Memory access

Vital for performance!

Memory types

Coalescing

Example of using shared memory

Memory types

Global

Shared

Constant (read only)

Texture cache (read only)

Local

Registers

Care about these when optimizing - not to begin with

Global memory

400-600 cycles latency!

Shared memory fast temporary storage

Coalesce memory access!

Continuous
Aligned on power of 2 boundary
Addressing follows thread numbering

Use shared memory for reorganizing data for coalescing!

Information Coding / Computer Graphics, ISY, LiTH

Using shared memory to reduce number of global memory accesses

Read blocks of data to shared memory

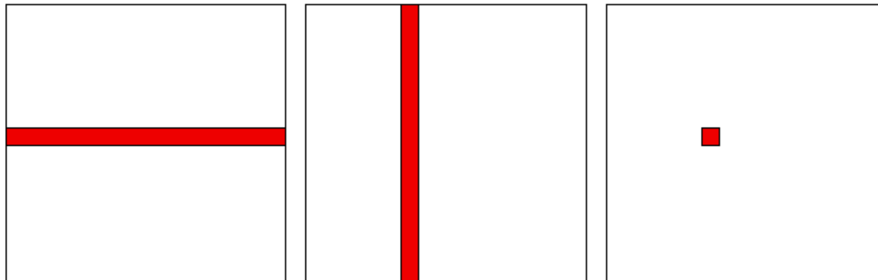
Process

Write back as needed

Shared memory as ”manual cache”

Example: Matrix multiplication

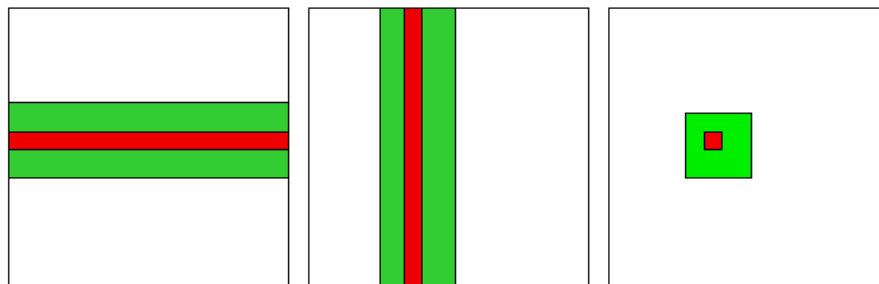
Matrix multiplication



To multiply two N*N matrices, every item will have to be accessed N times!

Naive implementation: $2N^3$ global memory accesses!

Matrix multiplication



Let each block handle a part of the output.

Load the parts of the matrix needed for the block into shared memory.

Matrix multiplication on CPU

Simple triple ”for” loop

```
void MatrixMultCPU(float *a, float *b, float *c, int theSize)
{
    int sum, i, j, k;

    // For every destination element
    for(i = 0; i < theSize; i++)
        for(j = 0; j < theSize; j++)
        {
            sum = 0;
            // Sum along a row in a and a column in b
            for(k = 0; k < theSize; k++)
                sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
            c[i*theSize + j] = sum;
        }
}
```

Information Coding / Computer Graphics, ISY, LiTH

Naive GPU version

Replace outer loops by thread indices

```
__global__ void MatrixMultNaive(float *a, float *b, float *c, int
theSize)
{
    int sum, i, j, k;

    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    // For every destination element
    sum = 0;
    // Sum along a row in a and a column in b
    for(k = 0; k < theSize; k++)
        sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
    c[i*theSize + j] = sum;
}
```

Naive GPU version inefficient

Every thread makes 2N global memory accesses!

Can be significantly reduced using shared memory

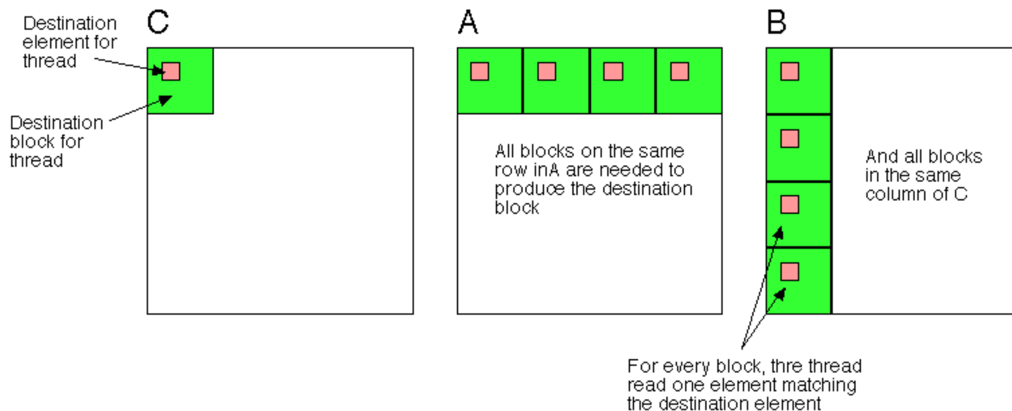
Optimized GPU version

Data split into blocks.

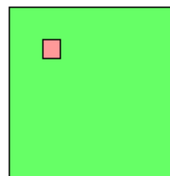
Every element takes part in all the blocks in the same *row* for A, *column* for B

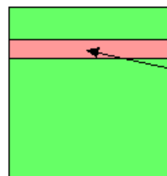
For every such block

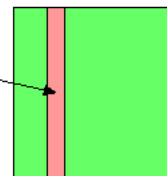
- Every thread reads *one* element to shared memory
- Then loop over the appropriate row and column for the block



For every block, we loop over the part of one row and column to perform that part of the computation







What one thread reads is used by everybody in the same row (A) or column (B)!

Information Coding / Computer Graphics, ISY, LiTH

Optimized GPU version

Loop over blocks (1D)

Allocate shared memory

Copy one element to shared memory

Loop over row/column in block, compute, accumulate result for one element

Write result to global memory

```
__global__ void MatrixMultOptimized( float* A, float* B, float* C, int theSize)
{
    int i, j, k, b, ii, jj;

// Global index for thread
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    float sum = 0.0;
    // for all source blocks
    for (b = 0; b < gridDim.x; b++)
    {
        __shared__ float As[BLOCKSIZE*BLOCKSIZE];
        __shared__ float Bs[BLOCKSIZE*BLOCKSIZE];

        // Index locked to block
        ii = b * blockDim.x + threadIdx.x;
        jj = b * blockDim.y + threadIdx.y;

        As[threadIdx.y*blockDim.x + threadIdx.x] = A[ii*theSize + j];
        Bs[threadIdx.y*blockDim.x + threadIdx.x] = B[i*theSize + jj];

        __syncthreads(); // Synchronize to make sure all data is loaded

        // Loop in block
        for (k = 0; k < blockDim.x; ++k)
            sum += As[threadIdx.y*blockDim.x + k]
             * Bs[k*blockDim.x + threadIdx.x];

        __syncthreads(); // Synch so nobody starts next pass prematurely
    }

    C[i*theSize + j] = sum;
}
```

Modified computing model:

Upload data to global GPU memory

For a number of parts, do:

Upload partial data to shared memory

Process partial data

Write partial data to global memory

Download result to host

Synchronization

As soon as you do something where one part of a computation depends on a result from another thread, you must synchronize!

__syncthreads()

Typical implementation:

- Read to shared memory
- __syncthreads()
- Process shared memory
- __synchthreads()
- Write result to global memory

Summary:

- Make threads and blocks to make the hardware occupied
- Access data depending on thread/block number
- Memory accesses are expensive!
- Shared memory is fast
- Make threads within a block cooperate
- Synchronize

That's all folks!

Next: More about memory management and optimization.