Detecting loop-level parallelism

Tuesday, February 9, 2021 8:51 AM

- Examples of no loop-carried dependencies
  - Jacobi iterations
  - SAXPY

```
for(i=0; i<N; i++)
    y[i] = a*x[i] + y[i];
```

iter 0: `y[0] = ax[0] + y[0]` ... iter 1: `y[1] = ax[1] + y[1]` ...

- Examples of loop-carried dependencies
  - Fibonacci series

0 1 1 2 3 5 8 13 ...

```
fib[0] = 0
fib[1] = 1
#pragma omp parallel for
for(i=2; i<N; i++)
    fib[i] = fib[i-1] + fib[i-2];
```

OpenMP does not check for loop carried dependencies

not correct

- How to have a compiler (such as *gcc*) automatically detect loop-carried dependencies?
  - Needed to safely perform loop unrolling
  - The GCD test
- The programmer must get involved for harder loops --- rewrite the loop appropriately
  - Example: infinite series for approximating \pi

a, b : constants
c, d : constants

```
for(i=0; i<n; i++){
    A[ai+b] = ...;
    ... = A[ci+d]
}
```

j, k : loop iterations

$j < k$

$j \neq k$

$0 \le j \le n;\ 0 \le k \le n$

iterations

```
j
A[aj+b] = ...;
... = A[cj+d];
...
```

```
k
A[ak+b] = ...;
... = A[ck+d];
...
```

Race condition?

Iter j is writing to a location

Iter k is reading from a location

$aj+b == ck+d$ *

→ Hazard → loop carried dependency

Test: greatest common divisor (GCD)

Test will pass if a loop-carried dependency exists

If a loop-carried dependency exists, GCD(a, c) must divide (d − b)

Examples :

```
for( i=0; i<100; i++)
    X[2i+3] = X[2i] + 5;
```

a = 2, b = 3, c = 2, d = 0

GCD(a, c) = GCD(2, 2) = 2

d − b = 0 − 3 = −3

2 does not divide −3 ⇒ no loop carried dependency.

```
for (i=0; i< 0; i++) {
    a[2i] = b[i];
    c[i] = a[4i+1];
}
```

a = 2, b = 0, c = 4, d = 1

GCD(a, c) = GCD(2, 4) = 2

d − b = 1 − 0 = 1

2 cannot devide 1

Sufficient condition

- Programmer may need to get involved.

$$\pi = 4\left[\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

Serial code:

```
plus_minus = 1;
sum = 0;
for(k=0; k<n; k++) {
    sum += plus_minus (2k+1);
    plus_minus = - plus_minus;
} approx_pi = 4 x sum;
```

⇒ loop carried dependency

```
sum=0;
#pragma omp parallel for reduction(+:sum)
for (k=0; k<n; k++){
    if (k%2 ==0)
        plus_minus =1;
    else  plus_minus =-1;
    sum += plus_minus/(2k+1)
}
approx_pi = 4*sum;
```

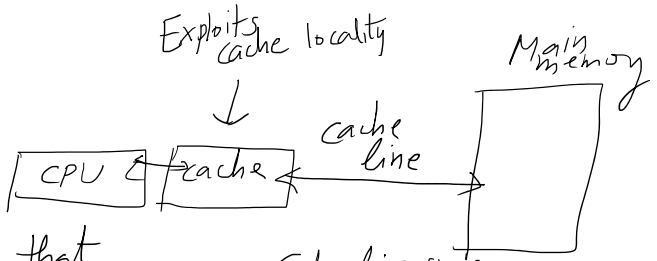
No loop carried dependency.

Loop optimizations

Tuesday, February 9, 2021 8:59 AM

- Approaches aim to:
  - Maximize locality in the cache (memory accesses within a single cache line ideally should have a stride length of 1)
    - Optimize memory access patterns accordingly
  - Reuse data brought into the cache

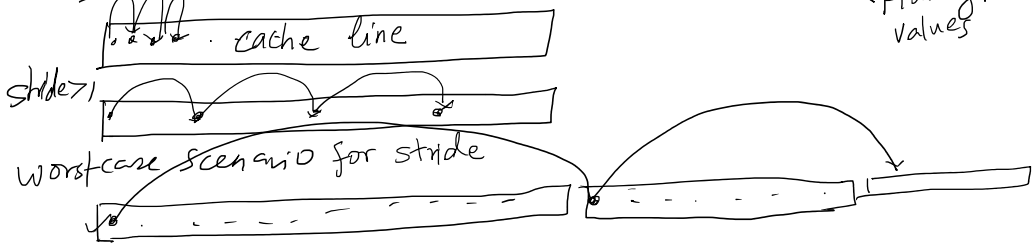
- Examples of loop optimization:
  - Loop fusion
  - Loop fission
  - Loop interchange
  - Loop tiling
    - Tiled matrix transpose
    - Tiles matrix multiplication



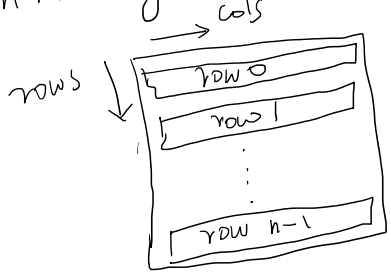
Memory patterns that
1) Maximize/exploit fully

Cache line sizes:
128 bytes = 32 x 4 bytes
256 bytes = 64 x 4 bytes
Floating point values

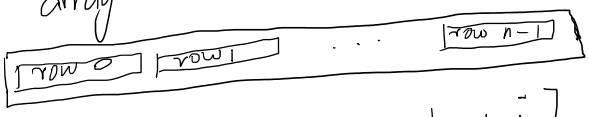
Stride=1 spatial locality



2D structure layout in memory:



1D array in row-major form:



num_cols: # of columns (or) width of matrix

$A[i][j] = A[i \times num\_cols + j]$

Matrix transpose: assume n x n matrix

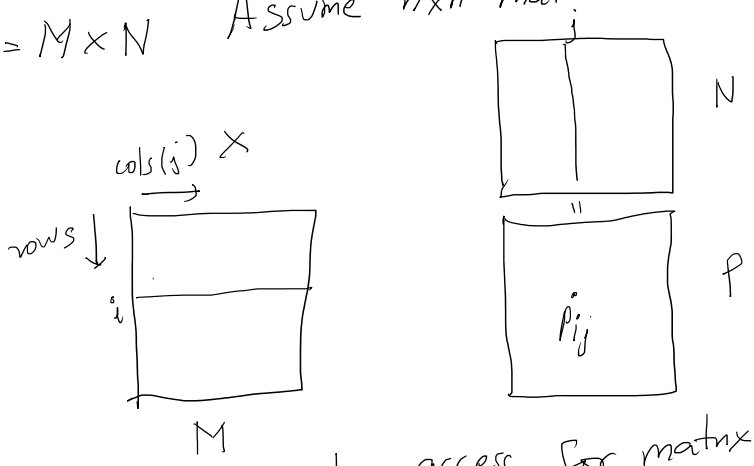
```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        B[i][j] = A[j][i];
```

Memory access along columns of A
- Very large striding of the 1D layout in memory
- Very large number of cache misses

Solution: maximize the reuse of data brought into cache
- Operate on smaller-sized tiles, one at a time (see loop-tiling-c)

Matrix multiplication:

$P = M \times N$ Assume $n \times n$ matrices



P_{ij} = dot product of (ith row of M) and (jth column of N)

Performance problem: Column access for matrix N

↳ Each access to a column element results in a cache miss (due to large stride)

Solution: tiled/blocked matrix multiplication.

Introduction to CUDA

Tuesday, February 9, 2021 9:03 AM

- Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs)

- Enables General Purpose Programming on the GPU (GPGPU)

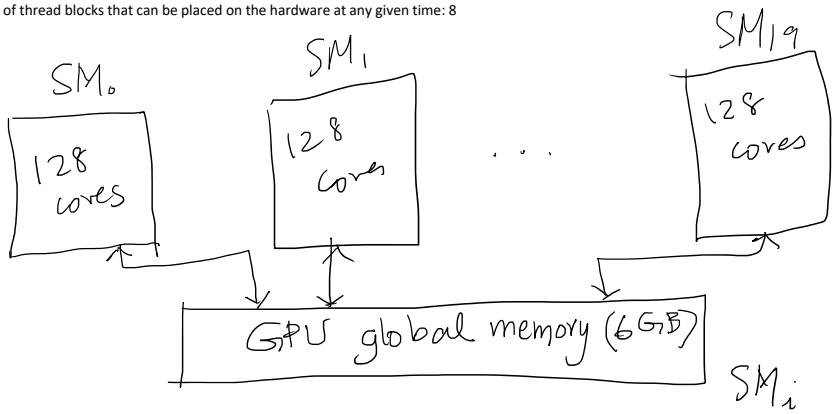
- Contains SPMD extensions for data-parallel programming on the GPU
    - SPMD: Single Program Multiple Data
    - STMD: Single Thread Multiple Data

GPU microarchitecture

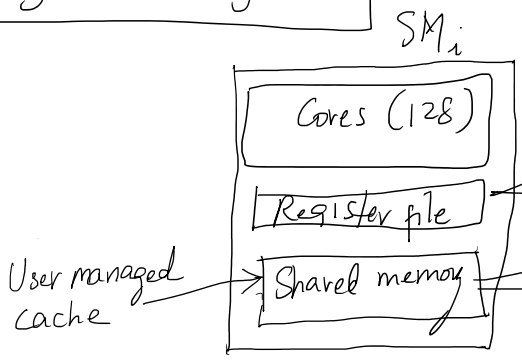
Tuesday, February 9, 2021 9:05 AM

- We will be using the GTX 1080 on xunil-05.coe.drexel.edu
- Feel free to debug and test code on your local GPU (if you have one on your machine)
- The GTX 1080 comprises 2560 cores
  - 20 streaming multiprocessors (SMs), each with 128 cores
  - 64K registers per SM
  - 48 KB of shard memory/cache per SM
  - 6 GB of global memory
  - 64 KB of constant memory
- Maximum number of threads that can be executed by the hardware at any given time: 2048 threads
- Maximum number of thread blocks that can be placed on the hardware at any given time: 8

Cores are organized in hierarchical fashion
- Operates at ~1.6GHz
- Supports single precision and double precision operations
- Supports branch statements



GPU memory hierarchy:



64K registers
latency: 1 cycle

48 KB
latency: 5–10 cycles

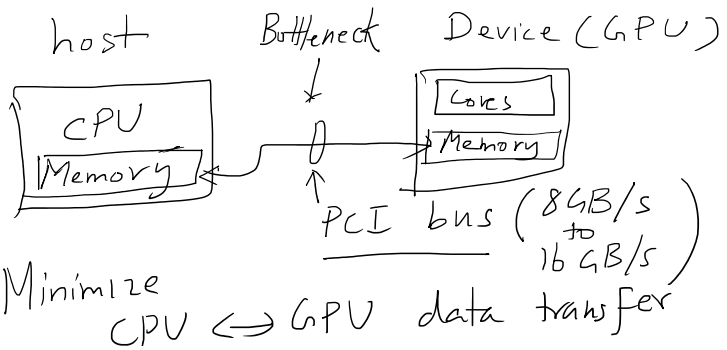
Global memory
latency: 200–300 cycles

Read only memories
- Constant memory → latency: ~25 cycles
- Texture memory → latency: ~25 cycles

The CUDA/C execution model

Tuesday, February 9, 2021 9:09 AM

- Integrated host and GPU program
- Serial or modestly parallel parts in host C code
- Highly parallel parts in GPU SPMD (single program multiple data) kernel C code
- CPU and GPU maintain separate memory maps
  - Use *cudaMemcpy()* to transfer data between CPU and GPU
- Memory allocation on the GPU is persistent across multiple kernel calls
  - *cudaMalloc()* allocates memory on the GPU
  - *cudaFree()* frees previously allocated memory



Minimize CPU ↔ GPU data transfer

Host side

1. Allocate memory on GPU
2. Transfer data to GPU (CPU → GPU)
3. Setup execution grid
4. Launch kernel
5. Transfer data to CPU (CPU ← GPU)
6. Free GPU global memory

Device side

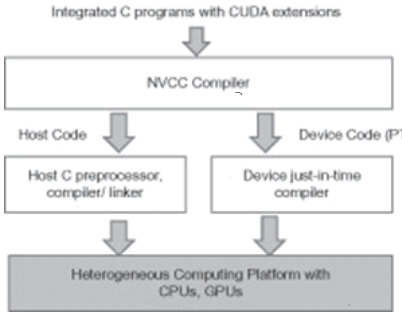
- kernel

GPU Global memory is persistent across multiple kernel calls.

Building the CUDA executable

Tuesday, February 9, 2021 9:16 AM

.cu + .cpp files with SIMD extensions



→ Parallel thread execution ISA

↑ instruction set architecture

CUDA threading model

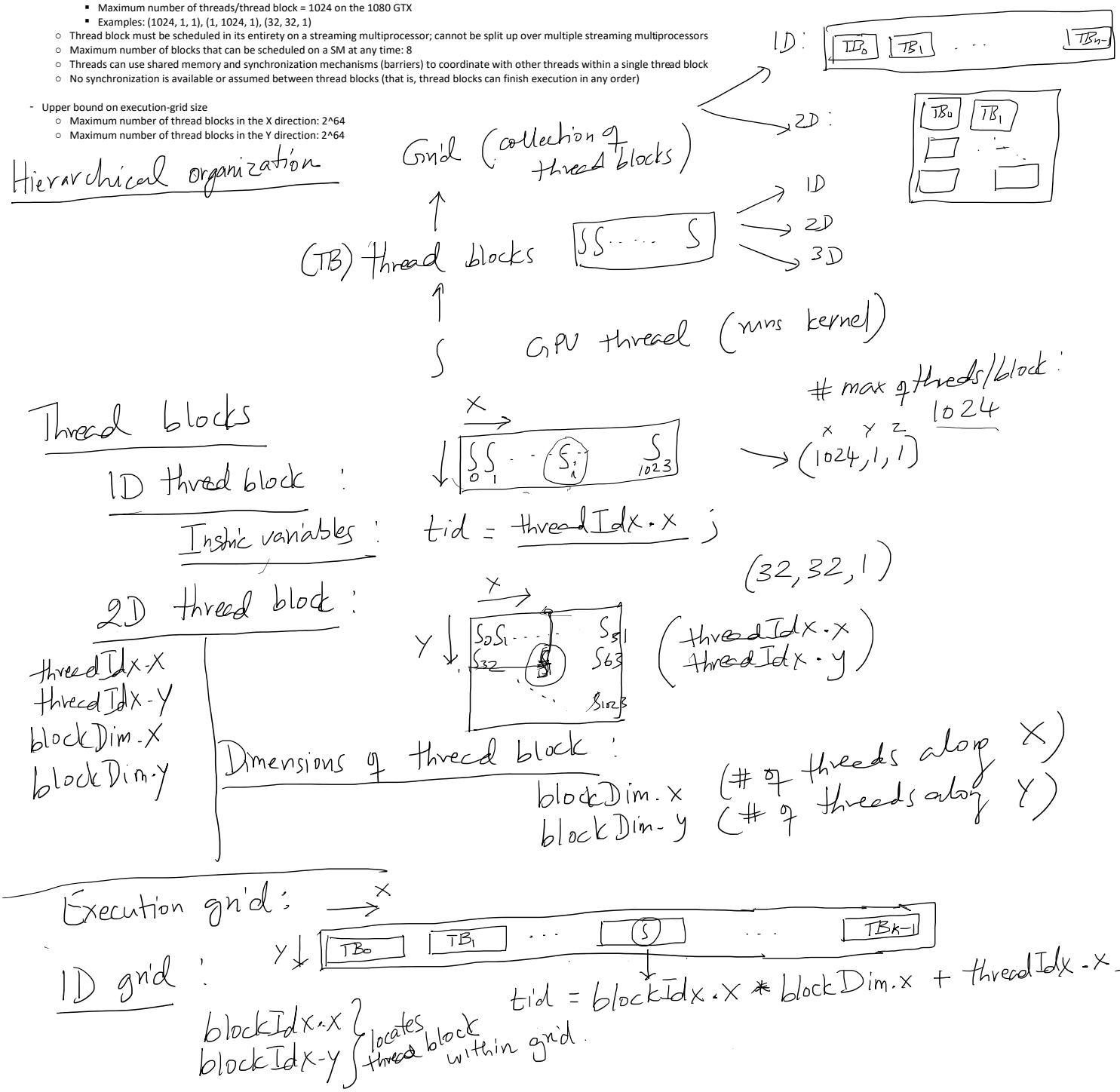
Tuesday, February 9, 2021 9:18 AM

- Hierarchical thread organization
  - Threads
  - Thread blocks
  - Execution grid

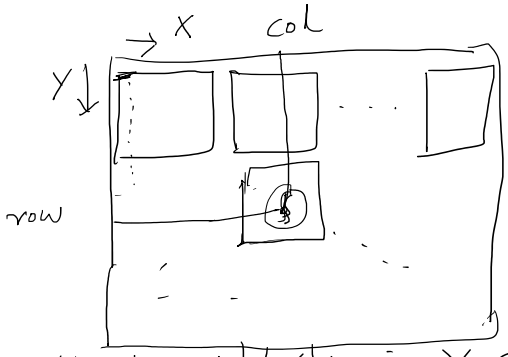
- CUDA provides intrinsic variables to help locate each thread within the execution grid
  - Needed for thread-to-data mapping for your application (think thread ID)

- More on thread blocks
  - Upper bound on thread-block size (CUDA version/GPU architecture dependent)
    - Maximum number of threads/thread block = 1024 on the 1080 GTX
    - Examples: (1024, 1, 1), (1, 1024, 1), (32, 32, 1)
  - Thread block must be scheduled in its entirety on a streaming multiprocessor; cannot be split up over multiple streaming multiprocessors
  - Maximum number of blocks that can be scheduled on a SM at any time: 8
  - Threads can use shared memory and synchronization mechanisms (barriers) to coordinate with other threads within a single thread block
  - No synchronization is available or assumed between thread blocks (that is, thread blocks can finish execution in any order)

- Upper bound on execution-grid size
  - Maximum number of thread blocks in the X direction: 2^64
  - Maximum number of thread blocks in the Y direction: 2^64



2D grid:



row = blockIdx.y * blockDim.y
+ threadIdx.y ;

col = blockIdx.x * bbckDim.x
+ thread Idx.x ;

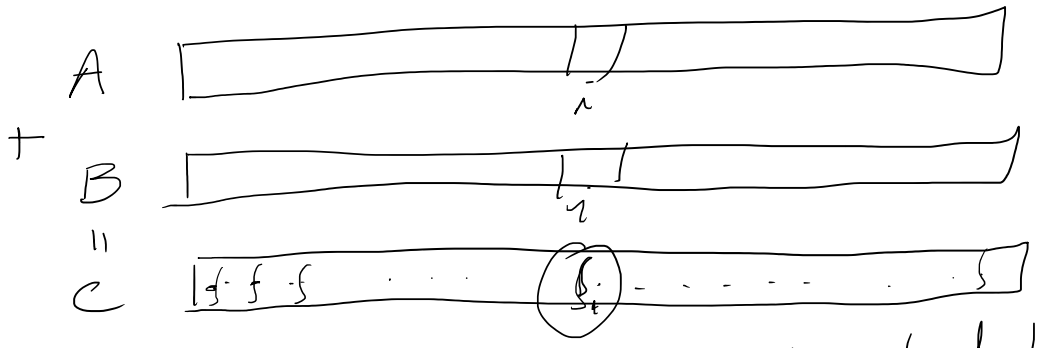
gridDim.x (# thread blocks in X direction)
gridDim.y (# thread blocks in Y direction)

Example: vector addition

Tuesday, February 9, 2021 11:35 AM

A, B : vectors

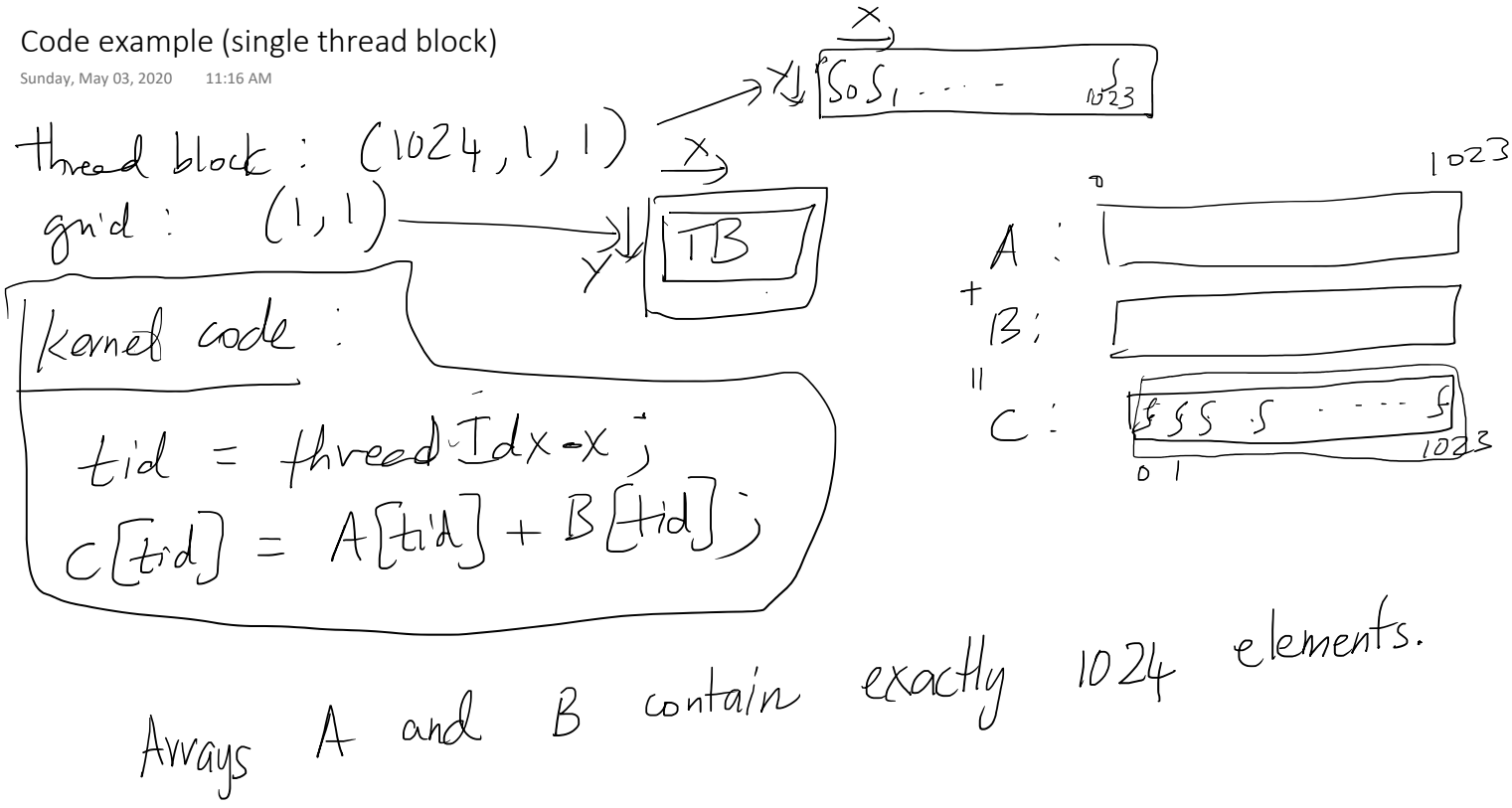
C = A + B



1) Strategy: each output element in C is computed by a thread.
2) If am a thread, what input elements do I need to calculate my output element

Code example (single thread block)

Sunday, May 03, 2020 11:16 AM



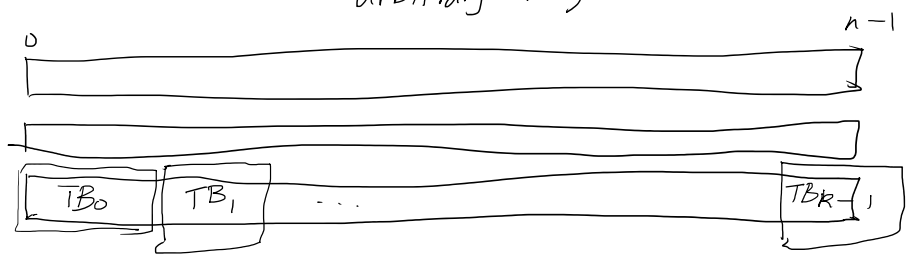
Arrays A and B contain exactly 1024 elements.

Code example (multiple thread blocks)

Sunday, May 03, 2020 11:20 AM

A and B are vectors of arbitrary length n

Version 1:



Execution grid:

thread block : (1024, 1, 1)

thread blocks : $\left\lceil \frac{n}{\text{thread block size}} \right\rceil = \left\lceil \frac{n}{1024} \right\rceil$

grid : (# thread blocks, 1)

Kernel code:

```
tid = blockIdx.x * blockDim.x + threadIdx.x;
if (tid < n){
    c[tid] = A[tid] + B[tid];
}
```

Performance issue: creation of large number of thread blocks ⟹ more work for the CUDA runtime to move thread blocks in and out of the hardware

---

Version 2:

- Create limited (fixed) number of thread blocks
- Have each thread calculate multiple output elements in C

How many thread blocks to create?

"Back of envelop" calculation:

- Each SM can accomodate 2048 threads at any time.

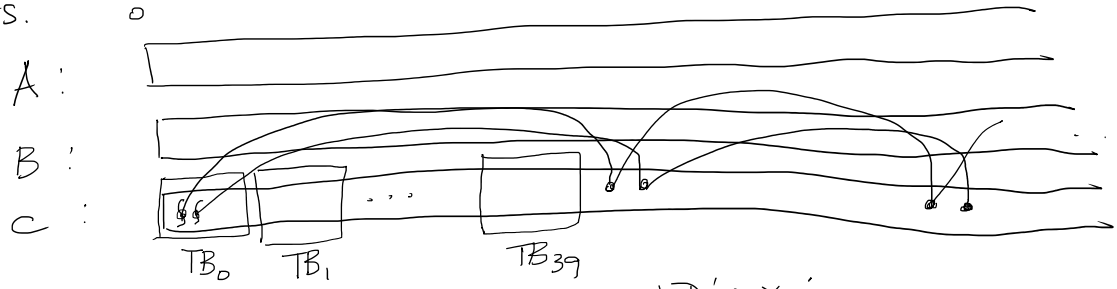
2 thread blocks of dimension (1024, 1, 1) per SM

. So, # of thread blocks = 2 x 20 = 40

Execution grid:

thread block: (1024, 1, 1)
#thread blocks: 40
grid: (#thread blocks, 1) = (40, 1)

We use the concept of striding to calculate output elements.



stride length = gridDim.x * blockDim.x;
total number of threads in the grid.

Kernel code:

```
tid = blockIdx.x * blockDim.x + threadIdx.x;
stride = gridDim.x * blockDim.x;
While (tid < n){
    c[tid] = A[tid] + B[tid]; // Calculate current element
    tid += stride; // Move on to next element
}
```