南開大學
Nankai University

计算机学院

并行程序实验报告

GPU 实验

姓名：唐明昊

学号：2113927

专业：计算机科学与技术

2023 年 6 月 27 日

目录

1 问题描述

1.1 问题引入

矩阵乘法是线性代数中的一个重要运算，两个矩阵相乘得到一个新的矩阵。矩阵乘法在许多领域都有广泛的应用，尤其在科学计算、工程领域和计算机图形学等方面发挥着重要作用。它被用于解决线性方程组、图像处理、数据压缩、神经网络和物理模拟等问题。

为了提高矩阵乘法的计算效率和性能，尤其是针对大规模矩阵的乘法运算，考虑采用并行化手段。

一般情况下，矩阵乘法可以通过 **GPU 进行任务划分**，由一个线程计算结果矩阵一个位置的元素，从而实现并行化。

为了优化计算性能，常常需要进行**子矩阵划分**：将一个大的矩阵划分成更小的子矩阵，对子矩阵进行并行计算。例如，将矩阵 A 划分成大小为 m × k 的子矩阵，将矩阵 B 划分成大小为 k × p 的子矩阵，然后使用并行计算的方式对这些子矩阵进行乘法运算。最后，将子矩阵的结果合并得到最终的矩阵乘积。

需要注意的是，在并行化矩阵乘法时还需要考虑一些问题：

负载均衡：确保每个处理单元或线程的工作负载均衡。要求将子矩阵划分均匀，每个处理单元耗时基本一致，保证整体性能。

数据依赖：合理安排矩阵乘法，保证结果正确性以及并行执行效率。

1.2 实验数据

实验中采用的数据由 **random_float()**（rand() / double(RAND_MAX)）随机生成。通过循环遍历的方式，随机生成浮点数赋值给数组 A 和 B 的各个元素。

实验数据生成

```
// init the A/B/C
for (int i = 0; i < M * K; i++) {// 随机生成矩阵A
    A[i] = random_float();
}
for (int i = 0; i < K * N; i++) {// 随机生成矩阵B
    B[i] = random_float();
}
for (int i = 0; i < M * N; i++) {// 初始化矩阵C
    C[i] = 0.0f;
    C_host[i] = 0.0f;
}
```

矩阵 A 的大小为 M × K，矩阵 B 的大小为 K × N。使用一维数组保存矩阵元素。

循环从 0 到 m * k - 1 进行迭代，依次访问 A 的每一个元素。在循环体内部，调用 random_float() 函数生成一个随机的浮点数，并将其赋值给 A[i]。矩阵 B 同理。矩阵 C 均初始化为 0.0f。

代码运行过程生成的矩阵 A、B 的规模均为 512*512。

1.3 实验环境

实验采用 DevCloud 云端环境，测试使用 oneAPI 进行 GPU 优化的加速效果。

1.4 实验概述

本实验从矩阵乘法的平凡算法出发，逐步探究使用 oneAPI 对普通算法进行 GPU 并行化、矩阵分块优化所带来的性能提升。

探究了任务不同划分粒度对性能的影响及其原因（问题三）。思考并测试输入寄存器的规模对于 GPU 资源利用率、算法性能的影响（问题四）。最后，对整个实验进行总结，指出在期末报告中运用 oneAPI 的可能性。

2 代码解释

2.1 oneAPI 并行化

首先将 CPU 单线程矩阵乘法并行化。使用 oneAPI 进行任务划分，将矩阵 C 划分为 M*N 个任务，每个线程读取一行以及一列，独立计算累计乘积，最后将结果放回矩阵。

矩阵乘法并行化

```
// 任务划分
auto grid_rows = (M + block_size - 1) / block_size * block_size;
auto grid_cols = (N + block_size - 1) / block_size * block_size;
auto local_ndrange  = range<2>(block_size, block_size);
auto global_ndrange = range<2>(grid_rows, grid_cols);
// GPU并行计算
auto e = q.submit([&](sycl::handler &h) {
  h.parallel_for<class k_name_t>(
      sycl::nd_range<2>(global_ndrange, local_ndrange),[=](sycl::nd_item<2> index) {
          int row = index.get_global_id(0);
          int col = index.get_global_id(1);
          float sum = 0.0f;
          for (int i = 0; i < K; i++) {
            sum += A[row * K + i] * B[i * N  + col];
          }
          C[row * N + col] = sum;
      });
```

2.2 分块优化

2.2.1 算法描述

为了加速矩阵乘法，采用分块进行算法优化。

分块算法的原理是**对行与列进行复用**，因为在矩阵相乘的过程中，矩阵 A 中的一行将与矩阵 B 中的 n 个列都相乘分别累加，然后得到 C 中的 n 个元素；对称的，矩阵 B 中的一列将与矩阵 A 中的 m 个行都相乘。由此可见，A 中的一行在矩阵乘法的过程中将被利用 n 次，而 B 中的一列将在乘法

过程中被利用 m 次。平凡的循环迭代算法里，每个子任务只是读出 A 的一行和 B 的一列，只使用一次，利用效率较低。

由以上的分析，可以得到矩阵乘法任务划分的新思路：**矩阵分块任务划分**。每个子任务中不在只包含结果矩阵 C 中的 1 个位置，而是包含 C 中的 tileX*tileY 个位置。对于 A 中的一行，会进行 tileY 次利用；相应的，对于 B 中的一列会进行 tileX 次利用。

2.2.2 代码实现

核心计算部分，使用每个工作线程的私有变量来保存计算的中间结果，即相当于利用 **Input Registers** 的方法，避免反复访问全局变量，从而减少访问同步开销。而输入寄存器的值，在计算过程中会被反复利用，以提高算法效率。

分块任务划分

```
// 任务划分
auto grid_rows = M / tileY;
auto grid_cols = N / tileX;
auto local_ndrange = range<2>(BLOCK, BLOCK);
auto global_ndrange = range<2>(grid_rows, grid_cols);
// GPU并行计算
auto e = q.submit([&](sycl::handler& h) {
    h.parallel_for<class k_name_t>(
        sycl::nd_range<2>(global_ndrange, local_ndrange), [=](sycl::nd_item<2> index)
            {
            int row = tileY * index.get_global_id(0);// 每个子矩阵有tileY行
            int col = tileX * index.get_global_id(1);// 每个子矩阵有tileX列
            .......
            // core computation
            for (int k = 0; k < N; k++) {
                // 读入输入寄存器subA, subB
                for (int m = 0; m < tileY; m++)
                    subA[m] = A[(row + m) * K + k];
                for (int p = 0; p < tileX; p++)
                    subB[p] = B[k * N + p + col];
                // 计算中间结果
                for (int m = 0; m < tileY; m++)
                    for (int p = 0; p < tileX; p++)
                        sum[m][p] += subA[m] * subB[p];
            }
            // 计算结果并写回矩阵
            for (int m = 0; m < tileY; m++)
                for (int p = 0; p < tileX; p++)
                  C[(row + m) * N + col + p] = sum[m][p];
        });
    });
e.wait();
```

通过上面的调整，仅需要在计算之前从全局内存中读取 (tileY+tileX)*N 次数据，最后再使用 tileX*tileY 次操作写回全局内存。

如果不利用输入寄存器策略，只是进行三**重循环的一般策略**，每次从全局读取数据并计算，则计算过程的代码如下：

一般策略

```
// 计算结果并写回矩阵
for (int m = 0; m < tileY; m++)
    for (int p = 0; p < tileX; p++)
        for(int k = 0; k < N; k++)
            C[(row + m) * N + col + p] += A[(row + m) * K + k] * B[k * N + p +
                col];
```

可以分析，该策略将对全局数据矩阵 A、B 读取 2*tileX*tileY*N 次，效率大幅落后于使用 Input Registers 策略。除了读取次数上的落后，还可以观察到，对于 B 的读取并不是顺序的，而是每次间隔 N 个元素读取。该方法空间局部性差于 Input Register 策略，会造成一定的性能损失。

当然，由于分块的优势，可以对该代码进行简单的改动，以重复利用 A 中一行元素 tileY 次，进而优化读取次数。但是对 C 的写入操作却无法优化，次数仍然远大于输入寄存器策略。

以上即是代码采用输入寄存器策略的原因。

3 问题探究

3.1 tileX、tileY 对性能的影响

改变算法中 tileX、tileY 的大小，探究其对性能的影响，得到实验结果如图3.1所示：

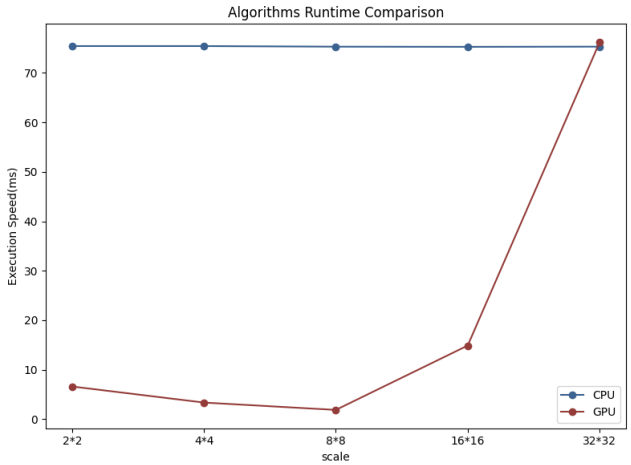

图 3.1: 并行算法不同规模执行时间以及与串行对比（单位：ms）

由图可知，tileX、tileY 的规模增大初期，GPU 并行化算法执行速度随之逐渐增大。直到规模达到 8*8 拐点出现，速度逐渐下降。继续观察图中数据可得，进一步增大规模，当 tileX*tileY 增长到 32*32 时，GPU 算法的性能已经低于 CPU 算法。

tileX、tileY 决定了任务划分粒度的大小：tileX、tileY 越大，任务划分的粒度越粗，每个工作线程得到的任务越多；反之任务越少。而经查阅资料得，任务量的大小将会影响开辟线程数的多少：tileX、tileY 越大，子任务的任务量更大，故 GPU 将开辟较少的线程；相应的，如果一个子任务的任务量较小，GPU 将开辟较多的线程。

据此分析，造成图3.1现象的原因可能是：

- tileX、tileY 过小时，每个线程所分配的任务量较小，由此**需要开辟较多的线程**。而由于线程的创建、销毁和切换都需要一定的时间和资源。另外，**线程的调度**也造成额外的开销：GPU **硬件资源限制**，所以线程共享 GPU 上面有限的硬件资源，如寄存器、共享内存、总线等；当线程数量较多时，会出现资源竞争而被迫等待的情况，影响执行效率。
- tileX、tileY 过大时，每个线程都将被分配大量工作，且总线程数减少。**少数线程负载大量任务**，而其他非工作线程则处于空闲状态，GPU 利用率过低，是对 GPU 计算资源的浪费。

3.2 输入寄存器大小对性能影响

3.2.1 问题描述

2.2.1节中分析了在核心计算部分，使用 Input Registers 策略能够有效的减少从全局内存中读取数据的次数，从而提高运行效率。

而在2.2.1节中所使用的输入寄存器总数为 suB 的 tileX + subA 的 tileY 个。思考，如果 GPU 还有剩余的存储资源未被利用，能否通过提高使用的输入寄存器个数，进而充分利用剩余的存储资源，从而取得更好的性能。

3.2.2 实验测试

将原有代码进行改动，参考循环展开的思想，每次循环读入的数据量提升至 length* (tileX+tileY) 个。

改动输入寄存器

```
for (int k = 0; k < N/length; k++) {
    // 读入输入寄存器subA[tileY][length], subB[tileX][length]
    for (int m = 0; m < tileY; m++)
        for (int f = 0; f < length; f++)
            subA[m][l] = A[(row + m) * K + k * length + f];
    for (int f = 0; f <length; f++)
        for (int p = 0; p < tileX; p++)
            subB[p][l] = B[(k * length + f) * N + p + col];
    // 计算中间结果sum
    for (int f = 0; f<length; f++)
        for (int m = 0; m < tileY; m++)
            for (int p = 0; p < tileX; p++)
                sum[m][p] += subA[m][f] * subB[p][f];
}
// write results back
```

3.2.3 结果分析

当 tileX，tileY 均为 2 时，改变 length 的大小即可调整所使用的输入寄存器个数，进而得到使用不同数量输入寄存器的情况下，算法运行效率如图3.2所示。(64+64 表示 subA 的规模为 tileY*length = 64，此时 subB 的规模也为 64。2+2 即表示为不进行改动，subA 和 subB 规模均为 2)

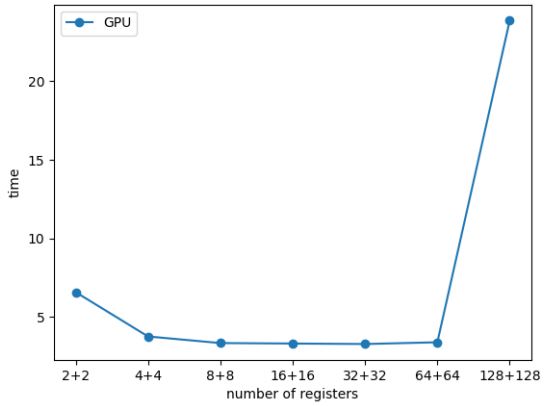

图 3.2: 不同输入寄存器个数下性能（单位：ms）

由图可知，随着使用的输入寄存器的个数逐渐增多，算法的运行速度先不断提升，达到拐点后，速度急剧降低。在 128+128 的规模下，算法的速度是不改变大小的约 1/4。

- 输入寄存器增多，充分利用了 GPU 的剩余资源，一次循环能够**读入更多的数据**。另外引入更多存储寄存器后，初始化 subA 能够连续读取多行，在一定程度上提高了程序的**空间局部性**。由图也可以看出，寄存器个数从 2+2 到 4+4 性能有明显提升。
- 继续增大使用的输入存储器的数量，GPU 资源已逐渐被消耗殆尽，但由于空间局部性平衡资源资源消耗的影响，性能仍然有略微提升。
- 输入寄存器规模从 64+64 提升到 128+128 后，**资源完全被消耗空**，过度利用存储器对算法造成巨大的性能损耗。

4 练习总结

在本次 oneAPI 编程练习中，使用分块矩阵的方法对矩阵相乘问题进行了并行化优化，并探究了两个问题：划分块大小对算法性能的影响；输入寄存器的数量多少对算法性能的影响。

通过实验测试具体的数值与查阅相关资料，探究分块优化与读入寄存器引起性能差异的具体原因。两个问题最终都指向了 GPU 资源的合理利用。

通过本次实验，我熟练掌握了 oneAPI 的运用，熟悉了 oneAPI 编程操作，任务划分方法等相关操作。深刻体悟到了 GPU 并行化的特点，以及对 GPU 资源使用应该注意的问题。在**期末报告**撰写中，我也会尝试引入 oneAPI，对自主选题进行并行优化，挖掘其更多价值。

5 线上学习记录

于 5 月 17 日与 5 月 23 日两次参加英特尔 oneAPI 校园黑客松系列培训，学习了“用 Intel@oneAPI 基础工具套件中的 C++/SYCL 直接编程语言实现异构编程” 和 “基于 Intel@oneAPI 的应用性能分析工具、优化方法”。过程中掌握了 oneAPI 的基本使用，了解了 oneAPI 工具套件的强大功能，为本次编程练习打下了坚实基础。

图 5.3: 云端课程学习截图

6 自主选题介绍

6.1 选题问题描述

倒排索引（inverted index），又名反向索引、置入文档等，多使用在全文搜索下，是一种通过映射来表示某个单词在一个文档或者一组文档中的存储位置的索引方法。在各种文档检索系统中，它是最常用的数据结构之一。

对于一个有 U 个网页或文档（Document）的数据集，若想将其整理成一个可索引的数据集，则可以为数据集中的每篇文档选取一个文档编号（DocID），使其范围在 [1,U] 中。其中的每一篇文档，都可以看做是一组词（Term）的序列。则对于文档中出现的任意一个词，都会有一个对应的文档序列集合，该集合通常按文档编号升序排列为一个升序列表，即称为倒排列表（Posting List）。所有词项的倒排列表组合起来就构成了整个数据集的倒排索引。

倒排列表求交（List Intersection）也称表求交或者集合求交，当用户提交了一个 k 个词的查询，查询词分别是 t_1，t_2，...，t_k，表求交算法返回 $\cap_{1\leq i\leq k} l\left(t_i\right)$。

首先，求交会按照倒排列表的长度对列表进行升序排序，使得：

$$|l(t_1)| \leq |l\left(t_2\right)| \leq \ldots \leq |l\left(t_k\right)|$$

例如查询 “2014 NBA Final”，搜索引擎首先在索引中找到 “2014”，“NBA”，“Final” 对应的倒排列表，并按照列表长度进行排序：

$$l\left(2014\right) = (13,\ 16,\ 17,\ 40,\ 50)$$

$$l\left(NBA\right) = (4,\ 8,\ 11, 13, 14,\ 16, 17, 39, 40, 42, 50)$$

$$l\left(Final\right) = (1,\ 2,\ 3,\ 5,\ 9,\ 10,\ 13,\ 16,\ 18,\ 20,\ 40,\ 50)$$

求交操作返回三个倒排列表的公共元素，即：

$$l(2014) \cap l(NBA) \cap l(Final) = (13,\ 16,\ 40,\ 50)$$

链表求交有两种方式：按表求交（SVS 算法）和按元素求交（ADP 算法）。另外，我们还找到拉链法、跳表法、Bitmap 位图法、Hash 优化法、递归法等方法。

6.2 数据集描述

给定的数据集是一个截取自 GOV2 数据集的子集，格式如下：

1) ExpIndex 是二进制倒排索引文件，所有数据均为四字节无符号整数（小端）。格式为：[数组 1] 长度，[数组 1]，[数组 2] 长度，[数组 2]....

2) ExpQuery 是文本文件，文件内每一行为一条查询记录；行中的每个数字对应索引文件的数组下标（term 编号）。

通过对数据集分析得到，ExpIndex 中每个倒排链表平均长度为 19899.4，数据最大值为 25205174，这个数据在**构建 hash 表和 BitMap 位图**时，对参数的确定至关重要。查询数据集 ExpQuery 一共 1000 条查询，单次查询最多输入 5 个列表，可以据此设计测试函数。

另外，在给定的数据集以外，我们还自行设置了小数据集（见 GitHub 仓库）：每个倒排链表的长度在 50 至 100，生成 1000 个倒排链表，总计 300 次查询，每次查询 5 个单词。该数据集用于验证算法的正确性以及测试不同算法在小规模输入下的性能表现。

6.3 实验环境

操作系统版本：Windows 11 家庭中文版 64-bit (10.0, Build 22621)
系统类型：64 位操作系统, 基于 x64 的处理器
处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz (8 CPUs)，3.1GHz
各级 cache 值：L1cache 320KB，L2cache 5MB，L3cache 8MB
内存容量：16384MB RAM
编译器版本：gcc (x86/64-posix-sjlj-rev0, Built by MinGW-W64 project) 8.1.0
GPU 名称：NVIDIA GeForce MX450
GPU 总内存：10011MB （显示内存 1928MB，共享内存 8083MB）

6.4 实验分工

本小组由唐明昊和朱世豪组成。本次 GPU 并行实验唐明昊同学负责 SVS 和 Hash 算法的并行化，朱世豪同学负责 ADP 算法的并行化。唐明昊同学提出粗粒度实现及其优化方法，朱世豪同学结合 SIMD 的思想提出细粒度实现。每个算法均对两种思想进行了实现。附唐明昊和朱世豪二人 GitHub 仓库链接。

6.5 实验概述

本次 GPU 实验分别从**粗粒度**和**细粒度**两个角度入手。

- 粗粒度算法对求交链表分段划分，GPU 的每个线程分到很少一部分的 DocId，再运用相应的 SVS 或 ADP 等算法与其他链表求交。

该方法的好处是任务划分简单，host 端与 device 端通信较少，数据转移成本小。

缺点是 SVS 和 ADP 串行算法有**提前结束**的特性。而在分段以后，位于后面段的元素失去了前面段的信息，需要从头往后寻找匹配，容易造成工作量不均等。

为了解决该问题，我们尝试通过精确化搜索和块间通信进行优化。

- 细粒度算法结合 **SIMD** 的思想，把 GPU 当作一个巨大的向量，一次可以进行比较若干个元素。

  该方法的好处是直接在算法上运用 SIMD 的思想，不需要再考虑算法提前结束特性的影响。

  缺点是 host 端与 device 端通信较多，需要反复装载数据到 GPU，通信成本高。

  为此，我们尝试了使用统一虚拟内存等方法。

7 算法实现

7.1 预处理倒排索引

由于之前的倒排索引均是使用 vector 存储的，为了使用 cuda 核函数，需要**将数据预处理转移到数组中去**。而用 cudaMalloc 二维数组较为复杂，决定采用一维数组表示二维数组，数据进行整体传输。

数据预处理

```
int* lengthArr = new int[num+1], int* gpuLengthArr;
ret = cudaMalloc((void**)&gpuLengthArr, (num+1) * sizeof(int));
if (ret != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed: %s\n", cudaGetErrorString(ret));
}
// 获取各段长度
lengthArr[0] = num;// 用0号位置来保存num
for (int j = 0; j < num; j++)
    ......
index = new int[totalLength];// 开辟一维数组
ret = cudaMalloc((void**)&gpuIndex, totalLength * sizeof(int));
if (ret != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed: %s\n", cudaGetErrorString(ret));
}
for (int j = 0; j < num; j++) {//
    倒排链表全部放进一个二维数组->一维数组表示二维数组
    // 复制到数组当中去
    copy((*invertedLists)[query[i][j]].docIdList.begin(),
        (*invertedLists)[query[i][j]].docIdList.end(), index + totalLength);
    totalLength += lengthArr[j+1];// 当前位置
}
// 传递长度数组到GPU端
ret = cudaMemcpy(gpuLengthArr, lengthArr, (num + 1) * sizeof(int),
    cudaMemcpyHostToDevice);
// 复制数据到GPU端
ret = cudaMemcpy(gpuIndex, index, totalLength * sizeof(int),
    cudaMemcpyHostToDevice);
```

为了将若干个链表存入一个一维数组，首先需要记录每个链表的长度，并使用链表的总长度来开辟数组空间。特别的，使用长度数组 0 号位置来记录链表总数 num。

获取到链表长度后，首先使用 copy 函数将 vector 数据传入到 host 端的 index 数组里。使用长度数组做为偏移量，即可将数据复制到对应位置。最后调用 cudaMemcpy 将 host 端数据转移到 device 端的 gpuIndex 数组里。

7.2 粗粒度实现

7.2.1 基本实现

粗粒度实现是使用类似于 pthread 和 openMP 并行化的方法，将 GPU 视为一个若干线程集合，每个线程取出当前链表的一部分（length/totalThreads）与当前 query 下的其他几个链表进行求交，求交结果放入数组 index 里。每个线程只会放入他所管理的那一段，所以并不会造成冲突的问题。

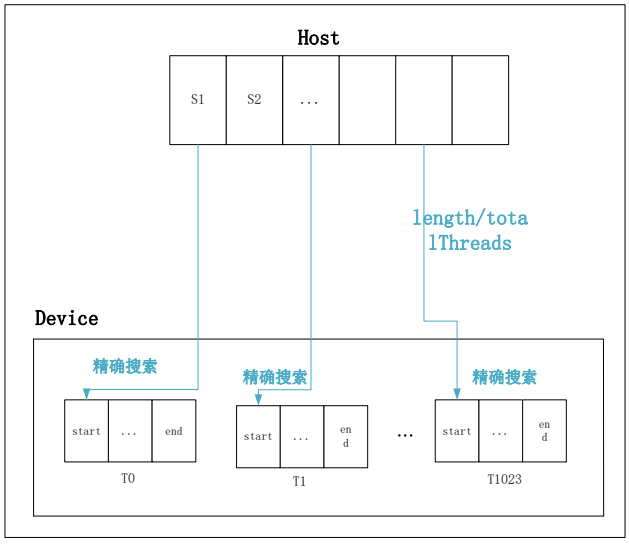

图 7.4: 粗粒度算法示意图

算法首先获取当前线程的索引，再计算总的线程数，据此计算出该线程所负责计算的链表部分。而后算法核心计算部分，每个线程在小范围内与其他链表进行求交。(算法计算核心逻辑前面实验已介绍，不再赘述)

粗粒度实现

```
__global__ void ADP_kernel(int* index,int* lengthArr) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;//计算线程索引
  int totalThreads = gridDim.x * blockDim.x;
  int num = lengthArr[0];
  // 线程起始，结束
  int start = lengthArr[1] / totalThreads * tid, end = min(lengthArr[1] /
      totalThreads * (tid + 1), lengthArr[1]);// 第一个链表拿来做划分
    ......
  for (int i = start; i < end; i++) {
    bool isFind = true;
    unsigned int e = index[i];
    for (int s = 1; s != num && isFind == true; s++) {
```

```
      isFind = false;
      while (list[s].cursor < list[s].end) {// 检查s列表
        .......
      }// 下一个链表
    }
    // 当前元素已被访问过
    if (isFind)
      index[position++] = e;
  }
}
```

7.2.2 精确化搜索

由于 SVS 和 ADP 串行算法为了提高表现，引入了拉链法，即链表前面的元素会帮助后面的元素移动求交链表中的指针，算法具有提前结束的特性。而在并行化分段以后，位于后面段的元素失去了前面段的信息，需要从头往后寻找匹配。又因为链表都是升序排列的，故在本链表中位于后半部分的元素，在另一链表中大概率也会位于后面。这将导致部分线程遍历链表的长度很大，也即工作量不均等，最终性能低下。

可以通过预处理的方式，在算法核心计算部分之前，通过二分查找的方式获取 1 到 num-1 号倒排链表中第一个大于等于该段起始 docId 的位置。该位置即为该线程的求交开始位置。

而对于求交的结束位置，对于第 i 号线程，由于它的求交结束位置恰好为第 i+1 号线程的求交开始位置，由此可以使用线程间的同步将索引位置进行通信，从而节省第 i 号线程再次使用二分索引查找确定求交结束点的时间。该思想参考了 Jacobi 迭代中相邻进程之间进行通信传输所需数据的方法，对倒排索引问题进行的一个特异性的优化。

精确搜素 + 块内同步

```
QueryItem* list = new QueryItem[num]();
  int count = lengthArr[1];
  for (int i = 1; i < num; i++){// 预处理
    // 起始结束位置
    list[i].cursor = find1stGreaterEqual(index,
        index[start],count,count+lengthArr[i+1]);
    count += lengthArr[i+1];
    endArr[(i-1)*totalThreads+tid]=list[i].cursor;// i的起始位置即为i-1的结束位置
  }
  __syncthreads();// 块内同步
  for(int i=1;i<num;i++){// 设置结束位置
    if(tid==totalThreads)
      list[i].end = count+lengthArr[i+1];
    else
      list[i].end=endArr[(i-1)*totalThreads+tid+1];
  }
```

7.3 细粒度实现

细粒度实现是使用类似于 **SIMD** 并行化的方法，将 GPU 视为一个超大的向量，一次进行比较若干个元素。

算法外层正常执行，无需并行分段（相应也不存在提前结束特性对算法效率造成的影响），当深入到函数内存循环比较部分时，调用核函数，一次比较若干个元素，加速求交过程。

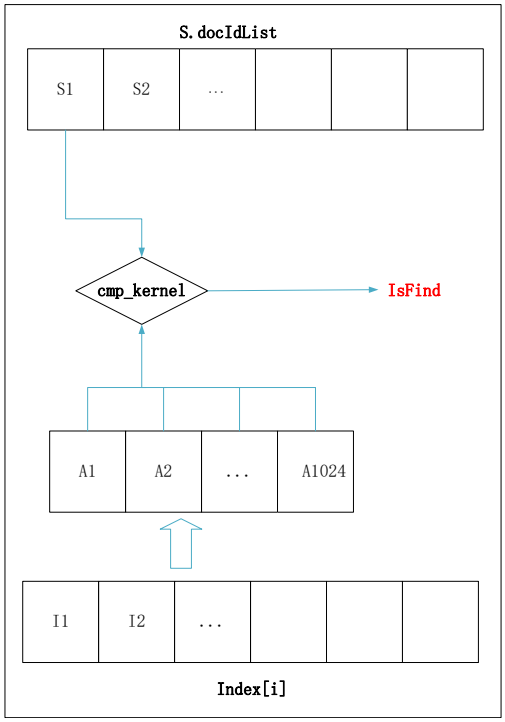

图 7.5: 细粒度算法示意图

定义核函数 cmp_kernel，负责将一个 docId 与待求交链表中 blockDim.x 个元素做比较，如果发生匹配，则将结果置为 true。因为若干个元素至多有一个元素与该 docId 相匹配，所以并不会发生冲突条件。

细粒度实现

```
while (list[s].cursor < list[s].end-1023) {// 检查s列表
    // 一次比较1024个
    cmp_kernel << <dimGrid, dimBlock >> > (gpuIndex, list[s].cursor, e,
        gpuIsFind);
    cudaDeviceSynchronize();// 同步
    // 将结果传回host端
    ret = cudaMemcpy(isFind, gpuIsFind, sizeof(bool), cudaMemcpyDeviceToHost);
    if (ret != cudaSuccess)
        fprintf(stderr, "cudaMemcpy failed: %s\n", cudaGetErrorString(ret));
    if (*isFind)
        break;
    if (e<index[list[s].cursor])
        break;
    list[s].cursor += 1024;// 当前访问过，且没找到合适的，往后移
}
```

8 实验结果分析

8.1 串行并行对比

本次实验对 SVS、ADP、Hash（参数设置为 512）三个算法进行了并行化优化。
使用并行分段的方法（粗粒度）进行算法优化效果如下所示。

	SVS	ADP	Hash(512)
串行时间	4379.99	3395.6	15422.8
并行时间（优化前）	352.97	397.21	
并行时间	199.27	180.24	361.25
加速比	21.98	18.78	42.69

表 1: 并行加速比

从表1可以看出，使用 GPU 并行加速取得了一定的优化效果。其中 SVS 算法和 ADP 算法加速比在 20 倍左右，而 Hash 算法加速比可以达到 40 倍，但三个算法都没有达到理想的百倍加速比。造成该现象的原因可能如下：

- 整个程序需要查询上千份 query，类似于 PThread 的**动态开辟线程**，查询每个 query 都需要重新对 device 端分配内存并从 host 端复制过去，造成了巨大的额外开销。依据阿姆达尔定律，算法对求交过程的加速效果需要平衡数据迁移、通信的成本，所以整体加速比并不理想。
- SVS 和 ADP 串行算法有提前结束的特性。在分段以后，位于后面段的元素失去了前面段的信息，需要从头往后寻找匹配，容易造成**工作量不均等**，进而加速不理想。虽然经过精确化搜索优化，但线程间**通信成本**也难免增大，只能是一定程度的效率提升。
- 而 Hash 算法为每个表构造 hash 表，将该表的 DocID 依次映射到对应的 hash 桶里，同一桶内元素按升序排列。在查找比较时，可以直接定位 hash 桶，再在 hash 桶小范围的查找，很少的步骤内即可查到，不需要从头进行遍历。所以加速效果相较于另外两个算法更明显。

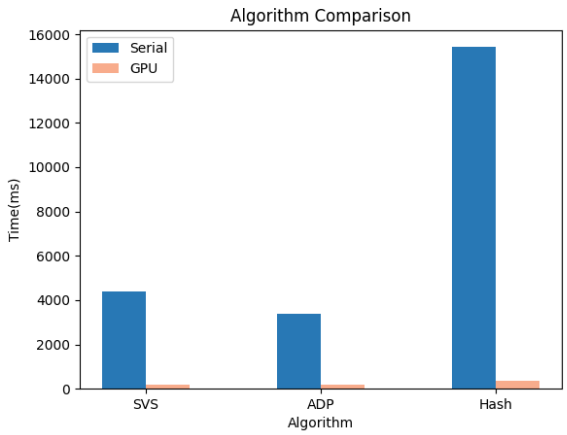

图 8.6: 分段并行加速效果

8.2 两种并行算法对比

如前所述，本次 GPU 实验共使用了两种并行手段：粗粒度和细粒度。粗粒度算法对求交链表分段划分，GPU 的每个线程分到很少一部分的 DocId，再运用相应的 SVS 等算法与其他链表求交。细粒度算法结合 SIMD 的思想，把 GPU 当作一个巨大的向量，一次可以进行比较若干个元素。两种并行手段加速效果对比如下：

	SVS	ADP	Hash(512)
串行时间	4379.99	3395.6	15422.8
粗粒度并行	199.27	180.74	361.25
加速比	21.98	18.78	42.69
细粒度并行	237.43	199.13	749.41
加速比	18.44	17.05	20.57

表 2: 两种并行方法对比

由表2可以看出，粗粒度并行的加速效果整体略优于细粒度并行，但在 Hash 算法上领先效果尤为突出。具体原因可能如下：

- 细粒度实现参照了 SIMD 的思想，在算法最内层比较元素时，一次取出大量元素放进向量，使用核函数一次性比较。虽然没有了分段并行提前结束特性的影响，但由于核函数位于最内层，除了需要传递倒排索引链表到 device 端，每次运行核函数还需要传输比较结果 isFind，下一次循环判断又需重新赋值通信，**通信开销成本高**。
- SVS 和 ADP 算法使用细粒度实现没有了提前结束的影响，但平衡上通信开销，导致两次并行化差距并不大，加速效果基本保持一致。而 Hash 算法本身没有提前结束特性，再使用细粒度实现后，额外增加了多余的通信开销，于是**加速效果回落**到与另外两个算法一致。

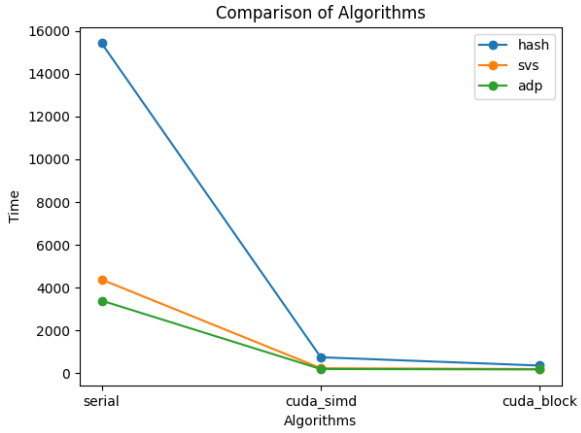

图 8.7: 加速效果对比

8.3 两种内存分配方式对比

为了探索是否可以减少 host 端与 device 端通信成本开销，使用了 **cudaMallocManaged** 进行统一内存分配，探究两种内存分配方式对性能影响。

下图展示了两种并行算法使用不同内存分配方式的加速效果对比。

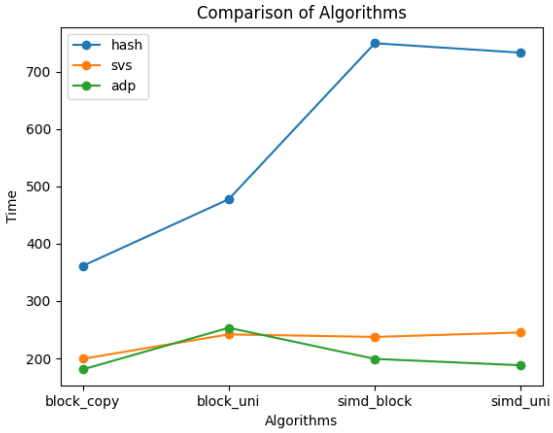

图 8.8: 两种不同内存分配方式

由图8.8可以看出，对于粗粒度并行化，使用统一内存分配的方式，效率有所降低；而细粒度并行化只对部分数据使用统一内存分配的方式，效率基本保持一致。我们通过查阅资料与实验，推测原因可能如下：

- 粗粒度并行化对每次取出的倒排索引列表进行统一内存分配，倒排索引列表在核函数内部需要**频繁的读写索引数组**。由实验指导手册得知，数据分配在 CPU 端，GPU 端的线程通过 **PCIE 总线**访问这块内存区域。每次隐式拷贝都需要走 PCIE 总线，频繁的对零拷贝内存进行读写，性能同样也会显著降低。

  虽然减少了拷贝数据的麻烦，但频繁访问，走总线运输数据对性能的影响是较大的，所以性能略有下降。

- 细粒度并行吸收了粗粒度并行的教训，没有对倒排索引列表进行统一内存分配，而是对核函数 cmp_kernel 需要用到的比较结果使用统一内存分配。省去循环拷贝 isFind，而且核函数一次只会有一个线程访问该变量，不会平凡的通过总线运输数据。于是统一内存分配略有一点提升，但由于拷贝 isFind 只占了整个算法很小一部分，所以提升并不明显，二者基本保持一致。

9 实验总结

本次实验对于倒排索引求交问题进行了 GPU 并行化研究，从细粒度和粗粒度两个角度入手，学习使用了 cuda，并对 SVS、ADP 和 Hash 算法进行了加速优化。另外，我们还探究了 cuda 的不同内存分配方式及其对性能的影响。

由于倒排索引求交这个问题不像矩阵乘法、高斯消元等算法适合使用 GPU 优化，过程中花费了较大精力优化，加速效果仍然一般。其中对于分块并行的精确化搜索，结合 SIMD 思想进行细粒度并行等都花费了较多时间处理。另外，由于倒排索引链表数据读取后都保存在 vector 中，数据从 CPU 端迁移到 GPU 端也是个棘手的问题。在尝试使用二维数组无果后，转而使用一维数组，并额外记录每个链表长度偏移量实现。

在过程中我们还了解到，对于**倒排索引压缩问题适合使用 GPU 优化**。由于时间有限，我们将在期末报告中对其进行研究。

通过本次实验，我们学习掌握了 GPU 并行的方法，熟悉了 CUDA 的使用，了解了许多相关接口。最重要的，还为我们期末研究报告指明了方向。如今，高性能计算逐渐成为热点（NVIDIA 公司凭借显卡产业如日中天），在以后的学习过程中，我们将更多的学习使用 CUDA 进行算法的并行化，充分利用 GPU 的强大计算能力为自己的学习与开发服务。