STL Allocator with Memory Pool

Object Oriented Programming

Final Project

2nd August 2017

Content

1 Introduction

1.1 Background

Up to now, as for allocating memory space, I believe, most of us just simply use `new/malloc` or `delete/free` So far, since only small program are built, so good. The only thing we need to do is to make a system call. Even when some careless programmer forgets to verify whether space is allocated normally, it always works.

However, hidden trouble exists.

- First, take implement of those system calls into consideration.

  Take `malloc/free` as an example. There is a free linked list which links free memory blocks. When `malloc` function is called, system will use some optiminum algorithm to search for a suitable space. If necessary, it will split a space block, return one to caller, and add another to linked list. When `free` function is called, system will links the space which needs to be deallocated to free lists.

  Thus, when it comes to a special program, where frequent `malloc/-free` calls for small object occur, the free list will finally hold numberous fragments. Then when another `malloc` call occurs, system must arrange those fragments first, the completment of space allocation will be delayed.

- Second, it's fact that even with no rearrangment, call for those function takes lots of time. When numberous calls occur, extra time are consumed.

According to the hidden trouble above, obviously, if a program needs to make numberous allocataion and deallocation calls, default space allocation methods are not recommendable. We must construct new memory management.

1.2 Memory Pool

Memory pool, also called fixed-size blocks allocation, is a good option [1]. In order not to suffer from fragmentation and to speed up memory

allocation, it preallocates a number of memory blocks with some fixed size.

Then application can allocate, access and free memory through the following interface:

1. **Allocate memory from the pools.**

   The function will determine the pool where the required block fits in. If all blocks of that pool are already reserved, the function tries to find one in the next bigger pool(s).

2. **Get an access pointer to the allocated memory.**

3. **Free the formerly allocated memory block by return it to pool.**

In general, pool allocators can help with allocating/freeing allocations of a certain size, in any order, in O(1) time.

1.3 Our Goal

Our task is to implement an allocator with memory pool to support `std::vector` and `std::list`.

The requirement is listed below:

- **The allocator should optimize memory allocation speed usind memory pool.**
- **The allocator should support arbitrary memory size allocation request.**

Then we should test the ability of the allocator with 10000 times of various size of vector construction and destruction.

2 Our Allocator with Memory Pool

2.1 STL Allocator Interface Implement

In C++ computer programming, `allocator`s are an important component of the C++ Standard Library. STL provides several data strucures

referred to as containers. And a common trait among these containers is their ability to change size during the execution of the program. To achieve this, some form of dynamic memory allocation is usually required. Allocators handle all the requests for allocation and deallocation of memory.

Any class that fulfills the **allocator requirements** can be used as an allocator. In particular, a class `A` capable of allocating memory for an object of type `T` must provide types for generically declaring objects and references (or pointers) to objects of type `T`, type that represent the largest size for an object in the allocation modele defined by `A`, and type that can represent the difference between any two pointers in the allocation model.

```
1  A::pointer
2  A::const_pointer
3  A::reference
4  A::const_reference
5  A::value_type
6  A::size_type
7  A::difference_type
```

STL separates the process of `new` or `delete` an object into two steps. As for `new` an object, first we allocate suitable space by `allocate` function, then we call `construct` function to construct object. And the `delete` function is similar. Thus we are able to optimize the first step seperately.

These two functions are to construct or destroy an object.

```
1 template <typename T> void A::construct(A::pointer p,
    A::const_reference t) { new ((void*) p) T(t); }
2 template <typename T>
3 void A::destroy(A::pointer p){ ((T*)p)->~T(); }
```

These two functions are to perform allocation and deallocation. The way we allocate memory is realized by these two functions.

```
1 void A::deallocate(A::pointer Ptr, A::size_type Count)
```

```
2 A::pointer A::allocate(A::size_type Count)
```

In our ideal, the two functions should work with memory pool. However, in one of the STL versions, P.J. Plauger version, which is adopted by Visual Studio, these two functions simply package `new / delete`, and make no effort to optimize memory allocation. Later, we will design the two functions and memory pool with inspiration of SGI STL, and optimize memory allocation.

Inspired by `alloc` in SGI STL [2], we design our own allocator. The three major member functions are listed below. The class memory pool will be stated in next sub-section.

```
1 void deallocate(pointer Ptr, size_type Count) {
2   size_type Size = Count * sizeof(T);
3   memoryPool::deallocate(Ptr, Size);
4 }
```

```
1 _DECLSPEC_ALLOCATOR pointer allocate(size_type Count)
    {
2   void *Ptr = NULL;
3   size_type Size = Count * sizeof(T);
4   if (Count) {
5     Ptr = memoryPool::allocate(Size);
6   }
7   return static_cast<pointer>(Ptr);
8 }
```

```
1 _DECLSPEC_ALLOCATOR pointer allocate(size_type Count,
    const void*) {
2   return allocate(Count);
3 }
```

These are some extra member functions that class A needs.

This function is to get the largest number of objects of type T that could be expected to be successfully allocated.

```
1 A::max_size(){ return UINT_MAX / sizeof(T); }
```

This function is to denote the address of an object.

```
1 A::pointer A::address(A::reference V){return &V}
```

In order to enable the possibility of obtaining a related allocator, Allocators are also required to supply a template member:

```
1 template <typename U> struct A::rebind { typedef A<U>
    other; };
```

2.2 Our Memory Pool

The declaration of class memoryPool:

```
1  class memoryPool {
2    typedef size_t size_type;
3
4  public:
5    static void *allocate(size_type N);
6    static void deallocate(void *Ptr, size_type N);
7
8  private:
9    enum{Align = 8};
10   enum{MaxBytes = 4096};
11   enum{SecondMaxBytes = 40960};
12   enum{NFreeLists = MaxBytes / Align};
13   enum{BigFreeLists = (SecondMaxBytes -
       MaxBytes)/Align};
```

```
14   enum{NumOfObj = 20};
15   enum{Increment = 8};
16
17   #define roundUp(x)   (((x) + Align - 1) & (~(Align
        - 1)))
18
19   union Obj {
20   Obj * freeListLink;
21   char Data;
22   };
23
24   static char *startFree;
25   static char *endFree;
26   static Obj * freeList[NFreeLists];
27   static Obj * BFreeList[BigFreeLists];
28   static int freeListCnt[NFreeLists];
29   static size_type heapSize;
30
31   static inline size_type freeListIndex(size_type N);
32   static void *refill(size_type N);
33 };
```

Here, the two public member functions `void *allocate(size_type N)` and `void deallocate(void *Ptr, size_type N)` are called in our allocator as is stated earlier.

In class memory pool, `allocate` and `deallocate` functions are both realized by three allocation methods in order to avoid memory fragments, speed up allocation, and reduce extra needed space. When the required space is big enough, we adopt the first allocation strategy: simply use `new` function to get space. If the required space is bigger than fisrt threshold we set, but smaller than second threshold, we adopt the second strategy with memory pool. If the required space is bigger than second threshold, which means it is big enough, we will adopt final strategy with memory pool too,

as 1 shows.

2.2.1 First-level allocation

In first-level allocation, since time of allcation for big space is much less than that for small space, we set a parameter `MaxBytes` as threshold, and if allocation block is bigger than the threshold, we just use `new` to allocate.

We design this method because as for big space block, compared with much more extra space to manage memory pool, the time consumed with directly allocation is acceptable, let alone that the requirment time is less.

Here `N` is the required bytes.

```
1 (void *)::operator new(N);
```

Here `Ptr` is a pointer pointing to the space needing free.

```
1 ::operator delete(Ptr);
```

2.2.2 Second-level allocation

In second-level allocation, since in test program, most of space required is for rather small object, and the times of the new operation and delete operation of small objects are so frequent, numberous space that could be saved for later use is wrongly freed. Thus, lots of time is wasted. So in this level, we design a allocation method to keep a free list to hold small space blocks and maintain a memory pool:

Everytime when a small block is required, allocate a big block, and maintain corresponding free list. Next time when same block is required, return one block held in free list. If it needs to be freed, allocator will put it back to free list, waiting for next call. And for sake of more convienent management, we will align space capacity by 8 bytes. For example, if client requires space with 30 bytes, we will regard it as 32 bytes. If the parameter is 4 bytes, then too many linked lists should be managed. If it is 16 bytes, then wasted space is too big. Thus we choose 8 bytes to align. In this way,

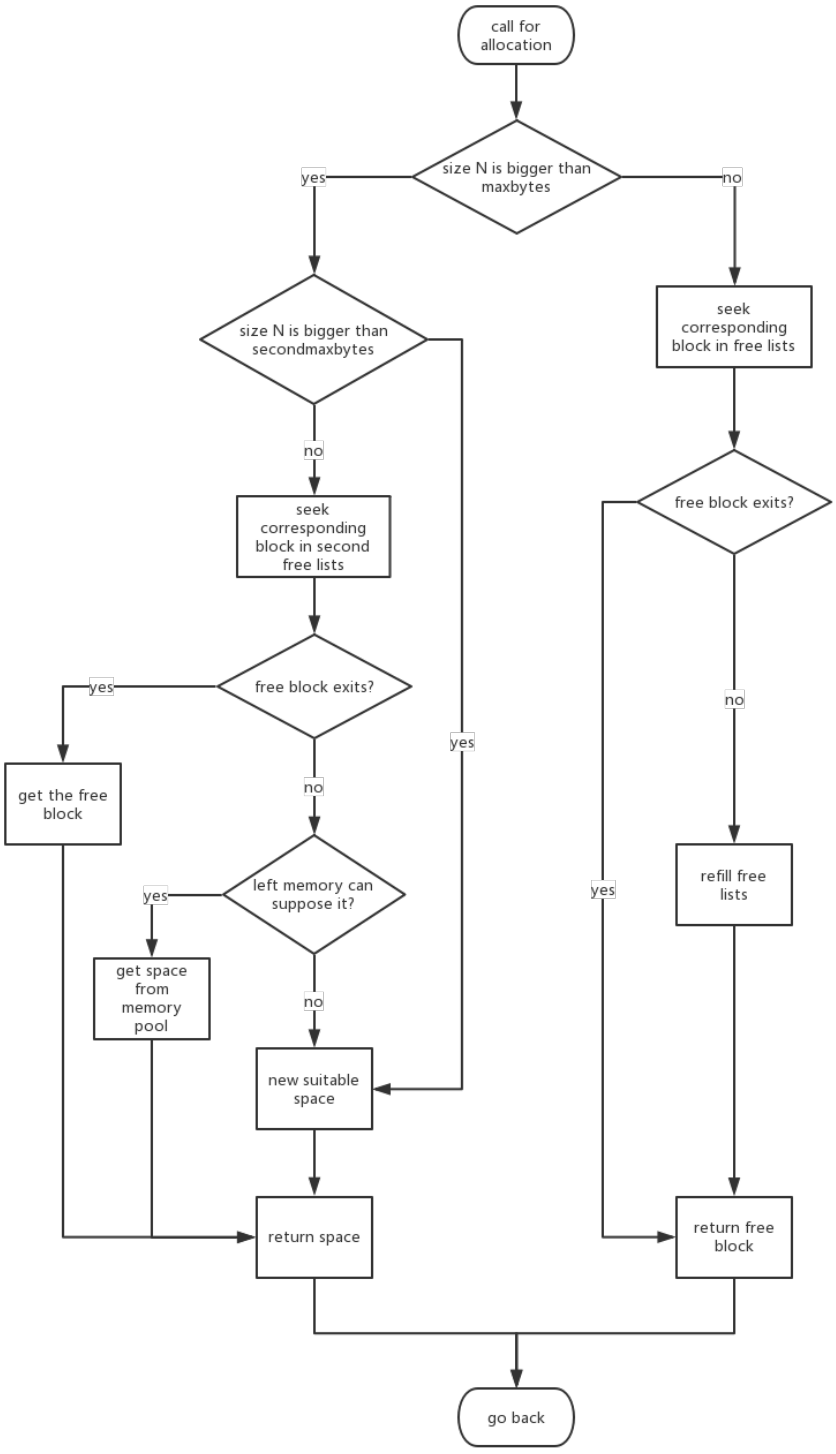

Diagram 1: Allocator framework

considering that `int, float` and pointer all occupy 4 bytes, the waste is acceptable, and management cost is also appropriate.

- Freelists

  `FreeLists` is a pointer array. It's element type is pointer implemented by union pointing to a linked list as 2 shows. Each linked list holds free small blocks with a corresponding fixed size like 3 shows. And we use `union` to implement pointer, in order to reduce extra cost for maintainment of freelists.

```
1 union Obj {
2   Obj * freeListLink;
3   char Data;
4 };
```

The allocation operation has two steps:

- **(1) Seek block in free list**

  In this step, we simply round up size of required space, and search in freelists for free block. If it exits, return it directly. If not, call `refill` function to refill freelists, then return a free block after refillment.

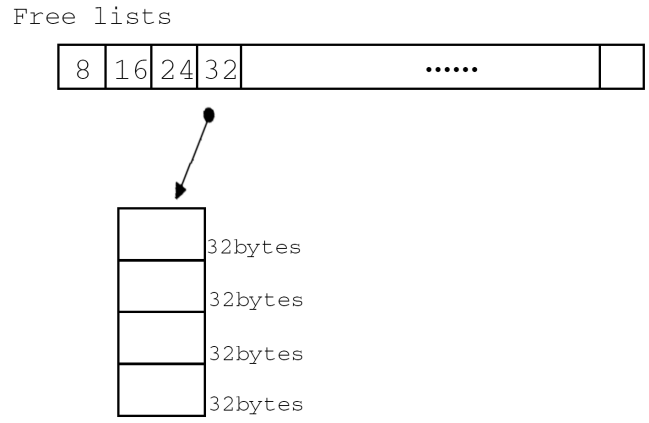

Diagram 2: Free lists

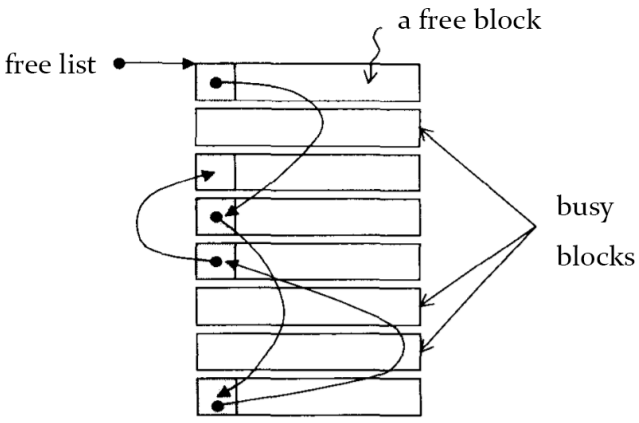

Diagram 3: One linked list in free lists

Algorithm 1: Seek block in free list

```
input size N
round up N to be multiple of 8
Calculate corresponding index
Get the pointer Ptr pointing to corresponding
  linked list
if Ptr is not NULL then
    Result ← Ptr
    Ptr ← (Ptr→next)
end
else
    Result ← refill freelists(N)
end
return Result
```

- **(2) Refill free list with memory pool**

  If a linked list is NULL for centain size block, we will refill it with many blocks. The number of blocks needed from memory pool depends.In general, an application usually calls a number of fixed size object,that means some freeList slot will be used more frequently. So, when refilling one freeList, we'd better give each size block differential treatment. The more one certain size used, the more blocks it will be refilled. To implement this idea, we use array `freeListCnt` to count the time of refillment for each size of block. And for next refillment, we will refill it with

  $$NumOfObj + Increment << Cnt$$

  blocks (if memory pool allows). Then if a size usually run out of its blocks, it will get more blocks next time. It can reduce the times of function `refill` calling which is the costly operation.

  If space in memory pool is smaller than one block, we will use `::operator new` to allocate new memory pool. Similar with refillment, the size of new memory pool is effected by the size accumulated. More

times we allocate a memory pool, bigger capacity we allocate.

Algorithm 2: Refillment

```
1  bytesLeft ← endFree - startFree
2  NObj ← NumOfObj + Increment << Cnt
3  totalBytes ← NObj * size of Type
4  if bytesLeft > totalBytes then
5  |  Result ← startFree
6  |  startFree += totalBytes
7  end
8  else if bytesLeft > size of block then
9  |  Result ← startFree
10 |  NObj ← bytesLeft / size of block
11 |  update totalbytes
12 |  startFree += size of block
13 end
14 else
15 |  if bytesLeft > 0 then
16 |  |  put left space into corresponding list
17 |  end
18 |  newBytes = roundUp(totalBytes * 2 + (heapSize
   |  >> 4))
19 |  heapSize ← heapSize + newBytes
20 |  startFree ← new (newBytes)
21 |  update totalBytes and NObj
22 |  endFree ← startFree + newBytes
23 |  Result ← startFree
24 end
25 startFree += size of block
26 foreach i < NObj do
27 |  split a block from space
28 |  add it to corresponding list
29 |  i++
30 end
31 return Result
```

2.2.3 Third-level allocation

In third-level allocation, there is also a freelists called `BFreelist`, some related defination is like this:

```
1 enum{BigFreeLists = (SecondMaxBytes -
    MaxBytes)/Align};
2 static Obj * BFreeList[BigFreeLists];
```

When the required space size is between `SecondMaxBytes` and `MaxBytes`, second-level allocation works. Here, the `SecondMaxBytes` is 40960, and the `MaxBytes` is 4096. Everything else in third-level allocation is the same as that in second-level allocation, except `refill` operation:

Since when `BFreelist` runs out, those space size is too big to refill lists with sevel new blocks, and taking the less frequencies of allocation into consideration, we will just allocate one corresponding space block rather than `NumOfObj + Increment << Cnt`.

In class memoryPool, as we all see, those member variables and functions are all static. So we can use them directly without creating an instance. It guarantees that for one program, during the whole running process, only one memory pool is held, no matter how many objects are created, resized, or deleted.

And all memory once has been allocated, it won't be returned until the program exits. In other word, the capacity of space that memory pool holds is always increasing.

What's more, the performance of our memory pool is depended on test program. If our allocator is tested with different test program, parameters (`Align, MaxBytes, SecondMaxBytes, NFreeLists, BigFreeLists, NumOfObj, Increment`) need ajusting to give a best performance. However, since memory pool is most used to optimize performance of one centain application, some prerequisite adjustment is common.

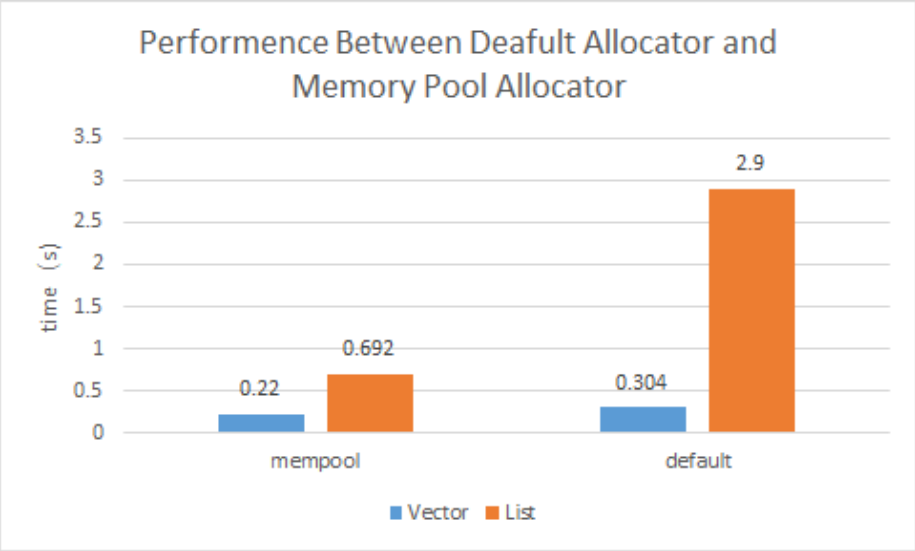

Diagram 4: Two allocator's performance

3 Testing results

We use vector and list which are most useful STL Containers to test our Memory Pool Allocator. We generate 12000 vector and 12000 list at random size. Then we stochastically choose 4000 container to be resized to a random size for both vector and list. We do same opertion to default allocator and calculate the optimizing rate $(T_d - T_m)/T_d$.

1. First, we test these two allocator's performance. Obviously, allocator with memory pool is much better than default allocator as 4 shows.

2. Second, we compare the performance between using normal distribution data and linear distribution data. For normal distribution data, we set mean = 256, variance = 10. Both up bound size is 500. We find that optimizing rate of vector is higher in normal distribution data obviously as 5 shows.

3. Then, We compare the performance between using Cnt and not.The conception of cnt has been introduced above. We find that using Cnt can do better in vector as 6 shows.

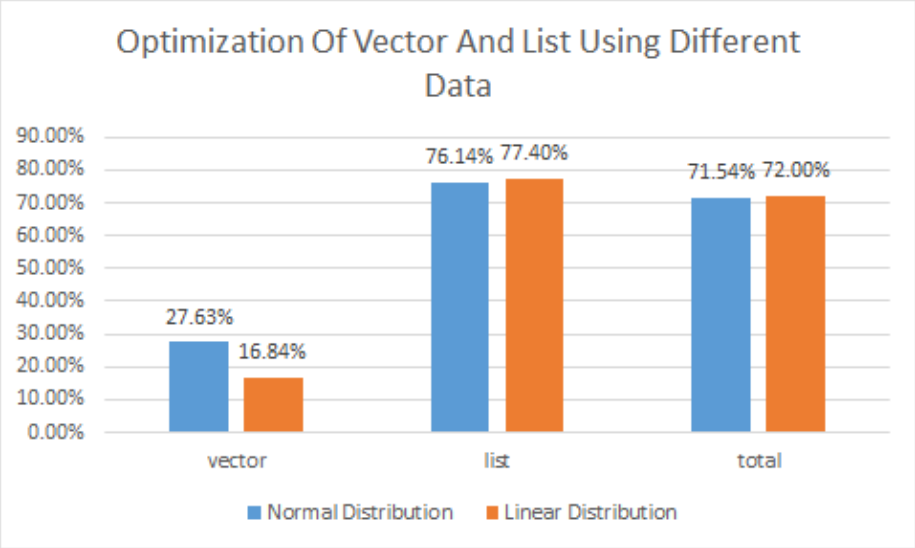

Diagram 5: Two data sets: normal distribution data and linear distribution data

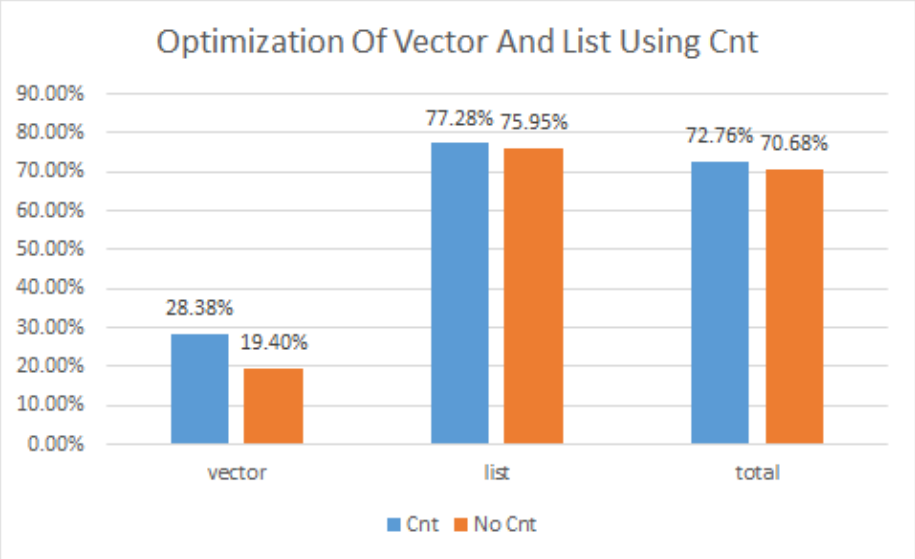

Diagram 6: Use Cnt and not

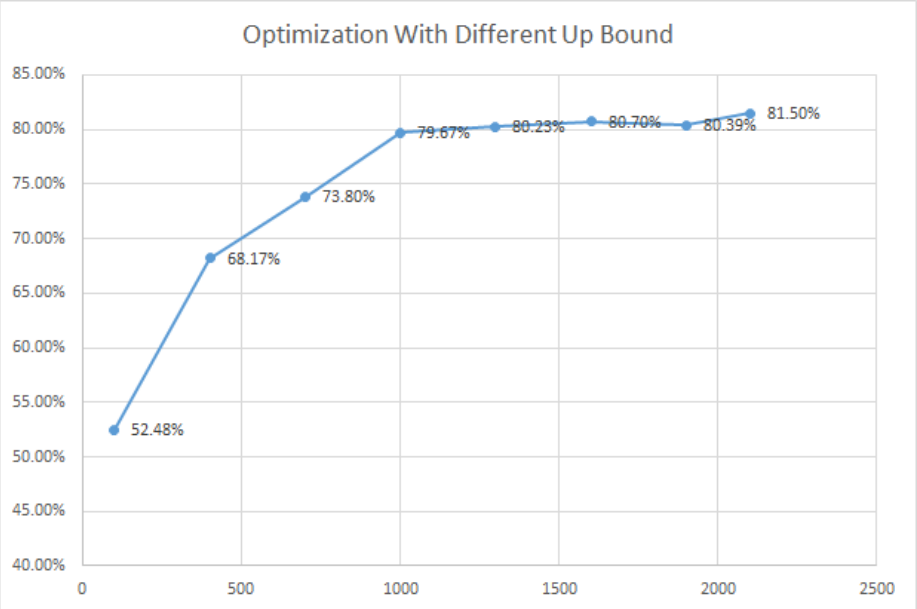

Diagram 7: Performance with different data size

4. Last, we compare the performance in different up bound of size. We find that with the up bound go up, optimizing rate goes up and to be convergent as 7 shows.

4 Reference

[1] WIKI Allocator(C++) https://en.wikipedia.org/wiki/Allocator_(C++)
[2] 侯捷. SGI STL 源码剖析. 华中科技大学出版社,2002