Exercise 3

360.252 - Computational Science on Many-Core Architectures
WS 2021

October 14, 2022

The following tasks are due by 23:59pm on Tuesday, November 8, 2022. Please document your answers in a PDF document and email the PDF (including your student ID in the file name 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!

> "Plagiarism is the fear of a blank page." — Mokokoma Mokhonoana

There is a dedicated environment set up for this exercise:

https://k40.360252.org/2022/ex3/
https://rtx3060.360252.org/2022/ex3/[1]

To have a common reference, please run all benchmarks for the report on both machines in order to also compare performance across different GPU generations.

Strided and Offset Memory Access (3 Points)

Reconsider the vector summation from Exercise 2 for vectors of size $N = 10^8$. Modify your GPU kernels so that

(a) only every k-th entry is summed, i.e. the parallel equivalent of (1 Point)

```
for (int i=0; i < N/k; ++i) z[i*k] = x[i*k] + y[i*k];
```

(b) the first k elements are skipped, i.e. the parallel equivalent of (1 Point)

```
for (int i=0; i < N-k; ++i) z[i+k] = x[i+k] + y[i+k];
```

and investigate values of k between 0 and 63.

Compute and plot the effective memory bandwidth (in GB/sec) for both cases (1 point). Only consider the entries that are actually touched (N/k and N-k, respectively) for the bandwidth calculation. What do you observe? Which general recommendation can you derive for future performance optimizations in more complicated scenarios?

[1] The machine equipped with the RTX 3060 GPU may not be available in the first few days after announcing the exercise.

Dense Matrix Transpose (6 Points)

Your friend is working with a dense matrix $A \in \mathbb{R}^{N \times N}$ and needs to transpose the matrix in-place.

In order to do so, your friend writes a specialized CUDA kernel. Unfortunately, your friend's kernel doesn't produce correct results and you need to help out.

Please help your friend as follows:

(a) Run your friend's kernel through `cuda-memcheck` and identify the problem(s). (1 Point)

(b) Fix your friend's kernel to yield correct results for all problem sizes, but do not optimize for performance. Determine the effective bandwidth (GB/sec) your GPU kernel achieves. (1 Point)

(c) As a first step to performance optimization, simplify the task by dropping the in-place requirement. That is, read from a matrix A and write the result (A^{T}) to a second matrix B. Optimize your code by avoiding strided memory access to global memory as much as possible. (1 Point)

(d) Now reconsider the in-place transposition. Use shared memory to load blocks of size 16×16, transpose them in shared memory, and write the result. Try to use non-strided global memory reads and writes for maximum bandwidth. Determine the effective bandwidth (GB/sec) you achieve. (2 Points)

(e) Compare the performance (GB/sec) you obtain for the non-optimized and the optimized kernel(s) for $N = 512, 1024, 2048, 4096$ on both GPU machines. What do you observe? (1 Point)

For simplicity, assume that N is a multiple of 16. Please make sure that the kernel computes correct results![2]

Full disclosure: Parts of this task are discussed in an NVIDIA developer blog post[3]. Feel free to get some inspiration from there.

Halloween Bonus: Trick AND Treat (no Points, but Chocolate)

Scare your lecturer by coming up with creative ways of making GPUs *appear* super-fast (100x speed-up and beyond) for vector additions. To demonstrate, write a program that benchmarks a (slow) *CPU version* of the vector addition from above and compare it to a *GPU version* you coded up above and in Exercise 2. You can play all tricks to make the CPU version appear slow and the GPU version appear fast. The result array has to be correct in both cases, but any other benchmarking crimes are permitted.

[2] It is better to have a slow kernel computing correct results rather than having a fast kernel computing wrong results ;-)

[3] https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/