Distributed and Parallel Computing

Lecture 03

Alan P. Sexton

University of Birmingham

Spring 2016

Multidimensional Grids and Blocks

Up till now we have specified only 1 dimensional grids and blocks:

```
unsigned int threadsPerB = 256;
unsigned int blocksPerG = 1 + ((n - 1) / threadsPerB);
vectorAdd<<<blocksPerG,threadsPerB>>>(d_A, d_B, d_C, n);
```

The more general way to do this is:

```
unsigned int threadsPerB = 256;
unsigned int blocksPerG = 1 + ((n - 1) / threadsPerB);

dim3 dimBlock (threadsPerB, 1, 1)
dim3 dimGrid (blocksPerG, 1, 1)
vectorAdd<<<dimGrid,dimBlock>>>(d_A, d_B, d_C, n);
```

The possible maximum values for each dimension are given by the output of *deviceQuery*:

```
Max dim size of a thread block (x,y,z): (1024, 1024, 64)
Max dim size of a grid size (x,y,z): (65535, 65535, 65535)
```

Given "`Maximum number of threads per block: 1024`", Then you must have that $x \times y \times z \leq 1024$ for the dimensions of a thread block.

Multidimensional Grids and Blocks

C supports multidimensional arrays: e.g. `A[row][col]`

- But this requires all except the last dimension size to be known at compile time
- Since we need dimensions to be chosen dynamically, encode multidimensionality in a 1-dimensional block of memory.

Row major: place all elements of the ***same row*** in consecutive locations:

- `M[row][col]` $\equiv$ `&M[row * row_width + col]`

Column major: place all elements of the ***same column*** in consecutive locations:

- `M[row][col]` $\equiv$ `&M[row + col * col_height]`

C uses row major layout. Fortran uses column major layout

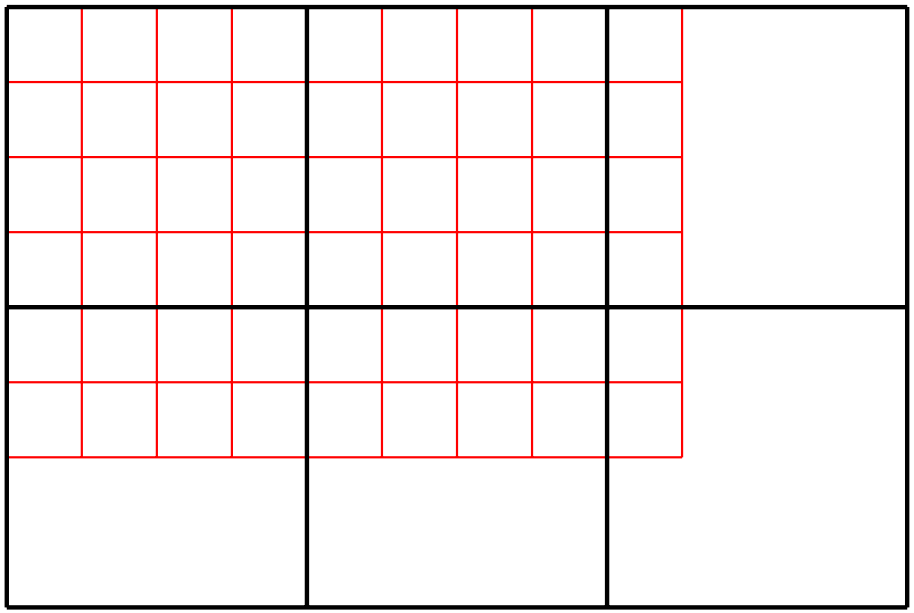
- Many libraries (e.g. BLAS) originally designed for Fortran and use column major
- In such cases, may need to transpose matrix to use the libraries (see library documentation)

Matrix with m rows and n columns

For 2-dimensional matrix or picture of size $m \times n$, 1 thread/cell:

- Total number of columns supported is `blockDim.x * gridDim.x`
- Total number of rows supported is `blockDim.y * gridDim.y`

Grid is (3, 2, 1)
Block is (4, 4, 1)
Matrix is 6×9



- Figure out row and column from block and thread ids:

```
row = blockIdx.y * blockDim.y + threadIdx.y;
col = blockIdx.x * blockDim.x + threadIdx.x;
if (row < m && col < n)
    M[row * n + col] ...
```

Large Matrices

How large a matrix can be handled?

- All data must fit in global memory:
  `Total amount of global memory:  1023 MBytes`
- If one thread/cell then limited by:
  max Block size $\times$ max number of grid cells in each dimension
- For Geforce GT 610
  - max 1024 threads/block $= 32 \times 32$ if kept square
  - max 65536 blocks per dimension of grid
  - hence $32 \times 65536 = 2,097,152$ in each dimension
- What if matrix is too big?
  - *Either* split matrix into submatrices and have host iterate the kernel calls
  - *Or* have each thread handle more than one cell

Choosing Configuration

Assume device has:

- Max 8 blocks per Streaming Multiprocessor (SM)
- max 1024 threads per SM
- max 512 threads per block

Consider a matrix/picture op: which block size is best 8×8, 16×16 or 32×32?

Choosing Configuration

- 8×8
  - 64 threads per block
  - SM supports 1024 threads $\Rightarrow$ need $1024/64 = 12$ blocks to fully utilise the SM.
  - Only allowed 8 blocks $\Rightarrow$ only $64 \times 8 = 512$ can be scheduled together
  - Thus fewer warps than SM supports $\Rightarrow$ might hit latency problems
- 16×16
  - 256 threads per block
  - Each SM takes $1024/256 = 4$ blocks
  - within 8 block limit and full SM utilisation $\Rightarrow$ good.
- 32×32
  - 1024 threads per block, greater than limit allowed $\Rightarrow$ fails

Choosing Configuration for Lab Machines

- 1 SM
- max 1536 $= 3 \times 512$ threads per SM
- max 1024 threads per block

Same Matrix/Picture operation:

- $8 \times 8 \Rightarrow 64$ threads/block $\Rightarrow 1536/64 = 24$ blocks to fully utilise SM $\Rightarrow$ Okay
- $16 \times 16 \Rightarrow 256$ threads/block $\Rightarrow 1536/256 = 6$ blocks to fully utilise SM $\Rightarrow$ Okay
- $32 \times 32 \Rightarrow 1024$ threads/block $\Rightarrow 1536/1024 = 1.5$ blocks to fully utilise SM. But only full blocks can be scheduled so wasting 1/3 of the SM $\Rightarrow$ runs but poor performance

Other Constraints

There are other contraints that might limit the configuration

- Register usage
  - automatic variables in device functions are stored in registers
  - 1 implicit register per thread
  - registers are private to each thread
  - `Total number of registers available per block: 32768`
- Shared Memory usage
  - Each thread uses some shared memory (more about this later)
  - `Total amount of shared memory per block: 49152 bytes`

Querying the GPU properties

SInce different GPUs have different capabilities, we should write CUDA code to adapt to the different GPUs. Therefore we need to be able to query those capabilities.

- There can be a number of GPUs in a computer

```
int dev_count;
err = cudaGetDeviceCount(&dev_count);
```

- Then can get the properties of a specific device:

```
cudaDeviceProp dev_prop;
int dev_number = 0;
err = cudaGetDeviceProperties(&dev_prop, dev_number);
```

- `cudaDeviceProp` is a structure with fields for all the properties you have already seen displayed from the `deviceQuery` program
- This information can be used, not only to choose a viable configuration for the device, but to identify a whole range of possible configurations and to automate tuning the application by exhaustively running timings over them.