Jarred Parr -- October 31, 2018

Point Source Pollution

Introduction

Point source pollution is a very common occurrence in our modern world. Whether it is just a simple spill, or a deadly break in an oil line, humans have long used modeling to predict the disaster surface of such events. This project is another example of such modeling. Utilizing a partial differential equation, models for a 1-dimensional river and 2-dimensional lake-like area have been able to be successfully simulated. This project served as an introduction to cuda-accelerated modeling of systems like this. Given the nature of the equation, it would prove to be very taxing on a normal CPU to attempt this degree of calculation for iterations sometimes in the billions. As a result, the problem specification mandated the use of a GPU and CUDA code to help mitigate these performance bottlenecks. With unlimited computing power this situation becomes trivial like in situations for researchers in national labs, however, given that resources of this scope were not available, a great deal of exploration was taken into proper optimization techniques, and further inquiry into getting every last ounce of performance from the CUDA code on limited hardware.

Motivation

Outside of the project specification, this project was particularly interesting to myself as a researcher because of my steadfast desire to learn more about simulations of phenomena that can have a direct impact on the environment. A great deal of my interest lies in the understanding of how such interactions exist naturally, and the ability to solve the problems that they present. The implications of very fine-tuned environmental modeling can speak volumes about ways to help further preserve the environment as a whole.

Program Architecture

My sequential code was about as simple as it gets. The code was able to be implement central difference theorem without much hassle, and this allowed more time to be devoted to building (and debugging) the parallel version. At first, it was an extreme challenge to get everything working. CUDA is very particular about its memory and any misstep caused a slew of hard-to-track-down errors. This enabled learning more of the "do's" and "dont's" of CUDA development. In terms of program operation, it was found that for certain numbers of cycles, there was a minimum number needed to see any reasonable results. For example, when run with a cylinder size of 1000 - 10000, a diffusion time that was even within the same order of magnitude of the cylinder size presented results that ended up producing values that were ~0. As a result, I opted to run each simulation a few times with varying numbers of iterations to see what had the nicest spread and the cleanest visualization surface. This gave a range of outputs and several reproducible ones as well.

Results

Unfortunately, I ran out of time when it came to parallelizing my 2d diffusion code, but I was able to still get extremely interesting findings for my final results which proved to be very enlightening and aligned well with my hypothesized outputs.It took quite a few runs to acquire a reliable rhythm for the tests, but once a pattern was found it was trivial to proceed. In terms of the CUDA code, block sizes of all kinds were used,

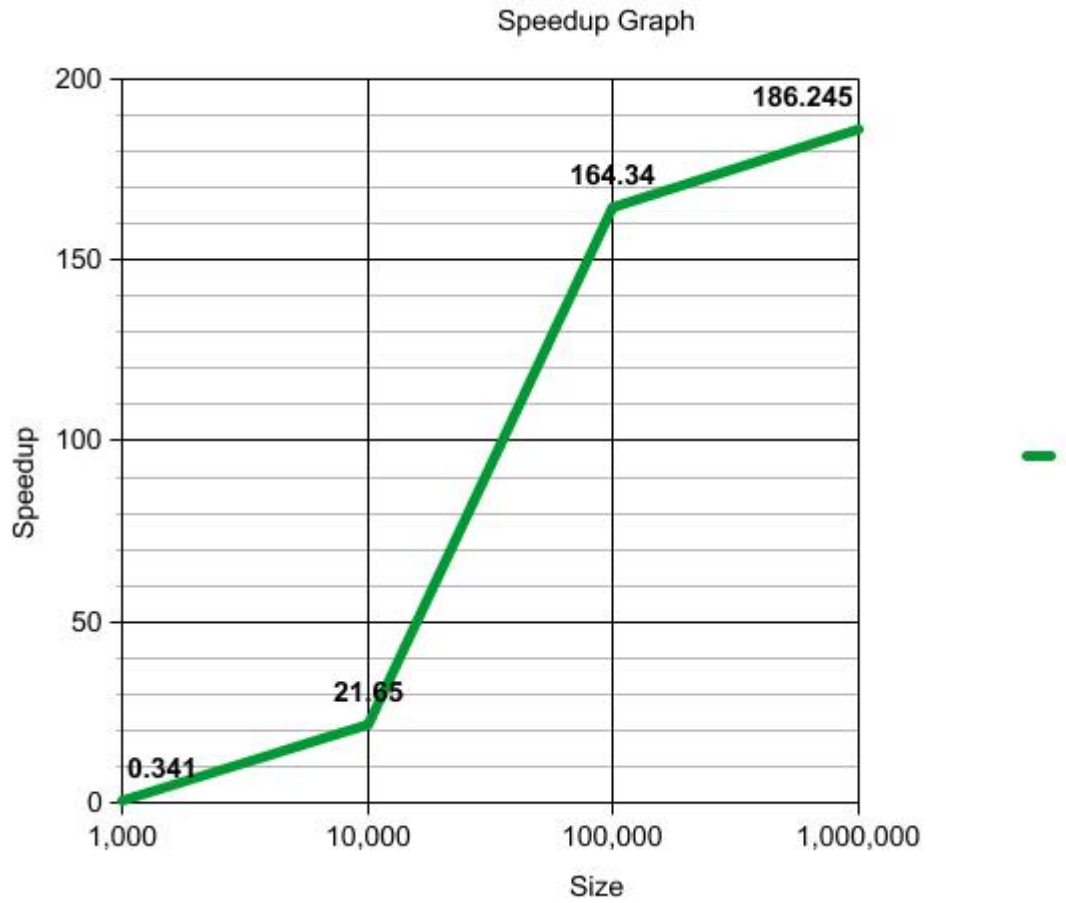
however, I consistently found that the best performance was for a block size of 1024.This eventually took a toll on my server's GPU so I took breaks to let the system cool and keep runs consistent.When tested on Seawolf I saw very marginal upgrades in terms of speed and as a result all times and data below is from my home server for the sake of ease of use as Seawolf saw a lot of traffic over the main time periods in which I was testing code. The following tests were run:

Test Results

	Size	Slice	Iterations	Concentration	Runtime	Sequential Runtime	Speedup
Run 0	1,000	700	100,000	200	0.88 s	0.3 s	.341
Run 1	10,000	7,000	1,000,000	200	3.15 s	68.2 s	21.65
Run 3	100,000	70,000	100,000,000	200	29.67 s	4876.04 s	164.34
Run 4	1,000,000	700,000	1,000,000,000	200	91.784 s	17,094.32 s	186.245

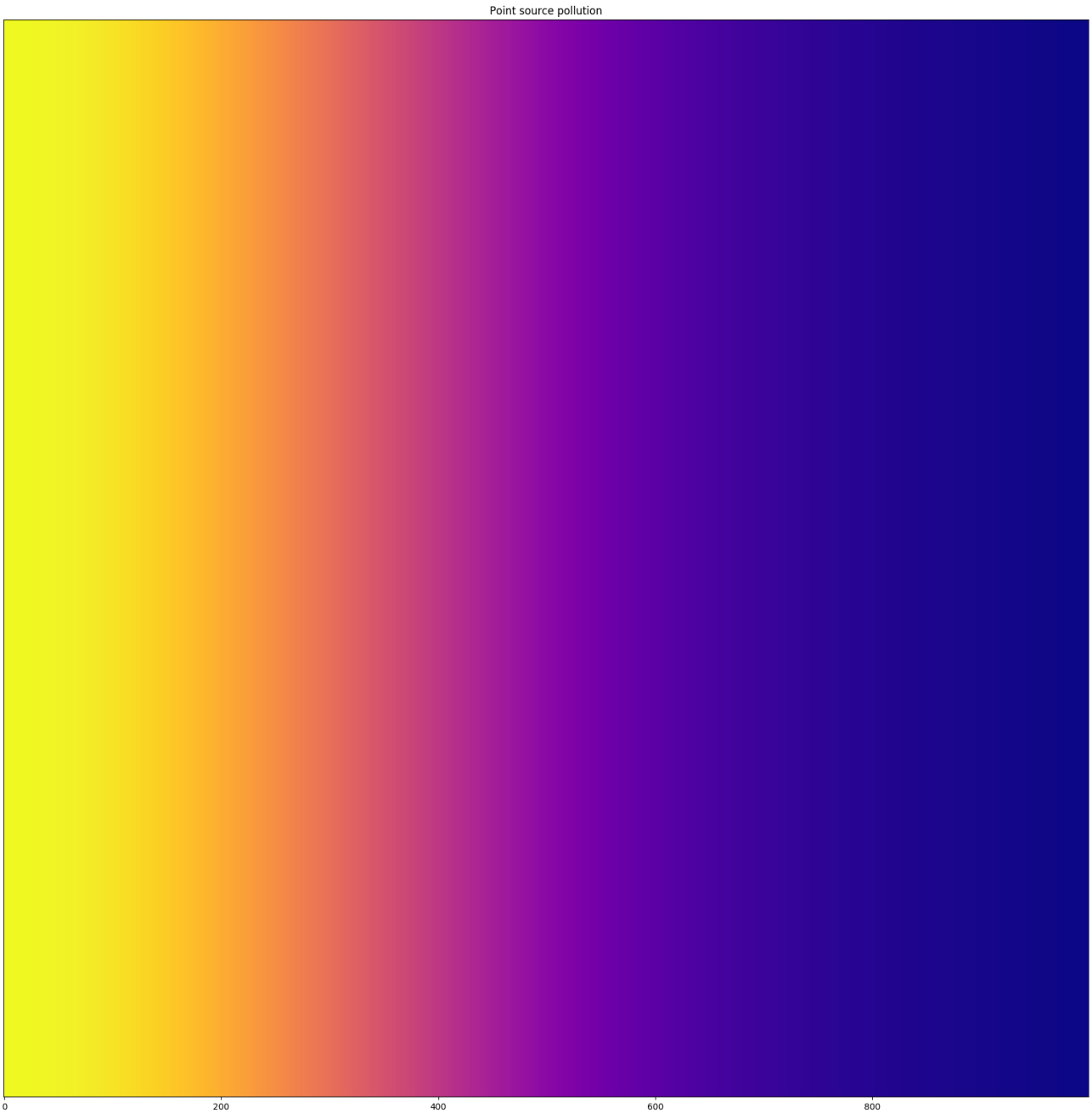
Speedup Graph

This speedup was measured with the size of the cylinder as the x-axis just to further emphasize the affect that the GPU can have on the end result.

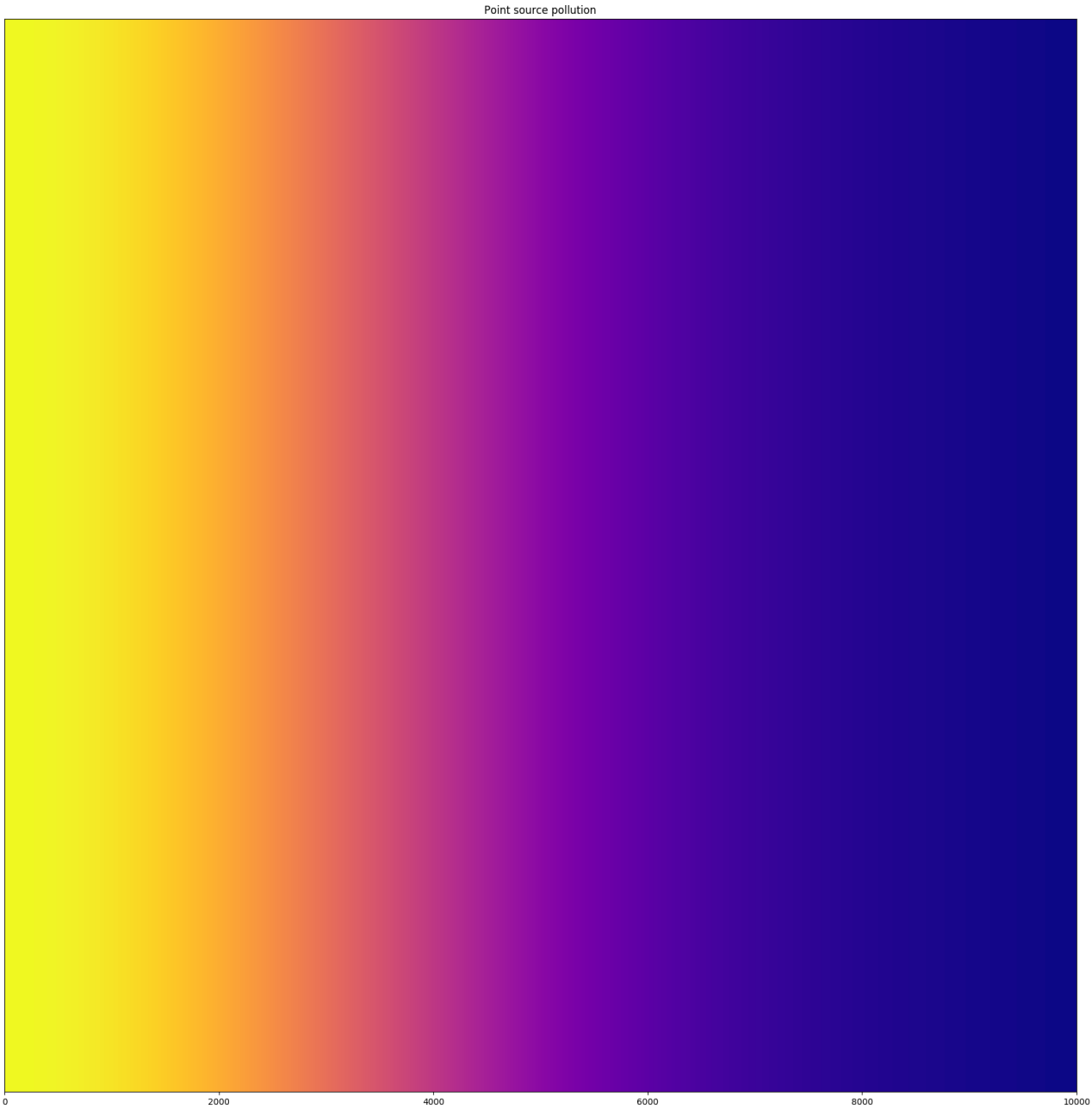


As can be seen by the table and accompanying graph, the speedup as the size increased went up dramatically as the overhead for the GPU implementation was able to be overcome by increasing the overall size of the cylinder. The simulation also produced some beautiful results which we able to be easily shown via matplotlib's heatmap functionality. They are as follows:

The 1000x1000 Graph



The 10000x10000 Graph



Even though they largely looked the same it was very cool to see the output so vividly for a given number if iterations. These were expanded since a cylinder shape typically has a pretty small height.

Conclusion

Overall this was a very enjoyable and thought provoking problem that shows the real world solutions that high performance computing can solve. I see that my speedups were a little low considering the scope of the project. Given more time I would have tried to fine tune my code as things like device synchronization and other small features for memory management and access seem to have severely hurt my overall runtime. I intent to extend this project further for greater insight into how to fix some of these problems as CUDA programming is a great interest of mine.

Source Code

Sequential

```
#include <algorithm>
#include <cstdint>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <random>
#include <tuple>
#include <vector>

class PointSourcePollution {
  public:
    explicit PointSourcePollution();
    ~PointSourcePollution() = default;
    std::vector<double> diffuse(
        uint64_t cylinder_size,
        uint64_t diffusion_time,
        uint64_t contaminant_concentration);
    std::vector<std::vector<double>> diffuse2d(
        uint64_t pool_rows,
        uint64_t pool_cols,
        uint64_t diffusion_time,
        uint64_t contaminant_concentration,
        uint64_t leaks,
        bool multiple);
    void end(const std::vector<double> & data);
    void end2d(const std::vector<std::vector<double>>& data);
  private:
    double central_difference_theorem(double left, double right) const;
    double central_difference_theorem2d(double left, double right, double up, double
down) const;
};

PointSourcePollution::PointSourcePollution() {}

void PointSourcePollution::end(const std::vector<double>& data) {
  std::ofstream payload;
  payload.open("output.txt");

  for (uint64_t i = 0; i < data.size(); ++i) {
    if (i != 0) {
      payload << " ";
    }
    payload << data[i];
  }

  payload.close();
}

void PointSourcePollution::end2d(const std::vector<std::vector<double>>& data) {
  std::ofstream payload;
  payload.open("output.txt");
```

```
  for (uint64_t i = 0; i < data.size(); ++i) {
    for (uint64_t j = 0; j < data[i].size(); ++j) {
      if (j != 0) {
        payload << " ";
      }

      payload << data[i][j];
    }
    payload << "\n";
  }
  payload.close();
}

double PointSourcePollution::central_difference_theorem2d(double left, double right,
double up, double down) const {
  return (left + right + up + down) / 4;
}

std::vector<std::vector<double>> PointSourcePollution::diffuse2d(
    uint64_t pool_rows,
    uint64_t pool_cols,
    uint64_t diffusion_time,
    uint64_t contaminant_concentration,
    uint64_t leaks,
    bool multiple
    ) {
  std::random_device rd;
  std::mt19937 g(rd());
  std::vector<std::tuple<uint64_t, uint64_t>> leak_locations;

  for (uint64_t i = 0; i < leaks; ++i) {
    std::uniform_int_distribution<uint64_t> leak_row(0, pool_rows);
    std::uniform_int_distribution<uint64_t> leak_col(0, pool_cols);
    std::tuple<uint64_t, uint64_t> val = std::make_tuple(leak_row(g), leak_col(g));
    leak_locations.push_back(val);
  }

  std::vector<std::vector<double>> pool(pool_rows, std::vector<double>(pool_cols, 0));
  std::vector<std::vector<double>> copy_pool(pool_rows, std::vector<double>(pool_cols,
0));
  std::vector<std::vector<double>> temp(pool_rows, std::vector<double>(pool_cols, 0));

  if (multiple) {
    for (uint64_t i = 0; i < leak_locations.size(); ++i) {
      auto row = std::get<0>(leak_locations[i]);
      auto col = std::get<1>(leak_locations[i]);

      pool[row][col] = contaminant_concentration;
      copy_pool[row][col] = contaminant_concentration;
    }
  } else {
    pool[0][0] = contaminant_concentration;
```

```
    copy_pool[0][0] = contaminant_concentration;
  }

  double left, right, up, down;

  for (uint64_t k = 0; k < diffusion_time; ++k) {
    for (uint64_t i = 0; i < pool_rows; ++i) {
      for (uint64_t j = 0; j < pool_cols; ++j) {
        left = j - 1 >= 0
          ? pool[i][j - 1]
          : pool[i][j];
        right = j + 1 <= pool_cols
          ? pool[i][j + 1]
          : pool[i][j];
        up = i - 1 >= 0
          ? pool[i - 1][j]
          : pool[i][j];
        down = i + 1 <= pool_rows
          ? pool[i + 1][j]
          : pool[i][j];
        copy_pool[i][j] = central_difference_theorem2d(left, right, up, down);
      }
    }
    temp = pool;
    pool = copy_pool;
    copy_pool = temp;
  }

  return pool;
}

std::vector<double> PointSourcePollution::diffuse(
    uint64_t cylinder_size,
    uint64_t diffusion_time,
    uint64_t contaminant_concentration
    ) {
  std::vector<double> cylinder(cylinder_size, 0);
  std::vector<double> copy_cylinder(cylinder_size, 0);
  double left, right;

  cylinder[0] = contaminant_concentration;
  copy_cylinder[0] = contaminant_concentration;

  for (uint64_t i = 1; i < diffusion_time; ++i) {
    for (uint64_t j = 1; j < cylinder_size; ++j) {
      left = cylinder[j - 1];
      right = cylinder[j + 1];

      copy_cylinder[j] = this->central_difference_theorem(left, right);
    }
    std::vector<double> temp(cylinder);
    cylinder = copy_cylinder;
    copy_cylinder = temp;
```

```
  }

  return cylinder;
}

double PointSourcePollution::central_difference_theorem(double left, double right)
const {
  return (left + right) / 2.0;
}

int main(int argc, char** argv) {
  uint64_t pool_rows, pool_cols, leaks, cylinder_size, slice_location, diffusion_time,
contaminant_concentration;
  bool multiple = false;

  if (argc < 5) {
    std::cerr << "usage: psp cylinder_size slice_location diffusion_time
contaminant_concentration" << std::endl;
    std::cerr << "usage: pool_rows pool_cols leaks contaminant_concentration
diffusion_time" << std::endl;
    return EXIT_FAILURE;
  }

  if (argc > 5) {
    pool_rows = atoi(argv[1]);
    pool_cols = atoi(argv[2]);
    leaks = atoi(argv[3]);
    if (leaks > 1) {
      multiple = true;
    }
    contaminant_concentration = atoi(argv[4]);
    diffusion_time = atoi(argv[5]);
  } else {
    cylinder_size = atoi(argv[1]);
    slice_location = atoi(argv[2]);
    diffusion_time = atoi(argv[3]);
    contaminant_concentration = atoi(argv[4]);
  }

  for (int i = 0; i < argc; ++i) {
    if (atoi(argv[i]) < 0) {
      std::cerr << "All inputs must be greater than 0" << std::endl;
      return EXIT_FAILURE;
    }
  }

  PointSourcePollution psp;
  if (argc == 5) {
    std::cout << "starting 1d diffusion..." << std::endl;
    std::vector<double> output = psp.diffuse(cylinder_size, diffusion_time,
contaminant_concentration);
    std::cout << "Answer at slice location: " << slice_location << " is " <<
output[slice_location] << std::endl;
```

```
    std::cout << "Now visualizing results..." << std::endl;
    psp.end(output);
    system("python plot.py output.txt");
  } else {
    std::cout << "starting 2d diffusion" << std::endl;
    std::vector<std::vector<double>> out(pool_rows, std::vector<double>(pool_cols));
    out = psp.diffuse2d(pool_rows, pool_cols, diffusion_time,
contaminant_concentration, leaks, multiple);
    std::cout << "Now visualizing results..." << std::endl;
    psp.end2d(out);
    system("python plot.py output.txt");
  }

  return EXIT_SUCCESS;
}
```

Parallel

```
#include <cstdint>
#include <cmath>
#include <fstream>
#include <iostream>
#include <vector>

const int BLOCK_SIZE = 1024;

class PointSourcePollution {
  public:
    PointSourcePollution() = default;
    ~PointSourcePollution() = default;
    void end(const double* data, uint64_t cylinder_size);
};

void PointSourcePollution::end(const double* data, uint64_t cylinder_size) {
  std::ofstream payload;
  payload.open("output.txt");

  for (uint64_t i = 0; i < cylinder_size; ++i) {
    if (i != 0) {
      payload << " ";
    }

    payload << data[i];
  }

  payload.close();
}

__device__
void central_difference_theorem(
    double left,
```

```
    double right,
    double& out
    ) {
  out = (left + right) / 2.0;
}

__global__
void diffuse(
    double* cylinder,
    double* copy_cylinder,
    double* temp,
    uint64_t cylinder_size,
    uint64_t diffusion_time,
    uint64_t contaminant_concentration
    ) {
  double left, right, cdt_out;
  int i = blockIdx.x * BLOCK_SIZE + threadIdx.x;

  if (i < cylinder_size) {
    if (i > 0)
      left = cylinder[i - 1];
    else
      left = cylinder[i];
    right = cylinder[i + 1];

    central_difference_theorem(left, right, cdt_out);
    cylinder[i] = cdt_out;
    temp = cylinder;
    cylinder = copy_cylinder;
    copy_cylinder = temp;
  }
}


int main(int argc, char** argv) {
  uint64_t cylinder_size, slice_location, diffusion_time, contaminant_concentration;

  if (argc < 5) {
    std::cerr << "usage: psp cylinder_size slice_location diffusion_time
contaminant_concentration" << std::endl;
    return EXIT_FAILURE;
  }

  for (int i = 0; i < argc; ++i) {
    if (atoi(argv[i]) < 0) {
      std::cerr << "All inputs must be greater than 0" << std::endl;
      return EXIT_FAILURE;
    }
  }

  PointSourcePollution psp;
  cylinder_size = atoi(argv[1]);
  slice_location = atoi(argv[2]);
```

```
  diffusion_time = atoi(argv[3]);
  contaminant_concentration = atoi(argv[4]);
  cudaError_t e;
  double *cylinder, *copy_cylinder, *temp;

  cudaMallocManaged(&cylinder, cylinder_size * sizeof(double));
  cudaMallocManaged(&copy_cylinder, cylinder_size * sizeof(double));
  cudaMallocManaged(&temp, cylinder_size * sizeof(double));

  // init our arrays
  for (int i = 0; i < cylinder_size; ++i) {
    if (i == 0) {
      cylinder[i] = contaminant_concentration;
      copy_cylinder[i] = contaminant_concentration;
    } else {
      cylinder[i] = 0.0;
      copy_cylinder[i] = 0.0;
    }
  }
  std::cout << cylinder[0] << copy_cylinder[0] << std::endl;

  e = cudaGetLastError();
  if (e != cudaSuccess) {
    std::cerr << "Error: " << cudaGetErrorString(e) << std::endl;
    return EXIT_FAILURE;
  }

  const uint64_t GRID_SIZE = ceil(cylinder_size / static_cast<double>(BLOCK_SIZE));
  for (int i = 0; i < diffusion_time; ++i) {
    diffuse<<<GRID_SIZE, BLOCK_SIZE>>>(
        cylinder,
        copy_cylinder,
        temp,
        cylinder_size,
        diffusion_time,
        contaminant_concentration);
  }
  cudaDeviceSynchronize();

  e = cudaGetLastError();
  if (e != cudaSuccess) {
    std::cerr << "Error2: " << cudaGetErrorString(e) << std::endl;
    return EXIT_FAILURE;
  }

  std::cout << "Answer at slice location: " << slice_location << " is " <<
cylinder[slice_location] << std::endl;
  std::cout << "Now visualizing results..." << std::endl;
  psp.end(cylinder, cylinder_size);

  cudaFree(cylinder);
  cudaFree(copy_cylinder);
```

```
  e = cudaGetLastError();
  if (e != cudaSuccess) {
    std::cerr << cudaGetErrorString(e) << std::endl;
    return EXIT_FAILURE;
  }


  system("python plot.py");
  return EXIT_SUCCESS;
}
```