
Exercise 5

360.252 - Computational Science on Many-Core Architectures
WS 2021

November 17, 2021

The following tasks are due by 23:59pm on Tuesday, November 23, 2021. 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!

“After some time passed in studying - and even imitating - the works of others, I would recommend the student to endeavour to be original, and to remember that originality should not be undiscovered plagiarism.” — Henry Peach Robinson

There is a dedicated environment set up for this exercise:

<https://gtx1080.360252.org/2021/ex5/>.

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

Multiple Dot Products (4 Points total)

Given a vector x of size N and K vectors $y_k, k = 1, \dots, K$ of size N and K being a multiple of 8, your friend coded up a method to compute the dot products

$$\langle x, y_1 \rangle, \dots, \langle x, y_K \rangle$$

by calling the vendor-tuned CUBLAS library. It turns out that this operation is the bottleneck for a very important application; your friend is proud that this approach is faster than a previous hand-rolled implementation and claims that “there’s nothing one can do to make it any faster”.

Based on the code provided at the URL above, demonstrate that you already have the skills to beat a vendor-tuned library approach:

1. Write a specialized kernel to compute 8 dot products concurrently (use `atomicAdd()` as in Exercise 2). (1 Point)
2. Add a loop around that kernel to call it $K/8$ times to correctly compute the K dot products $\langle x, y_1 \rangle, \dots, \langle x, y_K \rangle$. (1 Point)

3. Plot the execution times for vectors of size N between 10^3 and 10^6 for values of $K = 8, 16, 24, 32$. (1 Point)
4. Outline (but do not code) an approach that is similarly fast for general values of K . (1 Point)

Make sure that your code works for general values of N .

Pipelined Conjugate Gradients (4 Points total)

In the lecture we discussed the following pipelined conjugate gradient implementation:

```

1 Choose  $x_0$ ;
2  $p_0 = r_0 = b - Ax_0$ ;
3 Compute and store  $Ap_0$ ;
4  $\alpha_0 = \langle r_0, r_0 \rangle / \langle p_0, Ap_0 \rangle$ ;
5  $\beta_0 = \alpha_0^2 \langle Ap_0, Ap_0 \rangle / \langle r_0, r_0 \rangle - 1$ ;
6 for  $i = 1$  to convergence do
7    $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$ ;
8    $r_i = r_{i-1} - \alpha_{i-1} Ap_{i-1}$ ;
9    $p_i = r_i + \beta_{i-1} p_{i-1}$ ;
10  Compute and store  $Ap_i$ ;
11  Compute  $\langle Ap_i, Ap_i \rangle, \langle p_i, Ap_i \rangle$ ;
12  Compute  $\langle r_i, r_i \rangle$ ;
13   $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$ ;
14   $\beta_i = \alpha_i^2 \langle Ap_i, Ap_i \rangle / \langle r_i, r_i \rangle - 1$ ;
15 end
```

Implement this algorithm in CUDA using double precision arithmetic and data types. Make sure only two kernels are launched per iteration. (3 points) A reference implementation of a classical conjugate gradient implementation can be found at the URL above for reference. Feel free to use the initial guess $x_0 \equiv 0$.

Finally, compare the execution time per iteration for different unknowns (10^3 to about 10^7) for the classical implementation and your pipelined implementation. Which performance benefits do you identify? (1 Point)

Bonus point: Early Submission

Hand in your report by 23:59pm on Monday, November 22, 2021.