Contents

1 Requirement Analysis

The requirement of this project is to calculate the time consumed by performing dot multiplication operations on two vectors in C++ and Java and provide some reasonable explanations of the differences in time consumption of the computation between the languages. Its purpose is to help us deepen our understanding of both C++ and Java.

First of all, we should establish a preliminary cognition of dot product computation. For instance, the impact of the value range of each element of a vector, the impact of the length of a vector, and the size of the data volume before crossing the data boundary. All of these can help us design more practical experiments to compare the differences between C++ and Java.
Significantly, we need to consider the experiment of dot product computation of two vectors from different perspectives. This project compares C++ and Java from different data types, different data volumes, different optimization effects, and different memory overruns, which can deepen our understanding of their compilation and operation mechanisms, memory management, optimization mechanisms, and some other features.

2 Code Implementation

According to the requirements of the project, our code needs to implement dot product computation and calculate the time it takes. At the same time, in order to make the speed comparison between C++ and Java in dot multiplication operations universal and reliable, our code needs to achieve the random generation of different types of data and transfer the random numbers generated by C++ to Java. Finally, there are some code implementations for the convenience of data collecting.
Because C++ vector and Java ArrayList both have useful functions such as returning the length of an array and dynamically storing data, this project uses these two objects to store data in C++ and Java respectively.

2.1 Operation of dot product

The dot multiplication functions (methods) are written respectively in the class called DotProduct in C++ and Java.

```
//C++
class DotProduct
//Java
public class DotProduct
```

The implementation of the dot product is to multiply the elements of each of the two vectors at the same position, and then add the results of the multiplication together.

```
//C++
long long dot_product_int (const vector<int> & num1, const vector<int> & num2)
{
  if(num1.size()!=num2.size())
  {
    cout << "The size of the vectors are not fit!" << endl;
    return 0;
  }
  long long result=0;
  for(int i=0; i<num1.size();i++)
  result += num1[i] * num2[i];
  return result;
}
//dot_product_long, dot_product_float, dot_product_double are the same
//Java
public static long DotProductInteger(ArrayList<Integer> num1, ArrayList<Integer> num2)
{
    if(num1.size()!=num2.size())
    {
        System.out.println("The size of the vectors are not fit!");
        return 0;
    }
    long result=0;
    for(int i=0;i<num1.size();i++)
    result += num1.get(i) * num2.get(i);
    return result;
}
//DotProductLong, DotProductFloat, DotProductDouble are the same
```

2.2 Calculation of time

In C++, we call the function clock in the header file named time. h to record the start and end times of the computation, and then calculate the time spent by making a subtraction. (Precision in milliseconds)

```
#include <time.h>
clock_t int_start, int_finish;
int_start=clock();
//dot product operation
int_finish=clock();
```

In Java, we directly call the method nanoTime in the System to record the start and end times of the computation and calculate the time spent by performing a subtraction. (Precision in nanoseconds)

```
long int_start, int_finish;
int_start= System.nanoTime();
//dot product operation
int_finish= System.nanoTime();
```

2.3 Generate random data of different data types

The functions (methods) that generate random data are written in the class called CreateRandomNum in C++ and Java.

```
//C++
class CreateRandomNum
//Java
public class CreateRandomNum
```

In C++, we obtain random numbers by calculating formulas and obtaining the seeds of random numbers through the system clock. The function srand for obtaining random number seeds should be placed in the main function to generate multiple different random number arrays simultaneously.

```
//the calculation formulas for random numbers
#define randomInt(a,b)       (rand()% (b - a) + (a) )          //get a random integer of [a, b)
#define randomDouble(RAND_MAX) (rand() / double(RAND_MAX))     //get a random float between 0 and 1
#define randomFloat(RAND_MAX) (rand() / float(RAND_MAX))       //get a random double between 0 and 1

srand((int)time(0));//Using the system clock to generate different random number seeds
```

We use a loop to generate random numbers and use the push-back function to continuously add the number to the end of the vector. (the time of the loops is the length of the array) The function names are respectively called create-rand-int, create-rand-float, and create-rand-double.

```
vector<int> create_rand_int(int a, int b,int num)//get a random integer array of [a,b)
{
  vector<int> res;
  if(num<=0)
  {
    cout << "The input of the number is wrong" << endl;
    return res;
  }
  for(int i = 1 ; i <= num; i++)
  res.push_back(randomInt(a,b));
  return res;
}
vector<float> create_rand_float(int a, int b,int num)//get a random float array of (a,b+1)

{
  vector<float> res;
  if(num<=0)
```

```
  {
    cout << "The input of the number is wrong" << endl;
    return res;
  }
  for(int i=1; i<= num; i++)
  res.push_back((randomFloat(RAND_MAX) + randomInt(a,b)));
  return res;
}
//The method for generating random double arrays is similar to create_rand_float
```

In Java, the methods nextInt, nextFloat, and nextDouble in the class called Random are used to randomly generate data. And a loop is used to generate random numbers and continuously add them to the end of the ArrayList. (the time of loops is the length of the array) The function names are respectively called create-rand-intcreate-rand-float, and create-rand-double.

```
import java.util.Random;
public static ArrayList<Integer> create_rand_int(int a,int b,int num)//get a random integer array of [a,b)
    {
        if(num<=0)
        {
            System.out.println("Wrong input of the number");
            return null;
        }
        ArrayList<Integer> res = new ArrayList<Integer>();
        Random temp = new Random();
        for(int i=1;i<=num;i++)
        res.add(temp.nextInt(b-a)+a);
        return res;
    }
    public static ArrayList<Float> create_rand_float(int a, int b, int num)//get a random float array of (a,b+1)
    {
        if(num<=0)
        {
            System.out.println("Wrong input of the number");
            return null;
        }
        ArrayList<Float> res= new ArrayList<>();
        Random temp= new Random();
        for(int i=0;i<num;i++)
        res.add(temp.nextFloat()+temp.nextInt(b-a)+a);
        return res;
    }
//The method for generating random double arrays is similar to create_rand_float
```

2.4 Data transfer (from C++to Java)

To make the comparison between C++ and Java meaningful, we need to ensure that the data used for dot multiplication is the same. In this project, we store randomly generated data in C++ into a bin file, and then read the data in the bin file in Java.
In C++, we use the function fwrite in the header file named stdio.h to write data. The entire process of writing data is written in functions named SaveDataInt, SaveDataFloat, and SaveDataDouble. Here, we use the pass-by interference method to pass parameters, because the data volume of the vector may become large. If we use pass by value method, then a large amount of data will need to be copied, which is not efficient and takes too much space. The data is stored in a binary file with a filename suffix of.bin.

```
void SaveDataInt(const vector<int> &num, const string fileName)//The suffix for fileName is .bin
{
  FILE *fp = fopen( fileName.data() , "wb" );
  if( fp )
  {
    fwrite( &num[0], num.size() * sizeof(int), 1, fp );
    fclose( fp );
    fp = NULL;
  }
```

```
  else
  cout << "The file is not available" << endl;
}
//SaveDataFloat and SaveDataDouble are similar to SaveDataInt
```

The methods for Java to read data are written in a class called ReadFileFromC.

```
public class ReadFileFromC
```

There are two data storage sequences for data in binary files. One is called Big-Endian, which refers to storing the high bits of data in the low bits of the memory. The other is called Little-Endian, which is stored in the opposite order. For C++, the processing of data depends on the CPU it runs on (typically Little-endian), while for Java, the order of Big-Endian is used uniformly for storage. Therefore, when Java reads a bin file, it needs to switch the order of the bytes before and after.
First, we open the file through FileInputStream and FileChannel, then map the file to memory through MappedByteBuffer, adjust the byte order of the buffer to small-endian order, and then use the while loop to add all data to the end of the ArrayList through add method.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteOrder;
import java.nio.channels.FileChannel;
import java.nio.MappedByteBuffer;
public static void filereadInt (ArrayList<Integer> num, String filename)//The suffix for fileName is .bin
{
    File file= new File(filename);
    if(!file.exists())
    {
        System.out.println("file not exist");
        return;
    }
    try
    {
        //Open File
        FileInputStream fis = new FileInputStream(filename);
        FileChannel fileChannel = fis.getChannel();
        //Map files to memory
        MappedByteBuffer buffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, 0, fileChannel.size());
        buffer.order(ByteOrder.LITTLE_ENDIAN); //Set the byte order of the buffer to small end byte order
        //Read Data
        while (buffer.hasRemaining()) {
            num.add( buffer.getInt());
        }
        //Close File
        fileChannel.close();
        fis.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
//fileReadFloat and fileReadDouble are similar to fileReadInt
```

2.5 Data output

When conducting a large number of experiments to collect a large amount of data, we can first write the data to a CSV file, which is more convenient for us to process the data directly in the CSV file or copy and paste the data into our Excel file to process them.
In C++, we use the fstream class in the library to implement this function.

```
fstream csvFile("IntTimeCostFromCpp.csv", ios::out);
// Check if file is open
```

```
if (!csvFile.is_open()) {
    cout << "Error: Unable to open file." << endl;
}
else
{
    //write in a row
    csvFile << "data" << "," << "data";
    //move to the next row
    csvFile << endl;
}
// Close CSV file
csvFile.close();
```

In Java, we use the FileWriter class to implement this function.

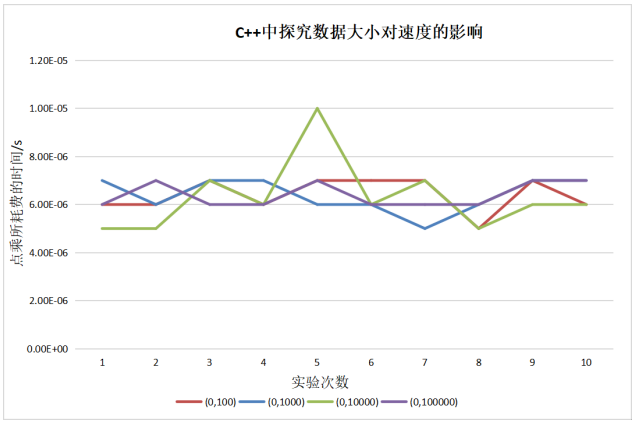
```
import java.io.FileWriter;
import java.io.IOException;

String csvFile = "FileName";//The suffix for fileName is .csv
String csvSplitBy = ",";
try (FileWriter fw = new FileWriter(csvFile))
{
    //write string in a row
    fw.append("data").append(csvSplitBy).append("data");
    //write data in a row
    fw.append(String.valueOf(data));
    //move to the next row
    fw.append("\n");
    // Close CSV file
    fw.close();
} catch (IOException e)
{
    e.printStackTrace();
}
```

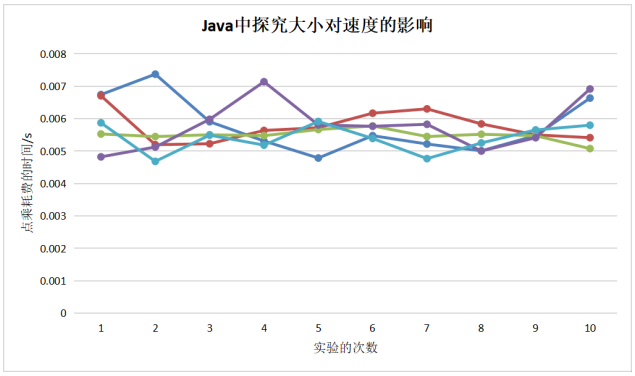
3 Experiments & Analysis

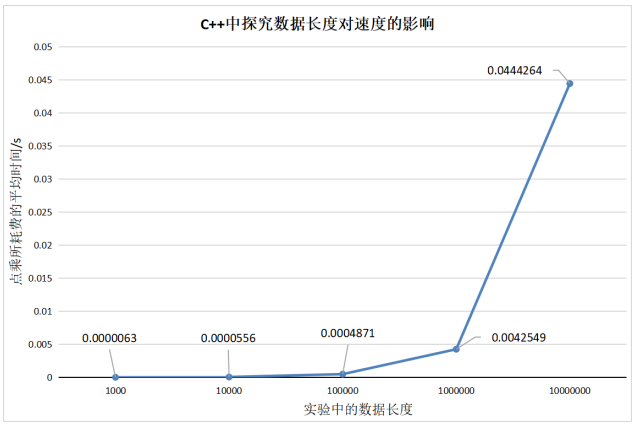
3.1 Preliminary cognition of data volume

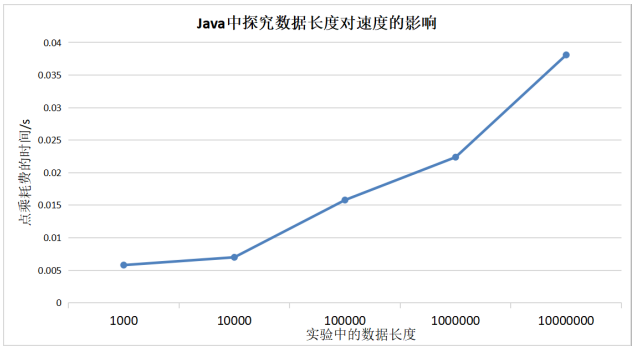
3.1.1 Experiments and Phenomena

We conducted two sets of experiments in total. The first group was to control the length of the data to 1000 and set its data to range (0,100), (0,1000), (0,10000), (0,100000), (0,1000000), (0,10000000), and (0,100000000) and perform each operation ten times. The second group controls the data range to (0,100), the data lengths of 1000, 10000, 10000, 10000, 1000000, 1000000, and 10000000 and perform ten times and calculate the average value. (Data type is int)
The results are as follows.









We will find that in C++ and Java, when the data range increases tenfold, its time does not significantly increase. As the amount of data increases tenfold, its time length increases apparently.

3.1.2 Cause analysis

C++ and Java programs typically use a hierarchy of cache levels, with smaller and faster caches closer to the processor and larger but slower caches farther away. When the data needed for an operation is already present in a cache, it can be accessed more quickly than if it needs to be retrieved from a slower level of the memory hierarchy. Thus, as the length of the vectors increases, there will be more data stored in the large caches farther away from the processor, and the time of doing the operation will increase apparently compare to the time increase due to multiplying larger numbers. Moreover, it will also cost more time to access the memory to retrieve the data for each element of the vectors.

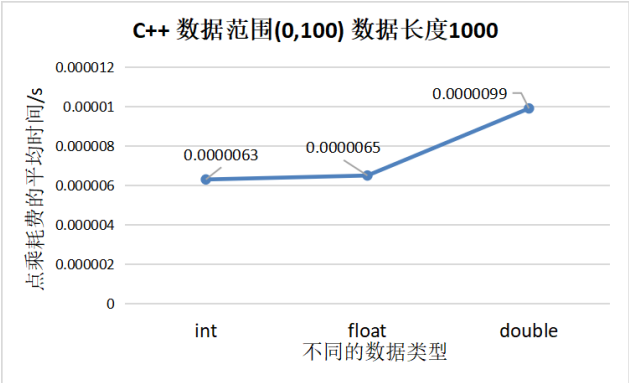
3.1.3 Conclusion

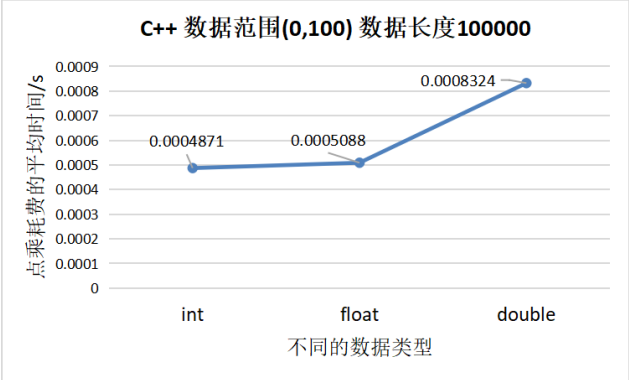
From the above experiments, we know that the impact of the length of data on the dot multiplication speed is greater than the impact of the size of the data amount on the dot multiplication speed. Therefore, in future experiments, we can use the length of data as an important indicator to measure the amount of data and control the data range to a relatively small range to prevent data from exceeding the bounds.

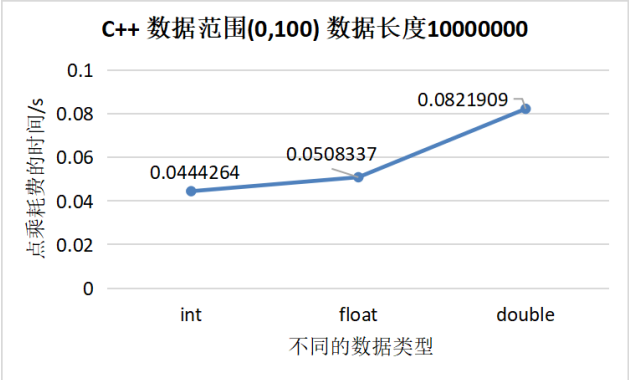
3.2 Preliminary cognition of dot product computation for different data types

3.2.1 Experiments and Phenomena

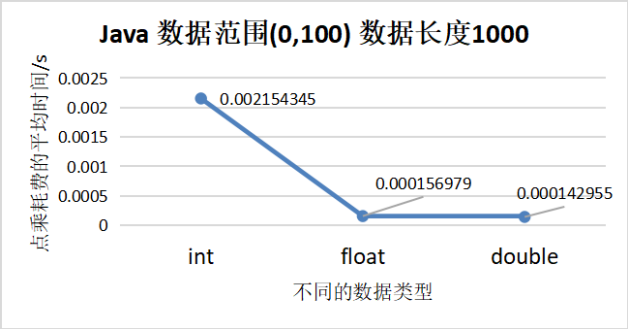
We randomly generate ten sets of random numbers of integer type with a data range of (0,100) and a data length of 1000. We store these ten sets of random numbers in the form of int, float, and double. Then calculate the time spent by their dot product computation and calculate the average time. Change the data range to 100000 and 10000000 and repeat the above experiment. The above experiments will be conducted in C++ and Java respectively.
The C++ data results in the line graph is drawn as follows.

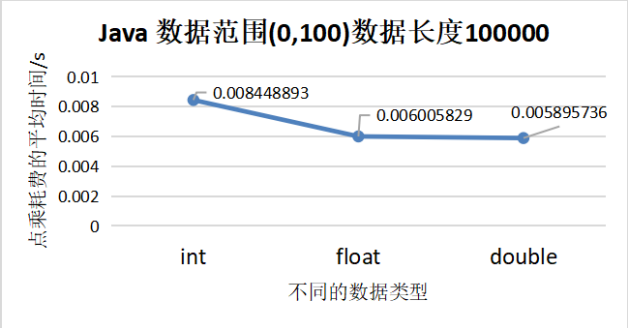


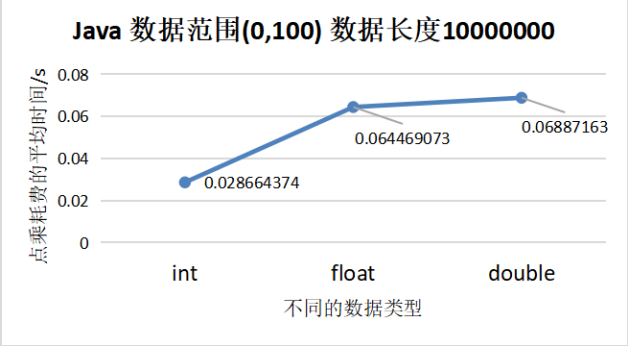




From these three line charts, we can see that regardless of the size of the data volume, int and float calculate the same set of data almost at the same speed, while double takes about twice as long as the two.
The Java data results in the line graph is drawn as follows.







The first image shows that int takes significantly longer than float and double, with a time difference of more than ten times. Float and double take approximately the same amount of time. In the second figure, int still takes longer than float and double, but their distances are significantly shorter, with a difference of less than twice. Float and double take approximately the same amount of time, but the distance between float and double is decreasing. In the third figure, int will consume less time than float, and float will also consume less time than double.
In summary, as the amount of data increases, calculating the same set of data using int will increase the time slower than using float, and the time of float will also increase slower than that of double.

3.2.2 Cause analysis

For C++, the reason why int consumes nearly as much time as float, and double consumes twice as much time as int and float is because of the data type size and the hardware architecture of the computer.
The size of int is usually 4 bytes, while float and double are usually 4 bytes and 8 bytes. When performing vector dot multiplication operations, the data has to be loaded from memory into the CPU registers. The CPU registers are temporary storage areas within the CPU that are used for the fast processing of data. Since the size of int and float is smaller than double, more int and float values can fit into a single CPU register.
When the CPU performs operations on data that is loaded into registers, it can perform multiple operations on the data in parallel. It allows for faster processing of data. However, when the data is larger than the size of a CPU register, the CPU has to load the data from memory multiple times, which slows down the processing.
Therefore, double values are approximately twice slower to process because they are twice the size of int and float, and require more memory access and processing time.
For Java, the reason for the phenomenon is due to the way modern computer architectures handle memory and data processing.

The reason why when the length of a vector is relatively small, the time taken for int to complete the dot multiplication of a vector is longer than the time float takes, and the time float takes is also longer than the time double takes is due to Java has some advanced processing features like SIMD instructions for float and double.
The reason why when the length of a vector is relatively large, the time taken for int to complete the dot multiplication of a vector is shorter than the time float takes, and the time float takes is also shorter than the time double takes is due to the time cost in accessing the memory where the vector data is stored. As we mentioned about CPU in the previous paragraph, since int is a smaller data type than float and double, more int values can fit into a single memory cache line, which means that fewer memory accesses are required to process the same amount of data. And when the length of the vectors becomes larger, the effect of more memory accesses is more significant than the benefits some advanced processing features in Java can bring.

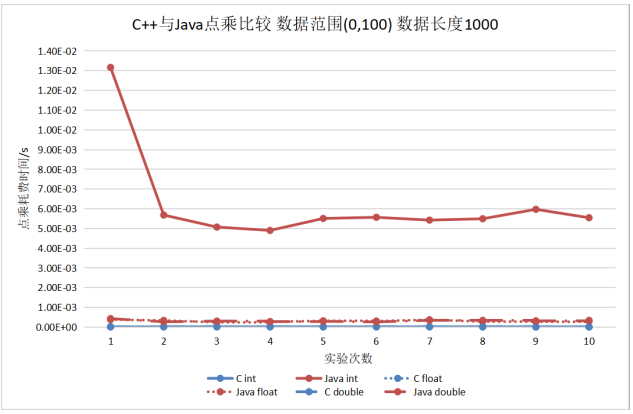
3.2.3 Conclusion

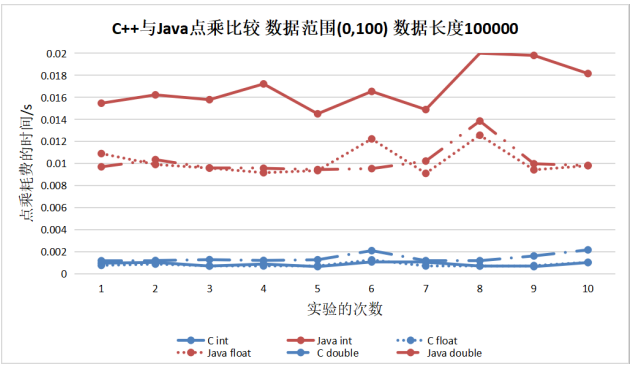
Int should be faster than the float and float should be faster than the double because the int has less memory access when dealing with the same amount of data. The phenomenon in C++ is corresponding to normal thought. However, due to the advanced processing features Java contained, the speed of processing dot product in the datatype of double and float is faster than in the type int. But, as the length of the vectors increases, the benefit will not cover the effect of more memory accessing bring.

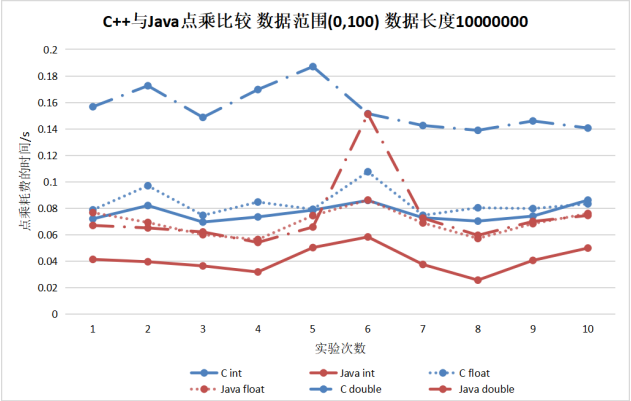
3.3 Compare the speed of dot product between C++and Java under different data volume

3.3.1 Experiments and Phenomena

We conducted three sets of experiments, each with a data range of (0,10) and a data length of 1000, 100000, 100000000. We randomly generated data of type int, float, and double (different) in C++, passed the data to Java, performed dot multiplication, and counted the time.
The results are drawn in line graphs as follows.







Through the line graphs, we can see that when the data length is 1000 and 100000, it will take a shorter time for all three data types in C++ than for Java, but when the data length is increased to 10000000, it will take a longer time for all three data types to accomplish the operation in C++ than for Java.

3.3.2 Cause analysis

The performance difference between C++ and Java in dot multiplication and vector length calculations is mainly due to differences in their different code compilation and execution mechanism, memory management, Safety detection, and optimization strategies.
Compilation and execution mechanism: The code in C++ is compiled into machine code in binary form and runs directly on the CPU, while the code in Java is compiled into Java bytecode, which is converted into machine code and runs on a virtual machine in Java. In theory, if other factors are not considered, Java will be one step slower in compilation and execution than C++, so Java will be slower than C++. Moreover, Java programs need to load class bytes from the network and then execute them, which is also the reason why Java runs slowly.
Memory management: C++ allows for low-level memory manipulation and provides more control over memory allocation and deallocation. This means that C++ can efficiently manage the memory used to store the vectors and optimize the computation of the dot product and vector length. While the garbage collector in Java has some time cost. Thus, when working with small vectors, C++ can be faster than Java.
Safety detection: Java has a lot of detections to ensure the program is running correctly. For instance, runtime overflow detection, runtime reference detection, and runtime type detection. These detections may increase the time of running a program. While C++ dose does not have such functions.
Optimization strategies: When working with large vectors, Java can be faster than C++ due to its automatic memory management and advanced optimization techniques. Java uses a garbage collector to manage memory, which can reduce the overhead of memory allocation and deallocation, especially for large arrays or vectors. Additionally, Java's Just-In-Time (JIT) compiler can optimize the code at runtime, which can result in better performance for computationally-intensive tasks like dot multiplication and vector length calculations.

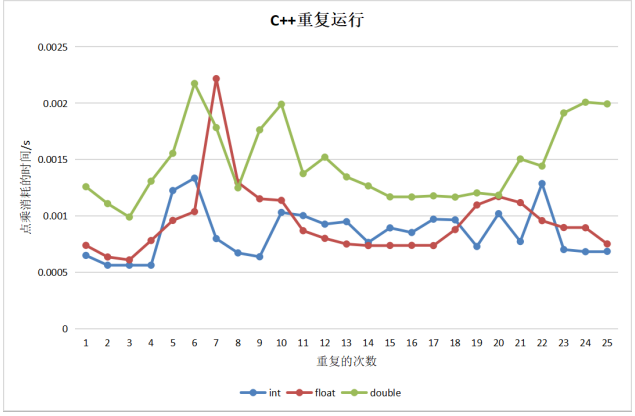
3.3.3 Conclusion

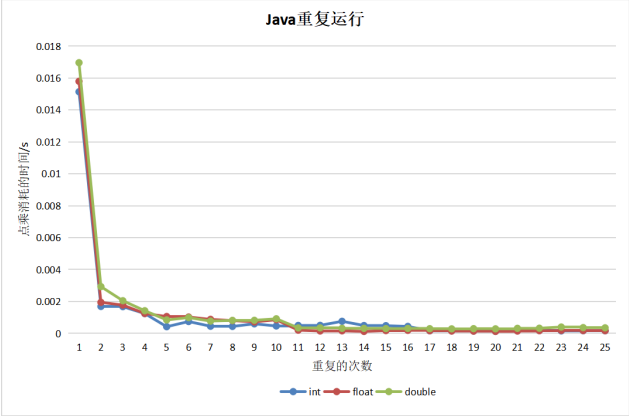
In summary, the speed of C++ and Java for dot multiplication depends on the size of the vectors being used. For small vectors, C++ may be faster due to its low-level memory manipulation capabilities and fewer step in compilation and execution, while for large vectors, Java may be faster due to its automatic memory management and advanced optimization techniques.

3.4 Comparison of speed changes before and after multiple dot product operations in a single run in C++and Java

3.4.1 Experiments and Phenomena

In C++ and Java, 25 dot product operations are repeated at one run and counted the time separately. Set the data range to (0100) and the data length 100000.
The running results of C++ and Java are as follows.





From the line graph, it can be seen that for C++, the speed of the three data types does not change much before and after the repeated operations. For Java, the time spent on the three data types will become one-tenth of the time spent on the first run when the program runs the second time, and the time spent on the tenth run will become one-hundredth of the time spent on the first run, and its speed will not be significantly improved in subsequent runs.

3.4.2 Cause analysis

The reason why the time consumed for dot multiplication decreases as we try it many times is due to a technique called ”Just-In-Time” (JIT) compilation used by the Java Virtual Machine (JVM).
When the JVM executes a Java program, it first interprets the bytecode and then compiles it into native machine code using the JIT compiler. The first time the dot multiplication is performed, the JVM will compile the code for the operation, which takes some time. However, when the operation is performed repeatedly, the JVM may decide to keep the compiled code in memory, reducing the time taken for subsequent executions of the operation. Because the compiled code is faster than interpreting the bytecode the time cost to accomplish the operation decreases.
Additionally, the JVM may optimize the code by analyzing the data being operated on and the pattern of usage. For example, if the two vectors are the same length and always contain the same values, the JVM may cache the result of the dot product to avoid performing the calculation again.

3.4.3 Conclusion

Overall, the combination of JIT compilation and code optimization in Java allows the JVM to improve the performance of frequently executed code. While C++ doesn’t contain those optimizations.

3.5 Comparison between C++’s - O3 optimization and Java’s optimization

3.5.1 Experiments and Phenomena

In C++, two vectors with a data range (0,100) and a data length of 100000 are randomly generated in the datatype of int, float, and double, respectively. Then, they are compiled using the - O3 instruction, and doing 25 dot product operations in a single run. Compare the average value with the average value without optimization in Experiment 3.4, and calculate the optimization ratio. Also, calculate the optimization ratio

based on the results of Java running in Experiment 3.4 (using the results of the last limit optimization and the first run time).
The results are shown in the figure.

C++的-O3与Java自带优化比较			
	int	float	double
03优化	0.0000436	0.0000978	0.00094188
无优化	0.00085084	0.00094932	0.00146576
优化比值	195.146789	9.706748466	1.556206735
java优化	1.62E-04	1.68E-04	3.47E-04
java无优化	0.015142814	0.015795841	0.016965193
优化比值	93.63136872	94.06709703	48.93165798

We will find that for int-type data, the optimization effect of the - O3 compiler instruction provided by C++ is better than that provided by Java. For float and double types, the optimization effect is not as good as that provided by Java.

3.5.2 Cause analysis

The possible reason why the optimization effect of the - O3 compiler instruction is better than that of Java for int-type data is that C++ allows for more low-level control over memory management, which can lead to more efficient use of CPU cache and better performance.
The possible reason why the optimization effect of the -O3 compiler instruction is worse than that of Java for float and double type data is that the compiler may not be able to automatically vectorize the code, which is essential for optimizing the dot product computation of two double or float vectors.

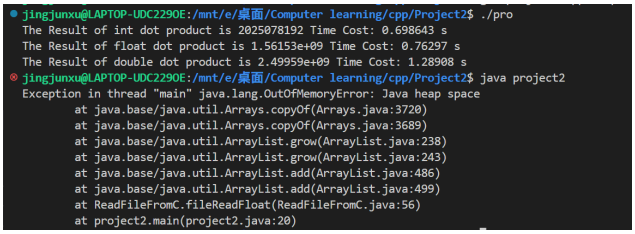
3.5.3 Conclusion

Overall, the exact reasons for the observed phenomenon will depend on the specific implementation of C++.

3.6 Java out of bounds

3.6.1 Experiments and Phenomena

We randomly generate data of int, float, and double with a data range of (0,100) and a data length of 100000000 on C++and store them in vector, and can successfully perform dot product operations. However, when we passed this set of data to Java, it experienced an out-of-memory error in heap space when writing the data into ArrayList.



3.6.2 Cause analysis

There are two possible reasons for this phenomenon.
One is likely due to the fact that the C++ program is able to manually control the amount of memory allocated to the container, while the Java program relies on the garbage collector and the JVM to manage memory automatically. Maybe the garbage collector was not able to free up enough memory to accommodate the new data due to some reason.
The other is that java objects have a fixed layout, while C++ objects can have variable layouts due to the use of virtual functions and inheritance. Thus, it is likely that ArrayList takes up more memory when storing the same set of data than vector does as it must have an object header which causes the out-of-memory error in heap space.

3.6.3 Conclusion

The memory allocation and object in C++ and Java are different.

4 Difficulties & Solutions

4.1 Difficulties in transferring data from C++ to Java

In the beginning, we chose to transfer data through the JNI (Java Native Interface) system. Spending a lot of time understanding its compilation principles, implementing methods for creating Java objects in C++, and obtaining and calling Java objects in C++. Unfortunately, the JNIEXPORT in JNI is not compatible with the main function in C++.
Then, we chose to write all the data in a binary file through C++ and open the file in Java to get the data. Because of the different ways of writing and reading a binary file in C++ and Java, we need to change the data from small-endian to big-endian in Java to read the data correctly. At first, we change the bytes one element by one element by formulas, but it is too slow to deal with a large amount of data. So, we learned to map an entire file to memory and use the MappedByteBuffer class in Java to operate, which is much faster.

4.2 Difficulties in collecting data

As we need to conduct a large number of experiments and collect a large amount of data. It is very difficult for us to collect the data by copying and pasting the results printing on the screen. Thus, in C++ we use the fstream class and in Java, we use the FileWriter class to write the data into a CSV file to collect the data.

5 Conclusion

Based on the experiments conducted in this project, we have found that although C++ and Java are similar in the ways of the mechanisms in computing, they differ in many aspects such as Compilation and execution of code, Memory management, Optimization, and Object structure.

5.1 Compilation and execution

C++ is a compiled language. The source code is first compiled into machine code and then executed in the CPU. Java, on the other hand, is both compiled and interpreted. The Java compiler converts the source code into bytecode, then the bytecode is translated into machine code in Java Virtual Machine (JVM) and executed in CPU. As one more step Java is used to execute, it is a possible reason why Java is slower than C++ in some situations. More specifically, Java compiles through a dynamic compiler (JIT), while C++ complies through a static compiler. In Java, the JIT compiler can bring optimizations and safety in running the program but take more time to run the program simultaneously.

5.2 Safety Detection

Java has a lot of detections to ensure the program is running correctly. For instance, runtime overflow detection, runtime reference detection, and runtime type detection. These detections may increase the time of running a program. While C++ dose does not have such functions.

5.3 Optimization

Java can optimize the code through the use of the Just-In-Time (JIT) compiler. While C++ does not have such a compiler.

5.4 Memory Management

C++ provides the programmer with complete control over memory management, we can allocate and deallocate memory manually. While, Java provides automatic memory management through a garbage collector, which frees the programmer from the burden of managing memory manually. C++ gives the programmer full control over how memory is used, which may lead to highly optimized programs. But we should be careful to avoid memory leaks and other issues that can arise from the incorrect use of memory. While Java is much safer.

5.5 Object structure

Java objects have a fixed layout, while C++ objects can have variable layouts due to the use of virtual functions and inheritance. In some ways, then a similar object in Java may take up more space as there must be a space for the object header, while in C++, it does not have to have that header.