实验二：全源最短路

1. 代码实现

首先在阅读过实验二的实验书后，确立了基本思路，参考给出的提示，`Cuda` 全源最短路径实现分为三个阶段：

- `first_stage`：在第一阶段，针对对角线中心块的内部执行 **Floyd-Warshall 算法**
- `second_stage`：在第二阶段，针对每一中心块上下左右的块组成十字块，用中心块的结果和十字块中原本的其他结果更新十字块。
- `third_stage`：在第三阶段，**用“十字块”的结果更新剩余的块。**

如此便实现了基于 `Cuda` 编程的全源最短路算法，具体的实现有如下的注意点：

- 二级分块策略
- 线程块大小设置
- 寄存器加速优化

初版实现

在第一版实现的时候，对于线程块的二级分块策略我采取了 1×5 的方式，每个线程块的大小为 32×32, 并没有采取额外的寄存器加速处理，如此情况下在第二和第三阶段的处理过程中，每个32×32的线程块可以处理 $32 \times 32 \times 5$ 个数据，同时利用`shared_memory` 将数据存在共享内存中进行计算加速。但是这种策略带来的优化效果并不明显，循环的迭代次数相比原本的基础实现也没有减少，最终在 $n = 10000$ 的数据规模下的最优时间为 $750ms$，未能达到性能线。

终版实现

在第一版实现的基础上，我首先优化了原本的二级分块策略，由原本的的一维分块转变成 $m \times m$ 的二级分块策略，在后续的参数调整过程中发现 $m = 3$ 的效果最佳，每个线程块的大小为 16×16 ，同时在第二和第三阶段：每个小块至于寄存器中，各由一个线程计算，如此可以获得更大的加速效果。最终在 $n = 10000$ 的数据规模下的最优时间为 $750ms$

2. 实验效果

下面主要展示终版实现和 `baseline` 的对比结果

数据规模	baseline	Mine	加速比
1000	14.925	1.496	9.976
2500	377.032	12.898	29.233
5000	2971.682	81.012	36.682
7500	10015.634	246.659	40.605
10000	22627.127	552.408	40.961