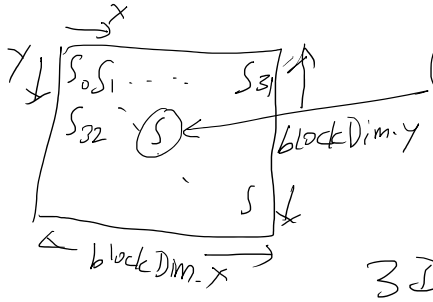
CUDA threading model

Sunday, February 14, 2021 11:50 AM

- Threads, thread blocks, and execution grid
- Thread to data mapping
  - Use CUDA intrinsic variables to locate thread within the execution grid

S : GPU thread (executes kernel code)

Thread block: 1D, 2D, 3D



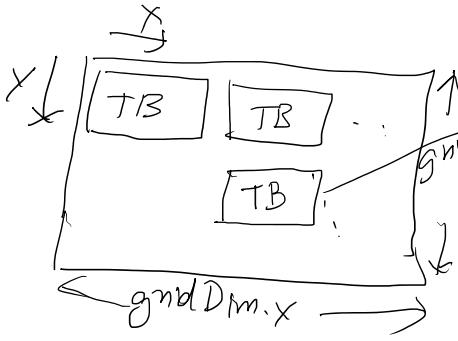
(threadIdx.x, threadIdx.y)

blockDim.x : # of threads in X direction
blockDim.y : # of threads in Y direction

3D : threadIdx.z } additional variables for 3D
blockDim.z } thread block

Execution grid : 1D or 2D

TB : thread block



blockIdx.x
blockIdx.y

gridDim.x : # of TBs in X
gridDim.y : # of TBs in Y

Vector sum

Sunday, February 14, 2021 11:52 AM

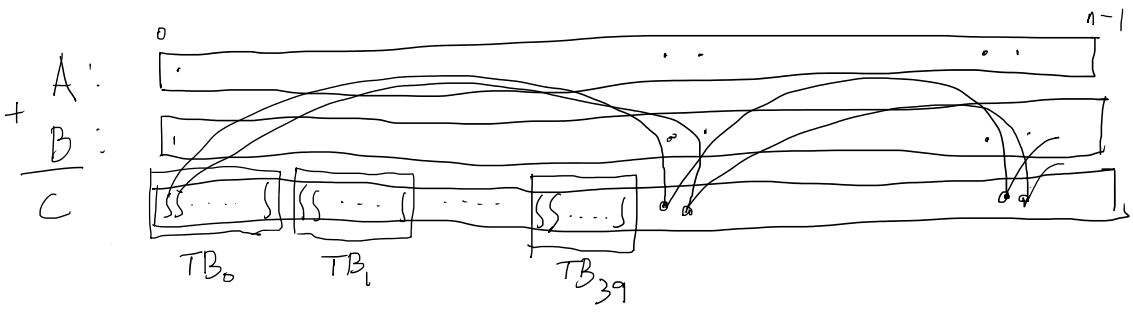
- Code example: vector_sum_large

A, B : vectors of arbitrary size n

C = A + B

thread block : (1024, 1, 1)

grid : (40, 1)



kernel :

```
tid = blockIdx.x * blockDim.x + threadIdx.x;
Stride = GridDim.x * blockDim.x; // Total # of threads in grid

While (tid < n){
    C[tid] = A[tid] + B[tid];
    tid += stride;
}
```

Advantages of this approach:

1) Creation of fewer number of thread blocks (less run-time scheduling overhead)
2) Each thread does more work (calculates multiple output elements)

Simple matrix multiplication (single thread block)

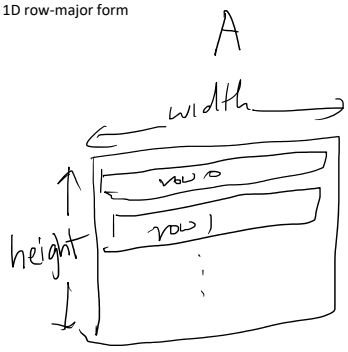
Sunday, February 14, 2021 11:53 AM

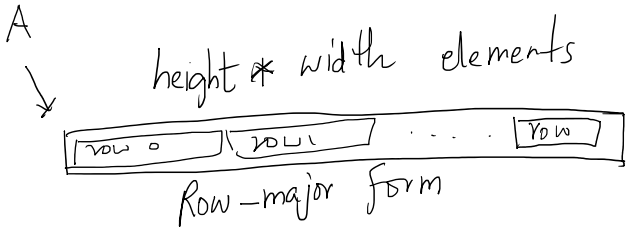
- Remember: 2D structures are laid out in memory in 1D row-major form
- Code example: simple_matrix_multiply

$P = M \times N$

n x n matrices

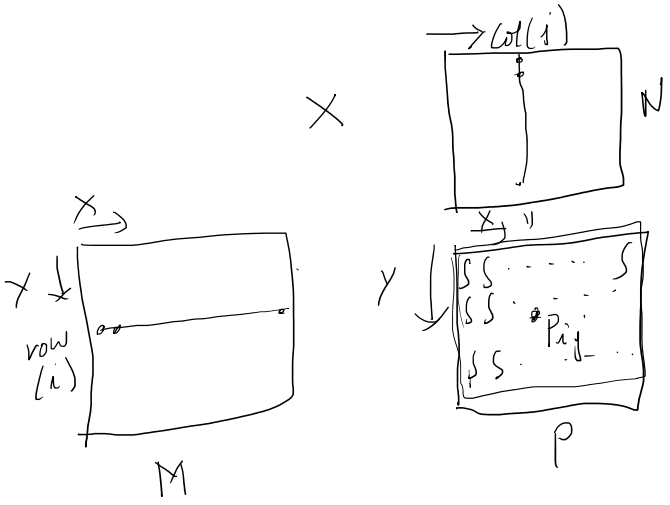
height, width





$A[i][j] = A[i * width + j]$

row, col



1) Decomposition -
   - Each P_{ij} can be calculated in parallel

$P_{ij} = dot(row\ i, column\ j)$

2) Each GPU thread separately computes P_{ij}

Assume 32 x 32 matrices.

thread block = (32, 32, 1)

grid = (1, 1)

Kernel:

```
row = threadIdx.y;
col = threadIdx.x;
P_ij = dot(row, col)
```

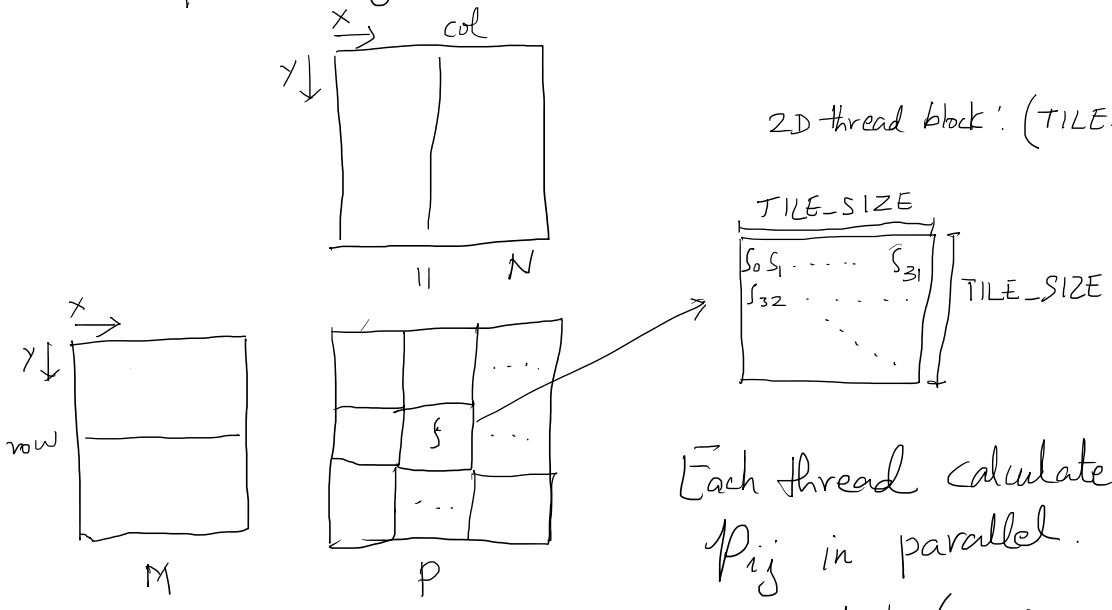
Tiled matrix multiplication

Sunday, February 14, 2021 11:54 AM

- Code example: tiled_matrix_multiply
- Remember:
  - Kernel launch on the device is asynchronous
    - Control returns to the host immediately while the kernel is running on the device
    - Use *cudaDeviceSynchronize()* to synchronize execution on the host and device

$P = M \times N$

$n \times n$ matrices of arbitrary dimensions.



2D thread block: (TILE_SIZE, TILE_SIZE)

Each thread calculates element P_{ij} in parallel.

$P_{ij} = dot(row_i, column_j)$

Thread block: (TILE_SIZE, TILE_SIZE)

grid: $\left(\left\lceil \frac{n}{TILE\_SIZE} \right\rceil, \left\lceil \frac{n}{TILE\_SIZE} \right\rceil\right)$ n: dimension of the matrix

Kernel:

```
row = blockIdx.y * blockDim.y + threadIdx.y;
col = blockIdx.x * blockDim.x + threadIdx.x;
if (row < n) && (col < n)
    dot(row, col);
```

Performance analysis of matrix multiplication

Sunday, February 14, 2021 11:54 AM

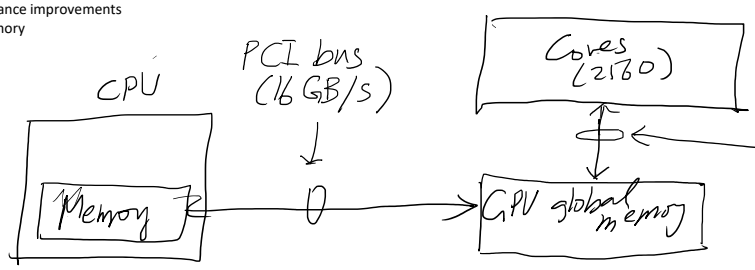
- Low arithmetic/operational intensity
  - Kernel is memory bound
  - Processing loop in the kernel code:

```
P_temp = 0.0;
for (int k = 0; k < matrix_size; k++) {
        M_element = M[matrix_size * row_number + k]; /* Row elements. */
        N_element = N[matrix_size * k + column_number]; /* Column elements. */
        P_temp += M_element * N_element;
}
```

- Large number of redundant loads from GPU global memory

- Key steps to achieving significant performance improvements
  - Coalesced access to GPU global memory
  - Judicious use of shared memory

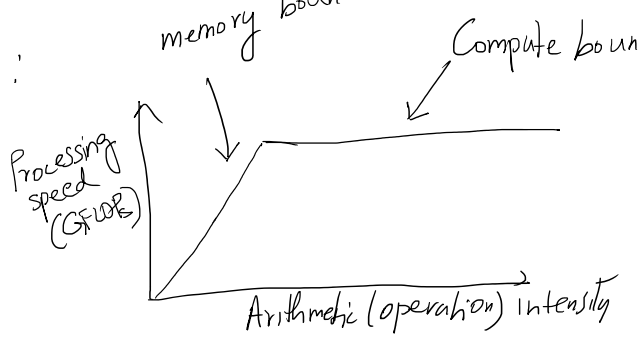
On the 1080 GTX:
2560 cores operating at ~1.6 GHz
8.9 TFLOPs.



320 GBytes/sec

Since single-precision floating point is 4 bytes,
320/4 GFPoperands/s
80 GFPoperands/s

Roofline model:

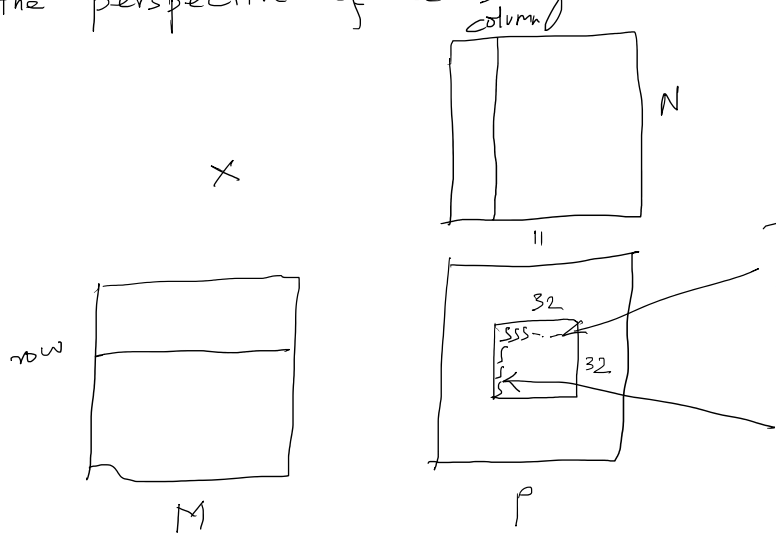


Arithmetic intensity of the multiplication kernel $= \frac{\#\text{ FP operations}}{\#\text{ Bytes transferred between main memory and core}} = \frac{1\text{ mult} + 1\text{ add}}{4 + 4\text{ bytes}} = \frac{1}{4}\ \text{ops/bytes}$

$$\text{Achievable FLOPs} = \min(\text{max FLOPs}, \text{arithmetic intensity} \times \text{memory bandwidth})$$
$$= \min\left(8900, \frac{1}{4} \times 320\right)$$
$$= \min(8900, 80)$$
$$= 80\ \text{GFLOPs}$$

Redundant loads from GPU global memory:

From the perspective of a single thread block (assume 32x32 thread block):



Threads within a thread block operate independent of each other

Threads along this row load each row element from N 32 times

Threads along this column load each column element from N 32 times.

Same issue with other threads in the thread block.

From the perspective of a thread block:

- Each row element is loaded 32 times.
- Each column element is loaded 32 times

That is 31 redundant loads.

Instruction scheduling

Sunday, February 14, 2021 12:28 PM

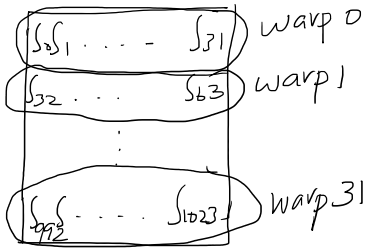
- Each thread block is executed as warps, which are the indivisible scheduling units within streaming multi-processors
  - Warp size: 32 threads
  - Number of threads per warp is an internal NVIDIA implementation decision, not part of the CUDA programming model
  - Example: if a thread block has 1024 threads, that translates to 32 warps.
- Thread IDs within a warp are consecutive and increasing. Warp 0 starts with thread 0
- All threads in a warp execute the same instruction (SIMD style)
- Do not rely on any ordering between warps; if there are dependencies between threads within the thread block, use synchronization to get correct results

→ Threads within a warp execute in lock step
⇒ Same program counter value to fetch the instruction

1D thread block:
(1024, 1, 1) for example:

The 1024 threads are organized as 32 warps, each containing 32 threads.

2D thread block:



Branch divergence

Sunday, February 14, 2021 12:31 PM

- Threads within a single warp take different paths
- Different execution paths taken by threads in a warp are serialized, that is traversed one at a time

Example of divergence:

If (threadIdx.x > 2) {

}
Threads 0, 1, and 2 follow a different path than the rest of the threads in the first warp

Example without divergence:

If (threadIdx.x/WARP_SIZE > 2) {

}

Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

If thread execution diverges, use a barrier to get execution back in sync

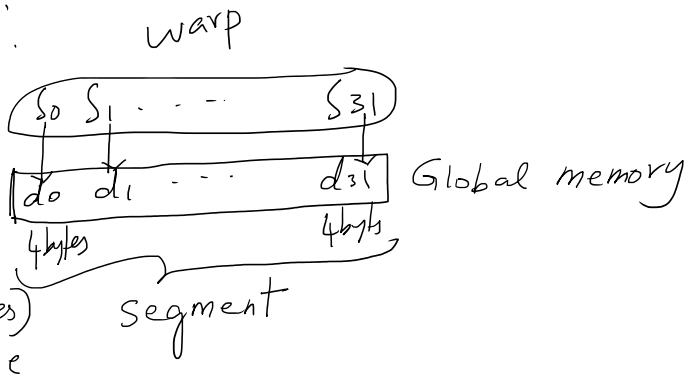
→ For example,

```
:
if (threadIdx.x == 0){
    // some code
}
__syncthreads(); // Other threads block at this barrier
```

Memory coalescing

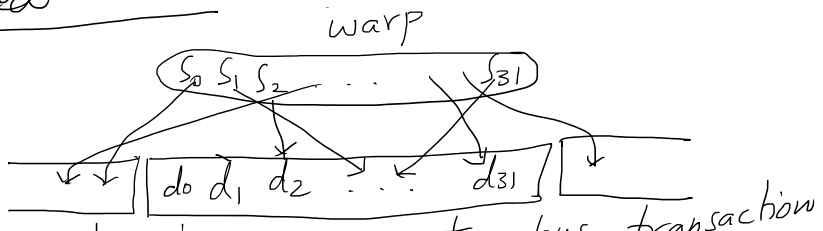
Sunday, February 14, 2021 12:31 PM

- Access to global memory is slow (~200 cycles)
- Coalesced access to global memory can improve transfer speed to cores
- Fully coalesced access:



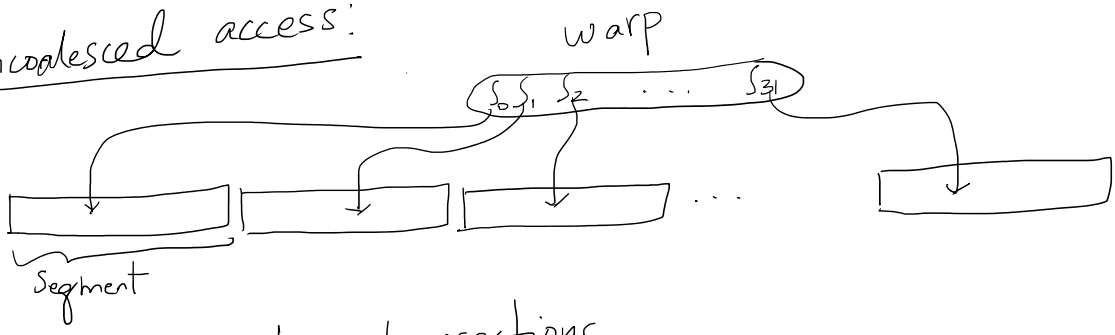
Data in segment (128 Bytes) is sent to core in one bus transaction.

- Somewhat coalesced access:



- Each segment is sent via a separate bus transaction
  ⇒ 3 bus transactions in above example

- Uncoalesced access:



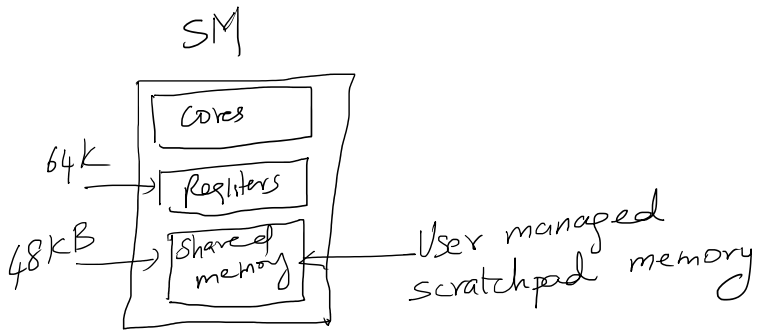
- Need 32 bus transactions

General design rule for better performance

- Threads within a warp have a memory-access stride as close to 1 as possible.

Use of shared memory on each SM

Sunday, February 14, 2021 12:29 PM



General strategy:

1) Allocate shared memory required for the thread block
2) Threads within thread block populate shared memory
3) Barrier sync to ensure that shared memory is populated.
4) Calculate using contents of shared memory

Important:

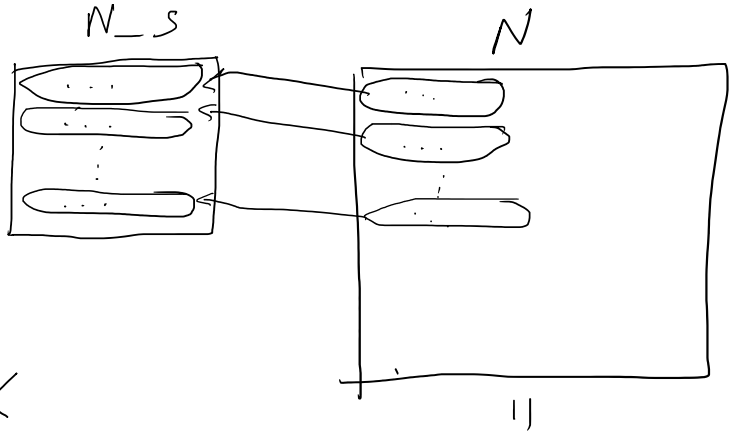
1) Shared memory is not persistent accross kernel calls
   ⇒ must be populated during each kernel call
2) Shared memory contents is not "zeroed" upon allocation
   ⇒ may contain garbage
   ⇒ must be populated correctly, including padding with zeros if necessary

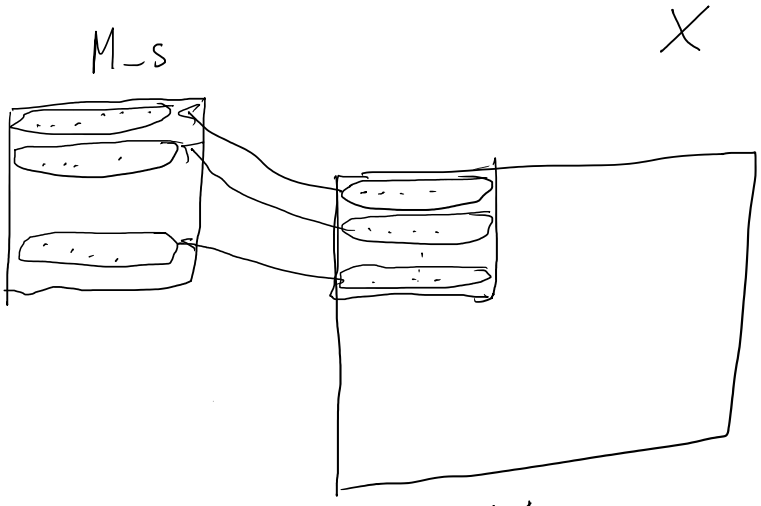
Matrix multiplication using shared memory

Sunday, February 14, 2021 12:29 PM

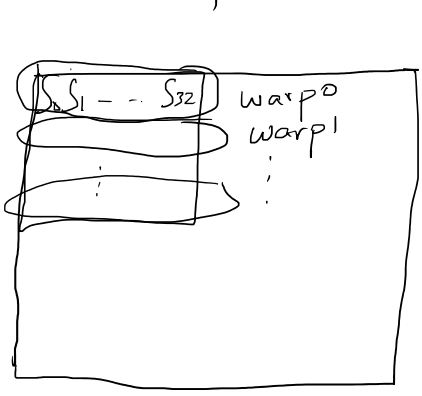
- Code example: tiled_matrix_multiply_shm

$P = M \times N$





X



M

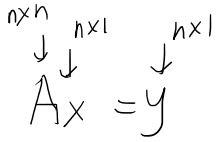
M_s and N_s are shared-memory allocation.

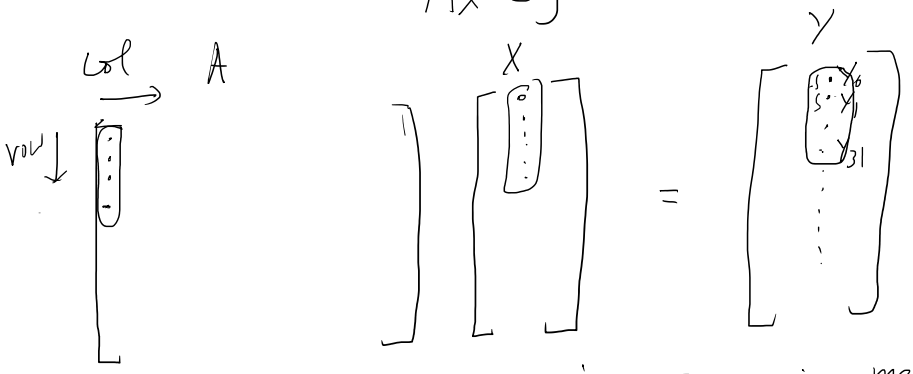
- A thread warp loads row elements from M into M_s
- Same thread warp loads column elements from N in N_s
- calculate partial result using M_s and N_s
- repeat

Impact of memory coalescing: Matrix-vector multiplication

Thursday, February 18, 2021 9:12 AM

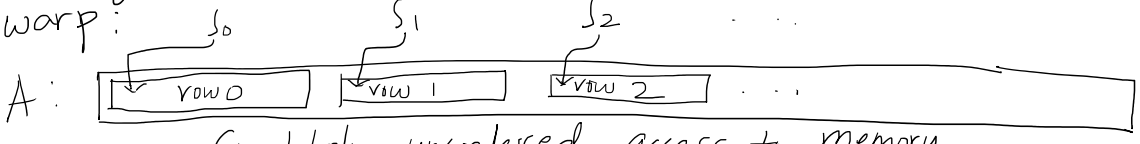
- Code example: matrix_vector_multiply





- Each Y_i can be calculated in parallel
- So create (1, 1024, 1) thread block.

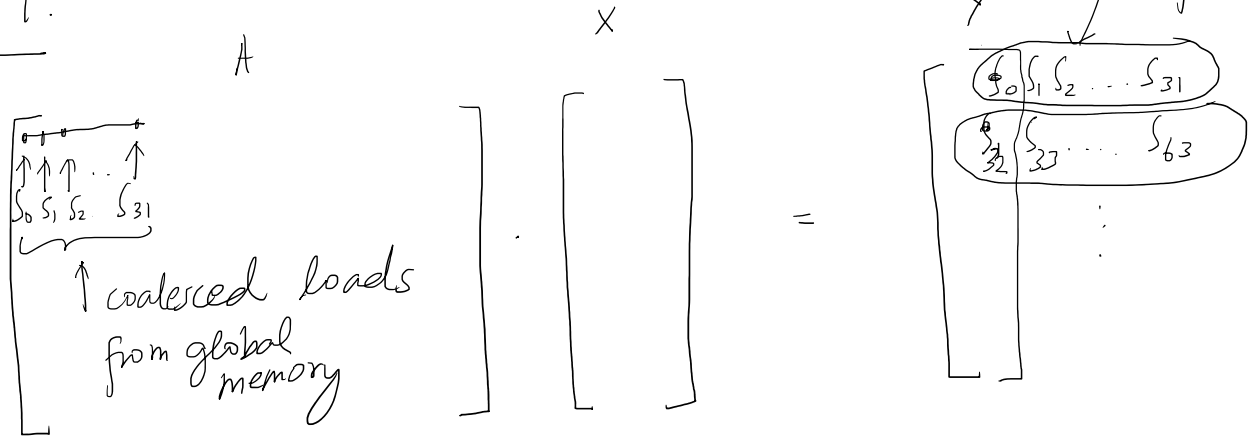
- A is laid out in row-major form in memory
- Memory access pattern to column elements of A by a thread warp:



- Completely uncoalesced access to memory

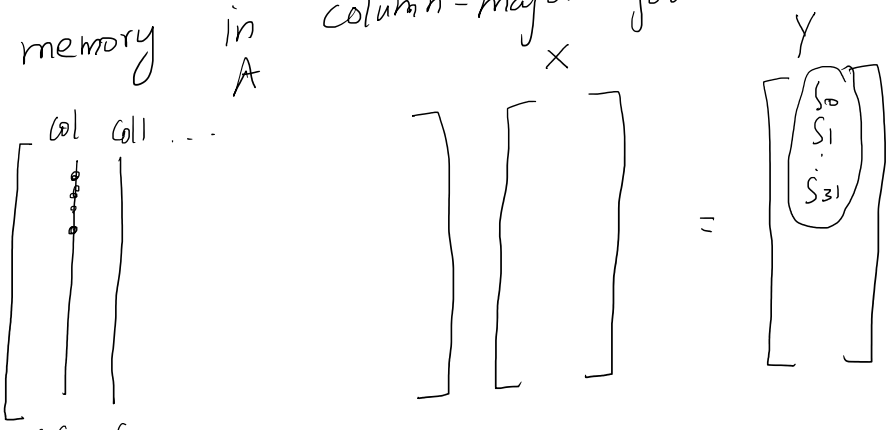
Solutions:

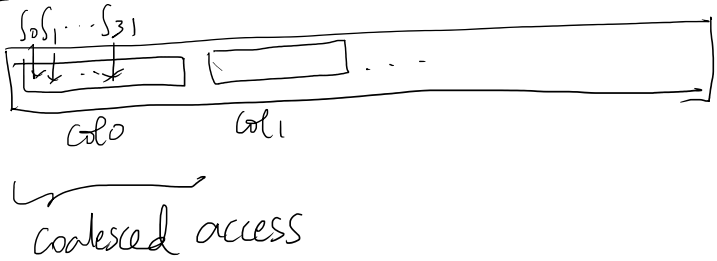
Version 1:



Version 2:

Store A in memory in column-major form





$A_{col-major}$:

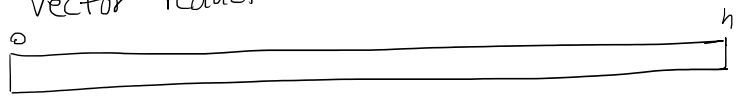
Coalesced access

- If A is stored in row-major form originally, re-format layout to column major
- Transfer $A_{col-major}$ to GPU
- Write kernel code assuming matrix is laid out in column major form.

Chunking versus striding patterns

Thursday, February 18, 2021 2:26 PM

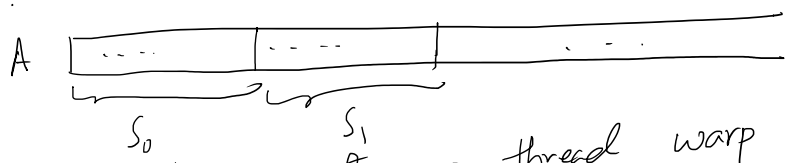
Example: vector reduction



$value = \sum_i a_i$

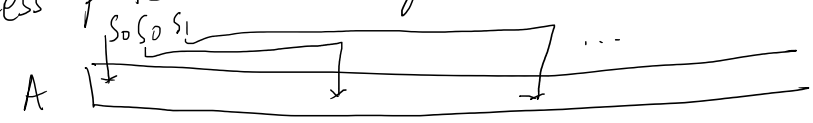
Options for parallelization:

Chunking: . Each thread S is responsible for calculating the partial sum for its chunk



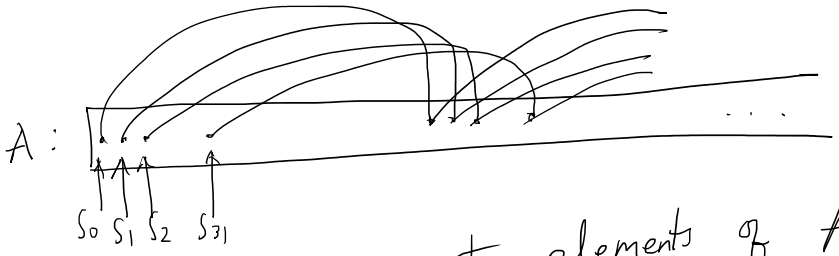
Suppose $S_0 S_1 \ldots$ belong to a thread warp

The access pattern to array A:



} Uncoalesced access to GPU global memory.

Striding:



Coalesced access to elements of A.

Basic idea:

- Chunking is preferred on CPU (maximizes cache locality)
- Striding is preferred on GPU (results in coalesced access to GPU global memory)