2018
fall

Real-time weights mapping between huge networks

EECS 587 FINAL PROJECT

ETHAN, ZHANG

shuruiz@umich.edu

1. Problem statement

Solving network optimization problem always includes network weights calculation. Sometimes, the weights cannot be obtained, or it is hard to calculate the network weights by using existing network. Or maybe the weights just rely on other networks weights. In both cases, we are supposed to use two networks to get the results. For the first case, we can construct another network and establish another easier optimization problem to calculate the weights and then use those weights the update the target network according to some mapping rules. By utilizing the new network, we can solve the weights of the new network and create a corresponding "bridge" that provide rules to mapping weights from the new network to the target network. We define this as a weights mapping problem because we have to map the weights from source network to target network with some rules.

An example is that, in my current research, which is to solve dynamic vehicle route assignment, suppose I have a timed network that contains vehicle origin-destination trip information and it tells me in that region, how many people go from station i to j at time t. I want to find a good trajectory for each vehicle trip in the target network. And on the other side, I have a detailed graph (source network) contains the shorted path information for trips, whose nodes are discretized in time and space. As a long target trip can be divided by multiple small trips, and shorted path information for those small trips are stored in the source network, we can add up the shortest path information of smaller trips and form our target link value. That is to say, in order to get a weight of my target network, I need to know whether that links is in the shortest path of every trip in the source network or not. The source network contains corresponding information, then add them up. The rules that show how each long trip is divided into small trips, are stored in a "bridge" matrix that connecting the source matrix and the target matrix.

Another example is that in reinforcement learning filed, when calculating temporal abstract, the first step is to calculate options using the options policy, and then using the calculated options policy to get the action values with a certain rule provided by an environment model [1]. To make the training process faster, the updating can always stop option earlier. We can do that in a shared memory machine and compare which option to adopt and determine when to interrupt so that we can allocate limited computation resources to other segments. However, stop earlier and not exploring the planned environment may make the agent lose some information. With GPU, it is possible to update the whole network in real-time without neglecting information

The goal of this project is mapping values from huge source network/matrix to huge target network/matrix in real-time and in a scalable way.

2. Existing weights mapping methods

2.1 Serial

Given a target network, for example, with size of 500k weights, its weights can be stored in a target matrix *T*. In order to obtain *T*, a high-level but easier-to-solve network might be built with size of 4 million, whose weight elements is stored in source matrix *S*. “Bridge” matrix *R* is generated deterministically according to the network architecture of the source network and the target network. The “Bridge” matrix shares the same length of *S* and its elements contains routing information, that maps weights from *S* to *T*. With *S, R* and *T*. For example, the address that an *S* value should be sent to. A mapping task can be done with those three components.

The index of *R*, for example *i-th* position in *R*, means the *i-th* position in *S*. The value $\mathrm{R[i]} = \mathrm{j}$ indicates that the corresponding value in S, which is *S[i]* should be routed to of $\mathrm{T[j]}$. If there are multiple elements in R share the same value, which indicates they points to the same position in T, the corresponding value in T is the summation of values in S in those position. For example, if there are several elements in R share the same value, which are $\mathrm{R[0]} = 2$ and $\mathrm{R[1]} = 2$, then the corresponding value in T, which is $\mathrm{T[2]} = \mathrm{S[0]} + \mathrm{S[1]}$. The max value in R is the maximum index of T. Since both the size of T and S are huge, for example, 4 million elements in S and about .5 million elements in t for a real-world taxi driver and customer matching problem network, lots of computation are needed.

The most widely used way, or the general way, to compute the weights mapping problem is to write serial code do a “search and update” program step by step. That is to say, for elements in T, if we want to update T[j], first of all go to “bridge” matrix R to search for j, for each j value and the its corresponding index of R, use that information, fetch the corresponding value in S and use it to update T[j]. For the serial method, if there are 500k elements in T that are needed to be updated, the computation time with a 4-million-size S is 3113.655152 seconds, which is about 50 minutes.

2.2 Pytorch-GPU

A faster way to mapping the weights is to use Pytorch-GPU [2], which is a deep learning tool that typically being used to build neural networks. With Pytorch-GPU, the weights in target network and in source network are treated as tensors. For position j in target matrix, use the build-in function Pytorch.sum() to sum all corresponding source values that given by the “bridge” matrix. The pseudo code is:

```
Import Pytorch-GPU
for T[j], given S, R,
T[j] = Pytorch.sum([S[i], where R[i] == j ] )
```

With Pytorch-GPU, one mapping for a 4 million-size S and 500k-size T takes about 5 minutes, which is much faster than serial case shown above. However, that is still slow because some problems may take 100 or even more mapping tasks. And solving them in real time becomes a challenge with traditional methods.

3. Methodologies

3.1 Parallel architecture and algorithm

Thinking of the architecture of GPU, it is like that we solve the problem in real-time with massive threads. The idea is storing copy all matrix from host to device. Since the index of a “bridge” element shows the position of element in source matrix that we are needed when updating the target matrix. Therefore, for each element in source matrix, allocate a thread to it to fetch the value, and let the thread execute a “router”, which is built based on “bridge” matrix R, to reach the target position. The mechanism is shown in the diagram below:

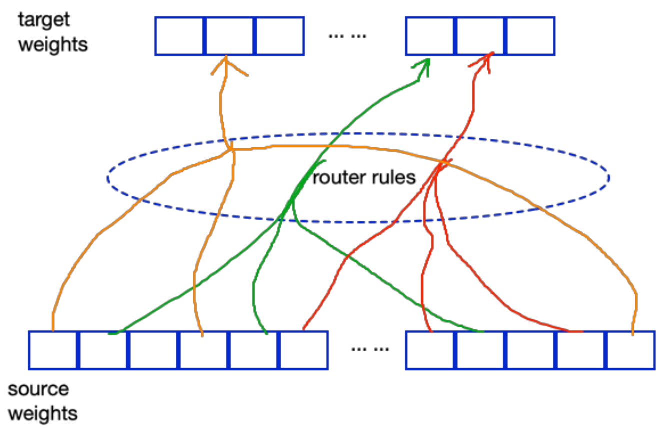

Figure 1, Algorithm illustration for weights mapping from source weights to target weights and the routing mechanism. Each arrow represents a thread that use a value in source weights matrix to update a value in target weights matrix. Router rules are generated from bridge matrix.

3.2 Threads management

Given source matrix S with size m (typically, m>10^6), bridge matrix R with size m and target matrix T with size n (n< m, typically n>10^5), use 2 dimensional block architecture and 2-dimensional grid architecture, with each block contains (32×32) threads and each grid has $[ceil(sqrt(m)/32) \ \times \ ceil(sqrt(m)/32)]$ blocks. This makes sure that each element in S will be assigned one thread to do the mapping task. Threads are managed by giving each thread a global index. The indexing method is shown below:

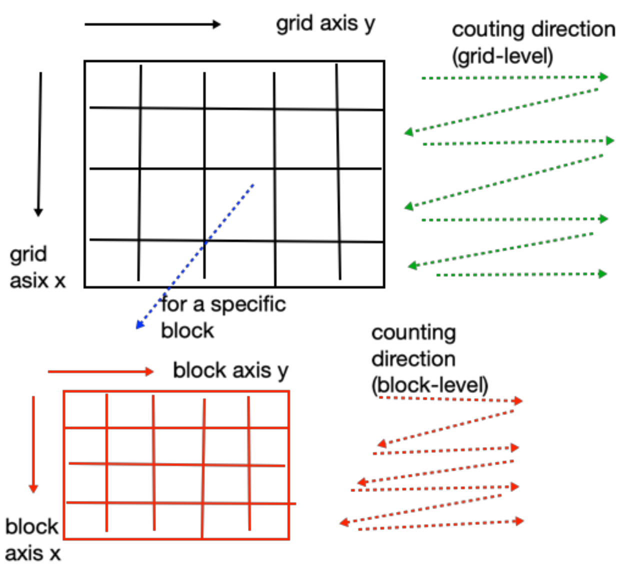

Figure 2, Threads management demonstration.

The indexing process contains two-level threads counting, which are block-level and grid-level. For a target thread, first of all, do a grid-level counting to sum all number of threads in previous blocks with the counting direction from shown above. Secondly, count all previous threads inside the block, adding the two counts to get a global index of the target thread.

3.3 Race condition issue

In the weights mapping problem, the size of target matrix is much smaller than source matrix S. According to the update rule of T, $\mathrm{T[j]} = \mathrm{S[a]} + \mathrm{S[b]} + \cdots$, it is possible that a race condition occurs when updating $\mathrm{T[j]}$ [3]. For example, when calculating $\mathrm{T[22]} = \mathrm{S[0]} + \mathrm{S[1]}$, S[0] is managed by thread 0 and S[1] is managed by thread 1. Both thread 0 and thread 1 can write T[22] after passing the router. If thread 1 reads the existing T[22] value after thread 0 did but before thread 0 write a new value back to T[22] address. The result of thread 0 will be covered by the result of thread 1, which result in a wrong update. The racing condition for the example is shown below:

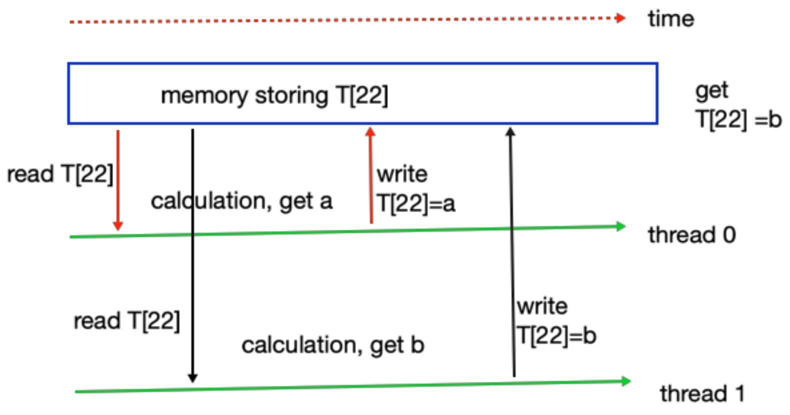

Figure 3, racing condition illustration when updating target matrix

In figure 3, we cannot get the correct update because we want to obtain $\mathrm{T}[22] = \mathrm{a} + \mathrm{b}$. However, with racing condition, we can only get what thread 1 obtains and the final output becomes$\mathrm{T}[22] = \mathrm{b}$. A write-protection mechanism should be constructed to make sure the target value cannot be updated asynchronously.

3.4 Reduction and atomic operation

Two methods to avoid the race condition, which are reduction and atomic operation [4], are always be adopted to solve the race condition. Reduction on GPU threads are generally tree-based approach used within each thread block. It is a good approach that do calculations without synchronization or barriers. An example of parallel reduction with interval addressing is shown below:

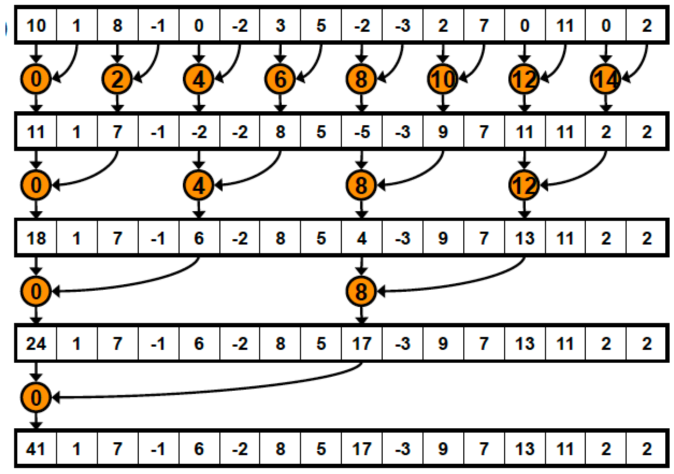

Figure 4, reduction with interval addressing [5].

However, reduction is always performed in block-level and threads used to do reduction are not randomly distributed. And global level reduction will be much complex than block-level reduction. In our problem, the threads are distributed randomly in the grid. For example, to update T[j], the threads that manage corresponding source values may located in different blocks like in block 1, block 2, block 9 and etc. The distribution of those threads has no pattern because generally the mapping process has to be done by may iteration and each iteration, we have different S, R and T. Using reduction on such a problem can only make the mapping process tend to get wrong results.

Considering this, setting locks on the target value is a more favorable way to do, which is an approach that atomic operations adopt.

In this problem, using atomicAdd() is a much wiser way to target value update comparing to traditional reduction methods. With atomicAdd, when the target value, e.g., T[j] is read by one thread, the thread adds a lock to T[j]. Before the thread writes its calculation result back to the address, other threads cannot access T[j]. Once the update is done by the thread, the corresponding lock is released and T[j] becomes available to other incoming threads. The mechanism is shown below:

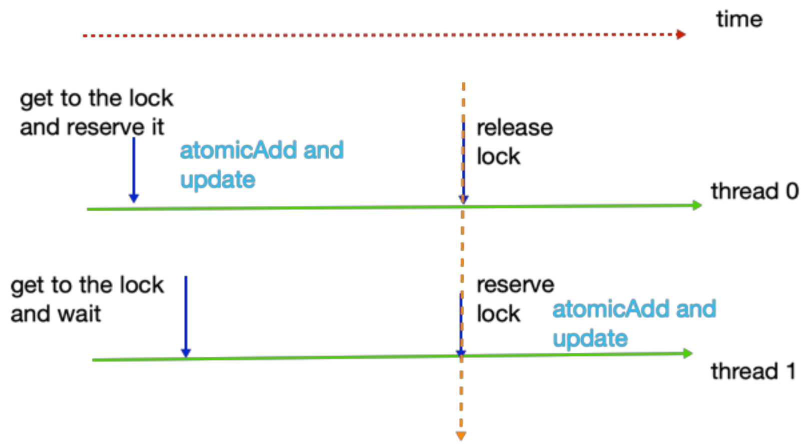

Figure 5, update target values with atomicAdd

With atomicAdd, the update of the target matrix value will occur in a synchronized way. Results of all corresponding threads can be stored and be used, that is to say, for target T[j], all elements in S that should be used are taken into account and no elements are skipped or neglected. This ensures the final result of T[j] is calculated correctly.

4. Performance analysis

4.1 General performance comparing to existing mapping methods, per iteration.

0.002464 ms for 4million size S, 500k million size T
0.002400 ms for 1million size S, 500k million size T
0.002368 ms for 20 million size S, 500k million size T
0.002496 ms for 40 million size S, 500k million size T

0.002464 ms for 4 million size S, 1 million size T
0.002432 ms for 4 million size S, 100k size T

0.015040 ms for 1 million size S, 10k size T
0.017696 ms for 4 million size S, 10k size T
0.018688 ms for 20 million size S, 10k size T
0.019328 ms for 40 million size S, 10k size T

0.017344 ms for 1 million size S, 20k size T
0.018720 ms for 4 million size S, 20k size T
0.019328 ms for 20 million size S, 20k size T
0.020032 ms for 40 million size S, 20k size T

Routing all tasks by 1 thread:
1219.812 ms for 400 million size S, 500k million size T (not scalable, time increases as size goes up)

For each thread i, since the job of finding the T[j] is already coded in the device function as a router, there is just one operation, which is to add its S value S[i] to the corresponding target address T[j]. According to the performance, we can claim that the speed limit for GPU threads in this task is about 0.002 ms.

4.2 Scalability

As we can see, even when dealing with 40 million size S and 1 million size T, the program can still handle it well. The reason is given in 1, that we only use GPU to a massive routing task and update the target address in parallel. The computation does not take too much time.

As we can see from 1, a really interesting thing is that the computation time (for each iteration) is largely depends on the size of T rather than the size of S. That is reasonable because with this parallel algorithm, even increase the size of S by 10 times not increase the computation tasks. As we can see from the architecture part, the number of blocks and threads are allocated based on the size of S to make sure each element in S is handled by one thread. And each thread just does one operation. However, with different size of T, the computation time is largely affected. With a large size of T, the run time is small because for each address in T, only few threads are racing to write their results to it. Each thread does not need to wait too long to write result to the target value. For example, when size of S is 4 million, as the size of T decreasing from 500k to 10k, for each address in T, number of average racing threads increase from 8 threads to 400, and the corresponding run time increase about 10 times. The total time for each thread that to finish its task can be formulated as below:

Total time = indexing time + routing time + time spending on waiting the atomic operation lock

Where the indexing time is time used to obtain the global index, routing time is the time that used to locate the source address and the target address with the global index and the routing rule. And the time spending on waiting the atomic operation lock is the time spending on race condition. Since the weights mapping only need to do one update (addition operation) for each thread, the update operation can be seen as real time.

Therefore, we can claim that the race condition dominates the time spending on this massive parallel algorithm.

For scalability, based on the performance, it is easy to find that this parallel weight mapping algorithm has a pretty good scalability when compared to traditional mapping methodology like Pytorch-GPU, the mapping task can be finished in real-time. By utilizing massive threads and a flexible grid to make sure each thread only does one mapping and updating task, the computation time issue brought by the huge weights size is solved by using the same size of threads. The increasing of size on source matrix does not affect computation time in the same rate. That is to say, increasing the source matrix S by 10 times does not increase the computation time by 10 times. As we can see, even with a 40 million size S, the total time still does not increase too much as the impact is partially eliminated by flexibly increasing the number of threads. The increasing on target network size plays an important role because it increases the waiting time for of atomic operation. And that effect cannot be dismissed by using more threads as the number of threads does not depends on the size of target matrix T.

4.3 GPU limit

In this problem, a speed limit of about 0.002 ms is achieved. According to the total time defined above, we can assume that indexing time and routing time are almost take 0 ms time because they are just simple addition and address indexing operations in global memory, and for each threads, there is only one update operation to finish its mapping task, we can claim that the GPU speed limit is about 0.002 ms for the mapping task and cannot be increased anymore.

5. Discussion

5.1 Achievements

In this project, I successfully parallelized the weights mapping task between huge tasks, which typically takes 5 minutes in Pytorch-GPU and about 50 minutes in serial hard-coded program for source network with a 4 million elements and target network with size 500k elements. For those traditional methods, weight elements addressing and updating are done in a serial way. That is to say, for an element in target network, search it in “bridge” matrix first and update it with corresponding source network weights and do it iteratively. The sequential addressing task takes lots of time because in serial case, there is only one “worker” and the “worker” need to do addressing one by one. Even one addressing task does not take too much time, the total time spent on mapping from one network to another network for the worker is tremendous when considering the huge network size.

The bottle network for the weights mapping task, as its name indicates, is the routing/mapping step one value in a network to another network. With the massive parallelism algorithm and its architecture provided in this project, we can flexibly construct millions of workers and let them do the routing and updating task at the same time. Since each routing and updating does not take too much time, the total time spent on the whole task is largely shortened and the weights mapping task is finished in real time. That is especially important in real world tasks. For example, in my current research, I have to deal with huge transportation networks. In order to get a link weight (target network), I need to know whether that link is in the shortest path of every trip matrix (bridge matrix) in a huge network which contains origin-destination information or not, and then fetch the corresponding value to update the target network weights. And to do dynamic route assignment, I have to finish as much mapping tasks as possible per second because real-world transportation network changes rapidly since lots of new trips are generated per second. Searching the bridge matrix is really time-consuming to check if the link in the bridge matrix or not. As I showed previously, 5 minutes for Pytorch-GPU to finish a full iteration mapping, which is totally not enough for real-time tasks like dynamic vehicle route assignment. With the parallel algorithm in this project, the task is solved in real-time. The bridge matrix contains both information of target network and the source network. Each thread is given a global index based on the index of bridge matrix. Therefore, it can easily obtain the corresponding target weight address and source weight value. Instead of letting the target network search in the bridge matrix, the mapping task is done from the source matrix side to the target network side. That avoid the searching step originally performed by the target matrix. For example, if the link is in the shortest path of every trip matrix, according to Figure 1, corresponding threads will be routed towards that link and update its value. If the link is not in the shortest path bridge matrix, no threads will be routed to that link therefore the weight on that link won’t be updated.

5.2 What to improve

This project uses massive parallelism to do real time weights mapping between huge networks. If thinking it more abstract, it can be thought as a compression operation that compress information of a huge network to a small network. By changing the routing and updating rules, it can be extended to many other areas. For example, feedforward artificial neural network [6], for weights in a target layer, their values are based on network weights of previous layer (source layer) and the corresponding activation function and node index (routing rules). Improvements can be made by making the algorithm can be more general and let it solve more general case.

Furthermore, since there are limits on number of total blocks in a grid and number of threads in a block for a Nvidia GPU, when testing the scalability, the program cannot solve extreme huge source matrix (with element size of 10^12 or above), though those extreme case are rare, there should be a method to solve them. For example, when solving national wide transportation networks, those extreme case can occur. Solving these cases using limited hardware source can also be treated as future improvements of the algorithm

Reference

[1] Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, *112*(1-2), 181-211.

[2] Paszke, A., Chintala, S., Collobert, R., Kavukcuoglu, K., Farabet, C., Bengio, S., ... & Mariethoz, J. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, may 2017.

[3] Boyer, M., Skadron, K., & Weimer, W. (2008, April). Automated dynamic analysis of CUDA programs. In Third Workshop on Software Tools for MultiCore Systems (p. 33).

[4] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008, August). Scalable parallel programming with CUDA. In ACM SIGGRAPH 2008 classes (p. 16). ACM.

[5] Mark, H. (2008). Optimizing parallel reduction in CUDA. NVIDIA CUDA SDK, 2.

[6] Montana, D. J., & Davis, L. (1989, August). Training Feedforward Neural Networks Using Genetic Algorithms. In IJCAI (Vol. 89, pp. 762-767).