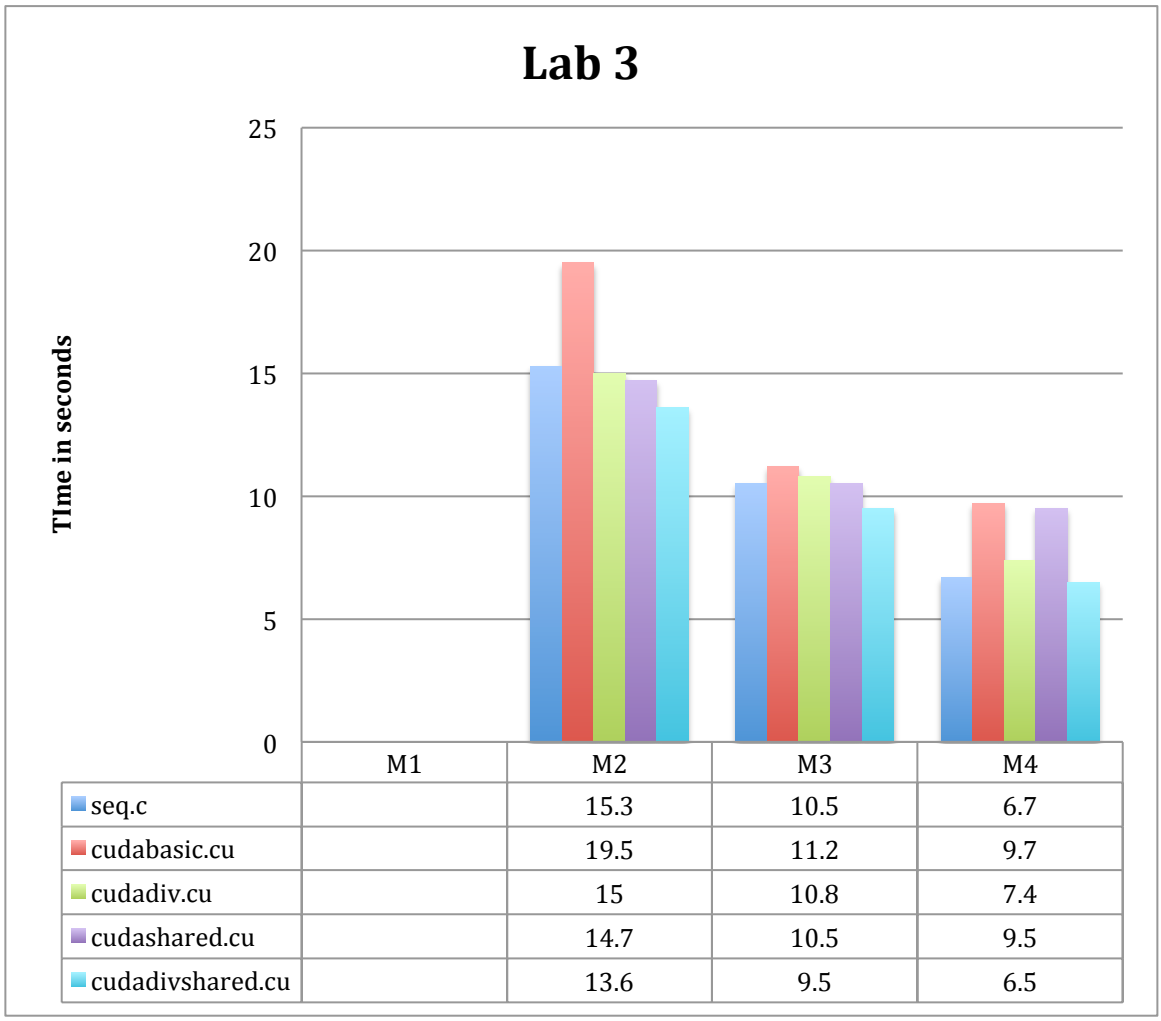
Connie Shi
Parallel Computing
Professor Zahran
Lab 3 – CUDA Parallel Max Reduction

* Note: I wrote an extra CUDA program called cudadivshared.cu that is the most optimized time that I could get CUDA to run parallel max reduction. cudashared.cu runs the fastest of the 3, but its optimizations actually did not reach peak performance. Cudadivshared.cu was the best performance I could manage.

M1 = Value where cudabasic.cu is better than seq.c = none
M2 = Value where cudadiv.cu is better than seq.c = 1,000,000,000
M3 = Value where cudashared.cu is better than seq.c = 700,000,000
M4 = Value where cudadivshared.cu is better than seq.c = 500,000,000



	M1	M2	M3	M4
seq.c		15.3	10.5	6.7
cudabasic.cu		19.5	11.2	9.7
cudadiv.cu		15	10.8	7.4
cudashared.cu		14.7	10.5	9.5
cudadivshared.cu		13.6	9.5	6.5

1) What is the relationship between M and the fact of the GPU version be better than the sequential version? Justify.

As M increases, more CUDA programs are able to beat sequential execution. At 500,000,000 elements, only cudadivshared.cu came close to sequential time, while the rest still performed worse. At 1,000,000,000, every CUDA program except for cudabasic.cu was able to beat sequential. Some factors that contribute to this effect are: overhead of CUDA run time, the need to cudaMemcpy elements from the CPU to the GPU and then back again, the low bandwidth of PCIe to copy the elements, and synchronization of the threads. These bottlenecks mean that unless M is sufficiently large, it will generate a poorer performance on the GPU than the CPU because there is not enough data to offset the cost of the bottlenecks.

2) Did you find that taking care of branch divergence makes the GPU version (i.e. cudadiv) better than the sequential version in lower M than the basic GPU implementation (i.e. is M2 higher or lower than M1)? Justify.

Yes. M1 is actually nonexistent. There is no M value within the memory limits constraint that I could test (about 100B) that was able to beat the time limits of sequential. This is because of branch divergence in cudabasic.cu along with the bottlenecks in performance mentioned in the previous question. Branch divergence in particular would happen in the first 5 runs of the program, when stride < 32 (warp size) – at the first run, each warp would only have 50% active threads doing work, the next run, it would have 25%, 12.5%, etc., until stride increases above 32 in which case after that, some warps would not be doing any work. This problem is fixed in cudadiv.cu, which does take thread divergence into account and stops once the stride gets to 32. Then thread 0 would iterate over the last 32 values sequentially to find the max of the block. Iteration over 32 values is very fast.

3) Similarly, was M3 higher or lower than M1? Justify.

M3 and M4 are lower than M1 and M2. Both M3 and M4 optimize on branch divergence and use shared memory to store data that is retrieved from global memory. Global memory is off chip so it is very slow to access in each iteration of the loop to compare every value. This is reduced by having each thread access global memory once to retrieve the value that it wants to read and store it in shared memory for future access. Subsequently, this cuts down global memory access. Shared memory access is very fast because it is on chip and private to the block. I tried to optimize cudashared.cu for each thread to read a variable number of elements and return the maximum of its assigned subset. I thought that having one thread read one element is a waste of thread creation and the overhead of CUDA, so giving more load to each thread might give better performance. While cudashared is better than cudadiv, the

optimization in addition to shared memory actually did not produce peak performance. Cudadivshared.cu removes this intermediate step of having each thread search sequentially within a subset of the data, and yields the best performance of all the CUDA programs here.

4) Beside branch-divergence and shared memory usage, what other optimizations have you used? State them in a bulleted list, with a justification of why you think these optimizations are effective for the problem at hand. If you haven't used any other optimizations, then say so and justify why you thought that the other optimizations wouldn't be effective here.

- **Prefetching** – Gets the next data elements while consuming current data elements. At the beginning of kernel, elements are retrieved from global memory and put in shared memory. When data elements that are read more than once, like checking sdata[tid] versus sdata[tid + stride], the latter is assigned to a variable to be assigned if needed.
- **Loop unrolling** – This was attempted but did not seem to yield a better performance. Ideally, this should have improved the run time, because it would reduce address arithmetic instructions, branch instructions, and increment instructions.
- **Different thread granularity** – This was program was rested using different granularities and seemed to perform best with the maximum number of threads per block. It is able to reduce a greater number of elements at every block to avoid recalling the kernel to do extra calculations, which does have run time overheads.
- **Cutting down problem size to multiple of 32** – It is tricky to deal with more elements than a multiple of 32 because you have to check memory bounds and accommodate an extra warp for less threads. Keeping it to a multiple of 32 means it fits in the warp and every warp gets its full potential.
- **Tiling –** Every block looks at a contiguous block of data, which improves cache hit rates.