Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters
2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

MCS 572 Lecture 21
Introduction to Supercomputing
Jan Verschelde, 14 October 2024

Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters

2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

grids, blocks, and threads

The code that runs on the GPU is defined in a function, the kernel.

A kernel launch

- creates a grid of blocks, and
- each block has one or more threads.

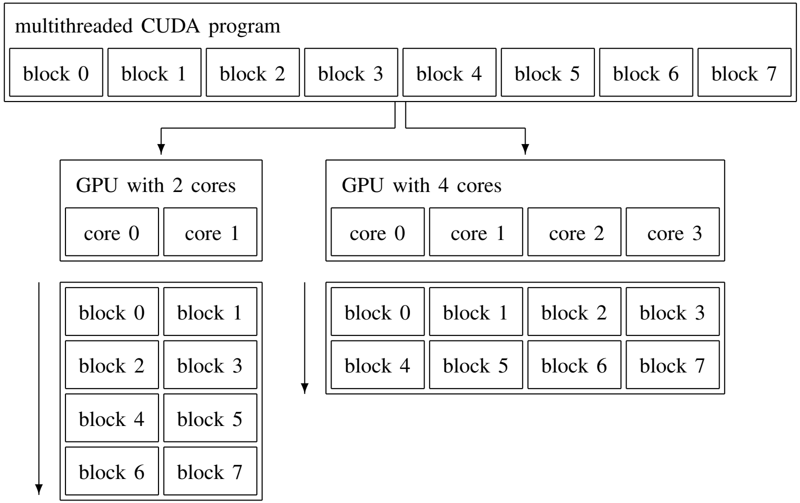
The organization of the grids and blocks can be 1D, 2D, or 3D.

During the running of the kernel:

- Threads in the same block are executed simultaneously.
- Blocks are scheduled by the streaming multiprocessors.

The P100 has 56 Streaming Multiprocessors (SMs) and
threads are executed in groups of 32 (the warp size).
Each SM has 64 cores.
This implies: $56 \times 64 = 3584$ threads can run simultaneously.
The A100 has 108 SMs, also with 64 cores each.

a scalable programming model



Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters

2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

identifying threads

All threads execute the same code, defined by the kernel.

The builtin variable `threadIdx`

- identifies every thread in a block uniquely; and
- defines the data processed by the thread.

The builtin variable `blockDim` holds the number of threads in a block.

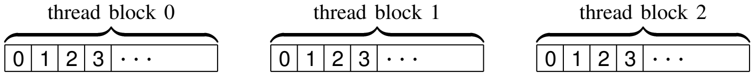
In a one dimensional organization, we use only `threadIdx.x` and `blockDim.x`. For 2D and 3D, the other components

- `threadIdx.y` belongs to the range 0..`blockDim.y`;
- `threadIdx.z` belongs to the range 0..`blockDim.z`.

data for each thread

The grid consists of N blocks, with `blockIdx.x` $\in \{0, N-1\}$.

Within each block, `threadIdx.x` $\in \{0,$ `blockDim.x` $- 1\}$.



```
int threadId = blockIdx.x *
   blockDim.x + threadIdx.x
...
float x = input[threadID]
float y = f(x)
output[threadID] = y
...
```

Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters
2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

setting the execution configuration parameters

Suppose the kernel is defined by the function `F` with input arguments `x` and output arguments `y`, then

```
dim3 dimGrid(128,1,1);
dim3 dimBlock(32,1,1);
F<<<dimGrid,dimBlock>>>(x,y);
```

launches a grid of 128 blocks. The grid is a one dimensional array.

Each block in the grid is also one dimensional and has 32 threads.

multidimensional thread organization

Limitations of the P100 and V100:

- Maximum number of threads per block: 1,024.
- Maximum sizes of each dimension of a block: $1,024 \times 1,024 \times 64$.
  Because 1,024 is the upper limit for the number of threads in a block, the largest square 2D block is 32×32, as $32^2 = 1,024$.
- Maximum sizes of each dimension of a grid:
  $2,147,483,647 \times 65,535 \times 65,535$.
  2,147,483,647 is the upper limit for the builtin variable `gridDim.x`, while 65,535 is the upper limit for the builtin variables `gridDim.y` and `gridDim.z`.

The same limitations apply for the A100.

a 3D example

Suppose the function `F` defines the kernel, with argument `x`, then

```
dim3 dimGrid(3,2,4);
dim3 dimBlock(5,6,2);
F<<<dimGrid,dimBlock>>>(x);
```

launches a grid with

- $3 \times 2 \times 4$ blocks; and
- each block has $5 \times 6 \times 2$ threads.

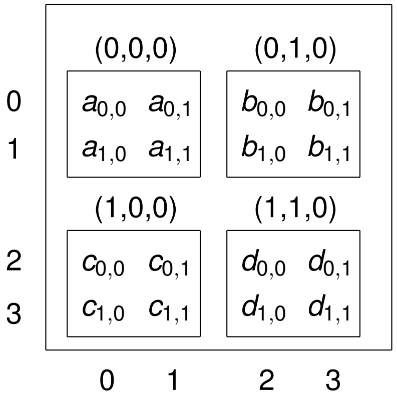
Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters

2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

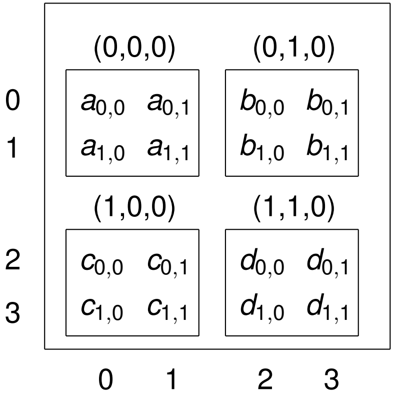
submatrices

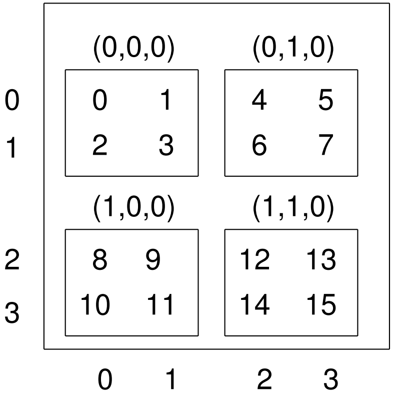
Consider a grid of dimension $2 \times 2 \times 1$
to store a 4-by-4 matrix in tiles of dimensions $2 \times 2 \times 1$:



mapping threads to entries in the matrix

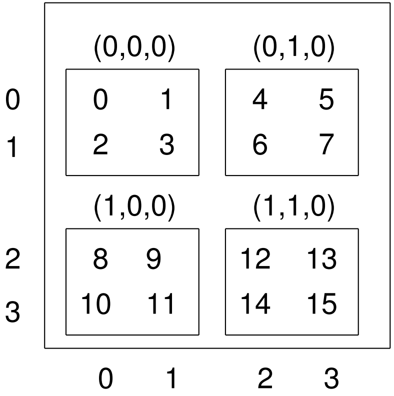
A kernel launch with a grid of dimensions $2 \times 2 \times 1$
where each block has dimensions $2 \times 2 \times 1$ creates 16 threads.


(0,0,0)
(0,1,0)
(1,0,0)
(1,1,0)
$a_{0,0}$ $a_{0,1}$ $a_{1,0}$ $a_{1,1}$
$b_{0,0}$ $b_{0,1}$ $b_{1,0}$ $b_{1,1}$
$c_{0,0}$ $c_{0,1}$ $c_{1,0}$ $c_{1,1}$
$d_{0,0}$ $d_{0,1}$ $d_{1,0}$ $d_{1,1}$
0 1 2 3
0 1 2 3



linear address calculation

A kernel launch with a grid of dimensions $2 \times 2 \times 1$
where each block has dimensions $2 \times 2 \times 1$ creates 16 threads.



```
x[0][0][0][0][0][0] = 0
x[0][0][0][0][1][0] = 1
x[0][0][0][1][0][0] = 2
x[0][0][0][1][1][0] = 3
x[0][1][0][0][0][0] = 4
x[0][1][0][0][1][0] = 5
x[0][1][0][1][0][0] = 6
x[0][1][0][1][1][0] = 7
x[1][0][0][0][0][0] = 8
x[1][0][0][0][1][0] = 9
x[1][0][0][1][0][0] = 10
x[1][0][0][1][1][0] = 11
x[1][1][0][0][0][0] = 12
x[1][1][0][0][1][0] = 13
x[1][1][0][1][0][0] = 14
x[1][1][0][1][1][0] = 15
```

Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters

2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

the main program

```
int main ( int argc, char* argv[] )
{
   const int xb = 2; /* gridDim.x */
   const int yb = 2; /* gridDim.y */
   const int zb = 1; /* gridDim.z */
   const int xt = 2; /* blockDim.x */
   const int yt = 2; /* blockDim.y */
   const int zt = 1; /* blockDim.z */
   const int n = xb*yb*zb*xt*yt*zt;

   printf("allocating array of length %d...\n",n);

   /* allocating and initializing on the host */

   int *xhost = (int*)calloc(n,sizeof(int));
   for(int i=0; i<n; i++) xhost[i] = -1.0;
```

copy to device and kernel launch

```
int *xdevice;
size_t sx = n*sizeof(int);
cudaMalloc((void**)&xdevice,sx);
cudaMemcpy(xdevice,xhost,sx,cudaMemcpyHostToDevice);

/* set the execution configuration for the kernel */

dim3 dimGrid(xb,yb,zb);
dim3 dimBlock(xt,yt,zt);
matrixFill<<<dimGrid,dimBlock>>>(xdevice);
```

the kernel definition

```
__global__ void matrixFill ( int *x )
/*
 * Fills the matrix using blockIdx and threadIdx. */
{
   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   int row = by*blockDim.y + ty;
   int col = bx*blockDim.x + tx;
   int dim = gridDim.x*blockDim.x;
   int i = row*dim + col;
   x[i] = i;
}
```

copying to host and writing the result

```
   /* copy data from device to host */
   cudaMemcpy(xhost,xdevice,sx,cudaMemcpyDeviceToHost);
   cudaFree(xdevice);

   int *p = xhost;
   for(int i1=0; i1 < xb; i1++)
     for(int i2=0; i2 < yb; i2++)
       for(int i3=0; i3 < zb; i3++)
         for(int i4=0; i4 < xt; i4++)
           for(int i5=0; i5 < yt; i5++)
              for(int i6=0; i6 < zt; i6++)
                printf("x[%d][%d][%d][%d][%d][%d] = %d\n",
                        i1,i2,i3,i4,i5,i6,*(p++));
   return 0;
}
```

Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters

2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

organization.jl

```
using CUDA

"""
    function matFill!(A)

fills the array using the blockIdx and threadIdx.
"""
function matFill!(A)
    bx = blockIdx().x - 1
    by = blockIdx().y - 1
    tx = threadIdx().x - 1
    ty = threadIdx().y - 1
    row = by*blockDim().y + ty
    col = bx*blockDim().x + tx
    dim = gridDim().x*blockDim().x
    idx = 1 + row*dim + col
    A[idx] = idx
    return nothing
end
```

organization.jl continued

```
xb = 2 # gridDim.x
yb = 2 # gridDim.y
zb = 1 # gridDim.z
xt = 2 # blockDim.x
yt = 2 # blockDim.y
zt = 1 # blockDim.z

dim = xb*yb*zb*xt*yt*zt
A_h = zeros(dim)
A_d = CuArray(A_h)

@cuda threads=(xt, yt, zt) blocks=(xb, yb, zb)
matFill!(A_d)

A_h = Array(A_d)
println(A_d)
println(A_h)
```

Thread Organization and Matrix Multiplication

thread synchronization

In a block all threads run independently.

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function:

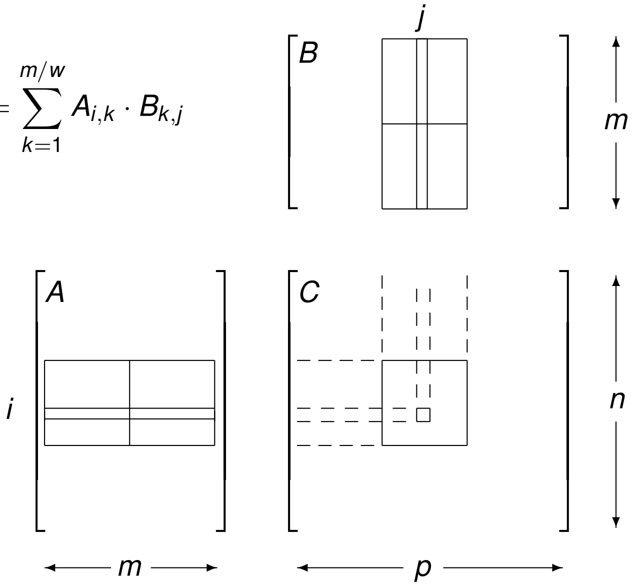
`__syncthreads().`

The thread executing `__syncthreads()` will be held at the calling location in the code until every thread in the block reaches the location.

Placing a `__syncthreads()` ensures that all threads in a block have completed a task before moving on.

applied to matrix multiplication with shared memory

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$$



application of `__syncthreads()`

With tiled matrix matrix multiplication using shared memory, all threads in the block collaborate to copy the tiles $A_{i,k}$ and $B_{k,j}$ from global memory to shared memory.

$\rightarrow$ Before the calculation of the inner products, all threads must finish their copy statement: they all execute the `__syncthreads()`.

Every thread computes one inner product.

$\rightarrow$ Before moving on to the next tile, all threads must finish, therefore, they all execute the `__syncthreads()` after computing their inner product and moving on to the next phase.

Thread Organization and Matrix Multiplication

1. Thread Organization
   - grids, blocks, and threads
   - using `threadIdx` and `blockIdx`
   - setting the execution configuration parameters

2. Matrix Matrix Multiplication
   - accessing submatrices with thread identifiers
   - CUDA code for thread organization
   - submatrices with threads in CUDA.jl
   - thread synchronization
   - revisiting the kernel of `matrixMul`

the kernel of `matrixMul`

```
template <int BLOCK_SIZE> __global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x;    // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;   // Thread index
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;
```

the submatrices

```
// Csub is used to store the element of the block sub-matrix
// that is computed by the thread
float Csub = 0;

// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
```

loading and multiplying

```
        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
```

the end of the kernel

```
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Recommended reading:

- NVIDIA CUDA Programming Guide.
  Available at `http://developer.nvdia.com`.
- Vasily Volkov and James W. Demmel: **Benchmarking GPUs to tune dense linear algebra.** In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008. Article No. 31.

summary and exercises

We covered more about the essentials of GPU acceleration.

1. Find the limitations of the grid and block sizes for the graphics card on your laptop or desktop.
2. Extend the simple code with the three dimensional thread organization to a tiled matrix-vector multiplication for numbers generated at random as 0 or 1.
3. Use Julia to define an accelerated tiled matrix matrix multiplication. Verify its correctness and examine its performance.