实现方式

总共使用了三个 GPU kernal——`selfKernel`，`crossKernel`，`globalKernal`，分别表示参考优化方法的三个阶段。

三个 kernal 的 Block 大小均为 $b \times b$，即分块的大小。（此处取 $b = 32$，即达到刚好最大线程数 1024）

`selfKernel` 中包含 1×1 个线程块，`crossKernel` 中包含 $\left\lceil \frac{n}{b} \right\rceil \times 2$ 个线程块（除去与当前块编号相同的，分别表示十字块的行和列），`globalKernel` 中包含 $\left\lceil \frac{n}{b} \right\rceil \times \left\lceil \frac{n}{b} \right\rceil$ 个线程块（表示剩余块）。

在 `selfKernel` 中使用一个 $b \times b$ 大小的共享内存数组，用于存储计算过程中的临时值。

在 `crossKernel` 中使用两个 $b \times b$ 大小的共享内存数组，用于临时存储当前十字架中块的值和当前中心块的值。

在 `globalKernel` 中使用两个 $b \times b$ 大小的共享内存数组，用于临时存储更新当前块所需两个十字架中块的值。

具体实现如下：

```
__global__ void kernel(int n, int k, int *graph) {
    auto i = blockIdx.y * blockDim.y + threadIdx.y;
    auto j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n && j < n) {
        graph[i * n + j] = min(graph[i * n + j], graph[i * n + k] + graph[k * n + j]);
    }
}

__global__ void selfKernel(int n, int p, int *graph) {
    __shared__ int sharedGraph[b][b];
    const auto i = threadIdx.y;
    const auto j = threadIdx.x;
    const auto realI = b * p + i;
    const auto realJ = b * p + j;
    const auto id = realI * n + realJ;
    if (realI < n && realJ < n) {
        sharedGraph[i][j] = graph[id];
    } else {
        sharedGraph[i][j] = 1000000000;
    }
    __syncthreads();
    int newDis;
    for (int k = 0; k < b; ++k) {
        newDis = sharedGraph[i][k] + sharedGraph[k][j];
        __syncthreads();
        if (newDis < sharedGraph[i][j]) {
            sharedGraph[i][j] = newDis;
        }
        __syncthreads();
    }
    if (realI < n && realJ < n) {
        graph[id] = sharedGraph[i][j];
    }
}

__global__ void crossKernel(int n, int p, int *graph) {
    if (blockIdx.x == p) return;
    __shared__ int sharedGraph[b][b];
    const auto i = threadIdx.y;
    const auto j = threadIdx.x;
    auto realI1 = b * p + i;
    auto realJ1 = b * p + j;
    const auto id1 = realI1 * n + realJ1;
    if (realI1 < n && realJ1 < n) {
        sharedGraph[i][j] = graph[id1];
    } else {
        sharedGraph[i][j] = 1000000000;
    }
    __syncthreads();
    auto realI2 = realI1;
    auto realJ2 = realJ1;
    if (blockIdx.y == 0) {
        realI2 = b * blockIdx.x + i;
    } else {
        realJ2 = b * blockIdx.x + j;
    }
    __shared__ int sharedNewGraph[b][b];
    const auto id2 = realI2 * n + realJ2;
    int minDis;
    if (realI2 < n && realJ2 < n) {
        minDis = sharedNewGraph[i][j] = graph[id2];
    } else {
        minDis = sharedNewGraph[i][j] = 1000000000;
    }
    __syncthreads();
    int newDis;
    if (blockIdx.y == 0) {
        for (int k = 0; k < b; ++k) {
            newDis = sharedNewGraph[i][k] + sharedGraph[k][j];
            if (newDis < minDis) {
                minDis = newDis;
            }
            __syncthreads();
            sharedNewGraph[i][j] = minDis;
            __syncthreads();
        }
    } else {
        for (int k = 0; k < b; ++k) {
            newDis = sharedGraph[i][k] + sharedNewGraph[k][j];
            if (newDis < minDis) {
                minDis = newDis;
            }
            __syncthreads();
            sharedNewGraph[i][j] = minDis;
            __syncthreads();
        }

    }
    if (realI2 < n && realJ2 < n) {
        graph[id2] = sharedNewGraph[i][j];
    }
}

__global__ void globalKernel(int n, int p, int *graph) {
    if (blockIdx.x == p || blockIdx.y == p) return;
    __shared__ int sharedRowGraph[b][b], sharedColGraph[b][b];
    const auto i = threadIdx.y;
    const auto j = threadIdx.x;
    auto realI1 = b * p + i;
    auto realJ1 = b * p + j;
    const auto id1 = realI1 * n + realJ1;
    auto realI2 = b * blockIdx.y + i;
    auto realJ2 = b * blockIdx.x + j;
    const auto id2 = realI2 * n + realJ2;
    if (realI1 < n && realJ2 < n) {
        sharedRowGraph[i][j] = graph[realI1 * n + realJ2];
    } else {
        sharedRowGraph[i][j] = 1000000000;
    }
    if (realI2 < n && realJ1 < n) {
        sharedColGraph[i][j] = graph[realI2 * n + realJ1];
    } else {
        sharedColGraph[i][j] = 1000000000;
    }
    __syncthreads();
    if (realI2 < n && realJ2 < n) {
        int minDis = graph[id2], newDis;
        for (int k = 0; k < b; ++k) {
            newDis = sharedColGraph[i][k] + sharedRowGraph[k][j];
            if (newDis < minDis) {
                minDis = newDis;
            }
        }
        graph[id2] = minDis;
    }
}

```

不同图规模加速比

图规模 n	朴素实现运行时间 (ms)	运行时间 (ms)	加速比
1000	14.756778	2.970092	4.97
2500	377.112280	25.343865	14.88
5000	2970.998835	158.762596	18.71
7500	10013.331400	515.264827	19.43
10000	22616.127009	1195.623236	18.91