COMP 605: Introduction to Parallel Computing
Lecture : CUDA Thread Parallelism

Mary Thomas

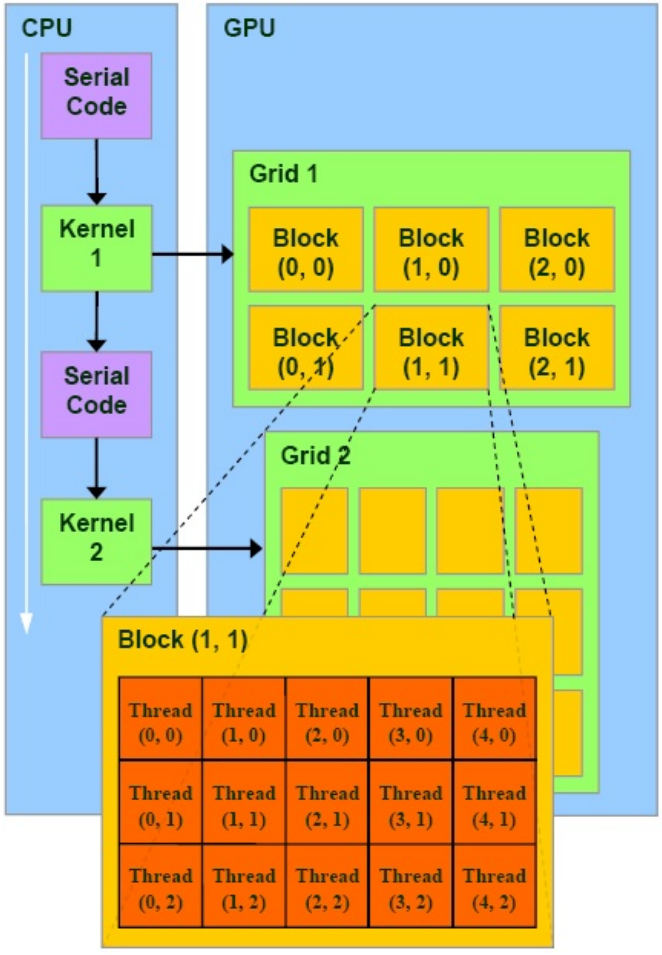
Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 04/25/17
Last Update: 04/25/17

Table of Contents

Recall: Defining GPU Threads and Blocks

- Looking at Device: Nvidia Tesla C1060
- Kernels run on GPU threads
- Grid: organized as 2D array of blocks:
  - Maximum sizes of each dimension: $[gridDim.x \times gridDim.y \times gridDim.z] = (65,536 \times 65,536 \times 1)$ blocks
- Block: $3D$ collection of threads
  - Max threads per block: 512
  - Max thread dimensions: (512, 512, 64) $[blockDim.x * blockDim.y * blockDim.z]$ $MaxThds/Block \leq 1024$
- threads composing a thread block must:
  - execute the same kernel
  - share data: issued to the same core
  - *Warp*: group of 32 threads; min size of data processed in SIMD fashion by CUDA multiprocessor.



Source: http://hothardware.com/Articles/NVIDIA-GF100-Architecture-and-Feature-Preview

Thread Parallelism

- We split the blocks into threads
- Threads can communicate with each other
- You can share information between blocks (using global memory and atomics, for example), but not global synchronization.
- Threads can be synchronized using *syncthreads*().
- Block parallelism: call kernel with N blocks, 1 thread per block
  add<<<N,1>>>(dev_a, dev_b, dev_c);
  N blocks x 1 Thread/block = N parallel threads
- Thread parallelism: call kernel with 1 block, N threads per block
  add<<<1,N>>>(dev_a, dev_b, dev_c);
  1 block x N Thread/block = N parallel threads
- Ultimately, we combine both models.

add_loop.cu, using 1 block and N threads

```
#include "../common/book.h"

#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc((void**)&dev_a,
            N*sizeof(int)));
    HANDLE_ERROR( cudaMalloc((void**)&dev_b,
            N*sizeof(int)));
    HANDLE_ERROR( cudaMalloc((void**)&dev_c,
            N*sizeof(int)));

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }
```

```
// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
        cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
        cudaMemcpyHostToDevice ) );


/* call kernel with 1 block, N threads per block */
add<<<1,N>>>( dev_a, dev_b, dev_c );


// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
        cudaMemcpyDeviceToHost ) );

// display the results
for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}

// free the memory allocated on the GPU
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );

return 0;
}
```

```
 /* Thread Parallelism: Using dynamic number of threads
 * Modified By:   Mary Thomas (mthomas@mail.sdsu.edu)
 * Based on:     CUDA SDK code add_loop_gpu.cu
 */
#include <stdio.h>
//#define N   65535+10
__device__ int d_Nthds;
__global__ void checkDeviceThdCount(int *t) {*t = d_Nthds;}
__global__ void add( int *a, int *b, int *c) {
   int tid = blockIdx.x;
   if (tid < d_Nthds) {
        c[tid] = a[tid] + b[tid];
    }
}
int main( int argc, char** argv ) {
if(argc != 2) {
    printf("Usage Error: %s <N> \n",argv[0]);}
    int h_N = atoi(argv[1]);

    int a[h_N], b[h_N], c[h_N];
    int *dev_a, *dev_b, *dev_c;
    int i,j,k;    int *d_N, d_Ntmp;
    float time;
    cudaEvent_t start, stop;
    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    cudaEventRecord(start, 0) ;

// set #threads to device variable d_Nthds
cudaMemcpyToSymbol(d_Nthds, &h_N, sizeof(int),
        0, cudaMemcpyHostToDevice);
cudaMalloc( (void**)&d_N, sizeof(int) );
checkDeviceThdCount<<<1,1>>>(d_N);
cudaMemcpy( &d_Ntmp, d_N, sizeof(int),
        cudaMemcpyDeviceToHost ) ;
cudaThreadSynchronize();
```

```
// fill the arrays 'a' and 'b' on the CPU
for (i=0; i<h_N; i++) {
        a[i] = i+1;
        b[i] = (i+1) * (i+1);
}
// allocate the memory on the GPU
cudaMalloc( (void**)&dev_a, h_N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_b, h_N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_c, h_N * sizeof(int) ) ;

// copy the arrays 'a' and 'b' to the GPU
cudaMemcpy( dev_a, a, h_N * sizeof(int),
        cudaMemcpyHostToDevice ) ;
cudaMemcpy( dev_b, b, h_N * sizeof(int),
        cudaMemcpyHostToDevice ) ;

add<<<1,h_N>>>( dev_a, dev_b, dev_c);

// copy the array 'c' back from the GPU to the CPU
cudaMemcpy( c, dev_c, h_N * sizeof(int),
        cudaMemcpyDeviceToHost ) ;

// free the memory allocated on the GPU
cudaFree( dev_a ) ;
cudaFree( dev_b ) ;
cudaFree( dev_c ) ;

//calculate elapsed time:
cudaEventRecord(stop, 0) ;
cudaEventSynchronize(stop) ;
//Compute elapsed time  (in milliseconds)
cudaEventElapsedTime(&time, start, stop) ;
printf("Nthreads=%ld, Telapsed(msec)= %26.16f\n",
        h_N,time);

return 0; }
```

Thread Parallelism:

```
[mthomas@tuckoo:cuda/add_loop] nvcc -arch=sm_20 -o add_loop_gpu add_loop_gpu.cu
[mthomas@tuckoo:cuda/add_loop] qsub -v T=10 bat.addloop
8709.tuckoo.sdsu.edu
[mthomas@tuckoo:cuda/add_loop] cat addloop.o8709
running add_loop_gpu using 10 threads
There are 2 CUDA devices.
CUDA Device #0
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
CUDA Device #1
Device Name: GeForce GTX 480
Maximum threads per block:     1024
Maximum dimensions of block: blockDim[0,1,2]=[ 1024  1024  64 ]
h_N = 10, d_N=1048576, d_Ntmp=10
1 + 1 = 2
2 + 4 = 6
3 + 9 = 12
4 + 16 = 20
5 + 25 = 30
6 + 36 = 42
7 + 49 = 56
8 + 64 = 72
9 + 81 = 90
10 + 100 = 110
Arr1[0]: 1 + 1 = 2
Arr1[5]: 6 + 36 = 42
Arr1[9]: 10 + 100 = 110
Arr2[0]: 1 + 1 = 2
Arr2[5]: 6 + 36 = 42
Arr2[9]: 10 + 100 = 110
GPU Nthreads=10, Telap(msec)=         0.4095999896526337
```

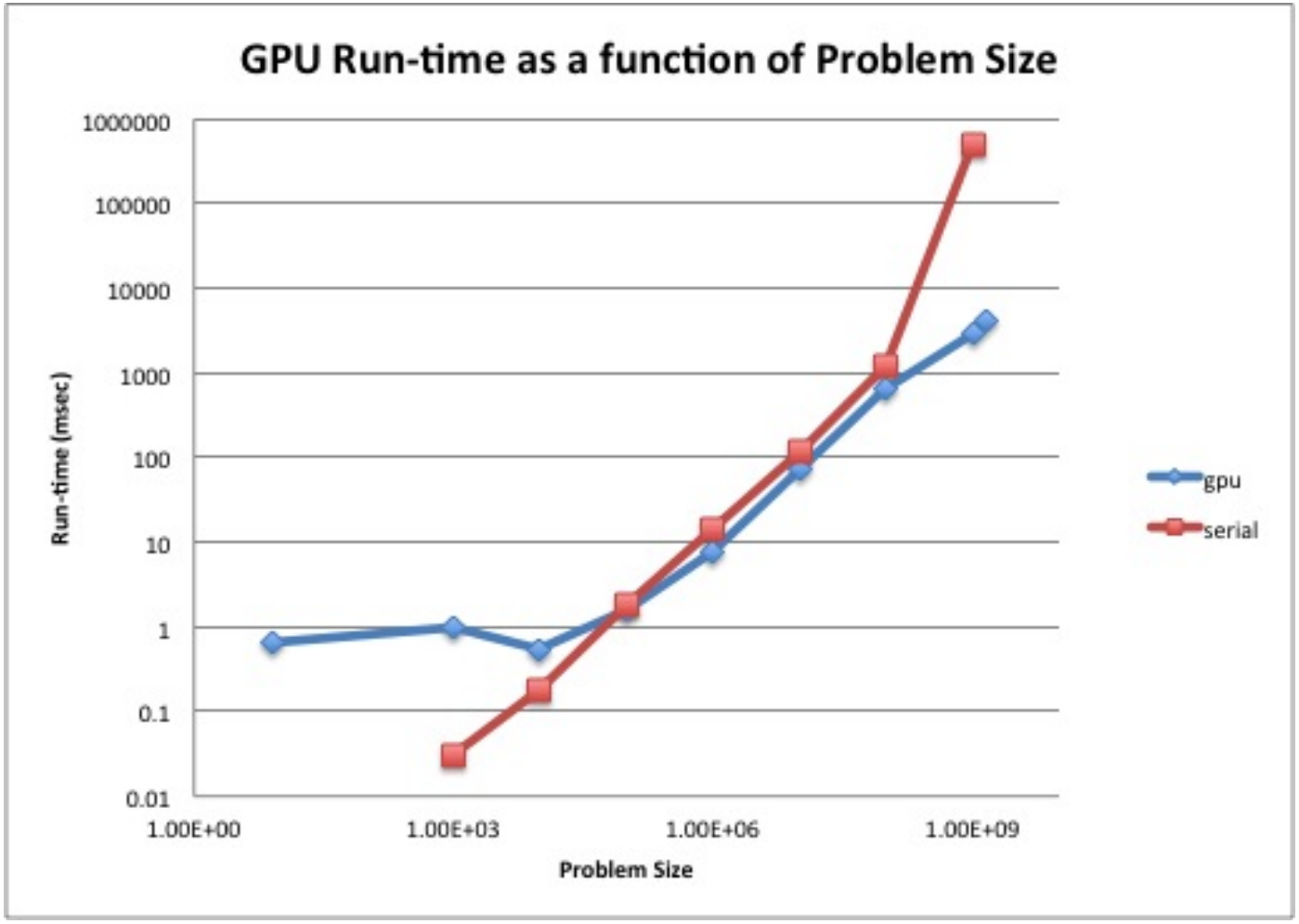
Thread Parallelism: add_loop_blocks.cu (output)

```
[mthomas@tuckoo]$ cat addloopser.o* | grep Telap
serial: Nthreads=10000, Telapsed(millisec) =                    184.0
serial: Nthreads=1000000, Telapsed(millisec) =                  15143.0
serial: Nthreads=100000000, Telapsed(millisec) =                 181107.0

[mthomas@tuckoo]$ cat addloopgpu.o* | grep Telap
GPU Nthreads=10000, Telap(msec)=          1.1845120191574097
GPU Nthreads=1000000, Telap(msec)=         11.1852159500122070
GPU Nthreads=100000000, Telap(msec)=        661.7844238281250000
GPU Nthreads=1410065408, Telap(msec)=       4061.6052246093750000
```

Loss of scaling when number of threads exceeds max threads per block (1024)

Performance comparison of serial vs GPU runtimes



Note, for small N, the GPU performance degrades after 10^3 but then improves for very large N.

What happens #threads is larger than #blocks*thds requested?

- You cannot exceed maxThreadsPerBlock
- Use device query to find out the max (1024 on tuckoo).
- You will loose parallel efficiency
- tuckoo.sdsu.edu (Spring 2014):
  - Max threads per block: 512 or 1024
  - Max thread dimensions: (512, 512, 64) or (1024x1024/64)
  - Max grid dimensions: (65535, 65535, 1)
- For large N, need 2D combination of threads and blocks
- Thread Rank: convert the 2D $[block, thread]$ space to a 1D indexing scheme

$$tid = threadIdx.x + blockIdx.x * blockDim.x;$$

Determining what the block & thread dimensions are on the device.

```
[mthomas@tuckoo:cuda/enum] cat /etc/motd
[snip]
GPUs
   ------------------------------------------
   node9  has 2     GTX 480 gpu cards (1.6GB dev ram ea.)
   node8  has 2     C2075   gpu cards (  6GB dev ram ea.)
   node7  has 2     C1060   gpu cards (  4GB dev ram ea.)
   node11 has 1     K40     gpu card  (                 )
[snip]
[mthomas@tuckoo:cuda/enum] cat enum_gpu.bat
#!/bin/sh
###PBS -l nodes=node9:ppn=1
#PBS -l nodes=node7:ppn=1
#PBS -N enum_gpu
#PBS -j oe
#PBS -q batch
cd $PBS_O_WORKDIR

./enum_gpu
```

```
----------------------------------------
NODE 7:   C1060
Name:  GeForce GT 240
   --- MP Information for device 2 ---
Multiprocessor count:  12
Shared mem per mp:  16384
Registers per mp:  16384
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
----------------------------------------
NODE 9:  GTX 480
Name:  GeForce GTX 480
   --- MP Information for device 1 ---
Multiprocessor count:  15
Shared mem per mp:  49152
Registers per mp:  32768
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (65535, 65535, 65535)
```

Translating thread row & column locations into unique thread IDs.

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Threads represent columns, blocks represent rows.

blockDim.x	**tid=threadIdx.x + blockIdx.x*blockDim.x**	**Th0**	**Th1**	**Th2**	**Th3**
0	$0 + 0 * 4 = 0$	0	1	2	3
1	$0 + 1 * 4 = 4$	4	5	6	7
2	$0 + 2 * 4 = 8$	8	9	10	11
3	$0 + 3 * 4 = 12$	12	13	14	15

Map to a vector of Thread ID's:

Elem(B,TID) =	[0,0]	[0,1]	[0,2]	[0,3]	[1,0]	[1,1]	[1,2]	[1,3]	[2,0]	[2,1]	[2,2]	[2,3]	[3,0]	[3,1]	[3,2]	[3,3]
Vector =	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]

- GPU hardware limits the number of blocks per grid and the number of threads per block
- Larger problems require use of both grid and blocks
- Need to control the number of threads, since they are smaller
- Fix number of threads and distributed chunks along the blocks:

```
add<<<128,128>>>( dev_a, dev_b, dev_c);
add<<<h_N,h_N>>>( dev_a, dev_b, dev_c);
add<<<ceil(h_N/128),128>>>( dev_a, dev_b, dev_c);
add<<<(h_N+127)/128,128>>>( dev_a, dev_b, dev_c);
```

- if maxTh == maximum number of threads per block:

```
add<<<(h_N+(maxTh-1))/maxTh, maxTh>>>( dev_a, dev_b, dev_c);
```

- Compute thread index as:
  $tid = threadIdx.x + blockIdx.x * blockDim.x;$

```
$tid = threadIdx.x + blockIdx.x * blockDim.x;$
```

```
/* CODE: add_loop_gpu for large # threads.
 */
#include <stdio.h>
//#define N   65535+10
__device__ int d_Nthds;
__global__ void checkDeviceThdCount(int *t) {*t = d_Nthds;}
__global__ void add( int *a, int *b, int *c) {
   tid = threadIdx.x + blockIdx.x * blockDim.x;
   if (tid < d_Nthds)
        c[tid] = a[tid] + b[tid];
}

int main( int argc, char** argv ) {
    /* get #threads from the command line */
    int h_N = atoi(argv[1]);
    int a[h_N], b[h_N], c[h_N];
    int *dev_a, *dev_b, *dev_c;
    int i,j,k,  *d_N, d_Ntmp;
    float time;
    cudaEvent_t start, stop;

    cudaEventCreate(&start) ;
    cudaEventCreate(&stop) ;
    cudaEventRecord(start, 0) ;

    // set the number of threads to device: d_Nthds
    cudaMemcpyToSymbol(d_Nthds, &h_N, sizeof(int),
        0, cudaMemcpyHostToDevice);

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, h_N * sizeof(int) ) ;
    cudaMalloc( (void**)&dev_b, h_N * sizeof(int) ) ;
    cudaMalloc( (void**)&dev_c, h_N * sizeof(int) ) ;
```

```
    // fill the arrays 'a' and 'b' on the CPU
    for (i=0; i<h_N; i++) {
        a[i] = i+1;     b[i] = (i+1) * (i+1);
    }

    // copy the arrays 'a' and 'b' to the GPU
     cudaMemcpy( dev_a, a, h_N * sizeof(int),
                cudaMemcpyHostToDevice ) ;
     cudaMemcpy( dev_b, b, h_N * sizeof(int),
                cudaMemcpyHostToDevice ) ;

    //add<<<128,128>>>( dev_a, dev_b, dev_c);
    //add<<<h_N,h_N>>>( dev_a, dev_b, dev_c);
    add<<<ceil(h_N/128),128>>>( dev_a, dev_b, dev_c);
    add<<<(h_N+127)/128,128>>>( dev_a, dev_b, dev_c);

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, h_N * sizeof(int),
                cudaMemcpyDeviceToHost ) ;

     // free the memory allocated on the GPU
     cudaFree( dev_a ) ; cudaFree( dev_b ) ;
     cudaFree( dev_c ) ;

    //Compute  elapsed time (in milliseconds)
    cudaEventRecord(stop, 0) ;
    cudaEventSynchronize(stop) ;
    cudaEventElapsedTime(&time, start, stop) ;
    printf("GPU Nthreads=%d, Telap(msec)= %26.16f\n",h_N,time);

    return 0;
}
```

Generalized kernel launch parameters dimGrid, dimBlock

- Distribute threads by thread blocks
- Kernel passes $<< \#blocks, \#threads >>$
- These are 3 dimensional objects, of type dim3 (C type)
- To distribute *h_N* threads, using the maximum number of threads per block, use:

```
int threadsperblock=maxthds;
blocksPerGridimin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
add<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_c);
```

OR, using the *dim3* object:

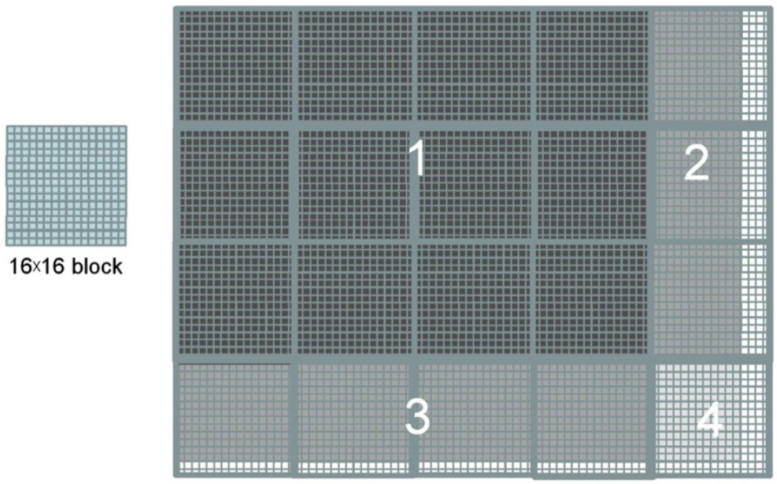
```
dim3 dimBlock(threadsperblock,1,1);
dim3 dimGrid(blocksPerGrid, 1, 1);
add<<<dimGrid, dimBlock>>>( dev_a, dev_b, dev_c);
```

- Calling a kernel for a 2D $m \times n$ matrix M[m][n], where $m < maxthds$ and $n < maxblocks$

```
dim3 dimGrid(n,1,1);
dim3 dimBlock(m, 1, 1);
add<<<dimGrid, dimBlock>>>( dev_a, dev_b, dev_c);
```

Mapping threads to multidimensional data

Example:
Covering a 76x62 picture with 16x16 blocks
m =76 horiz (x)
n = 62 vert (y) pixels




```
__global__ void kernel( unsigned char *ptr, int ticks )
{
// map from threadIdx/BlockIdx to pixel position
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;
. . .

dim3 dimBlock(16, 16, 1);
dim3 dimGrid(ceil(n/16.0), ceil(m/16.0), 1);
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

Row-major layout for 2D-C array

- The pixel data will have dynamic number of pixels
  - CUDA does not allow run-time allocation of a 2D matrix
  - Not allowed by the version of ANSI C used by CUDA (according to Kirk & Hu), but this may have changed by now).
- Need to linearize the array in $row - major$ order, into a vector which can be dynamic.
- 1 D array, where Element[row][col] is element [row*width+col]
- Thread mapping:

int x = threadIdx.x + blockIdx.x * blockDim.x;