Exercise 5

360.252 - Computational Science on Many-Core Architectures
WS 2022

November 13, 2022

The following tasks are due by 23:59pm on Tuesday, November 22, 2022. Please document your answers (please add code listings in the appendix) in a PDF document and submit the PDF together with the code in TUWEL.

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!

"To steal ideas from one person is plagiarism;
to steal from many is research." — Steven Wright

There is a dedicated environment set up for this exercise:

https://k40.360252.org/2022/ex5/
https://rtx3060.360252.org/2022/ex5/

To have a common reference, please run all benchmarks for the report on both machines in order to see differences across GPU generations.

Performance Modeling: Parameter Identification (5 Points total)

Estimate the following quantities based on suitable experiments (e.g. vector addition):

(a) PCI Express latency for `cudaMemcpy()` in microseconds (1 Point)

(b) Kernel launch latency in microseconds (1 Point)

(c) Practical peak memory bandwidth in GB/sec (1 Point)

(d) Maximum number of `atomicAdd()` updates per second on a single global memory address (1 Point)

(e) Peak floating point rate (multiply-add, i.e. $\alpha \mathrel{+}= \beta * \gamma$) in double precision as GFLOPs/sec (1 Point)

In each case, come up with a small code snippet to measure these quantities. For each case, please explain why your particular code snippet is a good way of measuring the respective performance parameter. Try to design your experiments such that other spurious effects like e.g. `for`-loop overhead have no significant impact on your measurements.

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

(a) a kernel for the matrix-vector product in line 4, (1 Point)

(b) kernels for the vector operations in lines 5 and 9 as well as 7, 8, and 12. (2 Points)

To obtain good performance, make sure that you have no repeated memory allocations while the CG algorithm is iterating along.

When your implementation is completed, analyze the following:

(c) 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)

(d) 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)

Hints

- A CPU-only code to start from is provided in the online exercise environment.
- A common choice for x_0 is to use a vector of zeros and thus avoid the computation of Ax_0 in line 2.
- Incrementally move computations to the GPU. Small relative differences due to round-off errors on the order of 10^{-15} are normal; no need to worry about these.

Bonus point: The Need for Speed

Try to optimize your CG code for best performance on either the K40 or the RTX3060 GPU (your choice). Try to minimize the number of kernel launches and host-device-communications in each iteration. Also, consider different thread block sizes for the various kernels. Compare your optimized version to your non-optimized version for different system sizes.