PyViennaCL

Toby St Clere Smithe; Karl Rupp; Philippe Tillet

8 July 2014

Outline

PyViennaCL

Making powerful scientific computing really transparently easy

- Abundant hardware, but how to make the most of it?

ViennaCL Overview

The Vienna Computing Library (*ViennaCL*) is primarily four things:

1. a set of kernel templates for computing elementary mathematical operations on vectors and matrices;
2. secondly, a generator for optimising the kernels for the types and shapes of the operands and the nature of the computational hardware;
3. thirdly, a scheduler for managing memory buffers, and the generation and dispatch of the kernels;
4. fourthly, a C++ template API to hide all this complexity from the user, who just wants to get her calculations done.

Provenance

- Developed largely by a group in the Institute for Microelectronics at the Vienna University of Technology, to support their simulations of sillicon dynamics using finite-element methods
  - but my interest is in complex systems, and my collaborator works on bio-inspired machine learning
- My involvement originates with the Google Summer of Code (both in 2013 and 2014)
- Liberal MIT licence; no copyright assignment
- `http://viennacl.sourceforge.net/`

Mathematical Features

All very familiar! ViennaCL can:

- represent scalars, vectors and matrices, including dense, sparse and structured matrices in a variety of formats and numerical data-types;
- compute basic linear algebra operations
  - BLAS 3 etc: elementwise elementary functions, additions, scaling, matrix and vector products;
- solve large linear systems given (optionally preconditioned) dense or sparse system matrices
  - iterative solvers: Conjugate Gradient, Stabilized BiConjugate Gradient, and Generalised Minimum Residual;
- compute eigenvalues, fast Fourier transforms, LU, QR, NM matrix factorisations.
  - LU factorisation with partial pivoting coming soon...

What is less interesting than how.

Portable performance

Common hardware-agnostic API

- CPU with OpenMP
- OpenCL, and its variety of hardware platforms
  - hardware database: GPUs, accelerators
  - device-specific kernels and tuned parameters
- CUDA
- future: heterogeneous and distributed computing

Kernel generator and scheduler

Representation: templates and expression trees

- Problem: (not just C++!)
  - Main ViennaCL API makes heavy use of C++ templates to represent operations.
  - But C++ templates are kind of compile-time dynamic-typing (details not important here); and
  - we can't hope to encode at compilation the multitude of operations that we might want at Python run-time..
- Solution: the ViennaCL scheduler and expression tree representation

Results: elimination of temporaries; kernel fusion

The combination of an intelligent scheduler and generator enables us to eliminate unnecessary temporary objects, and to minimize computation and memory accesses by fusing kernels.

Representation: expression trees

Consider the following pseudo-code, supposing the only use of x is as one of the summands:

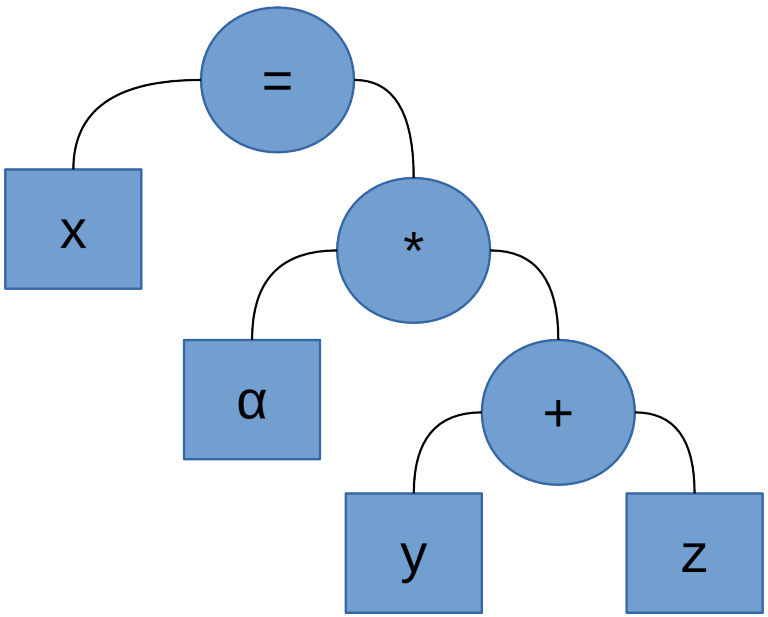
```
x = alpha * (y + z)
```

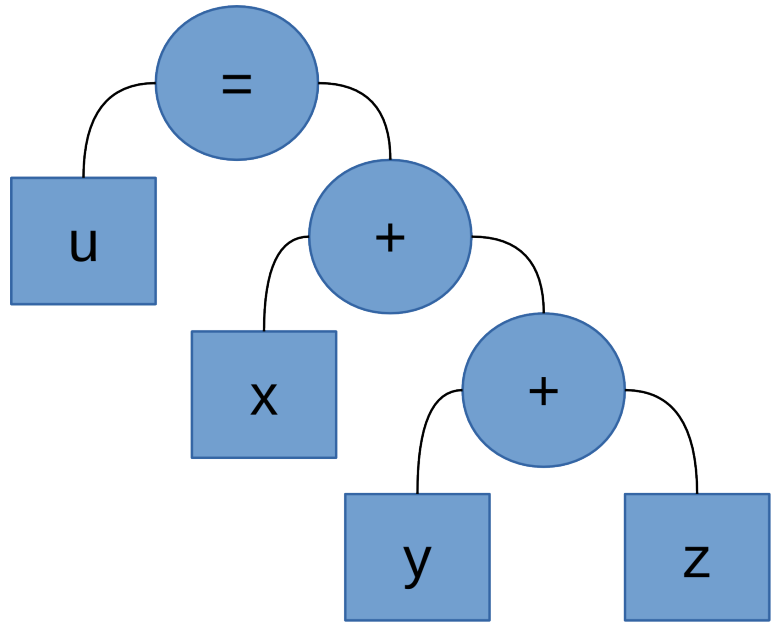
```
u = x + y + z
```

Naively parsing and computing this is wasteful both of memory and computational resources.

- Storing unneeded data; making more accesses than necessary;
- duplicating computations..
- But we don't have to be naive!

Expression tree example



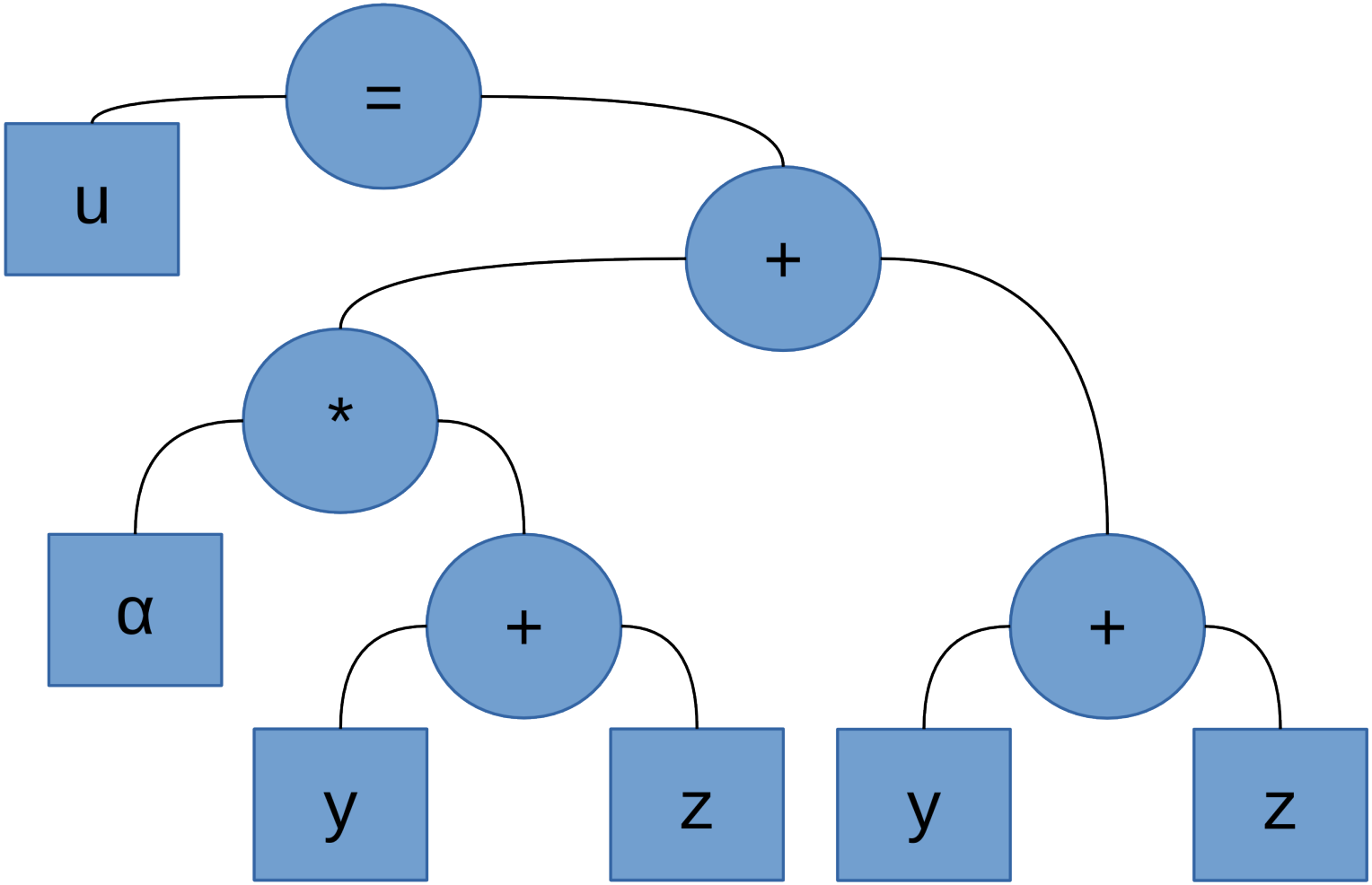


Notice the shared features!

Fused expression tree

Result: elimination of temporaries; kernel fusion

- minimise storage, memory access, computation time.



What is possible right now?

```
s0 = min(x)
s1 = max(x)
s2 = argmin(x)
s3 = argmax(x)
```

- 4 instructions
- Single fused kernel
- Reads `x` only once
- Device-specific (auto-tuned) kernel

```
u = x + y + z
v = x - y - z
```

- 2 instructions
- Single, device-specific kernel
- No temporary for `(y+z)` or `(y-z)`
- Reads `x`, `y`, `z` only once

Performance

Optimised, device-specific kernels

- How do we get them?

Intelligent auto-tuning

- work group sizes and number, elements computed per work item, vector type used, contiguous or strided access
- large parameter space..

We'll see some benchmarks shortly!

First: how about bringing all this to Python?

NumPy, SciPy, and BLAS

On Ubuntu, default NumPy / SciPy uses reference BLAS

Even an optimised BLAS would probably just use the CPU

- not necessarily a bad thing: the CPU is a sensible default choice

Theano, cudamat, and gnumpy

What if I don't have nVidia hardware?

What if I have a GPU and an accelerator like Intel's MIC, and want to make use of both?

What if I want flexible GPGPU with low execution overhead?

PyPy, Cython, PyOpenCL and PyCUDA

. . . if you care more about optimising algorithms than getting your calculations done!

PyViennaCL: are we Zen yet?

- *Simple is better than complex*
- *Complex is better than complicated*
- Like a good koan, PyViennaCL lets you be both.

PyViennaCL architecture and terminology

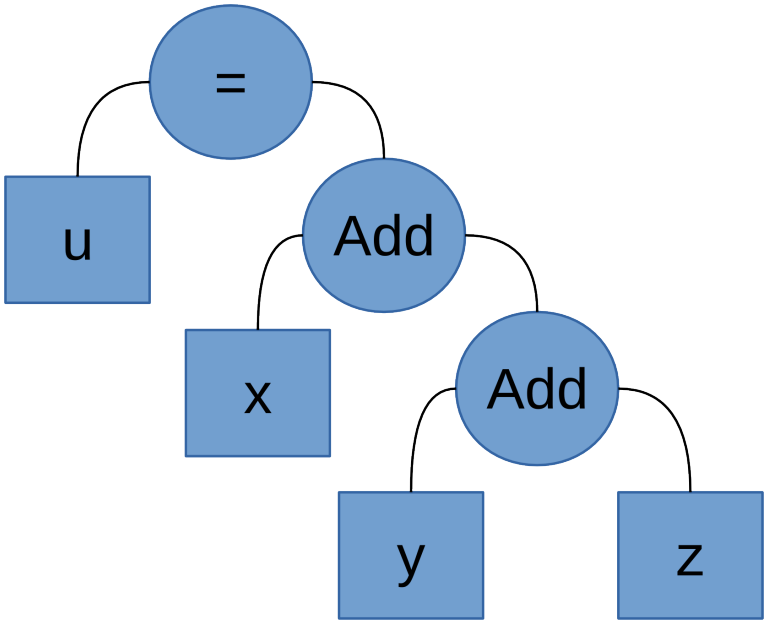
Thin, idiomatic Python layer above a `boost::python` wrapper

Boost may not be as fashionable as Cython, but with judicious use of preprocessor macros, it facilitates exporting ViennaCL's template-heavy API to dynamically-typed Python

Representing the ViennaCL expression tree in Python

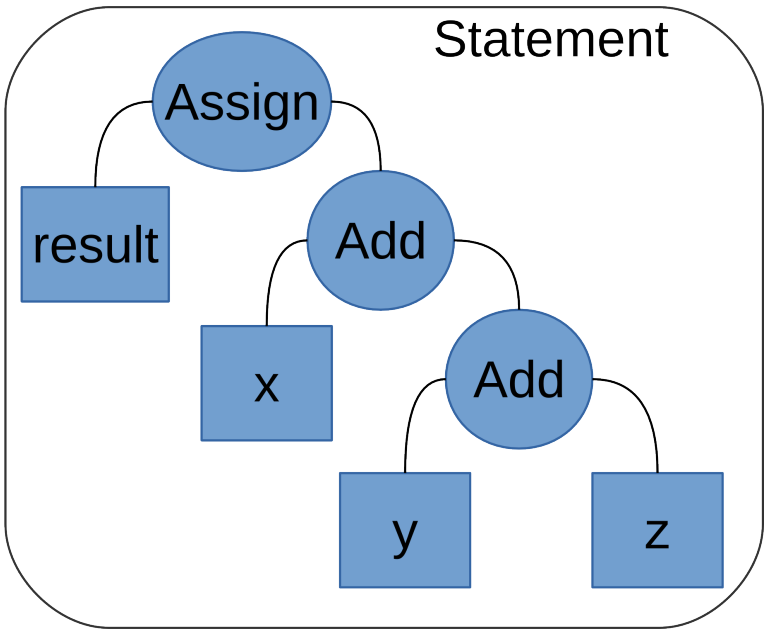
- Leaves, Nodes and Statements
  - `Leaf`: anything that holds data, such as a matrix or vector
  - `Node`: anything that computes data, such as a matrix product
  - `Statement`: represents the expression tree behind the scenes
- Does it matter?
  - `Node` instances adopt the semantics of the `Leaf` type associated with the result of the computation

Another look at the expression graph



- Now, `u` is an instance of `Add`, which is a `Node` subclass.
- When we use `u` outside of PyViennaCL, the following happens...

Enqueuing the expression



PyViennaCL:

- deduces the type and shape of object needed to hold the result of the computation;
- creates an `Assign` node at the top of the tree, so that ViennaCL knows where to store the result;
- schedules the computation;
- points the `result` attribute of `u` at the computed result;
- uses `u.result` in any future expressions involving `u`.

Example: simple matrix-vector product

NumPy code

```
import numpy as np

# Construct operands
A = np.random.rand(1024, 1024)
x = np.random.rand(1024)

# Compute result
b = A.dot(x) # b is another ndarray
```

Example: simple matrix-vector product

PyViennaCL code

Suppose we've constructed `A` and `x` using NumPy, as before.

```
import pyviennacl as vcl

# Transfer to compute device
A = vcl.Matrix(A)
x = vcl.Vector(x)

# Represent computation
b = A.dot(x) # b is a Node instance
# or...
b = A * x    # but PEP 465!!

# Do something with result
print(b) # This enqueues the operation,
         # and stores the result in host memory
```

Simple syntax, high performance

Objects inspired by NumPy / SciPy

- similar dtype interface
- similar object constructors
- similar attributes and function calls
  - *eg*: trans, dot product, solver interface
  - but uses * for dot product, not elementwise!!
- ranges and slices of objects provide proxy views, not copies

Convenience features

- Mix PyViennaCL and NumPy / SciPy matrices and arrays in PyViennaCL functions and expressions
- PyViennaCL will try and do a conversion
  - Conservative: not across dtypes or shapes
  - Explicit is better than implicit!
- Comparison operators (== etc) work, too!

Peformance versus current solutions

Does all this technology work?

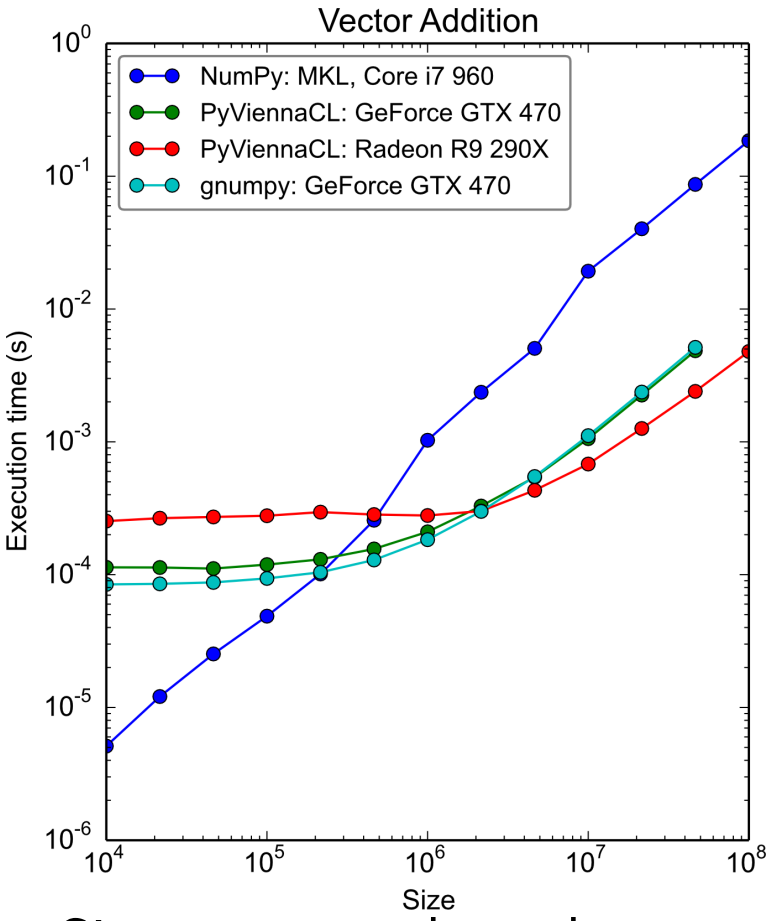
Benchmarks hot off the press!

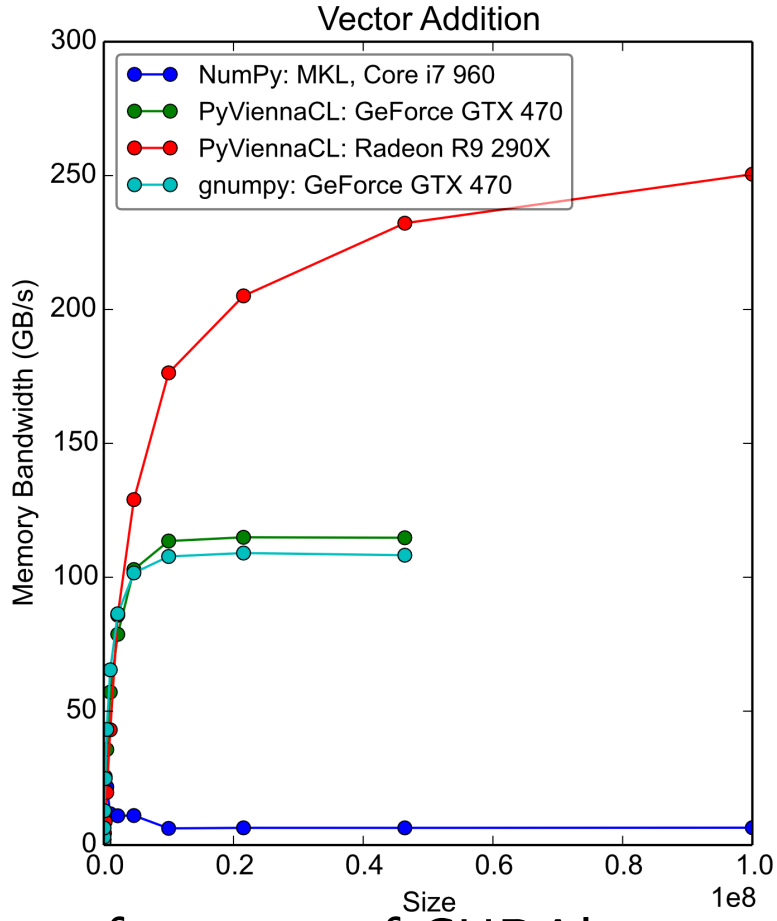
- Memory-bound
  - Vector addition
  - Matrix-vector product
- Compute-bound
  - Matrix-matrix product

Hardware-agnostic!

- Beyond the examples here, ViennaCL can be competitive on any OpenCL/CUDA-compliant hardware.

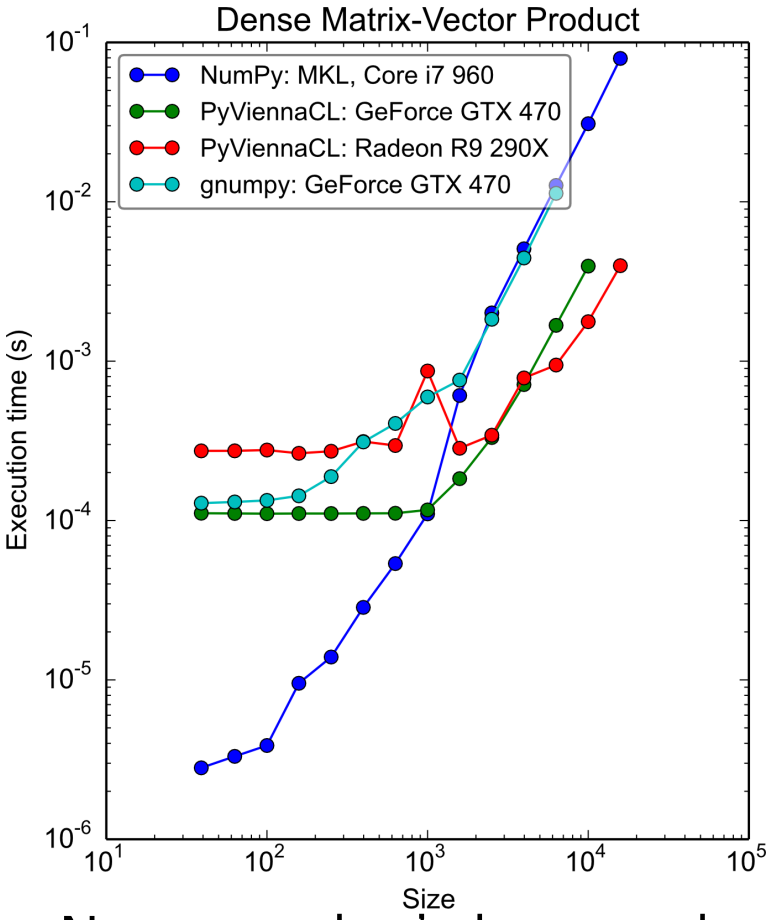
Vector addition

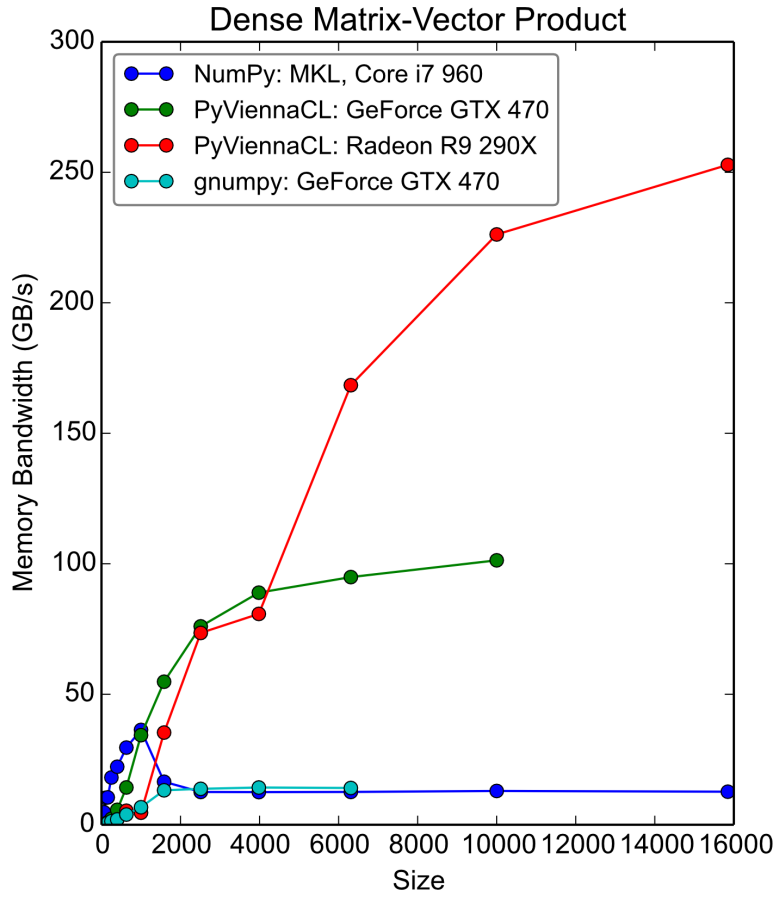




Since memory-bound, easy to reach performance of CUDA's cuBLAS. And lower PyViennaCL overhead on GTX amounts to ~5 GB/s better bandwidth.

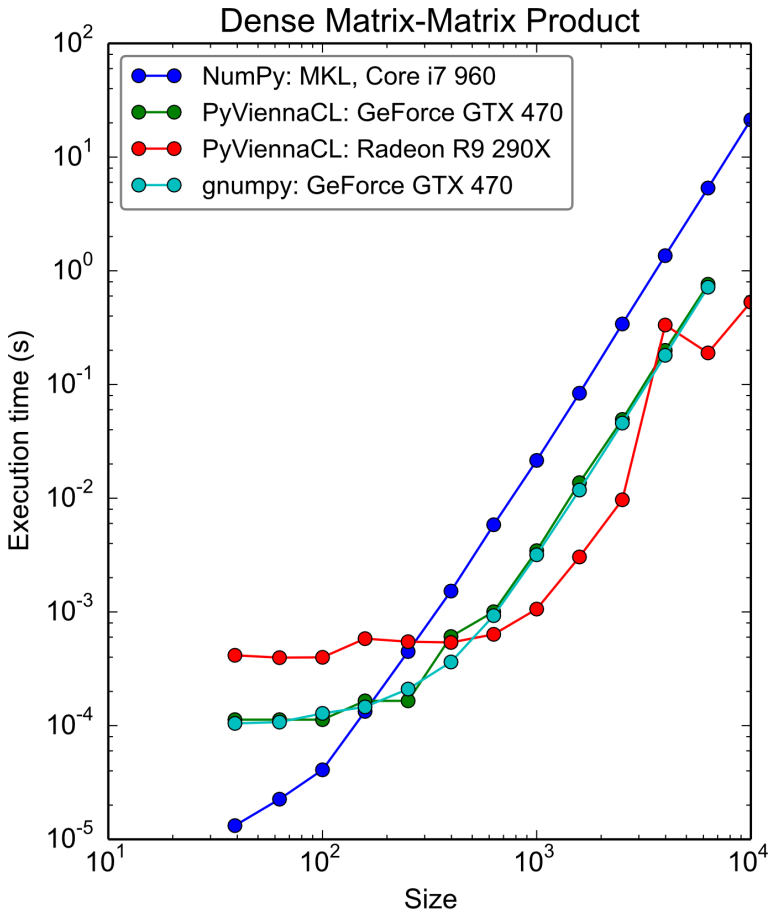
Matrix-vector product

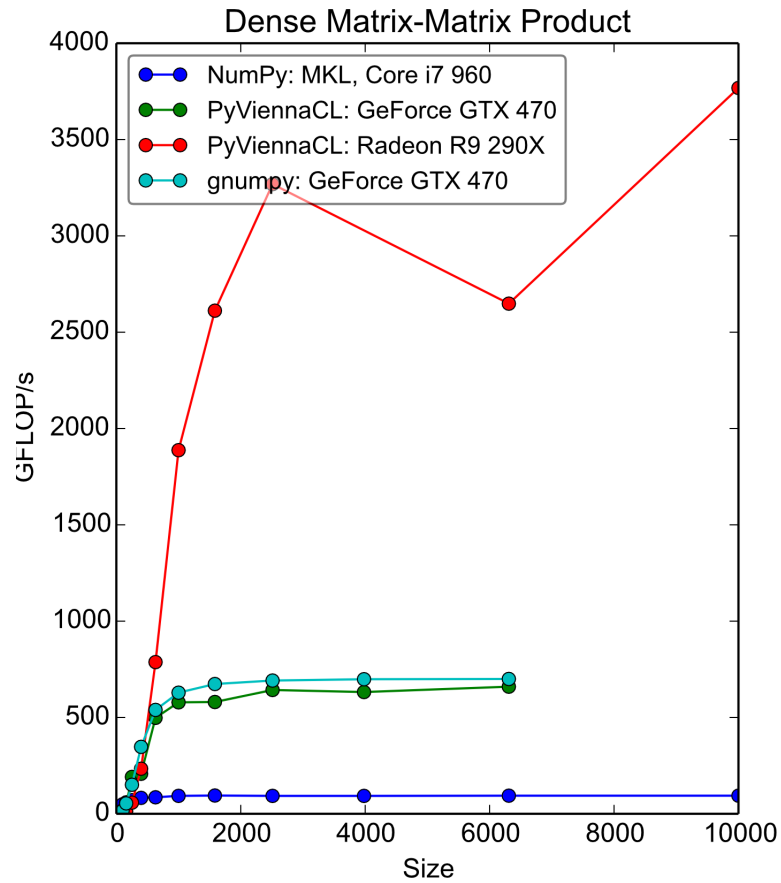




Not sure what's happened to gnumpy here..

Matrix-matrix product





Even when compute-bound, we approach cuBLAS, with more hardware support.

- Device-specific tuned kernels!
- Not quite peak performance yet..

Extending PyViennaCL: custom kernels and Nodes

What if you want to test an algorithm not in ViennaCL, using ViennaCL objects?

- Easy interoperability with PyOpenCL
- PyCUDA support coming soon
- Can integrate with the expression graph
  - `CustomNode` subclasses
- Some simple examples...

PyOpenCL interop example

```
# Suppose v, w, and x are PyViennaCL Vector instances
import pyopencl as cl

prg = cl.Program(ctx, """
__kernel void sum(__global const float *a,
                  __global const float *b,
                  __global float *c) {
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}
""").build()

prg.sum(queue, v.shape, None,
        v.handle.buffer,
        w.handle.buffer,
        x.handle.buffer)
# Now x holds result of v + w
```

CustomNode example

```
from pyviennacl import Vector, Matrix, CustomNode
from pyviennacl.backend import OpenCLMemory

class CustomSum(CustomNode):
    result_types = { ('Vector', 'Vector'): Vector,
                     ('Matrix', 'Matrix'): Matrix }

    kernels = { OpenCLMemory:
                { ('Vector', 'Vector'): vector_src,
                  ('Matrix', 'Matrix'): matrix_src } }

# Suppose v and w are as before
x = CustomSum(v, w) # x is a CustomSum instance

# Can do PyViennaCL operations involving the CustomSum
z = (x-(v+w)).norm(2) # Computed using ViennaCL
```

What's next?

Some bits of ViennaCL API not yet fully exposed:

- QR factorisation
- FFT
- Preconditioners
- Structured matrices

(Py)CUDA support

Further optimisation!

What's missing?

Users! (Hard to put a number on it...)

We've had a number of enquiries and external bug reports, and it's clearly a rough methodology... I'd be surprised if we had 20 regular users.

So:

- Try PyViennaCL!
  - `pip install pyviennacl`
  - New version out soon..
- Give us suggestions!
- Report bugs!
- Talk to me!

Thank you!

Any questions?