CUDA Memory Types

CPS343

Parallel and High Performance Computing

Spring 2013

Outline

1. Device memory
   - CUDA memory types and uses
   - CUDA Type qualifiers
   - Programming Scenarios
2. Matrix multiplication
   - Matrix-matrix multiplication
   - Global memory version
   - Shared memory version

Acknowledgements

Some material used in creating these slides comes from

- NVIDIA's CUDA C Programming Guide
- http://www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf

Outline

1. Device memory
   - CUDA memory types and uses
   - CUDA Type qualifiers
   - Programming Scenarios

2. Matrix multiplication
   - Matrix-matrix multiplication
   - Global memory version
   - Shared memory version

Compute Capability 1.x

- Global memory (read and write)
  - slow and uncached
  - requires sequential and aligned 16 byte read/writes to be fast (coalesed read/write)
- Texture memory (read only) – cache optimized for 2D access pattern
- Constant memory
  - where constants and kernel arguments are stored
  - slow, but with cache
- Shared memory (16KB per SM)
  - fast, but subject to bank conflicts
  - permits exchange of data between threads in block
- Local memory
  - used for whatever doesn't fit in to registers
  - part of global memory, so slow and uncached
- Registers – fast, has only thread scope

Compute Capability 2.x

- Global memory (read and write)
  - slow, but cached
- Texture memory (read only) – cache optimized for 2D access pattern
- Constant memory
  - where constants and kernel arguments are stored
  - special "LoaD Uniform" (LDU) instruction
- Shared memory (48KB per SM)
  - fast, but subject to (differnt) bank conflicts
- Local memory
  - used for whatever doesn't fit in to registers
  - part of global memory; slow but now cached
- Registers – 32768 32-bit registers per SM

Memory limitations

Global Memory

- Best if 64 or 128 bytes (16 or 32 single-precision, 8 or 16 double-precision) are read...
- *Coalesced* read/writes:
    - parallel read/writes from threads in a block
    - sequential memory locations...
    - ...with appropriate alignment
- ...otherwise up to 10x slower!

Shared Memory

- Fastest if all threads read from same shared memory location and/or all threads index a shared aray via permutation (e.g. linear read/writes)
- otherwise there can be bank conflicts

Outline

1. Device memory
   - CUDA memory types and uses
   - CUDA Type qualifiers
   - Programming Scenarios
2. Matrix multiplication
   - Matrix-matrix multiplication
   - Global memory version
   - Shared memory version

CUDA Type qualifiers

Variable declaration	Memory	Scope	Lifetime
`int localVar;`	register	thread	thread
`int localArray[10];`	local	thread	thread
`__shared__ int sharedVar;`	shared	block	block
`__device__ int globalVar;`	global	grid	application
`__constant__ int constantVar;`	constant	grid	application

- Automatic variables without any qualifier reside in a register...
- ...except arrays (reside in local memory)
- ...or if there are not enough registers

CUDA Type performance

Variable declaration	Memory	Performance penalty
`int localVar;`	register	1x
`int localArray[10];`	local	100x
`__shared__ int sharedVar;`	shared	1x
`__device__ int globalVar;`	global	100x
`__constant__ int constantVar;`	constant	1x

Outline

1. Device memory
   - CUDA memory types and uses
   - CUDA Type qualifiers
   - **Programming Scenarios**
2. Matrix multiplication
   - Matrix-matrix multiplication
   - Global memory version
   - Shared memory version

Scenario 1

- Task:
  - Load data from global memory
  - Do **thread-local** computations
  - Store result to global memory
- Solution:
  - Load data from global memory (coalesced)

    ```
    float a = d_ptr[blockIdx.x*blockDim.x + threadIdx.x];
    ```

  - Do computation with registers

    ```
    float res = f(a);
    ```

  - Store result (coalesced)

    ```
    d_ptr[blockIdx.x*blockDim.x + threadIdx.x] = res;
    ```

Scenario 2

- Task:
  - Load data from global memory
  - Do **block-local** computations
  - Store result to global memory

- Solution:
  - Load data to shared memory

    ```
    __shared__ float a_sh[BLOCK_SIZE];
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a_sh[threadIdx.x] = d_ptr[idx];
    __syncthreads(); // important!
    ```

  - Do computation

    ```
    float res = f(a_sh[threadIdx.x]);
    ```

  - Store result (coalesced)

    ```
    d_ptr[idx] = res;
    ```

Outline

1. Device memory
   - CUDA memory types and uses
   - CUDA Type qualifiers
   - Programming Scenarios

2. Matrix multiplication
   - Matrix-matrix multiplication
   - Global memory version
   - Shared memory version

Matrix-matrix multiplication

Consider our familiar matrix product $C = AB$:

```
for ( i = 0; i < A.height; i++ )
{
    for ( j = 0; j < B.width; j++ )
    {
        c[i][j] = 0;
        for ( k = 0; k < A.width; k++ )
        {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

- How many times is each element of matrix A accessed?
- How many times is each element of matrix B accessed?
- How many times is each element of matrix C accessed?

Matrix-matrix multiplication

Consider our familiar matrix product $C = AB$:

```
for ( i = 0; i < A.height; i++ )
{
    for ( j = 0; j < B.width; j++ )
    {
        c[i][j] = 0;
        for ( k = 0; k < A.width; k++ )
        {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

- How many times is each element of matrix A accessed? **B.width**
- How many times is each element of matrix B accessed?
- How many times is each element of matrix C accessed?

Matrix-matrix multiplication

Consider our familiar matrix product $C = AB$:

```
for ( i = 0; i < A.height; i++ )
{
    for ( j = 0; j < B.width; j++ )
    {
        c[i][j] = 0;
        for ( k = 0; k < A.width; k++ )
        {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

- How many times is each element of matrix A accessed? **B.width**
- How many times is each element of matrix B accessed? **A.height**
- How many times is each element of matrix C accessed?

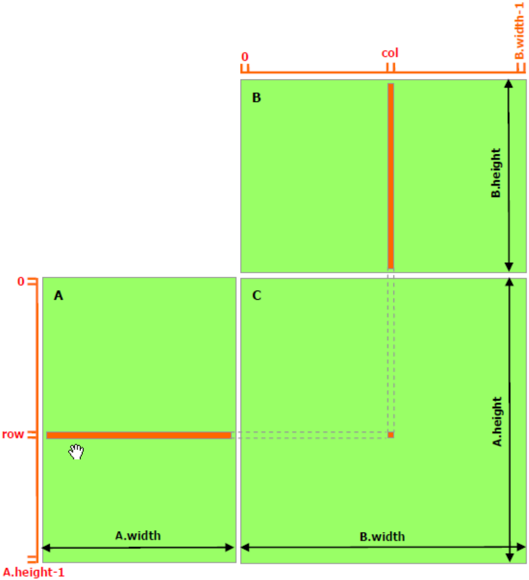
Matrix-matrix multiplication

Consider our familiar matrix product $C = AB$:

```
for ( i = 0; i < A.height; i++ )
{
    for ( j = 0; j < B.width; j++ )
    {
        c[i][j] = 0;
        for ( k = 0; k < A.width; k++ )
        {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

- How many times is each element of matrix A accessed? **B.width**
- How many times is each element of matrix B accessed? **A.height**
- How many times is each element of matrix C accessed? **A.width**

Matrix-matrix multiplication



- Consider an element `c[row][col]`. There are B.width elements on a row of C and A.height elements in a column of C.
- To compute each of these elements, we access a row of A and a column of B.
- We therefore access each row of A B.width times and each column of B A.height times.

Outline

1. Device memory
   - CUDA memory types and uses
   - CUDA Type qualifiers
   - Programming Scenarios
2. Matrix multiplication
   - Matrix-matrix multiplication
   - Global memory version
   - Shared memory version

Kernel development

- A CUDA kernel to compute the matrix product is straightforward
- In this simple implementation we assume that our matrices are square $N \times N$ and stored using linear arrays
- Access to the (i, j) element is faciliated via the macro

```
#define IDX(i,j,n) ((i)*(n)+j)
```

Matrix multiply: Global memory version – kernel

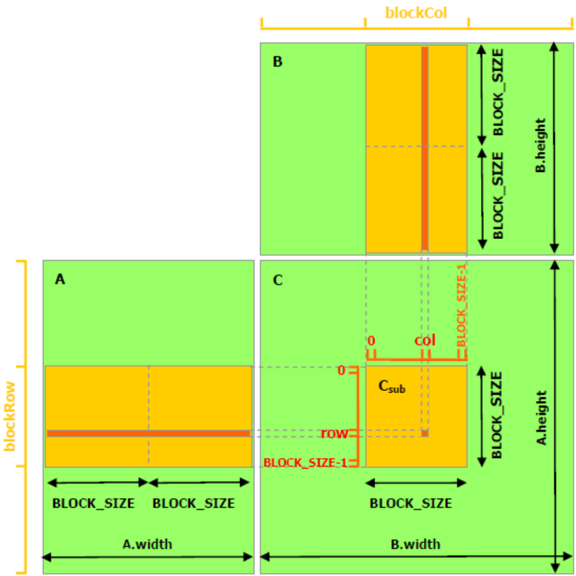
```
// matrix-matrix kernel using only global memory
__global__ void matmulGlobal( float* c, float* a, float* b,
                              int N )
{
    // compute row and column for our matrix element
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if ( col < N && row < N )
    {
        float sum = 0.0;
        for ( int k = 0; k < N; k++ )
        {
            sum += a[IDX(row,k,N)] * b[IDX(k,col,N)];
        }
        c[IDX(row,col,N)] = sum;
    }
}
```

Note: sum will be stored in a register so we this kernel only makes one reference to C.

Outline

1. Device memory
   - CUDA memory types and uses
   - CUDA Type qualifiers
   - Programming Scenarios
2. Matrix multiplication
   - Matrix-matrix multiplication
   - Global memory version
   - Shared memory version

Matrix-matrix multiplication



Matrix multiply: Shared memory version – kernel 1

```
// matrix-matrix kernel using global and shared memory
__global__ void matmulShared( float* c, float* a, float* b,
                              int N )
{
    // compute row and column for our matrix element
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // compute the number of blocks we need
    int M = ( N + BlockSize - 1 ) / BlockSize;
```

Matrix multiply: Shared memory version – kernel 2

```
float sum = 0.0;
for ( int m = 0; m < M; m++ )
{
    // all threads in block copy their element from
    // matrix a and matrix b to shared memory
    __shared__ float a_s[BlockSize][BlockSize];
    __shared__ float b_s[BlockSize][BlockSize];
    int c = m * BlockSize + threadIdx.x;
    int r = m * BlockSize + threadIdx.y;
    a_s[threadIdx.y][threadIdx.x] = a[IDX(row,c,N)];
    b_s[threadIdx.y][threadIdx.x] = b[IDX(r,col,N)];

    // make sure all threads are finished
    __syncthreads();
```

Matrix multiply: Shared memory version – kernel 3

```
        // compute partial sum using shared memory block
        // K is block size except at right or bottom since we
        //   may not have a full block of data there
        int K = (m == M - 1 ? N - m * BlockSize : BlockSize);
        for ( int k = 0; k < K; k++ )
        {
            sum += a_s[threadIdx.y][k] * b_s[k][threadIdx.x];
        }
        __syncthreads();
    }
    if ( col < N && row < N ) c[IDX(row,col,N)] = sum;
}
```