CUDA: Analysis

W Truong

1. **Parallel Reduction**
   a. *Bandwidth computation*

   Bandwidth (in GB/s) is calculated as follows:

   $$Bandwidth = N * sizeof(data\ type) / (t * 1x10^{-9})$$

   where *N* is the number of elements in an input array, *sizeof(data type)* is the number of bytes for the data type (4 bytes for a float as float is used in this parallel reduction), *t is the time to run the kernel*, and $1x10^{-9}$ is used to scale the bandwidth from bytes to GB.

   The time to run the kernel does not count the warm-up time (when the kernel is first called). The time refers to the steady-state performance. The kernel takes longer to run when it is warming up due to overhead such as just-in-time (JIT) compilation. All kernels use shared memory, giving a higher effective bandwidth than using global memory.

   The effective bandwidth is improved by removing divergence, using sequential addressing (replacing interleaved addressing), reducing the number of blocks with a load, unrolling the last warp to reduce loop overhead, and finally load as many elements as needed into each thread. The test results from each refinement of parallel reduction are provided below.

   b. *Test results for refining parallel reduction*

   All test results use 67108864 (2^{26}) elements. Step speedup is the speedup from the previous version of reduction to the current one (e.g. 1.69x speedup from naive to stride). Cumulative speedup is the overall speedup from the naive version to the current version (e.g. 4.116x overall speedup from naive to first_add).

Bandwidth and Speedups for all seven versions of parallel reduction

Version	Time (ms)	Bandwidth (GB/s)	Step Speedup	Cumulative Speedup
naive	2.696	99.57	N/A	N/A
strided access	1.594	168.4	1.691342535	1.691342535
sequential access	1.248	215.09	1.27724359	2.16025641
first add before reduce	0.655	409.83	1.905343511	4.116030534
unroll last warp	0.454	591.27	1.442731278	5.938325991
multiple (algorithm cascading)	0.355	756.16	1.278873239	7.594366197

Output files for all seven versions from naive to multiple are under reduce_naive.out, reduce_stride.out, reduce_seq.out, reduce_first_add.out, reduce_unroll.out, and reduce_mult.out respectively.

2. Matrix Transpose

a. Overview

Two versions of the transpose algorithm were implemented: naive and an efficient approach using shared memory and padding. We are more interested in the efficient version shown below.

```
__global__
void matTrans(dtype* AT, dtype* A, int N)  {
     __shared__ dtype temp[BLOCK_DIM][BLOCK_DIM+1];

     unsigned int xIdx = blockIdx.x * BLOCK_DIM + threadIdx.x;
     unsigned int yIdx = blockIdx.y * BLOCK_DIM + threadIdx.y;

     if ((xIdx < N) && (yIdx < N)) {
        unsigned int idx_in = xIdx + N * yIdx;
        temp[threadIdx.y][threadIdx.x] = A[idx_in];
     }

     __syncthreads();

     // transposed indices
     xIdx = blockIdx.y * BLOCK_DIM + threadIdx.x;
     yIdx = blockIdx.x * BLOCK_DIM + threadIdx.y;

     if ((xIdx < N) && (yIdx < N)) {
        unsigned int idx_out = xIdx + N * yIdx;
        AT[idx_out] = temp[threadIdx.x][threadIdx.y];
     }
}
```

Before running the kernel, the algorithm statically computes a 2D grid threads and blocks. The following formulas show how it computes the number of threads and blocks for the x and y dimensions.

$$Number\ of\ blocks\ =\ N\ /\ BLOCKDIM$$
$$Number\ of\ threads\ =\ BLOCKDIM$$
$$Total\ number\ of\ blocks\ =\ dim3\ (number\ of\ blocks,\ number\ of\ blocks,\ 1)$$
$$Total\ number\ of\ threads\ =\ dim3\ (number\ of\ threads,\ number\ of\ threads,\ 1)$$

where N is the number of elements, BLOCKDIM is the size of each block, and dim3 is a 3D representation (2D since the z-axis has a size of 1) of the grid and blocks.

The block size *must* be a power of 2 and less than or equal to 32 since the maximum number of threads in a block is 1024 (32x32 for 2D allocation). Thus, because the algorithm statically computes the number of blocks and threads (and because a group of 32 threads executes as a warp), N must be a power of 2.

The first step in optimization is to use shared memory instead of global memory. In both versions, the reads are coalesced. However, the naive version suffers from the very slow access times using global memory to do

non-coalesced writes. The efficient version solves this by using shared memory to calculate the indices twice. Like the naive version, the indices are first calculated to obtain the indices for the read. The difference, however, is that the indices are calculated a second time by changing the axes of the block indices used in each index to obtain the transposed indices, thus ensuring a coalesced write. To clarify, in the code snippet provided, the x-index uses the block index in the x-axis for the read, but it uses the block index in the y-axis for the write. __syncthreads() is required since the write locations to AT are different from the read locations of AT. By using shared memory, the efficient kernel avoids the large strides through global memory.

The second step in optimization is removing bank conflicts. Since the shared memory is allocated as BLOCKDIM x BLOCKDIM elements, the elements in a column map to the same shared memory bank, causing BLOCKDIM-way bank conflicts. By padding 1 element to the column, the same column indices in successive rows are on different banks, eliminating bank conflicts.

b. *Test results*

All test results were done with the number of elements as a power of 2. More specifically, the number of elements tested on was 1024x1024. Block size was changed from 32 to 16 and lastly to 8, all of these being powers of 2.

Overall results, showing throughput (in billion elements per second) and cumulative speedups at various block sizes (32, 16, and 8 respectively)

Version	32x32 blocks, 32x32 threads	64x64 blocks, 16x16 threads	128x128 blocks, 8x8 threads
Naive throughput	13.7971	23.8313	21.8453
shared memory only throughput	29.1271	40.3298	21.8453
shared memory + no bank conflict throughput	43.6907	45.4903	21.8453
Cumulative speedup from naive to efficient version w/o bank conflicts	3.166658211	1.908846769	1

The next section will focus on the results gathered with a block size of 16, but other block sizes will be discussed too.

Throughput and speedups for a block size of 16 (64x64 blocks in the grid, 16x16 threads per block)

Version	Throughput (billion elements per second)	Step speedup	Cumulative speedup
naive	23.8313	N/A	N/A
shared memory only	40.3298	1.692303819	1.692303819
shared memory + no bank conflict	45.4903	1.908846769	1.12795749

c. *Discussion*

Because the number of blocks and threads were computed statically, tuning had to be done to obtain the maximum throughput after all optimizations were done. Starting with a block size of 32 (the maximum block size) before moving to block sizes of 16 and 8, the throughputs at each block size was computed.

A block size of 16 was found to be optimal among the three different block sizes, with the kernel running with a block size of 32 (1024 threads per block) being slightly slower than that running with a block size of 16 (256 threads per block). However, running with a block size of 8 (64 threads per block) was found to be much slower than running with the other two sizes. In all cases, the overall thread count is still the same.

The GPU's specialty is hiding latency. On the V100, its compute capability is 7.0 with 80 SMs. An SM in this case will issue one instruction per warp over 1 clock cycle for 4 warps at a time. There is also a maximum of 32 thread blocks per SM (a maximum of 2560 blocks scheduled at a time), a maximum of 2048 threads per SM, and a maximum of 1024 threads per block.

All cases below will use the 1024x1024 matrix previously tested as an example, with the number of blocks and threads per block computed based on the block sizes. In all cases, only up to 2560 blocks can be in all the SMs at once, and the other waiting blocks cannot be assigned until all the warps in the currently running block(s) are done.

For the case with a block size of 8, the low throughput can be explained. There are 128*128 (16384) blocks, each block with 64 threads. This gives the maximum number of thread blocks and threads per SM but minimizes thread block size. Since only 2 warps are present in each block, each block transposes a very small chunk of the matrix. This means thread-level parallelism (instruction issued to another warp) is much less than if larger block sizes were used.

For the case with a block size of 16, there are 64*64 (4096) blocks and 16*16 (256) threads. In this case, 8 blocks can fit to each multiprocessor. It is possible that the few last “slowest” warps bottleneck the entire block in the SM while the rest of the warps are finished, especially at a synchronization point. This explains why a block size of 16 causes the kernel to run slightly faster than that with a block size of 32, since a block size of 32 has more warps (32 warps with a block size of 32 compared to 8 warps with a block size of 16).