BAB V

KESIMPULAN DAN SARAN

A. Kesimpulan

Berdasarkan hasil pengujian yang diperoleh, dapat disimpulkan bahwa:

1. Implementasi GPU CUDA untuk akselerasi proses *image inpainting* menggunakan metode *Alternating-Direction Implicit* (ADI) telah berhasil dikembangkan.
2. Berdasarkan hasil pengujian waktu proses pada beberapa citra uji, penggunaan GPU CUDA dapat mempercepat proses komputasi pada metode ADI hingga 6,08 kali dibandingkan dengan CPU, pada citra dengan resolusi 2736 x 1824.
3. Nilai *throughput* yang dihasilkan GPU akan semakin besar seiring dengan bersarnya resolusi sebuah citra.

B. Saran

Masih terdapat banyak peningkatan yang dapat dilakukan untuk mengembangkan penelitian ini, seperti:

1. Implementasi GPU CUDA pada skema implicit untuk menyelesaikan model *phase-field*.

2. Penggunaan *iterative solver* seperti *multigrid* atau lainnya, yang dikenal cepat kemudian ditambah implementasi GPU CUDA yang sudah terbukti dapat mempercepat proses komputasi.

DAFTAR PUSTAKA

Bertalmio, M., Sapiro, G., Caselles, V., & Ballester, C. (2000). Image Inpainting. *SIGGRAPH ACM* .

Bertozzi, A. L., Esedoglu, S., & Gillete, A. (2007). Inpainting of Binary Images Using the Cahn-Hilliard Equation. *IEEE Transactions on Image Processing* , 285-291.

Chan, T. F., Kang, S. H., & Shen, J. (2002). Euler's Elastica and Curvature-Based Image Inpainting. *SIAM Journal on Applied Mathematics* , 564-92.

Chan, T., & Shen, J. (2001). Mathematical Models for Local Nontexture Inpaintings. *SIAM Journal on Applied Mathematics* , 1019–43.

Cheng, J., Grossman, M., & McKercher, T. (2014). *Professional CUDA C Programming.* Indiana: John Wiley & Sons.

Darae, J., Seunggyu, L., Dongsun, L., Jaemin, S., & Junseok, K. (2016). Comparison study of numerical methods for solving the Allen–Cahn equation. *Computational Materials Science* , 131-136.

Darae, J., Yibao, L., Hyun Geun, L., & Junseok, K. (2009). Fast and Automatic Inpainting of Binary Images Using a Phase-Field Model. *J. KSIAM* , 225-236.

Esedoglu, S., & Shen, J. (2002). Digital Inpainting Based on The Mumford–Shah–Euler Image Model. *European Journal of Applied Mathematics* , 353-70.

Fjelland, B. (2012). *Thesis on applications of the Alternating Direction Implicit method.* Copenhagen: Copenhagen Business School.

Guillemot, C., & Le Meur, O. (2014). Image Inpainting : Overview and Recent Advances. *IEEE Signal Processing Magazine* , 127 - 144.

Hoffmann, K. A., & Chiang, S. T. (2000). *Computational Fluid Dynamics.* Kansas: A Publication of Engineering Education System.

Hore, A., & Ziou, D. (2010). Image Quality Metrics: PSNR vs SSIM. *ICPR* , 66-69.

Laszlo, E. (2016). *Parallelization of Numerical Methods on Parallel Processor Architectures.* Hungary: Pázmány Péter Catholic University.

Lin, Z., Zhang, W., & Tang, X. (2009). *Designing Partial Differential Equations for Image Processing by Combining Differential Invariants.* Microsoft Research Asia.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stones, J. E., & Phillips, J. C. (2008). GPU Computing. *Proceedings of the IEEE* , 879-899.

Panca, I. G. (2015). *Komputasi Paralel Berbasis Gpu Cuda untuk Pengembangan Image Inpainting dengan Metode Perona-Malik.* Yogyakarta: Universitas Atma Jaya Yogyakarta.

Prananta, E. (2015). *Akselerasi Proses Inpainting dengan Persamaan Diferensial Parsial Orde Keempat secara Paralel pada GPU CUDA.* Yogyakarta: Universitas Atma Jaya Yogyakarta.

Quarteroni, A., Sacco, R., & Saleri, F. (2000). *Numerical Mathematics.* New York: Springer.

Schonlieb, C.-B. (2009). *Modern PDE Techniques for Image Inpainting.* Cambridge University.

Tse, J. (2012). *Image Processing with CUDA.* Las Vegas: University of Nevada Las Vegas.

Wang, Z., & Bovik, A. C. (2009). Mean Squared Error: Love It or Leave It? A new look at signal fidelity measures. *IEEE Signal Processing Magazine* , 98-117.

Xu, L. (2011). *Parallel Computing based on GPGPU using Compute Unified Device Architecture.* Royal Institute of Technology (KTH).

Yibao, L., Darae, J., Jung-il, C., Seunggyu, L., & Junseok, K. (2015). Fast local image inpainting based on the Allen–Cahn model. *Digital Signal Processing* , 65-74.

Zhang, F., Chen, Y., Xiao, Z., Geng, L., Wu, J., Feng, T., et al. (2015). Partial Differential Equation Inpainting Method Based on Image Characteristics. *8th International Conference, ICIG* , 11-19.

LAMPIRAN

Lampiran 1 Pemrograman *Inpainting* Berbasis CPU

Hal pertama yang dilakukan dalam melakukan pemrograman pada CPU adalah mendefinisikan library yang akan digunakan. Library yang dipakai mencakup library OpenCV dan standard library C/C++.

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <string.h>
```

Library stdio.h digunakan untuk mengakses fungsi print. *Library time.h* digunakan untuk mengakses waktu pemrosesan. *Library math.h* digunakan untuk mengakses fungsi matematika seperti *pow* dan lainnya. *Library string.h* digunakan untuk mengakses fungsi *memcpy*.

```
#include <opencv2/opencv.hpp>
```

Kode diatas adalah kode untuk mengakses *library OpenCV* yang digunakan untuk membaca citra dan menyimpannya dalam kelas *Mat*.

```
cv::Mat img;
cv::Mat mask;
int WIDTH = 0;
int HEIGHT = 0;
double* h_img;
double* h_mask;
double* h_result;

int image_id;
#define IMAGE_1 1
#define IMAGE_2 2
#define IMAGE_3 3
```

```
#define IMAGE_4 4
#define IMAGE_5 5
```

Kode diatas merupakan definisi variable global yang akan digunakan selama proses inpainting.

```
#define ALPHA 1
#define DT 0.5
```

Tidak lupa untuk mendefinisikan variable perhitungan untuk proses inpainting seperti ALPHA dan DT (langkah waktu).

Setelah mempersiapkan beberapa library dan mendefinisikan variable yang dibutuhkan. Berikutnya adalah membuat fungsi untuk mengelompokkan pengerjaan pada proses inpainting. Langkah pertama adalah membaca citra dengan fungsi *imread*, kemudian menyimpannya dalam variable dengan tipe data *Mat* yang dimiliki oleh OpenCV.

```
void CPU_Init()
{
  // init original and mask image
  switch(image_id)
  {
    case IMAGE_1:
      img   = cv::imread("../images/original.png", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/mask.png", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    case IMAGE_2:
      img   = cv::imread("../images/lena_rusak.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/lena_mask.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    case IMAGE_3:
      img   = cv::imread("../images/car1.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/car_mask1.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    case IMAGE_4:
      img   = cv::imread("../images/car2.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/car_mask2.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
```

```
    case IMAGE_5:
      img   = cv::imread("../images/car3.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/car_mask3.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    default:
      printf("\nWrong image input!\n");
    break;
  }

  WIDTH   = img.cols;
  HEIGHT  = img.rows;

  img.convertTo(img, CV_64FC1);
  mask.convertTo(mask, CV_64FC1);

  h_img    = new double[WIDTH * HEIGHT];
  h_mask   = new double[WIDTH * HEIGHT];
  h_result = new double[WIDTH * HEIGHT];

  for (int i = 0; i < HEIGHT; i++)
  {
    for (int j = 0; j < WIDTH; j++)
    {
      int index = j + i * WIDTH;
      h_img[index]  = img.at<double>(i, j);
      h_mask[index] = 1-ceil(mask.at<double>(i, j) / 255);
    }
  }
  std::memcpy(h_result, h_img, WIDTH * HEIGHT * sizeof(double));
}
```

Langkah berikutnya adalah mengkonversi tipe data dalam matriks yang sebelumnya *unsigned char* ke tipe data *double precision* dengan bantuan fungsi *convertTo*. Setelah itu melakukan normalisasi nilai *mask*, yang sebelumnya berkisar 0-255 menjadi berkisar 0-1.

```
cv::Mat CPU_CN2D_ADI(int nIteration)
{
  cv::Mat result = img;
  u = img.clone();

  mask.convertTo(mask, CV_64FC1);
  for (int i = 0; i < HEIGHT; i++)
  {
    for (int j = 0; j < WIDTH; j++)
    {
```

```
      mask.at<double>(i, j) = 1-ceil(mask.at<double>(i, j) / 255);
    }
  }

  result.convertTo(result, CV_64FC1);

  //timer
  clock_t start, stop, selisih;
  float elapsedTime;
  start = clock();

  for(int i = 0; i < nIteration; i++)
  {
    cv::Mat temp = result;
    // solving X direction
    result = SweepDirectionX(temp, WIDTH, HEIGHT, ALPHA);
    // solving Y direction
    result = SweepDirectionY(temp, WIDTH, HEIGHT, ALPHA);
  }

  stop = clock();
  selisih = stop - start;
  elapsedTime = selisih / (float)CLOCKS_PER_SEC;
  printf("\nTime to generate CPU : %.3f ms\n", elapsedTime*1000);

  result.convertTo(result, CV_8UC1);
  return result;
}
```

Fungsi diatas adalah *entry-point* untuk proses inpainting pada CPU. Untuk merekam waktu proses *inpainting* digunakan tipe data *clock_t* dari *library time.h*. Diawal iterasi variable *start* merekam waktu sekarang, setelah iterasi selesai variable *stop* merekam waktu ketika proses iterasi telah selesai dijalankan. Waktu proses *inpainting* didapatkan dari selisih variable *stop* dan *start*. Untuk mendapatkan waktu dalam *millisecond* dilakukan pembagian terhadap selisih variable *start* dan *stop* dengan CLOCKS_PER_SEC.

```
cv::Mat SweepDirectionX(cv::Mat temp, int width, int height, int alpha)
{
```

```
  cv::Mat result = temp;
  cv::Mat lhs = cv::Mat::zeros(width - 2, width - 2, CV_64FC1);
  cv::Mat rhs = cv::Mat::zeros(width - 2, 1, CV_64FC1);
  double C = alpha * DT / 2;

  for (int i = 1; i < height - 1; i++)
  {
    for (int j = 1; j < width - 1; j++)
    {
      double laplacian = C * temp.at<double>(i-1, j) + (1 - 2 * C) *
        temp.at<double>(i, j) + C * temp.at<double>(i+1, j);

      if(j == 1)
      {
        lhs.at<double>(j-1, j-1) = 1 + 2 * C;
        lhs.at<double>(j-1, j) = -1 * C;
        rhs.at<double>(j-1, 0) = laplacian + C * temp.at<double>(i, j-1);
      }
      else if (j == width - 2)
      {
        lhs.at<double>(j-1, j-2)   = -1 * C;
        lhs.at<double>(j-1, j-1)   = 2 + 2 * C;
        rhs.at<double>(j-1, 0) = laplacian + C * temp.at<double>(i, j+1);
      }
      else
      {
        lhs.at<double>(j-1, j-2) = -1 * C;
        lhs.at<double>(j-1, j-1) = 1 + 2 * C;
        lhs.at<double>(j-1, j) = -1 * C;
        rhs.at<double>(j-1, 0) = laplacian;
      }
    }
    // solving tridiagonal
    cv::Mat solved = SolveTriDiagonal(lhs, rhs);

    for (int j = 1; j < width - 1; j++)
    {
      result.at<double>(i, j) = solved.at<double>(j - 1, 0) +
        mask.at<double>(i, j) * (u.at<double>(i, j) - result.at<double>(i, j));
    }
  }
  return result;
}
```

Kode diatas merupakan langkah pertama dari metode ADI yaitu melakukan *sweep* terhadap arah sumbu X. Langkah pertama adalah membentuk matriks *lhs* dan *rhs*, kemudian menyelesaikan matriks tersebut dengan Algoritma Thomas.

```
cv::Mat SweepDirectionY(cv::Mat temp, int width, int height, int alpha)
{
  cv::Mat result = temp;
  cv::Mat lhs = cv::Mat::zeros(height - 2, height - 2, CV_64FC1);
  cv::Mat rhs = cv::Mat::zeros(height - 2, 1, CV_64FC1);
  double C = alpha * DT / 2;

  for (int i = 1; i < width - 1; i++)
  {
    for (int j = 1; j < height - 1; j++)
    {
      double laplacian = C * temp.at<double>(j, i-1) + (1 - 2 * C) *
        temp.at<double>(j, i) + C * temp.at<double>(j, i+1);

      if (j == 1)
      {
        lhs.at<double>(j - 1, j - 1) = 1 + 2 * C;
        lhs.at<double>(j - 1, j) = -1 * C;
        rhs.at<double>(j - 1, 0) = laplacian + C * temp.at<double>(j - 1, i);
      }
      else if (j == height - 2)
      {
        lhs.at<double>(j - 1, j - 2) = -1 * C;
        lhs.at<double>(j - 1, j - 1) = 1 + 2 * C;
        rhs.at<double>(j - 1, 0) = laplacian + C * temp.at<double>(j + 1, i);
      }
      else
      {
        lhs.at<double>(j - 1, j - 2) = -1 * C;
        lhs.at<double>(j - 1, j - 1) = 1 + 2 * C;
        lhs.at<double>(j - 1, j) = -1 * C;
        rhs.at<double>(j - 1, 0) = laplacian;
      }
    }
    // solving tridiagonal
    cv::Mat solved = SolveTriDiagonal(lhs, rhs);

    for (int j = 1; j < height - 1; j++)
    {
      result.at<double>(j, i) = solved.at<double>(j - 1, 0) +
        mask.at<double>(j, i) * (u.at<double>(j, i) - result.at<double>(j, i));
    }
  }
  return result;
}
```

Kode diatas merupakan langkah kedua dari metode ADI yaitu melakukan *sweep* terhadap arah sumbu Y. Sama seperti langkah pada *sweep* arah sumbu X yaitu

membentuk matriks *lhs* dan *rhs*, kemudian menyelesaikan matriks tersebut dengan Algoritma Thomas.

```
cv::Mat SolveTriDiagonal(cv::Mat lhs, cv::Mat rhs)
{
  int n = lhs.rows;
  cv::Mat a           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat b           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat c           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat c_prime = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat d           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat u_new       = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);

  for (int i = 0; i < n; i++)
  {
    b.at<double>(i, 0) = lhs.at<double>(i, i);
  }

  for (int i = 0; i < n-1; i++)
  {
    a.at<double>(i+1, 0)  = lhs.at<double>(i+1, i);
       c.at<double>(i, 0)    = lhs.at<double>(i, i+1);
  }

  c_prime.at<double>(0, 0)  = c.at<double>(0, 0) / b.at<double>(0, 0);
  u_new.at<double>(0, 0)    = rhs.at<double>(0, 0) / b.at<double>(0, 0);

  for (int i = 1; i <  n; i++)
  {
    c_prime.at<double>(i, 0) = c.at<double>(i, 0) / (b.at<double>(i, 0) -
      c_prime.at<double>(i-1, 0) * a.at<double>(i, 0));
    u_new.at<double>(i, 0) = (rhs.at<double>(i, 0) - u_new.at<double>(i-1, 0) *
      a.at<double>(i, 0)) / (b.at<double>(i, 0) - c_prime.at<double>(i-1, 0) *
      a.at<double>(i, 0));
  }

  for (int i = n-2; i >= 0; i--)
  {
    u_new.at<double>(i, 0) -= (c_prime.at<double>(i, 0) *
      u_new.at<double>(i+1, 0));
  }
  return u_new;
}
```

Kode diatas merupakan penjabaran Algoritma Thomas untuk menyelesaikan persamaan matriks tridiagonal yang terbentuk dari *sweep* pada arah sumbu X maupun sumbu Y.

Lampiran 2 Pemrograman *Inpainting* Berbasis GPU

Untuk melakukan pemrograman pada GPU, terlebih dahulu harus menambahkan library CUDA.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
```

Menambahkan dua library yaitu *cuda_runtime.h* dan *device_launch_parameters.h* untuk mengakses kemampuan GPU dalam melakukan proses komputasi.

```
#define DIRECTION_X 0
#define DIRECTION_Y 1

int dimx = 16;
int dimy = 16;
dim3 block(dimx, dimy);
dim3 grid;
```

Mendefinisikan beberapa variable. Yang paling penting adalah definisi dari *block* dan *grid*. Dimensi *block* didefinisikan sejumlah 16 untuk X dan 16 untuk Y, sehingga jumlah *thread* dalam satu *block* adalah 256 *threads*.

```
void GPU_CN2D_ADI(int nIteration)
{
  double *d_result;
  double *d_mask;
  double *d_u;

  int device;
  cudaGetDevice(&device);
  cudaSetDevice(device);

  cudaMalloc((void**) &d_result, WIDTH * HEIGHT * sizeof(double));
  cudaMemcpy(d_result, h_result, WIDTH * HEIGHT * sizeof(double),
    cudaMemcpyHostToDevice);
  cudaMalloc((void**)&d_mask, WIDTH * HEIGHT * sizeof(double));
  cudaMemcpy(d_mask, h_mask, WIDTH * HEIGHT * sizeof(double),
    cudaMemcpyHostToDevice);
  cudaMalloc((void**) &d_u, WIDTH * HEIGHT * sizeof(double));
  cudaMemcpy(d_u, h_result, WIDTH * HEIGHT * sizeof(double),
    cudaMemcpyHostToDevice);
```

```
// allocate for lhs, rhs, and some variable to solve tridiagonal
int n;
if(WIDTH > HEIGHT)
{
  n = WIDTH;
  grid = dim3((int)ceil((float)n/ (block.x * block.y)));
}
else
{
  n = HEIGHT;
  grid = dim3((int)ceil((float)n/ (block.x * block.y)));
}

double *d_lhs;
double *d_rhs;
double *d_a;
double *d_b;
double *d_c;
double *d_c_prime;
double *d_solved;

cudaMallocManaged(&d_lhs, n * n * sizeof(double));
cudaMallocManaged(&d_rhs, n * sizeof(double));
cudaMallocManaged(&d_a, n * sizeof(double));
cudaMallocManaged(&d_b, n * sizeof(double));
cudaMallocManaged(&d_c, n * sizeof(double));
cudaMallocManaged(&d_c_prime, n * sizeof(double));
cudaMallocManaged(&d_solved, n * sizeof(double));

// timer
cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);

for(int i = 0; i < nIteration; i++)
{
     // solving X direction
  GPU_SweepDirection(d_result, d_mask,
    d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime, d_solved, d_u,
    WIDTH, HEIGHT, ALPHA, DIRECTION_X);

     // solving Y direction
  GPU_SweepDirection(d_result, d_mask,
    d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime, d_solved, d_u,
    WIDTH, HEIGHT, ALPHA, DIRECTION_Y);
}

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

```
  cudaEventElapsedTime(&elapsedTime, start, stop);
  printf("\nTime to generate GPU : %3.3f ms\n", elapsedTime);
  cudaEventDestroy(start);
  cudaEventDestroy(stop);

  cudaMemcpy(h_result, d_result, WIDTH * HEIGHT * sizeof(double),
    cudaMemcpyDeviceToHost);

  cudaFree(d_lhs);
  cudaFree(d_rhs);
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
  cudaFree(d_c_prime);
  cudaFree(d_solved);
  cudaFree(d_result);
  cudaFree(d_mask);
  cudaFree(d_u);

  cudaDeviceReset();
}
```

Terdapat beberapa fungsi dari GPU yang digunakan dalam fungsi `GPU_CN2D_ADI` seperti:

1. *cudaSetDevice*

   Berfungsi untuk menunjuk GPU yang akan digunakan.

2. *cudaMalloc*

   Berfungsi untuk mengalokasi memori pada GPU.

3. *cudaMallocManaged*

   Berfungsi untuk mengalokasi memori yang dapat digunakan pada CPU dan GPU.

4. *cudaMemcpy*

   Berfungsi untuk menyalin memori dari CPU ke GPU.

5. *cudaEventCreate*, *cudaEventRecord*, dan *cudaEventElapsedTime*

Berfungsi untuk merekam waktu proses *inpainting* yang terjadi dalam GPU.

6. *cudaEventDestroy*

   Berfungsi untuk dealokasi varibel yang digunakan untuk merekam waktu proses.

7. *cudaFree*

   Berfungsi untuk dealokasi memori pada GPU, agar tidak terjadi *memory leaks*.

```
void GPU_SweepDirection(double *d_result, double *d_mask,
  double *d_lhs, double *d_rhs, double *d_a, double *d_b,
  double *d_c, double *d_c_prime, double *d_solved, double *d_u,
  int width, int height, int alpha, int direction)
{
  double C = alpha * DT / 2;

  if (direction == DIRECTION_X)
  {
    int n = width - 2;
    for (int i = 1; i < height - 1; i++)
    {
      GPU_SweepDirectionX <<< grid, block >>>(i, width, height, C, d_result,
        d_lhs, d_rhs);
      cudaDeviceSynchronize();

      GPU_PrepareToSolveTriDiagonal <<< grid, block >>>(d_lhs, d_rhs, d_a, d_b,
        d_c, d_c_prime, d_solved, n);
      cudaDeviceSynchronize();

      CPU_SolveTriDiagonal(d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime,
        d_solved, n);

      GPU_CopyData <<< grid, block >>>(i, d_result, d_mask, d_solved, d_u,
        width, height, direction);
      cudaDeviceSynchronize();
    }
  }
  else
  {
    int n = height - 2;
    for (int i = 1; i < width - 1; i++)
    {
      GPU_SweepDirectionY <<< grid, block >>>(i, width, height, C, d_result,
        d_lhs, d_rhs);
      cudaDeviceSynchronize();
```

```
      GPU_PrepareToSolveTriDiagonal <<< grid, block >>>(d_lhs, d_rhs, d_a, d_b,
        d_c, d_c_prime, d_solved, n);
      cudaDeviceSynchronize();

      CPU_SolveTriDiagonal(d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime,
        d_solved, n);

      GPU_CopyData <<< grid, block >>>(i, d_result, d_mask, d_solved, d_u,
        width, height, direction);
      cudaDeviceSynchronize();
    }
  }
}
```

Terdapat pemanggilan kernel global seperti *GPU_SweepDirectionX* dan *GPU_SweepDirectionY*. Dapat dilihat bahwa disini perlu ditambahkan variabel *grid* dan *block* dalam setiap pemanggilan fungsi tersebut.

```
__global__ void GPU_SweepDirectionX(int i, int width, int height, double C,
  double *d_result, double *lhs, double *rhs)
{
  int n = width - 2;
  int ix      = threadIdx.x + blockIdx.x * blockDim.x;
  int iy      = threadIdx.y + blockIdx.y * blockDim.y;
  int idx     = iy * (blockDim.x * gridDim.x) + ix;

  if (idx > 0 && idx < width - 1)
  {
    double laplacian = C * d_result[idx + (i-1) * width] + (1 - 2 * C) *
      d_result[idx + i * width] + C * d_result[idx + (i+1) * width];

    if(idx == 1)
    {
      lhs[(idx-1) + (idx-1) * n]  = 1 + 2 * C;
      lhs[idx + (idx-1) * n]      = -1 * C;
      rhs[idx-1] = laplacian + C * d_result[(idx-1) + i * width];
    }
    else if (idx == width - 2)
    {
      lhs[(idx-2) + (idx-1) * n]  = -1 * C;
      lhs[(idx-1) + (idx-1) * n]  = 1 + 2 * C;
      rhs[idx-1] = laplacian + C * d_result[(idx+1) + i * width];
    }
    else
    {
      lhs[(idx-2) + (idx-1) * n]  = -1 * C;
      lhs[(idx-1) + (idx-1) * n]  = 1 + 2 * C;
      lhs[idx + (idx-1) * n]      = -1 * C;
```

```
      rhs[idx-1]                  = laplacian;
    }
  }
}
```

Kode diatas merupakan langkah pertama dari metode ADI yaitu melakukan *sweep* terhadap arah sumbu X. Didalam *kernel* ini, membentuk matriks *lhs* dan *rhs* untuk kemudian diselesaikan pada *kernel* lain.

```
__global__ void GPU_SweepDirectionY(int i, int width, int height, double C,
  double *d_result, double *lhs, double *rhs)
{
  int n = height - 2;
  int ix      = threadIdx.x + blockIdx.x * blockDim.x;
  int iy      = threadIdx.y + blockIdx.y * blockDim.y;
  int idx = iy * (blockDim.x * gridDim.x) + ix;

  if (idx > 0 && idx < height - 1)
  {
    double laplacian = C * d_result[(i-1) + idx * width] + (1 - 2 * C) *
      d_result[i + idx * width] + C * d_result[(i+1) + idx * width];

    if (idx == 1)
    {
      lhs[(idx - 1) + (idx - 1) * n]  = 1 + 2 * C;
      lhs[idx + (idx - 1) * n]        = -1 * C;
      rhs[idx - 1] = laplacian + C * d_result[i + (idx - 1) * width];
    }
    else if (idx == height - 2)
    {
      lhs[(idx - 2) + (idx - 1) * n]  = -1 * C;
      lhs[(idx - 1) + (idx - 1) * n]  = 1 + 2 * C;
      rhs[idx - 1] = laplacian + C * d_result[i + (idx + 1) * width];
    }
    else
    {
      lhs[(idx - 2) + (idx - 1) * n]  = -1 * C;
      lhs[(idx - 1) + (idx - 1) * n]  = 1 + 2 * C;
      lhs[idx + (idx - 1) * n]        = -1 * C;
      rhs[idx - 1]                    = laplacian;
    }
  }
}
```

Kode diatas merupakan langkah kedua dari metode ADI yaitu melakukan *sweep* terhadap arah sumbu Y. Sama seperti *sweep* terhadap sumbu X, didalam *kernel* ini membentuk matriks *lhs* dan *rhs* untuk kemudian diselesaikan pada *kernel* lain.

```
__global__ void GPU_PrepareToSolveTriDiagonal(double *d_lhs, double *d_rhs,
  double *d_a, double *d_b, double *d_c, double *d_c_prime,
  double *d_solved, int n)
{
  int ix  = threadIdx.x + blockIdx.x * blockDim.x;
  int iy  = threadIdx.y + blockIdx.y * blockDim.y;
  int i   = iy * (blockDim.x * gridDim.x) + ix;

  if(i < n)
  {
    d_b[i]    = d_lhs[i + i * n];
  }

  if(i < n-1)
  {
    d_a[i+1]  = d_lhs[(i+1) + i * n];
    d_c[i]    = d_lhs[i + (i+1) * n];
  }
}
```

Kode diatas adalah *kernel* untuk mempersiapkan nilai variable yang digunakan untuk menyelesaikan system matriks tridiagonal.

```
void CPU_SolveTriDiagonal(double *d_lhs, double *d_rhs, double *d_a,
  double *d_b, double *d_c, double *d_c_prime, double *d_solved, int n)
{
  d_c_prime[0] = d_c[0] / d_b[0];
  d_solved[0] = d_rhs[0] / d_b[0];

  for (int x = 1; x < n; x++)
  {
    d_c_prime[x] = d_c[x] / (d_b[x] - d_c_prime[x - 1] * d_a[x]);
    d_solved[x] = (d_rhs[x] - d_solved[x - 1] * d_a[x]) /
      (d_b[x] - d_c_prime[x - 1] * d_a[x]);
  }

  for (int backward = n - 2; backward >= 0; backward--)
  {
    d_solved[backward] -= (d_c_prime[backward] * d_solved[backward + 1]);
  }
}
```

Kode fungsi diatas digunakan untuk menyelesaikan system matriks tridiagonal dengan algoritma Thomas. Fungsi diatas dipanggil dari CPU. Untuk mencegah proses transfer variable antara CPU dan GPU yang tentunya memakan waktu, digunakan *shared memory* yang dipanggil dengan fungsi *cudaMallocManaged* yang sudah dijelaskan sebelumnya.

```
__global__ void GPU_CopyData(int i, double *d_result, double *d_mask,
  double *d_solved, double *d_u, int width, int height, int direction)
{
  int ix  = threadIdx.x + blockIdx.x * blockDim.x;
  int iy  = threadIdx.y + blockIdx.y * blockDim.y;
  int idx = iy * (blockDim.x * gridDim.x) + ix;

  if(direction == DIRECTION_X)
  {
    if(idx > 0 && idx < width - 1)
    {
      d_result[idx + i * width] = d_solved[(idx - 1)] +
        d_mask[idx + i * width] * (d_u[idx + i * width] –
        d_result[idx + i * width]);;
    }
  }
  else
  {
    if(idx > 0 && idx < height - 1)
    {
      d_result[i + idx * width] = d_solved[(idx - 1)] +
        d_mask[i + idx * width] * (d_u[i + idx * width] –
        d_result[i + idx * width]);;
    }
  }
}
```

Kode diatas berfungsi menyalin nilai hasil penyelesaian matriks tridiagonal untuk selanjutnya dilakukan proses iterasi berikutnya.

```
cv::Mat GetResultImage()
{
  cv::Mat result = cv::Mat::zeros(HEIGHT, WIDTH, CV_64FC1);

  for (int i = 0; i < HEIGHT; i++)
  {
```

```
    for (int j = 0; j < WIDTH; j++)
    {
      int index = j + i * WIDTH;
      result.at<double>(i, j) = h_result[index];
    }
  }

  result.convertTo(result, CV_8UC1);

  return result;
}
```

Kode diatas digunakan untuk mengkonversi pointer array menjadi bentuk Mat untuk selanjutnya ditampilkan.

Lampiran 3 Kode Lengkap *Inpainting* Berbasis CPU (cpu_adi.h)

```
////////////////////////////////////////////////////////////////////////
//
// Implicit Crank-Nicholson with ADI-solver to inpaint an image
//
////////////////////////////////////////////////////////////////////////

#define ALPHA 1
#define DT 0.5

////////////////////////////////////////////////////////////////////////
//
//  Function: SolveTriDiagonal
//
//  Applies Thomas's algorithm for solving tridiagonal matrices.
//  The primary use of this function is for solving the system of equations
//  that arises from Backward Euler.
//
////////////////////////////////////////////////////////////////////////
cv::Mat SolveTriDiagonal(cv::Mat lhs, cv::Mat rhs)
{
  int n = lhs.rows;
  cv::Mat a           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat b           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat c           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat c_prime = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat d           = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);
  cv::Mat u_new       = cv::Mat::zeros(lhs.rows, 1, CV_64FC1);

  for (int i = 0; i < n; i++)
  {
    b.at<double>(i, 0) = lhs.at<double>(i, i);
  }

  for (int i = 0; i < n-1; i++)
  {
    a.at<double>(i+1, 0)  = lhs.at<double>(i+1, i);
    c.at<double>(i, 0)    = lhs.at<double>(i, i+1);
  }

  c_prime.at<double>(0, 0)  = c.at<double>(0, 0) / b.at<double>(0, 0);
  u_new.at<double>(0, 0)    = rhs.at<double>(0, 0) / b.at<double>(0, 0);

  for (int i = 1; i <  n; i++)
  {
    c_prime.at<double>(i, 0) = c.at<double>(i, 0) / (b.at<double>(i, 0) -
      c_prime.at<double>(i-1, 0) * a.at<double>(i, 0));
    u_new.at<double>(i, 0)   = (rhs.at<double>(i, 0) - u_new.at<double>(i-1, 0)
      * a.at<double>(i, 0)) / (b.at<double>(i, 0) - c_prime.at<double>(i-1, 0)
      * a.at<double>(i, 0));
  }
```

```
  for (int i = n-2; i >= 0; i--)
  {
    u_new.at<double>(i, 0) -= (c_prime.at<double>(i, 0) *
      u_new.at<double>(i+1, 0));
  }

  return u_new;
}

//////////////////////////////////////////////////////
//
//  Function : SweepDirectionX
//  Sweep through X direction
//
//////////////////////////////////////////////////////
cv::Mat SweepDirectionX(cv::Mat temp, int width, int height, int alpha)
{
  cv::Mat result = temp;

  cv::Mat lhs = cv::Mat::zeros(width - 2, width - 2, CV_64FC1);
  cv::Mat rhs = cv::Mat::zeros(width - 2, 1, CV_64FC1);

  double C = alpha * DT / 2;

  for (int i = 1; i < height - 1; i++)
  {
    for (int j = 1; j < width - 1; j++)
    {
      double laplacian = C * temp.at<double>(i-1, j) + (1 - 2 * C) *
        temp.at<double>(i, j) + C * temp.at<double>(i+1, j);

      if(j == 1)
      {
        lhs.at<double>(j-1, j-1) = 1 + 2 * C;
        lhs.at<double>(j-1, j)     = -1 * C;
        rhs.at<double>(j-1, 0) = laplacian + C * temp.at<double>(i, j-1);
      }
      else if (j == width - 2)
      {
        lhs.at<double>(j-1, j-2)   = -1 * C;
        lhs.at<double>(j-1, j-1)   = 2 + 2 * C;
        rhs.at<double>(j-1, 0) = laplacian + C * temp.at<double>(i, j+1);
      }
      else
      {
        lhs.at<double>(j-1, j-2) = -1 * C;
        lhs.at<double>(j-1, j-1) = 1 + 2 * C;
        lhs.at<double>(j-1, j) = -1 * C;
        rhs.at<double>(j-1, 0) = laplacian;
      }
    }
```

```
    // solving tridiagonal
    cv::Mat solved = SolveTriDiagonal(lhs, rhs);

    for (int j = 1; j < width - 1; j++)
    {
      result.at<double>(i, j) = solved.at<double>(j - 1, 0) +
       mask.at<double>(i, j) * (u.at<double>(i, j) -
       result.at<double>(i, j));
    }
  }

  return result;
}

//////////////////////////////////////////////////////
//
//  Function : SweepDirectionY
//  Sweep through Y direction
//
//////////////////////////////////////////////////////
cv::Mat SweepDirectionY(cv::Mat temp, int width, int height, int alpha)
{
  cv::Mat result = temp;

  cv::Mat lhs = cv::Mat::zeros(height - 2, height - 2, CV_64FC1);
  cv::Mat rhs = cv::Mat::zeros(height - 2, 1, CV_64FC1);

  double C = alpha * DT / 2;

  for (int i = 1; i < width - 1; i++)
  {
    for (int j = 1; j < height - 1; j++)
    {
      double laplacian = C * temp.at<double>(j, i-1) + (1 - 2 * C) *
        temp.at<double>(j, i) + C * temp.at<double>(j, i+1);

      if (j == 1)
      {
        lhs.at<double>(j - 1, j - 1) = 1 + 2 * C;
        lhs.at<double>(j - 1, j) = -1 * C;
        rhs.at<double>(j - 1, 0) = laplacian + C *temp.at<double>(j - 1, i);
      }
      else if (j == height - 2)
      {
        lhs.at<double>(j - 1, j - 2) = -1 * C;
        lhs.at<double>(j - 1, j - 1) = 1 + 2 * C;
        rhs.at<double>(j - 1, 0) = laplacian + C *temp.at<double>(j + 1, i);
      }
      else
      {
        lhs.at<double>(j - 1, j - 2) = -1 * C;
        lhs.at<double>(j - 1, j - 1) = 1 + 2 * C;
        lhs.at<double>(j - 1, j) = -1 * C;
```

```
        rhs.at<double>(j - 1, 0) = laplacian;
      }
    }

    // solving tridiagonal
    cv::Mat solved = SolveTriDiagonal(lhs, rhs);

    for (int j = 1; j < height - 1; j++)
    {
      result.at<double>(j, i) = solved.at<double>(j - 1, 0) +
        mask.at<double>(j, i) * (u.at<double>(j, i) - result.at<double>(j, i));
    }
  }

  return result;
}

//////////////////////////////////////////////////////
//
//  Function : CPU_CN2D_ADI
//
//  Apply Crank-Nicholson 2D into given image
//  and solve matrix problem with Thomas's algorithm
//
//////////////////////////////////////////////////////
cv::Mat CPU_CN2D_ADI(int nIteration)
{
  cv::Mat result = img;
  u = img.clone();

  mask.convertTo(mask, CV_64FC1);
  for (int i = 0; i < HEIGHT; i++)
  {
    for (int j = 0; j < WIDTH; j++)
    {
      mask.at<double>(i, j) = 1-ceil(mask.at<double>(i, j) / 255);
    }
  }

  result.convertTo(result, CV_64FC1);

  //timer
  clock_t start, stop, selisih;
  float elapsedTime;
  start = clock();

  for(int i = 0; i < nIteration; i++)
  {
    cv::Mat temp = result;

    // solving X direction
    result = SweepDirectionX(temp, WIDTH, HEIGHT, ALPHA);
```

```
    // solving Y direction
    result = SweepDirectionY(temp, WIDTH, HEIGHT, ALPHA);
  }

  stop = clock();
  selisih = stop - start;
  elapsedTime = selisih / (float)CLOCKS_PER_SEC;
  printf("\nTime to generate CPU : %.3f ms\n", elapsedTime*1000);

  result.convertTo(result, CV_8UC1);
  return result;
}
```

Lampiran 4 Kode Lengkap *Inpainting* Berbasis GPU (gpu_adi.h)

```
////////////////////////////////////////////////////////////////////////////
//
// Implicit Crank-Nicholson with ADI-solver to inpaint an image
// Using CUDA to accelerate
//
////////////////////////////////////////////////////////////////////////////

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#define DIRECTION_X 0
#define DIRECTION_Y 1

// set grid and block dimension
int dimx = 16;
int dimy = 16;
dim3 block(dimx, dimy);
dim3 grid;

////////////////////////////////////////////////////
//
//  Function : GPU_SweepDirectionX
//  Sweep through X direction
//
////////////////////////////////////////////////////
__global__ void GPU_SweepDirectionX(int i, int width, int height, double C,
  double *d_result, double *lhs, double *rhs)
{
  int n = width - 2;

  int ix      = threadIdx.x + blockIdx.x * blockDim.x;
  int iy      = threadIdx.y + blockIdx.y * blockDim.y;
  int idx     = iy * (blockDim.x * gridDim.x) + ix;

  if (idx > 0 && idx < width - 1)
  {
    double laplacian = C * d_result[idx + (i-1) * width] + (1 - 2 * C) *
      d_result[idx + i * width] + C * d_result[idx + (i+1) * width];

    if(idx == 1)
    {
      lhs[(idx-1) + (idx-1) * n]  = 1 + 2 * C;
      lhs[idx + (idx-1) * n] = -1 * C;
      rhs[idx-1] = laplacian + C * d_result[(idx-1) + i * width];
    }
    else if (idx == width - 2)
    {
      lhs[(idx-2) + (idx-1) * n]  = -1 * C;
      lhs[(idx-1) + (idx-1) * n]  = 1 + 2 * C;
      rhs[idx-1] = laplacian + C * d_result[(idx+1) + i * width];
```

```
    }
    else
    {
      lhs[(idx-2) + (idx-1) * n]  = -1 * C;
      lhs[(idx-1) + (idx-1) * n]  = 1 + 2 * C;
      lhs[idx + (idx-1) * n]      = -1 * C;
      rhs[idx-1]                  = laplacian;
    }
  }
}

////////////////////////////////////////////////////
//
//  Function : GPU_SweepDirectionY
//
//  Sweep through Y direction
//
////////////////////////////////////////////////////
__global__ void GPU_SweepDirectionY(int i, int width, int height, double C,
  double *d_result, double *lhs, double *rhs)
{
  int n = height - 2;

  int ix  = threadIdx.x + blockIdx.x * blockDim.x;
  int iy  = threadIdx.y + blockIdx.y * blockDim.y;
  int idx = iy * (blockDim.x * gridDim.x) + ix;

  if (idx > 0 && idx < height - 1)
  {
    double laplacian = C * d_result[(i-1) + idx * width] + (1 - 2 * C) *
      d_result[i + idx * width] + C * d_result[(i+1) + idx * width];

    if (idx == 1)
    {
      lhs[(idx - 1) + (idx - 1) * n]  = 1 + 2 * C;
      lhs[idx + (idx - 1) * n]        = -1 * C;
      rhs[idx - 1]  = laplacian + C * d_result[i + (idx - 1) * width];
    }
    else if (idx == height - 2)
    {
      lhs[(idx - 2) + (idx - 1) * n]  = -1 * C;
      lhs[(idx - 1) + (idx - 1) * n]  = 1 + 2 * C;
      rhs[idx - 1] = laplacian + C * d_result[i + (idx + 1) * width];
    }
    else
    {
      lhs[(idx - 2) + (idx - 1) * n]  = -1 * C;
      lhs[(idx - 1) + (idx - 1) * n]  = 1 + 2 * C;
      lhs[idx + (idx - 1) * n]        = -1 * C;
      rhs[idx - 1]                    = laplacian;
    }
  }
}
```

```
////////////////////////////////////////////////////////////////////////////
//
//  Function: GPU_SolveTriDiagonal
//
//  Prepare some variable to be used to solve tridiagonal matrix
//
////////////////////////////////////////////////////////////////////////////
__global__ void GPU_PrepareToSolveTriDiagonal(double *d_lhs, double *d_rhs,
  double *d_a, double *d_b, double *d_c, double *d_c_prime,
  double *d_solved, int n)
{
  int ix  = threadIdx.x + blockIdx.x * blockDim.x;
  int iy  = threadIdx.y + blockIdx.y * blockDim.y;
  int i   = iy * (blockDim.x * gridDim.x) + ix;

  if(i < n)
  {
    d_b[i]    = d_lhs[i + i * n];
  }

  if(i < n-1)
  {
    d_a[i+1]  = d_lhs[(i+1) + i * n];
    d_c[i]    = d_lhs[i + (i+1) * n];
  }
}

////////////////////////////////////////////////////////////////////////////
//
//  Function: CPU_SolveTriDiagonal
//
//  Applies Thomas's algorithm for solving tridiagonal matrices.
//  The primary use of this function is for solving the system of equations
//  that arises from Backward Euler.
//
////////////////////////////////////////////////////////////////////////////
void CPU_SolveTriDiagonal(double *d_lhs, double *d_rhs, double *d_a, double
*d_b, double *d_c, double *d_c_prime, double *d_solved, int n)
{
  d_c_prime[0] = d_c[0] / d_b[0];
  d_solved[0] = d_rhs[0] / d_b[0];

  for (int x = 1; x < n; x++)
  {
    d_c_prime[x] = d_c[x] / (d_b[x] - d_c_prime[x - 1] * d_a[x]);
    d_solved[x] = (d_rhs[x] - d_solved[x - 1] * d_a[x]) / (d_b[x] -
      d_c_prime[x - 1] * d_a[x]);
  }

  for (int backward = n - 2; backward >= 0; backward--)
  {
    d_solved[backward] -= (d_c_prime[backward] * d_solved[backward + 1]);
```

```
  }
}

__global__ void GPU_CopyData(int i, double *d_result, double *d_mask, double
*d_solved, double *d_u, int width, int height, int direction)
{
  int ix  = threadIdx.x + blockIdx.x * blockDim.x;
  int iy  = threadIdx.y + blockIdx.y * blockDim.y;
  int idx = iy * (blockDim.x * gridDim.x) + ix;

  if(direction == DIRECTION_X)
  {
    if(idx > 0 && idx < width - 1)
    {
      d_result[idx + i * width] = d_solved[(idx - 1)] +
       d_mask[idx + i * width] * (d_u[idx + i * width] -
       d_result[idx + i * width]);
    }
  }
  else
  {
    if(idx > 0 && idx < height - 1)
    {
      d_result[i + idx * width] = d_solved[(idx - 1)] +
      d_mask[i + idx * width] * (d_u[i + idx * width] -
      d_result[i + idx * width]);
    }
  }
}

__global__ void GPU_PrintData(double *d_lhs, double *d_rhs, int i, int n)
{
  for (int k = 0; k < n; k++)
  {
    for(int j = 0; j < n;j++)
    {
      int index = j + k * n;
      printf("%.0f ", d_lhs[index]);
    }
    printf("| %.0f\n", d_rhs[k]);
  }
}

void GPU_SweepDirection(double *d_result, double *d_mask,
  double *d_lhs, double *d_rhs, double *d_a, double *d_b, double *d_c,
  double *d_c_prime, double *d_solved, double *d_u,
  int width, int height, int alpha, int direction)
{
  double C = alpha * DT / 2;

  if (direction == DIRECTION_X)
  {
    int n = width - 2;
```

```
    for (int i = 1; i < height - 1; i++)
    {
      GPU_SweepDirectionX <<< grid, block >>>(i, width, height, C, d_result,
        d_lhs, d_rhs);
      cudaDeviceSynchronize();

      GPU_PrepareToSolveTriDiagonal <<< grid, block >>>(d_lhs, d_rhs, d_a, d_b,
        d_c, d_c_prime, d_solved, n);
      cudaDeviceSynchronize();

      CPU_SolveTriDiagonal(d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime,
        d_solved, n);

      GPU_CopyData <<< grid, block >>>(i, d_result, d_mask, d_solved, d_u,
        width, height, direction);
      cudaDeviceSynchronize();
    }
  }
  else
  {
    int n = height - 2;

    for (int i = 1; i < width - 1; i++)
    {
      GPU_SweepDirectionY <<< grid, block >>>(i, width, height, C, d_result,
      d_lhs, d_rhs);
      cudaDeviceSynchronize();

      GPU_PrepareToSolveTriDiagonal <<< grid, block >>>(d_lhs, d_rhs, d_a, d_b,
        d_c, d_c_prime, d_solved, n);
      cudaDeviceSynchronize();

      CPU_SolveTriDiagonal(d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime,
        d_solved, n);

      GPU_CopyData <<< grid, block >>>(i, d_result, d_mask, d_solved, d_u,
        width, height, direction);
      cudaDeviceSynchronize();
    }
  }
}

void GPU_CN2D_ADI(int nIteration)
{
  double *d_result;
  double *d_mask;
  double *d_u;

  int device;
  cudaGetDevice(&device);
  cudaSetDevice(device);
```

```
cudaMalloc((void**) &d_result, WIDTH * HEIGHT * sizeof(double));
cudaMemcpy(d_result, h_result, WIDTH * HEIGHT * sizeof(double),
  cudaMemcpyHostToDevice);
cudaMalloc((void**)&d_mask, WIDTH * HEIGHT * sizeof(double));
cudaMemcpy(d_mask, h_mask, WIDTH * HEIGHT * sizeof(double),
  cudaMemcpyHostToDevice);
cudaMalloc((void**) &d_u, WIDTH * HEIGHT * sizeof(double));
cudaMemcpy(d_u, h_result, WIDTH * HEIGHT * sizeof(double),
  cudaMemcpyHostToDevice);

// allocate for lhs, rhs, and some variable to solve tridiagonal
int n;
if(WIDTH > HEIGHT)
{
  n = WIDTH;
  grid = dim3((int)ceil((float)n/ (block.x * block.y)));
}
else
{
  n = HEIGHT;
  grid = dim3((int)ceil((float)n/ (block.x * block.y)));
}

double *d_lhs;
double *d_rhs;
double *d_a;
double *d_b;
double *d_c;
double *d_c_prime;
double *d_solved;

cudaMallocManaged(&d_lhs, n * n * sizeof(double));
cudaMallocManaged(&d_rhs, n * sizeof(double));
cudaMallocManaged(&d_a, n * sizeof(double));
cudaMallocManaged(&d_b, n * sizeof(double));
cudaMallocManaged(&d_c, n * sizeof(double));
cudaMallocManaged(&d_c_prime, n * sizeof(double));
cudaMallocManaged(&d_solved, n * sizeof(double));

// timer
cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);

for(int i = 0; i < nIteration; i++)
{
  // solving X direction
  GPU_SweepDirection(d_result, d_mask,
    d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime, d_solved, d_u,
    WIDTH, HEIGHT, ALPHA, DIRECTION_X);
```

```
        // solving Y direction
    GPU_SweepDirection(d_result, d_mask,
      d_lhs, d_rhs, d_a, d_b, d_c, d_c_prime, d_solved, d_u,
      WIDTH, HEIGHT, ALPHA, DIRECTION_Y);
  }

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime, start, stop);
  printf("\nTime to generate GPU : %3.3f ms\n", elapsedTime);
  cudaEventDestroy(start);
  cudaEventDestroy(stop);

  cudaMemcpy(h_result, d_result, WIDTH * HEIGHT * sizeof(double),
    cudaMemcpyDeviceToHost);

  cudaFree(d_lhs);
  cudaFree(d_rhs);
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
  cudaFree(d_c_prime);
  cudaFree(d_solved);
  cudaFree(d_result);
  cudaFree(d_mask);
  cudaFree(d_u);

  cudaDeviceReset();
}
```

Lampiran 5 Kode Lengkap untuk Utilitas (utility.h)

```
////////////////////////////////////////
//
//  Used to convert image into 1D pointer
//  vice versa
//
////////////////////////////////////////

#include <stdio.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <opencv2/opencv.hpp>

int WIDTH = 0;
int HEIGHT = 0;

cv::Mat img;
cv::Mat mask;
cv::Mat u;

double* h_img;
double* h_mask;
double* h_result;

int image_id;
#define IMAGE_1 1
#define IMAGE_2 2
#define IMAGE_3 3
#define IMAGE_4 4
#define IMAGE_5 5

void DeleteResources()
{
  printf("====================== END OF SOURCE ========================\n");
  printf("delete all array data\n");
  delete[] h_img;
  delete[] h_mask;
  delete[] h_result;
}

void CPU_Init()
{
  // init original and mask image
  switch(image_id)
  {
    case IMAGE_1:
      img   = cv::imread("../images/original.png", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/mask.png", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    case IMAGE_2:
```

```
      img   = cv::imread("../images/lena_rusak.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/lena_mask.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    case IMAGE_3:
      img   = cv::imread("../images/car1.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/car_mask1.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    case IMAGE_4:
      img   = cv::imread("../images/car2.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/car_mask2.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    case IMAGE_5:
      img   = cv::imread("../images/car3.jpg", CV_LOAD_IMAGE_GRAYSCALE);
      mask  = cv::imread("../images/car_mask3.jpg", CV_LOAD_IMAGE_GRAYSCALE);
    break;
    default:
      printf("\nWrong image input!\n");
    break;
  }

  WIDTH   = img.cols;
  HEIGHT  = img.rows;

  img.convertTo(img, CV_64FC1);
  mask.convertTo(mask, CV_64FC1);

  h_img     = new double[WIDTH * HEIGHT];
  h_mask    = new double[WIDTH * HEIGHT];
  h_result  = new double[WIDTH * HEIGHT];

  for (int i = 0; i < HEIGHT; i++)
  {
    for (int j = 0; j < WIDTH; j++)
    {
      int index = j + i * WIDTH;
      h_img[index]  = img.at<double>(i, j);
      h_mask[index] = 1-ceil(mask.at<double>(i, j) / 255);
    }
  }

  std::memcpy(h_result, h_img, WIDTH * HEIGHT * sizeof(double));
}

cv::Mat GetResultImage()
{
  cv::Mat result = cv::Mat::zeros(HEIGHT, WIDTH, CV_64FC1);

  for (int i = 0; i < HEIGHT; i++)
  {
    for (int j = 0; j < WIDTH; j++)
    {
      int index = j + i * WIDTH;
      result.at<double>(i, j) = h_result[index];
```

```
    }
  }

  result.convertTo(result, CV_8UC1);

  return result;
}

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

  for (int i = 0; i < height; i++)
  {
    for (int 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 psnr(cv::Mat & img_src, cv::Mat & img_compressed)
{
  int D = 255;

  return (10 * log10((D*D) / mse(img_src, img_compressed)));
}

double sigma(cv::Mat & m, int i, int j, int block_size)
{
  double sd = 0;
  cv::Mat m_tmp = m(cv::Range(i, i + block_size),
    cv::Range(j, j + block_size));
  cv::Mat m_squared(block_size, block_size, CV_64F);
  multiply(m_tmp, m_tmp, m_squared);
  // E(x)
  double avg = cv::mean(m_tmp)[0];
  // E(x²)
  double avg_2 = cv::mean(m_squared)[0];

  sd = sqrt(avg_2 - avg * avg);

  return sd;
}

double cov(cv::Mat & m1, cv::Mat & m2, int i, int j, int block_size)
```

```
{
  cv::Mat m3 = cv::Mat::zeros(block_size, block_size, m1.depth());
  cv::Mat m1_tmp = m1(cv::Range(i, i + block_size),
    cv::Range(j, j + block_size));
  cv::Mat m2_tmp = m2(cv::Range(i, i + block_size),
    cv::Range(j, j + block_size));
  multiply(m1_tmp, m2_tmp, m3);

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

  return sd_ro;
}

double ssim(cv::Mat & img_src, cv::Mat & img_compressed, int block_size, bool
show_progress)
{
  // typically 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    = cv::mean(img_src(cv::Range(k, k + block_size),
        cv::Range(l, l + block_size)))[0];
      double avg_r    = cv::mean(img_compressed(cv::Range(k, k + block_size),
        cv::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)
    {
      printf("\r>>SSIM [%d%]", (int)((((double)k) / nbBlockPerHeight) * 100));
    }
  }

  ssim /= nbBlockPerHeight * nbBlockPerWidth;
```

```
  if (show_progress)
  {
    printf("SSIM : %.3f", ssim);
  }

  return ssim;
}
```

Lampiran 6 Kode Lengkap Entry-Point (kernel.cu)

```
#include "utility.h"
#include "cpu_adi.h"
#include "gpu_adi.h"
#include "cpu_explicit.h"

#define USE_CPU 1
#define USE_GPU 2

int main()
{
  int method;
  int number_of_iteration;
  printf("Image Inpainting using ADI\n\n");
  printf("Choose an image with size\n1. 483x405\n2. 1024x1024\n3. 1532x1021\n4.
2189x1459\n5. 2736x1824\n");
  printf("Please input your image choice : ");
  scanf("%d", &image_id);
  printf("Please input 1 for CPU and 2 for GPU : ");
  scanf("%d", &method);
  printf("Please input number of iteration : ");
  scanf("%d", &number_of_iteration);

  // init array data for CPU
  CPU_Init();

  cv::Mat result;

  switch(method)
  {
    case USE_CPU:
      printf("\nusing CPU with %d iterations...\n", number_of_iteration);

      // loop for given times and get the result
      result = CPU_CN2D_ADI(number_of_iteration);
    break;
    case USE_GPU:
      printf("\nusing GPU with %d iterations...\n", number_of_iteration);

      // loop for given times
      GPU_CN2D_ADI(number_of_iteration);
      // get result Image
      result = GetResultImage();
    break;
    default:
      printf("\nWrong method input!\n");
    break;
  }

  // show Image
  cv::namedWindow("original", CV_WINDOW_NORMAL);
```

```
cv::namedWindow("mask", CV_WINDOW_NORMAL);
cv::namedWindow("result", CV_WINDOW_NORMAL);

switch(image_id)
{
  case IMAGE_1:
    cv::imshow("original", cv::imread("../images/original.png",
      CV_LOAD_IMAGE_GRAYSCALE));
    cv::imshow("mask", cv::imread("../images/mask.png",
      CV_LOAD_IMAGE_GRAYSCALE));
  break;
  case IMAGE_2:
    cv::imshow("original", cv::imread("../images/lena_rusak.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
    cv::imshow("mask", cv::imread("../images/lena_mask.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
  break;
  case IMAGE_3:
    cv::imshow("original", cv::imread("../images/car1.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
    cv::imshow("mask", cv::imread("../images/car_mask1.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
  break;
  case IMAGE_4:
    cv::imshow("original", cv::imread("../images/car2.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
    cv::imshow("mask", cv::imread("../images/car_mask2.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
  break;
  case IMAGE_5:
    cv::imshow("original", cv::imread("../images/car3.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
    cv::imshow("mask", cv::imread("../images/car_mask3.jpg",
      CV_LOAD_IMAGE_GRAYSCALE));
  break;
  default:
    printf("\nWrong image input!\n");
  break;
}

cv::imshow("result", result);

if(image_id == IMAGE_5)
{
  cv::Mat asli = cv::imread("../images/car.jpg", CV_LOAD_IMAGE_GRAYSCALE);

  asli.convertTo(asli, CV_64FC1);
  result.convertTo(result, CV_64FC1);

  printf("\nQuality Measurement");
  printf("\nMSE : %.2f", mse(asli, result));
  printf("\nPSNR: %.2f", psnr(asli, result));
  printf("\nSSIM: %.2f\n", ssim(asli, result, 8, false));
```

```
    }

    DeleteResources();

    cv::waitKey();
    cv::destroyWindow("original");
    cv::destroyWindow("mask");
    cv::destroyWindow("result");

    return 0;
}
```