并行与分布式计算作业

“AVX 指令集与多线程编程”

第一次作业

姓名：施天予

班级：计科 2 班

学号：19335174

一、问题描述

- 在第一次课程中已经讲到，早期单节点计算系统并行的粒度分为：Bit级并行，指令级并行和线程级并行。现代处理器如Intel、ARM、AMD、Power以及国产CPU如华为鲲鹏等，均包含了并行指令集合。
  - 请调查这些处理器中的并行（向量）指令集，并选择其中一种如AVX，SSE等进行编程练习。
  - 此外，现代操作系统为了发挥多核的优势，支持多线程并行编程模型， 请利用多线程的方式实现N个整数的求和，编程语言不限，可以是Java， 也可以是 C/C++。
- Write an essay describing a research problem in your major that would benefit from the use of parallel computing. Provide a rough outline of how parallelism would be used. Would you use task- or data-parallelism?

二、解决方案

AVX指令集基本原理

数据类型

数据类型	描述
__m128	包含4个float类型数字的向量
__m128d	包含2个double类型数字的向量
__m128i	包含若干个整型数字的向量
__m256	包含8个float类型数字的向量
__m256d	包含4个double类型数字的向量
__m256i	包含若干个整型数字的向量

- 每一种类型，从2个下划线开头，接一个m，然后是vector的位长度。

- 如果向量类型是以d结束的，那么向量里面是double类型的数字。如果没有后缀，就代表向量只包含float类型的数字。
- 整形的向量可以包含各种类型的整形数，例如char, short, unsigned long long

函数命名

`_mm<bit_width>_<name>_<data_type>`

- `<bit_width>` 表明了向量的位长度，对于128位的向量，这个参数为空，对于256位的向量，这个参数为256。
- `<name>` 描述了内联函数的算术操作。
- `<data_type>` 标识函数主参数的数据类型。
  - `ps` 包含float类型的向量
  - `pd` 包含double类型的向量
  - `epi8/epi16/epi32/epi64` 包含8位/16位/32位/64位的有符号整数
  - `epu8/epu16/epu32/epu64` 包含8位/16位/32位/64位的无符号整数
  - `si128/si256` 未指定的128位或者256位向量
  - `m128/m128i/m128d/m256/m256i/m256d` 当输入向量类型与返回向量的类型不同时，标识输入向量类型

实验环境

硬件

- Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- NVIDIA GeForce GTX 1660Ti

软件

- Ubuntu 20.04.2
- gcc 9.3.0
- Vscode

实验过程

在本次实验中，我取 N = 1000000 ， 并把每个数设为1，进行N个数求和

串行编程

在串行程序中，直接用一个for循环求和

```
#include <stdio.h>
#include <sys/time.h>

/* 获取时间 */
#define  GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}

int N = 1000000;    /* 设有 N = 1000000 个数相加 */
int i, result;
double start, stop;

int main(int argc, char* argv[]) {
    int s[N];
    for (i = 0; i < N; ++i)     /* 将N个数设为1 */
        s[i] = 1;
    GET_TIME(start);
    for (i = 0; i < N; ++i)     /* 串行求和 */
        result += s[i];
    GET_TIME(stop);
    printf("%d\n", result);     /* 输出结果 */
    printf("Run time: %e\n", stop-start);   /* 计算时间 */
}
```

AVX编程

在AVX程序中，用 `_mm256_set1_epi32(1)` 将N个数8个一组（256位）设为1

用 `_mm256_add_epi32` 8个一组（256位）求和

再把最后得到的8个数加起来就是N个数的总和

```
#include <stdio.h>
#include <immintrin.h>
#include <sys/time.h>

/* 获取时间 */
#define  GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
```

```
}

int N = 1000000;    /* 设有 N = 1000000 个数相加 */
int i, result;
double start, stop;
__m256i n;

int main(int argc, char* argv[]) {
    __m256i s[N/8 + 1];
    for(int i = 0; i < N/8; ++i)    /* 将N个数设为1 */
        s[i] = _mm256_set1_epi32(1);
    GET_TIME(start);
    n = _mm256_load_si256((const __m256i *)(s));
    for (int i = 1; i < N/8; ++i)   /* 256位一组求和 */
        n = _mm256_add_epi32(n, s[i]);
    int sum[20];
    _mm256_store_ps((float *)sum, (__m256)n);
    for (i = 0; i < 8; ++i)         /* 8个数求和 */
        result += sum[i];
    GET_TIME(stop);
    printf("%d\n", result);     /* 输出结果 */
    printf("Run time: %e\n", stop-start);   /* 计算时间 */
}
```

多线程编程

多线程编程我使用了OpenMp编程，C语言编程

利用`# pragma omp parallel for`和`reduction`归约子句，实现N个数并行求和

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>

/* 获取时间 */
#define  GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}

int N = 1000000;    /* 设有 N = 1000000 个数相加 */
int thread_count, i, result;
double start, stop;

int main(int argc, char* argv[]) {
    if (argc == 2)      /* 获取线程数 */
```

```
        thread_count = strtol(argv[1], NULL, 10);
    int s[N];
    for (i = 0; i < N; ++i)      /* 将N个数设为1 */
        s[i] = 1;
    GET_TIME(start);
#   pragma omp parallel for num_threads(thread_count) \
        private(i) shared(s) reduction(+:result)
    for (i = 0; i < N; ++i)      /* 并行求和 */
        result += s[i];
    GET_TIME(stop);
    printf("%d\n", result);      /* 输出结果 */
    printf("Run time: %e\n", stop-start);   /* 计算时间 */
}
```

三、实验结果

AVX编程和串行编程的比较

在终端输入 `gcc serial_add.c -o serial_add` 命令编译，运行5次，结果如下：

```
sty@ubuntu:~$ gcc serial_add.c -o serial_add
sty@ubuntu:~$ ./serial_add
1000000
Run time: 3.388166e-03
sty@ubuntu:~$ ./serial_add
1000000
Run time: 2.215862e-03
sty@ubuntu:~$ ./serial_add
1000000
Run time: 2.229214e-03
sty@ubuntu:~$ ./serial_add
1000000
Run time: 2.242088e-03
sty@ubuntu:~$ ./serial_add
1000000
Run time: 2.259016e-03
```

在终端输入 `gcc avx_add.c -o avx_add -mavx2` 命令编译，运行5次，结果如下：

```
sty@ubuntu:~$ gcc avx_add.c -o avx_add -mavx2
sty@ubuntu:~$ ./avx_add
1000000
Run time: 6.380081e-04
sty@ubuntu:~$ ./avx_add
1000000
Run time: 5.950928e-04
sty@ubuntu:~$ ./avx_add
1000000
Run time: 7.240772e-04
sty@ubuntu:~$ ./avx_add
1000000
Run time: 8.850098e-04
sty@ubuntu:~$ ./avx_add
1000000
Run time: 5.769730e-04
```

统计两次运行结果：

运行次数	串行编程运行时间（μs）	AVX编程运行时间（μs）
1	3388	638
2	2215	595
3	2229	724
4	2242	885
5	2259	576
平均	2466.6	683.6

发现使用AVX指令集的运行速度更快，加速比$S = \frac{T_{串行}}{T_{AVX}} = 3.608$

多线程编程和串行编程的比较

在终端输入 `gcc -g -Wall -fopenmp -o parallel_add parallel_add.c` 命令编译，设置不同的线程数，对每种情况都运行5次，结果如下：

线程数为2

```
sty@ubuntu:~$ gcc -g -Wall -fopenmp -o parallel_add parallel_add.c
sty@ubuntu:~$ ./parallel_add 2
1000000
Run time: 8.308887e-04
sty@ubuntu:~$ ./parallel_add 2
1000000
Run time: 8.509159e-04
sty@ubuntu:~$ ./parallel_add 2
1000000
Run time: 8.380413e-04
sty@ubuntu:~$ ./parallel_add 2
1000000
Run time: 8.780956e-04
sty@ubuntu:~$ ./parallel_add 2
1000000
Run time: 8.339882e-04
```

线程数为4

```
sty@ubuntu:~$ gcc -g -Wall -fopenmp -o parallel_add parallel_add.c
sty@ubuntu:~$ ./parallel_add 4
1000000
Run time: 7.090569e-04
sty@ubuntu:~$ ./parallel_add 4
1000000
Run time: 6.000996e-04
sty@ubuntu:~$ ./parallel_add 4
1000000
Run time: 6.079674e-04
sty@ubuntu:~$ ./parallel_add 4
1000000
Run time: 6.339550e-04
sty@ubuntu:~$ ./parallel_add 4
1000000
Run time: 1.123190e-03
```

线程数为8

```
sty@ubuntu:~$ gcc -g -Wall -fopenmp -o parallel_add parallel_add.c
sty@ubuntu:~$ ./parallel_add 8
1000000
Run time: 8.161068e-04
sty@ubuntu:~$ ./parallel_add 8
1000000
Run time: 8.111000e-04
sty@ubuntu:~$ ./parallel_add 8
1000000
Run time: 1.038074e-03
sty@ubuntu:~$ ./parallel_add 8
1000000
Run time: 1.326084e-03
sty@ubuntu:~$ ./parallel_add 8
1000000
Run time: 8.771420e-04
```

线程数为10

```
sty@ubuntu:~$ gcc -g -Wall -fopenmp -o parallel_add parallel_add.c
sty@ubuntu:~$ ./parallel_add 10
1000000
Run time: 9.529591e-04
sty@ubuntu:~$ ./parallel_add 10
1000000
Run time: 8.721352e-04
sty@ubuntu:~$ ./parallel_add 10
1000000
Run time: 9.520054e-04
sty@ubuntu:~$ ./parallel_add 10
1000000
Run time: 9.648800e-04
sty@ubuntu:~$ ./parallel_add 10
1000000
Run time: 1.044989e-03
```

线程数为16

```
sty@ubuntu:~$ gcc -g -Wall -fopenmp -o parallel_add parallel_add.c
sty@ubuntu:~$ ./parallel_add 16
1000000
Run time: 2.085924e-03
sty@ubuntu:~$ ./parallel_add 16
1000000
Run time: 2.145052e-03
sty@ubuntu:~$ ./parallel_add 16
1000000
Run time: 2.071857e-03
sty@ubuntu:~$ ./parallel_add 16
1000000
Run time: 2.974033e-03
sty@ubuntu:~$ ./parallel_add 16
1000000
Run time: 1.987934e-03
```

取平均后得出下表：

线程数	2	4	8	10	16
平均运行时间（μs）	845.8	734.4	973.6	956.8	2252.4
加速比（$S=\frac{T_{串行}}{T_{并行}}$）	2.916	3.358	2.533	2.577	1.095

可以发现，随着线程数增多，运行时间减小，而当线程数增多到一定程度时，运行时间又会逐渐增加

这是因为当线程太多时，由于频繁切换线程等原因，会增大时间开销

四、遇到的问题及解决方法

1、刚开始发现串行程序的运行比多线程程序快，十分不合理

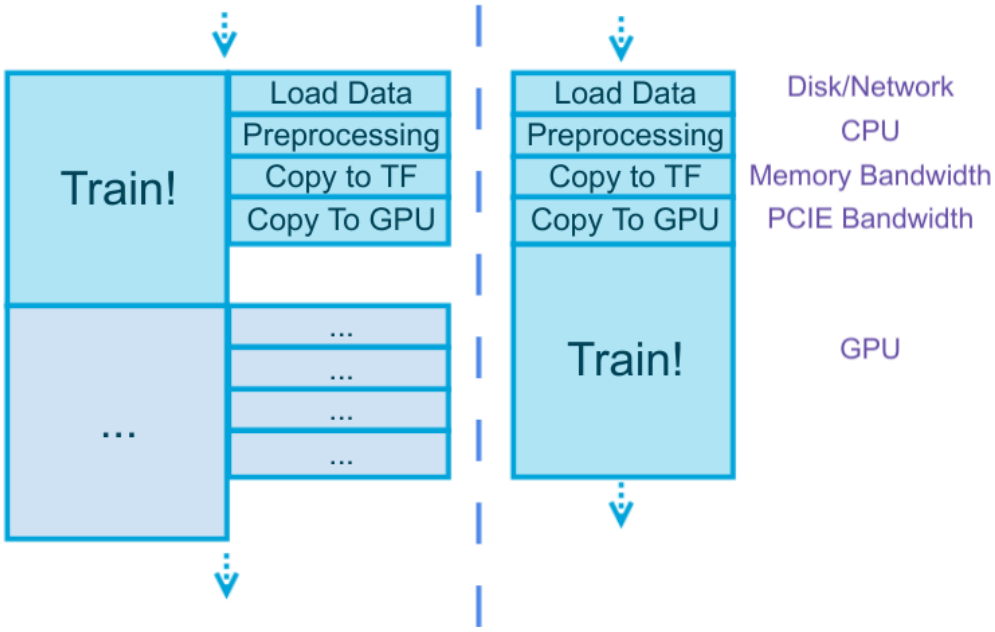
解决方法：增大问题规模，将N取值为1000000

2、在使用OpenMp编程时，使用parallel for发现求和的结果不对

解决方法：使用reduction归约子句，需要将所有线程求得的和加起来才是总和

五、关于并行计算在训练神经网络中的应用

在训练神经网络的时候，有时候数据的读取处理所占用的时间比较长，像数据量非常大的情况，或者是数据预处理比较耗内存的情况。此时我们希望并行化，即数据处理与训练同时进行，即使用**任务并行**。



- 并行化：在处理数据的时候并进行训练，当gpu空闲的时候，已经训练好的数据能够及时给到显存中。
- 串行化：必须对数据进行处理之后才可以送到gpu，gpu才可以开始工作，在数据处理完之前，gpu一直处于空闲状态，大大的浪费了资源。

在神经网络训练的任务并行过程中，需要在训练的同时准备数据。原因如下：

- 数据准备通常会消耗大量的时间。
- 数据准备通常使用与训练时使用完全不同的资源。可以在准备过程中进一步并行化为不同的阶段，因为它们也使用不同的资源，即不论并行还是串行使用的都是不同的资源。
- 数据准备通常不依赖于前一个训练步骤的结果。

总而言之，通过将数据处理和训练进行**任务并行**的方式，可以大大加快整个神经网络模型训练的时间！