CUDA Memory

Advanced Aspects

Albert García García

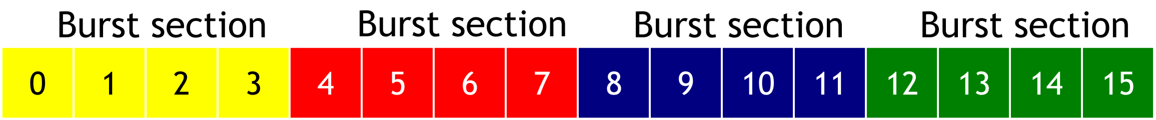
agarcia @ dtic.ua.es

Contents

Coalesced Access

Coalesced Access Revisited

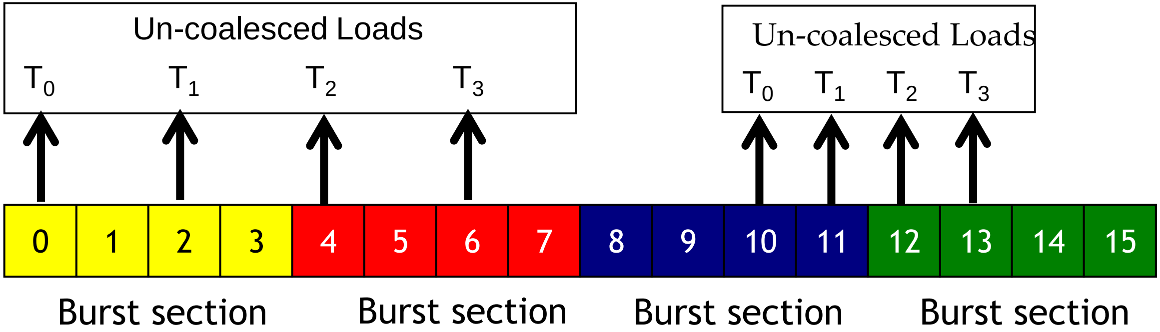
Memory bursting



- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor.
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more.

Coalesced Access Revisited

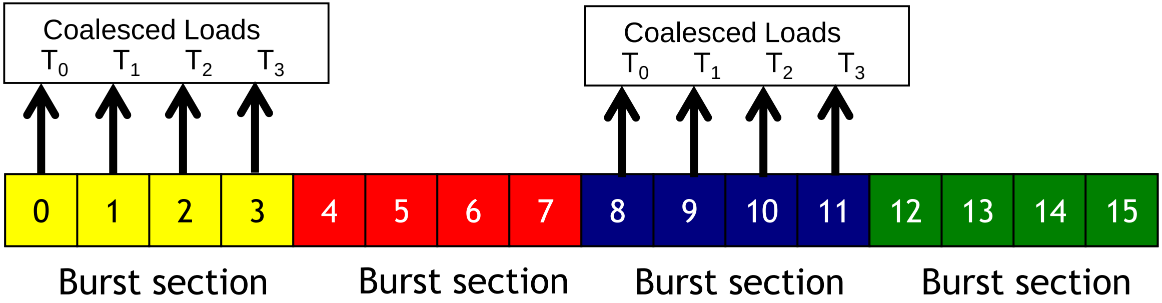
Memory bursting: Uncoalesced access



- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

Coalesced Access Revisited

Memory bursting: Coalesced access



– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

Coalesced Access Revisited

Colaesced access example

```
__global__ coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid] = threadIdx.x;
}
```

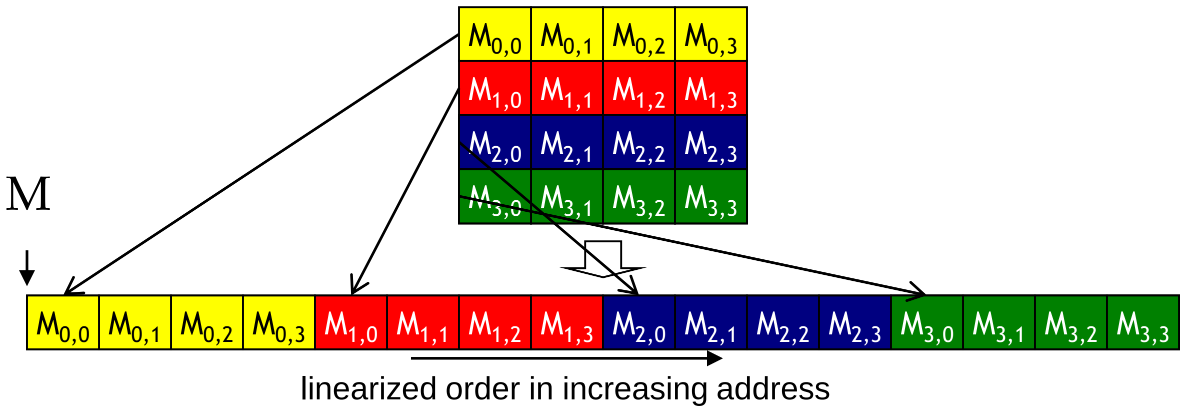
Coalesced Access Revisited

Non-colaesced access example

```
__global__ non_coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid*100] = threadIdx.x;
}
```

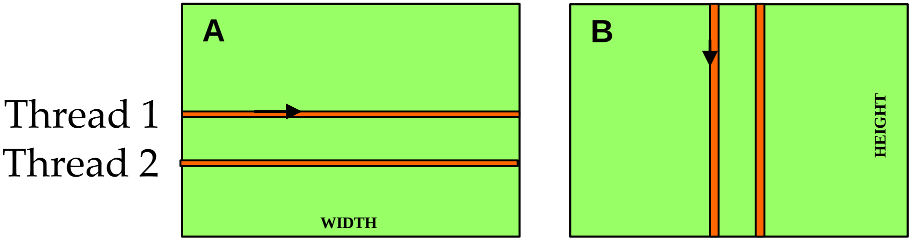
Coalesced Access Example

An image (2D array) is linear in memory space



Coalesced Access Example

One thread per row or one thread per column?



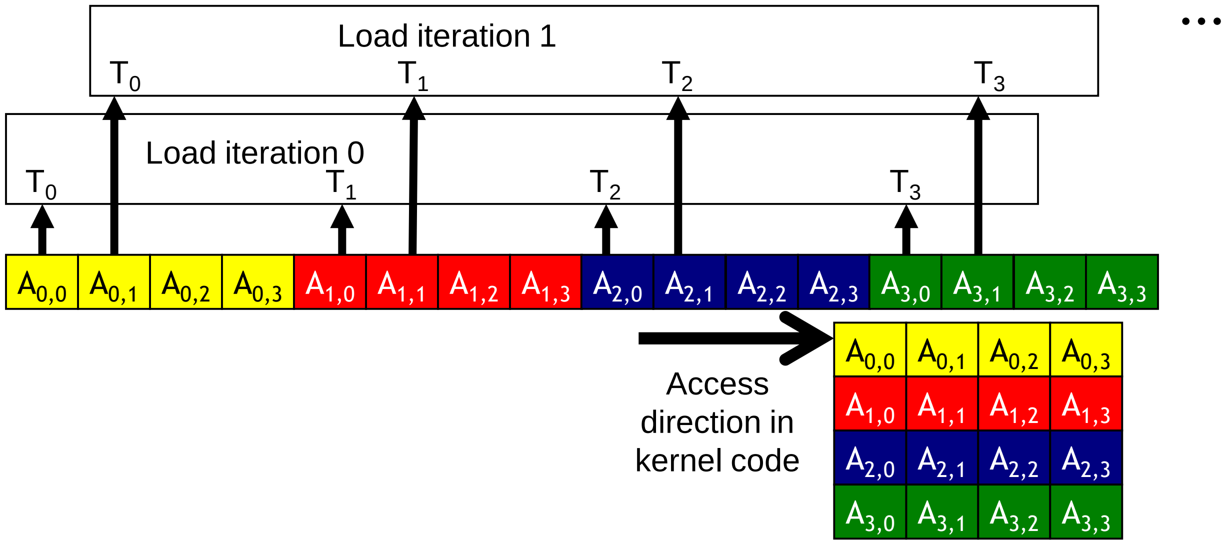
A[row*WITDH+i]   B[i*WIDTH+col]

i is the loop counter in the inner product loop of the kernel code

For A:
row = blockIdx.x*blockDim.x + threadIdx.x
For B:
col = blockIdx.x*blockDim.x + threadIdx.x

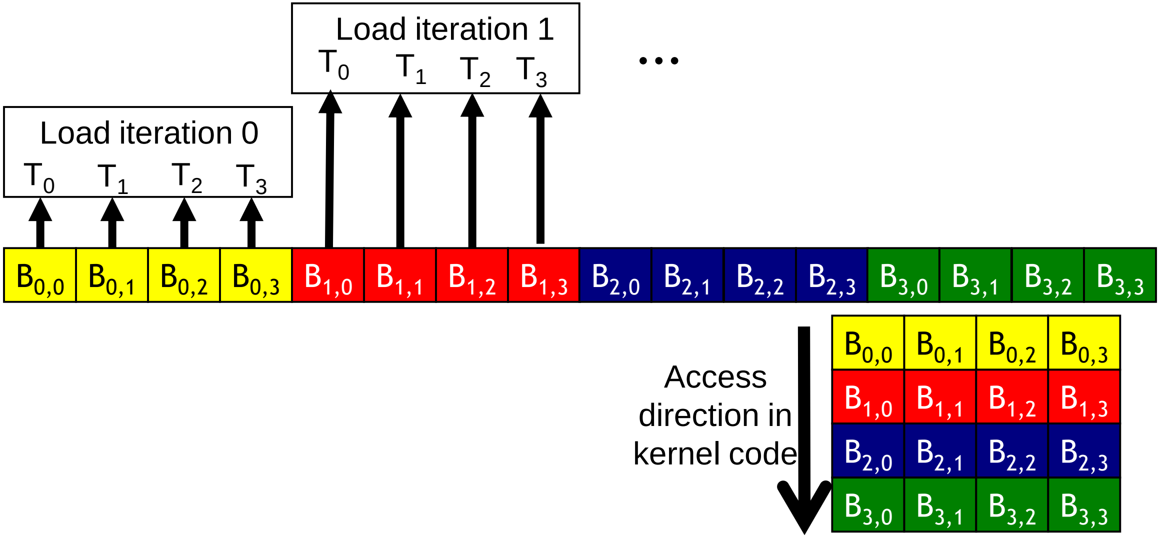
Coalesced Access Example

(A) Accesses are not coalesced



Coalesced Access Example

(B) Accesses are coalesced



Atomics

Atomics

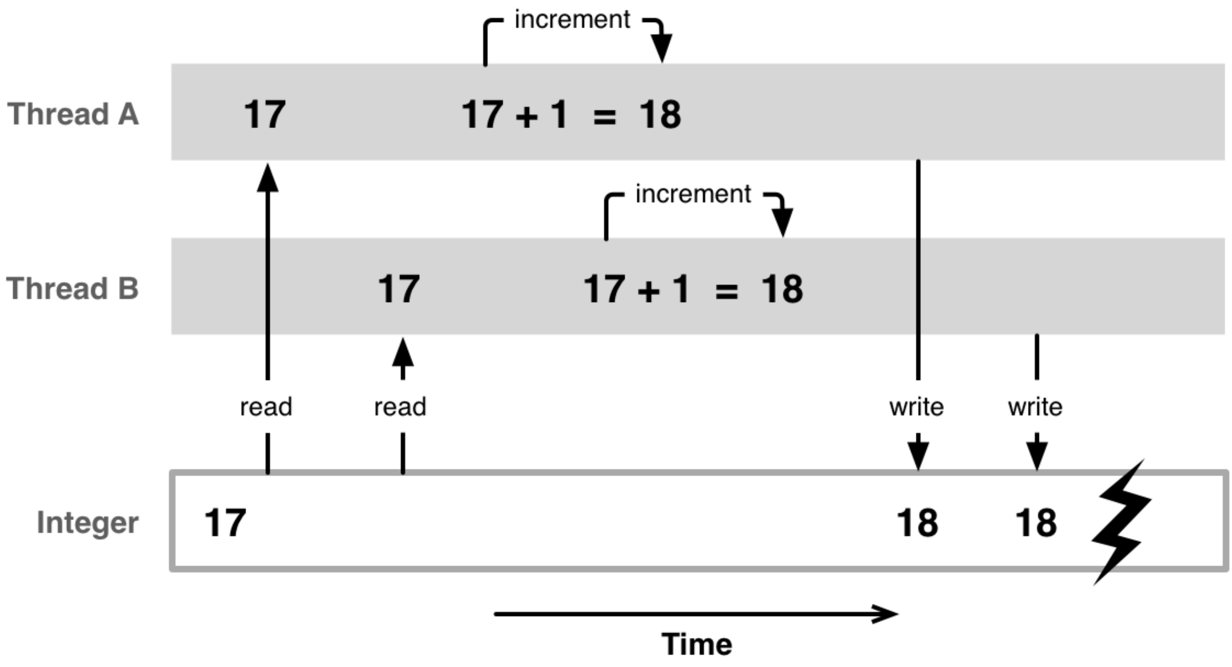
What’s the value of result[0]?

```
x = {3, 5, 2, 1}
result = {0}

__global__ guesswhat(float *x, float *result)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  result[0] += x[tid];
}
```

Atomics

Race Conditions on Memory Accesses



Atomics

Race condition definition

– Race Condition: A computational hazard that arises when the results of a program depend on the timing of uncontrollable events (e.g., the execution order of the threads):

Atomics

Race condition definition

- Race Condition: A computational hazard that arises when the results of a program depend on the timing of uncontrollable events (e.g., the execution order of the threads)
  - For instance, when more than one thread try to access the same memory location concurrently and at least one of them writes to it.

Atomics

What are atomic operations?

- An operation that is capable of reading, modifying, and writing a value back to memory without any other threads interfering it.
  - Guarantee that no race conditions may occur.
  - Impact on performance.
  - Parallel threads (memory access) are forced into a bottleneck in a lock-like fashion so each memory operation is executed one at a time.
  - CUDA provides various atomic functions:
    - atomicAdd()
    - atomicSub()
    - atomicMin()
    - atomicMax()
    - atomicInc()
    - atomicDec()
    - atomicAdd()
    - atomicExch()
    - atomicCAS()
    - atomicAnd(
    - atomicOr()
    - atomicXor()

Atomics

What’s the value of result[0]?

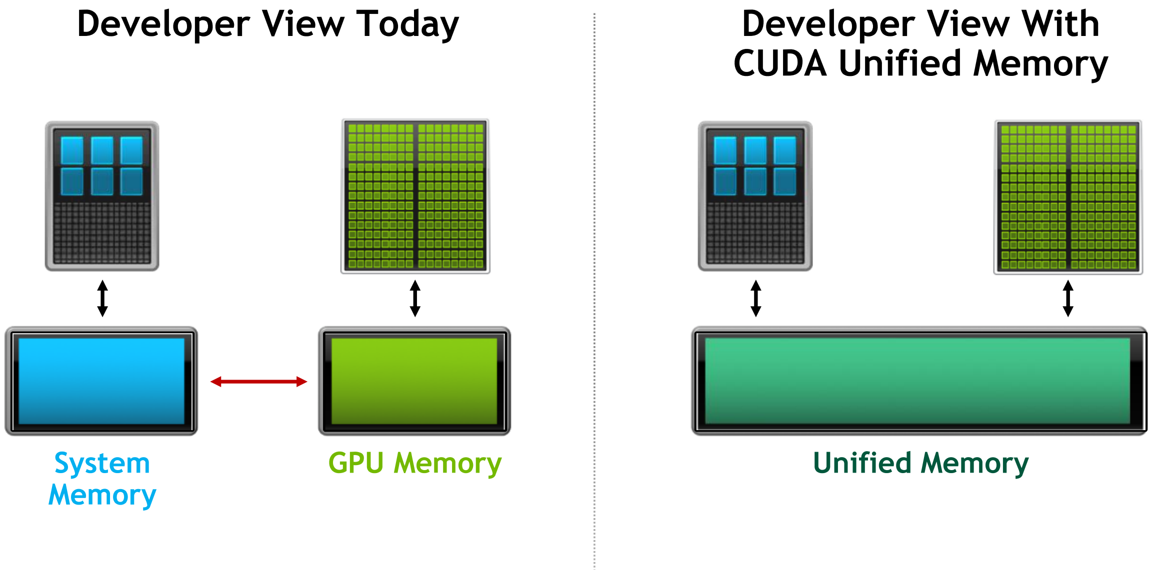
```
x = {3, 5, 2, 1}
result = {0}

__global__ guesswhat(float *x, float *result)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  atomicAdd(result[0], x[tid]);
}
```

Unified Memory

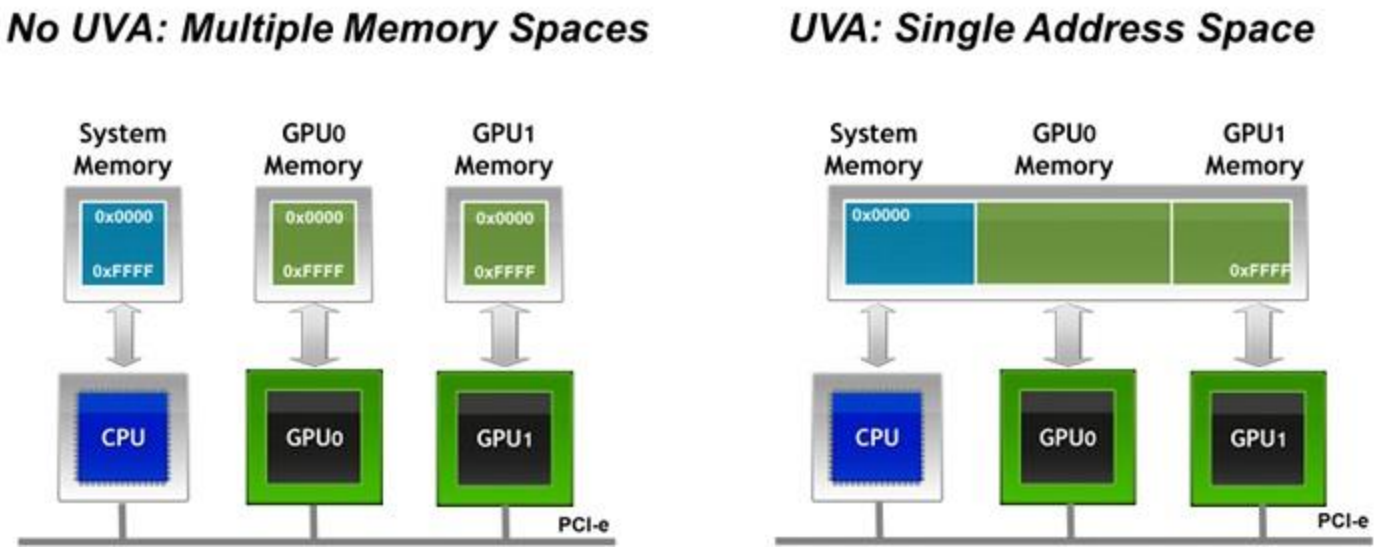
Unified Memory Revisited

Since CUDA 6.0 (and supported GPUs)!



Unified Memory Revisited

UVA (Unified Virtual Addressing)



Unified Memory

What is unified memory?

- A single memory address space accessible from any processor in a system. UVA allows allocating data only once and making it accessible to any GPUs and CPUs at any given time.
  - Replace cudaMalloc calls by cudaMallocManaged.
  - No need for explicit cudaMemcpy calls.
  - Memory is paged so GPU/CPU page faults trigger memory transfers on-demand.
    - GPUs prior to Pascal generation can't page fault! All data is migrated before a kernel launch just in case it is needed.

Unified Memory

Performance

- Depends on GPU generation:
  - Significant impact on GPUs before Pascal due to lack of page-fault mechanism.
  - Page Migration Engine on Pascal + Prefetching:
    - http://www.acceleware.com/blog/Unified-Memory-on-Tesla-P100-with-CUDA-8.0
    - “With Pascal GPUs and new CUDA 8.0 APIs, Unified Memory offers simplified programming AND matches performance of explicit memory management”.
- Not comparable to well-crafted application that makes uses of asynchronous copies to overlap computation and data transfer.

Shared Memory

Shared Memory Revisited

Hierarchy

Each SM has a limited amount of Shared memory

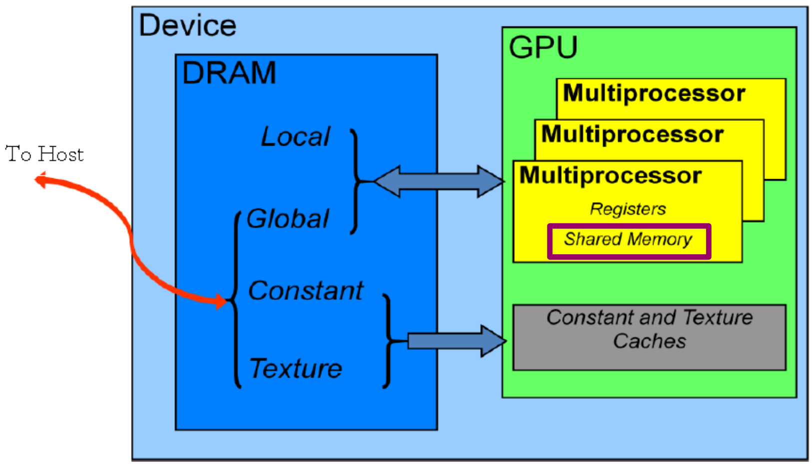
Shared across threads in a block

Low latency

Useful for data re-use

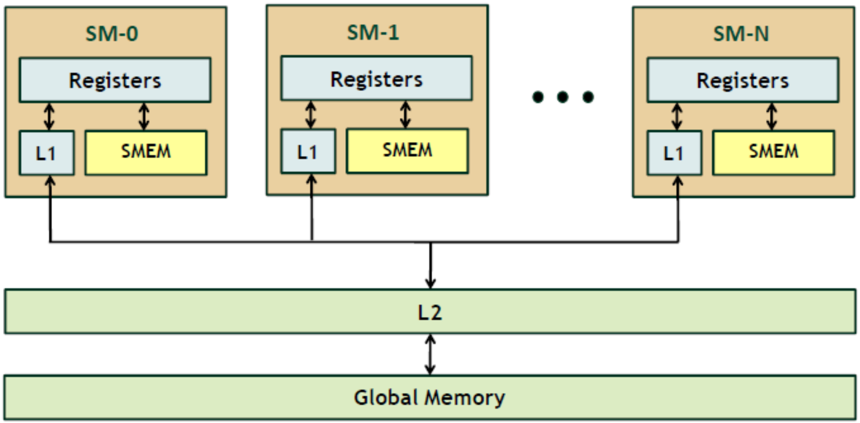
Can be allocated from the GPU (__shared__)

Manual management from CUDA Kernels



Shared Memory Revisited

Zoom on SM



- Serves a way of communicating or synchronizing threads in a block.
- Takes advantage of data reuse to reduce global/local memory accesses.
- Potentially reduces the number of registers needed for each thread.

Shared Memory Revisited

Limitations on GTX1080 Pascal

For each SM:

64 K registers

96 KiB shared memory

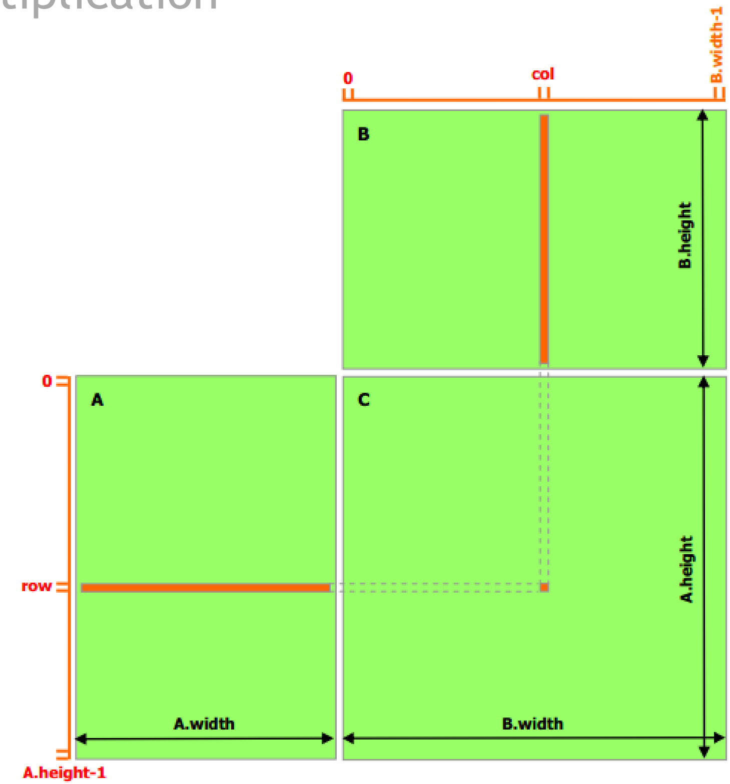
48 KiB L1 cache

16 KiB constant cache

2048 threads

Shared Memory

Matrix multiplication



Shared Memory

Matrix multiplication

```
__global__ void matrix_mul_kernel(float* Md, float* Nd, float* Pd, const int cWidth) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx_ = blockIdx.x;  int by_ = blockIdx.y;
    int tx_ = threadIdx.x;  int ty_ = threadIdx.y;

    int row_ = by_ * TILE_WIDTH + ty;
    int col_ = bx_ * TILE_WIDTH + tx;

    float p_value_ = 0.0f;
    for (int m = 0; m < cWidth / TILE_WIDTH; ++m) {
        Mds[ty_][tx_] = Md[row_ * cWidth + (m * TILE_WIDTH + tx_)];
        Nds[ty_][tx_] = Nd[(m * TILE_WIDTH + ty_) * cWidth + col_];
        __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k)
             p_value_ += Mds[ty_][k] * Nds[k][tx_];
       __syncthreads();
    }

    Pd[row_ * cWidth + col_] = p_value_;
}
```

CUDA Memory

Advanced Aspects

Thanks for your attention!

Slides created in collaboration with Sergio Orts-Escolano!

Albert García García

agarcia @ dtic.ua.es