Оценка эффективности разделяемой памяти в задаче решения уравнения Пуассона методом Якоби.

П. А. Пережогин

1 Постановка задачи.

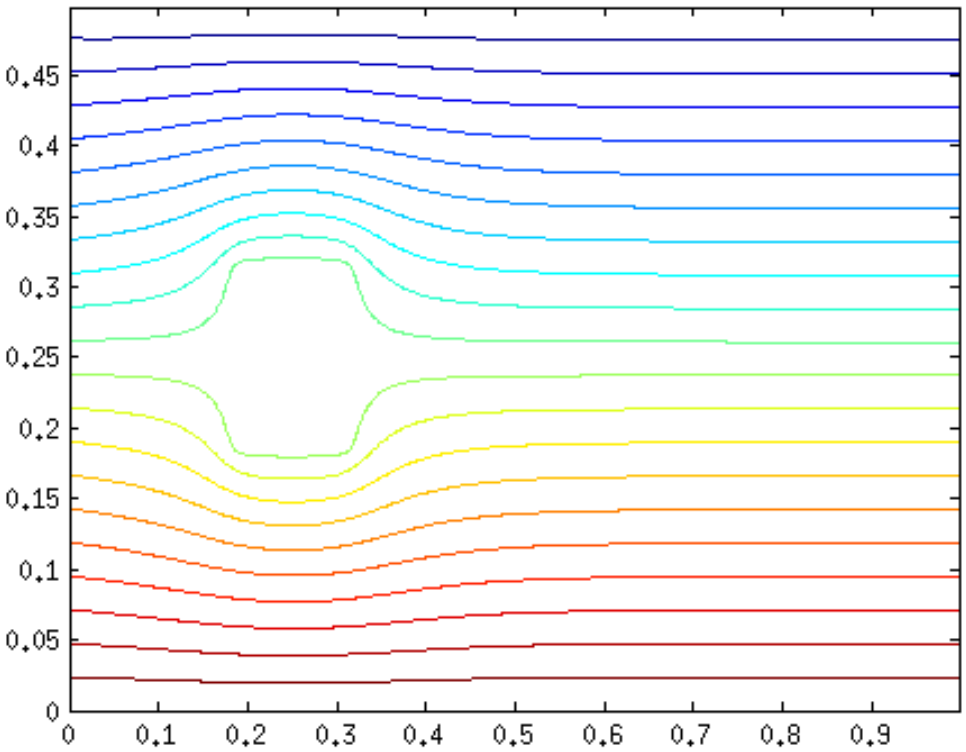

Рис. 1: Изолинии функции тока ψ при нулевой правой части. Изолинии совпадают с направлением течения.

Мы исследуем эффективность солвера, который может быть использован как один из компонентов для решения уравнений Навье-Стокса в задаче обтекания цилиндра:

$$\begin{cases} \frac{\partial \omega}{\partial t} + J(\psi, \omega) = \nu \triangle \omega, \\ \triangle \psi = \omega, \end{cases} \tag{1.1}$$

где ψ - функция тока, ω - завихренность, J - якобиан, а ν - вязкость. Второе из уравнений носит название уравнения Пуассона, и простейший способ его решить на равномерной сетке - это построить итерационный процесс по методу Якоби:

$$\frac{\psi_{i+1}^k - 2\psi_i^{k+1} + \psi_{i-1}^k}{h_x^2} + \frac{\psi_{j+1}^k - 2\psi_j^{k+1} + \psi_{j-1}^k}{h_y^2} = \omega_{ij}, \tag{1.2}$$

где индексом k обозначен номер итерации, а i, j - индексы пространственной области.

Расчетная область прямоугольная, а в середину области помещено обтекаемое тело, как показано на рисунке 1. Равномерная сетка имеет число узлов 512×256.

2 Реализация на видеокарте.

Для переноса алгоритма на видеокарту, разобьем расчетную область на плитки размером 32×32, а в каждой такой плитке будет работать блок нитей 32×4. Константы поместим в константную память, а ψ^k в разделяемую память размером 34×34 на один блок.

2.1 Константная память.

Сначала определим необходимые для выполнения алгоритма константы, которые помещены в структуры:

```
константы для итераций Якоби:
struct jacobi_s {
T a, b, c;
};

константы для определения расчетной плитки:
struct tile_s {
int y, xp2, xp3, nx, ny, ratio, distance, distance_b;
};

константы для определения обтекаемого тела:
struct square_s {
int x1, y1, x2, y2;
};
```

Константы помещены в одну структуру, которая размещается в константной памяти:

```
struct gpu_constants {
jacobi_s < T > jacobi;
tile_s tile;
square_s square;
};

__constant__ __device__ gpu_constants < Real > *gpu_c;
```

2.2 Вычислительные ядра (kernels).

2.2.1 kernel с использованием shared memory.

```
template < typename T >
__global__ void
jacobi_k ( T* psi, const T* psi_old, const T* w, const gpu_constants < T >* gpu_c ) {
int idx_g, idx_sb;
SharedMemory<T> smem;
T* psi_s = smem.getPointer(); выделение динамической разделяемой памяти

idx_g  = id_g  ( gpu_c->tile );
idx_sb = id_sb ( gpu_c->tile );

memcpy_cross ( psi_s, psi_old, idx_sb, idx_g, gpu_c->tile);
__syncthreads();

jacobi_d ( psi, psi_s, w, idx_g, idx_sb, gpu_c);
}
```

Здесь используется динамическая разделяемая память, которая, кроме того, может быть выделена под разный тип данных (float, double). Далее применяются device функции, которые для повышения производительности реализованы как inline.

Вычисление индекса в глобальной и разделяемой памяти для каждой нити.

```
inline __device__ int id_g ( const tile_s& tile ) {
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * tile.y + threadIdx.y * tile.ratio;
return i + j * tile.nx;
}

inline __device__ int id_sb ( const tile_s& tile ) {
return threadIdx.x + tile.xp3 + tile.distance_b * threadIdx.y;
}
```

Загрузка данных в разделяемую память.

```
template < typename T >
inline __device__ void memcpy_cross ( T* destination, const T* source,
   int idx_sb, int idx_g, const tile_s& tile ) {
int idx;

// copy boundary
if ( threadIdx.y == 0 ) {
idx = idx_g - tile.nx;
if ( outofBorder ( idx, tile ) == false )
destination[idx_sb - tile.xp2] = source[idx];
```

```
}

if ( threadIdx.y == 1 ) {
idx = blockDim.x * blockIdx.x - 1 + tile.nx * ( blockIdx.y * tile.y + threadIdx.x );
if ( outofBorder ( idx, tile ) == false )
if ( threadIdx.x < tile.y )
destination[( threadIdx.x + 1 ) * tile.xp2] = source[idx];
}

if ( threadIdx.y == 2 ) {
idx = blockDim.x * ( blockIdx.x + 1 ) + tile.nx * ( blockIdx.y * tile.y + threadIdx.x );
if ( outofBorder ( idx, tile ) == false )
if ( threadIdx.x < tile.y )
destination[( threadIdx.x + 2 ) * tile.xp2 - 1] = source[idx];
}

if ( threadIdx.y == blockDim.y - 1 ) {
idx = idx_g + tile.ratio * tile.nx;
if ( outofBorder ( idx, tile ) == false )
destination[idx_sb + tile.distance_b] = source[idx];
}

// copy interior of tile
for ( int offset = 0; offset < tile.ratio; offset++, idx_g += tile.nx, idx_sb += tile.xp2 )
destination[idx_sb] = source[idx_g];
}
```

Итерация Якоби, используя shared memory.

```
template < typename T >
inline __device__ void jacobi_d ( T* psi, const T* psi_s, const T* w,
int idx_g, int idx_sb, const gpu_constants < T >* gpu_c ) {
int mark;
const square_s &square = gpu_c->square;
const tile_s &tile = gpu_c->tile;
const jacobi_s < T > &jacobi = gpu_c->jacobi;

for ( int offset = 0; offset < tile.ratio; offset++, idx_g += tile.nx, idx_sb += tile.xp2 ) {
mark = inside ( idx_g, square, tile );
if ( mark == INSIDE ) {
psi[idx_g] = jacobi.a * ( psi_s[idx_sb + 1] + psi_s[idx_sb - 1] )
    + jacobi.b * ( psi_s[idx_sb + tile.xp2] + psi_s[idx_sb - tile.xp2] )
    + jacobi.c * w[idx_g];
}
else if ( mark == OUTFLOW )
psi[idx_g] = psi_s[idx_sb - 1];
}
```

```
}
```

Везде используются логические функции для определение принадлежности нити вычислительной области.

Логические функции.

```
inline __device__ bool isSquare ( const int& idx,
 const square_s& square, const tile_s& tile ) {
int i = idx % tile.nx;
int j = idx / tile.nx;
return ( ( i >= square.x1 ) && ( i <= square.x2 )
&& ( j >= square.y1 ) && (j <= square.y2) ) ? true:false;
}

inline __device__ bool outofBorder ( const int& idx, const tile_s& tile ) {
int i = idx % tile.nx;
int j = idx / tile.nx;
return ( ( i < 0 ) || ( j < 0 ) || ( i > tile.nx - 1 )
|| ( j > tile.ny - 1 ) ) ? true:false;
}
```

2.2.2 kernel без использования shared memory.

```
template < typename T >
__global__ void
jacobi_k_noshared ( T* psi, const T* psi_old, const T* w, const gpu_constants < T >* gpu_c )
int idx_g;

idx_g  = id_g  ( gpu_c->tile );

jacobi_d_noshared ( psi, psi_old, w, idx_g, gpu_c);
}
```

Здесь все функции идентичны тем, что показаны выше, только вместо разделяемой памяти используется глобальная.

Итерация Якоби, без shared memory.

```
template < typename T >
inline __device__ void jacobi_d_noshared ( T* psi, const T* psi_old, const T* w,
int idx_g, const gpu_constants < T >* gpu_c ) {
int mark;
const square_s &square = gpu_c->square;
const tile_s &tile = gpu_c->tile;
const jacobi_s < T > &jacobi = gpu_c->jacobi;

for ( int offset = 0; offset < tile.ratio; offset++, idx_g += tile.nx ) {
```

```
mark = inside ( idx_g, square, tile );
if ( mark == INSIDE ) {
psi[idx_g] = jacobi.a * ( psi_old[idx_g + 1] + psi_old[idx_g - 1] )
    + jacobi.b * ( psi_old[idx_g + tile.nx] + psi_old[idx_g - tile.nx] )
    + jacobi.c * w[idx_g];
}
else if ( mark == OUTFLOW )
psi[idx_g] = psi_old[idx_g - 1];
}
}
```

3 Проверка скорости работы вычислительных ядер (kernels).

Проведем 1000 итераций метода и измерим среднее время работы kernel с помощью Nvidia Profiler (nvprof) за одну итерацию. Изучим, влияет ли на результат размер плитки и размер блока. Вычисления с разделяемой памятью будем проводить с ключем $cudaFuncCachePreferShared$, а без разделяемой памяти с ключем $cudaFuncCachePreferL1$. Вычисления будем проводить с одинарной точностью. Результаты приведены в таблицах 1, 2.

Таблица 1: Время расчета одной итерации Якоби в среднем за 1000 итераций для видеокарты *Nvidia GTX* 570. Представлены различные размеры вычислительной плитки и блока. Разрешение 512x256.

плитка/блок	shared	no shared
32x32/32x4	41 мкс	29 мкс
32x16/32x4	38 мкс	27 мкс
32x8/32x4	43 мкс	28 мкс
32x4/32x4	58 мкс	32 мкс
32x32/32x8	38 мкс	26 мкс
32x16/32x8	41 мкс	28 мкс
32x32/32x16	42 мкс	29 мкс

Таблица 2: Время расчета одной итерации Якоби в среднем за 1000 итераций для видеокарты *Nvidia Tesla X*2070. Расчет произведен на кластере Ломоносов. Разрешение 512x256.

плитка/блок	shared	no shared
32x16/32x4	58 мкс	44 мкс

4 Заключение.

Как видно из результатов работы, применение разделяемой памяти не всегда оказывается эффективным. Использование $L1$ кэша оказывается предпочтительнее в том случае, когда алгоритм работает по принципу свертки начальных данных с некоторым небольшим фильтром, параметры которого хранятся в константной памяти. Возможно, это происходит по той причине, что кэш $L1$ адресуется

автоматически и для работы с ним не нужно тратить большое количество тактов на вычисление индексов. Использование кэша $L1$ может быть неэффективным только в том случае, когда имеет место большое количество кэш-промахов, но в представленных тестах $L1$ *cache hit rate* $= 70\%$.