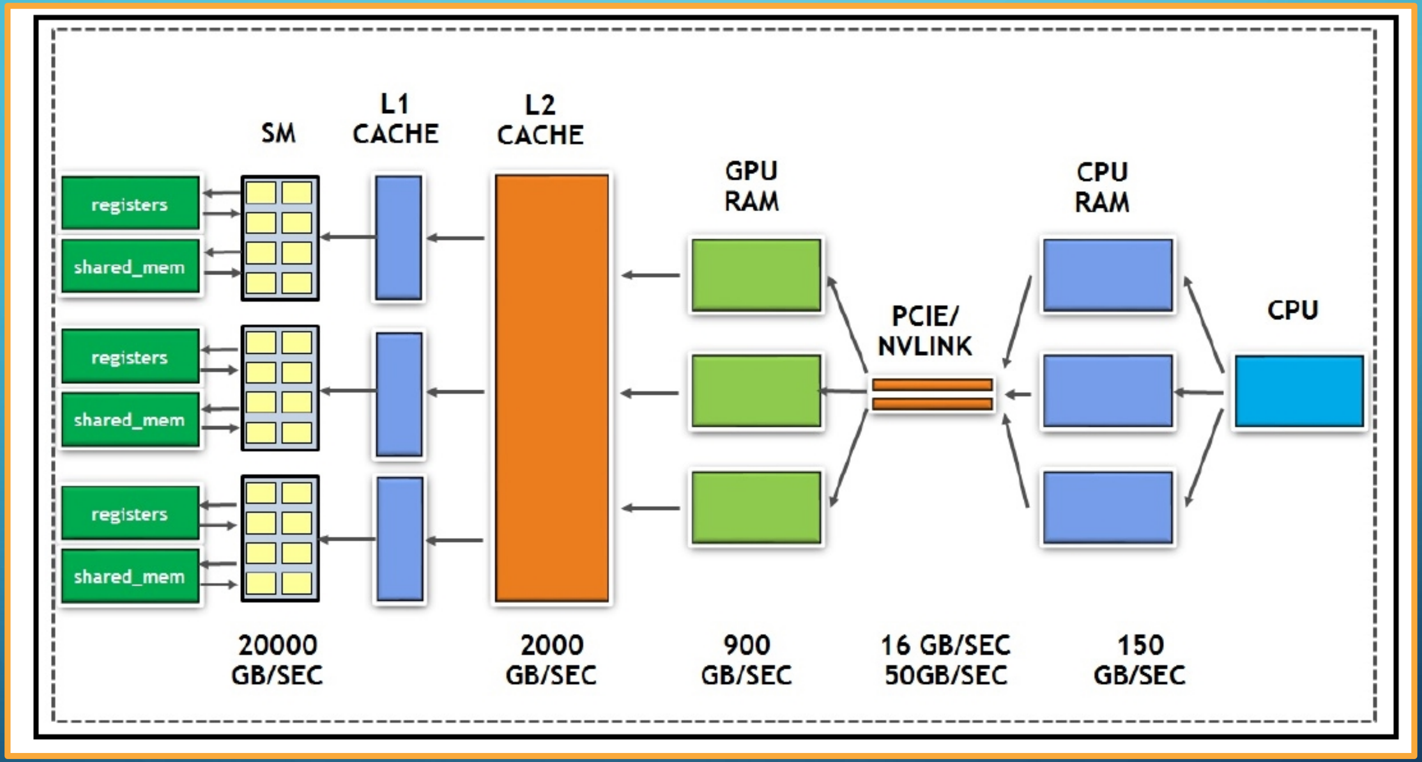
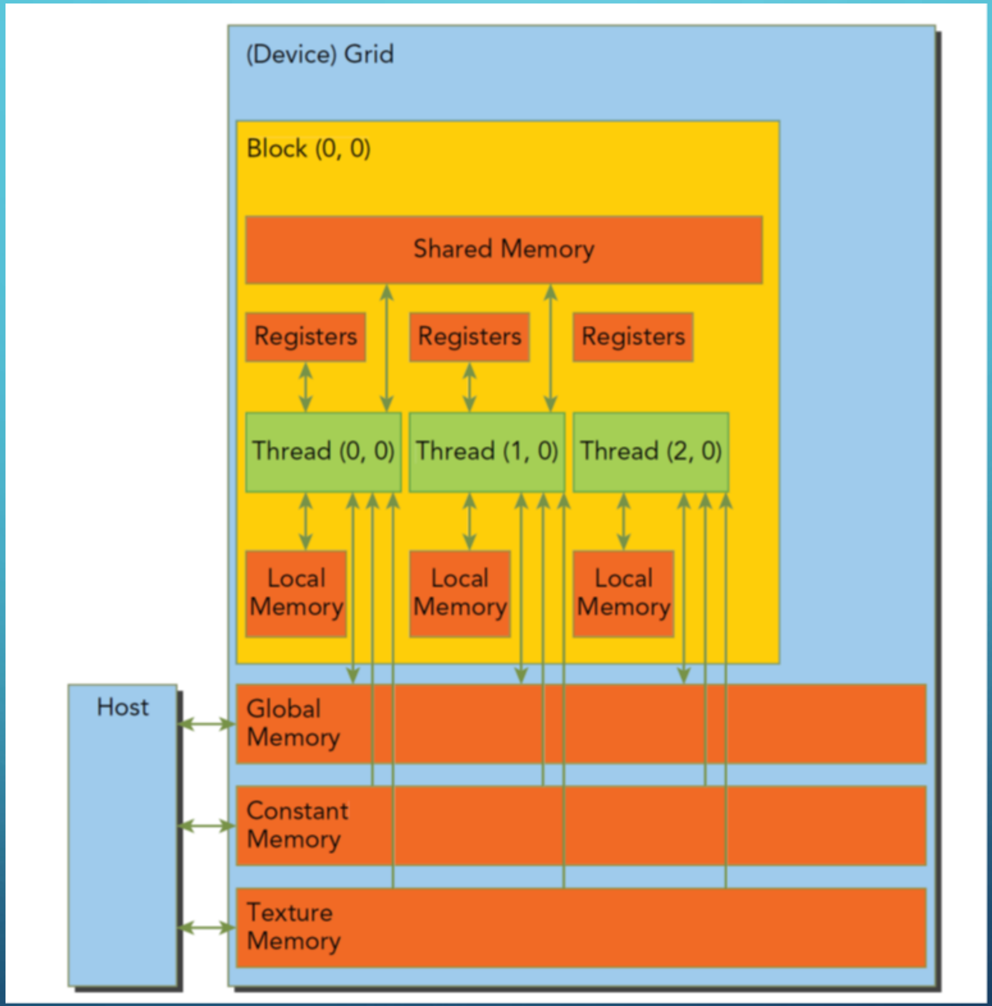
SHARED AND CONSTANT MEMORY

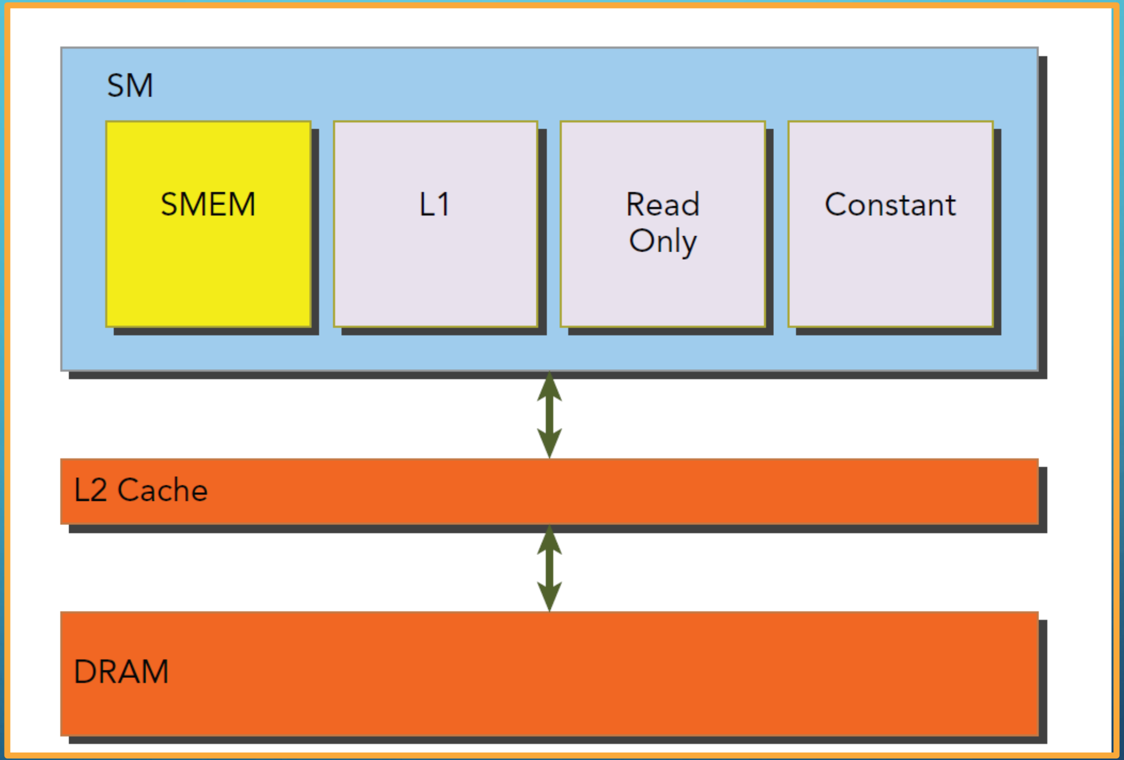
CUDA MEMORY MANAGEMENT



SHARED MEMORY



SHARED MEMORY



SHARED MEMORY

- Shared memory is one of the key components of the GPU.
- A fixed amount of shared memory is allocated to each thread block when it starts executing.
- This shared memory address space is shared by all threads in a thread block.
- Its contents have the same lifetime as the thread block in which it was created.
- Shared memory is partitioned among all resident thread blocks on an SM; therefore, shared memory is a critical resource that limits device parallelism.

SHARED MEMORY

SHARED MEMORY VERSUS GLOBAL MEMORY

GPU global memory resides in device memory (DRAM), and it is much slower to access than GPU shared memory. Compared to DRAM, shared memory has:

- 20 to 30 times lower latency than DRAM
- Greater than 10 times higher bandwidth than DRAM

SHARED MEMORY ALLOCATION

- You can allocate shared memory variables either statically or dynamically.
- Shared memory can also be declared as either local to a CUDA kernel or globally in a CUDA source code file.
- A shared memory variable is declared with the following qualifier:

__shared__

- The following code segment statically declares a shared memory 2D float array.

```
__shared__ float tile[size_y][size_x];
```

- If declared inside a kernel function, the scope of this variable is local to the kernel.
- If declared outside of any kernels in a file, the scope of this variable is global to all kernels.

SHARED MEMORY ALLOCATION

- If the size of shared memory is unknown at compile time, you can declare an un-sized array with the extern keyword.
- For example, the following code segment declares a shared memory 1D un-sized int array.
- This declaration can be made either inside a kernel or outside of all kernels.

```
extern __shared__ int tile[];
```

- Because the size of this array is unknown at compile-time, you need to dynamically allocate shared memory at each kernel invocation by specifying the desired size in bytes as a third argument inside the triple angled brackets, as follows:

```
kernel<<<grid, block, isize * sizeof(int)>>>(...)
```

SHARED MEMORY ALLOCATION

```
__global__ void setRowReadColDyn(int *out) {
    // dynamic shared memory
    extern  __shared__ int tile[];

    // mapping from thread index to global memory index
    unsigned int row_idx = threadIdx.y * blockDim.x + threadIdx.x;
    unsigned int col_idx = threadIdx.x * blockDim.y + threadIdx.y;

    // shared memory store operation
    tile[row_idx] = row_idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[row_idx] = tile[col_idx];
}
```

```
setRowReadColDyn<<<grid, block, BDIMX * BDIMY * sizeof(int)>>>(d_C);
```

SYNCHRONIZATION

- Synchronization among parallel threads is a key mechanism for any parallel computing language.
- As its name suggests, shared memory can be simultaneously accessed by multiple threads within a thread block.
- Doing so will cause inter-thread conflicts when the same shared memory location is modified by multiple threads without synchronization.
- In general, there are two basic approaches to synchronization:
  - Barriers
  - Memory fences
- At a barrier, all calling threads wait for all other calling threads to reach the barrier point.
- At a memory fence, all calling threads stall until all modifications to memory are visible to all other calling threads.
- To explicitly force a certain ordering for program correctness, memory fences and barriers must be inserted in application code.
- This is the only way to guarantee the correct behavior of a kernel that shares resources with other threads.

EXPLICIT BARRIER

- In CUDA, it is only possible to perform a barrier among threads in the same thread block.
- You can specify a barrier point in a kernel by calling the following intrinsic function:

void __syncthreads();

- __syncthreads acts as a barrier point at which threads in a block must wait until all threads have reached that point.
- __syncthreads also ensures that all global and shared memory accesses made by these threads prior to the barrier point are visible to all threads in the same block.
- __syncthreads is used to coordinate communication between the threads of the same block.

EXPLICIT BARRIER

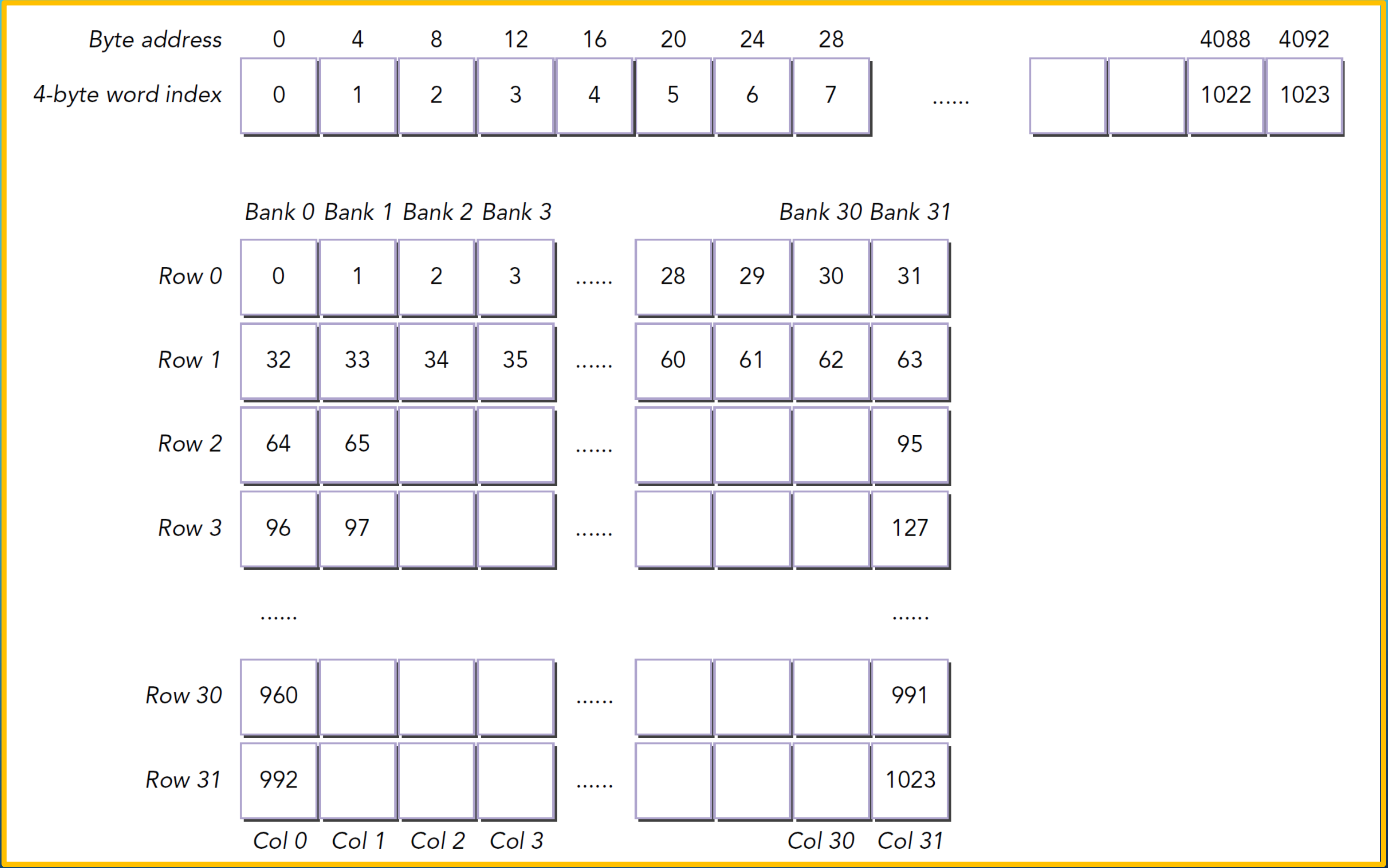
- You must be particularly careful when using __syncthreads in conditional code.
- It is only valid to call __syncthreads if a conditional is guaranteed to evaluate identically across the entire thread block.
- Otherwise execution is likely to hang or produce unintended side effects.
- For example, the following code segment may cause threads in a block to wait indefinitely for each other because all threads in a block never hit the same barrier point.

```
if (threadID % 2 == 0) {
    __syncthreads();
} else {
    __syncthreads();
}
```

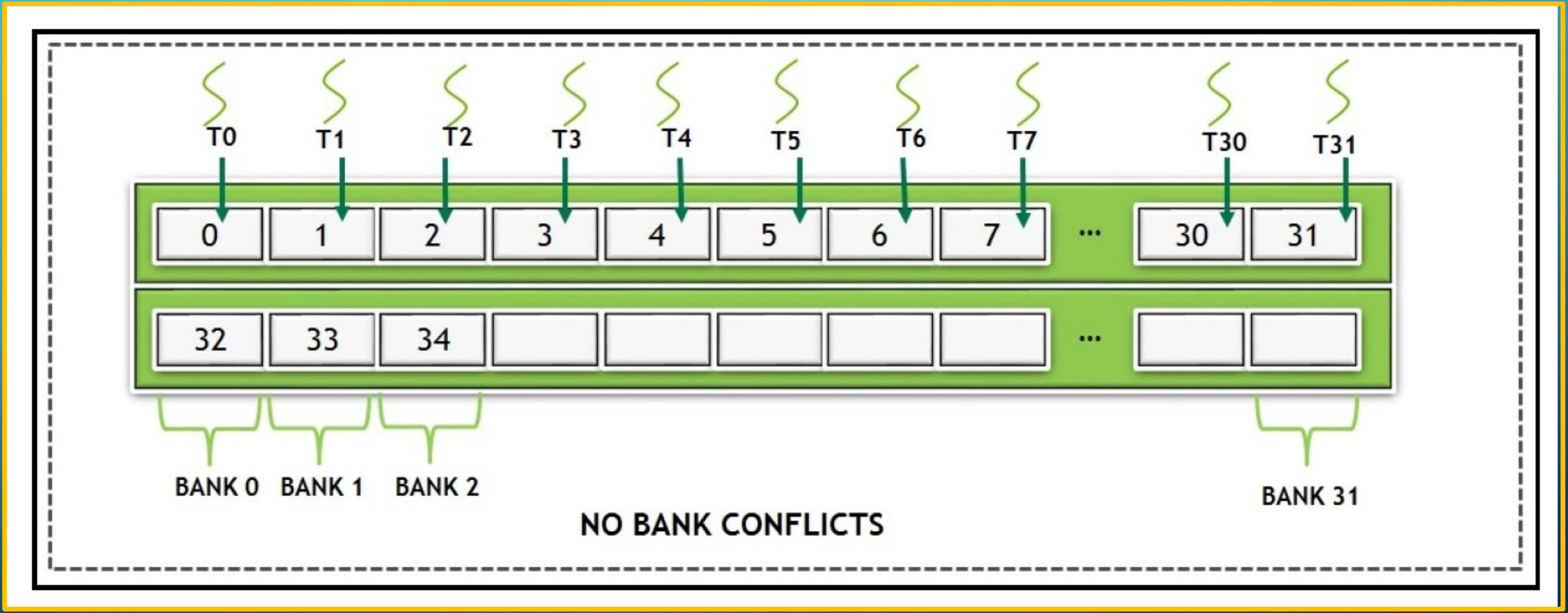
BANK CONFLICTS

- When multiple addresses in a shared memory request fall into the same memory bank, a bank conflict occurs, causing the request to be replayed.
- Three typical situations occur when a request to shared memory is issued by a warp:
- ➤ Parallel access: multiple addresses accessed across multiple banks
- ➤ Serial access: multiple addresses accessed within the same bank
- ➤ Broadcast access: a single address read in a single bank

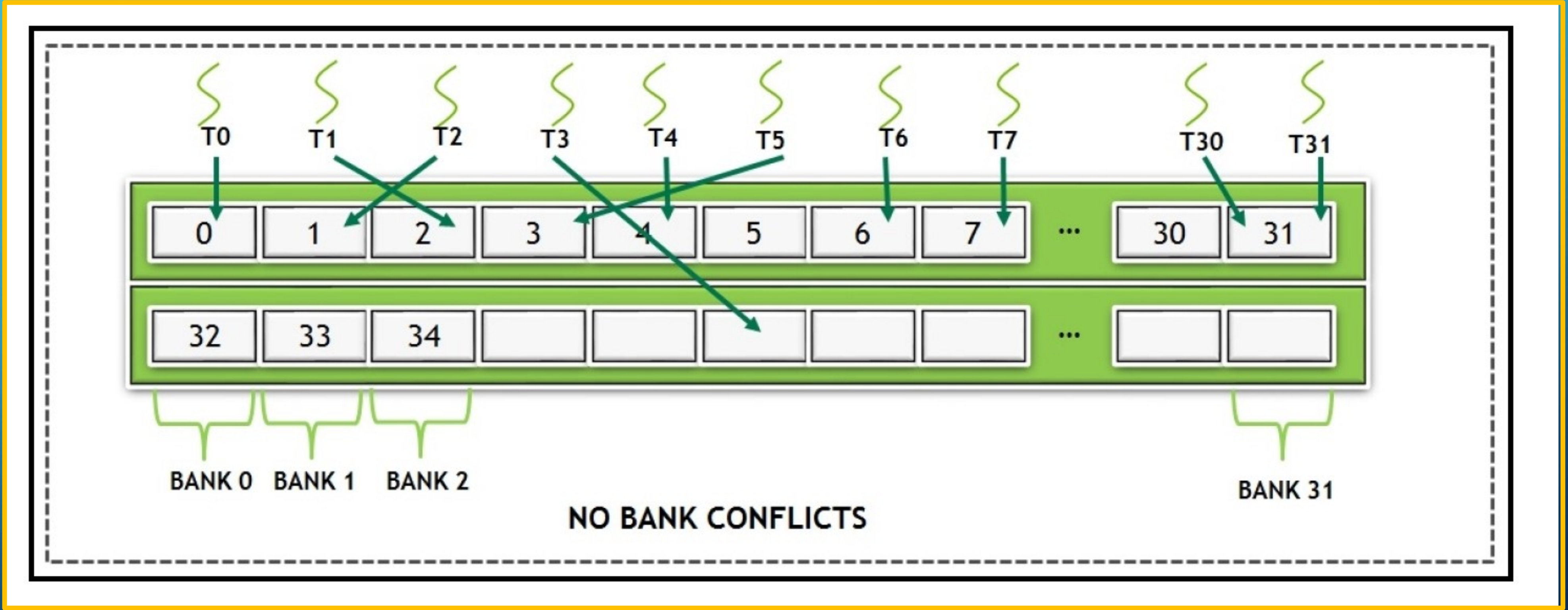
BANK CONFLICTS



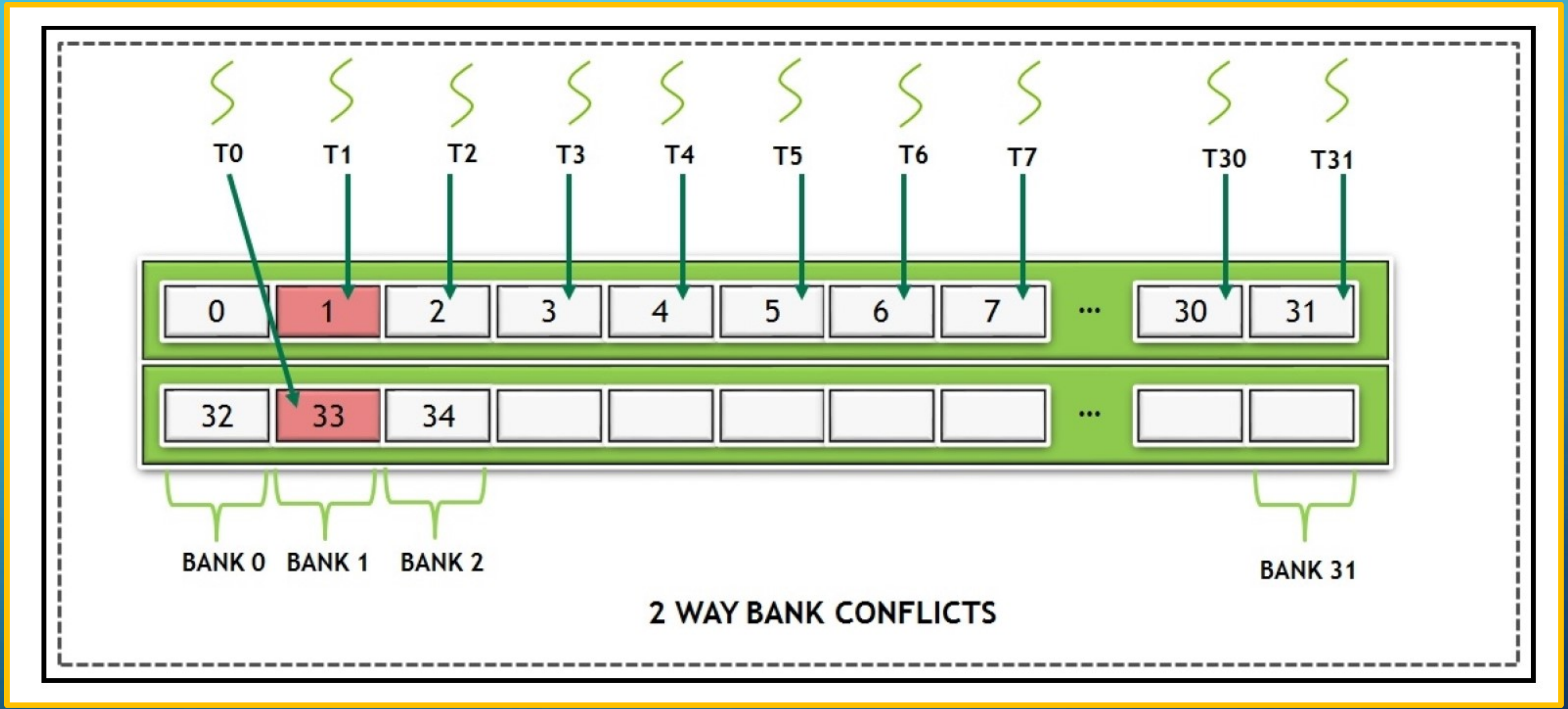
BANK CONFLICTS



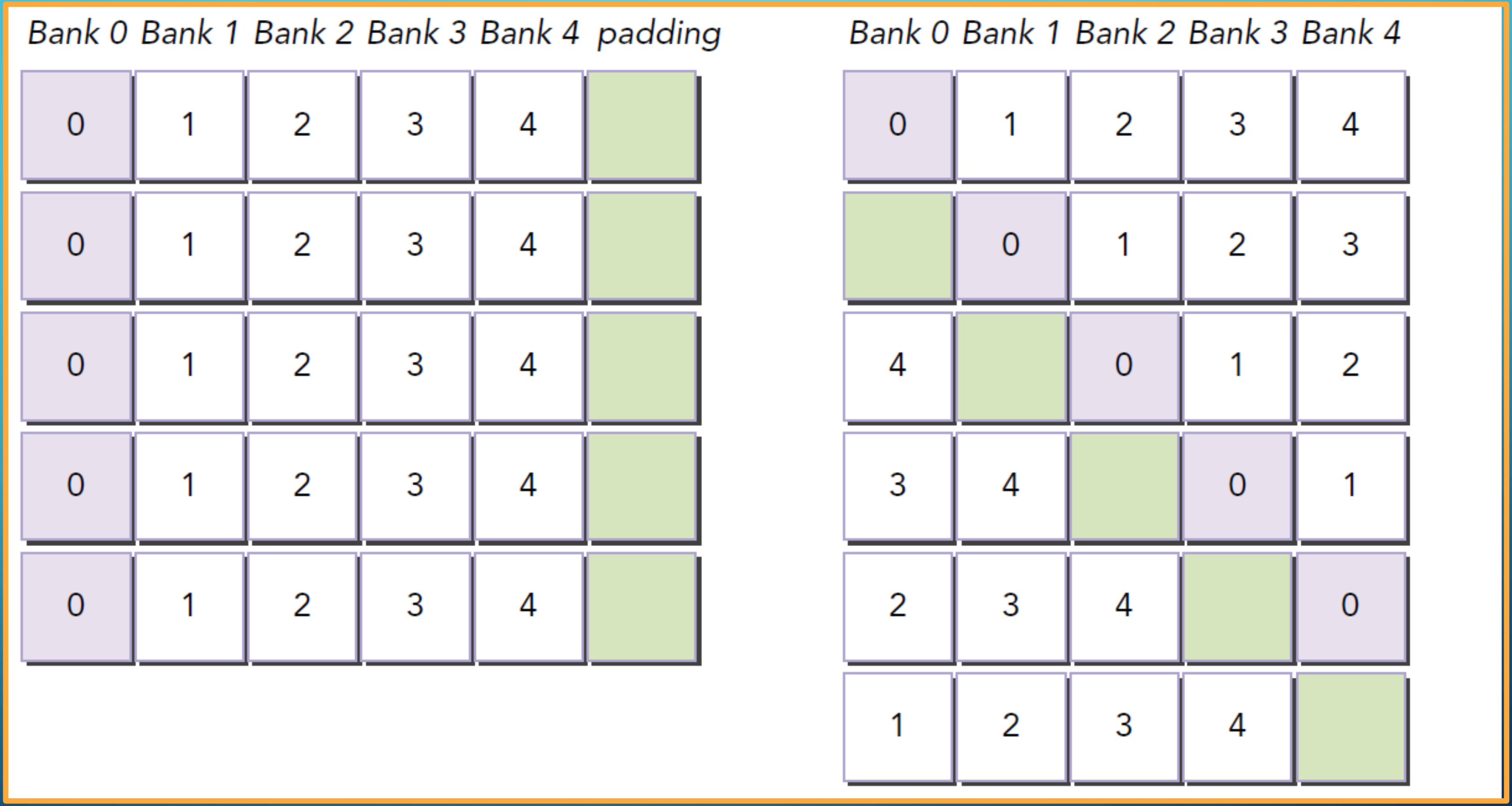
BANK CONFLICTS



BANK CONFLICTS



MEMORY PADDING



SQUARE SHARED MEMORY

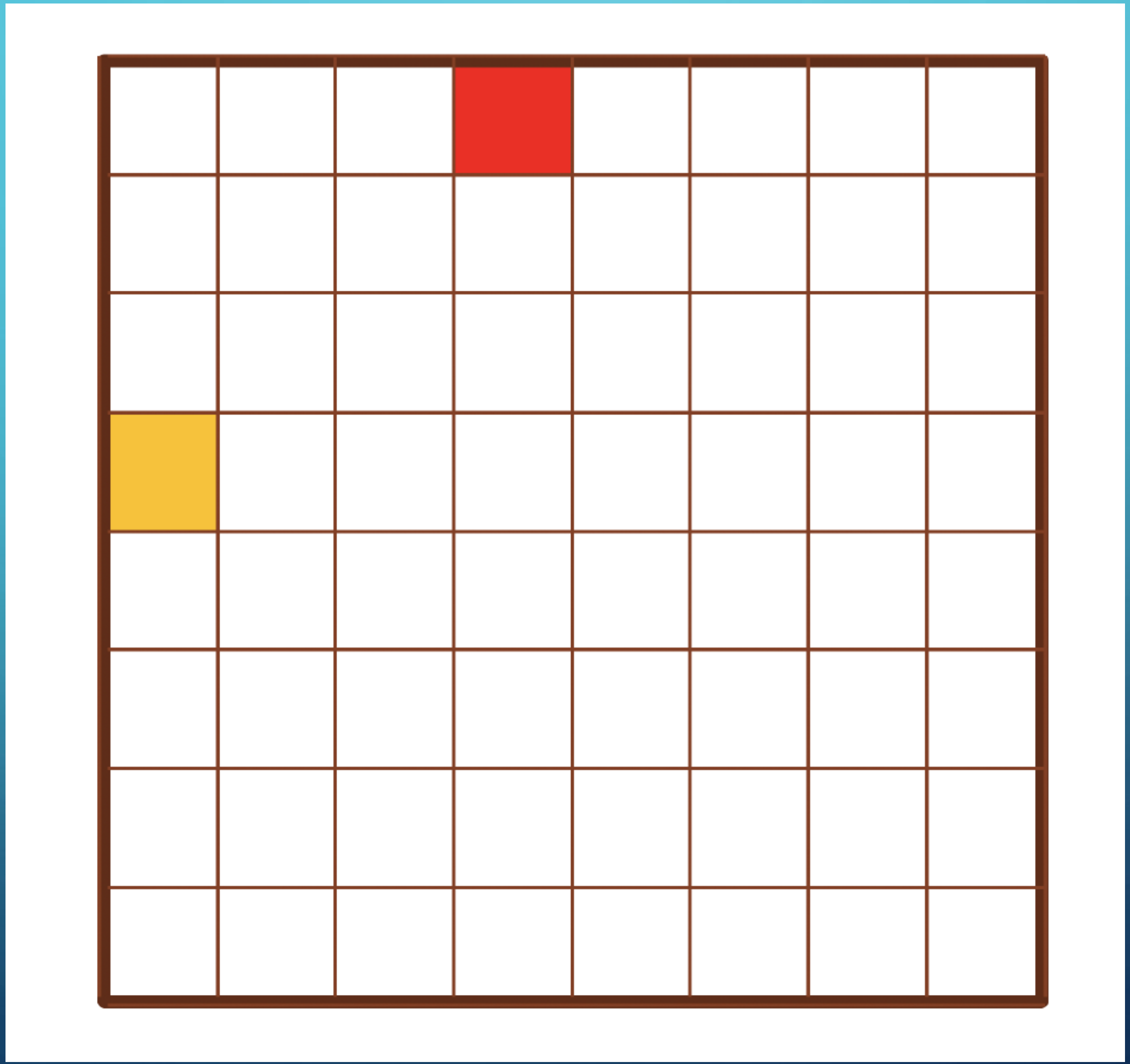
- You can declare a 2D shared memory variable statically, as follows:

```
__shared__ int tile[N][N];
```

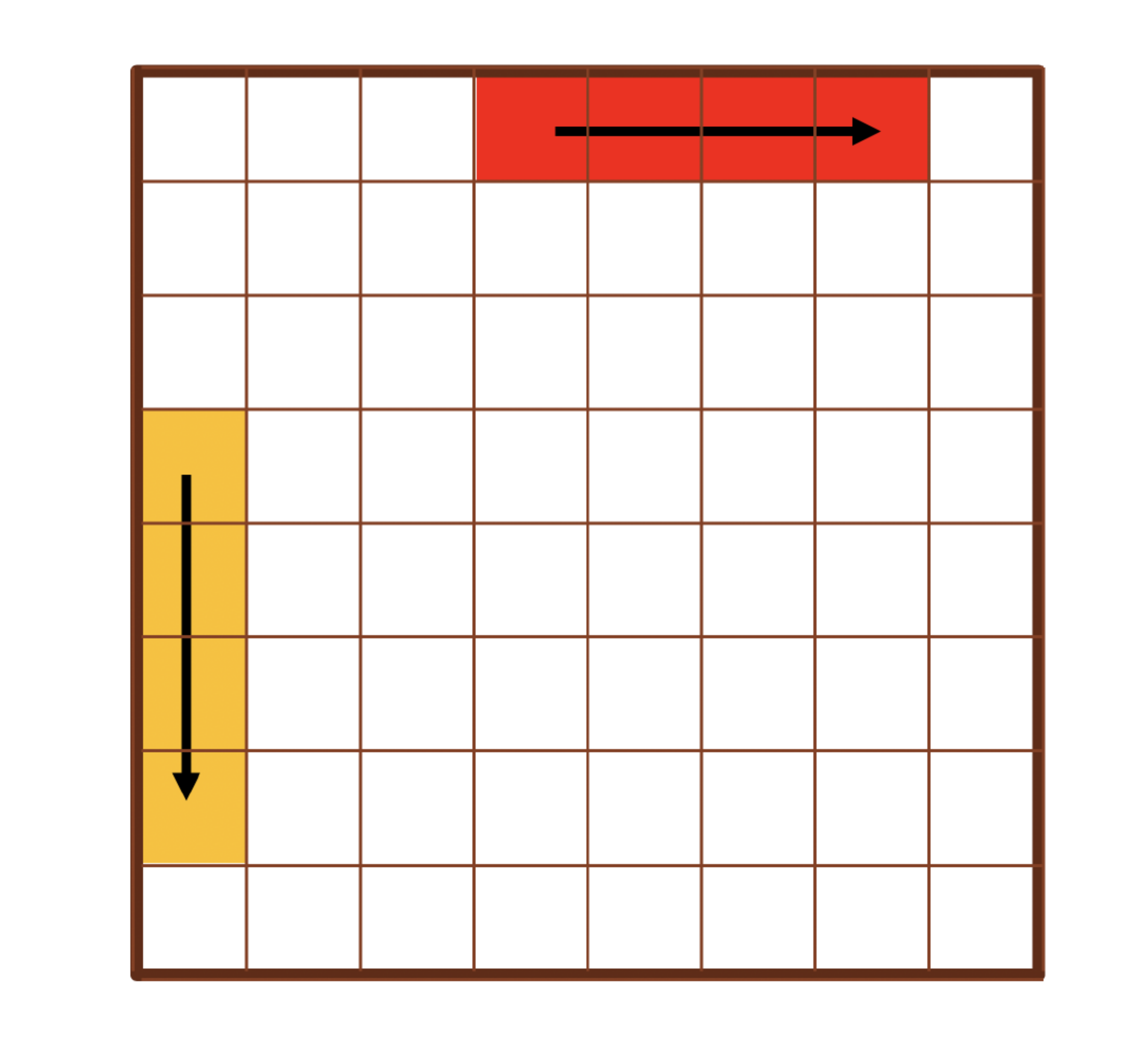
- Because this shared memory tile is square, you can choose to access it from a 2D thread block with neighboring threads accessing neighboring elements in either the x or y dimension:

```
tile[threadIdx.y][threadIdx.x]
tile[threadIdx.x][threadIdx.y]
```

MATRIX TRANSPOSE ON SHARED MEMORY



MATRIX TRANSPOSE ON SHARED MEMORY



MATRIX TRANSPOSE ON SHARED MEMORY(NAIVE)

```
__global__ void naiveGmem(float *out, float *in, const int nx, const int ny) {
   // matrix coordinate (ix,iy)
   unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
   unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;

   // transpose with boundary test
   if (ix < nx && iy < ny) {
      out[ix*ny+iy]= in[iy*nx+ix];
   }
}
```

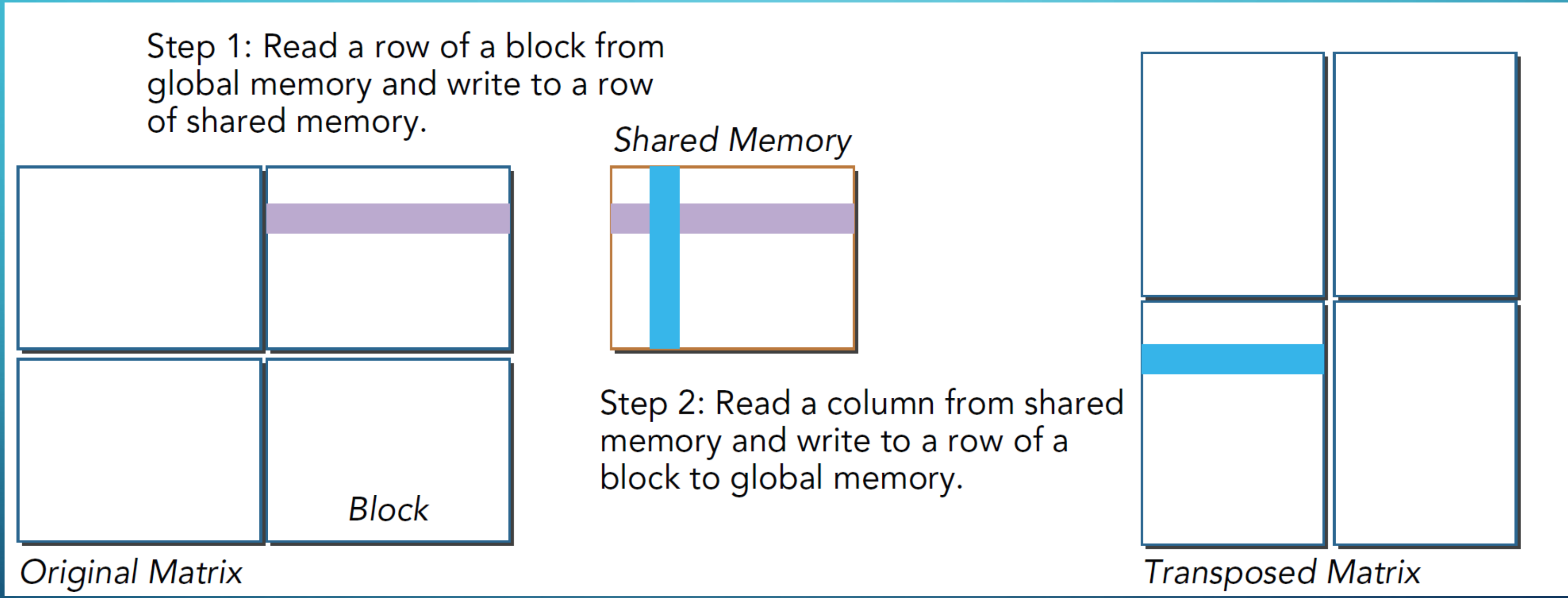
MATRIX TRANSPOSE ON SHARED MEMORY(COPYGMEM)

```
__global__ void copyGmem(float *out, float *in, const int nx, const int ny) {
   // matrix coordinate (ix,iy)
   unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
   unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;

   // transpose with boundary test
   if (ix < nx && iy < ny) {
      out[iy * nx + ix] = in[iy * nx + ix];
   }
}
```

PERFORMANCE OF TRANSPOSE KERNELS

KERNELS	TESLA M2090 (ECC OFF)		TESLA K40C (ECC OFF)	
	ELAPSED TIME (MS)	BANDWIDTH (GB/S)	ELAPSED TIME (MS)	BANDWIDTH (GB/S)
copyGmem	1.048	128.07	0.758	177.15
naiveGmem	3.611	37.19	1.947	68.98

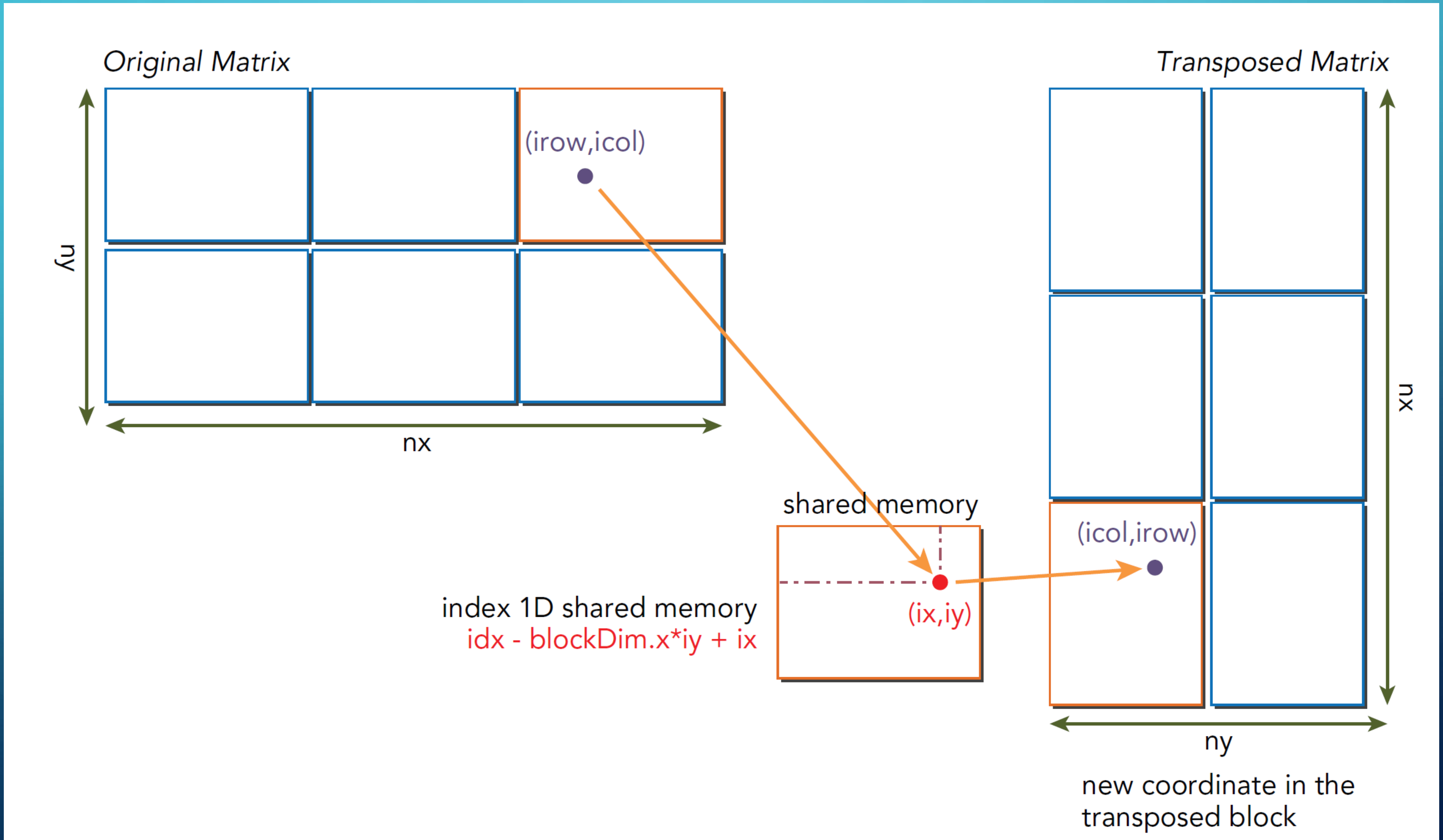
MATRIX TRANSPOSE ON SHARED MEMORY



MATRIX TRANSPOSE ON SHARED MEMORY

```
__global__ void transposeSmem(float *out, float *in, int nx, int ny) {
   // static shared memory
   __shared__ float tile[BDIMY][BDIMX];

   // coordinate in original matrix
   unsigned int ix,iy,ti,to;
   ix = blockIdx.x *blockDim.x + threadIdx.x;
   iy = blockIdx.y *blockDim.y + threadIdx.y;

   // linear global memory index for original matrix
   ti = iy*nx + ix;

   // thread index in transposed block
   unsigned int bidx,irow,icol;
   bidx = threadIdx.y*blockDim.x + threadIdx.x;
   irow = bidx/blockDim.y;
   icol = bidx%blockDim.y;

   // coordinate in transposed matrix
   ix = blockIdx.y * blockDim.y + icol;
   iy = blockIdx.x * blockDim.x  + irow;

   // linear global memory index for transposed matrix
   to = iy*ny + ix;

   // transpose with boundary test
   if (ix < nx && iy < ny)
   {
      // load data from global memory to shared memory
      tile[threadIdx.y][threadIdx.x] = in[ti];

      // thread synchronization
      __syncthreads();

      // store data to global memory from shared memory
      out[to] = tile[icol][irow];
   }
}
```

MATRIX TRANSPOSE ON SHARED MEMORY



PERFORMANCE OF TRANSPOSE KERNELS

KERNELS	TESLA M2090 (ECC OFF)		TESLA K40 (ECC OFF)	
	ELAPSED TIME (MS)	BANDWIDTH (GB/S)	ELAPSED TIME (MS)	BANDWIDTH (GB/S)
copyGmem	1.048	128.07	0.758	177.15
naiveGmem	3.611	37.19	1.947	68.98
transposeSmem	1.551	86.54	1.149	116.82
transposeSmemPad	1.416	94.79	1.102	121.83

CONSTANT MEMORY

- Constant memory is a special-purpose memory used for data that is read-only and accessed uniformly by threads in a warp.
- While constant memory is read-only from kernel codes, it is both readable and writable from the host.
- Constant variables must be declared in global scope with the following qualifier:

  __constant__

- Constant memory variables exist for the lifespan of the application and are accessible from all threads within a grid and from the host through runtime functions.

CONSTANT MEMORY

- Because the device is only able to read constant memory, values in constant memory must be initialized from host code using the following runtime function:

```
cudaError_t cudaMemcpyToSymbol(const void *symbol, const void * src,
size_t count, size_t offset, cudaMemcpyKind kind)
```

- The function `cudaMemcpyToSymbol` copies the data pointed to by `src` to the constant memory location specified by `symbol` on the device.
- The `enum` variable `kind` specifies the direction of the transfer.
- By default, `kind` is `cudaMemcpyHostToDevice`.

REFERENCES

- Professional CUDA C Programming, by John Cheng, Max Grossman, Ty McKercher, Wrox