HPCA Final Project Report

Dongkai Chen, Pengze Liu, Shuqi Yang[*1]

[1]**Dartmouth College**

ABSTRACT

We show how we implement conjugate gradient solver in vanilla CPU version, CPU version with OpenMP and GPU version.

Keywords: Cuda Programming, High Performance Computing, Conjugate Gradient Method

CPU AND OPENMP

According to the design of compressed row storage (CRS), we can use $ptr[i+1] - ptr[i]$ to index the *value* vector for a specific row i. We can use the col vector to index the position of candidate vector to do spare matrix-vector multiplication. The pseudo code of this implementation can be described as Alg.1 .

Algorithm 1 Pseudo code of CRS Matrix-Vecotr Multiplication

INPUT: val, col, ptr, n, v, mv

for $i \in n$ **do** $mv[i] = 0$

  for $j \in range(ptr[i], ptr[i+1])$ **do** $mv[i] = mv[i] + val[j] * mv[col[j]$

For the OpenMP part, we leverage the power of parallel computing, for each single loop code we use $\#pragma\ omp\ parallel\ for$ to call free threads working together in a machine to accelerate the speed. For tasks with respect to reduction, we call $\#pragma\ omp\ parallel\ for\ reduction(+:res)$ to decrease the runtime while prevent the res variable from non-atomic operation.

GPU IMPLEMENTATION

In our implementation, we fully leverage the advantages of Thrust. All the matrices and vectors are stored as $thrust :: device\_vector$ in our code, such that we do not care much about the data synchronization between the host and the device. Similar to the CPU version, a Conjugate Gradient Solver can be decomposed into three sub-tasks, i.e. vector dot-product, vector addition, and matrix multiplication. For Vector dot-product, we implemented it using $thrust :: innerproduct()$. For vector addition, initially, we tried $thrust :: transform()$. Nonetheless, it is proved that extra data copying may occur to avoid I/O conflicts in this implementation, so we replaced it by $thrust :: for\_each()$ to do the additions and subtractions in-place. The bottleneck of the solver lies in the sparse matrix multiplication operation. The sparse matrix A is converted into CRS format by default. To make this operation parallelizable, each thread captures a row-vector multiplication. Inspired by Mark Harris' Blog, we introduced Grid-Stride Loops in our MV kernel function.

OPTIMIZATION

Engineering We avoid allocating and coping additional variables by doing the vector self-increasing/-decreasing operations in-place. We also use some loop unrolling techniques for some for-loops.

Algorithm We implement the diagonal preconditioner in our code. However, in 2D Poisson problem, applying this preconditioner is equivalent to multiplying matrix A by a constant, since all the diagonal elements of A is 4. Therefore, this transformation would make no improvements to the properties of A. But it may be useful for other problems.

[*]equal contribution

Other possible improvements For matrix-vector multiplication part, we simply have each thread to calculate the result of each row. This works well in our problem, because the number of elements in each row is balanced. If using this solver for other problems, where the number of elements in each row is not balanced, this would cause workload imbalance to the threads and may not be a very high-performance solution. In such cases, parallelizing by elements instead of rows is a possible solution.

Some other preconditioners, such as Jacobi preconditioner, and incomplete Cholesky preconditioner, should be able to reduce the number of iterations.

PERFORMANCE RESULTS

Method	Time (ms)
CPU	1527.77
CPU-OpenMP	361.493
GPU	165.799

Table 1. CG computation for grid_size=256.

ACKNOWLEDGMENTS

Thanks Prof. Zhu's teaching and guidance in this term. Thanks the TAs. Thanks all the classmates and guest lecturers for their insightful presentations.