Pinned memory

Tuesday, March 2, 2021 8:35 AM

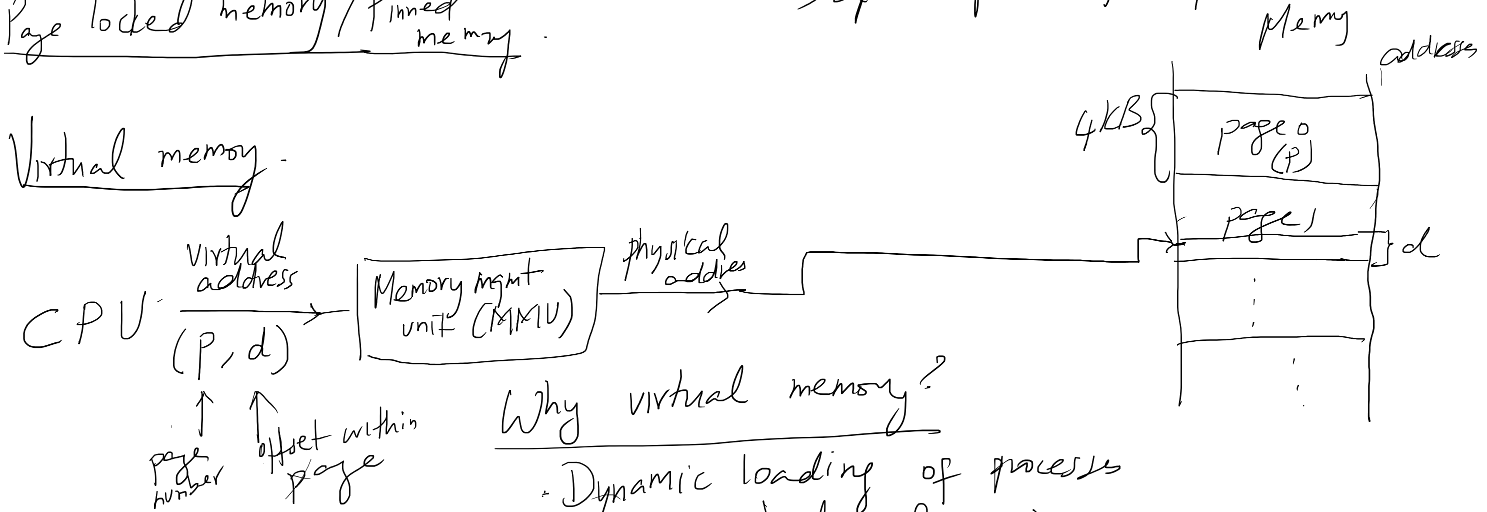
- Pinned or page-locked memory improves achieved bandwidth between host and device
    - Enables zero-copy transfers
- Use *cudaHostAlloc()* to request pinned pages from the Operating System
- Code example: *page_locked_memory*

bottleneck
CPU
GPU

↳ Speed up transfer of data?

· Page locked memory / Pinned memory.

· Virtual memory.

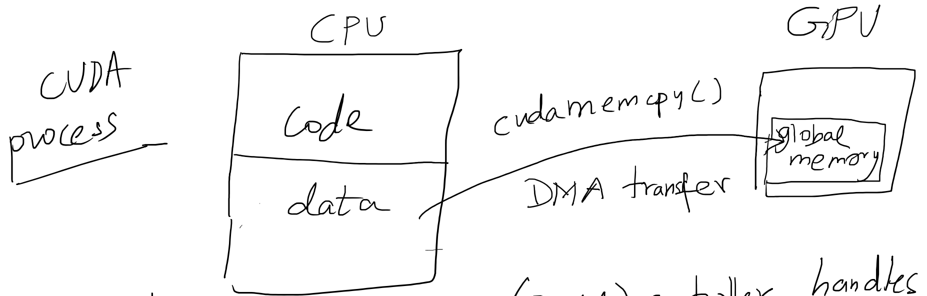


Why virtual memory?

· Dynamic loading of processes
↳ demand paging

Address space protection

Overhead due to address translation

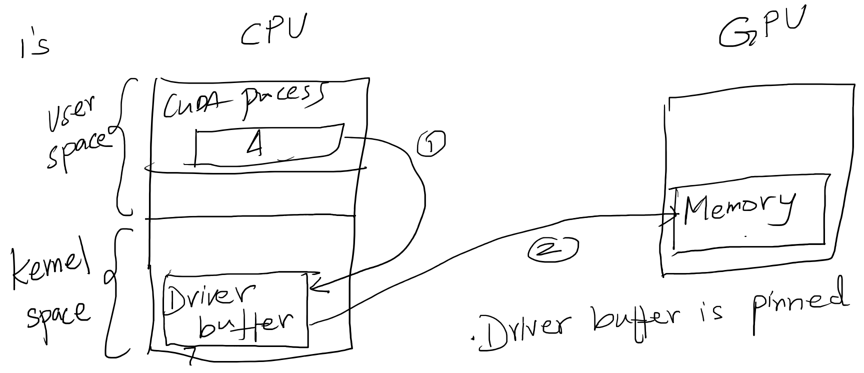


Direct memory access (DMA) controller handles data transfer

· src address (host)
· dest address (device)
· # of bytes to transfer

→ DMA happens in background
→ Frees up CPU for other jobs.

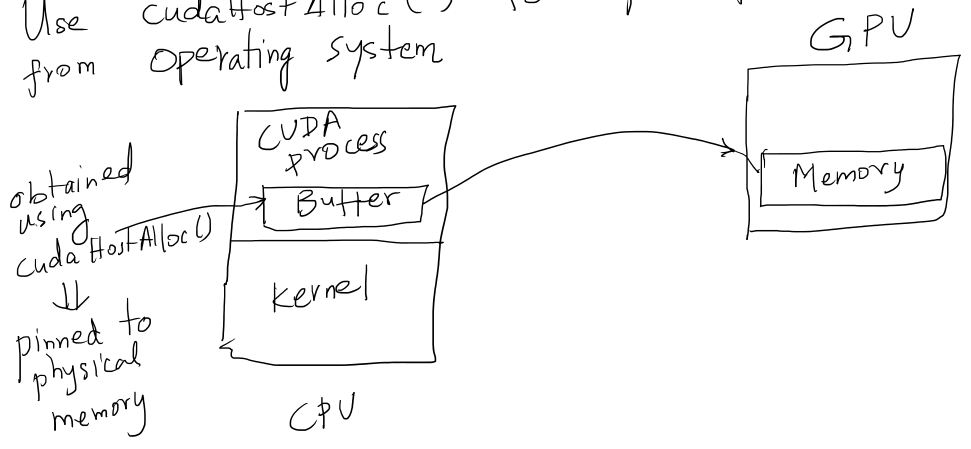
· DMA transfer is a two step process



· Driver buffer is pinned ~~to~~ memory

Zero-copy transfer:

Use cudaHostAlloc( ) to request pinned memory from operating system



Upto 2x speedup of CPU ⟷ GPU data transfer

CUDA streams

Tuesday, March 2, 2021 8:35 AM

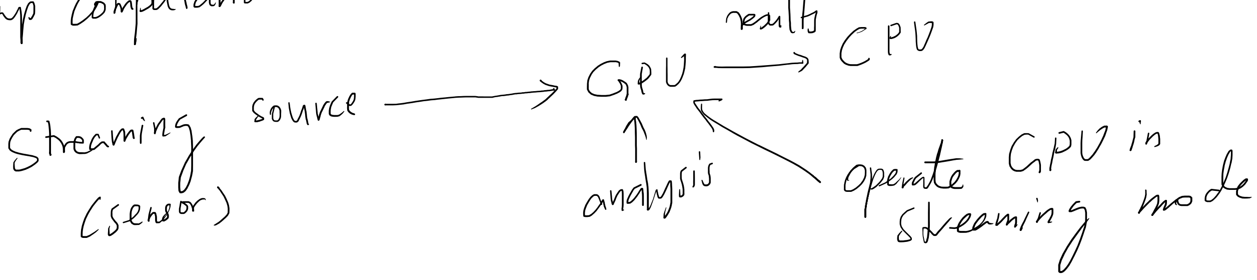
Code example: *streams*

So far: data is transferred in full before computation begins

$$\begin{tikzcd} \text{CPU} \arrow[r] & \text{GPU} \end{tikzcd}$$

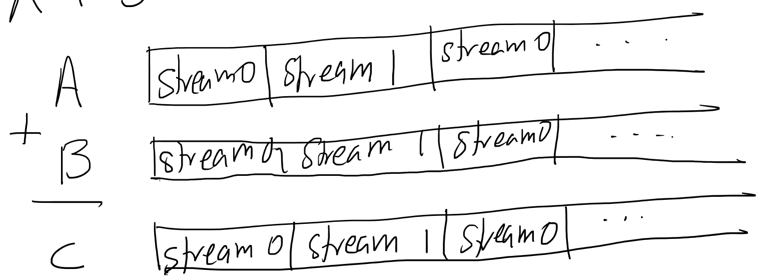
Streaming model:

1) Transfer data in smaller chunks
2) GPU can start processing chunk
3) Continue to transfer additional chunks to GPU
4) Overlap computation + communication

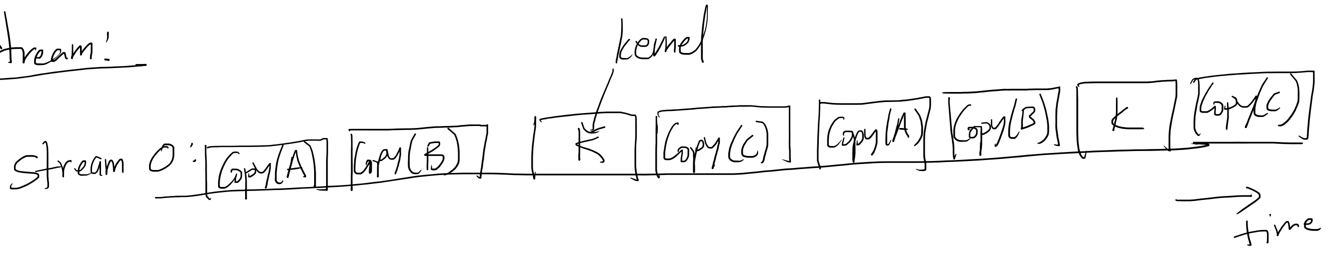


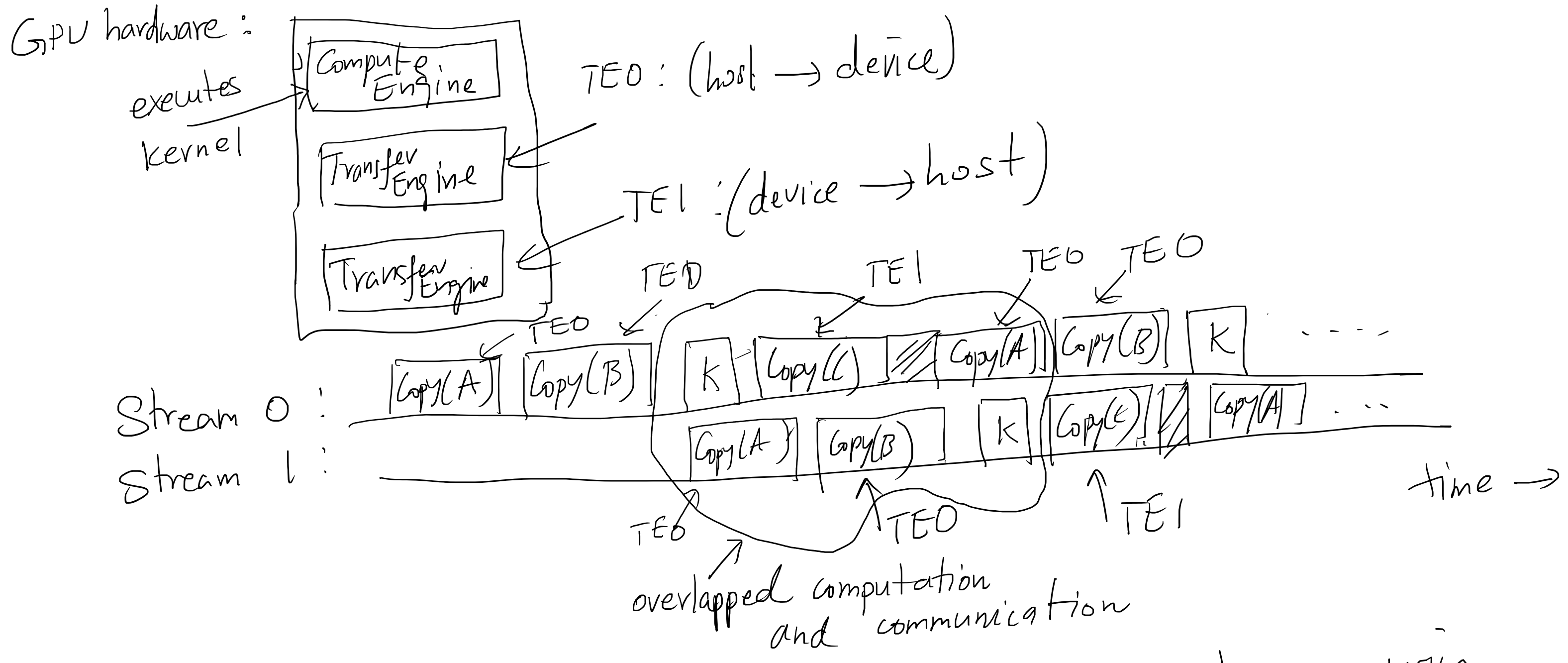
Vector addition:

$C = A + B$



Single stream:





• All operations are done in async fashion using pinned memory.

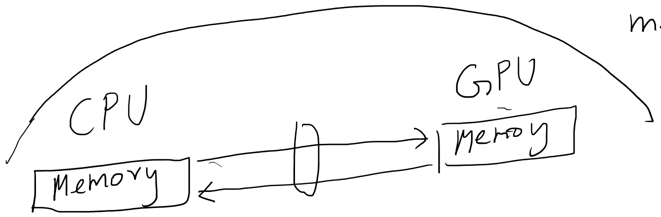
Unified memory

Tuesday, March 2, 2021 8:35 AM

https://devblogs.nvidia.com/unified-memory-cuda-beginners/

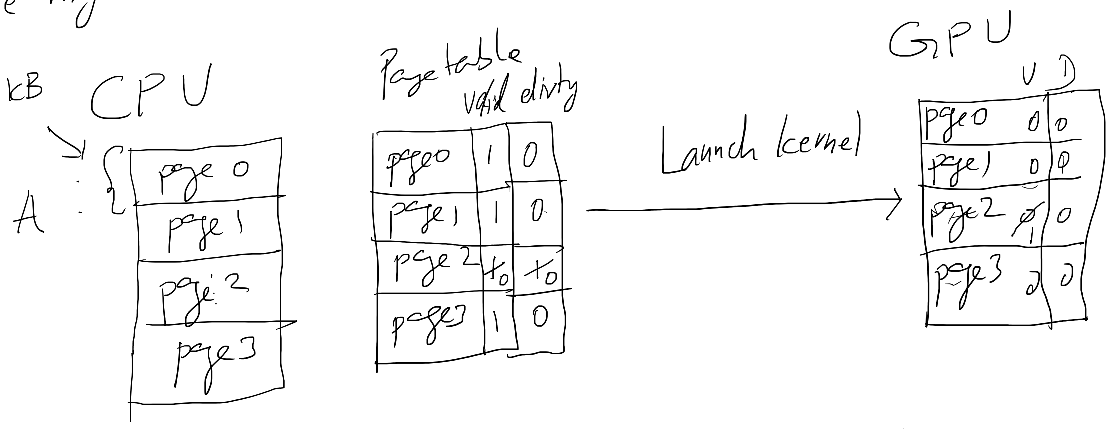
https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

Code example: *unified_mem*



Data transfer between the memories is managed by the CUDA runtime

Page migration between CPU and GPU:



Suppose kernel accesses data in page 2:

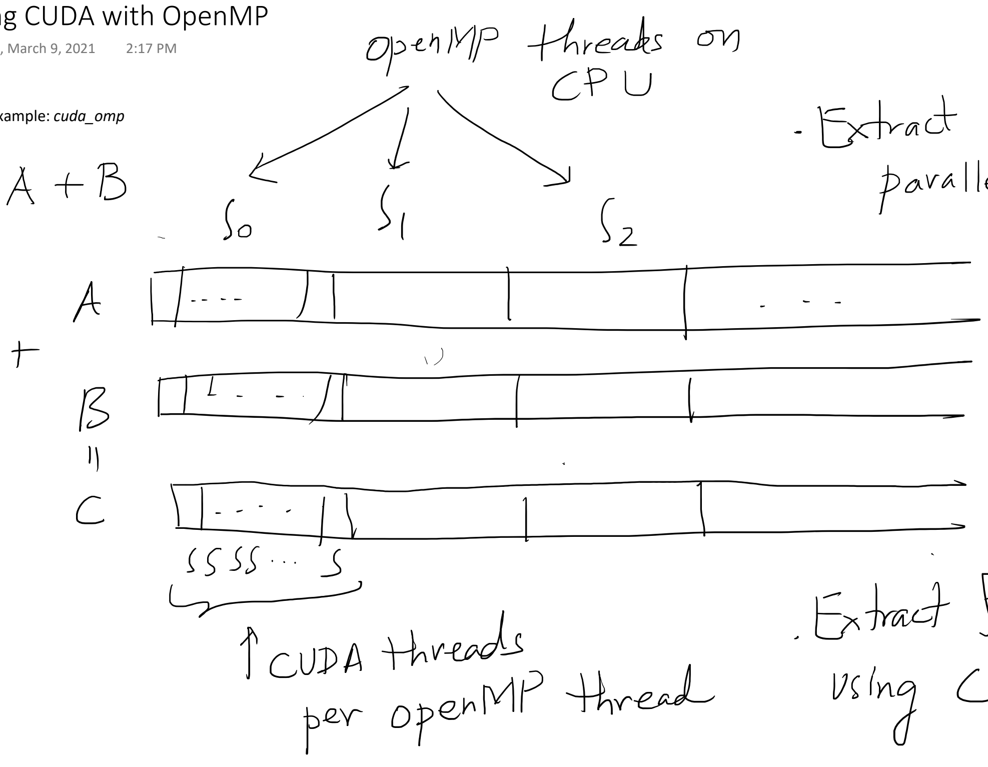
1) Page fault
2) Page 2 migrated from CPU to GPU
3) Page tables updated on GPU and CPU

Using CUDA with OpenMP

Tuesday, March 9, 2021 2:17 PM

Code example: *cuda_omp*

C = A + B



- Extract coarse-grained parallelism using OpenMP
- Extract fine-grained parallelism using CUDA

Concurrent kernel execution

Tuesday, March 9, 2021 2:17 PM

Code example: *concurrent_kernels*

SIMD fashion $\longrightarrow$ single kernel runs at any given time

MIMD fashion $\longrightarrow$ launchs $\underbrace{\text{concurrent kernels}}$ on the GPU

$\Downarrow$

<u>different kernels</u>

CUDA BLAS examples

Sunday, May 31, 2020 12:04 PM

https://docs.nvidia.com/cuda/cublas/index.html

Single precision AX plus Y (SAXPY)

Vector dot product (dot)

Single precision general matrix-matrix multiplication (sgemm)

Single precision general matrix-vector multiplication (sgemv)

See CUBLAS documentation on BBLearn

BLAS : Basic linear algebra subroutines

Vector processing on the CPU

Thursday, March 11, 2021 8:43 AM

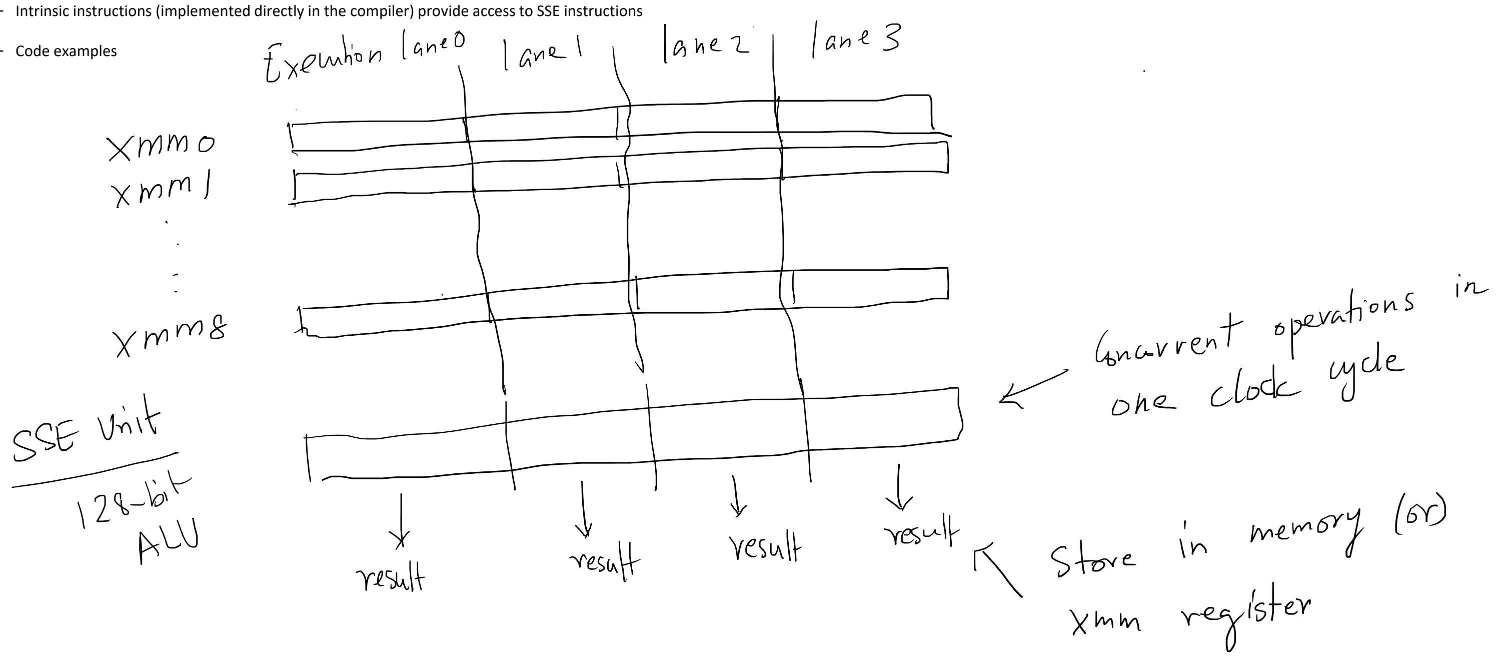
- *Streaming SIMD Extensions* (*SSE*) is a single instruction, multiple data (SIMD) instruction set extension to the x86 architectureop

- SSE2:
  - Eight new 128-bit registers known as XMM0 through XMM7
  - XMM registers can be configured to hold
    - Four 32-bit single-precision floating-point numbers or
    - Two 64-bit double-precision floating-point numbers or
    - Two 64-bit integers or
    - Four 32-bit integers or
    - Eight 16-bit short integers or
    - Sixteen 8-bit bytes or characters
  - The ISA supports both scalar and packed scalar (vector) instructions
    - Memory-to-register/register-to-memory/register-to-register data movement
    - Arithmetic operations (add. Subtract, multiply, divide)
    - Bit-wise logical operations
    - Compare and shuffle
    - …

- More recent developments: *Advanced Vector Extensions* (*AVX*)
  - AVX uses sixteen YMM registers, each of width 256 bits, to perform SIMD operations
    - Eight 32-bit single-precision floating point numbers or
    - Four 64-bit double-precision floating point numbers
  - AVX-512 uses 32 512-bit registers (ZMM0-ZMM31) for SIMD operations

- Intrinsic instructions (implemented directly in the compiler) provide access to SSE instructions

- Code examples



Gaussian elimination on the GPU

Thursday, March 11, 2021 8:57 AM

$$\underset{n\times n}{\underset{\uparrow}{A}}\, \underset{n\times 1}{\underset{\uparrow}{x}} = \underset{n\times 1}{\underset{\uparrow}{b}} \Longrightarrow \underset{\substack{\text{upper}\\\text{triangular}\\\text{matrix}}}{\underset{\uparrow}{U}}x = b'$$

$$\overset{U}{\begin{bmatrix} 1 & \cdots & \cdots & \cdots \\ & 1 & \cdots & \vdots \\ & & \ddots & \vdots \\ \text{zeros} & & 1 & a \\ & & & 1 \end{bmatrix}} \overset{x}{\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}} \uparrow = \overset{b'}{\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{bmatrix}}$$

Gaussian elimination

$$3x + 5y + 2z = 19$$
$$2x + 3y + z = 11$$
$$x + 2y + 2z = 11$$

$$A\begin{bmatrix} 3 & 5 & 2 \\ 2 & 3 & 1 \\ 1 & 2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \overset{b}{\begin{bmatrix} 19 \\ 11 \\ 11 \end{bmatrix}} \begin{matrix} R1 \\ R2 \\ R3 \end{matrix}$$

Back substitution

$$x_{n-1} = b_{n-1}$$
$$x_{n-2} + a\cdot x_{n-1} = b_{n-2}$$
$$x_{n-2} = b_{n-2} - a\cdot x_{n-1}$$
$$\vdots$$

1) Division step: $R1/3$

$$A' = \begin{bmatrix} 1 & 5/3 & 2/3 \\ 2 & 3 & 1 \\ 1 & 2 & 2 \end{bmatrix} \qquad b' = \begin{bmatrix} 19/3 \\ 11 \\ 11 \end{bmatrix}$$

2) Elimination step

$$R2 = R2 - 2R1$$
$$R3 = R3 - R1$$

$$\begin{bmatrix} 1 & 5/3 & 2/3 \\ 0 & -1/3 & -1/3 \\ 0 & 1/3 & 4/3 \end{bmatrix} \qquad b' = \begin{bmatrix} 19/3 \\ -5/3 \\ 14/3 \end{bmatrix} \begin{matrix} R1 \\ R2 \\ R3 \end{matrix}$$

3) Division step: $R2/-1/3$

$$A' = \begin{bmatrix} 1 & 5/3 & 2/3 \\ 0 & 1 & 1 \\ 0 & 1/3 & 4/3 \end{bmatrix} \qquad b' = \begin{bmatrix} 19/3 \\ 5 \\ 14/3 \end{bmatrix}$$

4) Elimination step:

$$R3 = R3 - \frac{1}{3}R2 \qquad A' = \begin{bmatrix} 1 & 5/3 & 2/3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad b' = \begin{bmatrix} 19/5 \\ 5 \\ 3 \end{bmatrix}$$

5) Division:

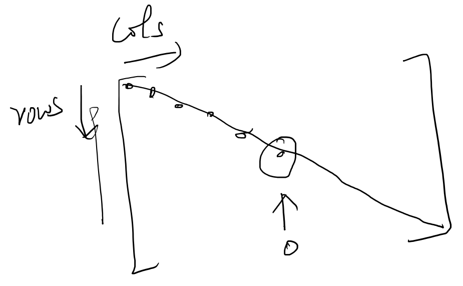
$$U = \begin{bmatrix} 1 & 5/3 & 2/3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad b' = \begin{bmatrix} 19/5 \\ 5 \\ 3 \end{bmatrix}$$

Back substitution :

$z = 3$
$y + z = 5$
$y = 5 - 3 = 2$
$x + 5/3\,y + 2/3\,z = 19/5$
$x = 1$

· Numerical stability

· Not stable



∘ Silent exception

$a/0 \Rightarrow NaN$

$NaN + NaN = NaN$

∘ Exceptions on GPU

- Division by zero

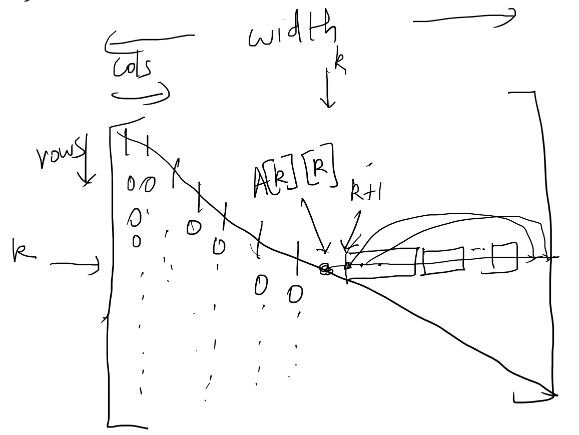
Host-side code :

```
Transfer A to device
for (k=0; k<n ; k++) {   // Process rows
    Call division kernel (A, k, n)
    sync.
    Call elimination kernel (A, k, n)
    sync.
}
Read A from device.
```

Device code:

- Design considerations:
  1) Fewer number of thread blocks
  2) Coalesced access to global memory

Division:

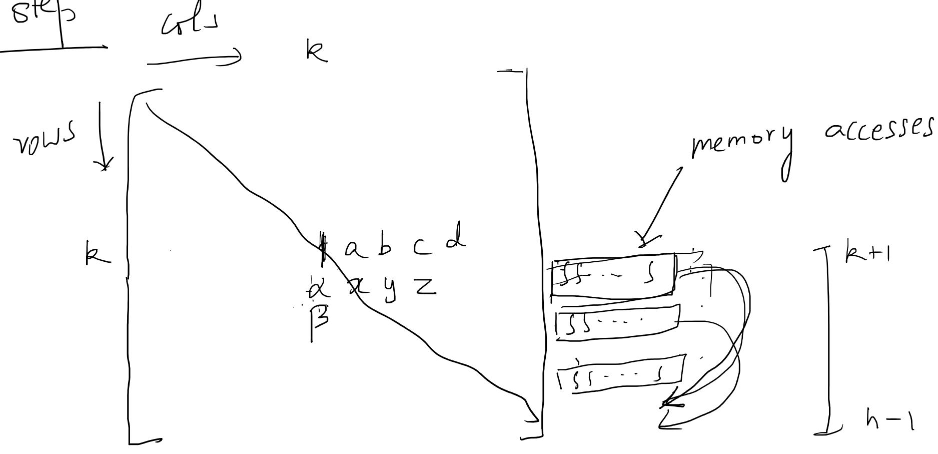


# thread blocks : m
thread block size: (p, 1, 1)
grid (m, 1)

Kernel code:

```
tid = blockIdx.x * blockDim.x
        + threadIdx.x
stride = gridDim.x * blockDim.x
pivot = A[k*width+k]
while (tid < n) {
    elem = A[k*width+k+1+tid]
    elem = elem/pivot
    tid = tid+stride
}
if (blockIdx.x == 0 &&
    threadIdx.x == 0)
    A[k * width + k] = 1
```

Elimination step



Kernel :

```
row = k+1 + blockIdx.x

while ( row < n ) {        // stride accross rows
    tid = threadIdx.x
    while( tid < n ) {     // stride along row
        x = A[row * width + k+1 + tid]
        x = x - α . a      // a ← A[k * width + k+1 + tid]
                           // α ← A[row * width + k]

        tid = tid + blockDim.x
    }
    sync
    if (threadIdx.x == 0)
        α = 0

    row = row + gridDim.x
}
```