Adeniyi Adeboye

CIS 677 – Project 3

**GPU-Accelerated computation of Voronoi Diagram**

**Overview**

The main idea is to write a CUDA-based massively multi-threaded application that when given a set of seeds and a 2-dimensional canvas, computes a discrete approximation of the corresponding Voronoi diagram.

**Background**

As explained by *Rong, G. and Tan, T., 2008*, A **Voronoi diagram** is described below:

*Let **G** be a space consisting of a set of points, and **S** be a subset of points in **G** that are designated as seeds. A **Voronoi diagram** of **S** is defined to be a partition of **G** into **subspaces**, called **Voronoi cells**, where each **subspace** corresponds to one seed **s** and contains all points of **G** that are closer to the corresponding seed **s** than to any other seed, with respect to some fixed distance function.*

*For simplicity, points of **G** that are of equal distance to two or more seeds are assigned arbitrarily to any of these seeds. A point of **G** that is closest to three or more seeds is a **Voronoi vertex** of these seeds.*

For this project, G will be a discrete space that can be stored as a texture unit in the GPU. *For example, G is a 2D grid of 512×512 grid points where each grid point/pixel is a texel (texture element).* As G is a discrete space, there may be none or more than one grid point with equal (closest) distance to three or more seeds. So, A Voronoi vertex refers to any grid point whose (***at most eight) neighbouring grid points*** (that sharing a side or a corner with the grid point) belong to two or more other Voronoi cells (*Rong, G. and Tan, T., 2008).*

**Method**

In this project, we will be using Jump Flood Algorithm to computes and implement Voronoi diagram on GPU. Jump Flood Algorithm (JFA) was first computed and used by Rong, G. and Tan, T., 2008, as an efficient and effective computational paradigm to compute the Voronoi diagram on a 2D and 3D grid texture unit on GPU.

**What is Jump Flood Algorithm (JFA)?**

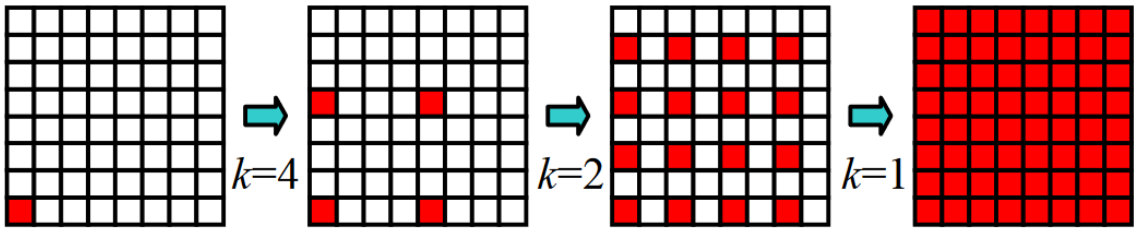
Jump Flood Algorithm (JFA) includes every grid point (x,y) in a texture unit/discrete space passing on its closest seed s found so far at an indicated position, to (at most eight) neighbouring grid points which are at (x+i, y+j) where i, j ∈ {–k, 0, k}. And each of these neighbouring grid points (x', y') will decide if the seed is closest to them based on their position relative to the seed position, if the seed is closest to them, then that neighbouring grid point content is updated, i.e. the grid point is assigned to its closest seed.

There are log n rounds of flood in JFA with step lengths of K (n/2, n/4 ……1), where n is the size of the grid (i.e., if 2D grid 512x512, then n = 512). This log n rounds of step length indicates how the flooding (i.e., passing on seed content) is being propagated, in each log n round every grid point passes on its content (i.e., seed) to other grid points at (x+i, y+j) where i, j ∈ **{–k, 0, k}**.

*Note: **k** indicates the offset (how wide is the distance) between a grid point and its neighbouring grid points on both x any y axis. As step length decreases the offset between each grid point and its neighbours decreases.*

As described by (Rong, G. and Tan, T., 2008), the step length is done in log n rounds by halving the step length iteratively until it reaches 1, the step length can also be done in a reverse way i.e., doubling the step length but starting from 1 instead of starting from half n/2 but this doubling approach is said to generate more errors in Jump Flooding than the halving approach.

The figure below gives a pictorial illustration of JFA propagation based on the halving step length approach:




Here, Jump Flooding is propagated with halving step approach, n(size) = 8, so the first step length is 4, followed by 2 and the finally followed by 1.

The distance metric used in this project and in Rong, G. and Tan, T., 2008 to assess how close seed positions are to their closest seed is **Euclidean distance**.

Side Note: **Flooding** – passing on grid point content to neighbouring grid points

**Jump** – step length (offset of neighbouring grid points to each grid point).

**Implementation of JFA for our project**

1. Create 3 vectors, one that will contain the seeds, one for the Voronoi with 2-D size, and last one for also for seeds but this will have the colours (r,g,b) for the respective seeds.

2. Generate x and y position values for each seed in the seed vector making sure the position is greater than zero and less than the size of the Voronoi grid.

3. Randomly assign seed number to some grid points (x,y) in the Voronoi vector.

4. Assign randomly generated colours to each of the seeds in the seed colour vector.

5. Allocate CUDA memory for new vectors that will be replicate of the seed vectors and Voronoi vectors on Device (i.e. on CUDA).

6. Transfer content of the of the seed vectors on host to device (i.e. CUDA) and transfer content of Voronoi vectors to two device vectors (Ping and Pong) (i.e. on CUDA).

7. Declare and instantiate the thread Size, Block dimension and grid dimension that will be used to run JFA on CUDA. Also, Mutex that will be used for the atomic operation on CUDA.

8. Run JFA for log n rounds on CUDA, after every round you swap ping and pong i.e. updating the seed content of every grid point after every round.

**JFA on GPU**

Here, the Jump flooding is done in parallel in which every grid point is passing its closest seed to its neighbouring grid points concurrently **BUT** the update of each neighbouring grid point with respect to its closest seed based on measuring the Euclidean distance between the grid point and the position of proposed seed to be assigned to, is done atomically using ***AtomicCAS().*** This is because we want to ***avoid race condition*** i.e., we do not want two or more threads updating the same grid point with the same or different seed numbers ***at the same time***, which we result to the grid point not being assigned to the correct seed or assigned to a seed that is also meant to have been **re-updated** by the most recent thread.

So, using ***atomicCAS()*** we **lock** the update part of the neighbouring grid points with new seeds based on Euclidean distance, with the help of **Mutex –** *which make sure while a thread is running code inside a lock, it shuts all the other threads out of the lock.*

Note that the threads running in the GPU are group of 32 threads (**warp**) that execute in lockstep, i.e., they synchronize after every lock step, this means each of the 32 threads wait for the remaining 32 (everyone) threads to exits the lock's loop before they all get release from **atomicCAS()**.

The update of the neighbouring grid point's seeds is written to new Pong vector for the next log n round, i.e., after the present log n round we will swap Ping and Pong vectors, and then used the new updated closest seed for that grid point in the next round.

Overall, the log n rounds for JFA is all done on GPU and by the end of the rounds i.e., when step length is == 1, we would have assigned each grid point in the Voronoi grid to its closest seed.

9. Transfer the Pong vector content on device (i.e., CUDA) back to the Voronoi vector on host.

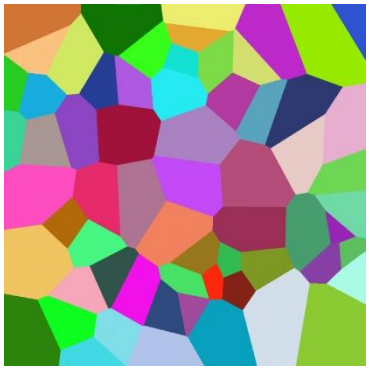
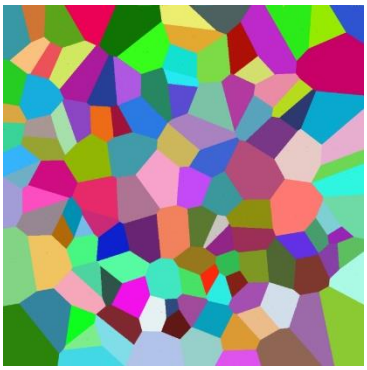
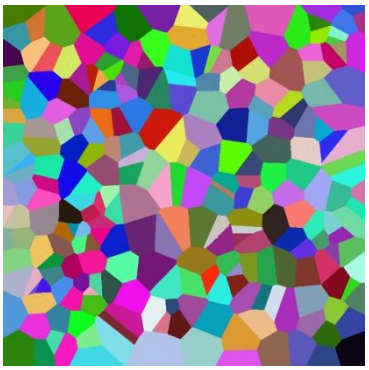
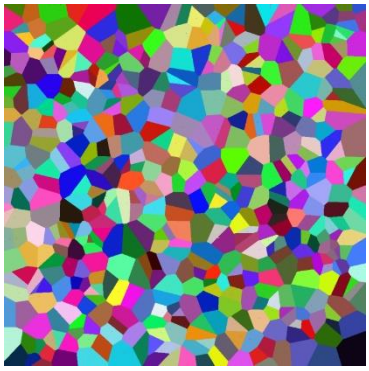
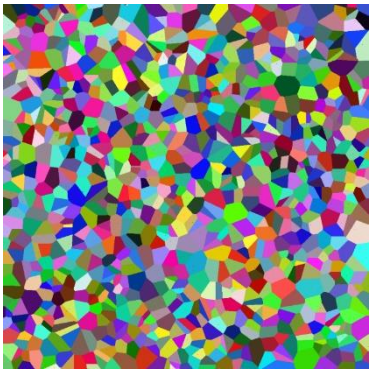
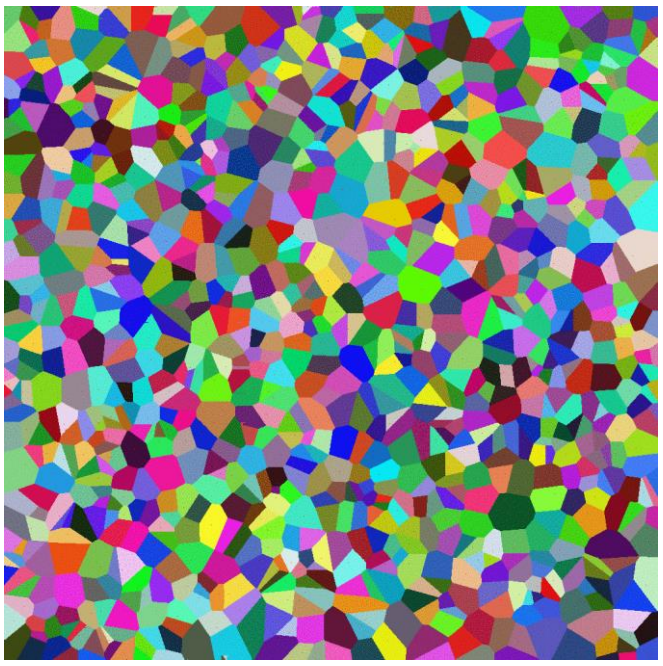
10. Free all CUDA devices

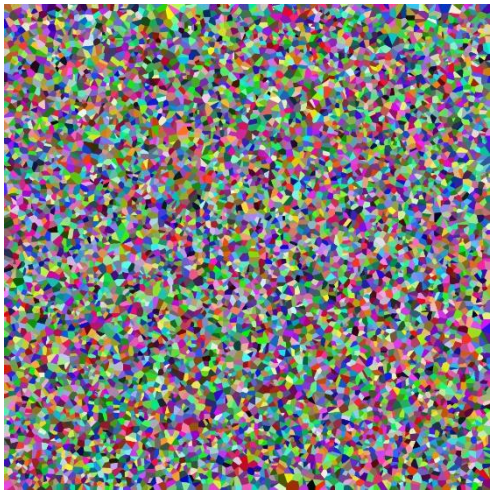
11. Assign the colour of each of the seeds to their respective grid points (i.e., grid points that are assigned to them) and write them as pixel into a .ppm image file.

**Results and Performance Analysis**

The images below show the Voronoi images for different seed sizes (*starting from 8, 16, 32, 64, 256, 512 and 1024*) for 2D grid size 1024 X 1024, that last one is a collection of the seeds with the same grid size (gif image):

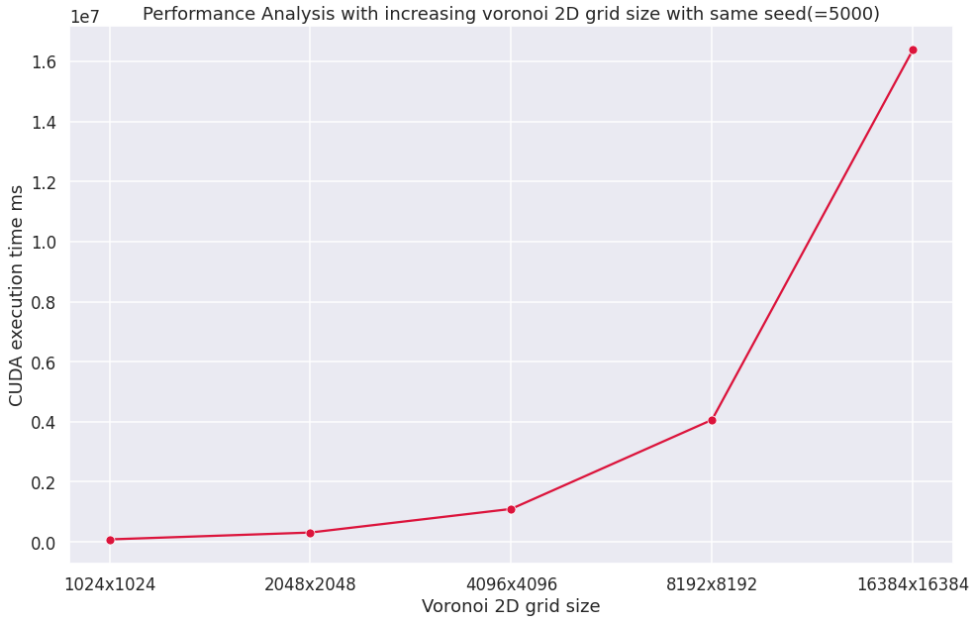


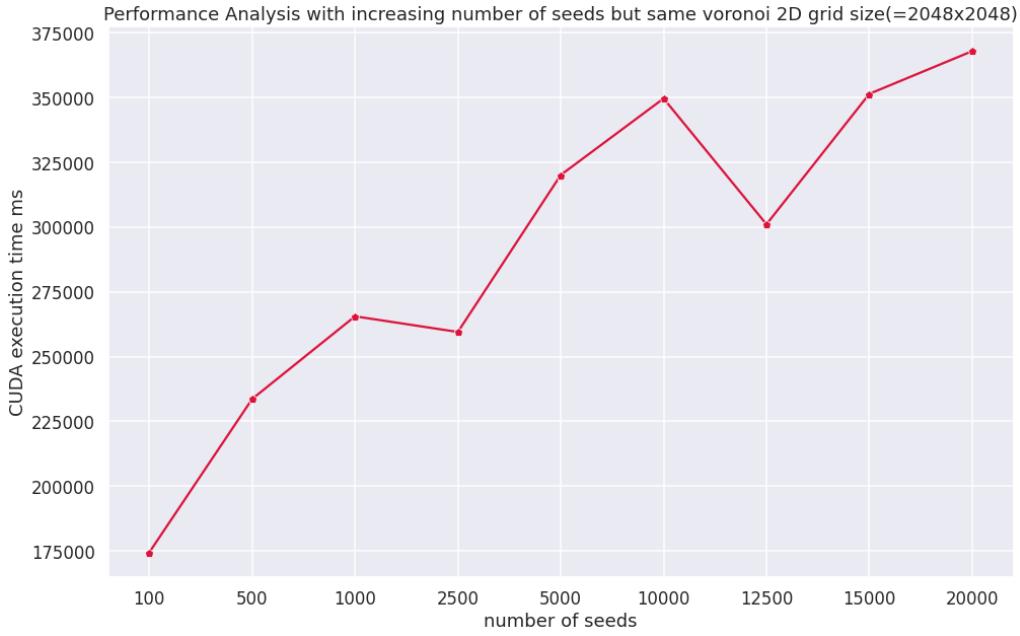


*This above image has 10000 seeds with Voronoi grid of 2048 x 2048.*

**Performance Analysis**




Based on the above figure, we observed that with increase in the Voronoi grid size for the same number of seeds, we seed steady linear increase in the CUDA execution time for smaller grid size i.e., below 4096x4096 size, but once we exceed that particular size, we see exponential increase in the CUDA execution time starting from 8192x8192 size. The linear increase in CUDA execution time with increasing grid size/image maybe because of the atomic operation that is used to update and assign closest seeds to corresponding grid points.




Based on the above figure, we observed that with increase in number of seeds for the same Voronoi grid size, we steady linear increase but also we observe some fluctuations in the increase, for example the CUDA execution time for 12,500 seeds is lower than the execution time for 5,000 seeds, this fluctuations in CUDA execution cannot be presently fully explained, but what we can generalize from these observations is that CUDA multithreading may be finding an optimized way of computing Voronoi images despite the atomic operation that is involved within the process.

Overall, the reason why we observed linear increase in the CUDA execution time for both increasing number of seeds for the same image and increasing image size, maybe because of the atomic operation involve in updating and assigning grid points to their closest seed.

**Summary**

To Conclude, we were able to compute Voronoi image with Jump Flood Algorithm which is a computational paradigm that involves each grid point in a discrete subspace passing on its content i.e., seed to at most eight neighbouring grid points at k step lengths for log n rounds.

We observe that there are linear increase in CUDA execution time when increasing seed and image sizes, this may be due to the atomic operation involve in the computational process.

**References**

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.8568&rep=rep1&type=pdf

https://www.comp.nus.edu.sg/~tants/jfa.html

Rong, G. and Tan, T., 2008. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. School of Computing, National University of Singapore

**CUDA APIs and Data Structures used for this project**

cudaGetDeviceCount() – get the number of CUDA devices in the Linux OS.

cudaMalloc() – allocate memory to device is CUDA based on the host replicate's size.

CudaDeviceProperty.warpSize – get the number of threads in a warp (32 threads)

CudaMemcpy – transfer data from device to host or host to device or device to device.

Kernel – Function to be executed on CUDA

atomicCAS() and Mutex – are used to perform the atomic operation, making sure only one thread is allowed at a time, to update the grid point's content i.e., assigning the grid point its closest seed without interference from any other thread.

cudaDeviceSynchronize() – forces the program to wait for all previously issued commands in all streams on the device (CUDA) to finish before continuing; In our case, the CPU is instead forced to be idle until all the GPU work has completed before doing anything else.

cudaFree() – free all the CUDA devices currently used.

****Voronoi 2D Image – Jump Flood Algorithm Code****

```
#include <cmath>

#include <cstdio>

#include <cstdlib>

#include <ctime>


#include <cuda_runtime.h>

#include <cuda_runtime_api.h>


#include <iostream>

#include <iterator>

#include <vector>

#include <chrono>
```

```
__global__ void Kernel( int SizeX , int SizeY , const float2 * SiteArray , const int * Ping , int * Pong , int k , int * Mutex )
{
  //
  const int CellX = threadIdx.x + blockIdx.x * blockDim.x ;
  const int CellY = threadIdx.y + blockIdx.y * blockDim.y ;

  const int CellIdx = CellX + CellY * SizeX ;
  const int Seed = Ping[CellIdx] ;
  if ( Seed < 0 )
  {
    return ;
  }

  //
  const int2 OffsetArray[8] = { { - 1 , - 1 } ,
                { 0 , - 1 } ,
                { 1 , - 1 } ,
                { - 1 , 0 } ,
                { 1 , 0 } ,
                { - 1 , 1 } ,
                { 0 , 1 } ,
                { 1 , 1 } } ;

  for ( int i = 0 ; i < 8 ; ++ i )
  {
    const int FillCellX = CellX + k * OffsetArray[i].x ;
    const int FillCellY = CellY + k * OffsetArray[i].y ;
    if ( FillCellX >= 0 && FillCellX < SizeX && FillCellY >= 0 && FillCellY < SizeY )
    {
```

```
//
const int FillCellIdx = FillCellX + FillCellY * SizeX ;

// Lock
//
while ( atomicCAS( Mutex , - 1 , FillCellIdx ) == FillCellIdx )
{
}

const int FillSeed = Pong[FillCellIdx] ;

if ( FillSeed < 0 )
{
   Pong[FillCellIdx] = Seed ;
}
else
{
   float2 P = make_float2( FillCellX + 0.5f , FillCellY + 0.5f ) ;

   float2 A = SiteArray[Seed] ;
   float2 PA = make_float2( A.x - P.x , A.y - P.y ) ;
   float PALength = PA.x * PA.x + PA.y * PA.y ;

   const float2 B = SiteArray[FillSeed] ;
   float2 PB = make_float2( B.x - P.x , B.y - P.y ) ;
   float PBLength = PB.x * PB.x + PB.y * PB.y ;

   if ( PALength < PBLength )
   {
      Pong[FillCellIdx] = Seed ;
   }
```

```
            }

            // Release
            //
            atomicExch( Mutex , - 1 ) ;
          }
      }
}

int main( int Argc , char * Argv[] )
{
   -- Argc , ++ Argv ;
   if ( Argc != 3 )
   {
      printf("SOMETHING IS WRONG") ;
      return EXIT_FAILURE ;
   }

   //numSeeds - Number of Seeds
   //Size - Voronoi grid size
   int numSeeds = atoi( Argv[0] ) ;
   int Size     = atoi( Argv[1] ) ;

   //
   int NumCudaDevice = 0 ;
   cudaGetDeviceCount( & NumCudaDevice ) ;
   if ( ! NumCudaDevice )
   {
      return EXIT_FAILURE ;
   }
```

```
//1. Generate x and y position values for the seeds in seedVec
//2. Randomly assign seed number to some grid points(x.y) in the voronoiVec
//3. Assign randomly generated colours to each of the seeds in randomcolourVec
std::vector< float2 > seedVec ;
std::vector< int >    voronoiVec( Size * Size , - 1 ) ;
std::vector< uchar3 > randomColorVec ;
for ( int i = 0 ; i < numSeeds ; ++ i )
{
   float X = static_cast< float >( rand() ) / RAND_MAX * Size ;
   float Y = static_cast< float >( rand() ) / RAND_MAX * Size ;
   int CellX = static_cast< int >( floorf( X ) ) ;
   int CellY = static_cast< int >( floorf( Y ) ) ;

   seedVec.push_back( make_float2( CellX + 0.5f , CellY + 0.5f ) ) ;
   voronoiVec[CellX + CellY * Size] = i ;
   //printf("SOMETHING IS GOOD");

   randomColorVec.push_back( make_uchar3( static_cast< unsigned char >( static_cast< float >( rand() ) / RAND_MAX * 255.0f ) ,
                           static_cast< unsigned char >( static_cast< float >( rand() ) / RAND_MAX * 255.0f ) ,
                           static_cast< unsigned char >( static_cast< float >( rand() ) / RAND_MAX * 255.0f ) ) ) ;
}

//
size_t seedSize = numSeeds * sizeof( float2 ) ;

float2 * seedArray = NULL ;
cudaMalloc( & seedArray , seedSize ) ;
cudaMemcpy( seedArray , & seedVec[0] , seedSize , cudaMemcpyHostToDevice ) ;
```

```
//BufferSize - Voronoi grid size (Size * Size)
size_t BufferSize = Size * Size * sizeof( int ) ;

int * Ping = NULL , * Pong = NULL ;
cudaMalloc( & Ping , BufferSize ) , cudaMemcpy( Ping , & voronoiVec[0] , BufferSize , cudaMemcpyHostToDevice ) ;
cudaMalloc( & Pong , BufferSize ) , cudaMemcpy( Pong , Ping , BufferSize , cudaMemcpyDeviceToDevice ) ;

//Mutex will be used in the kernel to lock and unlock atomic operation
int * Mutex = NULL ;
cudaMalloc( & Mutex , sizeof( int ) ) , cudaMemset( Mutex , - 1 , sizeof( int ) ) ;

//
//
cudaDeviceProp CudaDeviceProperty ;
cudaGetDeviceProperties( & CudaDeviceProperty , 0 ) ;

//warpsize = 32 threads
dim3 BlockDim( CudaDeviceProperty.warpSize , CudaDeviceProperty.warpSize ) ;
dim3 GridDim( ( Size + BlockDim.x - 1 ) / BlockDim.x ,
        ( Size + BlockDim.y - 1 ) / BlockDim.y ) ;

//run JFA for logn rounds
auto start = std::chrono::high_resolution_clock::now();
for ( int k = Size / 2 ; k > 0 ; k = k >> 1 )
{
   Kernel<<< GridDim , BlockDim >>>( Size , Size , seedArray , Ping , Pong , k , Mutex ) ;
   cudaDeviceSynchronize() ;

   cudaMemcpy( Ping , Pong , BufferSize , cudaMemcpyDeviceToDevice ) ;
   std::swap( Ping , Pong ) ;
```

```
}
auto stop = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop -start);
printf("Execution time %ld microseconds\n", duration.count());

cudaMemcpy( & voronoiVec[0] , Pong , BufferSize , cudaMemcpyDeviceToHost ) ;

//
cudaFree( seedArray ) ;
cudaFree( Ping ) ;
cudaFree( Pong ) ;
cudaFree( Mutex ) ;

//
//
FILE * Output = fopen( Argv[2], "wb" ) ;
fprintf( Output , "P6\n%d %d\n255\n" , Size , Size ) ;

std::vector< uchar3 > Pixels( Size * Size ) ;
for ( int y = 0 ; y < Size ; ++ y )
{
   for ( int x = 0 ; x < Size ; ++ x )
   {
      const int Seed = voronoiVec[x + y * Size] ;
      if ( Seed != - 1 )
      {
         Pixels[x + y * Size] = randomColorVec[Seed] ;
      }
   }
}
```

```
    for( std::vector< float2 >::const_iterator itr = seedVec.begin() ; itr != seedVec.end() ; ++ itr )
    {
        const int x = static_cast< int >( floorf( itr->x ) ) ;
        const int y = static_cast< int >( floorf( itr->y ) ) ;
        Pixels[x + y * Size] = make_uchar3( 255 , 0 , 0 ) ;
    }

    fwrite( & Pixels[0].x , 3 , Pixels.size() , Output ) ;
    fclose( Output ) ;

    return EXIT_SUCCESS ;
}
```