Иерархия памяти и эффективное программирование CUDA

Лекция 2

Запуск двух ядер на разных GPU

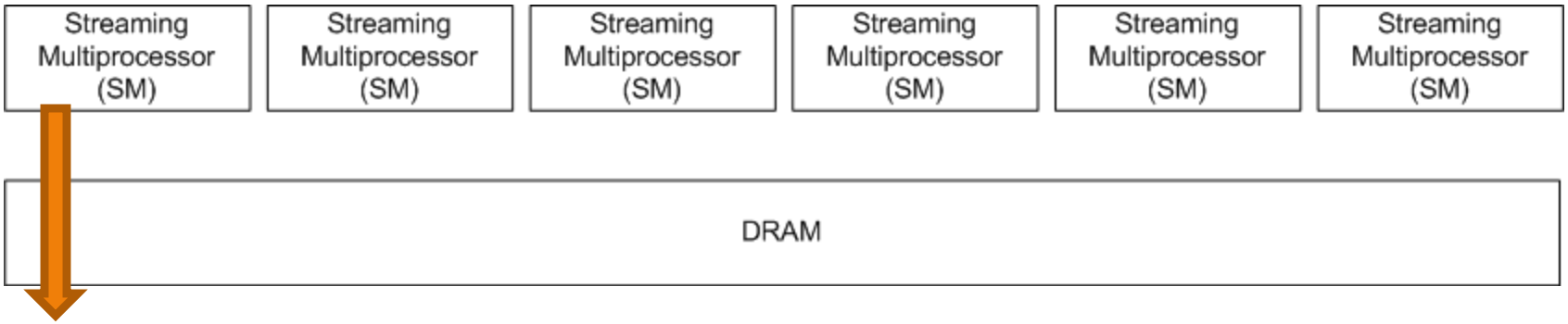
```
__global__ void add_kernel(int *c, const int *a, const int *b) //ядро 1
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
__global__ void sub_kernel(int *c, const int *a, const int *b) //ядро 2
{
    int i = threadIdx.x;
    c[i] = a[i] - b[i];
}
……
int main()
{
   // определяем количество GPU
   ……..
    // если GPU >= 2, то создаем стримы для двух разных GPU
    if (cudaSetDevice(0) != cudaSuccess) fprintf(stderr, "cudaSetDevice %d failed!", 0);
    cudaStream_t stream_gpu0;
    cudaStreamCreate(&stream_gpu0);

    if (cudaSetDevice(1) != cudaSuccess) fprintf(stderr, "cudaSetDevice %d failed!", 1);
    cudaStream_t stream_gpu1;
    cudaStreamCreate(&stream_gpu1);
    ……..
    // запуск ядер
    add_kernel<<<1, arraySize, 0, stream_gpu0>>>(dev0_c, dev0_a, dev0_b);
    sub_kernel<<<1, arraySize, 0, stream_gpu1>>>(dev1_c, dev1_a, dev1_b);
    …….
    return 0;
}
```

Типы памяти в CUDA

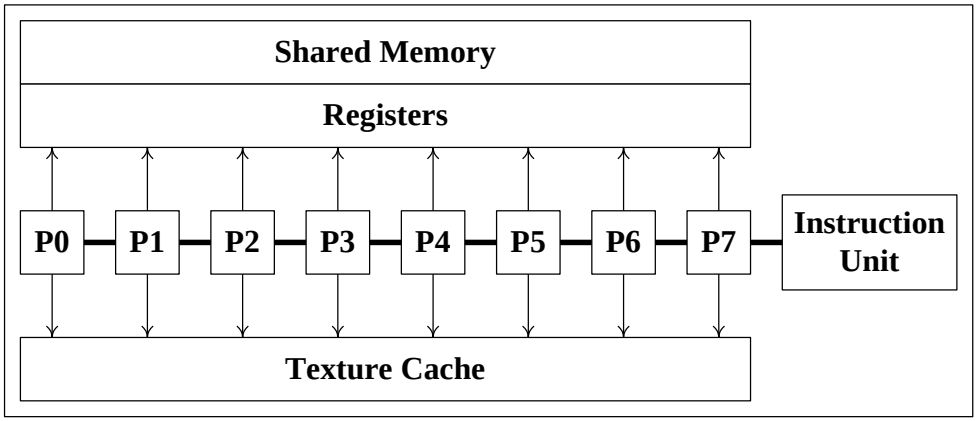
Тип памяти	Доступ	Уровень выделения	Скорость работы
Регистры	R/W	Per-thread	Высокая(on-chip)
Локальная	**R/W**	**Per-thread**	**Низкая (DRAM)**
Shared	R/W	Per-block	Высокая(on-chip)
Глобальная	**R/W**	**Per-grid**	**Низкая (DRAM)**
Constant	R/O	Per-grid	Высокая (L1 cache)
Texture	R/O	Per-grid	Высокая (L1 cache)

Пример многоядерных систем: архитектура G80 (2006 г)

Массив из потоковых мультипроцессоров



Streaming Multprocessor



Скорость доступа GTX-280

Тип памяти	Объем, Кб	Латентность, такты
Разделяемая память	16	40
Текстурный кеш L1	5	260
Текстурный кеш L2	256	370
Константный кеш L1	2	8
Константный кеш L2	8	81
Константный кеш L3	32	220
Кэш инструкций L1	4	-
Кэш инструкций L2	8	-
Кэш инструкций L3	32	-
Глобальная память	1024	450

Типы памяти в CUDA

- Самая быстрая – shared (on-chip) и регистры
- Самая медленная – глобальная (DRAM)
- Для ряда случаев можно использовать кэшируемую константную и текстурную память
- Доступ к памяти в CUDA идет отдельно для каждой половины warp’а (*half-warp*)

Работа с локальной памятью

- В архитектуре CUDA имеется аппаратное ограничение на количество используемых регистров одним потоков 63(CUDA < 3.0) и 255 (CUDA 3.5).
- Локальная память - это область в глобальной памяти, выделенная компилятором для хранения локальных значений потоков.
- Локальная память используется для хранения локальных данных потоков в случае нехватки регистров или объявления локальных массивов внутри ядра без ключевого слова __shared__.

Работа с константной памятью в CUDA

```
cudaError_t cudaMemcpyToSymbol(const char * symbol, const void *src,
                size_t count, size_t offset, enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyFromSymbol(void * dst, const char * symbol,
                size_t count, size_t offset, enum cudaMemcpyKind kind);
cudaError_t cudaMemcpyToSymbolAsync(const char * symbol, const void *src,
                size_t count, size_t offset, enum cudaMemcpyKind kind,
                cudaStream_t stream);
cudaError_t cudaMemcpyFromSymbolAsync(void * dst, const char * symbol,
                size_t count, size_t offset, enum cudaMemcpyKind kind,
                cudaStream_t stream);
```

Пример:

```
__constant__ float constData[256]; // константная память GPU
float              hostData[256]; // данные в памяти CPU

      // скопировать данные из памяти CPU в константную память GPU
cudaMemcpyToSymbol(constData, hostData, sizeof( data), 0,
cudaMemcpyHostToDevice);
```

Работа с глобальной памятью в CUDA без выравнивания

```
cudaError_t cudaMalloc      ( void ** devPtr, size_t size );
cudaError_t cudaFree        ( void * devPtr );
```

Доступ CPU к памяти:

```
cudaError_t cudaMemcpy(void * dst, const void * src, size_t size,
                       enum    cudaMemcpyKind kind);

cudaError_t cudaMemcpyAsync(void * dst, const void * src, size_t size,
                       enum    cudaMemcpyKind kind, cudaStream_t stream);

cudaError_t cudaMemcpy2D(void * dst, size_t dpitch, const void * src,
                       size_t spitch, size_t width, size_t height,
                       enum   cudaMemcpyKind kind, cudaStream_t stream);
cudaError_t cudaMemcpy2DAsync(void * dst, size_t dpitch, const void * src,
                       size_t spitch, size_t width, size_t height,
                       enum   cudaMemcpyKind kind, cudaStream_t stream);
```

Работа с глобальной памятью в CUDA с выравниванием

```
cudaError_t cudaMallocPitch ( void ** devPtr, size_t * pitch, size_t width,
                              size_t height );
```

Доступ к элементам матрицы:

```
T *item = (T*)((char*)baseAddress + row * pitch) + col;
```

Пример:

```
float * devPtr;              // pointer device memory
                             // allocate device memory
cudaMalloc ( (void **) &devPtr, 256*sizeof ( float );
                             // copy data from host to device memory
cudaMemcpy ( devPtr, hostPtr, 256*sizeof ( float ), cudaMemcpyHostToDevice );
                  // process data

                              // copy results from device to host
cudaMemcpy ( hostPtr, devPtr, 256*sizeof( float ), cudaMemcpyDeviceToHost );
                             // free device memory
cudaFree   ( devPtr );
```

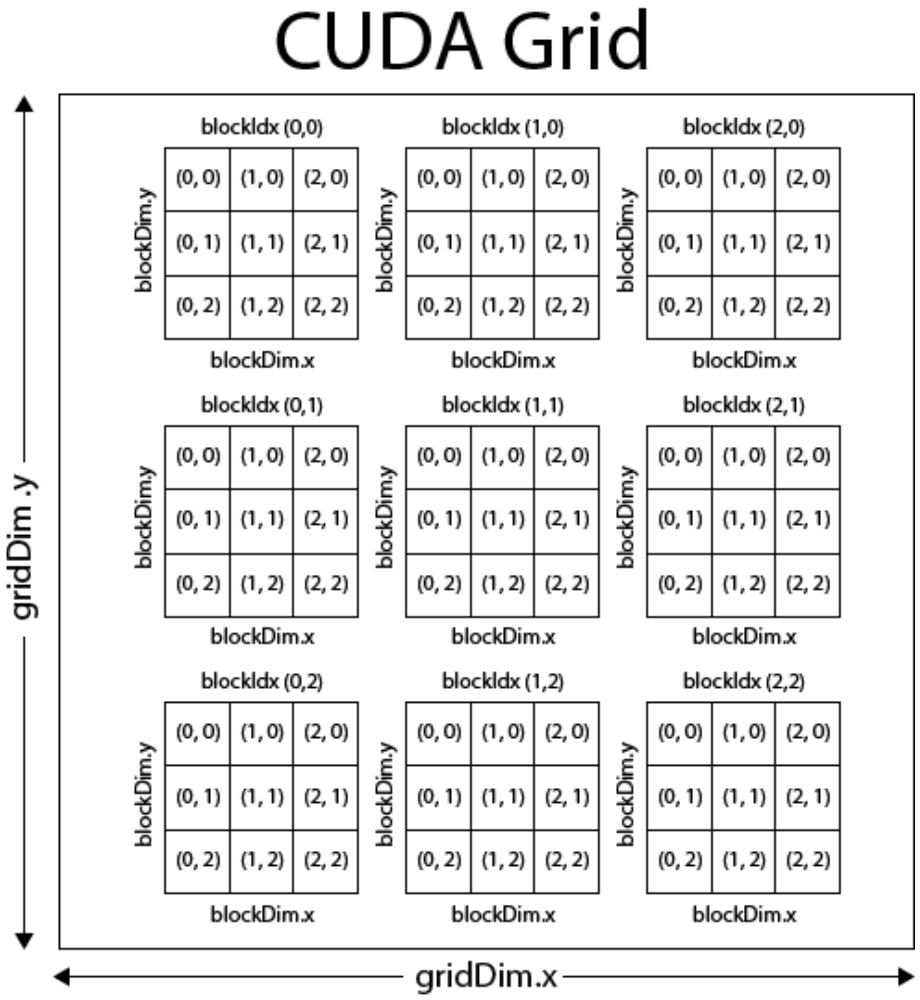
Пример: транспонирование двух матриц

Дана матрица A размером NxN

N кратна 16

Матрица расположена в глобальной памяти

Организуем нити в 2D-грид из блоков 16x16



Программная модель CUDA

```
__global__ void transpose1(float* inData, float* outData, int n)
{
  unsigned int xIndex = blockDim.x*blockIdx.x + threadIdx.x;
  unsigned int yIndex = blockDim.y*blockIdx.y + threadIdx.y;
  unsigned int inIndex = xIndex + n*yIndex;
  unsigned int outIndex = yIndex + n*xIndex;

  outData[outIndex] = inData[inIndex];
}
```

Пример: перемножение двух матриц

Даны матрицы A и B размером NxN

N кратна 16

Матрица расположена в глобальной памяти

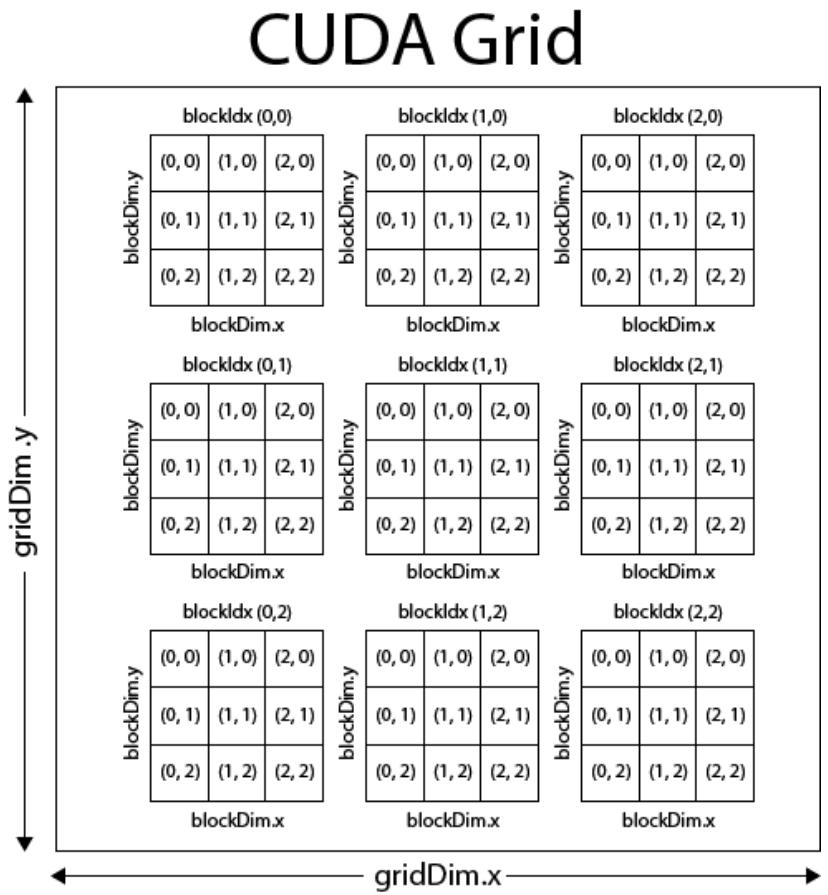
По одной нити на каждый элемент произведения

Организуем нити в 2D-грид из блоков 16x16

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

Пример: перемножение двух матриц. Использование глобальной памяти

```
__global__ void kernel ( const float * a, const float * b, int n, float * c )
{
   int bx = blockIdx.x;  // индексы блока
   int by = blockIdx.y;  //

   int tx = threadIdx.x;  // индексы нити внутри блока
   int ty = threadIdx.y;  //

   float sum = 0.0f;

   // смещение для a[i][0]
   int ia = n * BLOCK_SIZE * by + n * ty;

   // смещение для b[0][i]
   int ib =  BLOCK_SIZE * bx +  tx;

   // смещение для результата
   int ic= n * BLOCK_SIZE * by+ BLOCK_SIZE * bx;

   // перемножаем и суммируем
   for( int k = 0; k < n; k++ )
    sum+= a [ia+ k] * b [ib+ k*n];

   c [ic+ n * ty+ tx] = sum; // запоминаем результат
}
```



Оптимизация работы с памятью в CUDA

- Использование выравнивания
- Максимальное использование shared-памяти
- Объединение нескольких запросов к глобальной памяти в один (coalescing)
- Использование специальных паттернов доступа к памяти, гарантирующих эффективный доступ
  - Паттерны работают независимо в пределах каждого half-warp’а

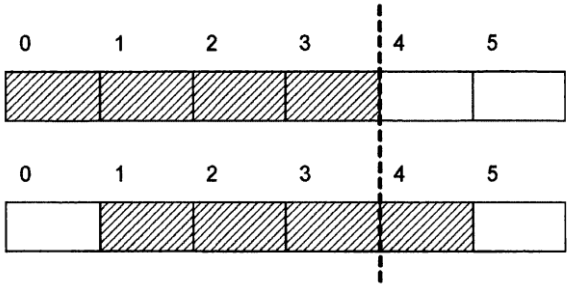
Использование выравнивания

```
struct vec3
{
  float x, y, z;
};
```

- Размер равен 12 байт
- Элементы массива не будут выровнены в памяти

```
struct __align__(16) vec3
{
  float x, y, z;
};
```

- Размер равен 16 байт
- Элементы массива всегда будут выровнены в памяти



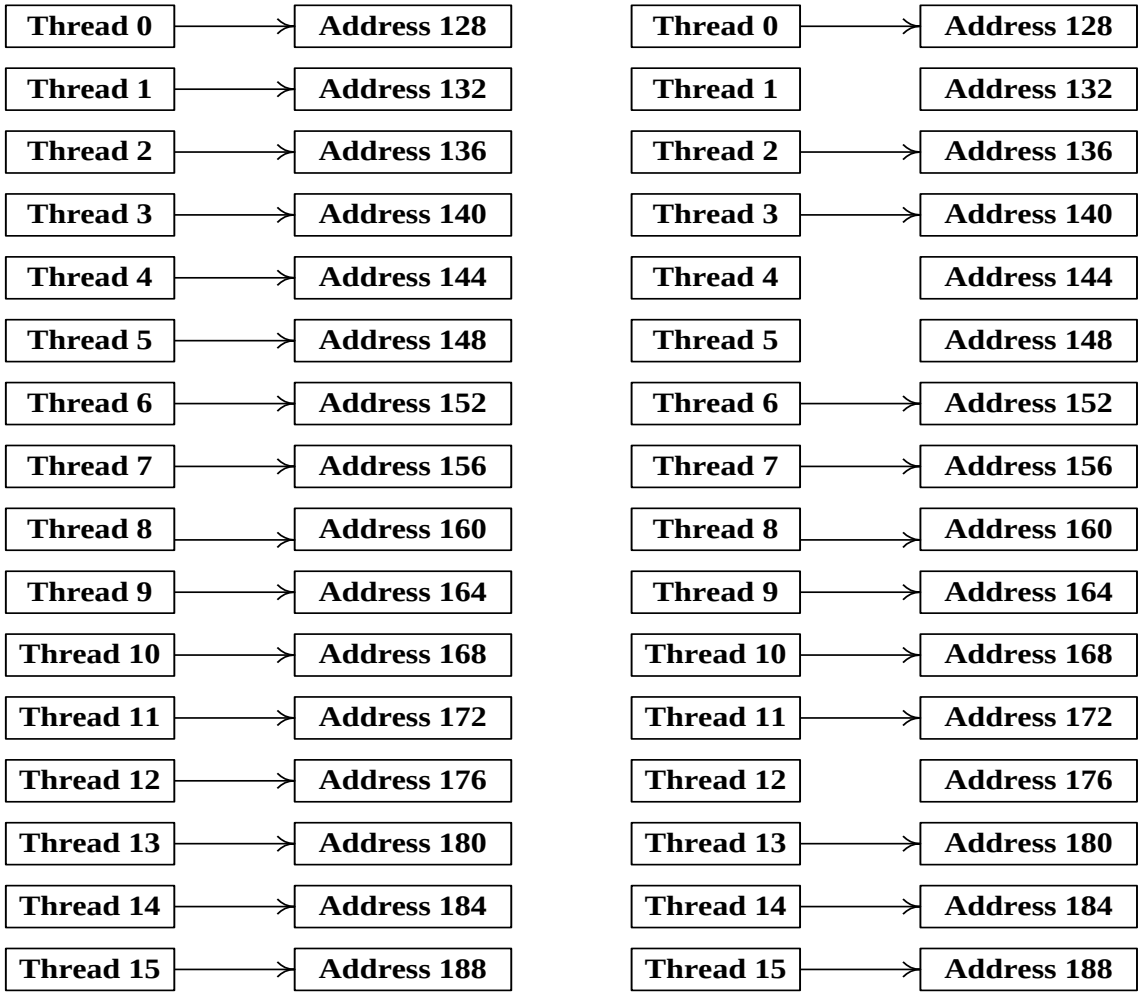
Объединение запросов к глобальной памяти

- GPU умеет объединять ряд запросов к глобальной памяти в один блок (транзакцию)
- Независимо происходит для каждого half-warp’а
- Длина блока должна быть 32/64/128 байт
- Блок должен быть выровнен по своему размеру

Объединение (coalescing) для GPU с CC 1.0/1.1

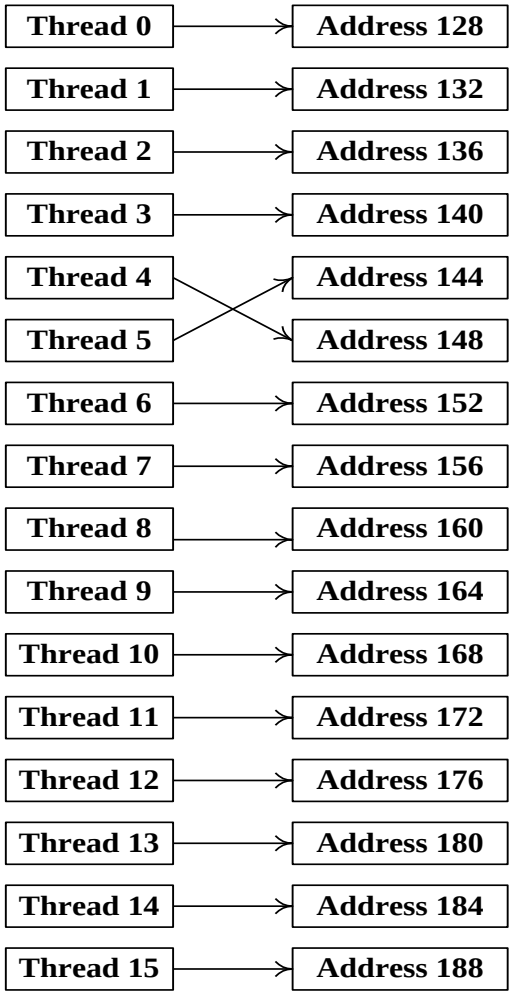
- Нити обращаются к
  - 32-битовым словам, давая 64-байтовый блок
  - 64-битовым словам, давая 128-байтовый блок
- Все 16 слов лежат в пределах блока
- *k*-ая нить half-warp'а обращается к *k*-му слову блока

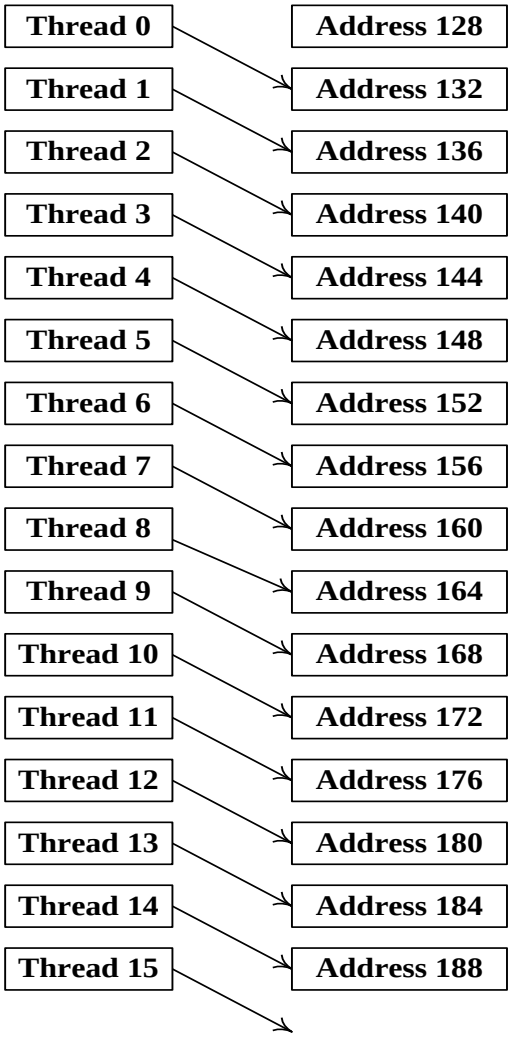
Объединение (coalescing) для GPU с CC 1.0/1.1



Coalescing

Объединение (coalescing) для GPU с CC 1.0/1.1





Not Coalescing

Доступ к структурам

```
struct A __align__(16)
{
    float a;
    float b;
    uint  c;
};

A array [1024];

. . .
A a = array [threadIdx.x];
```

```
float a [1024];
float b [1024];
uint  c [1024];
. . .
float fa = a [threadIdx.x];
float fb = b [threadIdx.x];
uint  uc = c [threadIdx.x];
```

Работа с разделяемой памятью

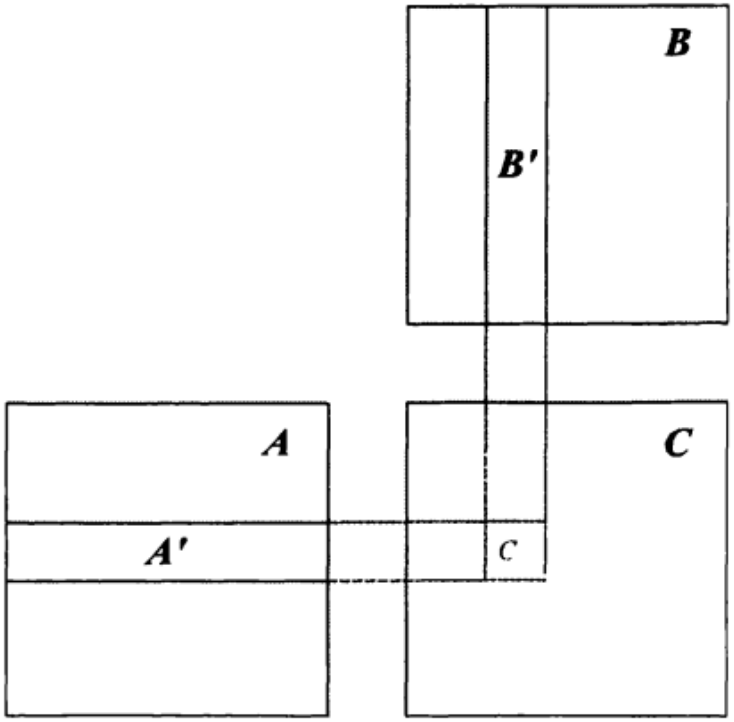
1. Явное задание размера массива выделяемой памяти:

```
__global__ void incKernal(float * a)
{
     //явно задаем выделение 256*4 байтов на блок
  __shared__ float buf[256];
     // запись значения из глобальной памяти в разделяемую
 buf[threadIdx.x] = a[blockIdx.x*blockDim.x + threadIdx.x];
}
```

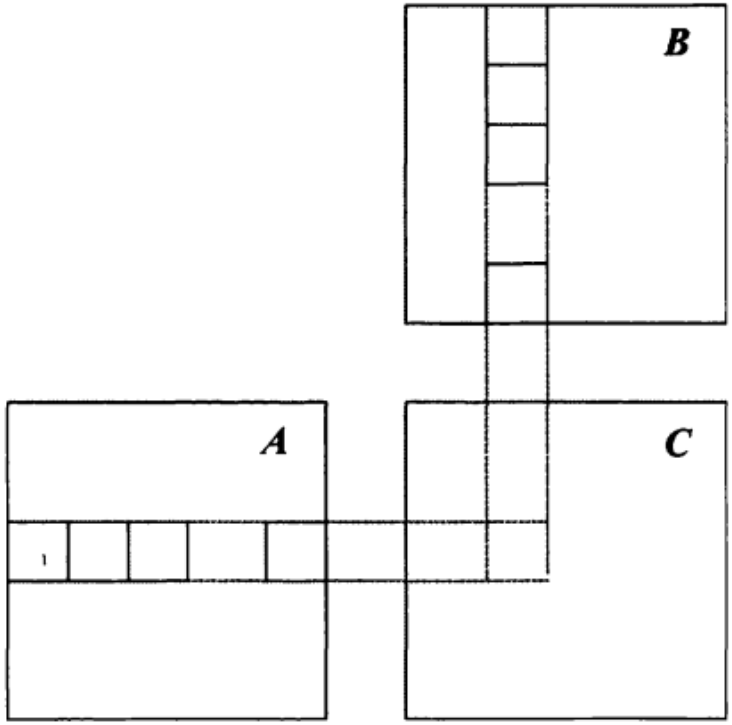
2. Задание размера массива при запуске ядра:

```
__global__ void kernal(float * a)
{
 __shared__ float buf[]; //размер явно не указан
     // запись значения из глобальной памяти в разделяемую
 buf[threadIdx.x] = a[blockIdx.x*blockDim.x + threadIdx.x];
 ….
}
// запустить ядро и задать выделяемый объем разделяемой памяти
 kernel<<<dim3(n/256), dim(256), k*sizeof(float)>>> (a);
```

Пример: перемножение двух матриц. Использование разделяемой памяти



Для вычисления C’ нужны A’ и B’



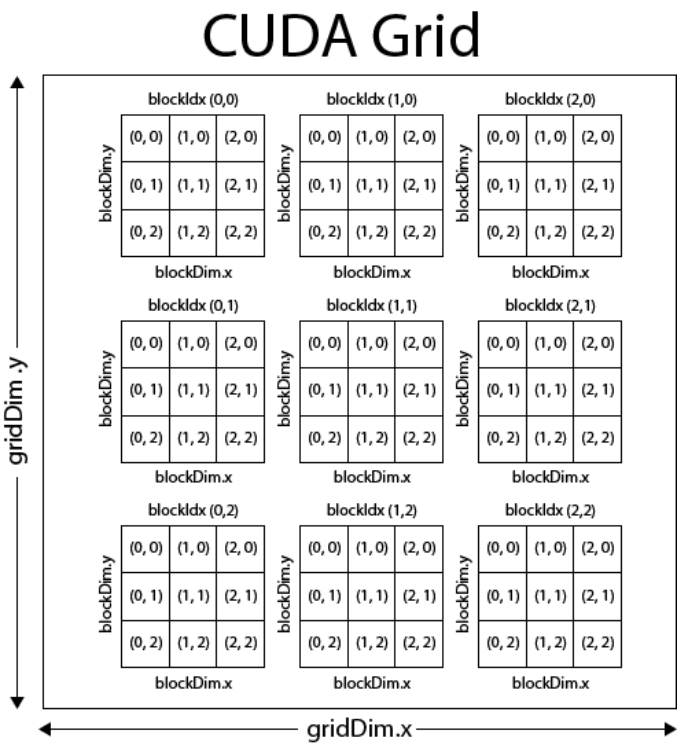
Разложение требуемой подматрицы в сумму произведений матрицы 16x16

$$C' = A'_1 \times B'_1 + A'_2 \times B'_2 + \ldots + A'_{N/16} \times B'_{N/16}.$$

Пример: умножение двух матриц. Использование разделяемой памяти

```
__global__ void kernel(const float * a, const float * b, int n, float * c)
{
   int bx = blockIdx.x;  // индексы блока
   int by = blockIdx.y;  //

   int tx = threadIdx.x;  // индексы нити внутри блока
   int ty = threadIdx.y;  //


int          aBegin = n * BLOCK_SIZE * by;
int          aEnd = aBegin + n -1;

int aStep = BLOCK_SIZE;

int          bBegin = bx * BLOCK_SIZE;
int bStep = BLOCK_SIZE*n;

float        sum = 0.0f;

for( int     ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )
{
__shared__ float          as [BLOCK_SIZE][BLOCK_SIZE];
__shared__ float          bs [BLOCK_SIZE][BLOCK_SIZE];
as [ty][tx] = a [ia + n * ty + tx];
bs [ty][tx] = b [ib + n * ty + tx];

__syncthreads (); // Убедимся, что подматрицы полностью загружены
for( int     k = 0; k < BLOCK_SIZE; k++ )
sum += as [ty][k] * bs [k][tx];
__syncthreads(); // Убедимся, что подматрицы никому больше не нужны
}
c [n * BLOCK_SIZE * by + BLOCK_SIZE * bx + n * ty + tx] = sum;
}
```

Ускорение в 18 раз



Число чтений из глобальной памяти 2N/16

Эффективная работа с shared-памятью

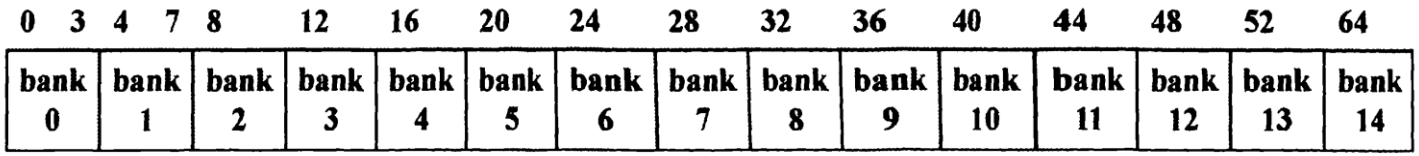
- Для повышения пропускной способности вся shared-память разбита на 16 (CUDA 1.x) или 32 (CUDA 2.x/3.x) банков
- Каждый банк работает независимо от других
- Можно одновременно выполнить до 16 обращений к shared-памяти
- Если идет несколько обращений к одному банку, то они выполняются последовательно

- Определение номера банка:

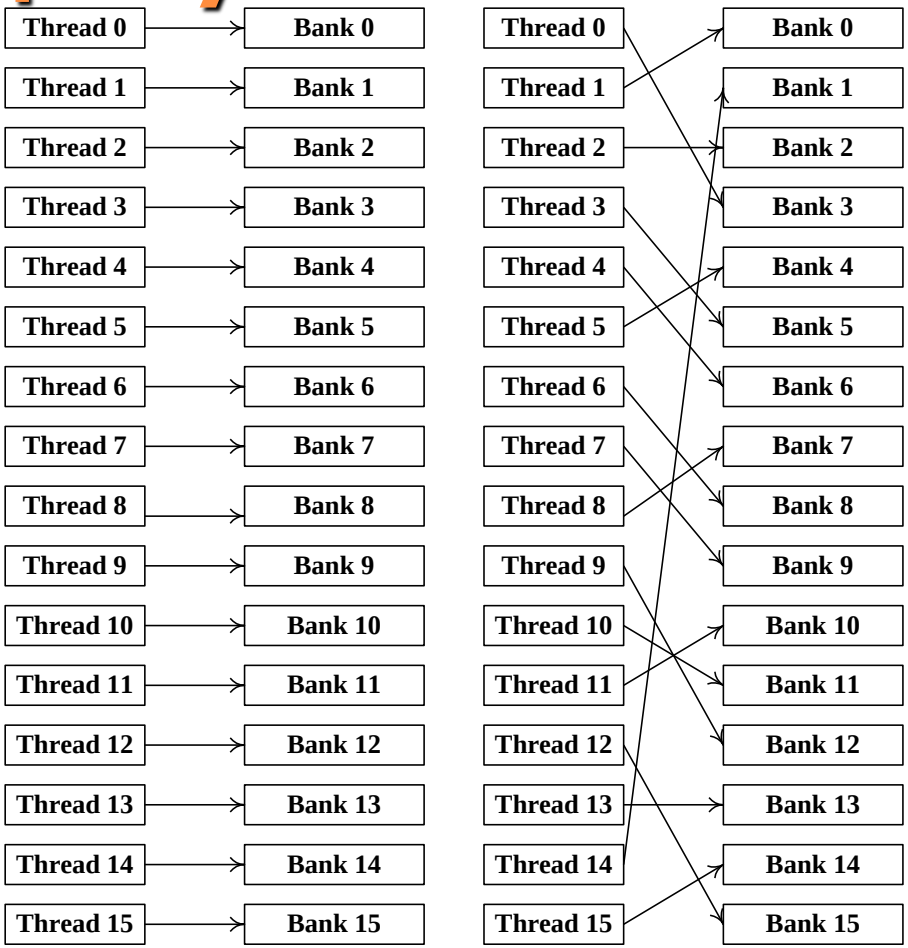
  Номер банка = (Адрес в байтах/4)%32 – для устройства версии 2.0
  Номер банка = (Адрес в байтах/4)%16 – для устройства версии 1.x

Эффективная работа с shared-памятью

- Банки строятся из 32-битовых слов
- Подряд идущие 32-битовые слова попадают в подряд идущие банки
- ***Bank conflict*** – несколько нитей из одного half-warp'а (CUDA 1.x) или из одного warp'а (CUDA 2.x/3.x) обращаются к одному и тому же банку
- Конфликта не происходит если все 16 нитей обращаются к одному слову (*broadcast*)

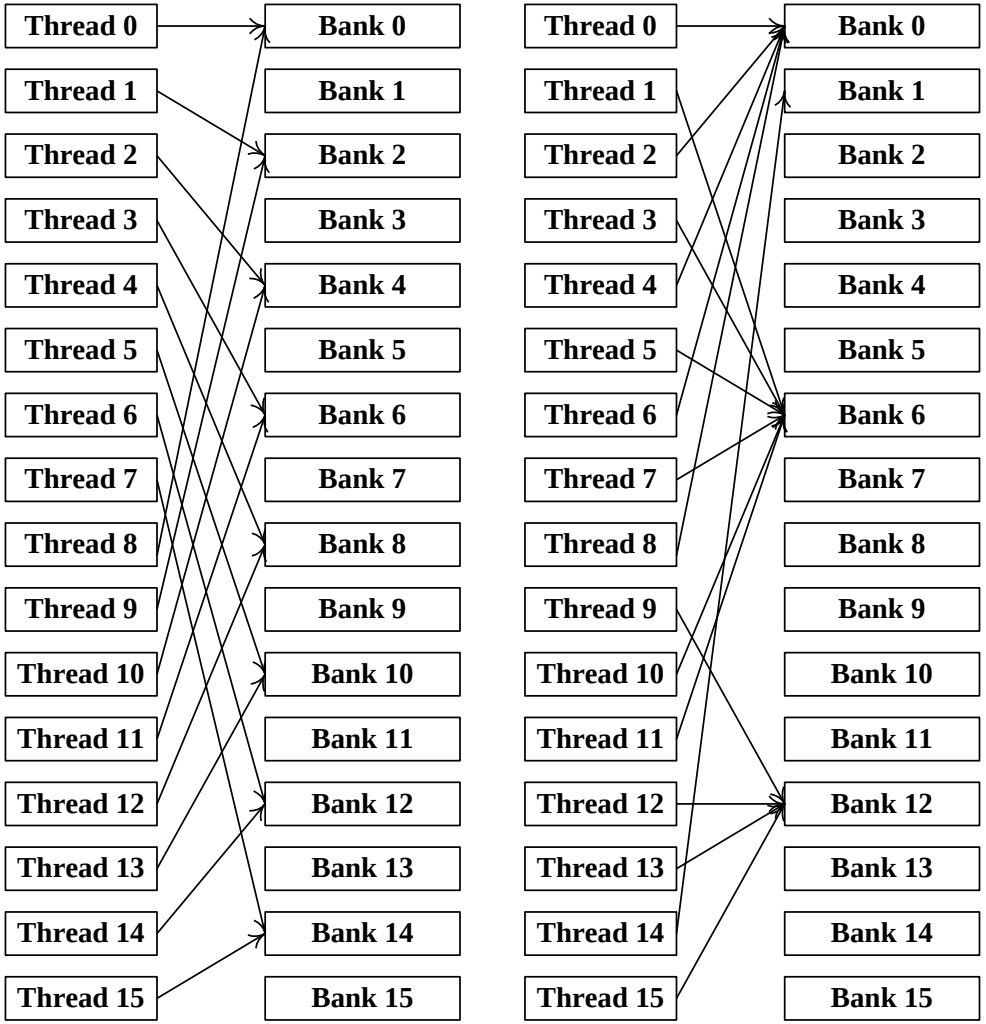


Бесконфликтные паттерны доступа




```
__shared__ float  buf [128];  // Объявили массив в разделяемой памяти.
                              // Каждая нить обращается к своему 32-битовому слову.
float             v = buf [baseIndex + threadIdx.x ];
```

Паттерны с конфликтами банков

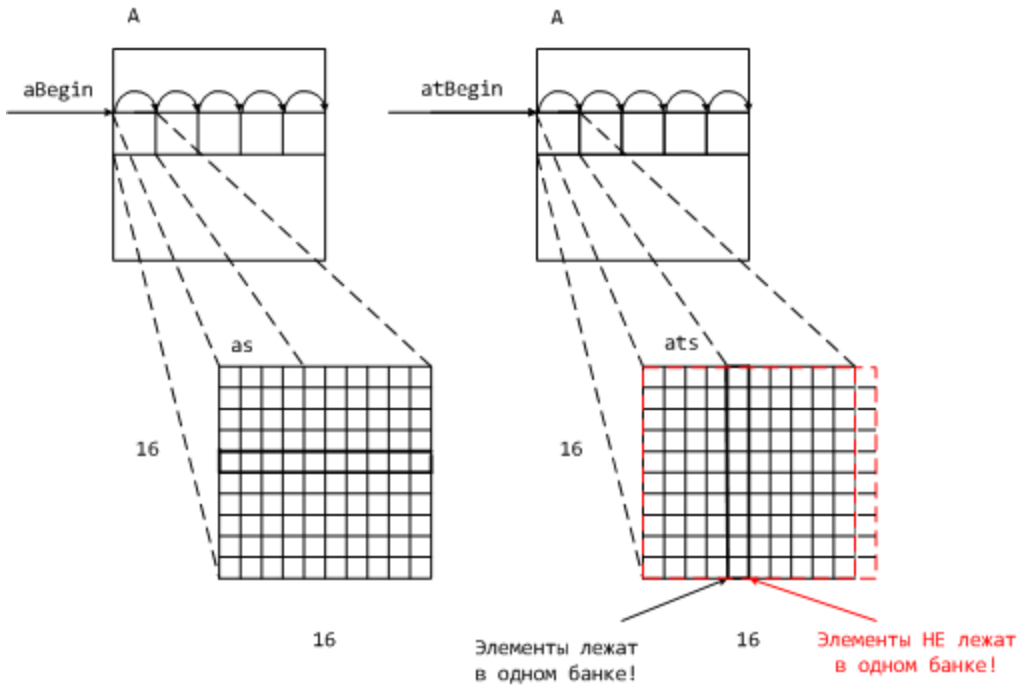


- **Слева – конфликт второго порядка – вдвое меньшая скорость**
- **Справа - несколько конфликтов, до 6-го порядка**

Пример: умножение матрицы на транспонированную.Использование разделяемой памяти

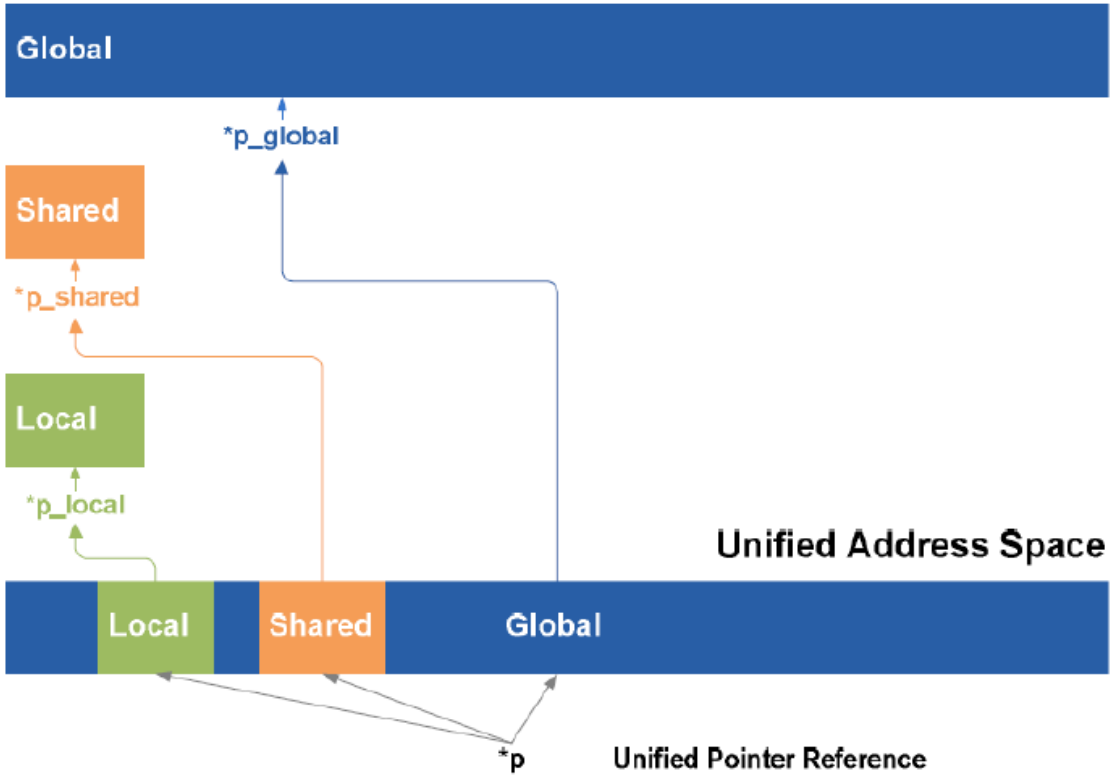
```
__global__ void kernel (float * a, int n, float * c )
{
   int bx = blockIdx.x;  // индексы блока
   int by = blockIdx.y;  //
   int tx = threadIdx.x;  // индексы нити внутри блока
   int ty = threadIdx.y;  //

   //индекс первой подматрицы A, обрабатываемой блоком
   int aBegin = n*BLOCK_SIZE*by;
   int aEnd = aBegin+n-1;

   //индекс первой подматрицы B, обрабатываемой блоком
   int    atBegin = n * BLOCK_SIZE * bx;
   float  sum = 0.0f;
   for( int    ia = aBegin, iat = atBegin; ia <= aEnd; ia += BLOCK_SIZE, iat += BLOCK_SIZE )
     {
          __shared__ float    as [BLOCK_SIZE][BLOCK_SIZE];
          __shared__ float    ats [BLOCK_SIZE][BLOCK_SIZE];
          as [ty][tx] = a [ia + n * ty + tx];
          ats [ty][tx] = a [iat + n * ty + tx];
          __syncthreads (); // Убедимся, что подматрицы полностью загружены
          for( int  k = 0; k < BLOCK_SIZE; k++ )
               sum += as [ty][k] * ats [tx][k];
          __syncthreads(); // Убедимся, что подматрицы никому больше не нужны
     }
     c [n * BLOCK_SIZE * by + BLOCK_SIZE * bx + n * ty + tx] = sum;
}
```

Пример: умножение матрицы на транспонированную.Использование разделяемой памяти



Единое адресное пространство

Начиная с CUDA 2.x реализовано единое адресное пространство.
Множество адресов поделено на участки
- локальной
- разделяемой
- глобальной памяти.



Существенно упрощает программирование алгоритмов с адресацией, зависящей от данных.

Влияние на производительность

- **Количество вычислительных потоков.**

От размера сетки блоков потоков и самого блока потоков зависит степень загруженности планировщиков графического ускорителя. Чем больше warp планирует планировщик, тем больше у него возможностей скрыть задержки, связанные с обращением в глобальную память. Кроме того, достаточно большие размеры сетки и блока обеспечат эффективное масштабирование на новых графических ускорителях без переписывания и перекомпилирования программы. Обычно рекомендуется порождать порядка 10^{5} потоков за один запуск ядра.

- **Равномерность загрузки вычислительных потоков.**

Неравномерность загрузки вычислительных потоков - очень частое явление, например, вычисления значений во внутренних и граничных узлах сетки явной разностной схемы обычно различаются. Однако, неравномерность загрузки потоков может привести к существенной деградации производительности. Во-первых это связанно с тем, что блок потоков завершается и освобождает мультипроцессор, когда все его потоки завершили исполнение. Во-вторых, запуск ядра завершается, когда все блоки потоков завершили исполнение.

Влияние на производительность

- **Преобладание вычислений по отношению к загрузкам данных.**

Для повышения производительности необходимо максимизировать количество производимых вычислений на единицу загружаемых данных из глобальной памяти. Минимизировать количество доступов в память помогает использование разделяемой памяти и кэшей, за счёт переиспользования одним и тем же или другими потоками близко расположенных данных.

- **Локальность загружаемых данных.**

Этот фактор напрямую следует из предыдущего. Увеличение локальности загружаемых данных на уровне warp позволяет уменьшить количество и объём транзакций в память и повысить эффективность работы кэшей. Увеличение локальности загружаемых данных на уровне блока потоков позволяет увеличить эффективность использования разделяемой памяти и работы кэшей.

Влияние на производительность

- **Деление** warp’ов **на условных переходах.**

В случае, когда разные потоки одного warp’а разбиваются по разным ветвям условного перехода, время исполнения условного перехода складывается из времён исполнения его ветвей. Следовательно, частое деление варпов по ветвям приводит к деградации производительности. Степень деградации зависит как от количества делений warp’ов, так и от размера ветвей условного перехода.

Требования к алгоритмам

- Преобладание вычислений по отношению к загрузкам данных из памяти
- Малая доля условных переходов
- Равномерность загрузки вычислительных потоков
- Локальность данных, с которыми будут работать потоки одного блока потоков
- Достаточное количество вычислительных потоков для полной загрузки планировщиков всех имеющихся мультипроцессоров (обычно более 10^5 потоков)

Лабораторная работа № 2

- Оптимизация алгоритма умножения матриц (по вариантам).

- Оптимизация алгоритма транспонирования матриц (по вариантам).

- Сравнительный график эффективности алгоритмов: CPU, GPU, GPU с оптимизацией в зависимости от размера данных.