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

Subject: Re: Impressive improvement in stl::vector when dealing with raw memory.
Posted by pvictor on Mon, 14 Nov 2022 08:46:13 GMT

View Forum Message <> Reply to Message

Hi!

Here's my results:

```
TIMING Upp::Vector<int>::insert: 478.83 s  - 15.96 s  (478.83 s  / 30 ), min: 15.82 s , max: 16.10 s , nesting: 0 - 30
TIMING std::vector<int>::insert: 478.96 s  - 15.97 s  (478.96 s  / 30 ), min: 15.82 s , max: 16.09 s , nesting: 0 - 30
TIMING Upp::Vector<int>::push_back: 30.00 ms - 999.97 us (30.00 ms / 30 ), min:  1.00 ms, max:  1.00 ms, nesting: 0 - 30
TIMING std::vector<int>::push_back: 32.00 ms -  1.07 ms (32.00 ms / 30 ), min:  1.00 ms, max:  2.00 ms, nesting: 0 - 30
TIMING Upp::Vector<Buff>::push_back:  2.34 s  - 78.10 ms ( 2.34 s  / 30 ), min: 76.00 ms, max: 82.00 ms, nesting: 0 - 30
```

TIMING std::vector<Buff>::push_back: 1.79 s - 59.70 ms (1.79 s / 30), min: 58.00 ms, max: 68.00 ms, nesting: 0 - 30

However, when I modify the code:

```
void TestCharBuffer() {
 for(int i=0; i < M; ++i) {
  {
   RTIMING("std::vector<Buff>::push_back");
   std::vector<Buff> v;
   v.reserve(N);   // +++
   for(int i = 0; i < N; i++) {
    Buff b;
    v.push_back(b);
   }
  }
  {
   RTIMING("Upp::Vector<Buff>::push_back");
   Upp::Vector<Buff> v;
   v.Reserve(N);   // +++
   for(int i = 0; i < N; i++){
    Buff b;
    v.Add(b);
   }
  }
 }
}
```

I get:

TIMING Upp::Vector<Buff>::push_back: 834.00 ms - 27.80 ms (834.00 ms / 30), min: 27.00 ms, max: 29.00 ms, nesting: 0 - 30
TIMING std::vector<Buff>::push_back: 834.00 ms - 27.80 ms (834.00 ms / 30), min: 27.00 ms, max: 30.00 ms, nesting: 0 - 30

It seems that Upp::Vector wastes more time for memory allocation.

Best regards,
Victor

Subject: Re: Impressive improvement in stl::vector when dealing with raw memory.
Posted by Lance on Mon, 14 Nov 2022 13:31:05 GMT
View Forum Message <> Reply to Message

Victor:

Thank you! I was suspecting it's probably just because my CPU lack certain optimisation.

The push_back()/emplace_back() action on raw bytes without reserve first is the most important indicator of efficiency. stl::vector is finally catching up , it seems.

insert/remove in the beginging is also story telling but this if of less importance, because a vector is not designed for frequent add/remove from position other than close to the end. Anyway in this respect, both std::vector and Upp::vector perform at par.

Subject: Re: Impressive improvement in stl::vector when dealing with raw memory.
Posted by Lance on Mon, 14 Nov 2022 13:52:57 GMT
View Forum Message <> Reply to Message

Why is operating on raw bytes a big deal? Isn't Upp still doing a lot better on Upp::Moveable objects like Upp::String? Well, it's just a matter of teaching stl::vector to treat Upp::String(and Upp::Moveable as a whole) as raw bytes and it will catch up or even outperform.

Well it all starts with testing Upp code for C++20 compliance. let's try theide first. The ide compiles fine on both GCC and CLANG with -std=c++20 option, except some complaints on capturing this by default is deprecated in C++20, which are easy to fix or safe to ignore for now. But it's a total different story with MSC. with standard set to C++17, MSC rejects a bunch of stuff like

return somecondition? "a literal string" : AString;

These are also easy to fix if you don't mind your local version is slightly different from the main stream.

When standard is set to C++20 or c++latest, Upp::Moveable AssertMoveable0() is start to causing compilation failure, this one seems to be quite difficult to fix.

I was thinking it's just a mechanism to communicate to the compiler that it can treat object of this class as raw bytes, maybe we can do it differently with so much more facilities available in more recent c++ library.

So I start to do some experiment.

Subject: Re: Impressive improvement in stl::vector when dealing with raw memory.
Posted by Lance on Mon, 14 Nov 2022 14:28:52 GMT
View Forum Message <> Reply to Message

Extract of <CoreExt/relocatable.hpp>

```
// a value of -1 indicate that we don't know if the class('s object) is trivially
// relocatable and if not, how to do adjustment after relocation
```

```
//
template <class T> struct relocate_traits { static constexpr int value = -1; };

// a value of 0 indicates object of T can be move around like raw bytes.
template <class T> requires std::is_trivial_v<T>
struct relocate_traits<T> { static constexpr int value = 0; };

template <class T>
concept UppMoveable = requires{
 typename T::MoveableBase; // Note, this require to add a typedef in Topt.h/Moveable definition
 requires std::derived_from<T, Upp::Moveable<T, typename T::MoveableBase> >;
};

// the following will teach compiler to treat all Upp::Moveable derivatives as
// trivially relocatable
template <UppMoveable T>
struct relocate_traits<T> { static constexpr int value = 0; };

template <class T>
inline constexpr bool is_trivially_relocatable_v = relocate_traits<T>::value == 0;
```

With that, the AssertMoveable part can be done without. This involves C++20 language features unfortunately (concept/requires). Maybe a extra #if c++version>2020(in the spirit) is needed to make both worlds happy.

And there are more opportunities opened with this approach.

The rest of my <CoreExt/relocatable.hpp>

```
// facility to get the class name from a member function pointer
//
template<class T> struct get_class;
template<class T, class R>
struct get_class<R T::*> { using type = T; };


template <class T>
requires requires(T t){
  { t.DoPostRelocationAdjustment() }noexcept;
  requires std::same_as<
   T, typename get_class<decltype(&T::DoPostRelocationAdjustment)>::type
  >;
 }
struct relocate_traits<T>{
 static constexpr int value = 1; // simple
 static void Do(T* obj)noexcept{ obj->DoPostRelocationAdjustment(); }
};
```

```
template <class T>
requires requires(T * p){// need to fix, pointer to derived class should be rejected
  {DoPostRelocationAdjustment(p) }noexcept;
 }
struct relocate_traits<T>{
 static constexpr int value = 1; // simple
 static void Do(T* obj)noexcept{ DoPostRelocationAdjustment(obj); }
};

template <class T>
requires requires(T t, const T* old){
  {t.DoPostRelocationAdjustment(old)}noexcept;
  requires std::same_as<
   T, typename get_class<decltype(&T::DoPostRelocationAdjustment)>::type
  >;
 }
struct relocate_traits<T>{
 static constexpr int value = 2; // old address is supplied.
 static void Do(T* obj, const T* old)noexcept{ obj->DoPostRelocationAdjustment(old); }
};

template <class T>
requires requires(T *obj, const T * old){
  {DoPostRelocationAdjustment(obj, old)}noexcept;
 }
struct relocate_traits<T>{
 static constexpr int value = 2; // old address is supplied.
 void Do(T* obj, const T* old)noexcept{ DoPostRelocationAdjustment(obj, old); }
};

template <class T>
requires requires(T t, const T* old, const T * from, const T * to){
  {t.DoPostRelocationAdjustment(old, from, to)}noexcept;
  requires std::same_as<
   T, typename get_class<decltype(&T::DoPostRelocationAdjustment)>::type
  >;
 }
struct relocate_traits<T>{
 static constexpr int value = 4; // 4 address version
 void Do(T* obj, const T* old, const T* from, const T* to)noexcept{
  obj->DoPostRelocationAdjustment(old, from, to);
 }
};

template <class T>
requires requires(T *p, const T * o, const T * from, const T* to){
  {DoPostRelocationAdjustment(p, o, from, to)}noexcept;
```

```
 }
struct relocate_traits<T>{
 static constexpr int value = 4; // 4 address version.
 void Do(T* obj, const T* old, const T* from, const T* to)noexcept{
  DoPostRelocationAdjustment(obj, old, from, to);
 }
};

template <class T>
concept relocatable = relocate_traits<T>::value != -1;

#define DECLARE_TRIVIALLY_RELOCATABLE( T ) namespace lz{\
 template <> struct relocate_traits<T>{ const static int value = 0; };\
}

#define RELOCATE_ADJUSTMENT_1(T, func) namespace lz{\
 struct relocate_traits<T>{\
  static constexpr int value = 1;\
  static void Do(T* obj)noexcept{ func(obj); }\
 };\
}

#define RELOCATE_ADJUSTMENT_2(T, func) namespace lz{\
 struct relocate_traits<T>{\
  static constexpr int value = 2;\
  static void Do(T* obj, const T* old)noexcept{ func(obj, old); }\
 };\
}

#define RELOCATE_ADJUSTMENT_4(T, func) namespace lz{\
 struct relocate_traits<T>{\
  static constexpr int value = 4;\
  static void Do(T* obj, const T* old, const T* start, const T* end)noexcept\
  { func(obj, old, start, end); }\
 };\
}
```

With this, we can teach a vector or Vector to handle objects that's not trivially relocatable, for example, Upp::Ctrl, or some one like this

```
class SomeClassWithBackpointer{
    struct Node{
        SomeClassWithBackpointer * owner;

        void SomeFunction(){}
```

```
        char buff[1024];
    }

    SomeClassWithBackpointer()=default;
    SomeClassWithBackpointer(const SomeClassWithBackpointer& ){...}

    void DoPostRelocationAdjustment()noexcept{
        if(node) node->owner = this;
    }

    Node * node = nullptr;
    char buff[1024]; // to make the class heavier
                     // otherwise, it's cheaper
                     // by simply move construct it.
};
```

With this definition, SomeClassWithBackpointer objects can be housed in a (revised)vector/Vector quite efficiently. I have done a trial implementation(incomplete) of such a vector.

Subject: Re: Impressive improvement in std::vector when dealing with raw memory.
Posted by Lance on Mon, 14 Nov 2022 16:24:16 GMT
View Forum Message <> Reply to Message

I have uploaded my (incomplete, simplified) std::vector implementation to GitHub. It's just a proof of concept. It's a refactoring of Upp::Vector, using std::vector interfaces and Upp::Vector memory management facilities and logic. It performs at par with(if not marginally faster than)Upp::Vector and should generate smaller executable size (which can be proved in theory and is tested true in practice).

Once I found it handles raw bytes slower than std::vector, I lose confidence/interests to continue. But it suffices to demonstrate my point: std::vector can be trained to handle trivially relocatable class object just as good as Upp::Vector (I expect it happen in not long future, if I can do that, why not all those a million times smarter guys), and more than that, in many situations, we can make the vectors work nicely with non-trivially relocatable objects, some times with tremendous performance gain.

I am able to relocate a Ctrl (even though that's really little point to do that --- in certain cases, it makes sense to put a lot of dynamically allocated child Ctrls of same [or similar, a different story] kind in a vector instead of Upp::Array for memory efficiency and less fragamentaion), and I make a std::basic_string relocatable, even though in this case I actually got a performance penalty: basic_string is too small to gain anything from move raw bytes then adjust affected pointers. But you can conceive there are lots of cases where moving raw bytes then adjust a couple of back pointers make sense.

Story of std::basic_string (GLIBCXX implementation)

It's surprising that a basic_string<ch> would cause trouble (core dump etc) when treated as raw bytes. Digging into its implementation (in <bits/basic_string.h>), we have the data members

```
      // Use empty-base optimization: http://www.cantrip.org/emptyopt.html
      struct _Alloc_hider : allocator_type // TODO check __is_final
      {
#if __cplusplus < 201103L
 _Alloc_hider(pointer __dat, const _Alloc& __a = _Alloc())
 : allocator_type(__a), _M_p(__dat) { }
#else
 _Alloc_hider(pointer __dat, const _Alloc& __a)
 : allocator_type(__a), _M_p(__dat) { }

 _Alloc_hider(pointer __dat, _Alloc&& __a = _Alloc())
 : allocator_type(std::move(__a)), _M_p(__dat) { }
#endif

 pointer _M_p; // The actual data.
      };

      _Alloc_hider _M_dataplus;
      size_type  _M_string_length;

      enum { _S_local_capacity = 15 / sizeof(_CharT) };

      union
      {
 _CharT           _M_local_buf[_S_local_capacity + 1];
 size_type        _M_allocated_capacity;
      };
```

Ignore the Allocator and empty base optimization stuff, the member variables can be translated into

```
      enum { _S_local_capacity = 15 / sizeof(_CharT) };

    pointer _M_p;
    size_type _M_string_length;
    union
    {
 _CharT        _M_local_buf[_S_local_capacity + 1];
 size_type _M_allocated_capacity;
```

```
};
```

Turns out, in the case the stored c-string can be fit in 15 bytes (one more for the null terminator), it stores the string locally and make _M_p point to it, thus created a class invariant that will break with raw move. It's disappointing that my copy raw bytes then do adjustment is less efficient than simply using move constructor, while it's logical that in the case when the object size is small comparing to adjustments that need to be made, adjusment-after-rawcopy will be more costly, let's try to blame somebody else.

Is basic_string has to be designed this way? I mean, for all it does, is the pointer to self action necessary? Indeed, it's not. We can make basic_string trivially relocatable without losing any functionality: just set _M_p to 1 when it's storing data locally!

A naive partial implementation of above idea looks like this

```
pointer p;
size_type _len;
size_type _capacity;

// fix: prepare for 16 - 2*sizeof(size_type) is 0!
char _dummy[ 16 - 2* sizeof(size_type) ];
enum{ local_capacity = 15 / sizeof(_CharT) };
// if string is store locally, how difficult
// is it to call strlen on a c-string with
// length less than 15? we can certainly use
// the space for _len for string storage too

bool local()const{ return as_int(p) = 1; }

// when no object in *this yet, ie, in constructor
void store_a_strong_raw(const chT *s){
    assert( s!= nullptr );
    if( strlen(s) <= local_capacity )
    {
       copy_string_to_local_buff();
       as_int(p) = 1;
    }else{
       p = allocate_string_in_heap();
    }
}
void store_a_string(const char * s){
    if( !local() )
        delete [] p;
    store_a_strong_raw (s);
}
```

```
~basic_string(){
    if( !local() )
       delete [] p;
}

size_type size()const{
    return local() ?
        getstrlen<chT>( local_storage_begin() ) :
        _len;
}

chT * local_storage_begin(){
    return reinterpret_cast<chT*>(
          &_len
        );
}
...
```

We have maintained the functionalities of original implementation, used less space, and most importantly, make basic_string objects trivially relocatable. Now basic_string objects will no longer be second class citizen in a vector/Vector world.

If you are more hackish: do we really need the full space of pointer p to determine if a string is stored locally? Depending on the endianness, we can potentially extract 7 more bytes on a 64-bit platform.

Subject: Re: Impressive improvement in std::vector when dealing with raw memory.
Posted by Lance on Sat, 19 Nov 2022 21:04:47 GMT
View Forum Message <> Reply to Message

Morale of basic_string story: sometimes (more than occasionally), it's possible to make a class trivially relocatable by slightly changing your design.

While I did not do a speed comparison of the underlying memory copy facilities (just move a volume of bytes around repeatedly for certain times) of std::vector and Upp::Vector, an intuitive explanation of Upp::Vector's performing well on small memory size and lagging behind when the memory block getting large is the difference in their respective memory management strategies.

A std::vector doubles it's capacity at each growth (until out of memory etc) while a Upp::Vector grows by 1/3 of its current capacity. Upp::Vector mitigates its supposedly more frequent allocation/relocation by doing TryRealloc, which, when success, housed objects relocation can be avoided. While the latter has more chances to succeed when allocated memory block is small (thus results in a amortized gain over std::vecotr), it tends to fail more often when the allocated memory block is big. In which case more frequent reallocation and relocation plus additional cost on (almost bound to fail) TryRealloc(will have to lock some mutex at least) drag the overall performance.

Subject: Re: Impressive improvement in std::vector when dealing with raw memory.
Posted by Lance on Sun, 20 Nov 2022 00:50:54 GMT
View Forum Message <> Reply to Message

Test the speed of copying raw memory of various utilities.

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

using namespace Upp;

const int N = 10000;
const int M = 3*1024*1024;

struct S{
    S()noexcept=default;
    S(const S&)noexcept=default;
    char buff[M];
};

CONSOLE_APP_MAIN
{
    S s;

    int64 t = 0;
    for(int i=0; i<N; ++i)
    {
        {
            RTIMING("the memory copy utility likely used by std::vector");
            char buff[M];
            new(buff)S(s);
            for(int i=0; i < M; ++i)
                t += buff[i];
        }
        {
            RTIMING("the memory copy utility used by Upp::Vector");
            char buff[M];
            memcpy_t((S*)buff, &s, 1);
            for(int i=0; i < M; ++i)
                t -= buff[i];
        }
        {
            RTIMING("memcpy function");
            char buff[M];
            memcpy(buff, &s, M);
            for(int i=0; i < M; ++i)
                t += buff[i];
        }
```

```
        for(int i=0; i<M; ++i)
            t -= s.buff[i];
    }

    RLOG(t);
}
```

Typical output

```
0
TIMING memcpy function: 12.25 s  -  1.22 ms (12.25 s  / 10000 ), min:  1.00 ms, max:  3.00 ms, nesting: 0 - 10000
TIMING the memory copy utility used by Upp::Vector:  8.72 s  - 871.98 us ( 8.72 s  / 10000 ), min:  0.00 ns, max:  3.00 ms, nesting: 0 - 10000
TIMING the memory copy utility likely used by std::vector: 11.63 s  -  1.16 ms (11.63 s  / 10000 ), min:  1.00 ms, max:  5.00 ms, nesting: 0 - 10000
```

It's confirmed Upp::memcpy_t with SIMD optimization is significantly faster. So it has to be because of memory allocation overhead.

Subject: Re: Impressive improvement in std::vector when dealing with raw memory.
Posted by mirek on Mon, 21 Nov 2022 15:34:07 GMT
View Forum Message <> Reply to Message

The most likely explanation:

std::vector usually grows by factor 2 which leads to average 50% overhead.

With Upp::Vector, I have decided that things being fast enough, we can use factor 1.5 and have only 25% overhead. Which means we do more reallocations to save memory.

EDIT: Sorry, now rereading the thread you have figured that out

Mirek

Subject: Re: Impressive improvement in std::vector when dealing with raw memory.
Posted by Lance on Tue, 22 Nov 2022 23:39:19 GMT
View Forum Message <> Reply to Message

Thanks, Mirek!
