CS219 Project 2: Computing the dot product of two vectors

蔡易霖 12312927

February 2025

1 分析与开始

这次 project 让我有点困惑，因为实现点乘和运行分析似乎并没有那么困难，而想再进行拓展却又感到束手无策，因为程序的优化对我而言是完全的未知领域。于是，我决定按部就班地先完成输入和运算的基本功能，再逐渐思考优化的事情...

2 基础功能的实现

2.1 不同数据类型的输入与储存

这个问题最矛盾的点再于在输入的时候，我无法预知将要输入的数据是什么类型的。因此，我使用了 union 数据类型来储存输入的数据。由于 union 数据类型会自动申请需要最多位的空间，也就是 int 和 double 的 64byte，所以使用 float 和 short 也就没有了什么意义。因此，我相当于自动将 float 和 short 类型的数据转化为 double 和 int，再进行运算。

这里注意到一点：题目中还让我们考虑 signed char 类型的输入。这是否意味着输入的数据可能不一定是数，还可能是一些特殊的字符？考虑到这一点，我选择当接收到的输入的字符不是数字的时候，将它根据 ASCII 码转化为一个数再进行运算。这样考虑下来，对于 signed char 的负数部分，我们仍然是转化为 int 类型进行运算；对于正数部分的可能的字符输入，我再做一个特殊转化处理。

2.2 点乘实现与时间统计

点乘的实现其实是一个比较简单的过程，无非是找到对应位置的数，并且将它们乘起来，再将这些乘积加起来。这里我考虑到的一个点是，如果相乘的两个数的数据类型不一样，应该按照 char 转 int，int 转 double 这样的向下兼容的优先级转换数据类型使得运算的各个数的数据类型相同并且不丢失过多的数据精度。

还有一个很有意思的点是，虽然兼容运算的时候有 double 我就会转化成 double 数据类型，但是如果输入的数据都是 int，但是乘积结果加起来超过了 int 的范围，它就会爆掉输出一个错误的结果。这是我不希望看到的！所以我在运算时监测它有没有超过当前数据范围，如果有超过就改用更高范围的数据范围。

然后，我还需要实现时间的统计。我需要统计的”execution time”，应当是包含了数据处理和计算的全部时间。因此，我将程序拆分字符串之前设置为起点，获得点乘结果后作为终点，获取这中间的时间作为程序的运行时间。

我注意到报告中有提到 c 语言中 clock() 函数得到的时间可能不准确，这让我感到好奇并且尝试了一下使用 time() 获得时间。在这一版程序下，我发现我在五十位向量之内的运算得到的运算时间都是相同的！

```
e0isa5@e0isa5-NbF-XX:~/c++/ysq/proj2$ ./dotproduct
Enter first vector:  1
Enter second vector: 1
Dot product: 1
Time elapsed: 0.000002 seconds
```

图 1: 一位向量乘法的时间

```
e0isa5@e0isa5-NbF-XX:~/c++/ysq/proj2$ ./dotproduct
Enter first vector:  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Enter second vector: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Dot product: 50
Time elapsed: 0.000002 seconds
```

图 2: 五十位向量乘法的时间

也就是说，time() 函数得到的时间的精度是不足够的，不足以体现出

一些范围内的数的运算的精度。于是，在 DeepSeek 的帮助下，我了解了 POSIX 中的一个高精度时间统计函数 **clock_gettime**。它能支持 POSIX 系统，包括 Linux 和 macOS 系统，并提供了纳秒级的时间精度。

在使用了使用了该函数之后，我得到的时间变得合理多了：

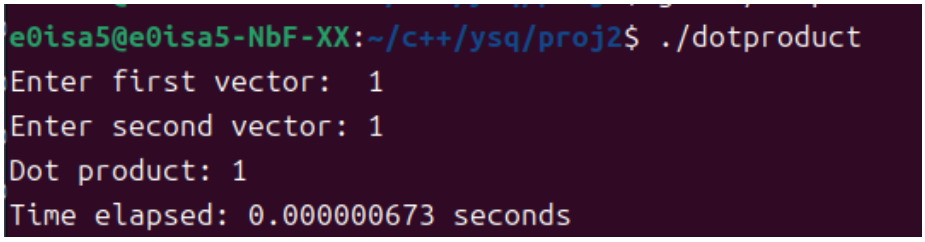

图 3: 修改后一位向量相乘

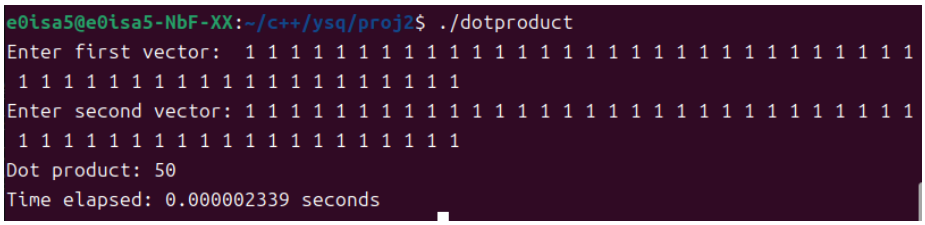

图 4: 修改后五十位向量相乘

我发现，这样的到的时间不仅随着数据增大而增大，还在小数据比 time() 得到的时间要短。在查阅资料后我发现 time() 得到的其实是系统的时间，而 clock_gettime 得到的是 cpu 的运行时间。这也就意味着，如果有其他程序占用 cpu，导致我的程序无法第一时间运算，time() 也会把等待时间记录在内，进而导致时间较长。而 clock_gettime 得到的则是程序的真正运行时间，比较准确。

有意思的是，为了启用 POSIX 的标准功能，DeepSeek 建议我使用以下宏定义：#define _POSIX_C_SOURCE 200112L。然而在使用了之后，我之前调用的 strchr 函数却出现了警告。

这让我百思不得其解（DeepSeek 也感到了困惑）。经过排查我才发现，strchr 其实也是 POSIX 标准的一部分。在我没有使用该宏定义的时候，当我调用 strchr，编译器其实自动帮我启用了高版本的 POSIX 标准。而当我定义了宏之后，使用的 POSIX 版本却不是最新的，这就导致了编译器警

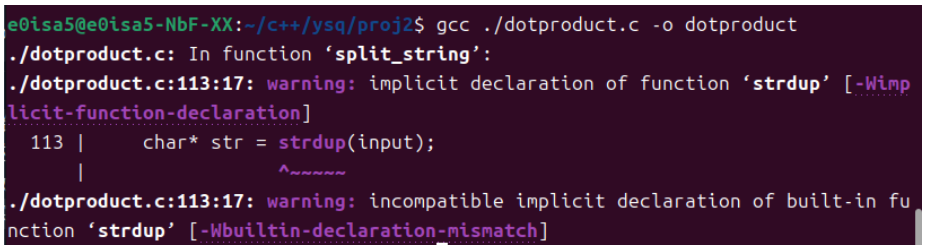

图 5: 奇怪的警告

告。因此，我把宏定义中的 POSIX 版本提高了，新的宏定义如下：#define _POSIX_C_SOURCE 200809L。这时，编译器就不会报警告了。

3 性能分析

终于来到了最让我感到恐惧的环节了... 这么说是因为在之前的时间测算的过程中，我就已经发现每一次运行的时间其实并不一样，如何通过浮动的时间测算程序运行的效率呢？我很自然的想到，可以通过多次测量取平均值的方式获得运行速度的期望值和方差，并且进行比较。因此，我编写了一个脚本，可以多次调用程序进行向量乘法并统计消耗的时间。

具体要调用多少次呢？为了探究这一点，我先针对长度 1000, 输入值范围在 $[-1000, 1000]$ 的向量进行了 100 次运算，得到的结果如下：

指标	**数值**
平均耗时	0.000009741 秒
耗时标准差	0.000004373 秒

此时，我希望我得到的平均时间的置信水平有 95%，也就是 $z = 1.96$，并且误差不超过 5%，也就是 0.0000005 秒左右。根据公式：

$$n = (\frac{zx}{E})^2$$

其中 $z = 1.96$，x 是平均值 $x = 0.000009741$，$E = 0.0000005$，带入计算可以得到 $n = 1458.07284148784$。也就是要得到这个精度，应当计算 1500 次或以上。我们就采取 1500 次计算取平均值作为运行的时间。

3.1 不同的数据类型对运行速率的影响

由于我的设计的 enum 一定需要 64byte 的空间，我将 float 和 short 分别转化为了 double 和 int，而 signed char 类型的变量本质上也是 int 类型的运算，所以这个运算实际上应该聚焦于对 int 和 double 的运行效率的计算。

于是，我对长度 1000, 数值大小 1000 以内的 int 和 double 类型的数据分别进行了 1500 次运算，得到的时间结果如下：

数据类型	平均耗时（秒）
int	0.000027511
double	0.000009604

如你所见，我发现 int 类型比 double 类型在这个数据区间慢了三倍！这是十分反直觉的！（DeepSeek 也觉得不合理！）于是我怀疑是输入数据有点小了，在这种特定区间下 double 运算更有优势，于是我把数值的范围扩大到 1000000000（接近 int 的最大值），此时程序运行状况如图：

数据类型	平均耗时（秒）
int	0.000020169
double	0.000007567

如你所见，这次的结果不但没有改变 int 和 double 的关系，甚至还缩短了程序运行的时间！数据越大算的越快？我已然是头昏眼花晕头转向了... 在仔细排查之后，我发现是因为我在运算 int 类型的数据类型时测算是否超过类型的最大值，导致每次循环的时间复杂度都增加了。我将溢出检测移除了，直接将 int 类型转化为 double 类型进行运算。以下是 1500 次 1000 长度 1000 数值的结果：

数据类型	平均耗时（秒）
int	0.000003636
double	0.000003553

这个情况下 int 和 double 的运行效率就差不多了！（而且都变快了！）以防万一，我也试了一下 1000000000 长度的输入：

数据类型	平均耗时（秒）
int	0.000003585
double	0.000003357

仍然是运行效率比小的数据更快！真的太神奇了！我将会在下一章节分析这一现象。

我还尝试了改变向量的长度，以下是长度 10000, 数据规模 1000 的运行结果:

数据类型	平均耗时（秒）
int	0.000031824
double	0.000028011

结果比较符合直觉地时间消耗增加了 10 倍左右。然而，double 类型的运行速率似乎比预期的又快了一些？非常奇怪... 我会在下一章探索这个问题。

3.2 java 与 c 的运行速率比较

这一部分的比较我认为应当不会出很大的问题（如果 java 跑得比 c 快那就天地倒转了）。我原先想直接套用 c 的脚本到 java 上，但后来发现 java 在 javac 后无法直接被之前的脚本识别，还需要修改输出格式。在调整好脚本后，我对 1500 次循环，1000 长度 1000 大小的向量运行结果如下：

数据类型	平均耗时（秒）
int	0.000475856
double	0.000454323

通过 int 类型的数据，我们可以发现 java 的运行效率至少是比 c 慢了 100 倍左右，这是一个很夸张的数字！我认为这可以反映出 c 语言的代码表现出了”significant speed advantages over Java”。

4 现有结果的理解与提速的思考

4.1 运算速度的分析

前面的部分有提到，int 类型的运算速度比 double 要慢上一些，这其实是反直觉的，DeepSeek 告诉我，int 类型的运算普遍是比 double 类型快的。然而，实际上，我的运算中的 int 类型一直都比 double 类型要慢上一些。在仔细的思考之后，我想起来原因是因为我将 int 的数据类型转化为 double 类型再进行运算的，所以 int 类型的运算不止实际上就是 double 类型的运算，还因为多了个转换的过程，所以效率比 double 还要低上一些。

同时，DeepSeek 告诉我 double 数据类型会更容易自动触发编译器的 SIMD 向量化优化。为了验证这一点，我将代码编译为汇编语言，并且成功在其中找到了 mulpd，也就是浮点 SIMD 乘法的指令。这说明编译器是有使用 SIMD 向量化来优化乘法的运行的。

而 DeepSeek 又告诉我，更大的数值很可能会除法更优的指令选择，也就是说当数值更大的时候，其更可能触发 SIMD 的优化，进而让运算的效率变得更高。

4.2 SIMD 向量化优化和编译器优化

既然编译器会自动帮我使用 SIMD 优化并且确实有效果，那我就不禁思考，为什么我不能直接手动使用 SIMD 全程优化我的代码呢？在 DeepSeek 的帮助下，我修改了我的代码，启用了 AVX 指令集，在 simd_dot_product 函数里调用了 256 位的寄存器，很底层地直接搬运数据，并且每次并行地提取四个对应位置的数并进行乘法，然后再将这四个 double 合并为一个总和。

同时，我知道了编译器的 o3 优化，我尝试用以下命令进行了编译：

```
gcc -O3 -mavx -march=native -o dotproduct dotproduct.c
```

之后，我再次运行了 1500 次 1000 位长，1000 数据大小的向量点乘，结果如下：

数据类型	平均耗时（秒）
int	0.000001598
double	0.000001418

对比之前的结果，我发现它的速度提高了大致三倍！所以使用 SIMD 向量化优化是一个有效果的优化方案！

这是合情合理的，因为 SIMD 的实际作用就是把数据拆成多条平行的线，然后并行地计算多个相同的操作。我的代码把数据拆成了四份，所以效率得到了三到四倍的优化是一件很合理的事。

4.3 改变数据的存储的想法

我了解过如果访问的内存连续，程序的运行速率会提高。所以我想到一个点：如果我将输入的数据按照需要被运算的顺序排成一列，这样读取内存的时候就能够沿着一条线从上往下读，这样就能提高读取数据的效率，进而提高程序运行的效率。于是，我设计了如下的结构体：

```
typedef struct __attribute__((aligned(32))) {
    double a;
    double b;
} Pair;

typedef struct {
    int length;
    Pair* data;
} VectorPair;
```

图 6: 数组结构体

它将输入的两个向量对应位置的变量像编麻花一样交织在了一起放到 Pair 类型的一个结构体中，并在合成的链中用一个 Pair* 的指针维护了这些变量。这样，访问的时候，它就可以顺着 Pair* 类型的 data 一路走下去，对每一个 Pair 进行乘法运算. 这看起来十分美好，于是我将其实现后测定了其对 1500 次测量 1000 长度 1000 数据大小的数据的运行速率：

数据类型	平均耗时（秒）
int	0.000001990
double	0.000001737

令人十分沮丧的是，我发现这样做的效率不如刚刚的效率。我询问了 DeepSeek 为什么会这样，在 DeepSeek 的点拨下，我意识到，如果我想要使用 SIMD 优化，我需要让输入的数据变为四行并行的形式再进行运算，而这个转换的过程可能比较耗时，消耗的时间超过了结构变换所节约的时间。想到这里，我又想到可以直接制作四行并行的结构体来储存数据。它长如下这样：

```
typedef struct __attribute__((aligned(64))) {
    __m256d a;
    __m256d b;
} Block4;

typedef struct {
    int total_pairs;
    int blocks_count;
    Block4* blocks;
    double* remain_a;
    double* remain_b;
    int remain_count;
} OptimizedVector;
```

图 7: 四行并行的数组结构体

其运行的结果如下

数据类型	**平均耗时（秒）**
int	0.000001672
double	0.000001501

令我高兴的是，这样做确实能够让效率提高，但可惜的是，其效率仍不如直接使用两个数组使用 SIMD 优化。大概是就连预处理所需要的时间都比改变结构需要的时间要多吧... 也不得不感叹一些封装好的优化的强大，明明我想出了看似很能节约时间的方法，但是最终的优化程度还是不如已有的优化。

5 结语

最终，在多次的对比下，我得到了效率最高的 c 语言计算向量点乘的方法：使用数组存储并使用 SIMD 优化。至于 Java... 既然它显著地比 c 慢，那在效率提升这一方面，与其优化它，为什么我不直接改用 c 呢?

这次 project 算是我第一次深度了解并思考程序运行的背后原理呢，想象内存的调用实在是蛮头大的... 无论如何，这次的 project 也算是完成了。在这次的 ddl 之后，我会将用到的脚本和调试用代码都放到 github 仓库上，网址是 https://github.com/10a5/CS202-Advanced-Programming，欢迎大家参观!