



Applied Parallel Computing
parallel-computing.pro

Иерархия памяти CUDA

Шевченко Александр

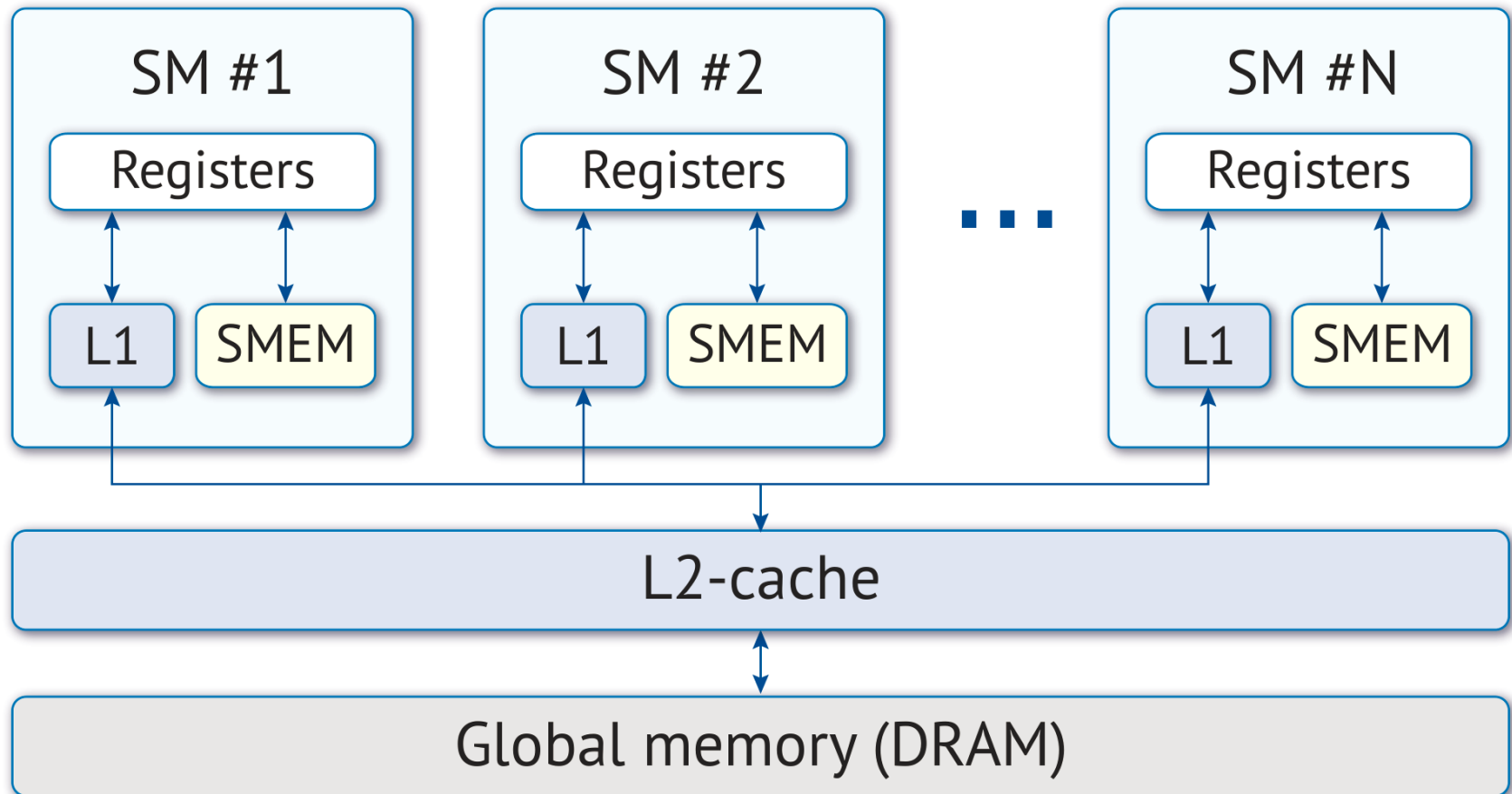


Типы памяти

Тип памяти	Расположение	Доступ	Види- мость	Время жизни
Регистры	SM	R/W	Нить	Нить
Локальная	DRAM GPU	R/W	Нить	Нить
Глобальная	DRAM GPU	R/W		
Разделяемая	SM	R/W	Блок	Блок
Константная	DRAM GPU	R/O		
Текстурная	DRAM GPU	R/O		
Общее адресное пространство	DRAM Host	R/W	Везде	



Иерархия памяти






Applied Parallel Computing
parallel-computing.pro

Регистровый файл



Регистровый файл

- В зависимости от Compute Capability, каждый мультипроцессор содержит **32768 (2.x-3.0)**, **65536 (3.5, 5.2)** 32-битных регистров.
- Доступ к регистрам других нитей запрещён. 
- Регистры распределяются между нитями блоков **во время компиляции.**



Регистровый файл

- В регистры по умолчанию попадают все переменные, которые создаются при исполнении функций на GPU

```
__global__ void sum_kernel(int *A, int *B, int *C) {  
    int threadIdx =  
        blockIdx.x * blockDim.x + threadIdx.x; //определить свой индекс  
    int elemA = A[threadIdx]; //считать нужный элемент A  
    int elemB = B[threadIdx]; // считать нужный элемент B  
    C[threadIdx] = elemA + elemB; //записать результат суммирования  
}
```

- Если регистров недостаточно, данные буферизуются в локальной памяти



Applied Parallel Computing
parallel-computing.pro

Локальная память



Локальная память

- Используется в случае нехватки регистров
- Расположена в DRAM GPU.
Время доступа: 400-800 тактов.
- Не может быть использована явно на уровне CUDA-программы
- Обладает упрощенной схемой адресации
- Негативный эффект может быть сглажен кешами





Applied Parallel Computing
parallel-computing.pro

Глобальная память



Глобальная память

- ❶ Основное хранилище данных GPU
- ❷ Наиболее медленная из всех типов памяти на GPU
- ❸ Функции работы с глобальной памятью
 - `cudaMalloc`
 - `cudaFree`
 - `cudaMemcpy`



Глобальная память. Цифры

- Размер глобальной памяти
 ≤ 12 Гб
- Скорость передачи данных через PCI-E 3.0:
6-10 Гб/с
- Скорость чтения из глобальной памяти:
150-250 Гб/с
- Ширина шины памяти DRAM GPU
до 384 бит



Произведение матриц

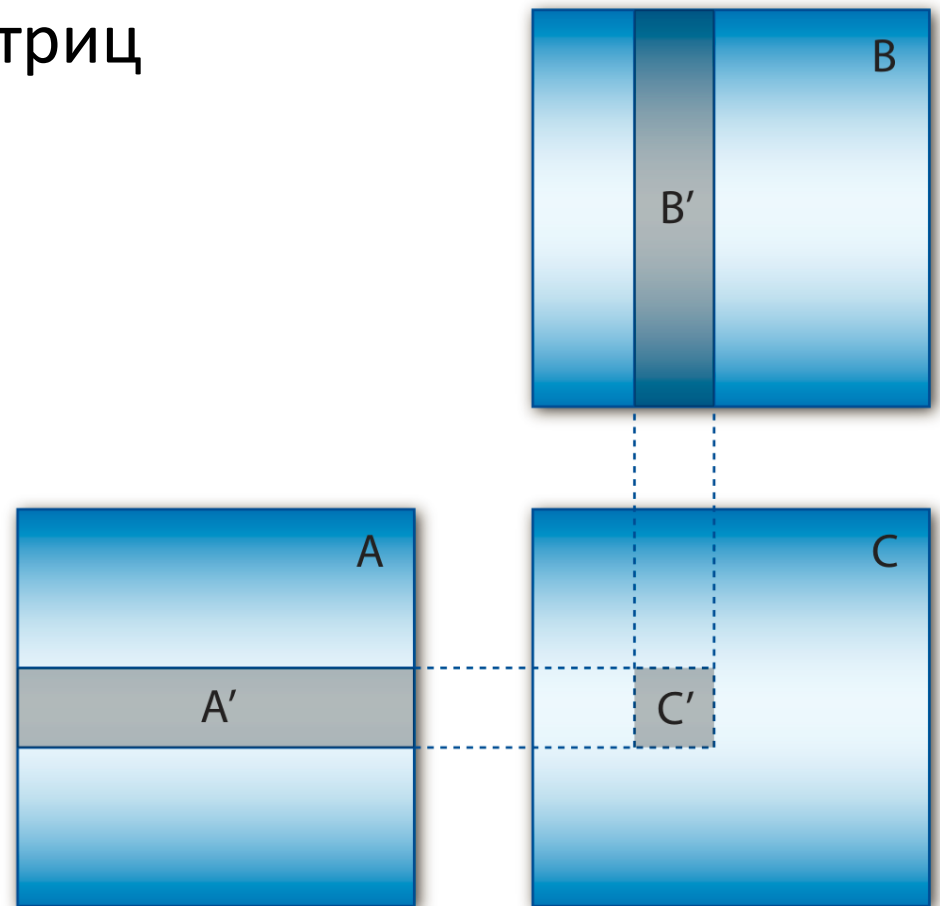
- Произведение двух матриц одинакового размера

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

- Используются **одномерные** массивы

a[i][j] =>

a[i * n + j]





Умножение матриц. Реализация

```
__global__ void matmul1 (float* a, float* b, int n, float* c){  
    // Индексы блока  
    int bx = blockIdx.x, by = blockIdx.y;  
    // Индексы нити внутри блока  
    int tx = threadIdx.x, ty = threadIdx.y;  
    // Смещение для a[i][0]  
    int ia = n * (BLOCK_SIZE * by + ty);  
    // Смещение для b[0][j]  
    int ib = BLOCK_SIZE * bx + tx;  
  
    // Переменная для накопления результата  
    float sum = 0.0f;  
    // Перемножить строку и столбец  
    for (int k = 0; k < n; k++)  
        sum += a [ia + k] * b [ib + k * n];  
    // Смещение для записываемого элемента  
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
    // Сохранить результат в глобальной памяти  
    c[ic + n * ty + tx] = sum;  
}
```



Дополнение по массивам

- Можно переложить на компилятор вопросы работы с многомерными массивами:

```
float* ptr = malloc(sizeof(float) * nx * ny);  
float (*arr)[nx] = (float(*)[nx])ptr;  
arr[2][3] = 1.0f;
```



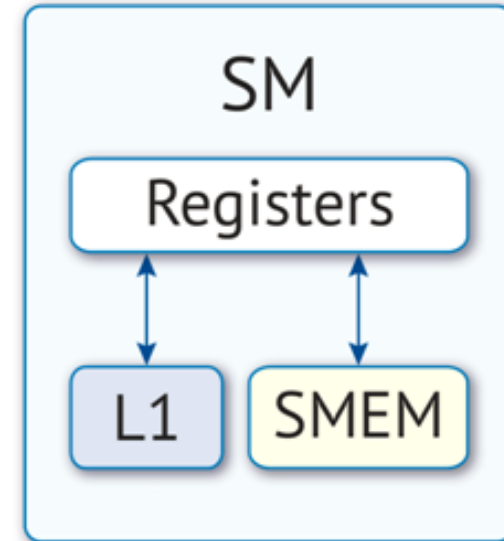
Applied Parallel Computing
parallel-computing.pro

Разделяемая память



Разделяемая память

- ❶ Располагается на мультипроцессоре
=> быстрая
- ❷ Делит пространство с L1 кэшем
 - 48 Кбайт (16 Кбайт L1 кэш)
 - 32 Кбайт (32 Кбайт L1 кэш)
 - 16 Кбайт (48 Кбайт L1 кэш)
- ❸ Является программно управляемым кэшем.
Одно из **самых важных** средств оптимизации





Выделение разделяемой памяти

Статически

- В GPU коде статический массив или переменная:

```
__shared__ int f[32]; //массив  
__shared__ float sum; //переменная
```

Динамически

- В GPU коде динамический массив:

```
extern __shared__ int g[];
```

- Размер указывается при запуске ядра:

```
kernel <<<100, 100, 40*sizeof(int)>>>();
```



Особенности использования

- Переменные, помеченные shared :
 - При объявлении вне функций ведут себя как статические (размер должен быть указан)
 - Приватны для каждого блока
 - Существуют только на время жизни блока
 - Не могут быть инициализированы при определении



Особенности использования

- Одновременно могут использоваться статические и динамические переменные в shared-памяти

```
__global__ void kernel () {  
    __shared__ float f;  
    extern __shared__ int buf [];  
}  
  
...  
kernel<<<10, 15, 30*sizeof(int)>>> ();
```



Особенности использования

- ❶ Все динамические указатели будут указывать на начало динамически выделенного блока

```
__global__ void kernel() {  
    extern __shared__ int buf1 [];  
    extern __shared__ int buf2 [];  
    buf2 = buf1 + 10;  
}  
  
...  
kernel<<<10, 15, 30*sizeof(int)>>>();
```



Синхронизация

`void __syncthreads ()`

- Локальная барьерная синхронизация для всех нитей блока
- Может вызываться только из Device-кода
- Может вызываться в условном операторе только если все нити блока дойдут до этой строчки
- Стоит вставлять синхронизацию между операциями чтения и записи в разделяемую память





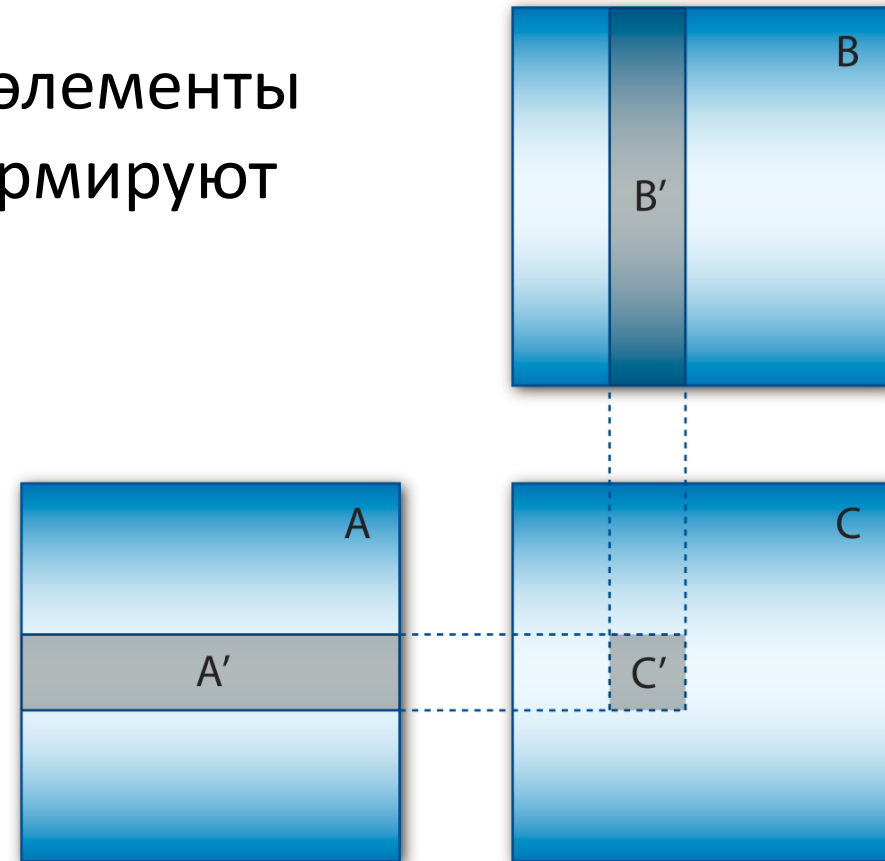
Типичное использование

- ④ Нити блока совместно загружают данные в разделяемую память
- ④ `__syncthreads ()`
- ④ Работа с разделяемой памятью
 - В том числе синхронизации, если нужно
- ④ `__syncthreads ()`
- ④ Выгрузка результатов расчета в глобальную память



Умножение матриц

- Постоянно используются элементы матриц A и B , которые формируют полосы
- Размер полос для реальных задач превышает размер разделяемой памяти



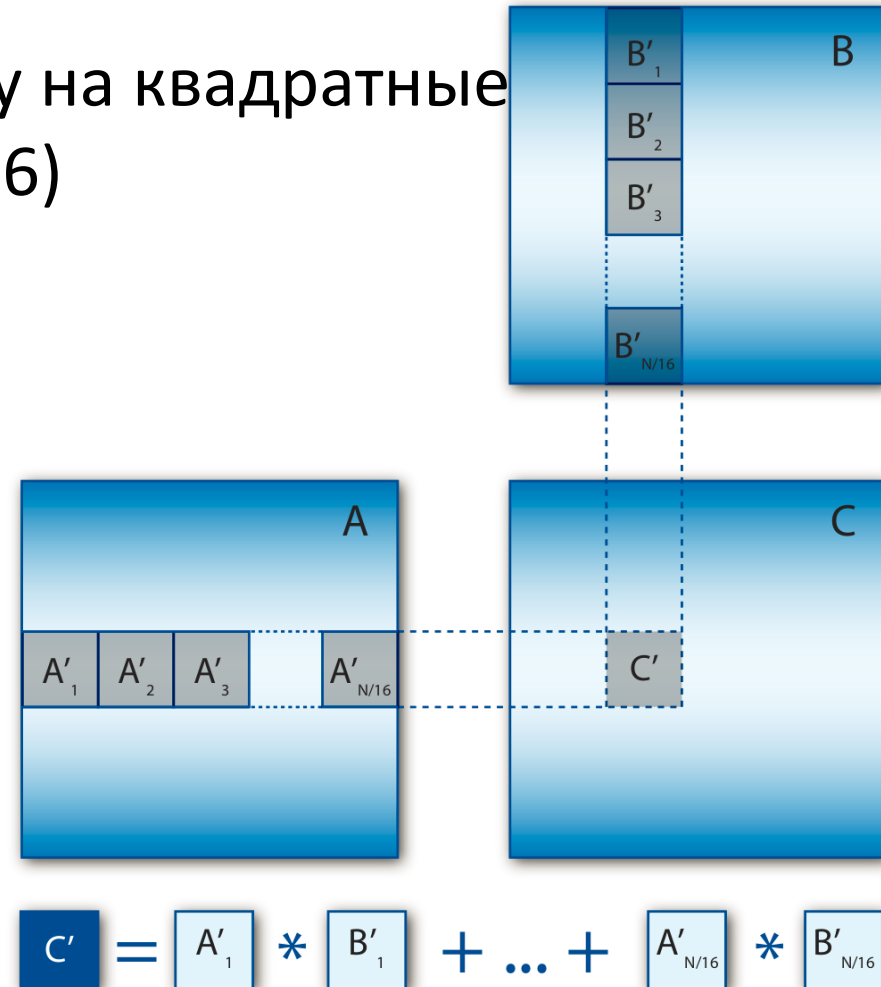


Умножение матриц

- Разбиваем каждую полосу на квадратные матрицы (например, 16x16)

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

- Для работы нужны всего две матрицы 16x16 в разделяемой памяти





Умножение матриц

- ⊗ На каждый элемент результата:
- ⊗ Прошлая реализация:
 - $2N$ арифметических операций
 - $2N$ обращений к глобальной памяти
- ⊗ Реализация с разделяемой памятью
 - $2N$ арифметических операций
 - $2N / 16$ обращений к глобальной памяти
 - Нужна явная синхронизация



Умножение матриц

```
__global__ void matmul2(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; // Индекс начала первой  
                                       // подматрицы A  
  
    int aEnd = aBegin + n - 1;  
    int aStep = BLOCK_SIZE;           // Шаг перебора подматриц A  
    int bBegin = BLOCK_SIZE * bx;     // Индекс первой подматрицы B  
    int bStep = BLOCK_SIZE * n;       // Шаг перебора подматриц B  
    float sum = 0.0f;                 // Вычисляемый элемент C'  
  
    __shared__ float as [BLOCK_SIZE] [BLOCK_SIZE];  
    __shared__ float bs [BLOCK_SIZE] [BLOCK_SIZE];
```



Умножение матриц

```
for (int ia = aBegin, ib = bBegin; ia <= aEnd;
    ia += aStep, ib += bStep) { // Цикл по всем подматрицам
    // Загрузить по одному элементу A и B в shared-память.
    as [ty][tx] = a [ia + n * ty + tx];
    bs [ty][tx] = b [ib + n * ty + tx];
    // Дождаться, пока обе матрицы будут загружены
    __syncthreads();
    // Вычислить элемент произведения загруженных подматриц.
    for (int k = 0; k < BLOCK_SIZE; k++)
        sum += as [ty][k] * bs [k][tx];
    // Дождаться пока все нити блока вычислят свои элементы
    __syncthreads();
}
int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
c [ic + n * ty + tx] = sum; // Записать результат
}
```



Быстродействие

Метод	Время, мс
Без использования разделяемой памяти	324.63
С использованием разделяемой памяти	93.26
С использованием CUBLAS	30.84

Время решения задачи 2048x2048, GPU: Tesla C2070



Спасибо за внимание!

Шевченко Александр
aleksandr.shevchenko@phystech.edu