Exercise 6

360.252 - Computational Science on Many-Core Architectures
WS 2020

November 25, 2020

The following tasks are due by 23:59pm on Tuesday, December 1, 2020. Please document your answers (please add code listings in the appendix) in a PDF document and email the PDF (including your student ID to get due credit) to karl.rupp@tuwien.ac.at.

You are free to discuss ideas with your peers. Keep in mind that you learn most if you come up with your own solutions. In any case, each student needs to write and hand in their own report. Please refrain from plagiarism!

> "I've been imitated so well I've heard people copy my mistakes." — Jimi Hendrix

There is a dedicated environment set up for this exercise:

https://gtx1080.360252.org/2020/ex6/.

To have a common reference, please run all benchmarks for the report on this machine.

Dot Product with Warp Shuffles (3 Points)

Given a vector x of size N, you need to compute

- the sum of all entries,
- the sum of the maximum value of all entries (1-norm),
- the sum of the square of all entries (squared 2-norm),
- the maximum value of the modulus of all entries (max-norm),
- the minimum value of all entries,
- the maximum value of all entries,
- the number of zero entries.

Your friend recommends to call the respective functions in CUBLAS, but you suspect that you can implement a faster version that computes all these values in a single kernel.

Implement and compare different versions of such a kernel for a fixed block size of 256:

1. Using shared memory like for the dot product. (1 Point)

2. Using warp shuffles only (no shared memory). Experiment with a fixed number of 128 thread blocks and $N/256$ blocks. (1 Point)

3. Compare the performance of these variants for different N. Also compare with the execution time for the dot-product $\langle x, x \rangle$. (2 Points)

In all cases, use atomics[1] for writing to global memory so that only a single kernel call is required. Also, transfer all result values with a single call to `cudaMemcpy`.

Sparse Matrix Times Dense Matrix (3 Points + 1 Bonus)

You are investigating a statistical effect in a transistor that is due to random dopant fluctuations when manufactoring tiny transistors. After doing the math, you find that you need to solve the Poisson equation with different right hand sides to analyze the effect.

Because a direct sparse solver requires too much memory for your application, you decide to use a conjugate gradient solver for each of the different right hand sides. Since you have already demonstrated your friend that you can beat CUBLAS, you start to wonder whether you can reduce your average solver times by handling multiple right hand sides simultaneously. Thus, instead of the standard linear system $Ax = b$ with sparse matrix A and right hand side vector b, you want to solve $AX = B$ for K different right hand sides $b_1, \ldots, b_K$. The columns of $X = \{x_1, \ldots, x_k\}$ represent your solutions for each of the right hand side columns of B.

Because a full CG solver with multiple right hand sides requires quite a bit of effort, you decide to focus only on the sparse matrix-vector products with multiple right hand sides and see which performance gains you can achieve.

1. Extend the sparse matrix-vector product kernel $y = Ax$ to $Y = AX$ with matrices X and Y of dimension $N \times K$ (with K small). Use one thread per row of the matrix A. (1 Point)

2. Compare the performance you obtain for:

   - storing X in a column-major fashion (one column after another)
   - storing X in a row-major fashion (one row after each other)
   - calling the standard matrix-vector product K times

   Investigate different values of K and N. (1 Point)

3. **Bonus:** Write a kernel where K threads work on each row (assume $K \in \{2, 4, 8, 16\}$) and X is stored in row-major fashion. Do you get better performance? Can you explain why? (1 Point)

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions