Exercise 7

360.252 - Computational Science on Many-Core Architectures
WS 2021

December 1, 2021

The following tasks are due by 23:59pm on Tuesday, December 14, 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!

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

There is a dedicated environment set up for this exercise:

https://gtx1080.360252.org/2021/ex7/.

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

Atomic Add, Warp Shuffles and Thread Configuration (3 Points)

In earlier exercises we have looked at `atomicAdd()` as a method to avoid a second kernel call for reductions. Let us have a closer look at performance implications:

1. Prepare the following kernels for computing the vector dot product $\langle x, y \rangle$ (1 Point):
   - No use of atomicAdd(). Copy partial result from each work group to the CPU, sum there.
   - No use of atomicAdd(), sum partial result from each work group on the GPU in a second kernel.
   - Using atomicAdd() once for each work group.
   - Using atomicAdd() once for each warp.

   Please consult your prior exercises - you may have coded these kernels there already.
2. Run performance comparisons for the above four scenarios for $N \in \{10^3, 3 \cdot 10^3, 10^4, \dots, 10^7\}$ as vector sizes and the following thread configurations (1 Point):
   - 128 blocks with 128 threads each
   - 256 blocks with 256 threads each

- 512 blocks with 512 threads each
- (N+255)/256 blocks with 256 threads each

3. Please comment on the performance you observe and explain any differences between the approaches. What difference does it make if you need the result on the CPU or on the GPU? (1 Point)

Dot Product with OpenCL (4 Points)

Given vectors of size N,

1. write an OpenCL kernel to compute the dot product of two vectors (1 Point) based on the provided code skeleton,
2. compare the performance of your dot product kernel with your existing CUDA implementation(s) on the GTX 1080 GPU. Make sure to select the NVIDIA OpenCL platform. (1 Point)
3. compare the performance of your dot product kernel on the CPU. Make sure to select an OpenCL platform with CPU support. (1 Point)
4. measure the time it takes to build OpenCL programs for different program sizes. Find a way to generate M different variations of the dot product kernel and measure the kernel compilation time with respect to $M \in \{1, 10, 20, 30, \ldots, 100\}$. Compare the time it takes to actually compile the kernels versus the time it takes to retrieve the kernels from cache. (1 Point)

OpenCL Sparse Matrix-Vector Product Kernels (3 Points)

You work on a new application for which you need fast sparse matrix-vector products. Unlike previous applications using finite differences on regular grids, you encounter matrices of two kinds: The first kind is the well-known 2D finite difference stencil, the second type consists of hundreds of nonzeros in each row (a random number between 200 and 2000).

Write fast sparse matrix-vector product kernels in OpenCL as follows:

1. Port the CUDA-based sparse matrix-vector product kernel from earlier exercises over to OpenCL, (1 Point)
2. write a fast kernel for sparse matrices with hundreds of nonzeros per row by using a full work group per row, (1 Point)
3. compare the performance of the two kernels for each of the two matrix types. (1 Point)

The necessary code to assemble the sparse matrices is provided. Please ensure that your kernels compute the correct results. :-)

Also, keep in mind that the matrix with more nonzeros per row will consume more memory. Thus, choose the matrix sizes appropriately.

Bonus: Implement CUDA+OpenCL (CUCL) Approach (1 Point)

In the lecture we discussed a method to support both CUDA and OpenCL with just a single kernel code base. Check out the skeleton code provided and add the missing code such that the dot product kernel in the file `dot.cucl` can be used with either CUDA or OpenCL.