全源最短路大作业 实验报告

管思源 2021012702

性能优化方式

> 为了叙述和理解的方便，以下我将依次叙述每次迭代所作出的优化，以及相应的加速效果

首先，我按照实验文档的参考优化方法编写了代码。此时有三个 `kernel` ，分别对应一个阶段，其相关参数如下：

Kernel	Block	Grid	Shared Memory
`stage1`	32×32	1	4 KB
`stage2`	32×32	(m-1)×2	8 KB
`stage3`	32×32	(m-1)×(m-1)	12 KB

上表中 `m = ceil(n / 32)` 。

关于 `Shared Memory` ，我的用法是在每个kernel启动时从 `Global Memory` 中将对应的数据拷贝，计算时 `Block` 内共享，最后再拷贝回 `Global Memory` 。对于 `stage1` ，这个拷贝范围是中心块；对于 `stage2` ，这个拷贝范围是中心块相应的一个十字块；对于 `stage3` ，拷贝的是 `Block` 所负责的块、以及对应的两个十字块。

下面是这种实现的加速效果：

n	100	1000	2500	5000	7500	10000
朴素实现	0.275	15.436	377.849	2987.084	10051.674	22837.635
我的实现	0.103	4.564	64.727	468.741	1571.627	3712.620
加速比	2.670	3.382	5.837	6.373	6.396	6.151

接下来，我尝试一些常规优化，如将 `if` 语句合并、展开所有循环（后来发现这个优化应当是没有效果的，编译器似乎自动做了展开），以下是这些常规优化的加速效果：

n	100	1000	2500	5000	7500	10000
用时	0.133	5.254	60.114	457.706	1528.729	3606.880
相对前步加速比	0.774	0.869	1.077	1.024	1.028	1.029
相对朴素加速比	2.068	2.938	6.286	6.526	6.575	6.332

然后，根据文档提示，我发现在阶段三不会更新 `Shared Memory` 中的十字块数据，计算时实际不需要 `__syncthreads()` ，于是删去这一步，有了较大的性能提升：

n	100	1000	2500	5000	7500	10000
用时	0.111	3.197	36.036	241.38	800.905	1883.782
相对前步加速比	1.198	1.643	1.668	1.896	1.909	1.915
相对朴素加速比	2.477	4.828	10.485	12.375	12.550	12.123

随后，我又进行了一系列常规优化，包括更多地使用寄存器而非 `Shared Memory` 储存中间变量，将循环中每次分支结果不变的if语句提到循环外面等，效果如下：

n	100	1000	2500	5000	7500	10000
用时	0.064	2.385	30.124	205.887	685.204	1615.857
相对前步加速比	1.734	1.340	1.196	1.172	1.169	1.166
相对朴素加速比	4.297	6.472	12.543	14.508	14.670	14.133

我观察发现，在 `stage3` 实际不需要将负责块的数据存入 `Shared Memory` ，直接从 `Global Memory` 读取入寄存器即可，从而减少 `Shared Memory` 用量，加速效果如下：

n	100	1000	2500	5000	7500	10000
用时	0.064	2.225	27.292	189.095	629.174	1483.424
相对前步加速比	1.000	1.072	1.104	1.089	1.089	1.089
相对朴素加速比	4.297	6.938	13.845	15.797	15.976	15.395

之后又是常规优化。通过观察代码，我发现程序每次访问 `Shared Memory` 都要计算偏移量，其中静态的乘法和加法可以提出循环。这一优化的效果如下：

n	100	1000	2500	5000	7500	10000
用时	0.061	2.116	25.755	179.221	593.207	1397.727
相对前步加速比	1.049	1.052	1.060	1.055	1.061	1.061
相对朴素加速比	4.508	7.295	14.671	16.667	16.945	16.339

在美化代码的过程中，我意外地发现使用一个常量定义 `BLOCK_LEN = 32` 来代替 `blockDim.x` 可以取得显著的加速，效果如下：

n	100	1000	2500	5000	7500	10000
用时	0.051	1.778	19.347	150.525	490.784	1155.117
相对前步加速比	1.196	1.190	1.331	1.191	1.209	1.210
相对朴素加速比	5.392	8.682	19.530	19.844	20.481	19.771

我推测这可能是常量定义有利与编译器对相关循环做出优化、同时可以提前进行一些静态的计算。

最后，我通过 `Occupancy Calculator` 的计算结果，发现我的 `Thread num` 和 `Register num` 较多，是 `SM` 调度的瓶颈，而 `Shared Memory` 的用量较少，反而影响了性能。为了充分利用 `Shared Memory` ，我在 `stage3` 采取了每个 `Block` 负责多个32×32块的策略，以下是测试结果：

块数量	100	1000	2500	5000	7500	10000
1×1	0.051	1.581	19.307	149.748	490.719	1154.779
2×2	0.071	1.566	22.354	161.878	542.113	1180.886
4×4	0.097	1.603	14.833	112.775	373.727	873.888
6×6	0.123	2.013	17.853	108.956	348.857	816.500

可以观察到，不同的负责块数量的设定分别在不同的图规模下取到最优。一般而言，更大的块数量有利于处理更大的图规模，而块数量为1则退化为之前的情形，可以在更小的图规模中减少overhead。因此，我对 `n` 进行了条件判断，对不同的图规模采用不同的块数量设定，效果如下：

n	100	1000	2500	5000	7500	10000
用时	0.050	1.397	14.818	107.435	346.922	814.095
相对前步加速比	1.020	1.273	1.306	1.401	1.415	1.419
相对朴素加速比	5.500	11.049	25.499	27.804	28.974	28.053

同时， `kernel` 也从三个变为六个，以下是它们的信息：

Kernel	Block	Grid	Shared Memory	Register
`stage1`	32×32	1	4 KB	29
`stage2`	32×32	(m-1)×2	8 KB	29
`stage3_1`	32×32	(m-1)×(m-1)	8 KB	28
`stage3_2`	32×32	ceil[(m-1)/2]×ceil[(m-1)/2]	16 KB	53
`stage3_4`	32×32	ceil[(m-1)/4]×ceil[(m-1)/4]	32 KB	54
`stage3_6`	32×32	ceil[(m-1)/6]×ceil[(m-1)/6]	48 KB	54

最终测试结果

n	1000	2500	5000	7500	10000
朴素实现	15.436	377.849	2987.084	10051.674	22837.635
我的实现	1.397	14.818	107.435	346.922	814.095
加速比	11.049	25.499	27.804	28.974	28.053