What happens inside Eigen, on a simple example

Consider the following example program:

```
#include<Eigen/Core>

int main()
{
  int size = 50;
  // VectorXf is a vector of floats, with dynamic size.
  Eigen::VectorXf u(size), v(size), w(size);
  u = v + w;
}
```

The goal of this page is to understand how Eigen compiles it, assuming that SSE2 vectorization is enabled (GCC option -msse2).

Why it's interesting

Maybe you think, that the above example program is so simple, that compiling it shouldn't involve anything interesting. So before starting, let us explain what is nontrivial in compiling it correctly - that is, producing optimized code - so that the complexity of Eigen, that we'll explain here, is really useful.

Look at the line of code

```
u = v + w;    //    (*)
```

The first important thing about compiling it, is that the arrays should be traversed only once, like

```
for(int i = 0; i < size; i++) u[i] = v[i] + w[i];
```

The problem is that if we make a naive C++ library where the VectorXf class has an operator+ returning a VectorXf, then the line of code (*) will amount to:

```
VectorXf tmp = v + w;
VectorXf u = tmp;
```

Obviously, the introduction of the temporary *tmp* here is useless. It has a very bad effect on performance, first because the creation of *tmp* requires a dynamic memory allocation in this context, and second as there are now two for loops:

```
for(int i = 0; i < size; i++) tmp[i] = v[i] + w[i];
for(int i = 0; i < size; i++) u[i] = tmp[i];
```

Traversing the arrays twice instead of once is terrible for performance, as it means that we do many redundant memory accesses.

The second important thing about compiling the above program, is to make correct use of SSE2 instructions. Notice that Eigen also supports AltiVec and that all the discussion that we make here applies also to AltiVec.

SSE2, like AltiVec, is a set of instructions allowing to perform computations on packets of 128 bits at once. Since a float is 32 bits, this means that SSE2 instructions can handle 4 floats at once. This means that, if correctly used, they can make our computation go up to 4x faster.

However, in the above program, we have chosen size=50, so our vectors consist of 50 float's, and 50 is not a multiple of 4. This means that we cannot hope to do all of that computation using SSE2 instructions. The second best thing, to which we should aim, is to handle the 48 first coefficients with SSE2 instructions, since 48 is the biggest multiple of 4 below 50, and then handle separately, without SSE2, the 49th and 50th coefficients. Something like this:

```
for(int i = 0; i < 4*(size/4); i+=4) u.packet(i)  = v.packet(i) + w.packet(i);
for(int i = 4*(size/4); i < size; i++) u[i] = v[i] + w[i];
```

So let us look line by line at our example program, and let's follow **Eigen** as it compiles it.

Constructing vectors

Let's analyze the first line:

```
Eigen::VectorXf u(size), v(size), w(size);
```

First of all, VectorXf is the following typedef:

```
typedef Matrix<float, Dynamic, 1> VectorXf;
```

The class template **Matrix** is declared in src/Core/util/ForwardDeclarations.h with 6 template parameters, but the last 3 are automatically determined by the first 3. So you don't need to worry about them for now. Here, **Matrix**<float, Dynamic, 1> means a matrix of floats, with a dynamic number of rows and 1 column.

The **Matrix** class inherits a base class, **MatrixBase**. Don't worry about it, for now it suffices to say that **MatrixBase** is what unifies matrices/vectors and all the expressions types - more on that below.

When we do

```
Eigen::VectorXf u(size);
```

the constructor that is called is Matrix::Matrix(int), in **src/Core/Matrix.h**. Besides some assertions, all it does is to construct the *m_storage* member, which is of type DenseStorage<float, Dynamic, Dynamic, 1>.

You may wonder, isn't it overengineering to have the storage in a separate class? The reason is that the **Matrix** class template covers all kinds of matrices and vector: both fixed-size and dynamic-size. The storage method is not the same in these two cases. For fixed-size, the matrix coefficients are stored as a

plain member array. For dynamic-size, the coefficients will be stored as a pointer to a dynamically-allocated array. Because of this, we need to abstract storage away from the **Matrix** class. That's DenseStorage.

Let's look at this constructor, in **src/Core/DenseStorage.h**. You can see that there are many partial template specializations of DenseStorages here, treating separately the cases where dimensions are Dynamic or fixed at compile-time. The partial specialization that we are looking at is:

```
template<typename T, int Cols_> class DenseStorage<T, Dynamic, Dynamic, Cols_>
```

Here, the constructor called is DenseStorage::DenseStorage(int size, int rows, int columns) with size=50, rows=50, columns=1.

Here is this constructor:

```
inline DenseStorage(int size, int rows, int) : m_data(internal::aligned_new<T>
      (size)), m_rows(rows) {}
```

Here, the *m_data* member is the actual array of coefficients of the matrix. As you see, it is dynamically allocated. Rather than calling new[] or malloc(), as you can see, we have our own internal::aligned_new defined in **src/Core/util/Memory.h**. What it does is that if vectorization is enabled, then it uses a platform-specific call to allocate a 128-bit-aligned array, as that is very useful for vectorization with both SSE2 and AltiVec. If vectorization is disabled, it amounts to the standard new[].

As you can see, the constructor also sets the *m_rows* member to *size*. Notice that there is no *m_columns* member: indeed, in this partial specialization of DenseStorage, we know the number of columns at compile-time, since the Cols_ template parameter is different from Dynamic. Namely, in our case, Cols_ is 1, which is to say that our vector is just a matrix with 1 column. Hence, there is no need to store the number of columns as a runtime variable.

When you call **VectorXf::data()** to get the pointer to the array of coefficients, it returns DenseStorage::data() which returns the *m_data* member.

When you call **VectorXf::size()** to get the size of the vector, this is actually a method in the base class **MatrixBase**. It determines that the vector is a column-vector, since ColsAtCompileTime==1 (this comes from the template parameters in the typedef VectorXf). It deduces that the size is the number of rows, so it returns **VectorXf::rows()**, which returns DenseStorage::rows(), which returns the *m_rows* member, which was set to *size* by the constructor.

Construction of the sum expression

Now that our vectors are constructed, let's move on to the next line:

```
u = v + w;
```

The executive summary is that operator+ returns a "sum of vectors" expression, but doesn't actually perform the computation. It is the operator=, whose call occurs thereafter, that does the computation.

Let us now see what **Eigen** does when it sees this:

```
v + w
```

Here, v and w are of type VectorXf, which is a typedef for a specialization of **Matrix** (as we explained above), which is a subclass of **MatrixBase**. So what is being called is

```
MatrixBase::operator+(const MatrixBase&)
```

The return type of this operator is

```
CwiseBinaryOp<internal::scalar_sum_op<float>, VectorXf, VectorXf>
```

The **CwiseBinaryOp** class is our first encounter with an expression template. As we said, the operator+ doesn't by itself perform any computation, it just returns an abstract "sum of vectors" expression. Since there are also "difference of vectors" and "coefficient-wise product of vectors" expressions, we unify them all as "coefficient-wise binary operations", which we abbreviate as "CwiseBinaryOp". "Coefficient-wise" means that the operations is performed coefficient by coefficient. "binary" means that there are two operands - we are adding two vectors with one another.

Now you might ask, what if we did something like

```
v + w + u;
```

The first v + w would return a **CwiseBinaryOp** as above, so in order for this to compile, we'd need to define an operator+ also in the class **CwiseBinaryOp**... at this point it starts looking like a nightmare: are we going to have to define all operators in each of the expression classes (as you guessed, **CwiseBinaryOp** is only one of many) ? This looks like a dead end!

The solution is that **CwiseBinaryOp** itself, as well as **Matrix** and all the other expression types, is a subclass of **MatrixBase**. So it is enough to define once and for all the operators in class **MatrixBase**.

Since **MatrixBase** is the common base class of different subclasses, the aspects that depend on the subclass must be abstracted from **MatrixBase**. This is called polymorphism.

The classical approach to polymorphism in C++ is by means of virtual functions. This is dynamic polymorphism. Here we don't want dynamic polymorphism because the whole design of **Eigen** is based around the assumption that all the complexity, all the abstraction, gets resolved at compile-time. This is crucial: if the abstraction can't get resolved at compile-time, **Eigen**'s compile-time

optimization mechanisms become useless, not to mention that if that abstraction has to be resolved at runtime it'll incur an overhead by itself.

Here, what we want is to have a single class **MatrixBase** as the base of many subclasses, in such a way that each **MatrixBase** object (be it a matrix, or vector, or any kind of expression) knows at compile-time (as opposed to run-time) of which particular subclass it is an object (i.e. whether it is a matrix, or an expression, and what kind of expression).

The solution is the Curiously Recurring Template Pattern. Let's do the break now. Hopefully you can read this wikipedia page during the break if needed, but it won't be allowed during the exam.

In short, **MatrixBase** takes a template parameter *Derived*. Whenever we define a subclass Subclass, we actually make Subclass inherit **MatrixBase**<Subclass>. The point is that different subclasses inherit different **MatrixBase** types. Thanks to this, whenever we have an object of a subclass, and we call on it some **MatrixBase** method, we still remember even from inside the **MatrixBase** method which particular subclass we're talking about.

This means that we can put almost all the methods and operators in the base class **MatrixBase**, and have only the bare minimum in the subclasses. If you look at the subclasses in **Eigen**, like for instance the **CwiseBinaryOp** class, they have very few methods. There are coeff() and sometimes coeffRef() methods for access to the coefficients, there are rows() and cols() methods returning the number of rows and columns, but there isn't much more than that. All the meat is in **MatrixBase**, so it only needs to be coded once for all kinds of expressions, matrices, and vectors.

So let's end this digression and come back to the piece of code from our example program that we were currently analyzing,

```
v + w
```

Now that **MatrixBase** is a good friend, let's write fully the prototype of the operator+ that gets called here (this code is from **src/Core/MatrixBase.h**):

```
template<typename Derived>
class MatrixBase
{
  // ...

  template<typename OtherDerived>
  const CwiseBinaryOp<internal::scalar_sum_op<typename
      internal::traits<Derived>::Scalar>, Derived, OtherDerived>
  operator+(const MatrixBase<OtherDerived> &other) const;

  // ...
};
```

Here of course, *Derived* and *OtherDerived* are VectorXf.

As we said, **CwiseBinaryOp** is also used for other operations such as substration, so it takes another template parameter determining the operation that will be applied to coefficients. This template parameter is a functor, that is, a class in which we have an operator() so it behaves like a function. Here, the functor used is internal::scalar_sum_op. It is defined in src/Core/Functors.h.

Let us now explain the internal::traits here. The internal::scalar_sum_op class takes one template parameter: the type of the numbers to handle. Here of course we want to pass the scalar type (a.k.a. numeric type) of VectorXf, which is float. How do we determine which is the scalar type of *Derived* ? Throughout **Eigen**, all matrix and expression types define a typedef *Scalar* which gives its scalar type. For example, VectorXf::Scalar is a typedef for float. So here, if life was easy, we could find the numeric type of *Derived* as just

```
typename Derived::Scalar
```

Unfortunately, we can't do that here, as the compiler would complain that the type Derived hasn't yet been defined. So we use a workaround: in src/Core/util/ForwardDeclarations.h, we declared (not defined!) all our subclasses, like **Matrix**, and we also declared the following class template:

```
template<typename T> struct internal::traits;
```

In **src/Core/Matrix.h**, right *before* the definition of class **Matrix**, we define a partial specialization of internal::traits for T=**Matrix**<any template parameters>. In this specialization of internal::traits, we define the Scalar typedef. So when we actually define **Matrix**, it is legal to refer to "typename internal::traits\<Matrix\>::Scalar".

Anyway, we have declared our operator+. In our case, where *Derived* and *OtherDerived* are VectorXf, the above declaration amounts to:

```
class MatrixBase<VectorXf>
{
  // ...

  const CwiseBinaryOp<internal::scalar_sum_op<float>, VectorXf, VectorXf>
  operator+(const MatrixBase<VectorXf> &other) const;

  // ...
};
```

Let's now jump to **src/Core/CwiseBinaryOp.h** to see how it is defined. As you can see there, all it does is to return a **CwiseBinaryOp** object, and this object is just storing references to the left-hand-side and right-hand-side expressions - here, these are the vectors *v* and *w*. Well, the **CwiseBinaryOp** object is also storing an instance of the (empty) functor class, but you shouldn't worry about it as that is a minor implementation detail.

Thus, the operator+ hasn't performed any actual computation. To summarize, the operation *v* + *w* just returned an object of type **CwiseBinaryOp** which did nothing

else than just storing references to *v* and *w*.

The assignment

PLEASE HELP US IMPROVING THIS SECTION. This page reflects how Eigen worked until 3.2, but since Eigen 3.3 the assignment is more sophisticated as it involves an Assignment expression, and the creation of so called evaluator which are responsible for the evaluation of each kind of expressions.

At this point, the expression *v* + *w* has finished evaluating, so, in the process of compiling the line of code

```
u = v + w;
```

we now enter the operator=.

What operator= is being called here? The vector u is an object of class VectorXf, i.e. **Matrix**. In **src/Core/Matrix.h**, inside the definition of class **Matrix**, we see this:

```
template<typename OtherDerived>
inline Matrix& operator=(const MatrixBase<OtherDerived>& other)
{
  eigen_assert(m_storage.data()!=0 && "you cannot use operator= with a non
      initialized matrix (instead use set()");
  return Base::operator=(other.derived());
}
```

Here, Base is a typedef for **MatrixBase**<**Matrix**>. So, what is being called is the operator= of **MatrixBase**. Let's see its prototype in **src/Core/MatrixBase.h**:

```
template<typename OtherDerived>
Derived& operator=(const MatrixBase<OtherDerived>& other);
```

Here, *Derived* is VectorXf (since u is a VectorXf) and *OtherDerived* is **CwiseBinaryOp**. More specifically, as explained in the previous section, *OtherDerived* is:

```
CwiseBinaryOp<internal::scalar_sum_op<float>, VectorXf, VectorXf>
```

So the full prototype of the operator= being called is:

```
VectorXf& MatrixBase<VectorXf>::operator=(const
      MatrixBase<CwiseBinaryOp<internal::scalar_sum_op<float>, VectorXf,
      VectorXf> > & other);
```

This operator= literally reads "copying a sum of two VectorXf's into another VectorXf".

Let's now look at the implementation of this operator=. It resides in the file **src/Core/Assign.h**.

What we can see there is:

```
template<typename Derived>
template<typename OtherDerived>
inline Derived& MatrixBase<Derived>
  ::operator=(const MatrixBase<OtherDerived>& other)
{
  return internal::assign_selector<Derived,OtherDerived>::run(derived(),
      other.derived());
}
```

OK so our next task is to understand internal::assign_selector :)

Here is its declaration (all that is still in the same file **src/Core/Assign.h**)

```
template<typename Derived, typename OtherDerived,
         bool EvalBeforeAssigning = int(OtherDerived::Flags) &
      EvalBeforeAssigningBit,
         bool NeedToTranspose = Derived::IsVectorAtCompileTime
                && OtherDerived::IsVectorAtCompileTime
                && int(Derived::RowsAtCompileTime) ==
      int(OtherDerived::ColsAtCompileTime)
                && int(Derived::ColsAtCompileTime) ==
      int(OtherDerived::RowsAtCompileTime)
                && int(Derived::SizeAtCompileTime) != 1>
struct internal::assign_selector;
```

So internal::assign_selector takes 4 template parameters, but the 2 last ones are automatically determined by the 2 first ones.

EvalBeforeAssigning is here to enforce the EvalBeforeAssigningBit. As explained here, certain expressions have this flag which makes them automatically evaluate into temporaries before assigning them to another expression. This is the case of the **Product** expression, in order to avoid strange aliasing effects when doing "m = m * m;" However, of course here our **CwiseBinaryOp** expression doesn't have the EvalBeforeAssigningBit: we said since the beginning that we didn't want a temporary to be introduced here. So if you go to **src/Core/CwiseBinaryOp.h**, you'll see that the Flags in internal::traits<**CwiseBinaryOp**> don't include the EvalBeforeAssigningBit. The Flags member of **CwiseBinaryOp** is then imported from the internal::traits by the EIGEN_GENERIC_PUBLIC_INTERFACE macro. Anyway, here the template parameter EvalBeforeAssigning has the value false.

NeedToTranspose is here for the case where the user wants to copy a row-vector into a column-vector. We allow this as a special exception to the general rule that in assignments we require the dimesions to match. Anyway, here both the left-hand and right-hand sides are column vectors, in the sense that ColsAtCompileTime is equal to 1. So NeedToTranspose is false too.

So, here we are in the partial specialization:

```
internal::assign_selector<Derived, OtherDerived, false, false>
```

Here's how it is defined:

```
template<typename Derived, typename OtherDerived>
struct internal::assign_selector<Derived,OtherDerived,false,false> {
```

```
  static Derived& run(Derived& dst, const OtherDerived& other) { return
      dst.lazyAssign(other.derived()); }
};
```

OK so now our next job is to understand how lazyAssign works :)

```
template<typename Derived>
template<typename OtherDerived>
inline Derived& MatrixBase<Derived>
  ::lazyAssign(const MatrixBase<OtherDerived>& other)
{
  EIGEN_STATIC_ASSERT_SAME_MATRIX_SIZE(Derived,OtherDerived)
  eigen_assert(rows() == other.rows() && cols() == other.cols());
  internal::assign_impl<Derived,
      OtherDerived>::run(derived(),other.derived());
  return derived();
}
```

What do we see here? Some assertions, and then the only interesting line is:

```
internal::assign_impl<Derived, OtherDerived>::run(derived(),other.derived());
```

OK so now we want to know what is inside internal::assign_impl.

Here is its declaration:

```
template<typename Derived1, typename Derived2,
         int Vectorization = internal::assign_traits<Derived1,
      Derived2>::Vectorization,
         int Unrolling = internal::assign_traits<Derived1,
      Derived2>::Unrolling>
struct internal::assign_impl;
```

Again, internal::assign_selector takes 4 template parameters, but the 2 last ones are automatically determined by the 2 first ones.

These two parameters *Vectorization* and *Unrolling* are determined by a helper class internal::assign_traits. Its job is to determine which vectorization strategy to use (that is *Vectorization*) and which unrolling strategy to use (that is *Unrolling*).

We'll not enter into the details of how these strategies are chosen (this is in the implementation of internal::assign_traits at the top of the same file). Let's just say that here *Vectorization* has the value *LinearVectorization*, and *Unrolling* has the value *NoUnrolling* (the latter is obvious since our vectors have dynamic size so there's no way to unroll the loop at compile-time).

So the partial specialization of internal::assign_impl that we're looking at is:

```
internal::assign_impl<Derived1, Derived2, LinearVectorization, NoUnrolling>
```

Here is how it's defined:

```
template<typename Derived1, typename Derived2>
struct internal::assign_impl<Derived1, Derived2, LinearVectorization,
      NoUnrolling>
{
```

```
  static void run(Derived1 &dst, const Derived2 &src)
  {
    const int size = dst.size();
    const int packetSize = internal::packet_traits<typename
      Derived1::Scalar>::size;
    const int alignedStart =
      internal::assign_traits<Derived1,Derived2>::DstIsAligned ? 0
                              : internal::first_aligned(&dst.coeffRef(0), size);
    const int alignedEnd = alignedStart + ((size-
      alignedStart)/packetSize)*packetSize;

    for(int index = 0; index < alignedStart; index++)
      dst.copyCoeff(index, src);

    for(int index = alignedStart; index < alignedEnd; index += packetSize)
    {
      dst.template copyPacket<Derived2, Aligned,
      internal::assign_traits<Derived1,Derived2>::SrcAlignment>(index, src);
    }

    for(int index = alignedEnd; index < size; index++)
      dst.copyCoeff(index, src);
  }
};
```

Here's how it works. *LinearVectorization* means that the left-hand and right-hand side expression can be accessed linearly i.e. you can refer to their coefficients by one integer *index*, as opposed to having to refer to its coefficients by two integers *row*, *column*.

As we said at the beginning, vectorization works with blocks of 4 floats. Here, *PacketSize* is 4.

There are two potential problems that we need to deal with:

- first, vectorization works much better if the packets are 128-bit-aligned. This is especially important for write access. So when writing to the coefficients of *dst*, we want to group these coefficients by packets of 4 such that each of these packets is 128-bit-aligned. In general, this requires to skip a few coefficients at the beginning of *dst*. This is the purpose of *alignedStart*. We then copy these first few coefficients one by one, not by packets. However, in our case, the *dst* expression is a VectorXf and remember that in the construction of the vectors we allocated aligned arrays. Thanks to *DstIsAligned*, **Eigen** remembers that without having to do any runtime check, so *alignedStart* is zero and this part is avoided altogether.
- second, the number of coefficients to copy is not in general a multiple of *packetSize*. Here, there are 50 coefficients to copy and *packetSize* is 4. So we'll have to copy the last 2 coefficients one by one, not by packets. Here, *alignedEnd* is 48.

Now come the actual loops.

First, the vectorized part: the 48 first coefficients out of 50 will be copied by packets of 4:

```
for(int index = alignedStart; index < alignedEnd; index += packetSize)
{
  dst.template copyPacket<Derived2, Aligned,
      internal::assign_traits<Derived1,Derived2>::SrcAlignment>(index, src);
}
```

What is copyPacket? It is defined in src/Core/Coeffs.h:

```
template<typename Derived>
template<typename OtherDerived, int StoreMode, int LoadMode>
inline void MatrixBase<Derived>::copyPacket(int index, const
      MatrixBase<OtherDerived>& other)
{
  eigen_internal_assert(index ⩾ 0 && index < size());
  derived().template writePacket<StoreMode>(index,
    other.derived().template packet<LoadMode>(index));
}
```

OK, what are writePacket() and packet() here?

First, writePacket() here is a method on the left-hand side VectorXf. So we go to **src/Core/Matrix.h** to look at its definition:

```
template<int StoreMode>
inline void writePacket(int index, const PacketScalar& x)
{
  internal::pstoret<Scalar, PacketScalar, StoreMode>(m_storage.data() + index,
      x);
}
```

Here, *StoreMode* is ***Aligned***, indicating that we are doing a 128-bit-aligned write access, *PacketScalar* is a type representing a "SSE packet of 4 floats" and internal::pstoret is a function writing such a packet in memory. Their definitions are architecture-specific, we find them in **src/Core/arch/SSE/PacketMath.h**:

The line in **src/Core/arch/SSE/PacketMath.h** that determines the PacketScalar type (via a typedef in **Matrix.h**) is:

```
template<> struct internal::packet_traits<float>  { typedef __m128  type; enum
      {size=4}; };
```

Here, __m128 is a SSE-specific type. Notice that the enum *size* here is what was used to define *packetSize* above.

And here is the implementation of internal::pstoret:

```
template<> inline void internal::pstore(float*  to, const __m128&  from) {
      _mm_store_ps(to, from); }
```

Here, __mm_store_ps is a SSE-specific intrinsic function, representing a single SSE instruction. The difference between internal::pstore and internal::pstoret

is that internal::pstoret is a dispatcher handling both the aligned and unaligned cases, you find its definition in **src/Core/GenericPacketMath.h**:

```
template<typename Scalar, typename Packet, int LoadMode>
inline void internal::pstoret(Scalar* to, const Packet& from)
{
  if(LoadMode == Aligned)
    internal::pstore(to, from);
  else
    internal::pstoreu(to, from);
}
```

OK, that explains how writePacket() works. Now let's look into the packet() call. Remember that we are analyzing this line of code inside copyPacket():

```
derived().template writePacket<StoreMode>(index,
    other.derived().template packet<LoadMode>(index));
```

Here, *other* is our sum expression *v* + *w*. The .derived() is just casting from **MatrixBase** to the subclass which here is **CwiseBinaryOp**. So let's go to **src/Core/CwiseBinaryOp.h**:

```
class CwiseBinaryOp
{
  // ...
    template<int LoadMode>
    inline PacketScalar packet(int index) const
    {
      return m_functor.packetOp(m_lhs.template packet<LoadMode>(index),
      m_rhs.template packet<LoadMode>(index));
    }
};
```

Here, *m_lhs* is the vector *v*, and *m_rhs* is the vector *w*. So the packet() function here is Matrix::packet(). The template parameter *LoadMode* is ***Aligned***. So we're looking at

```
class Matrix
{
  // ...
    template<int LoadMode>
    inline PacketScalar packet(int index) const
    {
      return internal::ploadt<Scalar, LoadMode>(m_storage.data() + index);
    }
};
```

We let you look up the definition of internal::ploadt in **GenericPacketMath.h** and the internal::pload in **src/Core/arch/SSE/PacketMath.h**. It is very similar to the above for internal::pstore.

Let's go back to CwiseBinaryOp::packet(). Once the packets from the vectors *v* and *w* have been returned, what does this function do? It calls m_functor.packetOp() on them. What is m_functor? Here we must remember what particular template specialization of **CwiseBinaryOp** we're dealing with:

```
CwiseBinaryOp<internal::scalar_sum_op<float>, VectorXf, VectorXf>
```

So m_functor is an object of the empty class internal::scalar_sum_op<float>. As we mentioned above, don't worry about why we constructed an object of this empty class at all - it's an implementation detail, the point is that some other functors need to store member data.

Anyway, internal::scalar_sum_op is defined in src/Core/Functors.h:

```
template<typename Scalar> struct internal::scalar_sum_op EIGEN_EMPTY_STRUCT {
  inline const Scalar operator() (const Scalar& a, const Scalar& b) const {
      return a + b; }
  template<typename PacketScalar>
  inline const PacketScalar packetOp(const PacketScalar& a, const
      PacketScalar& b) const
  { return internal::padd(a,b); }
};
```

As you can see, all what packetOp() does is to call internal::padd on the two packets. Here is the definition of internal::padd from **src/Core/arch/SSE/PacketMath.h**:

```
template<> inline __m128  internal::padd(const __m128&  a, const __m128&  b) {
      return _mm_add_ps(a,b); }
```

Here, _mm_add_ps is a SSE-specific intrinsic function, representing a single SSE instruction.

To summarize, the loop

```
for(int index = alignedStart; index < alignedEnd; index += packetSize)
{
  dst.template copyPacket<Derived2, Aligned,
      internal::assign_traits<Derived1,Derived2>::SrcAlignment>(index, src);
}
```

has been compiled to the following code: for *index* going from 0 to the 11 (= 48/4 - 1), read the i-th packet (of 4 floats) from the vector v and the i-th packet from the vector w using two __mm_load_ps SSE instructions, then add them together using a __mm_add_ps instruction, then store the result using a __mm_store_ps instruction.

There remains the second loop handling the last few (here, the last 2) coefficients:

```
for(int index = alignedEnd; index < size; index++)
  dst.copyCoeff(index, src);
```

However, it works just like the one we just explained, it is just simpler because there is no SSE vectorization involved here. copyPacket() becomes copyCoeff(), packet() becomes coeff(), writePacket() becomes coeffRef(). If you followed us this far, you can probably understand this part by yourself.

We see that all the C++ abstraction of **Eigen** goes away during compilation and that we indeed are precisely controlling which assembly instructions we emit.

Such is the beauty of C++! Since we have such precise control over the emitted assembly instructions, but such complex logic to choose the right instructions, we can say that **Eigen** really behaves like an optimizing compiler. If you prefer, you could say that **Eigen** behaves like a script for the compiler. In a sense, C++ template metaprogramming is scripting the compiler – and it's been shown that this scripting language is Turing-complete. See Wikipedia.