02614 High-Performance Computing - 2023 January 17, 2023

Assignment 3: GPU Matrix Multiplication GPU Poisson Problem

Your report must be handed in electronically on DTU Learn in PDF format!
Please add your source code in a ZIP file!
Remember the Addendum with the responsibilities!

Deadline: Friday, January 20, 2023 - at midnight!

Background

Matrix-matrix multiplication has the potential to be a **compute-bound** algorithm for large matrices sizes (e.g., for square matrices of size N it requires $\mathcal{O}(N^2)$ memory accesses and $\mathcal{O}(N^3)$ arithmetic operations). It is therefore a great example for showing the GPU's computing capabilities.

NVIDIA has developed a BLAS library for GPUs - called CUBLAS - which also includes a highly optimized DGEMM routine that takes the same function parameters as CBLAS. This library is able to reach the peak performance of the GPU.

The Jacobi method, on the other hand, is a **memory-bound** operation; it performs a constant small amount of flops per memory access for all N. The performance is limited by the effective memory bandwidth that can be reached, which is currently about 7x larger for GPUs compared to CPUs. Also the data transfers between host and device can be a significant bottleneck.

The Assignment

The purpose of this assignment is to gain experience with high-performance GPU programming by writing and optimizing OpenMP offload versions of the matrix-matrix multiplication function and the Jacobi iteration in the Possion problem.

We again consider the general matrix-matrix multiplication,

$$\mathbf{C} = \mathbf{AB},$$

which you have worked with in Assignment 1. The matrix sizes of $\mathbf{A}$ and $\mathbf{B}$ are $m \times k$ and $k \times n$, respectively, so that $\mathbf{C}$ has size $m \times n$, where m, n, and k are arbitrary positive integers.

We provide a new framework for matrix-matrix multiplication on DTU Learn with a new driver - similar to the one you used in Assignment 1 - but now for the NVIDIA compiler `nvc++`. For more information see the `README` provided with the driver and re-read the background text of Assignment 1.

You will also solve the Poisson problem again for the heat distribution in a small square room. Please re-read the background text of Assignment 2 for the details.

Part 1: Matrix-matrix multiplication

Again in this assignment, you will have to develop a library of functions, that all carry out matrix-matrix multiplication, as specified below. Your library functions will later be evaluated with the same driver as you are using, so you have to write your library according to the specifications provided.

1. **Standard OpenMP versions:** Write standard OpenMP implementations of the `mkn` version and the `blocked` version for matrix-matrix multiplication from Assignment 1. Also write the `lib` matrix-matrix multiplication function so that it can be used with a multi-threaded library. Measure and discuss the performance you see for these implementations running on 1 CPU. Hints:
   - Use your code from week 1 as the outset and make it work with the `nvc++` compiler (adopt the new Makefile provided on DTU Learn).
   - The new driver looks for `matmult_mkn_omp` and `matmult_blk_omp` as the function names in you library.
   - Note that your `matmult_lib` from Assignment 1 can be re-used without any changes to the code. The new driver has already been linked to a multi-threaded version of CBLAS (Intel's MKL library) and the number of threads can be controlled with `OMP_NUM_THREADS` just like your own OpenMP implementations, which allows for fair comparisons.

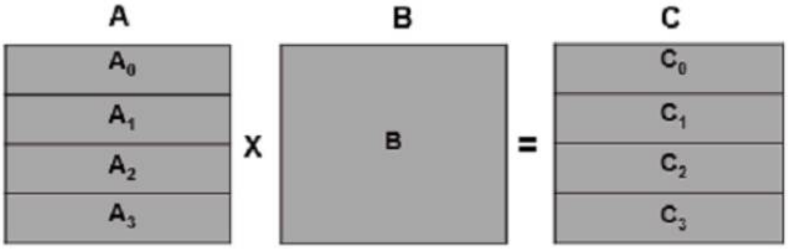
2. **OpenMP mkn_offload / mnk_offload versions:** Write an OpenMP offload implementation of the `mkn` version for matrix-matrix multiplication from Assignment 1. Discuss what limits the performance of the `mkn` version on the GPU and what you can do to improve it?

   Write an OpenMP offload implementation of the `mnk` version for matrix-matrix multiplication from Assignment 1. Experiment with the number of teams and threads per team and report your efforts. Discuss what limits the performance of the `mnk` version on the GPU and what you can do to improve it?

3. **OpenMP blk_offload version:** Modify the OpenMP offload implementation of the `mnk` version from question 2 to a new OpenMP offload `blk` version, where each thread computes more than one element of the C matrix. Experiement with the number of elements computed per thread and report your efforts. Discuss what limits the performance of this version on the GPU and what you can do to improve it? Hints:
   - The simplest way is to block the outermost `m`-loop (and only the outermost loop) in the same manner as in Assignment 1.
   - Use a constant (known at compile time) for the block size and not the `bs` function argument. It is important that the innermost loop is completely unrolled by the compiler for this blocking to be efficient: Look for `(completely unrolled)` in the compiler output.

4. **OpenMP asy_offload version:** The data transfer times can have a significant impact on the performance achieved in offloaded matrix-matrix multiplication. Find a way to measure how much time is used for data transfers in your implementations and discuss the implications for the performance of your previous offload versions. Then write a new version of matrix-matrix

multiplication (`asy_offload`) that uses asynchronous data transfers and offloads in order to overlap the data transfer and computation during execution. To do this the matrix-matrix multiplication has to be split into several independent sub-calculations, e.g.:



Notice that the four horizontal slabs $\mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2$, and $\mathbf{C}_3$, can be calculated idependently (why not apply vertical slabs - explain?). Use the best of the previous offload implementations as the starting point. Did you see any improvement (explain why or why not)?

For simplicity in the `asy` version, you may assume that m, n and k are integer multiples of the number of slabs.

5. **OpenMP `lib_offload` version:** Finally implement the `DGEMM` function for GPUs provided in the NVIDIA CUBLAS library. Explain your choices of arguments when calling the function interface. Compare and discuss the performance and speed-up observed between `lib` and `lib_offload`. Comment on the observed utilization of the hardware for these highly tuned libraries.

Notes (for everything above)

- The implementations should work for all matrix sizes (exception in 4.).
- Measure and discuss the performance for all your versions.
- Use profiler tools (`nsys` and `nv-nsight-cu`) repeatedly to analyze your implementations and use it to explain what limits their performance.
- Find usable information in the NVIDIA HPC Compilers User's Guide: https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/#openmp-use
- How would you suggest to improve your versions further? [*Note:* If you decide to improve further on any of the versions because you have some time to spare, please use the extra function `exp_offload` for this and leave the previous five functions in your library implemented acording to the questions in the assignment.]

Part 2: Poisson Problem

6. **OpenMP offload Jacobi:** Write an OpenMP offload implementation for solving the Poisson problem with Jacobi iterations on the GPU that uses the `map` clause to transfer data between the host and the device. Use your code from Assignment 2 as a starting point and use your fastest OpenMP version as the reference CPU Jacobi method in the following.

For simplicity in questions 6. and 7. it is sufficient to use the maximum iteration limit as the stopping criteria. For fair comparison

in these questions, you should remove any norm calculation from your reference code as well.

Write a second OpenMP offload implementation for solving the Poisson problem with Jacobi iterations on the GPU that uses `omp_target_memcpy()` to transfer data between the host and the device. Hints:

- Try first to make device versions of the functions `d_malloc_3d()` and `d_free_3d()` provided in the file `alloc_3d.cpp` with Assignment 2. Use `omp_target_alloc()` and return both pointers `***p` and `*a` (why both when we didn't in Assignment 2?). Make sure they work properly.
- When you transfer data between 3d volumes on the host and the device, leave the pointer arrays (`p`) untouched, and copy only between the data arrays (`a`).

7. **OpenMP offload Jacobi - dual GPUs:** Modify your second OpenMP offload implementation for solving the Poisson problem so that it can run simultaneously on two GPUs by splitting the task equally between them (you may assume that the grid size is an equal number). Hints:

   - Make two separate target regions - one for GPU0 and one for GPU1.
   - Use the `cudaDeviceEnablePeerAccess()` trick from the slides to access directly into arrays that have been allocated on the other GPU.
   - Asynchronous data transfers allows you to transfer to and from both GPUs simultaneously, saving time. Use the `nsys` timeline to check this.
   - Do not try to make this implementation with `map` clauses!

   What speed-up do you get from using two GPUs compared to one GPU? Try for different grid sizes and iterations. Is it as you would expect - explain?

8. **OpenMP offload norm calculation:** Make a second version of your first implementation from question 6 that supports the norm based stopping criteria used in Assignment 2. What is the observed performance hit from introducing the norm based stopping criteria in your GPU kernel? What is the performance hit in the CPU reference version? Discuss your findings.

Notes (for all part 2 questions above)

- Time your implementations for different grid sizes. Comment on the observed performance of your implementations and compare to the reference.
- Use profiler tools (`nsys` and `nv-nsight-cu`) to analyze your implementations and use it to explain their performance.