南開大學
Nankai University

计算机学院

并行程序设计实验报告

自选题目：基于 COO 格式的稀疏矩阵乘法 CUDA 优化

姓名：孟笑朵

学号：2010349

专业：计算机科学与技术

2022 年 6 月 15 日

Abstract

在本次实验中，利用常规的稀疏矩阵存储方式 COO 进行存储，在英伟达的 NOVIDIA 平台上实现了稀疏矩阵乘法 SpMM 和稀疏矩阵向量乘法 SpMV 的 CUDA 优化并行算法，并结合 nsys 以及可视化工具对 CUDA 程序进行进一步分析.

关键字： SpMV，SpMM，NOVIDIA，CUDA, nsys

目录

1 性能测试环境

英伟达 NVIDIA 虚拟环境

- 编译器: nvcc
- 中心处理器：Tesla T4
- CUDA Version: 11.2
- 存储：15109MiB

2 自选题介绍

本文是作为同杨鑫合作的《WMD 算法的并行化优化》的一个子问题的解决方案，该问题聚焦于 WMD 算法中最终求解的 Sinkhorn-Knopp 算法的并行优化，算法中涉及到多次稀疏矩阵与稠密向量矩阵相乘，因此，实现稀疏矩阵乘法的并行化十分有必要。不仅在本算法中，稀疏矩阵/张量在科学计算、数据分析、机器学习等应用中也十分常见，而稀疏矩阵的间接内存访问模式又给代码优化带来了巨大挑战，实现一种高效的稀疏矩阵乘法算法意义重大。

在《基于 COO 格式的稀疏矩阵乘法的 SIMD 编程》中，我们已经详细阐述了关于稀疏矩阵向量乘法的一些概念,并对稀疏矩阵压缩格式进行简单的介绍。在《基于 COO 格式的稀疏矩阵乘法的 pThread 编程》中，我们进行了稀疏矩阵向量乘的 pThread 并行优化并进行了性能测试，并对 HYB(HYBrid) 格式表示的稀疏矩阵进行了简单介绍。在《基于 COO 格式的稀疏矩阵乘法的 open+ 编程》中，我们实现了 openMP 的稀疏矩阵乘法并行化优化，并结合 SIMD 进行程序的进一步优化，并且对比了 pThread 的优化效果，进行了一系列性能测试。

在本次实验中，我们继续使用 COO 格式的稀疏矩阵乘法的并行化优化，在大作业中使用 HYB 格式的稀疏矩阵乘法并行化优化，为之后进行 HYB 格式的稀疏矩阵乘法做准备。本实验中继续将稀疏矩阵乘法划分为两类：一类是稀疏矩阵稠密向量相乘（SpMV），一类是稀疏矩阵稠密矩阵相乘（SpMM），将这两类问题分别实现 CPU 的 CUDA 并行化加速，并比较分析其相比于平凡算法的性能提升。

3 并行算法实现与性能分析

3.1 SpMV 串行算法与并行分析

基于 COO 的稀疏矩阵格式与向量相乘的算法伪代码如下：

$$\text{for } l = 0 \text{ to } k-1 \text{ do}$$
$$y[\text{ rowind }[l]] \leftarrow \text{y[rowind }[l]] + \text{val}[l] \cdot \text{x[colind}[l]]$$

在 SIMD 中，我们对这种算法进行了访存分析，发现这种算法至少要进行五次 memory load 才能进行一次运算，虽然我们也进行了循环展开的 SIMD 编程，但是由于这种算法本身的局限性，即空间访问并不具备连续性，强行进行 SIMD 的向量化反而适得其反，使得 SIMD 并行化算法反而比平凡算法性能更差。在 pThread 编程中，我们优化了 COO 的稀疏矩阵表示格式，并说明了对外层的多线程优化要明显优于对内层的多线程并行化。结合 CUDA 并行化优化的特点，在本次实验中，我们继续采用下面这种稀疏矩阵格式进行 SpMV 并行化优化。

COO 存储改进格式

```
class COOMatrix
{
    float* value,//存储非零元素的值
    int* row,//存储非零元素的行下标
    int* col,//存储非零元素的列下标
    int* index//存储每行第一个非零元素在row中的下标
};
```

利用上述形式进行 SpMV 平凡算法计算的算法伪代码如下:

```
int i,j;
for(i=0;i<nozerorows;i++)
{
    for(j=index[i];j<index[i+1];j++)
    {
        yy[row[j]]+=value[j]*vec[col[j]];
    }
}
```

3.2 SpMV 的 GPU 并行化优化

结合 CUDA 编程的特点，需要自行设置配置变量，包括线程块的数目以及线程块内线程的数目，有时候会出现网格工作量不匹配的情况，我们对解决方案是如下设置线程块数以及线程块内线程的个数:

```
size_t threadsPerBlock;
size_t numberOfBlocks;

threadsPerBlock = 128;//包含多个线程（32 的倍数）的块通常具有性能优势
numberOfBlocks = (nozerorows + threadsPerBlock - 1) / threadsPerBlock;
```

结合平凡算法的特点, 我们设置了 SpMV 计算的核函数, 代码如下:

```
__global__
void spmv_gpu(int nozerorows,float* vec,int* row,int* col,
                float* value,int* indexVec,float* y){
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    for(int i=index;i<nozerorows;i+=stride)
    {
        for(int j=indexVec[i];j<indexVec[i+1];j++)
```

```
9           {
10              y[row[j]]+=value[j]*vec[col[i]];
11          }
12      }
13  }
```

3.3 SpMV 的 GPU 并行化算法的性能测试

由于经过测试,NVIDIA 的实验环境相比于本地的实验环境还具有一定的性能差距, 故本次实验只对比平凡算法和 CUDA 算法之间的性能, 我们发现, 当数据规模较小时,GPU 的运算优势并不明显, 甚至要差于平凡算法, 但是当数据规模较大时,CUDA 算法的性能优势就有所体现, 具体性能测试结果如下表可见:

数据规模	CPU 时间	GPU 时间
1024	0.122080 ms	0.326368 ms
4096	0.695456 ms	0.806016 ms
8192	2.948384 ms	1.719904 ms

另外, 我们结合 nsys 工具对程序性能做了进一步分析, 我们主要查看了三种信息 CUDA API Statistics、CUDA 内核统计信息以及 CUDA 内存操作统计信息（时间和大小）, 具体如下图所示

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
55.2	579676	2	289838.0	288734	290942	[CUDA memset]
28.2	295609	71	4163.5	2175	44063	[CUDA Unified Memory memcpy HtoD]
16.6	174555	6	29092.5	1535	109470	[CUDA Unified Memory memcpy DtoH]

图 3.1: spMV 的 CUDA API Statistics

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
131072.000	2	65536.000	65536.000	65536.000	[CUDA memset]
1088.000	6	181.333	4.000	700.000	[CUDA Unified Memory memcpy DtoH]
1088.000	71	15.324	4.000	256.000	[CUDA Unified Memory memcpy HtoD]

图 3.2: spMV 的 CUDA 内核统计信息

Operating System Runtime API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
85.8	1856659968	97	19140824.4	57646	100149618	poll
8.4	182619101	85	2148460.0	16689	20566153	sem_timedwait
5.4	117586272	691	170168.3	1009	18243453	ioctl
0.1	3022585	94	32155.2	1427	813687	mmap
0.1	2219065	82	27061.8	4640	46955	open64
0.0	215174	3	71724.7	69517	75235	fgets
0.0	195093	4	48773.3	37161	58057	pthread_create
0.0	149141	25	5965.6	1700	25279	fopen
0.0	122215	11	11110.5	4512	18031	write
0.0	56672	39	1453.1	1004	6863	fcntl
0.0	52177	11	4743.4	2133	8861	munmap
0.0	38233	4	9558.3	1202	18297	pthread_rwlock_timedwrlock
0.0	36158	5	7231.6	3568	9907	open
0.0	30020	18	1667.8	1181	5228	fclose
0.0	27454	6	4575.7	1053	12432	fgetc
0.0	23556	12	1963.0	1368	3396	read
0.0	19914	2	9957.0	9216	10698	socket
0.0	12575	3	4191.7	2199	6215	fread
0.0	10360	1	10360.0	10360	10360	pipe2
0.0	9749	1	9749.0	9749	9749	connect
0.0	8956	4	2239.0	1909	2920	mprotect
0.0	3230	1	3230.0	3230	3230	bind
0.0	1857	1	1857.0	1857	1857	listen

图 3.3: spMV 的 CUDA 内存操作统计信息（时间和大小）

3.4 SpMM 串行算法与 GPU 并行化优化

在 SIMD 实验中，我们进行简单的 Cache 优化所得到稀疏矩阵相乘算法伪代码如下：

```
void coo_multiply_matrix_serial(){
    for (int i=0;i<nonzeros;i++)
        for(int k=0;k<n;k++)
            c[row[i]][k] += value[i] * b[col[i]][k];
}
```

对于稀疏矩阵乘法，我们采取的 CUDA 并行化思路和 openMP 以及 pThread 中并行化思路类似，在稀疏矩阵乘法的外层循环进行划分。结合 CUDA 编程的特点进行设计算法，具体算法代码如下：

```
__global__
void spmm_gpu(int nonzeros,int n,int* row,int* col,
        float* value,float**b,float**c){
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    for (int i=index;i<nonzeros;i++)
        for(int k=0;k<n;k++)
            c[row[i]][k] += value[i] * b[col[i]][k];
}
```

3.5 SpMM 的并行化算法性能分析

但是在进行 spMM 的 CUDA 虚拟环境的实验时，出现了 Segmentation fault (core dumped) 报错，最初以为是申请内存空间过大引起的，在多次修改数据规模后依旧报告这个错误，数次排错未果，受到时间限制，不得已暂时记录下来代之后进行仔细排错。

受作者水平所限，这部分实验暂时没有执行完毕，实为一大遗憾!

在数次尝试未果后，笔者为了弥补工作量的不足，进行了矩阵乘法的 GPU 优化算法实验具体如下:

3.6 矩阵乘法的 GPU 并行算法

矩阵优化的 GPU 的并行化函数的关键在于如何将矩阵乘法的三层循环进行优化，在 GPU 的并行化中使用的是三维的线程块来进行优化的，巧妙地将三层循环化为一层循环:

```
__global__ void matrixMulGPU( int * a, int * b, int * c )
{
  int val = 0;

  int row = blockIdx.x * blockDim.x + threadIdx.x;
  int col = blockIdx.y * blockDim.y + threadIdx.y;

  if (row < N && col < N)
  {
    for ( int k = 0; k < N; ++k )
      val += a[row * N + k] * b[k * N + col];
    c[row * N + col] = val;
  }
}
```

```
int main()
{
  int *a, *b, *c_cpu, *c_gpu;

  int size = N * N * sizeof (int); // Number of bytes of an N x N matrix

  // Allocate memory
  cudaMallocManaged (&a, size);
  cudaMallocManaged (&b, size);
  cudaMallocManaged (&c_cpu, size);
  cudaMallocManaged (&c_gpu, size);

  // Initialize memory
  for( int row = 0; row < N; ++row )
```

```
15      for( int col = 0; col < N; ++col )
16      {
17        a[row*N + col] = row;
18        b[row*N + col] = col+2;
19        c_cpu[row*N + col] = 0;
20        c_gpu[row*N + col] = 0;
21      }
22
23    dim3 threads_per_block (16, 16, 1); // A 16 x 16 block threads
24    dim3 number_of_blocks ((N / threads_per_block.x) + 1, (N / threads_per_block.y) +    1, 1);
25
26    matrixMulGPU <<< number_of_blocks, threads_per_block >>> ( a, b, c_gpu );
27
28    cudaDeviceSynchronize(); // Wait for the GPU to finish before proceeding
29
30    // Call the CPU version to check our work
31    matrixMulCPU( a, b, c_cpu );
32
33    // Free all our allocated memory
34    cudaFree(a); cudaFree(b);
35    cudaFree( c_cpu ); cudaFree( c_gpu );
36  }
```

4 总结与收获

在本次实验中，得到了以下几点收获：

1. 学会了利用 NVIDIA 的实验环境进行基本的 GPU 编程;
2. 学会了使用 nsys 以及 Nsight Systems 进行可视化分析;

5 代码上传

全部代码素材已上传GitHub.