Spring 2022: Advanced Topics in Numerical Analysis: High Performance Computing
Homework 4
https://github.com/anmolsinghal/HPC-HW4

Submitted by: Anmol Singhal
as15151@nyu.edu

1. **Matrix-vector operations on a GPU**
   Pls find the code in the github repo. Below I report the bandwith achieved with matrix and vector of different sizes. Code has error checking for relative error using CPU multiplication as reference
   Dimension of matrix = size x size
   Dimension of vector = size x 1
   Bandwith reported in GB/s

   - CUDA server 1 using GTX TITAN black

Size	Bandwith
16	0.001722
32	0.024000
64	0.062320
128	0.140868
256	0.275677
512	0.561500
1024	1.142022
2048	2.271501
4096	2.915998
8192	3.121584

   - CUDA server 2 using GeForce RTX 2080 Ti

Size	Bandwith
16	0.003335
32	0.050305
64	0.175913
128	0.467391
256	0.787739
512	1.736775
1024	3.436714
2048	6.825378
4096	12.277954
8192	19.854298

- CUDA server 3 using TITAN V

Size	Bandwith
16	0.003018
32	0.056995
64	0.199693
128	0.476015
256	0.685791
512	1.605320
1024	3.365829
2048	6.847475
4096	13.753369
8192	22.793453

- CUDA server 4 using GTX TITAN X

Size	Bandwith
16	0.002691
32	0.039286
64	0.121382
128	0.257742
256	0.637519
512	1.393312
1024	2.592066
2048	5.138759
4096	6.186121
8192	6.006633

- CUDA server 5 using GTX TITAN Z

Size	Bandwith
16	0.000953
32	0.018697
64	0.049149
128	0.110021
256	0.228648
512	0.466921
1024	0.942962
2048	1.890439
4096	2.494926
8192	2.777564

2. **2D Jacobi method on a GPU**
   Pls find the code in the github repo. The code can be compiled using the `make all` command and run using `jacobi2d-cuda <N> <Num of iterations>` The code has checking for correctness and compares the same using CPU

3. **Update on final project**

   - **Technology and Literature Survey:** The goal was to understand the math and the existing literature on the given topic. In summary the math we have decided to proceed is as follows. The method we choose to follow is different from the traditional approached for matrix multiplication in that we employ a nonlinear processing function and reduce the problem to table lookups. This does not require any multiply-add operations instead we use vector quantization methods for similarity. In this wa y we have eliminated the multiply-ads by introducing a family of quantization functions.
   - **Problem formulation:** Let us assume that we have two matrices $A \in \mathbb{R}^{N \mathtt{x} D}$ and $B \in \mathbb{R}^{N \mathtt{x} M}$ with $N >> D \geq M$. We have a time budget τ and now our task becomes to construct three functions $g(.), h(.)$ and $f(.)$ along with constants α, β such that: $\|\alpha f(g(A), h(B)) + \beta - AB\|_F < \epsilon(\tau)\|AB\|_F$ for as small an error $\epsilon(\tau)$ as possible. The constants α and β are separate from $f(.,.)$ so that it can produce low bitwidth outputs (for e.g in range [0,255]) even when the entries of AB do not fall in the range.

   - **Methodology:** Here we assume the existence of a training set $\tilde{A}$, whose rows are selected from the same distribution as that of A. The first task that we achieved was product quantization. It consists of the following steps:
     - **Prototype learning:** Here we used offline training where we cluster the rows of $\tilde{A}$ (training set) using K-means algorithm on each of C disjoint subspaces to create C sets of K prototypes.
     - **Encoding function,** $g(a)$: Here we stored the most similar prototype to a as integer indices using $C \log_2(K)$ bits.
     - **Table Construction,** $h(B)$: Then we precomputed and stored the dot products between b and each prototype in each subspace as C lookup tables of size K.

  - **Aggregation,** $f(.,.)$: Finally we used the indices and the tables constructed above to lookup the estimated partial $a^T b$ in each subspace then aggregate the results across all C subspaces using the sum operation.

- **Hash Function Family, g(a):** Currently, we are working in developing the hash function. We have constructed the basic hash function to map the vector x to a set of 4 indices using 4 arrays (v^1, v^2, v^3, v^4) with 4 split thresholds. We lookup the split threshold for every node at all the four levels. If the vector element is above the threshold we map it to the index using the equation $i = 2i$ or $2i - 1$ based on whether it is assigned to left child or the right one in the hashing tree.
- **Modifications in the CPU implementation:** Used Quick sort for sorting the indices. Flattened the call heirarchy. Refactored out the parallelizable functions.
- **Future Goals:** Our next goals are to work on the algorithm and code to add next level to the hashing tree, work on optimizing the prototyping method further, using fast 8-bit aggregation for $f(.,.)$. Also, currently, we have implemented the above algorithms in CPU, hence we are also working to scale these to GPU for efficient GPU training(accelerating the clustering process) and inferencing(matrix multiplication).