МГТУ
им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

ПРОГРАММИРОВАНИЕ CUDA C/C++, АНАЛИЗ ИЗОБРАЖЕНИЙ И DEEP LEARNING

Лекция №2

Спасёнов Алексей

Иерархия памяти в CUDA

1. Типы памяти в CUDA
2. Регистры и локальная память
3. Работа с глобальной памятью
4. Оптимизация работы с глобальной памятью
5. CUDA-потоки
6. Использование pinned-памяти
7. Разделяемая память
8. Оптимизация работы с разделяемой памятью

Основные понятия CUDA

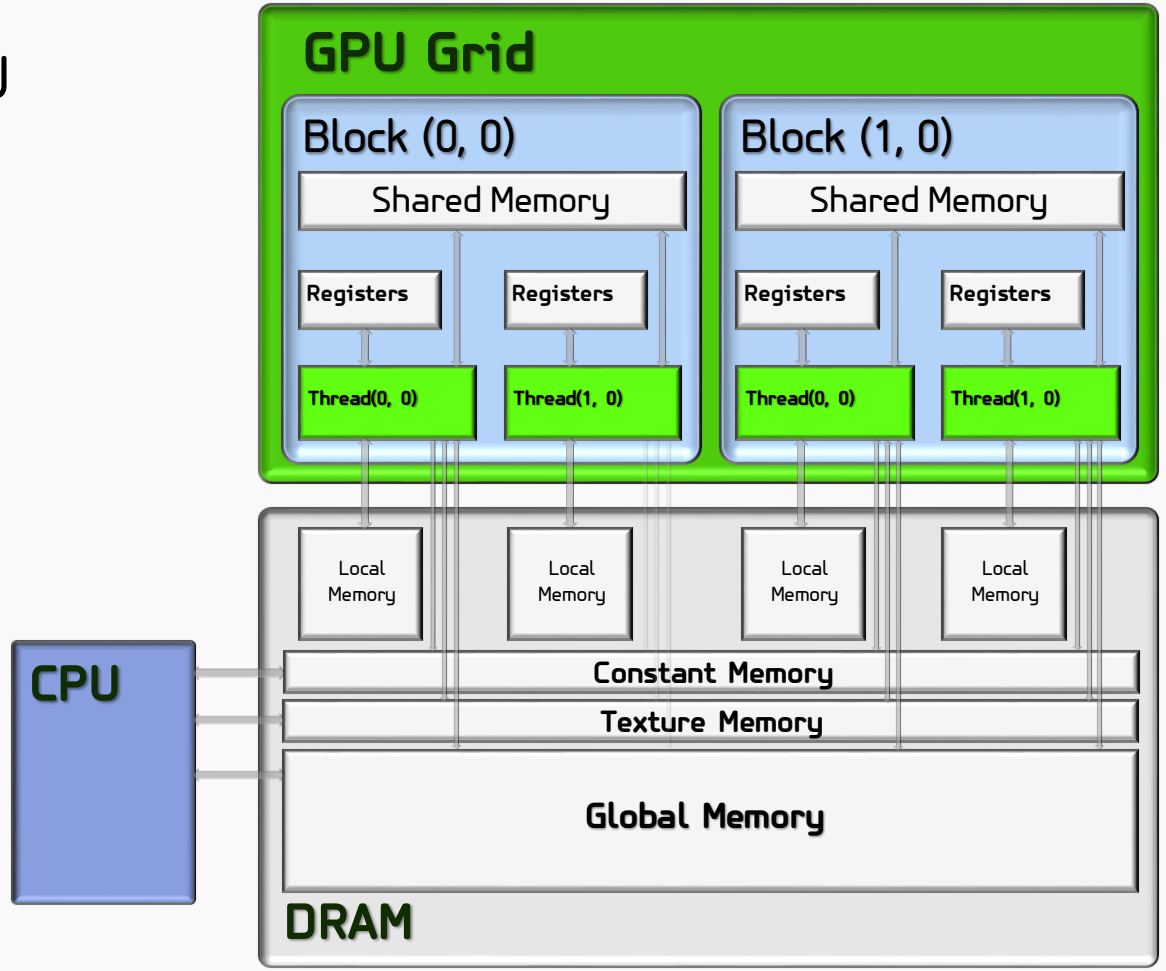
- Kernel – Параллельная часть алгоритма, выполняемая на Grid
- Grid – Объединение блоков, которые выполняются на одном устройстве
- Block – Объединение потоков, которое выполняется целиком на одном SM.
  Имеет свой уникальный идентификатор внутри Grid.
- Thread – единица выполнения программы.
  Имеет свой уникальный идентификатор внутри Block.
- Warp – 32 последовательно идущих Thread

Типы памяти в CUDA

Тип памяти	Расположение	Кешируется	Доступ	Уровень доступа	Время жизни
Регистры	Мультипроцессор	Нет	R/w	Per-thread	Нить
Локальная	DRAM	Нет	R/w	Per-thread	Нить
Разделяемая	Мультипроцессор	Нет	R/w	Все нити блока	Блок
Глобальная	DRAM	Нет	R/w	Все нити и CPU	Выделяется CPU
Константная	DRAM	Да	R/o	Все нити и CPU	Выделяется CPU
Текстурная	DRAM	Да	R/o	Все нити и CPU	Выделяется CPU

Типы памяти в CUDA

Модель памяти на GPU



Работа с глобальной памятью

Выделение и освобождение глобальной памяти

```
1. // выделение памяти на device
2. cudaError_t cudaMalloc ( void ** devPtr, size_t size );

3. // выделение памяти на device под двумерные массивы
4. cudaError_t cudaMallocPitch ( void ** devPtr, size_t * pitch,
5.                               size_t width, size_t height );

6. // освобождение памяти
7. cudaError_t cudaFree ( void * devPtr );
```

Работа с глобальной памятью

Копирование памяти

1. cudaError_t **cudaMemcpy** (void * dst, const void * src, size_t size, enum cudaMemcpyKind kind);
2. cudaError_t **cudaMemcpyAsync** (void * dst, const void * src, size_t size, enum cudaMemcpyKind kind, cudaStream_t stream);
3. cudaError_t **cudaMemcpy2D** (void * dst, size_t dpitch , const void * src, size_t width, size_t height, enum cudaMemcpyKind kind);
4. cudaError_t **cudaMemcpy2DAsync** (void * dst, size_t dpitch , const void * src, size_t width, size_t height, enum cudaMemcpyKind kind , cudaStream_t stream);

Работа с глобальной памятью

Копирование памяти

Направление копирования

1. cudaMemcpyHostToDevice
2. cudaMemcpyDeviceToHost
3. cudaMemcpyDeviceToDevice
4. cudaMemcpyHostToHost

CUDA events

Событие – объект типа cudaEvent_t

```
    // …
1.  cudaEvent_t start, stop;
2.  float GPUTime = 0.0f;

3.  cudaEventCreate ( &start );
4.  cudaEventCreate ( &stop );

5.  cudaEventRecord (start , 0);
```

Работа с глобальной памятью

CUDA events

```
6.  kernel <<< B, T >>> (…); // Запуск функции-ядра

7.  cudaEventRecord ( stop , 0);
8.  cudaEventSynchronize ( stop );

9.  cudaEventElapsedTime ( &GPUTime, start, stop);
10. printf(“GPU time: %.3f ms\n”, GPUTime);

11. cudaEventDestroy ( start );
12. cudaEventDestroy ( stop  );
```

Работа с глобальной памятью

Пример №1

Сложение векторов

$$C_i = A_i + B_i,$$

$$A_i = \sqrt{i},\ B_i = 2 \cdot i,$$

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

- $N = 512 \cdot 50000$
- 512 нитей в блоке, 50000 блоков

Работа с глобальной памятью

Функция-ядро

```
1.  __global__ void addKernel(const float *A, const float *B, float *C, const int size) {
2.      int i = threadIdx.x + blockIdx.x * blockDim.x;
3.      if (i < size) {
4.          C[i] = A[i] + B[i];
5.      }
6.  }
```

Инициализация массивов A и B

```
1.  void initialization(float *hostA, float *hostB, const int size) {
2.      for (int i = 0; i < size; i++) {
3.          hostA[i] = sqrtf(i);
4.          hostB[i] = 2.*i;
5.      }
6.  }
```

Работа с глобальной памятью

Описание и определение переменных

```
1.   void workFunction() {
2.       float *hostA, *hostB, *hostC;
3.       float *devA, *devB, *devC;
4.       int arraySize = 512*50000;

5.       cudaEvent_t GPUstart, GPUstop;
6.       float CPUstart, CPUstop;

7.       float GPUtime = 0.0f;
8.       float CPUtime = 0.0f;

9.       int N_threads = 512; // Количество нитей в блоке
10.      int N_blocks;
```

Работа с глобальной памятью

Инициализация массивов и определение размера блока

```
20.     initialization (hostA, hostB, arraySize);

21.     if ((arraySize % N_threads) == 0) {
22.             N_blocks = (arraySize / N_threads);
23.     }
24.     else {
25.             N_blocks = (arraySize / N_threads) + 1;
26.     }

27.     dim3 Threads(N_threads);
28.     dim3 Blocks (N_blocks);
```

Работа с глобальной памятью

Выделение памяти на CPU и GPU

```
29.     size_t mem_size = sizeof(float)* arraySize;

30.     hostA = (float*)malloc(mem_size);
31.     hostB = (float*)malloc(mem_size);
32.     hostC = (float*)malloc(mem_size);

33.     HANDLE_ERROR(cudaMalloc((void**)&devA, mem_size));
34.     HANDLE_ERROR(cudaMalloc((void**)&devB, mem_size));
35.     HANDLE_ERROR(cudaMalloc((void**)&devC, mem_size));
```

Работа с глобальной памятью

Вычисление на GPU

```
36.     cudaEventCreate(&GPUstart);
37.     cudaEventCreate(&GPUstop);

38.     cudaEventRecord(GPUstart, 0);

39.     cudaMemcpy(devA, hostA, mem_size, cudaMemcpyHostToDevice);
40.     cudaMemcpy(devB, hostB, mem_size, cudaMemcpyHostToDevice);
41.     cudaMemset(devC, 0, mem_size);

42.     addKernel <<< Blocks, Threads >>> (devA, devB, devC, arraySize);

43.     cudaMemcpy(hostC, devC, mem_size, cudaMemcpyDeviceToHost);

44.     cudaEventRecord(GPUstop, 0);
45.     cudaEventSynchronize(GPUstop);

46.     cudaEventElapsedTime(&GPUtime, GPUstart, GPUstop);
47.     printf("GPU time : %.3f ms\n", GPUtime);
```

Работа с глобальной памятью

Вычисление на CPU

```
36.     CPUstart = clock();

37.     for (int i = 0; i < arraySize; i++) {
38.             hostC[i] = hostA[i] + hostB[i];
39.     }

40.     CPUstop = clock();
41.     CPUtime = 1000.*(CPUstop - CPUstart) / CLOCKS_PER_SEC;
42.     printf("CPU time : %.3f ms\n", CPUtime);
```

Работа с глобальной памятью

Освобождение памяти

```
43.     free(hostA);
44.     free(hostB);
45.     free(hostC);
46.     HANDLE_ERROR(cudaFree(devA));
47.     HANDLE_ERROR(cudaFree(devB));
48.     HANDLE_ERROR(cudaFree(devC));

49.     HANDLE_ERROR(cudaEventDestroy(GPUstart));
50.     HANDLE_ERROR(cudaEventDestroy(GPUstop));
51. }
```

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (1 CPU) GPU GeForce 650M

GPU time: 107.797 ms

CPU time: 24.000 ms

Rate: 0.223

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 107.797 ms

CPU time: 18.000 ms

Rate: 0.167

- Копирование данных с “**host**” -> “**device**”
- Выполнение “**функции-ядра**”
- Копирование данных с “**device**” -> “**host**”

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 11.699 ms

CPU time: 18.000 ms

Rate: 1.54

- Выполнение “**функции-ядра**”

Работа с глобальной памятью

Пример №2

Вычисление сложной функции

$$C_i = \sum_{j=1}^{100} (\cos(\sqrt{A_i} \cdot \tan(B_i)) \cdot \sqrt{j}),$$

$$A_i = \sqrt{i},\ B_i = 2 \cdot i,$$

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

- $N = 512 \cdot 50000$
- 512 нитей в блоке, 50000 блоков

Работа с глобальной памятью

Функция-ядро

```
1.  __global__ void funcKernel(const float *A, const float *B, float *C, const int size) {
2.      int i = threadIdx.x + blockIdx.x * blockDim.x;
3.      if (i < size) {
4.          for (int j = 0; j < 100; j++) {
5.                      C[i] += cosf(sqrtf(A[i]) * tanhf(B[i])) * sqrtf(j);
6.          }
7.      }
8.  }
```

Работа с глобальной памятью

Вычисление на GPU

```
36.     cudaEventCreate(&GPUstart);
37.     cudaEventCreate(&GPUstop);

38.     cudaEventRecord(GPUstart, 0);

39.     cudaMemcpy(devA, hostA, mem_size, cudaMemcpyHostToDevice);
40.     cudaMemcpy(devB, hostB, mem_size, cudaMemcpyHostToDevice);
41.     cudaMemset(devC, 0, mem_size);

42.     funcKernel <<< Blocks, Threads >>> (devA, devB, devC, arraySize);

43.     cudaMemcpy(hostC, devC, mem_size, cudaMemcpyDeviceToHost);

44.     cudaEventRecord(GPUstop, 0);
45.     cudaEventSynchronize(GPUstop);

46.     cudaEventElapsedTime(&GPUtime, GPUstart, GPUstop);
47.     printf("GPU time : %.3f ms\n", GPUtime);
```

Работа с глобальной памятью

Вычисление на CPU

```
36.     CPUstart = clock();

37.     for (int i = 0; i < arraySize; i++) {
38.             for (int j = 0; j < 100; j++) {
39.                     hostC[i] += cosf(sqrtf(hostA[i]) * tanf(hostB[i])) * sqrtf(j);
40.             }
41.     }

42.     CPUstop = clock();
43.     CPUtime = 1000.*(CPUstop - CPUstart) / CLOCKS_PER_SEC;
44.     printf("CPU time : %.3f ms\n", CPUtime);

45.     printf("Rate : %.3f \n", CPUtime/GPUtime);
```

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (1 CPU) GPU GeForce 650M

GPU time: 218.957 ms

CPU time: 60199.000 ms

Rate: 274.935

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**”
3) Копирование данных с “**device**” -> “**host**”

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (1 CPU) GPU GeForce 650M

GPU time: 121.262 ms

CPU time: 60199.000 ms

Rate: 496.438

1) Выполнение “**функции-ядра**”

Работа с глобальной памятью

Вычисление на CPU

```
36.     CPUstart = clock();

37.     #pragma omp parallel
38.     {
39.         #pragma omp for
40.         for (int i = 0; i < arraySize; i++)
41.             for (int j = 0; j < 100; j++) {
42.                 hostC[i] += cosf(sqrtf(hostA[i]) * tanhf(hostB[i])) * sqrtf(j);
43.             }
44.     }

45.     CPUstop = clock();
46.     CPUtime = 1000.*(CPUstop - CPUstart) / CLOCKS_PER_SEC;
47.     printf("CPU time : %.3f ms\n", CPUtime);
```

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 218.957 ms

CPU time: 15091.000 ms

Rate: 68.922

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**”
3) Копирование данных с “**device**” -> “**host**”

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 121.262 ms

CPU time: 15091.000 ms

Rate: 124.45

1) Выполнение “**функции-ядра**”

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 64.567 ms

CPU time: 15091.000 ms

Rate: 233.726

1) Выполнение “**функции-ядра**” + Fast Math

Работа с глобальной памятью

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 156.404 ms

CPU time: 15091.000 ms

Rate: 96.487

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**” + Fast Math
3) Копирование данных с “**device**” -> “**host**”

CUDA-потоки

Функция-ядро

KernelName <<< nBlock, nThread, nShMem, **nStream** >>> (param);

nStream - номер потока из которого запускается функция-ядро

CUDA events

1. **cudaEvent_t** start;
2. **cudaEventCreate** (&start);
3. **cudaEventRecort** (start , 0);

CUDA-потоки

Использование закреплённой памяти (pinned-memory)

cudaHostAlloc() / cudaFreeHost()

Использование асинхронного копирования

cudaMemcpyAsync()

Синхронизация потока с CPU

cudaStreamSynchronize()

CUDA-потоки

Поддержка перекрытия операций (device overlap)

```
1.      cudaDeviceProp devProperties;
2.      int currentDevice;

3.      HANDLE_ERROR( cudaGetDevice( &currentDevice ) );
4.      HANDLE_ERROR( cudaGetDivecеProperties( &devProperties, currentDevice ) );

5.      if( devProperties.deviceOverlap ) {
6.          printf(“Using CUDA streams…\n”);
7.      }
```

CUDA-потоки

Пример №3

Вычисление сложной функции с использованием CUDA-streams

$$C_i = \sum_{j=1}^{100} (\cos(\sqrt{A_i} \cdot \tan(B_i)) \cdot \sqrt{j}),$$

$$A_i = \sqrt{i},\ B_i = 2 \cdot i,$$

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

- $N = 512 \cdot 50000$
- 512 нитей в блоке, 50000 блоков

CUDA-потоки

Функция-ядро

```
1.  __global__ void funcKernel(const float *A, const float *B, float *C, const int size) {
2.      int i = threadIdx.x + blockIdx.x * blockDim.x;
3.      if (i < size) {
4.          for (int j = 0; j < 100; j++) {
5.              C[i] += cosf(sqrtf(A[i]) * tanhf(B[i])) * sqrtf(j);
6.          }
7.      }
8.  }
```

CUDA-потоки

Определение и описание переменных

```
1.  #define N_STREAMS 2


2.  //...
3.  void workFunction() {
4.      float *hostA, *hostB, *hostC;
5.      float *devA,  *devB,  *devC;
6.      int arraySize = ( 512 * 50000  ) / N_STREAMS;

7.      cudaEvent_t GPUstart, GPUstop;
8.      float CPUstart, CPUstop;

9.      float GPUtime = 0.0f;
10.     float CPUtime = 0.0f;

11.     int N_threads = 512;
12.     int N_blocks;
```

CUDA-потоки

Определение и описание CUDA-потоков

```
13.     cudaStream_t stream[N_STREAMS];

14.     for (int i = 0; i < N_STREAMS; i++) {
15.             cudaStreamCreate( &stream[i] );
16.     }
```

CUDA-потоки

Выделение памяти на CPU и GPU

```
hostA = (float*)malloc(mem_size);
hostB = (float*)malloc(mem_size);
hostC = (float*)malloc(mem_size);
```

```
17.     size_t mem_size = sizeof(float)*arraySize;

18.     HANDLE_ERROR(cudaMallocHost((void**)&hostA, mem_size*N_STREAMS));
19.     HANDLE_ERROR(cudaMallocHost((void**)&hostB, mem_size*N_STREAMS));
20.     HANDLE_ERROR(cudaMallocHost((void**)&hostC, mem_size*N_STREAMS));

21.     HANDLE_ERROR(cudaMalloc((void**)&devA, mem_size*N_STREAMS));
22.     HANDLE_ERROR(cudaMalloc((void**)&devB, mem_size*N_STREAMS));
23.     HANDLE_ERROR(cudaMalloc((void**)&devC, mem_size*N_STREAMS));
```

CUDA-потоки

Инициализация массивов и определение размера блока

```
24.     initialization (hostA, hostB, arraySize);

25.     if ((arraySize % N_threads) == 0) {
26.             N_blocks = (arraySize / N_threads);
27.     }
28.     else {
29.             N_blocks = (arraySize / N_threads) + 1;
30.     }

31.     dim3 Threads(N_threads);
32.     dim3 Blocks (N_blocks);
```

CUDA-потоки

Копирование массивов с Host на Device

```
33.     cudaMemset(devC, 0, mem_size);

34.     for (int i = 0; i < N_STREAMS; ++i) {
35.             HANDLE_ERROR(cudaMemcpyAsync(devA + i*arraySize / N_STREAMS,
                                             hostA + i*arraySize / N_STREAMS, mem_size,
                                             cudaMemcpyHostToDevice, stream[i]) );
36.             HANDLE_ERROR(cudaMemcpyAsync(devB + i*arraySize / N_STREAMS,
                                             hostB + i*arraySize / N_STREAMS, mem_size,
                                             cudaMemcpyHostToDevice, stream[i]) );
37.     }
```

CUDA-потоки

Выполнение Функции-ядра

```
38.     cudaEventCreate(&GPUstart);
39.     cudaEventCreate(&GPUstop);

40.     cudaEventRecord(GPUstart, 0);

41.     for (int i = 0; i < N_STREAMS; i++) {
42.         func1Kernel <<< Blocks, Threads, 0, stream[i] >>> (devA + i*arraySize / N_STREAMS,
                                                               devB + i*arraySize / N_STREAMS,
                                                               devC + i*arraySize / N_STREAMS,
                                                               arraySize);
43.     }
44.     HANDLE_ERROR(cudaGetLastError());

45.     cudaEventRecord(GPUstop, 0);
46.     cudaEventSynchronize(GPUstop);

47.     cudaEventElapsedTime(&GPUtime, GPUstart, GPUstop);
48.     printf("GPU time : %.3f ms\n", GPUtime);
```

CUDA-потоки

Освобождение памяти

```
49.     HANDLE_ERROR(cudaFreeHost(hostA));
50.     HANDLE_ERROR(cudaFreeHost(hostB));
51.     HANDLE_ERROR(cudaFreeHost(hostC));

52.     HANDLE_ERROR(cudaFree(devA));
53.     HANDLE_ERROR(cudaFree(devB));
54.     HANDLE_ERROR(cudaFree(devC));

55.     HANDLE_ERROR(cudaEventDestroy(GPUstart));
56.     HANDLE_ERROR(cudaEventDestroy(GPUstop));
57. }
```

CUDA-потоки

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M (1 CUDA-Stream)

GPU time: 156.404 ms

CPU time: 15091.000 ms

Rate: 96.487

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**” + Fast Math
3) Копирование данных с “**device**” -> “**host**”

CUDA-потоки

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M (2 CUDA-Stream)

GPU time: 139.541 ms (156)

CPU time: 15091.000 ms

Rate: 108.568

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**” + Fast Math
3) Копирование данных с “**device**” -> “**host**”

CUDA-потоки

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M (4 CUDA-Stream)

GPU time: 96.353 ms (139) (156)

(64 - выполнение “**функции-ядра**” + Fast Math)

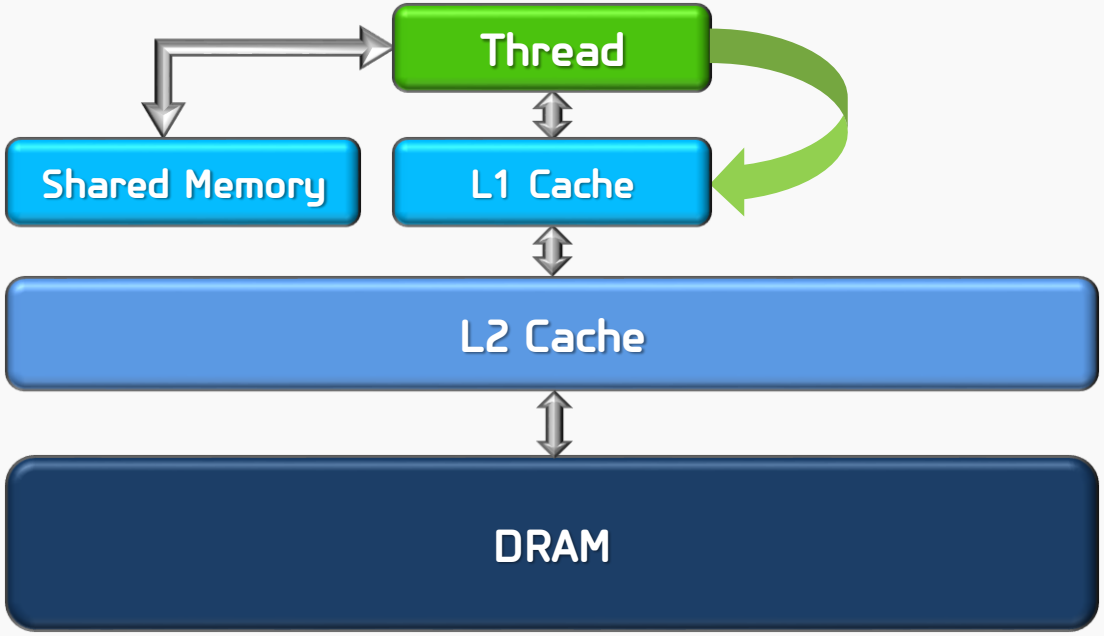
CPU time: 15091.000 ms

Rate: 156.622

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**” + Fast Math
3) Копирование данных с “**device**” -> “**host**”

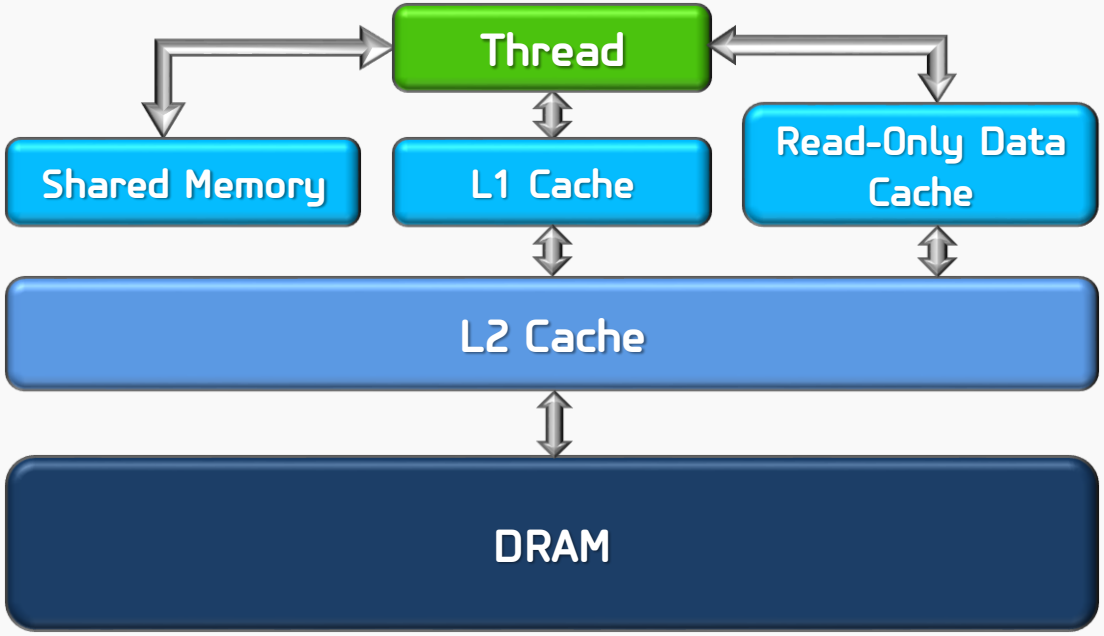
Работа с глобальной памятью

Подсистема памяти для CC 2.X



Работа с глобальной памятью

Подсистема памяти для CC 3.X



Работа с глобальной памятью

Read-Only Data Cache

Использование классификаторов const и __restrict__

```
1.  __global__ void KernelFunction (int* __restrict__ output,
                                    const int* __ restrict__ input  ) {
2.      // тело фукнкции
3.       output[idx1] = input[idx2];
4.  }
```

Использование __ldg ()

```
1.  __global__ void KernelFunction (int* output, int* input) {
2.      // тело фукнкции
3.       output[idx1] = __ldg( &input[idx2] );
4.  }
```

Работа с глобальной памятью

Read-Only Data Cache

Конфигурация Shared-memory и L1-Cache

- 48 КБ Shared-memory / 16 КБ L1-Cache
  cudaFuncCachePreferShared
- 16 КБ Shared-memory / 48 КБ L1-Cache
  cudaFuncCachePreferL1
- Без предпочтения (в зависимости от контекста)
  cudaFuncCachePreferNone

Работа с глобальной памятью

Конфигурация с предпочтением L1-cache

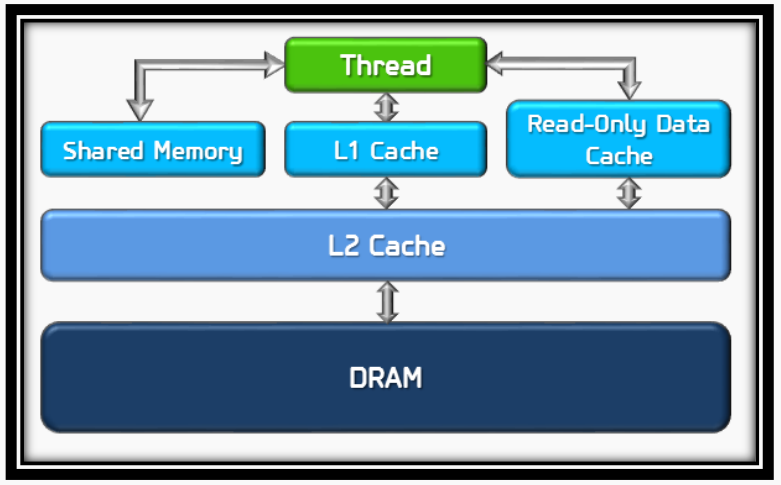
```
    // device-code
1.      __global__ void KernelFunction(…) {
2.          // Тело функции
3.      }

    // host-code
4.      int main(…) {
5.          // …

6.          cudaFuncSetCacheConfig ( KernelFunction, cudaFuncCachePreferL1 );

7.          // …
8.          return 0;
9.      }
```

Работа с глобальной памятью

L1-cache и L2-cache

Конфигурация L1 и L2

- Использование L2 -Xptxas –dlcm=ca
- Использование L1 и L2 -Xptxas –dlcm=cg



Работа с глобальной памятью

Особенности работы с глобальной памятью

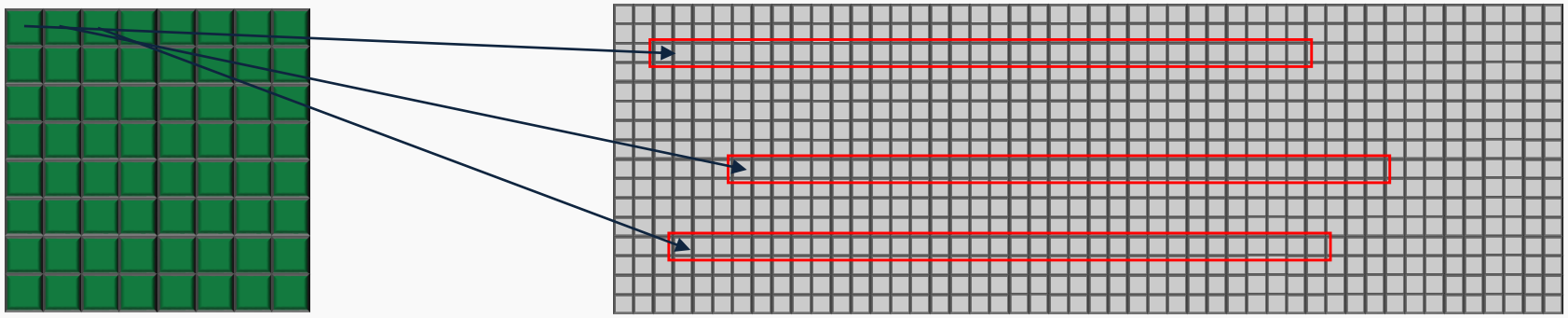
- GPU объединяет ряд запросов к глобальной памяти в 1 блок - транзакцию
- Объединение запросов на уровне варпов
- Кеш-линия 128 Б
- Выравнивание по 128 Б в глобальной памяти

Работа с глобальной памятью

Обращение в глобальную память

L1-cache выключен – запросы по 32 Б

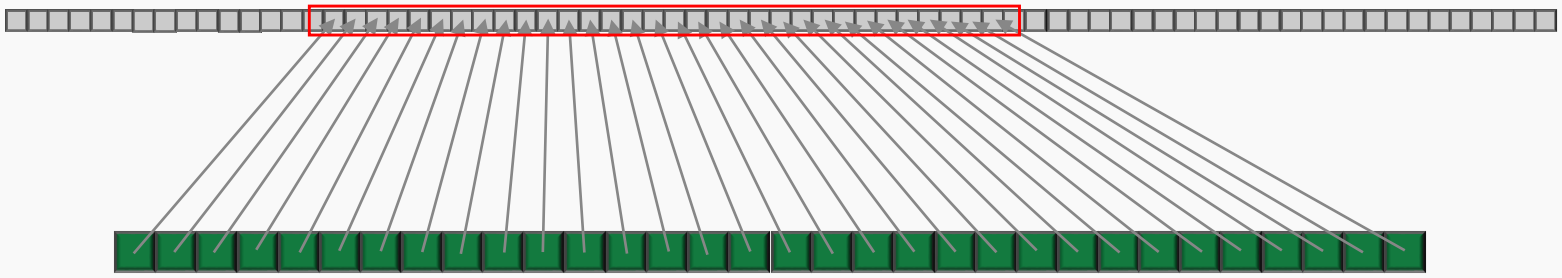
Использование для разреженного доступа в память



Работа с глобальной памятью

Обращение в глобальную память

L1-cache включен – запросы по 128 Б с кешированием в L1-cache



Работа с глобальной памятью

Пример №4

Перемножение матриц

$$C = AB,$$

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} B_{k,j},$$

$$A_i = \sqrt{i},\ B_i = \sin(i),$$

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

- $N\text{x}N = 1024\text{x}1024$
- $32\text{x}32$ нитей в блоке, $(N\text{x}N)/(32\text{x}32)$ блоков

Работа с глобальной памятью

Функция-ядро

```
1.      __global__ void mult(float *A, float *B, float *C) {
2.          unsigned int idx_X = threadIdx.x + blockIdx.x * blockDim.x;
3.          unsigned int idx_Y = threadIdx.y + blockIdx.y * blockDim.y;
4.          float sum = 0.;

5.          if ((idx_X < N) && (idx_Y < N)) {
6.              for (int i = 0; i < N; i++) {
7.                  sum += A[idx_X*N + i] * B[idx_Y + i*N];
8.              }
9.              C[idx_X*N + idx_Y] = sum;
10.         }
11.     }
```

Работа с глобальной памятью

Функция main

```
12. int main(void) {
13.     cudaEvent_t GPUstart, GPUstop;
14.     float GPUtime = 0.0f;

15.     float *hostA, *hostB;
16.     float *hostC;

17.     float *devA, *devB;
18.     float *devC;

19.     size_t mem_size = N*N*sizeof(float);
```

Работа с глобальной памятью

Функция main

```
20.     hostA = (float *)malloc(mem_size);
21.     hostB = (float *)malloc(mem_size);
22.     hostC = (float *)malloc(mem_size);

23.     cudaMalloc((void**)&devA, mem_size);
24.     cudaMalloc((void**)&devB, mem_size);
25.     cudaMalloc((void**)&devC, mem_size);
```

Работа с глобальной памятью

Инициализация массивов и копирование на GPU

```
26.     for (int i = 0; i < N*N; i++) {
27.             hostA[i] = sqrtf(i);
28.             hostB[i] = sinf(i);
29.     }

30.     cudaMemcpy(devA, hostA, mem_size, cudaMemcpyHostToDevice);
31.     cudaMemcpy(devB, hostB, mem_size, cudaMemcpyHostToDevice);
32.     cudaMemset(devC, 0, mem_size);
```

Работа с глобальной памятью

Инициализация массивов и копирование на GPU

```
33.     int N_Threads = 32;
34.     int N_Blocks = 0;

35.     if (((N) % N_Threads) == 0) {
36.             N_Blocks = ((N) / N_Threads);
37.     }
38.     else {
39.             N_Blocks = ((N) / N_Threads) + 1;
40.     }

41.     dim3 Threads(N_Threads,N_Threads);
42.     dim3 Blocks(N_Blocks,N_Blocks);
```

Работа с глобальной памятью

Запуск функции-ядра

```
43.     cudaEventCreate(&GPUstart);
44.     cudaEventCreate(&GPUstop);

45.     cudaEventRecord(GPUstart, 0);

46.     mult <<< Blocks, Threads >>> (devA, devB, devC);

47.     cudaEventRecord(GPUstop, 0);
48.     cudaEventSynchronize(GPUstop);

49.     cudaEventElapsedTime(&GPUtime, GPUstart, GPUstop);
50.     printf("GPU time : %.3f ms\n", GPUtime);

51.    cudaMemcpy(hostC, devC, mem_size, cudaMemcpyDeviceToHost);
```

Работа с глобальной памятью

Инициализация массивов и копирование на GPU

```
52.      cudaFree(devA);
53.      cudaFree(devB);
54.      cudaFree(devC);
55.      free(hostA);
56.      free(hostB);
57.      free(hostC);

58.      return 0;
59. }
```

Работа с глобальной памятью

Анализ результатов

GPU GeForce 650M

GPU time: 801.613 ms

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**”
3) Копирование данных с “**device**” -> “**host**”

Работа с глобальной памятью

Анализ результатов

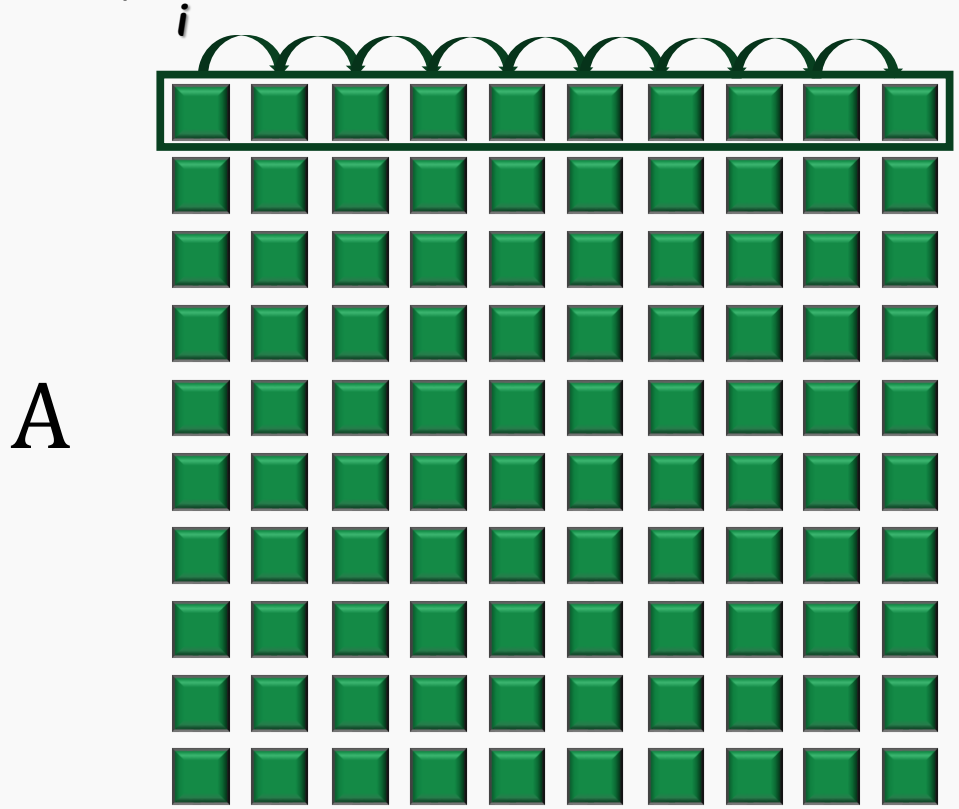
GPU GeForce 650M

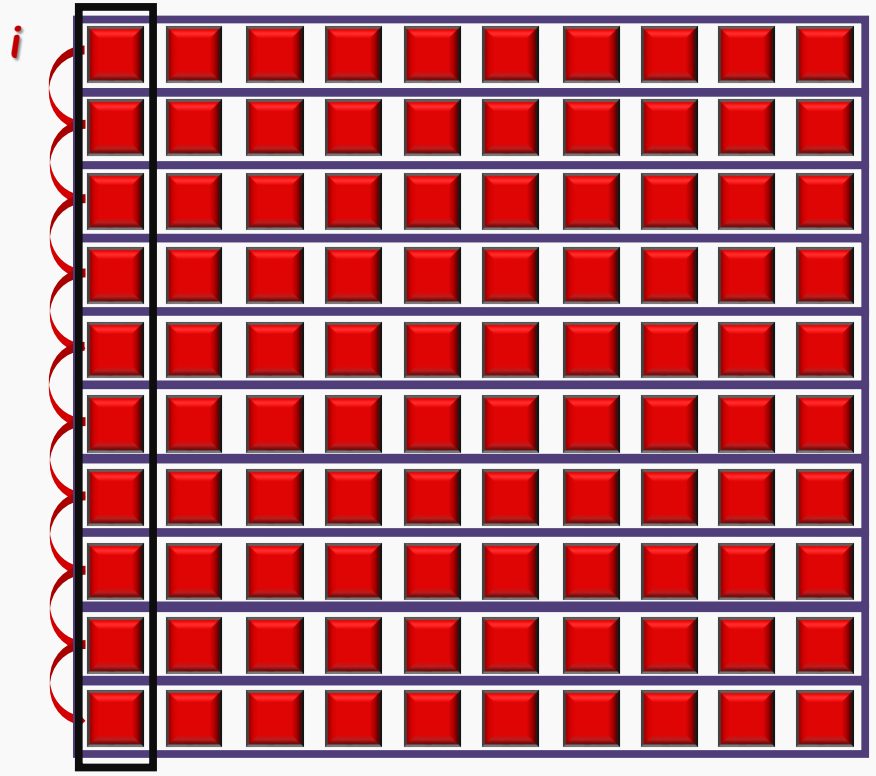
GPU time: 793.149 ms

1) Выполнение “**функции-ядра**”

Работа с глобальной памятью

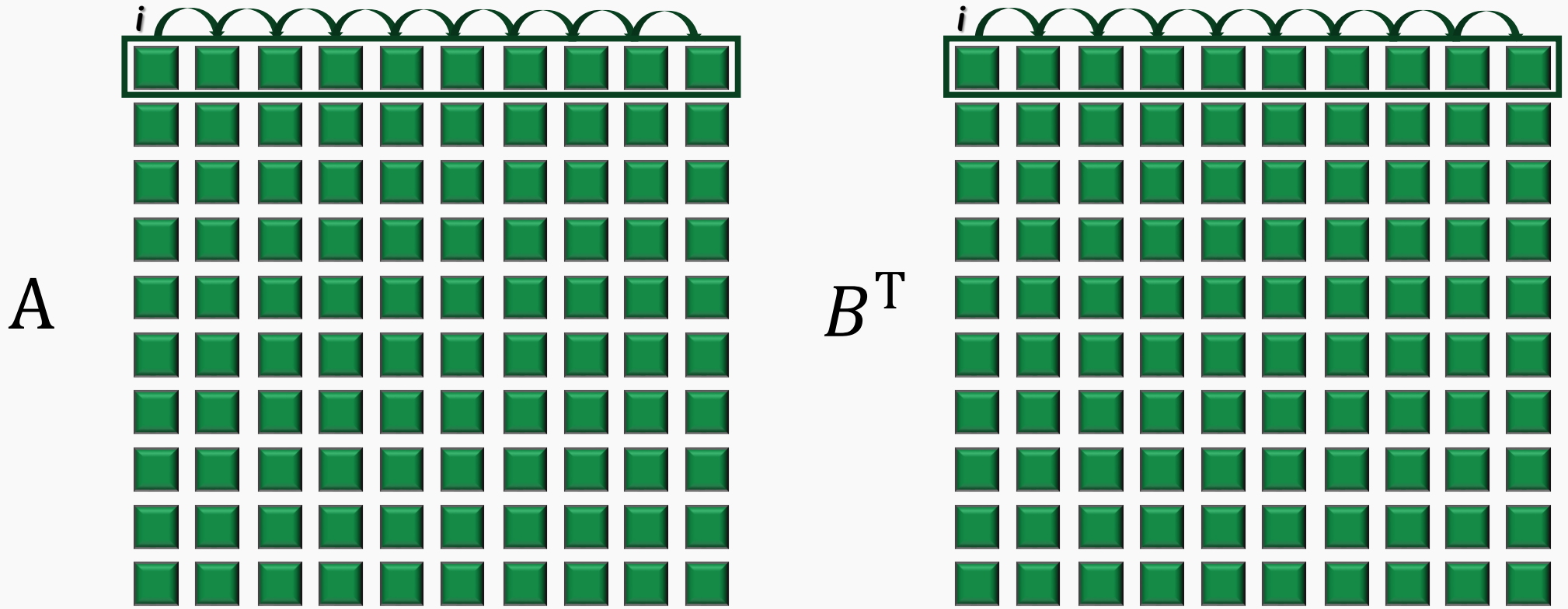
Анализ результатов





Работа с глобальной памятью

Анализ результатов



Работа с глобальной памятью

Анализ результатов

GPU GeForce 650M

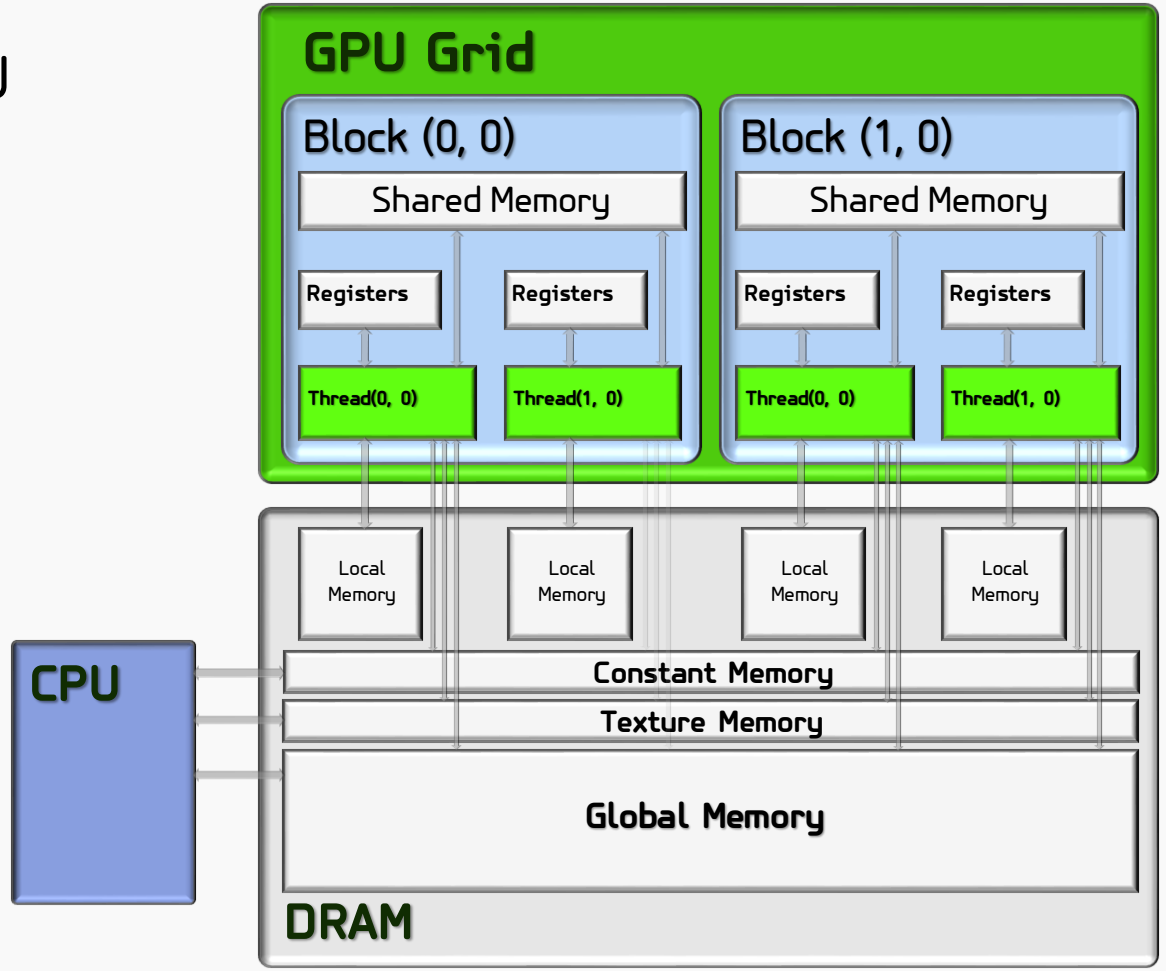
GPU time: 248.291 ms (793)

Rate: 3.198

1) Выполнение “**функции-ядра**” + coalescing

Типы памяти в CUDA

Модель памяти на GPU



Разделяемая память

Работа с shared-memory

Выделение памяти

- Статический способ

```
__shared__ float  P[32];  // Массив
__shared__ short L;       // Переменная
```

- Динамический способ

```
extern __shared__ float [ ];
```

Разделяемая память

Работа с shared-memory

Выделение памяти

```
1.  __global__ void Kernel(float *A) {
2.      __shared__ float buf[];
3.      // ...
4.      buf[threadIdx.x] =A[threadIdx.x + blockIdx.x * blockDim.x];
5.      // Тело функции
6.  }


1.  //
2.  Kernel <<< dim3(Blocks), dim3(Threads), N*sizeof(float) >>> ( devA );
3.  //
```

Разделяемая память

Работа с shared-memory

Выделение памяти

```
1.  __global__ void Kernel(float *A, int n) {
2.      __shared__ float buf1[];
3.      __shared__ float buf2[];
4.      // ...
5.      buf1[threadIdx.x] =A[threadIdx.x + blockIdx.x * blockDim.x];
6.      buf2[threadIdx.x + n] =A[threadIdx.x + blockIdx.x * blockDim.x + n];
7.      // Тело функции
8.  }
```

Разделяемая память

Пример №5

Перемножение матриц + shared memory

$$C = AB,$$

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} B_{k,j},$$

$$A_i = \sqrt{i},\ B_i = \sin(i),$$

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

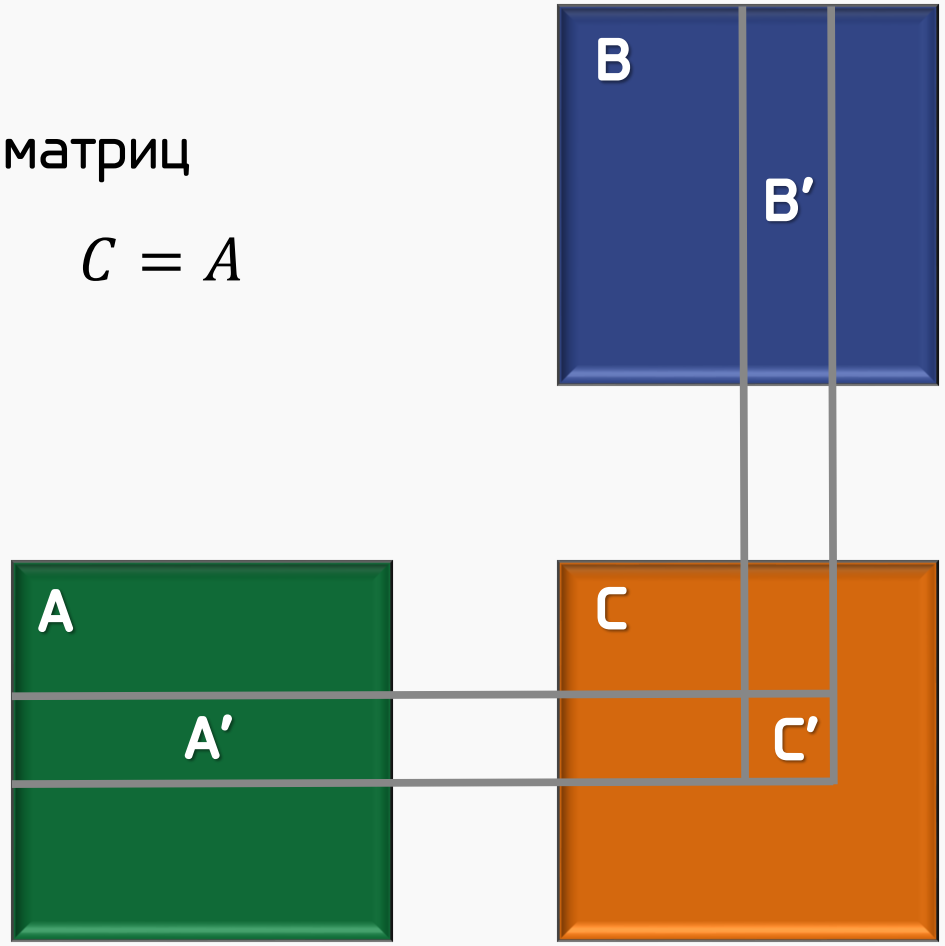
- $N\text{x}N = 1024\text{x}1024$
- $32\text{x}32$ нитей в блоке, $(N\text{x}N)/(32\text{x}32)$ блоков

Разделяемая память

Пример №5

Перемножение матриц

$$C = A$$



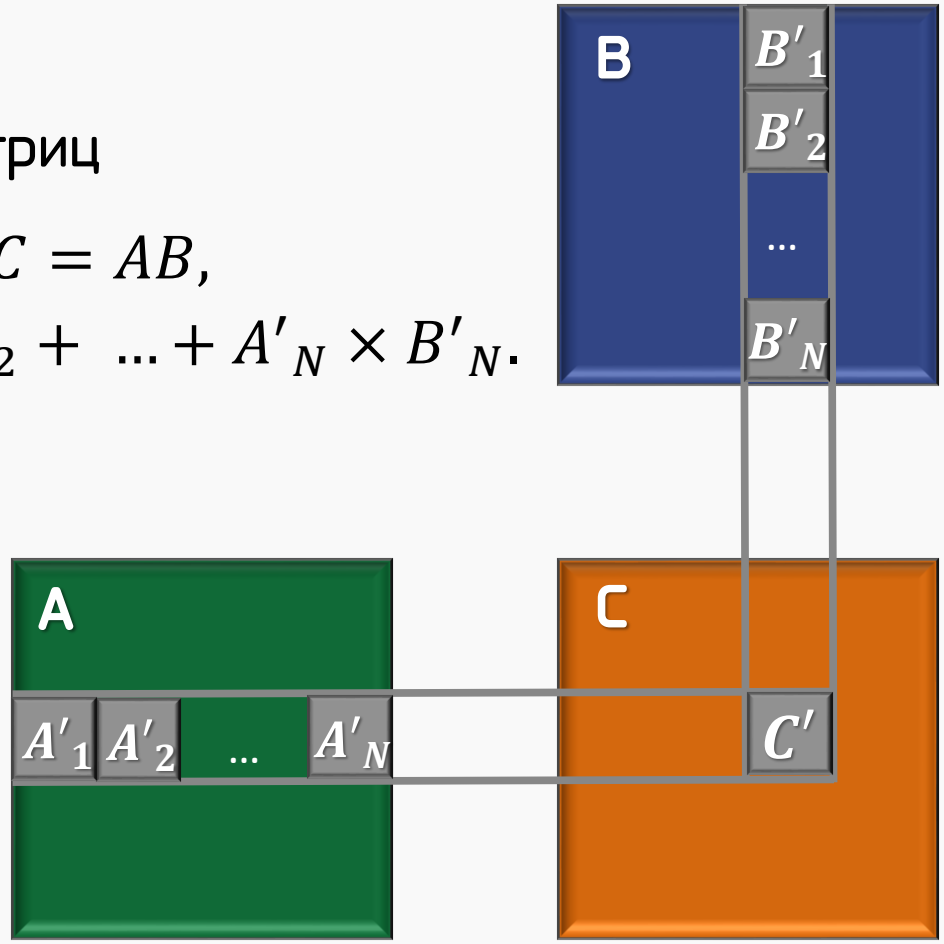
Разделяемая память

Пример №5

Перемножение матриц

$$C = AB,$$

$$C' = A'_1 \times B'_1 + A'_2 \times B'_2 + \ \ldots + A'_N \times B'_N.$$



Разделяемая память

Функция-ядро

```
1.      #define BLOCK_SIZE 16

2.      __global__ void kernel ( float * a, float * b, float * c, int n )
3.      {
4.          int bx = blockIdx.x, by = blockIdx.y;
5.          int tx = threadIdx.x, ty = threadIdx.y;

6.          int aBegin = n * BLOCK_SIZE * by;
7.          int aEnd = aBegin + n - 1;
8.          int bBegin = BLOCK_SIZE * bx;
9.          int aStep  = BLOCK_SIZE, bStep = BLOCK_SIZE * n;

10.         float sum = 0.0f;
```

Разделяемая память

Функция-ядро

```
11.     for (int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep)
12.     {
13.       __shared__ float as[BLOCK_SIZE][BLOCK_SIZE]; // Статический способ выделения памяти
14.       __shared__ float bs[BLOCK_SIZE][BLOCK_SIZE];
15.       as[ty][tx] = a[ia + n * ty + tx];
16.       bs[ty][tx] = b[ib + n * ty + tx];
17.       __syncthreads();          // Синхронизация нитей

18.       for (int k = 0; k < BLOCK_SIZE; k++) sum += as[ty][k] * bs[k][tx];
19.       __syncthreads();
20.     }

21.     c[n * BLOCK_SIZE * by + BLOCK_SIZE * bx + n * ty + tx] = sum;
22.   }
```

Разделяемая память

Функция main

```
23. int main() {
24.     int N = 1024;
25.     int m, n, k;
26.     float timerValueGPU, timerValueCPU;
27.     cudaEvent_t start, stop;
28.     cudaEventCreate(&start);
29.     cudaEventCreate(&stop);

30.     size_t mem_size = N*N* sizeof(float);
31.     float *devA, *devB, *devC, *a, *b, *c, *cc, *bT, *aT;

32.     a =  (float*)malloc(mem_size);
33.     b =  (float*)malloc(mem_size);
34.     bT = (float*)malloc(mem_size);
35.     aT = (float*)malloc(mem_size);
36.     c =  (float*)malloc(mem_size);
37.     cc = (float*)malloc(mem_size);
```

Разделяемая память

Функция main

```
38.     for (n = 0; n<N; n++)
39.     {
40.         for (m = 0; m<N; m++)
41.         {
42.             a[m + n*N] = 2.0f*m + n;
43.             b[m + n*N] = m - n;
44.             aT[m + n*N] = m + n*2.0f;
45.             bT[m + n*N] = n - m;
46.         }
47.     }

48.     cudaMalloc((void**)&devA, mem_size);
49.     cudaMalloc((void**)&devB, mem_size);
50.     cudaMalloc((void**)&devC, mem_size);
```

Разделяемая память

Функция main

```
51.     dim3 Threads(BLOCK_SIZE, BLOCK_SIZE);
52.     dim3 Blocks(N / threads.x, N / threads.y);

53.     cudaEventRecord(start, 0);

54.     cudaMemcpy(adev, a, mem_size, cudaMemcpyHostToDevice);
55.     cudaMemcpy(bdev, b, mem_size, cudaMemcpyHostToDevice);

56.     kernel <<< Blocks, Threads >>> ( devA, devB, devC, N );

57.     cudaMemcpy(c, cdev, mem_size, cudaMemcpyDeviceToHost);

58.     cudaEventRecord (stop, 0);
59.     cudaEventSynchronize (stop);
60.     cudaEventElapsedTime (&timerValueGPU, start, stop);

61.     printf("\n GPU calculation time %f msec\n", timerValueGPU);
```

Разделяемая память

Функция main

```
62.      cudaFree(devA);
63.      cudaFree(devB);
64.      cudaFree(devC);
65.
66.      free(a);
67.      free(b);
68.      free(bT);
69.      free(aT);
70.      free(c);
71.      free(cc);

72.      cudaEventDestroy(start);
73.      cudaEventDestroy(stop);

74.      return 0;
75. }
```

Разделяемая память

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 246.570 ms

CPU time: 4678.000 ms

Rate: 18.972

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**” – global memory + coalescing
3) Копирование данных с “**device**” -> “**host**”

Разделяемая память

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 176.428 ms

CPU time: 4649.000 ms

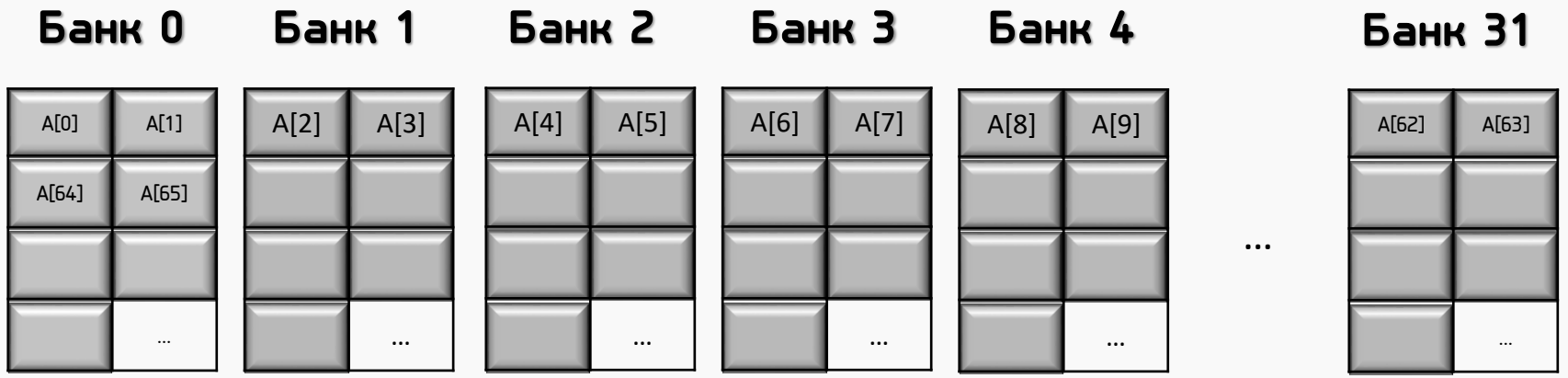
Rate: 26.351

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**” – shared memory
3) Копирование данных с “**device**” -> “**host**”

Разделяемая память

Разделение разделяемой памяти на банки

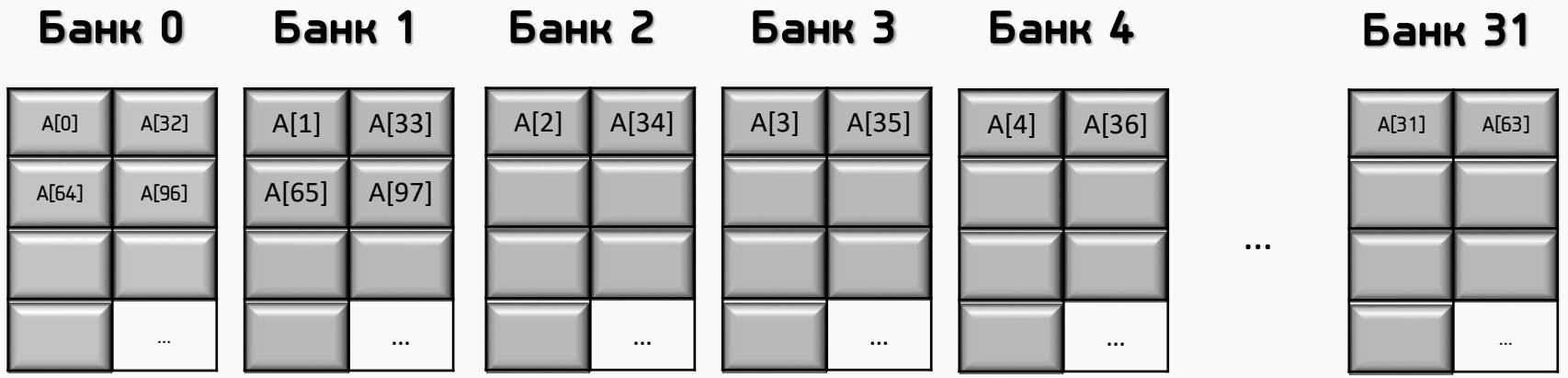
8-Байтовый режим доступа



Разделяемая память

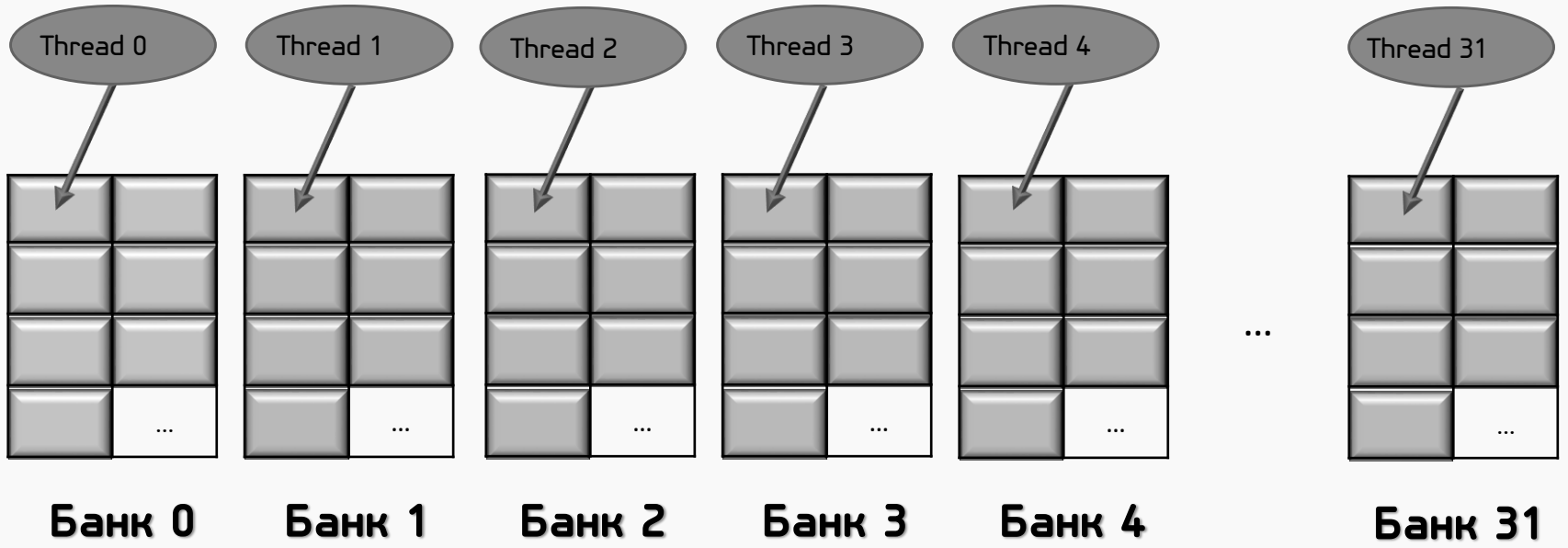
Разделение разделяемой памяти на банки

4-Байтовый режим доступа



Разделяемая память

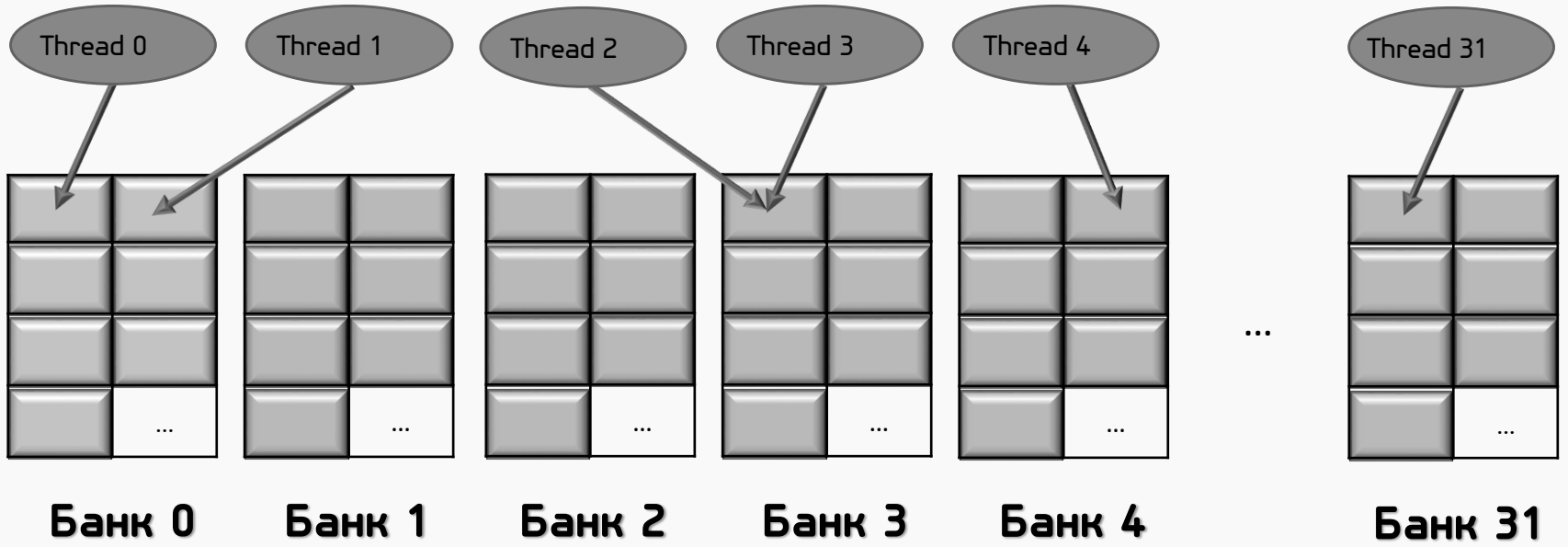
Разделение разделяемой памяти на банки



Нет банк конфликтов

Разделяемая память

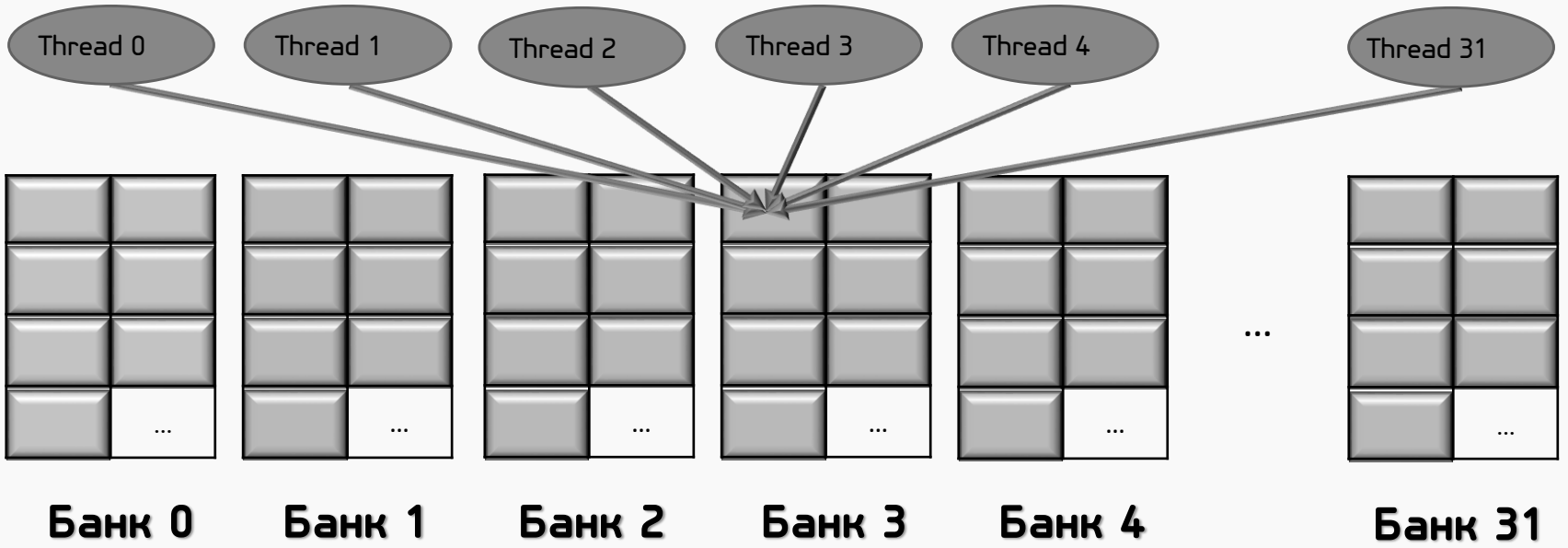
Разделение разделяемой памяти на банки



Нет банк конфликтов

Разделяемая память

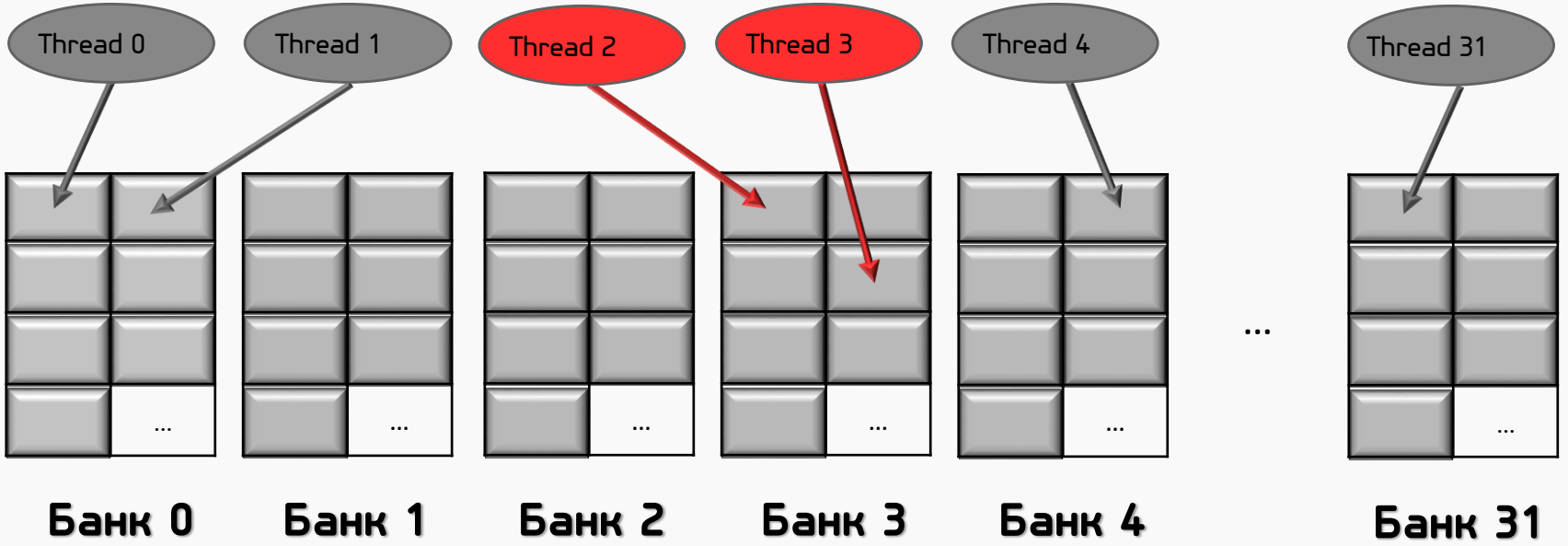
Разделение разделяемой памяти на банки



Нет банк конфликтов (broadcast)

Разделяемая память

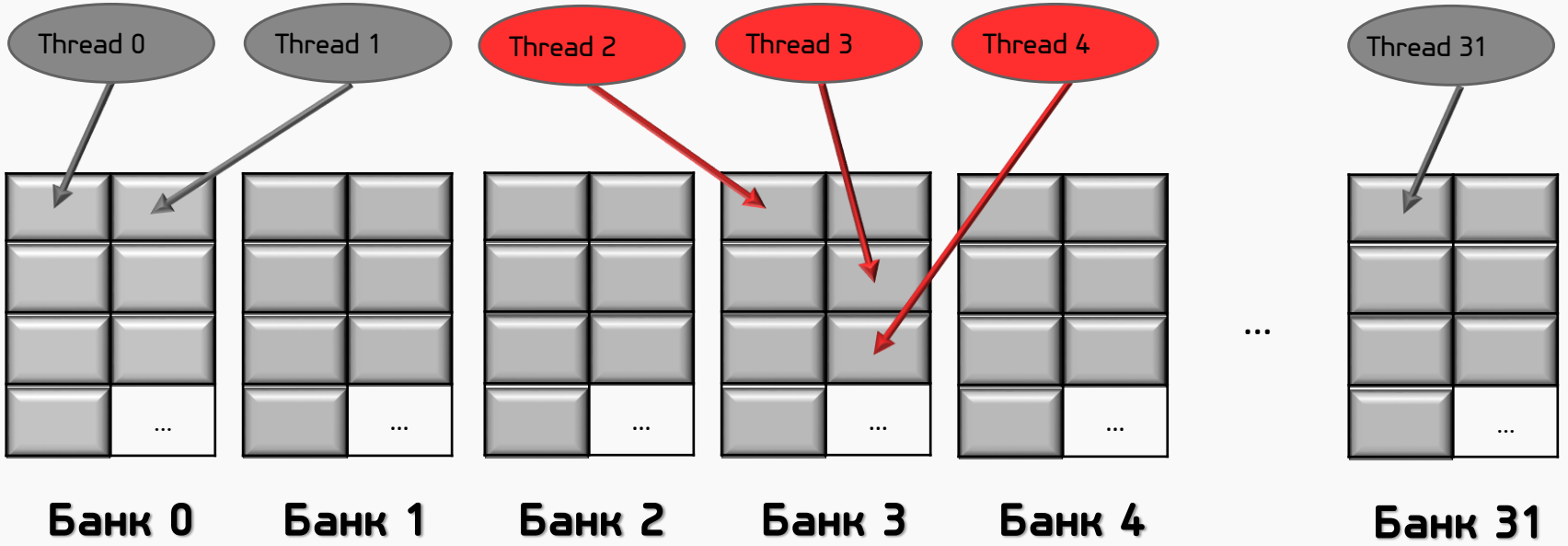
Разделение разделяемой памяти на банки



Банк конфликт 2-го порядка

Разделяемая память

Разделение разделяемой памяти на банки



Банк конфликт 3-го порядка

Разделяемая память

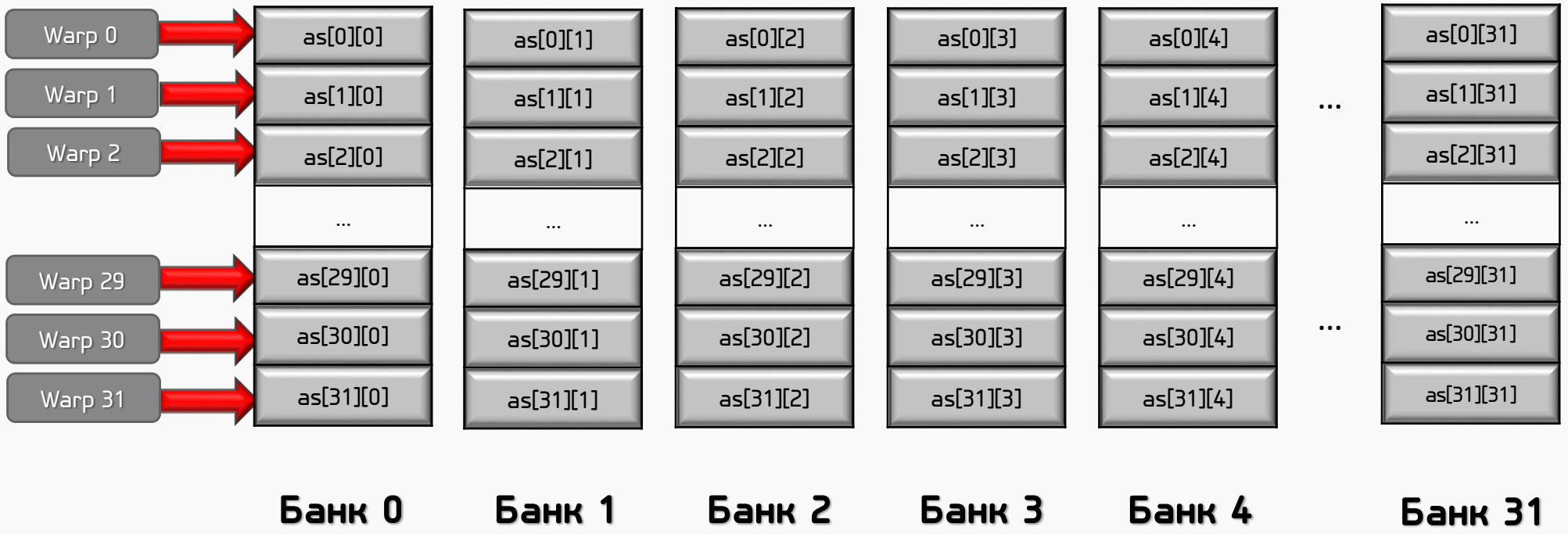
Функция-ядро

```
11.     for (int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep)
12.     {
13.       __shared__ float as[BLOCK_SIZE][BLOCK_SIZE]; // Статический способ выделения памяти
14.       __shared__ float bs[BLOCK_SIZE][BLOCK_SIZE];
15.       as[ty][tx] = a[ia + n * ty + tx];
16.       bs[ty][tx] = b[ib + n * ty + tx];
17.       __syncthreads();          // Синхронизация нитей

18.       for (int k = 0; k < BLOCK_SIZE; k++) sum += as[ty][k] * bs[k][tx];
19.       __syncthreads();
20.     }

21.     c[n * BLOCK_SIZE * by + BLOCK_SIZE * bx + n * ty + tx] = sum;
22.   }
```

Разделяемая память

Разделение разделяемой памяти на банки

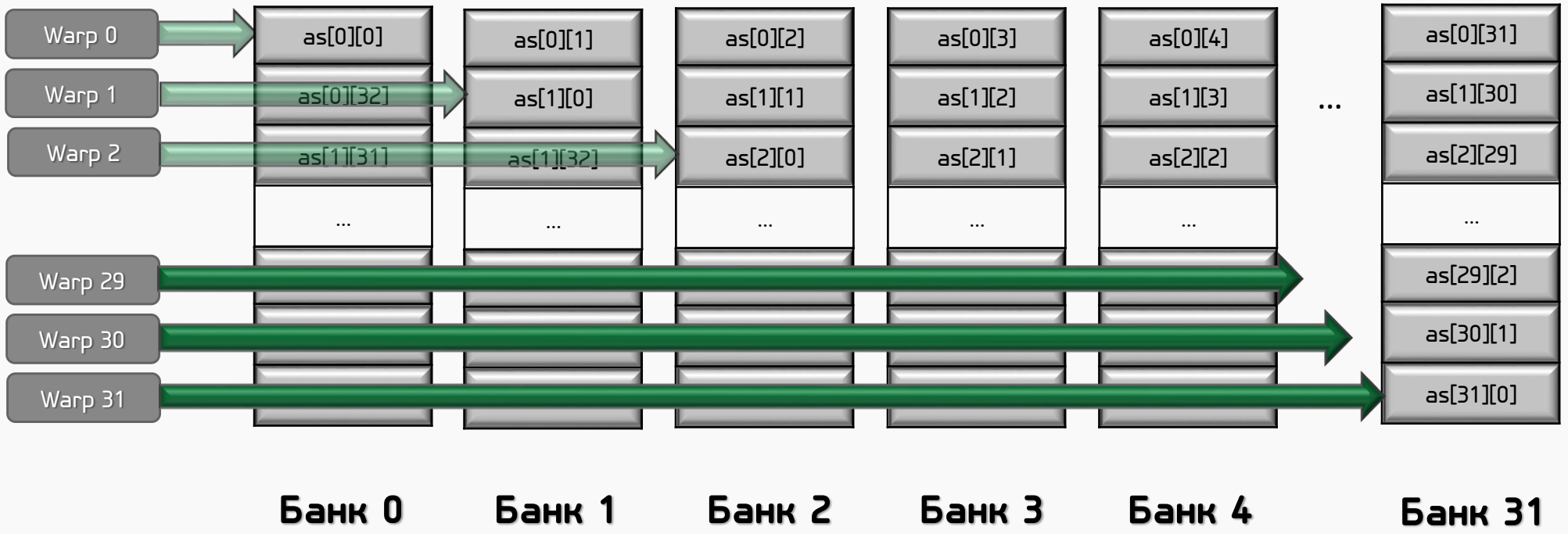
__shared __ double as[32][32];



Разделяемая память

Разделение разделяемой памяти на банки

__shared __ double as[32][33];



Разделяемая память

Функция-ядро

```
11.     for (int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep)
12.     {
13.      __shared__ float as[BLOCK_SIZE][BLOCK_SIZE +1 ]; //Статический способ выделения памяти
14.      __shared__ float bs[BLOCK_SIZE][BLOCK_SIZE +1 ];
15.      as[ty][tx] = a[ia + n * ty + tx];
16.      bs[ty][tx] = b[ib + n * ty + tx];
17.      __syncthreads();         // Синхронизация нитей

18.      for (int k = 0; k < BLOCK_SIZE; k++) sum += as[ty][k] * bs[k][tx];
19.      __syncthreads();
20.     }

21.     c[n * BLOCK_SIZE * by + BLOCK_SIZE * bx + n * ty + tx] = sum;
22.  }
```

Разделяемая память

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time: 69.209 ms

CPU time: 4649.000 ms

Rate: 67.377

1) Копирование данных с “**host**” -> “**device**”
2) Выполнение “**функции-ядра**” – shared memory + opt
3) Копирование данных с “**device**” -> “**host**”

Разделяемая память

Анализ результатов

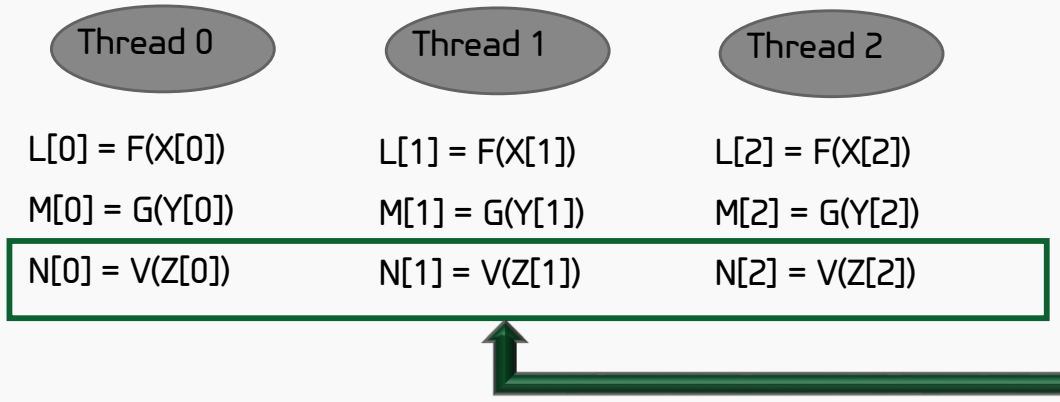
CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time:	793.149 ms	Rate: 6	Global memory
GPU time:	246.570 ms	Rate: 19	Global memory + coalescing
GPU time:	176.428 ms	Rate: 27	Shared memory
GPU time:	69.209 ms	Rate: 68	Shared memory + opt

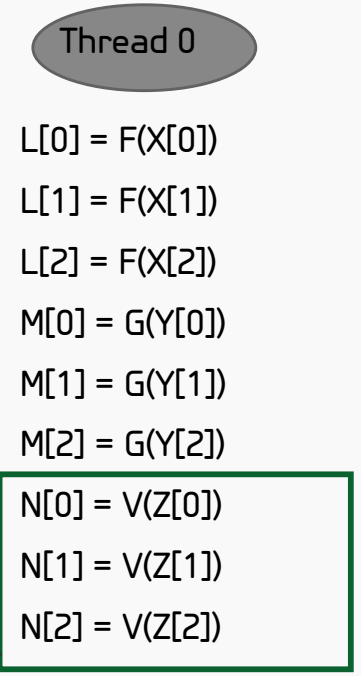
TLP и ILP

Volkov GTC 2010

Thread-Level Parallelism (TLP)



Instruction-Level Parallelism (ILP)



TLP и ILP

Функция-ядро

```
1.        as[ty][tx] = a[ia + n * ty + tx];
2.        bs[ty][tx] = b[ib + n * ty + tx];
3.        sum += as[ty][k] * bs[k][tx];
4.        c[n * BLOCK_SIZE * by + BLOCK_SIZE * bx + n * ty + tx]  sum;
```

```
1.        as[ty][tx] = a[ia + n * ty + tx];
2.        bs[ty][tx] = b[ib + n * ty + tx];
3.        as[ty][tx] = a[ia + n * (ty+16) + tx];
4.        bs[ty][tx] = b[ib + n * (ty+16) + tx];
5.        //…
6.        sum += as[ty][k] * bs[k][tx];
7.        sum += as[ty+16][k] * bs[k][tx];
8.        //…
9.        c[aBegin + bBegin + ty * n + tx] = sum1;
10.       c[aBegin + bBegin + (ty + 16) * n + tx] = sum2;
```

TLP и ILP

Запуск функции-ядра

```
// Определение числа нитей в блоке
1.      dim3 Threads (BLOCK_SIZE, BLOCK_SIZE / 2)

// Запуск Kernel
2.      kernel <<< Blocks, Threads >>> (devA, devB, N, devC);
```

TLP и ILP

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M (x2 inst)

GPU time: 46.957 ms

CPU time: 4649.000 ms

Rate: 99

TLP и ILP

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M (x4 inst)

GPU time: 34.253 ms

CPU time: 4649.000 ms

Rate: 135.725

Оптимизация

Анализ результатов

CPU Core i7-3610QM 2.30GHz (8 CPUs) GPU GeForce 650M

GPU time:	793.149 ms	Rate: 6	Global memory
GPU time:	246.570 ms	Rate: 19	Global memory + coalescing
GPU time:	176.428 ms	Rate: 27	Shared memory
GPU time:	69.209 ms	Rate: 68	Shared memory + opt
GPU time:	46.957 ms	Rate: 99	Shared memory + opt + 2x inst
GPU time:	34.253 ms	Rate: 136	Shared memory + opt + 4x inst

Контакты:
a.spasenov@mail.ru
alex_spasenov (Skype)

Спасибо за внимание!