

ASSIGNMENT 3: GPU MATRIX MULTIPLICATION GPU POISSON PROBLEM

Your report must be handed in electronically on Campusnet in PDF format!

Deadline: latest on January 20, 2017 at 24:00!

Background

Matrix-matrix multiplication has for medium and large size matrices the potential to be a **compute-bound** operation (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 arguments as the CPU version.

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 . In that case, the performance is limited by the effective memory bandwidth that can be reached, which is currently about 3x larger for GPUs compared to the CPUs. Also the transfer CPU \leftrightarrow GPU add to the actual running times.

The Assignment

The prerequisite for this assignment is the previous two assignments in the course.

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 again provide a framework for matrix multiplication with a driver program on Campusnet, similar to the one you used in Assignment 1, but this time for the CUDA compiler `nvcc`. For more information see the README provided with the driver.

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

The purpose of this assignment is to gain experience with high-performance CUDA programming by writing and optimizing a matrix multiplication kernel and a Jacobi iteration kernel for the GPU. We begin with naive versions and proceed by stepwise refinement to get hopefully better performance.

Matrix multiplication

1. You will need a reference matrix multiply on the CPU for estimating speed-ups of the later GPU versions. For this you should use the DGEMM call you made for Assignment 1

linked to a multithreaded version of CBLAS (`-lptcblas`). The driver we have provided for Assignment 3 is linked to this version of CBLAS and allows for a fair comparison between the performance of CPU and GPU.

2. Write a first sequential implementation (`matmult_gpu1()`) of matrix multiplication on the GPU that uses only a single thread. It should work for all matrix sizes. Hints:
 - You need CUDA code to allocate memory on the GPU, transfer **A** and **B** to the GPU, transfer **C** back to the CPU, and free the allocated memory.

Time your kernel for small matrix sizes and compare to the reference DGEMM on the CPU.

3. Write a second naive implementation (`matmult_gpu2()`) of matrix multiplication on the GPU that uses one thread per element of **C**. It should work for all matrix sizes. Hints:
 - In your kernel, each thread should read one row of **A** and one column of **B** and compute and store the corresponding element of **C**.
 - It is convenient to use 2D thread blocks.

Time your kernel for different matrix sizes and compare to the reference DGEMM on the CPU. How much of the running time is used for CPU ↔ GPU transfers? Use a profiler tool (e.g., `nvvp`) to analyze your kernel and try to explain what limits its performance on the current hardware. Report your findings.

4. Improve the naive kernel by writing a third version (`matmult_gpu3()`) of matrix multiplication, where each thread computes exactly two elements of **C**. Hints:
 - Think about which second element a thread should compute (right neighbor, below neighbor, or others?). Not all choices are equally good.

Modify the third version into a new version (`matmult_gpu4()`), where each thread computes exactly 4 elements of **C**.

Time your kernels for different matrix sizes and compare to the reference DGEMM on the CPU and your other kernels. Did you get any speed-up and can you explain why/why not? Use a profiler tool (e.g., `nvvp`) to analyze your kernel and try to explain what limits its performance on the current hardware. Report your findings.

5. Return to the naive kernel again and modify it into a fifth version (`matmult_gpu5()`) that uses shared memory for reading the **A** and **B** matrix in order to improve the performance.

For simplicity in the fifth version, you may assume that m, n and k are integer multiples of the thread block size.

Time your kernels for different matrix sizes and compare to the reference DGEMM on the CPU and to your other kernels. Did you get any speed-up and can you explain why/why not?

Use a profiler tool (e.g., `nvvp`) to analyze your kernels and try to explain what limits its performance on the current hardware. Report your findings.

How would you suggest to improve this kernel further? [*Note: If you decide to improve further on any of the kernels because you have some time to spare, please use the extra function `matmult_gpu6()` for this and leave the previous five functions in your library implemented according to the questions in the assignment.*]

6. Finally compare with the DGEMM function for GPUs provided by Nvidia in the CUBLAS library by implementing the `matmult_gpulib()` function. Hints:

- `cublasDgemm()` is only for column major, so remember the exercise from week 1.

Poisson Problem

7. Use your code from Assignment 2 as a starting point for solving the Poisson problem and use the fastest CPU version as the reference Jacobi method in the following. For this assignment it is sufficient to use the maximum iteration limit as the stopping criteria, so you should remove any norm calculation from your reference code.

Write a sequential kernel for solving the Poisson problem with Jacobi iterations on the GPU that uses only one thread. Hints:

- Let the CPU initialize vectors u and f , then do a CPU \rightarrow GPU transfer of them.
- Call your Jacobi iteration kernel inside a CPU controlled iteration loop to get global synchronization between each iteration step.
- When all iterations are done, transfer the result from GPU \rightarrow CPU.

Time your kernel for small grid sizes and compare to the reference

8. Write a naive kernel for solving the Poisson problem with Jacobi iterations on the GPU that uses one thread per grid point and does NOT use shared memory.

Use a profiler tool (e.g., `nvvp`) to analyze your kernel and try to explain what limits its performance on the current hardware. Report your findings.

How would you suggest to improve this kernel further?

9. Modify your kernel 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 kernels - one for GPU0 and one for GPU1.
- Use `cudaDeviceEnablePeerAccess()` to avoid explicit copying of ghost points.

Comment on the observed performance of your implementation.

Goals

The following concepts are covered in this assignment

- How to write GPU kernels having 2D thread blocks and 2D grids.
- Using shared memory and "blocking" of matrices.
- How to improve performance by optimizing instructions throughput for GPU code.
- Performance profiling of incremental improvements to kernel code.
- Calling the CUBLAS linear algebra library.