Aligned Memory Allocator

Dec 6th, 2014

Introduction

In a previous article about SIMD wrappers, I suggested to design a dedicated memory allocator to handle SIMD memory alignment constraints, but I didn't give any details on how to do it. That's the purpose of this article. The C++ standard describes a set of requirements our allocator must respect to work with standard containers. After a survey of these standard requirements, we'll see how to implement an aligned memory allocator that meets them.

1. The C++ standard requirements

The standard requires the allocator to define the following type:

- **value_type**: the type of allocated elements, T
- **pointer** and **const_pointer**: pointer and constant pointer to T
- **reference** and **constant_reference**: reference and constant reference to T
- **size_type**: an unsigned integral type that can represent the size of the largest object in the allocation model
- **difference_type**: a signed integral type that can represent the difference between any two pointers in the allocation model

The standard then requires a template class **rebind** member, which is a template typedef: Allocator<T>::rebind<U>::other is the same type as Allocator<U>. This member is used by container that allocates memory for internal structures that hold T elements instead of allocating memory for T elements. For instance, std::list<T> allocates memory for Node<T> instead of T, but you don't want to use Allocator<Node<T> > as template parameter since this would expose implementation details in the interface. Thus you use Allocator<T> and the internal allocation is done with Allocator<T>::rebind<U>::other.allocate(n).

Then we have to provide the **address** functions, which return the address of a given object. Two overloads are provided, one for references and one for constant references.

The two following functions are the essential part of the allocator: **allocate** and **deallocate**, which allocates/deallocates memory for n objects of type T. These functions are low-level memory management functions and are not responsible for constructing or destroying objects, this has to be done in specific functions: **construct** and **destroy**.

The last specific function is **max_size**, a function that returns the maximum value that can be passed to allocate.

Finally, the allocator must provide default and copy constructors, and equality check operators.

2. Aligned memory allocator interface

Since we must handle different memory alignment bounds, our aligned memory allocator will take two template parameters: **T**, the type of allocated objects, and **N**, the aligment bound. Given the requirements of the previous section, the allocator interface looks like:

aligned_allocator.hpp

```
template <class T, int N>
    class aligned_allocator
    {

    public:

        typedef T value_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef T* pointer;
        typedef const T* const_pointer;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        template <class U>
            struct rebind
            {
                typedef aligned_allocator<U,N> other;
            };

        inline aligned_allocator() throw() {}
        inline aligned_allocator(const aligned_allocator&) throw() {}

        template <class U>
            inline aligned_allocator(const aligned_allocator<U,N>&) throw() {}

        inline ~aligned_allocator() throw() {}

        inline pointer address(reference r) { return &r; }
        inline const_pointer address(const_reference r) const { return &r; }

        pointer allocate(size_type n, typename std::allocator<void>::const_pointer hint = 0);
        inline void deallocate(pointer p, size_type);

        inline void construct(pointer p, const_reference value) { new (p) value_type(value); }
        inline void destroy(pointer p) { p->~value_type(); }

        inline size_type max_size() const throw() { return size_type(-1) / sizeof(T); }

        inline bool operator==(const aligned_allocator&) { return true; }
        inline bool operator!=(const aligned_allocator& rhs) { return !operator==(rhs); }
    };
```

Nothing special to say here, the **construct** function calls the copy constructor of T through the placement new operator but does not allocate memory for the element, this is the responsibility of the **allocate** function. Same thing for the **destroy** function, it calls the destructor of T but it doesn't deallocate memory, this has to be done after with a call to the **deallocate** function.

3. Allocate and deallocate implementation

We can now focus on the **allocate** and **deallocate** implementation. Depending on our platform, aligned memory allocation function may be available:

- On Windows 64 bits, FreeBSD (except on ARM and MIPS architectures), and Apple, the **malloc** function is already 16-bytes aligned
- Systems implementing POSIX provide the **posix_memalign** function
- SSE intrinsics provide the **_mm_malloc** function
- Visual Studio provides the **_aligned_malloc** function

Except for the 16-bytes aligned malloc, every function takes an alignment parameter that must be a power of 2, thus the **N** template parameter of our allocator should be a power of 2 so it can works with these aligned memory allocation functions. Note that many of these functions can be available on a same platform.

Assume we can detect at compile time if such functions are available (we'll come back on this later); we can provide a function that selects the aligned memory allocation to use if such a function is available, otherwise forwards to our own implementation:

aligned_allocator.hpp

```
namespace detail
{
    inline void* _aligned_malloc(size_t size, size_t alignment)
    {
        void* res = 0;
        void* ptr = malloc(size+alignment);
        if(ptr != 0)
        {
            res = reinterpret_cast<void*>((reinterpret_cast<size_t>(ptr) & ~(size_t(alignment-1))) + alignment);
            *(reinterpret_cast<void**>(res) - 1) = ptr;
        }
        return res;
    }
}

inline void* aligned_malloc(size_t size, size_t alignment)
{
    #if MALLOC_ALREADY_ALIGNED
        return malloc(size);
    #elif HAS_MM_MALLOC
        return _mm_malloc(size,alignment);
    #elif HAS_POSIX_MEMALIGN
        void* res;
        const int failed = posix_memalign(&res,size,alignment);
        if(failed) res = 0;
        return res;
    #elif (defined _MSC_VER)
        return _aligned_malloc(size, alignment);
    #else
        return detail::_aligned_malloc(size,alignment);
    #endif
}
```

The idea in the **_aligned_malloc** function is to search for the first aligned memory address (*res*) after the one returned by the classic malloc function (*ptr*), and to use it as return value. But since we must ensure *size* bytes are available after *res*, we must allocate more than *size* bytes; the minimum size to allocate to prevent buffer overflow is *size+alignment*. The we store the *ptr* value just before *res* so the **_aligned_free** function can easily retrieve and pass it to the classic free function:

aligned_allocator.hpp

```
namespace detail
{
    inline void _aligned_free(void* ptr)
    {
        if(ptr != 0)
            free(*(reinterpret_cast<void**>(ptr)-1));
    }
}

inline void aligned_free(void* ptr)
{
    #if MALLOC_ALREADY_ALIGNED
        free(ptr);
    #elif HAS_MM_MALLOC
        _mm_free(ptr);
    #elif HAS_POSIX_MEMALIGN
        free(ptr);
    #elif defined(_MSC_VER)
        _aligned_free(ptr);
    #else
        detail::_aligned_free(ptr);
    #endif
    }
```

The **aligned_free** function is the symmetric of **aligned_malloc**: it selects the aligned memory function available or forwards to the **_aligned_free** function.

We can now write the **allocate** and **deallocate** functions of the allocator:

aligned_allocator.hpp

```
template <class T, int N>
    typename aligned_allocator<T,N>::pointer
    aligned_allocator<T,N>::allocate(size_type n, typename std::allocator<void>::const_pointer hint)
    {
        pointer res = reinterpret_cast<pointer>(aligned_malloc(sizeof(T)*n,N));
        if(res == 0)
            throw std::bad_alloc();
        return res;
    }

template <class T, int N>
    typename aligned_allocator<T,N>::pointer
    aligned_allocator<T,N>::deallocate(pointer p, size_type)
    {
    aligned_free(p);
    }
```

Here we see the advantage to have encapsulated aligned memory allocation selection in a dedicated function: the allocate function of the allocator simply forwards to this dedicated function and then handles possible bad allocation. The result is a simple and easy-to-read code. Another advantage is that you can use **aligned_malloc** and **aligned_free** functions outside the **aligned_allocator** class if you need.

Note: the call to malloc after the MALLOC_ALREADY_ALIGNED preprocessor token should be available for 16-bytes aligned memory allocator only (the same applies to the call to free). Thus we should provide two versions of **aligned_malloc** and **aligned_free** and a specialization of the allocator ofr **N = 16**.

4. Detecting available aligned memory allocation

Now that we have implemented the allocation and deallocation methods, we can come back to the preprocessor tokens. Defining these tokens is not simple because you have to refer to the documentation of a lot of various systems and architectures. Thus there's a chance that we may not be comprehensive, but at least we can cover the most common platforms.

Let's start with the GNU world; according to this documentation, "The address of a block returned by malloc or realloc in GNU systems is always a multiple of eight (or sixteen on 64-bit systems)". According to this one, page 114, "[The] LP64 model […] is used by all 64-bit UNIX ports", therefore we should use this predefined macro instead of __x86_64__ (this last one won't work on PowerPC or SPARC). Thus we can define the following macro:

aligned_allocator.hpp

```
#if defined(__GLIBC__) && ((__GLIBC__>=2 && __GLIBC_MINOR__ >= 8) || __GLIBC__>2) \
 && defined(__LP64__)
  #define GLIBC_MALLOC_ALREADY_ALIGNED 1
#else
  #define GLIBC_MALLOC_ALREADY_ALIGNED 0
#endif
```

FreeBSD has 16-byte aligned malloc, except on ARM and MIPS architectures (see this documentation):

aligned_allocator.hpp

```
#if defined(__FreeBSD__) && !defined(__arm__) && !defined(__mips__)
  #define FREEBSD_MALLOC_ALREADY_ALIGNED 1
#else
  #define FREEBSD_MALLOC_ALREADY_ALIGNED 0
#endif
```

On windows 64 bits and Apple OS, the malloc function is also already aligned, thus we can define the **MALLOC_ALREADY_ALIGNED** macro, based on these information and the macros previously defined:

aligned_allocator.hpp

```
#if (defined(__APPLE__) \
 || defined(_WIN64) \
```

```
 || GLIBC_MALLOC_ALREADY_ALIGNED \
 || FREEBSD_MALLOC_ALREADY_ALIGNED)
  #define MALLOC_ALREADY_ALIGNED 1
#else
  #define MALLOC_ALREADY_ALIGNED 0
#endif

```

To handle systems implementing POSIX:

aligned_allocator.hpp

```
#if ((defined __QNXNTO__) || (defined _GNU_SOURCE) || ((defined _XOPEN_SOURCE) && (_XOPEN_SOURCE >= 600))) \
 && (defined _POSIX_ADVISORY_INFO) && (_POSIX_ADVISORY_INFO > 0)
  #define HAS_POSIX_MEMALIGN 1
#else
  #define HAS_POSIX_MEMALIGN 0
#endif

```

The last macro to define is **HAS_MM_MALLOC**; the _mm_malloc function is provided with SSE intrinsics, thus we can rely on the macros defined in this article:

aligned_allocator.hpp

```
#if SSE_INSTR_SET > 0
    #define HAS_MM_MALLOC 1
#else
    #define HAS_MM_MALLOC 0
#endif

```

That's it, some architectures may be missing but it shouldn't be too complicated to handle them with appropriate documentation.

Conclusion

The aligned memory allocator designed in this article meets the standard requirements and can therefore be used with any container of the STL. If you work with intrinsics wrappers and std::vector, this allocator will allow you to load the data from memory with the load_a method, faster than load_u (the same applies for storing data to memory):

sample.cpp

```
typedef std::vector<double,aligned_allocator<double,16> > vector_type;
vector_type v1,v2,v3;
// code filling v1 and v2
for(size_t i = 0; i < v1.size(); i += simd_traits<double>::size)
{
    vector2d v1d = load_a(&v1[i]);
    vector2d v2d = load_a(&v2[i]);
    vector2d v3d = v1d + v2d;
    store_a(&v3[i],v3d);
}
```

But as we will see in a forthcoming article, std::vector may not be the most appropriate container for efficient numerical analysis.

Posted by Johan Mabille Dec 6th, 2014 Memory

« Performance considerations about SIMD wrappers

Comments