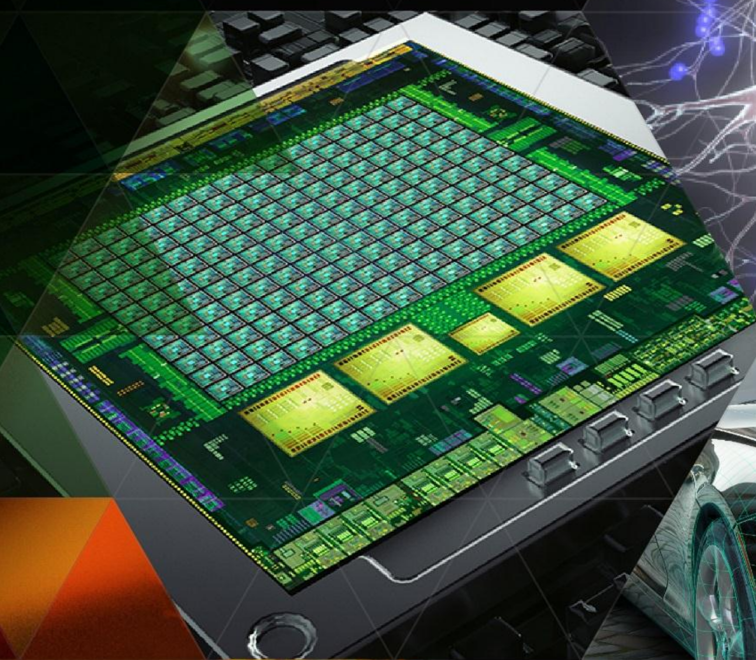
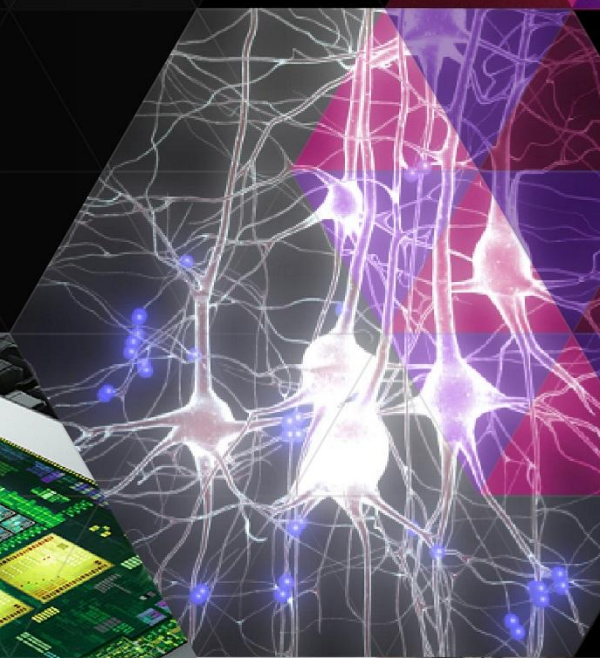
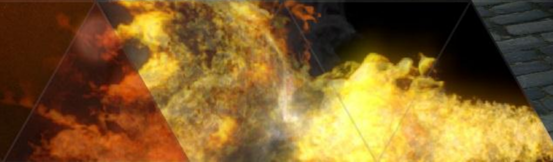
NVIDIA CUDA И OPENACC
ЛЕКЦИЯ 4

Перепёлкин Евгений



СОДЕРЖАНИЕ

Лекция 4

- Разделяемая память
- Шаблон работы с разделяемой памятью
- Пример. Задача N-тел
- Оптимизация работы с разделяемой памятью
- Пример. Перемножение матриц

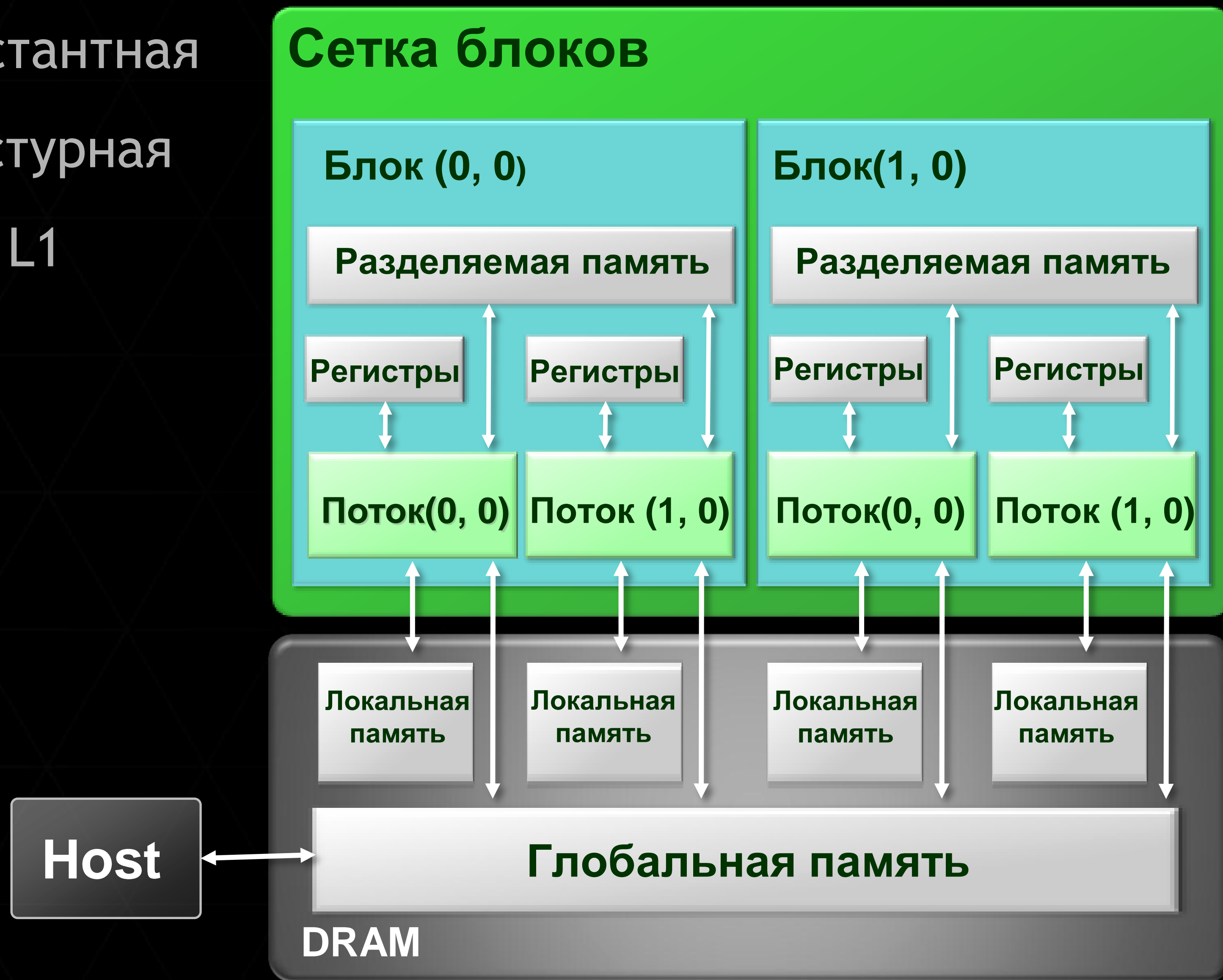
Разделяемая память

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы
Register (регистровая)	RW	Per-thread	Высокая (on-chip)
Local (локальная)	RW	Per-thread	Низкая (DRAM)
Global (глобальная)	RW	Per-grid	Низкая (DRAM)
Shared (разделяемая)	RW	Per-block	Высокая (on-chip)
Constant (константная)	RO	Per-grid	Высокая (L1 cache)
Texture (текстурная)	RO	Per-grid	Высокая (L1 cache)

- Константная
- Текстурная
- Кеш L1

ДОСТУП К ПАМЯТИ НА GPU



РАЗДЕЛЯЕМАЯ ПАМЯТЬ

Способы выделения и синхронизация

- Статический способ:
  - `__shared__ double F [64]; // массив F`
  - `__shared__ float G;          // переменная G`
- Динамический способ:
  - `extern __shared__ float [ ];   // 64 элемента`
  - `Kernel <<< nBlock, nTread, 64 * sizeof (float) >>> (...);`
- Барьерная синхронизация:
  - `__syncthreads ()`

ШАБЛОН

работы с разделяемой памятью

```
__global__ void My_Kernel ( float *a, float *b )
{
 int tx = threadIdx.x; // определение номера нити
 int bx = blockIdx.x; // определение номера блока
 // выделение разделяемой памяти для массива as
 __shared__ float as [BLOCK_SIZE];
 // параллельное копирование нитями блока данных из
 // массива a, расположенного в глобальной памяти в
 // массив as, расположенного в разделяемой памяти
 as [tx] = a [tx + bx * BLOCK_SIZE];
 __syncthreads (); // барьерная синхронизация нитей одного блока
 {...} // вычисление величины res, связанное с элементами массива as
 b [tx + bx * BLOCK_SIZE] = res; // параллельная запись в глобальную память
 __syncthreads (); // барьерная синхронизация нитей одного блока
}
```

Пример. Задача N-тел

ЗАДАЧА N-ТЕЛ

$$\vec{a}_{n,i} = \frac{\vec{F}_{n,i}}{m} = Gm \sum_{k \neq n}^{N-1} \frac{\vec{r}_{k,i} - \vec{r}_{n,i}}{\left|\vec{r}_{k,i} - \vec{r}_{n,i}\right|^3},$$

$$\vec{v}_{n,i+1} = \vec{v}_{n,i} + \vec{a}_{n,i}\tau,$$

$$\vec{r}_{n,i+1} = \vec{r}_{n,i} + \vec{v}_{n,i}\tau + \vec{a}_{n,i}\frac{\tau^2}{2},$$

$$t_i = t_0 + i\tau,$$

$$\left|\vec{r}_{k,i} - \vec{r}_{n,i}\right| < 0.01\text{м}, \vec{F}_{n,i} = 0,$$

$$mG = 10\,\text{Нм}^2/\text{кг}\,, \tau = 0.001c$$

ПРИМЕР. КОД ПРОГРАММЫ

Часть 1. Функция Acceleration_CPU

```
// CPU – вариант. Вычисление ускорения
void Acceleration_CPU (float *X, float *Y, float *AX, float *AY,
                       int nt, int N, int id)
{float ax = 0.f; float ay = 0.f; float xx, yy, rr; int sh = (nt - 1) * N;

 for ( int j = 0; j < N; j++ )  // цикл по частицам
 {if ( j != id )                // поверка самодействия
  {xx = X[j + sh] - X[id + sh]; yy = Y[j + sh] - Y[id + sh];
   rr = sqrtf (xx * xx + yy * yy);
   if ( rr > 0.01f )            // минимальное расстояние 0.01 м
   {rr = 10.f / (rr * rr * rr); ax += xx * rr; ay += yy * rr;
   } // if rr
  } // if id
 } // for
 AX[id] = ax; AY[id] = ay;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 2. Функция Position_CPU

```
// CPU-вариант. Пересчет координат
void Position_CPU (float *X, float *Y, float *VX,
                   float *VY, float *AX, float *AY,
                   float tau, int nt, int Np, int id)
{
 int sh = (nt - 1) * Np;
 X[id + nt * Np] = X[id + sh] + VX[id] * tau + AX[id] * tau * tau * 0.5f;
 Y[id + nt * Np] = Y[id + sh] + VY[id] * tau + AY[id] * tau * tau * 0.5f;

 VX[id] += AX[id] * tau;
 VY[id] += AY[id] * tau;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 3. Функция-ядро Acceleration_GPU

```
// GPU-вариант. Расчет ускорения
__global__ void Acceleration_GPU (float *X, float *Y,
                                  float *AX, float *AY, int nt, int N)
{int id = threadIdx.x + blockIdx.x * blockDim.x;
 float ax = 0.f; float ay = 0.f; float xx, yy, rr; int sh = (nt - 1) * N;
 for ( int j = 0; j < N; j++ )        // цикл по частицам
 {if (j != id)                        // проверка самодействия
  {xx = X[j + sh] - X[id + sh]; yy = Y[j + sh] - Y[id + sh];
   rr = sqrtf (xx * xx + yy * yy);
   if (rr > 0.01f)                    // минимальное расстояние 0.01 м
   {rr = 10.f / (rr * rr * rr); ax += xx * rr; ay += yy * rr;
   } // if rr
  } // if id
 } // for j
 AX[id] = ax; AY[id] = ay;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 4. Функция-ядро Position_GPU

```
// GPU-вариант. Пересчет координат
__global__ void Position_GPU (float *X, float *Y, float *VX, float *VY,
                              float *AX, float *AY, float tau, int nt, int Np)
{
 int id = threadIdx.x + blockIdx.x * blockDim.x;
 int sh = (nt - 1) * Np;

 X[id + nt * Np] = X[id + sh] + VX[id] * tau + AX[id] * tau * tau * 0.5f;
 Y[id + nt * Np] = Y[id + sh] + VY[id] * tau + AY[id] * tau * tau * 0.5f;

 VX[id] += AX[id] * tau;
 VY[id] += AY[id] * tau;
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 5. Функция main

```
int main (int argc, char* argv[])
{float timerValueGPU, timerValueCPU;
 cudaEvent_t start, stop;  // определение переменных-событий для таймера
 cudaEventCreate ( &start ); cudaEventCreate ( &stop );

 int N = 10240;          // число частиц (2-й вариант 20480)
 int NT = 10;            // число шагов по времени (для анимации - 800)
 float tau = 0.001f;   // шаг по времени 0.001 с

// создание массивов на host
 float *hX, *hY, *hVX, *hVY, *hAX, *hAY;
 unsigned int mem_size     = sizeof (float) * N;
 unsigned int mem_size_big = sizeof (float) * NT * N;
 hX  = (float*) malloc (mem_size_big); hY  = (float*) malloc (mem_size_big);
 hVX = (float*) malloc (mem_size);     hVY = (float*) malloc (mem_size);
 hAX = (float*) malloc (mem_size);     hAY = (float*) malloc (mem_size);
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 6. Функция main

```
// задание начальных условий на host
 float vv,phi;
 for ( j = 0; j < N; j++ )
 {phi = (float) rand();
  hX[j]  = rand() * cosf(phi) * 1.e-4f; hY[j] = rand() * sinf(phi) * 1.e-4f;
  vv = (hX[j] * hX[j] + hY[j] * hY[j]) * 10.f;
  hVX[j] = - vv * sinf(phi); hVY[j] = vv * cosf(phi);
 }
// создание на device массивов
 float *dX, *dY, *dVX, *dVY, *dAX, *dAY;
 cudaMalloc((void**) &dX, mem_size_big);
 cudaMalloc((void**) &dY, mem_size_big);
 cudaMalloc((void**) &dVX, mem_size); cudaMalloc((void**) &dVY, mem_size);
 cudaMalloc((void**) &dAX, mem_size); cudaMalloc((void**) &dAY, mem_size);
// задание сетки нитей и блоков
 int N_thread = 256; int N_block = N / N_thread;
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 7. Функция main

```
// ------------------GPU-вариант--------------------------
cudaEventRecord ( start, 0 );
// копирование данных на device
cudaMemcpy ( dX, hX, mem_size_big, cudaMemcpyHostToDevice );
cudaMemcpy ( dY, hY, mem_size_big, cudaMemcpyHostToDevice );
cudaMemcpy ( dVX, hVX, mem_size, cudaMemcpyHostToDevice );
cudaMemcpy ( dVY, hVY, mem_size, cudaMemcpyHostToDevice );

for ( j = 1; j < NT; j++ )
{// расчет ускорения
 Acceleration_GPU <<< N_block, N_thread >>> ( dX, dY, dAX, dAY, j, N );
 // пересчет координат
 Position_GPU <<< N_block, N_thread >>>
                      ( dX, dY, dVX, dVY, dAX, dAY, tau, j, N );
}
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 8. Функция main

```
// копирование траекторий с device на host
cudaMemcpy ( hX, dX, mem_size_big, cudaMemcpyDeviceToHost );
cudaMemcpy ( hY, dY, mem_size_big, cudaMemcpyDeviceToHost );

// определение времени выполнения GPU-варианта
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &timerValueGPU, start, stop );
printf ("\n GPU calculation time %f msec\n", timerValueGPU);

{...} // сохранение траекторий в файл, GPU-вариант
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 9. Функция main

```
//------------------CPU-вариант------------------------
cudaEventRecord ( start, 0 );
for ( j = 1; j < NT; j++ )
{for ( id = 0; id < N; id++ )
 {Acceleration_CPU (hX, hY, hAX, hAY, j, N, id);
  Position_CPU (hX, hY, hVX, hVY, hAX, hAY, tau, j, N, id);
 }
}
// определение времени выполнения CPU-варианта
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &timerValueCPU, start, stop );
printf ("\n CPU calculation time %f msec\n", timerValueCPU);
printf ("\n Rate %f x\n",timerValueCPU/timerValueGPU);
{...} // сохранение траекторий, CPU-вариант
```

ПРИМЕР. КОД ПРОГРАММЫ

Часть 10. Функция main

```
// освобождение памяти
 free (hX); free (hY); free (hVX); free (hVY); free (hAX); free (hAY);
 cudaFree (dX); cudaFree (dY); cudaFree (dVX); cudaFree (dVY);

 // уничтожение переменных-событий
 cudaEventDestroy ( start );
 cudaEventDestroy ( stop );

 return 0;
}
```

ПРИМЕР. ЗАДАЧА N-ТЕЛ

ПРИМЕР. ЗАДАЧА N-ТЕЛ

РЕЗУЛЬТАТ

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Number of particles : 10240      20480
GPU calculation time: 96.7 ms    215.7 ms
CPU calculation time: 3442 ms    13821 ms
Rate                : 35 x       64 x
```

Время расчета GPU-варианта включает в себя:

- копирование данных с «host» на «device»;
- выполнение «функции-ядра»;
- копирование данных с «device» на «host».

КОД ПРОГРАММЫ НА CUDA

```
__global__ void Acceleration_Shared (float *X, float *Y, float *AX, float *AY,
                                     int nt, int N, int N_block)
{int id = threadIdx.x + blockIdx.x * blockDim.x;
 float ax = 0.f; float ay = 0.f; float xx, yy, rr; int sh = (nt - 1) * N;
 float xxx = X[id + sh]; float yyy = Y[id + sh];
 __shared__ float Xs[256]; __shared__ float Ys[256]; // выделение разделяемой памяти
 for ( int i = 0; i < N_block; i++ )                          // основной цикл по блокам
 {Xs[threadIdx.x] = X[threadIdx.x + i * blockDim.x + sh]; // копирование из глобальной
  Ys[threadIdx.x] = Y[threadIdx.x + i * blockDim.x + sh]; // в разделяемую память
  __syncthreads ();                                      // синхронизация
  for ( int j = 0; j < blockDim.x; j++ )                 // вычислительная часть
  {if ( ( j + i * blockDim.x ) != id )
   {xx = Xs[j] - xxx; yy = Ys[j] - yyy; rr = sqrtf ( xx * xx + yy * yy );
    if ( rr > 0.01f ) {rr = 10.f /(rr * rr * rr); ax += xx * rr; ay += yy * rr;} //if
   } // if id
  } // for j
  __syncthreads ();                                      // синхронизация
 } // for i
 AX[id] = ax; AY[id] = ay;
}
```

РЕЗУЛЬТАТ

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Number of particles : 10240      20480
GPU calculation time: 74.4 ms    173.6 ms
CPU calculation time: 3442 ms    13821 ms
Rate                : 46 x       79 x
```

Время расчета GPU-варианта включает в себя:

- копирование данных с «host» на «device»;
- выполнение «функции-ядра»;
- копирование данных с «device» на «host».

Оптимизация работы с разделяемой памятью

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНКИ

Compute Capability 3.x

- 32 банка, ширина 8 Байт
  - пропускная способность 8 Байт за такт на SMX
  - варп (32 потока) считывает 256 Байт за такт на SMX
- Два режима доступа
  - 4-Байтовый `cudaSharedMemBankSizeFourByte` (по умолчанию)
  - 8-Байтовый `cudaSharedMemBankSizeEightByte`
  - задается функцией `cudaDeviceSetSharedMemConfig ()`

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНКИ

8-Байтовый режим доступа

Банк 0		Банк 1		Банк 2		Банк 3		…	Банк 30		Банк 31	
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]		A[60]	A[61]	A[62]	A[63]
A[64]	A[65]							…				
…	…	…	…	…	…	…	…	…	…	…	…	…

4 Байта = 32 бита

```
__shared__ float A [ N ];

float x = A [ threadIdx.x ];
```

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНКИ

4-Байтовый режим доступа

Банк 0		Банк 1		Банк 2		Банк 3		…	Банк 30		Банк 31	
A[0]	A[32]	A[1]	A[33]	A[2]	A[34]	A[3]	A[35]		A[30]	A[62]	A[31]	A[63]
A[64]	A[128]	A[65]						…				
…	…	…	…	…	…	…	…	…	…	…	…	…

4 Байта = 32 бита

```
__shared__ float A [ N ];

float x = A [ threadIdx.x ];
```

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. БАНК КОНФЛИКТЫ

Compute Capability 3.x

- Банк конфликты возникают, когда:
  - две или более нитей одного варпа обращаются к разным 8-Байтовым словам, лежащим в одном банке
  - банк-конфликт имеет порядок N когда конфликтуют N нитей одного варпа
- Банк конфликтов нет, когда:
  - разные нити варпа обращаются к одному слову
  - разные нити варпа обращаются к различным байтам одного и того же слова

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Нет банк конфликтов

В каждый банк по одному обращению

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Нет банк конфликтов

В каждый банк по одному обращению

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Нет банк конфликтов

Несколько обращение в один банк, но к одному и тому же слову

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Банк конфликт 2-го порядка

Обращение к двум разным словам, лежащим в одном банке

РАЗДЕЛЯЕМАЯ ПАМЯТЬ. ПРИМЕР ДОСТУПА

Compute Capability 3.x

Банк конфликт 3-го порядка

Обращение к трем разным словам, лежащим в одном банке

Пример. Перемножение матриц

ПРИМЕР ПЕРЕМНОЖЕНИЯ МАТРИЦ

$$C = AB,$$

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} b_{k,j} ,$$

$$a_{i,j} = 2j + i, \qquad b_{i,j} = j - i,$$

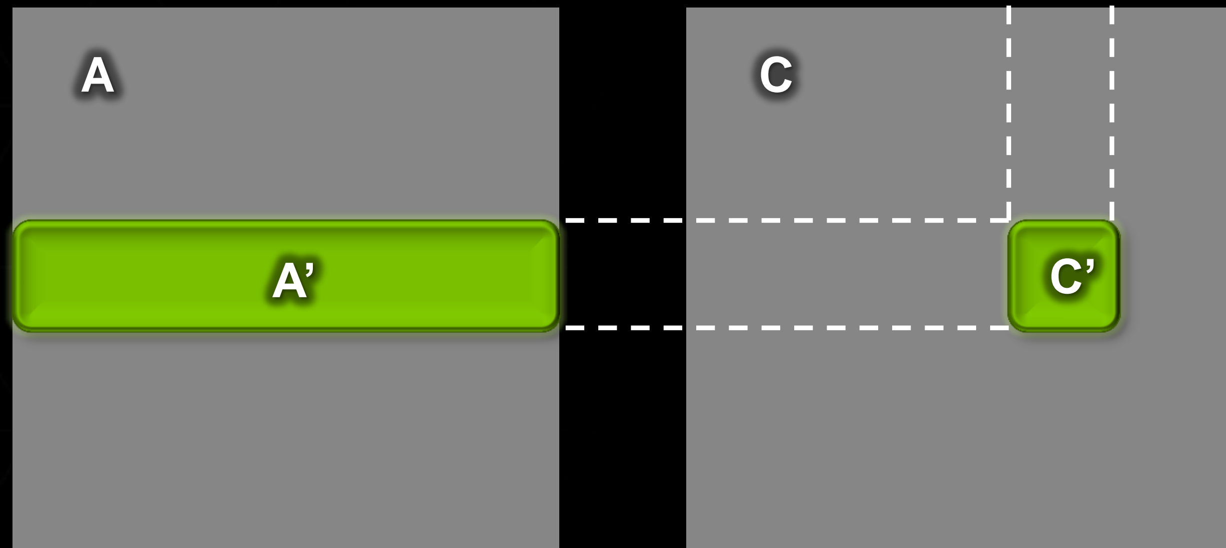
$$i, j = 0, \dots, N - 1$$

N x N = 2048 x 2048, BLOCK_SIZE = 32

(tx, ty) – (32, 32) – нити внутри блока

(bx, by) – (64, 64) – число блоков

ВАРИАНТ «GLOBAL»



КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

Часть 1. Функция-ядро

```
#define BLOCK_SIZE 32

__global__ void kernel_global ( float *a, float *b, int n, float *c )
{int    bx   = blockIdx.x;  // номер блока по x
 int    by   = blockIdx.y;  // номер блока по y
 int    tx   = threadIdx.x; // номер нити в блоке по x
 int    ty   = threadIdx.y; // номер нити в блоке по y
 float sum = 0.0f;
 int ia = n * ( BLOCK_SIZE * by + ty );  // номер строки из A′
 int ib = BLOCK_SIZE * bx + tx;          // номер столбца из B′
 int ic = ia + ib;                       // номер элемента из C′
// вычисление элемента матрицы C
 for ( int k = 0; k < n; k++ ) sum += a[ia + k] * b[ib + k * n];
 c[ic] = sum;
}
```

КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

Часть 2. Функция main

```
int main()
{int N = 2048;
 int m, n, k;
 // создание переменных-событий
 float timerValueGPU, timerValueCPU;
 cudaEvent_t start, stop;
 cudaEventCreate (&start); cudaEventCreate (&stop);

 int numBytes = N * N * sizeof (float );
 float *adev, *bdev, *cdev, *a, *b, *c, *cc, *bT;
 // выделение памяти на host
 a  = (float *) malloc (numBytes); //матрица A
 b  = (float *) malloc (numBytes); //матрица B
 bT = (float *) malloc (numBytes); //транспонированная матрица B
 c  = (float *) malloc (numBytes); //матрица С для GPU-варианта
 cc = (float *) malloc (numBytes); //матрица С для CPU-варианта
```

КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

Часть 3. Функция main

```
// задание матрицы A, B и транспонированной матрицы B
for ( n = 0; n < N; n++ )
{for ( m = 0; m < N; m++ )
 {a[m + n * N] = 2.0f * m + n; b[m + n * N] = m - n; bT[m + n * N] = n - m;
 }
}
// задание сетки нитей и блоков
dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
dim3 blocks ( N / threads.x, N / threads.y);
// выделение памяти на GPU
cudaMalloc ( (void**)&adev, numBytes );
cudaMalloc ( (void**)&bdev, numBytes );
cudaMalloc ( (void**)&cdev, numBytes );
```

КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

Часть 4. Функция main

```
// ---------------- GPU-вариант ----------------------
// копирование матриц A и B с host на device
cudaMemcpy ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy ( bdev, b, numBytes, cudaMemcpyHostToDevice );
// запуск таймера
cudaEventRecord (start, 0);
// запуск функции-ядра
kernel_global <<< blocks, threads >>> ( adev, bdev, N, cdev );
// оценка времени вычисления GPU-варианта
cudaThreadSynchronize ();
cudaEventRecord (stop, 0);
cudaEventSynchronize (stop);
cudaEventElapsedTime (&timerValueGPU, start, stop);
printf ("\n GPU calculation time %f msec\n", timerValueGPU);
// копирование, вычисленной матрицы C с device на host
cudaMemcpy ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
```

КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

Часть 5. Функция main

```
// -------------------- CPU-вариант --------------------
// запуск таймера
 cudaEventRecord (start, 0);
// вычисление матрицы C
 for ( n = 0; n < N; n++ )
 {for ( m = 0; m < N; m++ )
  {cc[m+n*N] = 0.f;
   for( k = 0; k < N; k++ ) cc[m+n*N] += a[k+n*N] * bT[k+m*N]; // bT !!!
  }
 }
// оценка времени вычисления CPU-варианта
 cudaEventRecord (stop, 0);
 cudaEventSynchronize (stop);
 cudaEventElapsedTime (&timerValueCPU, start, stop);
 printf ("\n CPU calculation time %f msec\n",timerValueCPU);
 printf ("\n Rate %f x\n",timerValueCPU/timerValueGPU);
```

КОД ПРОГРАММЫ. ВАРИАНТ «GLOBAL»

Часть 6. Функция main

```
// освобождение памяти на GPU и CPU
 cudaFree ( adev );
 cudaFree ( bdev );
 cudaFree ( cdev );
 free ( a );
 free ( b );
 free ( bT );
 free ( c );
 free ( cc );
 // уничтожение переменных-событий
 cudaEventDestroy ( start );
 cudaEventDestroy ( stop );

 return 0;
}
```

РЕЗУЛЬТАТ. ВАРИАНТ «GLOBAL»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Precision            : float      double
GPU calculation time : 134  ms    220  ms
CPU calculation time : 4622 ms    9154 ms
Rate                 : 34 x       41 x
```

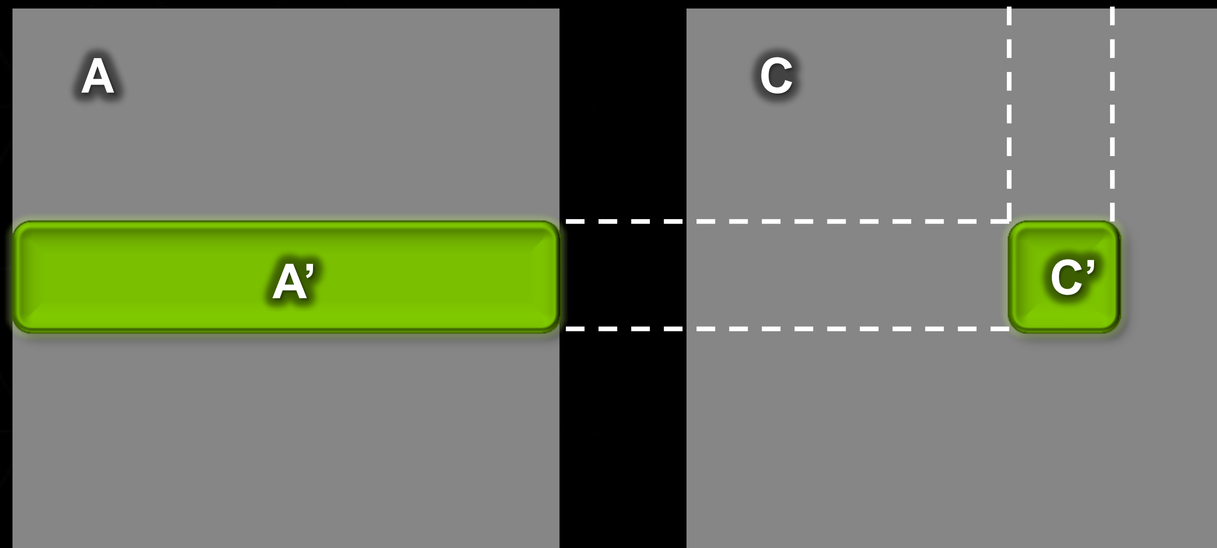
При вычислении

$c_{i,0}, c_{i,1}, \ldots, c_{i,N-1}$

N-раз считывается строка

$a_{i,0}, a_{i,1}, \ldots, a_{i,N-1}$

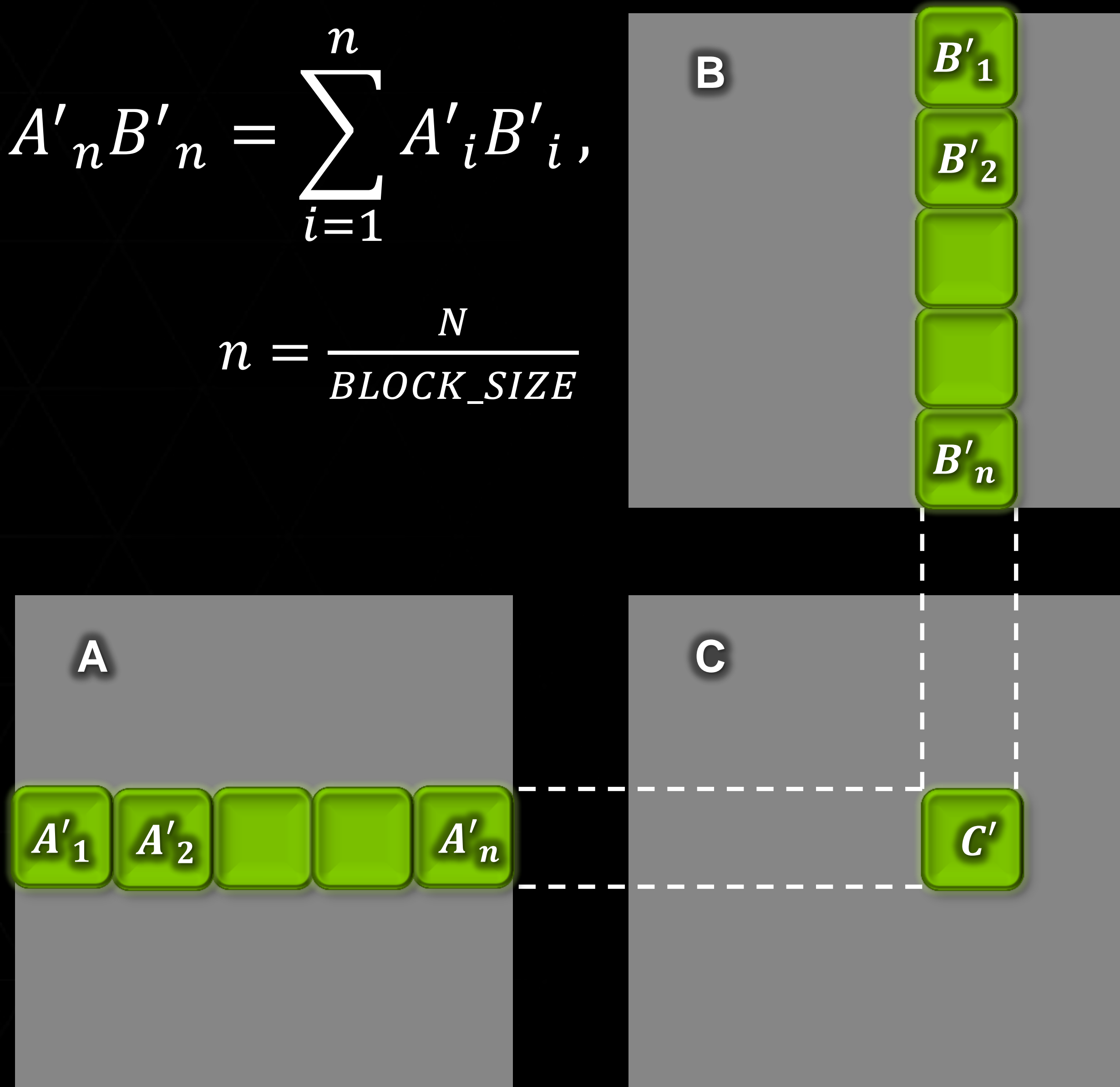
НЕДОСТАТОК ВАРИАНТА «GLOBAL»



$$C' = A'_1 B'_1 + A'_2 B'_2 + \cdots + A'_n B'_n = \sum_{i=1}^{n} A'_i B'_i\,,$$

$$n = \frac{N}{BLOCK_SIZE}$$

ВАРИАНТ «SMEM»



КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-1»

Часть 1. Функция-ядро

```
__global__ void kernel_smem_1 ( float *a, float *b, int n, float *c )
{int bx = blockIdx.x,  by = blockIdx.y;
 int tx = threadIdx.x, ty = threadIdx.y;
 int aBegin = n * BLOCK_SIZE * by, aEnd = aBegin + n - 1;
 int bBegin = BLOCK_SIZE * bx, aStep = BLOCK_SIZE, bStep = BLOCK_SIZE * n;
 float sum  = 0.0f;
 __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
 __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
 for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )
 {as [tx][ty] = a [ia + n * ty + tx]; bs [tx][ty] = b [ib + n * ty + tx];
  __syncthreads ();
  for ( int k = 0; k < BLOCK_SIZE; k++ ) sum += as [k][ty] * bs [tx][k];
  __syncthreads ();
 }
 c [aBegin + bBegin + ty * n + tx] = sum;
}
```

РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-1»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Precision               : float      double
GPU calculation time: 150  ms     476  ms
CPU calculation time: 4622 ms     9154 ms
Rate                    : 30 x       19 x
```

```
as [tx][ty] = a [ia + n * ty + tx]; // копирование из глобальной
bs [tx][ty] = b [ib + n * ty + tx]; // в разделяемую память
```

```
  (*) ind  = tx + ty * BLOCK_SIZE
 (**) indM = ty + tx * BLOCK_SIZE
```

- линейный номер нити
- линейный номер элементов в матрицах «as» и «bs»

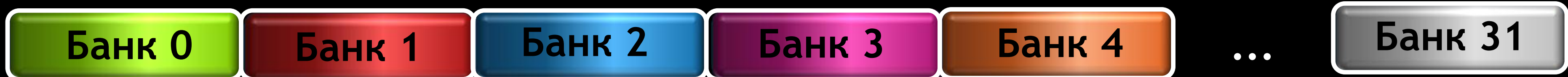
БАНК КОНФЛИКТЫ

```
__shared__ double B [32][32];
```

	Варп 0	Варп 1	Варп 2	…	Варп 31
Варп 0	B[0][0]	B[0][1]	B[0][2]	…	B[0][31]
Варп 1	B[1][0]	B[1][1]	B[1][2]	…	B[1][31]
Варп 2	B[2][0]	B[2][1]	B[2][2]	…	B[2][31]
…	…	…	…	…	…
Варп 31	B[31][0]	B[31][1]	B[31][2]	…	B[31][31]

32 элемента

Обращение по столбцу – дает банк конфликт 32 порядка



НЕТ БАНК КОНФЛИКТОВ

```
__shared__ double B [32][33];
```

	Варп 0	Варп 1	Варп 2	…	Варп 31	
Варп 0	B [0][0]	B[0][1]	B[0][2]	…	B[0][31]	B[0][32]
Варп 1	B[1][0]	B[1][1]	B[1][2]	…	B[1][31]	B[1][32]
Варп 2	B[2][0]	B[2][1]	B[2][2]	…	B[2][31]	B[2][32]
…	…	…	…	…	…	…
Варп 31	B[31][0]	B[31][1]	B[31][2]	…	B[31][31]	B[31][32]

Обращение по столбцу и по строке без банк конфликтов

РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-2»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Precision           : float         double (4B)    double (8B)
GPU calculation time: 56 ms (150)   87 ms (476)    62 ms (476)
CPU calculation time: 4622 ms       9154 ms        9154 ms
Rate                : 82 x          105 x          147 x
```

В «функции-ядре» строки (SMEM-1):

```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
```

Заменили на строки (SMEM-2):

```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE + 1];
__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE + 1];
```

КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-3»

Функция-ядро «SMEM-3»

```
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
```

Вместо строк (SMEM-1, SMEM-2):

```
as [tx][ty] = a [ia + n * ty + tx];
bs [tx][ty] = b [ib + n * ty + tx];
sum += as [k][ty] * bs [tx][k];
```

Поставим стоки (SMEM-3):

```
as [ty][tx] = a [ia + n * ty + tx];
bs [ty][tx] = b [ib + n * ty + tx];
sum += as [ty][k] * bs [k][tx];
```

`ind  = tx + ty * BLOCK_SIZE` – линейный номер нити

`indM = tx + ty * BLOCK_SIZE` – линейный номер элементов в матрицах «as» и «bs»

РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-3»

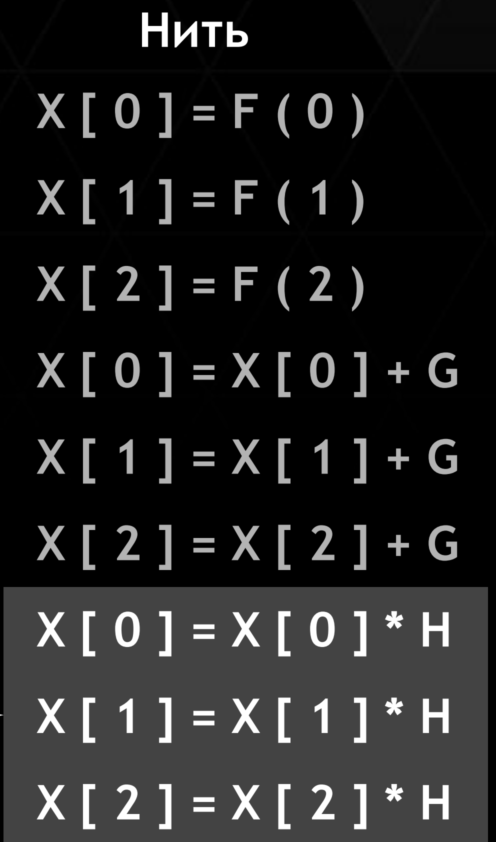
CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Precision           : float         double (4B, 8B)
GPU calculation time: 46 ms  (56)   80 ms   (87, 62)
CPU calculation time: 4622 ms       9154 ms
Rate                : 100 x         114 x
```

ПАРАЛЛЕЛИЗМ ПО НИТЯМ И ПО ИНСТРУКЦИЯМ

Thread-Level Parallelism (TLP)

Instruction-Level Parallelism (ILP)



Три независимых операции

КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-4»

Функция-ядро «SMEM-4»

Вместо строк (SMEM-3):

```
float sum = 0.0f;
as [ty][tx] = a [ia + n * ty + tx]; bs [ty][tx] = b [ib + n * ty + tx];
sum += as [ty][k] * bs [k][tx];
c [aBegin + bBegin + ty * n + tx] = sum;
```

Поставим строки (SMEM-4):

```
float sum1 = 0.0f, sum2 = 0.0f;
as [ty][tx] = a [ia + n * ty + tx];
bs [ty][tx] = b [ib + n * ty + tx];
as [ty + 16][tx] = a [ia + n * ( ty + 16 ) + tx];
bs [ty + 16][tx] = b [ib + n * ( ty + 16 ) + tx];
sum1 += as [ty][k] * bs [k][tx];
sum2 += as [ty + 16][k] * bs [k][tx];
c [aBegin + bBegin + ty * n + tx] = sum1;
c [aBegin + bBegin + ( ty + 16 ) * n + tx] = sum2;
```

КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-4»

Функция main

Добавим строки для блока нитей (SMEM-4):

```
dim3 threads_4 ( BLOCK_SIZE, BLOCK_SIZE / 2);
```

для запуска новой «функции-ядра» (SMEM-4):

```
kernel_smem_4 <<< blocks, threads_4 >>> ( adev, bdev, N, cdev );
```

РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-4»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Precision            : float          double (4B, 8B)
GPU calculation time: 33 ms  (46)    50 ms  (80, 80)
CPU calculation time: 4622 ms        9154 ms
Rate                 : 140 x          183 x
```

КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-5»

Функция-ядро «SMEM-5»

Добавим новые строки при копировании данных:

```
float sum1 = 0.0f, sum2 = 0.0f, sum3 = 0.0f, sum4 = 0.0f;
as [ty][tx] = a [ia + n * ty + tx];
bs [ty][tx] = b [ib + n * ty + tx];
as [ty + 8][tx] = a [ia + n * ( ty + 8 ) + tx];
bs [ty + 8][tx] = b [ib + n * ( ty + 8 ) + tx];
as [ty + 16][tx] = a [ia + n * ( ty + 16 ) + tx];
bs [ty + 16][tx] = b [ib + n * ( ty + 16 ) + tx];
as [ty + 24][tx] = a [ia + n * ( ty + 24 ) + tx];
bs [ty + 24][tx] = b [ib + n * ( ty + 24 ) + tx];
```

КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-5»

Функция-ядро «SMEM-5»

Изменения при перемножении матриц:

```
sum1 += as [ty][k] * bs [k][tx];
sum2 += as [ty + 8][k] * bs [k][tx];
sum3 += as [ty + 16][k] * bs [k][tx];
sum4 += as [ty + 24][k] * bs [k][tx];
```

при сохранении данных:

```
c [aBegin + bBegin + ty * n + tx] = sum1;
c [aBegin + bBegin + ( ty + 8 ) * n + tx] = sum2;
c [aBegin + bBegin + ( ty + 16 ) * n + tx] = sum3;
c [aBegin + bBegin + ( ty + 24 ) * n + tx] = sum4;
```

КОД ПРОГРАММЫ. ВАРИАНТ «SMEM-5»

Функция «main»

Добавим строки для блока нитей (SMEM-5):

```
dim3 threads_5 ( BLOCK_SIZE, BLOCK_SIZE / 4);
```

для новой «функции-ядра» (SMEM-5):

```
kernel_smem_5 <<< blocks, threads_5 >>> ( adev, bdev, N, cdev );
```

РЕЗУЛЬТАТ. ВАРИАНТ «SMEM-5»

CPU Core2 Quad Q8300 2.5 ГГц ICC x64 1-ядро GPU Tesla K40c CUDA 6.0

```
Precision                : float           double   (4B, 8B)
GPU calculation time: 26 ms  (33)     45.5 ms (50, 50)
CPU calculation time: 4622 ms         9154 ms
Rate                 : 177 x           201 x
```

Вариант «SMEM-3»

```
GPU calculation time: 46 ms           80 ms    (80, 80)
Rate                 : 100 x           114 x
```

Сравнение вариантов

		Global	SMEM-1	SMEM-2	SMEM-3	SMEM-4	SMEM-5
float		134 мс	150 мс	56 мс	46 мс	33 мс	26 мс
		34 x	30 x	82 x	100 x	140 x	177 x
double		220 мс	476 мс	62 мс*	80 мс	50 мс	45.5 мс
		41 x	19 x	147 x*	114 x	183 x	201 x

* 8-Байтовый режим доступа