CS179: GPU PROGRAMMING

RECITATION 2

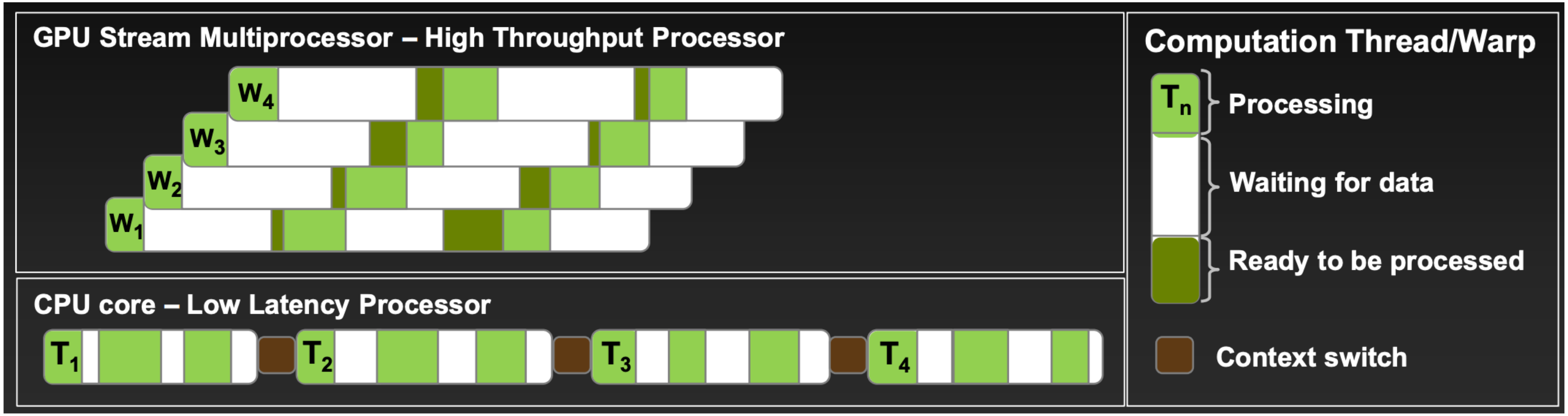
GPU Memory
Synchronization
Instruction-level parallelism
Latency hiding
Matrix Transpose

MAIN REQUIREMENTS FOR GPU PERFORMANCE

- Sufficient parallelism
  - Latency hiding and occupancy
  - Instruction-level parallelism
  - Coherent execution within warps of thread
- Efficient memory usage
  - Coalesced memory access for global memory
  - Shared memory and bank conflicts

LATENCY HIDING

Idea: have enough warps to keep the GPU busy during the waiting time.



LOOP UNROLLING AND ILP

```
for (i = 0; i < 10; i++) {
 output[i] = a[i] + b[i];
}
```

```
output[0] = a[0] + b[0];
output[1] = a[1] + b[1];
output[2] = a[2] + b[2];
…
```

- Reduce loop overhead
- Increase parallelism when each iteration of the loop is independent
- Can increase register usage

SYNCHRONIZATION

`__syncthreads()`

- Synchronizes all threads in a block
- Warps are already synchronized! (Can reduce __syncthreads() calls)

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

- Works in global and shared memory

SYNCHRONIZATION 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

Smarter solution:

- Each thread computes its own sum in register
- Use shared memory to sum across a block (Next week: Reduction)
- Each block does a single atomic increment in global memory

LAB 2

Part 1: Conceptual questions

1. Latency hiding
2. Thread divergence
3. Coalesced memory access
4. Bank conflicts and instruction dependencies

Part 2: Matrix Transpose Optimization

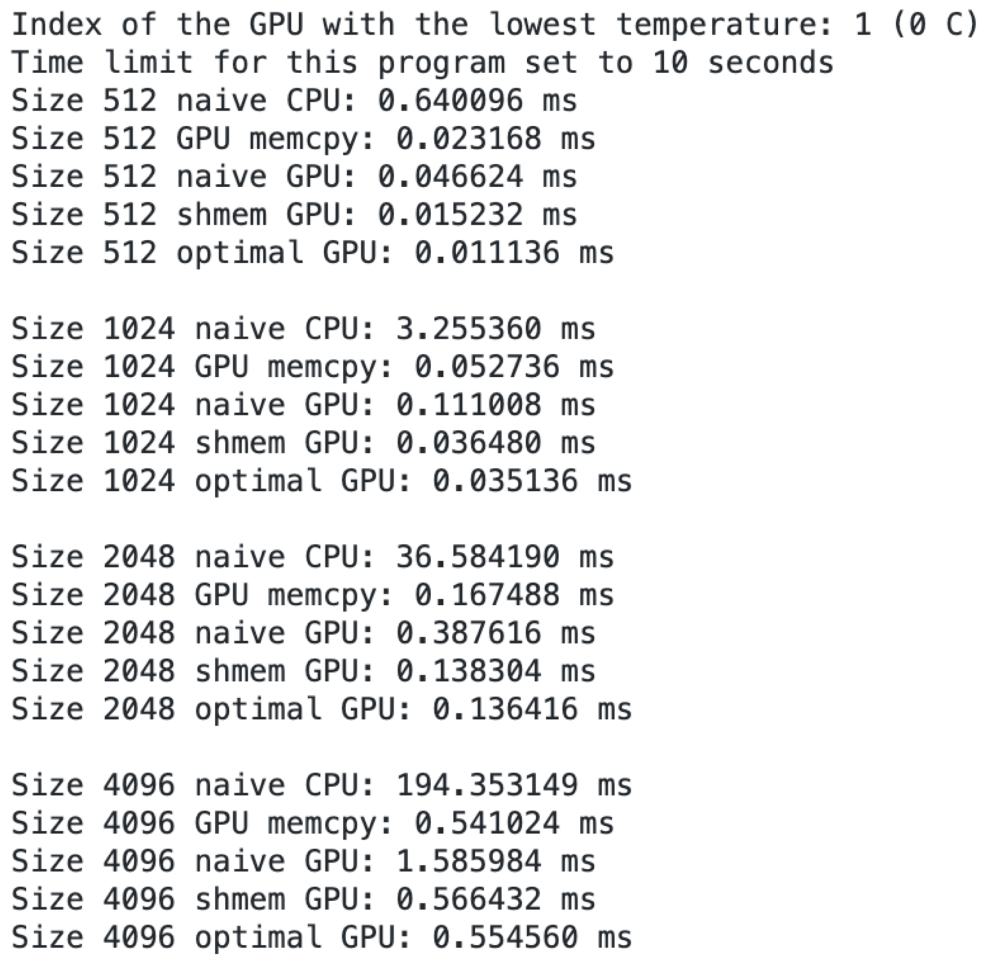
1. Naïve matrix transpose (given to you)
2. Shared memory matrix transpose
3. Optimal matrix transpose

Need to comment on all non-coalesced memory accesses and bank conflicts in provided kernel code

MATRIX TRANSPOSE

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.

The example output compares performance among CPU implementation and different GPU implementations.



```
Index of the GPU with the lowest temperature: 1 (0 C)
Time limit for this program set to 10 seconds
Size 512 naive CPU: 0.640096 ms
Size 512 GPU memcpy: 0.023168 ms
Size 512 naive GPU: 0.046624 ms
Size 512 shmem GPU: 0.015232 ms
Size 512 optimal GPU: 0.011136 ms

Size 1024 naive CPU: 3.255360 ms
Size 1024 GPU memcpy: 0.052736 ms
Size 1024 naive GPU: 0.111008 ms
Size 1024 shmem GPU: 0.036480 ms
Size 1024 optimal GPU: 0.035136 ms

Size 2048 naive CPU: 36.584190 ms
Size 2048 GPU memcpy: 0.167488 ms
Size 2048 naive GPU: 0.387616 ms
Size 2048 shmem GPU: 0.138304 ms
Size 2048 optimal GPU: 0.136416 ms

Size 4096 naive CPU: 194.353149 ms
Size 4096 GPU memcpy: 0.541024 ms
Size 4096 naive GPU: 1.585984 ms
Size 4096 shmem GPU: 0.566432 ms
Size 4096 optimal GPU: 0.554560 ms
```

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

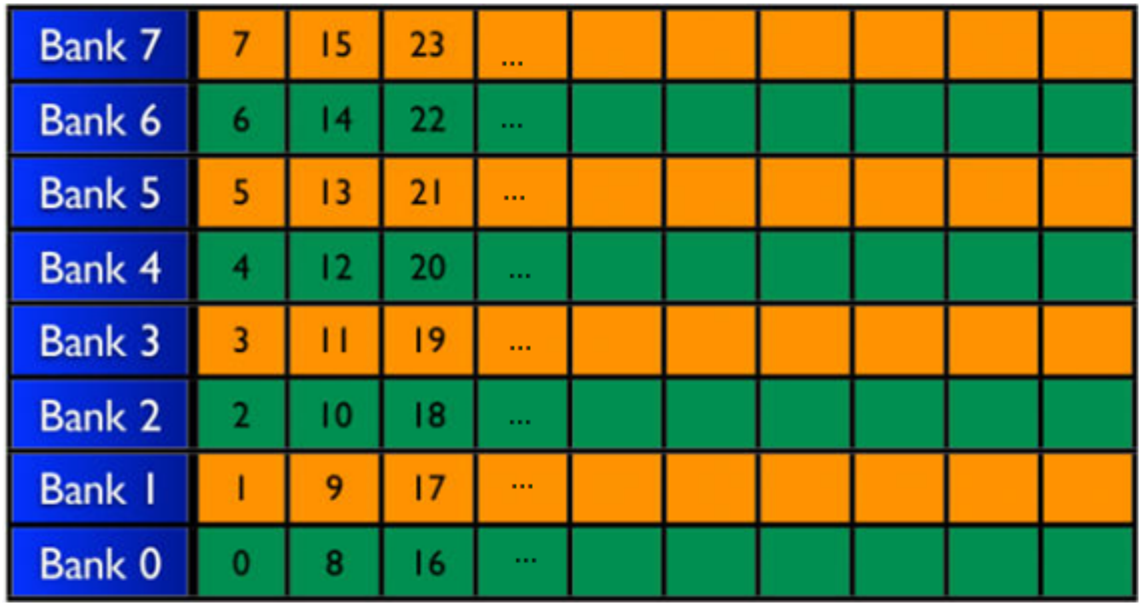
Idea to avoid non-coalesced accesses:

- 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

Need to choose values of x and y to perform the transpose

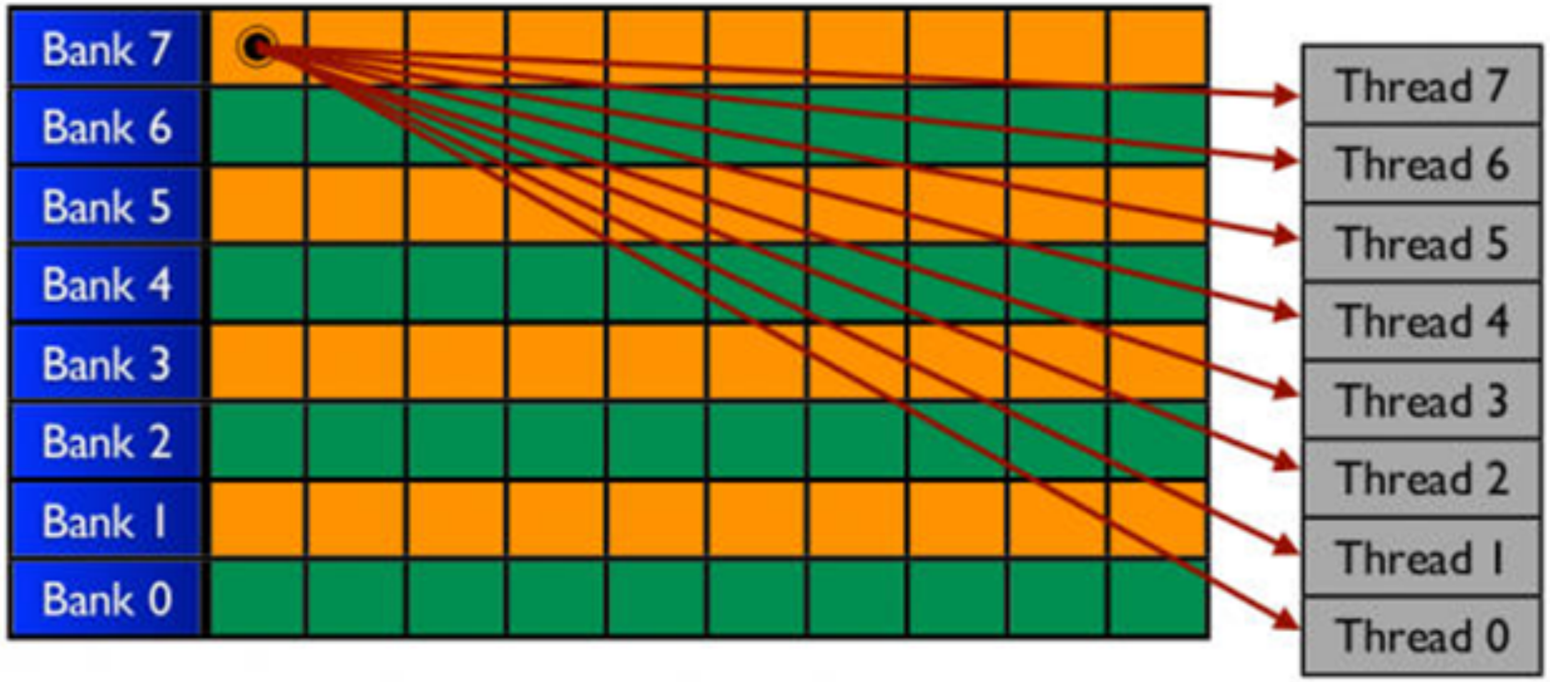
EXAMPLE OF A SHARED MEMORY CACHE

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



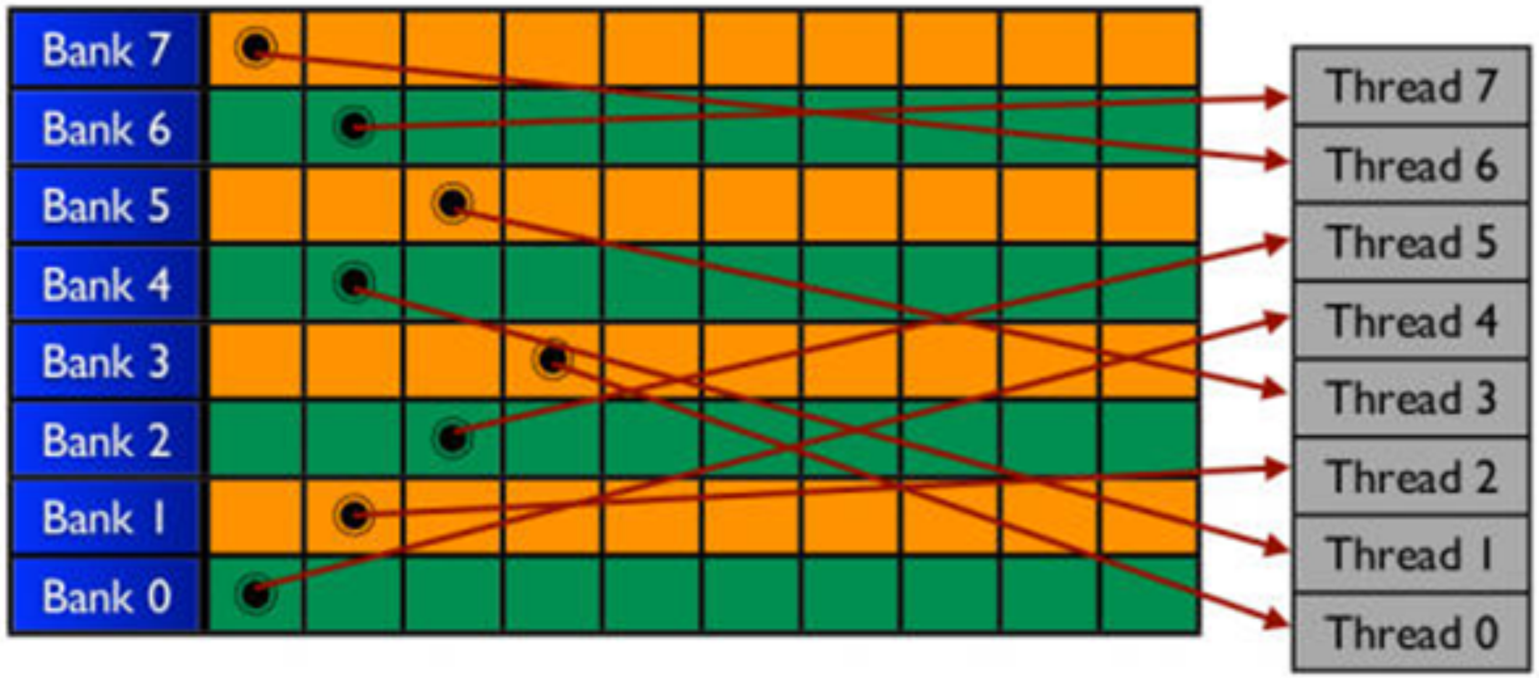
Bank 7	7	15	23	…						
Bank 6	6	14	22	…						
Bank 5	5	13	21	…						
Bank 4	4	12	20	…						
Bank 3	3	11	19	…						
Bank 2	2	10	18	…						
Bank 1	1	9	17	…						
Bank 0	0	8	16	…						

EXAMPLE OF A SHARED MEMORY CACHE



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

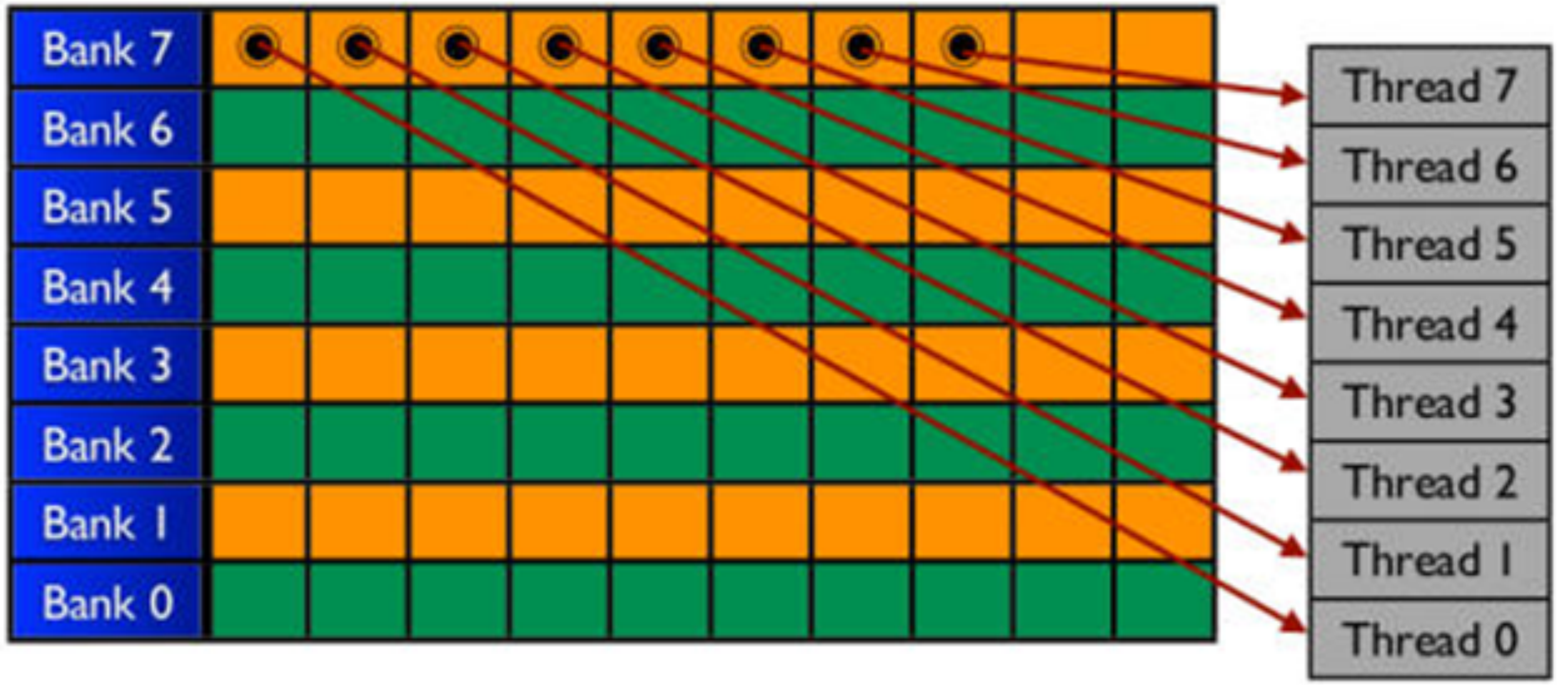
EXAMPLE OF A SHARED MEMORY CACHE



OK:

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

EXAMPLE OF A 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.

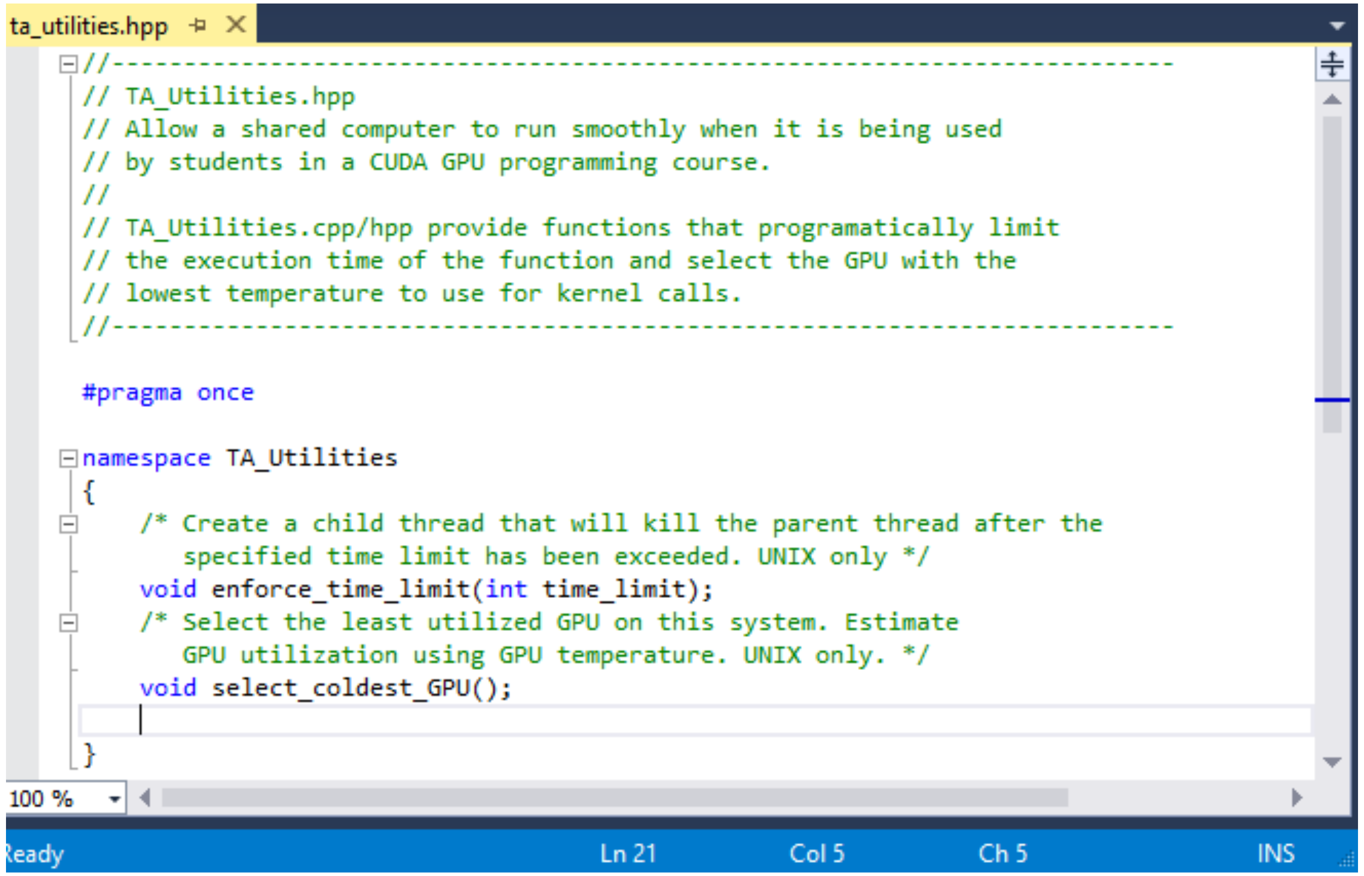
`Remember that there are 32 banks and the GPU runs threads in batches of 32 (called warps).`

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

TA_UTILS.CPP

DO NOT DELETE THIS CODE!

- 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 Titan/Haru/Maki
- 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();
    
}
```