

实验二：全源最短路（APSP）

计04 何秉翔 2020010944

1. 实现方法

我们的实现方法基于实验文档里介绍的**基于分块思想的 Floyd-Warshall 算法优化方法**，我们采用的线程块大小为 32×32 ，按照实验文档的方法，我们将图邻接矩阵 D 分为大小为 64×64 的分块，即每个 32×32 的线程块处理一个 64×64 的邻接矩阵分块。

因此算法总共分为 $\lceil \frac{n}{64} \rceil$ 步，每步分为三阶段，在下面将一一介绍具体实现，第 p 步 ($0 \leq p < \lceil \frac{n}{64} \rceil$) 对应于原算法中 $pb \leq k < (p+1)b$ 的迭代。最外层循环遍历每一步，对于每一步的每一个阶段，我们分别开一个 GPU kernel 来实现对应阶段。kernel 的线程分配情况与存储访问情况（次数）如下：

kernel	grid	block	global	shared
kernel_1	(1, 1)	(32, 32)	4×2	$4 \times (1 + 64)$
kernel_2	$(\lceil \frac{n}{64} \rceil, 2)$	(32, 32)	4×3	$4 \times (2 + 64)$
kernel_3	$(\lceil \frac{n}{64} \rceil, \lceil \frac{n}{64} \rceil)$	(32, 32)	4×4	$4 \times (2 + 64)$

1.1 阶段一

在阶段一中，只有邻接矩阵的 $\lceil \frac{n}{64} \rceil$ 个 64×64 对角块需要被处理。而对于每一步，我们只需处理一个 64×64 对角块，因此对于阶段一的 kernel_1，我们设置 grid 规模为 (1, 1)，block 规模为 (32, 32)，共一个线程块，用于处理第 p 个 64×64 的对角块。

我们只使用一块 64×64 的 int 的共享内存，用于保存当前第 p 个 64×64 的对角块。第 (i, j) 个线程，其中 $0 \leq i, j < 32$ ，负责将对角块中 $(2i, 2j)$ 、 $(2i+1, 2j)$ 、 $(2i, 2j+1)$ 、 $(2i+1, 2j+1)$ 索引位置的 2×2 小方块载入到共享内存中，同时用四个寄存器保存这 2×2 小方块的值，在后续循环内访问时用到。另外在访问共享内存（包括 load 和 store）的过程中注意判断是否越界，越界的值赋值为 66666666（一个较大的值）。

kernel 内循环 64 次用于在中心块内部执行 Floyd-Warshall 算法，对于每一循环步 k ，我们在循环内对 2×2 小方块进行更新，在循环内访问共享内存时，我们注意到有重复访问的情况，比如小方块内同一行的行坐标相同，此时对于列坐标为 k 的访问是重复的，此时我们可以在循环内部开几个寄存器，先将会重复访问的共享内存元素放到寄存器内再进行更新。

对每个线程，存储访问情况如下：

1.2 阶段二

在阶段二中，对于每一步的一个 64×64 对角块，我们需要对相应的 $2 \times (\lceil \frac{n}{64} \rceil - 1)$ 个十字块进行更新。因此对于阶段二的 kernel_2，我们设置 grid 规模为 $(\lceil \frac{n}{64} \rceil, 2)$ ，block 规模为 (32, 32)，共 $2 \times \lceil \frac{n}{64} \rceil$ 个线程块，用于处理十字块，由于中心块内部的逻辑即阶段一，因此我们可以统一处理。

我们使用两块 64×64 的 int 共享内存，用于保存当前第 p 个 64×64 的对角块以及当前线程对应的十字块，载入共享内存的逻辑与阶段一一致，在 load 进十字块对应的共享内存时，我们用四个寄存器保存 2×2 小方块的值，在后续循环内访问时用到。另外在访问共享内存（包括 load 和 store）的过程中注意判断是否越界，越界的值赋值为 66666666（一个较大的值）。

我们根据 blockIdx.y 来区分用于处理行十字块还是列十字块，具体而言，我们约定 $\text{blockIdx.y} == 0$ 时当前线程块处理行十字块， $\text{blockIdx.y} == 1$ 时处理列十字块，注意索引的正确计算即可。

最后我们分别对行和列来实现阶段二的循环主逻辑，与阶段一一致，我们在循环内部也使用寄存器来保存需要多次访问的共享内存元素。

1.3 阶段三

再阶段三中，对于每一步的一个 64×64 对角块，我们需要对剩余的所有非十字块进行更新。因此对于阶段三的 `kernel_3`，我们设置 `grid` 规模为 $(\lceil \frac{n}{64} \rceil, \lceil \frac{n}{64} \rceil)$ ，`block` 规模为 $(32, 32)$ ，共 $\lceil \frac{n}{64} \rceil \times \lceil \frac{n}{64} \rceil$ 个线程块，用于处理剩余的非十字块，由于对于十字块处理的逻辑即阶段二，因此我们仍然可以统一处理。

我们使用两块 64×64 的 `int` 共享内存，对于当前线程对应的非十字块 (i, j) ，以及当前第 p 个 64×64 的对角块 (ip, jp) ，我们将 (ip, j) 以及 (i, jp) 这两个与对角块同行或同列的十字块载入共享内存中，另外，我们将当前线程对应的非十字块所要处理的 2×2 小方块载入四个寄存器。另外在访问共享内存（包括 `load` 和 `store`）或者载入寄存器的过程中注意判断是否越界，越界的值赋值为 66666666（一个较大的值）。

最后我们实现阶段三的循环主逻辑，与阶段一一致，我们在循环内部也使用寄存器来保存需要多次访问的共享内存元素。

2. 运行时间 & 加速比

在本部分我们汇报 `apsp` 在 $n = 1000, 2500, 5000, 7500, 10000$ 时的运行时间，及相对于助教提供的朴素实现 `apsp_ref` 的加速比。

n	<code>apsp</code>	<code>apsp_ref</code>	加速比
1000	1.191226 <i>ms</i>	14.805482 <i>ms</i>	12.43
2500	12.449682 <i>ms</i>	377.098447 <i>ms</i>	30.29
5000	76.350631 <i>ms</i>	2971.935243 <i>ms</i>	38.92
7500	261.490285 <i>ms</i>	10015.080836 <i>ms</i>	38.30
10000	599.162586 <i>ms</i>	22626.396011 <i>ms</i>	37.76