PA3 全源最短路（APSP）实验报告

1 GPU kernel 的实现

1.1 约定

所有的 kernel 函数的线程块规模都是 64×4，每个 kernel 处理一个 64×64 的矩阵，读入矩阵时每个线程读入 16个 int ，若超过数组的范围，则用 INF 填充。注意到全局内存到共享内存的拷贝是巨大的瓶颈，因此采取如下方法读入，从而实现对 cache 的尽可能的利用。

```
int i = threadIdx.x
int j = threadIdx.y;
for (int iter = 0; iter < 16; iter ++) {
  src[j][i] = src_core[IDX(src_start + j, i, BLOCK)];
  j += NUM;
}
```

令当前批量加入的节点为 [p*b,(p+1)*b) ，设 $A = [p*b,(p+1)*b) \cap \mathbb{N}$，将 $A \to A$ 称为 core -> core ，将 $\complement_U A \to A$ 称为 src -> core ，将 $A \to \complement_U A$ 称为 core -> dest ，将 $\complement_U A \to \complement_U A$ 称为 src -> dest 。

1.2 raw_floyd

第一阶段，完成 core -> core 的计算，先将矩阵从 graph[] 拷贝到位于共享内存的 core[64][65] 中，然后执行常规的 Floyd 算法，最后将结果拷贝回 graph[] 。注意由于 core 同时需要满足横竖两种访问模式，为了避免 bank conflict ，必须将数组最后一维设为 65。

特别的，将结果同时也拷贝到一个自定义的 core_core[64*64] 的数组中，若第二阶段如果从 graph[] 中读取，则相邻两次读取之间相差了 n*sizeof(int) ，局部性不好，而 core_core[] 数组则没有这个问题，因而能提高第二阶段的读入效率。

1.3 expand_row

第二阶段，完成 core -> dest 的计算，先将矩阵从 graph[] 和 core_core[] 拷贝到位于共享内存的 dest[64][65] 与 core[64][65] 中，然后执行算法，最后将结果同时拷贝到 graph[] 与一个 core_dest 数组上，这个 core_dest 与 core_core 的功能是相似的，实际测试中能减少第三阶段的 IO 耗时约 40ms。

1.4 expand_col

第二阶段，完成 src -> core 的计算，具体方式与 expand_row 相似，不再赘述。

1.5 expand_all

第三阶段，完成 src -> dest 的计算，先将数据从 src_core[],core_dest[] 拷贝到共享内存中，将数据从 graph[] 中直接读到寄存器上。

接下来，每个线程处理一个 4×4 的小块，这样，相比 1×16 的小块，从共享内存中读取的数据量从 $(1+16) \times 64$ 减小到 $(4+4) \times 64$。每次从共享内存中读取两个 4×4 的小块，将计算结果累积到 4×4 的小块上，最后再写回到 graph[] 中。这一共 3 个小块一共 48 个 int ，可以被完全放在寄存器上，避免了一般实现中常见的 bank conflict 问题，且显著地提升了计算的性能。

在这一阶段，每一轮从共享内存读取的数据量为 $(4+4) \times 4 = 32$，而运算次数为 $4 \times 4 \times 4 \times 2 = 128$，能较好地发挥全部的运算和访存能力。

1.6 一点心得

1. 编译器非常喜欢上下界固定的循环，因此尽量将循环的上下界写成固定的数字，方便编译器优化。
2. 为了让编译器将一个数组自动展开到寄存器上，访问数组时必须用常数、上下界固定的循环，一定要避免使用变量作为数组索引。
3. 访存优化是极其关键的瓶颈，一是从全局内存到共享内存，二是从共享内存到寄存器。

2 运行效率测试

n	naive(ms)	optimized(ms)	speedup
1000	14.9	3.64	4.09
2500	377.1	23.0	16.40
5000	2972.3	107.5	27.65
7500	10015.5	356.9	28.06
10000	22627.4	684.3	33.06