CUDA C++ Exercise: Basic Linear Algebra Kernels: GEMM Optimization Strategies

Dmitry Lyakh

Scientific Computing

Oak Ridge Leadership Computing Facility

Oak Ridge National Laboratory

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Installing CUDA Basic Linear Algebra (BLA) Library

- Log in to BlueWaters @ NCSA:
  ssh <user_id>@bwbay.ncsa.illinois.edu

- Clone the BLA exercise repository (either way):
  git clone https://github.com/DmitryLyakh/CUDA_Tutorial.git
  or
  git clone /projects/training/bayr/CUDA_Tutorial

- If already cloned, **cd CUDA_Tutorial** and **git pull**

- Adjust Cray environment modules:
  module swap PrgEnv-cray PrgEnv-gnu
  module swap gcc/8.2.0
  module load cudatoolkit

- Copy Makefile: **/projects/training/bayr/CUDA_Tutorial/Makefile** into your CUDA_Tutorial directory
  or make sure CUDA_HOST, CUDA_INC, CUDA_LIB match in the Makefile

- Run **make**: Builds binary **bla_test.x**

- Running **bla_test.x** on BlueWaters:
  - Open interactive session **once** (for one hour): `qsub -I -l nodes=1:ppn=16:xk -l walltime=01:00:00`
  - Adjust Cray modules again as described above (**once** per interactive session)
  - **cd CUDA_Tutorial** (enter your CUDA_Tutorial path)
  - **aprun –n1 –N1 –d16 ./bla_test.x**

CUDA BLA Library Concepts: Matrix

- In file matrix.hpp: **class Matrix<T>**, **T** = {**float**, **double**}
- Matrix constructor: **Matrix(nrows, ncols)**: No storage yet!
- Matrix storage: **Matrix.allocateBody(int device)**:
  CPU Host: device = -1
  NVIDIA GPU: device = 0,1,2,… (only one GPU on BlueWaters)
  May simultaneously reside on Host and GPU: Needs sync (below)!
- Set to zero on given device: **Matrix.zeroBody(int device)**
- Set to some random value on Host: **Matrix.setBodyHost()**
- Synchronize value on multiple devices:
  Matrix.syncBody(int device, int source_device)

CUDA BLA Library Concepts: Matrix Operations

- Compute sum of the squares of all elements (on given device):
  double Matrix.computeNorm(int device)
- Add one matrix to another matrix (on given device):
  Matrix.add(Matrix & Amat, T alpha, int device)
- Multiply two matrices and add the result to another matrix:
  Matrix.multiplyAdd(bool left_transp, bool right_transp,
  Matrix & Amat, Matrix & Bmat, int device)
- Default Matrix.multiplyAdd GPU implementation expects:
  left_transp = **false**, right_transp = **false**
- **Your exercise is to implement GPU kernels for all transposition cases: FalseTrue, TrueFalse, TrueTrue**

CUDA BLA Library Implementation Benchmark

- Our test driver code: main.cpp: Function **use_bla()**
- Creates matrices **A(m,k)**, **B(k,n)**, **C(m,n)** with some **m**, **n**, **k**
- Computes the total **flop count** for matrix multiplication:
  Flop = 2*m*n*k, where factor of 2 is (multiply + add) = 2 Flop
- Executes matrix multiplication/accumulation (**C.multiplyAdd**):
  C(m,n) += A(m,k) * B(k,n)
- Function **bla::reset_gemm_algorithm(int)** chooses between:
  - 7: Highly optimized cuBLAS GEMM implementation
  - 2: Shared memory + registers based BLA GEMM (bla_lib.cu: **gpu_gemm_sh_reg_nn**)
  - 1: Shared memory based BLA GEMM (bla_lib.cu: **gpu_gemm_sh_nn**)
  - 0: Naïve BLA GEMM implementation (bla_lib.cu: **gpu_gemm_nn**)

CUDA BLA Library Implementation Benchmark

Testing your BLA GPU kernel implementation (main.cpp: use_bla() function):

```
for(int repeat = 0; repeat < 2; ++repeat){ //repeat experiment twice
 C.zeroBody(0); //set matrix C body to zero on GPU#0
 bla::reset_gemm_algorithm(0); //choose your algorithm: {0,1,2,7}
 std::cout << "Performing matrix multiplication C+=A*B with BLA GEMM brute-force ... ";
 double tms = bla::time_sys_sec(); //timer start
 C.multiplyAdd(false,false,A,B,0); //default case {false,false}: You goal is {false,true}, {true,false}, {true,true}
 double tmf = bla::time_sys_sec(); //timer stop
 std::cout << "Done: Time = " << tmf-tms << " s: Gflop/s = " << flops/(tmf-tms)/1e9 << std::endl;
 //Check correctness on GPU#0:
 C.add(D,1.0f,0); //adding the correct result with a minus sign (matrix D) should give you zero matrix
 auto norm_diff = C.computeNorm(0); //check its norm
 std::cout << "Norm of the matrix C deviation from correct = " << norm_diff << std::endl;
 if(std::abs(norm_diff) > 1e-7){ //report if norm is not zero enough
  std::cout << "#FATAL: Matrix C is incorrect, fix your GPU kernel implementation!" << std::endl;
  std::exit(1);
 }
}
```

This benchmark is run for all available BLA GEMM algorithms: 0, 1, 2, 7 for the {false,false} case. Your goal is to implement and run other cases: {false,true}, {true,false}, {true,true}!

CUDA BLA Library: GEMM cases

- **Matrix A(m,n)** employs column-wise storage (standard BLAS):
  - **A(0,0), A(1,0), A(2,0), …, A(m-1,0), A(0,1), A(1,1), A(2,1), …, A(m-1,n-1)**
  - **Linear offset** of element (j,k) in storage is **L(j,k) = (j + k*m)**
  - **m** is the leading dimension extent in this case
- Matrix multiplication {**false**,**false**} case (implemented):
  - **C(m,n) += A(m,k) * B(k,n)**
- Matrix multiplication {**false**,**true**} case (your exercise):
  - **C(m,n) += A(m,k) * B(n,k)**
- Matrix multiplication {**true**,**false**} case (your exercise):
  - **C(m,n) += A(k,m) * B(k,n)**
- Matrix multiplication {**true**,**true**} case (your exercise):
  - **C(m,n) += A(k,m) * B(n,k)**

CUDA BLA Library: GEMM algorithms

- You will work inside **bla_lib.cu** source file directly with CUDA GEMM kernels
- Matrix multiplication {**false**,**false**} case (implemented):
  - **C(m,n) += A(m,k) * B(k,n)**
  - **CUDA kernels: gpu_gemm_nn, gpu_gemm_sh_nn, gpu_gemm_sh_reg_nn**
- Matrix multiplication {**false**,**true**} case (your exercise):
  - **C(m,n) += A(m,k) * B(n,k)**
  - **CUDA kernels: gpu_gemm_nt, gpu_gemm_sh_nt, gpu_gemm_sh_reg_nt**
- Matrix multiplication {**true**,**false**} case (your exercise):
  - **C(m,n) += A(k,m) * B(k,n)**
  - **CUDA kernels: gpu_gemm_tn, gpu_gemm_sh_tn, gpu_gemm_sh_reg_tn**
- Matrix multiplication {**true**,**true**} case (your exercise):
  - **C(m,n) += A(k,m) * B(n,k)**
  - **CUDA kernels: gpu_gemm_tt, gpu_gemm_sh_tt, gpu_gemm_sh_reg_tt**

CUDA BLA Library: Matrix Addition

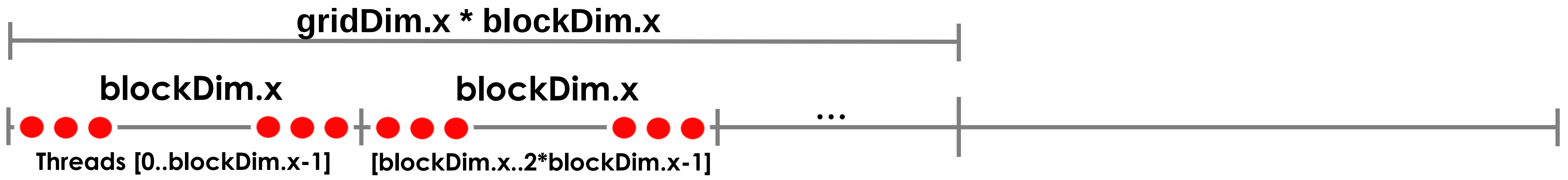
float or double

No pointer aliasing

```
template <typename T>
__global__ void gpu_array_add(size_t arr_size,   //in: array size
                              T * __restrict__ arr0,          //inout: pointer to arr0[arr_size]
                              const T * __restrict__ arr1,    //in: pointer to arr1[arr_size]
                              T alpha)                        //in: scaling factor
{
 size_t n = gridDim.x * blockDim.x;
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) arr0[i] += arr1[i] * alpha;
 return;
}
```

CUDA BLA Library: Matrix Addition

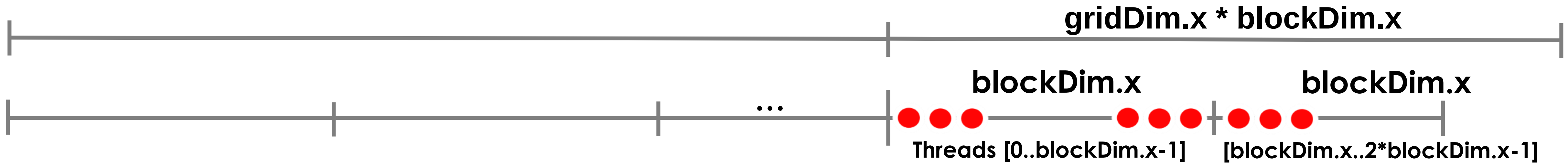
```
template <typename T>
__global__ void gpu_array_add(size_t arr_size,   //in: array size
                             T * __restrict__ arr0,          //inout: pointer to arr0[arr_size]
                             const T * __restrict__ arr1,    //in: pointer to arr1[arr_size]
                             T alpha)                        //in: scaling factor
{
 size_t n = gridDim.x * blockDim.x; //total number of threads in the kernel
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) arr0[i] += arr1[i] * alpha;
 return;
}
```



Array elements: [0..N-1]

CUDA BLA Library: Matrix Addition

```
template <typename T>
__global__ void gpu_array_add(size_t arr_size,   //in: array size
                        T * __restrict__ arr0,          //inout: pointer to arr0[arr_size]
                        const T * __restrict__ arr1,    //in: pointer to arr1[arr_size]
                        T alpha)                        //in: scaling factor
{
 size_t n = gridDim.x * blockDim.x; //total number of threads in the kernel
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) arr0[i] += arr1[i] * alpha;
 return;
}
```



Array elements: [0..N-1]

CUDA BLA Library: Matrix “Norm” (sum of squares)

float or double

```
template <typename T>
__global__ void gpu_array_norm2(size_t arr_size,        //in: array size
                                const T * __restrict__ arr,        //in: pointer to arr[arr_size]
                                volatile T * norm)        //inout: sum of the squared elements of the array
{
 extern __shared__ double thread_norm[]; //dynamic shared memory of size blockDim.x

 size_t n = gridDim.x * blockDim.x;
 double tnorm = 0.0;
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) tnorm += arr[i] * arr[i];
 thread_norm[threadIdx.x] = tnorm;
 __syncthreads();

 unsigned int s = blockDim.x;
 while(s > 1){
  unsigned int j = (s+1U)>>1; //=(s+1)/2
  if(threadIdx.x + j < s) thread_norm[threadIdx.x] += thread_norm[threadIdx.x+j];
  __syncthreads();
  s = j;
 }

 if(threadIdx.x == 0){
  unsigned int j = 1;
  while(j){j = atomicMax(&norm_wr_lock,1);} //lock
  __threadfence();
  *norm += thread_norm[0]; //accumulate
  __threadfence();
  j=atomicExch(&norm_wr_lock,0); //unlock
 }
 __syncthreads();
 return;
}
```

CUDA BLA Library: Matrix “Norm” (sum of squares)

```
template <typename T>
__global__ void gpu_array_norm2(size_t arr_size,        //in: array size
                                const T * __restrict__ arr,      //in: pointer to arr[arr_size]
                                volatile T * norm)        //inout: sum of the squared elements of the array
{
 extern __shared__ double thread_norm[]; //dynamic shared memory of size blockDim.x

 size_t n = gridDim.x * blockDim.x;
 double tnorm = 0.0;
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) tnorm += arr[i] * arr[i];
 thread_norm[threadIdx.x] = tnorm;
 __syncthreads();

 unsigned int s = blockDim.x;
 while(s > 1){
  unsigned int j = (s+1U)>>1; //=(s+1)/2
  if(threadIdx.x + j < s) thread_norm[threadIdx.x] += thread_norm[threadIdx.x+j];
  __syncthreads();
  s = j;
 }

 if(threadIdx.x == 0){
  unsigned int j = 1;
  while(j){j = atomicMax(&norm_wr_lock,1);} //lock
  __threadfence();
  *norm += thread_norm[0]; //accumulate
  __threadfence();
  j=atomicExch(&norm_wr_lock,0); //unlock
 }
 __syncthreads();
 return;
}
```

Each thread computes its contribution over the entire subrange and stores its contribution in shared memory array (per block)

CUDA BLA Library: Matrix “Norm” (sum of squares)

```
template <typename T>
__global__ void gpu_array_norm2(size_t arr_size,          //in: array size
                               const T * __restrict__ arr,  //in: pointer to arr[arr_size]
                               volatile T * norm)            //inout: sum of the squared elements of the array
{
 extern __shared__ double thread_norm[]; //dynamic shared memory of size blockDim.x

 size_t n = gridDim.x * blockDim.x;
 double tnorm = 0.0;
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) tnorm += arr[i] * arr[i];
 thread_norm[threadIdx.x] = tnorm;
 __syncthreads(); Sync threads within a block

 unsigned int s = blockDim.x;
 while(s > 1){
  unsigned int j = (s+1U)>>1; //=(s+1)/2
  if(threadIdx.x + j < s) thread_norm[threadIdx.x] += thread_norm[threadIdx.x+j];
  __syncthreads();
  s = j;
 }

 if(threadIdx.x == 0){
  unsigned int j = 1;
  while(j){j = atomicMax(&norm_wr_lock,1);} //lock
  __threadfence();
  *norm += thread_norm[0]; //accumulate
  __threadfence();
  j=atomicExch(&norm_wr_lock,0); //unlock
 }
 __syncthreads();
 return;
}
```

CUDA BLA Library: Matrix “Norm” (sum of squares)

```
template <typename T>
__global__ void gpu_array_norm2(size_t arr_size,        //in: array size
                                const T * __restrict__ arr,          //in: pointer to arr[arr_size]
                                volatile T * norm)                  //inout: sum of the squared elements of the array
{
 extern __shared__ double thread_norm[]; //dynamic shared memory of size blockDim.x

 size_t n = gridDim.x * blockDim.x;
 double tnorm = 0.0;
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) tnorm += arr[i] * arr[i];
 thread_norm[threadIdx.x] = tnorm;
 __syncthreads();

 unsigned int s = blockDim.x;
 while(s > 1){
  unsigned int j = (s+1U)>>1; //=(s+1)/2
  if(threadIdx.x + j < s) thread_norm[threadIdx.x] += thread_norm[threadIdx.x+j];
  __syncthreads();
  s = j;
 }

 if(threadIdx.x == 0){
  unsigned int j = 1;
  while(j){j = atomicMax(&norm_wr_lock,1);} //lock
  __threadfence();
  *norm += thread_norm[0]; //accumulate
  __threadfence();
  j=atomicExch(&norm_wr_lock,0); //unlock
 }
 __syncthreads();
 return;
}
```

Threads within a thread block perform reduction into thread_norm[0]

CUDA BLA Library: Matrix "Norm" (sum of squares)

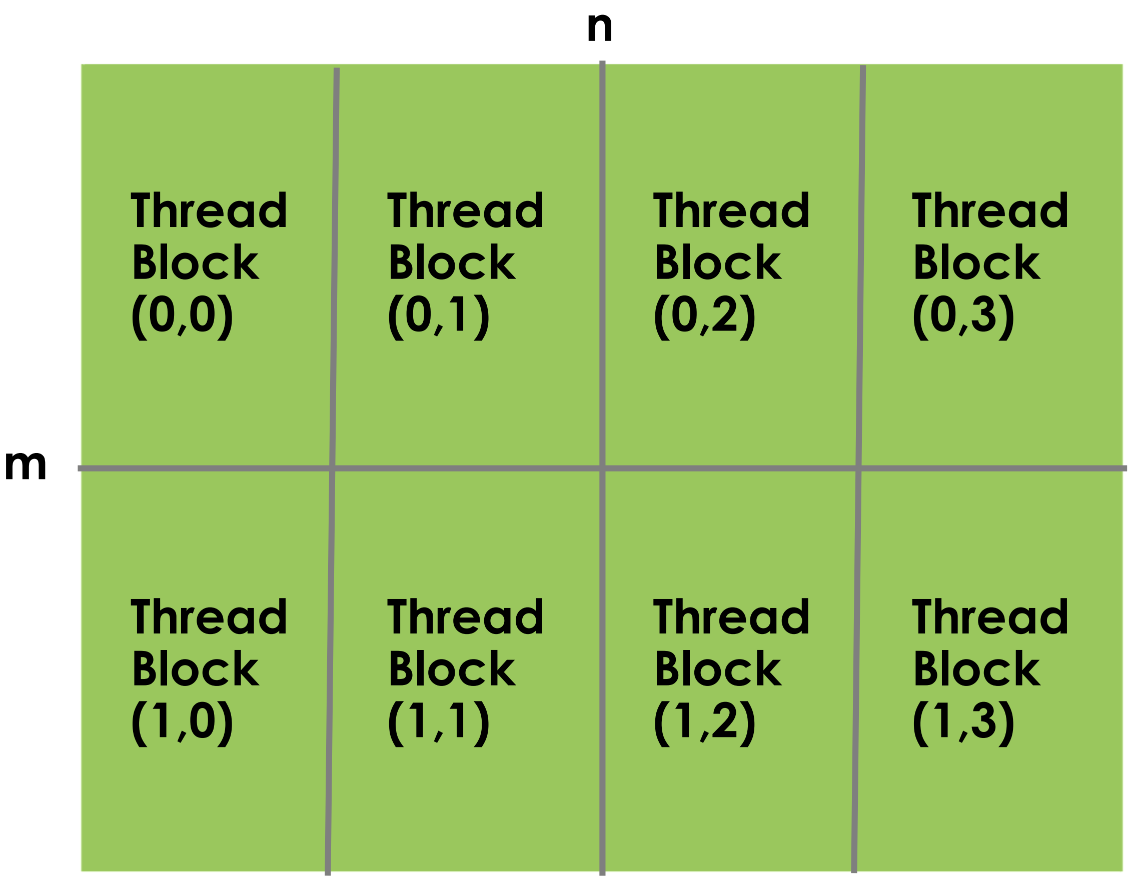
```
template <typename T>
__global__ void gpu_array_norm2(size_t arr_size,        //in: array size
                                const T * __restrict__ arr,     //in: pointer to arr[arr_size]
                                volatile T * norm)      //inout: sum of the squared elements of the array
{
 extern __shared__ double thread_norm[]; //dynamic shared memory of size blockDim.x

 size_t n = gridDim.x * blockDim.x;
 double tnorm = 0.0;
 for(size_t i = blockIdx.x * blockDim.x + threadIdx.x; i < arr_size; i += n) tnorm += arr[i] * arr[i];
 thread_norm[threadIdx.x] = tnorm;
 __syncthreads();

 unsigned int s = blockDim.x;
 while(s > 1){
  unsigned int j = (s+1U)>>1; //=(s+1)/2
  if(threadIdx.x + j < s) thread_norm[threadIdx.x] += thread_norm[threadIdx.x+j];
  __syncthreads();
  s = j;
 }

 if(threadIdx.x == 0){
  unsigned int j = 1;
  while(j){j = atomicMax(&norm_wr_lock,1);} //lock
  __threadfence();
  *norm += thread_norm[0]; //accumulate
  __threadfence();
  j=atomicExch(&norm_wr_lock,0); //unlock
 }
 __syncthreads();

return;
}
```

Thread 0 of each thread block accumulates the result into global memory (norm)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



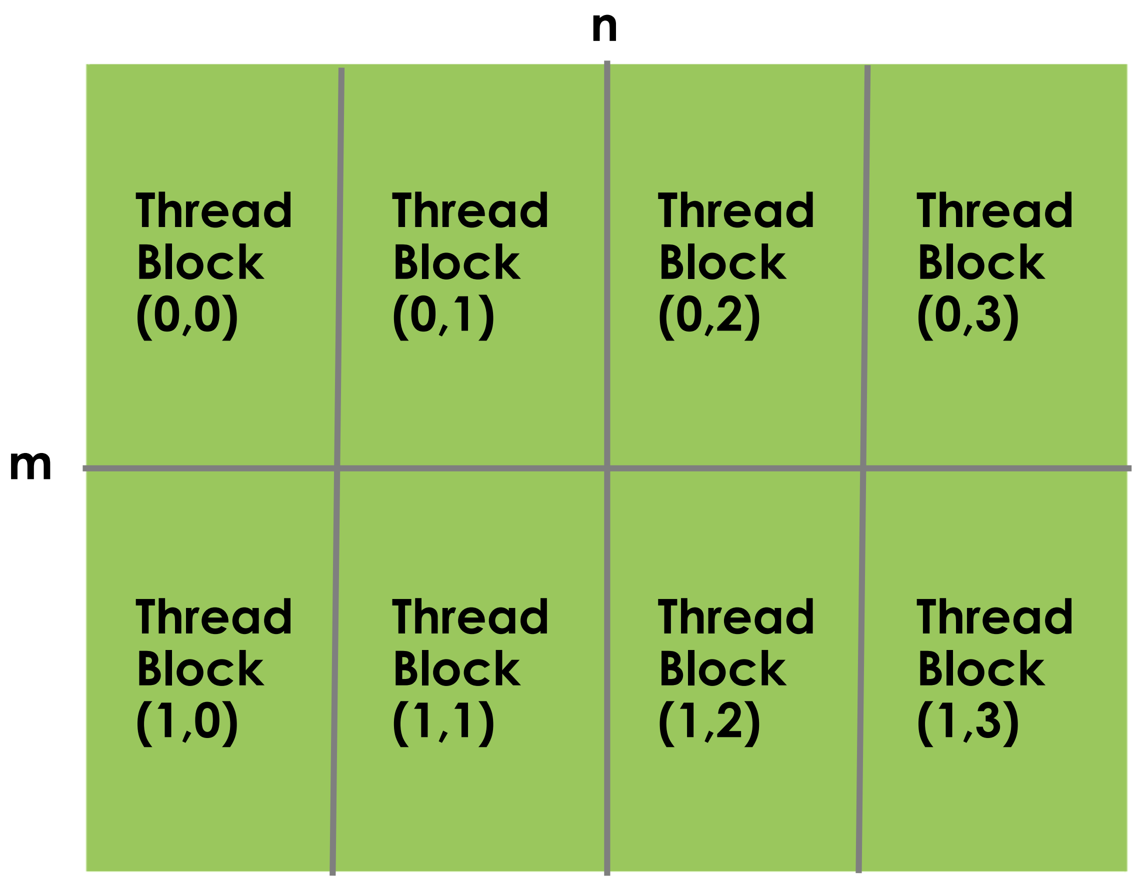
Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

float or double

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



Matrix C(m,n)

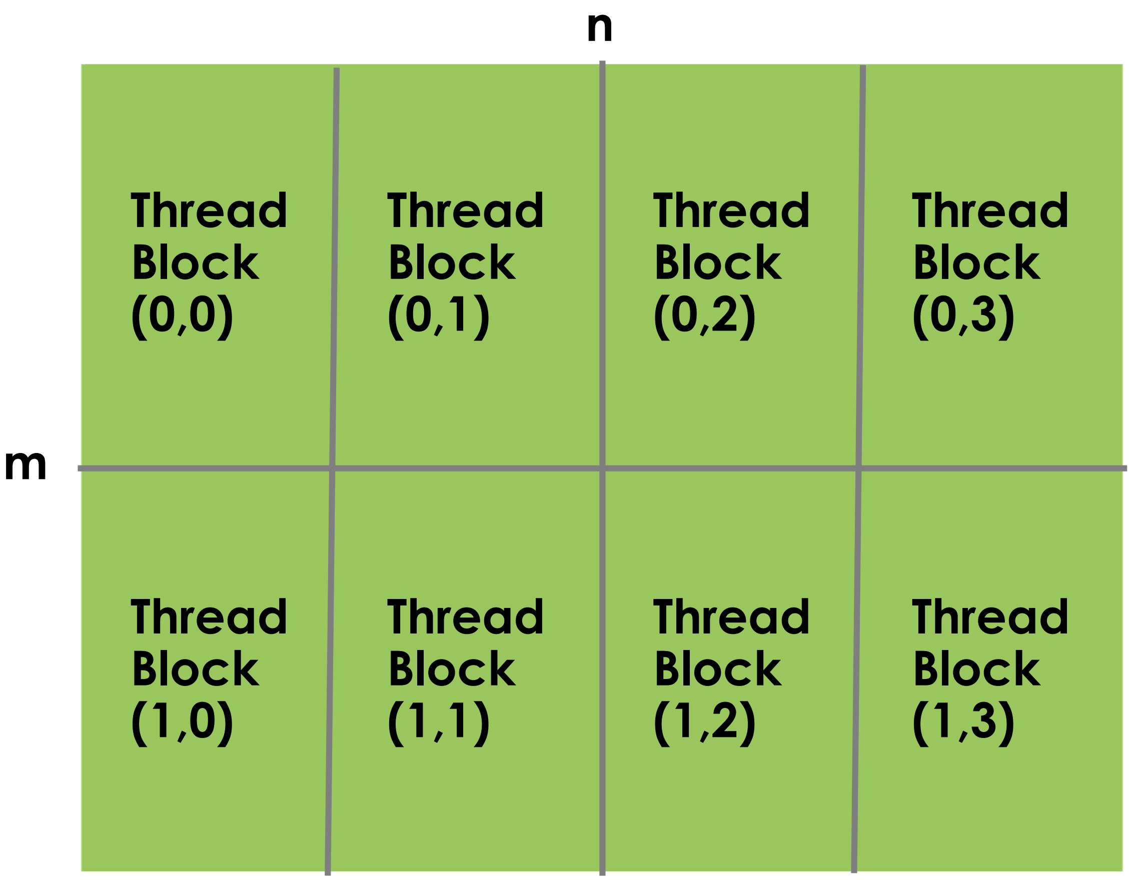
CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

No pointer aliasing

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

C A B

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

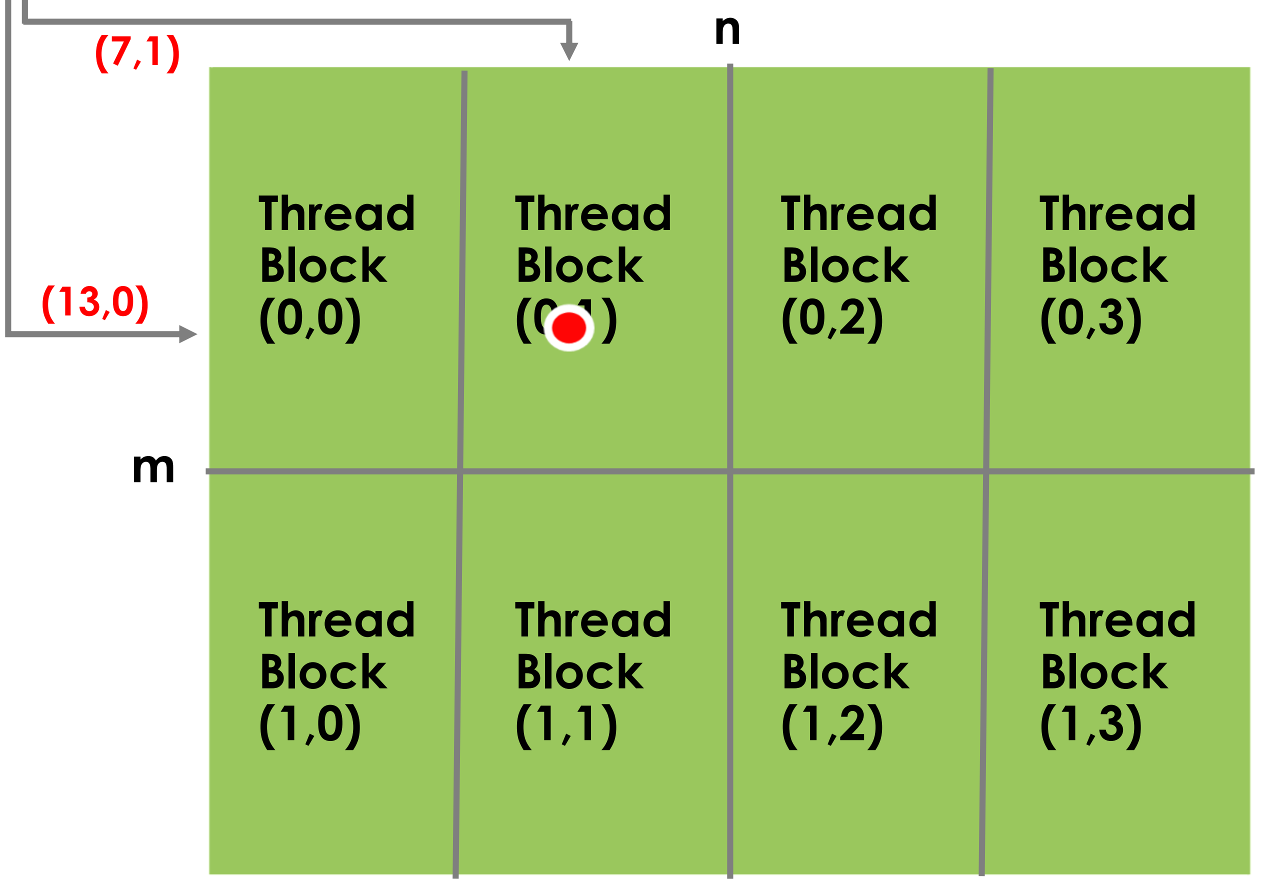


Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



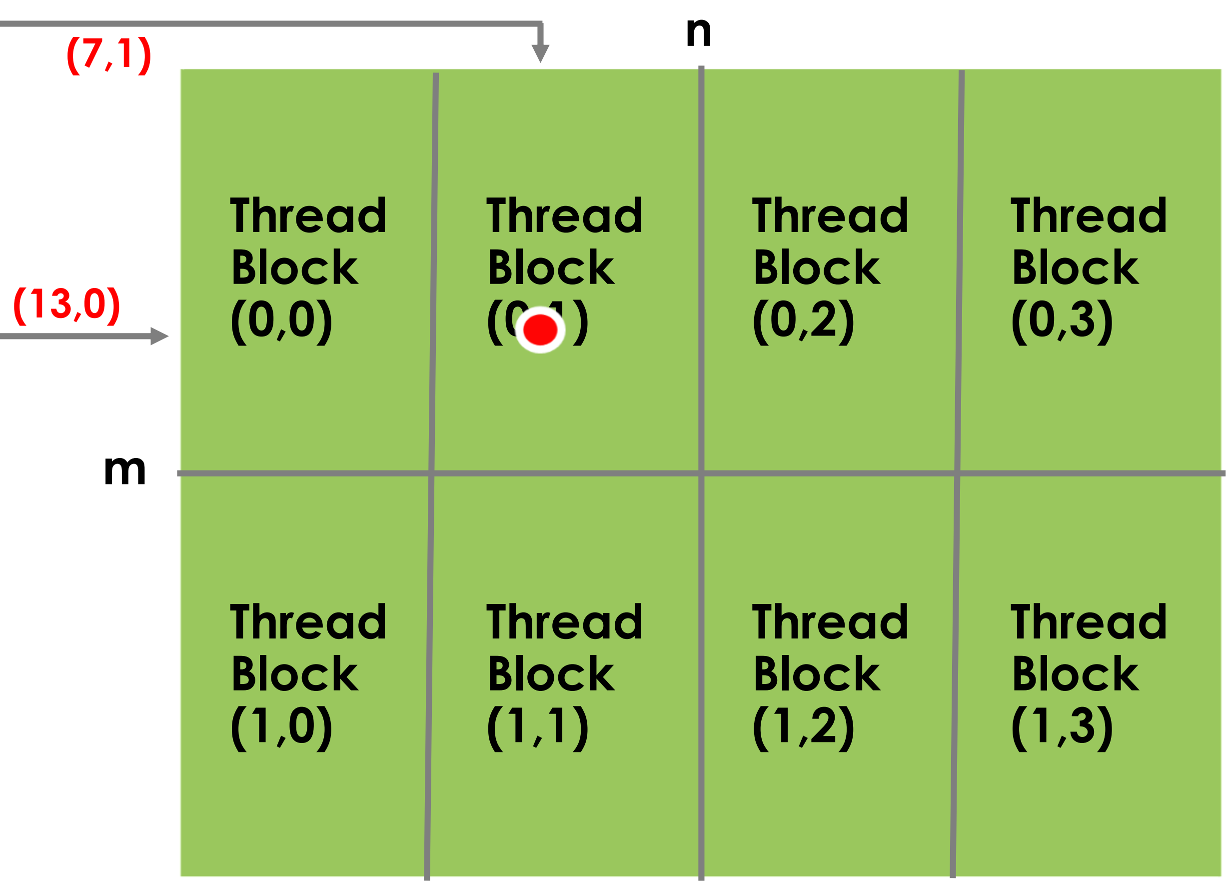
Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Bounds guards

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

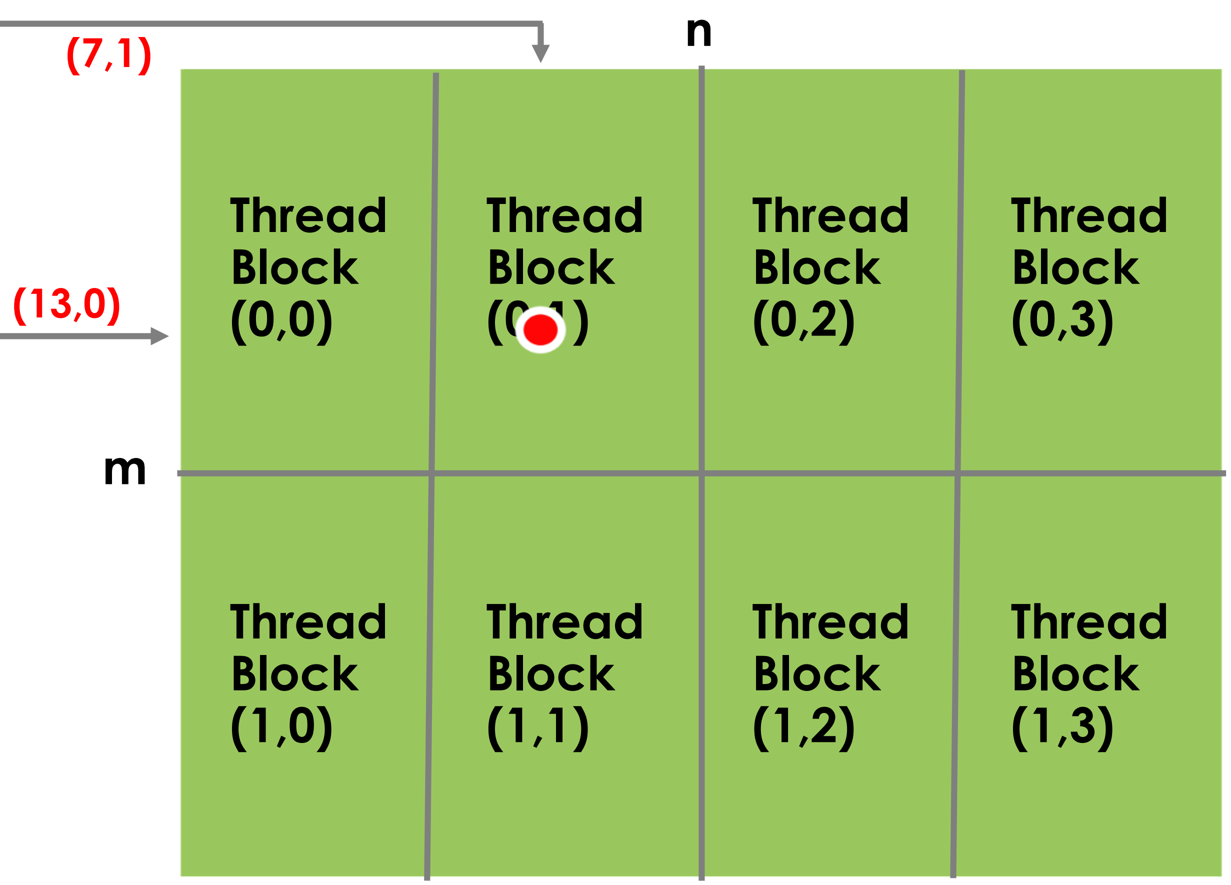


Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){
                       Init accumulator register
   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



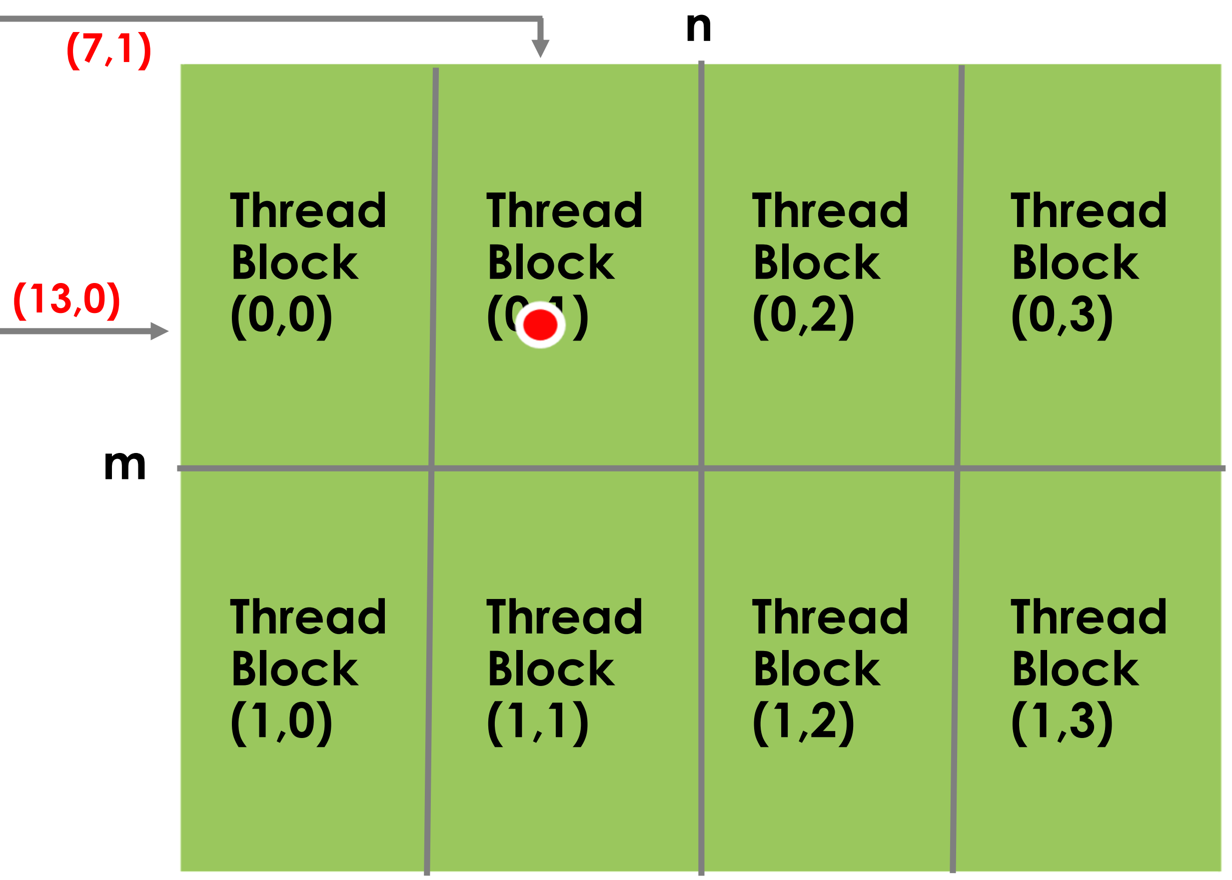
Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Loop over entire k dim

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



Matrix C(m,n)

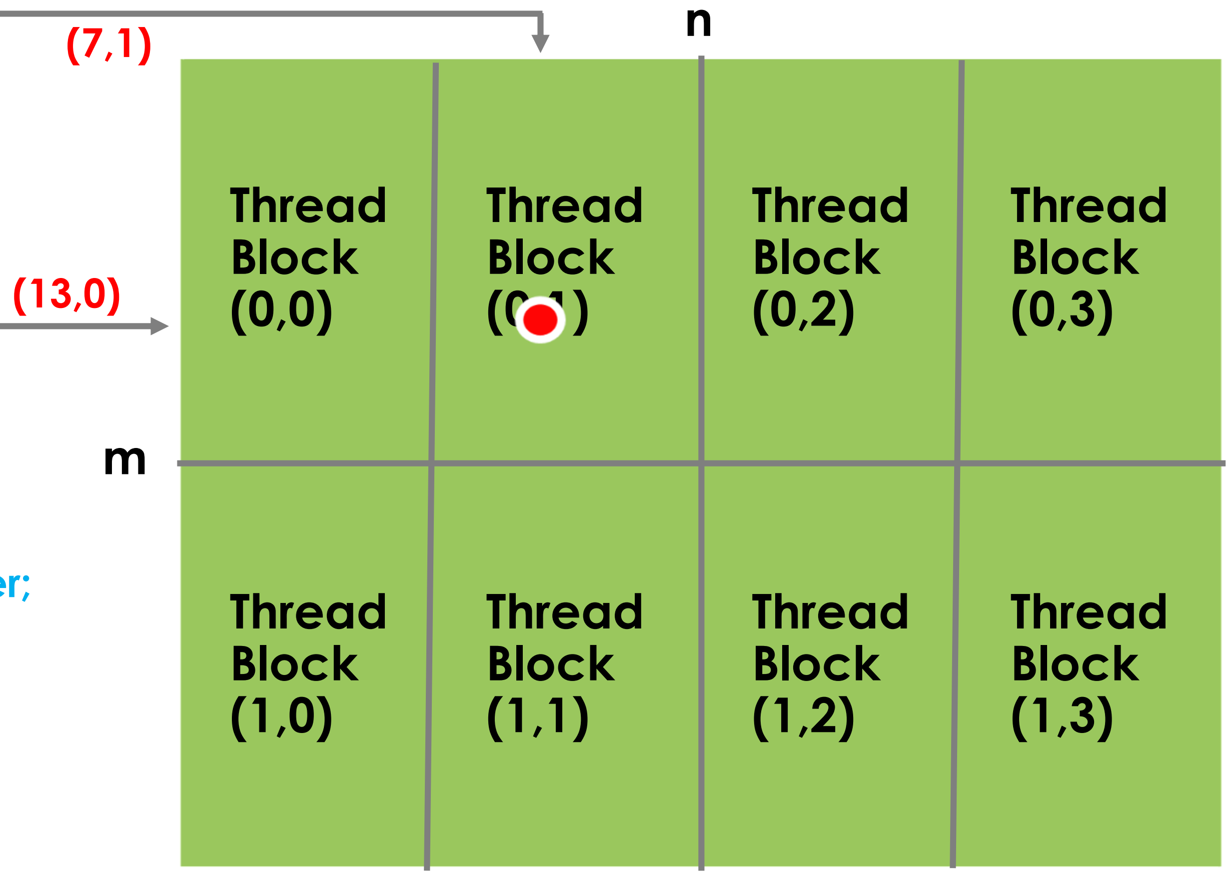
CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

Linear offsets are used for addressing A and B storage

Load element of A;
Load element of B;
Multiply;
Accumulate into register;



Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

Linear offsets are used for addressing A and B storage

Load element of A;
Load element of B;
Multiply;
Accumulate into register;

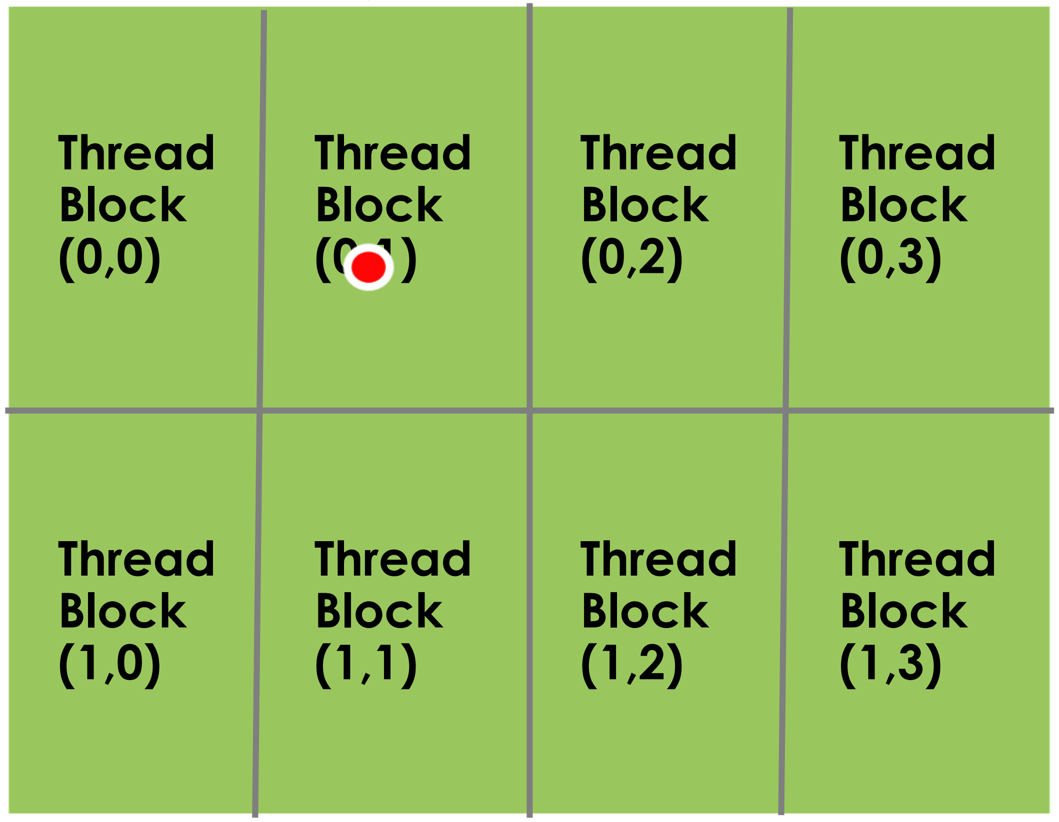
Global memory accesses to A and B are coalesced: threadIdx.x is the minor component

(7,1)

(13,0)

n

m



Matrix C(m,n)

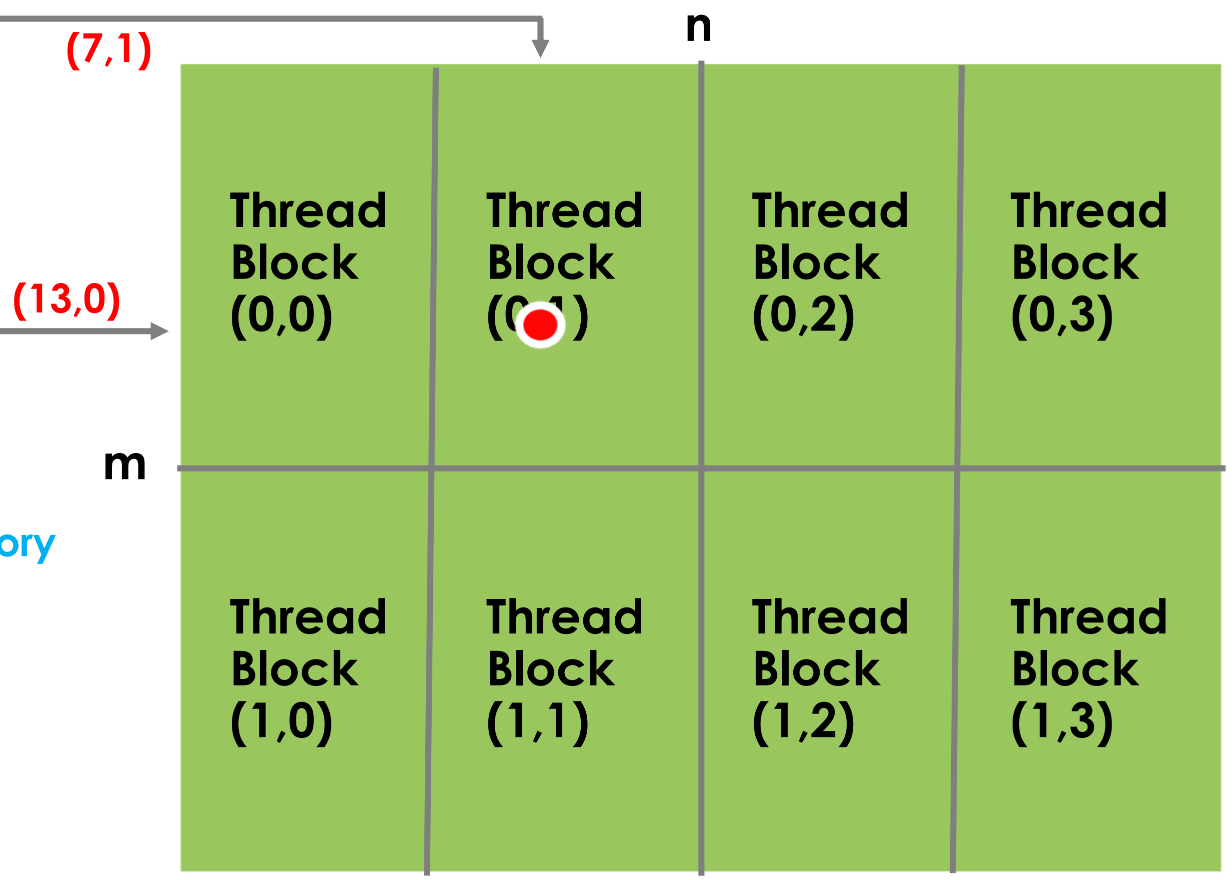
OAK RIDGE National Laboratory

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Upload register to global memory

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



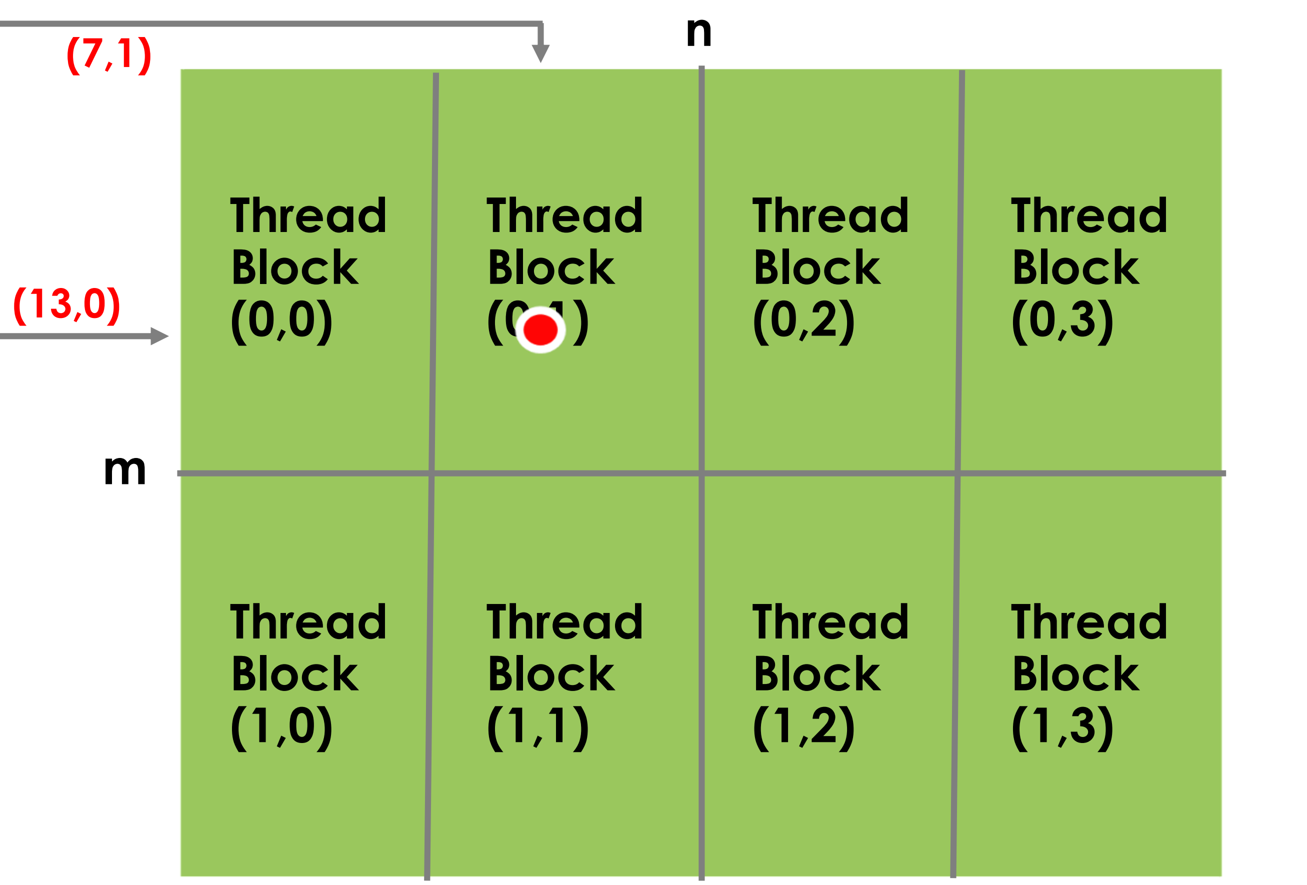
Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Loop over X and Y dims of C in case CUDA thread blocks do not cover full matrix C

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

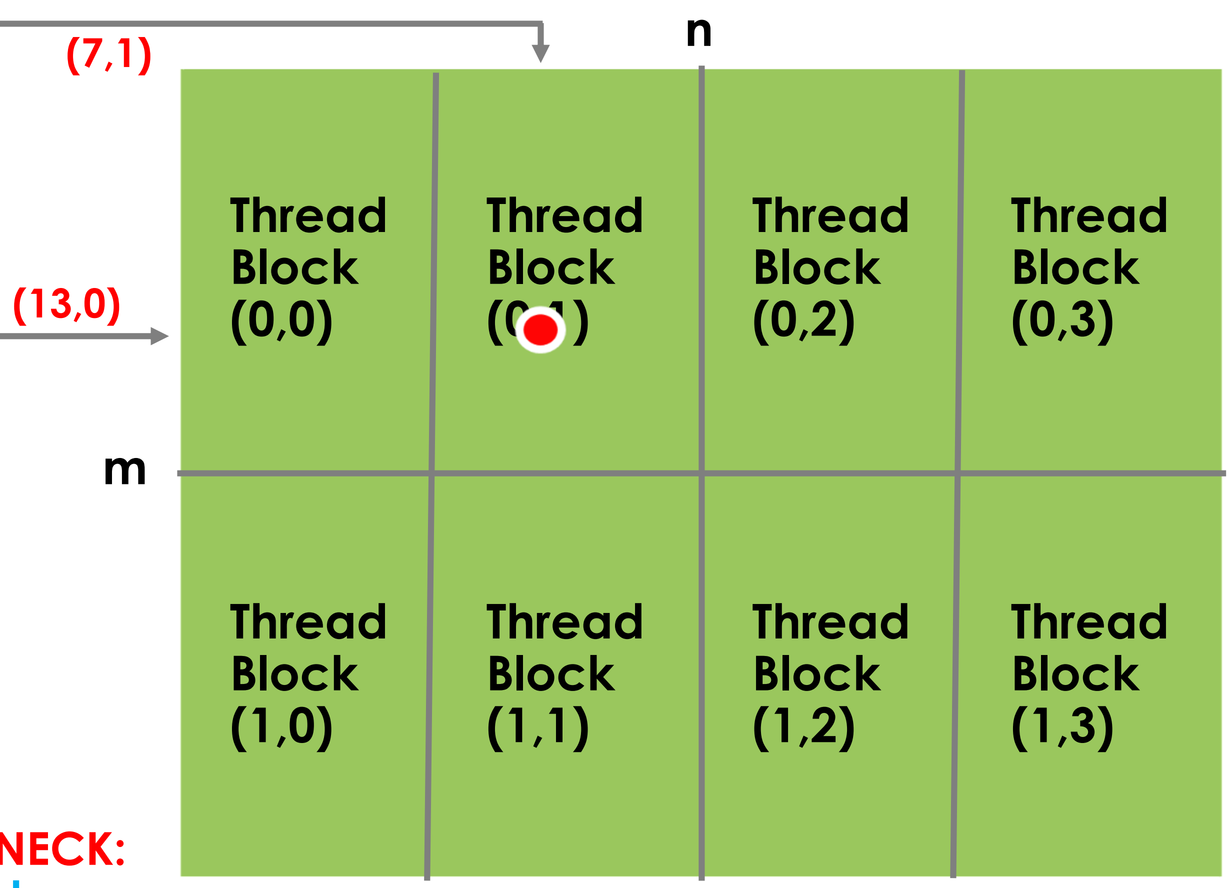


Matrix C(m,n)

CUDA BLA Library: Naïve GEMM kernel (algorithm 0)

```
template <typename T>
__global__ void gpu_gemm_nn(int m, int n, int k, T * __restrict__ dest, const T * __restrict__ left, const T * __restrict__ right)
{
 size_t ty = blockIdx.y*blockDim.y + threadIdx.y;
 size_t tx = blockIdx.x*blockDim.x + threadIdx.x;

 size_t n_pos = ty;
 while(n_pos < n){

  size_t m_pos = tx;
  while(m_pos < m){

   T tmp = static_cast<T>(0.0);
   for(size_t k_pos = 0; k_pos < k; ++k_pos){
    tmp += left[k_pos*m + m_pos] * right[n_pos*k + k_pos];
   }
   dest[n_pos*m + m_pos] += tmp;

   m_pos += gridDim.x*blockDim.x;
  }

  n_pos += gridDim.y*blockDim.y;
 }
 return;
}
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



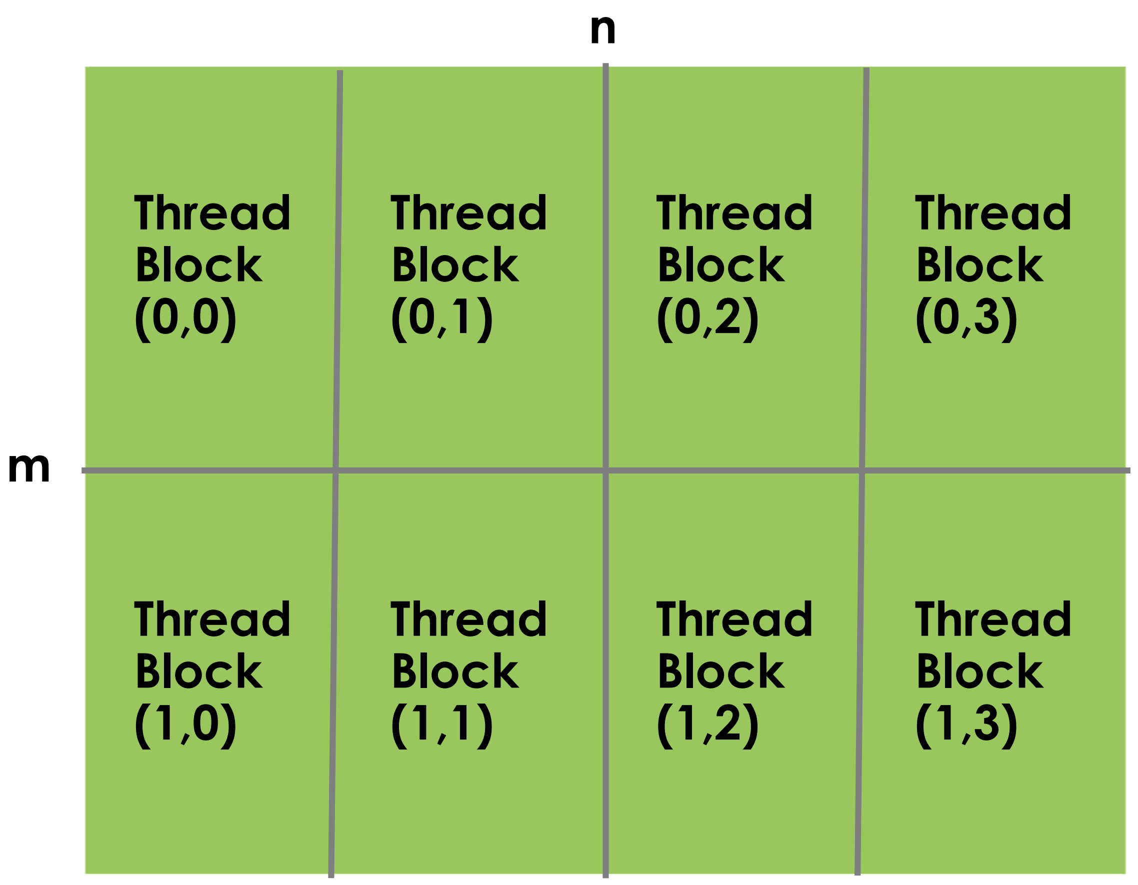
Matrix C(m,n)

GLOBAL MEMORY ACCESS BOTTLENECK:
2*bDim.y*bDim.x*k loads per block
2*m*n*k loads per kernel

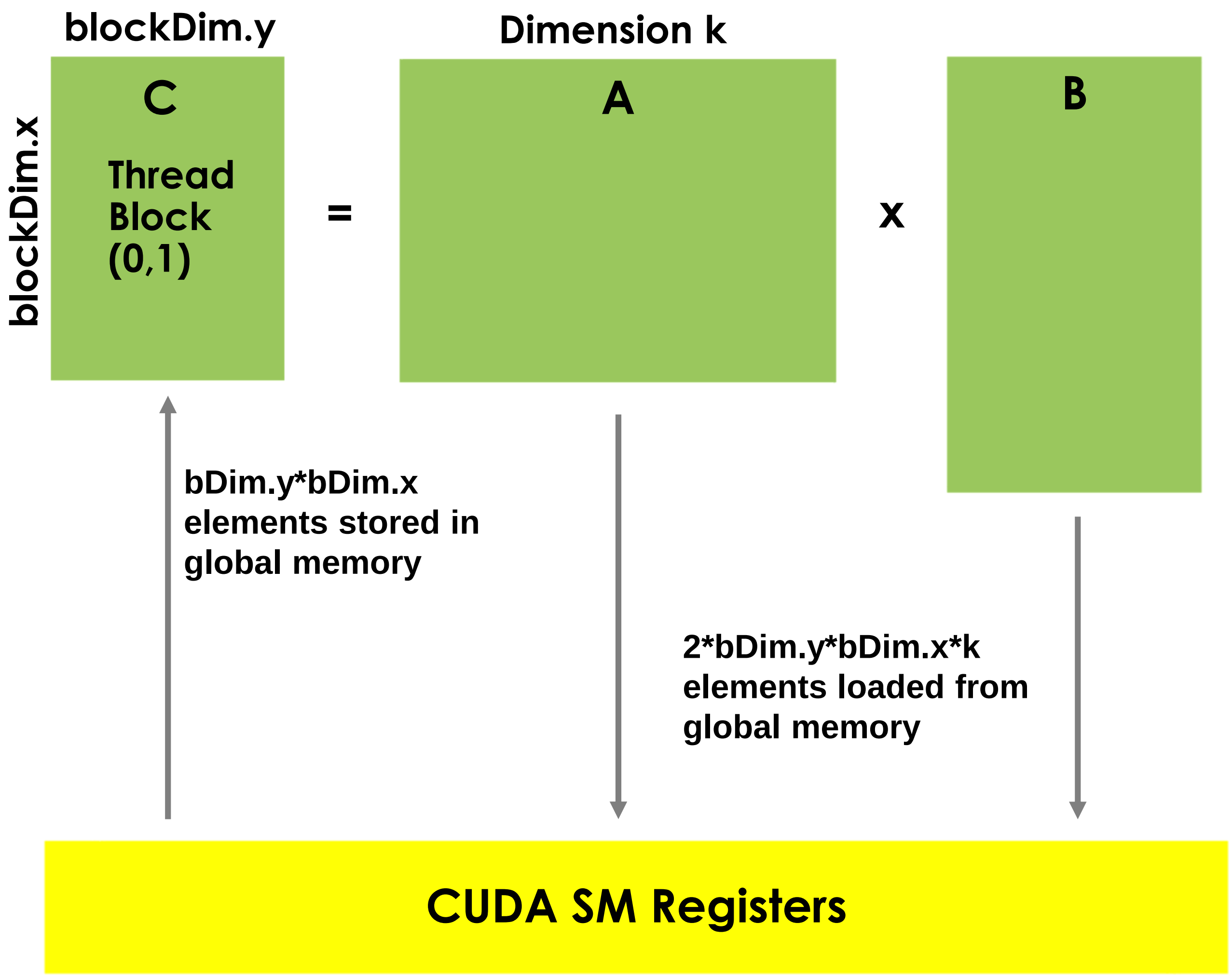
CUDA BLA Library: Shared Memory GEMM need

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

~2*m*n*k/bDim.x global loads per kernel



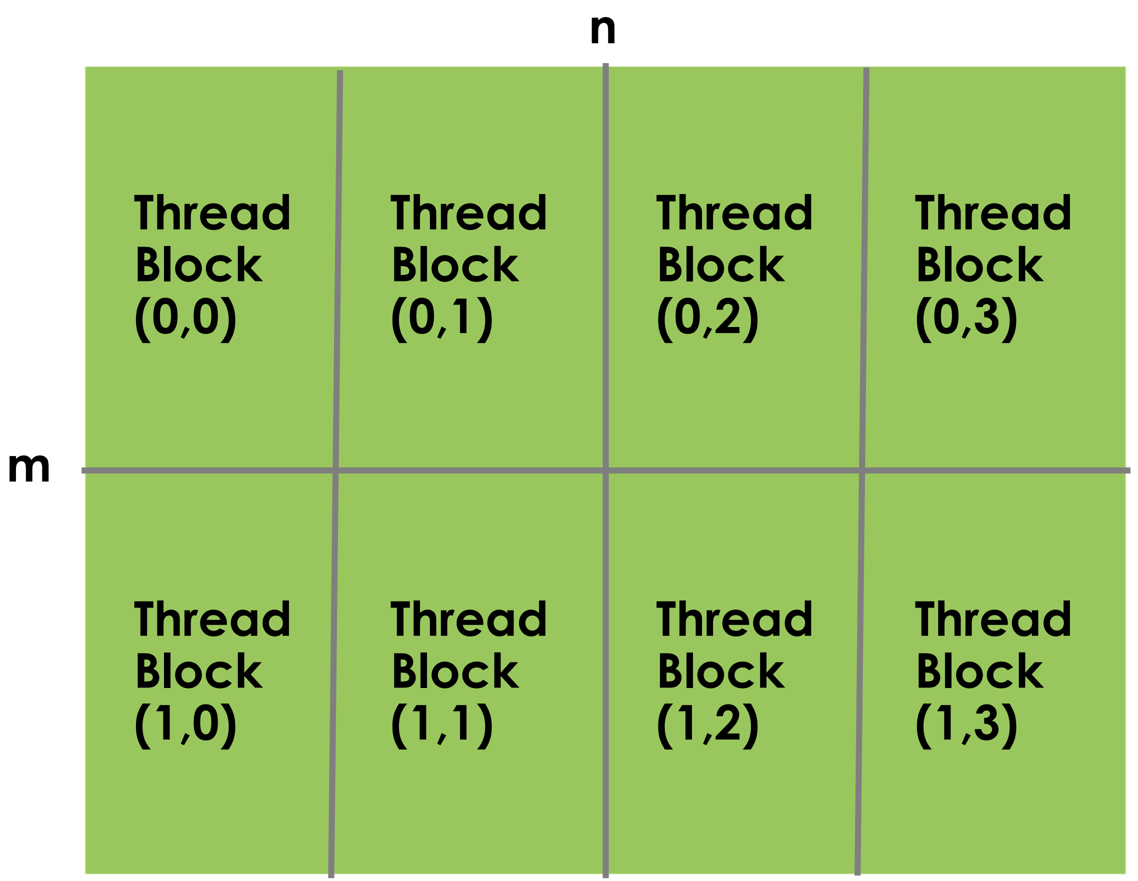
Matrix C(m,n)



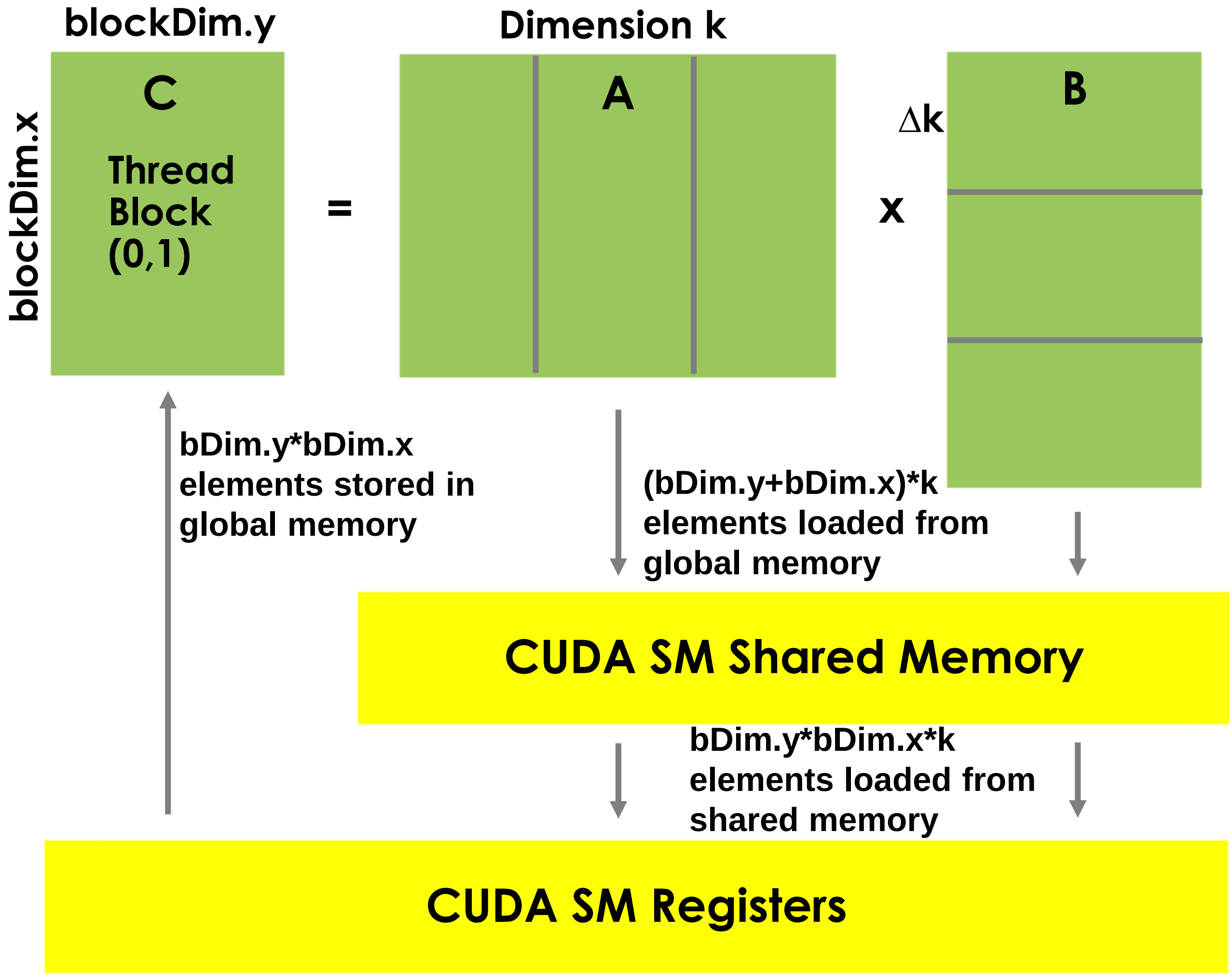
CUDA BLA Library: Shared Memory GEMM (algorithm 1)

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

~2*m*n*k/bDim.x global loads per kernel



Matrix C(m,n)

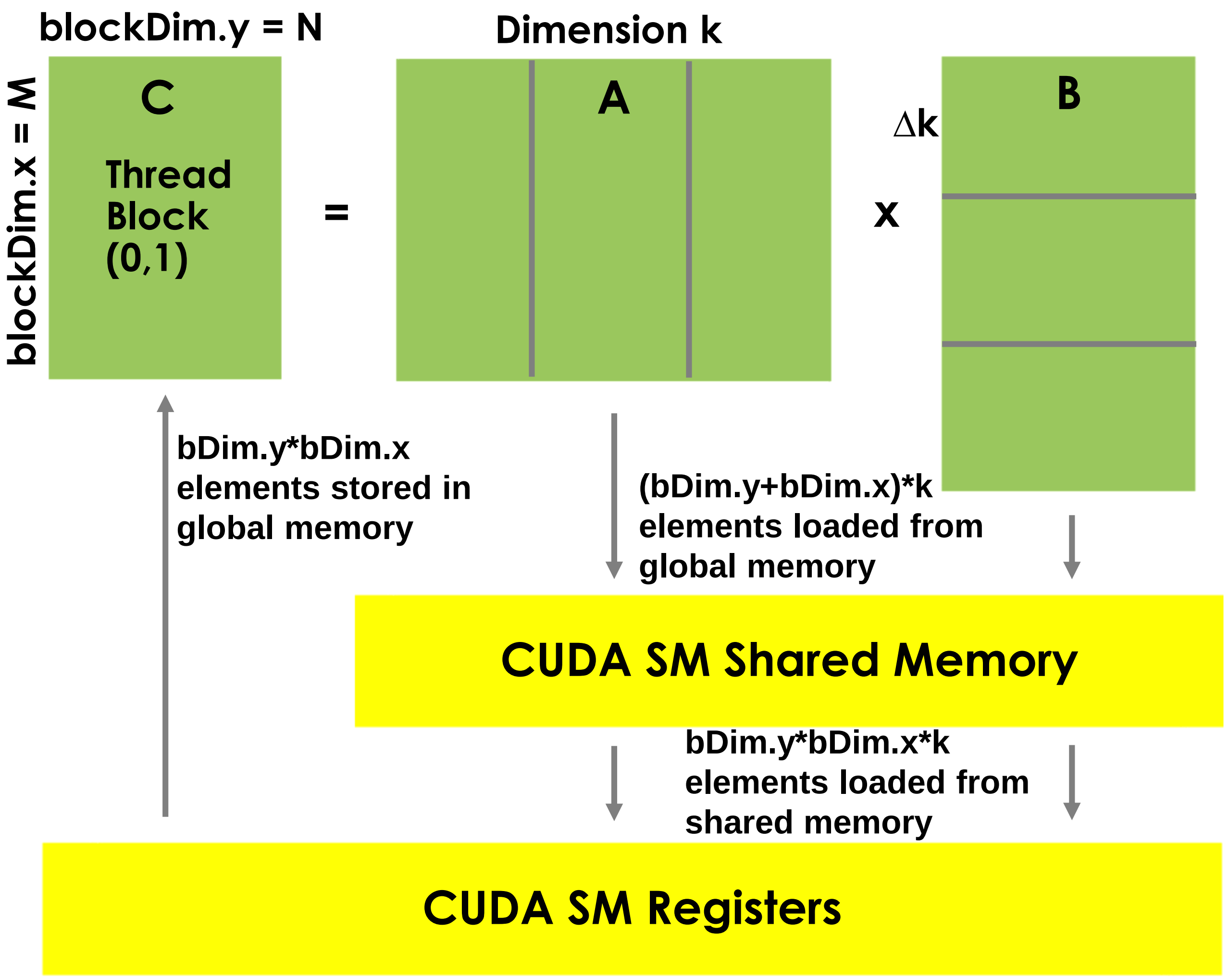


CUDA BLA Library: Shared Memory GEMM (algorithm 1)

TileC(M,N), TileA(M,K), TileB(K,N)

```
template <typename T, int TILE_EXT_N, int TILE_EXT_M, int TILE_EXT_K>
__global__ void gpu_gemm_sh_nn(int m, int n, int k,  //in: matrix dimensions: C(m,n)+=A(m,k)*B(k,n)
                               T * __restrict__ dest,          //inout: pointer to C matrix data
                               const T * __restrict__ left,    //in: pointer to A matrix data
                               const T * __restrict__ right) //in: pointer to B matrix data
{
 using int_t = int; //either int or size_t   Shared memory buffers (per thread block)
 __shared__ T lbuf[TILE_EXT_K][TILE_EXT_M], rbuf[TILE_EXT_N][TILE_EXT_K];

 for(int_t n_pos = blockIdx.y*blockDim.y; n_pos < n; n_pos += gridDim.y*blockDim.y){ //tile offset in Y

  for(int_t m_pos = blockIdx.x*blockDim.x; m_pos < m; m_pos += gridDim.x*blockDim.x){ //tile offset in X

   T tmp = static_cast<T>(0.0); //accumulator

   for(int_t k_pos = 0; k_pos < k; k_pos += TILE_EXT_K){ //tile begin position along dimension K
    int_t k_end = k_pos + TILE_EXT_K; if(k_end > k) k_end = k;

    //Load a tile of matrix A(m_pos:TILE_EXT_M, k_pos:TILE_EXT_K):
    if(m_pos + threadIdx.x < m){
     for(int_t k_loc = k_pos + threadIdx.y; k_loc < k_end; k_loc += blockDim.y){
      lbuf[k_loc-k_pos][threadIdx.x] = left[k_loc*m + (m_pos+threadIdx.x)];
     }
    }

    //Load a tile of matrix B(k_pos:TILE_EXT_K, n_pos:TILE_EXT_N):
    if(n_pos + threadIdx.y < n){
     for(int_t k_loc = k_pos + threadIdx.x; k_loc < k_end; k_loc += blockDim.x){
      rbuf[threadIdx.y][k_loc-k_pos] = right[(n_pos+threadIdx.y)*k + k_loc];
     }
    }
    __syncthreads();
```

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



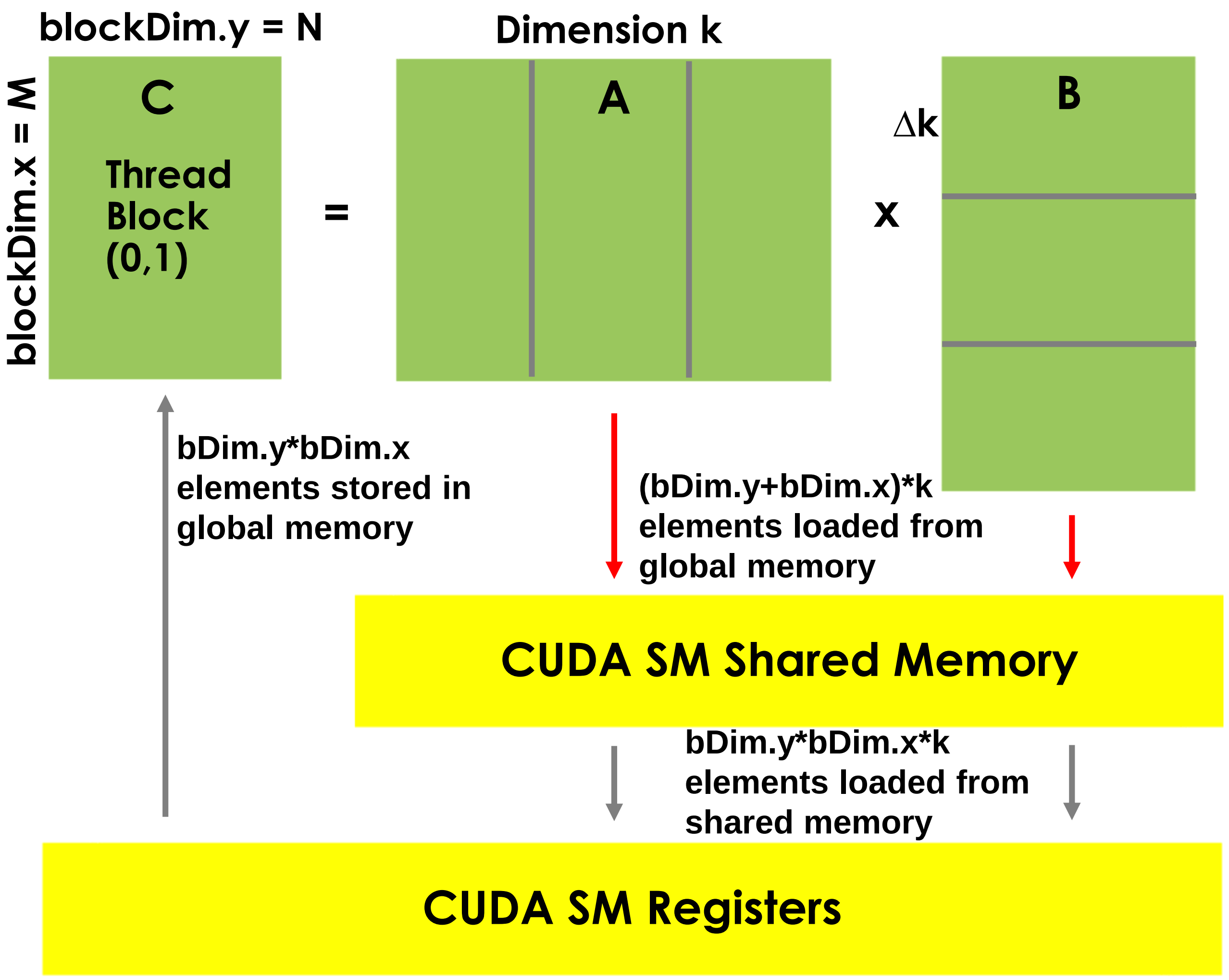
CUDA BLA Library: Shared Memory GEMM (algorithm 1)

TileC(M,N), TileA(M,K), TileB(K,N)

```
template <typename T, int TILE_EXT_N, int TILE_EXT_M, int TILE_EXT_K>
__global__ void gpu_gemm_sh_nn(int m, int n, int k,  //in: matrix dimensions: C(m,n)+=A(m,k)*B(k,n)
                               T * __restrict__ dest,            //inout: pointer to C matrix data
                               const T * __restrict__ left,    //in: pointer to A matrix data
                               const T * __restrict__ right) //in: pointer to B matrix data
{
 using int_t = int; //either int or size_t
 __shared__ T lbuf[TILE_EXT_K][TILE_EXT_M], rbuf[TILE_EXT_N][TILE_EXT_K];

 for(int_t n_pos = blockIdx.y*blockDim.y; n_pos < n; n_pos += gridDim.y*blockDim.y){ //tile offset in Y

  for(int_t m_pos = blockIdx.x*blockDim.x; m_pos < m; m_pos += gridDim.x*blockDim.x){ //tile offset in X

   T tmp = static_cast<T>(0.0); //accumulator

   for(int_t k_pos = 0; k_pos < k; k_pos += TILE_EXT_K){ //tile begin position along dimension K
    int_t k_end = k_pos + TILE_EXT_K; if(k_end > k) k_end = k;

    //Load a tile of matrix A(m_pos:TILE_EXT_M, k_pos:TILE_EXT_K):
    if(m_pos + threadIdx.x < m){
     for(int_t k_loc = k_pos + threadIdx.y; k_loc < k_end; k_loc += blockDim.y){
      lbuf[k_loc-k_pos][threadIdx.x] = left[k_loc*m + (m_pos+threadIdx.x)];
     }
    }

    //Load a tile of matrix B(k_pos:TILE_EXT_K, n_pos:TILE_EXT_N):
    if(n_pos + threadIdx.y < n){
     for(int_t k_loc = k_pos + threadIdx.x; k_loc < k_end; k_loc += blockDim.x){
      rbuf[threadIdx.y][k_loc-k_pos] = right[(n_pos+threadIdx.y)*k + k_loc];
     }
    }
    __syncthreads();
```

Loading shared memory buffers

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



CUDA BLA Library: Shared Memory GEMM (algorithm 1)

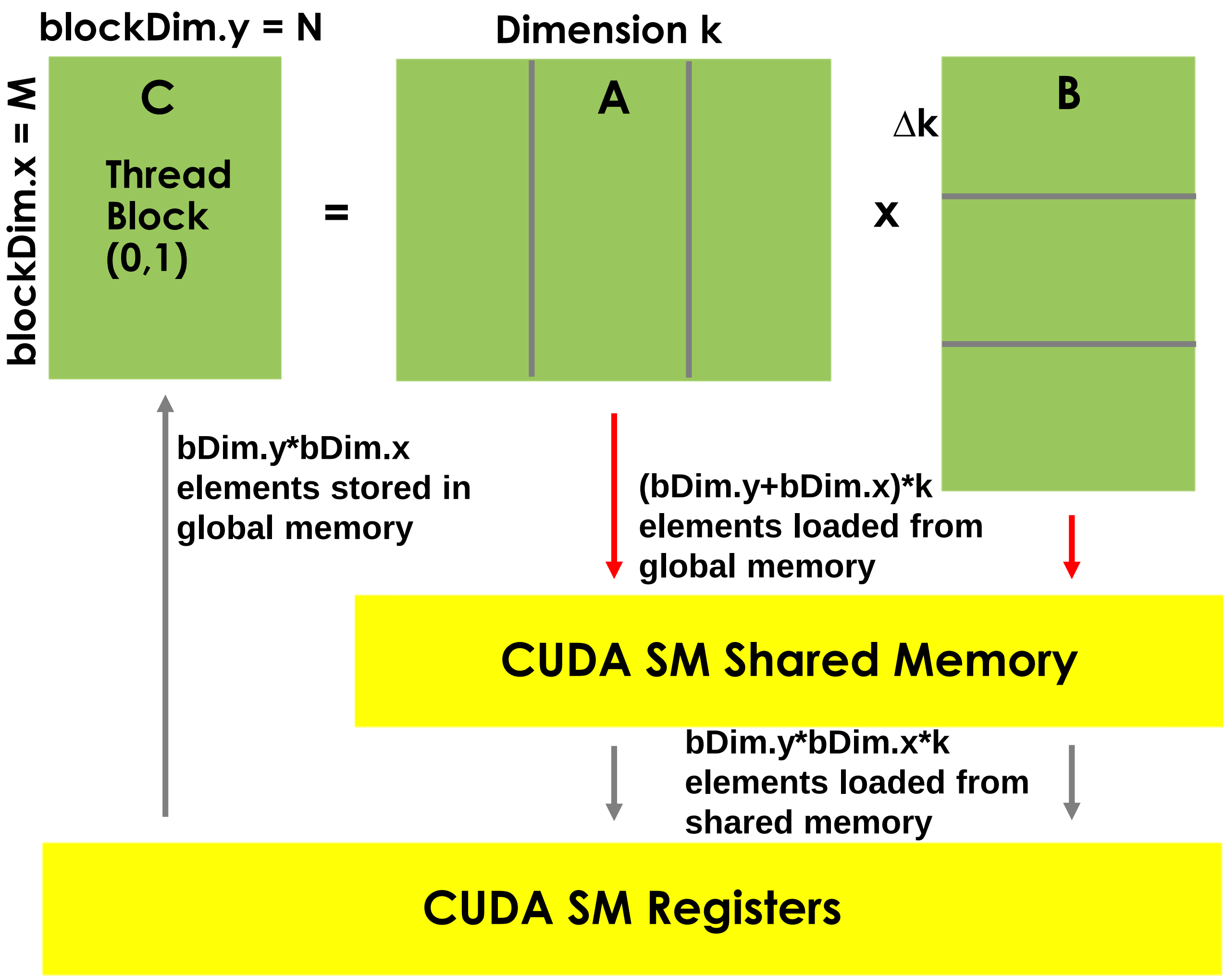
TileC(M,N), TileA(M,K), TileB(K,N)

```
template <typename T, int TILE_EXT_N, int TILE_EXT_M, int TILE_EXT_K>
__global__ void gpu_gemm_sh_nn(int m, int n, int k,  //in: matrix dimensions: C(m,n)+=A(m,k)*B(k,n)
                               T * __restrict__ dest,          //inout: pointer to C matrix data
                               const T * __restrict__ left,    //in: pointer to A matrix data
                               const T * __restrict__ right)   //in: pointer to B matrix data
{
 using int_t = int; //either int or size_t
 __shared__ T lbuf[TILE_EXT_K][TILE_EXT_M], rbuf[TILE_EXT_N][TILE_EXT_K];

 for(int_t n_pos = blockIdx.y*blockDim.y; n_pos < n; n_pos += gridDim.y*blockDim.y){ //tile offset in Y

  for(int_t m_pos = blockIdx.x*blockDim.x; m_pos < m; m_pos += gridDim.x*blockDim.x){ //tile offset in X

   T tmp = static_cast<T>(0.0); //accumulator

   for(int_t k_pos = 0; k_pos < k; k_pos += TILE_EXT_K){ //tile begin position along dimension K
    int_t k_end = k_pos + TILE_EXT_K; if(k_end > k) k_end = k;

    //Load a tile of matrix A(m_pos:TILE_EXT_M, k_pos:TILE_EXT_K):
    if(m_pos + threadIdx.x < m){
     for(int_t k_loc = k_pos + threadIdx.y; k_loc < k_end; k_loc += blockDim.y){
      lbuf[k_loc-k_pos][threadIdx.x] = left[k_loc*m + (m_pos+threadIdx.x)];
     }
    }

    //Load a tile of matrix B(k_pos:TILE_EXT_K, n_pos:TILE_EXT_N):
    if(n_pos + threadIdx.y < n){
     for(int_t k_loc = k_pos + threadIdx.x; k_loc < k_end; k_loc += blockDim.x){
      rbuf[threadIdx.y][k_loc-k_pos] = right[(n_pos+threadIdx.y)*k + k_loc];
     }
    }
    __syncthreads();
```

Loading shared memory buffers

Global memory accesses to A and B are coalesced: threadIdx.x is the minor component

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



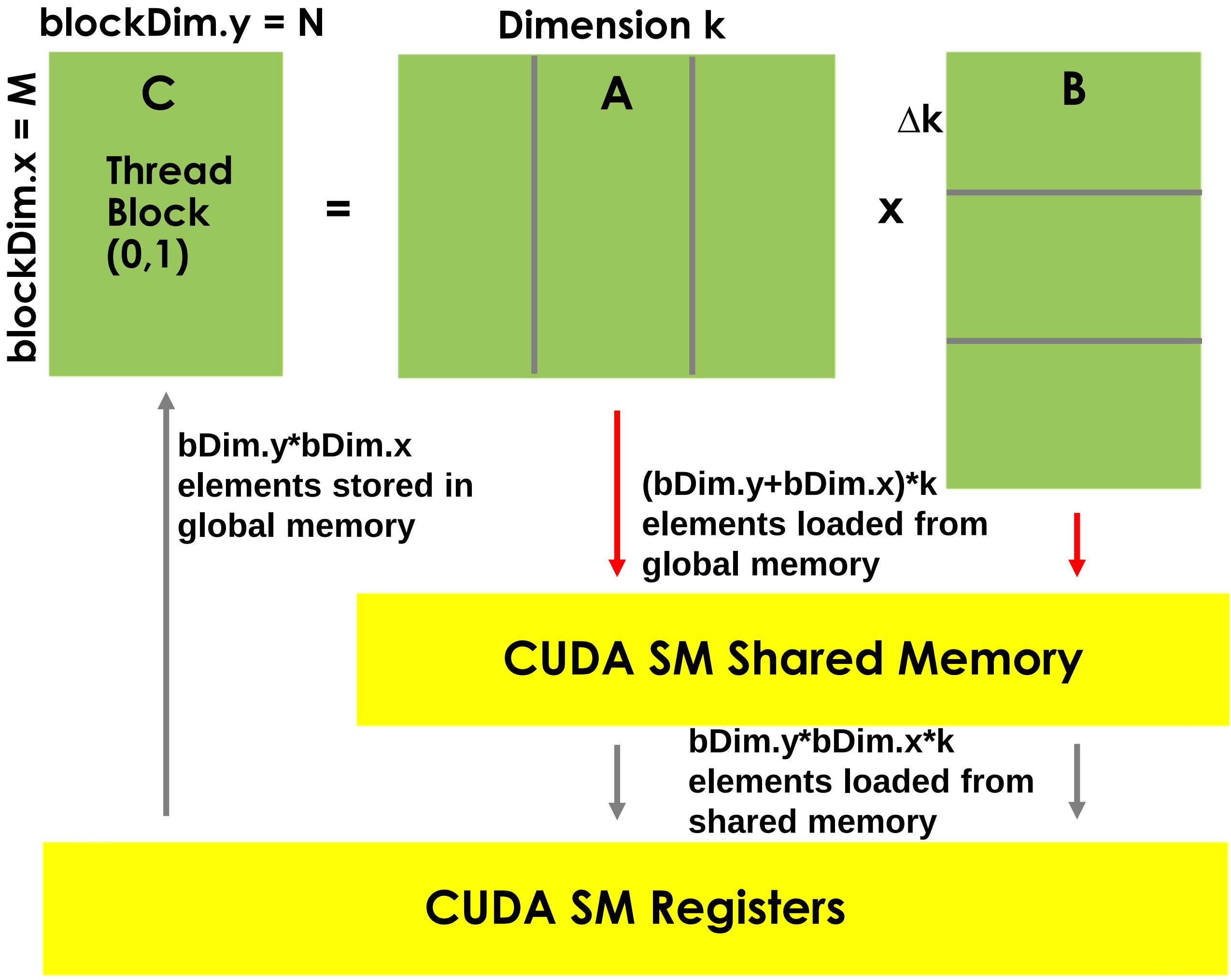
CUDA BLA Library: Shared Memory GEMM (algorithm 1)

TileC(M,N), TileA(M,K), TileB(K,N)

```
template <typename T, int TILE_EXT_N, int TILE_EXT_M, int TILE_EXT_K>
__global__ void gpu_gemm_sh_nn(int m, int n, int k,  //in: matrix dimensions: C(m,n)+=A(m,k)*B(k,n)
                               T * __restrict__ dest,          //inout: pointer to C matrix data
                               const T * __restrict__ left,    //in: pointer to A matrix data
                               const T * __restrict__ right) //in: pointer to B matrix data
{
 using int_t = int; //either int or size_t
 __shared__ T lbuf[TILE_EXT_K][TILE_EXT_M], rbuf[TILE_EXT_N][TILE_EXT_K];

 for(int_t n_pos = blockIdx.y*blockDim.y; n_pos < n; n_pos += gridDim.y*blockDim.y){ //tile offset in Y

  for(int_t m_pos = blockIdx.x*blockDim.x; m_pos < m; m_pos += gridDim.x*blockDim.x){ //tile offset in X

   T tmp = static_cast<T>(0.0); //accumulator

   for(int_t k_pos = 0; k_pos < k; k_pos += TILE_EXT_K){ //tile begin position along dimension K
    int_t k_end = k_pos + TILE_EXT_K; if(k_end > k) k_end = k;

    //Load a tile of matrix A(m_pos:TILE_EXT_M, k_pos:TILE_EXT_K):
    if(m_pos + threadIdx.x < m){
     for(int_t k_loc = k_pos + threadIdx.y; k_loc < k_end; k_loc += blockDim.y){
      lbuf[k_loc-k_pos][threadIdx.x] = left[k_loc*m + (m_pos+threadIdx.x)];
     }
    }

    //Load a tile of matrix B(k_pos:TILE_EXT_K, n_pos:TILE_EXT_N):
    if(n_pos + threadIdx.y < n){
     for(int_t k_loc = k_pos + threadIdx.x; k_loc < k_end; k_loc += blockDim.x){
      rbuf[threadIdx.y][k_loc-k_pos] = right[(n_pos+threadIdx.y)*k + k_loc];
     }
    }
    __syncthreads();
```

Synchronizes threads in a thread block

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

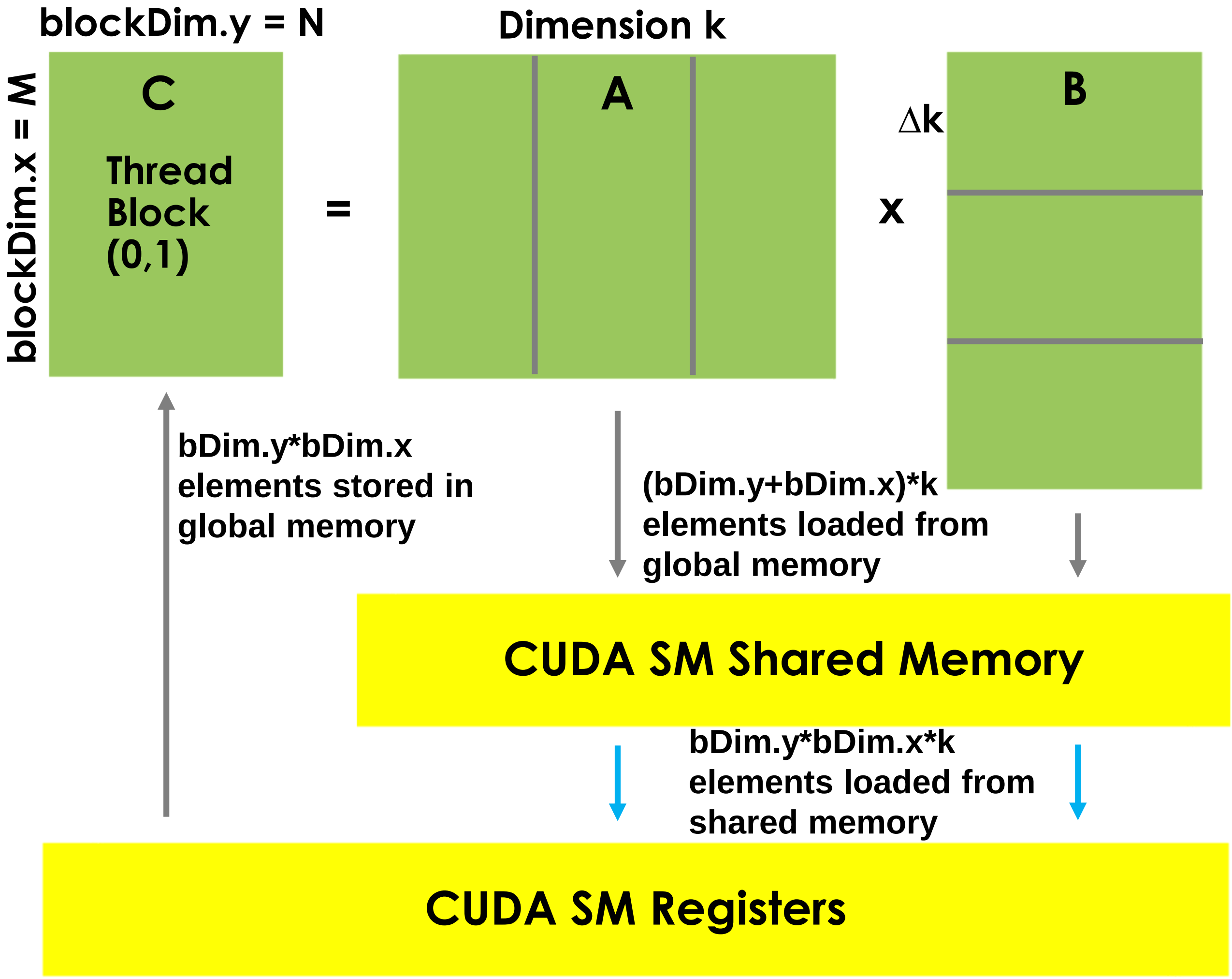


Oak Ridge National Laboratory

CUDA BLA Library: Shared Memory GEMM (algorithm 1)

```
//Multiply two loaded tiles to produce a tile of matrix:
    if(m_pos + threadIdx.x < m && n_pos + threadIdx.y < n){
     if(k_end - k_pos == TILE_EXT_K){ //known loop count: Unroll
#pragma unroll Unroll loop for performance
      for(int_t l = 0; l < TILE_EXT_K; ++l){
       tmp += lbuf[l][threadIdx.x] * rbuf[threadIdx.y][l];
      }
     }else{ //number of loop iterations is not known at compile time
      for(int_t l = 0; l < (k_end - k_pos); ++l){
       tmp += lbuf[l][threadIdx.x] * rbuf[threadIdx.y][l];
      }                    Performing matrix multiplication
     }                     from shared memory buffers
    }
    __syncthreads();

   } //k_pos

   //Store element of the C matrix in global memory:
   if(m_pos + threadIdx.x < m && n_pos + threadIdx.y < n)
    dest[(n_pos+threadIdx.y)*m + (m_pos+threadIdx.x)] += tmp;
```

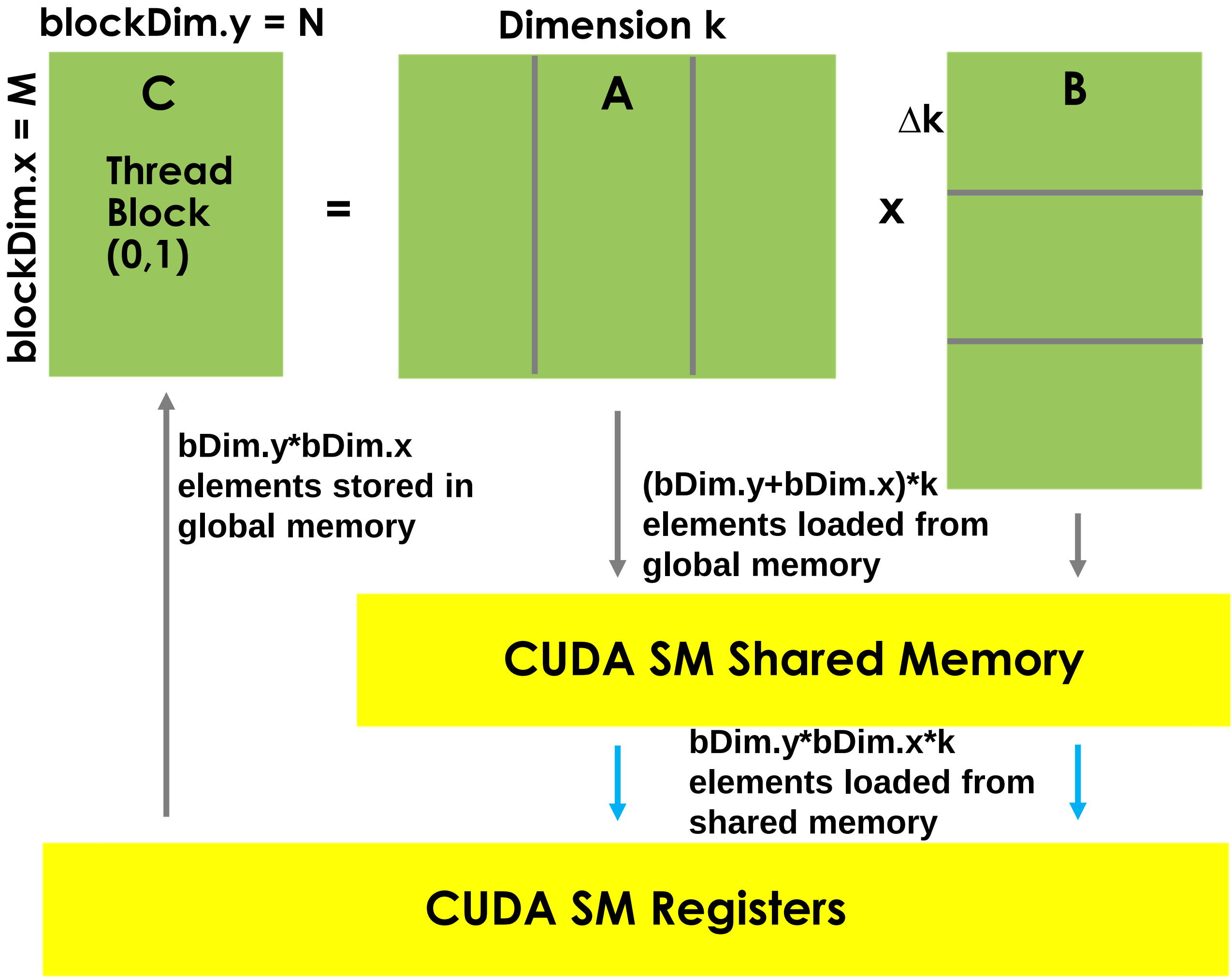
Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



CUDA BLA Library: Shared Memory GEMM (algorithm 1)

```
//Multiply two loaded tiles to produce a tile of matrix:
    if(m_pos + threadIdx.x < m && n_pos + threadIdx.y < n){
     if(k_end - k_pos == TILE_EXT_K){ //known loop count: Unroll
#pragma unroll Unroll loop for performance
      for(int_t l = 0; l < TILE_EXT_K; ++l){
       tmp += lbuf[l][threadIdx.x] * rbuf[threadIdx.y][l];
      }
     }else{ //number of loop iterations is not known at compile time
      for(int_t l = 0; l < (k_end - k_pos); ++l){
       tmp += lbuf[l][threadIdx.x] * rbuf[threadIdx.y][l];
      }                   Performing matrix multiplication
     }                    from shared memory buffers
    }
    __syncthreads(); Synchronizes threads in a thread block

   } //k_pos

   //Store element of the C matrix in global memory:
   if(m_pos + threadIdx.x < m && n_pos + threadIdx.y < n)
    dest[(n_pos+threadIdx.y)*m + (m_pos+threadIdx.x)] += tmp;
      Upload register to global memory (as before)
```

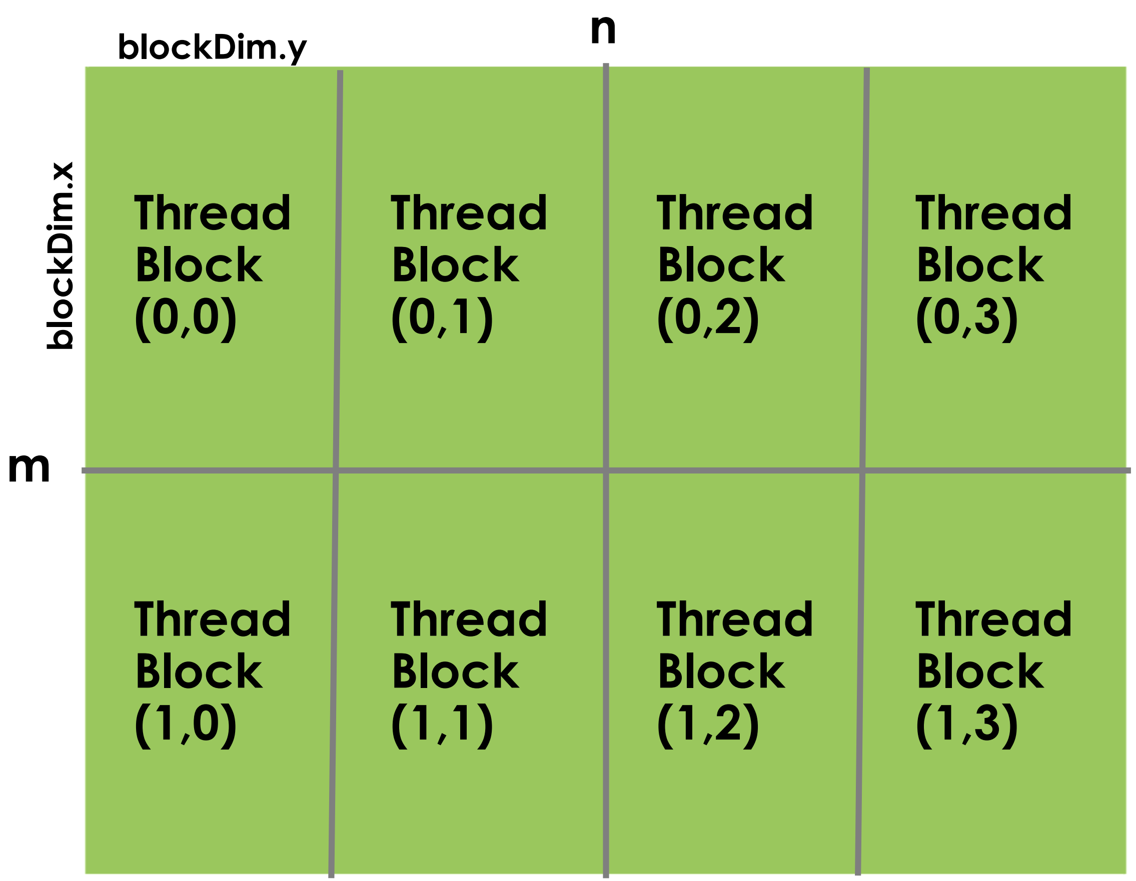
Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)



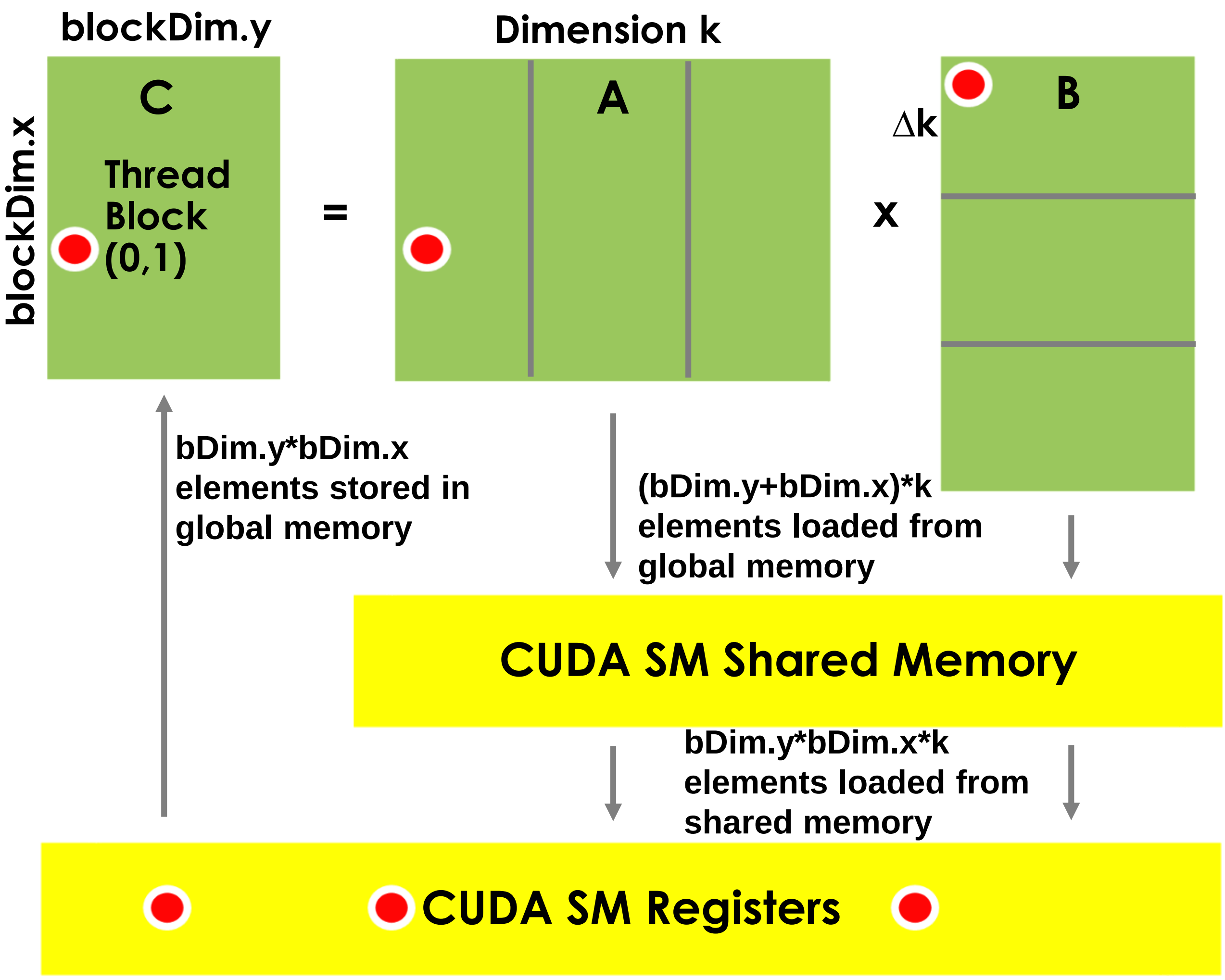
CUDA BLA Library: +Registers GEMM need

Each CUDA thread block computes:
C(blockDim.x, blockDim.y) += A(blockDim.x, k) * B(k, blockDim.y)

~2*m*n*k/bDim.x global loads per kernel



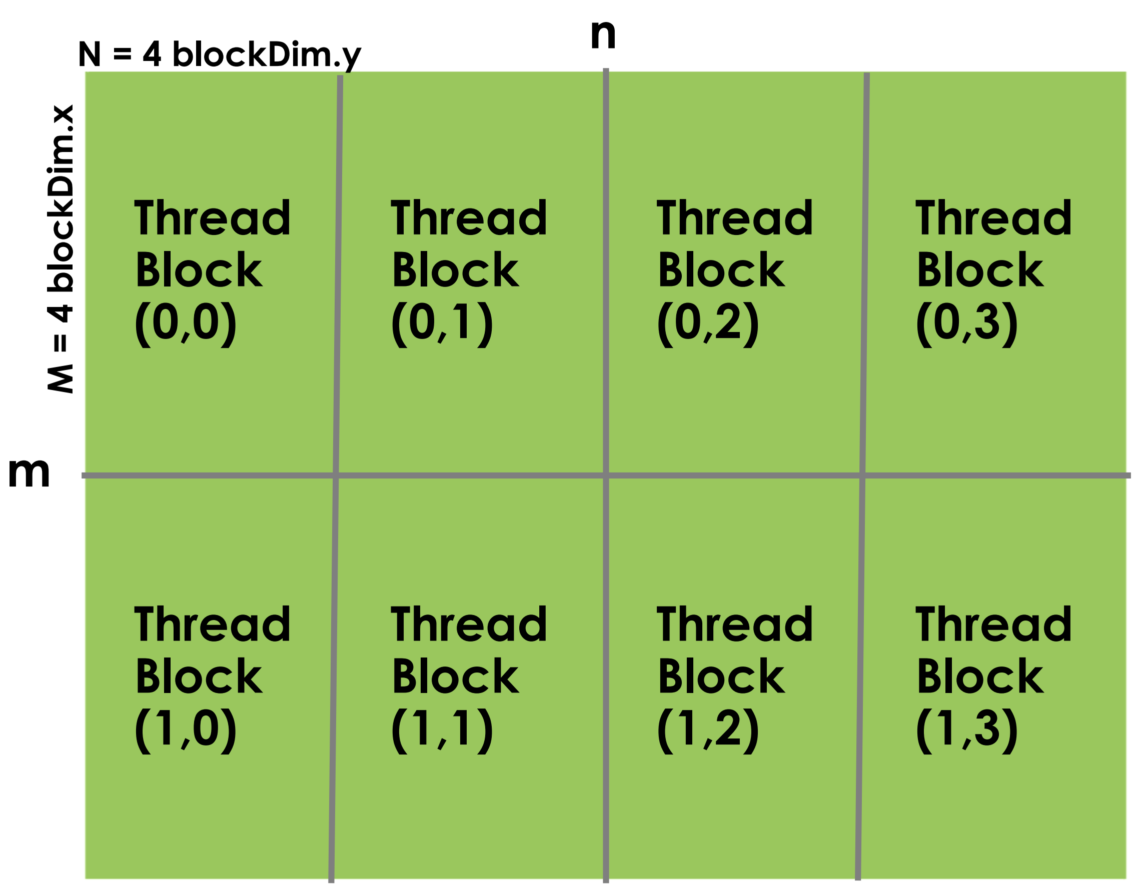
Matrix C(m,n)



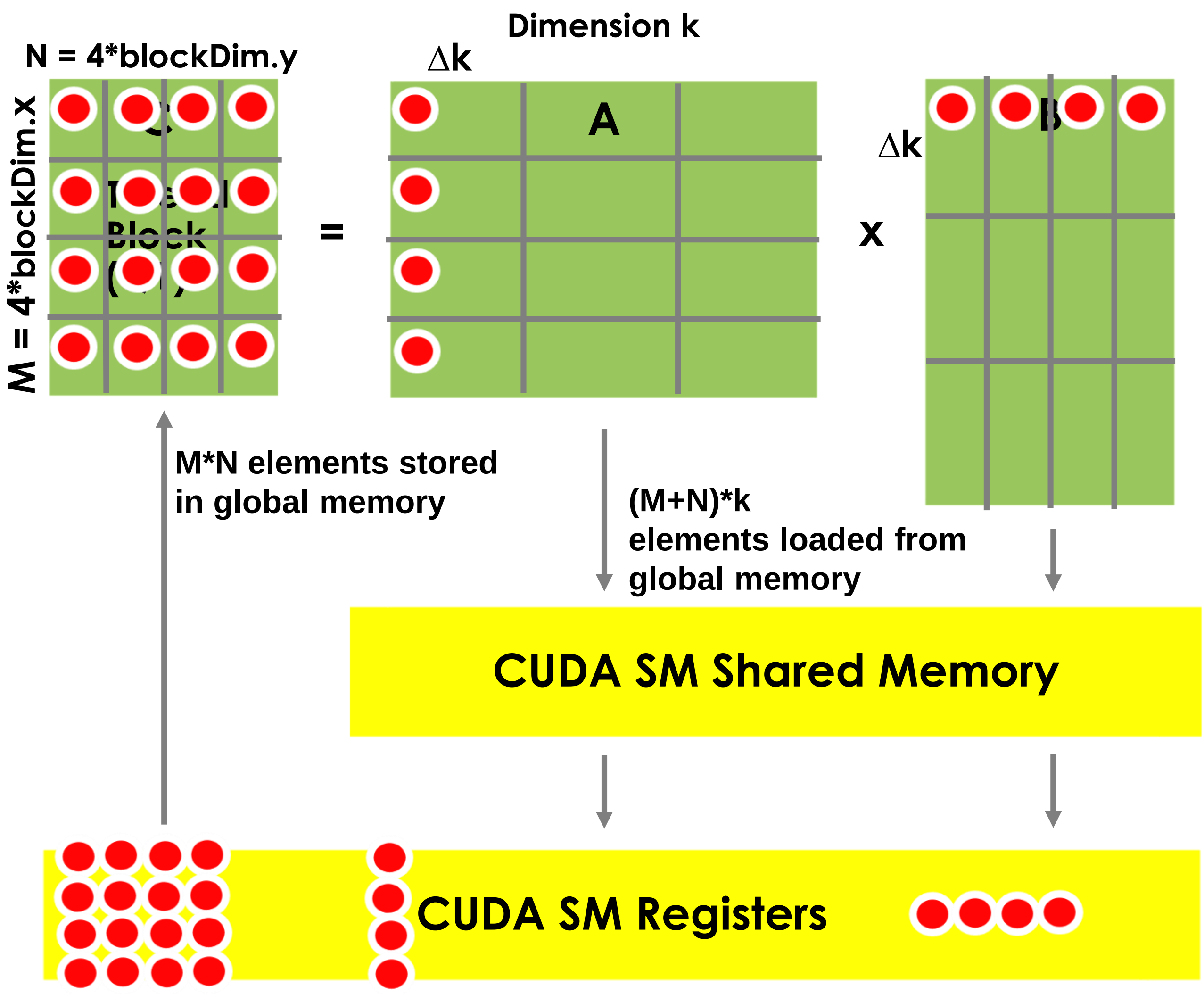
CUDA BLA Library: +Registers GEMM (algorithm 2)

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)

~2*m*n*k/M global loads per kernel



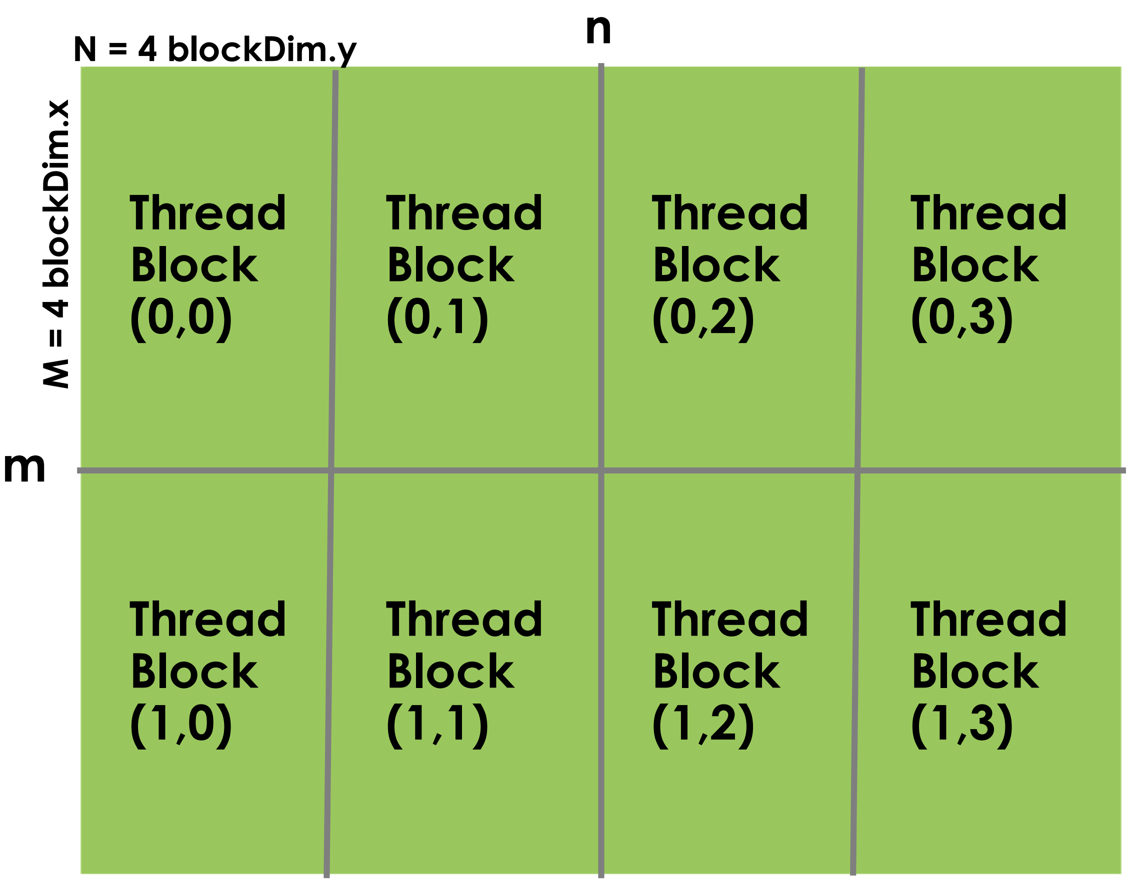
Matrix C(m,n)



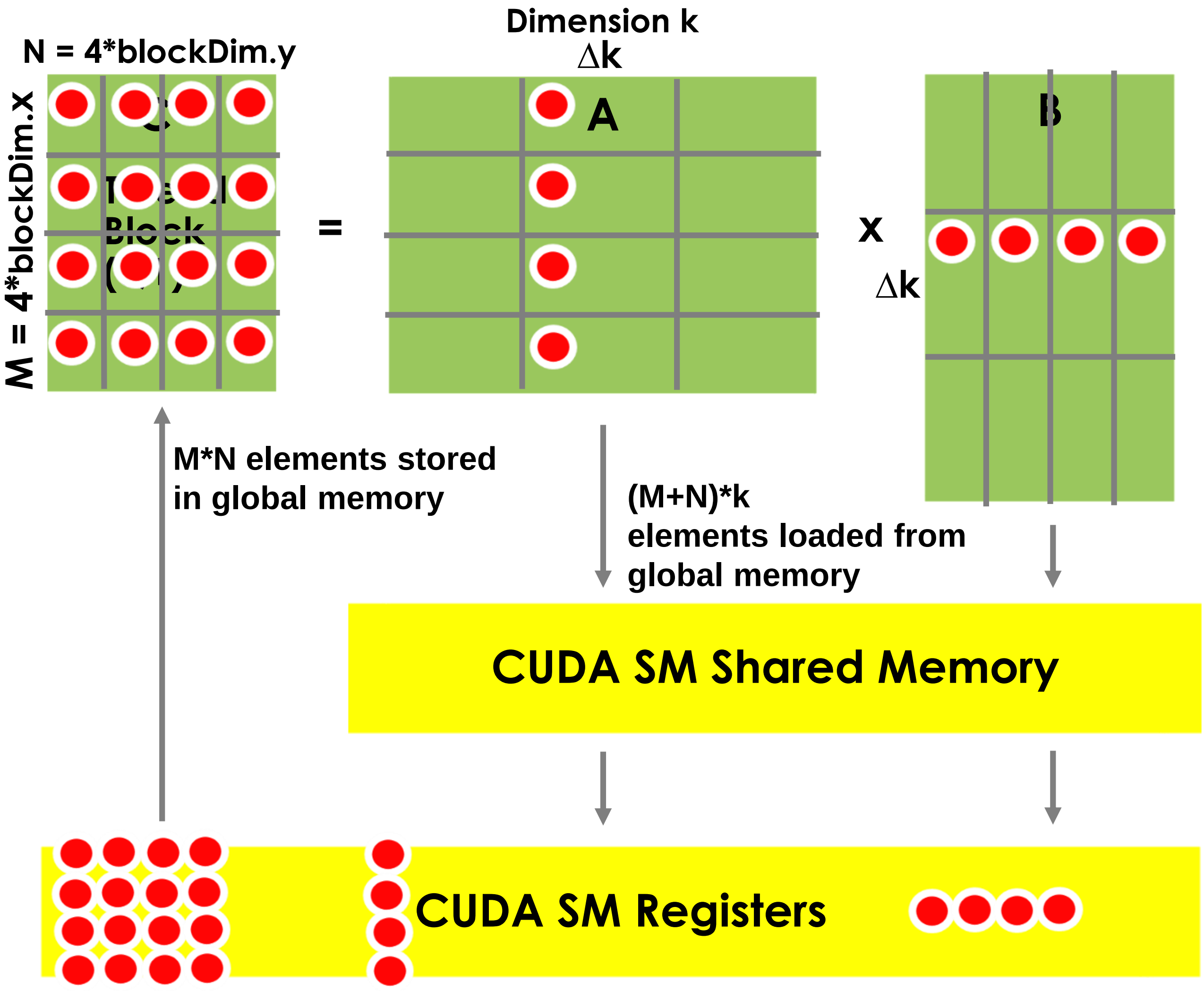
CUDA BLA Library: +Registers GEMM (algorithm 2)

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)

~2*m*n*k/M global loads per kernel



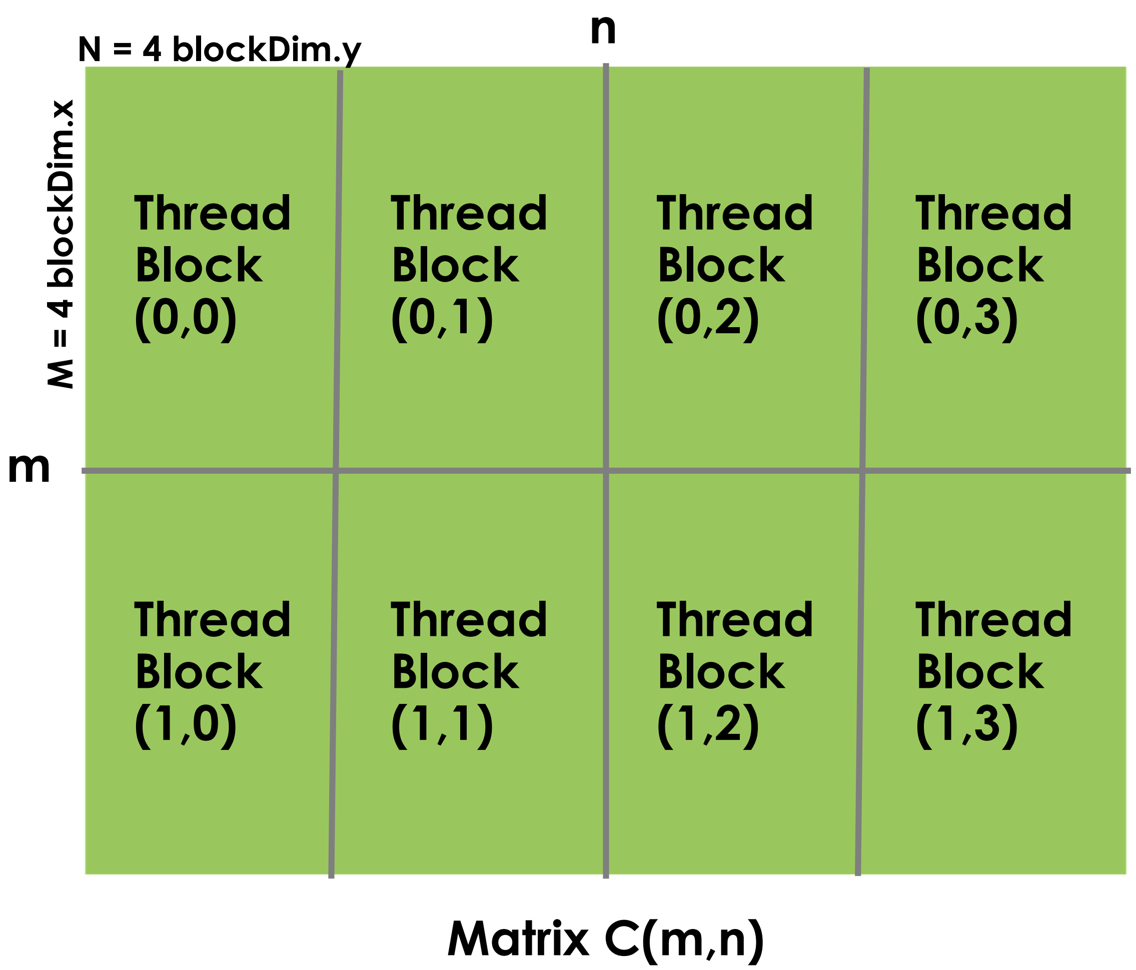
Matrix C(m,n)

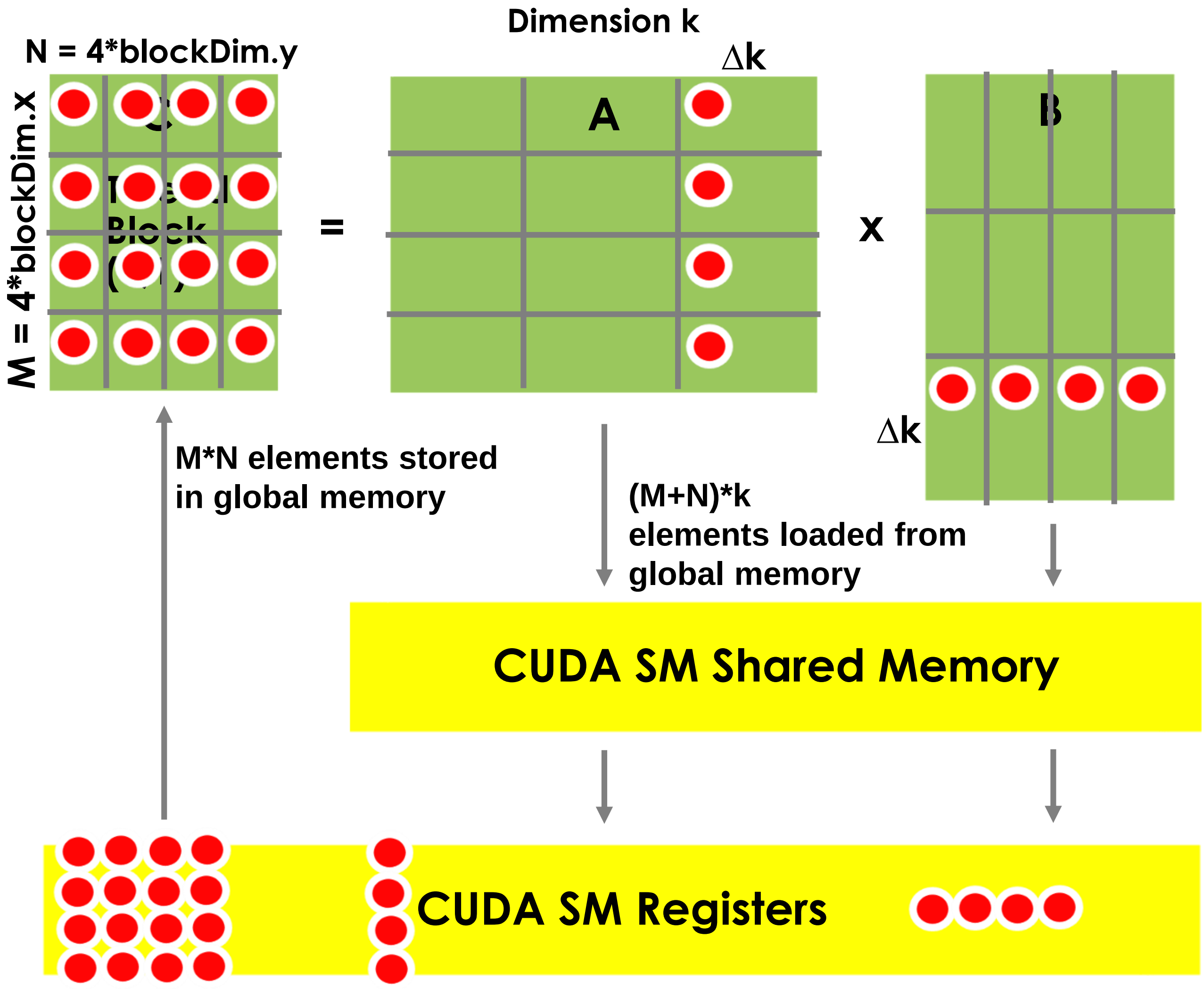


CUDA BLA Library: +Registers GEMM (algorithm 2)

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)

~2*m*n*k/M global loads per kernel

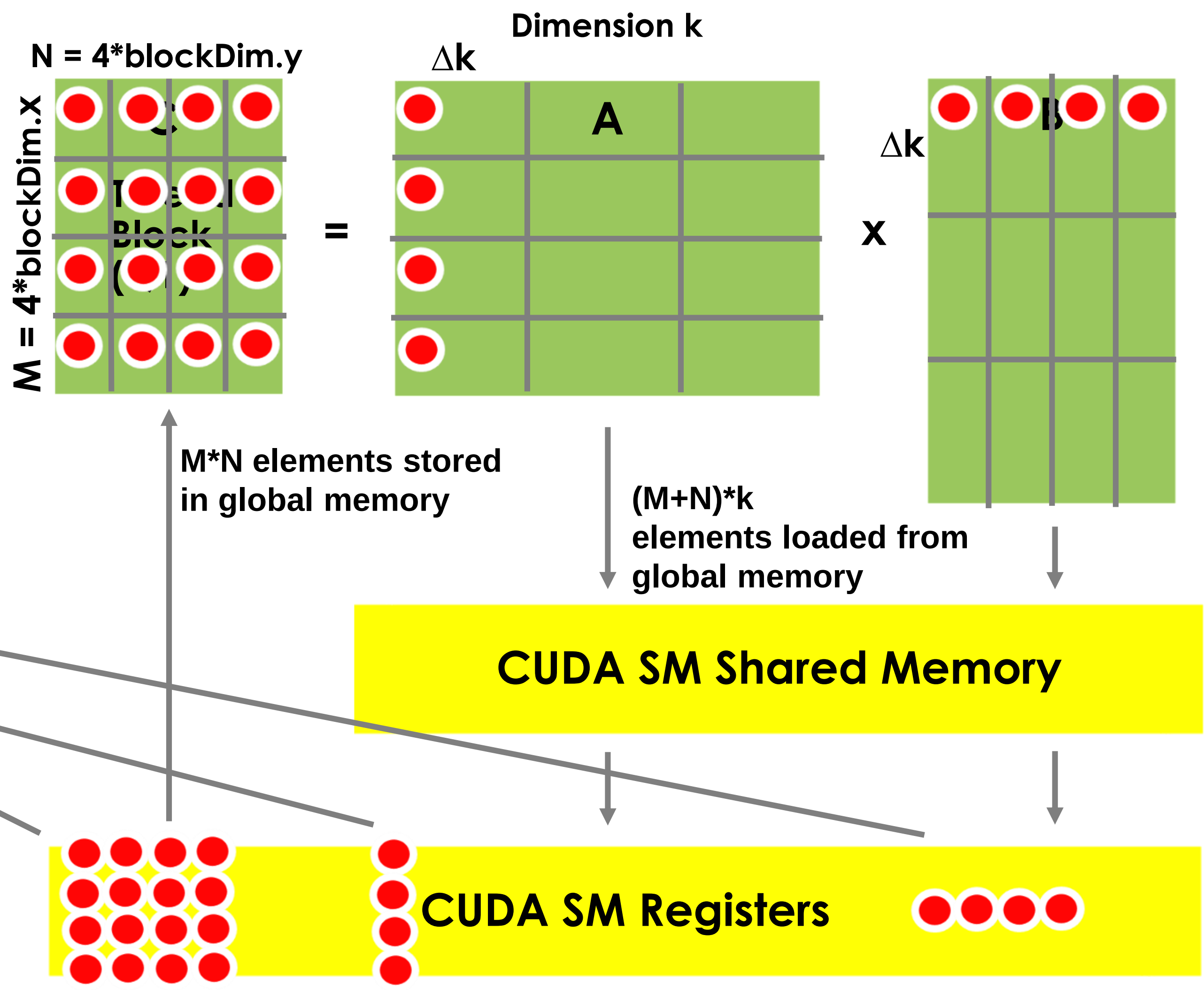




CUDA BLA Library: +Registers GEMM (algorithm 2)

```
template <typename T, int TILE_EXT_N, int TILE_EXT_M, int TILE_EXT_K>
__global__ void gpu_gemm_sh_reg_nn(int m, int n, int k,
                                   T * __restrict__ dest,        //inout: pointer to C matrix data
                                   const T * __restrict__ left,  //in: pointer to A matrix data
                                   const T * __restrict__ right) //in: pointer to B matrix data
{
 using int_t = int; //either int or size_t
 __shared__ T lbuf[TILE_EXT_K][TILE_EXT_M], rbuf[TILE_EXT_N][TILE_EXT_K];

 for(int_t n_pos = blockIdx.y*TILE_EXT_N; n_pos < n; n_pos += gridDim.y*TILE_EXT_N){
  int_t n_end = n_pos + TILE_EXT_N; if(n_end > n) n_end = n;

  for(int_t m_pos = blockIdx.x*TILE_EXT_M; m_pos < m; m_pos += gridDim.x*TILE_EXT_M){
   int_t m_end = m_pos + TILE_EXT_M; if(m_end > m) m_end = m;

   if((m_end - m_pos == TILE_EXT_M) && (n_end - n_pos == TILE_EXT_N)){

    //Initialize registers to zero:
    T dreg[4][4] = {static_cast<T>(0.0)};
    T rreg[4] = {static_cast<T>(0.0)};
    T lreg[4] = {static_cast<T>(0.0)};
```

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)

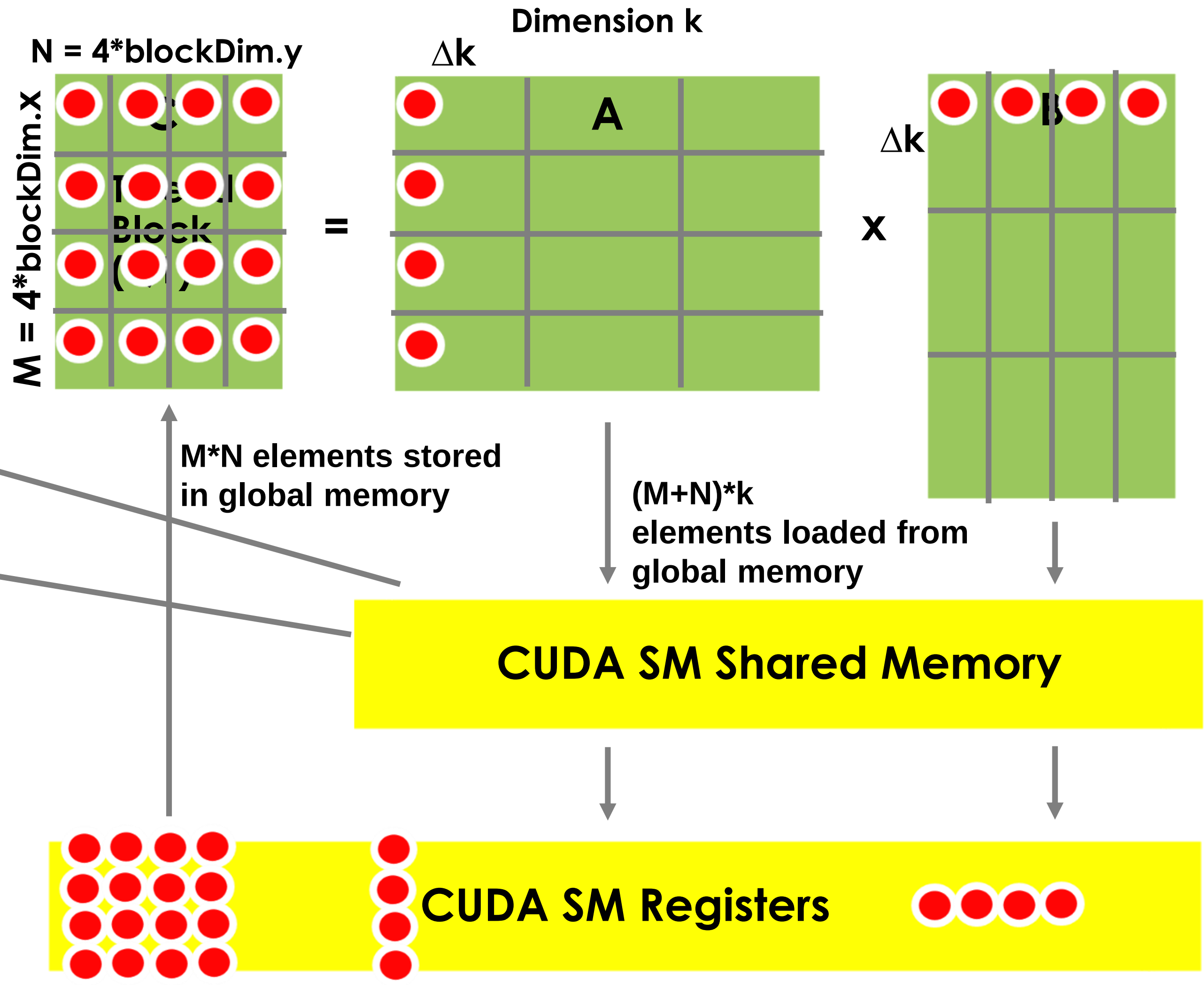


CUDA BLA Library: +Registers GEMM (algorithm 2)

```
for(int_t k_pos = 0; k_pos < k; k_pos += TILE_EXT_K){ //k_pos is the position of the CUDA thread along the K dimension
    int_t k_end = k_pos + TILE_EXT_K; if(k_end > k) k_end = k;

    //Load a tile of matrix A(m_pos:TILE_EXT_M, k_pos:TILE_EXT_K):
    for(int_t m_loc = m_pos + threadIdx.x; m_loc < m_end; m_loc += blockDim.x){
     for(int_t k_loc = k_pos + threadIdx.y; k_loc < k_end; k_loc += blockDim.y){
      lbuf[k_loc - k_pos][m_loc - m_pos] = left[k_loc*m + m_loc];
     }
    }

    //Load a tile of matrix B(k_pos:TILE_EXT_K, n_pos:TILE_EXT_N):
    for(int_t n_loc = n_pos + threadIdx.y; n_loc < n_end; n_loc += blockDim.y){
     for(int_t k_loc = k_pos + threadIdx.x; k_loc < k_end; k_loc += blockDim.x){
      rbuf[n_loc - n_pos][k_loc - k_pos] = right[n_loc*k + k_loc];
     }
    }
    __syncthreads();
```

Loop: M > blockDim.x;
Loop: N > blockDim.y;

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)



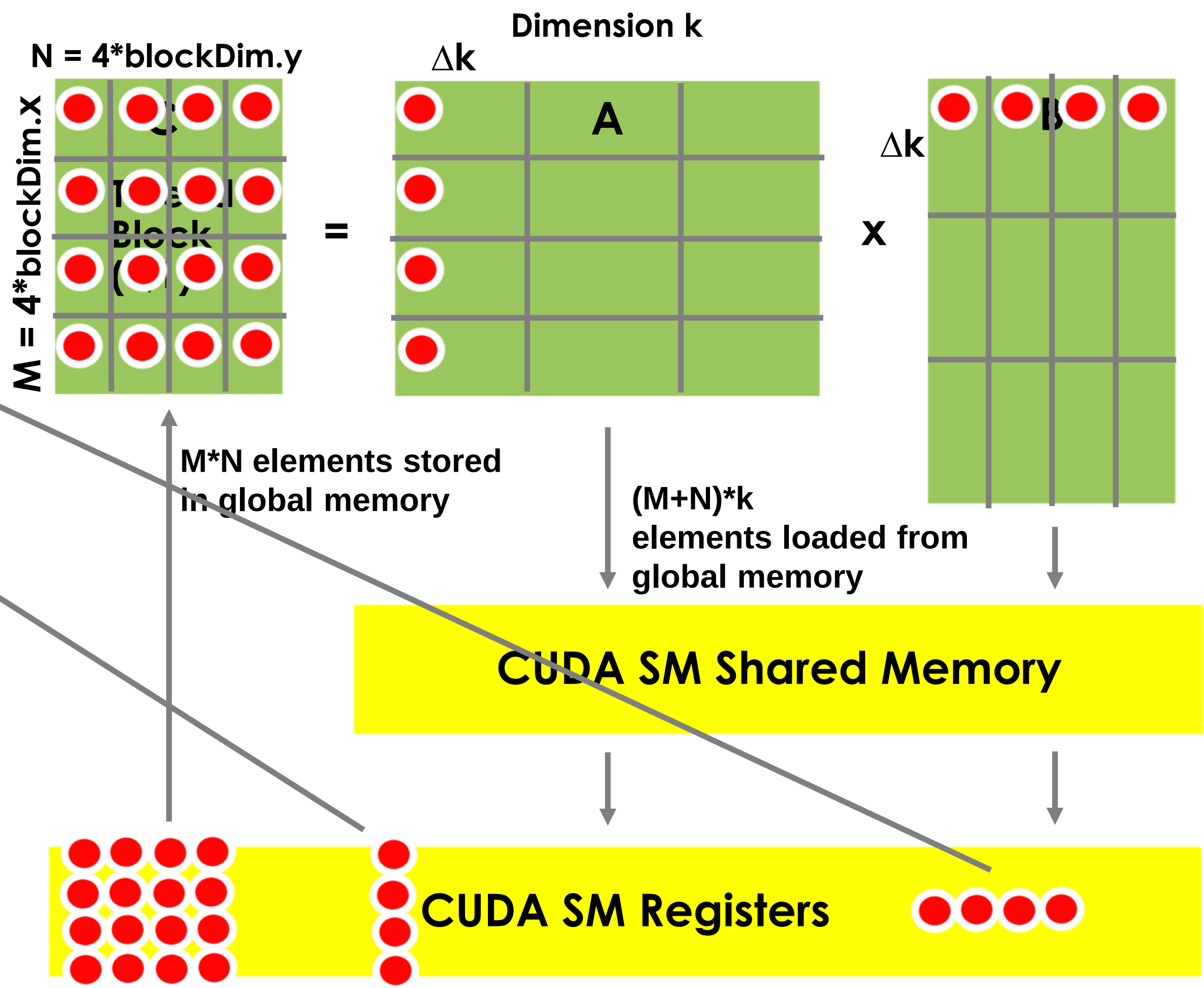
CUDA BLA Library: +Registers GEMM (algorithm 2)

```
//Multiply two loaded tiles to produce a tile of matrix
     if(k_end - k_pos == TILE_EXT_K){
#pragma unroll
      for(int_t l = 0; l < TILE_EXT_K; ++l){
#pragma unroll
       for(int_t j = 0; j < 4; ++j) rreg[j] = rbuf[threadIdx.y + blockDim.y*j][l];
#pragma unroll
       for(int_t j = 0; j < 4; ++j) lreg[j] = lbuf[l][threadIdx.x + blockDim.x*j];
#pragma unroll
       for(int_t j = 0; j < 4; ++j){
#pragma unroll
        for(int_t i = 0; i < 4; ++i){
         dreg[j][i] += lreg[i] * rreg[j];
        }
       }
      }
     }else{
      for(int_t l = 0; l < (k_end - k_pos); ++l){
#pragma unroll
       for(int_t j = 0; j < 4; ++j) rreg[j] = rbuf[threadIdx.y + blockDim.y*j][l];
#pragma unroll
       for(int_t j = 0; j < 4; ++j) lreg[j] = lbuf[l][threadIdx.x + blockDim.x*j];
#pragma unroll
       for(int_t j = 0; j < 4; ++j){
#pragma unroll
        for(int_t i = 0; i < 4; ++i){
         dreg[j][i] += lreg[i] * rreg[j];
        }
       }
      }
     }
     __syncthreads();
```

Load registers

Load registers

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)

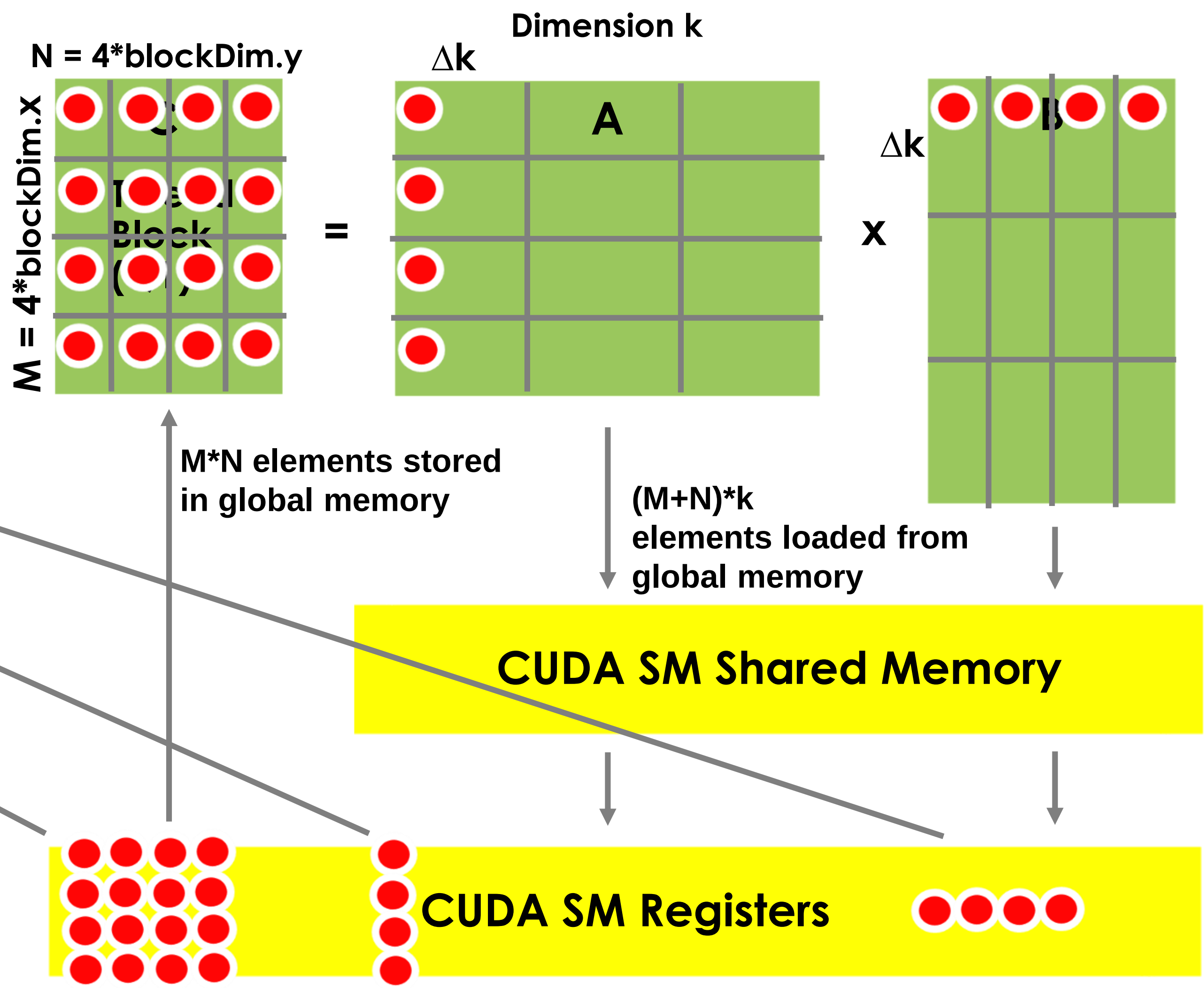


CUDA BLA Library: +Registers GEMM (algorithm 2)

```
//Multiply two loaded tiles to produce a tile of matrix
     if(k_end - k_pos == TILE_EXT_K){
#pragma unroll
      for(int_t l = 0; l < TILE_EXT_K; ++l){
#pragma unroll
       for(int_t j = 0; j < 4; ++j) rreg[j] = rbuf[threadIdx.y + blockDim.y*j][l];
#pragma unroll
       for(int_t j = 0; j < 4; ++j) lreg[j] = lbuf[l][threadIdx.x + blockDim.x*j];
#pragma unroll
       for(int_t j = 0; j < 4; ++j){
#pragma unroll
        for(int_t i = 0; i < 4; ++i){
         dreg[j][i] += lreg[i] * rreg[j];
        }
       }
      }
     }else{
      for(int_t l = 0; l < (k_end - k_pos); ++l){
#pragma unroll
       for(int_t j = 0; j < 4; ++j) rreg[j] = rbuf[threadIdx.y + blockDim.y*j][l];
#pragma unroll
       for(int_t j = 0; j < 4; ++j) lreg[j] = lbuf[l][threadIdx.x + blockDim.x*j];
#pragma unroll
       for(int_t j = 0; j < 4; ++j){
#pragma unroll
        for(int_t i = 0; i < 4; ++i){
         dreg[j][i] += lreg[i] * rreg[j];
        }
       }
      }
     }
     __syncthreads();
```

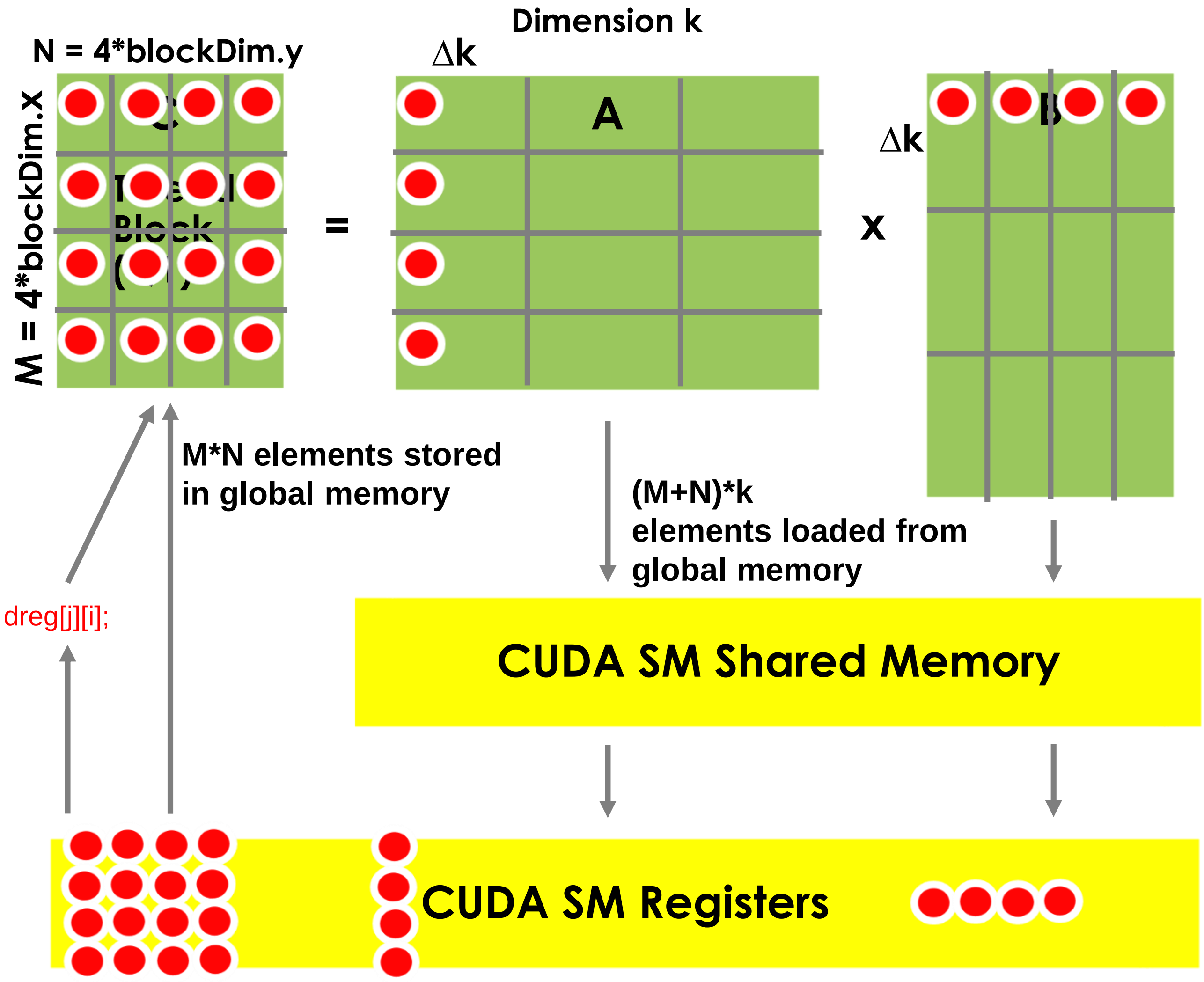
4x4 matrix outer product from registers by each thread

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)



CUDA BLA Library: +Registers GEMM (algorithm 2)

Each CUDA thread block computes:
C(M = 4*blockDim.x, N = 4*blockDim.y) += A(M, k) * B(k, N)




```
//Store elements of the C matrix in global memory:
#pragma unroll
   for(int_t j = 0; j < 4; ++j){
#pragma unroll
    for(int_t i = 0; i < 4; ++i){
     dest[(n_pos + threadIdx.y + blockDim.y*j)*m + (m_pos + threadIdx.x + blockDim.x*i)] += dreg[j][i];
    }
   }
```

Upload (4x4) registers to global memory

CUDA BLA Library: Implement Your GEMM Algorithms

- You will work inside **bla_lib.cu** source file directly with CUDA GEMM kernels
- Matrix multiplication {**false**,**false**} case (already implemented):
  - **C(m,n) += A(m,k) * B(k,n)**
  - **CUDA kernels: gpu_gemm_nn, gpu_gemm_sh_nn, gpu_gemm_sh_reg_nn**
- Matrix multiplication {**false**,**true**} case (your exercise):
  - **C(m,n) += A(m,k) * B(n,k)**
  - **CUDA kernels: gpu_gemm_nt, gpu_gemm_sh_nt, gpu_gemm_sh_reg_nt**
- Matrix multiplication {**true**,**false**} case (your exercise):
  - **C(m,n) += A(k,m) * B(k,n)**
  - **CUDA kernels: gpu_gemm_tn, gpu_gemm_sh_tn, gpu_gemm_sh_reg_tn**
- Matrix multiplication {**true**,**true**} case (your exercise):
  - **C(m,n) += A(k,m) * B(n,k)**
  - **CUDA kernels: gpu_gemm_tt, gpu_gemm_sh_tt, gpu_gemm_sh_reg_tt**

CUDA BLA Library Implementation Benchmark

Testing your BLA GPU kernel implementation (main.cpp: use_bla() function):

```
for(int repeat = 0; repeat < 2; ++repeat){ //repeat experiment twice
 C.zeroBody(0); //set matrix C body to zero on GPU#0
 bla::reset_gemm_algorithm(0); //choose your algorithm: {0,1,2,7}
 std::cout << "Performing matrix multiplication C+=A*B with BLA GEMM brute-force ... ";
 double tms = bla::time_sys_sec(); //timer start
 C.multiplyAdd(false,false,A,B,0); //default case {false,false}: You goal is {false,true}, {true,false}, {true,true}
 double tmf = bla::time_sys_sec(); //timer stop
 std::cout << "Done: Time = " << tmf-tms << " s: Gflop/s = " << flops/(tmf-tms)/1e9 << std::endl;
 //Check correctness on GPU#0:
 C.add(D,1.0f,0); //adding the correct result with a minus sign (matrix D) should give you zero matrix
 auto norm_diff = C.computeNorm(0); //check its norm
 std::cout << "Norm of the matrix C deviation from correct = " << norm_diff << std::endl;
 if(std::abs(norm_diff) > 1e-7){ //report if norm is not zero enough
  std::cout << "#FATAL: Matrix C is incorrect, fix your GPU kernel implementation!" << std::endl;
  std::exit(1);
 }
}
```

**This benchmark is run for all available BLA GEMM algorithms: 0, 1, 2, 7 for the {false,false} case.
Your goal is to implement and run other cases: {false,true}, {true,false}, {true,true}!
Use TEST_BLA_GEMM_BRUTE, TEST_BLA_GEMM_SHARED, TEST_BLA_GEMM_REGISTER switches
in main.cpp:use_bla() to turn on/off specific GEMM algorithms (0,1,2, respectively).**