Memory concept

Grid concept, Synchronization

GPU Programming

http://cuda.nik.uni-obuda.hu

Szénási Sándor
szenasi.sandor@nik.uni-obuda.hu

GPU Education Center of Óbuda University

NVIDIA GPU EDUCATION CENTER

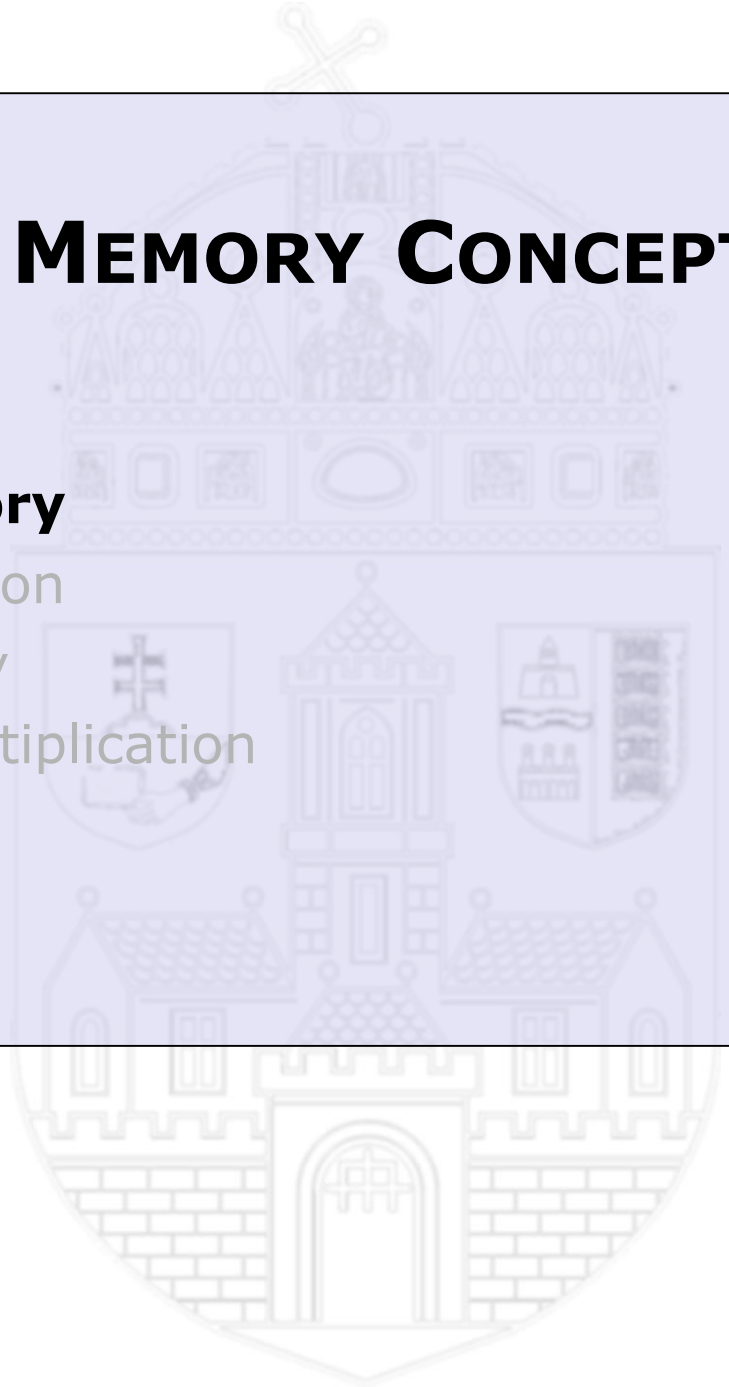
Memory Concept

Off-chip memory

Dynamic allocation

On-chip memory

Tiled matrix multiplication



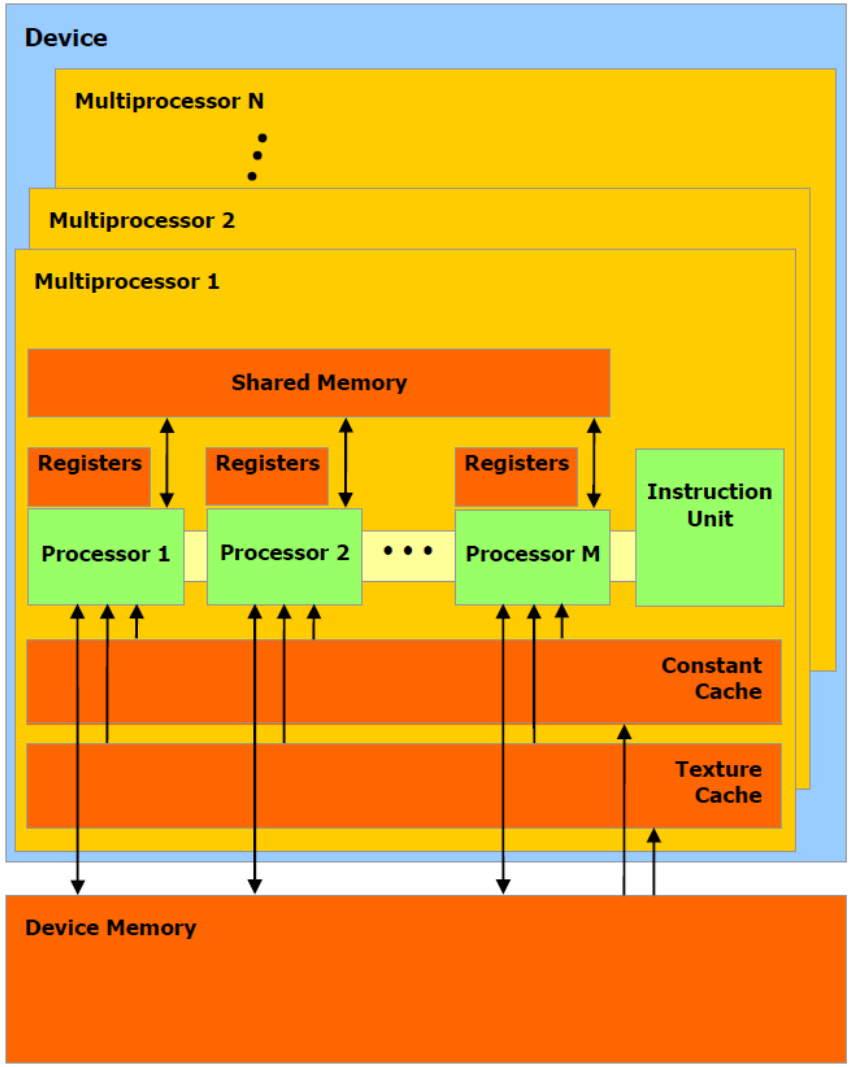
Inside one CUDA device

Memory overview

- This figure illustrates the CUDA hardware model for a device
- Every device contains one or more multiprocessors, and these multiprocessors contains one or (more frequently) more SIMT execution units

Inside one multiprocessor

- SIMT execution units
- Registers
- Shared memory (available for all threads)
- Read-only constant and texture cache



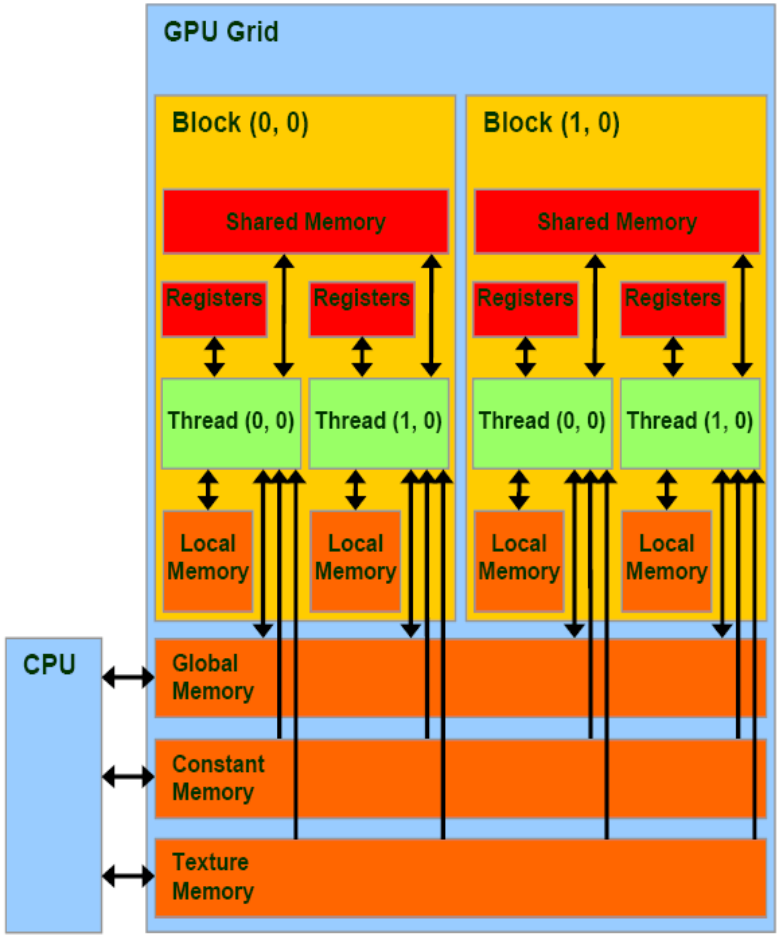
The memory concept

From the developer's perspective

- Thread level
  - Private registers (R/W)
  - Local memory (R/W)
- Block level
  - Shared memory (R/W)
  - Constant memory (R)
- Grid level
  - Global memory (R/W)
  - Texture memory (R)

Device-host communication

- The global, constant and texture memory spaces can be read from or written to by the CPU and are persistent across kernel launches by the same application



Global memory

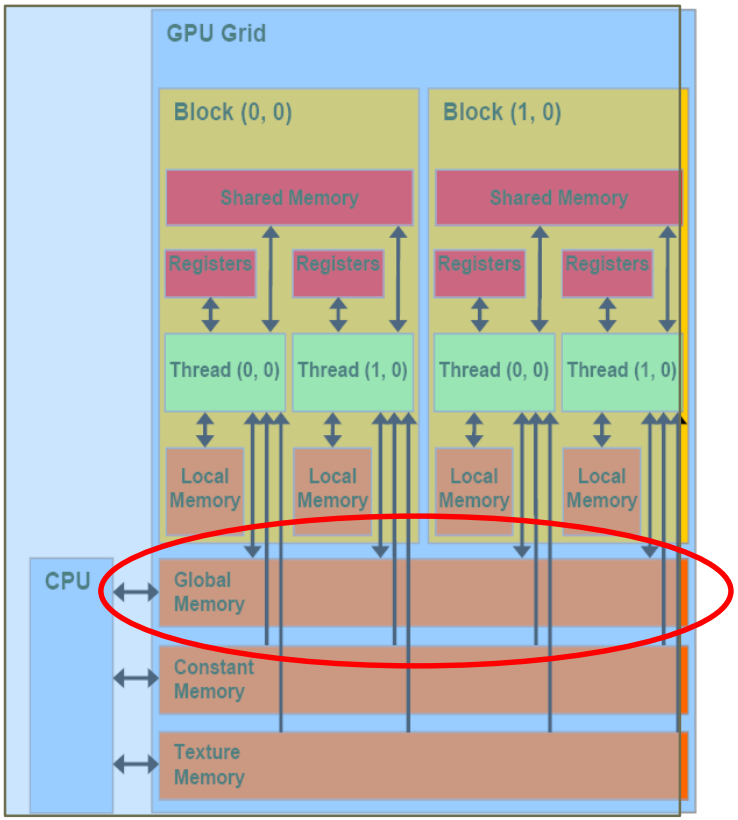
Details

- Has the lifetime of the application
- Accessible for all blocks/threads
- Accessible for the host
- Readable/writeable
- Large
- Quite slow

Static declaration

- Use the __device__ keyword
- Example

```
__device__ float *devPtr;
__device__ float devPtr[1024];
```



CUDA memory model – constant memory

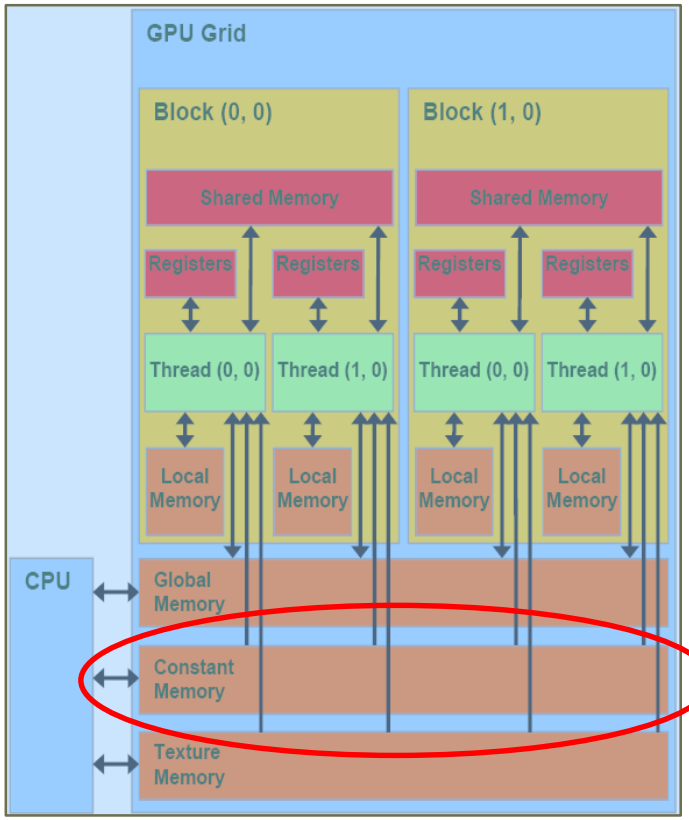
Details

- Has the lifetime of the application
- Accessible for all blocks/threads
- Accessible for the host
- Readable/writeable for the host
- Readable for the device
- Cached

Statuc declaration

- Use the __constant__ keyword
- Example:

```
__constant__ float *devPtr;
__constant__ float devPtr[1024];
```



Copy to static device variable (global or constant)

From host to device

- Function name: cudaMemcpyToSymbol
- Parameters
  - symbol - Symbol destination on device (host or device)
  - src - Source memory address
  - count - Size in bytes to copy
  - offset - Offset from start of symbol in bytes
  - kind - Type of transfer
- Copies count bytes from the memory area pointed to by src to the memory area pointed to by offset bytes from the start of symbol symbol. Symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space
- Asynchronous version: cudaMemcpyToSymbolAsync

From device to device

- The kind parameter can be
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToDevice.

Copy from static device variable (global or constant)

From device to host

- Function name: cudaMemcpyFromSymbol
- Parameters
  - dst - Destination memory address (host or device)
  - symbol - Symbol source from device
  - count - Size in bytes to copy
  - offset - Offset from start of symbol in bytes
  - kind - Type of transfer
- Copies count bytes from the memory area pointed to by offset bytes from the start of symbol symbol to the memory area pointed to by dst. Symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space
- Asynchronous version: cudaMemcpyFromSymbolAsync

From device to device

- The kind parameter can be
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToDevice.

Get pointer for static device variables

To acquire a pointer to a static variable

- Function name: cudaGetSymbolAddress
- Parameters
  - devPtr - Return device pointer associated with symbol
  - symbol - Global variable or string symbol to search for
- Returns in *devPtr the address of symbol symbol on the device. symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space.
- Example

```
__constant__ float dev_A[100];
…
float *dev_A_ptr;
cudaGetSymbolAddress((void**)&dev_A_ptr, dev_A);
```

Get the size of a static device variable

- Function name: cudaGetSymbolSize
- Parameters
  - size - Size of object associated with symbol
  - symbol - Global variable or string symbol to find size of

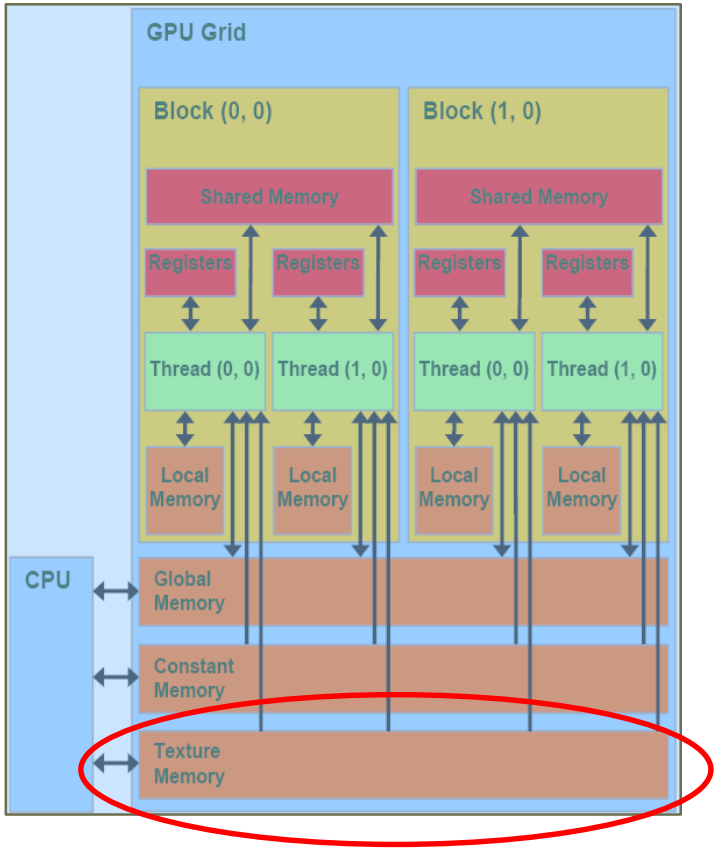
CUDA memory model – texture memory

Details

- Has the lifetime of the application
- Accessible for all blocks/threads
- Accessible for the host
- Readable/writeable for the host
- Readable for the device
- Available for image manipulating functions (texturing etc.). Not a common byte based array.

Declaration

- We do not discuss



Memory Concept

Off-chip memory

Dynamic allocation

On-chip memory

Tiled matrix multiplication

Memory handling

Static allocation

- Variabled declared as usual in C languages
- The declaration contains one of the previously introduced keywords (__device__, __constant__ etc.)
- The variable is accessible as usual in C languages, we can use them as operands and function parameters etc.

Dynamic allocation

- The CUDA class library have several memory handling functions. With these function we can
  - allocate memory
  - copy memory
  - free memory
- The memory is accessible via pointers
- Pointer usage is the same as common in C languages but it is important to note that the device has a separated address space (device and host memory pointers are exchangeable)

Dynamic allocation – allocate and free memory

Allocate device memory

- Function name: cudaMalloc
- Parameters
  - devPtr - Pointer to allocated device memory
  - size - Requested allocation size in bytes
- Allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared
- Example

```
float *dev_Ptr;
cudaMalloc((void**)&dev_Ptr, 256 * sizeof(float));
```

- Warning: host memory and device memory have separate address spaces. It's the programmers responsibility to use them in the right context

Free device memory

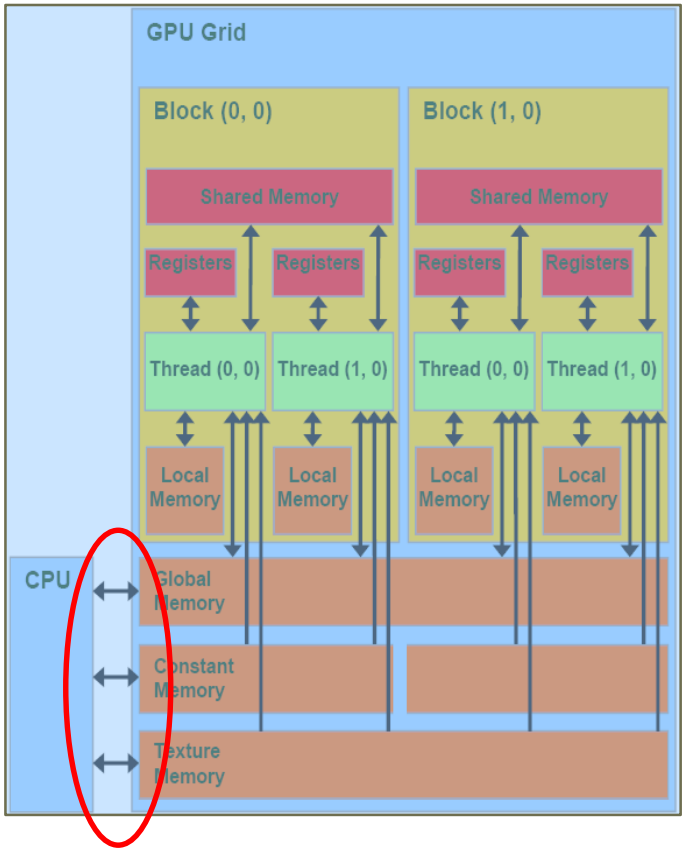
- Function name: cudaFree
- Parameters
  - devPtr - Pointer to allocated device memory
- Example

```
cudaFree(dev_Ptr);
```

Transfer in device memory

Copy between memory regions

- Function name: cudaMemcpy
- For asynchronous copy: cudaMemcpyAsync
- Parameters
  - dst - Destination memory address
  - src - Source memory address
  - count - Size in bytes to copy
  - kind - Direction of transfer
- Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst
- Valid values for direction
  - host → host (cudaMemcpyHostToHost)
  - host → device (cudaMemcpyHostToDevice)
  - device → host (cudaMemcpyDeviceToHost)
  - device → device (cudaMemcpyDeviceToDevice)




- ```
  float *hostPtr = ...;
  float *devPtr = ...;
  cudaMemcpy(devPtr, hostPtr, 256 * sizeof(float), cudaMemcpyHostToDevice);
  ```
  ```

Pinned memory

- In the host side we can allocate pinned memory. This memory object is always stored in the physical memory, therefore the GPU can fetch it without the help of the CPU
- Non-pinned memory can stored in swap (in practice in the hard drive) therefore it can cause page faults on access. So the driver needs to check every access
- To use asynchronous memory transfers the memory must be allocated by the special CUDA functions (instead of standard C malloc function):
  - cudaHostAlloc()
  - cudaFreeHost()
- It has several benefits:
  - copies between pinned memory and device memory can be performed concurrently with kernel execution for some devices
  - pinned memory can be mapped to the address space of the device on some GPUs
  - on systems with a front-side bus, bandwidth of memory transfer is higher in case of using pinned memory in the host
- Obviously the OS can not allocate as many page-locked memory as pageable. And the using of too much page-locked memory can decrease the overall system performance

Zero-copy memory

Zero copy memory

- A special version of the pinned memory is the zero-copy memory. In this case we don't need to transfer data from host to the device, the kernel can directly access the host memory
- Also called mapped memory because in this case the this memory region is mapped into the CUDA address space
- Useful when
  - the GPU has no memory and uses the system RAM
  - the host side wants to access to data while kernel is still running
  - the data does not fit into GPU memory
  - we want to execute enough calculation to hide the memory transfer latency
- Mapped memory is shared between host and device therefore the application must synchronize memory access using streams or events
- The CUDA device properties structures has information the capabilities of the GPU: canMapHostMemory = 1 if the mapping feature is available

Portable pinned memory

- Pinned memory allowed to move between host threads (in case of multi-GPU environments)

Additional memory handling functions

Get free memory

- Function name: cudaMemGetInfo
- Parameters
  - free - Returned free memory in bytes
  - total - Returned total memory in bytes
- Returns in *free and *total respectively, the free and total amount of memory available for allocation by the device in bytes

Set memory region

- Function name: cudaMemSet
- Asynchronous function: cudaMemSetAsync
- Parameters
  - devPtr - Pointer to device memory
  - value - Value to set for each byte of specified memory
  - count - Size in bytes to set
- Initializes or sets device memory to a value

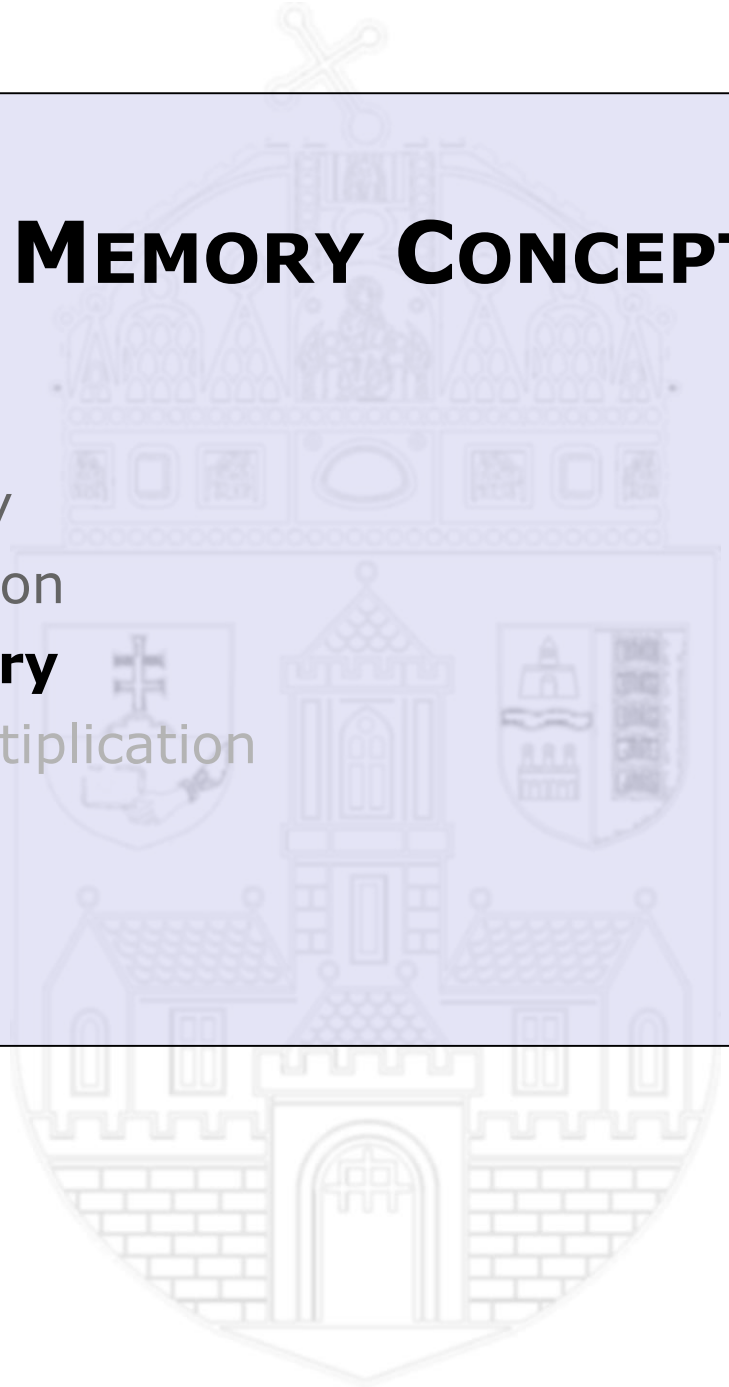
Memory Concept

Off-chip memory

Dynamic allocation

On-chip memory

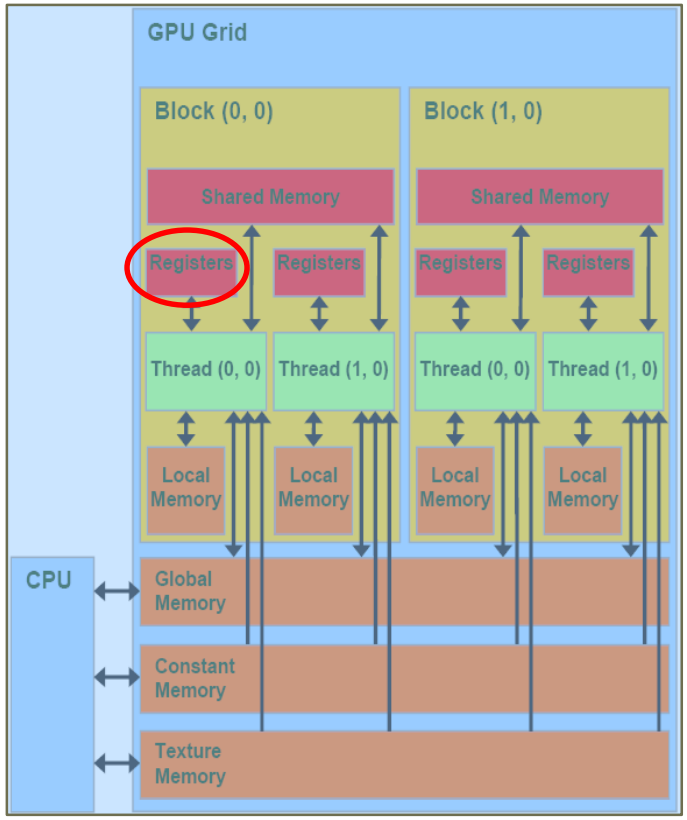
Tiled matrix multiplication



CUDA memory model - registers

Details

- Has the lifetime of the thread
- Accessible for only the owner thread
- Not accessible for the host/other threads
- Readable/writeable
- Quite fast
- Limited number of registers
- Not dedicated registers, the GPU have a fixed size register set



Declaration

- Default storing area for local variables
- Example

```
__global__ void kernel {
    int regVar;
}
```

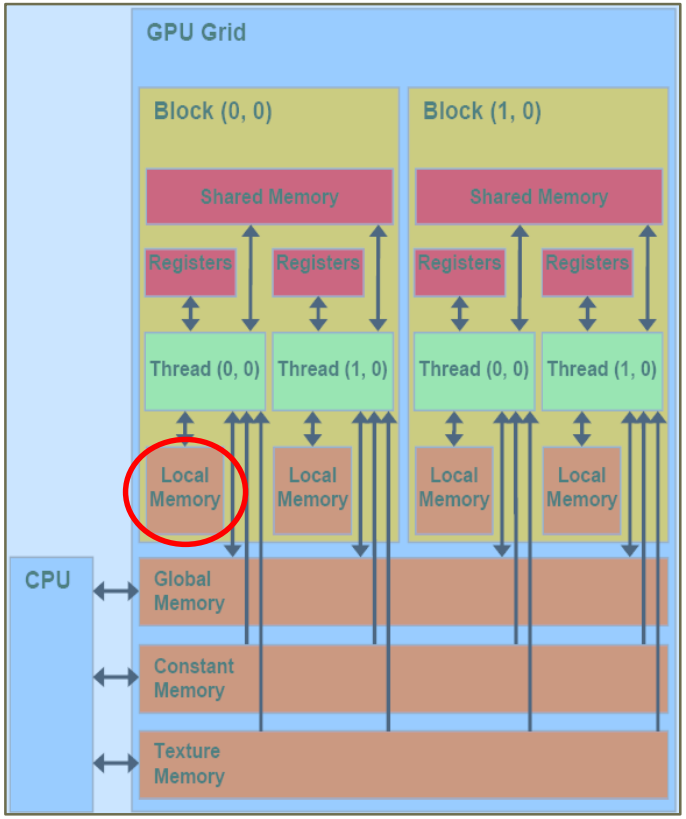
Notes

- The compiler gives information about the number of used registers
- Compiler parameters can limit the maximum number of registers

CUDA memory model – local memory

Details

- Has the lifetime of the thread
- Accessible for only the owner thread
- Not accessible for the host/other threads
- Readable/writeable
- Quite slow



Declaration

- Looks like a normal register, but these variables are stored in the „global" memory
- If there aren't any space for registers, the compiler will automatically create the variables in local memory
- Example

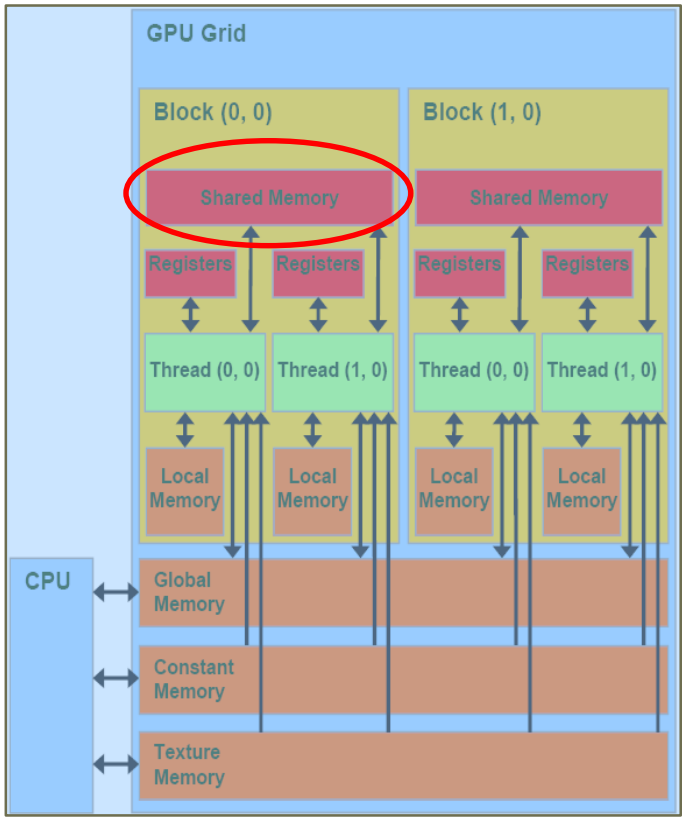
```
__global__ void kernel {
    int regVar;
}
```

Notes

- Arrays are stored in the local memory by default

Shared memory

- Has the lifetime of the block
- Accessible for all threads in this block
- Not accessible for the host
- Readable/writeable for threads
- Quite fast
- Size is strongly limited (see kernel start)



Static declaration

- Use the __shared__ keyword
- Example

```
__shared__ float a;
__global__ void kernel {
    __shared__ float devPtr[1024];
}
```

- Easy way, when the amount of required memory is already known at compile time

Shared memory dynamic allocation

Runtime allocation of shared memory

- The third kernel launch parameter is the size of the required shared memory
- Every block will have an array in the shared memory with the given size
- This array can be referenced by an unsized extern array
- Only a low level access is available in this case

```
extern __shared__ float s[];
__global__ void Kernel {
      s[threadIdx.x] = dev_A[threadIdx.x]
}
```

- The 3rd execution configuration parameter is the allocated shared memory size per thread blocks (in bytes)

```
Kernel<<<10, 20, 20 * sizeof(float)>>>( )
```

- In the case of multiple arrays, we can use the following syntax

```
extern __shared__ int s[];
int *integerData = s;
float *floatData = (float*)&integerData[nI];
char *charData = (char*)&floatData[nF];
```

```
Kernel<<<10, 20, nI*sizeof(int) + nF*sizeof(float) + nC*sizeof(char) >>>( )
```

Cache and shared memory configuration

Configurable cache control

- The on-chip memory can be partitioned between L1 cache and shared memory using 3 (≥CC2.0) or 2 (≥CC3.0) options

Set cache configuration

- Function name: cudaFuncSetCacheConfig
- Parameters
  - func - Device function symbol
  - cacheConfig - Requested cache configuration
- Possible cache configurations
  - cudaFuncCachePreferNone
    no preference for shared memory or L1 (default)
  - cudaFuncCachePreferShared
    prefer larger shared memory and smaller L1 cache
  - cudaFuncCachePreferL1
    prefer larger L1 cache and smaller shared memory
  - cudaFuncCachePreferEqual
    prefer equal size L1 cache and shared memory
- Sets through cacheConfig the preferred cache configuration for the function specified via func. This is only a preference
- Similar function to configure the device: cudaDeviceSetCacheConfig

Physical implementation of the CUDA memory model

Dedicated hardware memory

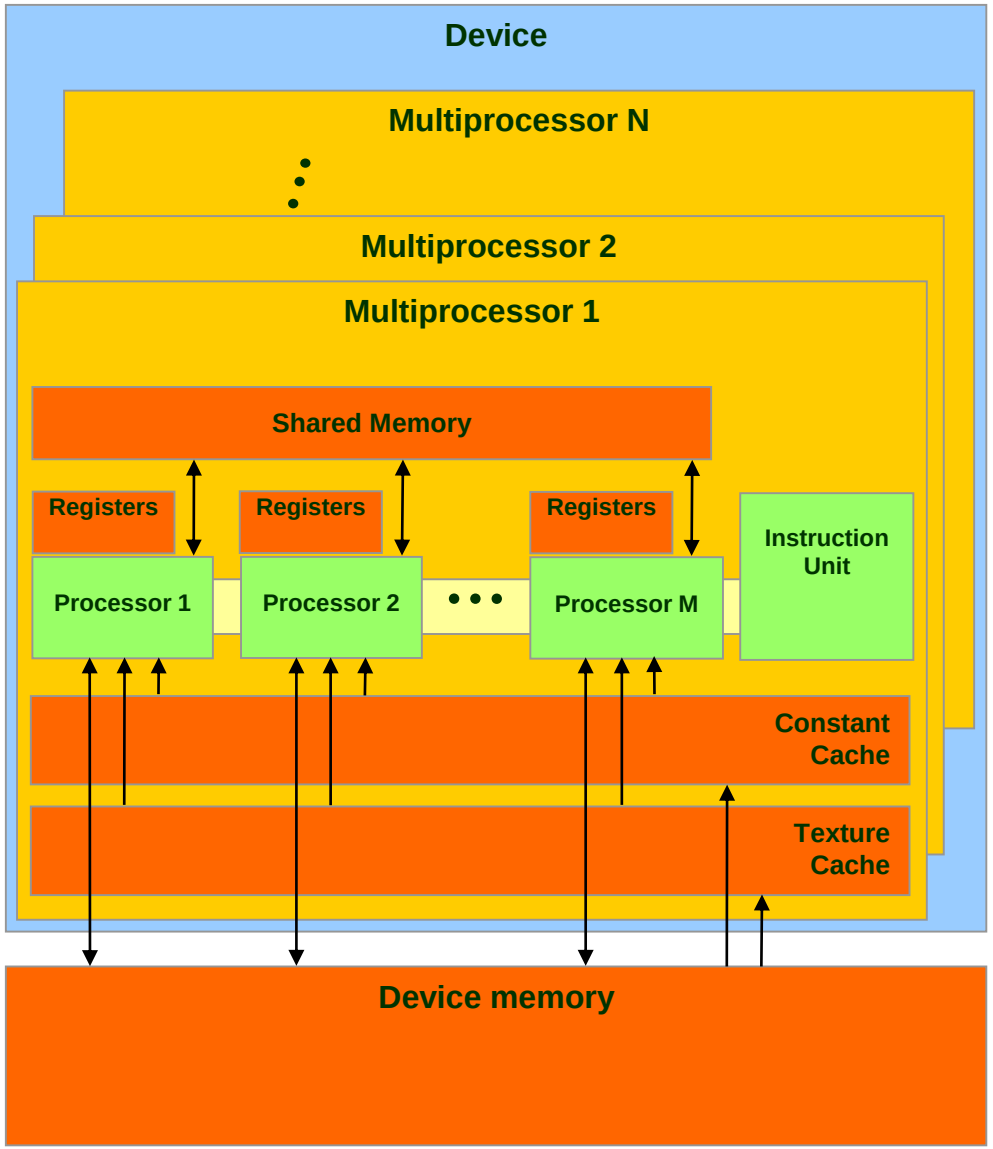
- The compiler will map here the
  - registers,
  - shared memory
- ~1 cycle

Device memory without cache

- The compiler will map here the
  - local variables,
  - global memory
- ~100 cycle

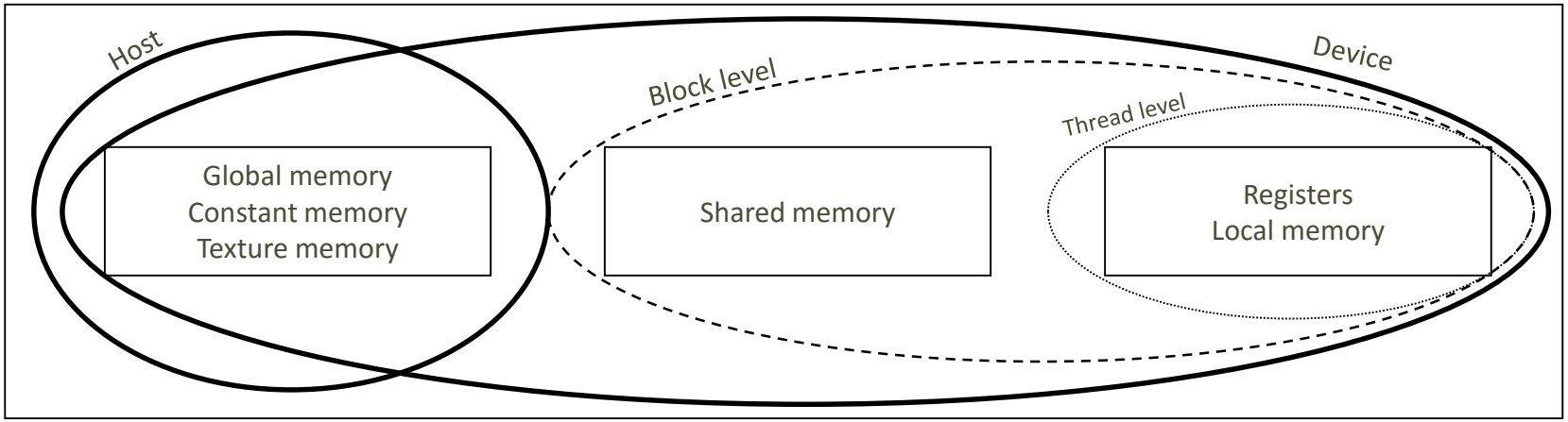
Device memory with cache

- The compiler will map here the
  - constant memory,
  - texture memory,
  - instruction cache
- ~1-10-100 cycle



CUDA memory regions

Grouped by visibility



Grouped by accessibility

	Global	Constant Texture	Shared	Registers Local memory
Host	Dynamic allocation R/W	Dynamic allocation R/W	Dynamic allocation -	- -
Eszköz	- R/W	Static allocation R	Static allocation R/W	Static allocation R/W

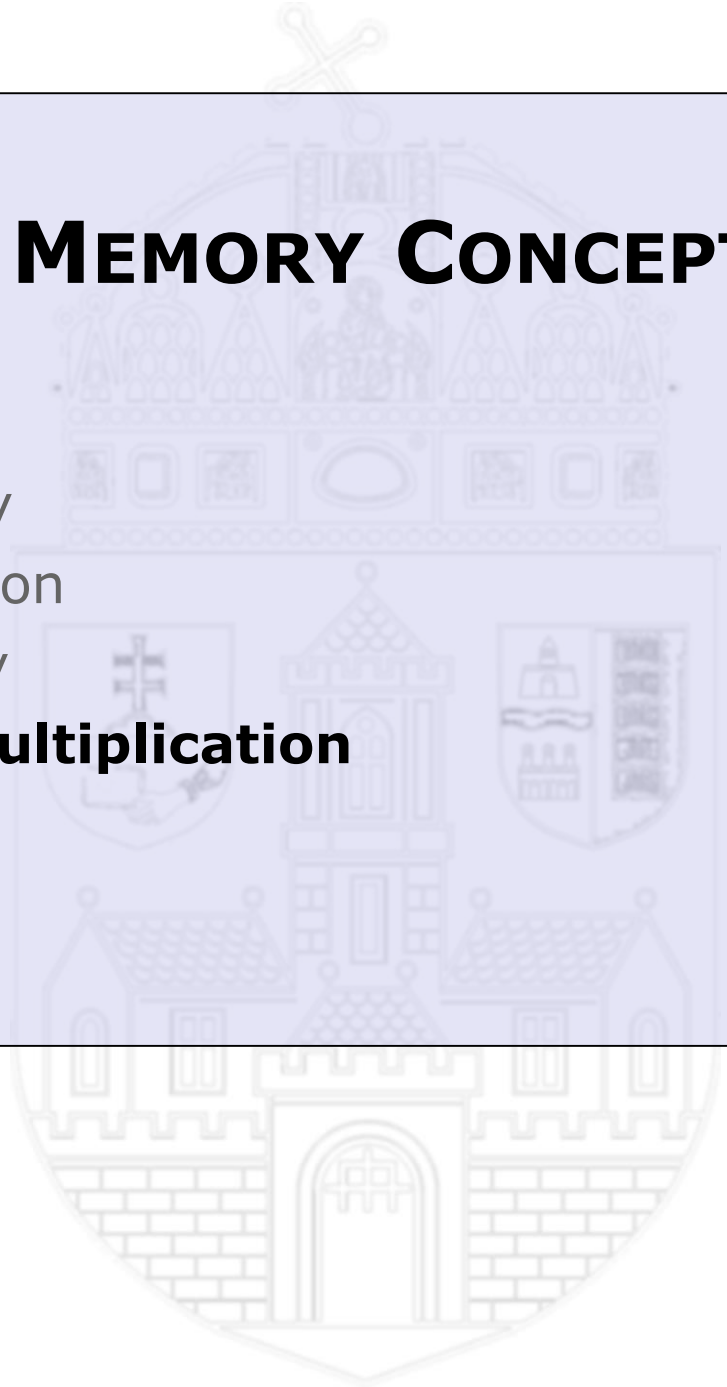
Memory Concept

Off-chip memory

Dynamic allocation

On-chip memory

Tiled matrix multiplication



Using shared memory

Matrix multiplication

- Matrix multiplication uses relatively small amount of arithmetic operations for the amount of memory transfers
- We need as many operations as possible to hide the latency caused by memory transfers (the GPU tries to schedule the execution units in case of memory latencies but without a lot of operations this is not possible)
- Out goal is to increase the ratio of arithmetic operations / memory transfers

Available solutions

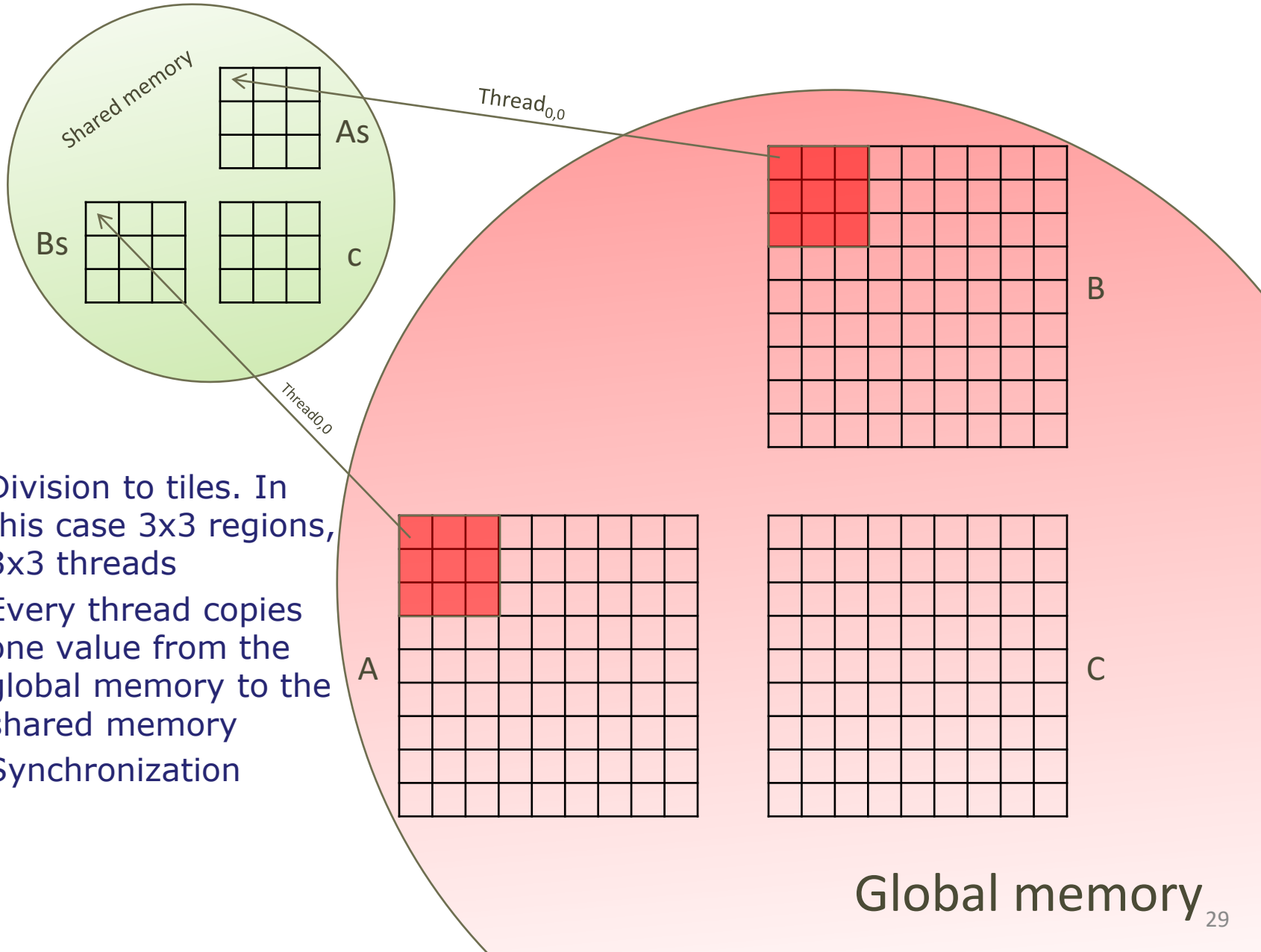
- Increase parallelism (in this case it is not possible)
- Decrease the number of memory transfers (in practice this means manually programmed caching)
  - holding as many variables in registers as possible
  - using the shared memory
- Find another solution

Tiled matrix multiplication

Tiled matrix technique

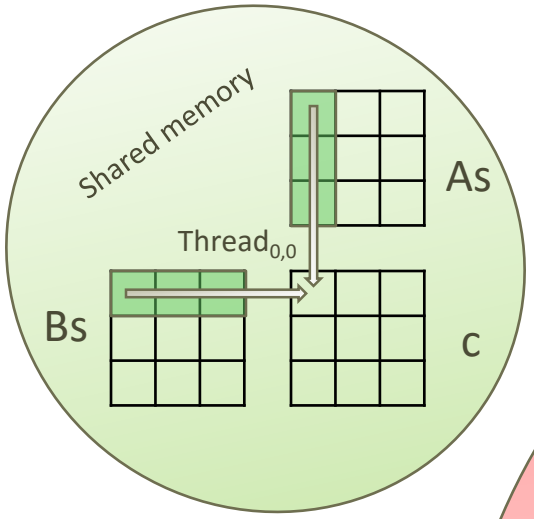
- One input cell is necessary for the calculations of more than one output cell. In the not optimized version of the algorithm, more than one thread will read the same input value from global memory
- It would be practical to harmonize these thread's work:
  - divide the entire output matrix to small regions (tiles)
  - allocate shared memory for one region
  - in the region, every thread loads the corresponding value from the input matrices to the shared memory
  - every thread calculates one partial result based on the values in the shared memory
- The size of the shared memory is limited therefore the steps above are usually executable only in more than one steps. We have to divide the input matrix to more than one tiles, and at the and of the kernel executions we have to summarize the values in these tiles
- The latter case it is necessary to synchronize the threads. Every thread must wait until all of the other threads loads the values from global memory to the shared memory, and after that the threads must wait again until all of them finished calculation before load the next value

Optimized matrix multiplication

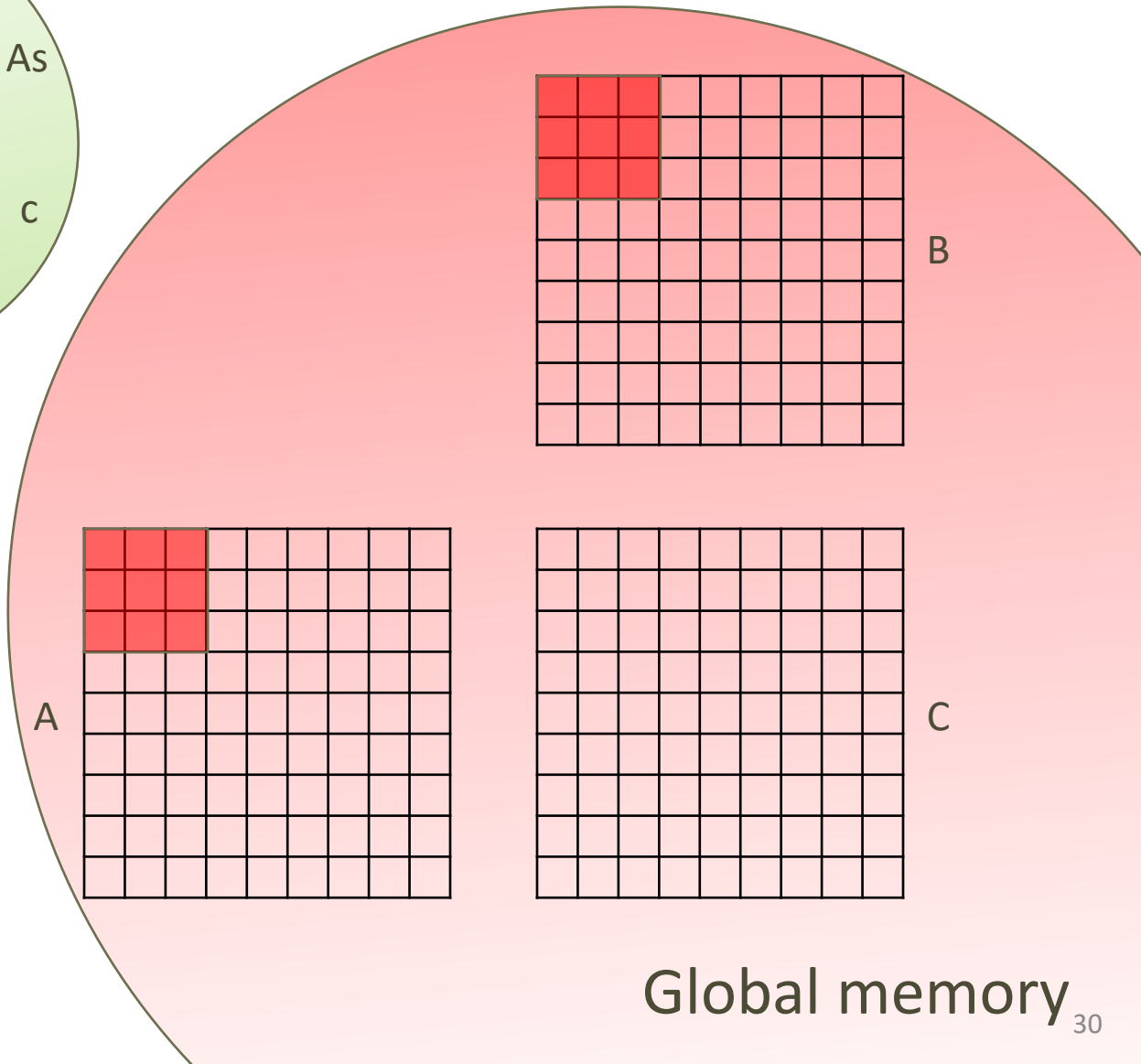


1. Division to tiles. In this case 3x3 regions, 3x3 threads
2. Every thread copies one value from the global memory to the shared memory
3. Synchronization

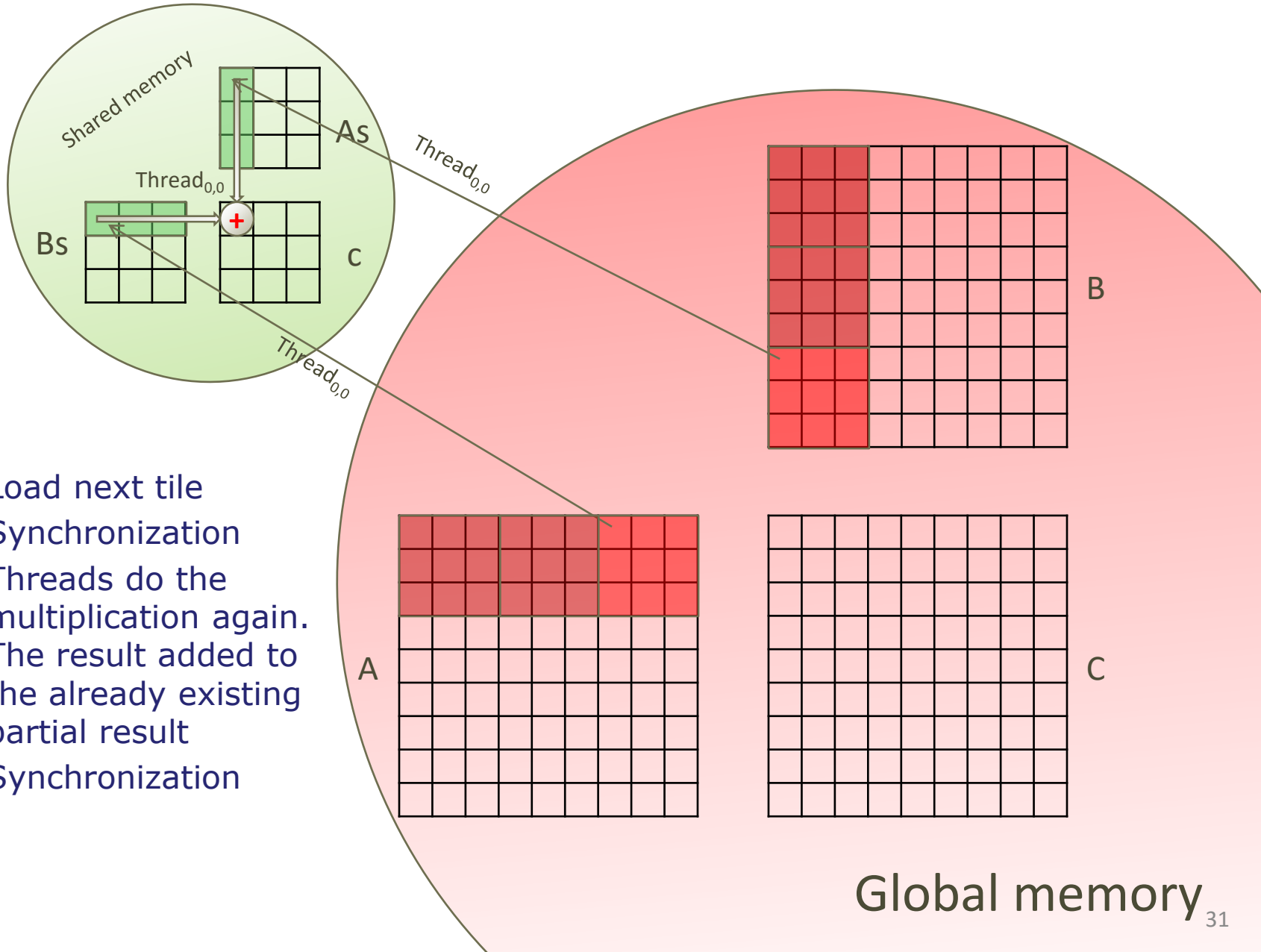
Optimized matrix multiplication (2)



4. Every thread calculated one cell's result in the shared memory
5. Synchronization

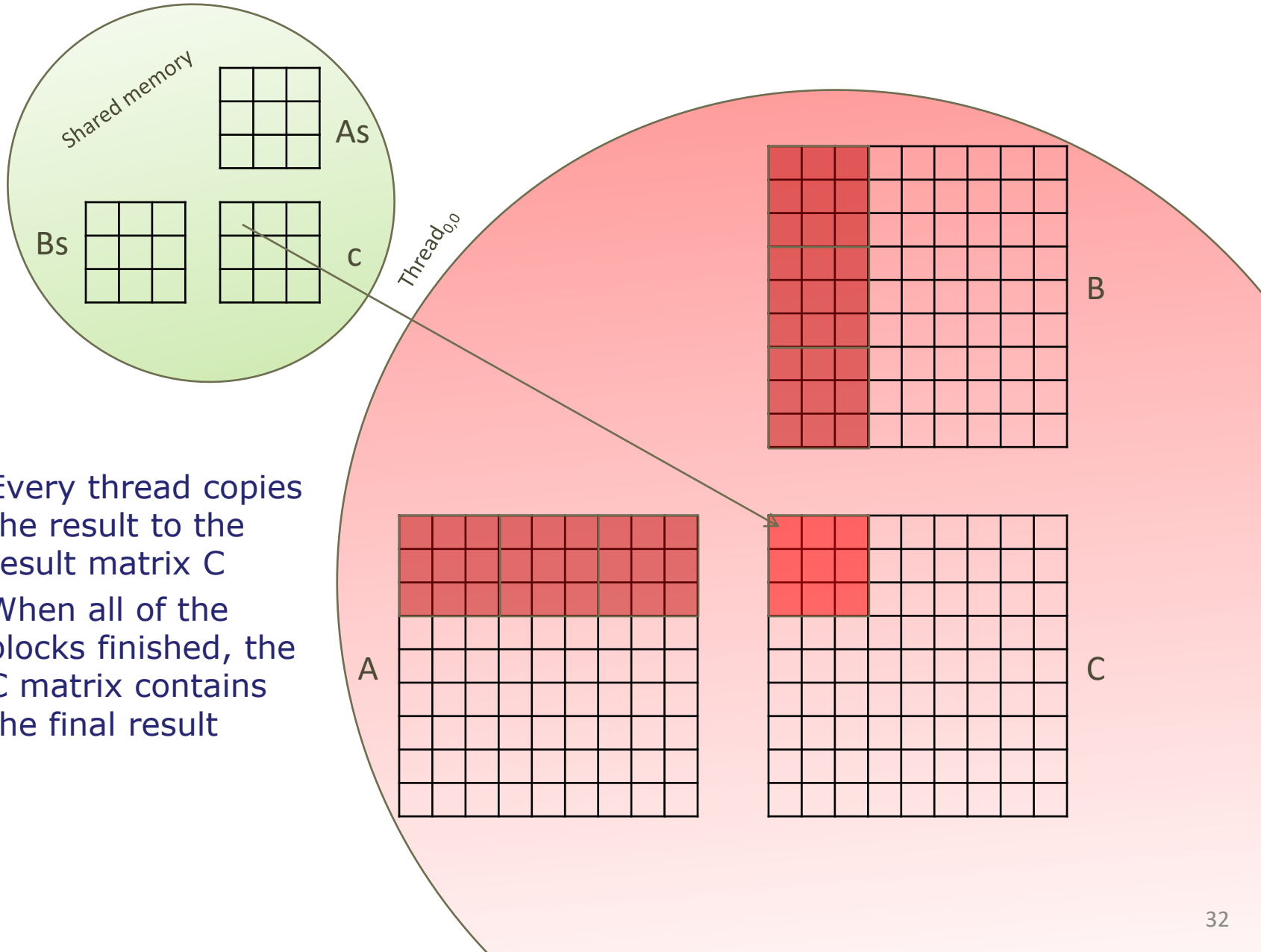


Optimized matrix multiplication (3)



6. Load next tile
7. Synchronization
8. Threads do the multiplication again. The result added to the already existing partial result
9. Synchronization

Optimized matrix multiplication (4)



10. Every thread copies the result to the result matrix C
11. When all of the blocks finished, the C matrix contains the final result

Optimized matrix multiplication source code

- Kernel invocation is the same as the not-optimized version

```
1  __global__ void MatrixMulGPUTiled(float *devA, float *devB, float *devC) {
2      __shared__ float shr_A[BLOCK_SIZE][BLOCK_SIZE];
3      __shared__ float shr_B[BLOCK_SIZE][BLOCK_SIZE];
4
5      int indx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
6      int indy = blockIdx.y * BLOCK_SIZE + threadIdx.y;
7
8      float c = 0;
9      for (int k = 0; k < N / BLOCK_SIZE; k++) {
10         shr_A[threadIdx.y][threadIdx.x] = devA[k * BLOCK_SIZE + threadIdx.x + indy * N];
11         shr_B[threadIdx.y][threadIdx.x] = devB[indx + (k* BLOCK_SIZE + threadIdx.y) * N];
12         __syncthreads();
13
14         for (int l = 0; l < BLOCK_SIZE; l++) {
15             c += shr_A[threadIdx.y][l] * shr_B[l][threadIdx.x];
16         }
17         __syncthreads();
18     }
19     devC[indx + indy * N] = c;
20 }
```

Comparing runtime of original and tiled algorithms

Runtime of algorithms

- Horizontal axis: size of matrix (N)
- Vertical axis: runtime (second)