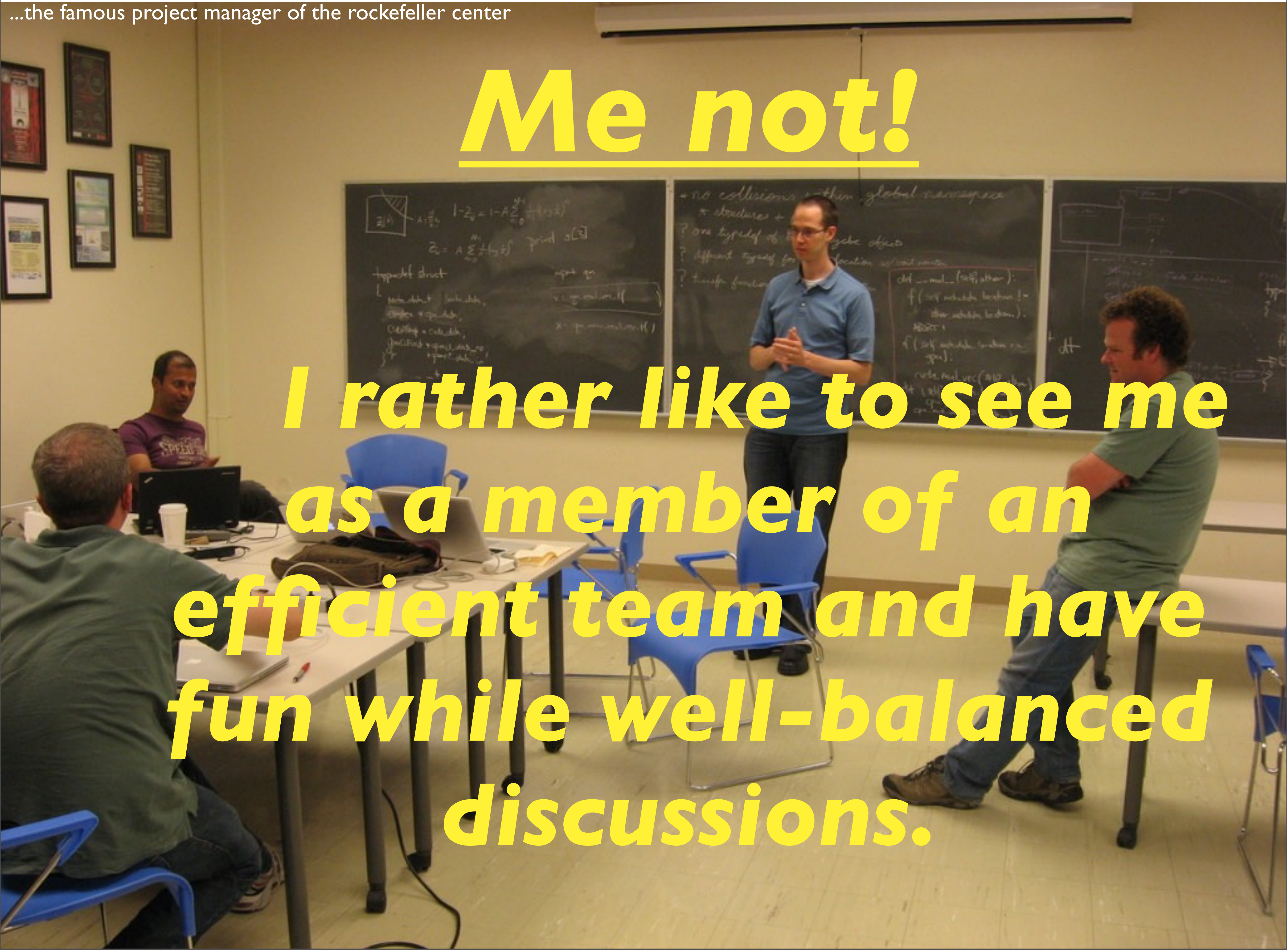
"I like having a lot of people against me…"

–JOHN R. TODD



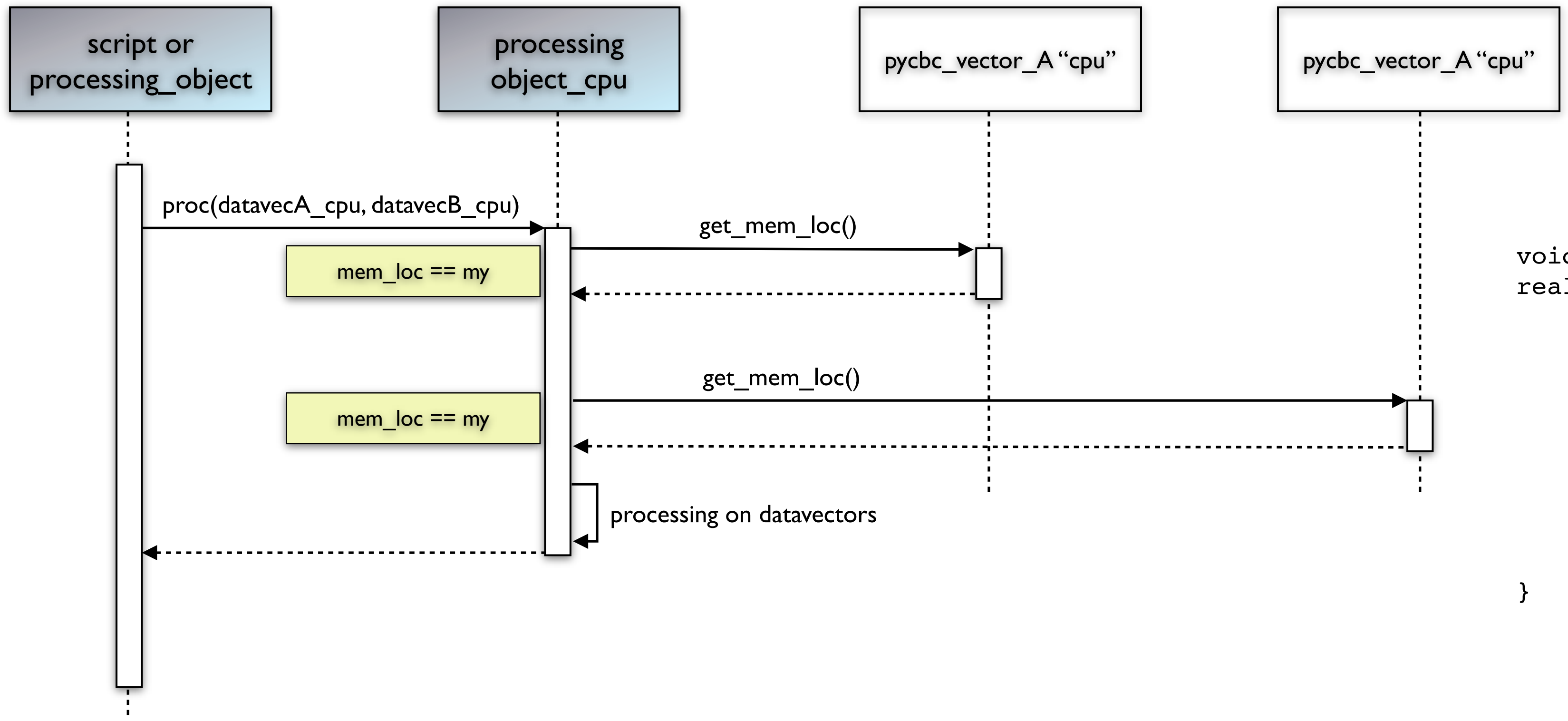
Me not!

I rather like to see me as a member of an efficient team and have fun while well-balanced discussions.

Content

- automagic (vs.:)
- coherent object oriented python control layer
- discussons/threads

Device memory flow to and from cpu with "automagic" get_mem_loc function and generic "void" pycbc vector

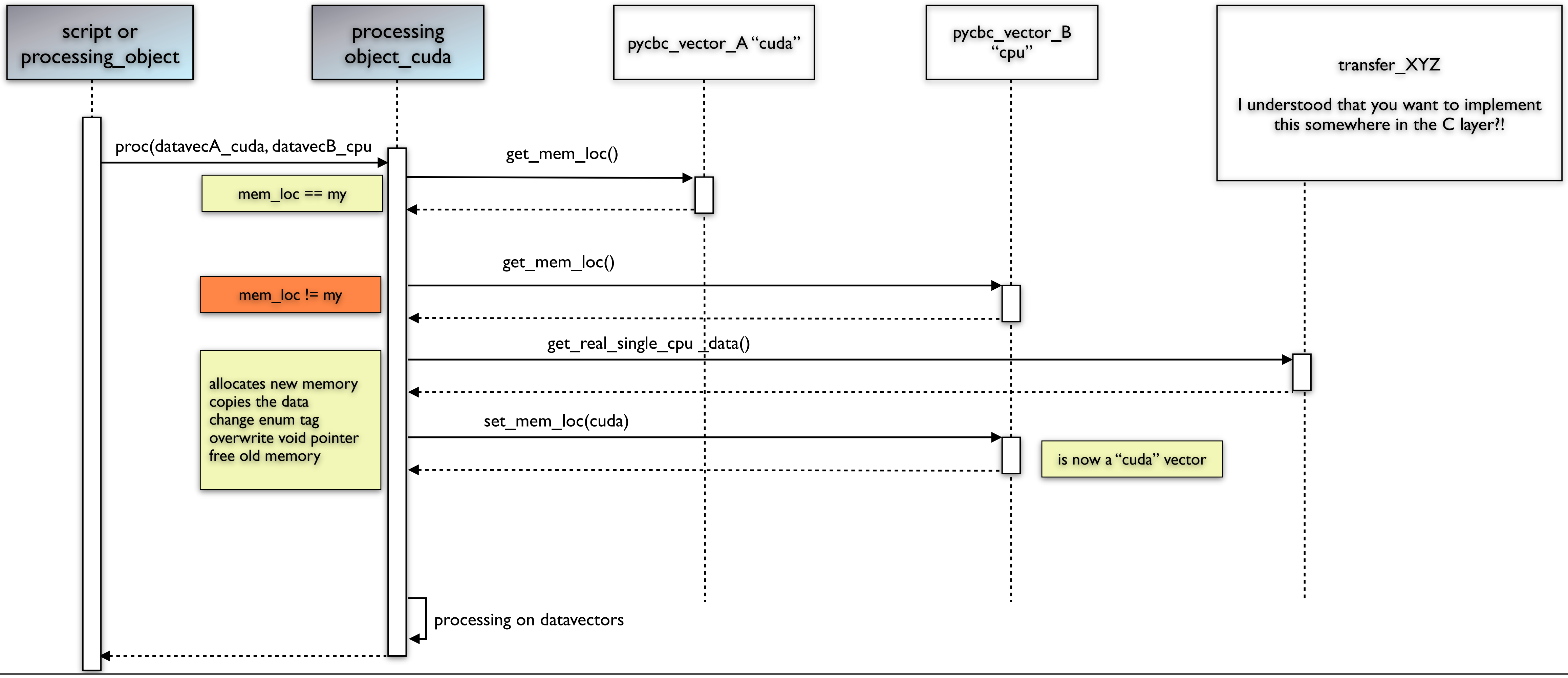


```
void MyFavoriteFunction(real_single_datavector *input,
real_double_datavector *output, ...){
      float *input_data;
      double *output_data;

      input_data = get_real_single_cpu_data(input);
      output_data = get_real_double_cpu_data(output);

        ...

      return;
}
```



There are some risks within this concept

- *get_ - functions shall do the appropriate typecast*
  - *where do they live?*
  - *how do they know the desired type?*

- *They do the memory transfer if mem_loc doesn't fit (currently they throw an error if they are not capable of doing this)*

  - *Let me resume:They:*
    - *probably allocate device memory*
    - *do a cpu-device transfer*
    - *update the mem_loc enum*
    - *update the pointer*
    - *free the memory*

who is in charge to free this memory at which time?

all permutations of data transfer in one function with switch case statements?

who allocated this memory?

allocation and freeing of memory is decoupled. This is dangerous in terms of robustness. A memory factory module could solve the problem (like in lalsuite)

"automagic" - what is this? Why don't you want to use "polymorphism" - a strong, clean and well accepted feature of object oriented design in millions of software projects?

But I agree on the datavector naming topic:

```
# I agree on changing the all too generic naming of datavectors in the C layer
# to real_vector_single_cpu_t real_vector_single_cuda_t real_vector_single_opencl_t
# and so on for the other data types

The data transfer functions in the C layer would have to distinguish the datavectors by
name. Also it makes certain decisions at the “data_in()” functions in the Python layer
easier.
```

I like: The object oriented python control layer

- Coherent object oriented design

- Keep the former made main requirements of the inspiral.c redesign in mind:

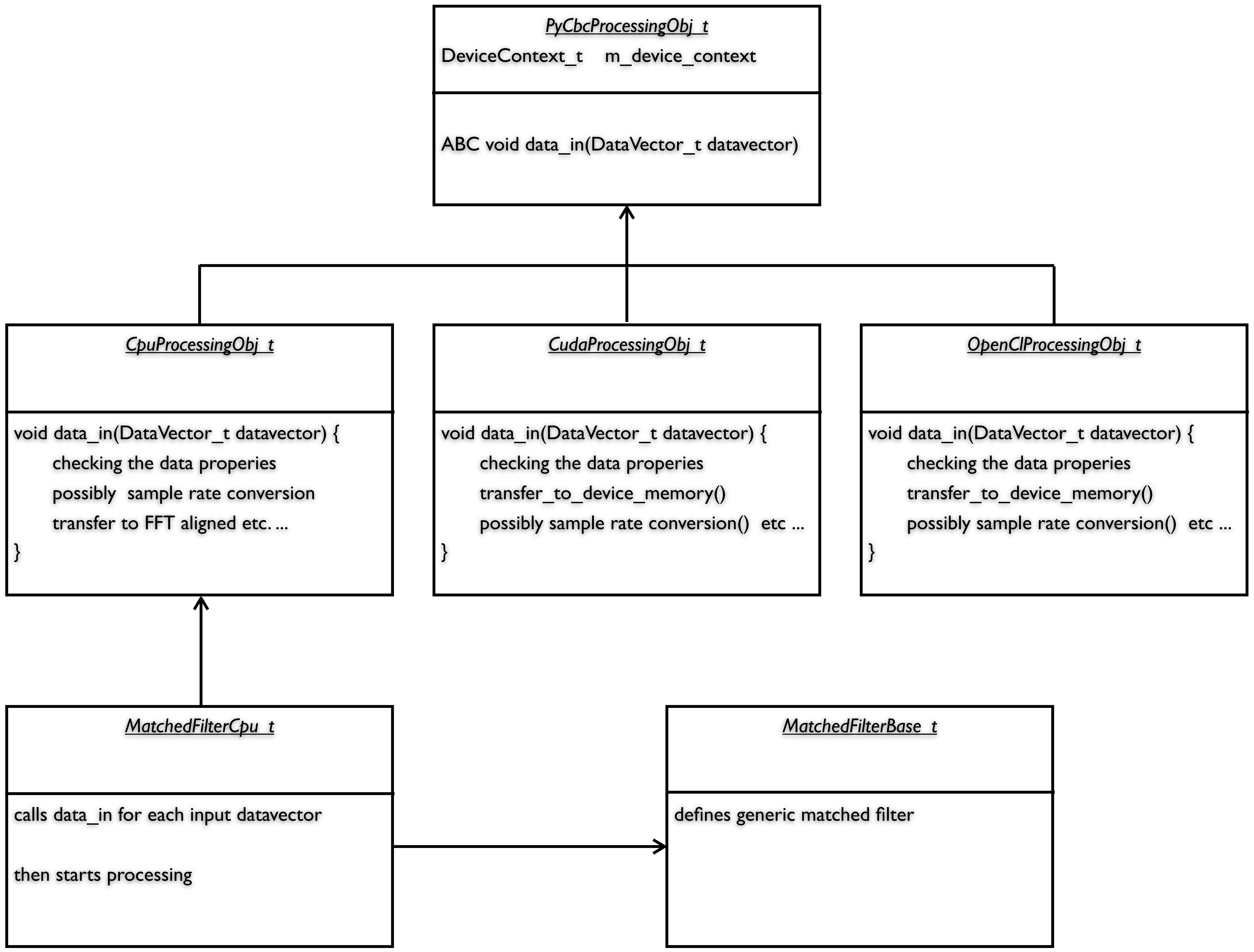
  - modularity - open for extension, closed for modification - clear datastructures - self explanatory - test driven development by unittest

- The distribution of responsibilities is very clear

  - The pipeline, the top layer algorithm, dataflow and memory management is exclusive done in the python layer (Btw. it can not be done in the C layer because plain ansi C is not object oriented)
  - All high performance computing is done or issued to GPUs in the C layer. The C layer and it's underlying processing architecture shall be seen as Coprocessors. They are attached to the python control layer in a master-slave-style

- Programmers of GPU/CPU C layer functions are forced to partition their algorithms into clean functions (that do only one thing) that work upon the data structures that are given to them by the object oriented python framework. Note that they are free to allocate their own internal memory if they really need to but "datavectors" belong to the framework and are managed by the framework.

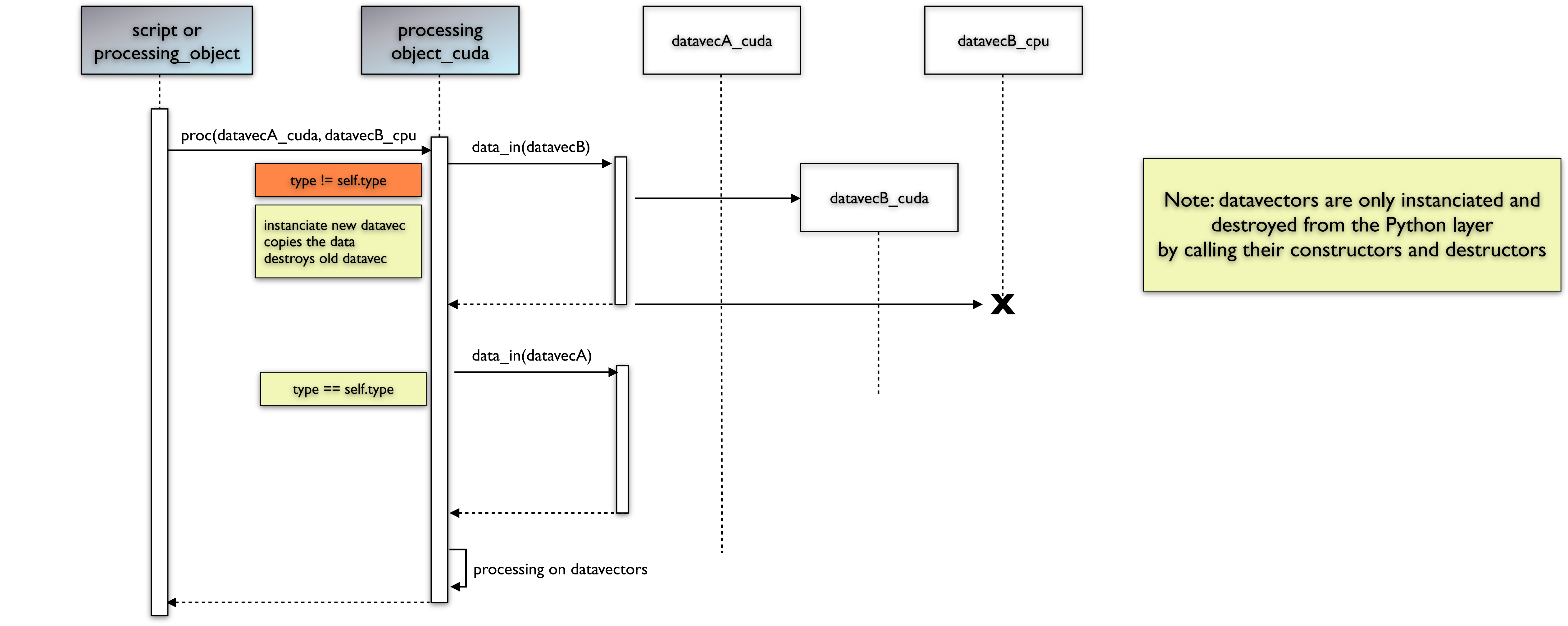
Object oriented python control layer - inheritance for example matched filter (text description follows)


PyCbcProcessingObj_t
DeviceContext_t m_device_context
ABC void data_in(DataVector_t datavector)
CpuProcessingObj_t
void data_in(DataVector_t datavector) {
checking the data properies
possibly sample rate conversion
transfer to FFT aligned etc. ...
}
CudaProcessingObj_t
void data_in(DataVector_t datavector) {
checking the data properies
transfer_to_device_memory()
possibly sample rate conversion() etc ...
}
OpenClProcessingObj_t
void data_in(DataVector_t datavector) {
checking the data properies
transfer_to_device_memory()
possibly sample rate conversion() etc ...
}
MatchedFilterCpu_t
calls data_in for each input datavector
then starts processing
MatchedFilterBase_t
defines generic matched filter

Object oriented python control layer - inheritance in words

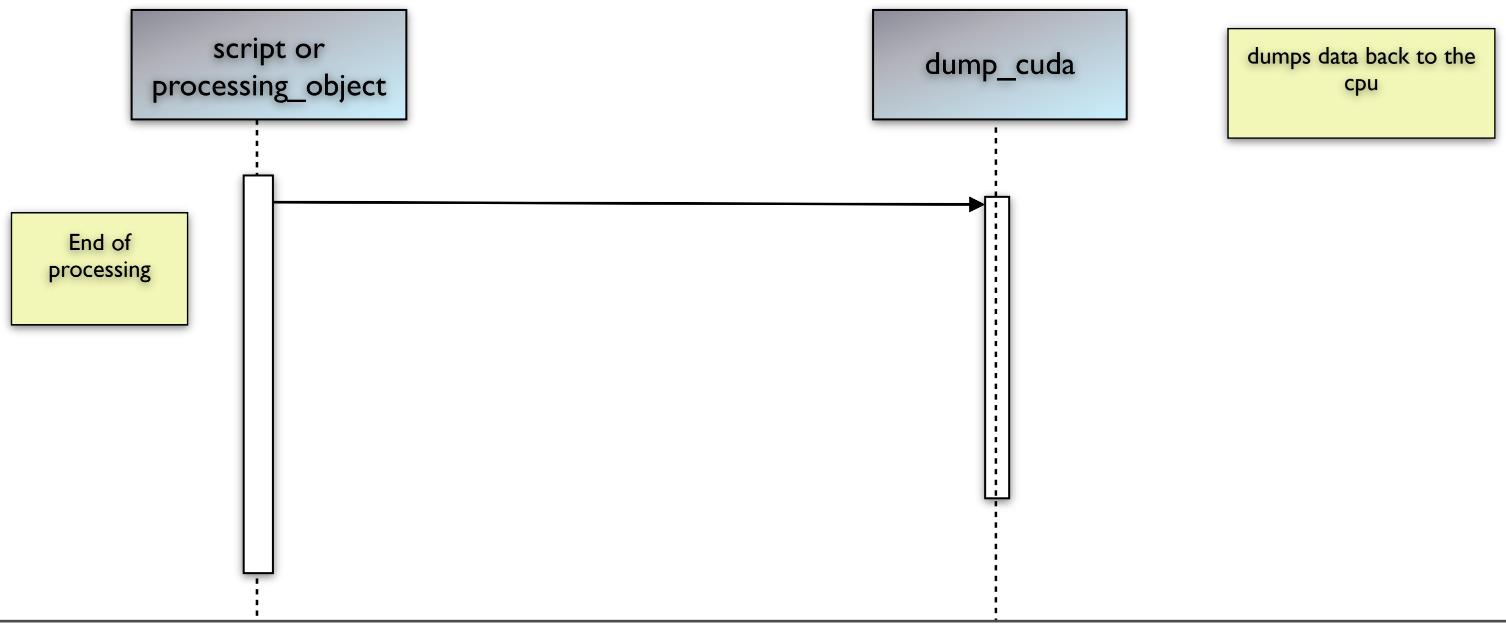
- Each processing object inherits from two base class systems
  - From pycbc_proc_obj it inherits it's framework related properties:
    - The context that holds the current device context
    - The data_in method that possibly converts data and transfers data to the objects processing architecture. A CudaProcessingObj_t for example knows it's proper transfer functionality because it _has_ the right method to transfer data to the GPU. The transfer methods are implemented only once. Each for Cuda, OpenCl etc. They are bound to the right processing object by inheritance, hence at compile time (no error prone switch case structures at runtime necessary). *If required we can also couple the transfer functions at runtime. Then we would have to make them members of the Context_t class.*
- From <algorithm>_base it inherits it's algorithm related properties
  - The <algorithm>_base class defines the interface of the processing object so that a matched filter, for example, always looks the same (_has_: generate_snr(self, stilde, htilde ,snr) and max(self, snr)) independent on which processing architecture it runs. Thereby providing that objects are interchangeable in terms of the processing architecture. This enables the "Decoupling algorithm from implementation" feature.

Device memory flow to and from cpu object orientated


script or processing_object
processing object_cuda
datavecA_cuda
datavecB_cpu
proc(datavecA_cuda, datavecB_cpu
data_in(datavecB)
type != self.type
instanciate new datavec copies the data destroys old datavec
datavecB_cuda
data_in(datavecA)
type == self.type
processing on datavectors

Note: datavectors are only instanciated and destroyed from the Python layer by calling their constructors and destructors

End of processing or testing



```
 All processing objects have to inherit from this base class via ...
lass PyCbcProcessingObj:

   __metaclass__ = ABCMeta

   def __init__(self, device_context):
       self._logger= logging.getLogger('pycbc.pycbc')
       self._devicecontext = device_context

   @abstractmethod
   def data_in(self, datavector):
       pass

 ... their correct derivative according to their processing architecture:
lass CpuProcessingObj(PyCbcProcessingObj):

   def __init__(self, device_context):

       super(CpuProcessingObj, self).__init__(device_context)

   # data_in() is to be called for every input datavector.
   # in case of an alien datavector (does not fit to self-architecture) data_in
   # create the new datavector, copies the data by calling the proper transfer
   # function in the C layer and set new_datavector = old_datavector
   # (thus destroy the old datavector)

   def data_in(self, datavector):

       print 'data_in of ' + repr(self) + ' called'
       print 'with ' + repr(datavector)

       if repr(datavector).find("datavectorcpu") >= 0:
           # it is one of us
           return datavector

       else:
           print 'aliendatavector found:'
           alien_repr_str= repr(datavector)
           print alien_repr_str
           # find correct new datatype. by parsing alien_repr_str.
           # and instanciate the correct thing, " cloning " from the alien
           new_arch_vector = real_vector_single_cpu_t(len(datavector), datavector.get_delta_x())

           # call the transfer function in the C layer
           # prototyping it here:
           for i in range(len(datavector)):
               new_arch_vector[i] = datavector[i]

           return new_arch_vector
```

```
class  GenSnrImplementationCpu(GenSnrImplementationBase):

    def __init__(self, owner_mfilt):

        self.__logger= logging.getLogger('pycbc.GenSnrImplementationCpu')
        self.__logger.debug("instanciated GenSnrImplementationCpu")

        super(GenSnrImplementationCpu, self).__init__(owner_mfilt)

    def generate_snr(self, context, snr, stilde, htilde):
        """
        Process matched filtering by generating snr timeseries \rho(t)
        """

        # for the purpose of testing data_in()
        self._aliendatavector = alien_datavector_t(len(snr), stilde.get_delta_x())
        print
        print "self._aliendatavector before data_in() : "
        print self._aliendatavector
        self._aliendatavector = self._owner_mfilt.data_in(self._aliendatavector)
        print "self._aliendatavector after data_in() : "
        print self._aliendatavector
        print "after data_in() call aliendatavector is now correct: "+ repr(self._aliendatavector)
        print

        print
        print "snr before data_in() : "
        print snr
        snr =   self._owner_mfilt.data_in(snr)
        print "snr after data_in() : "
        print snr
        print

        stilde= self._owner_mfilt.data_in(stilde)
        htilde= self._owner_mfilt.data_in(htilde)

        gen_snr_cpu(context, snr, stilde, htilde, self._owner_mfilt._q,
                    self._owner_mfilt._qtilde, 1.0, 1.0) # just for prototyping
                                                         # f_min and sigma_sq
                                                         # are magic numbers
```

```
self._aliendatavector before data_in() :
<real_vector_single_t, length 4096, data ptr 0x10095d200>
data_in of <pycbc.matchedfilter.matchedfilter_cpu.MatchedFilterCpu object at 0x10066fad0> called
with <pycbc.datavector.datavectoropencl.real_vector_single_t; proxy of <Swig Object of type 'real_vector_single_t *' at 0x100675270> >
aliendatavector found:
<pycbc.datavector.datavectoropencl.real_vector_single_t; proxy of <Swig Object of type 'real_vector_single_t *' at 0x100675270> >
self._aliendatavector after data_in() :
<real_vector_single_t, length 4096, data ptr 0x100961200>
after data_in() call aliendatavector is now correct: <pycbc.datavector.datavectorcpu.real_vector_single_t; proxy of <Swig Object of type 'real_vector_single_t *' at 0x100675240> >


snr before data_in() :
<real_vector_single_t, length 4096, data ptr 0x100941200>
data_in of <pycbc.matchedfilter.matchedfilter_cpu.MatchedFilterCpu object at 0x10066fad0> called
with <pycbc.datavector.datavectorcpu.real_vector_single_t; proxy of <Swig Object of type 'real_vector_single_t *' at 0x100675180> >
snr after data_in() :
<real_vector_single_t, length 4096, data ptr 0x100941200>
```

Python and only Python should manage datavectors

Josh: The important point is that even the automagic functions would at most ***modify*** a datavector struct, but they would not explicitly create or destroy it. To Python, the object has remained the same, and was created somewhere in they Python layer and will be destroyed by the garbage collector when it is no longer needed. So Python handles creation/ destruction of objects. But it is really the ***Python*** layer (more than the C) that doesn't care where the object lives: it's always just a data vector (that has a "type" of real single, etc) but where it lives may change as it flows through the pipeline. The GPU versions of processing functions or objects will of course call SWIG wrapped versions of the corresponding GPU C functions, which would in turn move the memory to where they needed it, so there is still control at the Python layer (based on how you import things at the beginning of the executable) of where things live. But sufficiently far in the future if all of the automagic stuff is implemented, it might be possible to string together a pipeline that moves things around from CPU to CUDA to OpenCL and back. Such a thing would probably be stupid and slow, but if you had the right hardware it would actually work.

Python always knows precisely WHERE and WHAT (kind of type and device memory_location) a datavector is: It can be queried with the repr() function and then checked against the current requirements:

```
print repr(strain_data.time_series)    # outputs:

<pycbc.datavector.datavectorcpu.real_vector_double_t;
  proxy of <Swig Object of type 'real_vector_double_t *' at
0x100520510> >
```

I like this very much!

- I agree to the point: We will create fancy pipelines where data will flow from CPUs to GPUs and vice versa. There might come even a mix of different GPU - languages (Cuda/OpenCL). Earlier then that I see partitioning of processing to multiple GPUs (multi - context) and a coherent analysis pipeline. Those two last issues are the most important future challenges.
- In my proposal there is no "automagic". Modification of datavectors is not allowed. There is only one layer that is responsible for the memory management. It is object oriented Python!

- Nothing will be allocated or freed in the C layer except in the (to be swig wrapped) constructors and destructors i.g.: new_real_vector_single_t() and delete_real_vector_single_t().

- It is absolutely clear who is in charge of creating and destroying. It is the datavector itself. This is one of the strongest features of OOD

- A datavector will be modified never in it's type or it's specification. Though it's data content might change. If the memory location doesn't fit a new datavector will be created (with the same specs but different memory location) The data will be copied, then the old datavector will be destroyed. This can be done by Python explicitly with the del() function.

- With Python an object will be precisely created when it is instanciated. Wether it will be created as a top object by the script or as a member of another object like i.g. the datavector: complex_vector_single_t "qtilde" as property of a Matched Filter. The object will be deleted when the owner object is being killed _or_ there is no reference to it anymore _or_ we explicitly destroy it with del(). This is absolutely what we want!

- The garbage collection of Python does the job for us. We don't have to implement memory management like in lalsuite. Additionally we will use explicit destruction of objects (like at the end of a context)

We have two different concepts - that's good competition - let the better win

Josh: So I do not favor swig wrapped methods that in Python explicitly check and move things around.

Nothing will be moved around in Python. Datavectors are being instanciated (born) and deleted (killed) by the control of the Python Layer. And they will be dispatched to the appropriate C layer functions. The Swig wrapped C layer functions appear in Python as members of so called Implementation Objects. So we are able to create Pipelines of Datavectors and Implementation-Objects which do the processing. PyCBC is a library basically composed of objects of those two classes that are fully interchangeable in terms of the underlying compute architecture (even at runtime!). By that we decouple algorithms from implementations!

The C layer will ***have*** to know where stuff lives, and it's cleaner to keep that knowledge in the C layer alone.

Nope. The C layer defines only Constructors and Destructors for the architecture depending objects like datavectors, device-contexts, FFT-Plans etc. And it provides plain C functions that do the processing, by probably calling kernels. All the management is done by object oriented Python. It was one of our main requirements to do the redesign of inspiral.c "Object Oriented". C is not "Object Oriented" and it is complicated and error prune to implement OO in plain C.

Otherwise, every single Python object or method we write has to handle the logic that Karsten displayed on his slides, checking where something lives and moving it around if it wants to. If a new datavector location is added, or something changes about that logic, it has to be fixed everywhere. The way proposed above encapsulates all of that into the get_real_single_cpu_data() functions. And something like that has to happen anyway, since we will always have to typecast the void pointer at the very least. This way we handle multiple problems at once, and the code lives only in one place.

You are right to say we have to implement the transfer functions somewhere and anyway. We agree on the basic concept, that datavectors are being checked at the begin of a processing unit. Eventually the data has then to be transferred to a different device memory. But in opposite of plugging the structure into a single "automagic" function that lives somewhere and has to be modified for every architectural change. I propose a very clean object oriented design architecture where the structure of the pipeline and it's dataflow it reflected in classes that describe the composition and responsibilities of objects. The code of data transfer functionality is coupled strongly with the correct data types and processing architecture by compile time (inheritance) and is implemented on only one place only one time. In other words:
Each processing object will be coupled with it's fitting transfer/convert methods at the timepoint of it's creation. From then it knows how to transfer/convert Data by itself.

You would never have to change existing and tested code. You can always extend the structure by adding new classes (new processing architectures) that inherit the basic data transfer behavior.

In turn the automagic functions must do this at runtime. You have to tell them what to do for each transaction you want to make. Can you assure to tell them always to do the right thing? You'll have to implement all the different permutations of transfers at one place. I see large switch case code fragments showing up. Imagine if new architectures appear. You would have to touch/manipulate this code again and again.

Alex: I think it is important to emphasize, as you implied, that we have two different types of objects we are dealing with. We have data objects that are conceptually the same no matter where they live, and we have processing functions that will perform differently depending on how they are built.

I completely agree. Excellent description :-)

The python level should treat all data objects the same and this extends to not doing the location checking, moving etc in the python level. I agree that it is cleaner and more convenient to do the data retrieval/moving in the clayer, especially if it handles the typecasting in a single set of helper functions.

I see it in the opposite way! Python is our main object oriented control facility. In my concept there is no need of typecasting and modifying in the C layer. The C layer consists basically only of Constructors, Destructors and processing functions that might call kernels. They are free to allocate local scratch memory by itself at their own responsibility. But "datavectors" will be given to them only by Godfather Python Layer. The partitioning of algorithms into functions has to be done to fulfill this main requirement of the framework! Think of a "Control Layer" and a " High Performance Processing Layer".

(1) Josh: It can actually be a little confusing to correctly typecast and dereference the void pointers in C, and if you don't have all the compiler warnings turned on (or do and ignore warnings it gives you) the this is one place where its very easy to shoot yourself in the head.

We should make it as a requirement of design to allow only warning free code to be deployed. Other than that I would suggest to not allow deployment of code that doesn't pass the unittest

(2) Karsten, Alex, and I managed to get ourselves confused about this while coding up some stuff in Syracuse. If we put these into separate functions, then there's one place we have to do it right, everyone else who writes C is just working with good old pointers to C datatypes. If every author of a C function is responsible for correctly typecasting/dereferencing a void pointer, then we are sure to have many bugs flowing from that as we go forward. Some will be hard to find.

I agree. That is why in my concept there is no typecasting of void pointers at all. Because there are no void pointers. If a C function gets a wrong type of datavector it will crash at compiletime not at runtime!

(2) The ability to put error checking into those functions is also useful.

I would rather like to have the error checking in Python it is very much more convenient there and the logging functionality is there and exceptions actually makes sense only in a Language with a garbage collection. There will be a straightforward error handler in the C layer (like the one we agreed upon in Syracuse).

(3) If you say that the C layer doesn't need to know about where things live (as Karsten did this morning) then at best that means that you are writing C code that doesn't check that its memory lives where it's supposed to, because it's assuming that Python has done that before handing it off. But now suppose you want to write an Einstein@Home application using this codebase, so that there is no Python layer. Then now all of your C functions (and you may still want to combine CPU/GPU functions in this scenario) never check that they are given sane input, and you either have a very dangerous C program or you have to go back and put that sanity check into every single C function.

Good point :-) In that case it will be a great pleasure for me to enter the realm of C++ again and write/convert the whole Phython - layer in C++ with all it's sophisticated class architecture and error checking and in that going as well getting complete rid of the Swig-Wrapping because it is then only a native C/C++ function-call-interface making it a straightforward monolithic application. Btw.: This is the reason for my appeal to not write too much "pythonic" python code.

Unittest!

We have ignored unittest in our discussions. Unittests should be an inherent part of our design and our design methodology. I like to bring out a very serious appeal about using unittest at a very early time point of design. In my concept the unittests are defined and they reside only in Python. They will test each processing object from the python layer down to the very lowest C layer and possibly GPU kernels! The key is that you have to write test code anyways. Usually you would manipulate the top script and later change it back to the origin. Then all your testing code is away. The effort you spend in writing unittest REMAINS all over the projects lifetime. It's fun to work with the straindata unittest. Whenever I tweak around in straindata a simple call to it's unittest gives me the confidence that everything is OK. You'll have to extend the unittest of your module while development and so it evolves with your module.

"Prerequisite for unittest is the object oriented concept. Things like the automagic functions can not be covert by unittest in python. If the datatypes are not known by python (void) they can not be tested with the unittest!"

1st dark clouds appear over PyCBC with the automagic concept regarding the OpenCl complex datavector:

It is a dangerous style in terms of a consistent, selfexplanatory and robust framework to utilize a single buffer datavector as a device memory structure to hold OpenCl complex vectors by splitting the buffer by half and let the 1st part hold real numbers and the 2nd part imaginary numbers!

A much better way is already defined very cleanly in the code by following the current object oriented concept (see datavectoropencl.c)