


Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program of the Lobachevsky State University of Nizhni Novgorod among the world's research and education centers

Strategic initiative “Achieving leading positions in the field of supercomputer technology and high-performance computing”

Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Introduction to GPU programming

05 Lecture
CUDA thread execution and memory hierarchy

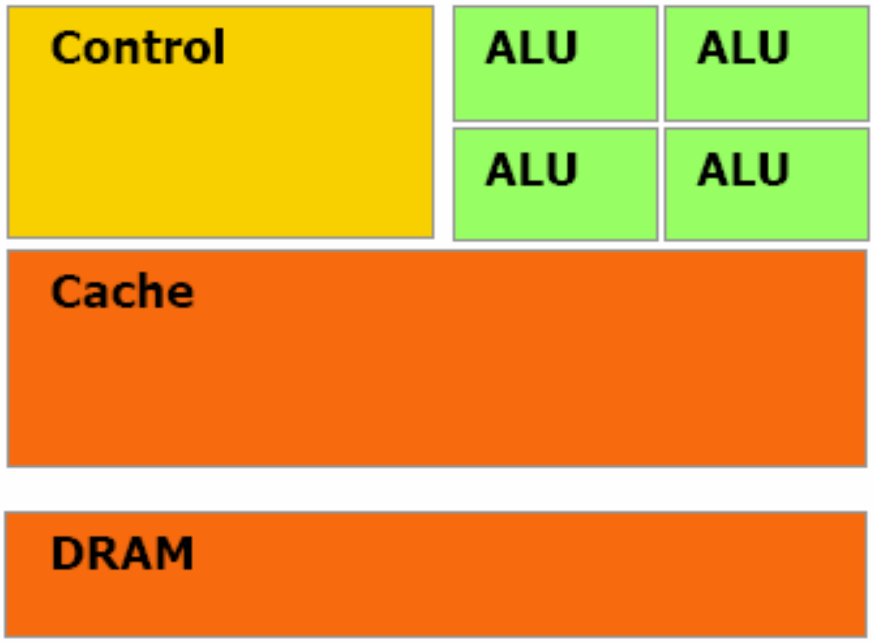
Bastrakov S.I.
Software department

Contents

- GPU architecture
- Thread execution
- Memory hierarchy

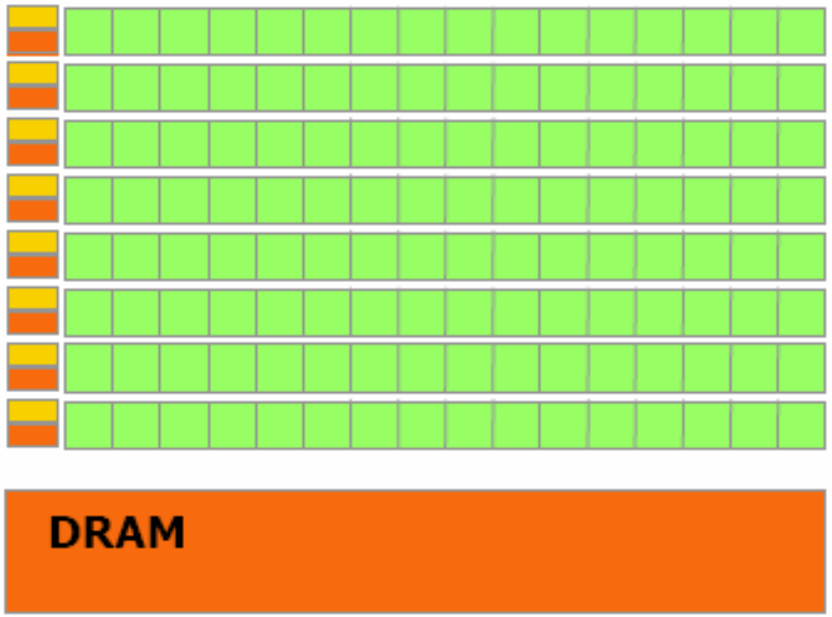
GPU architecture

CPU and GPU architecture



CPU

"cache-oriented"



GPU

"cache-miss oriented"

Image source: NVIDIA CUDA C Programming Guide v. 6.5

Tesla 8 multiprocessor

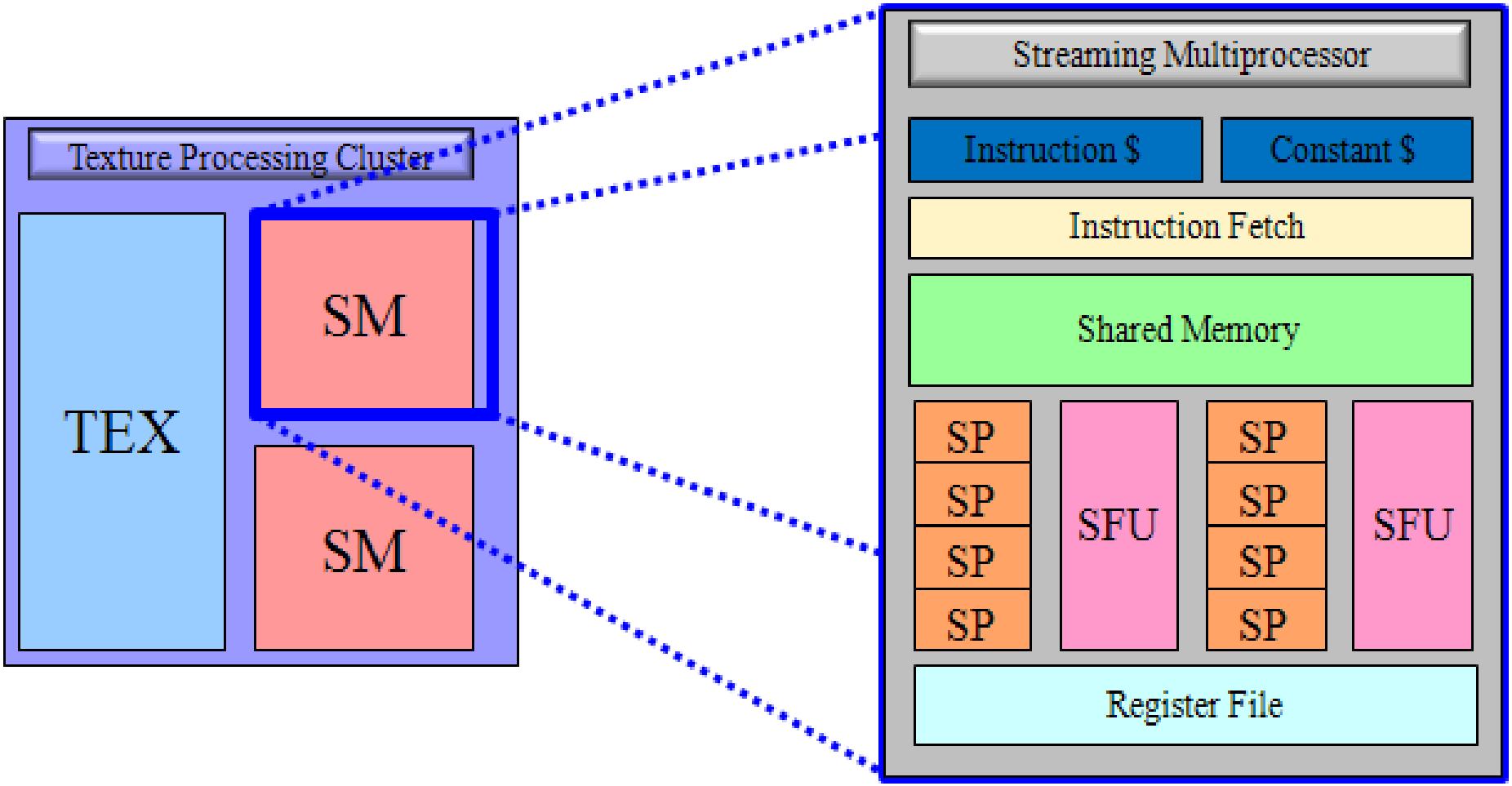


Image source:A.V. Boreskov, A.A. Harlamov "Architecture and programming of massively parallel computational systems".

Tesla 10 multiprocessor

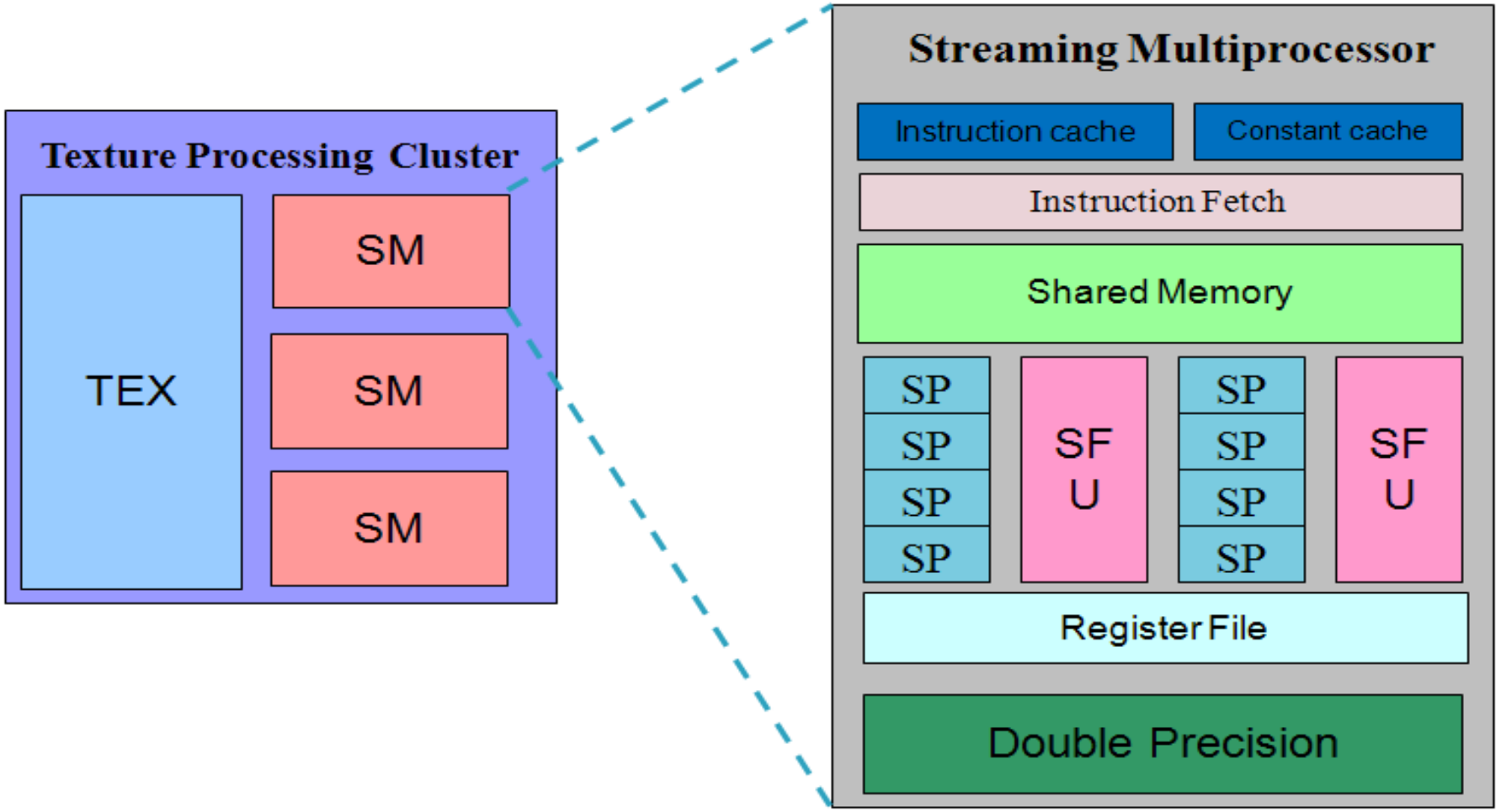


Image source:A.V. Boreskov, A.A. Harlamov "Architecture and programming of massively parallel computational systems".

Tesla 10 memory hierarchy

- device/global
- shared
- constant cache
- texture cache
- register
- local

Fermi multiprocessor

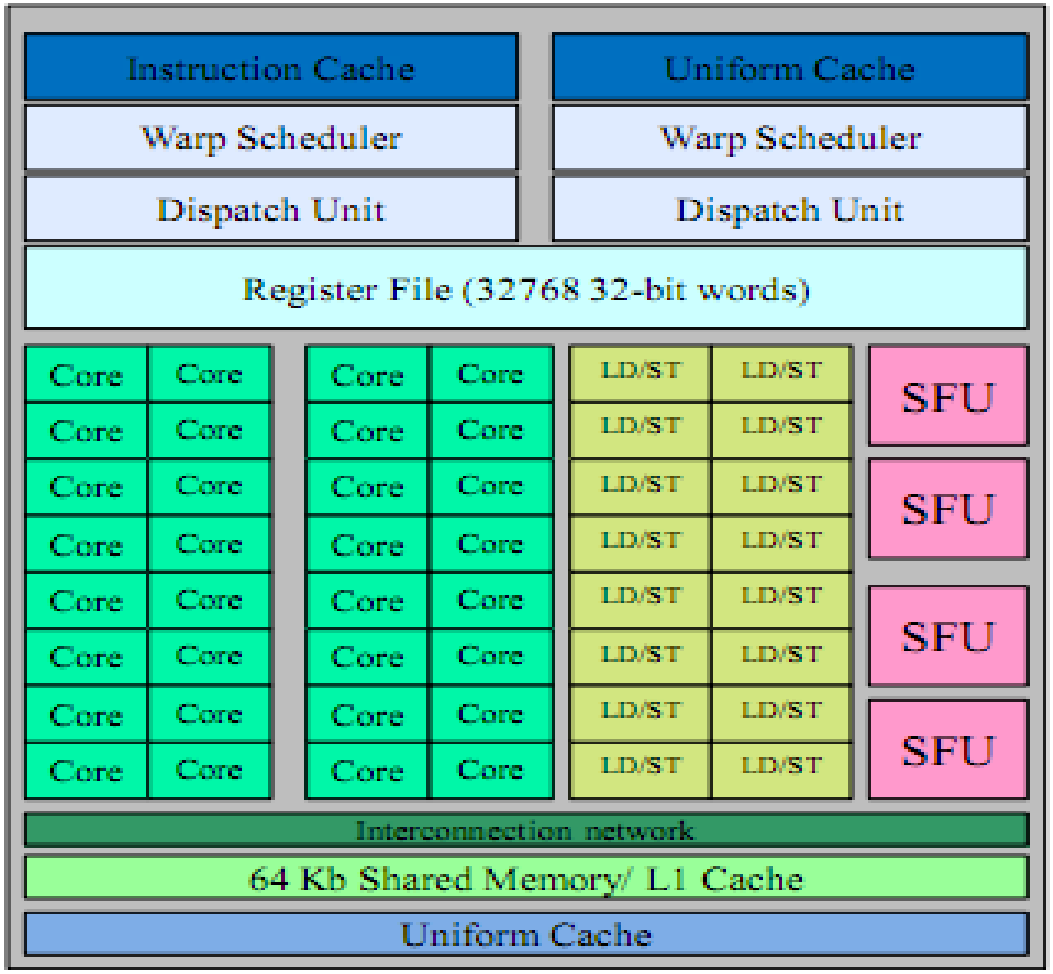


Image source:A.V. Boreskov, A.A. Harlamov "Architecture and programming of massively parallel computational systems".

Thread execution

Thread communication

- Threads of the same block are executed on the same multiprocessor, they can communicate using:
  - shared memory;
  - __**syncthreads()**.
- Threads of different block can communicate only via global memory.
- There are atomic operations in shared and global memory.

Kernel execution

- Blocks are distributed between multiprocessors
- Threads of a block are executed by CUDA-cores of a multiprocessor
- If there are enough resources, several blocks can be concurrently executed on the same multiprocessor.
- Large amount of blocks helps automatic scaling.

Automatic scaling

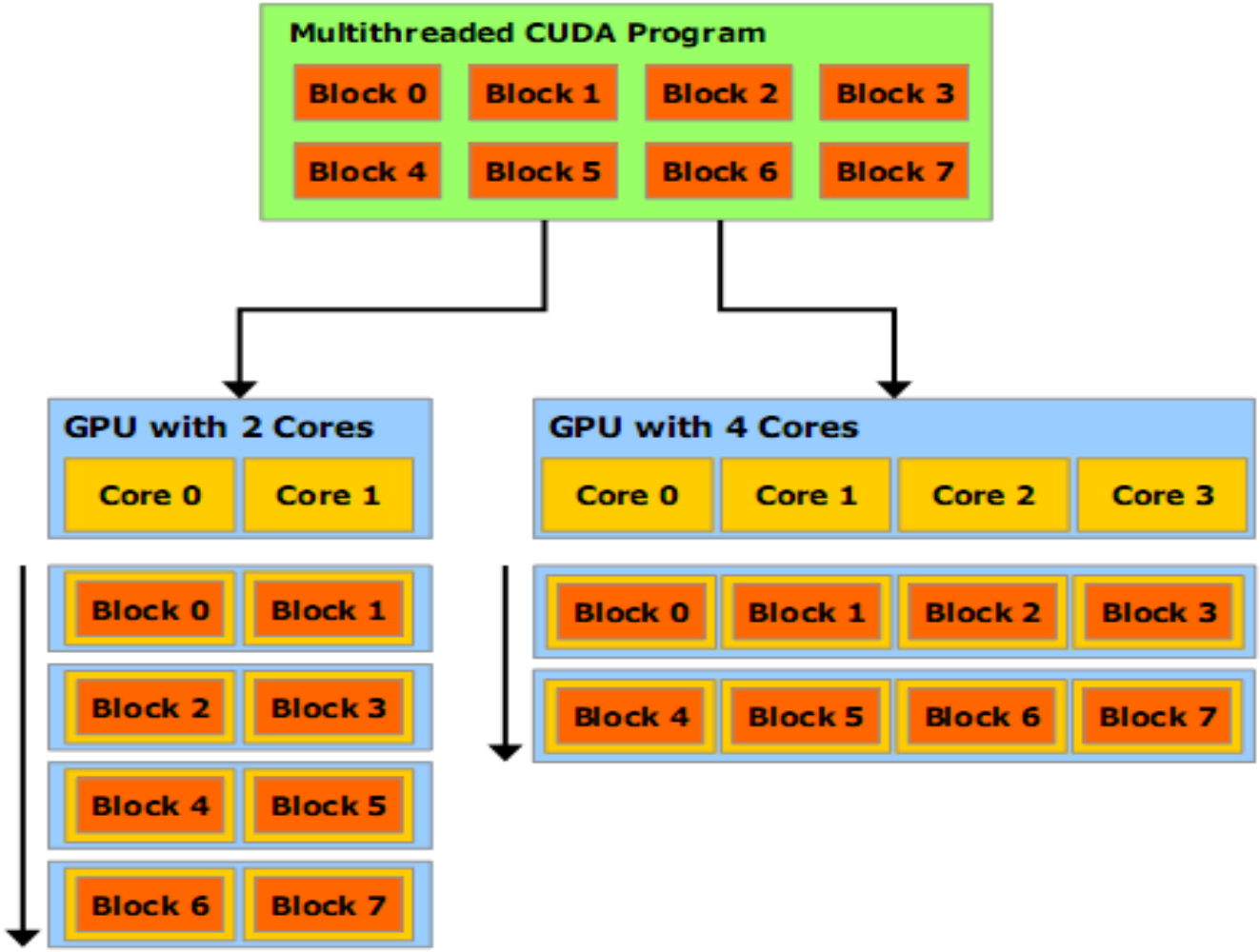


Image source: NVIDIA CUDA C Programming Guide v. 6.5

SIMT

- All threads running on the same multiprocessor are grouped in **warps**, a warp consists of threads with sequential identifiers.
  - Warp size is currently 32.
- Warp scheduling is done automatically.
- CUDA-cores of a multiprocessor execute same instruction for all threads in a warp (**SIMT**, Single Instruction Multiple Thread).
  - Threads of a warp are always synchronized.
- Programming in scalar terms: kernel code for one thread, can use conditions.

Example of warp execution

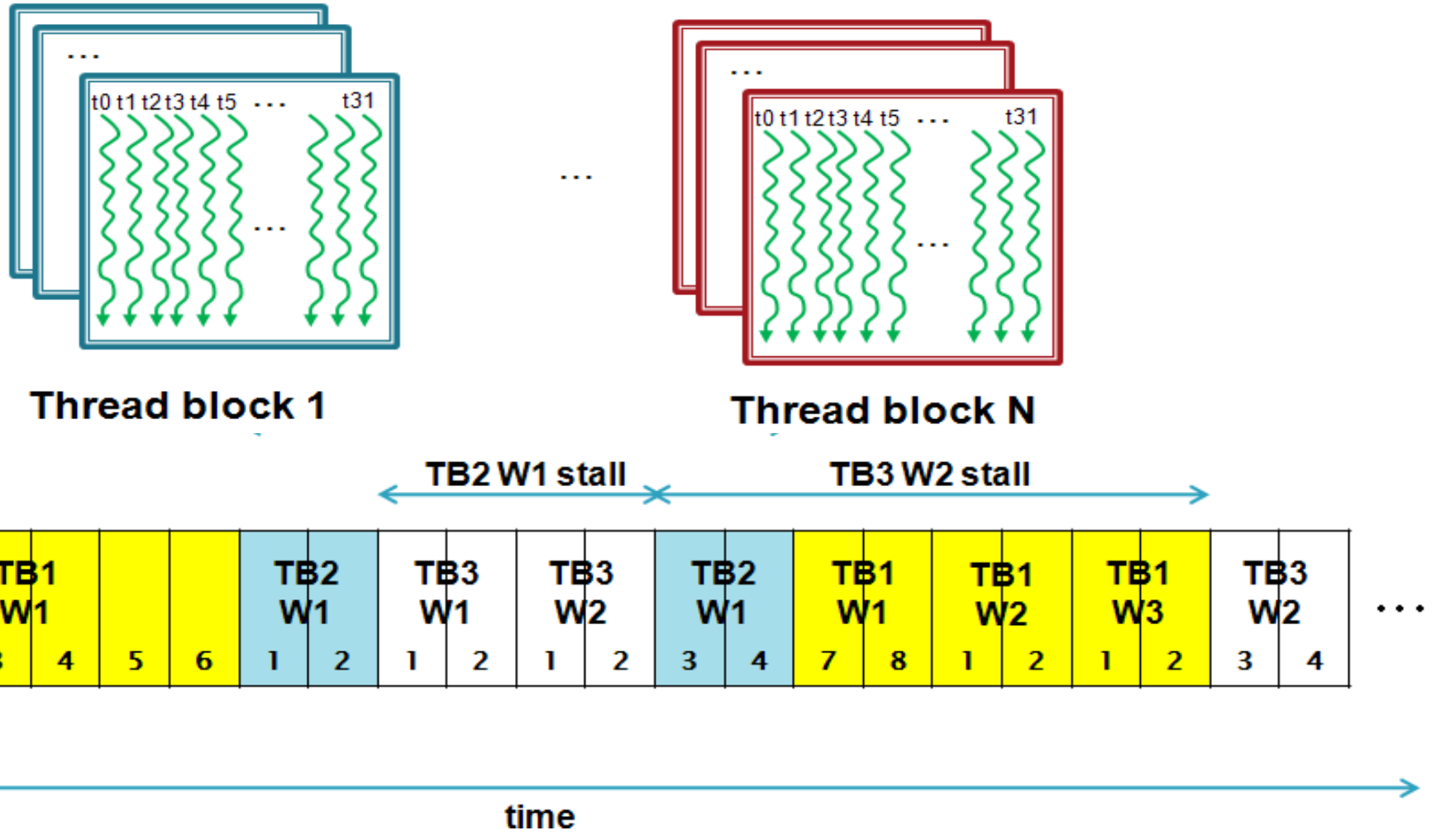


Image source: NVIDIA CUDA C Programming Guide v. 6.5

Memory hierarchy

Memory hierarchy

Memory type	Access	Scope	Speed
Register	R/W	Per-thread	High (on-chip)
Local	R/W	Per-thread	Low (DRAM)
Shared	R/W	Per-block	High (on-chip)
Global*	R/W	Per-grid	High if hit cache (DRAM)
Constant	R/O	Per-grid	High if hit cache
Texture	R/O	Per-grid	High if hit cache

*+ L1/L2 cache from Fermi

Image source:A.V. Boreskov, A.A. Harlamov "Architecture and programming of massively parallel computational systems".

Using global memory

- cudaError_t **cudaMalloc** (void** devPtr, size_t count
- cudaError_t **cudaFree** (void* devPtr)
- cudaError_t **cudaMemcpy** (void* dst, const void* src, size_t count, enum cudaMemcpyKind kind),

  kind can be cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost.
- **cudaMemcpyAsync** – asynchronous version.

Example

- Allocate arrays on host and device:

```
int n = 1000;
float * a = new float[n], * a_gpu;
cudaMalloc((void**)&a_gpu, n * sizeof(float));
```

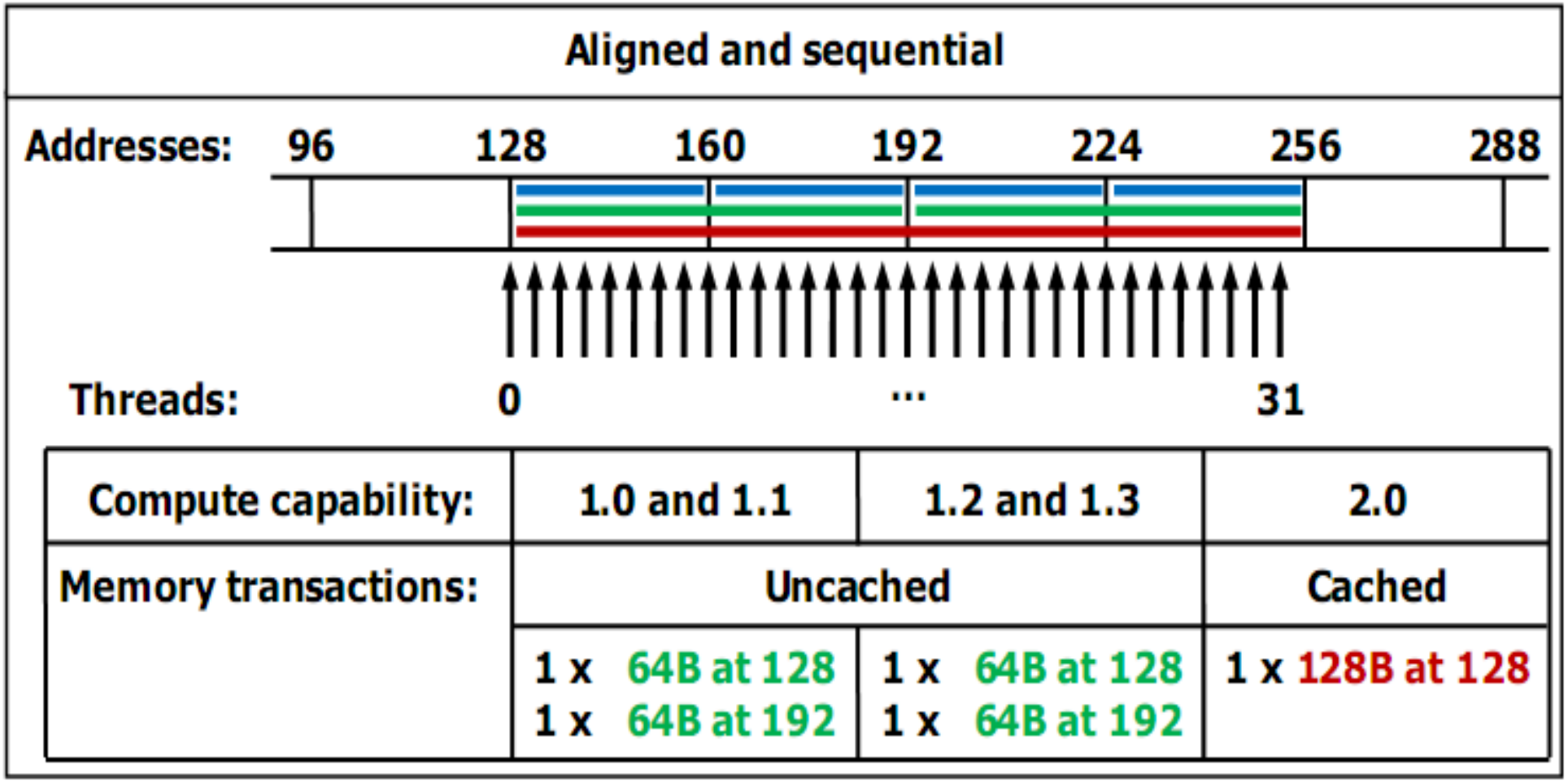
- Copy from host to device:

```
cudaMemcpy(a_gpu, a, n * sizeof(float),
    cudaMemcpyHostToDevice);
```

- Copy from device to host:

```
cudaMemcpy(a, a_gpu, n * sizeof(float),
    cudaMemcpyDeviceToHost);
```

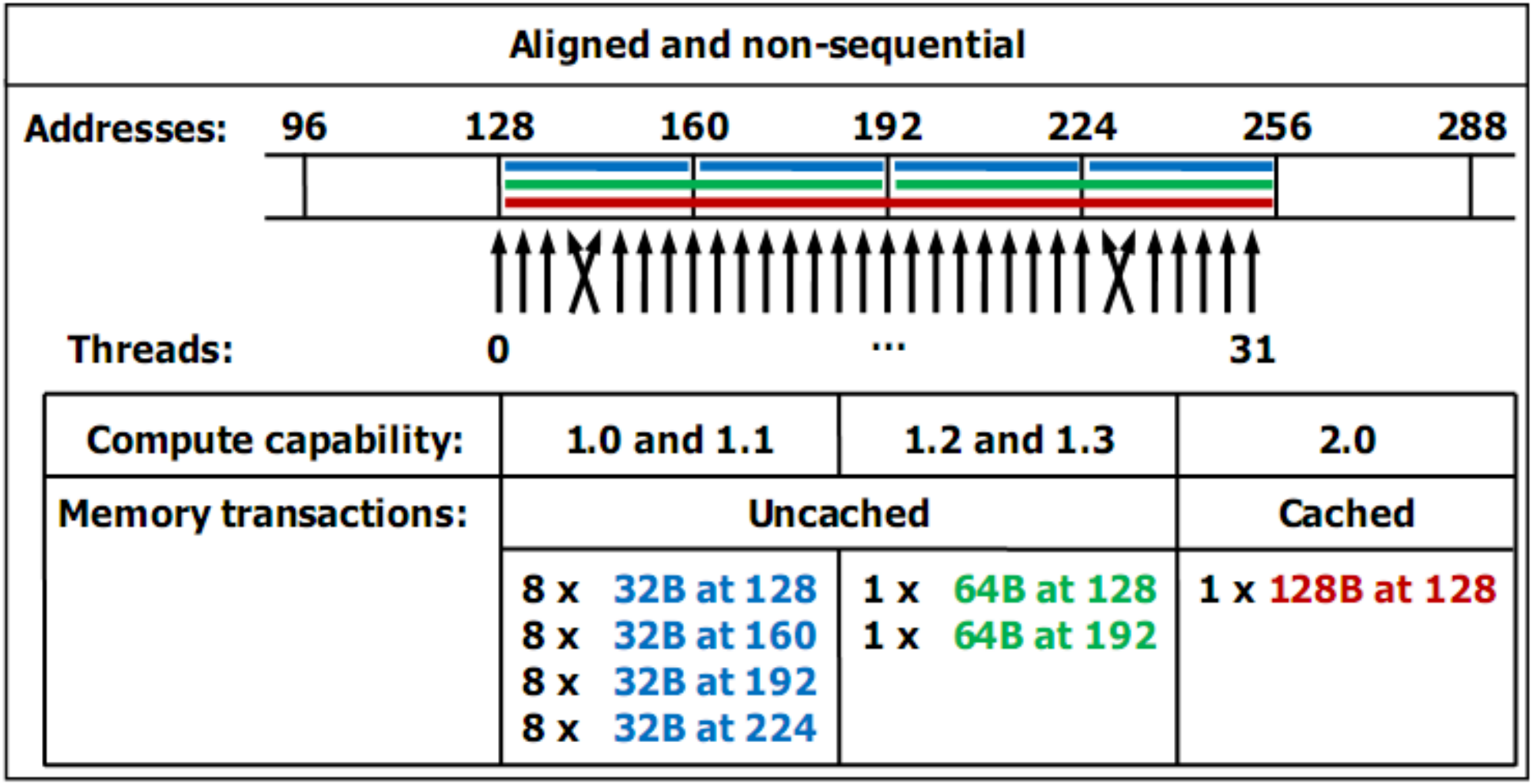
Coalescing for global memory…



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Image source: NVIDIA CUDA C Programming Guide v. 6.5

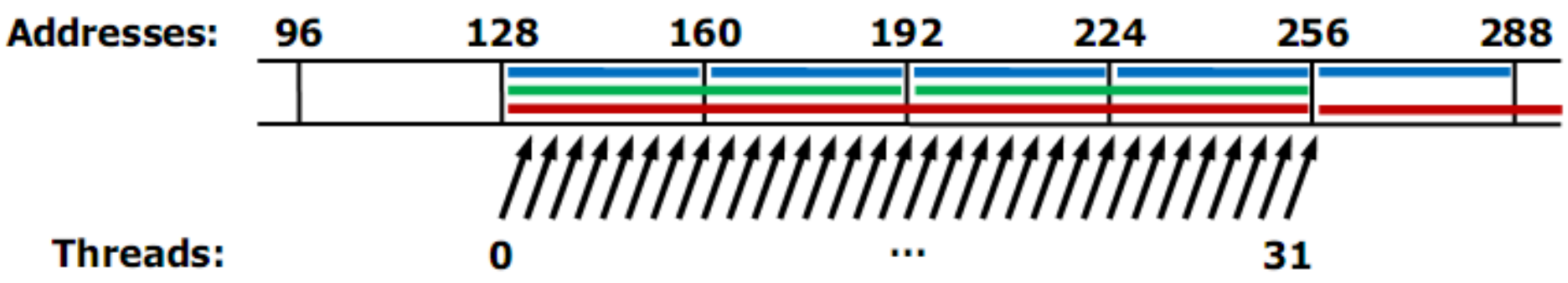
Coalescing for global memory…



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Coalescing for global memory

Misaligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Image source: NVIDIA CUDA C Programming Guide v. 6.5

Example: Jacobi method

```
__global__ void kernel (float * a, float * f, float * x0,    float
   * x1, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int ia = n * idx;
  float sum = 0.0f;
  for (int i = 0; i < n; i++)
    sum += a[ia + i] * x0[i]; /* no coalescing! */
  float alpha = 1.0f / a[ia + idx];
  x1[idx] = x0[idx] + alpha * (f[idx] - sum);
}
```

Working with shared memory

- Latency 4 clocks.
- Small size (16 to 48 KB).
- Typical use pattern:
  - Load intensively used data from global memory;
  - Synchronize if necessary;
  - Compute using loaded data;
  - Synchronize if necessary;
  - Write results back to global memory.

Example

```
__global__ void kernel(int * a)
{
   int globalIdx = blockIdx.x * blockDim.x + threadIdx.x;
   __shared__ int shared_a[NUM_THREADS];
   shared_a[threadIdx.x] = a[globalIdx];
   __syncthreads();
   // all threads of a block can use shared_a
}
```

Dynamically allocated shared memory

- array0 of type short length 2n, array1 of type float length n and array2 of type int length 3n.

```
extern __shared__ float array[];
__global__ void kernel(int n) {
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[2 * n];
    int*   array2 =   (int*)&array1[n];
}
…
int shared_mem_size = sizeof(short) * 2 * n + sizeof(float) * n +
                                  sizeof(int) * 3 * n;
kernel<<<num_blocks, num_threads, shared_mem_size>>>(n);
```

No bank conflicts

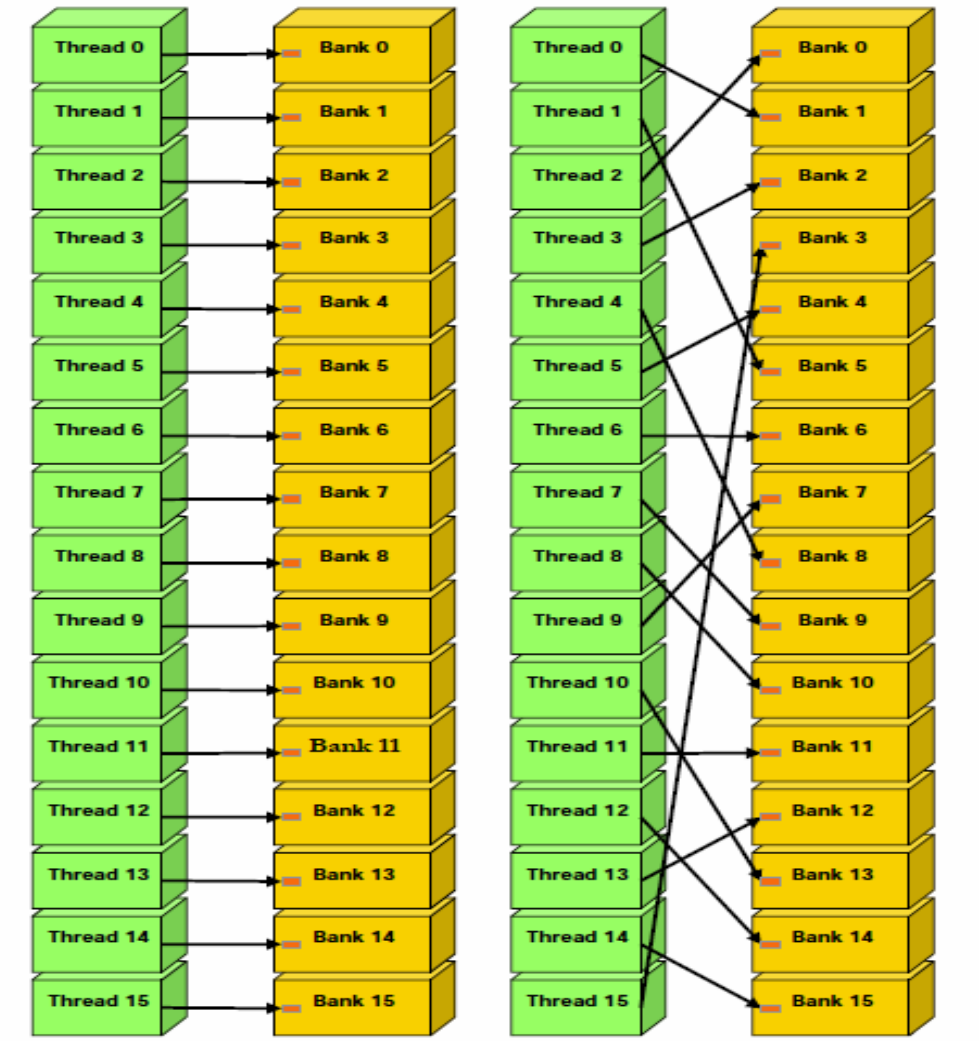


Image source: NVIDIA CUDA C Programming Guide v. 6.5

Bank conflicts

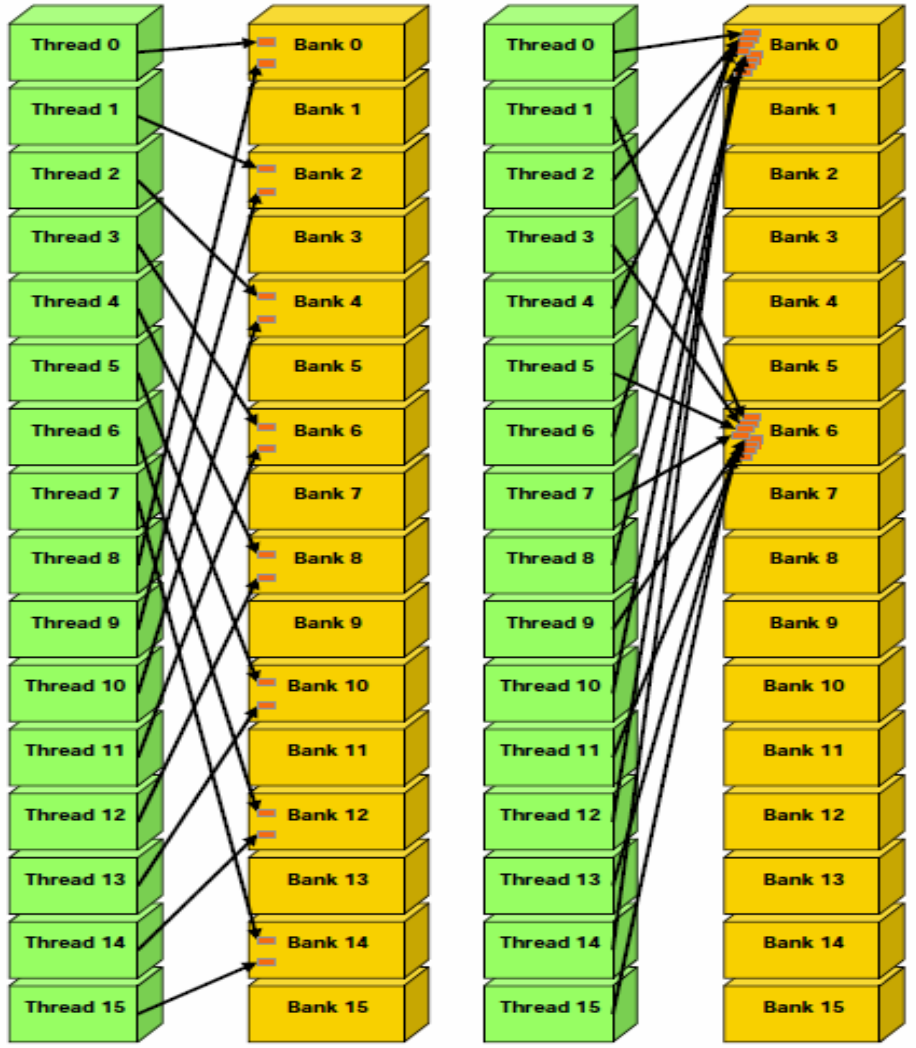


Image source: NVIDIA CUDA C Programming Guide v. 6.5

References

- Sanders J., Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. – Addison-Wesley Professional, 2010. – 312 p.
- Farber R. CUDA Application Design and Development. – Morgan Kaufmann, 2011. – 336 p.
- NVIDIA CUDA C Programming Guide. [http://docs.nvidia.com/cuda/cuda-c-programming-guide/].

Authors

- Bastrakov S.I.,
  Assistant of the Software department of CMC faculty.
  bastrakov@vmk.unn.ru