15-418 Project Proposal

Yijun (Jack) Dong, Enzhe Lu
yijund, enzhel
April 11, 2017

TITLE:

CuGB: Parallelizing Gradient Boosting on GPU *by Jack Dong and Enzhe Lu*

SUMMARY:

We are going to implement a parallel gradient boosting on the NVIDIA GPUs with CUDA.

BACKGROUND:

Gradient Boosting is a computational-dense algorithm in machine learning mainly used for decision trees. It combines several weak predictors to form a strong predicted model. The basic idea about Gradient Boosting Decision Tree is to utilize gradient decent and boosting technique to obtain a predictor that combines results from different decision trees to obtain better results.

Input: training set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:
$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$
2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:
    $$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n$$
    2. Fit a base learner $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^n$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:
    $$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right)$$
    4. Update the model:
    $$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$
3. Output $F_M(x)$.

Figure 1: Pseudocode of Gradient Boosting Decision Tree

Gradient Boosting started with a single constant predictor, usually used as the mean value of the training set. In each iteration, it computes the "pseudo-residual" of the observation and the last model result. Then it constructs a decision tree of that pseudo-residual with same input feature x. Gradient Boosting algorithm then computes a one-dimensional optimization of weight to minimize the lost of the weighted combination of last model and predictor of residual. The model then

updated as the weighted combination of last model and the residual predictor with the weighted just compute, and inter next iteration.

In gradient boosting, most of the computation steps could be paralleled to improve the computation speed. Specifically, the residual predictor is the most computational-dense part. It needs to browse the whole residual dataset to construct the predictor. Currently, the most recent XGBoost library for Gradient Boosting Computation is built on distributed system with the message passing parallel framework. It does not support single machine level parallelization. Thus a parallelization in single machine could benefit for small set problems with low budget.

RESOURCES:

We would like to use NVIDIA graphic processors in the GHC machines to test and run our code. Our project is based on the current distributed version Gradient Boosting python framework xgboost and past year OpenMP parallelized version project.

The start code would be xgboost and the past year openMP parallel project. Since the OpenMP version only utilizes the threads on CPU, it will only obtain some speed up, but not much. Since GPU has more computing power than CPU, our CUDA version implementation could achieve better performance speed up by utilizing more cores in the same time. Our project will also base on the CUDA C/C++ framework.

THE CHALLENGE:

In our implementation, to better utilize the GPU architecture, we might not use the same top-down approach like OpenMP parallelization. Since decision tree construction has to browse all dataset, bottom-up approach would be much better in CUDA environment. Thus, The first challenge point we will face is the data parallelization when constructing each level of the decision tree. The memory access pattern is a scan-like pattern when constructing each level of the decision tree. Thus we need to build a GPU version data structure to represent the whole decision tree.

The second challenge we will face is efficient data communication in CUDA. Currently, CUDA only supports shared memory (global memory) I/O as communication among all threads. This would be a huge bottleneck for us to improve. Since we need to communicate and split the dataset between each iteration, the expected communication will be O(N), as the computation is also O(N).

There might also be some other technique problems that we haven't considered during implementation. This project would be a great challenge to both of us. We hope that we could learn something from it.

GOALS AND DELIVERABLES:

PLAN TO ACHIEVE: We expect to achieve significant speed up from the xgboost by the end of our project.

HOPE TO ACHIEVE: We hope to achieve better speed up than the OpenMP CPU version. We will run xgboost on Xeon E5 as our reference. If we have significant speed up during our project,

we would like to combine our code with xgboost in order to make some contribution to the open source community.

DEMO: In the final computation, we would run xgboost version and our version of Gradient Boosting, and report the runtime as well as the speedup using graphs.

PLATFORM CHOICE:

We will use C++ as the programming language. CUDA platform integrates well with C++, and comparing with C, C++ has `std` library, which we can greatly benefit from.

We will use the GHC machines that contain NVIDIA GeForce GTX 1080 to develop and test our code. The GTX 1080 GPUs support CUDA, and the GHC machines have already installed NVIDIA CUDA C/C++ Complier. Meanwhile, we have experience with programming in CUDA on GHC machines from Assignment 2.

SCHEDULE:

Week 1 (April 16^{th}): Successfully run reference implementations to provide baseline and develop test harnesses for the project. Study gradient boosting in depth, and explore opportunities for parallelization.

Week 2 (April 23^{rd}): Develop a first and naive version of parallelized gradient boosting with CUDA. Test for correctness and performance against baseline implementation.

Checkpoint (April 25^{th}): Finish baseline, test harnesses and naive version of parallelized gradient boosting with CUDA.

Week 3 (April 30^{th}): Finish Checkpoint writeup. Explore all potential opportunities for parallelism in the code. Develop a second approach of parallelization.

Week 4 (May 7^{th}): Continue to optimize code by testing performance and finding bottlenecks in the implementation. Potentially explore a third approach of parallelization, if a reasonable one exists.

Week 5 (May 12^{th}): Finish final writeup and prepare for the presentation.