Introduction to HPC
HW6 Report

姓名: 任一

学号:2018011423

ry18@mails.tsinghua.edu.cn

2020 年 5 月 15 日

实验环境	
集群 GPU 节点:	cn006-cn007
nvcc 版本:	Cuda compilation tools, release 9.0, V9.0.176

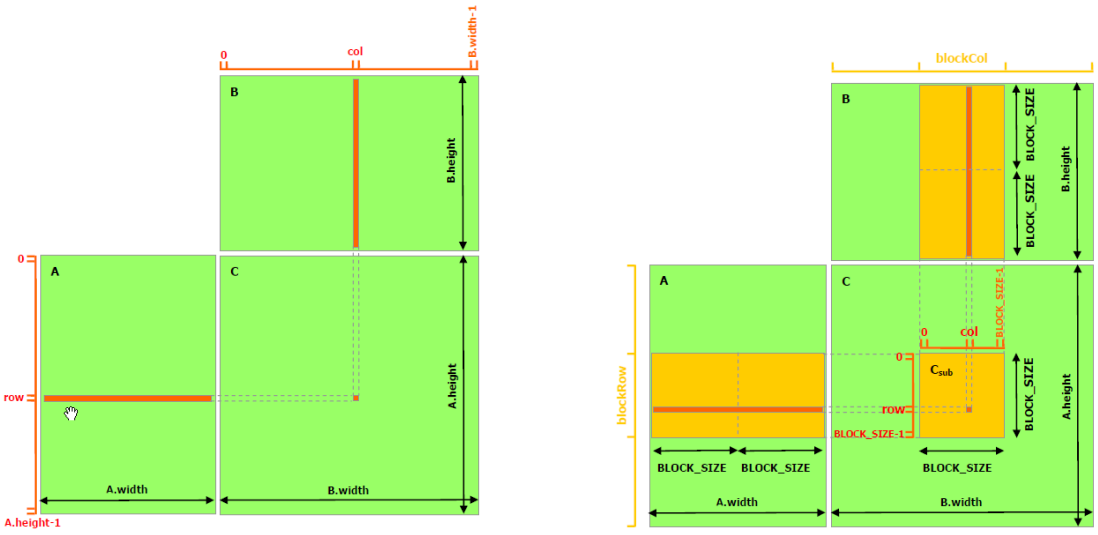
1 实验概述

本次作业中，我通过所学的 CUDA 编程知识，结合对 GPU 架构的理解，实现了优于 baseline 的 gemm 算法，即计算 $C = \alpha AB + \beta C$, 其中 A, B, C 的矩阵大小分别为 $M \times K, K \times N, M \times N$. 我做的优化也主要集中于 AB 的矩阵乘法运算。提交目录中的 gemm_2018011423.h 即为我实现的 gemm 算法头文件。

2 实验思路

本次实验中，我主要利用了 GPU 每个 block 中的 shared memory，来减少从 global memory 中读取数据，以加快矩阵乘法计算速度。

具体来说，我将每个 block 的形状设置为 32 * 32，对于每个待计算的 Block，令 Block 中的每个 Thread 都从当前 Block 计算所涉及的 A 矩阵和 B 矩阵中的某一部分，读取相应的一个元素到 A_shared 和 B_shared 这样的 shared memory 中的矩阵里面。由于 shared memory 大小有限，我固定 A_shared 和 B_shared 矩阵的大小均为 32 * 32. 这样就需要在一个 Block 的计算中，对涉及的 A 矩阵和 B 矩阵的某一部分进行循环遍历，每次遍历都需要更新 A_shared 和 B_shared 中的数据，以得到最终该 Block 内元素的计算结果。实现示意图如图 1(b) 所示[1]。



(a) baseline 的实现方法　　(b) 我的实现方法

图 1: baseline 实现方法与我的实现方法对比

[1]图源https://docs.nvidia.com/cuda/cuda-c-programming-guide/#shared-memory

3 实验中的改进思路

本部分中我将介绍在进行本实验的过程中，我的一些想法和改进思路。

3.1 使用动态分配的 **shared memory**，难以突破 **shared memory** 大小的瓶颈

当我对整个实验框架和 baseline 的实现思路进行学习后，我认为 baseline 很慢的很大一部分原因在于，没有使用 shared memory，每个线程只是从 global memory 当中读取数据，这样就没有很好地利用 GPU 的体系结构。

一个最直观的改进想法是，将实验中用到的矩阵 A, B, C 都存在 shared memory 中. 但经过资料的查找，我发现实验平台上 GPU 的 shared memory 大小为 48KB [2]，这样小的存储空间难以存下 A, B, C 这 3 个矩阵，同时也不利于任意大小矩阵的计算。

为了减少对 shared memory 大小的占用，我发现每个 Block 在计算的过程中，只需要 A, B 矩阵的一部分内容，即可完成 Block 内的元素计算。举例来说，使用框架最初默认的 32×8 的 Block 大小，$M = 300, N = 400, K = 500$. 在这样的参数下，每个 Block 所需要矩阵 A 中 $8 \times K = 4000$ 个元素，需要矩阵 B 中 $32 \times K = 16000$ 个元素，总计需要 20000 个元素。每个 double 型变量大小为 8Byte, 因此该 Block 需要的元素所占的空间为 160KB, 这比 shared memory 的 48KB 要大, 因此不可行。此外这个数字还依赖于 K 的大小，也不利于任意大小矩阵的计算。

3.2 固定所需的 **shared memory** 大小，效果较好

为了解决该问题中，shared memory 大小有限带来的瓶颈，我上网查阅了一些资料，在 Nvidia 官方文档中找到了较为合适的解决方案 [3]，并参考官方的解决方法，顺利完成了该实验，并得到了优于 baseline 的较好的结果。

具体来说，shared memory 一次难以存下每个 Block 所需的矩阵元素，一个解决方法就是固定每次 shared memory 存储的元素个数，分多次将所需的矩阵元素放入 shared memory 中进行计算。示意图如图 1(b) 所示。使用这个方法可以充分利用 GPU 中的 shared memory, 同时也巧妙地解决了 shared memory 大小有限的问题，并且该解决方案也不依赖于输入矩阵的大小，具有很强的灵活性。不过该算法对于 M, N, K 不被 Block 大小整除的情况，需要一些特殊处理，经过一定的调试，我也完成了对该情况的处理，从而能够处理任意大小矩阵的乘法运算，

[2] 资料来源于 https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

[3] 参考了 Nvidia 文档 https://docs.nvidia.com/cuda/cuda-c-programming-guide/#shared-memory

为了进一步优化该算法，我试图调整每个 Block 的大小，从 16×16 调整到了 32×32. 在 $M = 978, N = 782, K = 633$ 的 double 型运算中，得到的实验结果如下表：从表中可以看出，我实现的算

表 1: 使用不同大小的 BlockSize 对性能的影响

	time(s)	GFLOPS	SpeedUp Ratio
BlockSize=16	0.005441	177.94	2.50
BlockSize=32	0.004417	219.20	3.09
Cublas	0.001247	776.36	10.95
Baseline	0.013540	70.91	1.00

法性能较 baseline 有很大的提升，此外 BlockSize 较大时性能相较 BlockSize 较小时的性能更好。这让我思考上面两个现象的原因。我从对 global memory 读取次数角度，来看我的算法相较 baseline 算法的提升之处，以及 BlockSize 对算法性能的影响。

为了分析的简便起见，我们在分析时认为 M, N, K 都可以被 BlockSize 整除，且仅考虑矩阵乘法的部分 (即 $C = AB$ 两个矩阵的乘法). 对于 baseline 算法来说，C 中每个元素的计算，都需要从 global memory 中读取所需的元素。C 中每个元素的计算需要 A, B 两个矩阵中的 $2 \times K$ 个元素，C 中共有 $M \times N$ 个元素. 因此使用 baseline 算法，共需从 global memory 中读取 $2MNK$ 个元素.

对于我的算法来说，C 中每个 Block 需要从 global memory 中读取 $2 \times BlockSize \times K$ 个元素，C 中共有 $\frac{MN}{BlockSize^2}$ 个 Block. 因此使用我的方法需要从 global memory 中读取 $\frac{2MNK}{BlockSize}$ 个元素，需要从 global memory 中读取的元素少于 Baseline 算法。

由上面的分析，相较 baseline 算法，我的算法可以显著降低从 global memory 读取元素的次数，且从 global memory 中读取元素的个数与 BlockSize 成反比。这样就可以很好地解释在表 1 的实验中的两个实验现象，即我实现的算法性能较 baseline 有很大的提升，以及 BlockSize 较大时性能相较 BlockSize 较小时的性能更好。

4 实验结果分析

本部分中，我将通过不同尺寸的矩阵，对我的算法进行充分的性能测试，并与 Cublas 和 Baseline 算法进行对比分析。在表格和图中，MyGemm-16 和 MyGemm-32 分别表示 BlockSize 为 16 和 32 的我的算法。为了展示清晰，图中的横坐标均为以 2 为底的对数坐标。在如下所有测试中，我的算法结果均与正确结果保持一致。

4.1 固定 K, 改变 M, N

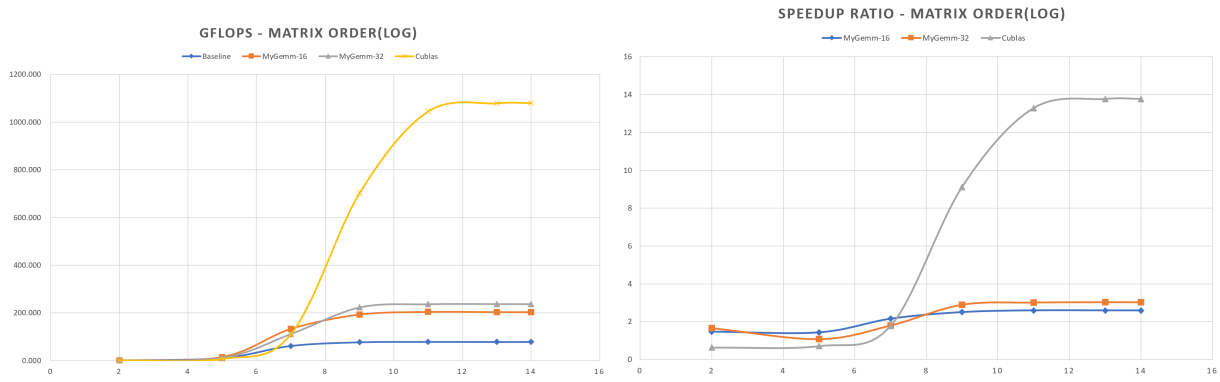
在本部分测试时，为了方便起见，我令 $K = 1024, M = N = MatrixOrder$. 得到性能分析图表如下：

表 2: GFLOPS-矩阵阶数表

MatrixOrder	4	32	128	512	2048	8192	16384
Baseline	0.172	9.982	61.494	77.081	78.642	78.363	78.397
MyGemm-16	0.253	14.306	132.436	193.081	204.026	203.275	203.108
MyGemm-32	0.286	10.821	110.799	223.122	236.552	237.245	237.023
Cublas	0.109	7.081	109.683	702.897	1044.848	1078.783	1079.668

表 3: 加速比-矩阵阶数表

MatrixOrder	4	32	128	512	2048	8192	16384
MyGemm-16	1.469	1.433	2.153	2.504	2.594	2.594	2.591
MyGemm-32	1.660	1.084	1.802	2.895	3.008	3.028	3.023
Cublas	0.631	0.709	1.784	9.119	13.286	13.767	13.772



(a) GFLOPS 与矩阵阶数折线图　　(b) 加速比与矩阵阶数折线图

图 2: 固定 K, 改变 M, N 时的性能分析

从图 2 中可以看出，随着矩阵阶数的增加，GFLOPS 和加速比都在增加。当矩阵阶数达到 2048 及更大的时候，GFLOPS 和加速比在各算法中的变化都趋于平缓。这样的现象可以解释为，当矩阵阶数增加时，并行计算的效率逐渐显现出来，当矩阵阶数达到一定值之后，并行计算的效率就逐渐趋于饱和，不再有很大的增长。

此外，对比各算法，当矩阵阶数很大的时候 (例如 512 及以上)，Cublas 能够表现出很好的性能，其次是 MyGemm-32 和 MyGemm-16, 这也佐证了 3.2 中的分析，即 BlockSize 较大时，性能会更好一些。不过在矩阵阶数很小的时候 (例如 32 及一下)，Cublas 的性能甚至比 Baseline 算法还要慢一些，MyGemm-16 和 MyGemm-32 算法略优于 Baseline 算法。这可以理解为，Cublas 对大型矩阵计算做了很多较为复杂的优化，但当矩阵规模很小时，这些优化的开销相较优化带来的收益更大，因此矩阵规模小时 Cublas 的性能较低。而我实现的 MyGemm 算法没有过于复杂的优化技术，因此对于较小规模的矩阵运算也能表现出一定的优势。

4.2 固定 M, N, 改变 K

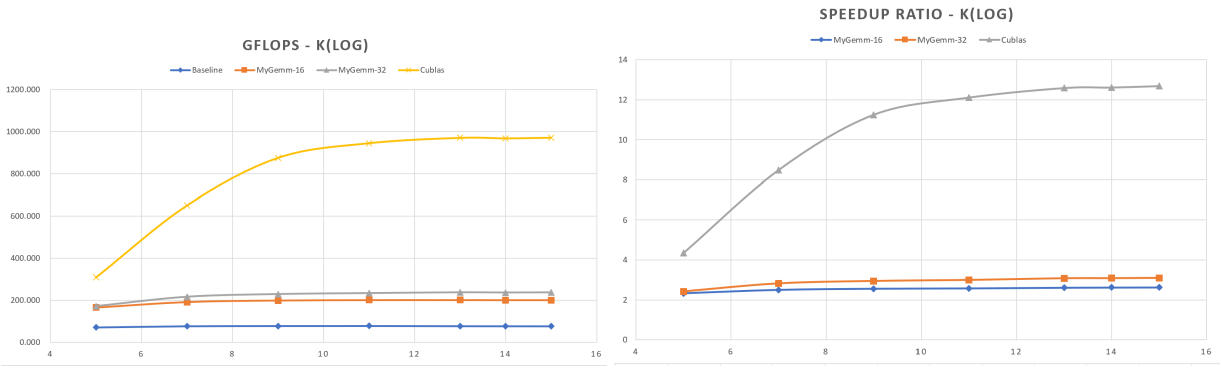
本部分测试中，我固定 $M = N = 1024$, 改变 K, 得到性能分析图表如下：

表 4: GFLOPS-K 数据表

K	32	128	512	2048	8192	16384	32768
Baseline	71.131	76.723	77.846	78.042	77.098	76.750	76.556
MyGemm-16	165.632	192.100	199.415	201.285	201.376	201.115	201.118
MyGemm-32	172.557	216.842	229.456	234.087	237.662	236.905	237.452
Cublas	309.196	651.014	876.057	945.675	970.994	968.644	971.713

表 5: 加速比-K 数据表

K	32	128	512	2048	8192	16384	32768
MyGemm-16	2.329	2.504	2.562	2.579	2.612	2.620	2.627
MyGemm-32	2.426	2.826	2.948	3.000	3.083	3.087	3.102
Cublas	4.347	8.485	11.254	12.118	12.594	12.621	12.693



(a) GFLOPS 与 K 折线图　　(b) 加速比与 K 折线图

图 3: 固定 M, N, 改变 K 时的性能分析

从图 3 中可以得到和 4.1 中类似的结论，即当 K 增加时，各个算法的 GFLOPS 和加速比都有一定的提升，在矩阵规模很大时，这样的提升就趋于平缓。此外，Cublas 展现出了优异的性能，我实现的 MyGemm 算法相较 Baseline 算法也有较大的提升，并且 BlockSize 大的性能会更好一些。这些现象和结论与 4.1 中的非常相似，由于我在 4.1 中已经进行了详细的分析，在此就不再赘述了。

5 总结

在本次作业中，结合对 GPU 体系结构的理解，我尝试了 CUDA 编程。这让我加深了对于 GPU 的理解，也体验到了 GPU 编程与 CPU 编程的显著差异，从中也收获了很多，感谢老师和助教的悉心指教！