Subject: Impressive improvement in stl::vector when dealing with raw memory.
Posted by Lance on Mon, 14 Nov 2022 00:48:14 GMT
View Forum Message <> Reply to Message

Roughly 2 years ago, Mirek wrote this article. Some of the facts, like the speed of NTL containers versus standard library counterparts, which are well know to us, obviously surprised other readers. In the comment section of the article, Mirek and Espen Harlinn had an in depth discussion:

Here is a quote that kind of initiated the interesting discussion:
Quote:
OK ...

U++ appears to be an impressive piece of work, but:

You are making some remarkable claims with regard to the performance of your library, and how you have achieved this alleged performance boost.

You claim that memcpy/memmove is faster than std::copy, while in my experience the performance of memcpy/memmove is the same as for std::copy/std::copy_backward.

Your string class is supposed to be faster than std::string. While this may be true for some operations, it is probably not true for the most important ones, and for situations where your implementation is faster, you will probably get similar performance using std::string_view.

Statements like:
Quote:
it is still very useful and using memmove for this task easily results in 5 times speedup of the operation.

implies that the standard library is really bad. If it were true, than that would be rather embarrassing ...

I've made similar, if not so bold, claims in the past, but C++ and the standard library has evolved to a point where I would be hesitant to do so again.

I am also not plagued by memory leaks since I am mostly using std::unique_ptr and std::shared_ptr to manage memory resource ownership.

Best regards
Espen Harlinn

I reread the article a few weeks ago, and decided to do a short test. Guess what, I am surprised by the test result. I want to share my findings with the community and please do similar test on your own machine --- either to confirm or disprove my test.

I basically used the benchmarks/Vector package but tailored it to builtin types.

```
#include <Core/Core.h>
#include <vector>

using namespace Upp;

const int N = 400000;
const int M = 30;
const size_t buffsize = 128;

struct Buff{
   Buff()=default;
   Buff(const Buff&)=default;
   Buff(Buff&&)=default;

   char buff[buffsize];
};

namespace Upp{
NTL_MOVEABLE(Buff);
}

void TestInt();
void TestIntInsert();
void TestCharBuffer();

CONSOLE_APP_MAIN
{
   TestCharBuffer();
//    TestInt();
//    TestIntInsert();
}

void TestCharBuffer()
{
   for(int i=0; i < M; ++i)
   {
  {
         RTIMING("std::vector<Buff>::push_back");
    std::vector<Buff> v;
    for(int i = 0; i < N; i++){
       Buff b;
  v.push_back(b);
    }
 }

 {
```

```
    RTIMING("Upp::Vector<Buff>::push_back");
    Upp::Vector<Buff> v;
    for(int i = 0; i < N; i++){
  Buff b;
  v.Add(b);
    }
 }

   }
}

void TestInt()
{
   for(int i=0; i < M; ++i)
   {
 {
    RTIMING("std::vector<int>::push_back");
    std::vector<int> v;
    for(int i = 0; i < N; i++)
  v.push_back(i);
 }
 {
    RTIMING("Upp::Vector<int>::push_back");
    Upp::Vector<int> v;
    for(int i = 0; i < N; i++)
  v.push_back(i);
 }

   }
}

void TestIntInsert()
{
   for(int i=0; i < M; ++i)
   {
 {
    RTIMING("std::vector<int>::insert");
    std::vector<int> v;
    for(int i = 0; i < N; i++)
  v.insert(v.begin(), i);
 }
 {
    RTIMING("Upp::Vector<int>::insert");
    Upp::Vector<int> v;
    for(int i = 0; i < N; i++)
  v.Insert(0, i);
 }
   }
```

}

Some of the test results:

TIMING Upp::Vector<Buff>::push_back: 1.99 s - 66.33 ms (1.99 s / 30), min: 63.00 ms, max: 73.00 ms, nesting: 0 - 30
TIMING std::vector<Buff>::push_back: 1.23 s - 41.07 ms (1.23 s / 30), min: 39.00 ms, max: 47.00 ms, nesting: 0 - 30

The number fluctuate quite a lot, but mostly the result is in favour of std::vector (when handling raw bytes).

BTW, testing insertion is very time consuming, considering start from small number for N and M, then gradually increase. It appears std::vector excels when N are big.

My CPU:

```
Architecture:          x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
    CPU family:          6
    Model:               142
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            10
    CPU max MHz:         4200.0000
    CPU min MHz:         400.0000
    BogoMIPS:            4199.88
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall
                         nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq
                         dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer
                        aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi fle
                        xpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt
                         xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d arch_capabilities
```

Virtualization features:
  Virtualization: VT-x
Caches (sum of all):
  L1d: 128 KiB (4 instances)
  L1i: 128 KiB (4 instances)
  L2:   1 MiB (4 instances)
  L3:   8 MiB (1 instance)
NUMA:
  NUMA node(s):   1
  NUMA node0 CPU(s):   0-7
Vulnerabilities:
  Itlb multihit: KVM: Mitigation: VMX disabled
  L1tf: Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
  Mds: Mitigation; Clear CPU buffers; SMT vulnerable
  Meltdown: Mitigation; PTI
  Mmio stale data: Mitigation; Clear CPU buffers; SMT vulnerable
  Retbleed: Mitigation; IBRS
  Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
  Spectre v1:   Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:   Mitigation; IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS Not affected
  Srbds: Mitigation; Microcode
  Tsx async abort: Mitigation; TSX disabled

I would appreciate if you can do the test and share your results.

BR,
Lance