МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Курсовой проект**

**по курсу «Программирование графических процессоров»**

**Моделирование поведения роя частиц на GPU**

Выполнил: Н.А. Зайцев

Группа: 8О-408Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2016

**Условие**

Описание задачи:

**Цель работы.** Научиться использовать GPU для моделирования поведения косяка рыб (стаи птиц) при решении задачи глобальной оптимизации.

**Задание.**

1. Реализовать алгоритм роя частиц для задачи глобальной оптимизации Inertia Weighted SPO[1]. Коэффициент инерции w и параметры a1, a2 задаются пользователем.
2. Для решения проблемы коллизий, необходимо ввести силу отталкивания обратно пропорциональную расстоянию в некоторой степени (например, в четвертой) между частицами. Производить учёт только локальных взаимодействий. Для локализации частиц требуется использовать разбиение пространства.
3. 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 должна быть возможность интерактивно изменять масштаб.

На разработанном программном обеспечении выполнить исследование зависимости скорости сходимости от параметров w, a1 и a2, и отразить результаты в отчете. Так же необходимо подобрать такие параметры w, a1 и a2, при которых поведение роя частиц будет наиболее соответствовать поведению стаи птиц(косяка рыб) в природе. Провести сравнение производительности gpu и cpu(т.е. дополнительно нужно реализовать алгоритм без использования CUDA).

**Вариант 4:** Функция Розенброка [2].

Курсовая была реализована без разбиения пространства. Сила отталкивания была введена обратно пропорциональной расстоянию в четвёртой степени. Частицы отображаются белыми точками, а карта высот (температурная карта) оттенками зелёного.

Закон описывающий поведение частиц:

,

где w – коэффициент инерции, a1 и a2 коэффициенты ускорения, pbest – лучшая найденная частицей точка, gbest - лучшая точка из пройденных всеми частицами системы, k — эмпирический коэффициент, f — вектор-сила, действующая на данную частицу, x – текущее положение частицы, v – вектор скорости, r1 и r2 – случайные значения в диапазоне от 0 до 1 включительно.

**Программное и аппаратное обеспечение**

**GPU:**

Name : GeForce GTX 650

Compute capability : 3.0

Total Global Memory : 2147483648

Shared memory per block : 49152

Registers per block : 65536

Warp size : 32

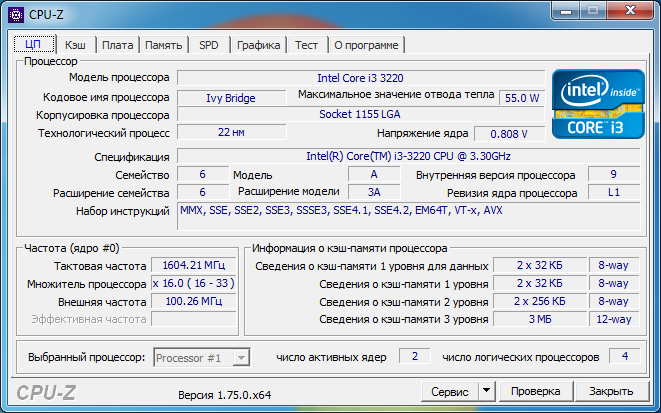
Max threads per block : (1024, 1024, 64)

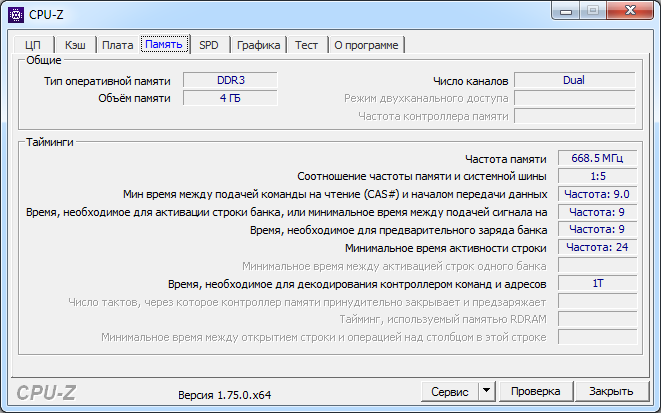
Max block : (2147483647, 65535, 65535)

Total constant memory : 65536

Multiprocessors count : 2

**CPU:**





**HDD:**

Производитель: Seagate

Модель: ST2000DM001-1CH164 ATA Device

Размер : 1,82 ТБ

**OS:**

Windows 7 Professional SP1 64-bit

**IDE:**

Microsoft Visual Studio 2013 + CUDA Plugins

**Compiler:**

nvcc

**Debugger**:

Nvidia CUDA Debugger

**Метод решения**

В начальный этап времени частицам задаётся случайная координата в пределах [0, 100] по оcи x и в пределах [0, 100] по оси y. И высчитывается наилучшее глобальное значение по сгенерированным точкам. После чего запускается процесс минимизации функции. Первое ядро заполняет массив, являющийся прообразом нашей температурной карты и размером с видимое окно программы, значениями функции, затем при помощи редукции находятся максимумы и минимумы значений функции в этом массиве для последующей нормировки. Затем, заполняется массив пикселей, используя значения из предыдущего массива с последующей нормировкой. Температурная карта построена. Следующее ядро высчитывает силы отталкивания всех частичек друг с другом. Следующее ядро производит подсчёт новых координат по приведённому ранее закону и отрисовывает на соответствующему частичке месте белый пиксель в массиве пикселей. При помощи редукции находится новая глобальная лучшая точка. Затем массив пикселей отрисовывается на экран в окне приложения. Процесс повторяется. Таким образом, ждём пока частицы не найдут глобальный минимум в окрестности точки (1, 1) – это минимум функции Розенброка [2].

**Описание программы**

Программа состоит из одного файла, но разбита на функцию main, несколько ядер и функций.

Структура particle:

struct particle

{

double2 x;

double2 v;

double2 p\_best;

double2 f;

};

Эта структура описывает параметры частички: координата, вектор скорости, наилучшее значение и вектор силы отталкивания, действующий на эту частичку.

Ядро kernel\_particles\_init:

\_\_global\_\_ void kernel\_particles\_init(particle \*p\_arr, int p\_number, curandState \* state, unsigned long seed)

{

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

while (tid < p\_number)

{

p\_arr[tid].v = double2();

p\_arr[tid].f = double2();

curand\_init(seed, tid, 0, &state[tid]); // U(0,1)

p\_arr[tid].x.x = curand\_uniform(&state[tid]) \* 100;

p\_arr[tid].x.y = curand\_uniform(&state[tid]) \* 100;

p\_arr[tid].p\_best = p\_arr[tid].x;

tid += blockDim.x \* gridDim.x;

}

}

Используется одномерная сетка с 16-ю блоками по 16 потоков. Ядро производит стартовую инициализацию массива частичек. Начальную координата выставляется в пределах [0, 100] по оcи x и в пределах [0, 100] по оси y, вектор скорости и вектор силы равны нулевому вектору. Наилучшее значение выставляется равным координате.

Ядро kernel\_g\_best\_find:

\_\_global\_\_ void kernel\_g\_best\_find(particle \*p\_arr, double2 \*partial\_g\_best, int p\_number)

{

\_\_shared\_\_ double2 seq[32];

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

seq[threadIdx.x] = p\_arr[tid].x;

\_\_syncthreads();

int pow = 2;

while (pow <= 32)

{

if (threadIdx.x \* pow + pow - 1 < 32)

{

seq[threadIdx.x \* pow + pow - 1] = (fun(seq[threadIdx.x \* pow + pow - 1]) < fun(seq[threadIdx.x \* pow + pow - pow / 2 - 1])) ? seq[threadIdx.x \* pow + pow - 1] : seq[threadIdx.x \* pow + pow - pow / 2 - 1];

}

\_\_syncthreads();

pow \*= 2;

}

if (threadIdx.x == 0)

{

partial\_g\_best[blockIdx.x] = seq[31];

}

}

Используется одномерная сетка с количеством блоков равным количеству точек целочисленно делённым на 32, 32 – количество потоков в блоке. При помощи редукции находит наилучшее значение функции среди подпоследовательностей (число которых равно количеству блоков) точек длиной 32.

Ядро kernel\_g\_best\_find\_final:

\_\_global\_\_ void kernel\_g\_best\_find\_final(particle \*p\_arr, double2 \*partial\_g\_best, int size, int p\_number)

{

double2 max;

if (size > 0)

{

max = partial\_g\_best[0];

}

else

{

max = p\_arr[0].x;

}

for (int i = 1; i < size; i++)

{

if (fun(partial\_g\_best[i]) < fun(partial\_g\_best[i - 1]))

max = partial\_g\_best[i];

}

for (int i = 32 \* size; i < p\_number; i++)

{

if (fun(p\_arr[i].x) < fun(max))

max = p\_arr[i].x;

}

if(fun(max) < fun(g\_best))

g\_best = max;

}

Ядро запускается в однопоточном режиме с одним блоком. Производит окончательный поиск наилучшего значения функции среди лучших значений подпоследовательностей и остатка последовательности точек. В случае если оно оказывается лучше глобального, то глобальное значение обновляется на найденное.

Ядро kernel\_g\_best\_find\_final\_init:

\_\_global\_\_ void kernel\_g\_best\_find\_final\_init(particle \*p\_arr, double2 \*partial\_g\_best, int size, int p\_number)

{

double2 max;

if (size > 0)

{

max = partial\_g\_best[0];

}

else

{

max = p\_arr[0].x;

}

for (int i = 1; i < size; i++)

{

if (fun(partial\_g\_best[i]) < fun(partial\_g\_best[i - 1]))

max = partial\_g\_best[i];

}

for (int i = 32 \* size; i < p\_number; i++)

{

if (fun(p\_arr[i].x) < fun(max))

max = p\_arr[i].x;

}

g\_best = max;

}

От предыдущего ядра отличается только тем, что используется на этапе инициализации, и это ядро инициализирует наилучшее глобальное значение найденным значением.

Ядро kernel\_mass\_center:

\_\_global\_\_ void kernel\_mass\_center(particle \*p\_arr, int p\_number)

{

double2 tmp = double2();

for (int i = 0; i < p\_number; i++)

{

tmp.x += p\_arr[i].x.x;

tmp.y += p\_arr[i].x.y;

}

tmp.x /= p\_number;

tmp.y /= p\_number;

dev\_xc = tmp.x;

dev\_yc = tmp.y;

}

Ядро запускается в однопоточном режиме и подсчитывает центр масс, который затем используется как точка центра отрисовки экрана.

Ядро kernel\_data\_raw:

\_\_global\_\_ void kernel\_data\_raw(double scale\_x, double scale\_y) {

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

int idy = blockIdx.y \* blockDim.y + threadIdx.y;

int offsetx = blockDim.x \* gridDim.x;

int offsety = blockDim.y \* gridDim.y;

int i, j;

for (j = idy; j < height; j += offsety)

{

for (i = idx; i < width; i += offsetx)

{

data\_raw[j \* width + i] = fun(i, j, scale\_x, scale\_y);

}

}

}

Используется двумерная сетка по 16х16 потоков в каждом из 32х32 блоков. Производит заполнение массива размером с экран значениями функции.

Ядро kernel\_find\_max:

\_\_global\_\_ void kernel\_find\_max(double \*partial\_max)

{

\_\_shared\_\_ double seq[reduce\_threads\_count];

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

if (tid < width \* height)

{

seq[threadIdx.x] = data\_raw[tid];

}

else

{

seq[threadIdx.x] = -1; // fun() always >= 0;

}

\_\_syncthreads();

int pow = 2;

while (pow <= reduce\_threads\_count)

{

if (threadIdx.x \* pow + pow - 1 < reduce\_threads\_count)

{

seq[threadIdx.x \* pow + pow - 1] = (seq[threadIdx.x \* pow + pow - 1] > seq[threadIdx.x \* pow + pow - pow / 2 - 1]) ? seq[threadIdx.x \* pow + pow - 1] : seq[threadIdx.x \* pow + pow - pow / 2 - 1];

}

\_\_syncthreads();

pow \*= 2;

}

if (threadIdx.x == 0)

{

partial\_max[blockIdx.x] = seq[reduce\_threads\_count - 1];

}

}

Используется одномерная сетка по 1024 потока в каждом из width \* height / 1024 + 1 блоков, где width – ширина экрана, height – его высота. При помощи редукции находит наилучшее значение функции среди подпоследовательностей точек длиной 1024, в качестве незначащего элемента используется -1, так как функция всегда больше этого значения.

Ядро kernel\_find\_max\_final:

\_\_global\_\_ void kernel\_find\_max\_final(double \*partial\_max, int size)

{

double max = partial\_max[0];

for (int i = 1; i < size; i++)

{

if (partial\_max[i] > partial\_max[i - 1])

max = partial\_max[i];

}

dev\_maxf = max;

}

Подсчитывает в однопоточном режиме окончательный результат. Находит максимум среди максимумов на подпоследовательностях.

Ядро kernel\_find\_min:

\_\_global\_\_ void kernel\_find\_min(double \*partial\_min)

{

\_\_shared\_\_ double seq[reduce\_threads\_count];

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

if (tid < width \* height)

{

seq[threadIdx.x] = data\_raw[tid];

}

else

{

seq[threadIdx.x] = INFINITY;

}

\_\_syncthreads();

int pow = 2;

while (pow <= reduce\_threads\_count)

{

if (threadIdx.x \* pow + pow - 1 < reduce\_threads\_count)

{

seq[threadIdx.x \* pow + pow - 1] = (seq[threadIdx.x \* pow + pow - 1] < seq[threadIdx.x \* pow + pow - pow / 2 - 1]) ? seq[threadIdx.x \* pow + pow - 1] : seq[threadIdx.x \* pow + pow - pow / 2 - 1];

}

\_\_syncthreads();

pow \*= 2;

}

if (threadIdx.x == 0)

{

partial\_min[blockIdx.x] = seq[reduce\_threads\_count - 1];

}

}

Всё аналогично kernel\_find\_max, только ищется минимум.

Ядро kernel\_find\_min\_final:

\_\_global\_\_ void kernel\_find\_min\_final(double \*partial\_min, int size)

{

double min = partial\_min[0];

for (int i = 1; i < size; i++)

{

if (partial\_min[i] < partial\_min[i - 1])

min = partial\_min[i];

}

dev\_minf = min;

}

Всё аналогично kernel\_find\_max\_final, только ищется минимум.

Ядро kernel\_data:

\_\_global\_\_ void kernel\_data(uchar4\* data)

{

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

int idy = blockIdx.y \* blockDim.y + threadIdx.y;

int offsetx = blockDim.x \* gridDim.x;

int offsety = blockDim.y \* gridDim.y;

int i, j;

for (j = idy; j < height; j += offsety)

{

for (i = idx; i < width; i += offsetx)

{

double f = (data\_raw[j \* width + i] - dev\_minf) / (dev\_maxf - dev\_minf);

data[j \* width + i] = make\_uchar4(0, (int)(f \* 255), 0, 255);

}

}

}

Используется двумерная сетка по 16х16 потоков в каждом из 32х32 блоков. Производит нормировку высотной карты по найденным в предыдущих ядрах минимуму и максимуму функции, окрашивает пиксель в соответствующий высоте оттенок зелёного – чем выше, тем зеленее, чем ниже, тем темнее вплоть до чёрного цвета. Таким образом строится температурная карта.

Ядро kernel\_calc\_force:

#define sqr(x) ((x)\*(x))

\_\_global\_\_ void kernel\_calc\_force(particle \*p\_arr, int p\_number)

{

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

while (tid < p\_number)

{

p\_arr[tid].f = double2();

for (int i = 0; i < p\_number; i++)

{

if (i == tid)

continue;

double d2 = sqr(p\_arr[i].x.x - p\_arr[tid].x.x) + sqr(p\_arr[i].x.y - p\_arr[tid].x.y);

p\_arr[tid].f.x -= (p\_arr[i].x.x - p\_arr[tid].x.x) / (sqr(d2) + 1e-3);

p\_arr[tid].f.y -= (p\_arr[i].x.y - p\_arr[tid].x.y) / (sqr(d2) + 1e-3);

}

tid += blockDim.x \* gridDim.x;

}

}

Используется одномерная сетка по 64 потока в каждом из 64-х блоков. Ядро подсчитывает силы отталкивания всех частичек друг с другом.

Функция calc:

\_\_device\_\_ void calc(particle & p, double w, double a1, double a2, double dt, double k, curandState \* state)

{

p.v.x = w \* p.v.x + (a1 \* curand\_uniform(state) \* (p.p\_best.x - p.x.x) + a2 \* curand\_uniform(state) \* (g\_best.x - p.x.x) + k \* p.f.x) \* dt;

p.v.y = w \* p.v.y + (a1 \* curand\_uniform(state) \* (p.p\_best.y - p.x.y) + a2 \* curand\_uniform(state) \* (g\_best.y - p.x.y) + k \* p.f.y) \* dt;

p.x.x += p.v.x \* dt;

p.x.y += p.v.y \* dt;

if (fun(p.x.x, p.x.y) < fun(p.p\_best.x, p.p\_best.y))

{

p.p\_best.x = p.x.x;

p.p\_best.y = p.x.y;

}

}

Функция подсчитывает для частицы новые значения координаты, вектора скорости, а также лучшего значения точки. Запускается из ядра kernel\_main.

Ядро kernel\_main:

\_\_global\_\_ void kernel\_main(double w, double a1, double a2, double dt, double k, particle \*p\_arr, int p\_number, uchar4\* data, double scale\_x, double scale\_y, curandState \* state)

{

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

while (tid < p\_number)

{

calc(p\_arr[tid], w, a1, a2, dt, k, &state[tid]);

double2 tmp = p\_arr[tid].x;

tmp.x -= dev\_xc;

tmp.x /= scale\_x;

tmp.x += 1;

tmp.x \*= (double)(width - 1);

tmp.x /= 2;

tmp.y -= dev\_yc;

tmp.y /= scale\_y;

tmp.y \*= -1;

tmp.y += 1;

tmp.y \*= (double)(height - 1);

tmp.y /= 2;

int2 tmp2;

tmp2.x = (int)tmp.x;

tmp2.y = (int)tmp.y;

if (tmp2.x > 0 && tmp2.x < width && tmp2.y > 0 && tmp2.y < height)

{

data[tmp2.y \* width + tmp2.x] = make\_uchar4(255, 255, 255, 255);

}

tid += blockDim.x \* gridDim.x;

}

}

Используется одномерная сетка по 32 потока в каждом из 32-х блоков. Ядро подсчитывает новые значения параметров частиц и затем, если они находятся в поле видимости окна, отрисовываются на экране белым пикселем.

Ядро kernel\_printf:

\_\_global\_\_ void kernel\_printf(double2 \*g\_best\_to\_host)

{

\*g\_best\_to\_host = g\_best;

}

Ядро работает в однопоточном режиме и нужно лишь только для того, чтобы скопировать значение глобального минимума на хост.

Функция update:

void update() {

uchar4\* dev\_data;

size\_t size;

auto start\_time = chrono::high\_resolution\_clock::now();

CSC(cudaGraphicsMapResources(1, &res, 0));

CSC(cudaGraphicsResourceGetMappedPointer((void\*\*) &dev\_data, &size, res));

kernel\_mass\_center << <1, 1 >> >(p\_arr\_dev, p\_number);

kernel\_data\_raw << <blocks, threads >> >(scale\_x, scale\_y);

kernel\_find\_max << <reduce\_blocks\_count, reduce\_threads\_count >> >(partial\_max\_dev);

kernel\_find\_max\_final << <1, 1 >> >(partial\_max\_dev, reduce\_blocks\_count);

kernel\_find\_min << <reduce\_blocks\_count, reduce\_threads\_count >> >(partial\_min\_dev);

kernel\_find\_min\_final << <1, 1 >> >(partial\_min\_dev, reduce\_blocks\_count);

kernel\_data<<<blocks, threads>>>(dev\_data);

kernel\_calc\_force << <64, 64 >> >(p\_arr\_dev, p\_number);

kernel\_main << <32, 32 >> >(w, a1, a2, dt, k, p\_arr\_dev, p\_number, dev\_data, scale\_x, scale\_y, devStates);

kernel\_g\_best\_find << <p\_number / 32, 32 >> >(p\_arr\_dev, partial\_g\_best, p\_number);

kernel\_g\_best\_find\_final << <1, 1 >> >(p\_arr\_dev, partial\_g\_best, p\_number / 32, p\_number);

kernel\_printf << <1, 1 >> >(g\_best\_dev);

CSC(cudaDeviceSynchronize());

CSC(cudaGraphicsUnmapResources(1, &res, 0));

CSC(cudaMemcpy(g\_best\_host, g\_best\_dev, sizeof(double2), cudaMemcpyDeviceToHost));

auto end\_time = chrono::high\_resolution\_clock::now();

printf("%f %f\n", (\*g\_best\_host).x, (\*g\_best\_host).y);

cout << chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time).count() << "\n";

glutPostRedisplay();

}

Эта функция запускается циклически и предназначена для обновления массива пикселей, которые после её выполнения будут отрисованы на экране. Здесь же можно проследить последовательность вызова всех ядер, кроме ядер инициализации.

Функция MyKeyboardFunc:

void MyKeyboardFunc(unsigned char Key, int x, int y)

{

switch (Key)

{

case '-':

scale\_x += 0.5;

scale\_y = scale\_x \* height / width;

break;

case '+':

if (scale\_x > 1)

{

scale\_x -= 0.5;

scale\_y = scale\_x \* height / width;

}

break;

};

}

Эта функция вызывается в случае нажатия на клавиши клавиатуры, и меняет коэффициент масштаба изображения на экране.

Перегрузки функции fun:

\_\_device\_\_ double fun(double2 arg) {

return sqr((1 - arg.x)) + 100 \* sqr((arg.y - sqr(arg.x)));

}

\_\_device\_\_ double fun(double x, double y) {

return sqr((1 - x)) + 100 \* sqr((y - sqr(x)));

}

Эти две функции отличаются только типом аргумента, а делают одно и то же – по заданному значению x и y возвращают значение функции Розенброка в этой точке.

\_\_device\_\_ double fun(int i, int j, double scale\_x, double scale\_y) {

double x = 2.0f \* i / (double)(width - 1) - 1.0f;

double y = 2.0f \* j / (double)(height - 1) - 1.0f;

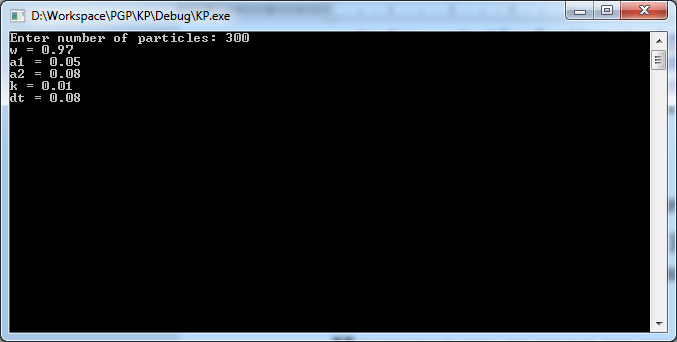
return fun(x \* scale\_x + dev\_xc, -y \* scale\_y + dev\_yc);

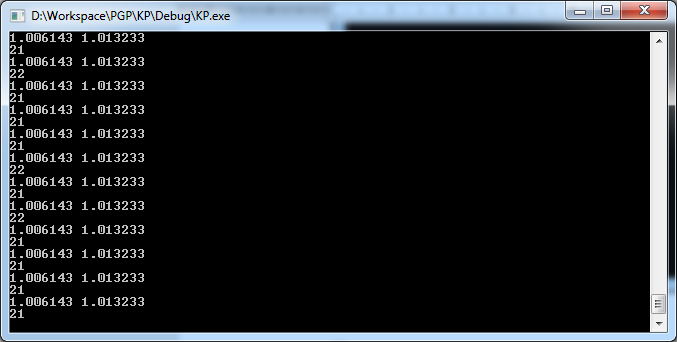
}

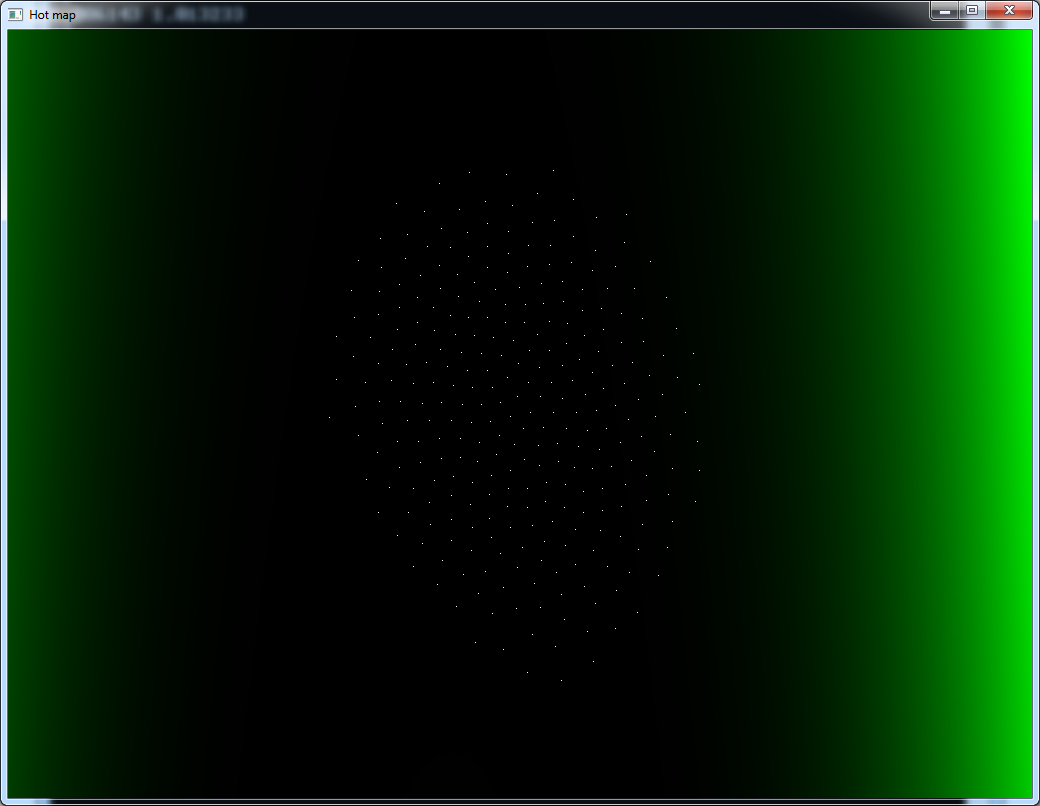
Эта же функция переводит значения индексов массива пикселей в значения функции с учётом масштабирования и расположения центра экрана. Используется на этапе первичного заполнения массива пикселей до нормировки.

**Исследовательская часть**

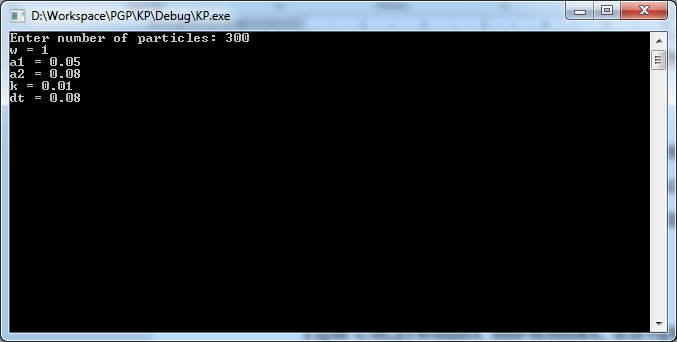
При следующих значениях, алгоритм сошёлся где-то за минуту:

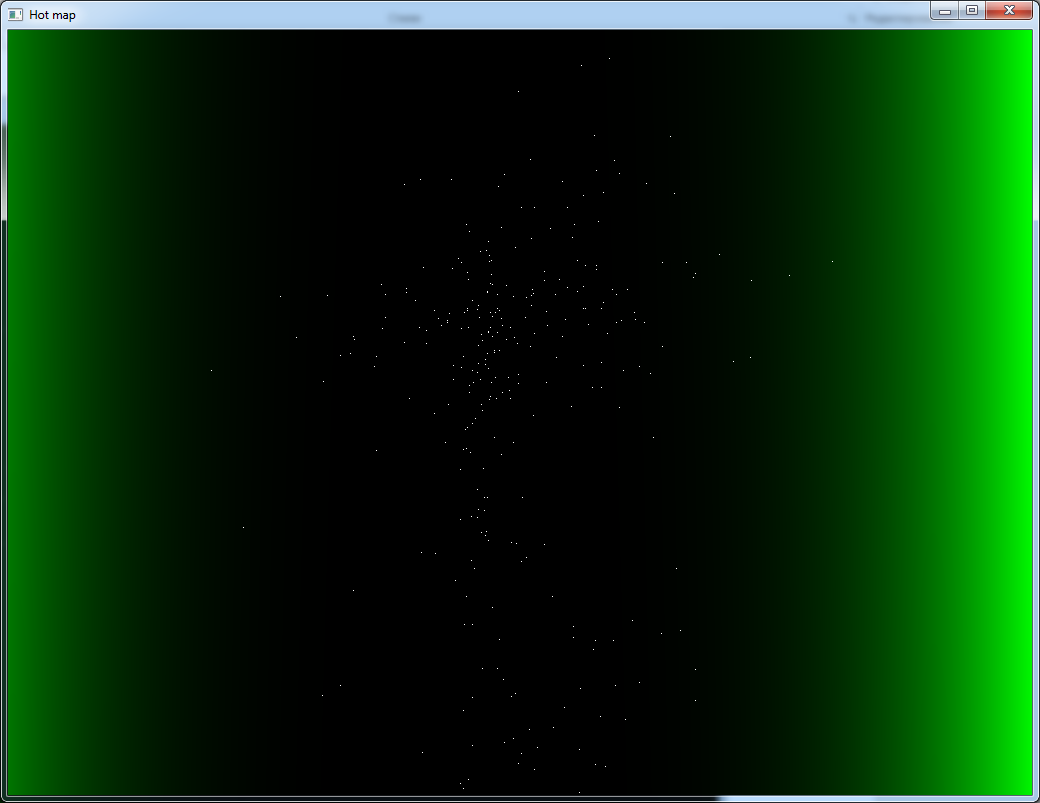
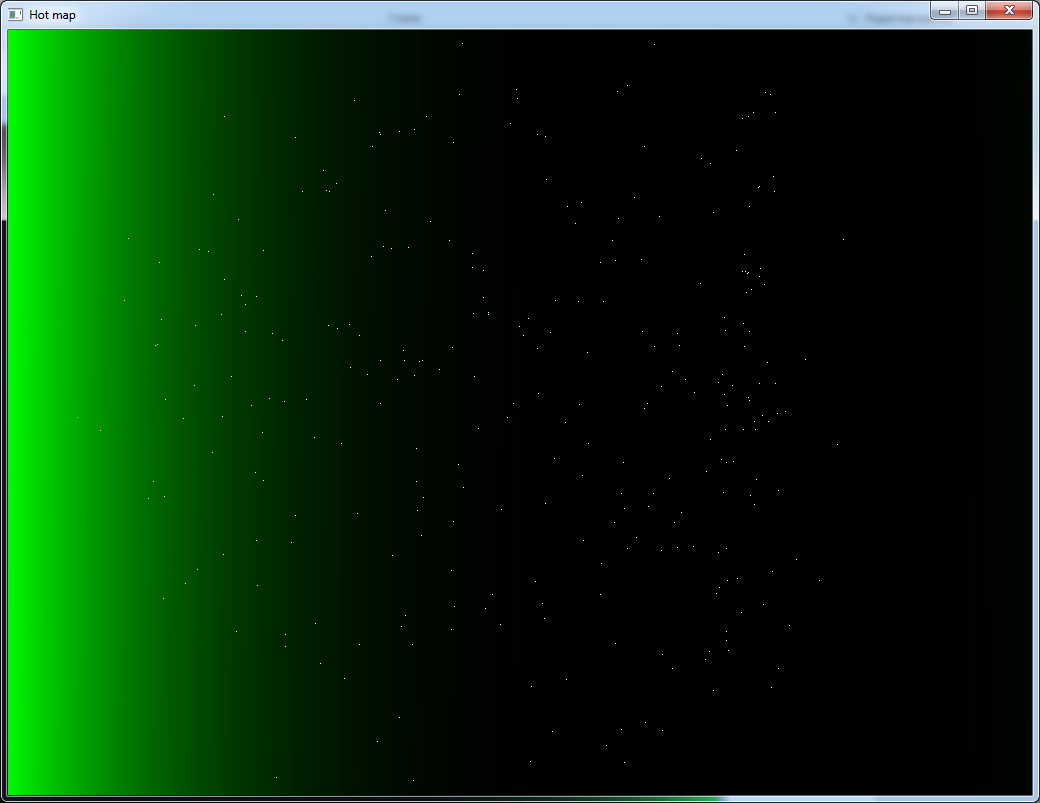
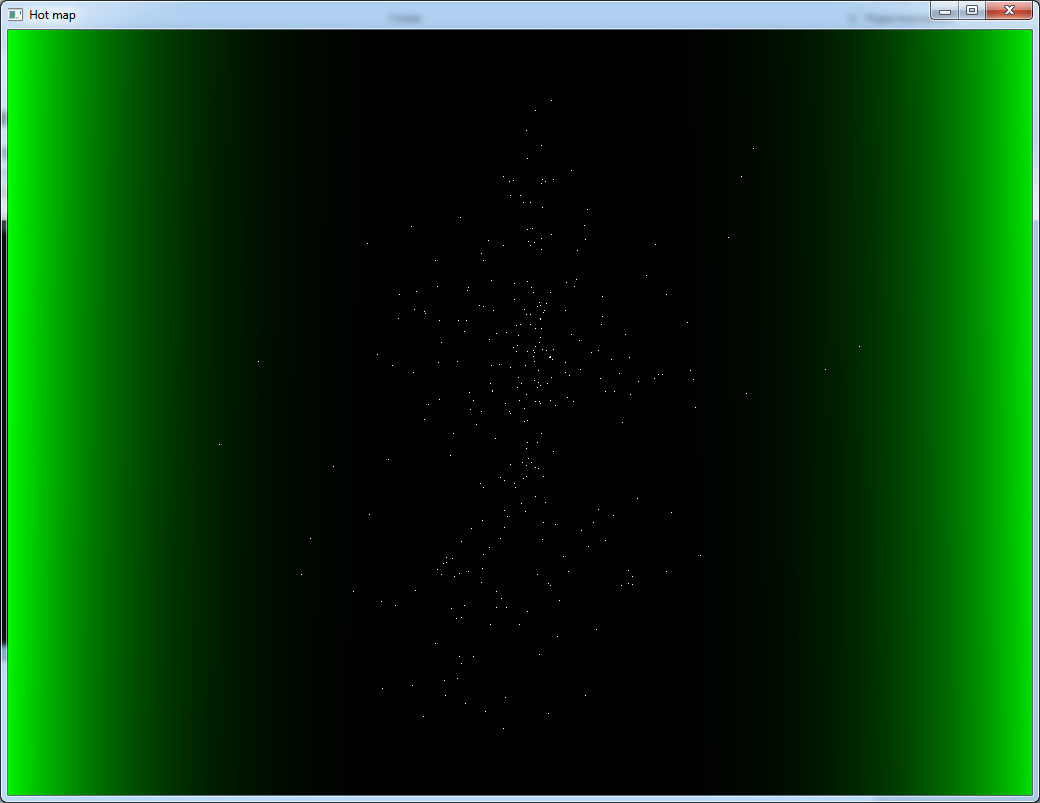




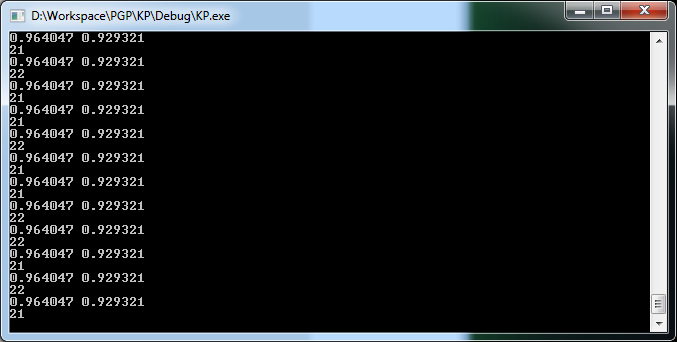


Метод будет неустойчивым, если коэффициент инерции w >= 1:



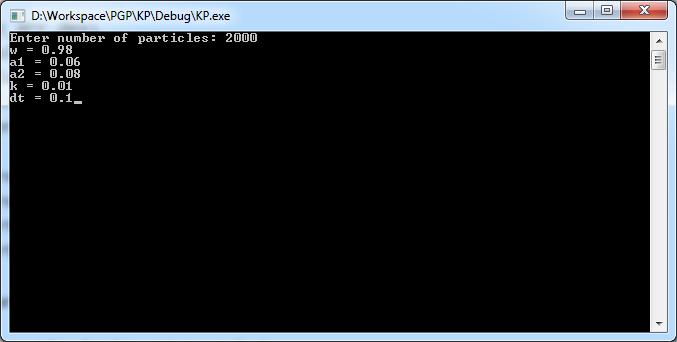


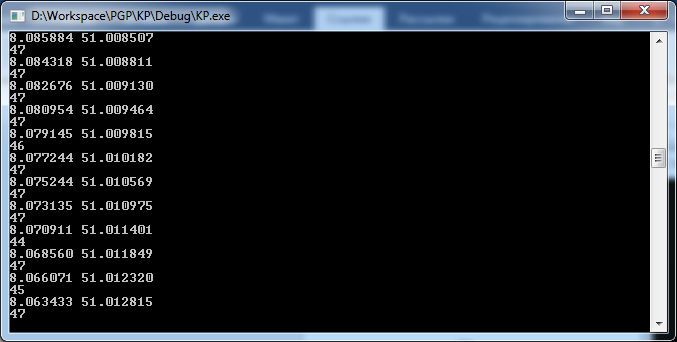
Хотя глобальный минимум был всё-таки найден:



**Результаты**

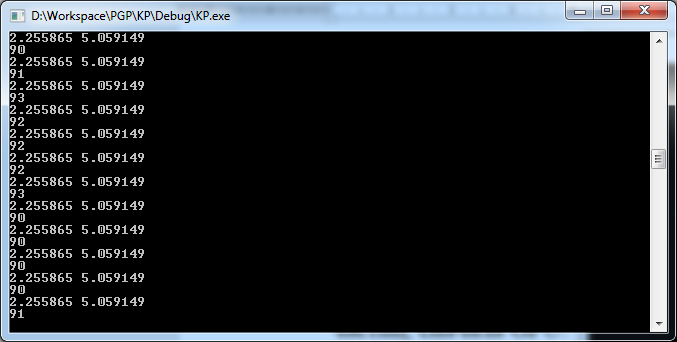
Давайте сравним производительность программы, реализованной на CPU с программой, реализованной на GPU с использованием CUDA, на тесте c несколькими тысячами частиц, сначала GPU:





Видим, что программа тратит в среднем 46-47 мс на отрисовку одного кадра.

Посмотрим какая производительность будет на CPU на том же тесте:



Программа на CPU тратит в среднем 92 мс, что почти в два раза медленнее чем на GPU.

**Выводы**

В результате выполнения данной работы я научился использовать GPU для моделирования поведения косяка рыб (стаи птиц) при решении задачи глобальной оптимизации.

**Использованные источники**

1. <http://nauchforum.ru/en/node/3179>
2. https://ru.wikipedia.org/wiki/Функция\_Розенброка