THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

Ballot Counting for Optimal Binary Prefix Sum

Anthony Skjellum David Whittaker (Contact: dpwhitt@uab.edu) Purushotham Bangalore
Department of Computer and Information Sciences, University of Alabama at Birmingham

__ballot()

evaluates predicate for all threads of the warp and returns an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp. This function is only supported by devices of compute capability 2.0.[1]

- Compiles to a single instruction
- Functions like MPI_ALLTOALL, for a single bit
- Intra-warp communication without shared mem.

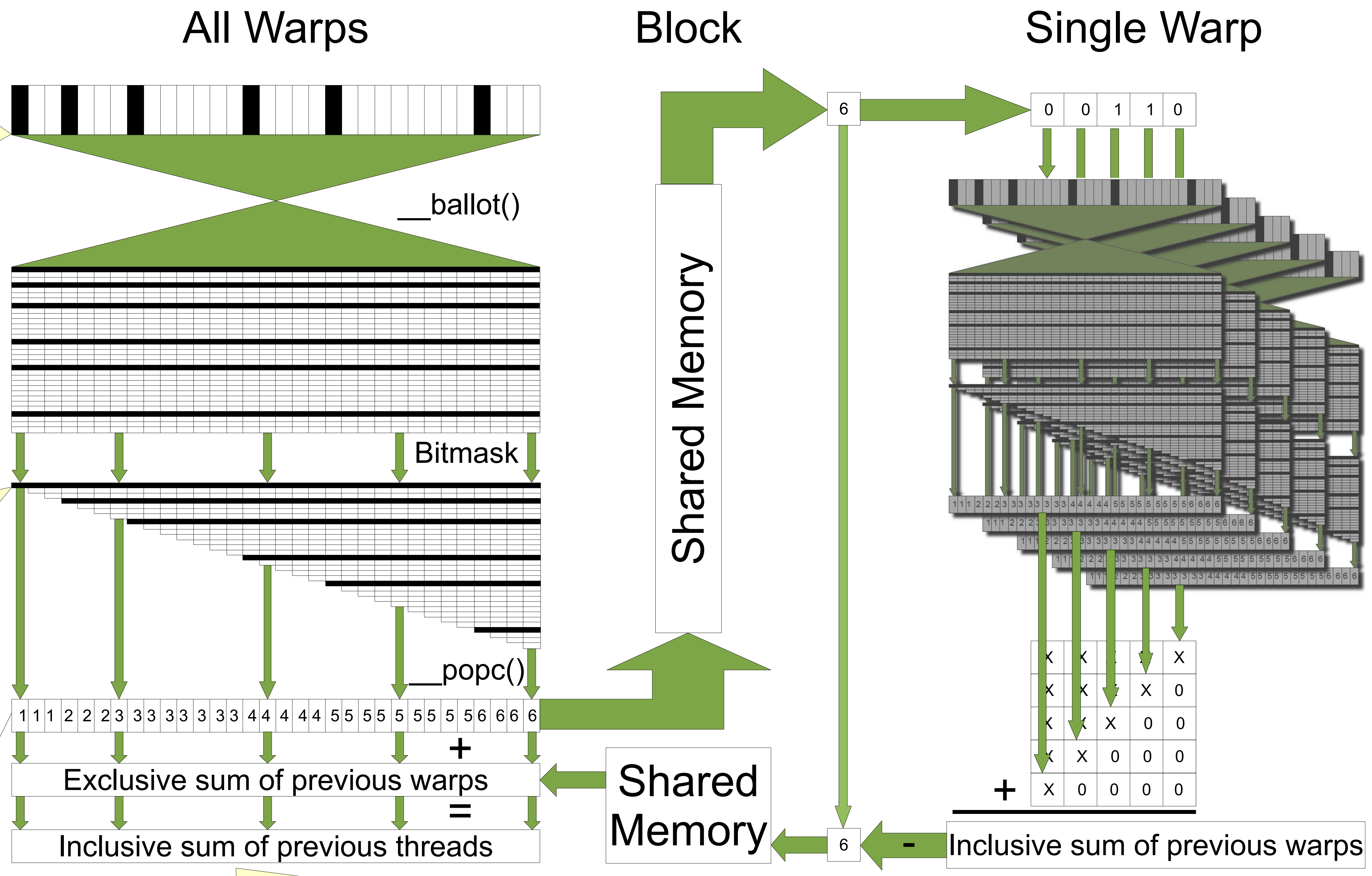
Bitmask

- Ballot returns an aggregate of all threads in warp
- To retrieve only previous threads, higher-order bits should be masked away
- In PTX Assembly, %lanemask_le[2]
- In C, 0xFFFFFFFF >> (31 - (threadIdx.x & 31))
- Logical AND with result of __ballot()

__popc(x)

returns the number of bits that are set to 1 in the binary representation of 32-bit integer parameter **x**.[1]

- Compiles to a single instruction on Fermi
- Gives the sum of previous threads in current warp



Final Calculation

The sum of previous warps is then added to the sum of previous threads in this warp. The result is the number of non-zero predicates in threads before and including each thread. The predicate can be subtracted from this value to give the exclusive prefix sum.

Inter-Warp Sum

- Once the Intra-warp sums have been calculated, the final value from each warp must be scanned.
- The last thread in each warp places its value (the sum of all threads in the warp) in shared memory.
- Then, the first warp reads these values.
- This warp then repeats the __ballot(), bitmask, and __popc() routines on each bit
- Since there are 32 threads per warp, the maximum number of bits to check is 6.
- Each result is shifted left by the bit position that was evaluated.
- The results are added together to give a prefix sum of the preceding warps' sums.
- This value is placed back in shared memory for each warp to retrieve in the final calculation.

Note: this approach uses the binary scan to do integer scans only for integers < 64. Larger numbers require more bits, and become less efficient than the standard approach. However, the standard approach is limited by shared memory latency in its warp reduction stage. Future work will test the general applicability of replacing the warp reduction stage with a 32-bit ballot count.

Application: Radix Sort

The radix sort algorithm[3] is among the fastest GPU sorting algorithms currently published. It sorts keys by proceeding from the Least Significant Bit to the Most Significant Bit. At each step, it performs a stable sort of the keys on the current bit only. It accomplishes this by performing a binary prefix sum to determine the number of set bits appearing before each element. Subtracting from the thread index also gives the number of unset bits. Finally, the last thread broadcasts the total number of unset bits to all threads. With this information, each thread can calculate its unique index in the stable order for this bit.

This requires as many Binary Prefix Sums as there are bits in the key value. The Binary Prefix Sum represents the majority of the workload. So, having a faster Binary Prefix Sum yields a significant improvement over current algorithms.

We tested against a simplified version of the Radix Sort implemented in the CUDPP[4] library. Our version removed the multiple element per thread optimizations that would speed up both algorithms equally for simplicity, but otherwise used the same approach.

Our testing reveals an 80% speed improvement over current techniques

Future Applications: Suffix Array Construction, BWT, BZIP Compression

A suffix array is an array of integers giving the starting positions of suffixes of a string in lexicographical order.[5] Lexicographical sorting must use a slight variation on the theme that Radix Sort employs: it starts from the Most Significant Bit and works downward, but only sorts within the range where the previously sorted bit matches its neighbors. This can be handled by having each thread keep track of the first element that currently matches its own data, as well as the number of such elements. When this number of elements is 1, the element is in the correct position. However, the thread will still have to wait until all the threads in the block have sorted correctly. Some instances can be conceived for which this worst case is easily possible, for instance, a book which has a repeated sentence or paragraph. A data structure should be utilized which keeps active elements separate from sorted elements and uses fewer and fewer blocks as the sorting grows more accurate. In any case, this should prove faster than comparison-based sorting, which would have the same issue with every comparison.

The Burrows Wheeler Transform, which comprises a major step in BZIP compression, is a variation on the suffix array. In fact, if the string is seen as having a null character appended that sorts before every other character lexicographically, then the two can be seen as computationally identical. We hope to use this approach to develop a fast BZIP codec for Fermi GPUs.

[1] NVIDIA CUDA C Programming Guide [2] PTX: Parallel Thread Execution ISA Version 2.1 [3] Michael Garland et. al., Designing Efficient Sorting Algorithms for Manycore GPUs [4] CUDPP: CUDA data parallel primitives library. http://www.gpgpu.org/developer/cudpp/ [5] http://en.wikipedia.org/wiki/Suffix_array

This work supported by the National Science Foundation Grant CNS-0821497