viennacl-gsoc-proposal

Toby St Clere Smithe

April 28, 2013

Contents

1 Proposal title

Integrating ViennaCL into the Python scientific ecosystem

2 Short description

PyViennaCL will provide Python bindings for ViennaCL along with documentation and regression tests. The bindings will provide Pythonic objects and routines that represent the ViennaCL API, at least up to and including BLAS level 3. The bindings will also integrate coherently into the current Python scientific ecosystem, by supporting concurrent use of the NumPy ndarray, as well as aiming to provide a drop-in subset of the full NumPy API using ViennaCL as a back-end where possible.

3 Content

3.1 Personal background

I am a finalist undergraduate at the University of Oxford, studying for a degree in psychology and philosophy. My main academic interests are in computational neuroscience – in particular, the areas overlapping theoretical neuroscience, computation, and dynamical systems theory; I am particularly interested in what we can learn about computation from mathematical and physical theory, and physical systems such as neural networks.

I am also a keen supporter of free software (I maintain a couple of packages in Debian and Ubuntu, and occasionally contribute patches for bugs I come across). This January, I completed a project in connectionist neural modelling, writing >2000 lines of Python and C++. I have experience with Python bindings, too, though that is a few years back now: I once wrote, as an exercise, some partial bindings for D-Bus.

I have applications pending for doctoral work in computational neuroscience at a couple of institutions here in the UK, and indeed this is very much the path I would like to pursue for the moment. A few weeks ago, having acquired a laptop with a modern nVidia GPU, I decided to spend some time over the summer working on an OpenCL implementation of a neural network simulator, mainly to teach myself about GPGPU programming.

To work on the Python bindings for ViennaCL as a Google Summer of Code project would be perfect, since I could combine my already extant summer plans with my love of Python, begin work in scientific computing – and have some reusable code come out of it in the end! In particular, I would be able to test my bindings with my own personal neural network project.

3.2 About the project

Amongst others, there are two parallel trends in scientific computing which this project aims to bring seamlessly together: firstly, the increasing use of Python for scientific computations, including systems and mock-ups; and secondly, the increasing interest in highly parallel GPGPU computing, which is now available even on commodity hardware. However, until recently, GPGPU programming has been highly technical, and unintegrated with common tools.

ViennaCL makes GPGPU computing much more straightforward, by providing a library of compute kernels (principally using OpenCL or CUDA), and a BLAS-like C++ interface, for linear algebra. A set of Python bindings would therefore provide this rich API to an increasingly popular computational ecosystem.

However, the Python world already has frameworks for scientific computing and linear algebra – principally NumPy and SciPy; in particular, NumPy provides a powerful and widely used array class, the ndarray (for "N-dimensional array"). This project therefore proposes not to discard this work, and so aims to provide NumPy compatibility.

The project will provide NumPy compatibility in two forms: firstly, and as a required deliverable, objects will be easily created from and converted to ndarrays, and procedures will, where possible, accept ndarrays as well as native types. But this type fungibility incurs context switching and slow memory copying, so the second piece of NumPy compatibility will be in the form of a submodule as fully as possible providing a NumPy-compatible API – in addition to the lower-level direct ViennaCL API.

The requisites of the summer project are limited in scope to a stable base API, in order to provide a workable but deliverable system come "pencils down". But the larger project is grander, and so I have many aims beyond the summer and its few "required deliverables". Ultimately, I aim for PyViennaCL to be a formidable competitor to systems such as Theano, providing dynamic access to advanced features of ViennaCL such as runtime kernel generation, and a pluggable interface for custom kernels.

3.3 Benefits for ViennaCL/Python/science

By providing Python bindings for ViennaCL, the Python-using scientific community would gain access to a straightforward library for GPGPU linear algebra. As well as the direct benefits of this for the library consumers, this would

bring exposure to ViennaCL itself, exposing bugs and potentially attracting contributions.

By integrating with the Python scientific computing ecosystem through NumPy, the "barrier to entry" would be especially low – users would have to make few modifications to existing code in order to take advantage of GPU hardware.

And of course there are benefits for me: I would gain experience in GPGPU programming, and at the boundary of Python and C++ – areas which are increasingly popular in general, but which are particularly useful in the context of my interests. After all, I plan on using my own work on PyViennaCL to build neural network simulations.

3.4 Technical details

A prototype branch is available on GitHub, which will be referred to occasionally below. This branch provides an answer to benchmark puzzle using PyViennaCL, implemented as a test unit using the prototype pyviennacl code. The branch build pyviennacl, being fully integrated into the CMake build and test system, but it has extra dependencies: Python3 (interpreter, headers and libraries); NumPy (inc. headers); Boost::Python (for Python3); and, included as a git submodule, Boost::NumPy. A further limitation is that the pyviennacl test will not run unless pyviennacl is installed (ie, run `make install && ldconfig`).

The pyviennacl prototype implementation exhibits many of the features described below, such as API abstraction, an expression class, delayed execution, `numpy.ndarray` support, and partial artithmetic support. It defaults to using the OpenCL compute back-end.

3.4.1 Implementation details and dependencies

ViennaCL is a largely self-contained header-only C++ library, making use of such C++ meta-programming techniques as expression templates to enable compile-time kernel generation and expression optimisation; available back-ends, for instance, are decided by compiler defines. These features are useful for a C++ system, but make building a Python one more complex.

For this reason, PyViennaCL is implemented with a core in C++ using Boost::Python – also depending on Boost::NumPy (and thus by extension Python and NumPy themselves) – beneath a more user-friendly Python wrapper. Compile-time techniques such as expression templates are only available in a limited form for Python code, since the bindings are pre-built and invoked at runtime. Consequently, Boost::Python is used to wrap a subset of the C++ templates (eg, the viennacl::vector_expression class) into coherent objects (eg, the pyviennacl.vector object). Boost::Python also translates C++ features such as operator overloading neatly to and from Python, allowing ViennaCL objects to be constructed and manipulated in terms of native Python types.

3.4.2 PyViennaCL object model and API

PyViennaCL will expose the ViennaCL at three levels of increasing abstraction: at base, the `pyviennacl._viennacl` is the Boost C++ core, and thereby provides low-level access to the ViennaCL API; above this is the pyviennacl module itself, which is a thin Python wrapper around `pyviennacl._viennacl`, and which provides routines for expression construction, delayed execution, and type safety and conversion – all of which would be much more cumbersome to write in Boost::Python C++, since Python is fairly loosely typed, and C++ is not; whilst it's important to be careful in loosely typed situations, it does make this situation easier, by eliminating a lot of "boiler-plate" code. Finally, `pyviennacl.numpy` provides a drop-in substitute for `numpy`, created by importing `numpy` itself, and overloading its objects and functions where possible.

The `pyviennacl._viennacl` module is publicly accessible, but should not be considered stable. On the other hand, in the initial release, it will be recommended for people to use the main `pyviennacl` module, which provides a stable API expressing the ViennaCL functionality in a Pythonic manner.

All classes in the main `pyviennacl` module provide attributes `result` and `value`: `result` performs any pending operations and returns an object of the resultant type; and `value` performs any pending operations and returns an `ndarray` containing the values of the resulting object. Unlike those of ViennaCL itself, `pyviennacl` objects do not provide bracket operators or iterators for elementwise access to vector/matrix contents; instead, users should force the object to be computed and copied to CPU RAM by using the `value` attribute, which does provide brackets, slices, and iterators.

See the specific sections below for more details.

3.4.3 Required deliverables

- `pyviennacl.double` **implements** `viennacl::scalar<double>`
  Constructible from Python `float`, and provides `value` and `result` attributes. Delayed execution, to

mitigate slowness of accessing the compute device. Otherwise, provides arithmetic operators, so usable in expressions just as a Python native float would be. As usual, the type result of an expression is a `pyviennacl.expression` object.

- `pyviennacl.vector` **implements** `viennacl::vector<double, 1>`
  Initially, memory alignment need not be configurable, and so alignment choice constitutes a "nice-to-have".

Table 1: pyviennacl.vector – see Table 3.2 in the ViennaCL manual

ViennaCL Interface	pyviennacl Interface	Comment
`CTOR()`	`vector()`	Empty vector, size unspecified
`CTOR(n)`	`vector(n)`	Empty vector, size specified by `n`
`scalar_vector<double> .. .. (n, value)`	`vector(n, value)`	Vector of length `n` with each element set to `value`
`CTOR(v)`	`vector(v)`	New vector using size and values of old vector `v`
`N/A`	`vector(nd)`	New vector taking values from ndarray `nd`
`N/A`	`vector(l)`	New vector taking values from list `l`
`v.clear()`	`v.clear()`	Set all entries to `0`
`v.resize(n, preserve)`	`v.resize(n, preserve)`	Set new size to `n`, preserving values according to bool `preserve` [1]
`v.size()`	`v.size`	In pyviennacl, `v.size` is an attribute, not a function [1]
`v.internal_size()`	`v.internal_size`	Again, attribute, not function
`v.swap(v2)`	`t = v; v = v2; v2 = t`	Python maintains references to objects, so the equality operator does not incur copying
`v.empty()`	`v.size == 0`	The explicit check is much clearer
`v.handle()`	?	This is a "nice-to-have", and comes with the later work on pyviennacl custom kernels

[1] I am unsure whether or not to implement resizing using a function, or just to allow the `size` attribute to be read-write. The latter case is more elegant, but it is not obvious in this situation what to do about value preservation. I might just have it that the `size` attribute is read-write, and values are always discarded.

- `pyviennacl.matrix` **implements** `viennacl::matrix<double, row_major, 1>`
  Initially, neither memory alignment nor storage layout need be configurable, and so these both constitute "nice-to-haves". This is true for all matrix types.

Table 2: pyviennacl.matrix – see Table 3.3 in the ViennaCL manual

ViennaCL Interface	pyviennacl Interface	Comment
`CTOR()`	`matrix()`	Empty matrix, size unspecified
`CTOR(n, m)`	`matrix(n, m)`	Empty matrix of size `n` rows by `m` columns
`scalar_matrix<double> .. .. (n, m, value)`	`matrix(n, m, value)`	Matrix of size `n` by `m`, with each element set to `value`
`CTOR(mat)`	`matrix(mat)`	New matrix using size and values of old matrix `mat`
N/A	`matrix(nd)`	New matrix taking values from ndarray nd
`mat.clear()`	`mat.clear()`	Set all entries to 0
`mat.resize(m, n, preserve)`	`mat.resize(m, n, .. .. preserve)`	Set new size to `m` by `n`; `preserve` not honoured yet [2] [3]
`mat.size1()`	`mat.size1` [4]	Attribute describing number of rows in `mat` [3]
`mat.internal_size1()`	`mat.internal_size1` [4]	Again, attribute, not function
`mat.size2()`	`mat.size2` [5]	Attribute describing number of columns in `mat` [3]
`mat.internal_size2()`	`mat.internal_size2` [5]	Again, attribute, not function
`mat.handle()`	?	Nice-to-have, requires later work on custom kernels

[2] It might be plausible to implement `preserve`-honouring in Python, thereby working around the deficiency in the underlying library.

[3] On the other hand, see note 1 in the `vector` interface section above: I might just have it so that the row/column `size` attributes are read-write, and that values are always discarded. In this case, the behaviour would match that of ViennaCL anyway.

[4] Or `mat.rows` and `mat.internal_rows` – which is more readable, but deviates from the underlying interface.

[5] Or `mat.cols` and `mat.internal_cols` – which is more readable, but deviates from the underlying interface.

- `pyviennacl.compressed_matrix` **implements** `viennacl::compressed_matrix<double, 1>`

Table 3: pyviennacl.compressed_matrix – see Table 3.4 in the ViennaCL manual

ViennaCL Interface	pyviennacl Interface	Comment
`CTOR()`	`compressed_matrix()`	Empty matrix, size unspecified
`CTOR(n, m, nnz)`	`compressed_matrix(n, m, nnz)`	Empty matrix of size `n` rows by `m` columns, with space for `nnz` non-zeroes
`CTOR(csr)`	`compressed_matrix(csr)`	New matrix using size and values of old compressed_matrix `csr`
N/A	`compressed_matrix(nd)`	New matrix taking values from ndarray `nd` [6]
`csr.clear()`	`csr.clear()`	Set all entries to 0
`csr.resize(m, n, .. .. preserve)`	`csr.resize(m, n, preserve)`	Set new size to `m` by `n`; `preserve` not honoured yet [7] [8]
`csr.reserve(nnz)`	`csr.reserve(nnz)`	Reserve memory for `nnz` nonzero entries
`csr.size1()`	`csr.size1` [9]	Attribute describing number of rows in `mat` [8]
`csr.internal_size1()`	`csr.internal_size1` [9]	Again, attribute, not function
`csr.size2()`	`csr.size2` [10]	Attribute describing number of columns in `mat` [8]
`csr.internal_size2()`	`csr.internal_size2` [10]	Again, attribute, not function
`csr.handle()`	?	Nice-to-have, requires later work on custom kernels; also need to implement index handles

[6] Or perhaps better using `scipy.sparse.csr_matrix`? But not sure how easily accessible this is from C++ code. In any case, the constructor will count `nnz` of the given matrix argument, and construct then copy the relevant matrices.

[7] It might be plausible to implement `preserve`-honouring in Python, thereby working around the deficiency in the underlying library.

[8] On the other hand, see note 1 in the vector interface section above: I might just have it so that the row/column `size` attributes are read-write, and that values are always discarded. In this case, the behaviour would match that of ViennaCL anyway.

[9] Or `mat.rows` and `mat.internal_rows` – which is more readable, but deviates from the underlying interface.

[10] Or `mat.cols` and `mat.internal_cols` – which is more readable, but deviates from the underlying interface.

- `pyviennacl.coordinate_matrix` **implements** `viennacl::coordinate_matrix<double, 1>`
  This is implemented largely as for `pyviennacl.compressed_matrix`, but looking to `scipy.sparse.coo_matrix` as a model.

- `pyviennacl.ell_matrix` **implements** `viennacl::ell_matrix`
  Again, this is implemented as with other sparse matrix types.

- `pyviennacl.hybrid_matrix` **implements** `viennacl::hyb_matrix`
  Implemented as the other sparse matrix types, but unsure whether to use `pyviennacl.hybrid_matrix` or `pyviennacl.hyb_matrix` as the class name; I'm not sure that there's much weight to the length argument, since we already have `compressed_matrix` and `coordinate_matrix`, and `hyb` isn't immediately obvious as to function.

- `pyviennacl.expression` **implements a binary heap representing supported computations (eg, vector addition, matrix multiplication)**
  This does not map directly onto any ViennaCL C++ class, instead providing the 'delayed execution' functionality supplied by C++ expression templates. The expression class does most of the "heavy lifting" of computations; whereas in C++, an addition operation on two `viennacl::vector` instances would produce a `viennacl::vector_expression` instance, in PyViennaCL, it produces a `pyviennacl.expression` object.

  Python performs arithmetic operations in binary pairs, and so the `pyviennacl.expression` class represents arithmetic expressions as a binary heap. No computations are actually performed until the result of the computation is required to be known, either for algorithmic reasons, or where the user accesses the result.

  Depending on how ViennaCL issue 8 is solved, this should serve to improve performance, and minimise copying and the creation of temporary objects.

- **BLAS operations are implemented where possible using operator overloading and the** `pyviennacl.expression` `class`
  Where operator overloading is not possible or ambiguous, BLAS operations are implemented as named functions in the `pyviennacl` namespace, returning `pyviennacl.expression` objects where possible. See the details below.

  Furthermore, the main PyViennaCL API is aimed to match closely that of ViennaCL and uBLAS; the `pyviennacl.numpy` sub-module will ultimately provide a `numpy` and `scipy` idiomatic version.

  Finally, the $\star$ operator is reserved for the scalar product.

- **BLAS Level 1**
  Where `p = pyviennacl`, `x` and `y` are both vectors, and `a` and `b` are scalars,

Table 4: BLAS Level 1 Syntax – see Table 4.1 in the ViennaCL manual

ViennaCL	PyViennaCL	Comment
`swap(x, y);`	`t = x; x = y; y = t`	Python objects are just references, so no need for a specific `swap` function.
`x *= a;`	`x *= a`	
`y = x;`	`y = x`	
`y += a * x;`	`y += a * x`	
`y -= a * x;`	`y -= a * x`	
`y = element_prod(x, z);`	`y = p.element_prod(x, z)`	Also provide an `element_mul(..)` function, to be Pythonic. Or `y = x.element_prod(z)`
`y = element_div(x, z);`	`y = p.element_div(x, z)`	Or `y = x.element_div(z)`
`inner_prod(x, y);`	`p.inner_prod(x, y)`	Or `x.inner_prod(y)`, or `x.dot(y)`
`a = norm_1(x);`	`a = x.norm_1`	
`a = norm_2(x);`	`a = x.norm_2`	
`a = norm_inf(x);`	`a = x.norm_inf`	
`i = index_norm_inf(x);`	`i = x.index_norm_inf`	
`plane_rotation(a, b, x, y);`	`p.plane_rotation(a, b, x, y)`	

- **BLAS Level 2**
  Where `p = pyviennacl`, `A` is a matrix, `x` and `y` are vectors, and `a` and `b` are scalars,

Table 5: BLAS Level 2 Syntax – see Table 4.2 in the ViennaCL manual

ViennaCL	PyViennaCL	Comment
`y = prod(A, x);`	`y = p.prod(A, x)` [11]	Or `y = A.prod(x)` [11]
`y = prod(trans(A), x);`	`y = p.prod(A.trans, x)` [11]	Or `y = A.trans.prod(x)` [11]
`x = prod(A, x);`	`x = p.prod(A, x)` [11]	Or `x = A.prod(x)` [11]
`x = prod(trans(A), x);`	`x = p.prod(A.trans, x)` [11]	Or `x = A.trans.prod(x)` [11]
`y = a * prod(A, x) + b * y;`	`y = a * p.prod(A, x) + b * y` [11]	Or `y = a * A.prod(x) + b * y` [11]
`y = a * prod(trans(A), x) ...` `... + b * y;`	`y = a * p.prod(A.trans, x) ...` [11] `... + b * y`	Or `y = a * A.trans.prod(x) + b * y` [11]
`y = solve(A, x, tag);` [12]	`y = p.solve(A, x, p.tag)`	Or `y = A.solve(x, p.tag)`
`y = solve(trans(A), x, tag);` [12]	`y = p.solve(A.trans, x, p.tag)`	Or `y = A.trans.solve(x, p.tag)`
`inplace_solve(A, x, tag);` [12]	`x = p.solve(A, x, tag)`	Or `x = A.solve(x, p.tag)`
`inplace_solve(trans(A), x, tag);` [12]	`x = p.solve(A.trans, x, tag)`	Or `x = A.trans.solve(x, p.tag)`
`A += a * outer_prod(x, y);`	`A += a * p.outer_prod(x, y)`	Or `A += a * x.outer_prod(y)` [13]
`A += a * outer_prod(x, x);`	`A += a * p.outer_prod(x, x)` [14]	Or `A += a * x.outer_prod(x)` [15]
`A += a * outer_prod(x, y) ...` `... + a * outer_prod(y, x);`		

[11] NB: PyViennaCL also provides `mul` functions wherever ViennaCL provides `prod`. Except for particular technical vocabulary such as the inner/outer product functions.

[12] `tag` is one out of `lower_tag`, `unit_lower_tag`, `upper_tag`, and `unit_upper_tag` – each of which is provided as an attribute in the `pyviennacl` module.

[13] For the outer product functions, there is a temptation to provide a variant of `p.prod` such that `x.outer_prod(y) == x.prod(y.trans)`. But similarly, there is a temptation not to provide such a variant, for clarity's sake.

[14] `A += a * p.outer_prod(x, y) + a * p.outer_prod(y, x)`

[15] `A += a * x.outer_prod(y) + a * y.outer_prod(x)`

- **BLAS Level 3**
  Where `p = pyviennacl`, `A`, `B` and `C` are matrices, and `x` and `y` are vectors,

Table 6: BLAS Level 3 Syntax – see Table 4.3 in the ViennaCL manual

ViennaCL	PyViennaCL	Comment
`C = prod(A, B);`	`C = p.prod(A, B)` [16]	Or `C = A.prod(B)` [16]
`C = prod(A, trans(B));`	`C = p.prod(A, B.trans)` [16]	Or `C = A.prod(B.trans)` [16]
`C = prod(trans(A), B);`	`C = p.prod(A.trans, B)` [16]	Or `C = A.trans.prod(B)` [16]
`C = prod(trans(A), trans(B));`	`C = p.prod(A.trans, B.trans)` [16]	Or `C = A.trans.prod(B.trans)` [16]
`C = solve(A, B, tag);` [17]	`C = p.solve(A, B, p.tag)`	Or `C = A.solve(B, p.tag)`
`C = solve(trans(A), B, tag);` [17]	`C = p.solve(A.trans, B, p.tag)`	Or `C = A.trans.solve(B, p.tag)`
`C = solve(A, trans(B), tag);` [17]	`C = p.solve(A, B.trans, p.tag)`	Or `C = A.solve(B.trans, p.tag)`
`C = solve(trans(A), ... ... trans(B), tag);`	`C = p.solve(A.trans, B.trans, p.tag)`	Or `C = A.trans.solve(B.trans, p.tag)`
`inplace_solve(A, trans(B), tag);` [17] [18]	`B = p.solve(A, B, p.tag)`	Or `B = A.solve(B, p.tag)`
`inplace_solve(trans(A), x, tag);` [17] [19]	`B = p.solve(A.trans, x, p.tag)`	Or `B = A.trans.solve(x, p.tag)`
`inplace_solve(A, trans(B), tag);` [17]	`B = p.solve(A, B.trans, p.tag)`	Or `B = A.solve(B.trans, p.tag)`
`inplace_solve(trans(A), x, tag);` [17] [19]	`B = p.solve(A.trans, B.trans, p.tag)`	Or `B = A.trans.solve(B.trans, p.tag)`

[16] NB: PyViennaCL also provides `mul` functions wherever ViennaCL provides `prod`. Except for particular technical vocabulary such as the inner/outer product functions.

[17] `tag` is one out of `lower_tag`, `unit_lower_tag`, `upper_tag`, and `unit_upper_tag` – each of which is provided as an attribute in the `pyviennacl` module.

[18] (Why is this `trans(B)`, given the mathematical expression in Table 4.3 at this point?)

[19] (Why `x` in Table 4.3 at this point?)

- **Direct and iterative solvers**
  These are provided using the `pyviennacl.solve` function as simple wrappers around the `viennacl::linalg` functions, using the `viennacl` tags in order to choose the algorithm. Substitute the desired tag for `p.tag` in the tables above.

- **Eigenvalue computations**
  `pyviennacl.eig(A, p.eig_tag)` computes the largest eigenvalue(s) of matrix `A`, with the algorithm determined by the choice of `p.eig_tag`.

- **Integration of types and operations with the NumPy framework (via ndarray) wherever possible**
  Each class can be constructed from an `ndarray` – of the right form – and accessing the attribute `o.value` on any pyviennacl matrix/vector/expression object `o` performs any pending operations and returns an `ndarray` object, or `scipy` sparse array object, representing the object's state.

- **API documentation**
  Documentation is important, particularly in the few places where the Python API differs from the C++ API. Nonetheless, because the main `pyviennacl` API tries to stay as faithful to the construction of ViennaCL in C++ as possible, the documentation task is made all the more easier. The detail of this proposal will also make a good starting-point for documentation.

- **Regression tests**
  ViennaCL uses the CMake testing framework, CTest, for regression testing. Since PyViennaCL will be built out of the main ViennaCL tree, it integrates into that same framework. In the prototype tree, one CTest unit is implemented, as part of the solution to Puzzle 1 (see below). By pencils down, at least 75% of the required PyViennaCL API should be covered by tests, with the remaining 25% at least in progress. The tests could follow the examples of those currently included in C++, to demonstrate translating code between the two languages; the test framework provides a good opportunity to provide example code.

- **Portability**

  - **Portability concerns about dependencies**
    The use of Boost requires some thought about portability, since different Boost libraries are occasionally API- and ABI- incompatible, and the same versions are not available everywhere. Nonetheless, the permissive licensing of Boost makes it straightforward to provide statically linked versions of PyViennaCL for more obscure or controlled platforms.

  - **QR factorisation**
    QR factorisation is only a nice-to-have, because of its external dependency on uBLAS; see below.

3.4.4 Nice to have

The following are ordered in descending order of desirability; that is, most desirable comes first.

- **Full coverage of the ViennaCL algorithm library**
  - **Preconditioners**
    This requires some thought about how best to present the C++ API in a Pythonic fashion. Because the API here should look like the rest of `pyviennacl`, this is a "nice to have", to be implemented once the main API has taken shape and is working well.
  - **QR factorisation**
    (NB external uBLAS dependency)
  - **Compute-back-end-dependent functionality**
    To implement these functions in PyViennaCL requires the work on run-time back-end choice, which is itself a nice-to-have. In the graph of features, a "required deliverable" feature cannot depend on a "nice to have" feature.

    This functionality includes: additional iterative solvers and preconditioners; the fast Fourier transform; bandwidth-reduction algorithms; and algorithms for non-negative matrix factorisation.
  - **Structured matrix types and algorithms**
- **Configurable memory alignment and storage layout for vector and matrix types.**
- **API compatibility with NumPy/SciPy, letting PyViennaCL and NumPy be interchangeable**
  - **PyViennaCL implements a `numpy.ndarray` compatible class, performing operations on the requested compute back-end.**
    The aim here is that PyViennaCL can ultimately provide a GPGPU drop-in for many NumPy/SciPy functions. As such, the API would look just like the relevant parts of NumPy/SciPy.
- **Parameter tuning**
  ViennaCL provides an API for determining global and local compute work sizes (i.e., number of work groups, and work items per group) on the OpenCL back-end. These tunables have a "considerable impact on the obtained device performance"[1]. As such, it is worthwhile exposing this interface in PyViennaCL.
- **Run-time kernel generation and execution**
  A C++ interface for automated user kernel generation is documented in chapter 11 of the ViennaCL manual. This could usefully be exploited in PyViennaCL for the automatic generation of kernels for the evaluation of complex expressions produced by the pyviennacl.expression class. The feasibility of this feature depends on the flexibility of the ViennaCL kernel generator interface, which is currently classed as "experimental". As such, this is only a "nice to have" feature.
- **Run-time switching of compute back-end**
  Because the compute back-end is determined by a compiler `#define`, this feature requires that the `pyviennacl._viennacl` implementation code is abstracted into back-end-specific modules, eg, `pyviennacl._viennacl_ocl` and `pyviennacl._viennacl_cuda`. These would be API-compatible, and types from one should be transparently sharable with types from the other. In this way, Python code could call both (say) OpenCL and CUDA kernels as desired.
- **Interface for custom kernels**
  Chapter 9 of the ViennaCL manual describes a C++ interface for using custom OpenCL kernels in ViennaCL applications. ViennaCL provides helper classes and functions in the `viennacl::ocl` namespace for kernel compilation and launching, and it would be useful to be able to take advantage of this work in Python programs.
- **Intelligent back-end choice (according to hardware and algorithmic demands)**
  This would require a set of performance heuristics; in the end, it might just be in the user's best interest (re performance) for back-end choice to be necessarily manual.

[1] ViennaCL manual, version 1.4.1, page 54.

3.4.5 Expected problems and solutions

- **Performance optimisation**
  - **PCI-e latency and CPU/GPU copying; delayed execution**
    A substantial hindrance to the performance of GPGPU computing is "set-up costs": kernel initialisation takes much longer than actual computation for small work loads (see solution to Puzzle 1 below). Even after initialisation, copying to and from the compute device over the bus is substantially slower than copying within the device memory domain: PCI-express bandwidth and latency, for instance, is orders of magnitudes slower than (G)DDR bandwidth and latency.

    As a result, work to mitigate much initialisation or copying is worthwhile. In PyViennaCL, this work appears largely in the use of the `pyviennacl.expression` class, which evaluates expressions – ultimately by the construction of complex composite kernels – only when necessary.
  - **Parameter tuning**
    Compute devices vary in their performance characteristics. As such, a desirable "nice-to-have" is the ability to tune work load parameters, as is the ability to be able to switch back-ends at runtime.
- **Limitated access to compile-time optimisations in Python run-time (eg, expression templates)**
  ViennaCL makes much use of C++ compile-time optimisation via meta-programming techniques like expression templates. However, these cannot easily be translated into run-time optimisations, since it is not known at run-time which expressions – after a set of primitives – will be needed. Without hard-coding a complex and unmaintainable recursive type hierarchy in the `pyviennacl._viennacl` C++ module, then, this work is not available.

  As a result, in order to avoid a large number of slow GPU/CPU copies and kernel initialisations, an equivalent means of optimisation is required. This is implemented in the `pyviennacl.expression` class binary heap.
- **Single precision devices and the Python `float` type**
  The Python `float` type represents a C++ `double`. However, some compute devices may not support double-precision floating point types. As a result, at least at first, these devices will not be supported by PyViennaCL.

3.5 Project schedule

3.5.1 *2013-06-07 Mon*: Week 0

End of exam period.

3.5.2 *2013-06-17 Mon*: Week 1

Work begins on implementing and documenting basic types and tests

3.5.3 *2013-06-24 Mon*: Week 2

(work continues)

3.5.4 *2013-07-01 Mon*: Week 3

Basic types and arithmetic operations implemented; `pyviennacl.expression` class has a naive implementation

3.5.5 *2013-07-08 Mon*: Week 4

Remaining matrix types implemented; BLAS Level 1 complete

3.5.6 *2013-07-15 Mon*: Week 5

BLAS Level 2 complete; optimisation work on `pyviennacl.expression` class continues in earnest

3.5.7 *2013-07-22 Mon*: Week 6

(work continues)

3.5.8 *2013-07-29 Mon*: Week 7

BLAS Level 3 complete; `pyviennacl.expression` and algorithm API complete to the required extent

3.5.9 *2013-08-05 Mon*: Week 8

Documentation of finished work completed; any remaining functions have documented stubs; regression test coverage hits at least 50%

3.5.10 *2013-08-12 Mon*: Week 9

Spare week, catch-up if falling behind schedule; start working on portability

3.5.11 *2013-08-19 Mon*: Week 10

Work on build system and cross-platform portability is complete

3.5.12 *2013-08-26 Mon*: Week 11

Code review: go back over code, documentation and tests; implement remaining ViennaCL algorithm wrappers

3.5.13 *2013-09-02 Mon*: Week 12

Work on nice-to-have features: plan NumPy/SciPy implementation; implement parameter tuning

3.5.14 *2013-09-09 Mon*: Week 13

Work on NumPy/SciPy reaches an early usable stage; parameter tuning implementation done; run-time backend/kernel work progress

3.5.15 *2013-09-16 Mon*: Week 14

Soft "pencils down": code clean-up; check API consistency

3.5.16 *2013-09-23 Mon*: Week 15

Firm "pencils down"

3.6 Answers to puzzles

http://www.iue.tuwien.ac.at/cse/wiki/doku.php?id=python_wrapper#puzzles

3.6.1 Benchmark puzzle

(Graphs are contained in the pyviennacl-puzzle.pdf file, which is produced by the `pyviennacl` test.)

Every time an operation is performed on the GPU, the relevant kernel has to be copied over (and whatever initialisation necessary done). This is of constant time complexity, but it takes a time long enough to dwarf the time it takes to do the actual operation, until the vector becomes large enough. On my nVidia GeForce 610M GPU (using driver version 319), this critical point is at 2**12 vector entries.

Alternatively, plotting the execution time *per entry* shows strictly linearly decreasing time up to 2**12, and constant time thereafter. The negative slope up to 2**12 entires is because the kernel initialisation takes a constant time, which is always greater than the actual calculation time; the flat slope after this point is because the actual calculation takes longer than initialisation, and the time taken is directly proportional to vector length.

The operation `x = y1 + y2 + y1 + y2` takes a constant multiple of time longer than the operation `x = y1 + y2` because it involves a second operation, but it does not take simply twice as long; instead, it takes around four times longer. This is because `y1 + y2` is computed twice – with only one initialisation of the corresponding kernel – and then the result is substituted into the computation `[(y1 + y2) + (y1 + y2)]`, which uses a different kernel: `(y1 + y2)` uses `avbv_cpu_cpu` in this case, while here `[(y1 + y2) + (y1 + y2)]` uses `avbv_gpu_gpu`, and this requires the results of the first calculations to be stored, the re-initialisation of the compute device, and then the final calculation to be performed.

So the two major determining processes are as follows: firstly, initialisation of kernels on the compute device; secondly, the actual computation.

3.6.2 Advantages and disadvantages of C++ expression templates

Expression templates are a C++ metaprogramming technique wherein the type of a class instance recursively represents the operation which produced that instance; C++ *templates* are used to construct the *expression* (hence "expression templates"), using the "curiously recurring template pattern" (CRTP) and, usually, operator overloading. By representing the expression in the instance type, the expression does not have to be evaluated at the point in the C++ code at which it is created, but only when it is accessed; in effect, part of the calculation is performed at compile time, with the rest performed at some unspecified point. For instance, the addition of two instances of type `Vector` might produce an instance of type `VectorExpression<Vector, Vector>` such that the actual addition is only performed on accessing some element of the `VectorExpression` instance.

Expression templates are therefore a powerful and, unsurprisingly, expressive technique. But, complex expressions necessarily produce complex types. In most C++ code, this doesn't pose a difficulty, because the compiler is able to resolve the types through inheritance. ViennaCL, because it has to mediate interaction between C++ code on the CPU and OpenCL/CUDA code on the GPU, implements a limited form of expression templating, up to a recursive depth of two. This means that even fairly simple expressions, like `x = y1 + y2 + y1 + y2`, are computed in a somewhat indirect fashion using a temporary object, rather than a neat single expression type. Work is on-going in ViennaCL itself to mitigate this.

Nonetheless, even expression templates to a depth of two are not easily available at run-time, because they are a compile-time optimisation, determined by the compiler. As a result, Python code cannot take advantage of a substantial hierarchy of templates, without bearing a complex maintenance burden. Instead, another solution is required. In PyViennaCL, this takes the form of the `pyviennacl.expression` class; see above for details.