Universität Heidelberg
Institut für Technische Informatik

Robert Strzodka
Application Specific Computing

Advanced Parallel Algorithms

Exercise 4

- **Submit electronically (MOODLE) as one archive named after you until Thursday, 08.12.2022 9:00.**
- **Archive should encompass a PDF and your code.**
- **Code should be compileable (include Makefile or similar).**
- **Include names on the top of the sheets.**
- **A maximum of two students is allowed to work jointly on the exercises.**

Reading assignment

Chapter 10 and Sparse Matrix Storage Formats:

- http://www.netlib.org/linalg/html_templates/node90.html
- https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/appendix-a-linear-solvers-basics/sparse-matrix-storage-formats.html

Freshen up your linear algebra knowledge:

- vector x, norm $\|x\|_2$, scalar product $x \cdot y$
- matrix A, matrix-vector product $A\,x$, matrix-matrix product $A\,B$, inverse A^{-1}
- linear equation system $A\,x = b$

4.1 Histogram with more bins (30 points)

Using the example `2_Concepts_and_Techniques/histogram` we discussed three different histogram implementations:

1. Highest number of bins but possibly atomic collisions between all threads.
2. Medium number of bins with atomic collisions only between threads of the same warp.
3. Smallest number of bins with no atomic collisions

To support larger number of bins we want an intermediate solution between (1) and (2) which allows atomic collisions between threads of `Wc` warps, where `Wc` is given by the user on the command line. Make also the number of bins `binNum` a command line parameter. Change `2_Concepts_and_Techniques/histogram/histogram256.cu` appropriately, now operating on ints instead of chars. Make a performance plot with three curves one for `Wc=1`, one for `Wc=2` one for `Wc=4`, where on the x-axis we have `binNum=256,...,8192`.

4.2 Bitonic sort (3 $\times$ 5 points)

Look at the implementation of bitonic sort in `2_Concepts_and_Techniques/sortingNetworks/bitonicSort.cu` and in particular at the stride-loop L308 and the functions that it calls. Something looks strange here. Answer:

- Why is there a `break` after `bitonicMergeShared()`?
- What happens if we omit the `break`, do we still get the correct results?
- Express the code more clearly without using a `break`?

4.3 Extra: scan with transpose access (50 extra points)

Function `shfl_scan_test()` in example `2_Concepts_and_Techniques/shfl_scan` uses `__shfl_up_sync()` exclusively for efficient scan computation. Let us use j to index the warp number and i to index the lane within the warp. Then the function proceeds in five steps (line numbers refer to file `shfl_scan.cu`)

1. L51: all warps read data d_{ij} into register
2. L61-66: all warps make partial sums along index j: $v_{ij} := \text{scan}(d_{ij} \,|\, i = 0, \dots, 31)$
3. L88-96: one warp makes a scan of the partial sums
   $s_i := \text{scan}(v_{31,j} \,|\, j = 0, \dots, warpNum - 1)$
4. L103-108: all warps correct the partial sums $v_{ij} := v_{ij} + (j > 0) \,?\, s_{j-1} : 0$
5. L113: all warps write out v_{ij} to memory

Let us try a different implementation based on the idea of fast data transposition in shared memory shown in the example `6_Performance/transpose`. Above the indexing is one dimensional but we interpreted the data d_{ij} as a two dimensional array of size $(i : 32 \;\times\; j : warpNum)$ with row major storage to emphasize the action of the individual lanes in the warps. Now without changing the location of any value, let us simply reinterpret this data as a two dimensional array of size $(i : warpNum \;\times\; j : 32)$ with row major storage. It is still exactly the same data in exactly the same order we only look at it differently.

Following this idea replace `shfl_scan_test()` with the following procedure:

1. All warps read data d_{ij} into shared memory in normal order
2. Interpret ds_{ij} as an $(i : warpNum \;\times\; j : 32)$ array with row major storage. One warp makes all the partial sums along index i: $v_{ij} := \text{scan}(d_{ij} \,|\, i = 0, \dots, warpNum - 1)$. Every lane $j = 0, \dots, 31$ simply makes $warpNum$ serial summations along the $i = 0, \dots, warpNum - 1$ index.
3. One warp makes a scan of the partial sums $s_j := \text{scan}(v_{warpNum-1,j} \,|\, j = 0, \dots, 31)$ using `__shfl_up_sync()` similar to step 3 above.
4. All warps correct the partial sums $v_{ij} := v_{ij} + (j > 0) \,?\, s_{j-1} : 0$
5. All warps write out v_{ij} to memory

Measure the performance against the previous version and answer the following questions:

- Depending on *warpNum*, how many additions are performed in the original and in the new version?
- Does the new version produce bank conflicts, can we avoid them?
- What limits performance in the new version?