Flatten 2D matrix

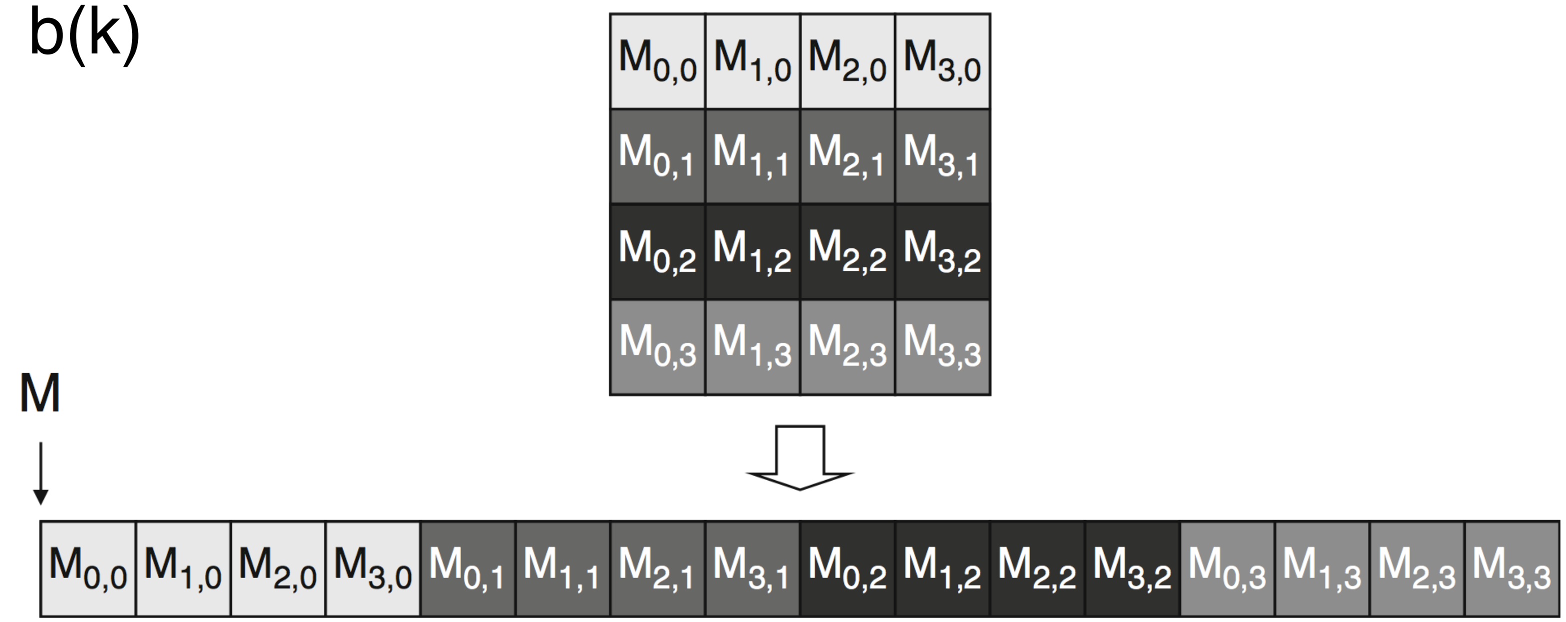
2D matrix to 1D array and back again

C++ uses **row major order**: n x m, which are the number of rows and columns
also called the height and the width

a(i,j) can be flatten to 1D array b(k)

where k= i*m + j

```
for (int i=0; i < n; i++)
  { for (int j =0; j< m; j++)
        b[i*m+j] = a[i][j];
   }
```



To get back to 2D matrix from A(k)

```
i= k/m;  //rounding down
j = k -(i*m);              or j= k %m
```

(where modulus gives remainder)

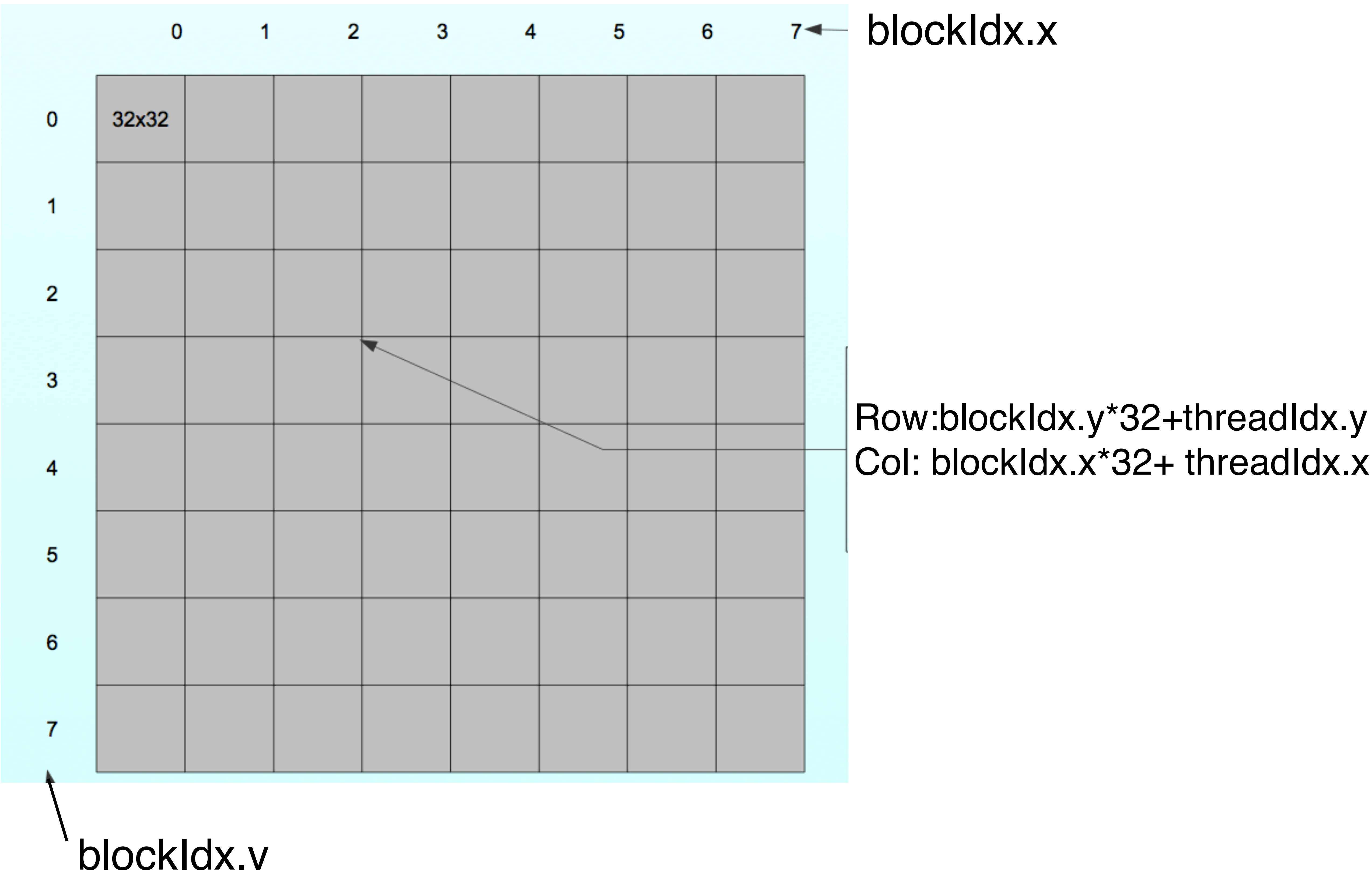
Matrix Copy

Problem: copy matrix a(n,m) into b(n,m). Here **n=m=256;** multiple of 32

Solution: matcopy.cu with flattened matrices

```
__global__ void copymat(float * input, float * output)
{
 int x = blockIdx.x * blockDim.x + threadIdx.x;        //using 2-D location in matrix
 int y = blockIdx.y * blockDim.y + threadIdx.y;

 int length = gridDim.x*blockDim.x;    //width of a row
 output[y*length+x] = input[y*length+x];
}

int main(){
  dim3 block(32,32);               //NOTE:  can not use block(32,32,0)
  dim3 gridDim(8,8);               //8 x 32 = 256 (perfect fit)
  copymat<<<gridDim,block>>>(d_input, d_output);
}
```

Matrix: gridDim(8,8)



Matrix Copy

Instead of an 8x8 grid of 32 x 32 blocks, use 32x8 blocks four times in y direction; **grid stride.**

```
dim3 block(32,32);
dim3 gridDim(2,8);
copymat<<<gridDim,block>>>(d_input, d_output);
```

What is the kernel?

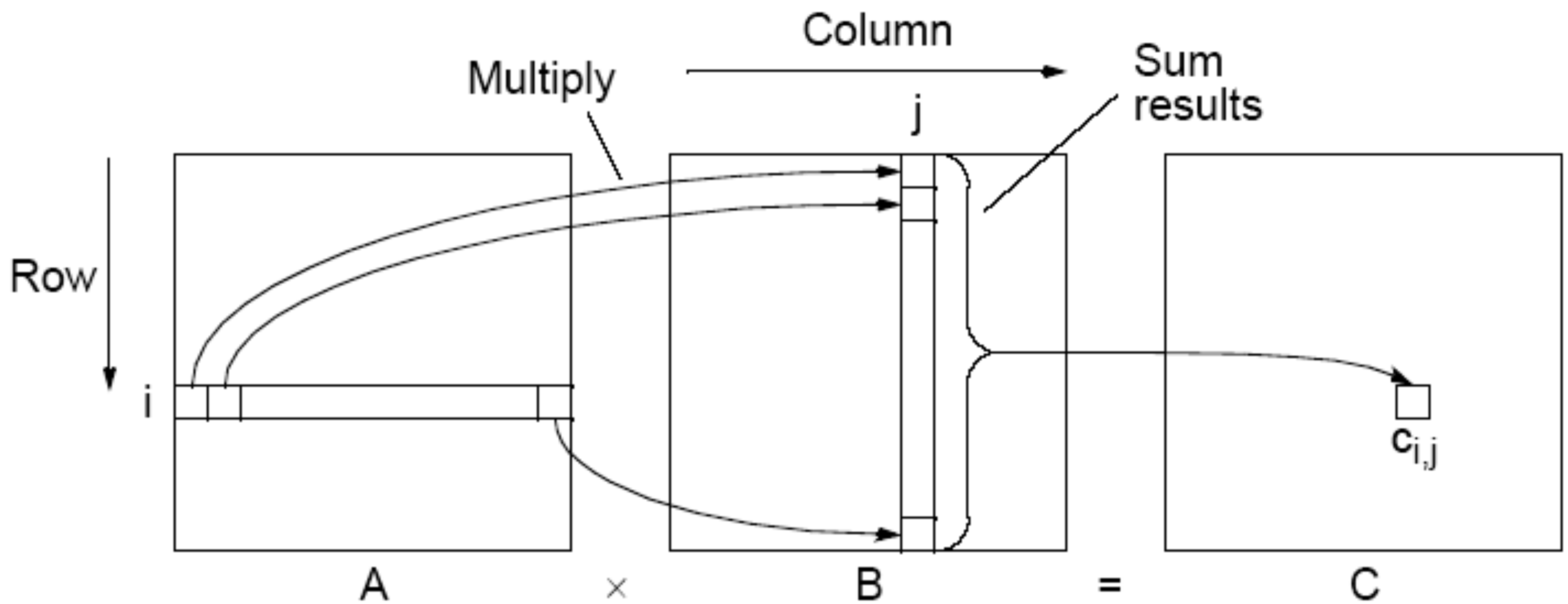
Why do it? Thread reuse—it is actually faster.

Matrix Copy by 4 (using grid stride in y)

```
__global__ void copymat(float * input, float * output)
{
 int x = blockIdx.x * blockDim.x + threadIdx.x;
 int y = blockIdx.y * blockDim.y + threadIdx.y;


 int length = gridDim.x*blockDim.x;

 for (int j=0; j < 4*gridDim.y*blockDim.y; j+=gridDim.y*blockDim.y)
     output[(y+j)*length+x] = input[(y+j)*length+x];

}
```

Matrix Multiply

Two square matrices: N x N



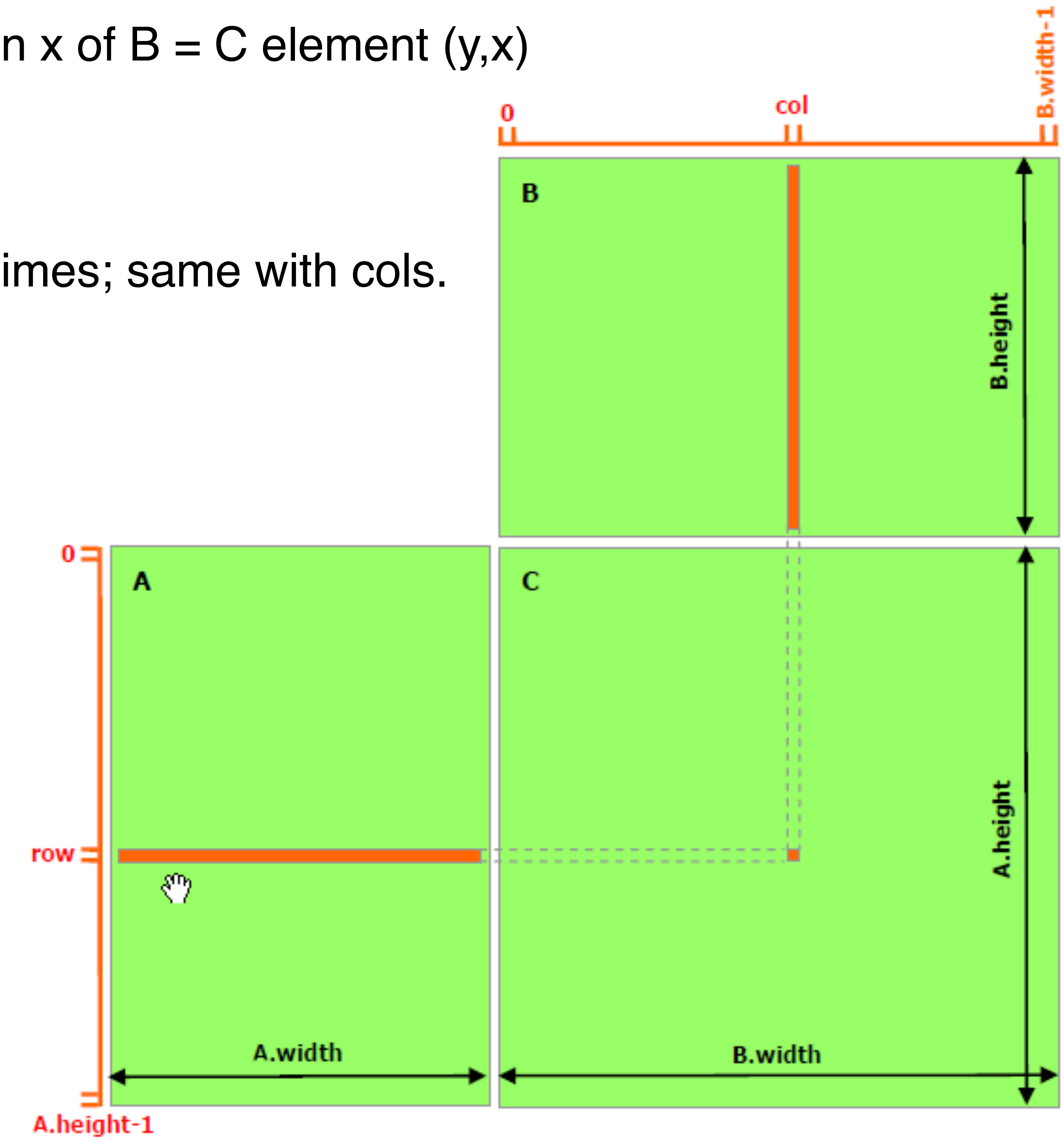
$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} * b_{k,j}$$

Square Matrix Multiply

Simple matrix multiply with square matrices: C=A*B with size WIDTH*WIDTH

Procedure: row y of A times column x of B = C element (y,x)

Note that A rows are read WIDTH times; same with cols.



C++ Code

```
for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; k < N; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
```

Requires n^3 multiplications and n^3 additions

CUDA with flattened **a, b,** and **c**

one thread for each $c_{i,j}$

```
#define WIDTH 256

__global__  void mult_mat(float * a, float * b, float * c)
{
 int x = blockIdx.x * blockDim.x + threadIdx.x;     //location in c
 int y = blockIdx.y * blockDim.y + threadIdx.y;
 int idx = y*WIDTH + x;

 if (idx <WIDTH*WIDTH)
   {
     float Cvalue=0.0;
     for (int i=0; i<WIDTH; i++)
      {
        Cvalue  +=  a[y*WIDTH + i] * b[i*WIDTH+x];
      }
      c[y*WIDTH+x]=Cvalue;
   }
}
```

Using Shared Memory

Option: Put all of matrix a and b into shared memory

  Problem: only one block possible as memory only shared in block

Option: Put all of a row into a shared memory; better than before

  Problem: reuse of all the columns

Option: Use tiles.

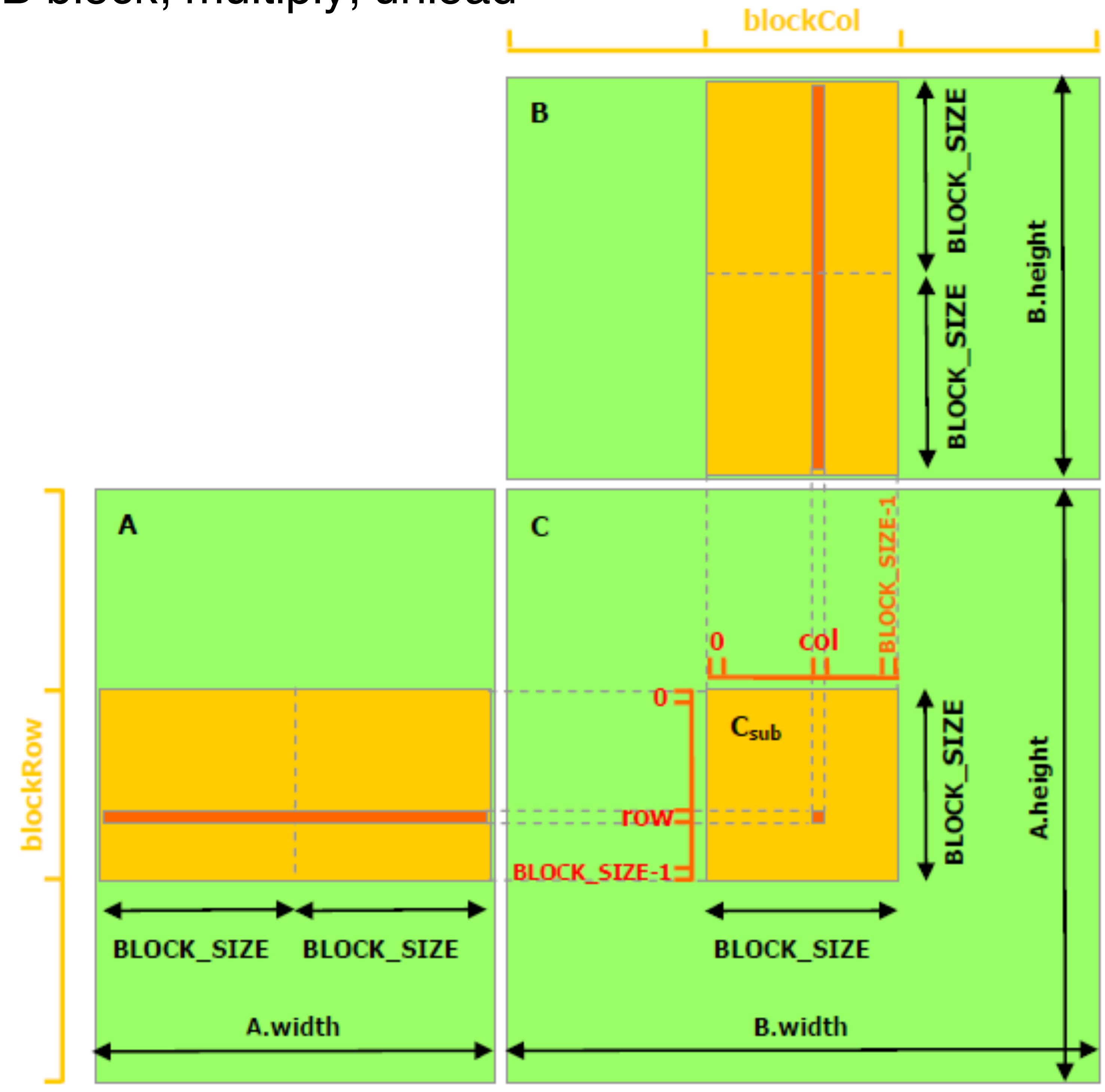
Tiled Matrix Multiply

Use blocks:

load first A block and the first B block into shared memory; multiply; unload shared memory

load second A block and the second B block; multiply; unload

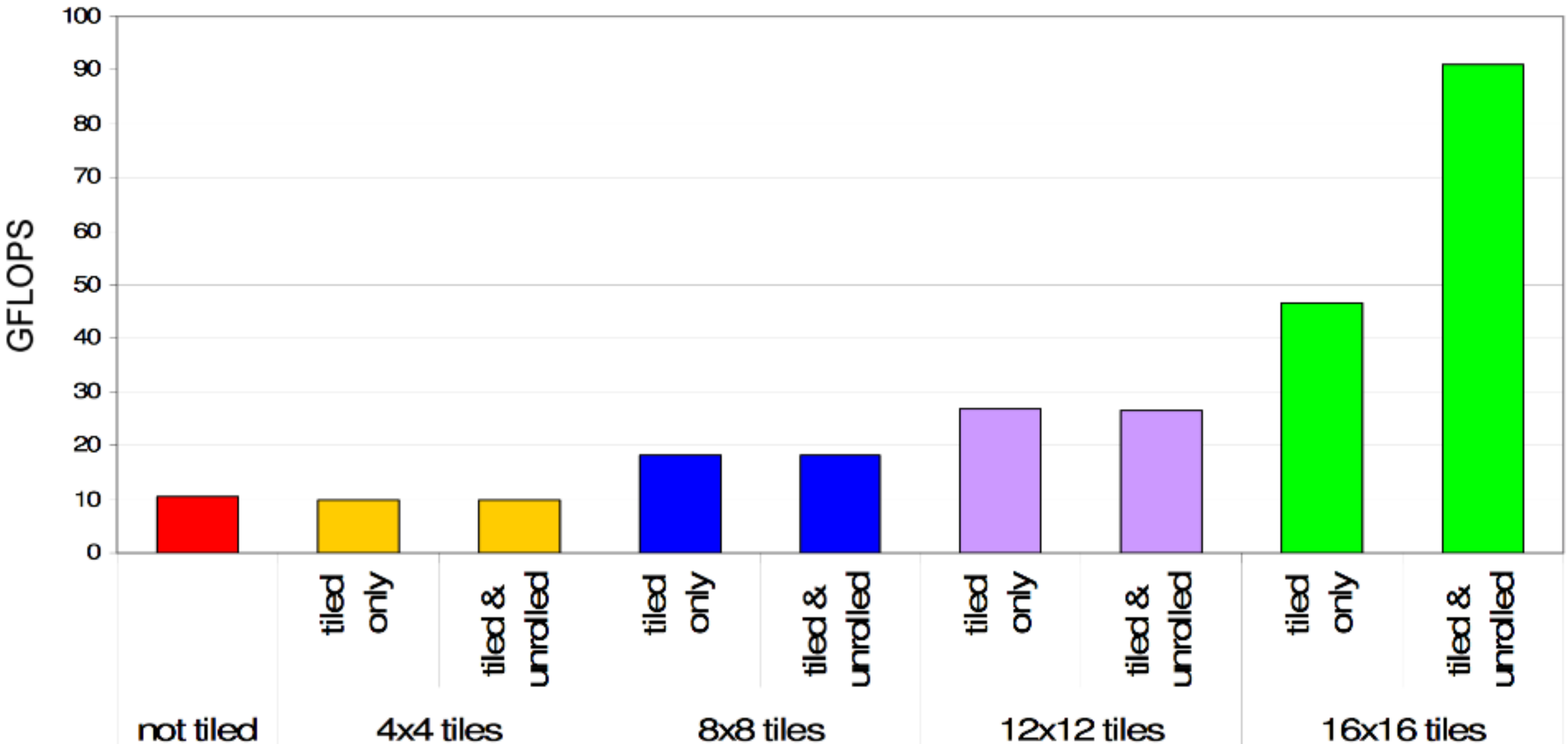
add sums



Tiled Matrix Multiply

```
__global__ void mat_Mul(float* A, float* B, float* C, int ARows, int ACols, int BRows, int BCols, int CRows, int CCols) {

    float CValue = 0;

    int Row = blockIdx.y*TILE_DIM + threadIdx.y;
    int Col = blockIdx.x*TILE_DIM + threadIdx.x;

    __shared__ float As[TILE_DIM][TILE_DIM];
    __shared__ float Bs[TILE_DIM][TILE_DIM];

    for (int k = 0; k < (TILE_DIM + ACols - 1)/TILE_DIM; k++) {

        if (k*TILE_DIM + threadIdx.x < ACols && Row < ARows)   As[threadIdx.y][threadIdx.x] = A[Row*ACols + k*TILE_DIM + 
threadIdx.x];
        else                                                   As[threadIdx.y][threadIdx.x] = 0.0;

        if (k*TILE_DIM + threadIdx.y < BRows && Col < BCols)   Bs[threadIdx.y][threadIdx.x] = B[(k*TILE_DIM + 
threadIdx.y)*BCols + Col];
        else                                                   Bs[threadIdx.y][threadIdx.x] = 0.0;

        __syncthreads();

        for (int n = 0; n < TILE_DIM; ++n) CValue += As[threadIdx.y][n] * Bs[n][threadIdx.x];

        __syncthreads();
    }

    if (Row < CRows && Col < CCols) C[((blockIdx.y * blockDim.y + threadIdx.y)*CCols)+(blockIdx.x*blockDim.x)
+threadIdx.x]=CValue;
}
```

Timing



http://www.umiacs.umd.edu/~ramani/cmsc828e_gpusci/Lecture5.pdf

BLAS library

Basic Linear Algebra Subprograms

Quick Reference Guide: http://www.netlib.org/blas/blasqr.pdf

http://www.netlib.org/blas/faq.html#8

Matrix Multiply with CUBLAS

CUBLAS is the CUDA implementation of BLAS
(Basic Linear Algebra Subprograms)
http://docs.nvidia.com/cuda/cublas/

Consider scalars α,β, vectors x, y, and matrices A, B, C.

Three levels of CUBLAS programs:

Level 1 BLAS 1: Scalar and vector based functions

$y \rightarrow \alpha x + y$

Level 2 BLAS 2: Matrix-vector operations

$y \rightarrow \alpha A x + \beta y$

Level 3 BLAS 3: Matrix-matrix operations

$C \rightarrow \alpha AB + \beta C$

BLAS

Most routines have the following in their names:

s	single precision
d	double precision
c	complex single precision
z	complex double precision

Matrix types:

general	ge
symmetric	sy
Hermitian	he
triangular	tr

Example: dgemm is a double precision general matrix multiply (Level 3)

Legacy FORTRAN

NOTE:

BLAS originally written in FORTRAN. Arrays stored column major in FORTRAN, while they are row major in C++. Further, indexing starts at 1, rather than 0.

Need to have arrays in the correct format.

All arrays are flattened. From column major linear array to C++ 2D array:

1	1	2	3	5	8	13	21	34	55	89	144

encodes matrix B,

$$\begin{bmatrix} 1 & 5 & 32 \\ 1 & 8 & 55 \\ 2 & 13 & 89 \\ 3 & 21 & 144 \end{bmatrix}$$

#define IDX2C(i,j,ld) (((j)*(ld))+(i)) takes 2D array indices (i,j) into column major index where ld is leading dimension (# rows)

Check out IDX2C define for column major arrays

```
/* simple examination of IDX2C
*/
#include <iostream>
#define IDX2C(i,j,ld) ((j)*(ld) + (i))
using namespace std;

/*
my array is 1  2  3  4
            5  6  7  8
            9 10 11 12
*/
```

Note: ld = 3
leading dimension is the number of elements to count in linear array to get to the next column.

```
int main(){
  int a[12];
  cout<<endl <<"column major array\n";
  // define column major array using IDX2C
  // this is the way to use C++ for-loops to get column major linear array
  for (int i = 0; i<3; i++) //rows
   {for (int j = 0; j<4; j++)  //columns
      { a[IDX2C(i,j,3)] =i*4+j+1;
       cout<<"a["<<IDX2C(i,j,3)<<"]=" <<i*4+j+1<<",";
       }
       cout<<"\n";
   }
   cout<<"\n encoded linear array: "<<endl;
   for (int i=0; i<12; i++)
    cout<<a[i]<<", ";

    cout<<endl<<endl;
   //print out column major linear array in 2D form.
   for (int i = 0; i<3; i++)
     {for(int j =0; j<4; j++)
        {cout<<a[IDX2C(i,j,3)]<<“, “;}
        cout << endl;
     }
}
```

Output: testIDX2C.cpp

```
/*
my array is 1  2  3  4
            5  6  7  8
            9 10 11 12
*/
```

```
column major array
a[0]=1,a[3]=2,a[6]=3,a[9]=4,
a[1]=5,a[4]=6,a[7]=7,a[10]=8,
a[2]=9,a[5]=10,a[8]=11,a[11]=12,

 encoded linear array:
1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12,

1, 2, 3, 4,
5, 6, 7, 8,
9, 10, 11, 12, _
```

Matrix Multiply with CUBLAS (kernel)

```
// Multiply the arrays A and B on GPU and save the result in C
// C(m,n) = A(m,k) * B(k,n)
void gpu_blas_mmul(const float *A, const float *B, float *C, const int m, const int k,
const int n) {
	int lda=m,ldb=k,ldc=m;
	const float alf = 1;
	const float bet = 0;
	const float *alpha = &alf;
	const float *beta = &bet;

	// Create a handle for CUBLAS
	cublasHandle_t handle;
	cublasCreate(&handle);

	// Do the actual multiplication
	cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, alpha, A, lda,
B, ldb, beta, C, ldc);

	// Destroy the handle
	cublasDestroy(handle);
}
```

From CUDA Samples

nvcc matrixMulCUBLAS.cpp -I/Developer/NVIDIA/CUDA-6.5/samples/common/inc -lcublas