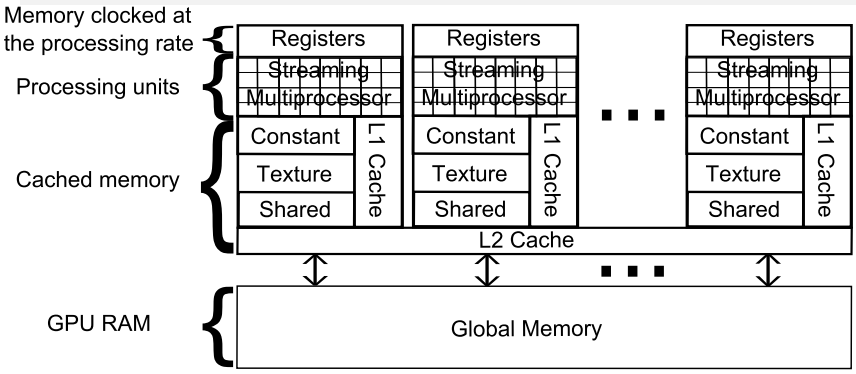
Shared: Second fastest memory



Shared memory

- Cached memory visible to all threads of the same block
- Has a lifetime of a kernel
- Static allocation of arrays: `__shared__ float A[100];`
- Dynamic allocation of arrays: `extern __shared__ float A[];`
  kernel call: `myKernel<<<..., ..., 100*sizeof(float)>>>(...);`

High bandwidth

- Divided into equally sized memory modules (banks)
- Any memory load/store of n addresses ($n \leq 32$) in n distinct memory banks can be performed simultaneously
- Multiple accesses to the same memory bank are serialized, except for the same memory location accessed by warp threads (broadcast)

Increase the size of shared in GPUs

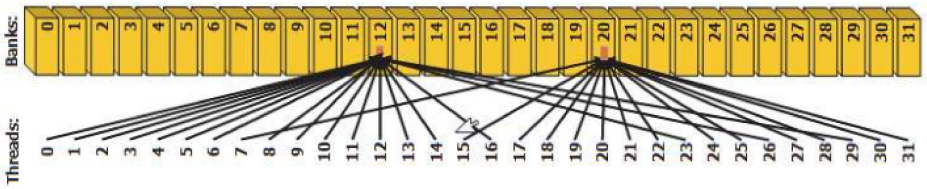
with `cudaFuncAttributeMaxDynamicSharedMemorySize` in `cudaFuncSetAttribute()`

EUMaster4HPC

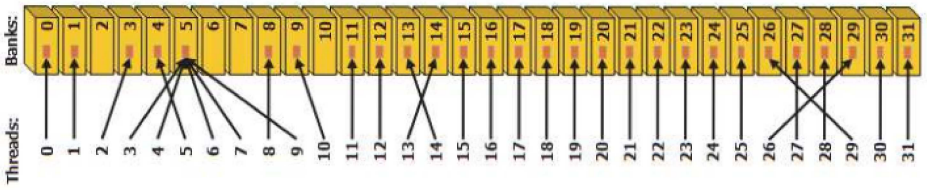
Sarr is an array in the shared memory

Figure from CUDA programming guide: Nvidia

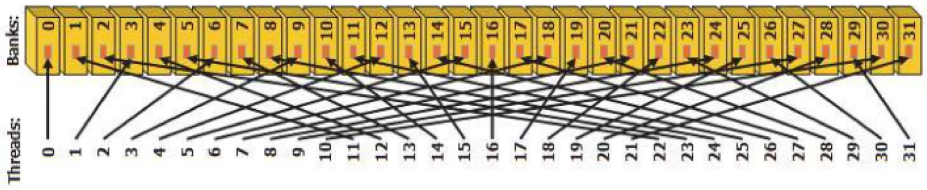
no bank conflict (broadcast)



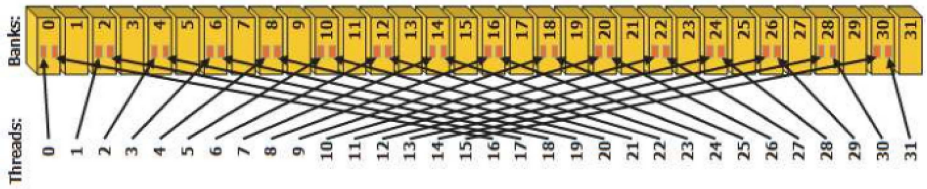
no bank conflict



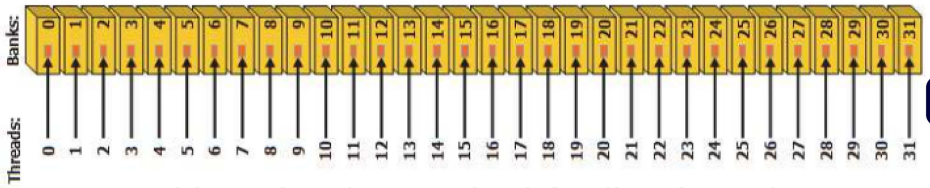
Sarr[3*threadIdx.x] no bank conflict



Sarr[2*threadIdx.x] two-way bank conflict



Sarr[threadIdx.x] no bank conflict



EUMaster4HPC

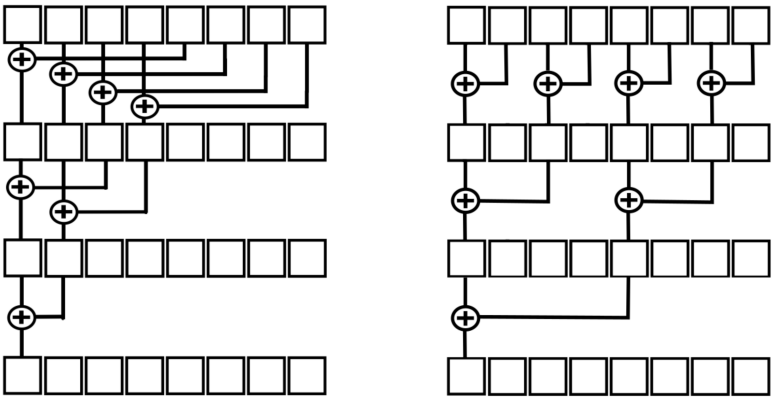
Avoiding bank conflicts

Some facts

- Threads can access to any memory space of the shared memory of their block
- The synchronization barrier `__syncthreads();` ensures that threads of the same block wait for all other threads of the same block.
- Threads of different blocks cannot exchange values within the same kernel

Dot product

- Store the product result in the shared memory then perform a reduction using the following scheme (left) with a `__syncthreads();` at each step



- `atomicAdd` is used for the reduction through blocks
- What happens when `atomicAdd` is used for the whole sum?

EUMaster4HPC

```
1  __global__ void ShaAtom_k(float* A, float* B, float* C) {
2
3      int idx = threadIdx.x + blockIdx.x * blockDim.x;
4      int i;
5      __shared__ float sC[NTPB];
6
7      sC[threadIdx.x] = A[idx] * B[idx];
8      __syncthreads();
9
10     i = blockDim.x / 2;
11     while (i != 0) {
12         if (threadIdx.x < i) {
13             sC[threadIdx.x] += sC[threadIdx.x + i];
14         }
15         __syncthreads();
16         i /= 2;
17     }
18
19     if (threadIdx.x == 0) {
20         atomicAdd(C, sC[0]);
21     }
22 }
```

EUMaster4HPC

```
1  ////////////////Bad alternative//////////////////////
2  int i = blockDim.x / 2;
3  while (i != 0) {
4      if (threadIdx.x < i) {
5          sC[threadIdx.x] += sC[threadIdx.x + i];
6          __syncthreads();
7      }
8      i /= 2;
9  }
10
11 ////////////another bad example//////////////////////
12 if (/*boolean that depends on each thread*/) {
13     /*some operations*/
14     __syncthreads();
15 }
16
17 ////////////another bad example//////////////////////
18 for (int i = 0; i < threadIdx.x * threadIdx.x; i++) {
19     /*some operations*/
20     __syncthreads();
21 }
```

EUMaster4HPC