homework 1

> 我们在第一次课程中已经降到，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，1.请调查这些处理器中的并行指令集，并选择其中一种进行编程练习，计算两个各包含10^6个整数的向量之和。2.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，请将问题1用多线程的方式实现，线程实现的语言不限，可以是Java，也可以是C/C++

项目结构

```
.
├── CMakeLists.txt
├── include
│   ├── common.hpp
│   └── functions.hpp
├── lib
│   ├── addNormal.cpp //不用AVX和多线程下的向量加法
│   ├── addUsingAVX.cpp //使用AVX的向量加法
│   ├── addUsingThreads.cpp //使用多线程的向量加法
│   ├── CMakeLists.txt
│   └── dataMaker.cpp //生成两个10^6的向量
├── README.md
└── src
    └── main.cpp
```

注：CMakeLists.txt中编译选项如下：

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -mavx -mavx2 -O0")
```

编译运行

```
mkdir build && cd build
cmake ..
make
./add
```

输出结果示例

运行后，build子文件夹结构如下：

```
./build
├── add //项目可执行文件
├── addNormal_output //addNormal的结果输出
├── addUsingAVX_output  //addUsingAVX的结果输出
├── addUsingThreads_output //addUsingThreads的结果输出
├── CMakeCache.txt
├── CMakeFiles
├── cmake_install.cmake
├── lib
├── Makefile
├── vector1 //makeData生成的vector1
└── vector2 //makeData生成的vector2
```

可能的输出结果：

```
gwen@gwen-G3-3579:/media/gwen/DATA/A_GW/Courses/Parallel_and_distributed_computing/hw1/build$ ./add
Make two 1*1000000 vectors which store in vector1 and vector2

Now try to add them normally, the output stored in addNormal_output
elapsed time: 0.00308347s

Now try to add them using AVX, the output stored in addUsingAVX_output
elapsed time: 0.00162792s

Now try to add them using threads, the output stored in addUsingThreads_output
elapsed time: 0.00328742s
```

分析

1 使用AVX与普通向量加法的加速比

1.1 本地测试

核心计时代码：

```
//using AVX
for(int i = 0; i < VectorSize / 8; i ++)
        ans_int[i] = _mm256_add_epi32(int_vec1[i], int_vec2[i]);
//normal
for(int i = 0; i < VectorSize; i ++)
        ans_vec[i] = v1[i] + v2[i];
```

在我的机器上测试，使用AVX的向量加法和普通的向量加法，在对两个10^6大小的向量相加时，加速比只有1.9左右，与预期的接近8有较大差距。

1.2 原因猜想

由Amdahl's Law可知，$Speedup = \frac{1}{(1-p)+\frac{p}{k}}$

计时包括了多次访存读写数据的开销，该开销占总运行时间比重较大，导致加速比相比预期减小。

1.3 实验验证

为使计时结果加速比更接近两者加法的加速比，进行一下两个操作使加法占运行时间的比重更高：

1) 对上述循环执行多次，由于多次访问同样的数据，由内存管理的知识，推测访问该数据的速度会更快；

2) 只执行加法操作，省略将结果写入数组的过程。

修改代码，进行试验：

```
//using AVX
for(int t = 0; t < 1000; t ++)
    for(int i = 0; i < N; i ++)
        _mm256_add_epi32(int_vec1[i], int_vec2[i]);
//normal
for(int t = 0; t < 1000; t ++)
    for(int i = 0; i < VectorSize; i ++)
        v1[i] + v2[i];
```

得结果如下：

```
gwen@DESKTOP-HVHN85U:/mnt/d/A_GW/Courses/Parallel_and_distributed_computing/hw1/build$ ./add
Make two 1*1000000 vectors which store in vector1 and vector2

Now try to add them normally, the output stored in addNormal_output
elapsed time: 1.47087s

Now try to add them using AVX, the output stored in addUsingAVX_output
elapsed time: 0.302819s
```

可见，加速比升至约为4.86，更接近理想的加法加速比。

2 使用多线程与普通向量加法的加速比

2.1 本地测试

在我的机器上测试，使用两个线程和普通的向量加法，在对两个10^6大小的向量相加时，速度接近，与预期的加速比约为2有较大差距。

2.2 原因猜想

猜想，这是由于相加两个规模为10^6的向量对于计算机是较小的工作量，启动多个线程的开销和线程间的调度的开销大于多线程带来的性能优化，由此影响了加速比。

2.3 实验验证

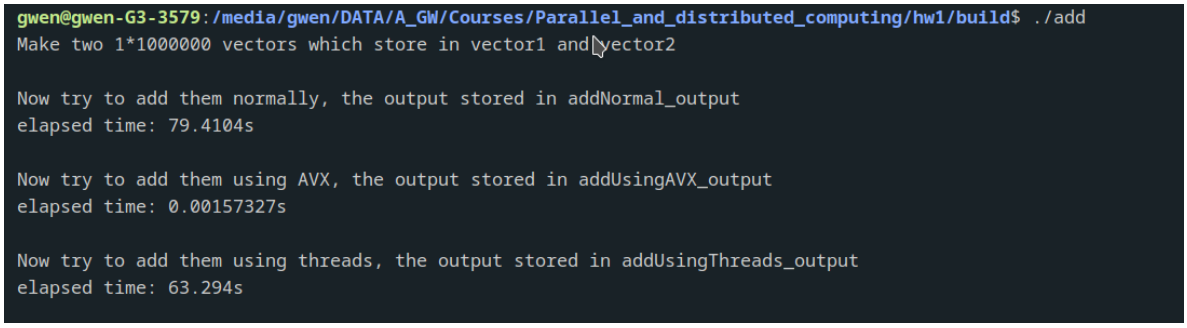
于是，将上述两个向量重复相加40000遍以增加工作量。

修改代码，进行试验：

```
//normal

for(int t = 0; t < 40000; t++)
    for(int i = 0; i < VectorSize; i ++)
        ans_vec[i] = v1[i] + v2[i];

//using two threads

void addUnit(int st, int ed){
    for(int t = 0; t < 40000; t++)
        for(int i = st; i < ed; i ++)
            ans[i] = vec1[i] + vec2[i];
}
```

```
const int len = VectorSize / 2;
for(int i = 0; i < 2; i ++){
    thread t{addUnit, i * len, (i + 1) * len};
    t.join();
}
```

得结果如下：(没有修改使用AVX的程序)



可见，加速比升至约为1.25，更接近理想的加速比。

3 结论

使用AVX和多线程均能加速向量加法，但实际加速比与理想加速比有较大差距，主要是由于工作量不足，访存开销、线程开销等开销占程序运行时间比重大。

由于“相加两个10^6的向量”为工作量小的任务，性能提升不如理想情况，但是在实验中增加工作量以后，加速比明显更接近理想的加速比，程序性能有了明显的提升。