C Project II 报告

指导老师：于仕琪

撰写人员：12112510 李涵

第一部分：项目简述（目录）

本次项目要求计算两个向量的点积。我分别使用C++和java完成了相关程序的编写。

最终运行时间：

- **基本要求的实现：**

1. 基本向量点积运算的实现
2. 输入错误时，不会程序崩溃，而是让用户重新输入
3. 允许用户自己输入测试数据，或者选择程序自动生成指定规模的数据
4. 用户正确输入完毕后，可选择继续输入或者退出
5. 精度相关的问题

- **效率提升的方法：**

1. 函数值传递更改为引用传递
2. 多线程的并行计算（以OPNEMP作为实现方式）
3. 寄存器运算
4. GPU加速（以CUDA作为实现方式）
5. O3编译优化
6. 访存优化
7. 空间处理和回收
8. 随机数生成效率的提升
9. 提高C++的打印效率

- **测试环境：**

CPU：AMD Ryzen 7 5800H（8核）
显卡：NVDIA GEFROCE RTX 3060 Laptop GPU 6GB
内存：16GB（3200MHz）
运行系统：Windows 10
C++编译器：mingw-GCC
C++版本：C++ 8.1.0
java版本：java 11.0.15
C++的IDE：Visual Studio Code 1.76.0
java的IDE：IntelliJ IDEA 2022.3(Ultimate Edition)

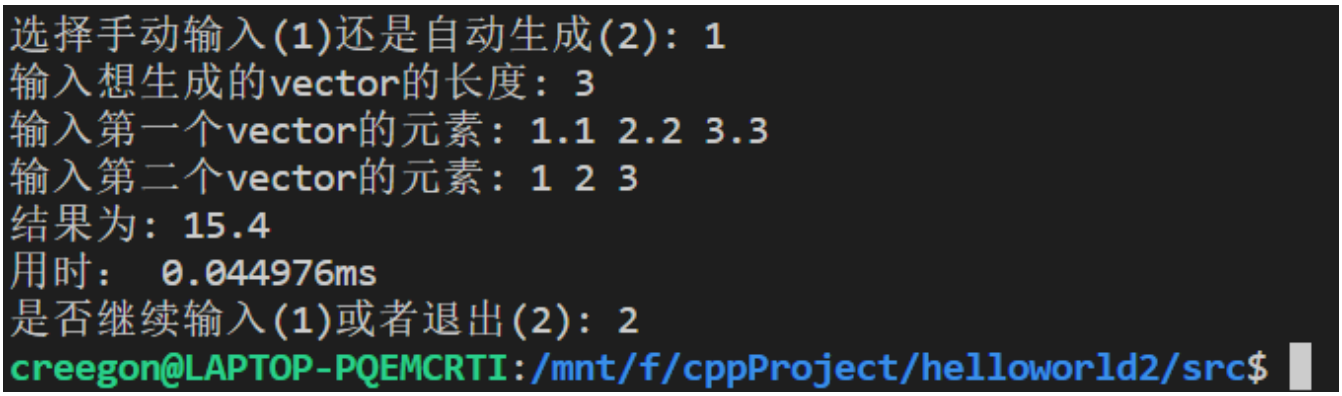
第二部分：基本要求的实现

以下操作的示例代码和截图，皆取自C++程序。

1. 基本向量点积运算的实现

运算规则：dot_product = v1[0] * v2[0] + v1[1] * v2[1] + ... + v1[n] * v2[n]; （注意：两个向量的**长度**需相同）

实际运行效果：

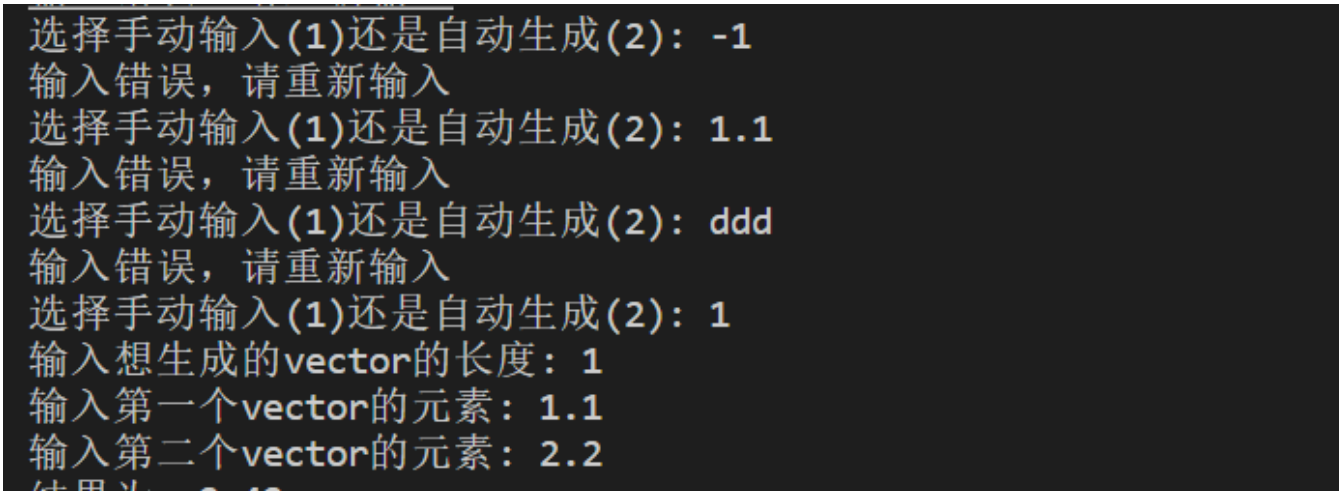


2. 输入错误时，不会程序崩溃，而是让用户重新输入

如果用户在输入长度时，输入了非正整数或者字符串（会被自动转化为0），程序会退回到起始位置，并要求用户重新输入。在输入向量元素时，如果有多个小数点或者存在其他字符，则同理。

注意：在每次输入错误后，应当先使用 `cin.clear()` 清除流状态标志，再用 `cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')` 来忽略输入缓冲区中的所有字符，从而防止可能导致的程序异常

实际运行效果：



3. 允许用户自己输入测试数据，或者选择程序自动生成指定规模的数据

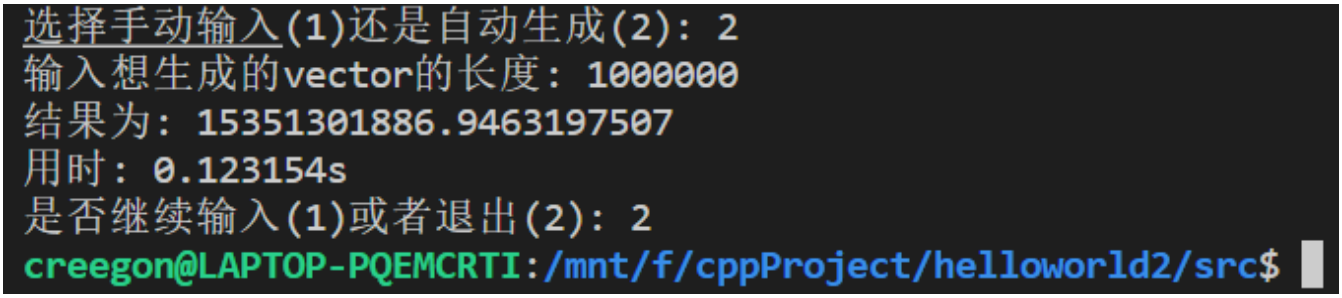
实际运行效果：

4. 用户正确输入完毕后，可选择继续输入或者退出

详见前面几张图的最后一行

5. 精度相关的问题

尽管C++内部的运算方法的精度已然很高，但为了更进一步，我试着安装了GNU高精度算术运算库（英文名：GNU Multiple Precision Arithmetic Library，详细的安装过程参见 https://blog.csdn.net/yang889999888/article/details/73442356 (https://blog.csdn.net/yang889999888/article/details/73442356)）。
在具体使用上，先将 `double` 类型的向量转化为 `mpf_t` 类型的数组，然后调用 `mpf_mul()` 和 `mpf_add()` 的方法获得结果。要注意的是，应当在运算完后，使用 `mpf_clear()` 清除掉创建的数据，从而释放掉内存空间，避免数据泄露。
实际运行效果：



但是因为使用了库后，很多优化便无法实施（……库已高度集合相关措施），故不在后续，仍然采用最原始

在计算机内部，浮点数采用二进制科学计数法进行存储。但由于计算机内存是有限的，因此只能存储一定位数的二进制数。这就导致了浮点数的精度受到限制，最终可能会出现精度误差的情况。
虽然 float 和 double 都是浮点类型，但它们的精度不同。 float 类型通常占用4个字节，精度为6-7位有效数字，表示范围为1.2E-38到3.4E+38。而 double 类型通常占用8个字节，精度为15-16位有效数字，表示范围为2.2E-308到1.8E+308。
所以在本次项目中，统一使用 double 类型。

第三部分：效率提升的方法

以下操作的示例代码和截图，皆取自C++程序。在逐个实现的同时，我同样也会附上对应的时间消耗。测试样例自动生成，数量为三千万，生成的数据范围为（-10000，10000）。

未经优化的效果：

```
选择手动输入(1)还是自动生成(2): 2
输入想生成的vector的长度: 30000000
结果为: 9.67457e+10
用时:  281.908ms
是否继续输入(1)或者退出(2): 2
creegon@LAPTOP-PQEMCRTI:/mnt/f/cppProject/helloworld2/src$
```

1. O3编译优化

1.1 原理

O3是GCC编译器的优化选项之一。该选项会开启多个优化，包括循环展开、函数内联、常量折叠等，以便优化代码执行效率。它还会采取很多向量化算法，提高代码的并行执行程度，利用现代CPU中的流水线，Cache等。
~~其实02和03才是设计者想让C用户使用的编译选项，不加优化的选项通常只会用于调试~~

1.2 代码实现

在 makeFile 文件的 CXXFLAGS 后面，加上 -O3 即可。
实际运行效果：

```
选择手动输入(1)还是自动生成(2): 2
输入想生成的vector的长度: 30000000
结果为: 3.38124e+11
用时:  182.665ms
是否继续输入(1)或者退出(2): 2
creegon@LAPTOP-PQEMCRTI:/mnt/f/cppProject/helloworld2/src$
```

2. 函数值传递更改为引用传递

2.1 原理

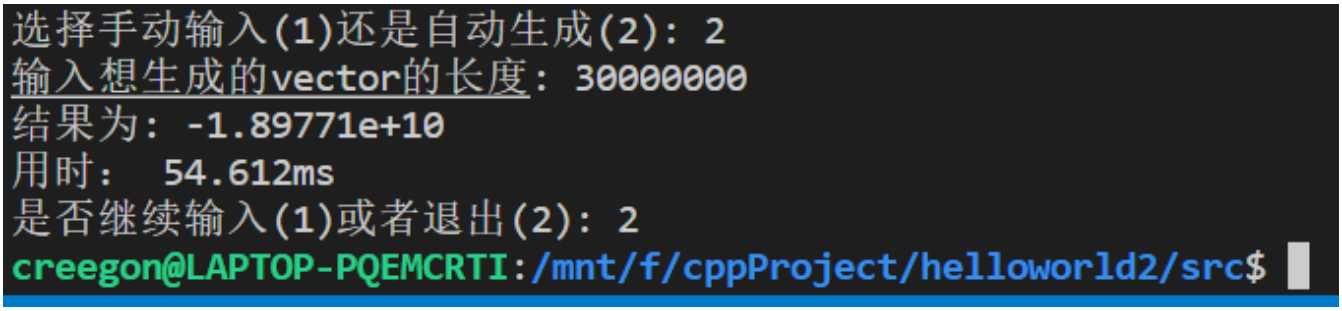
C++中，函数的参数可以使用值传递或引用传递两种方式进行传递。当函数参数使用值传递时，函数会创建该参数的本地副本，并对该副本进行操作。而当函数参数使用引用传递时，函数会直接使用传递的参数，而不会创建本地副本。如此，它能提升程序运行效率，避免内存问题。

2.2 注意事项

- 如果函数返回后，参数所引用的对象已经被销毁，那么引用就变成了悬空引用，这会导致程序出现未定义的行为。
- 避免使用未初始化的引用。
- 注意引用和指针之间的差异，避免混淆。

2.3 代码实现

在传递参数的过程中，加上 & 即可。
实际运行效果：



3. 多线程的并行计算（以OpenMP作为实现方式）

3.1 原理

多线程并行处理是指在同一进程内启动多个线程，每个线程执行不同的任务，以达到加快程序处理速度的目的。多线程并行处理可以有效利用多核CPU的优势，提高程序的运行效率。

而OpenMP是多线程的一种实现方式。OpenMP（Open Multi-Processing）是一种并行编程模型，它可以在共享内存多处理器系统上实现并行计算。它能：

- 创建一个并行计算环境，将 for 循环的迭代计算任务分配到多个线程中进行处理。
- 将 for 循环的控制变量（即循环变量）的迭代范围分区给不同的线程，使每个线程都可以计算一部分迭代任务。
- 自动处理线程之间的同步和负载均衡，确保计算任务被分配到每个线程中，并且每个线程的计算工作量相近。

3.2 注意事项

OpenMP并行化技术的使用并不能保证一定会加速程序运行，有时可能会导致性能反而变差。原因可能如下：

- 并行化代码的开销（如线程创建、同步）会消耗掉部分计算时间，导致程序反而变慢。
- 计算密度较低（即计算操作与访问内存操作之间的比例较小），所以结果不如预期。

另一种C++的多线程优化方式，是直接调用其内部自带的线程库。虽然它拥有更灵活的线程创建和管理方式，但是它需要手动管理线程并进行同步和互斥操作，操作更为复杂。（但它仍然也是可选项之一）

注意： 不能使用自带的clock来计时！因为它记录的是cpu的周期数，当多个并行线程运行时，它的滴答数也会成倍增加！应该使用openmp提供的omp_get_wtime()才能记录真正的时间！（我在这里困惑了很久，感谢 http://t.csdn.cn/lJpeY (http://t.csdn.cn/lJpeY))

3.3 代码实现

使用了 `omp_set_num_threads(4)` ,标明设置四个线程同时工作。在中间的求和的 `for` 循环上面，加上 `#pragma omp parallel for reduction(+ : sum)` ，表示 `sum` 是一个共享变量，在最后才将所有线程的 `sum` 加到一起。

实际运行效果：

```
选择手动输入(1)还是自动生成(2): 2
输入想生成的vector的长度: 30000000
结果为: 1.22095e+11
用时:  14.7815ms
是否继续输入(1)或者退出(2): 2
creegon@LAPTOP-PQEMCRTI:/mnt/f/cppProject/helloworld2/src$
```

4. GPU加速（以CUDA作为实现方式）

4.1 原理

虽然GPU和CPU都可以用于程序的加速，但GPU的设计目的是尽可能快地完成大规模的并行计算任务，例如图形渲染、物理模拟、机器学习等。GPU拥有大量的处理单元（数百个到数千个），并行执行相同的操作。此外，GPU还具有更高的内存带宽和更大的内存容量，这使得在需要大量数据处理的场景中，GPU通常比CPU更有效率。

CUDA是NVIDIA开发的并行计算平台和编程模型，可以利用GPU的大规模并行性来加速计算密集型应用程序。在数据传输时，它可以通过DMA引擎来实现，而不需要CPU的干预（能减少传输时间和CPU的负担）。它还提供了诸如内存共享、数据流水线和指令重排等算法优化技术。因为我的显卡是NVDIA的，所以在本次项目中，我选择使用CUDA进行加速。

通常来说，CUDA和OpenMP并不会同时使用，所以在此我仅仅使用CUDA加速。并且为了**普适性**（N卡并不是所有人都有），我的主体部分仍然采用OpenMP加速。

4.2 注意事项

- 因为GPU的内存访问速度相对于CPU较慢，所以需要尽量减少内存访问次数（还可采用缓存等技术来提高访问效率）。
- 因为CUDA中不支持使用C++标准库中的cout输出函数（CUDA的核函数是在GPU设备上执行的，并不支持访问主机CPU上的标准输出流）（要想在核函数中输出结果，得使用CUDA提供的printf函数），所以我将用CUDA加速的方法，单独放到了 `dotVector2.cpp` 文件下。

4.3 代码实现

具体实现流程较为复杂，在此我会详细说明：

- 通过 `__global__` 关键词，定义 `dotProductKernel` 方法为核函数，用于计算两个向量的对应元素之积。
- 使用CUDA中的 `cudaMalloc` 函数在设备端（GPU）分配一段指定大小的内存空间，再用 `cudaMemcpy` 函数将两个 `double` 类型的向量拷贝到GPU内存中。
- 设置线程池的大小为**256**（这样正好能充分利用NVIDIA GPU的SIMD指令集架构，保证GPU核心计算资源占用最佳性能），再调用 `dotProductKernel` 方法运算，最后将结果cpy回来并求和。
- 记住最后还需用 `cudaFree` 释放GPU内存，防止爆掉。

实际运行效果如下：

4.4 安装过程

因为CUDA的安装和配置过程有一定难度，在此我列出基本步骤和常见困惑。

1. 在ubuntu终端中依次输入以下命令，下载CUDA Toolkit安装程序：

```
wget
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-ubuntu2004.pin
(https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-ubuntu2004.pin)
sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget https://developer.download.nvidia.com/compute/cuda/11.4.2/local_installers/cuda-repo-ubuntu2004-11-4-local_11.4.2-470.57.02-1_amd64.deb
(https://developer.download.nvidia.com/compute/cuda/11.4.2/local_installers/cuda-repo-ubuntu2004-11-4-local_11.4.2-470.57.02-1_amd64.deb)
sudo dpkg -i cuda-repo-ubuntu2004-11-4-local_11.4.2-470.57.02-1_amd64.deb
sudo apt-key add /var/cuda-repo-ubuntu2004-11-4-local/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

2. 添加CUDA的环境变量，将以下行添加到~/.bashrc文件的末尾（将￥改为dollar符号）：

```
export PATH=￥PATH:/usr/local/cuda/bin
export LD_LIBRARY_PATH=￥LD_LIBRARY_PATH:/usr/local/cuda/lib64
```

3. 在安装完CUDA库后，仍然找不到 device_launch_parameters.h 头文件：

> 我的方法是，通过该指令 sudo find / -name "device_launch_parameters.h"，找到其位置，然后手动将其添加到 makefile 中（如下）：
> CXXFLAGS = -Wall -std=c++11 -O3 -I/usr/include

5. 寄存器运算

在程序执行完之后，操作系统并不会自动回收动态分配的内存，导致内存泄漏问题。

5.1 原理

寄存器是CPU内部的一种高速缓存器，位于CPU核心内部，其读取和写入速度非常快，可以在瞬间完成运算操作。寄存器运算通常用于一些频繁执行的简单运算，例如加减乘除、位运算等。将这些运算操作存储在寄存器中，可以避免频繁访问内存的开销，从而提高程序的运行效率。在C++中，可以通过使用关键字 register 来声明寄存器变量。例如： register int a;

5.2 注意事项

常规注意事项：

- 受限于CPU，寄存器的数量和大小都是有限的（通常只能存32位的字节）。
- 寄存器不能用于地址运算。
- 无法利用 & 符号获取到寄存器的地址。

为什么优化效果不明显：

- 在编译器优化的帮助下，在某些情况下， register 也可能被忽略。
- 而且现代处理器上，使用CPU寄存器进行计算通常对性能的提升非常有限，因为现代处理器具有广泛的指令重排和流水线技术，可以进行非常高效的计算。

5.3 代码实现

实际运行效果：

```
选择手动输入(1)还是自动生成(2): 2
输入想生成的vector的长度: 30000000
结果为: 3.3774e+09
用时:  14.5523ms
是否继续输入(1)或者退出(2): 2
creegon@LAPTOP-PQEMCRTI:/mnt/f/cppProject/helloworld2/src$
```

6. 访存优化

6.1 原理

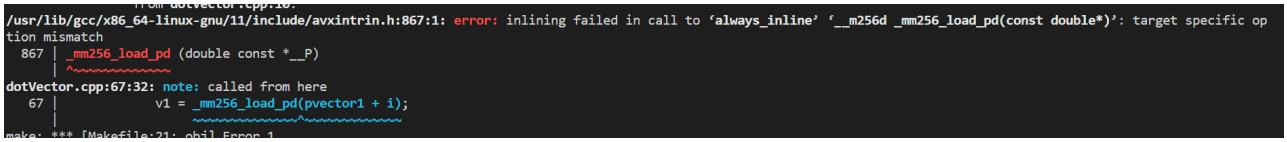
程序在执行过程中需要不断地访问内存中的数据，而内存速度较慢，频繁的内存访问会导致CPU等待内存响应，从而造成性能瓶颈。而访存优化能通过改善程序的内存访问模式，从而缓解内存瓶颈，提高程序性能。常见方式有：空间局部性优化、时间局部性优化、数据预取优化和数据对齐优化等。

在本次项目中，我使用了AVX指令集来进行这个操作。

6.2 注意事项

以下是实现过程中的一些常见的困惑（指我遇到的）：

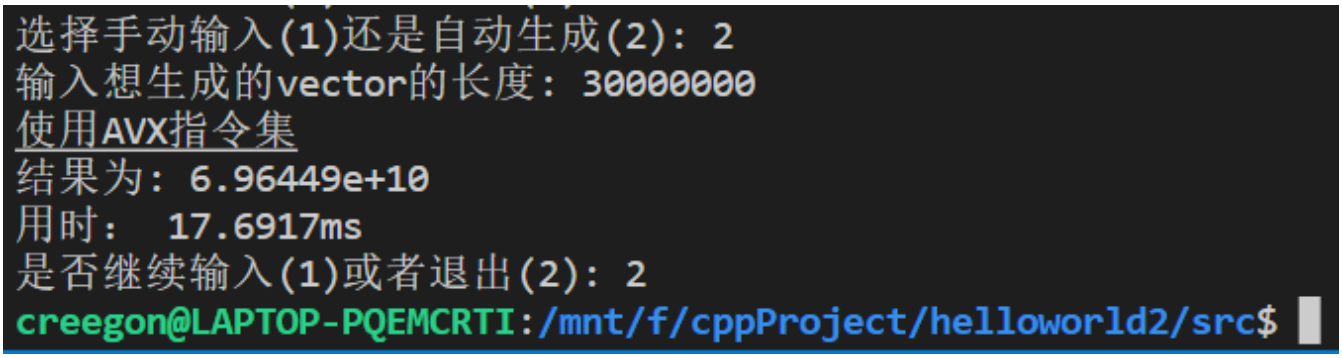
- Q：出现如图所示的报错：



- **A：在makefile中的CXXFLAG项，添加-mavx2 -mfma -msse4.1 -ffast-math（原理：提示编译器启用AVX、FMA和SSE等指令集）**
- Q：难以实现数据的对齐，从而很难去访存优化?
- **A:可以使用内存对齐优化指令 `_mm256_loadu_pd` 和 `_mm256_storeu_pd`，而不是使用 `_mm256_load_pd` 和 `_mm256_store_pd` (它们不要求内存地址对齐，可以避免在对齐时引入额外开销)**
- Q：加速效果很弱?
  其实，在第一步做的**03**优化，就已经基本包含了AVX指令集的访存优化。所以在其他条件一样的情况下，我的手动实现和直接运算的效率差距微乎其微（略微快一点）。这也提醒了我们，应该要查明每一步优化的具体原理，否则很容易会陷入困惑（悲）

6.3 代码实现

当向量的大小可以被4整除时，使用AVX指令集和OpenMP并行化循环计算向量内积，通过将向量分成多个部分，每个线程负责一个部分的计算。循环的每一步中，使用 `_mm256_loadu_pd` 函数加载两个double型数，分别存储在AVX寄存器中。接着使用 `_mm256_mul_pd` 函数对两个向量的每个元素进行相乘，再用 `_mm256_add_pd` 函数将乘积累加到一个AVX寄存器中。最后，将寄存器中的结果通过 `_mm256_storeu_pd` 函数存入线程私有**（防止数据污染）**的double数组中。在循环完成之后，使用 `#pragma omp atomic` 语句将线程私有的double数组中的元素加到公共的double类型变量sum中。
实际运行效果（电脑性能莫名下降了，前面的结果也变成了17ms-18ms左右）：



7. 空间处理和回收

7.1 原理

C++没有自带的垃圾回收机制，需要我们手动去清理。对于堆内存，我们需要使用 `new` 和 `delete` 去创建和释放它。

7.2 代码实现

实际例子如： `_aligned_free(alignsum);` （注意需要导入头文件 `#include <malloc.h>` ）和 `delete[] vec1;`

8. 随机数生成效率的提升

8.1 原理

本来我打算使用python脚本，编写一个随机向量生成器并储存到txt文件中，供C++和java程序读入。但因为文件的I/O需要耗费一定的时间，所以最终改为直接在程序内生成。提升效率的原理和前文提到的多线程并行运算一样。在C++中，我选择了 `default_random_engine` 类来创建，在java中，我选择了 `ThreadLocalRandom` 类来创建。

8.2 代码实现

详见C++中的 `generateVector()` 方法。

9. 提高C++的打印效率

9.1 原理

C++中的 cout 是标准输出流，它的输出效率可能不如直接写入文件或者使用低级I/O函数等方式快。在此用如下的方法来提高效率：

- 将 cout 和 cin 的同步关闭，因为它们默认是同步的，关闭同步后可以减少不必要的同步等待时间，从而提高效率。可以使用以下语句关闭同步：

```
std::ios_base::sync_with_stdio(false);
```

- 使用换行符 \n 代替 endl ，因为 endl 会强制刷新缓冲区，从而降低效率。如果需要刷新缓冲区，可以手动调用 flush() 函数。
- 使用带缓冲的 I/O 函数，例如 fwrite() 和 fputs() 等，因为它们可以减少 I/O 操作的次数，从而提高效率。

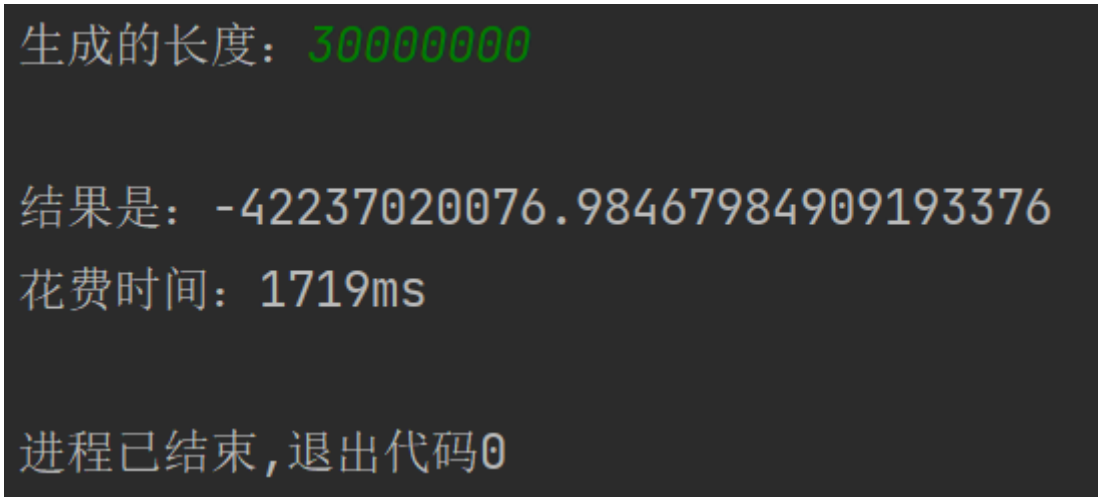
9.2 代码实现

详见每一个输出的语句和刷新流语句

第四部分：与其他语言的程序的比较

在此比较的便是java程序。为精简，java程序只有程序自动生成大量随机数的选项（采用并行流的方式生成）。

经过多次测试后，在生成同样数据规模（三千万）的情况下，java程序平均用时在1700ms左右（下图为示例）：



对结果的分析：
C++程序显著快于java程序，个人猜测原因可能如下：

1. C++是静态类型语言，而 Java 是动态类型语言。C++编译时就确定了每个变量的类型，因此在运行时不需要进行类型检查和转换，可以减少运行时开销。
2. C++有更好的内存管理控制，可以手动分配和释放内存。而 Java 的垃圾回收机制会造成一定的开销。

3. C++编译器可以进行更高效的优化，例如函数内联、循环展开、指令重排等。
4. C++中，我们可以手动进行一些代码优化，例如使用位运算、手动内存池等。

第五部分：总结

在本次报告中，我收获良多~~（被迫）~~，总结如下：

- 知道了像gmp库这样高效强大的C++大数计算库
- 对-O1,-O2,-O3,-Os,-Ofast等编译优化方式有了一定的认知，大致明白了其工作原理
- 对地址引用的所节省的时间成本（不用再完整赋值）有了一定的认识
- 知道了CPU和GPU的并行流计算的基本使用
- 对C++程序计时的原理有了新的认知
- 对如何使用指令集来访存加速略有收获
- 垃圾回收，调用随机数库，加快打印流 and so on...
- 对C++相对于java的强大的主观能动性（误）有了直观的认知

第六部分：致谢

在报告的最后，我想对所有帮助过我完成这份报告的人/网站/AI表达谢意。
在此感谢：

- chatgpt plus和new bing
- 于老师的B站中文网课
- 学长的报告（仅有小部分借鉴！！无抄袭！！）和几位已经上了C++课程的同学
- CSDN和stackoverflow和github的一些大神