Floyd-Warshall Algorithm using CUDA and PyBind11

Anshul Devnani ECE 277 Fall 2023

Introduction

- The Floyd-Warshall an APSP algorithm. It's used for finding the shortest paths in a weighted graph. It considers all pairs of vertices in the graph and updates the shortest path between every pair if a shorter path is found.
  - $O(n^3)$ runtime , $O(n^2)$ – space complexity

- Purpose: Investigate whether implementing the Floyd-Warshall algorithm on CUDA GPUs yields a speedup compared to a CPU implementation.

CPU implementation

```
for (int k = 0; k < vertices; k++) {
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            if (graph[i][k] != INF && graph[k][j] != INF &&
                graph[i][k] + graph[k][j] < graph[i][j]) {
                graph[i][j] = graph[i][k] + graph[k][j];
            }
        }
    }
}
```

- The algorithm uses three nested loops to iterate over all pairs of vertices (i, j) and an intermediate vertex k.
    - The outer loop (for (int k = 0; k < vertices; k++)) iterates over all vertices, considering each vertex as a potential intermediate vertex.
    - The middle loop (for (int i = 0; i < vertices; i++)) represents the source vertex.
    - The inner loop (for (int j = 0; j < vertices; j++)) represents the destination vertex.
- The if statement checks whether there is a shorter path from vertex i to vertex j through the intermediate vertex k
- The algorithm continues to iterate through all possible pairs of vertices and intermediate vertices until all shortest paths have been found.

Naïve GPU implementation

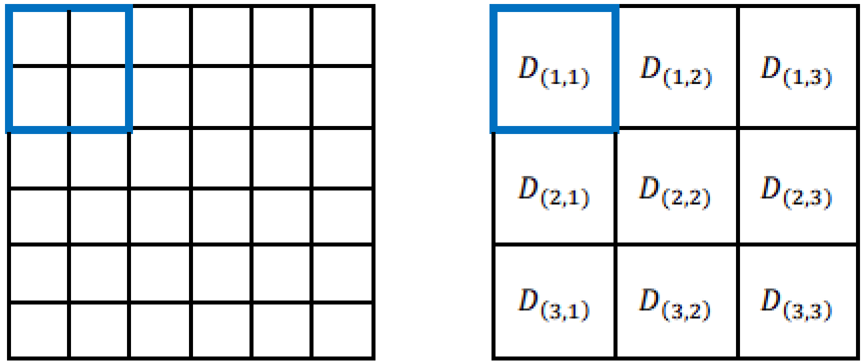
```
__global__ void floydWarshall(int* graph, int vertices, int k) {
    // Calculate the row index for the current thread within the grid
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index for the current thread within the grid
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    // Check if the thread's indices are within the valid range of vertices
    if (i < vertices && j < vertices) {
        // Calculate the flattened index corresponding to the 2D indices (i, j)
        int index = i * vertices + j;

        // Calculate the sum of distances from vertex i to k and from k to j
        int ikj = graph[i * vertices + k] + graph[k * vertices + j];

        // Update the graph matrix at index (i, j) if the computed path is shorter
        graph[index] = (graph[index] > ikj) ? ikj : graph[index];
    }
}
```

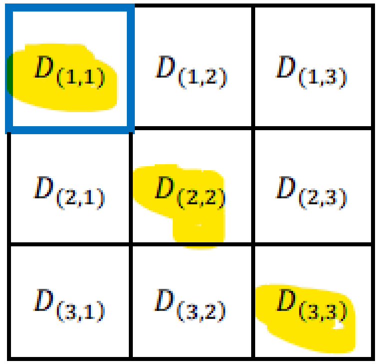
```
for (int k = 0; k < number_of_vert; k++) {
    floydWarshall << <numBlocks, threadsPerBlock >> > (d_matrix, number_of_vert, k);
    cudaDeviceSynchronize();
}
```

- **Not fully parallel**
  - K+1 iteration of algorithm is dependent on the Kth iteration
- Naïve approach
  - The algorithm is divided into a series of CUDA kernels, each responsible for updating a subset of the graph matrix.
  - Each thread within a block handles the computation for a specific pair of vertices.
  - The CUDA kernel is launched in a loop over all possible intermediate vertices.
  - Each iteration corresponds to one step in the Floyd-Warshall algorithm, updating distances through the chosen intermediate vertex.
- This approach is faster than CPU implementation but not optimal

Blocked GPU implementation

- Naïve approached can be optimized even further by using blocked approach – much better runtime
  - Divide the adjacency matrix into blocks and iterate over:
    - floor $(number\_of\_vertices\ +\ block\_\mathrm{dim}-\ 1)\ /\ block\_\mathrm{dim}$ times
      - 6 + 2 – 1 / 2 = floor(3.5) = 3 times
- 3 phase approach
  - Phase 1 – Self Dependent Phase
  - In the K-th iteration, the 1st phase is to compute $B \times B$ pivot block $D_{K,K}^{(K*B)}$
    - Essentially **compute Floyd Warshall Algorithm for all blocks down the diagonal**
    - However, can't be done in parallel since need k iteration computed in order to do k+1 computation



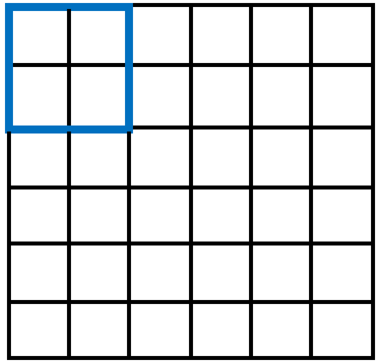


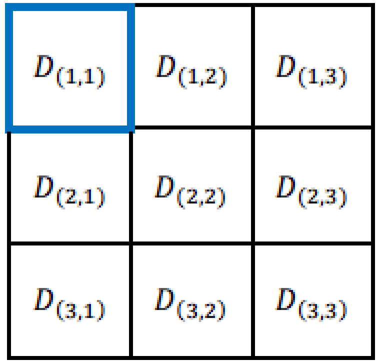
$$d^{(1)}(1,1) = \min\left(d^{(0)}(1,1), d^{(0)}(1,1) + d^{(0)}(1,1)\right)$$
$$d^{(1)}(1,2) = \min\left(d^{(0)}(1,2), d^{(0)}(1,1) + d^{(0)}(1,2)\right)$$
$$d^{(1)}(2,1) = \min\left(d^{(0)}(2,1), d^{(0)}(2,1) + d^{(0)}(1,1)\right)$$
$$d^{(1)}(2,2) = \min\left(d^{(0)}(2,2), d^{(0)}(2,1) + d^{(0)}(1,2)\right)$$

$$d^{(2)}(1,1) = \min\left(d^{(1)}(1,1), d^{(1)}(1,2) + d^{(1)}(2,1)\right)$$
$$d^{(2)}(1,2) = \min\left(d^{(1)}(1,2), d^{(1)}(1,2) + d^{(1)}(2,2)\right)$$
$$d^{(2)}(2,1) = \min\left(d^{(1)}(2,1), d^{(1)}(2,2) + d^{(1)}(2,1)\right)$$
$$d^{(2)}(2,2) = \min\left(d^{(1)}(2,2), d^{(1)}(2,2) + d^{(1)}(2,2)\right)$$

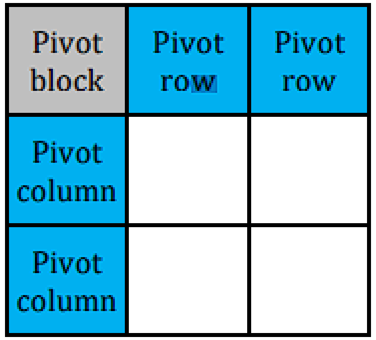
Images taken from https://ocw.nthu.edu.tw/ocw/upload/news_attach/482/10710%E5%91%A8%E5%BF%97%E9%81%A0%E6%95%99%E6%8E%88_%E5%B9%B3%E8%A1%8C%E7%A8%8B%E5%BC%8F%E8%A8%AD%E8%A8%88%E4%BD%9C%E6%A5%AD_HW4_rev3.pdf

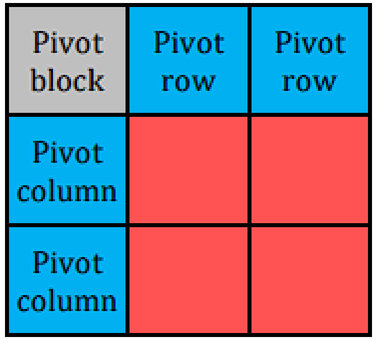
Blocked GPU implementation

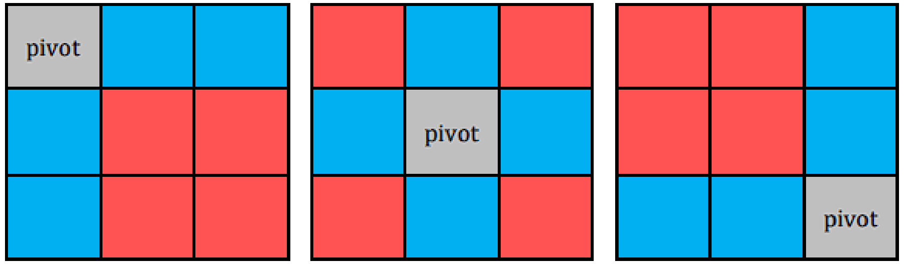




- 3 phase approach
  - Phase 2 – Compute Row and Column Blocks from the diagonal block
    - In the K-th iteration, it computes all $D_{K,h}^{(K*B)}$ and $D_{h,K}^{(K*B)}$ where $h \neq K$.
    - Essentially **compute FW for the kth row and the kth column of blocks**
  - Phase 3 – Compute all other block
    - In the K-th iteration, it computes all $D_{(h_1),(h_2)}^{(K*B)}$ where h_1 and h_2 $\neq K$.
    - Essentially **compute FW for all other blocks**
- There are 3 total iterations in this example







(a) Round 1 (b) Round 2 (c) Round 3

Images taken from https://ocw.nthu.edu.tw/ocw/upload/news_attach/482/10710%E5%91%A8%E5%BF%97%E9%81%A0%E6%95%99%E6%8E%88_%E5%B9%B3%E8%A1%8C%E7%A8%8B%E5%BC%8F%E8%A8%AD%E8%A8%88%E4%BD%9C%E6%A5%AD_HW4_rev3.pdf

Blocked GPU implementation

- Kernel Call

```
for (int k = 0; k < blocks; k++) {

    fw_phase_1_kernel << <1, block_dim >> > (d_matrix, number_of_vert, k);

    fw_phase_2_kernel << <blocks, block_dim >> > (d_matrix, number_of_vert, k);

    fw_phase_3_kernel << <phase3_grid, block_dim >> > (d_matrix, number_of_vert, k);

}
```

- Additional optimizations
  - Because FW is a memory bound application, faster memory access is very beneficial
    - **Shared memory** in all phases
      - Load each sub graph so all threads can have quick access to the data (TB/s vs GB/s)
      - Do computation with the shared memory object
      - Then copy back into global memory

```
__global__ void fw_phase_1_kernel(int* graph, int n, int k )
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Calculate offset for the current iteration
    int offset = (k * BLOCK_DIM);

    // Load data from global memory to shared memory
    __shared__ int shared_graph[BLOCK_DIM][BLOCK_DIM];

    // sync
    __syncthreads();

    shared_graph[ty][tx] = graph[(offset + ty) * n + (offset + tx)];

    // perform sync
    __syncthreads();

    // Perform the Floyd-Warshall computation
    int sum = 0;
    for (int k = 0; k < BLOCK_DIM; ++k) {
        sum = shared_graph[ty][k] + shared_graph[k][tx];

        // Update the minimum distance in the shared memory
        if (sum < shared_graph[ty][tx]) {
            shared_graph[ty][tx] = sum;
        }
    }
    __syncthreads();

    graph[(offset + ty) * n + (offset + tx)] = shared_graph[ty][tx];
}
```

Using Pybind11

- 4 functions wrapped with pybind11

```
PYBIND11_MODULE(FloydWarshallCuda, m) {
    m.def("run_blocked_fw_cuda", &py_run_blocked_fw_cuda, "Run Blocked Floyd-Warshall on GPU");
    m.def("run_naive_fw_cuda", &py_run_naive_fw_cuda, "Run Naive Floyd-Warshall on GPU");
    m.def("generateRandomAdjacencyMatrix", &py_generateRandomAdjacencyMatrix, "Generate random adjacency matrix");
    m.def("floydWarshall_CPU", &py_floydWarshall_CPU, "Run Floyd-Warshall algorithm on CPU");
}
```

- Profile.py
  - Python script to profile the CPU / Naive / Blocked approaches to FW Algo
    - Used to gather results in the next slides
  - Explained in depth in the Demo

Experiments / Results

- 2 Experiments
  - Measure speedup and runtime between the CPU approach vs the naïve/blocked GPU approach's with **varying the number of vertices but keeping block size constant (16)**
  - Measure speedup between the CPU approach vs the block GPU approach **with varying the block size but keeping number of vertices at 2048**
- How was time measured?
  - CPU – std::chrono
  - GPU – cudaEvents (start and stop)
  - **Only measures computation time, does not consider memory transfer time on GPU**
- Used ITS-E4309-02
  - GPU: NVIDIA RTX A2000
  - CPU: i7-11700k

Number of Vertices	FW CPU Runtime (ms)	Speedup [Runtime (ms)] Naive FW GPU	Speedup [Runtime (ms)] Blocked FW GPU	All Resulting Matrices Match?
16	0.01	0.02 [0.41]	0.43 [0.02]	True
32	0.05	0.03 [1.76]	2.17 [0.02]	True
64	0.27	0.13 [2.1]	6.87 [0.04]	True
128	1.69	0.42 [4.0]	17.58 [0.1]	True
256	11.44	1.41 [8.13]	50.54 [0.23]	True
512	80.17	4.1 [19.55]	96.05 [0.83]	True
1024	605.98	7.51 [80.74]	117.35 [5.16]	True
2048	5024.83	11.61 [432.83]	141.25 [35.57]	True
4096	40021.29	12.21 [3276.42]	187.46 [213.49]	True
8192	314419.16	14.47 [21724.48]	167.64 [1875.53]	True
16384	2484668.0	14.6 [170129.89]	117.55 [21136.68]	True

# Vertices	Block Size	Speed Up over CPU
2048	4	~37 times
2048	8	~98 times
2048	16	~381 times
2048	32	~558 times
2048	64	~ 27218 times

References

1. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Asmita-Gautam-Spring-2019.pdf
2. https://ieeexplore.ieee.org/document/9066330
3. https://saadmahmud14.medium.com/parallel-programming-with-cuda-tutorial-part-4-the-floyd-warshall-algorithm-5e1281c46bf6
4. https://arxiv.org/abs/1811.01201
5. https://dl.acm.org/doi/pdf/10.1145/3431379.3460651

Demo!
* Please look at README to run code