МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет «Информационные технологии и прикладная математика»
Кафедра «Вычислительная математика и программирование»

Курсовой проект
по курсу «Программирование графических процессоров»

Моделирование поведения роя частиц на GPU

Выполнил: И.И. Тишин
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2019

Условие

Цель работы:

Использование GPU для моделирования поведения косяка рыб или стаи птиц. Решение задачи глобальной оптимизации. Взаимодействие технологий CUDA и OpenGL. Решение проблемы коллизий многих объектов. Построение регулярного разбиения пространства.

Задание:

Реализовать алгоритм роя частиц для задачи глобальной оптимизации Inertia Weighted SPO[1]. Коэффициент инерции w и параметры a1, a2 задаются пользователем. Для решения проблемы коллизий, необходимо ввести силу отталкивания обратно пропорциональную расстоянию в некоторой степени(например в четвертой) между частицами.

Должна присутствовать 2D визуализация работы алгоритма. Частицы (рыбы/птицы) можно отобразить кружочками. На экран необходимо вывести температурную карту рассматриваемой функции(задается вариантом), по которой будут перемещаться частицы. Камера должна следовать за роем частиц(например, за центром масс), т.е. если частицы переместились за область видимости экрана, то камера перемещается в след за ними. У пользователя должнабыть возможность интерактивно изменять масштаб.

Вариант №6:

6. Функция Швефеля[3].

$$f(\mathbf{X})=\sum_{i=1}^{n}\left[-x_i \sin\left(\sqrt{|x_i|}\right)\right]$$

Программное и аппаратное обеспечение

Major/Minor version number: 6.1

Total amount of global memory: 6078 MBytes (6373572608 bytes)

(10) Multiprocessors, (128) CUDA Cores/MP: 1280 CUDA Cores

GPU Max Clock rate: 1480 MHz (1.48 GHz)

Memory Clock rate: 4004 Mhz

Memory Bus Width: 192-bit

L2 Cache Size: 1572864 bytes

Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)

Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers

Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Total number of registers available per block: 65536

Warp size: 32

Maximum number of threads per multiprocessor: 2048

Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)

Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

Maximum memory pitch: 2147483647 bytes

Texture alignment: 512 bytes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1,

NumDevs = 1

CPU:

Архитектура: x86_64

CPU op-mode(s): 32-bit, 64-bit

CPU(s): 12

Потоков на ядро: 2

Ядер на сокет: 6

Имя модели: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

CPU max MHz: 4100,0000

CPU min MHz: 800,0000

BogoMIPS: 4416.00

Виртуализация: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 9216K

Ram:

32 GiB, 2666 MHz

HDD:

500 GB

Программное обеспечение:

Ubuntu 18.04.2 LTS

g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0

nvcc: release 10.1, V10.1.105

Метод решения

Сгенерировать начальное положение частиц.

В цикле выполняем следующие действия: рассчитать карту в текущей отображаемой области, пересчитать положение частиц, обновить локальные минимумы частиц и глобальный минимум, отобразить частицы.

Карта отображается в соответствии со значением функции в данной точке пространства. В рассматриваемом решении цвет задается так:

```
__device__ uchar4 get_color(float f) {
  float k = 1.0 / 5.0;
  if (f <  k)
        return make_uchar4(255, (int)((f - k) * 255 / k), 0, 0);
  if (f < 2 * k)
        return make_uchar4(255, 255, (int)((f - 2 * k) * 255 / k), 0);
  if (f < 3 * k)
        return make_uchar4(255 - (int)((f - 3 * k) * 255 / k), 255, 255, 0);
  if (f < 4 * k)
        return make_uchar4(0, 255 - (int)((f - 4 * k) * 255 / k), 255, 0);
  if (f < 5 * k)
        return make_uchar4(0, 0, 255 - (int)((f - 5 * k) * 255 / k), 0);
  return make_uchar4(0, 0, 0, 0);
}
f = (fun(i, j) - minf) / (maxf - minf);
get_color(f);
```

Обновление происходит следующими действиями:

- Генерируются случайные векторы r_1 и r_2 из равномерного распределения с параметрами (0, 1).
- Обновляется скорость частицы:
  $v_{ix} = wv_{ix} + [a_1r_1\times(min_{local} – x_i) + a_2r_2\times(min_{global} – x_i) + F_{ix}] \times dt$. Здесь v_{ix} – проекция скорости i компоненты на ось Ox; w – коэффициент, характеризующий инерционные свойства частиц; a_1 и a_2 – определяют значимость для частицы своего лучшего положения и лучшего среди всего роя положения; min_{local} – лучшее положение i частицы; min_{global} – лучшее положение среди всего роя.
- Обновляется положение частицы:
  $x_i = x_i + v_{ix} \times dt$
  $y_i = y_i + v_{iy} \times dt$

Для решения проблемы коллизий частиц вводится сила взаимодействия между частицами. Заряды принимаются единичными, расстояние учитывается в 4 степени. После проецирования силы на оси, например ось Ox, получим $F_i = G \times \sum (x_i - x_j) / (dist(x_i, x_j) \wedge 3+0.0001)$. Здесь G – коэффициент пропорциональности (принимаем за 0,01).

Для отображения точек необходимо вычислить, попадают ли они в текущую область. Для тех, что попадают, вычисляется позиция в буфере по формуле (для оси Ox) ((x - xc + sx) / (2 * sx / (w - 1))). Здесь xc – смещение окна от начальной позиции, sx – масштаб карты, а w – размер экрана по горизонтали. Точки отображаются зеленым цветом.

Описание программы

Сначала инициализируется openGL, объявляются функции для отрисовки и отслеживающие нажатие клавиш. Создается матрица проекций размером w на h, где w и h размеры окна. Далее создается буфер и привязываем к нему данные об отображаемой текстуре. Также создается буфер для взаимодействия с cuda, который отображает данные openGL буфера.

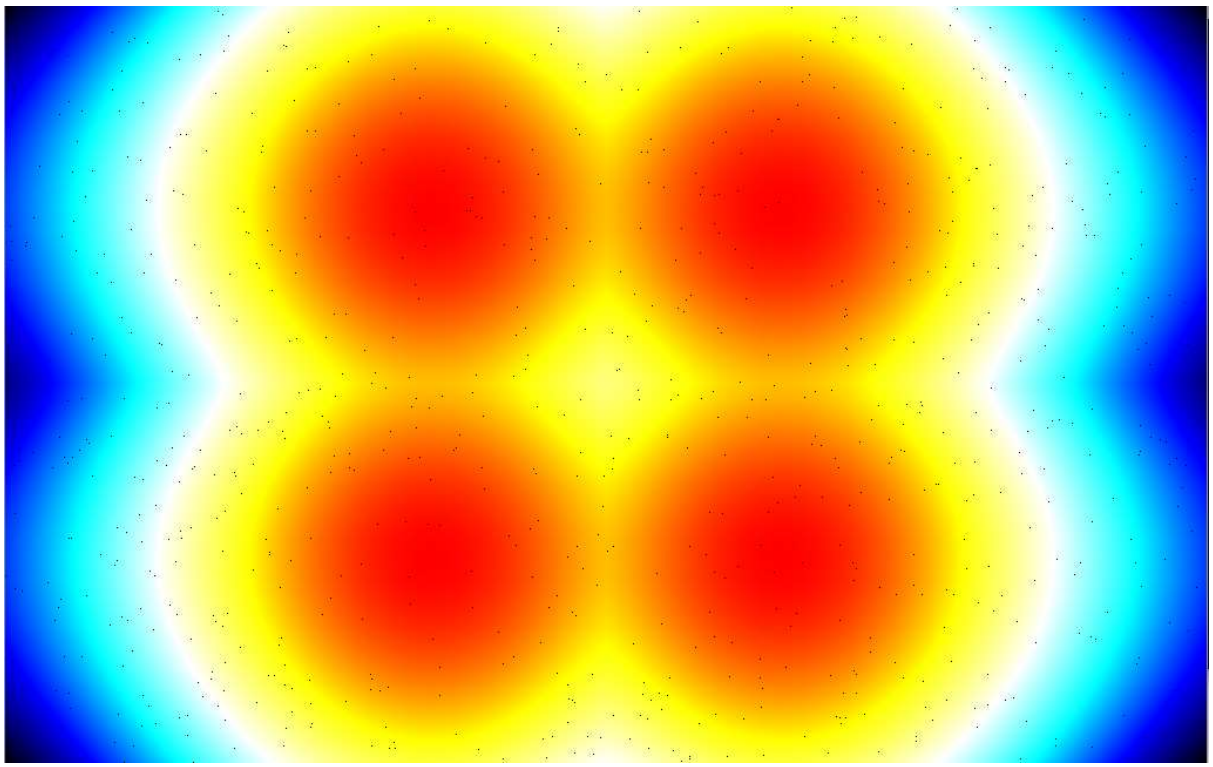
После инициализаций необходимо сгенерировать начальное положение частиц. Для этого в данной работе применяется модуль <curand.h>. Также у каждой точки выставляется нулевая начальная скорость и лучшее положение в исходных координатах.

В функции, отвечающей за обновление данных на экране, получаем указатель на буфер с отображенными из openGL данными о текстуре. Далее вычисляем центр масс частиц и если он изменился со времени предыдущего вызова функции больше чем на ε, сдвигаем экран в новый центр и перерисовываем карту. После этого обновляем координаты частиц, обновляем минимумы и отрисовываем частицы. При инициализации openGL было указано использование двойного буфера, поэтому после всех изменений нужно свапнуть экранные буферы.

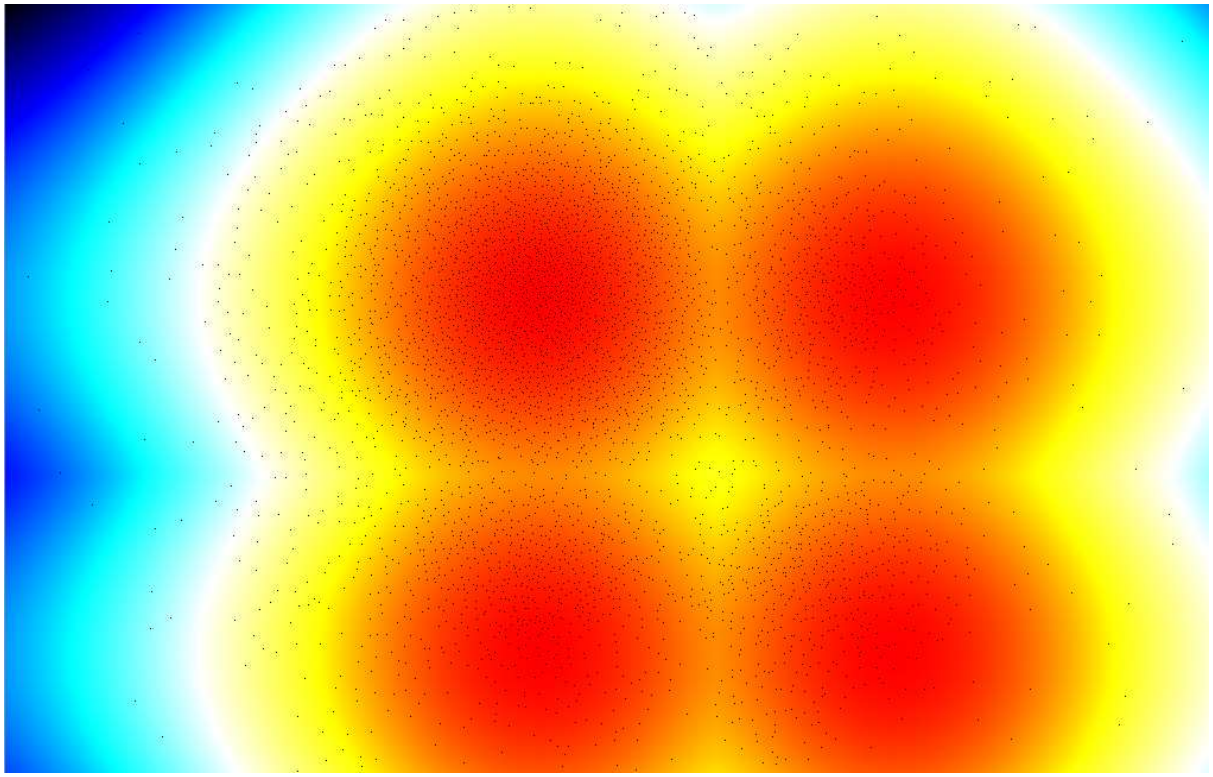
Исследовательская часть

При преобладании a_1 над a_2 частицы склонны выбирать локальные минимумы, а при обратной ситуации – глобальные. При увеличении w взаимодействие частиц ускоряется, вследствие чего они быстрее находят оптимальную позицию. В данной работе поведение частиц близко к поведению стаи при $a_1 = 0.8$ $a_2 = 0.2$ $w = 0.99$.

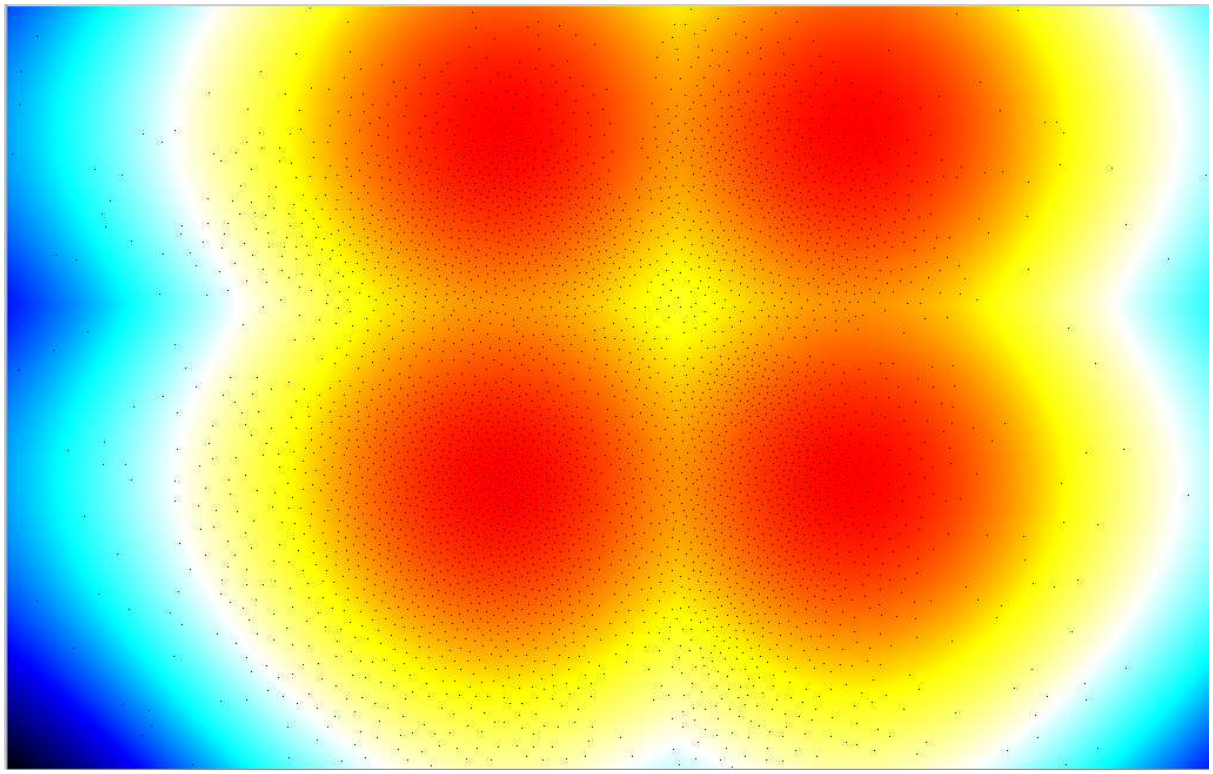
Начальное расположение частиц



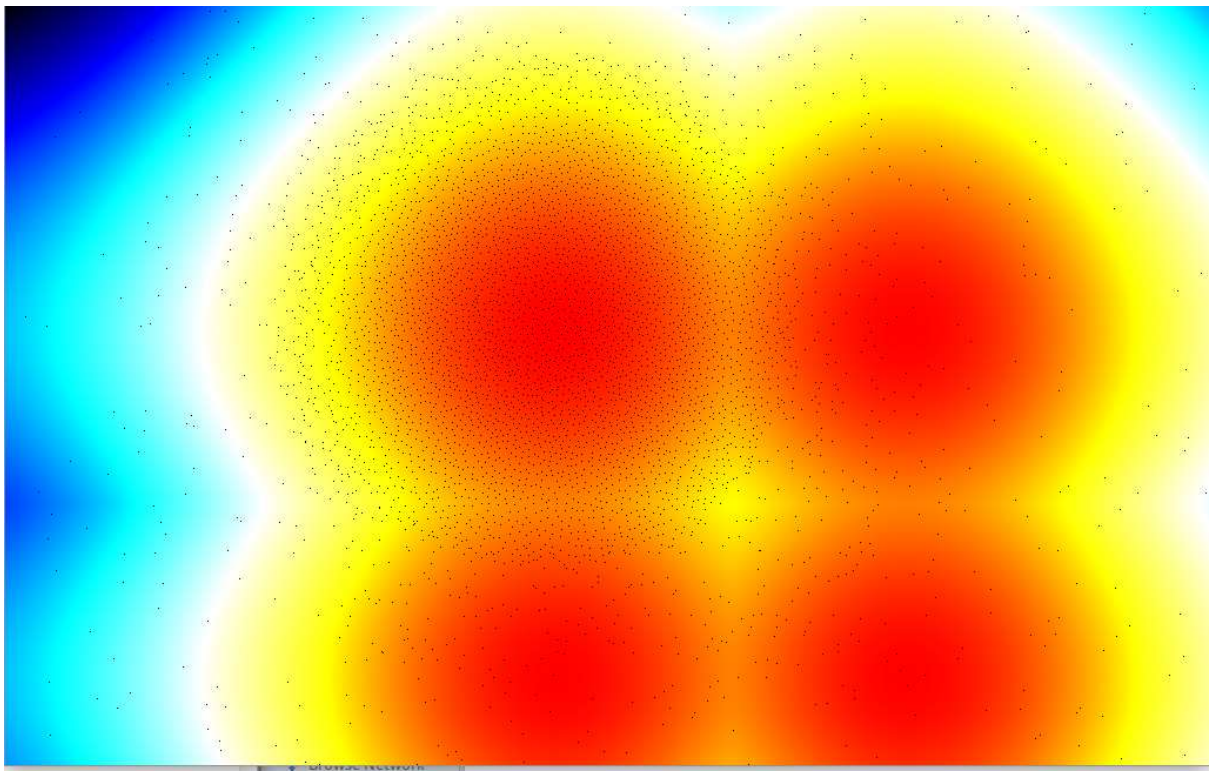
Частицы слетаются к минимуму

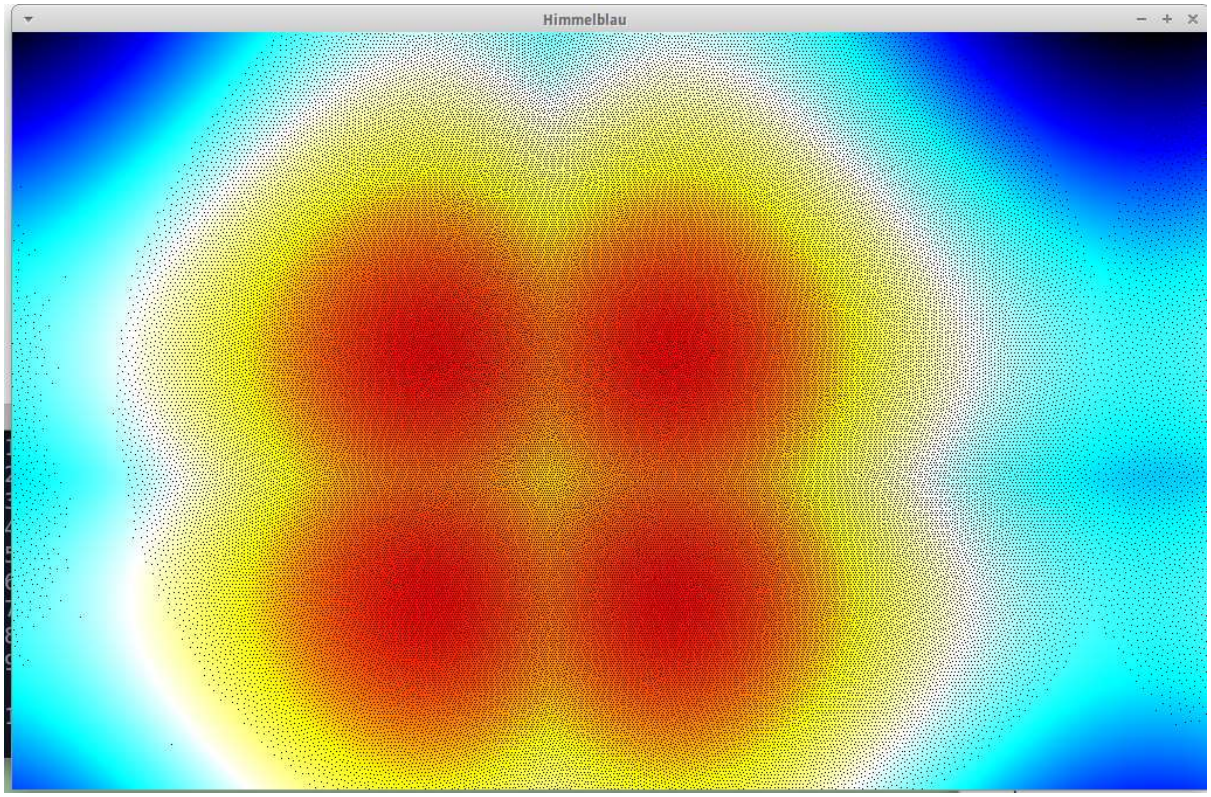


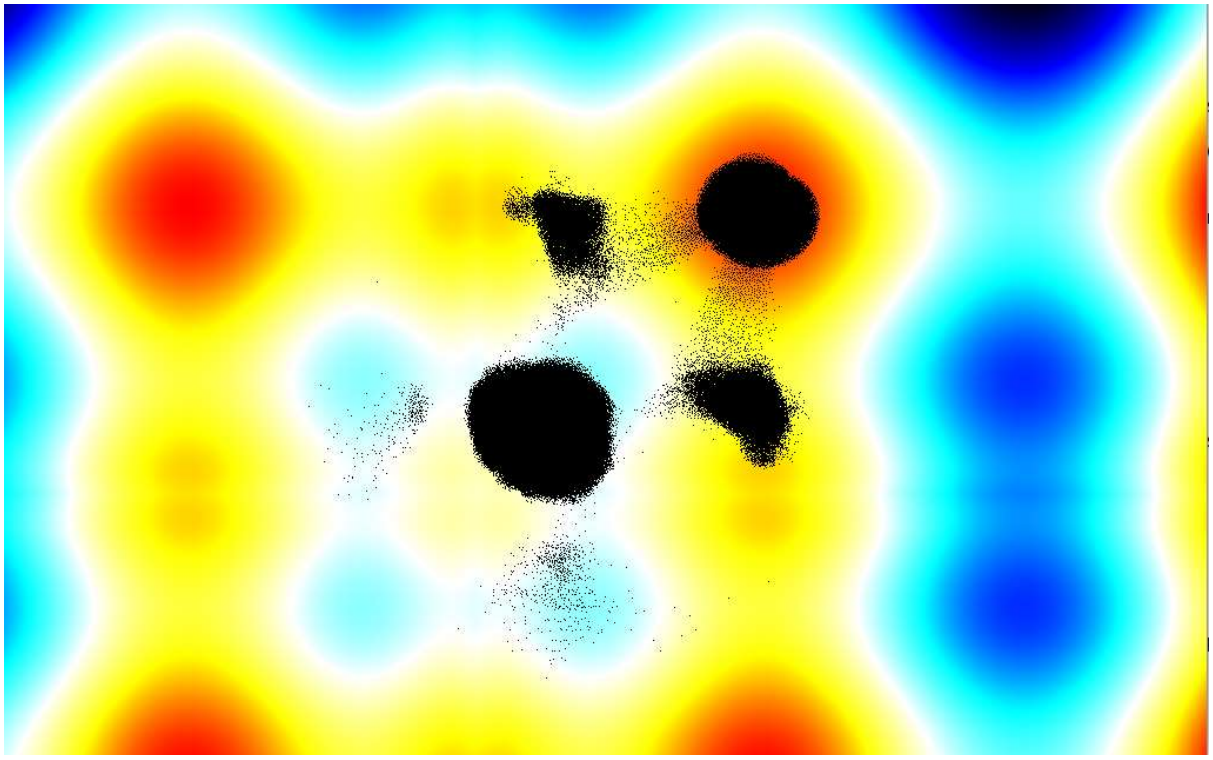
Преобладание локального минимума

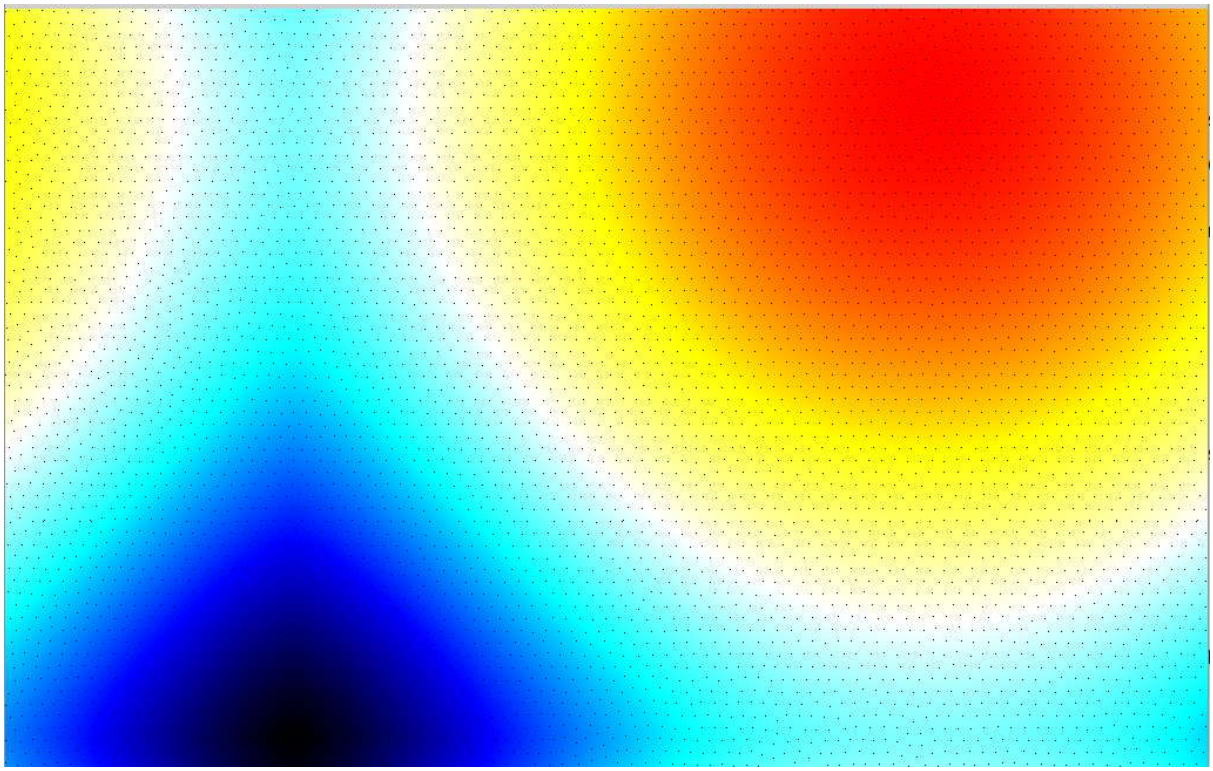


Преобладание глобального минимума




Himmelblau





Результаты

Размер окна	Число частиц	Время на CPU ms	Время на GPU ms
1024x648	1000	365,26	5.826
	100000	363,43	21,778
1920x1080	1000	1163,05	7,577
	100000	1162,59	47,73

Из-за того, что количество пикселей в окне много больше чем количество частиц, они почти не привносят вклад в вычислительную работу. Но зато хорошо видно, что gpu отлично справляется как с первым так и со вторым размером, разница чуть больше миллисекунды. В то время как cpu тратит в 3 раза больше времени (больше почти на 800 мс).

Выводы

Алгоритм роя частиц широко применяется в задачах оптимизации сложных многомерных нелинейных функций. По эффективности он может соперничать с другими методами глобальной оптимизации, а низкая алгоритмическая сложность способствует простоте его реализации. Также метод применяется в задачах машинного обучения (обучение нейросетей и распознавания изображений).