Exercise 3

360.252 - Computational Science on Many-Core Architectures
WS 2020

November 4, 2020

The following tasks are due by 23:59pm on Tuesday, November 10, 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/ex3/.

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

Strided and Offset Memory Access (2 Points)

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

1. 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];
```

2. 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. 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?

Conjugate Gradients (5 Points total)

The conjugate gradient algorithm for solving a system of equations $Ax = b$ with symmetric and positive definite system matrix A can be formulated as follows:

Algorithm 1: Classical CG

1 Choose x_0;
2 $p_0 = r_0 = b - Ax_0$;
3 **for** $i = 0$ *to convergence* **do**
4 Compute and store Ap_i;
5 Compute $\langle p_i, Ap_i \rangle$;
6 $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$;
7 $x_{i+1} = x_i + \alpha_i p_i$;
8 $r_{i+1} = r_i - \alpha_i Ap_i$;
9 Compute $\langle r_{i+1}, r_{i+1} \rangle$;
10 Stop if $\langle r_{i+1}, r_{i+1} \rangle$ is small enough;
11 $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$;
12 $p_{i+1} = r_{i+1} + \beta_i p_i$;
13 **end**

Implement this algorithm in CUDA using double precision arithmetic and data types. Implement

1. a kernel for the matrix-vector product in line 4, (1 Point)
2. kernels for the vector operations in lines 5 and 9 as well as 7, 8, and 12. (1 Point)

When your implementation is completed, analyze the following:

1. Plot the time needed for convergence (reduction of the residual norm r by a factor of 10^6 compared to r_0) and the number of iterations for different system sizes. (1 Point)
2. Break down the total solution time for small system sizes (about 1000 unknows) and large system sizes (about 10^7 unknowns) on a per-kernel basis. Which parts of the implementation are worthwhile to optimize in later work? (1 Point)

Finally, develop a simple performance model based on latency and memory bandwidth for the required data transfers (assume five nonzero entries per row in the A). Compare it with the actual timings obtained above for different system sizes. Which approximate values for the latency and memory bandwidth do you get? (1 Point)

Hints

- Check `https://gtx1080.360252.org/2020/ex3/` for a code skeleton to start from.
- A common choice for x_0 is to use a vector of zeros and thus avoid the computation of Ax_0 in line 2.
- Consider starting with a sequential CPU-version and then incrementally move computations to the GPU.

Bonus point: Early Submission

Hand in your report by 23:59pm on Monday, November 9, 2020.