Applied Parallel Computing

parallel-computing.pro

Стационарное уравнение Пуассона

к.т.н. Алексей Ивахненко

Poisson-Dirichlet process

- Задача Дирихле для уравнения Пуассона:

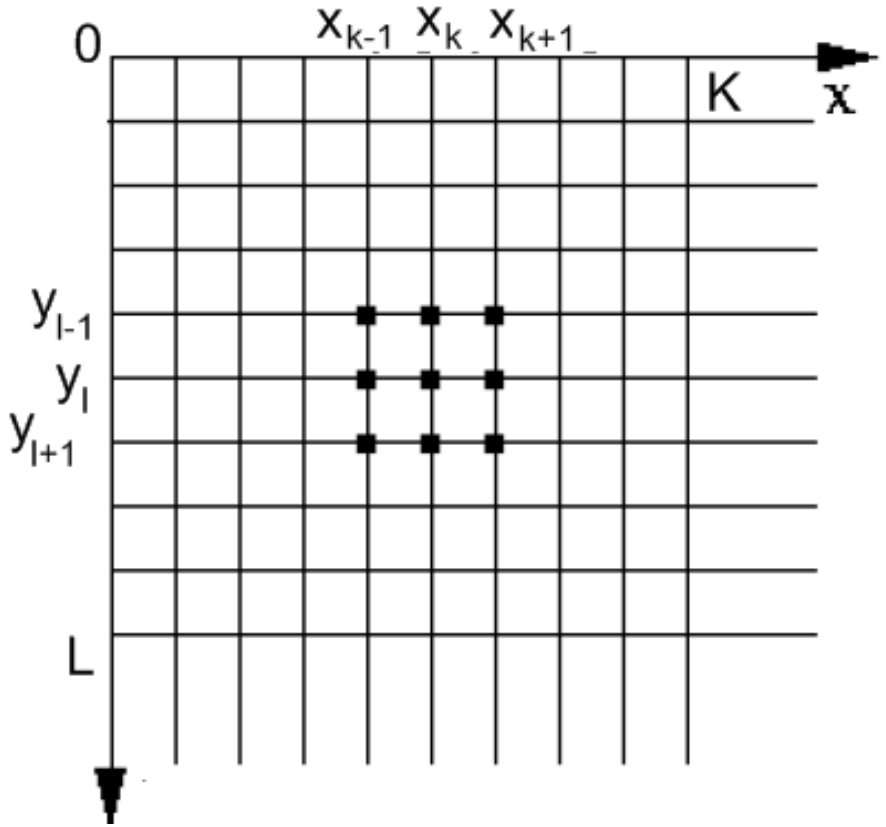
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x, y) \qquad (4)$$

В прямоугольной области

$a \le x \le b,\ c \le y \le d$

- Область делится на равные

сегменты по измерениям x и y



Метод конечных разностей

- После применения метода конечных разностей:

$$U_{k,l} = U_{k,l} - \xi_{k,l} \quad (5)$$

- Где:

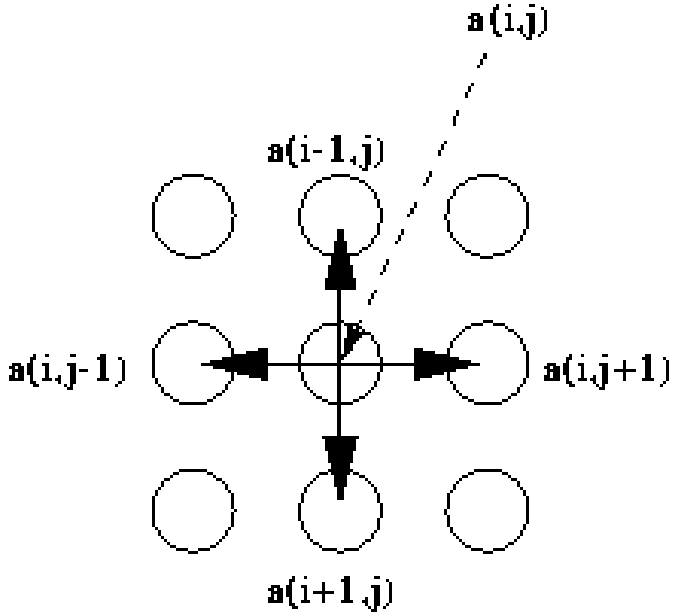
$$U_{k,l} = (U_{k+1,l} + U_{k-1,l} + U_{k,l-1} + U_{k,l+1} - \Delta^2 \rho_{k,l})/4, \quad (6)$$

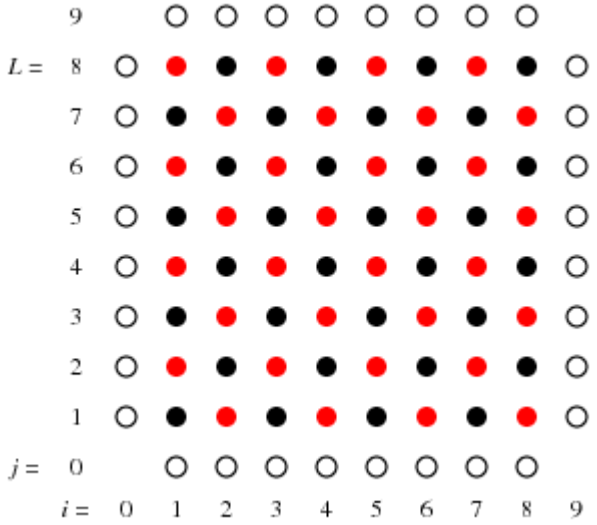
$$\xi_{k,l} = U_{k,l} - (U_{k+1,l} + U_{k-1,l} + U_{k,l-1} + U_{k,l+1} - \Delta^2 \rho_{k,l})/4, \quad (7)$$

ξ - невязка, и она рассчитывается из разницы левой и правой части уравнения

SOR Метод

- Successive over-relaxation (SOR) – метод решения линейных систем уравнений типа Ax=b полученный путем обобщения метода Gauss-Seidel.





SOR Метод

- Если добавить ω в формулу (5) мы получим основное уравнение SOR:

$$U_{k,l}^{new} = U_{k,l}^{old} - \omega\xi \qquad (6)$$

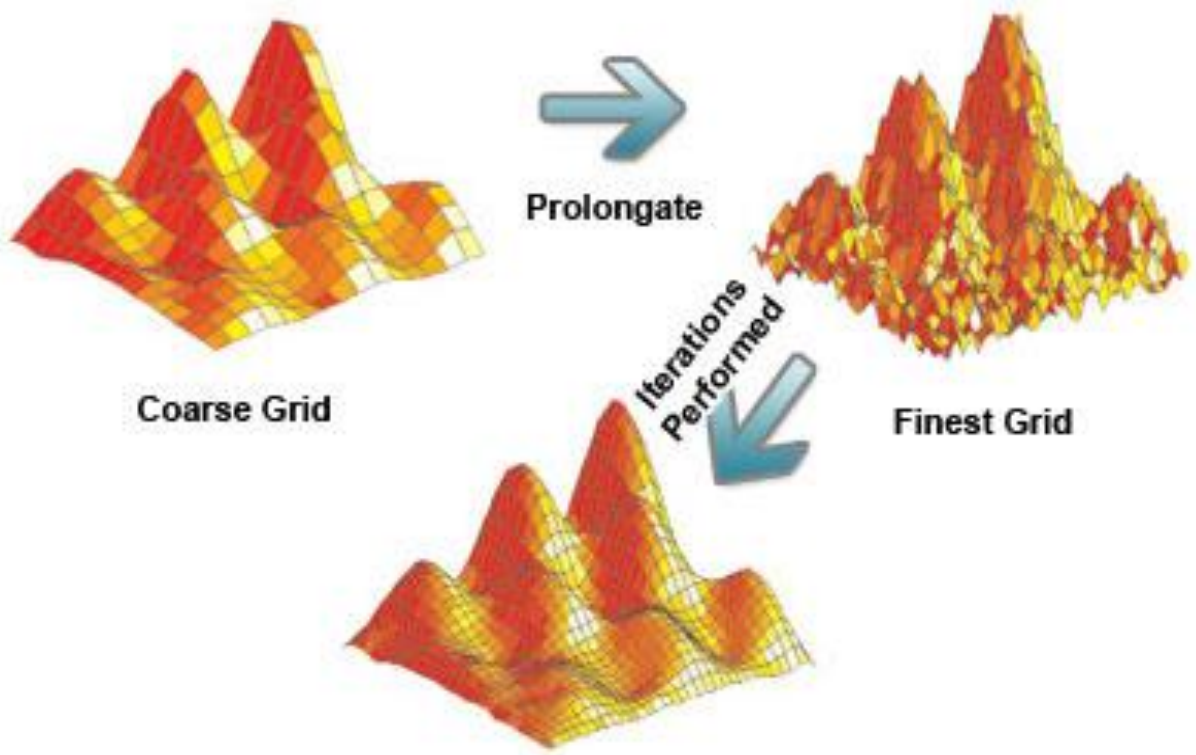
- ω обычно не равно 1
- Для лучшей сходимости выбирают значения между 1 и 1,5

Мультигрид

- Порядок сеток:

Точная <-> Грубая

- Smoothing: уменьшение высокочастотных гармоник
- Restriction: точная сетка -> грубая сетка
- Prolongation: грубая сетка -> точная сетка
- Solver: поиск решения с заданной точностью



Реализация

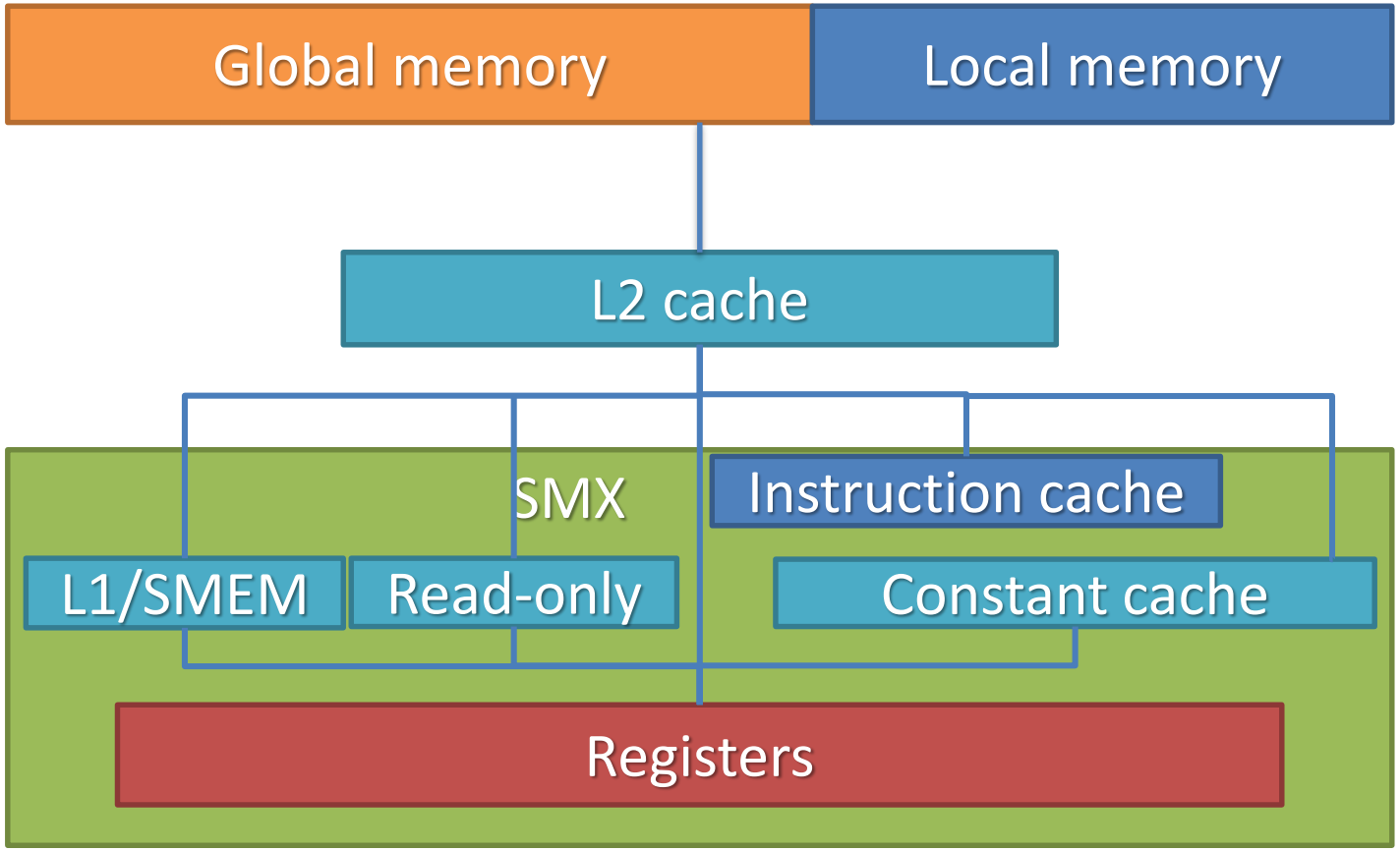
- Входные данные: размер домена задачи, задающее воздействие, количество сеток и заданная величина невязки
- Solver использует SOR с multigrid или без него (управляется дополнительными входными параметрами)
- Вывод приложения: расчетные значения, визуализация в формате BOV (Paraview) или с помощью OpenGL

Hardware

Stream Processors	2496
Core Clock	706MHz
Memory Clock	5.2GHz GDDR5
Memory Bus Width	320-bit
VRAM	5GB
Single Precision	3.52 TFLOPS
Double Precision	1.17 TFLOPS (1/3)
Transistor Count	7.1B
TDP	225W
Manufacturing Process	TSMC 28nm
Architecture	Kepler

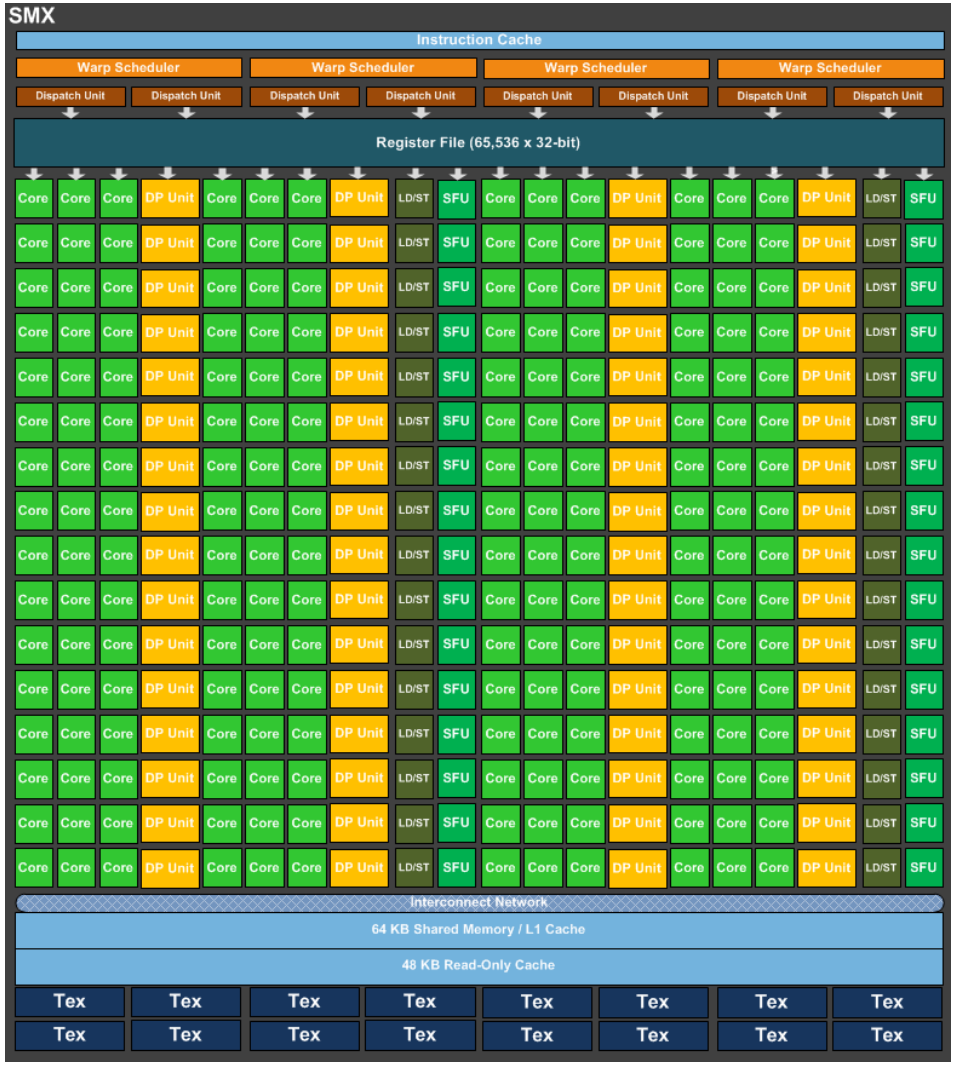
Иерархия памяти

- Управляемая:
  - Глобальная память
  - Разделяемая память
- Частично управляемая:
  - Локальная память
  - Регистры
  - L1 кэш
  - Read-only кэш



SMX: основные компоненты

- SMX:
  - 192 CUDA ядра
  - 64 DP Юнита
  - 32 LD/ST Юнита
  - 32 SFU
  - 64 KB L1 кэш/ разделяемая память
  - 16 текстурных процессоров
  - 48 KB read-only кэш
  - 65536 32-bit регистра
  - 4 планировщика варпов
  - 8 dispatch юнитов



Режимы работы

Solver modes:

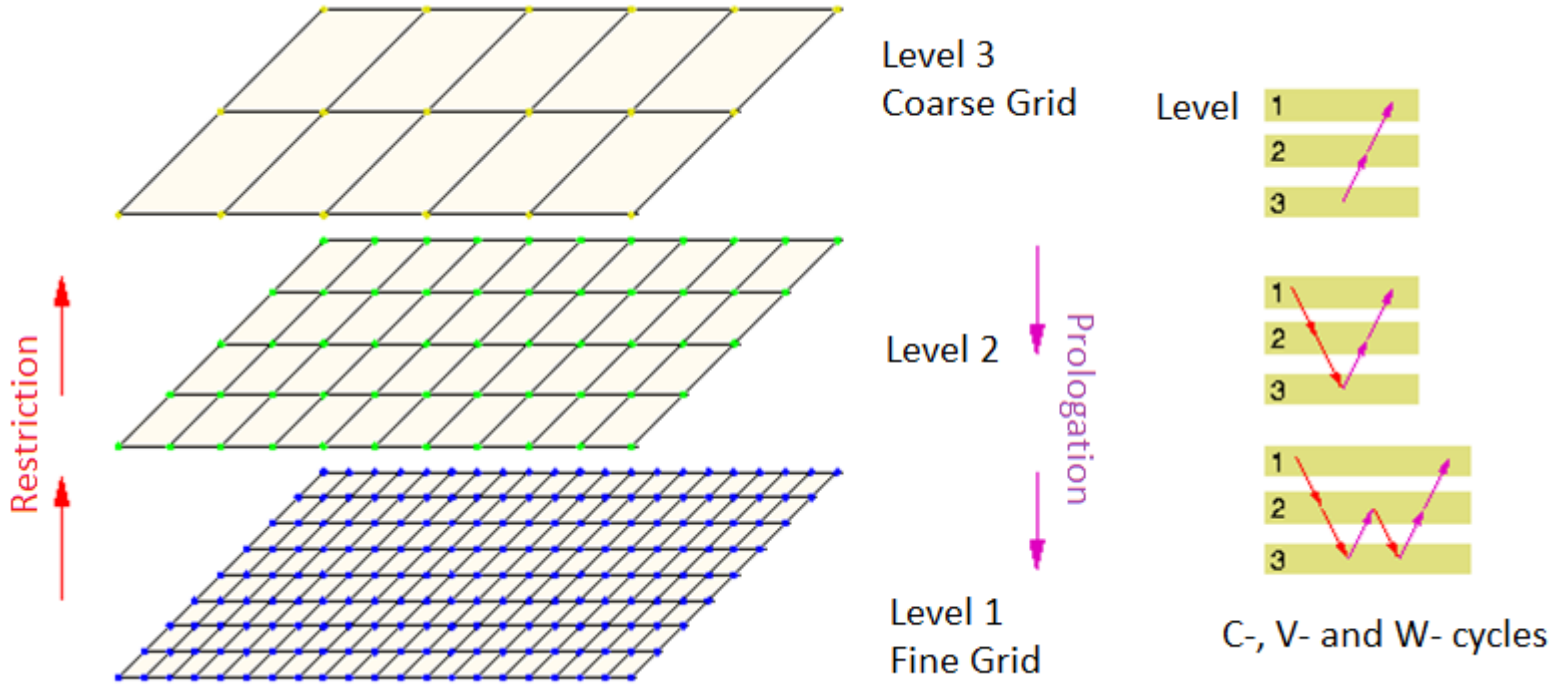
1) SOR only (without multigrid))

2) C – cycle (with multigrid)

3) V – cycle (with multigrid)

4) W – cycle (with multigrid)

Multigrid principles:



Аргументы программы

Example:

```
./MGCUDA x y M N S err Q
```

Arguments:

x,y – размер домена задачи

M,N – количество точек сетки по 'x‘ и 'y'

S – количество уровней multigrid

Err - невязка

Q – режим (0 = SOR, 1 = C-cycle, 2 = V-cycle, 3 = W-cycle)

Исходная производительность

- Mode 1: 389,91 sec (Usual SOR)

```
./MGCUDA 32 32 4096 4096 0 1e-6 0 (Mode 1)
```

- Mode 2: 208,44 sec (Multigrid -> C-cycle)

```
./MGCUDA 32 32 4096 4096 3 1e-6 1 (Mode 2)
```

- Mode 3: 210,8 sec (Multigrid -> V-cycle)

```
./MGCUDA 32 32 4096 4096 3 1e-6 2 (Mode 3)
```

- Mode 4: 257,02 sec (Multigrid -> W-cycle)

```
./MGCUDA 32 32 4096 4096 3 1e-6 3 (Mode 4)
```

NVIDIA Visual Profiler

Low Compute / Memcpy Efficiency [76,307 ms / 83,765 ms = 0,911]
The amount of time performing compute is low relative to the amount of time required for memcpy. More...

Low Memcpy/Compute Overlap [0 ns / 76,307 ms = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

Low Kernel Concurrency [0 ns / 76,307 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [1,008 MB/s avg, for memcpys accounting for 0% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

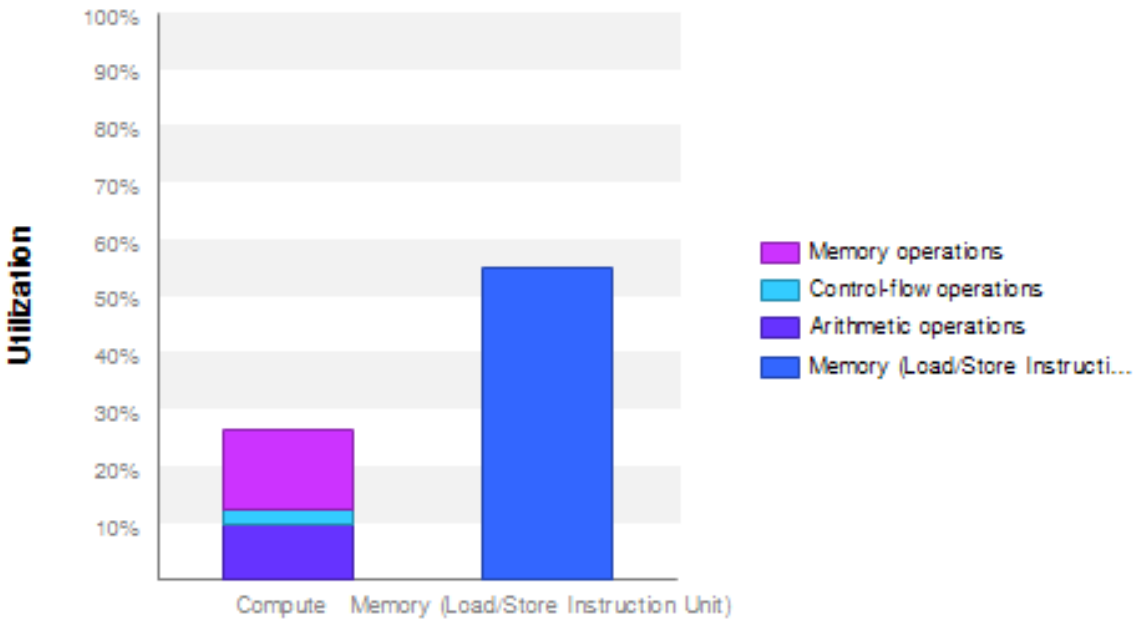
Low Memcpy Overlap [0 ns / 34,044 ms = 0%]
The percentage of time when two memory copies are being performed in parallel is low. More...

Low Compute Utilization [76,307 ms / 17,199 s = 0,4%]
The multiprocessors of one or more GPUs are mostly idle. More...

Compute Utilization
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.

NVIDIA Visual Profiler

- Profiled GPU utilization and effective memory bandwidths:



Warp Execution Efficiency	65,2	%
Global Memory Store Efficiency	12,5	%
Global Memory Load Efficiency	12,5	%
Warp Non-Predicated Execution Efficiency	61,9	%
L2 throughput (Texture Reads)	0	GB/s
Texture Cache Transactions	0	times

Шаги оптимизации

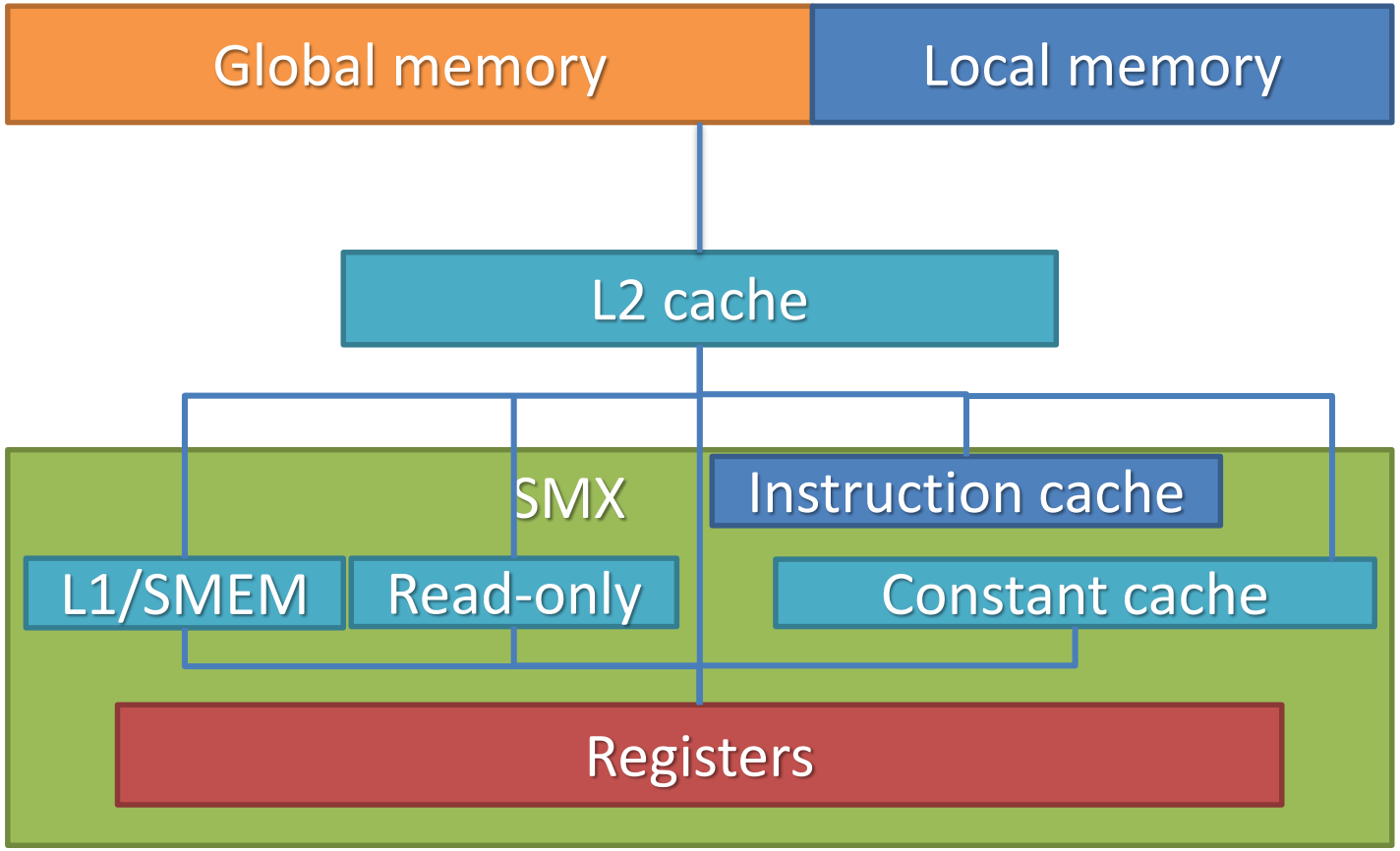
- Read-only кэш
- Оптимизация кода ядра
- AoS to SoA
- Поворот блоков
- Swap указателей вместо явного копирования
- Custom memory allocator в Thrust

Все оптимизации были проверены на следующих параметрах:

- `./MGCUDA 32 32 4096 4096 0 1e-6 0 (Mode 1)`
- `./MGCUDA 32 32 4096 4096 3 1e-6 1 (Mode 2)`
- `./MGCUDA 32 32 4096 4096 3 1e-6 2 (Mode 3)`
- `./MGCUDA 32 32 4096 4096 3 1e-6 3 (Mode 4)`

Read-only кэш

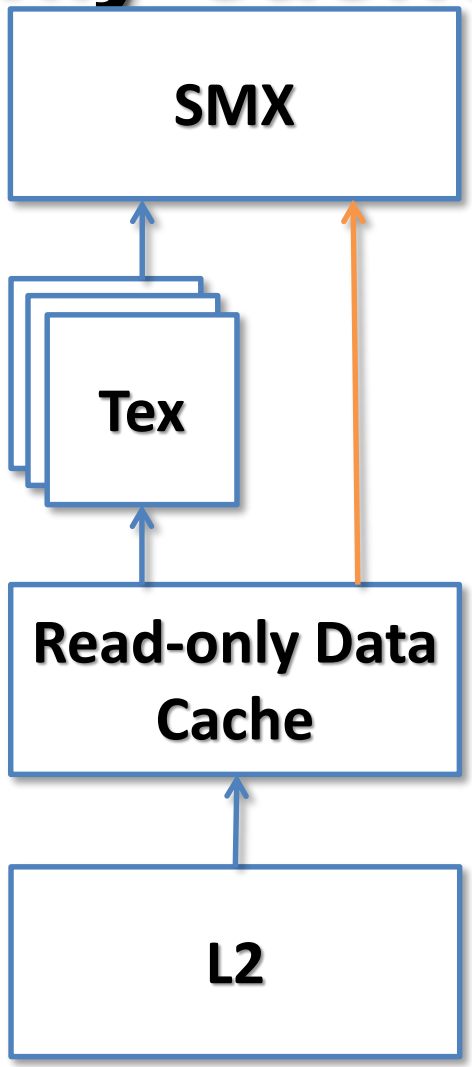
- 48 KB на SMX
- Отдельный от L1 кэша
- Латентность аналогична L1
- Кэширование 1D, 2D и 3D arrays
- Встроенные алгоритмы интерполяции



Read-only cache

- `const __restrict`
- В обход текстурного блока
- Кэшированный доступ к глобальной памяти
- Не требует явного использования текстур

- Для чего?
  - Отдельно от smem/L1 кэша
  - Высокая пропускная способность к L2
  - Невыровненный доступ
  - 1D, 2D и 3D массивы



const __restrict

- Указать `const __restrict` для параметров
- Компилятор сгенерирует инструкцию загрузки через read only кэш

```
__global__
void saxpy (float x, float y,
            const float * __restrict input,
            float * output){
  size_t offset = threatIdx.x +
                  (blockIdx.x * blockDim.x);

  // Compiler will automatically use texture
  // for 'input'
  output[offset] = (input[offset] * x) + y;
}
```

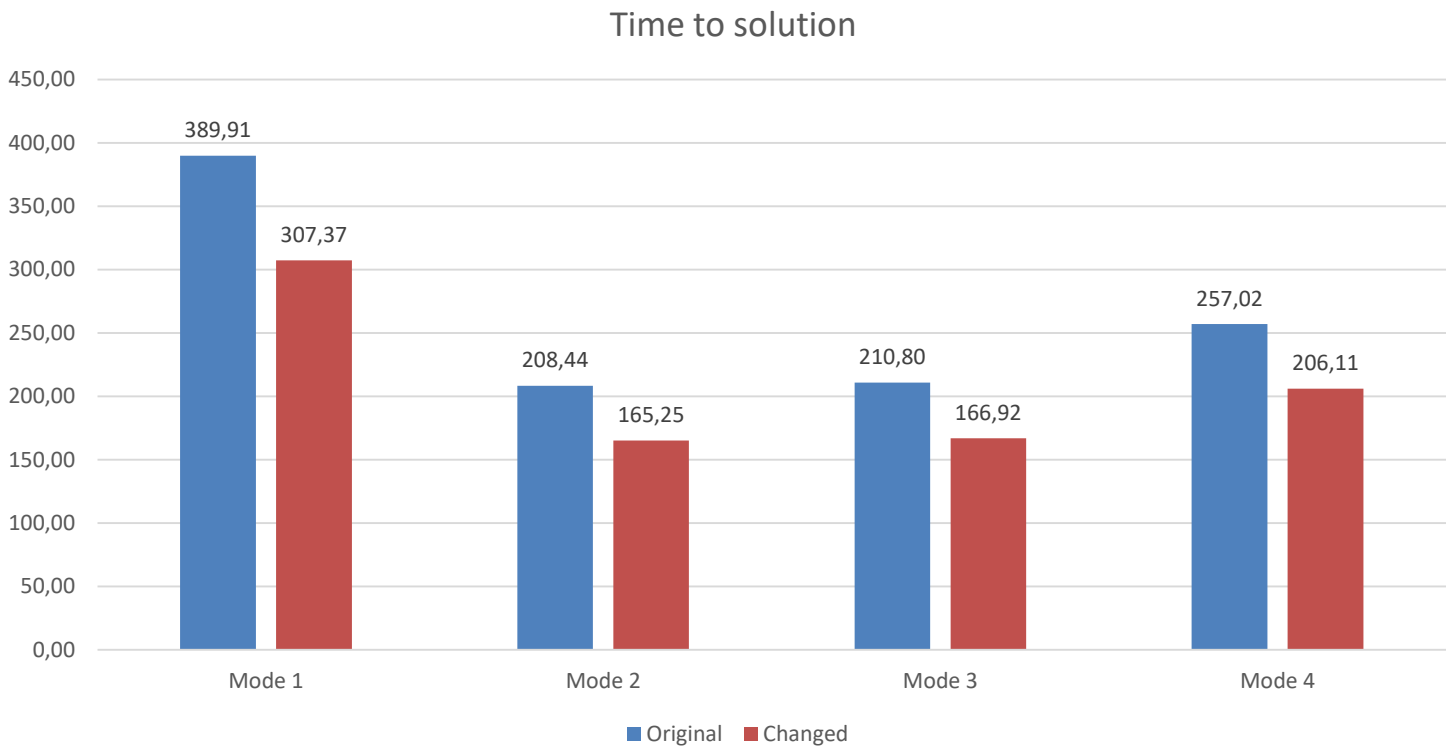
Сравнение кода

```
__global__ void calculate_residual(float dx,
float dy, int M, int N, float* x, float*
x_old)
…
__global__ void project_up(int M_s, int N_s,
float* x_s, float* x_d, int M_d, int N_d)
…

__global__ void project_down(int M_s, int N_s,
float* x_s, float* x_d, int M_d, int N_d)
…

__global__ void
calculate_single_poisson_iteration(int M, int
N, float dx, float dy, float* x,  float* x0,
float _w, int rb)
```

```
__global__ void calculate_residual(float dx,
float dy, int M, int N, const float*
__restrict__ x, float* x_old)
…
__global__ void project_up(int M_s, int N_s,
const float* __restrict__ x_s, float* x_d, int
M_d, int N_d
…
__global__ void project_down(int M_s, int N_s,
const float* __restrict__ x_s, float* x_d, int
M_d, int N_d)
…
__global__ void
calculate_single_poisson_iteration(int M, int
N, float dx, float dy, float* x, const float*
__restrict__ x0, float _w, int rb)
```

Итог

- Ускорение в 1.27x, в зависимости от режима работы.



Optimized version works:
389.91/307.37 = 1.27x faster (Mode 1)
208.44/165.25 = 1.26x faster (Mode 2)
210.80/166.92 = 1.26x faster (Mode 3)
257.02/206.11 = 1.25x faster (Mode 4)

Оптимизация кода ядра

- Исходная программа простаивает 50% GPU ядра из-за индексов нитей:

```
…
calculate_single_poisson_iteration<<<blocks, threads_1>>>(M, N, dx, dy, x, x0, _w, 0);
calculate_single_poisson_iteration<<<blocks, threads_1>>>(M, N, dx, dy, x, x0, _w, 1);
…
__global__ void calculate_single_poisson_iteration(int M, int N, float dx, float dy,
float* x,  float* x0, float _w, int rb){
…
int i = __umul24(blockIdx.x ,blockDim.x) + threadIdx.x;
int j = __umul24(blockIdx.y ,blockDim.y) + threadIdx.y;
…
if( (i>0)&&(i<M-1)&&(j>0)&&(j<N-1) )
    if((i + j) % 2 == rb )
    {
        float resid=(-x0[I2(i,j)] + (x[I2(i-1,j)]+x[I2(i+1,j)])*dxdx+
                    (x[I2(i,j+1)]+x[I2(i,j-1)])*dydy)/c;
        x[I2(i,j)]=x[I2(i,j)]*(1.0-_w)+_w*(resid);     //calculate SOR
    }
…
```

Сравнение кода

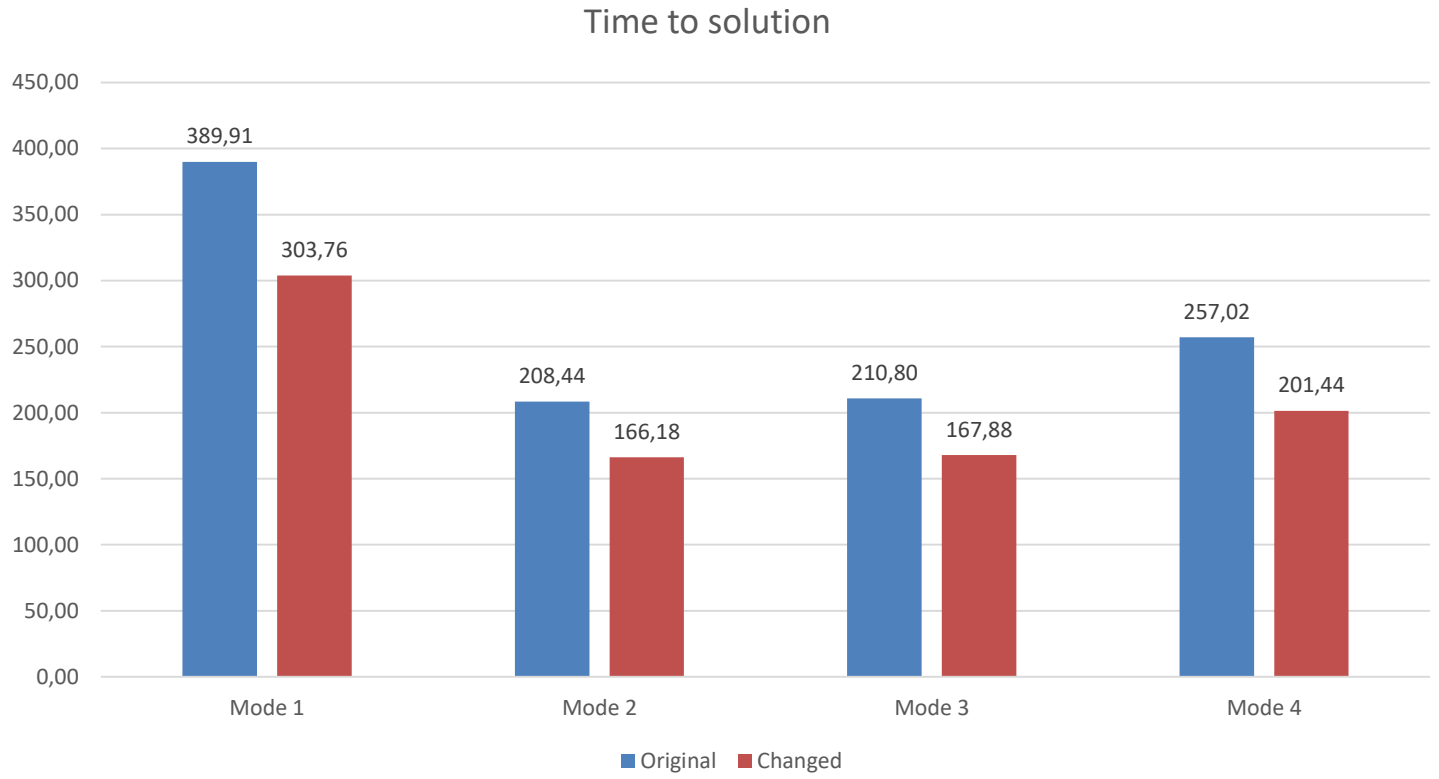
```
calculate_single_poisson_iteration<<<blocks, threads_1>>>(M, N, dx, dy, x, x0, _w, 0);
calculate_single_poisson_iteration<<<blocks, threads_1>>>(M, N, dx, dy, x, x0, _w, 1);
…
```

```
__global__ void calculate_single_poisson_iteration(int M, int N, float dx, float dy, float* x,  float* x0, float _w, int rb){
…
int i = __umul24(blockIdx.x ,blockDim.x) + threadIdx.x;
int j = __umul24(blockIdx.y ,blockDim.y) + threadIdx.y;
…
if( (i>0)&&(i<M-1)&&(j>0)&&(j<N-1) )
          if((i + j) % 2 == rb ){
float resid=(-x0[I2(i,j)] + (x[I2(i-1,j)]+x[I2(i+1,j)])*dxdx+(x[I2(i,j+1)]+x[I2(i,j-1)])*dydy)/c;
x[I2(i,j)]=x[I2(i,j)]*(1.0-_w)+_w*(resid);     //calculate SOR
}
```

```
calculate_single_poisson_iteration<<<blocks, threads>>>(M, N, dx, dy, x, x0, _w, 1, 1);
calculate_single_poisson_iteration<<<blocks, threads>>>(M, N, dx, dy, x, x0, _w, 2, 2);
calculate_single_poisson_iteration<<<blocks, threads>>>(M, N, dx, dy, x, x0, _w, 1, 2);
calculate_single_poisson_iteration<<<blocks, threads>>>(M, N, dx, dy, x, x0, _w, 2, 1);
…
__global__ void calculate_single_poisson_iteration(int M, int N, float dx, float dy, float* x, float*  x0, float _w, int si, int sj){
…
float res;
int i = (__umul24(blockIdx.x ,blockDim.x) + threadIdx.x)*2 + si;
int j = (__umul24(blockIdx.y ,blockDim.y) + threadIdx.y)*2 + sj;
…
if( (i>0)&&(i<M-1)&&(j>0)&&(j<N-1) )
{
float resid=(-x0[I2(i,j)] + (x[I2(i-1,j)]+x[I2(i+1,j)])*dxdx+(x[I2(i,j+1)]+x[I2(i,j-1)])*dydy)/c;
res=x[I2(i,j)]*(1.0-_w)+_w*(resid);

x[I2(i,j)] = res;
}
```

Итог

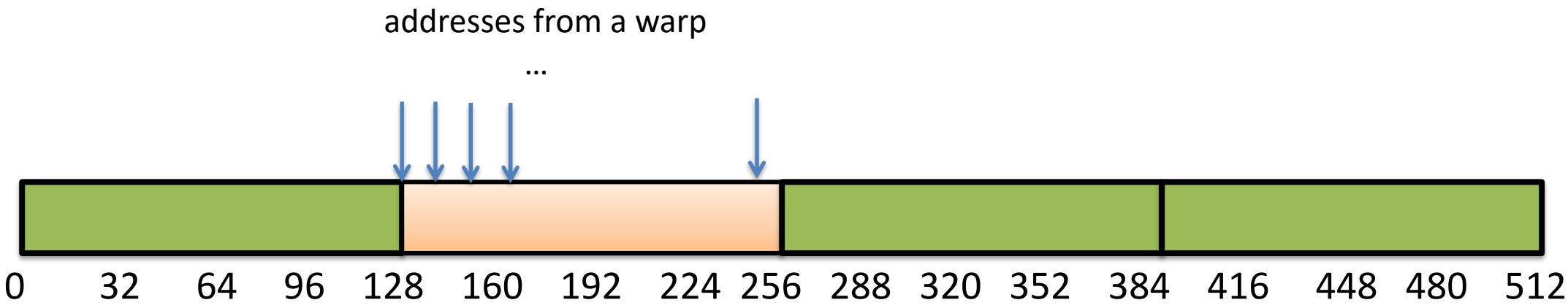
- Ускорение в 1.28x, в зависимости от режима работы



Optimized version works:
389.91/303.76 = 1.28x faster (Mode 1)
208.44/166.18 = 1.25x faster (Mode 2)
210.80/167.88 = 1.26x faster (Mode 3)
257.02/201.44 = 1.28x faster (Mode 4)

AoS to SoA

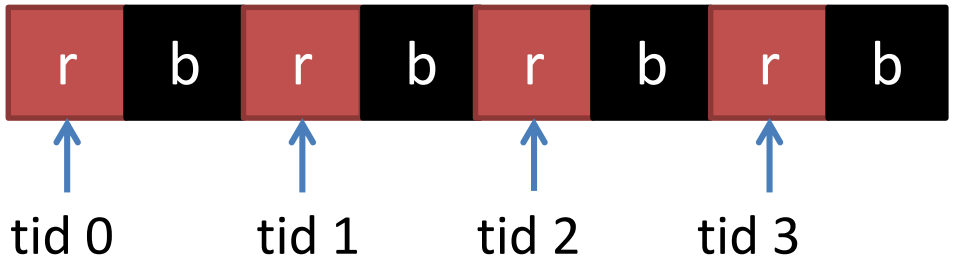
- **Оптимальный доступ к памяти:**
  - Запрос 32 последовательных 4-байтных слов, 1 слово на 1 нить варпа, выровнены по 128-байт
- **Запросы в глобальную память попадают в 1 транзакцию**
  - Варп загружает 128 байт 1-й кэш линией
  - Использование шины: 100%

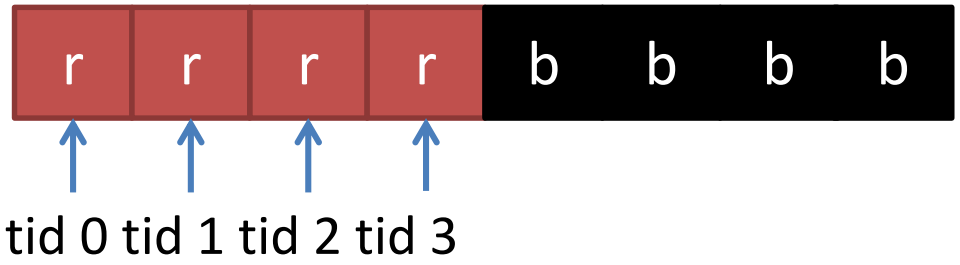


AoS to SoA

```
struct A{
    float r;
    float b;
};
struct A myArray[n]
```

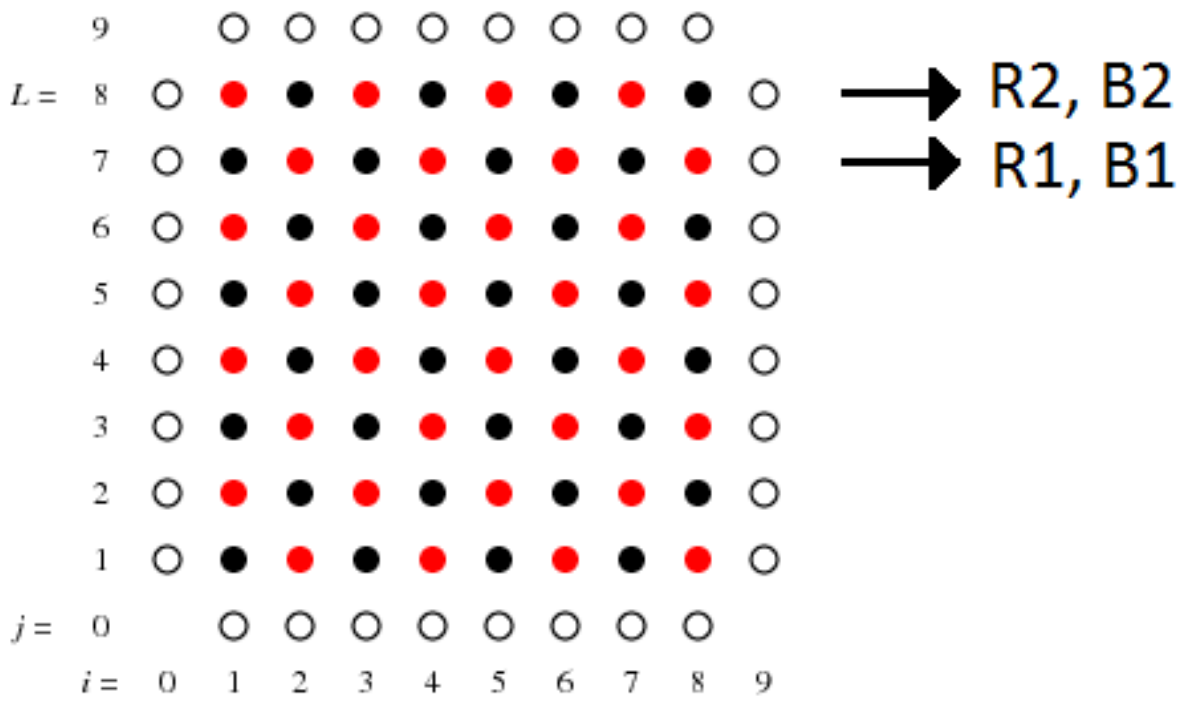
```
struct A{
    float r [n];
    float b [n];
};
struct A myArray
```





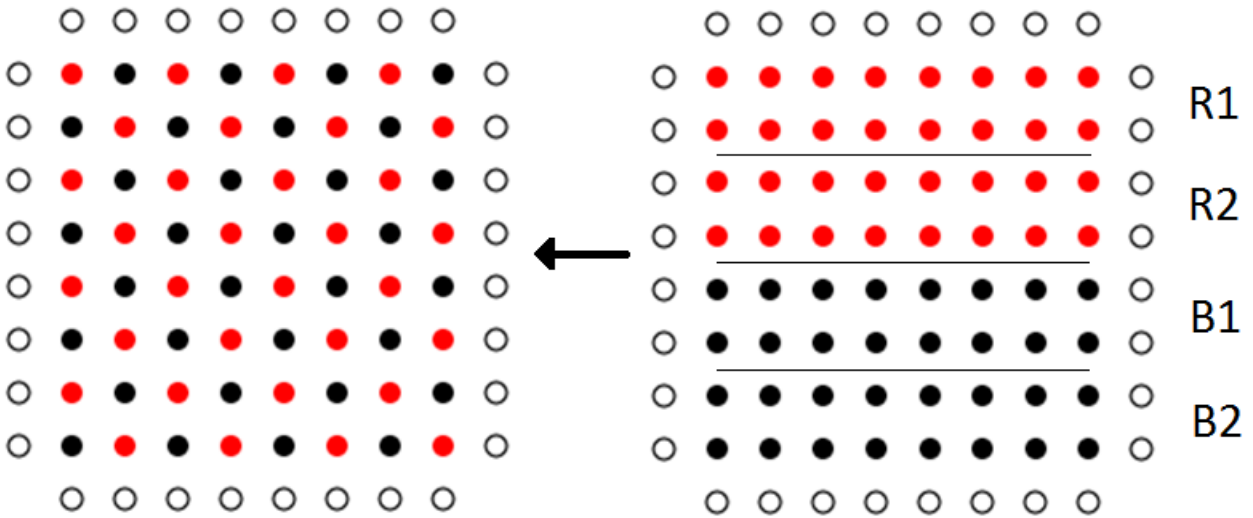
AoS to SoA

- Разделить 1 массив на 4 индивидуальных массива R1, R2, B2, B2. Доступ к элементам каждого из этих массивов будет оптимальным



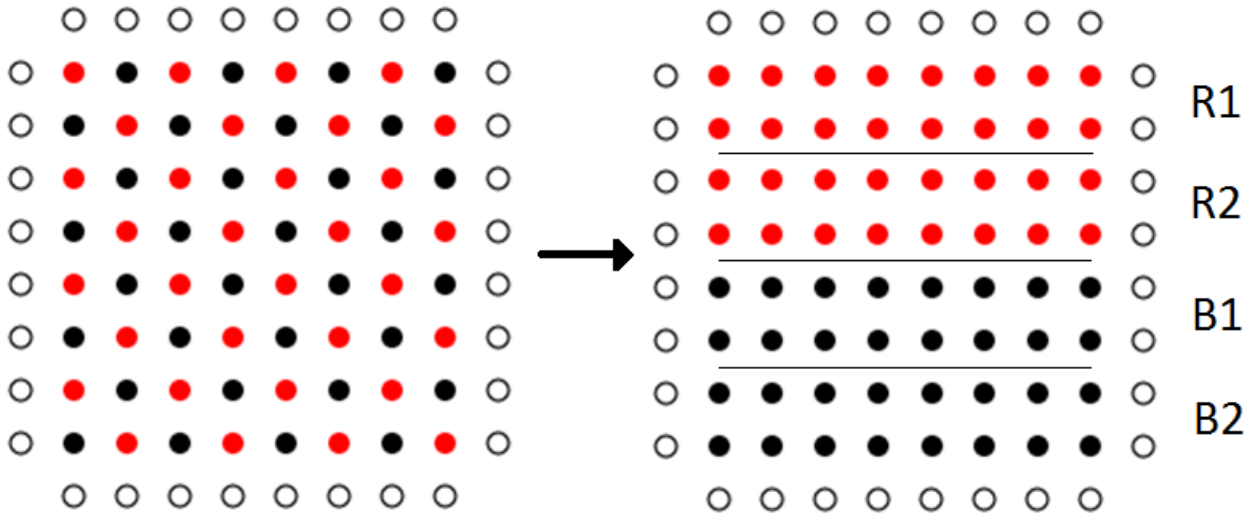
Сравнение кода

```
#define I2(i,j) (j)*(Y)+(i)
#define I2_r(i,j) (j)*(Y2)+(i)

void Make_AOS_order (int M, int N, float* x, float* x_r1, float* x_r2, float* x_b1, float* x_b2){
   int Y = M;
   int Y2 = Y/2;
      for (int j=0; j<(N/2); j++)
      {
         for (int i=0; i<(M/2); i++)
         {
            x[I2(i*2,   j*2)] = x_r2[I2_r(i,j)];
            if(i<((M-1)/2))

                  x[I2(i*2+1, j*2)]   = x_b2[I2_r(i,j)];
            else
                  x[I2(i*2+1, j*2)] = 0.0f;
            if(j<((N-1)/2))
                  x[I2(i*2,   j*2+1)] = x_b1[I2_r(i,j)];
            else
                  x[I2(i*2,   j*2+1)] = 0.0f;
            if ((i<((M-1)/2)) && j<((N-1)/2))
                  x[I2(i*2+1, j*2+1)] = x_r1[I2_r(i,j)];
            else
                  x[I2(i*2+1, j*2+1)] = 0.0f;
         }
      }
}
```



Сравнение кода

```
#define I2(i,j) (j)*(Y)+(i)
#define I2_r(i,j) (j)*(Y2)+(i)

void Make_SOA_order (int M, int N, float* x, float* x_r1, float* x_r2, float* x_b1, float* x_b2 ){
int Y = M;
int Y2 = Y/2;
   for (int j=0; j<(N/2); j++){
      for (int i=0; i<(M/2); i++){
            x_r2[I2_r(i,j)] = x[I2(i*2,   j*2)];
            if(i<((M-1)/2))
                  x_b2[I2_r(i,j)] = x[I2(i*2+1, j*2)]
            else
                  x_b2[I2_r(i,j)] = 0.0f;
            if(j<((N-1)/2))
                  x_b1[I2_r(i,j)] = x[I2(i*2,   j*2+1
            else
                  x_b1[I2_r(i,j)] = 0.0f;
            if ( (i<((M-1)/2)) && j<((N-1)/2) )
                  x_r1[I2_r(i,j)] = x[I2(i*2+1, j*2+1)];
            else
                  x_r1[I2_r(i,j)] = 0.0f;
            }
      }
}
```

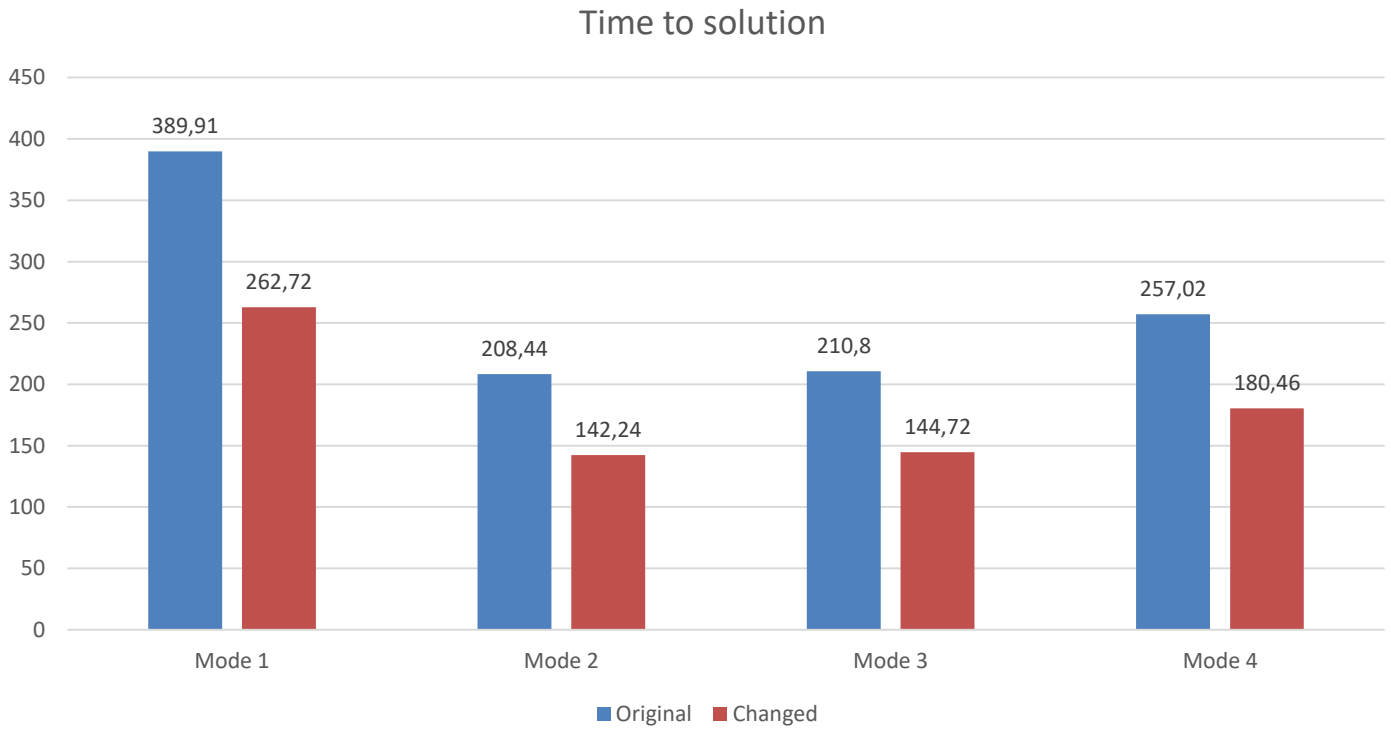


Сравнение кода

- `__global__ void calculate_single_poisson_iteration_r1(int M, int N, float dx, float dy, float* r1, float* r2, float* b1, float* b2, float* x0, float _w)`
- `__global__ void calculate_single_poisson_iteration_r2(int M, int N, float dx, float dy, float* r1, float* r2, float* b1, float* b2, float* x0, float _w)`
- `__global__ void calculate_single_poisson_iteration_b1(int M, int N, float dx, float dy, float* r1, float* r2, float* b1, float* b2, float* x0, float _w)`
- `__global__ void calculate_single_poisson_iteration_b2(int M, int N, float dx, float dy, float* r1, float* r2, float* b1, float* b2, float* x0, float _w)`

Итог

- Ускорение в 1.48x, в зависимости от режима работы



Optimized version works:
389.91/262.72 = 1.48x faster (Mode 1)
208.44/142.24 = 1.47x faster (Mode 2)
210.80/144.72 = 1.46x faster (Mode 3)
257.02/180.46 = 1.42x faster (Mode 4)

Поворот блоков

A(0,0)	A(1,0)	A(2,0)	A(3,0)
A(0,1)	A(1,1)	A(2,1)	A(3,1)
A(0,2)	A(1,2)	A(2,2)	A(3,2)
A(0,3)	A(1,3)	A(2,3)	A(3,3)

X direction

A(0,0)	A(1,0)	A(2,0)	A(3,0)
A(0,1)	A(1,1)	A(2,1)	A(3,1)
A(0,2)	A(1,2)	A(2,2)	A(3,2)
A(0,3)	A(1,3)	A(2,3)	A(3,3)

Y direction

A(0,0)	A(1,0)	A(2,0)	A(3,0)
A(0,1)	A(1,1)	A(2,1)	A(3,1)
A(0,2)	A(1,2)	A(2,2)	A(3,2)
A(0,3)	A(1,3)	A(2,3)	A(3,3)

Сравнение кода

- Original version:

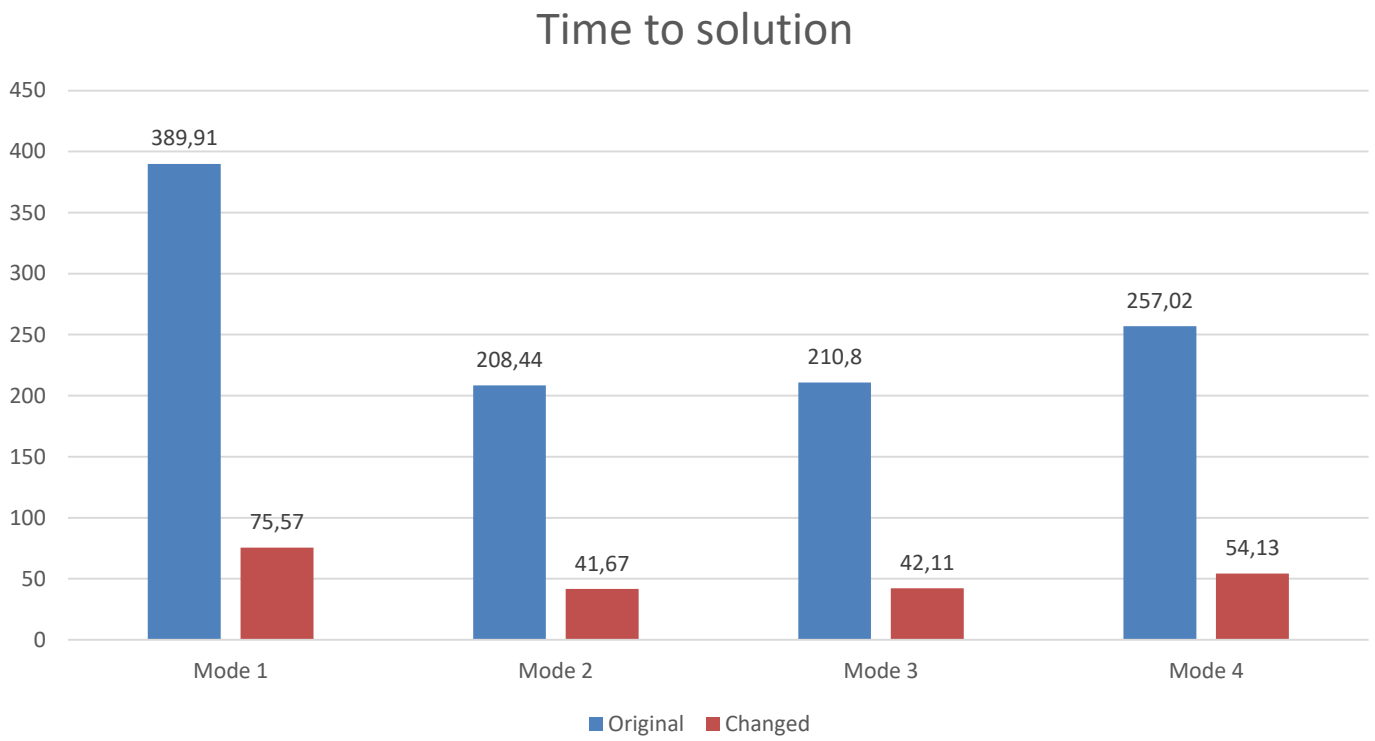
```
#define I2(i,j) (i)*(Y)+(j)
//calculating index
#define I2_1(i,j) (i)*(Z)+(j)
//calculating index
…
#define BLOCK_SIZE_x 32
#define BLOCK_SIZE_y (512 / BLOCK_SIZE_x)
…
int Y=N;                          //for macros
definition
int i = __umul24(blockIdx.x ,blockDim.x)
+ threadIdx.x;
int j = __umul24(blockIdx.y ,blockDim.y)
+ threadIdx.y;
```

- Optimized verison:

```
#define I2(i,j) (j)*(Y)+(i)
//calculating index
#define I2_1(i,j) (j)*(Z)+(i)
//calculating index
…
#define BLOCK_SIZE_x 128
#define BLOCK_SIZE_y (512 / BLOCK_SIZE_x)
…
int Y=M; //for macros definition
int i = __umul24(blockIdx.x ,blockDim.x)
+ threadIdx.x;
int j = __umul24(blockIdx.y ,blockDim.y)
+ threadIdx.y;
```

Итог

- Ускорение в 5.16x, в зависимости от режима работы



Optimized version works:
389.91/75.57 = 5.16x faster (Mode 1)
208.44/41.67 = 5.00x faster (Mode 2)
210.80/42.11 = 5.01x faster (Mode 3)
257.02/54.13 = 4.75x faster (Mode 4)

cudaMemcpy -> pointers swap

```
…
cudaMemcpy(x_old, x, sizeof(float)*(M)*(N), cudaMemcpyDeviceToDevice);
dim3 threads_1( BLOCK_SIZE_x, BLOCK_SIZE_y );

calculate_single_poisson_iteration<<<blocks, threads_1>>>(M, N, dx, dy, x, x0, _w, 0);
calculate_single_poisson_iteration<<<blocks, threads_1>>>(M, N, dx, dy, x, x0, _w, 1);

calculate_residual<<<blocks, threads_1>>>(dx, dy, M, N, x, x_old);

thrust::device_ptr<float> Max_Ptr = thrust::device_pointer_cast(x_old);
float Max = 0.0f;
Max = thrust::reduce(Max_Ptr, Max_Ptr + (M)*(N),  0.0f, thrust::maximum<float>());

return(Max);
…
```

Сравнение кода

```
float one_complete_step(dim3 blocks, dim3
threads, float dx, float dy, int M, int N,
float* x, float* x_old, float* x0, float
_w){
cudaMemcpy(x_old, x, sizeof(float)*(M)*(N),
cudaMemcpyDeviceToDevice); //copy: x->x_old
…

eps=one_complete_step(blocks, threads,
x/(float)M[ss], y/(float)N[ss], M[ss],
N[ss], &P_GPU[shift], &Pold_GPU[shift],
&D_GPU[shift], sor_w[ss]);
```

```
float one_complete_step(dim3 blocks, dim3
threads, float dx, float dy, int M, int N,
float* x, float* x_old, float* x0, float _w){



…
swap_arrays(ss);
eps=one_complete_step(blocks, threads,
x/(float)M[ss], y/(float)N[ss], M[ss], N[ss],
P_GPU[ss], Pold_GPU[ss], &D_GPU[shift],
sor_w[ss]);
…
void swap_arrays (int s)
{
    float* temp   = P_GPU[s];
    P_GPU[s]      = Pold_GPU[s];
    Pold_GPU[s]   = temp;
}
```

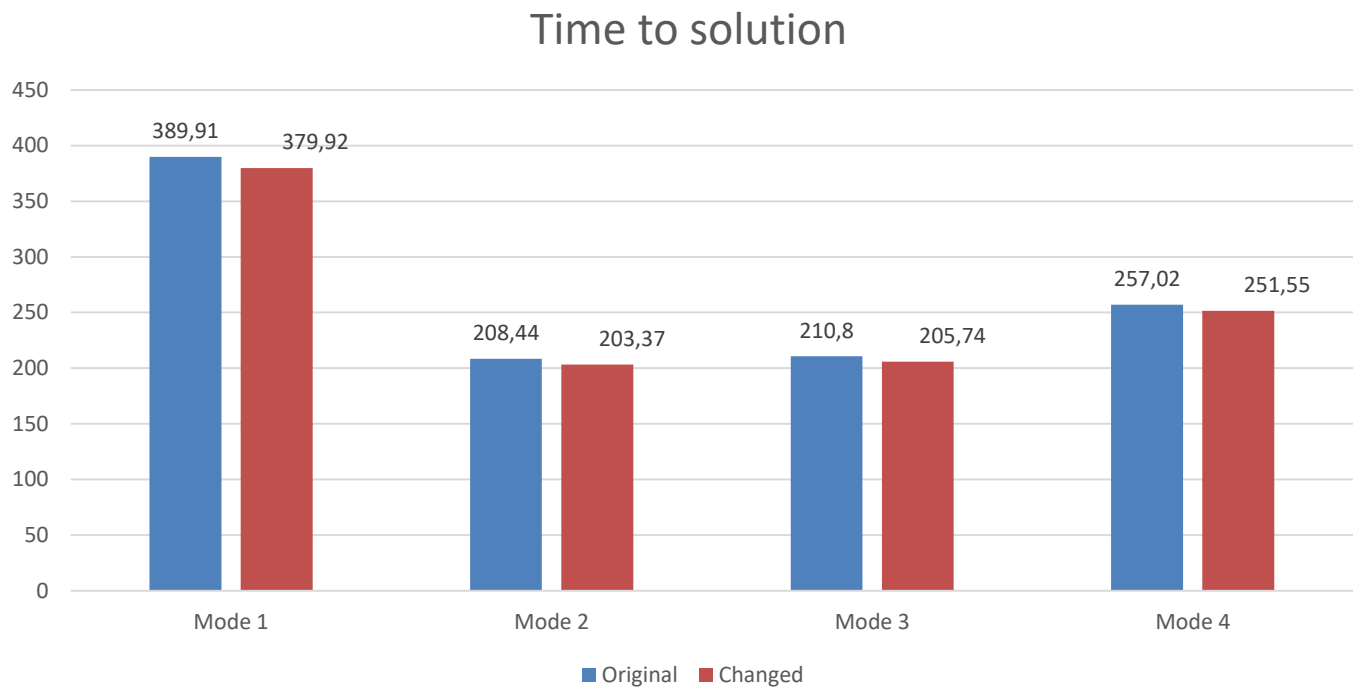
Сравнение кода

```
P=(float*)allocate(total_size,P);
P_old=(float*)allocate(total_size,P_old);
P_GPU=allocate_device_mem(total_size,P_GPU);
Pold_GPU=allocate_device_mem(total_size,
Pold_GPU);
```

```
P=(float**)malloc((S+1)*sizeof(float*));
POLD=(float**)malloc((S+1)*sizeof(float*));
P_GPU=(float**)malloc((S+1)*sizeof(float*));
Pold_GPU=(float**)malloc((S+1)*sizeof(float*
));

for(int i=0;i<=S;i++){

P[i]=(float*)malloc((M[i]*N[i])*sizeof(float
));
POLD[i]=(float*)malloc((M[i]*N[i])*sizeof(fl
oat));
cudaMalloc(&P_GPU[i],M[i]*N[i]*sizeof(float)
);
cudaMalloc(&Pold_GPU[i],M[i]*N[i]*sizeof(flo
at));
}
```

Итог

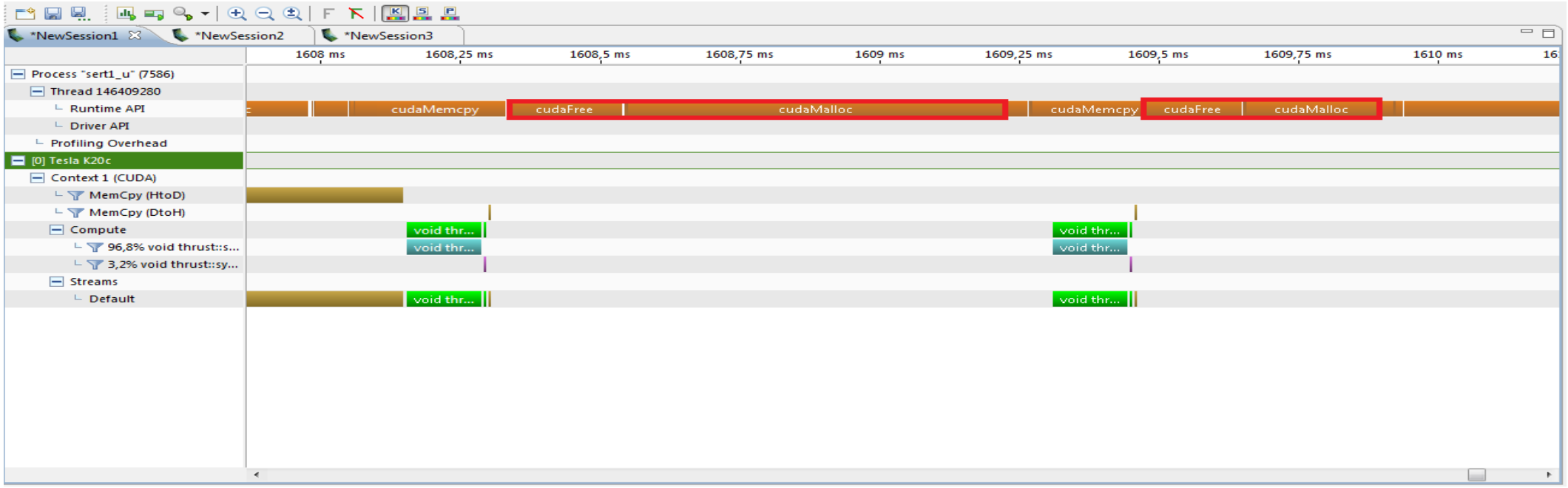
- Постоянная экономия 5-10 секунд (3% в исходном варианте, 15% в финальной версии)



Optimized version works:
389.91/379.92 = 1.03x faster (Mode 1)
208.44/203.37 = 1.02x faster (Mode 2)
210.80/205.74 = 1.02x faster (Mode 3)
257.02/251.55 = 1.02x faster (Mode 4)

Custom memory allocator in Thrust

- Мы используем thrust::reduce для нахождения максимального значения на каждом шаге. Алгоритм вызывает cudaMalloc/cudaFree для управления промежуточным буфером. С использованием custom memory allocator можно сохранять буфер между итерациями.



Сравнение кода

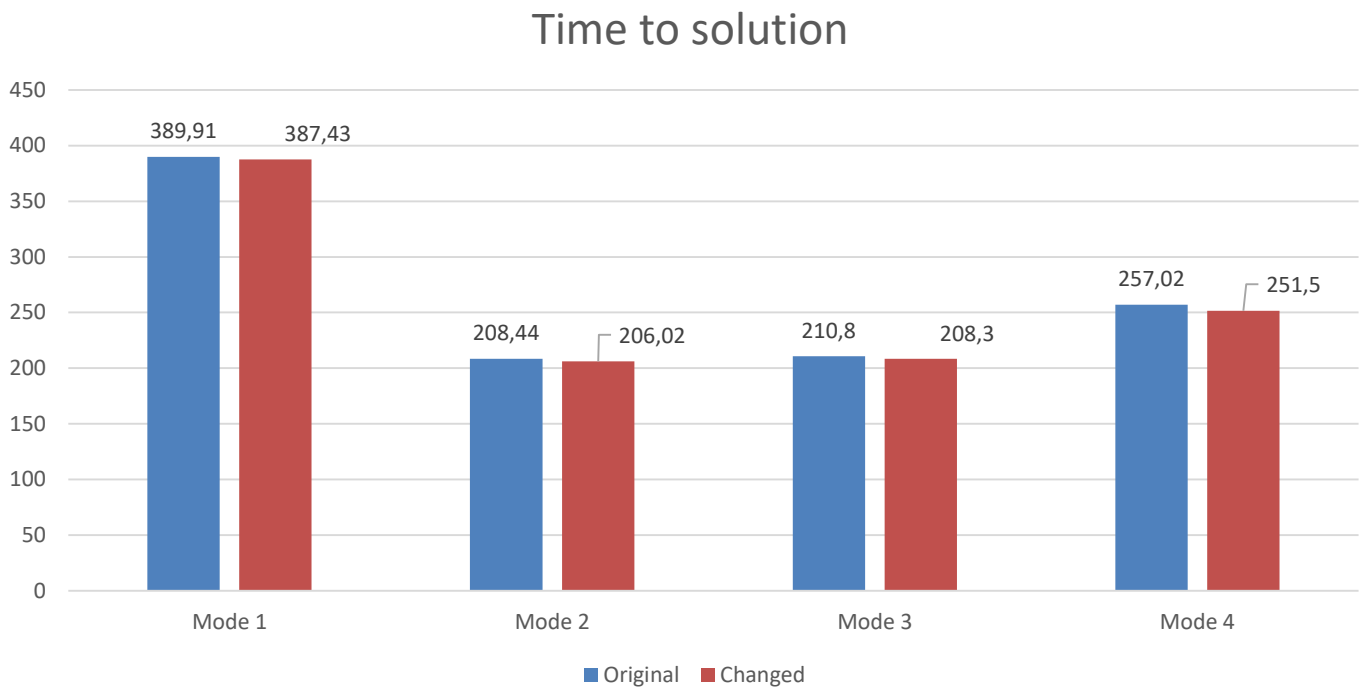
```
Max = thrust::reduce(Max_Ptr,
Max_Ptr + (M)*(N),  0.0f,
thrust::maximum<float>());
```

```
//Allocator
#include "allocator.h"
…
static cached_allocator alloc;
…
Max =
thrust::reduce(thrust::cuda::par(a
lloc), Max_Ptr, Max_Ptr + (M)*(N),
0.0f, thrust::maximum<float>());
```

- See additional information and code in the post on our Website "https://parallel-computing.pro/index.php/9-cuda/34-thrust-cuda-tip-reuse-temporary-buffers-across-transforms"

Итог

- Постоянная экономия 2-6 секунд (2% в исходном варианте, 10% в финальной версии)



Optimized version works:
389.91/387.43 = 1.01x faster (Mode 1)
208.44/206.02 = 1.01x faster (Mode 2)
210.80/208.3 = 1.01x faster (Mode 3)
257.02/251.5 = 1.02x faster (Mode 4)

Общий эффект оптимизаций

- Read-only cache

(Achieved speedup of up to 1.27x)

- Kernel code optimizations

(Achieved speedup of up to 1.28x)

- AOS to SOA

(Achieved speedup of up to 1.48x)

- Block rotation

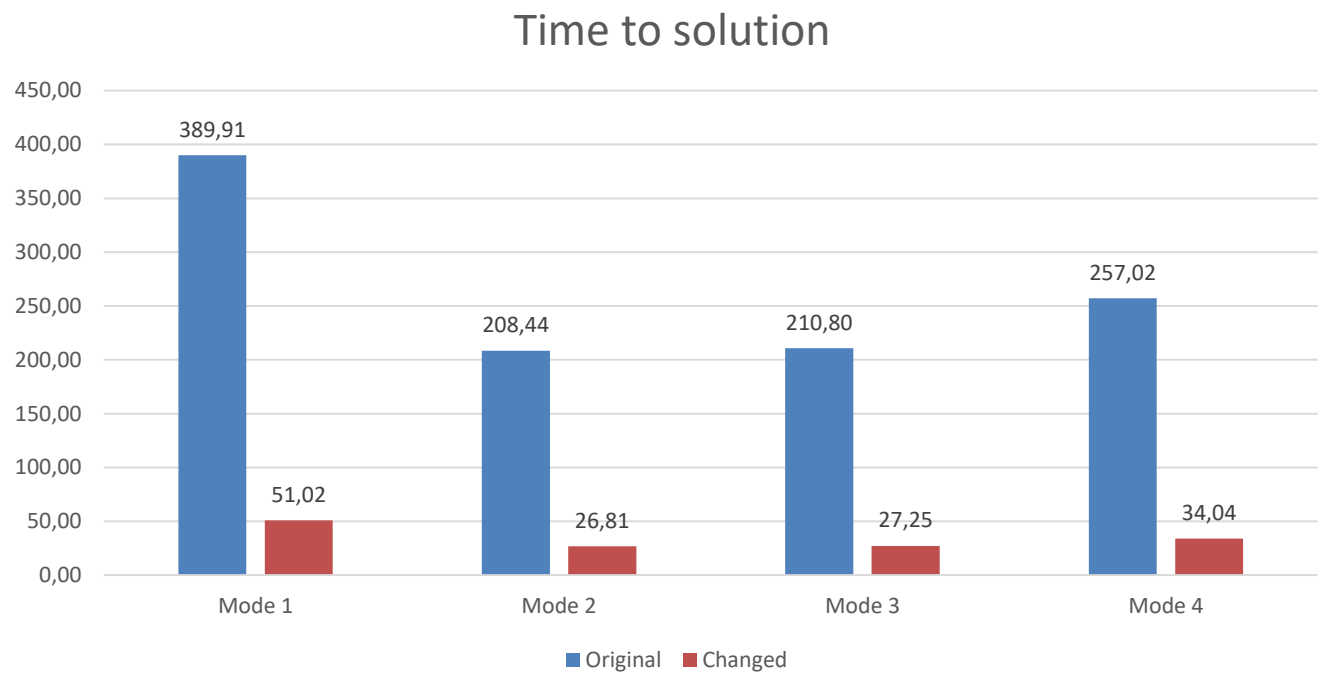
(Achieved speedup of up to 5.16x)

- cudaMemcpy -> pointers swap

(Achieved speedup of up to 1.03x)

- Custom memory allocator in Thrust

(Achieved speedup of up to 1.02x)



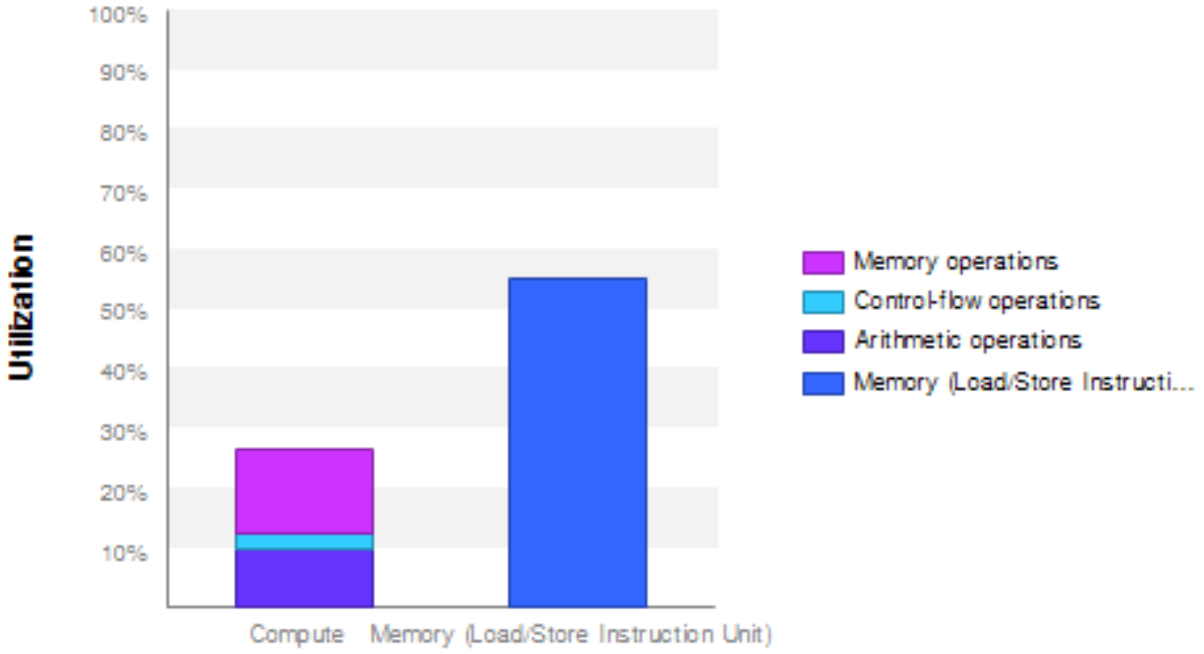
Optimized version works:
389.91/51.02 = 7.64x faster (Mode 1)
208.44/26.81 = 7.77x faster (Mode 2)
210.80/27.25 = 7.74x faster (Mode 3)
257.02/34.04 = 7.55x faster (Mode 4)

Эффект в профайлере

	original	final	Difference	
Warp Execution Efficiency	65,20	99,90	34,70	%
L2 throughput (Texture Reads)	0,00	125,46	125,46	GB/s
L2 throughput (L1 Reads)	118,00	22,83	-95,17	GB/s
Issue Slot Utilization	26,20	53,10	26,90	%
Global Load Transaction	100565016,00	1048064,00	-98,96	%
Global Store Transaction	16760838,00	10480672,00	-37,47	%
Control-Flow instruction	247255086,00	230436920,00	-6,80	%
Texture Cache Transactions	0,00	20961280,00	20961280,00	Times
Issued Control-Flow Instructions	24650222,00	11538432,00	-53,19	%
Warp Non-Predicated Execution Efficiency	61,90	95,40	33,50	%
Issued Load/Store Instructions	138673220,00	4754584,00	-96,57	%
Instructions Executed	130331460,00	62019584,00	-52,41	%
Global Memory Store Efficiency	12,50	100,00	87,50	%
Global Memory Load Efficiency	12,50	100,00	87,50	%
Instructions Issued	277422440,00	64629020,00	-76,70	%
Issued IPC	1,11	2,45	120,50	%
Executed IPC	0,52	2,35	349,90	%

Эффект в профайлере

Original



Final