ПРИМЕНЕНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ ДЛЯ МИНИМИЗАЦИИ ФУНКЦИОНАЛОВ, ВОЗНИКАЮЩИХ В ПРОЦЕССЕ РЕШЕНИЯ ЗАДАЧ ОПРЕДЕЛЕНИЯ ПАРАМЕТРОВ ИНФОРМАЦИОННЫХ СИГНАЛОВ С ПОМОЩЬЮ ТЕОРИИ МАКСИМАЛЬНОГО ПРАВДОПОДОБИЯ

В. И. Строков

Балтийский федеральный университет им. И. Канта, Калининград

Статья получена 18 января 2015 г.

Аннотация. В последнее время в литературе все чаще упоминается метод максимального правдоподобия как способ решения той или иной задачи, однако практических результатов, полученных на реальных данных мало. Чаще всего это связано с неразрешимостью задачи в аналитическом виде и огромной вычислительной сложностью при решении задачи численно. В данной статье описана возможность решения задач, возникающих при использовании теории оптимального приема, основанная на применении графических процессоров, что позволяет существенно уменьшить время получения конечного результата.

Ключевые слова: метод максимального правдоподобия, функционал правдоподобия, минимизация функционала.

Abstract: In the recent literature the maximum likelihood method is increasingly referred to, as a way of solving a problem, but practical results obtained on real data is not enough. Most often this is due to unsolvable problems analytically and enormous computational complexity for solving the problem numerically. This article describes the possibility of solving problems arising from the application of the theory of optimal reception, based on the use of graphics processors that can significantly reduce the time of receipt of the result.

Key words: maximum likelihood method, the functional likelihood, minimization of functional.

Как известно, в процессе решения задачи оценки параметров сигналов с помощью теории максимального правдоподобия возникает системный

функционал, зависящий от оцениваемых параметров информационного сигнала, требующий минимизации для нахождения искомого решения:

$$F\left(\xi_i\right) \to \min,$$

где ξ_i - оцениваемые параметры сигнала (амплитуда, фаза, время приема, частота и т.п.).

Обычно, в случае оценки частоты или времени приема сигнала минимизация функционала в аналитическом виде неосуществима, и приходится прибегать к численным методам решения задачи. Однако данный подход связан с высокой вычислительной сложностью данной процедуры, что и ограничивает применение теории оптимального приема для оценки таких параметров сигнала.

Выходом из данной ситуации может быть следующее:

1. Нахождение подходящего метода оптимизации, применимого для данной задачи.
2. Применение графических процессоров для ускорения процесса минимизации по указанной сетке.

Первый метод основан на применении, например, эволюционных алгоритмов оптимизации для нахождения решения. Однако, в случае большого количества шумовых минимумов и (или) в случае недостаточного количества итерационных шагов алгоритма возможна сходимость к ложному решению.

Приведем пример функционала правдоподобия.

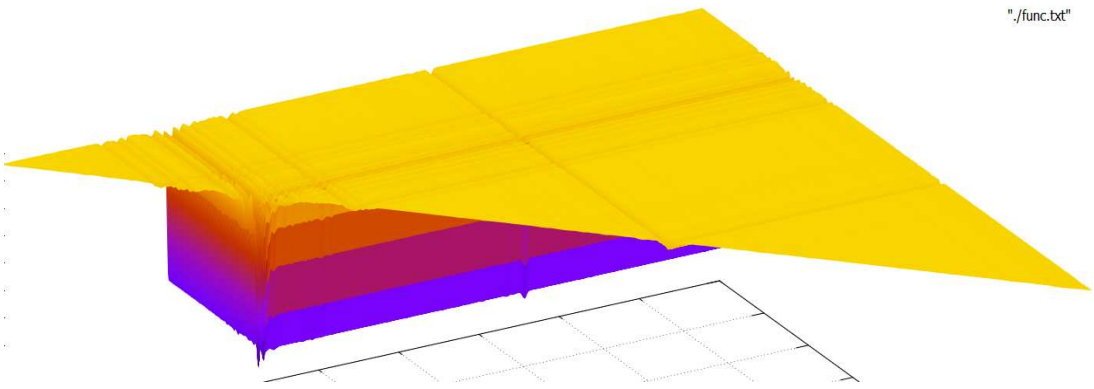

Рисунок 1 - Пример функционала правдоподобия.

Функционал имеет один очень узкий глобальный минимум и множество локальных минимумов, поэтому задача подбора подходящего итерационного алгоритма не является тривиальной. В то же время возможность быстрого решения задачи по заданной сетке позволяет решить задачу точно с учетом выбранного шага.

Данная статья нацелена на обзор второй возможности, а именно – применение математических неграфических вычислений на графических процессорах (все сказанное ниже относится исключительно к архитектуре CUDA от NVIDIA).

CUDA (*Compute Unified Device Architecture*) – это архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров) [1].

Архитектура типичного видеоадаптера CUDA представлена на рисунке 2. Устройство состоит N SIMD мультипроцессоров, каждый из которых состоит из M процессоров. Так же на устройстве имеется память различных видов. Работа с архитектурой CUDA заключается в написании ядра – kernel.

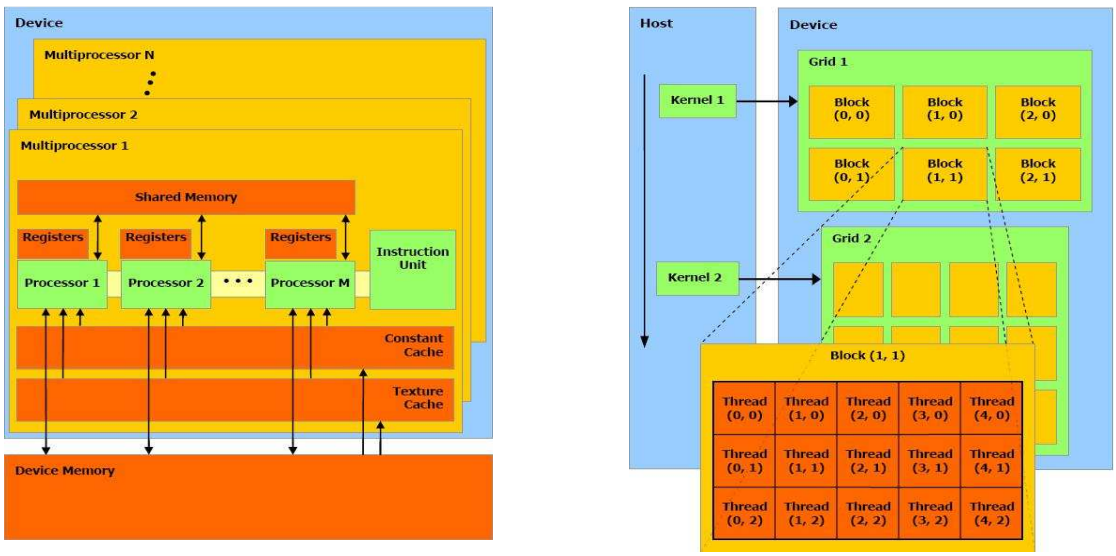

Рисунок 2 - Архитектура CUDA(слева), работа ядра(справа).

Kernel принимает параметры – число блоков (Blocks) и число нитей (Threads) запускаемых в блоке. Таким образом создается Blocks* Threads параллельных потоков для выполнения операций, описанных в ядре kernel.

Рассмотрим основные шаги, необходимые, для организации вычисления минимума функционала с применением параллельных потоков (предполагается, что есть сформированная функция функционала правдоподобия, реализованная в виде: *float func_cpu(float x)*, где x – аргумент, принимаемый функцией (рассматриваем одномерный случай)).

Шаг 1. Определяем минимизируемую функцию для вызова с устройства (device), путем добавления специального квалификатора __device__ и незначительных модификаций тела функции. Минимизирумая функция примет вид:

__device__ float func_dev(float x).

Пример: Пусть функция, которую мы хотим минимизировать представляет собой Schwefel function

$$f\left(x\right)=418.9829d-\sum_{i=1}^{d}x_i\sin\left(\sqrt{\left|x_i\right|}\right),$$

тогда функции func_cpu() и func_dev() будут иметь вид:

```
float Schwefel_cpu(float x)
{
    return 418.9829 - x*sinf(sqrt(fabs(x);
}
__device__ float Schwefel_dev(float x)
{
    return 418.9829f - x*__sinf(sqrt(fabs(x);
}.
```

Как видно, изменения в теле функции минимальны и заключаются лишь в изменении функции sinf() на функцию __sinf(), которая позволяет вычислять тригонометрические значения синуса на видеоадаптере с меньшей точность, но с большей скоростью, чем стандартная функция c++ из math.h (аналогичные функции существуют и для других тригонометрических выражений).

Шаг 2. Реализуем ядро, осуществляющее расчет значений полученной функции с сохранением данных значений в массив.

Представим ядро на примере минимизации Schwefel function. Данное ядро принимает следующие параметры: **result** – массив, куда будут записаны рассчитанные значения Schwefel function на линейной сетке с числом узлов – **size**, расстоянием между узлами **step** и начальным положением **begin**.

```
__global__ void Schwefel_kernel(float *result, uint64_t size, float step, float begin)
{
    uint64_t tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < size)
        result[tid] = Schwefel(begin + step*tid);
}
```

Т.к. при запуске ядра создается много параллельных потоков, то мы имеем возможность рассчитывать значения минимизируемой функции одновременно для разных значений параметра x, что позволяет существенно сократить время нахождения минимума функционала. Т.к. запускаемые параллельно нити будут рассчитывать свое значение Schwefel function, то нам не нужно использовать цикл for(..), который мы бы использовали для минимизации по сетке на CPU.

В теле ядра мы видим переменную tid, которая идентифицирует номер запускаемого потока, она складывается из номера блока blockIdx.x по оси x, threadIdx.x – номера нити в блоке по этой же оси (CUDA поддерживает трехмерную индексацию, в одномерном случае воспользуемся осью x).

Константа blockDim.x указывает максимальное число нитей в одном блоке вдоль заданного направления.

Условие if (tid < size) не дает ядру выйти за пределы массива, т.к. обычно число запускаемых потоков больше размера массив. Квалификатор __global__ указывает на то, что ядро, будучи вызванным на основной машине (host), будет исполнено на видеоадаптере (device). Использованный ранее квалификатор __device__ указывает на то, что функция будет вызвана с устройства (device) и выполнена на нем.

Шаг 3. Написание функции, реализующей поиск минимального элемента в массиве насчитанных значений.

Основная функция в этом блоке - void distance_min(..), остальные функции вспомогательные. Функция atomicMin – атомарная функция, принимающая следующие параметры: val_adress – адрес, куда будет записано минимальное из чисел *val_adress и val, pos_adress – адрес, хранящий позицию минимального элемента. Позиция элемента val передается параметром pos. Действия функций pos_min(..) и minimum(..) в комментариях не нуждаются.

```
__device__ __forceinline__ float atomicMin(float *val_adress, float val, uint64_t *pos_adress, uint64_t pos)
{
int ret = __float_as_int(*val_adress);
while (val < __int_as_float(ret)) {
	int old = ret;
	if ((ret = atomicCAS((int *)val_adress, old, __float_as_int(val))) == old) {
		atomicCAS(pos_adress, *pos_adress, pos);
		break;
	}
}
return __int_as_float(ret);
}
```

```
__device__ uint64_t pos_min(float a, float b, uint64_t p1, uint64_t p2)
{
	if (a < b) return p1;
	return p2;
}

__device__ float minimum(float a, float b)
{
	if (a < b) return a;
	return b;
}

__global__ void distance_min(float *a, uint64_t size, float *min, uint64_t *pos)
{
	__shared__ float cache[threadsPerBlock]; // threadsPerBlock – число потоков, //запускаемых в одном блоке (THREADS)
	__shared__ uint64_t pos_cache[threadsPerBlock];
	uint64_t tid = threadIdx.x + blockIdx.x * blockDim.x;
	uint64_t cacheIndex = threadIdx.x;

	if (tid == 0)
	*min = CUDART_INF_F; //возвращаем значение равное бесконечности

	cache[cacheIndex] = CUDART_INF_F;
	pos_cache[cacheIndex] = 0;

	if (tid < size)
	{
		cache[cacheIndex] = a[tid];
		pos_cache[cacheIndex] = tid;
```

```
    }

    __syncthreads(); // синхронизация всех нитей

    uint64_t i = blockDim.x / 2;
    while (i != 0) {
        if (cacheIndex < i)
        {
            pos_cache[cacheIndex] = pos_min(cache[cacheIndex], cache[cacheIndex + i], pos_cache[cacheIndex], pos_cache[cacheIndex + i]);
            cache[cacheIndex] = minimum(cache[cacheIndex], cache[cacheIndex + i]);
        }
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        atomicMin(min, cache[0], pos, pos_cache[0]);
}
```

Функция distance_min() с квалификатором __global__ осуществляет поиск минимального элемента в массиве **a** размера **size**, **pos** – указатель на позицию найденного минимального элемента в массиве, **min** – указатель на значение минимального элемента.

Поясним принцип работы функции. Для каждого блока мы создаем два массива: **cache** и **pos_cache**, размером равным числу нитей в блоке. Для каждого блока заполняем эти массивы данными из входного массива **a**, так что массив **cache** будет хранить значения, а массив **pos_cache** – положения элементов. Далее алгоритмом, похожим на алгоритм редукции, в параллельной секции попарно сравниваем элементы данных массивов, запоминая наименьшее

значение и его положение в исходном массиве **a**, до тех пор, пока в каждом блоке не останется по одному элементу. Затем на последнем шаге находим минимальный элемент из оставшихся.

Шаг 4. Осуществляем минимизацию функции.

*float *dev_funcvals; //создаем указатель на память, где будем хранить насчитанные значения функции*
*cudaMalloc((void**)&dev_funcvals, number*sizeof(float));//выделяем память под массив* ***dev_funcvals*** *размера* ***number***
//создаем указатели на нуль-копируемую память для хранения результатов расчетов
*float *host_minumum, *dev_minimum;*
*uint64_t *host_position, *dev_position;*
//выделяем память
*cudaHostAlloc((void**)&host_minumum, sizeof(float), cudaHostAllocWriteCombined | cudaHostAllocMapped);*
cudaHostGetDevicePointer(&dev_minimum, host_minumum, 0);
*cudaHostAlloc((void**)&host_position, sizeof(uint64_t), cudaHostAllocWriteCombined | cudaHostAllocMapped);*
cudaHostGetDevicePointer(&dev_position, host_position, 0);

//вызов ядра расчета значений функции
Schwefel_kernel << <BLOCKS, THREADS >> >(dev_funcvals, number, step, begin);
//вызов ядра для поиска минимального элемента в массиве dev_funcvals
distance_min << <BLOCKS, THREADS >> >(dev_funcvals, number, dev_minimum, dev_position);
//ожидание завершения расчета
cudaThreadSynchronize();

BLOCKS, THREADS – параметры запуска ядра. BLOCKS – число блоков, задействованных на решение задачи, описанной в ядре, THREADS – число нитей запущенных на блок.

Выбирая THREADS равным некоторому значению, BLOCKS можно определить по формуле, указанной ниже:

```
uint64_t THREADS = 1024;
uint64_t BLOCKS = (number + THREADS - 1) / THREADS;
```

где number – число узлов в сетке, по которой производится расчет значений функционала.

В теории оптимального приема для расчёта параметров радиотехнических сигналов (путем минимизации функционала) часто приходится выполнять следующие арифметические операции:

1. скалярные произведения,
2. вычисление тригонометрических функций,
3. вычисление обратных матриц,
4. умножение матриц.

Поэтому, для теста производительности представленной системы выберем функцию, содержащую все эти элементы. Пример ядра такой функции представлен ниже. Данная функция производит оценку частоты помехи в некотором сигнале, моделируемом с помощью константы const_sig.

```
__global__ void func_kernel(float *result, uint64_t M, float step, float begin)
{
uint64_t tid = threadIdx.x + blockIdx.x * blockDim.x;
while (tid < M)
{
    float rcolumn[4], lcolumn[4];
    unsigned short int kod = 0xc44b;
```

```
float corrmatrix[4][4], inv_corrmatrix[4][4];
for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
                corrmatrix[i][j] = 0;
for (int i = 0; i < 4; i++)
        rcolumn[i] = 0;
float const_cos, const_sin, const_sig;
float image_cos, image_sin;
float Q = 0;

bool bit;
for (int i = 0; i <= 15; i++)
{
        if ((kod & 0x8000) > 0)
                bit = true;
        else
                bit = false;
        for (int j = i * 150; j < i * 150 + 150; j++)
        {
                if (bit)
                {
                image_cos = __cosf(2.0f*3.14159f*465000.0f*2.0e-7f*j);
                image_sin = __sinf(2.0f*3.14159f*456000.0f*2.0e-7f*j);
                }
                else
                {
                image_cos = -__cosf(2.0f*3.14159f*465000.0f*2.0e-7f*j);
                image_sin = -__sinf(2.0f*3.14159f*465000.0f*2.0e-7f*j);
                }
                const_cos = __cosf(2.0f*3.14159f*2.0e-7f*(begin + step*tid)*j);
```

```
            const_sin = __sinf(2.0f*3.14159f*2.0e-7f*(begin + step*tid)*j);
            const_sig = __sinf(2.0f*3.14159f*2.0e-7f*467321.36f*j);

            corrmatrix[0][0] += image_cos * image_cos;
            corrmatrix[0][1] += image_cos * image_sin;
            corrmatrix[0][2] += image_cos * const_cos;
            corrmatrix[0][3] += image_cos * const_sin;

            corrmatrix[1][1] += image_sin * image_sin;
            corrmatrix[1][2] += image_sin * const_cos;
            corrmatrix[1][3] += image_sin * const_sin;

            Q += const_sig*const_sig;

            rcolumn[0] += image_cos * const_sig;
            rcolumn[1] += image_sin * const_sig;
            rcolumn[2] += const_cos * const_sig;
            rcolumn[3] += const_sin * const_sig;
        }
        kod = kod << 1;
}
corrmatrix[2][2] = dev_ccsum(begin + step*tid);
corrmatrix[2][3] = dev_cssum(begin + step*tid);
corrmatrix[3][3] = dev_sssum(begin + step*tid);

for (int i = 0; i < 4; i++)
        for (int j = i + 1; j < 4; j++)
            corrmatrix[j][i] = corrmatrix[i][j];

inv(corrmatrix, inv_corrmatrix);
```

```
        mprod(inv_corrmatrix, rcolumn, lcolumn);
        float S = dot_product(rcolumn, lcolumn, 4);
        result[tid] = Q - S;
        tid += blockDim.x * gridDim.x;
}
}
```

Произведем сравнение производительности нахождения минимума такой функции на CPU и на GPU.

Таблица 1. Сравнение производительности GPU/CPU.

Число итераций	Время выполнения на GPU: T_GPU, ms	Время выполнения на CPU: T_CPU, ms	Отношение времен T_CPU/T_GPU
130	0,40	29,78	74,40
260	0,43	59,57	138,45
520	0,57	119,14	208,87
1300	0,82	298,21	363,41
2600	0,83	598,92	721,08
13000	2,30	2984,81	1296,83
26000	3,98	5986,23	1503,02
52000	7,69	12122,57	1577,35
130000	18,13	30154,31	1662,97
520000	70,63	118292,14	1674,91
1300000	175,56	-	-
2600000	350,73	-	-
13000000	1760,69	-	-
14444445	1953,15	-	-

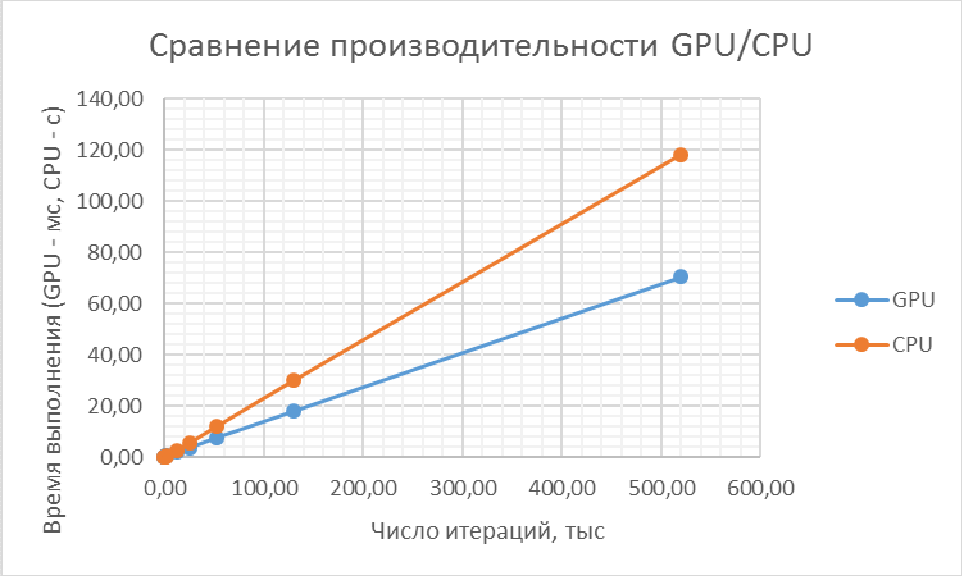

Рисунок 3 - Сравнение времен выполнения минимизации функционала на CPU/GPU.

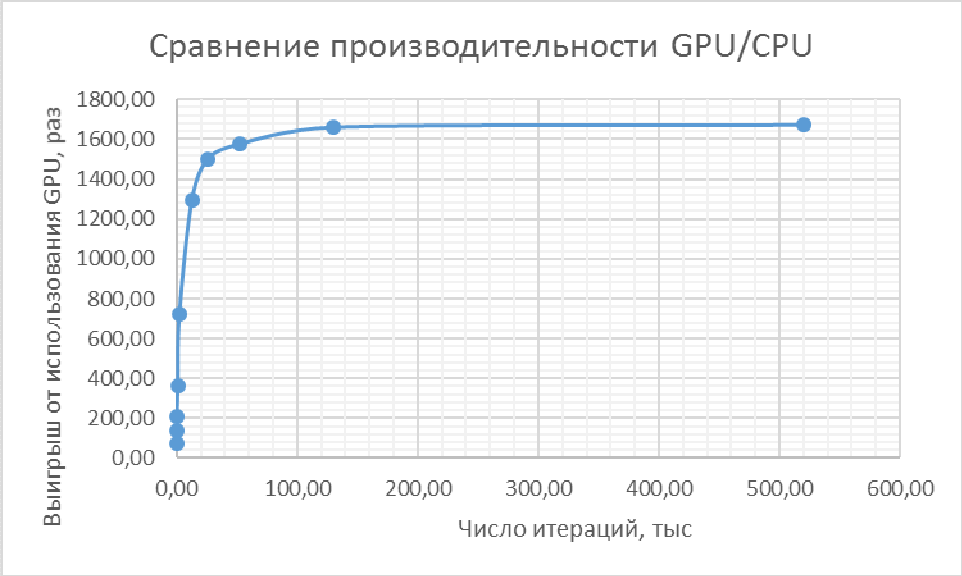

Рисунок 4 - Выигрыш, получаемый от использования GPU в расчетах минимума функционала.

Как показывают графики, использование GPU вместо CPU позволяет существенно ускорить процедуру минимизации искомой функции. Конфигурация оборудования, на которой производилось сравнение производительности GPU/CPU представлена в таблице 2.

Таблица 2 - Характеристики CPU/GPU.

Тип и параметры GPU	Тип и параметры CPU
• тип: i5-4210H • базовая тактовая частота: 2.9 GHz • Максимальная тактовая частота с технологией Turbo Boost: 3.5 GHz	• тип: GTX 860M • Число CUDA ядер: 640 • Частота: Up to 2500 MHz • Производительность CUDA: 5.0

В случае минимизации двумерного функционала, алгоритм представленный выше останется применимым, но с небольшими модификациями. Покажем это на примере минимизации двумерной Schwefel function. Сама функция и ядро представлены ниже.

```
__device__ float Schwefel_2d(float x, float y)
{
    return 2.0f * 418.9829f - x*sin(sqrt(fabs(x))) - y*sin(sqrt(fabs(y)));
}
__global__ void Schwefel_kernel_2d(float *result, uint64_t size, float step, float begin)
{
    uint64_t idx = threadIdx.x + blockIdx.x * blockDim.x;
    uint64_t idy = threadIdx.y + blockIdx.y * blockDim.y;
    if (idx < size && idy < size)
    {
    *(result + size*idx + idy) = Schwefel_2d(begin + step*idx, begin + step*idy);
    }
}
```

Как видно, в данном случае мы воспользовались двойной индексацией, однако значения, как и прежде сохраняем в линейный массив, что позволяет использовать описанную выше процедуру минимизации. Тогда искомое решение представится в виде

*float x = (pos / size)*step + begin;*
*float y = (pos % size)*step + begin;*
где pos – позиция минимального элемента в массиве result.

Еще одно замечание касается параметров запуска ядра Schwefel_kernel_2d. Для двумерной индексации его нужно запускать на сетке, например, фиксирую число нитей вдоль одного направления блока равным 32, найдем значения для числа блоков.
dim3 grids((number + 32 - 1) / 32, (number + 32 - 1) / 32);
dim3 threads(32, 32);
где number - число узлов в сетке вдоль одного направления, по которой производится расчет значений функционала.

Запуск ядра в данном случае будет выглядеть следующим образом:

Schwefel_kernel_2d << <grids, threads >> >(dev_funcvals, number, step, begin);

Описанная в данной статье процедура поиска минимума функционала позволяет получить существенный прирост в скорости работы приложения, осуществляющего оценку параметров сигнала на основе теории оптимального приема, в то же время, решение, получаемое с помощью описанного алгоритма, является истинным глобальным минимумом, что позволяет снизить ошибки, связанные с промахами при применении каких-либо алгоритмов оптимизации. Кроме того, описанный в статье способ минимизации может быть применен и к

другим задачам, возникающим в радиотехнике и требующим для решения минимизацию некоторой функции.

Литература

1. Параллельные вычисления CUDA [электронный ресурс.] URL: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html

2. Джейсон Сандерс, Эдвард Кэндрот «Технология CUDA в примерах», ДМК-Пресс, 2011.