CME213/ME339
Lecture 6

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Debugging

- "Invalid configuration argument" means that either your block dimensions or grid dimensions were invalid
- Can be because they were too large or zero in all dimensions
- This is easy to diagnose — just print out your grid/block dimensions before you launch the kernel
- "Unspecified launch failure" usually means that an out of bounds memory access has occurred
- This is the GPU equivalent of a segfault
- Use cuda-memcheck to catch out of bounds memory access.

Debugging

- Add -G to the nvcc command to enable GPU side debugging information
- `nvcc -o test test.cu -arch=sm_20 -G`
- `cuda-memcheck ./test`
- Similar but not as powerful as valgrind for the cpu.

```
__global__
void myKernel(int *in) {
  in[threadIdx.x] += 1;
}

int main(void) {
  int *dIn;
  cudaMalloc(&dIn, sizeof(int));

  myKernel<<<1, 128>>>(dIn);
  return 0;
}
```

Debugging

Results of `cuda-memcheck ./test` are:

```
========= CUDA-MEMCHECK
========= Invalid __global__ read of size 4
=========     at 0x00000060 in test.cu:4:myKernel
=========     by thread (32,0,0) in block (0,0,0)
=========     Address 0x00201080 is out of bounds
=========
========= ERROR SUMMARY: 1 error
```

Line number is off by one, but it will give a general idea of where to look.

Debugging

- You can use `printf` from inside a kernel!
- You need to make sure the kernel is compiled for at least device capability 2.0
- That's the `-arch=sm_20` option we pass to `nvcc`
- We made a mistake in the Makefile for HW1 that didn't allow `printf` to be used without modifying the Makefile
- This is fixed

Debugging

Don't do this:

```
__global__
void test(...) {
    const int tid = blockDim.x * blockIdx.x + threadIdx.x;
    //...
    printf("%d %d\n", foo, bar);
    //...
}
```

- Every thread in every block will try and print something - tons of data!
- The default size of the print buffer is 8MB
- The buffer is circular
- Main idea: protect the print statement somehow

Debugging

Print only for one block:

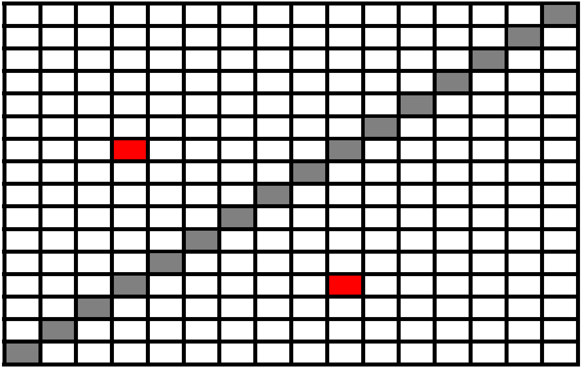
```
if (blockIdx.x == 0)
  printf("%d %\n", foo, bar);
```

or even better for only one thread:

```
const int gtid = threadIdx.x + blockIdx.x * blockDim.x;
if (gtid == 4732)
  printf("%d %d\n", foo, bar);
```

Matrix Transposition

M rows; N columns $\rightarrow$ N rows; M columns



$$A(i,j) \rightarrow A(j,i)$$

Simple Implementation

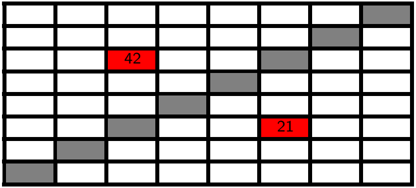
```
__global__
void simpleTranspose(int *array_in, int *array_out,
                     int M, int N)
{
  const int tid = threadIdx.x + blockDim.x * blockIdx.x;

  int n = tid % N; //m, n in INPUT
  int m = tid / N;

  array_out[n * M + m] = array_in[m * N + n];
}
```

- We assume `M * N` is a multiple of `blockDim.x`
- Each block is 1D and the grid is also 1D
- Loc (n, m) in an array of size (M, N) is given by `m * N + n`

Indexing Example



- The output array is (N, M) and the transposed location is (m, n)
- Accounting for the formula `n * M + m`
- M = N = 8
- m: `21 / 8 = 2`
- n: `21 % 8 = 5`
- New Location: `5 * 8 + 2 = 42`

Performance

# Array Dimensions	GB/sec
(256, 256)	32
(512, 512)	38
(1024, 1024)	28
(2048, 2048)	22
(4096, 4096)	-

- Invalid configuration argument at the last size due to requesting 65536 blocks
- The max is 65535 — move to a 2D grid
- Theoretical peak is 152 GB/sec — why such poor performance?
- And why does performance drop with increasing grid size?

Move to 2D

```
__global__
void simpleTranspose2D(int *array_in, int *array_out,
                       int M, int N)
{
  const int n = threadIdx.x + blockDim.x * blockIdx.x;
  const int m = threadIdx.y + blockDim.y * blockIdx.y;

  array_out[n * M + m] = array_in[m * N + n];
}
```

- Here we are assuming that M is a multiple of `blockDim.y` and N is a multiple of `blockDim.x`
- Superficially similar, we trade a modulo and division for a multiply and add

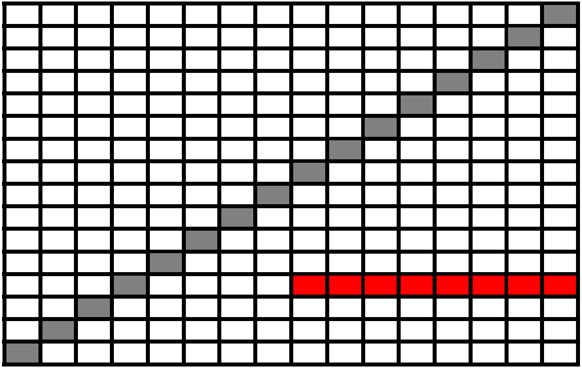
Performance

# Array Dimensions	GB/sec
(256, 256)	53
(512, 512)	63
(1024, 1024)	70
(2048, 2048)	69
(4096, 4096)	71

- Performance curve behaves as expected
- Increases with problem size until a plateau is reached
- Performance level has more than doubled
- Why such a difference?

Memory Access Pattern #1

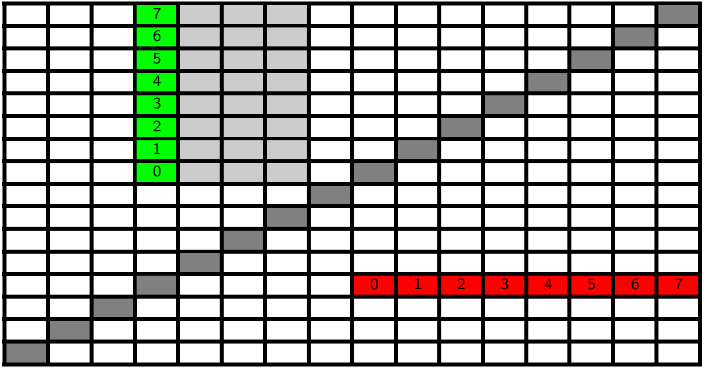
Warp Size of 8



Reads are coalesced

Memory Access Pattern #1

Warp Size of 8

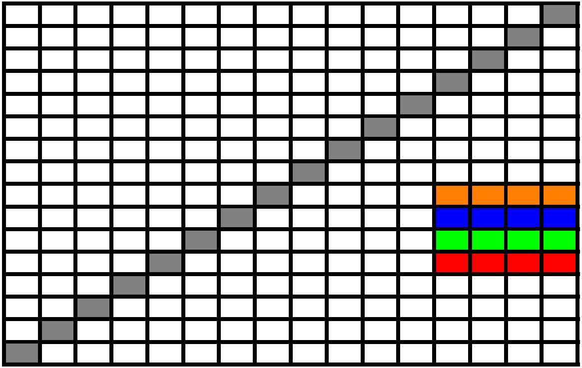


Writes are NOT!

Memory Access Pattern #1

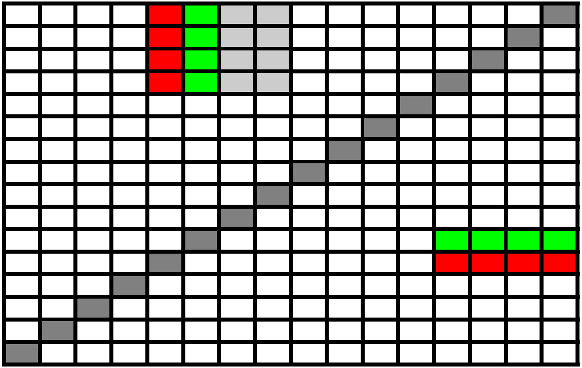
- The reads are fully coalesced
- However, the writes are completely uncoalesced
- For each item that is written a complete 128 byte transaction is made and 124 bytes are wasted
- Explains the low performance, but not why performance drops with increasing size

Memory Access Pattern #2



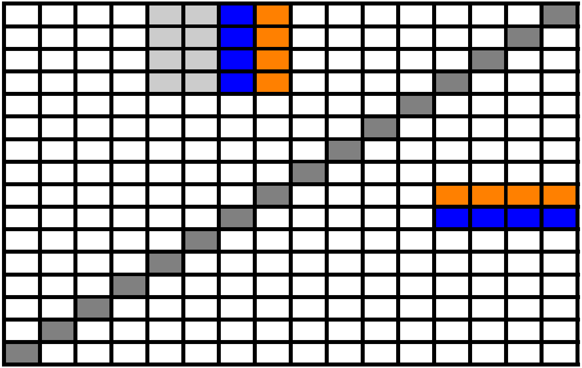
- Memory access pattern is 2D
- Reads might actually be less coalesced depending on dimensions of block
- Previous measurements were taken for 16×16, so each read was only half used but the performance still went up

Memory Access Pattern #2



- Notice that we have now written *two* values in each transaction

Memory Access Pattern #2



- Caching!
- The writes of the second warp occur to memory that has been cached because of the first warp's transaction

Cache Hierarchy

- One 768KB L2 cache that serves ALL the SMs
- Each SM has an 8KB L1 cache
- In the first case then L2 cache is able to hold relevant data for later warps before being evicted
- As the size increases most data gets evicted before it is used again
- In the 2D case we enforce a memory access pattern that ensures we can use data in the L2 and even L1 before it is evicted