CS 179: GPU Computing

Recitation 2: Synchronization, Shared memory, Matrix Transpose

Synchronization

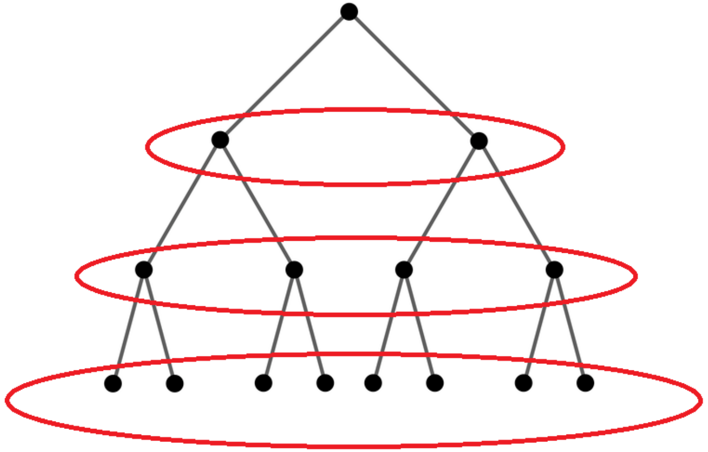
Ideal case for parallelism:

- no resources shared between threads
- no communication between threads

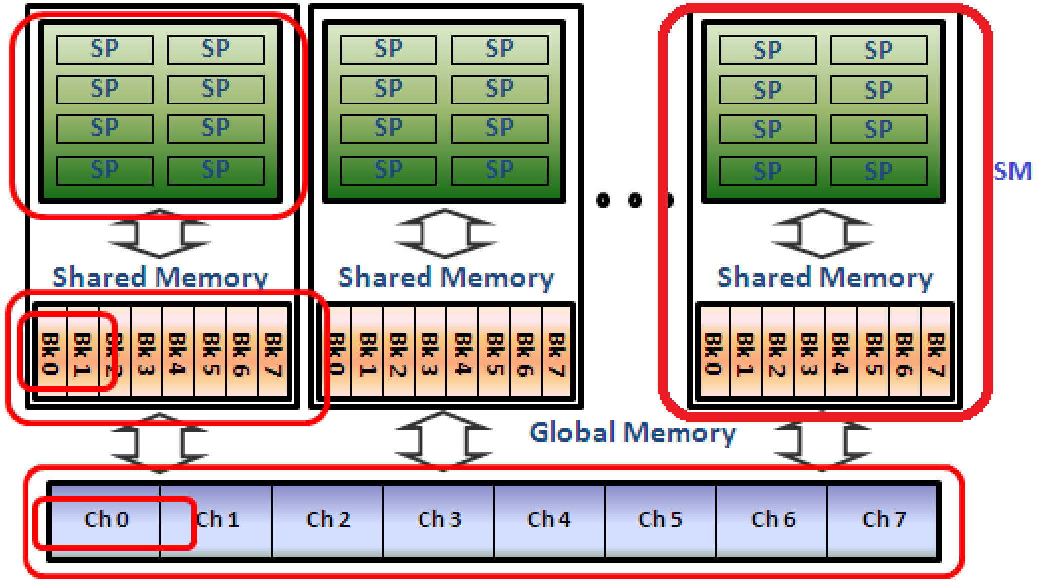
Many algorithms that require just a little bit of resource sharing can still be accelerated by massive parallelism of GPU

Examples needing synchronization

(1) Parallel BFS
(2) Summing a list of numbers
(3) Loading data into a GPU’s shared memory



__syncthreads()



- `__syncthreads()` synchronizes all threads in a block.
- Remember that shared memory is per block. Every block that is launched will have to allocate shared memory for its own itself on its resident SM.
- This __synchthreads() call is very useful for kernels using shared memory.

Atomic instructions: motivation

Two threads try to increment variable `x=42` concurrently.

Final value should be 44.

Possible execution order:

```
thread 0 load x (=42) into register r0
thread 1 load x (=42) into register r1
thread 0 increment r0 to 43
thread 1 increment r1 to 43
thread 0 store r0 (=43) into x
thread 1 store r1 (=43) into x
```

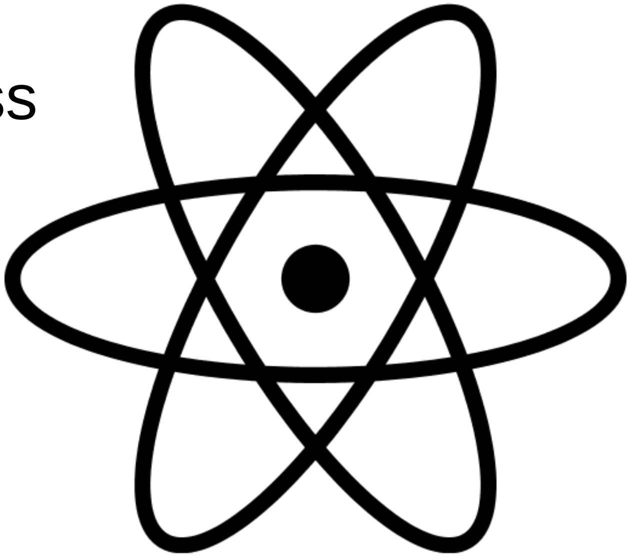
Actual final value of `x: 43`

:(

Atomic instructions

- An atomic instruction executes as a single unit, cannot be interrupted.

- Serializes access



Atomic instructions on CUDA

```
atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS,
          And, Or, Xor}
```

Syntax: `atomicAdd(float *address, float val)`

Work in both global and shared memory!

(Synchronization) budget advice

Do more cheap things and fewer expensive things!

Example: computing sum of list of numbers

Naive:
each thread atomically increments each number to accumulator in global memory

Sum example

Smarter solution:

- each thread computes its own sum in register
- use warp shuffle (next slide) to compute sum over warp
- each warp does a single atomic increment to accumulator in global memory
- Reduce number of atomic instructions by a factor of 32 (warp size)

Warp-synchronous programming

What if I only need to synchronize between all threads in a warp?

Warps are already synchronized!

Can reduce `__syncthreads()` calls

Warp shuffle

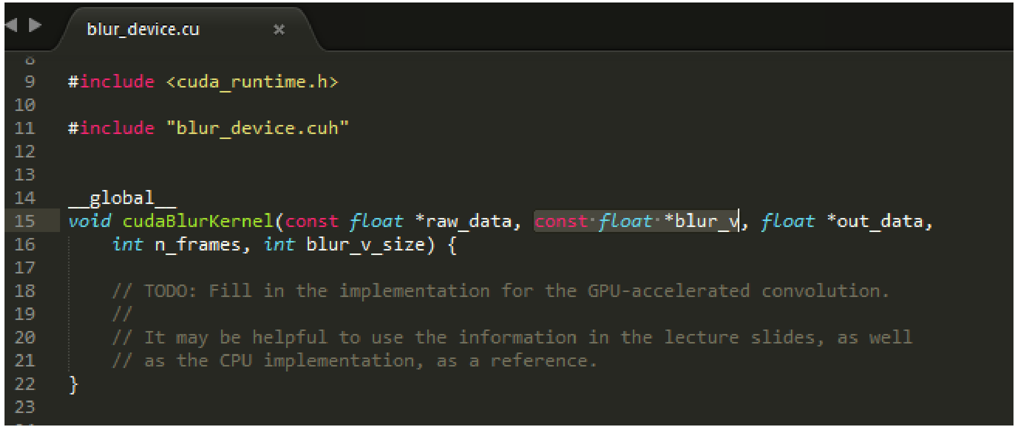
Read value of register from another thread in warp.

```
int __shfl(int var, int srcLane, int width=warpSize)
```

Extremely useful to compute sum of values across a warp.

First available on Kepler (no Fermi, only CC >= 3.0)

Quick Aside: blur_v from Lab 1



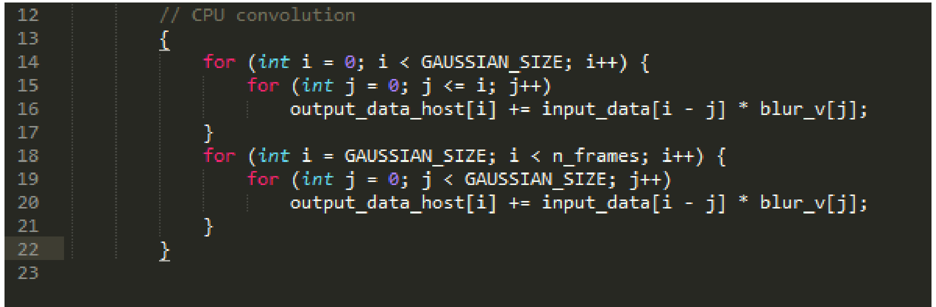
blur_device.cu

```
#include <cuda_runtime.h>

#include "blur_device.cuh"


__global__
void cudaBlurKernel(const float *raw_data, const float *blur_v, float *out_data,
    int n_frames, int blur_v_size) {

    // TODO: Fill in the implementation for the GPU-accelerated convolution.
    //
    // It may be helpful to use the information in the lecture slides, as well
    // as the CPU implementation, as a reference.
}
```



```
        // CPU convolution
        {
            for (int i = 0; i < GAUSSIAN_SIZE; i++) {
                for (int j = 0; j <= i; j++)
                    output_data_host[i] += input_data[i - j] * blur_v[j];
            }
            for (int i = GAUSSIAN_SIZE; i < n_frames; i++) {
                for (int j = 0; j < GAUSSIAN_SIZE; j++)
                    output_data_host[i] += input_data[i - j] * blur_v[j];
            }
        }
```

Shared memory is great place to put blur_v.

1) blur_v is relatively small and easily fits in shared memory.

2) Every thread reads from blur_v

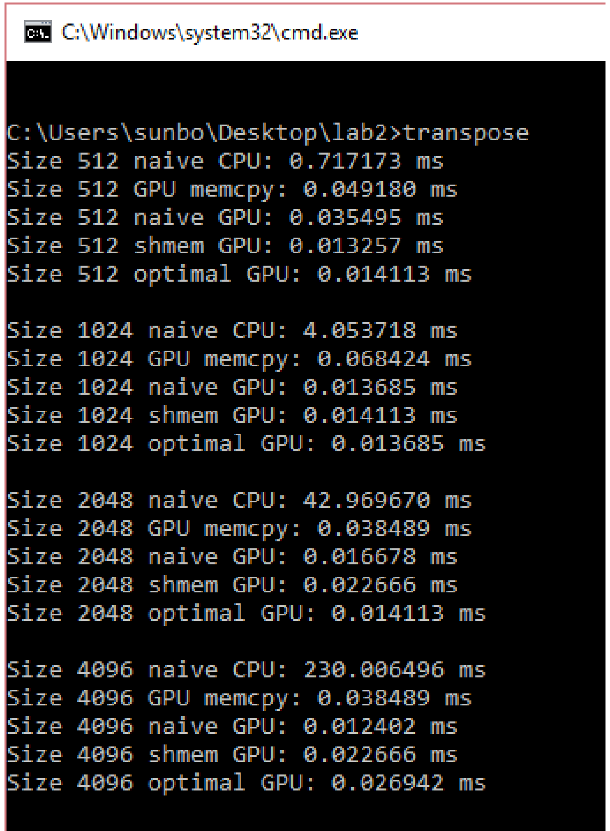
3) Stride 0 access. No bank conflicts when i > GAUSSIAN_SIZE (majority of threads)

Lab 2

(1) Questions on latency hiding, thread divergence, coalesced memory access, bank conflicts, instruction dependencies

(2) What you actually have to do: Need to comment on all non-coalesced memory accesses and bank conflicts in provided kernel code. Lastly, improve the matrix transpose kernel by using cache and memory optimizations.

Matrix Transpose


C:\Windows\system32\cmd.exe

```
C:\Users\sunbo\Desktop\lab2>transpose
Size 512 naive CPU: 0.717173 ms
Size 512 GPU memcpy: 0.049180 ms
Size 512 naive GPU: 0.035495 ms
Size 512 shmem GPU: 0.013257 ms
Size 512 optimal GPU: 0.014113 ms

Size 1024 naive CPU: 4.053718 ms
Size 1024 GPU memcpy: 0.068424 ms
Size 1024 naive GPU: 0.013685 ms
Size 1024 shmem GPU: 0.014113 ms
Size 1024 optimal GPU: 0.013685 ms

Size 2048 naive CPU: 42.969670 ms
Size 2048 GPU memcpy: 0.038489 ms
Size 2048 naive GPU: 0.016678 ms
Size 2048 shmem GPU: 0.022666 ms
Size 2048 optimal GPU: 0.014113 ms

Size 4096 naive CPU: 230.006496 ms
Size 4096 GPU memcpy: 0.038489 ms
Size 4096 naive GPU: 0.012402 ms
Size 4096 shmem GPU: 0.022666 ms
Size 4096 optimal GPU: 0.026942 ms
```

An interesting IO problem, because you have a stride 1 access and a stride n access. Not a trivial access pattern like “blur_v” from Lab 1.

Transpose is just a fancy `memcpy`, so `memcpy` provides a great performance target.

Note: This example output is for a clean project without the shmem and optimal kernels completed. Your final output should show a decline in kernel time for the different kernels.

Matrix Transpose

```
__global__
void naiveTransposeKernel(const float *input, float *output, int n) {
 // launched with (64, 16) block size and (n / 64, n / 64) grid size
 // each block transposes a 64x64 block

 const int i = threadIdx.x + 64 * blockIdx.x;
 int j = 4 * threadIdx.y + 64 * blockIdx.y;
 const int end_j = j + 4;

 for (; j < end_j; j++) {
        output[j + n * i] = input[i + n * j];
 }
}
```

Shared memory & matrix transpose

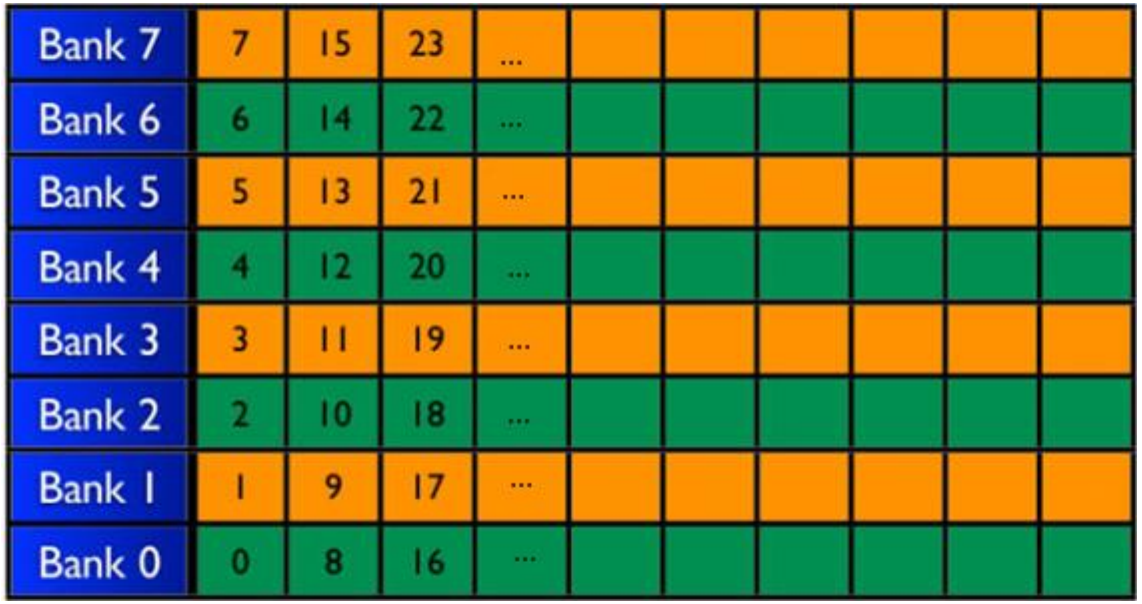
Idea to avoid non-coalesced accesses:

- Load from global memory with stride 1
- Store into shared memory with stride x
- `__syncthreads()`
- Load from shared memory with stride y
- Store to global memory with stride 1

Choose values of x and y perform the transpose.

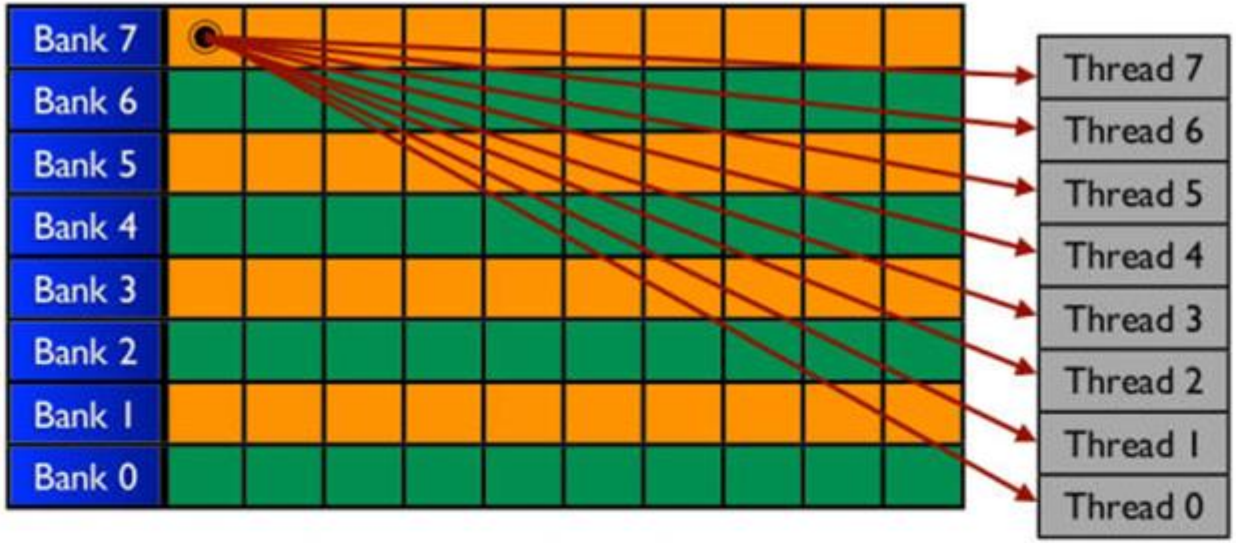
Example of an SM’s shared memory cache

Let’s populate shared memory with random integers.
Here’s what the first 8 of 32 banks look like:



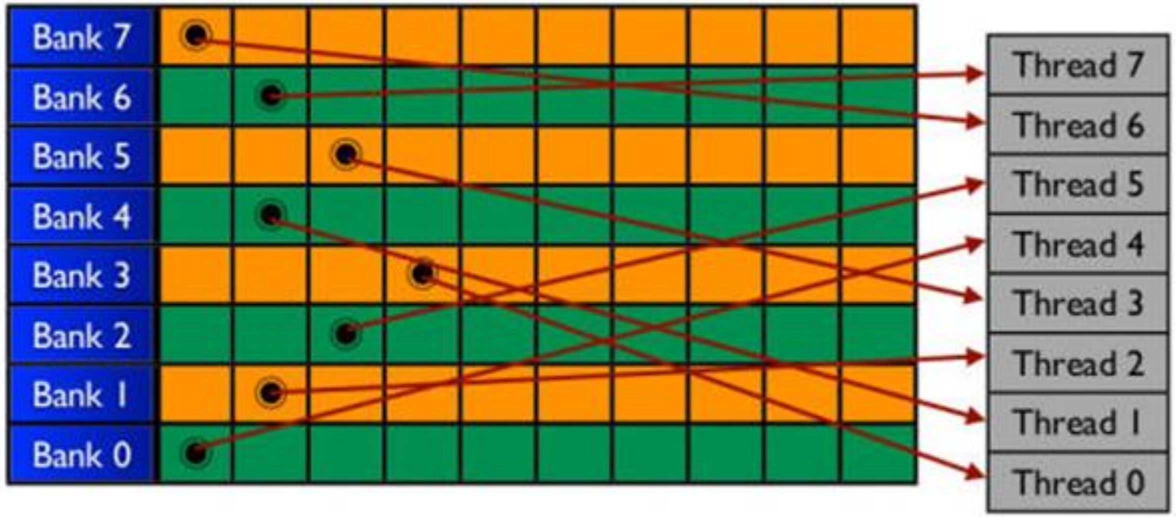
Bank 7	7	15	23	...						
Bank 6	6	14	22	...						
Bank 5	5	13	21	...						
Bank 4	4	12	20	...						
Bank 3	3	11	19	...						
Bank 2	2	10	18	...						
Bank 1	1	9	17	...						
Bank 0	0	8	16	...						

Example of an SM’s shared memory cache



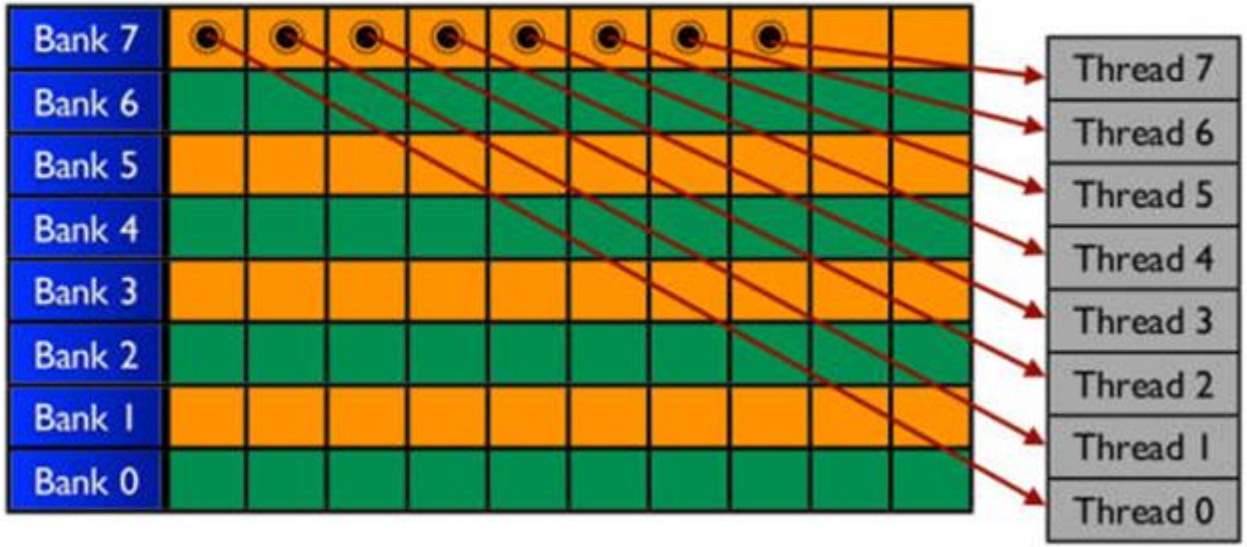
OK: No Bank conflicts when all threads read from the same bank

Example of an SM's shared memory cache



OK: No Bank conflicts as long as each bank is only accessed once.

Example of an SM's shared memory cache



Not OK:

Multiple threads accessing the same bank. Loads become serialized.

Avoiding bank conflicts

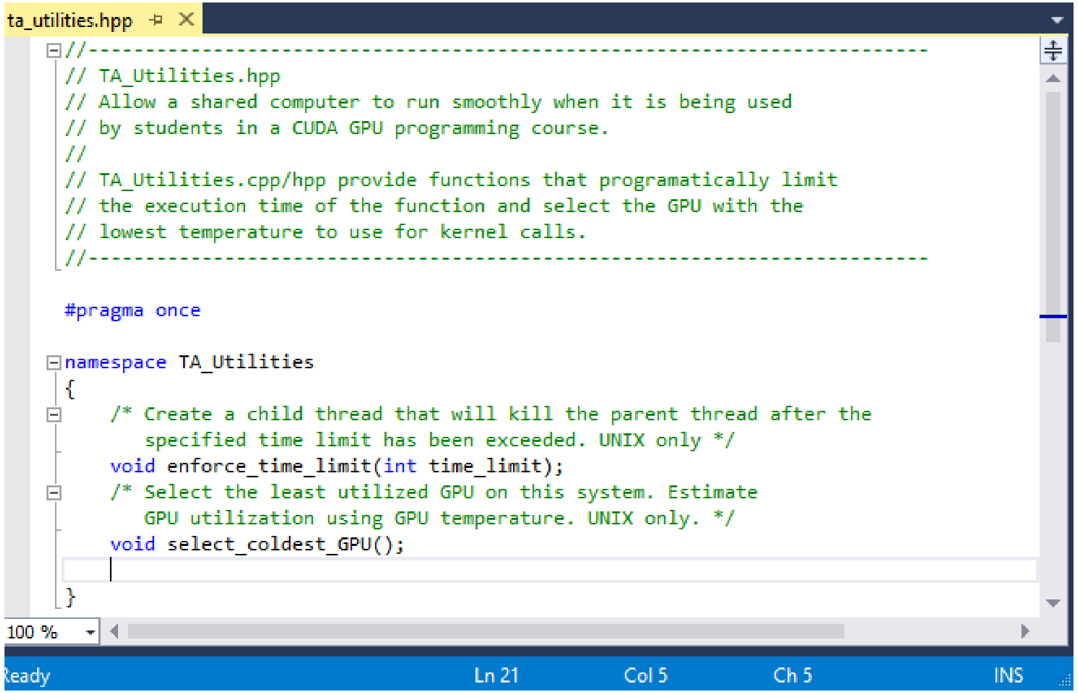
You can choose `x` and `y` to avoid bank conflicts.

`Remember that there are 32 banks and the GPU runs threads in batches of 32 (called warps).`

A stride `n` access to shared memory avoids bank conflicts iff `gcd(n, 32) == 1`.

ta_utils.cpp

- Included in the UNIX version of this set
- Should minimize lag or infinite waits on GPU function calls.
- Please leave these functions in the code if you are using Haru
- Namespace TA_Utilities



```
//----------------------------------------------------------------------------
// TA_Utilities.hpp
// Allow a shared computer to run smoothly when it is being used
// by students in a CUDA GPU programming course.
//
// TA_Utilities.cpp/hpp provide functions that programatically limit
// the execution time of the function and select the GPU with the
// lowest temperature to use for kernel calls.
//----------------------------------------------------------------------------

#pragma once

namespace TA_Utilities
{
    /* Create a child thread that will kill the parent thread after the
       specified time limit has been exceeded. UNIX only */
    void enforce_time_limit(int time_limit);
    /* Select the least utilized GPU on this system. Estimate
       GPU utilization using GPU temperature. UNIX only. */
    void select_coldest_GPU();

}
```