (unsigned char*)mask.data;
	double *h_denominator = (double*)Denominator.data;
	double *h_lambda1 = (double*)Lambda1.data;
	double *h_lambda2 = (double*)Lambda2.data;

	double *h_doubleSrc = (double*)malloc(sizeof(double)*src.cols*src.rows);
	double *h_doubleLamda = (double*)malloc(sizeof(double)*src.cols*src.rows);
```

```
        double *h_out = (double*)malloc(sizeof(double)*src.cols*src.rows);
        double *h_curve = (double*)malloc(sizeof(double)*src.cols*src.rows);
        float *elapsedtime = 0;

        gpu_image2double(h_src, h_lamda, h_doubleSrc, h_doubleLamda, rows, cols);
        gpu_inpaintinghighorder(h_doubleSrc, h_doubleLamda, h_denominator,
h_out,h_curve, 2000, elapsedtime, rows,cols);

        Mat outputsrc(src.rows, src.cols, vartype, h_doubleSrc);
        Mat outpuinpainting(src.rows, src.cols, vartype, h_out);
        Mat outputcurve(src.rows, src.cols, vartype, h_curve);

        imshow("lambda asli", mask);
        printf("Print Inpainting 10 first data\n");
        for (int j = 0; j < 10; j++)
        {
                cout << outpuinpainting.at<double>(0, j)<<" ";
        }
        printf("\n\nPrint Src 10 first data\n");
        for (int j = 0; j < 10; j++)
        {
                cout << outputsrc.at<double>(0, j) << " ";
        }

        printf("\n\nPrint Curve 10 first data\n");
        for (int j = 0; j < 10; j++)
        {
                cout << outputcurve.at<double>(0, j) << " ";
        }

        imshow("soruce", outputsrc);
        imshow("inpainting", outpuinpainting);
        outpuinpainting.convertTo(outpuinpainting, CV_8UC1, 255.0);
        imwrite("outputGPU.png", outpuinpainting);
        imshow("Curve", outputcurve);
        //cout << "Elapsed time: " << elapsedtime << "s\n";
        waitKey();
    }
```

Lampiran 7. Kode Lengkap Inpainting Pemrosesan Paralel GPU

```
#include "cuda_runtime.h"
#include "cufft.h"
#include "utils.h"
#include <stdio.h>
#include <algorithm>
#include "assert.h"


__constant__ int dt = 100;
__constant__ double epsilon = 0.005;
__constant__ double ep2 =0.000025;  //ep2 = epsilon^2;
                                    //__constant__ int lpower = 2;
__constant__ double lambda0 = 100;  //lambda0 = 10 ^ lpower;
__constant__ double c1 = 200;           //1 / epsilon;
__constant__ double  c2 = 100;          //c2 = lambda0;

__global__
void maxreduce(double * d_in, double * d_out, int numRows, int numCols)
{
	extern __shared__ double sdata[];
	// each thread loads one element from global to shared mem
	unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;     // current row
	unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;     // current column

	if ((row >= numRows) || (col >= numCols))
	{
		return;
	}
	unsigned int tid = threadIdx.x;
	unsigned int idx = col + numCols * row;         // current pixel index
	sdata[tid] = d_in[idx];
	__syncthreads();
	// do reduction in shared mem
	for
		(unsigned int s = 1; s < blockDim.x; s *= 2) {
		if (tid % (2 * s) == 0) {
			sdata[tid] = max(sdata[tid], sdata[tid + s]);
			//sdata[tid] =sdata[tid]+sdata[tid + s];sum
		}
		__syncthreads();
	}
	// write result for this block to global mem
	if (tid == 0) d_out[blockIdx.x] = sdata[0];
}
__global__
void Convert2double(unsigned char* const inputChannel, double* const outputChannel,
int numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;       // current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;       // current column
	int idx = col + numCols * row;           // current pixel index

	if ((row >= numRows) || (col >= numCols))
	{
		return;
	}
	double intensity = inputChannel[idx];
	outputChannel[idx] = intensity / 255;
}
__global__
```

```
void computelubarcomplex(double *u, double* lambda, cufftDoubleComplex *clubar, int
numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	int index = col + numCols * row;			// current pixel index
	double value = 0;
	value = u[index] * lambda[index];
	clubar[index].x =value;
	clubar[index].y = 0.0f;
}
__global__
void computeLU(double *u, double* lambda, cufftDoubleComplex* lubar, int numRows, int
numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	double value = 0;
	int index = col + numCols * row;			// current pixel index
	value = u[index] * lambda[index];
	lubar[index].x= value;
	lubar[index].y = 0;
}
__global__
void computelapcurvbar(cufftDoubleComplex * curve, double *Denominator,
cufftDoubleComplex * result, int numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	double x,y = 0;
	int index = col + numCols * row;			// current pixel index
	x = curve[index].x * Denominator[index];
	y = curve[index].y * Denominator[index];
	result[index].x = x;
	result[index].y = y;
}
__global__
void copyData(cufftDoubleComplex * lu0bar, cufftDoubleComplex * lubar, int numRows,
int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	int index = col + numCols * row;
	cufftDoubleComplex temp;
	temp = lu0bar[index];
	lubar[index] = temp;
}
__global__
void Inverse2double(unsigned char* const inputChannel, double* const outputChannel,
int numRows, int numCols)
```

```
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	int idx = col + numCols * row;			// current pixel index

	if ((row >= numRows) || (col >= numCols))
	{
		return;
	}
	double intensity = inputChannel[idx];
	outputChannel[idx] = (1 - intensity / 255) * lambda0; //lambda=lambda *.
lambda0;
}
__global__	void real2complex(double * in, cufftDoubleComplex  *out, int numRows,
int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	int index = col + numCols * row;
	out[index].x = in[index];
	out[index].y = 0.0f;
}
__global__	void complex2real(cufftDoubleComplex  *in, double * out, int numRows,
int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	int index = col + numCols * row;
	double result = in[index].x / ((double)numRows*(double)numCols);
	out[index] = result;
}
static __device__ __host__ inline cufftDoubleComplex ComplexScale(cufftDoubleComplex
a, double s)
{
	cufftDoubleComplex c;
	c.x = s * a.x;
	c.y = s * a.y;
	return c;
}
static __device__ __host__ inline cufftDoubleComplex ComplexSub(cufftDoubleComplex a,
cufftDoubleComplex b)
{
	cufftDoubleComplex c;
	c.x = a.x - b.x;
	c.y = a.y - b.y;
	return c;
}
static __device__ __host__ inline cufftDoubleComplex ComplexAdd(cufftDoubleComplex a,
cufftDoubleComplex b)
{
	cufftDoubleComplex c;
	c.x = a.x + b.x;
	c.y = a.y + b.y;
	return c;
}
```

```
static __device__ __host__ inline cufftDoubleComplex
ComplexScaleDiv(cufftDoubleComplex a, double s)
{
	cufftDoubleComplex c;
	c.x = a.x/s;
	c.y = a.y/s;
	return c;
}

__global__
void ComputeCurve(double *u, cufftDoubleComplex *curv, double *cur,  int size, int
numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;		// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;		// current column

	if ((row >= numRows) || (col >= numCols)){
		return;
	}

	int index = col + numCols * row;

	int idown = index + numCols;
	int iup = index - numCols;
	int iright = index + 1;
	int ileft = index - 1;
	////for diagonal
	int iupleft = iup - 1;
	int iupright = iup + 1;
	int idownleft = idown - 1;
	int idownright = idown + 1;

	//menangani pojok
	iup<0 ? iup = index : iup = iup;
	ileft < 0 ? ileft = index : ileft = ileft;
	idown >= size ? idown = index : idown = idown;
	iright >= size ? iright = index : iright = iright;

	////menangani pojok
	if (row == 0){
		iupleft = ileft;
		iupright = iright;
		////idownleft = idown;
	}
	else if (row == numRows - 1){
		idownleft = ileft;
		idownright = iright;
	}
	////supaya 0,0 mengikuti colom
	if (col == 0){
		ileft = index;  //menangani selisih kanan kiri
		iupleft = iup; //menangani diagonal pojok
		idownleft = idown;
	}
	else if (col == numCols - 1){
		iright = index;
		iupright = iup;
		idownright = idown;
	}

	double x = (u[ileft] - u[iright])/2;
	double y = (u[idown] - u[iup])/2;
```

```
	double xx = u[ileft] + u[iright] - 2 * u[index];
	double yy = u[iup] + u[idown] - 2 * u[index];
	double Dp = u[iupleft] + u[idownright];
	double Dm = u[iupright] + u[idownleft];
	double xy = (Dp - Dm) / 4;
	double Num = xx * (ep2 + y * y) - 2 * x * y * xy + yy * (ep2 + x * x);
	double Den = powf((ep2 + x * x + y * y), 1.5);
	double Curv = Num / Den;

	//Curv complex format
	curv[index].x = Curv;
	curv[index].y = 0;
	cur[index] = Curv;
	__syncthreads();
}
__global__
void normalize(cufftDoubleComplex *din, double * dout, int numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;	// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;	// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	int index = col + numCols * row;
	double value = din[index].x/(numCols*numRows);
	dout[index] = value;
}
__global__
void CahnHilliardHighOrder(double * ubar, double * Denominator, double * lapcurvbar,
double * lu0bar, double * lubar, double *newubar, int numRows, int numCols)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;	// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;	// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}
	int index = col + numCols * row;
	newubar[index] = ((1 + c2*dt +
dt*c1*Denominator[index]*Denominator[index])*ubar[index] - dt*lapcurvbar[index] +
dt*(lu0bar[index] - lubar[index]))/ (1 + dt*c2 + dt*c1*Denominator[index] *
Denominator[index]);
}
__global__
void CahnHilliardHighOrderComplex(cufftDoubleComplex * ubar, double * Denominator,
cufftDoubleComplex * lapcurvbar, cufftDoubleComplex * lu0bar, cufftDoubleComplex *
lubar, cufftDoubleComplex *newubar, cufftDoubleComplex *temp, int numRows, int
numCols)
{
	double dt = 100;
	double c1 = 200;
	double c2 = 100;

	int row = blockIdx.y * blockDim.y + threadIdx.y;	// current row
	int col = blockIdx.x * blockDim.x + threadIdx.x;	// current column
	if ((row >= numRows) || (col >= numCols)){
		return;
	}

	int index = col + numCols * row;
	double alpha = 1 + c2*dt + dt*c1*(Denominator[index] * Denominator[index]);
	cufftDoubleComplex result123 = ComplexSub(lubar[index], lu0bar[index]);
```

```
	cufftDoubleComplex result = ComplexSub(ComplexScale(ubar[index], alpha),
ComplexAdd(ComplexScale(lapcurvbar[index], dt), ComplexScale(ComplexSub(lubar[index],
lu0bar[index] ), dt)));
	result = ComplexScaleDiv(result, alpha);
	newubar[index] = result;
	temp[index] = result123;
}
void gpu_image2double(unsigned char *h_src, unsigned char *h_lamda, double* h_outsrc,
double *h_outlamda, int numRows, int numCols)
{
	double *d_outlamda, *d_outsrc;
	unsigned char *d_src, *d_lamda;
	int size = numRows*numCols;

	const dim3 blockSize(32, 32, 1);
	const dim3 gridSize((numCols / 32) + 1, (numRows / 32) + 1, 1);

	checkCudaErrors(cudaMalloc(&d_src, sizeof(unsigned char)* size));
	checkCudaErrors(cudaMalloc(&d_lamda, sizeof(unsigned char)* size));
	checkCudaErrors(cudaMalloc(&d_outlamda, sizeof(double)* size));
	checkCudaErrors(cudaMalloc(&d_outsrc, sizeof(double)* size));

	checkCudaErrors(cudaMemcpy(d_src, h_src, sizeof(unsigned char)*size,
cudaMemcpyHostToDevice));
	checkCudaErrors(cudaMemcpy(d_lamda, h_lamda, sizeof(unsigned char)*size,
cudaMemcpyHostToDevice));

	checkCudaErrors(cudaMemset(d_outlamda, 0, sizeof(double)*size));
	checkCudaErrors(cudaMemset(d_outsrc, 0, sizeof(double)*size));

	Convert2double << <gridSize, blockSize >> >(d_src, d_outsrc, numRows, numCols);
	Inverse2double << <gridSize, blockSize >> >(d_lamda, d_outlamda, numRows,
numCols);

	checkCudaErrors(cudaMemcpy(h_outsrc, d_outsrc, sizeof(double)* size,
cudaMemcpyDeviceToHost));
	checkCudaErrors(cudaMemcpy(h_outlamda, d_outlamda, sizeof(double)* size,
cudaMemcpyDeviceToHost));

	cudaFree(d_src);
	cudaFree(d_lamda);
	cudaFree(d_outlamda);
	cudaFree(d_outsrc);
}
void gpu_inpaintinghighorder(double *h_src, double *h_lamda, double *h_denominator,
double* h_output, double* h_curve, int iteration, float *elapsedTime, int numRows, int
numCols)
{
	const dim3 blockSize(32, 32, 1);
	const dim3 gridSize((numCols / 32) + 1, (numRows / 32) + 1, 1);

	int sizedata = numRows*numCols;
	int sizedouble = sizeof(cufftDoubleReal)*sizedata;
	int sizecomplex = sizeof(cufftDoubleComplex)*sizedata;

	cufftDoubleReal * d_u, *d_lambda, *d_denominator, *d_output;
	cufftDoubleReal * ux, *uy, *uxx, *uyy, *uxy, *dp, *dm;
	cufftDoubleReal *LUbar, *curve, *result;
	cufftDoubleComplex  *cu, *cubar, *cLU0bar, *cLU, *cLUbar, *ccurve, *ccurvebar,
*lapcurvbar, *cresult, *cresult2;

	cudaEvent_t start, stop;
```

```
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	checkCudaErrors(cudaMalloc(&d_u, sizedouble));
	checkCudaErrors(cudaMalloc(&result, sizedouble));
	checkCudaErrors(cudaMalloc(&d_output, sizedouble));

	checkCudaErrors(cudaMalloc(&d_lambda, sizedouble));
	checkCudaErrors(cudaMalloc(&d_denominator, sizedouble));

	checkCudaErrors(cudaMalloc(&ux, sizedouble));
	checkCudaErrors(cudaMalloc(&uy, sizedouble));
	checkCudaErrors(cudaMalloc(&uxx, sizedouble));
	checkCudaErrors(cudaMalloc(&uyy, sizedouble));
	checkCudaErrors(cudaMalloc(&dp, sizedouble));
	checkCudaErrors(cudaMalloc(&dm, sizedouble));
	checkCudaErrors(cudaMalloc(&uxy, sizedouble));

	checkCudaErrors(cudaMalloc(&curve, sizedouble));
	checkCudaErrors(cudaMalloc(&LUbar, sizedouble));

	checkCudaErrors(cudaMalloc(&cubar, sizecomplex));
	checkCudaErrors(cudaMalloc(&cu, sizecomplex));
	checkCudaErrors(cudaMalloc(&cLU0bar, sizecomplex));
	checkCudaErrors(cudaMalloc(&cLU, sizecomplex));
	checkCudaErrors(cudaMalloc(&cLUbar, sizecomplex));
	checkCudaErrors(cudaMalloc(&ccurve, sizecomplex));
	checkCudaErrors(cudaMalloc(&ccurvebar, sizecomplex));
	checkCudaErrors(cudaMalloc(&lapcurvbar, sizecomplex));
	checkCudaErrors(cudaMalloc(&cresult, sizecomplex));
	checkCudaErrors(cudaMalloc(&cresult2, sizecomplex));

	checkCudaErrors(cudaMemcpy(d_u, h_src, sizedouble, cudaMemcpyHostToDevice));
	checkCudaErrors(cudaMemcpy(d_lambda, h_lamda, sizedouble,
cudaMemcpyHostToDevice));
	checkCudaErrors(cudaMemcpy(d_denominator, h_denominator, sizedouble,
cudaMemcpyHostToDevice));

	cufftHandle pFFT;
	cufftHandle pIFFT;

	if (cufftPlan2d(&pFFT,numRows,numCols, CUFFT_Z2Z) != CUFFT_SUCCESS){
		fprintf(stderr, "CUFFT error:01.FFT Plan creation Real 2 Complex
failed");
		return;
	}

	if (cufftPlan2d(&pIFFT,numRows,numCols, CUFFT_Z2Z) != CUFFT_SUCCESS){
		fprintf(stderr, "CUFFT error:02.IFFT Plan creation Complex 2 Real
failed");
		return;
	}

	computeLU << <gridSize, blockSize >> >(d_u, d_lambda, cLU, numRows, numCols);
	if (cufftExecZ2Z(pFFT, cLU, cLUbar, CUFFT_FORWARD) != CUFFT_SUCCESS){
		fprintf(stderr, "CUFFT error: Exec FFT lubar to Complex lubar failed");
		return;
	}

	copyData << <gridSize, blockSize >> >(cLUbar, cLU0bar, numRows, numCols);
	//lu0bar copy!!!!
	real2complex << <gridSize, blockSize >> >(d_u, cu, numRows, numCols);
```

```
	if (cufftExecZ2Z(pFFT, cu, cubar, CUFFT_FORWARD) != CUFFT_SUCCESS){
		fprintf(stderr, "CUFFT error: Exec FFT u to Complex ubar failed");
		return;
	}

	cudaEventRecord(start);

	for (int i = 0; i < iteration; i++)
	{
		ComputeCurve << < gridSize, blockSize >> >(d_u, ccurve,curve, sizedata,
numRows, numCols);
		if (cufftExecZ2Z(pFFT, ccurve, ccurvebar, CUFFT_FORWARD) !=
CUFFT_SUCCESS){
			fprintf(stderr, "CUFFT error: Exec FFT curv to Complex failed");
			return;
		}
		computelapcurvbar << <gridSize, blockSize >> >(ccurvebar, d_denominator,
lapcurvbar, numRows, numCols);  //lapcurvbar = Denominator.*fft2(curv);
		//lapruvbar agak beda sedikit, karena nilai 0nya.
		CahnHilliardHighOrderComplex << <gridSize, blockSize >> >(cubar,
d_denominator, lapcurvbar, cLU0bar, cLUbar, cresult,cresult2, numRows, numCols);
		if (cufftExecZ2Z(pIFFT, cresult, cu, CUFFT_INVERSE) != CUFFT_SUCCESS){
//ifft
			fprintf(stderr, "CUFFT error: Exec IFFT curv to Complex failed");
			return;
		}
		copyData << <gridSize, blockSize >> >(cresult, cubar, numRows, numCols);
	//lu0bar copy!!!!
		complex2real << <gridSize, blockSize >> >(cu, d_u, numRows, numCols);
		computeLU << <gridSize, blockSize >> >(d_u, d_lambda, cLU, numRows,
numCols); //new LU
		if (cufftExecZ2Z(pFFT, cLU, cLUbar, CUFFT_FORWARD) != CUFFT_SUCCESS){
//LUbar
			fprintf(stderr, "CUFFT error: Exec IFFT lubar to Complex
failed");
			return;
		}
	}

	if (cufftExecZ2Z(pIFFT, cresult2, cresult2, CUFFT_INVERSE) != CUFFT_SUCCESS){
		fprintf(stderr, "CUFFT error: Exec IFFT u to Complex result failed");
		return;
	};
	cudaEventRecord(stop);
	normalize << <gridSize, blockSize >> >(cresult2, curve, numRows, numCols);

	checkCudaErrors(cudaMemcpy(h_output, d_u, sizedouble, cudaMemcpyDeviceToHost));
	checkCudaErrors(cudaMemcpy(h_curve, curve, sizedouble,
cudaMemcpyDeviceToHost));

	cudaEventSynchronize(stop);
	float elapsedTime1 = 0;
	cudaEventElapsedTime(&elapsedTime1, start, stop);

	printf("GPU: %f ms", elapsedTime1);
	cufftDestroy(pFFT);
	cufftDestroy(pIFFT);


	cudaFree(d_u);
	cudaFree(d_lambda);
```

```
    cudaFree(d_denominator);
    cudaFree(d_output);
    cudaFree(cubar);
    cudaFree(LUbar);
    cudaFree(cLUbar);
    cudaFree(cLU0bar);
    cudaFree(curve);
    cudaFree(ccurve);
    cudaFree(lapcurvbar);
    cudaFree(cresult);
}
```