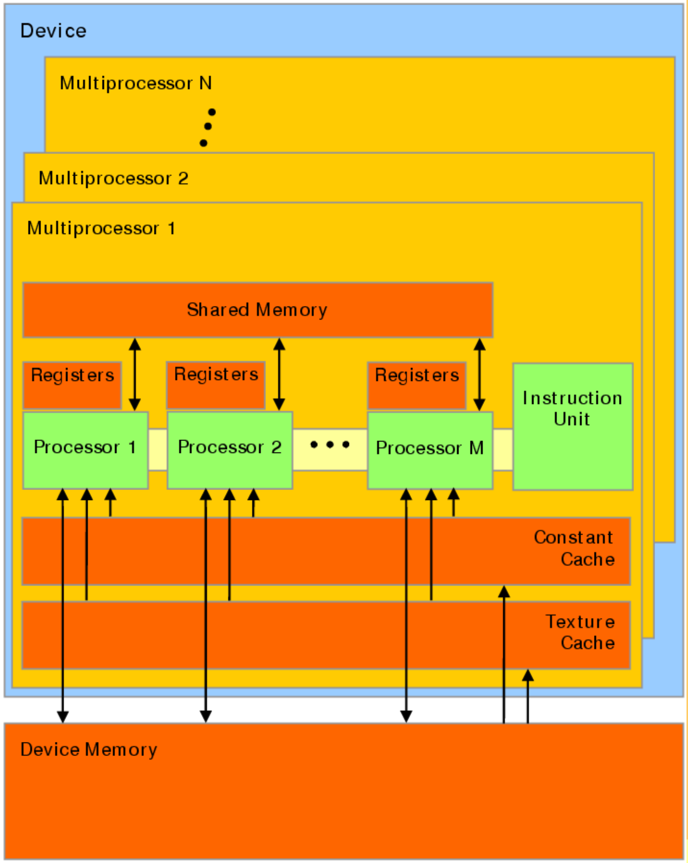
CUDA

Parallel programming on pixels

Hardware

- A GPU has many multiprocessors.
- A multiprocessor has
  - Core for computation
  - Register for computation
  - Memory for storage

An Illustration of GPU



A set of SIMT multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model

Execution

- Kernel function is run in grid.
  - <<<grid-dim, block-dim>>>
- A grid has many blocks.
  - The layout of the grid is specified by grid-dim.
  - The layout of the block is specified by block-dim.

Block and Multiprocessor

- A thread block runs on a multiprocessor as a unit.
  - No “partial” blocks
  - No migration
- A multiprocessor may run multiple blocks.
  - A multiprocessor has the resource to run many threads simultaneously.
  - The number of threads is limited by resource. If one wish to have more threads, he should get more blocks, not more threads per block.

Synchronization

- Threads with the same block can synchronize with __syncthreads().
- Kernel waits for all previous CUDA calls returns, but the control returns to host immediately.
- cudaMemcpy() starts after all previous CUDA calls returns, and return only when memory operation finishes.

Machine Configuration

- Use cudaGetDeviceCount(&device_count) to determine the number of devices in the system.
- Use cudaGetDeviceProperties(cudaDeviceProp *, int deviceid) to determine the important parameters in performance tuning.
  - Warp size
  - Maximum threads per block

cudaDeviceProp

- char name[256];
- size_t totalGlobalMem;
- size_t sharedMemPerBlock;
- int regsPerBlock;
- int warpSize;
- size_t memPitch;
- int maxThreadsPerBlock;
- int maxThreadsDim[3];
- int maxGridSize[3];

cudaDeviceProp

- size_t totalConstMem;
- int major;
- int minor;
- int clockRate;
- size_t textureAlignment;
- int deviceOverlap;
- int multiProcessorCount;

Function Qualifiers

- __global__
  - Host code invokes, runs on device.
- __device__
  - Device code invokes, runs on device.
- __host__
  - Host code invokes, runs on host.

__global__ Restrictions

- Must return void.
- No recursion
- No static variables
- No variable number of arguments

Combination

- __host__ and __device__ can be combined.
  - The same code could be run on both device and host for performance and convenience.
  - Same source but different binary codes for CPU and GPU.

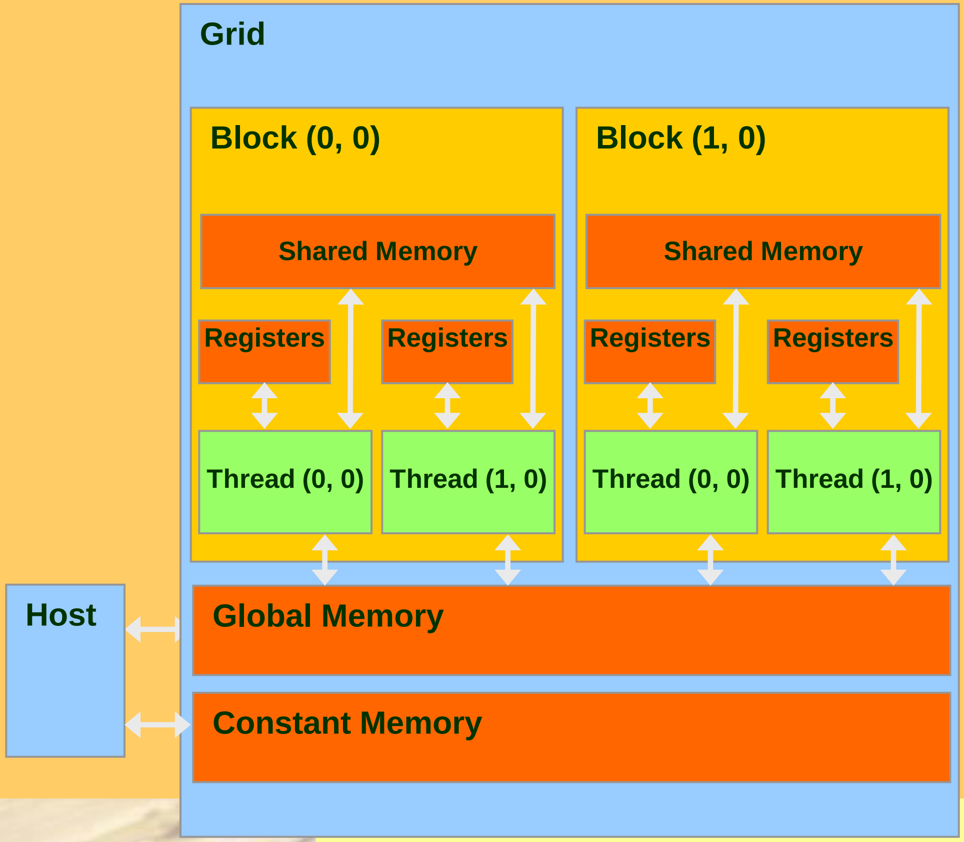
Built-in Device Variables

- All of type dims
  - With constructor, e.g. (N, N).
  - Has component x, y, and z.
- Block id and grid dimension
  - gridDim
  - blockIdx
- Thread id and block dimension
  - threadIdx
  - blockDim

Memory Hierarchy

- Global and constant memory
  - Large
  - Shared by all threads and host
  - Off-chip
- Shared memory
  - Small
  - Shared by all threads in the same block
  - On-chip

An Illustration



Implications

- A large number of threads hide instruction and memory latency for each other.
- Threads of the same block can share data via fast shared memory.
- All threads can share data via global memory.

GPU Variable Types

- Classification
  - __device__
  - __shared__
  - __constant__
  - __local__

GPU Variable Types

- Where the variables will be allocated
- The lifetime of the variable

__device__ Variables

- So-called global memory
- Large and slow
- Allocated with cudaMalloc
- Accessible by all threads and host
- Valid throughout the entire execution

__shared__ Variables

- So called shared memory
- Small and fast
- Allocated by declaration or kernel invocation
- Accessible by threads in the same block, but values assigned are guaranteed to be visible by other threads only after __syncthreads().
- Valid only during kernel execution

__shared__ by Declaration

```
__global__ void kernel(…)
{
    __shared__ float sData[256];
    …
}

int main(void)
{
    kernel<<<griddim, blockdim>>>(…);
}
```

__shared__ by Kernel Invocation

```
__global__ void kernel(…)
{
    extern shared float sData[]; /* we will have 256 floats here. */
    …
}

int main(void)
{
    kernel<<<griddim, blockdim, 256 * sizeof(float)>>>(…);
}
```

Kernel Invocation

- <<<griddim, blockdim, sharedmemsize>>>
- sharedmemsize is optional, and the default value is 0.
- If not 0 then it specifies the number of bytes of shared memory allocated for each block.
  - The allocate shared memory can be accessed as sh_data.

__constant__ Variables

- Stored in constant memory
- Read-only on device, but can be set on host.
- Accessible by all threads and host
- Valid throughout the entire execution

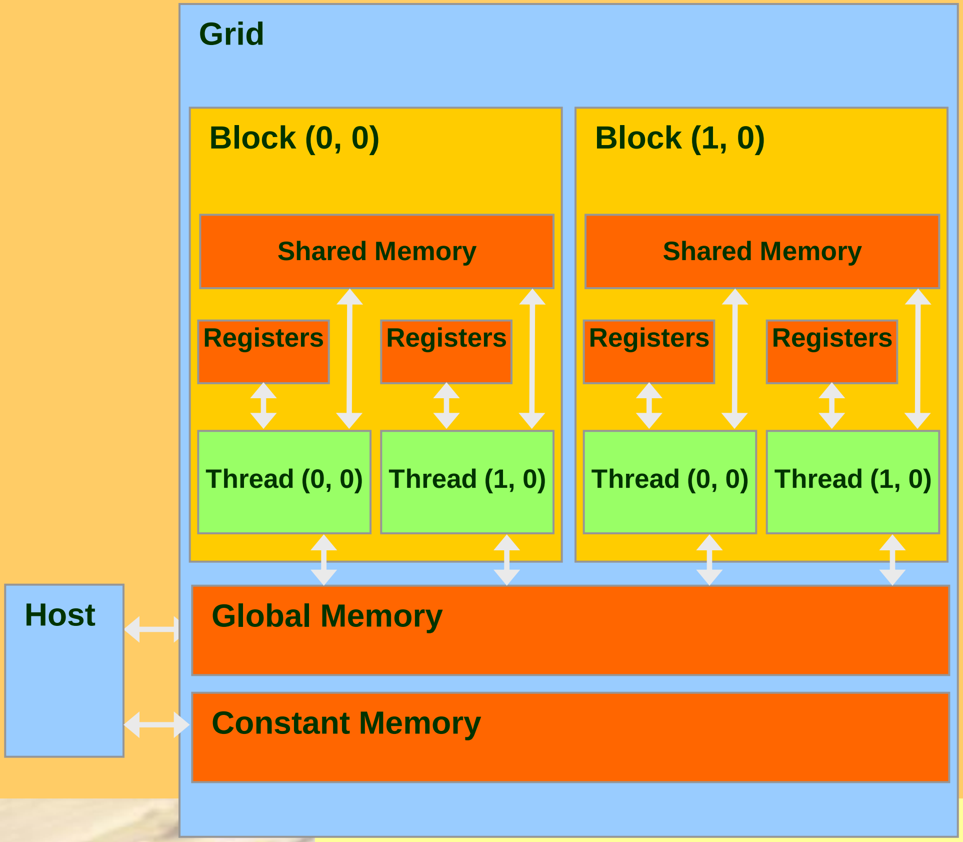
__local__ Variables

- Stored in registers if not array
- Allocated by declaration
- Accessible by the thread that declares it.
- Valid when the thread is active

Access

- Host may access
  - __global__
  - __constant__
- Kernel only
  - __local__
  - __shared__

An Illustration



Restriction

- No qualifier on
  - Members of a structure
  - Host local variables
- __shared__ cannot be initialized during declaration.

Strategy

- Shared memory is much faster than global memory
  - Host moves data into global memory.
  - Thread moves data from global memory into private memory for processing.
  - Thread computes.
  - Copy results from private to global memory.

Timing

- The clock() routine returns the wall clock time in clock ticks as a clock_t.
- The number of clock ticks per second is in the constant CLOCKS_PER_SEC.
- Before calling clock make sure all CUDA routines finish by calling cudaThreadSynchronize() at host.

Thread Synchronization

- __syncthreads()
  - Calls from device code
  - Synchronizes all threads in the same block
  - Enforces consistent view on shared memory
- cudaThreadSynchronize()
  - Calls from host code.
  - Synchronizes all threads
  - Enforces correct timing on kernel

Matrix Multiplication

- Two versions to compute C = A x B
- Both A and B are in global memory
  - Version 1 retrieves data from global memory directly.
  - Version 2 retrieves data from global memory and stores in shared memory.

Using Shared Memory

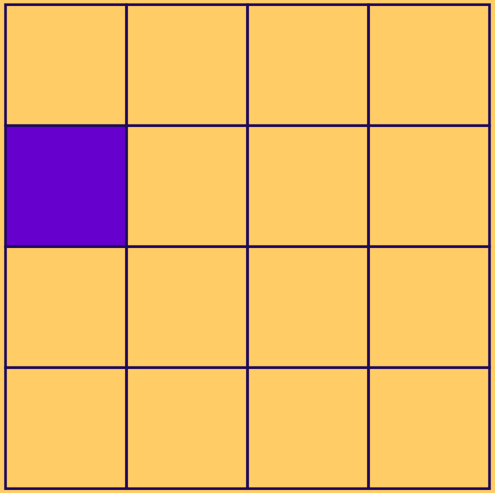
- Organize the grid as a N/b by N/b grids, and each block has b by b threads.
- Each thread block will declare two b by b shared arrays in shared memory (for A and B).
- Each thread in a block will be responsible for loading an element from A, and an element from B.

Matrix Loading

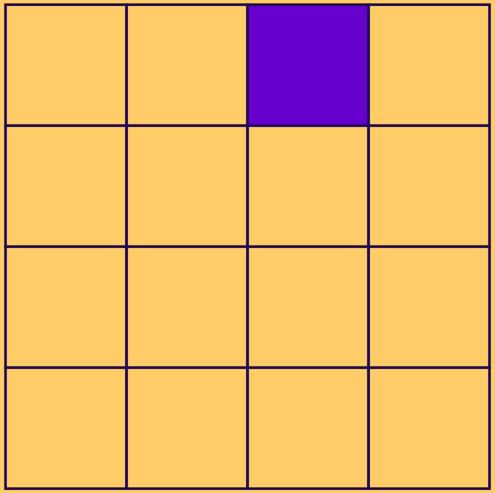
- The matrix A and B will be divided into N/b * N/b sub-matrices, each of size b by b.
- If a block is in the i-th row and j-th column of the grid, then the threads in the block will load the first sub-matrix (of size b by b) from the i-th sub-matrix row of A, and the first sub-matrix (of size b by b) from the j-th sub-matrix column of B, then the threads will load from the second sub-matrix, and so on.

An Illustration

- First step



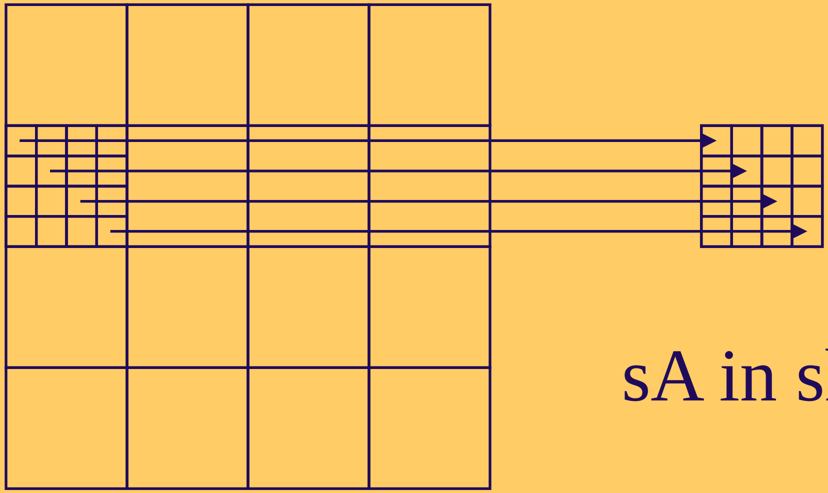
A



B

An Illustration

- Each thread moves an element in A.

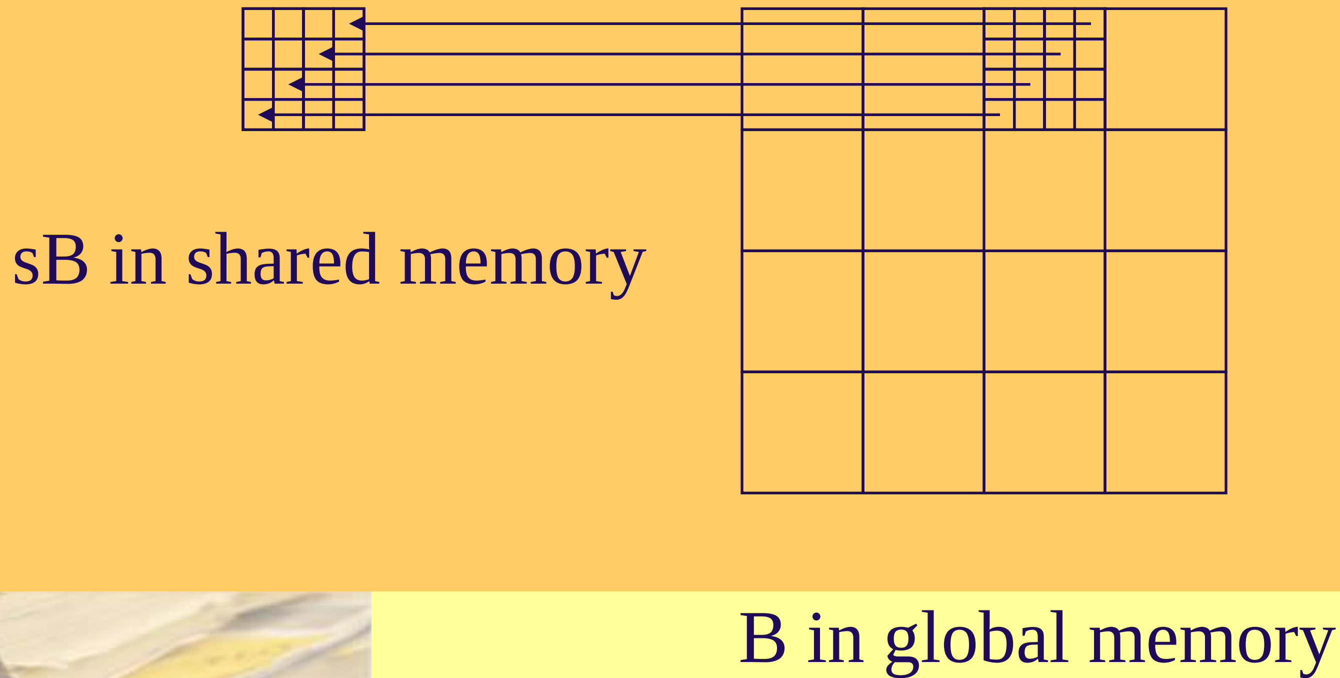


sA in shared memory

A in global memory

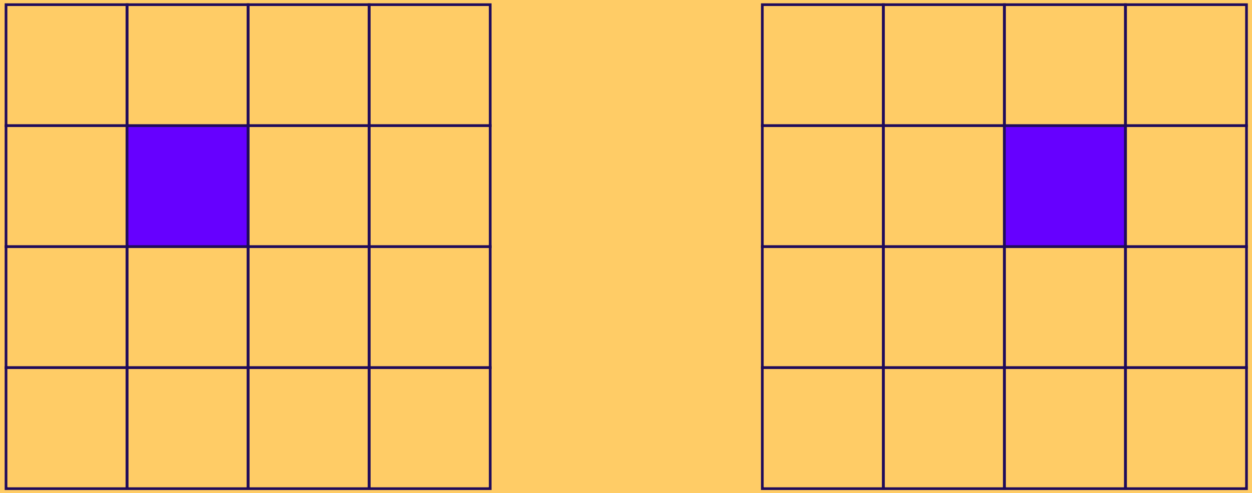
An Illustration

- Each thread moves an element in B



An Illustration

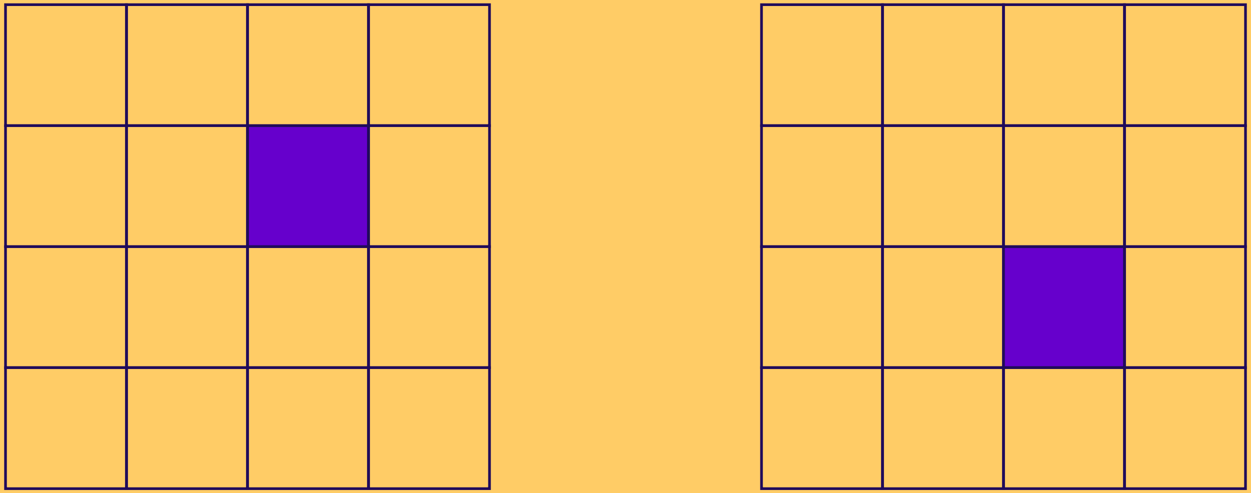
- Second step



A B

An Illustration

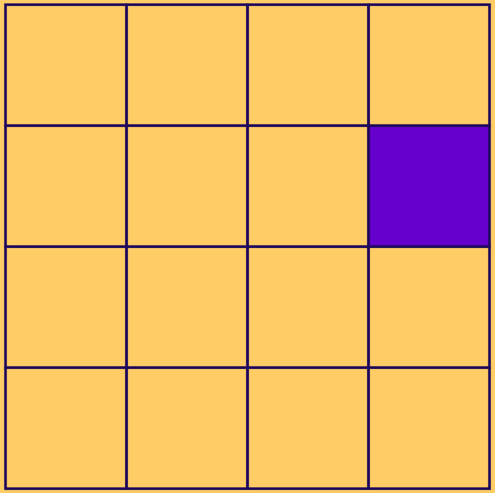
- Third step



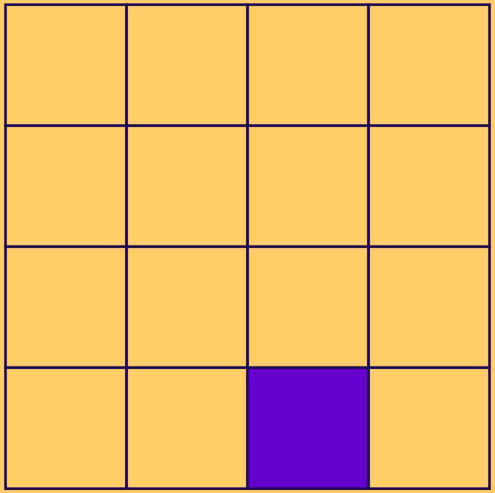
A B

An Illustration

- Final step



A



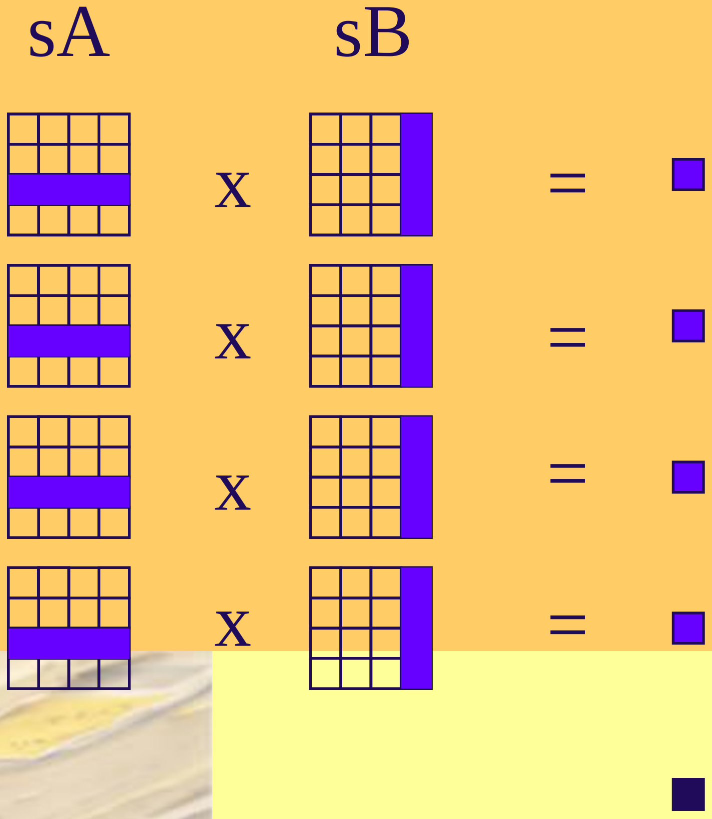
B

Inner Product

- Every time the threads load two sub-matrix (A and B), it performs an inner product it is responsible for.
  - The thread at the ith-row and j-th column of a thread block will compute the inner product of the i-th row of sub-matrix A and j-th column of sub-matrix B.
- The sum of these inner products will be the final C_{ij}

An Illustration

- Each thread computes an element in C



Comparison

- Each thread of version 1
  - Load 2N element (A, B) from global memory for inner product.
- Each thread of version 2
  - Load 2(N / b) element (A, B) from global memory.
  - Load 2N elements from shared memory (for inner product).
  - The threads “share” the loading from slow global memory.