CUDA: Solving All-Pairs Shortest Paths in Parallel

Xi Deng

Dartmouth College

ABSTRACT

In this project, I parallelized the dynamic programming solution of all-pairs shortest paths in CUDA. Trying to Improv the performance of parallelizing by take the advantage of shared memory of Blocks.

Keywords

CUDA, parallel computing, All-pairs shortest paths.

1. INTRODUCTION

All-pairs shortest paths problem is to find the shortest path between any pair of vertices in a graph. That is, given a weight directed graph **G=<V, E>** and a weight function **w: E=>R** which maps each edge to a real value, we need to find an output matrix whose position in row u and column v stands for the least weight path from u to v.

By solving all-pairs shortest paths problem, we can get the diagonal path of a graph which means the longest least weight path between any pair of vertices in the graph.

There are several methods to solve this problem. One way is to do single source shortest path for |V| times. Another is dynamic programming. Here I mainly worked on the dynamic programming method. The sequential dynamic programming solution of the problem will take O($|V|^3 \log |V|$) time, in which $||V|^3$ is related to a matrix multiplication operation which let me see the potential to parallelize.

2. SEQUENTIAL SOLUTION

Assuming that there is no negative-weight cycle which will make the weight decrease each cycle. Define an Adjacency matrix $W = (w_{IJ})$. In matrix W, w_{ij} represent the weight of edge (i , j). Define p as the shortest path from vertex i to vertex j within at most m edges. The shortest path between any pair of vertex has at most |V|-1 edges.

2.1 Structure of Shortest Path

If vertex i equals vertex j then it's obvious that p have 0 edges. If vertex i is not equal to vertex j, then decompose path p into p' (which is the shortest path goes from i to k) and edge (k, j). `

2.2 Recursive Solution

Define $l_{ij}^{(m)}$ as minimum weight of any paths from i to j with m edges. When there is no edge between any pair of vertices,

$$l_{ij}^{(0)} = \begin{cases} 0, & if\ i = j \\ \infty, & if\ i \neq j \end{cases}$$

When there are more than one edges, recursively calculate $l_{ij}^{(m)}$ from $l_{ij}^{(m-1)}$ with $l_{ij}^{(m)} = \min_{i \leq k \leq n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}$ until we reach the limit of edges. When m reaches |V|-1, matrix L = $l_{ij}^{(m)}$ turns out to be the output matrix of all-pairs shortest paths problem.

2.3 Sequential Pseudocode

This is the whole process of computing shortest paths between any pairs of vertices.

FAST-ALL-PAIRS-SHORTEST-PATHS(W)
1. n = W.rows
2. $L^{(1)} = W$
3. m = 1
4. while m < n – 1
5. let $L^{(2m)}$ be a new n x n matrix
6. $L^{(2m)}$ = EXTEND-SHORTEST-PATHS($L^{(m)}, L^{(m)}$)
7. m = 2m
8.return $L^{(m)}$

As m increase from 0 to |V|-1. The matrix L keeps changing. When m becomes larger than |V|-1, L turns out to be the result matrix and remain unchanged as m grow. So we let 2m to update m each time to cut down the iteration into lg(|V|-1).

In each iteration we compute L^{2m} from L^m in following function.

EXTEND-SHORTEST-PATHS(L,W)
1. n = L.rows
2. let L' = (l'_{ij}) be a new n x n matrix
3. for i = 1 to n
4. for j = 1 to n
5. $l'_{ij} = \infty$
6. for k = 1 to n
7. $l'_{ij} = \min (l'_{ij}, l_{ik} + w_{kj})$
8. return L

This structure is similar as matrix multiplication, which remind me of parallelize matrix multiplication with GPU. Accumulating sum on each location for the result matrix is independent to each other. We could consider the FAST-ALL-PAIRS-SHORTEST-PATHS function as a function executed on the host(CPU) and then parallelize the for loop part of matrix multiplication on devices(GPU) with CUDA,

3. PARALLEL SCHEMA

I chose CUDA as parallel method, because the matrix will have large number of entries as vertexes of the graph grows and CUDA has grids and blocks and threads which allows me to assign each location in matrix to a processor.

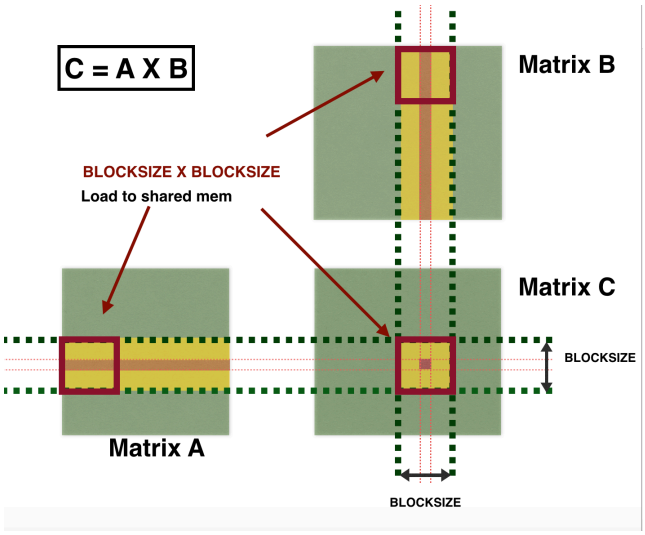
3.1 Naive Method

Consider C = A×B. Matrixes A, B, C are n × n matrixes, n is the number of vertices in the graph. The naïve way is assign each position in output matrix C to a thread in GPU, and compute all the position in parallel by looping over columns of A and row of B, and accumulating the sum. In this way, for every processor, each row of A will be read the number of columns B times from the device memory to thread blocks. So I consider using the shared memory in thread blocks to improve the performance.

3.2 Shared Memory

It takes time to fetch value from the global memory to block and threads. If we load the matrixes A, B to local memory, that will save a lot of time. However, the shared memory in GPU blocks is limit, for Capacity 2.0, the shared memory per block is 48KB. That is to say, if the matrix is large, we can't load the whole matrix to each block. So we need load only a part of matrix of A and B to shared memory in advance.

For each block we assign a same size sub matrix of C to it. Suppose we divide C into $n/blocksize \times n/blocksize$ of submatrices, each submatrix has a size of $blocksize \times blocksize$. Each position in this submatrix will be calculated by corresponding thread in a same size block. Thus, the thread block only needs rows and columns of A and B which are used to calculate the $blocksize \times blocksize$ submatrix of C. Considered the limit of shared memory, we load the same size submatrix of A and B to shared memory instead of once loading whole rows and columns. Then loop over A and B with a step of $blocksize$ length and accumulate the sum of corresponding block in C.



By using the shared memory, each row of A is read number of columns in B over $blocksize$ times, which would turn out to be time saving in parallel.

4. IMPLEMENT DETAILS

4.1 Host Code

Host code is the code that executes on CPU. The procedure goes as fellow: First, declare the matrix L as adjacent matrix on host, declare d_a and d_b as the matrixes that will be used on GPU device. I store the matrixes in arrays. Allocate CPU memory for L and allocate GPU global memory for d_a and d_b. Then use function genreateGraph to randomly generate directed weight graph adjacent matrix W and copy it to d_a on the device. While m < n – 1, do matrix multiplication like operation in parallel on the device(GPU). After the while loop, we copy the final result from device to host and compare with sequential result. Finally, release the memory on host and device.

4.1.1 Randomly Generate Graph

To generate a random graph as input of the problem. We use a simple function. It generates edge with probability p, And generate a random number ranging from 1 to range as weight.

Generate Graph

```
float* generateGraph(int n, int range, float p){

    float* l = (float *)malloc(n*n*sizeof(float));
    int seed = time(NULL);
    srand(seed);

    for(int j = 0 ; j < n ; ++j){
      for(int i = 0 ; i < n ; ++i){
        float po = ((float)rand()/(float)(RAND_MAX));
        if(po > p)
              l[j*n+i] = (float)(rand()%range)+1;
        else l[j*n+i] = FLT_MAX;            ;
      }
      l[j*n+j] = 0;//form vertex to itself is 0
    }
    return l;
}
```

4.1.2 Result Check

To check the result, just compare the result of parallel program with sequential one.

4.2 Kernel Code

Kernel code is code that will be executed on each thread. Kernel can't have any return value and recursive structure. Thus we need to make sure where should we put the result to and write the result to global memory in kernel. To do this, we compute index with blockIdx and threadIdx.

4.2.1 Naive Method

In naive way, the thread and block all have one dimension. The index stands for the result location in matrix C which current thread is related to.

Naive Method

```
__global__
void Mamulti(int n , float *a, float *b, float *c){
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int row = index / n;
    int col = index % n;
    col[index] = FLT_MAX;
    for (int i = 0 ; i < n ; i++)
      c[index] = min(c[index], a[row * n + i] + b[i * n + col]);
}
```

4.2.2 Shared Memory

When we use shared memory, there may be some write and read confliction. Therefore, we need to make sure when each thread read the Matrix in local memory, all position in matrix is ready. We also need to ensure that before loading new submatrix to local the previous computing in each thread have already finished. To do this, I use sync before and after read and write the shared memory.

Because each block is assigned to a block of same size in result matrix C, and each location in C is assigned to corresponding thread in that block, both block and thread have two dimension. We use index of block and thread to compute where do the corresponding submatrix of A and B starts.

The Machine I use has 4 CUDA devices, each one allows a total shared memory of 48KB per block. The maximum dimension 0 of block is 1024 and the maximum dimension 1 of block is 1024, within each block there are at most 1024 threads. So I divide C into sub blocks of size 32×32.

Parallel with Shared Memory

```
template <int BLOCK_SIZE>
__global__
void Mamulti(int n , float *A, float *B, float *C){
   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   int Abegin = by * BLOCK_SIZE * n;
   int Astep = BLOCK_SIZE;
   int Aend = Abegin + n - 1;
   int Bbegin = bx * BLOCK_SIZE;
   int Bstep = BLOCK_SIZE * n;
   int index = BLOCK_SIZE * by * n + BLOCK_SIZE * bx;
  float tempc = FLT_MAX;

   for(int a = Abegin,b = Bbegin;
     a <= Aend;
     a += Astep,b += Bstep){
 //declare shared mem
    __shared__ float Asub[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bsub[BLOCK_SIZE][BLOCK_SIZE];
    Asub[ty][tx] = A[a + n * ty + tx];
    Bsub[ty][tx] = B[b + n * ty + tx];
 //sync all threads
      __syncthreads();
    for(int k = 0; k < BLOCK_SIZE ; k++)
      tempc = min(tempc,Asub[ty][k] + Bsub[k][tx]);
    __syncthreads();
  }
  C[index + ty * n + tx] = tempc;//write the result
}
```

5. ANALYSIS OF RESULT

The sequential solution is O($n^3 logn$). By using the parallel method, the time complexity will turn down to O($nlogn$).

For shared memory, it reduces the number of operation that reading A 'row from global memory. In naïve way, each row of A is read from global to local for number of B's column times. In shared memory way, each row of A is read $\frac{number\ of\ B's\ column}{blocksize}$ times, and then read $number\ of\ B's\ column - \frac{number\ of\ B's\ column}{blocksize}$ times from local. Because read from local is much faster than read from global memory. That may turn outs to be time saving.

I use block size of 32×32 and the run parallel program on anthill. I use $2^n \times 2^n$ matrices(n = 5,6,…,14) as input matrix, run them in sequential, naive parallel way and shared memory parallel way respectively for 7 times and take the average running time of each kind . Table 1 shows the result.

Table 1. Running time Comparison

Matrix Size ($2^n \times 2^n$)	Sequential (sec)	Naïve Parallel (sec)	Shared Mem (sec)
n=5	0.001	0.00019	0.00010
n=6	0.011	0.00024	0.00014
n=7	0.100	0.001	0.00027
n=8	1.071	0.001	0.00028
n=9	11.937	0.06	0.007
n=10	108.041	1.091	0.155
n=11	911.511	0.02	0.833
n=12	---	0.07	5.237
n=13	---	16.45	40.105
n=14	---	18.013	385.826

The running time of parallel is 100+ times faster than sequential one. It's also interesting that, when the matrix is less or equal to 1024 * 1024 the parallel solution with shared memory turns out to be the fastest one, but as the size of matrix grow, speed of the shared mem solution decrease surprisingly. Let's consider the time used in memory read and load in both method.

For each row of A, naïve way read totally number of B's column times n. Suppose each time, it take tg to load data from global memory once, then total time is $tg \times n$. Suppose tl is the time that load data from shared memory, the total load time for row A is

$$T_{mem} = \left(n - \frac{n}{blocksize}\right) \times tl + \frac{n}{blocksize} \times tg.$$

Then, it may save,

$$\Delta T = (tg - tl) \times (n - \frac{n}{bs})$$

It seems that as n grows, the time that shared mem method saves will decrease. Remember we use a __syncthreads() function to wait for all the thread. I guess that may be the time consuming part. What's more, if the matrix that we load doesn't fit local memory, that will take more time. However, the parallel computing for the matrix multiplication like operation is worthwhile.

5.1 Conclusion

The parallel solution is 100+ times faster than the sequential one. The parallel solution with shared memory is faster than the naive way when the matrix size if less the 1024*1024.

6. REFERENCES

[1] CLRS. *Introduction to Algorithm, Third Edition..*

[2] Robert, Hochberg. 2012. *Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model*