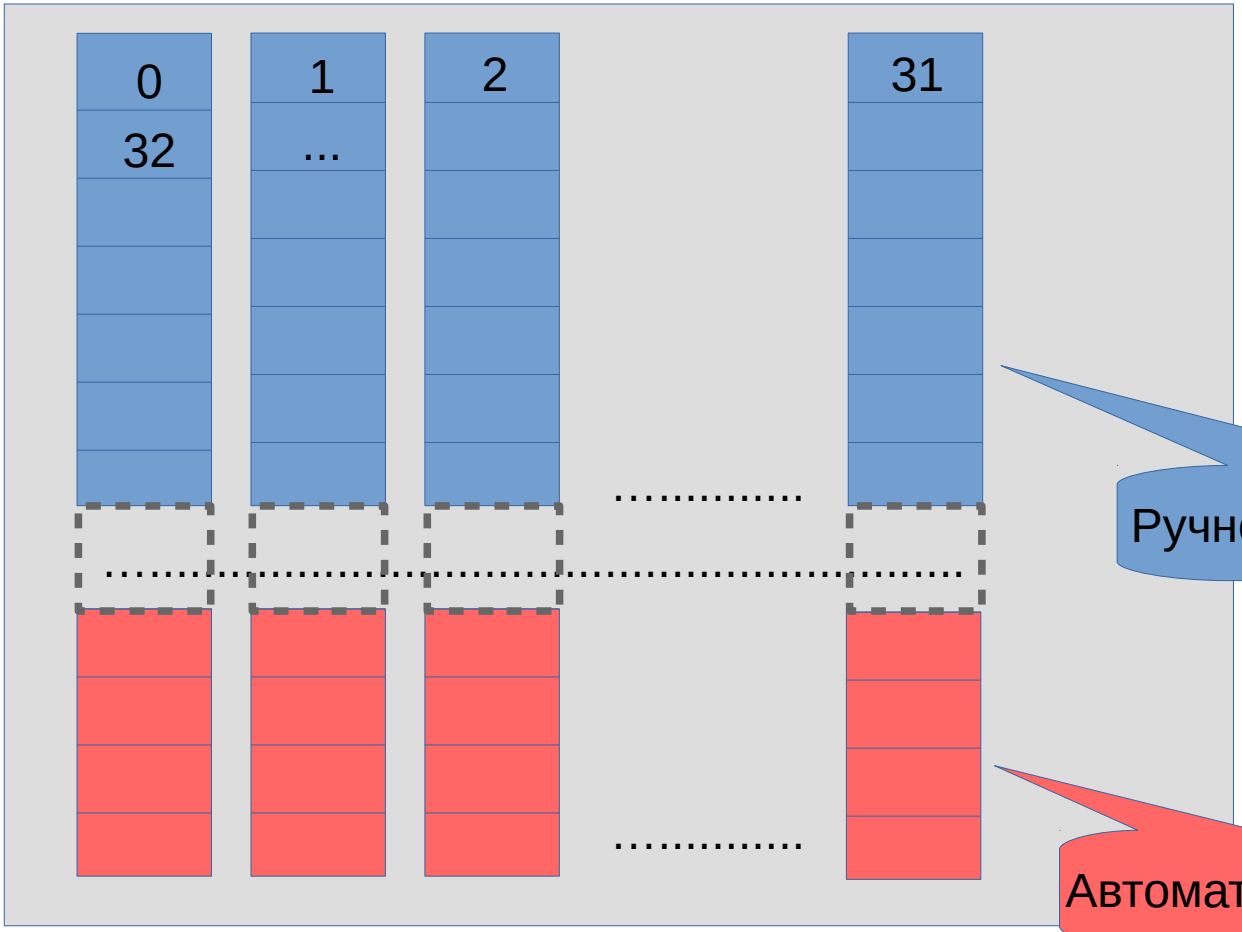
Лекция 4

СОДЕРЖАНИЕ

- ***разделяемая память (shared memory)***

L1 кэш память



32 банка памяти

1 строка =32*4 байта = 128 байт

Размер кэша L1:
512 строки * 128 байт =**64 K**

Регулируемое соотношение:
16K/48K или **48K/16K**

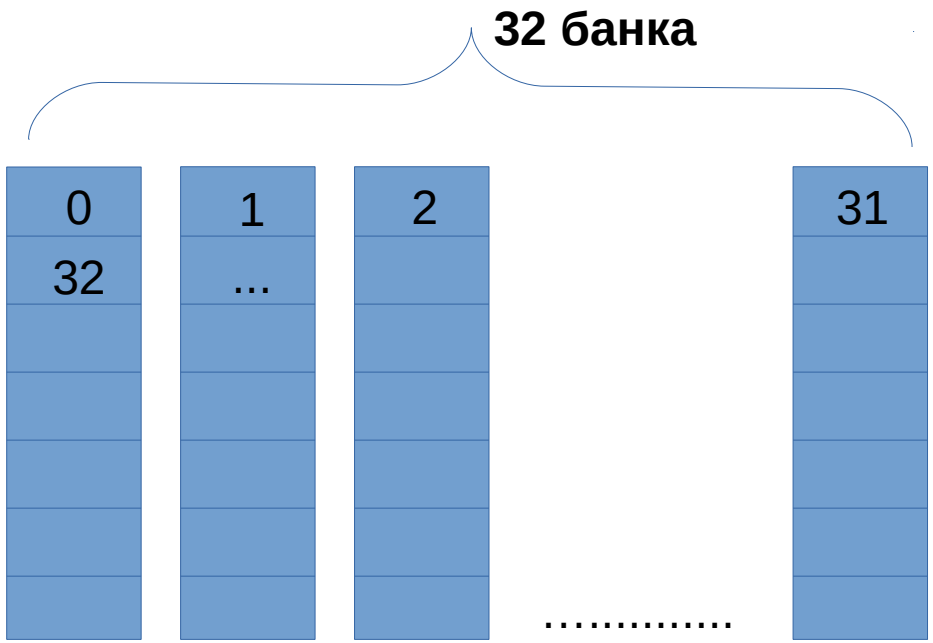
cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);

```
__global__ void gTest(...){
    ……..
    return;
}

int main(){
    cudaFuncSetCacheConfig(gTest, cudaFuncCachePreferL1);
    ……..
    gTest<<<num_th, num_bl>>>(...);

    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.

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*sizeof(float)>
>>();
```

3-й параметр – размер разделяемой памяти.

Вывод матрицы

```
#include <stdio.h>

void Output(float* a, int N){
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++)
      fprintf(stdout,"%g\t",a[j+i*N]);
    fprintf(stdout,"\n");
  }
     fprintf(stdout,"\n\n\n");
}
```

Инициализация матрицы

```
__global__ void gInitializeStorage(float* storage_d){
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;
    int N=blockDim.x*gridDim.x;

    storage_d[i+j*N]=(float)(i+j*N);
}
```

Простое транспонирование

```
__global__ void gTranspose0(float* storage_d, float* storage_d_t){
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;
    int N=blockDim.x*gridDim.x;

    storage_d_t[j+i*N]=storage_d[i+j*N];
}
```

Наивное использование shared memory (динамическое выделение памяти)

```
__global__ void gTranspose11(float* storage_d,float* storage_d_t){
    extern __shared__ float buffer[];
    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;
    int N=blockDim.x*gridDim.x;

    buffer[threadIdx.y+threadIdx.x*blockDim.y]=storage_d[i+j*N];
  __syncthreads();



    i=threadIdx.x+blockIdx.y*blockDim.x;
    j=threadIdx.y+blockIdx.x*blockDim.y;

    storage_d_t[i+j*N]=buffer[threadIdx.x+threadIdx.y*blockDim.x];
}
```

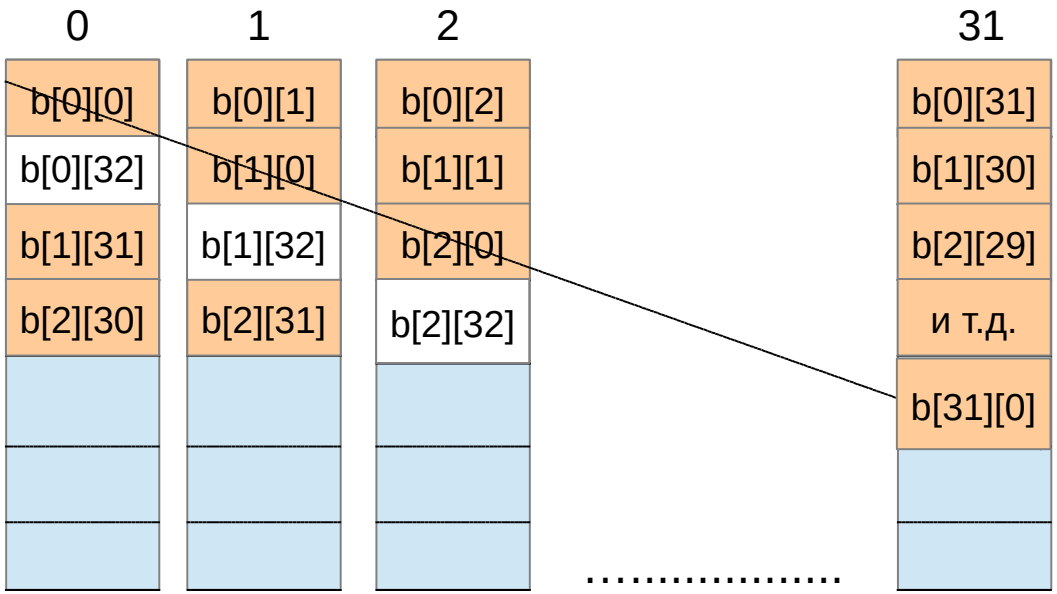
Наивное использование shared memory (статическое выделение памяти)

```
#define SH_DIM 32
__global__ void gTranspose12(float* storage_d,float* storage_d_t){
    __shared__  float buffer_s[SH_DIM][SH_DIM];

    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;
    int N=blockDim.x*gridDim.x;

    buffer_s[threadIdx.y][threadIdx.x]=storage_d[i+j*N];
    __syncthreads();

    i=threadIdx.x+blockIdx.y*blockDim.x;
    j=threadIdx.y+blockIdx.x*blockDim.y;
    storage_d_t[i+j*N]=buffer_s[threadIdx.x][threadIdx.y];
}
```

Как избежать конфликта банков разделяемой памяти

__shared__ float buffer[SH_DIM][SH_DIM+1];

Размещение массива buffer в разделяемой памяти (shared memory):



Использование shared memory с разрешением конфликтов банков

```
__global__ void gTranspose2(float* storage_d,float* storage_d_t){
    __shared__  float buffer[SH_DIM][SH_DIM+1];

    int i=threadIdx.x+blockIdx.x*blockDim.x;
    int j=threadIdx.y+blockIdx.y*blockDim.y;
    int N=blockDim.x*gridDim.x;

    buffer[threadIdx.y][threadIdx.x]=storage_d[i+j*N];
    __syncthreads();

    i=threadIdx.x+blockIdx.y*blockDim.x;
    j=threadIdx.y+blockIdx.x*blockDim.y;
    storage_d_t[i+j*N]=buffer[threadIdx.x][threadIdx.y];
}
```

Ввод размерностей матрицы и блока нитей

```
int main(int argc, char* argv[]){
    if(argc<3){
      fprintf(stderr, "USAGE: matrix  <dimension of matrix>
                                      <dimension_of_threads>\n");
      return -1;}
    int N=atoi(argv[1]);
    int dim_of_threads=atoi(argv[2]);
    if(N%dim_of_threads){
            fprintf(stderr, "change dimensions\n");
            return -1;}
    int dim_of_blocks=N/dim_of_threads;
    const int max_size=1<<8;
    if(dim_of_blocks>max_size){
            fprintf(stderr, "too many blocks\n");
            return -1;  }
```

Выделение памяти и инициализация матрицы

```
float *storage_d, *storage_d_t, *storage_h;

cudaMalloc((void**)&storage_d, N*N*sizeof(float));
cudaMalloc((void**)&storage_d_t, N*N*sizeof(float));
storage_h=(float*)calloc(N*N, sizeof(float));

gInitializeStorage<<<dim3(dim_of_blocks,dim_of_blocks),
                     dim3(dim_of_threads,dim_of_threads)>>>(storage_d);
cudaDeviceSynchronize();

memset(storage_h,0.0,N*N*sizeof(float));
cudaMemcpy(storage_h,  storage_d, N*N*sizeof(float),
                                          cudaMemcpyDeviceToHost);
Output(storage_h, N);
```

Запуск gTranspose0

```
gTranspose0<<<dim3(dim_of_blocks, dim_of_blocks),
      dim3(dim_of_threads,dim_of_threads)>>>(storage_d,storage_d_t);
cudaDeviceSynchronize();

memset(storage_h,0.0,N*N*sizeof(float));
cudaMemcpy(storage_h,  storage_d_t, N*N*sizeof(float),
                                                cudaMemcpyDeviceToHost);
Output(storage_h, N);
```

Запуск gTranspose1

```
gTranspose11<<<dim3(dim_of_blocks, dim_of_blocks),
                dim3(dim_of_threads,dim_of_threads),
                dim_of_threads*dim_of_threads*sizeof(float)>>>
                                        (storage_d,storage_d_t);
cudaDeviceSynchronize();

memset(storage_h,0.0,N*N*sizeof(float));
cudaMemcpy(storage_h,  storage_d_t, N*N*sizeof(float),
                                        cudaMemcpyDeviceToHost);
Output(storage_h, N);
```

Запуск gTranspose12

```
gTranspose12<<<dim3(dim_of_blocks, dim_of_blocks),
                        dim3(dim_of_threads,dim_of_threads)>>>
                                                    (storage_d,storage_d_t);
cudaDeviceSynchronize();

memset(storage_h,0.0,N*N*sizeof(float));
cudaMemcpy(storage_h,  storage_d_t, N*N*sizeof(float),
                                                    cudaMemcpyDeviceToHost);
Output(storage_h, N);
```

Запуск gTranspose2

```
gTranspose2<<<dim3(dim_of_blocks, dim_of_blocks),
                        dim3(dim_of_threads,dim_of_threads)>>>
                                                    (storage_d,storage_d_t);
cudaDeviceSynchronize();

memset(storage_h,0.0,N*N*sizeof(float));
cudaMemcpy(storage_h,  storage_d_t, N*N*sizeof(float),
                                                    cudaMemcpyDeviceToHost);
Output(storage_h, N);
```

Освобождение ресурсов

```
  cudaFree(storage_d);
  cudaFree(storage_d_t);
  free(storage_h);

  return 0;
}
```

Ускорение за счет грамотного использования разделяемой памяти

```
...>nvprof ./matrix 256 32 > tmp
==3642== NVPROF is profiling process 3642, command: ./matrix 256 32
256     32      8
==3642== Profiling application: ./matrix2 256 32
==3642== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 63.08%  202.84us         5  40.567us  40.349us  41.054us  [CUDA memcpy DtoH]
  9.63%  30.958us         1  30.958us  30.958us  30.958us  gTranspose0(float*, float*)
  9.53%  30.642us         1  30.642us  30.642us  30.642us  gTranspose11(float*, float*)
  9.16%  29.441us         1  29.441us  29.441us  29.441us  gTranspose12(float*, float*)
  5.19%  16.678us         1  16.678us  16.678us  16.678us  gTranspose2(float*, float*)
  3.42%  11.011us         1  11.011us  11.011us  11.011us  InitializeStorage(float*)
```

Определение эффективности использования разделяемой памяти

```
...> nvprof -m shared_efficiency  ./matrix 256  32 > tmp
Invocations       Metric Name         Metric Description
                  Min         Max         Avg
Device "GeForce GTX 560 Ti (0)"
Kernel: gTranspose0(float*, float*)
          1         shared_efficiency   Shared Memory Efficiency
                  0.00%       0.00%       0.00%
Kernel: gTranspose2(float*, float*)
                100.00%     100.00%     100.00%
Kernel: gTranspose11(float*, float*)
                  6.06%       6.06%       6.06%
Kernel: gTranspose12(float*, float*)
                  6.06%       6.06%       6.06%
```