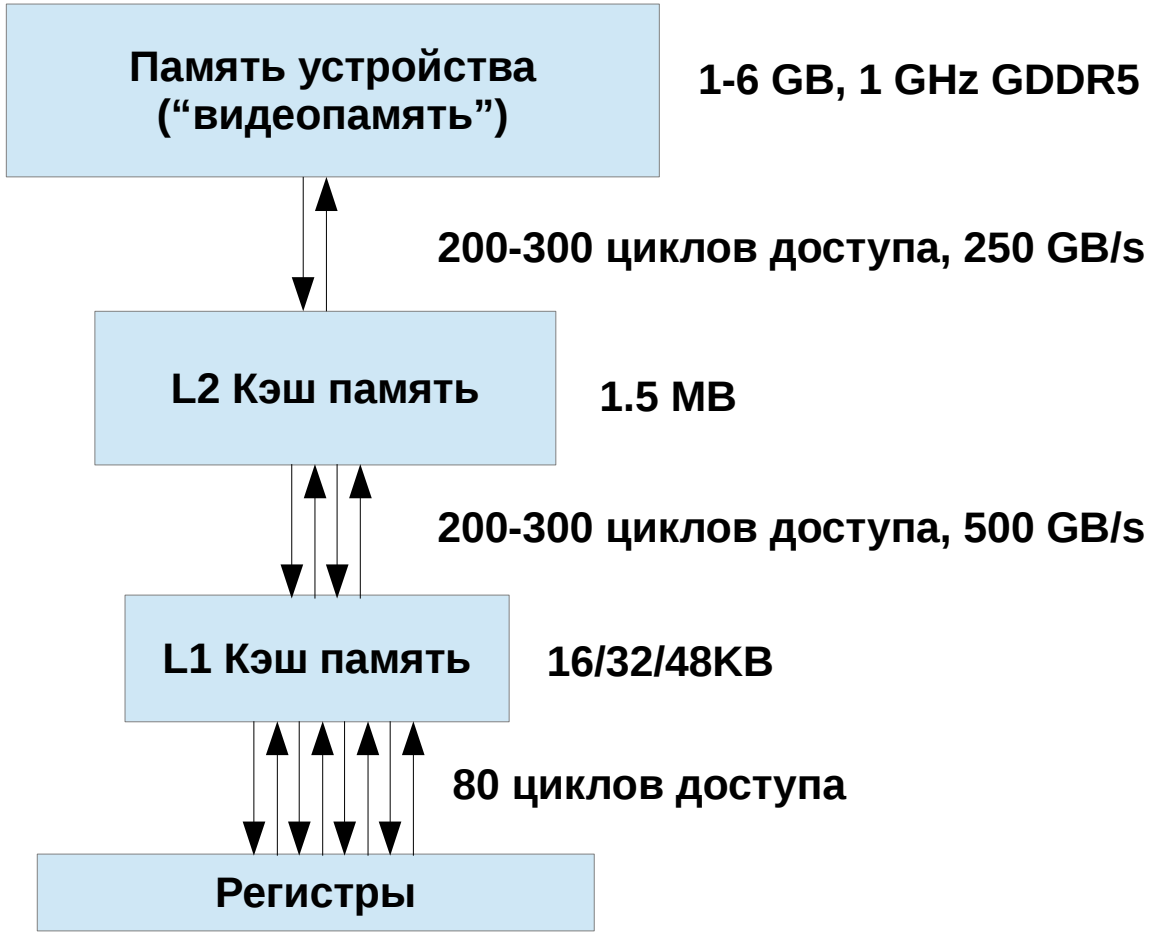
Лекция 3

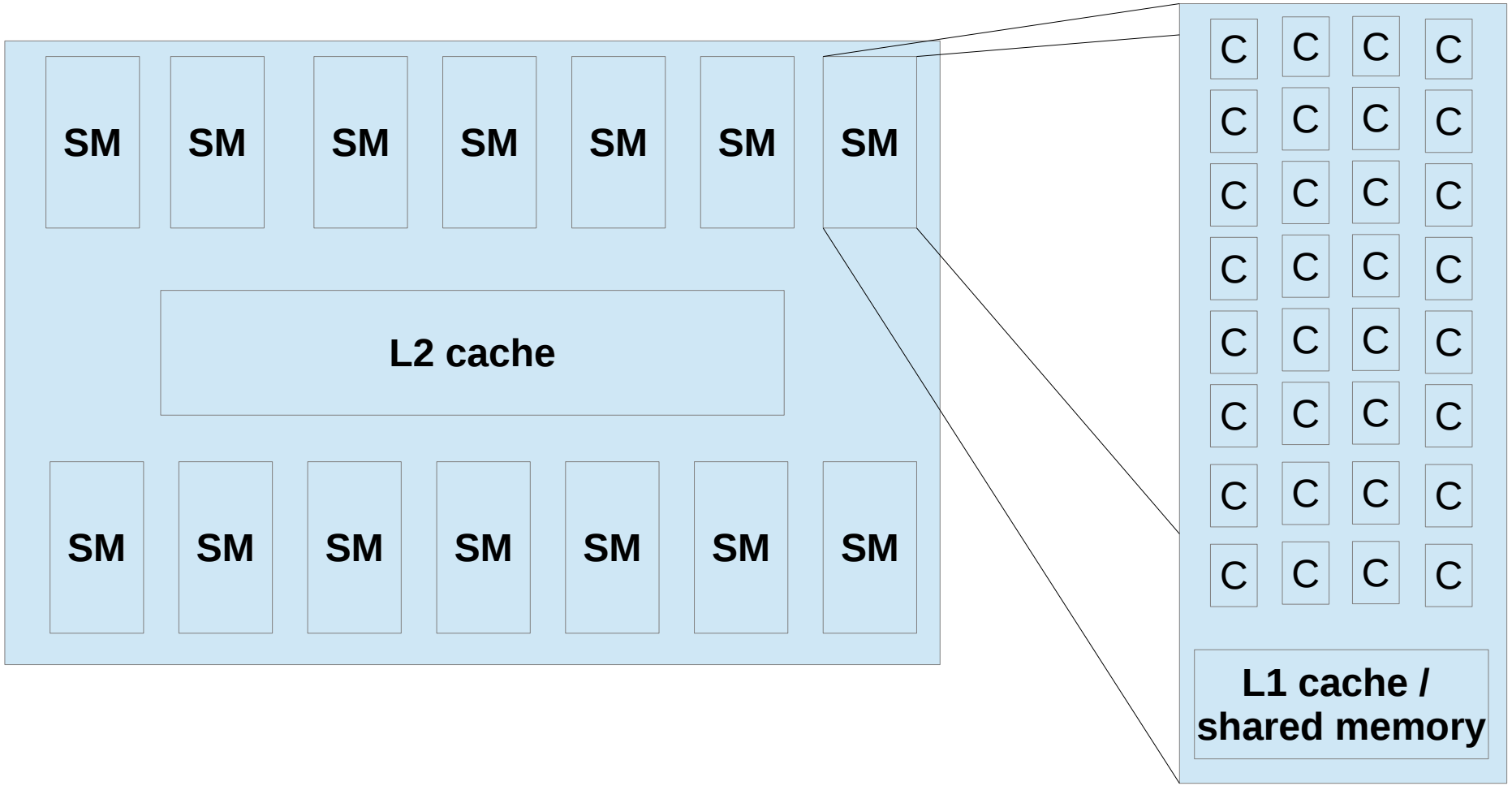
СОДЕРЖАНИЕ

- *иерархия памяти;*
- *объединение запросов к глобальной памяти (coalescing);*
- *разделяемая память (shared memory)*

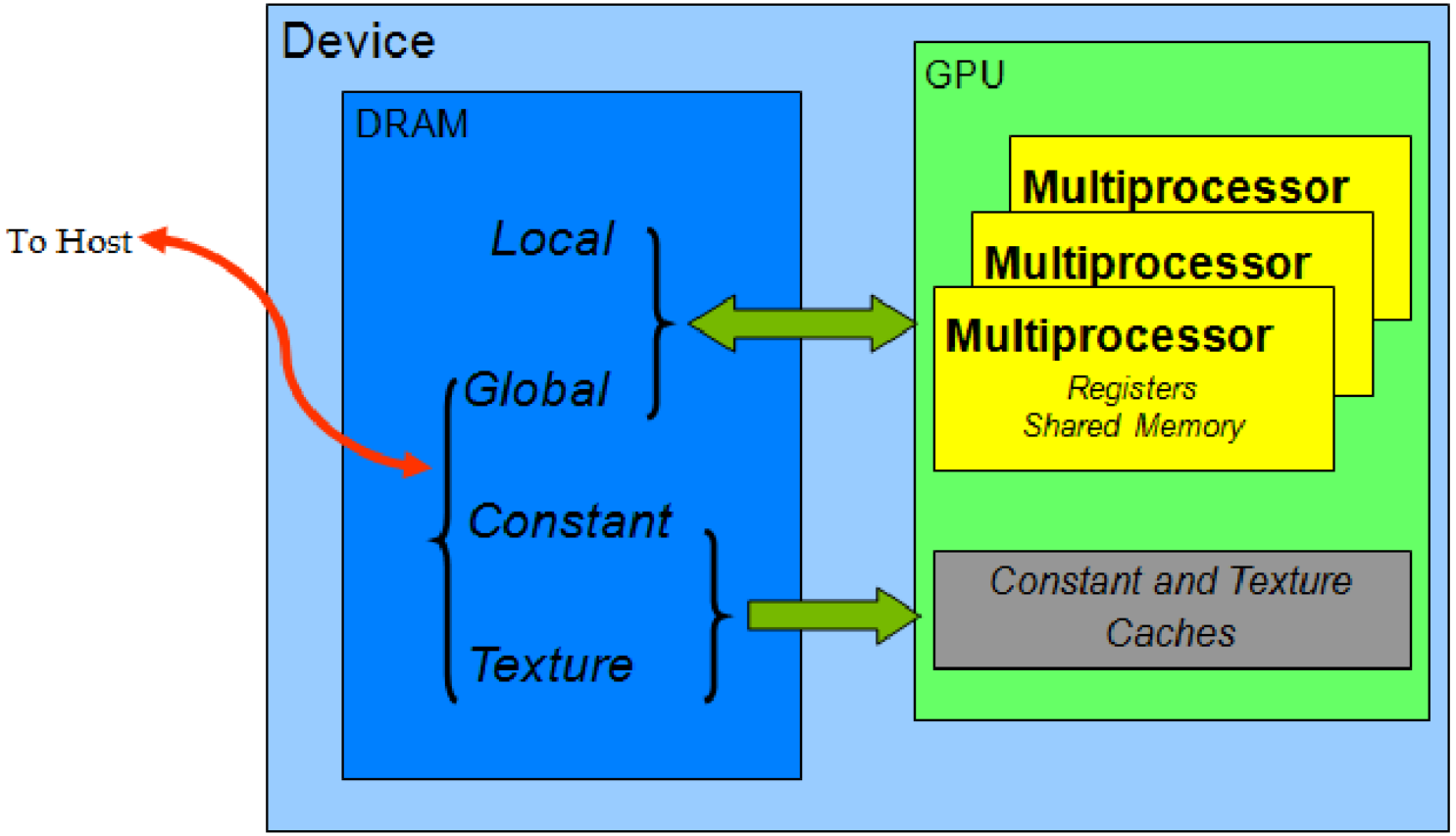
Иерархия памяти графического процессора (Fermi, Kepler)



Архитектура Ферми



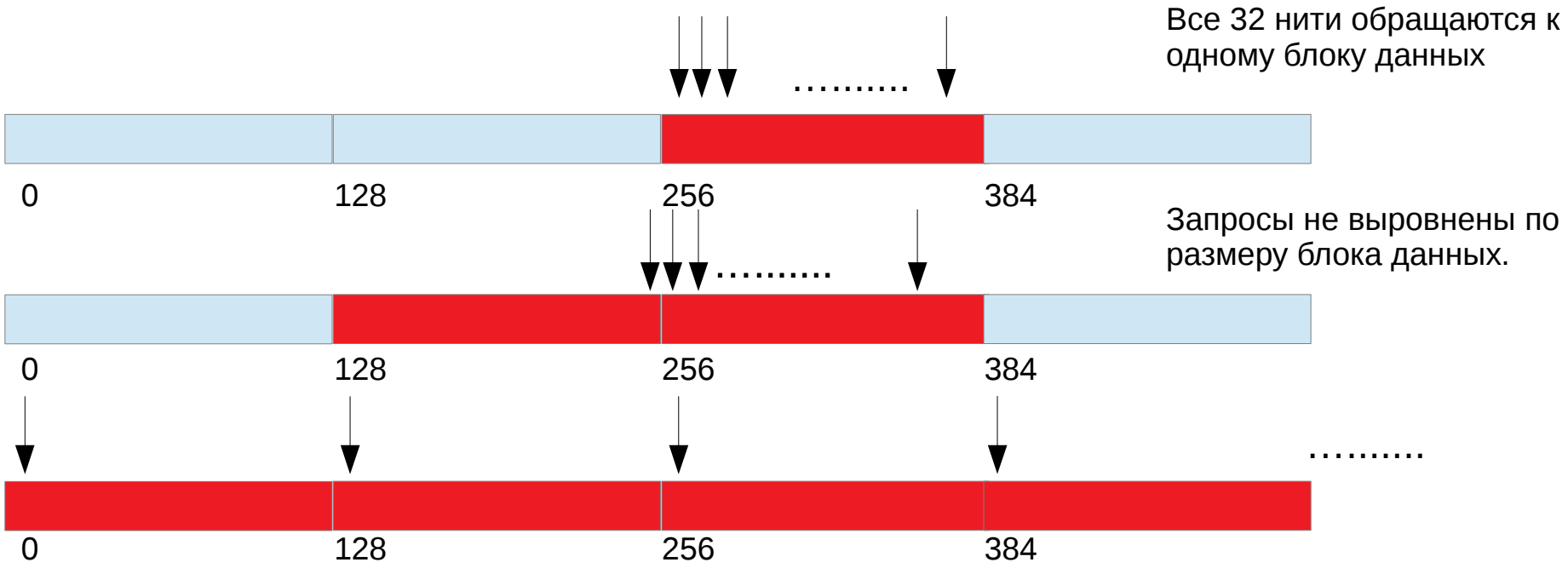
Иерархия памяти устройства CUDA



CUDA C Best Practices Guid, Nvidia, october 2012

Объединение запросов к глобальной памяти (coalescing)

Запросы на чтение и запись к глобальной памяти нитями одного варпа (*a warp*) объединяются в транзакции, количество которых равно количеству необходимых для выполнения запросов блоков данных (*cache lines*) L1 кэша размером в 128 байт. **COMPUTE CAPABILITIES 2.* !**



Тестирование параллельных алгоритмов с использованием coalescing'a на GPU с архитектурой Tesla, Fermi и Kepler (задание 1)

Исследовать влияние coalescing'a на примере сложения векторов и инициализации матриц.

```
cudaMalloc((void**)&dVx, (N+offset)*sizeof(float));
…..
(dVx+offset)[threadIdx.x]=.....
```

Нарушение выравнивания на границы кратные 128, выполняемого cudaMalloc.

```
A) df[threadIdx.x+blockIdx.x*blockDim.x]=.....
B) df[blockIdx.x+threadIdx.x*gridDim.x]=.....
```

Нарушение одновременного доступа нитей одного блока CUDA к одному блоку данных.

```
A) df[threadIdx.x+blockIdx.x*blockDim.x]=.....
B) df[(blockDim.x-threadIdx.x-1)+blockIdx.x*blockDim.x]=.....
```

Нарушение порядка доступа к блку данных полуварпа (*CC 1.**).

Исследовать зависимость времени копирования от величины страйда массива.

```
df1[(threadIdx.x+blockIdx.x*blockDim.x)*stride]=
            df2[(threadIdx.x+blockIdx.x*blockDim.x)*stride];
```

Исследовать влияние выравнивания на операции с масивами структур.

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

Приложение: шаблон для тестирования эффективности coalescing'a

```
#include <cuda.h>
#include <curand_kernel.h>
#include <stdio.h>
#include <malloc.h>

__global__  void gInit(float* a, float* b, int N){
    int thread_id=threadIdx.x+blockIdx.x*blockDim.x;

    unsigned int seed=thread_id;
    curandState s;
    curand_init(seed,0,0, &s);

    for(int i=thread_id; i<N; i+=blockDim.x*gridDim.x){
      a[i]=curand_uniform(&s);
      b[i]=curand_uniform(&s);
    }
}

__global__ void gSum(float* a, float* b, float* c, int N){
    int thread_id=threadIdx.x+blockIdx.x*blockDim.x;
    for(int i=thread_id; i<N; i+=blockDim.x*gridDim.x)
      c[i]=a[i]+b[i];
  }
```

шаблон (продолжение)

```
int main(int argc, char* argv[]){
  float *a, *b, *c;
  float *ha, *hb, *hc;
   if(argc<4) { fprintf(stderr, "USAGE: prog <blocks>  <threads>  <offset>\n"); return 1;}
  //int N=atoi(argv[1]);
  int num_of_blocks=atoi(argv[1]);
  int threads_per_block=atoi(argv[2]);
  int N=num_of_blocks*threads_per_block;
  int offset=atoi(argv[3]);

  cudaMalloc((void**)&a, (N+offset)*sizeof(float));
  cudaMalloc((void**)&b, N*sizeof(float));
  cudaMalloc((void**)&c, N*sizeof(float));

  ha=(float*)calloc(N, sizeof(float));
  hb=(float*)calloc(N, sizeof(float));
  hc=(float*)calloc(N, sizeof(float));

  gInit<<<num_of_blocks,threads_per_block>>>(a+offset,b,N);
  cudaThreadSynchronize();

cudaEvent_t  start,stop;
float elapsedTime;
```

шаблон (продолжение)

```
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start,0);

  gSum<<<num_of_blocks,threads_per_block>>>(a+offset,b,c,N);
//  cudaThreadSynchronize();

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime,start,stop);
fprintf(stderr,"gTest took %g\n", elapsedTime);

  cudaMemcpy(ha,a+offset, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaMemcpy(hb,b,N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaMemcpy(hc,c,N*sizeof(float), cudaMemcpyDeviceToHost);

  for(int i=0;i<N;i++)
    printf("%g\t%g\t%g\n", ha[i],hb[i], hc[i]);
```

шаблон (окончание)

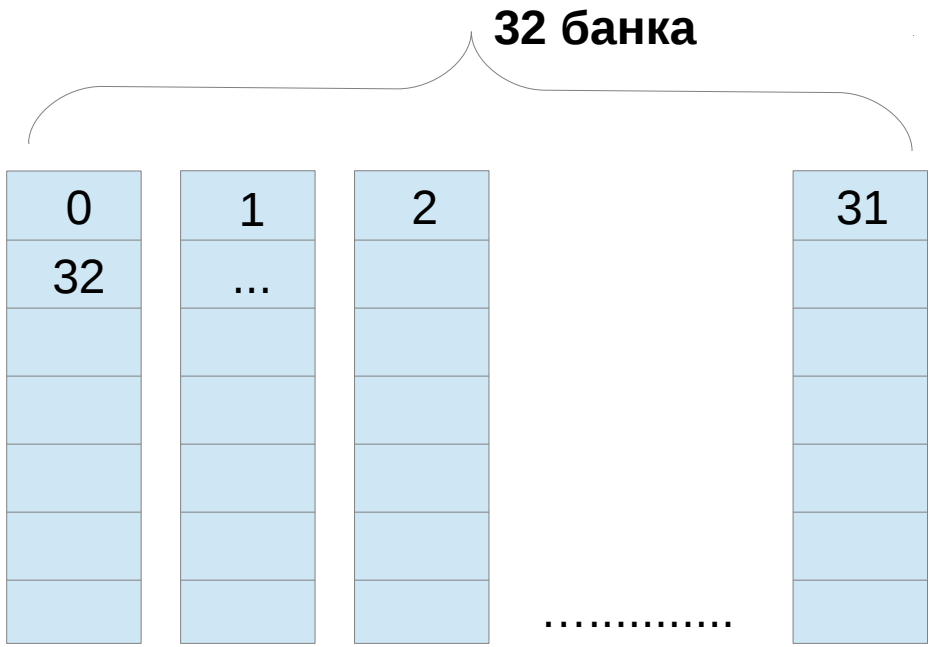
```
  cudaFree(a);
  cudaFree(b);
  cudaFree(c);
  free(ha);
  free(hb);
  free(hc);

  return 0;
}
```

Разделяемая память (shared memory)

Разделяемая память CUDA – память с низкой латентностью и высокой пропускной способностью.

Высокая пропускная способность обеспечивается параллельным выполнением запросов, благодаря разделению памяти на отдельные модули, банки памяти.



Если более одной нити варпа обращаются к одному и тому же банку, то происходит конфликт, который разрешается сериализацией выполнения запроса.

Например:

```
__shared__ float s[1024];
.....
s[threadIdx.x*2]=.....
```

Выделение разделяемой памяти

Разделяемая память выделяется (статически или динамически) только на устройстве. Область видимости – нити одного блока. Время жизни – время выполнения ядра.

Статическое выделение:

```
#define N 3
#define M 512
__global__ void gTest(){
   __shared__ float s[N][M];
…….....................................
}
```

Динамическое выделение:

```
extern __shared__ float s[];
__global__ void gTest2(){
        float* a=(float*)s;
        float* b=(float*)&s[512];
        float* c=(float*)&s[1024];
…….....................................
}
```

```
gTest2<<<100,32,N*M>>>();
```

3-й параметр – размер разделяемой памяти.

Задание (лабораторная 3)

Оптимизировать операцию транспонирования матриц используя *coalescing* и *shared memory*.