CUDA&CUDA 和 OpenMP 混合编程实验报告

姓名

2024 年 5 月 30 日

1 实验环境

Windows11 Professional 22H2 x64:

- 处理器：Intel(R) Core(TM) i7-11800H CPU @ 4.60GHz 16 线程
- 内存：32.0 GB
- 显卡：NVIDIA GeForce RTX 3070
- 显卡驱动版本：11.6
- 编程语言：C++
- 编译器：nvcc
- 编程平台：Visual Studio 2019

2 实验目的

1. 了解 CUDA 的安装方式和注意问题。通过简洁明了地列出 CUDA 安装的步骤和关键注意事项，目的在于为 CUDA 新用户提供一个清晰、易于遵循的安装指南。这不仅帮助用户高效完成 CUDA 环境的配置，也旨在通过分享实践经验，降低并行编程技术的入门门槛，尤其是对于那些首次尝试在个人计算机上进行 CUDA 编程的用户。

2. 使用 CUDA 或者 CUDA 和 OpenMP 混合编程实现矩阵乘法并且根据自己的电脑配置进行优化。现矩阵乘法的 CUDA 版本以及可能的 OpenMP-CUDA 混合编程版本，并基于个人机器的具体配置，从多个角度对比其加速比。此实验目的在于探究 CUDA 和 OpenMP-CUDA 混合编程技术在执行复杂运算任务时的性能差异，以及评估在不同计算资源配置下的执行效率。通过实验，旨在深入理解 GPU 并行计算的优势与局限，以及混合编程模式在提升计算性能方面的潜力，同时，通过附上源码，增加实验的可重复性和透明度，为并行计算领域的学习者和研究者提供参考。

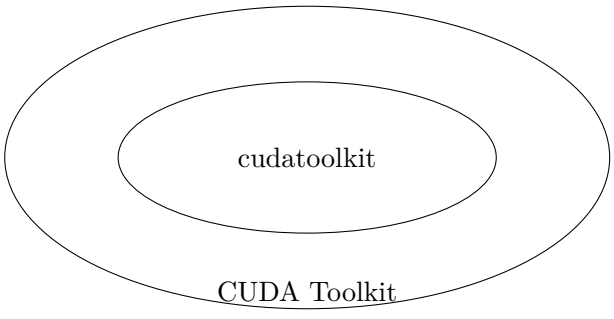
3 实验步骤

3.1 CUDA 的安装

1. 下载 **CUDA Toolkit**：这一步需要区分任务性质，首先确定是进行深度学习还是进行 CUDA 编程，如果只是进行深度学习，那么没没有必要安装完整的 CUDA Toolkit，只需使用 conda 在虚拟环境中安装 cudatoolkit 即可，步骤如 (b)；如果是进行 CUDA 编程，那么需要下载完整的 CUDA Toolkit 如步骤 (a)。

CUDAToolkit =	NVIDIA CUDA Development	包含进行 CUDA 编程所需的库、头文件和编译工具。
	NVIDIA CUDA Documentation	提供 CUDA 的官方文档，包括开发者指南、API 参考、编程指南等。
	NVIDIA CUDA Nsight NVTX	性能分析工具，用于标记和组织 GPU 性能分析的时序事件。
	NVIDIA CUDA Runtime	包含 CUDA 运行时库，支持 CUDA 程序的执行和运行。
	NVIDIA CUDA Samples	一组演示如何使用 CUDA 进行编程的示例。
	NVIDIA CUDA Visual Studio Integration	CUDA 与 Visual Studio 集成的工具和插件。
	NVIDIA Nsight Compute	用于 CUDA 应用的性能分析工具，提供深入的硬件级别性能分析。
	NVIDIA Nsight Systems	系统级性能分析工具，提供全面的系统性能分析，包括 CPU 和 GPU 活动。

Conda cudatoolkit =	NVIDIA CUDA Runtime	包含 CUDA 运行时库，支持 CUDA 程序的执行和运行。
	部分 NVIDIA CUDA Development	可能包括一些必要的库和头文件，用于基础的 CUDA 开发。



cudatoolkit 和 CUDA Toolkit 包含关系

(a) 访问 NVIDIA 官网，下载适用于自己操作系统的 CUDA Toolkit。下载地址：`https://developer.nvidia.com/cuda-toolkit-archive`。选择自己的系统相关信息如下图所示：

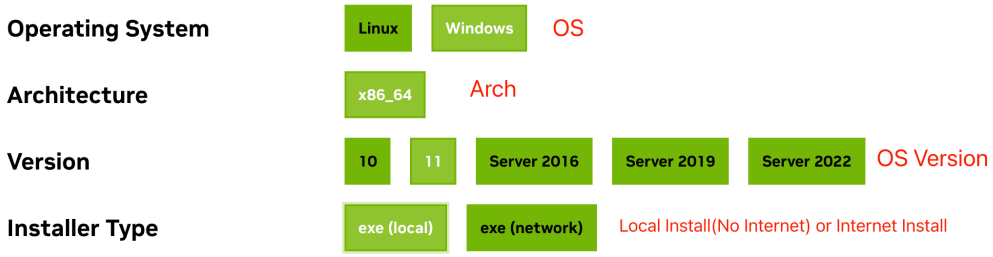


图 1: CUDA Toolkit 下载

(b) 打开`https://anaconda.org/`，搜索 cudatoolkit，找到 Pytorch/Tensorflow 需要的 cudatoolkit 版本，然后进复制下载命令进行下载比如下面下载 11.6 版本的 cudatoolkit。

```
conda install deepmodeling::cudatoolkit
```

2. 运行刚刚下载可执行文件，全部默认安装即可，如果不在 C 盘安装会出现一些问题，有的程序会默认去 C 盘找 CUDA 的驱动文件导致无法运行。Linux 不分盘所以无所谓。

3. 安装完成后，需要配置环境变量，需要配置如下有名称的环境变量：

```
CUDA_PATH = C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6
CUDA_PATH_V11_6 = C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11
    .6
```

和下面 Path 变量中的条目：

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6\bin
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6\libnvvp
```

其中一个系统中可以有多个驱动版本，如果你想要使用什么版本，可以修改 `CUDA_PATH` 以及将 Path 中的相应版本的路径上移到所有版本的路径之上即可实现快速的 CUDA 版本切换。

4. 这时已经基本配置完成，`win+R` 输入 `powershell` 打开 powershell，输入下面的指令和看到下面输出的结果之后，说明 CUDA 已经安装成功。

```
(base) PS C:\Users\24692> nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Fri_Dec_17_18:28:54_Pacific_Standard_Time_2021
Cuda compilation tools, release 11.6, V11.6.55
Build cuda_11.6.r11.6/compiler.30794723_0
(base) PS C:\Users\24692> nvidia-smi
Mon Apr  8 09:38:47 2024
NVIDIA-SMI 551.86    Driver Version: 551.86    CUDA Version: 12.4
```

这里的 CUDA Version 和 CUDA Toolkit 的版本不一样是因为这里的 CUDA Version 是显卡支持的最高 CUDA 版本，而 CUDA Toolkit 是你安装的 CUDA 版本。

5. 按需安装 cuDNN，cuDNN 是深度学习底层的算子库，为一些深度学习中的基本运算提供加速。

3.2 Ablation Study

本实验将这里的矩阵乘法任务分为下面两个子任务：

- 数据的初始化：相乘的两个矩阵的随机数初始化的阶段。
- 矩阵乘法：两个矩阵相乘的阶段 (CUDA 编程还包括将数据复制回主存的时间)。

然后本实验针对上面这两个阶段进行编程的设计。其中数据的初始化需要两层循环进行初始化，但是可以预见到，这里的两层循环的任务是可以并行的，因为每个矩阵元素的初始化操作都是独立的。所以这里可以使用 `#pragma omp parallel for collapse` 进行并行化。而矩阵乘法的阶段的最后结果的每一个元素的计算都是相互独立的，所以可以并行计算，这里可以使用 OpenMP 和 CUDA 进行并行化计算。实验的源代码可以见附录。

Config	1	2	3	4	5	6
init omp	-	True	-	-	True	True
cal omp	-	-	True	-	True	-
cal gpu	-	-	-	True	-	True
1	2.67	2.75	0.46	0.0816	0.48	0.1912
2	2.66	2.77	0.47	0.0812	0.47	0.1855
3	2.70	2.76	0.45	0.0742	0.49	0.1852
4	2.67	2.73	0.47	0.0788	0.47	0.1901
5	2.66	2.77	0.46	0.0796	0.47	0.1935
ave	2.67200	2.75600	0.46200	0.07908	0.47600	0.18910
SpeedUp	-	0.969521	5.783550	33.788569	5.613445	14.130090

表 1: Ablation Of Matrix Sized 1000(s)

Config	1	2	3	4	5	6
init omp	-	True	-	-	True	True
cal omp	-	-	True	-	True	-
cal gpu	-	-	-	True	-	True
1	32.65	34.13	5.67	0.3685	5.78	0.1695
2	31.95	32.15	5.62	0.3670	5.69	0.1718
3	32.01	32.70	5.70	0.3702	5.67	0.1909
4	32.20	32.22	5.64	0.3764	5.79	0.2055
5	31.97	33.20	5.66	0.3718	5.90	0.1749
ave	32.15600	32.88000	5.65800	0.37078	5.76600	0.18252
SpeedUp	-	0.977981	5.683280	86.725282	5.576830	176.177953

表 2: Ablation Of Matrix Sized 2000(s)

Config	1	2	3	4	5	6
init omp	-	True	-	-	True	True
cal omp	-	-	True	-	True	-
cal gpu	-	-	-	True	-	True
1	146.68	158.80	21.79	0.9331	22.01	0.5067
2	143.14	144.90	21.66	0.9471	21.90	0.5045
3	144.12	144.10	21.63	0.9199	21.80	0.4886
4	143.30	146.13	21.88	0.9233	21.89	0.5018
5	144.22	145.32	21.70	0.9545	22.02	0.4909
ave	144.292	147.85	21.732	0.93558	21.924	0.4985
SpeedUp	-	0.976	6.64	154.23	6.58	289.45

表 3: Ablation Of Matrix Sized 3000(s)

通过上面的消融实验的数据，可以得到下面的结果：

1. GPU 的有效加速：使用 GPU 对矩阵乘法这种高密度高并行高独立性的计算的提升是巨大的，可以从表1表2 表3中看出，从一开始的 1000 大小矩阵的加速比 33.79 到最后的 3000 大小矩阵的加速比 289.45(相比于基线数据)，其中相比于其他的并行化操作，比如只使用 OpenMP 优化乘法，同时使用 OpenMP 优化初始化和乘法也有着较为明显的加速。可以看出越大规模的矩阵运算，GPU 的加速效果越高，具体的原因分析有下：

   (a) 并行计算能力

   - 硬件架构优势：GPU 含有成百上千个核心 (RTX3070 Laptop 5120 个)，能够同时执行成千上万的线程。这种大规模并行处理能力使得 GPU 非常适合执行数据并行任务，如矩阵乘法。
   - 数据并行：在矩阵乘法中，计算每个输出元素的操作是独立的。CUDA 能够将这些操作分布到多个线程中并行执行，显著减少总的计算时间。

   (b) 内存带宽

   - 高内存带宽：GPU 具有非常高的内存带宽，能够更快地读写数据，这对于数据密集型的计算任务尤其重要，因为矩阵乘法需要频繁地访问内存中的数据。

   (c) 优化的内存访问

   - 共享内存和缓存：CUDA 允许开发者使用 GPU 上的共享内存，这是一种低延迟、高带宽的内存。合理利用共享内存可以减少对全局内存的访问次数，提高数据访问效率。此外，通过优化内存访问模式，可以进一步提高性能。

2. 使用 OpenMP 不一定会产生加速：从上面可以看出使用 OpenMP 并行优化数据的初始化并不一定能带来很明显的加速效果。其中只使用 OpenMP 并行优化数据初始化操作对比普通的串行程序来看加速比并不高，甚至还低于 1，这是因为 OpenMP 的并行化操作也会带来一些额外的时间开销，这些时间开销可能会抵消掉 OpenMP 并行化优化的时间。在只使用 OpenMP 优化乘法操作以及同时使用 OpenMP 优化初始化和乘法操作的对比中，也不是很明显。更加明显的是只有 GPU 加速乘法以及 OpenMP 加速初始化和 GPU 优化乘法，这两组对比中可以看出，在 1000 大小的矩阵上 OpenMP 居然出现了负优化的现象，但是在后续的 2000 大小和 3000 大小矩阵中却有着明显的加速效果，可以总结为以下几点：

   (a) **OpenMP** 并行化开销：可以看出，如果本身的操作速度已经很快，那么没有必要一定要生搬硬套 OpenMP，OpenMP 启动并行化会有额外的开销，如果这种开销大于并行化所带来的加

速效果，那么使用 OpenMP 只能是得不偿失，会对程序产生负优化，很好的例子就是 1000 大小矩阵表1中单独 GPU 加速乘法以及使用 OpenMP 优化初始化和 GPU 加速乘法的负优化的例子。从这里也可以总结出 OpenMP 应该使用的场景。如果一个场景有很多的串行算法模块，$O(1), O(n), O(n^2), o(n^3) \dots O(n^x)$，那么就需要使用 OpenMP 并行化处理高复杂度的算法模块 (如果可以的话，要求算法模块内部有一定的无关性)，并且前提是 $T_{parallelcost} > T(Algorithm_{O(n^x)})$ ，如果 OpenMP 的并行化开销大于任何一个模块的优化的时间，那么所有的优化都是没有意义的，并且使用 OpenMP 优化复杂度高的算法，降低复杂度，这样使得总体的复杂度降低，以此来进行加速。

(b) 普通的程序的瓶颈不仅在 **CPU** 而且在访存：为什么串行程序和 OpenMP 优化矩阵乘法加上 OpenMP 优化初始化的加速比都没有显著的提升，并且 GPU 在加上了初始化的 OpenMP 优化会产生越来越大的初始化操作，一方面的原因是因为 GPU 的并行化远高于 GPU，另一点是普通的程序在使用大量的时间在访存上。访存时间成为了程序的瓶颈，再怎么使用 OpenMP 开多线程也没有什么作用都要停止等待内存数据搬运到 GPU。并且 GPU 加上 OpenMP 优化初始化的加速比随着矩阵大小越来越大，这说明，GPU 没有访存的高额开销，瓶颈就是在初始化的优化程度上。

4 实验感想

通过这次实验，我对 GPU 计算的潜力和并行计算的效率有了更深刻的认识。通过做实验记录数据，我了解了 GPU 在处理矩阵乘法这类高度并行化的任务上，相较于传统 CPU 具有显著的加速优势。通过了消融实验，我被 GPU 架构带来的硬件优势和内存带宽的高效利用所震撼。CUDA 技术真正实现了通过并行处理大幅度减少计算时间的可能性，而这一点在规模更大的矩阵乘法中表现得尤为明显。

在进行 OpenMP 并行化时，实验教会了我并行化是一把双刃剑。OpenMP 的引入并不总是能带来性能上的提升，有时它的并行化开销甚至可能导致性能的下降。尤其在处理较小规模的数据时，OpenMP 的加速效果并不明显，这提示我们在决定使用并行计算时，必须考虑到算法的复杂性和计算负担。

此外，我也意识到，普通程序的访存操作在串行计算中占据了大量的时间，成为性能提升的瓶颈。而 GPU 的高效计算能力和内存操作显然突破了这一限制，尤其是在矩阵规模增大时，GPU 计算的优势更加凸显。我认为，理解和合理利用硬件特性对于优化程序性能至关重要。

总之，这次实验不仅加深了我对并行计算技术的理解，而且也让我更加清晰地认识到了在实际应用中如何根据计算任务的特点和规模来选择合适的优化策略。这将对我未来在计算机科学和深度学习领域的学习和研究提供极大的帮助。

A 源代码

```
#include <iostream>
#include <cstdlib>
#include <cuda_runtime.h>
#include <omp.h>
#define TILE_SIZE 16
#define BLOCK_SIZE 16
__global__ void MatrixMulCUDA(int* A, int* B, int* C, int N) {
    int ROW = blockIdx.y * blockDim.y + threadIdx.y;
    int COL = blockIdx.x * blockDim.x + threadIdx.x;
    int tmpSum = 0;
    if (ROW < N && COL < N) {
        for (int i = 0; i < N; i++) {
            tmpSum += A[ROW * N + i] * B[i * N + COL];
        }
    }
    C[ROW * N + COL] = tmpSum;
}
int main() {
    int sizes[] = { 1000, 2000, 3000 };
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);
    for (int n = 0; n < numSizes; n++) {
        int N = sizes[n];
        size_t bytes = N * N * sizeof(int);
        int* A, * B, * C;
        int* d_A, * d_B, * d_C;
        double start = omp_get_wtime();
        A = (int*)malloc(bytes);
        B = (int*)malloc(bytes);
        C = (int*)malloc(bytes);
#pragma omp parallel for collapse(2)
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                A[i * N + j] = rand() % 10;
                B[i * N + j] = rand() % 10;
            }
        }
        cudaMalloc(&d_A, bytes);
        cudaMalloc(&d_B, bytes);
        cudaMalloc(&d_C, bytes);
        cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
        dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
```

```
        dim3 blocksPerGrid((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N +
            BLOCK_SIZE - 1) / BLOCK_SIZE);
        MatrixMulCUDA << <blocksPerGrid, threadsPerBlock >> > (d_A,
            d_B, d_C, N);
        cudaDeviceSynchronize();
        cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
        double end = omp_get_wtime();
        std::cout << "Total execution time for " << N << "x" << N <<
            " matrix: " << (end - start) * 1000 << " milliseconds" <<
            std::endl;
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
        free(A);
        free(B);
        free(C);
    }
    return 0;
}
```

Listing 1: OpenMP 和 CUDA 混合编程

```
#include <iostream>
#include <cstdlib>
#include <cuda_runtime.h>
__global__ void matrixMultiply(int* A, int* B, int* C, int N) {
    int ROW = blockIdx.y * blockDim.y + threadIdx.y;
    int COL = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if (ROW < N && COL < N) {
        for (int i = 0; i < N; i++) {
            sum += A[ROW * N + i] * B[i * N + COL];
        }
        C[ROW * N + COL] = sum;
    }
}
void runMatrixMultiply(int N) {
    size_t bytes = N * N * sizeof(int);
    int* h_A, * h_B, * h_C;
    int* d_A, * d_B, * d_C;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);
    h_A = (int*)malloc(bytes);
```

```
    h_B = (int*)malloc(bytes);
    h_C = (int*)malloc(bytes);
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);
    for (int i = 0; i < N * N; i++) {
        h_A[i] = rand() % 10;
        h_B[i] = rand() % 10;
    }
    cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);
    int blockSize = 16;
    dim3 dimBlock(blockSize, blockSize, 1);
    dim3 dimGrid((N + blockSize - 1) / blockSize, (N + blockSize - 1)
        / blockSize, 1);
    matrixMultiply << <dimGrid, dimBlock >> > (d_A, d_B, d_C, N);
    cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    std::cout << "Total execution time for " << N << "x" << N << "
       matrix: " << milliseconds << " milliseconds" << std::endl;
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);
}
int main() {
    int sizes[] = { 1000, 2000, 3000 };
    for (int i = 0; i < 3; i++) {
        runMatrixMultiply(sizes[i]);
    }
    return 0;
}
```

Listing 2: CUDA 编程