codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

SYCL-BLAS: Leveraging Expression Trees for Linear Algebra

Jose Aliaga (Universitat Jaume I, Castellon, Spain),
Ruyman Reyes, Mehdi Goli (Codeplay Software)

About me...

- Phd in Compilers and Parallel Programming
  - Created the first Open Source OpenACC implementation
- Background in HPC, programming models and compilers
  - Worked in HPC Scientific Code (ScaLAPACK, GROMACs, CP2K)
- Contributor to SYCL Specification
- Product Lead of ComputeCpp (Codeplay's SYCL implementation)
- Coordinating the work on SYCL Parallel STL

What is SYCL-BLAS

SYCL-BLAS is an implementation of BLAS functionality using Expression Trees on SYCL and C++.

Although it offers a BLAS interface, the important part is the expression trees for the common operations, that can be re-used for different functionality or to fuse multiple operations on a single kernel.

SYCL-BLAS is a collaboration between Codeplay and Universitat Jaume I of Castellon (Spain).

Why SYCL-BLAS

BLAS is used in many different machine learning and scientific does as the core computational library. Many libraries are built on top of BLAS, or use some of its computational cores (such as gemm).
BLAS interface is divided into three levels, **vector**, **matrix-vector** and **matrix-matrix** operations

SYCL-BLAS offers a C++/SYCL friendly interface that allows mapping STL containers or SYCL Buffers to views, and apply operations to those views.

Expression trees are created at compile time to implement the BLAS interface.

Kernels are built with offline compiler (no runtime compilation)

CLBlast: The tuned OpenCL BLAS library

	master	development
Linux/OS X	build passing	build passing
Windows	build passing	build passing

CLBlast is a modern, lightweight, performant and tunable OpenCL BLAS library written in C++11. It is designed to leverage the full performance potential of a wide variety of OpenCL devices from different vendors, including desktop and laptop GPUs, embedded GPUs, and other accelerators. CLBlast implements BLAS routines: basic linear algebra subprograms operating on vectors and matrices.

This preview-version is not yet tuned for all OpenCL devices: **out-of-the-box performance on some devices might be poor**. See below for a list of already tuned devices and instructions on how to tune yourself and contribute to future releases of the CLBlast library.

Build Status

Build branch	master	develop
GCC/Clang x64	build passing	build passing
Visual Studio x64	build passing	build passing

clBLAS

This repository houses the code for the OpenCL™ BLAS portion of clMath. The complete set of BLAS level 1, 2 & 3 routines is implemented. Please see Netlib BLAS for the list of supported routines. In addition to GPU devices, the library also supports running on CPU devices to facilitate debugging and multicore programming. APPML 1.12 is the most current generally available pre-packaged binary version of the library available for download for both Linux and Windows platforms.

The primary goal of clBLAS is to make it easier for developers to utilize the inherent performance and power efficiency benefits of heterogeneous computing. clBLAS interfaces do not hide nor wrap OpenCL interfaces, but rather leaves OpenCL state management to the control of the user to allow for maximum performance and flexibility. The clBLAS library does generate and enqueue optimized OpenCL kernels, relieving the user from the task of writing, optimizing and maintaining kernel code themselves.

VexCL documentation

VexCL is a vector expression template library for OpenCL/CUDA. It has been created for ease of GPGPU development with C++. VexCL strives to reduce amount of boilerplate code needed to develop GPGPU applications. The library provides convenient and intuitive notation for vector arithmetic, reduction, sparse matrix-vectork products, etc. Multi-device and even multi-platform computations are supported.

The library source code is available under MIT license at https://github.com/ddemidov/vexcl.

ViennaCL

Similar approaches

- Generate kernels at runtime via string composition
- Runtime compilation of kernels, with caching
- All define vector and matrix classes
- All wrap kernel execution on a host library call
- Some have a more C or a more C++ interface
- The Higher level ones require defining host semantics and types (e.g, a gpu vector)
- Writing these libraries, integrating them and combining multiple is challenging

```
/* Setup clBLAS */
err = clblasSetup( );

/* Prepare OpenCL memory objects and place matrices inside them. */
bufA = clCreateBuffer( ctx, CL_MEM_READ_ONLY, M * K * sizeof(*A),
                      NULL, &err );
bufB = clCreateBuffer( ctx, CL_MEM_READ_ONLY, K * N * sizeof(*B),
                      NULL, &err );
bufC = clCreateBuffer( ctx, CL_MEM_READ_WRITE, M * N * sizeof(*C),
                      NULL, &err );

err = clEnqueueWriteBuffer( queue, bufA, CL_TRUE, 0,
    M * K * sizeof( *A ), A, 0, NULL, NULL );
err = clEnqueueWriteBuffer( queue, bufB, CL_TRUE, 0,
    K * N * sizeof( *B ), B, 0, NULL, NULL );
err = clEnqueueWriteBuffer( queue, bufC, CL_TRUE, 0,
    M * N * sizeof( *C ), C, 0, NULL, NULL );

    /* Call clBLAS extended function. Perform gemm for the lower right sub-matrices */
    err = clblasSgemm( clblasRowMajor, clblasNoTrans, clblasNoTrans,
                            M, N, K,
                            alpha, bufA, 0, lda,
                            bufB, 0, ldb, beta,
                            bufC, 0, ldc,
                            1, &queue, 0, NULL, &event );

/* Wait for calculations to be finished. */
err = clWaitForEvents( 1, &event );

/* Fetch results of calculations from GPU memory. */
err = clEnqueueReadBuffer( queue, bufC, CL_TRUE, 0,
                            M * N * sizeof(*result),
                            result, 0, NULL, NULL );
```

Pure OpenCL interface

```
typedef float         ScalarType;
//typedef double     ScalarType; //use this if your GPU supports double precision

// Set up some ViennaCL objects
viennacl::vector<ScalarType> vcl_rhs;
viennacl::vector<ScalarType> vcl_result;
viennacl::matrix<ScalarType> vcl_matrix;

/* Set up and fill matrix in std_matrix here */
/* Set up and fill load vector in std_rhs here */

// copy data to GPU:
copy(std_rhs.begin(), std_rhs.end(), vcl_rhs.begin());
copy(matrix, vcl_matrix);

// Compute matrix-vector products
vcl_result = viennacl::linalg::prod(vcl_matrix, vcl_rhs);     //the ViennaCL way

// Compute transposed matrix-vector products
vcl_result_trans = viennacl::linalg::prod(trans(vcl_matrix), vcl_rhs_trans);
```

Using custom types for CPU/GPU

The SYCL-BLAS approach

SYCL already define all the host integration interface and semantics.

Developers can focus on kernel and performance.

Any SYCL-based library automatically integrates with other libraries and with C++.

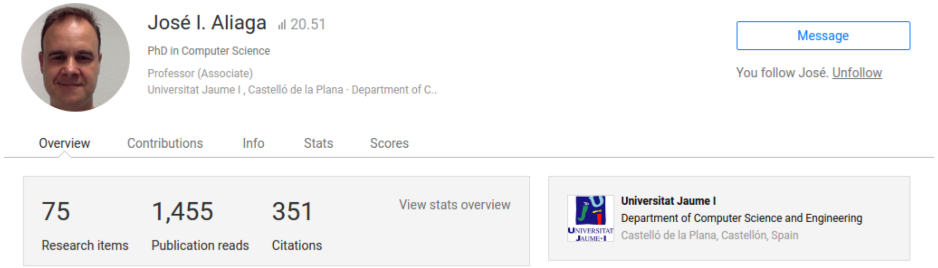
(almost) No need for custom backends: SYCL implementation provides the final mile

```
Executor<SYCL> ex(q);

{
  // CREATION OF THE BUFFERS
  buffer<double, 1> bX(vX.data(), range<1>{vX.size()});
  buffer<double, 1> bY(vY.data(), range<1>{vY.size()});

  // BUILDING A SYCL VIEW OF THE BUFFERS
  BufferVectorView<double> bvX(bX);
  BufferVectorView<double> bvY(bY);

  // EXECUTION OF THE ROUTINES
  _axpy<SYCL>(ex, bX.get_count(), 1.0, bvX, 1, bvY, 1);
}
```

Calling axpy on SYCL BLAS. Only Red boxes are library specific.

Demonstrating C++/SYCL productivity

~80% of SYCL-BLAS code so far has been implemented by Dr. Aliaga (co-author), during a research visit to Codeplay Edinburgh, from July to August 2016 (1.5 Months). Dr. Aliaga has a strong background in Dense and Sparse Linear Algebra, Clusters and CUDA, working mainly in **Fortran** and C.



He didn't know SYCL or advanced C++ when he started in July. He got a crash course on template metaprogramming and some assistance. **Started SYCL-BLAS from scratch.**

There was no public version of ComputeCpp at the moment. He worked with the internal development version of the time.

At the end of the visit, SYCL-BLAS had level 1 and 2 functions implemented, together with an initial gemm implementation.

BLAS Level 1

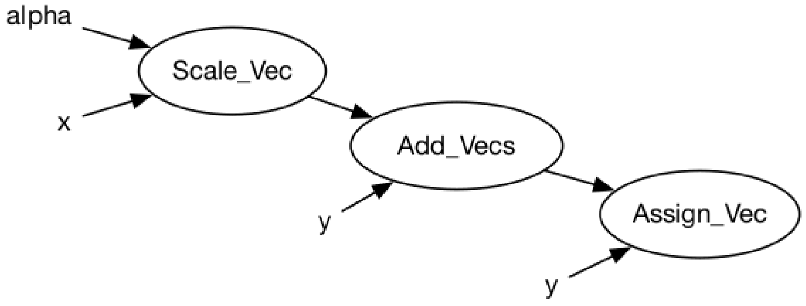
Name	Operation	Actions
_copy	$y = x$	Assign_Vector
_swap	$y \Leftrightarrow x$	Swap_Vectors
_scal	$y = \alpha x$	Scale_Vector & Assign_Vector
_axpy	$y = \alpha x + y$	Scale_Vector & Add_Vectors & Assign_Vector
_asum	$res = \sum \|x_i\|$	Reduction
i_amax	$k, \|x_k\| = \max \|x_i\|$	Special_Reduction
_dot	$res = y^T x$	Prd_Vectors & Reduction
_nrm2	$res = \sqrt{x^T x}$	Prd_Vector & Reduction & Scal_Oper
_rotg	$(c, s) = (\alpha, \beta)$	Scalar_Oper
_rot	$x = s * y + c * x$ $y = c * y - s * x$	Scale_Vector & Add_Vectors & Doble_Assign_Vectors

Expression Tree Structure

There are three types of nodes

- **Views**: Wraps a reference to a container with some extra information (e.g. stride)
- **Operations**: Classes that define operations involving views or scalars
- **Executors**: Evaluates the expression tree

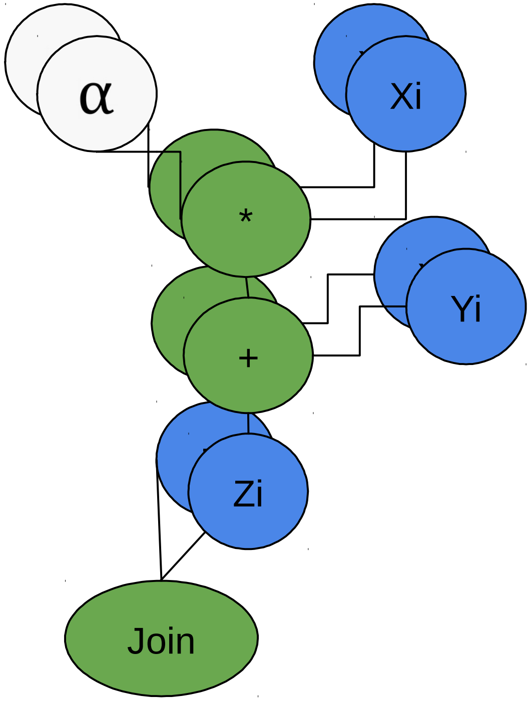
We use make functions to create the nodes and enable auto-deduction.




```
template <typename ExecutorType, typename T, typename ContainerT>
void _axpy(Executor<ExecutorType> ex, int _N, T _alpha,
           OperVectorView<T, ContainerT> _vx, int _incx,
           OperVectorView<T, ContainerT> _vy, int _incy) {
  // Creation of the operands and constants
  auto my_vx = OperVectorView<T, ContainerT>(_vx, _vx.getDisp(), _incx, _N);
  auto my_vy = OperVectorView<T, ContainerT>(_vy, _vy.getDisp(), _incy, _N);
  // Definition of the expression tree
  auto scalOp = make_prdScalar(_alpha, my_vx);
  auto addOp = make_addBinary(my_vy, scalOp);
  auto assignOp = make_assign(my_vy, addOp);
  // Execution of the expression tree
  ex.execute(assignOp);
}
```

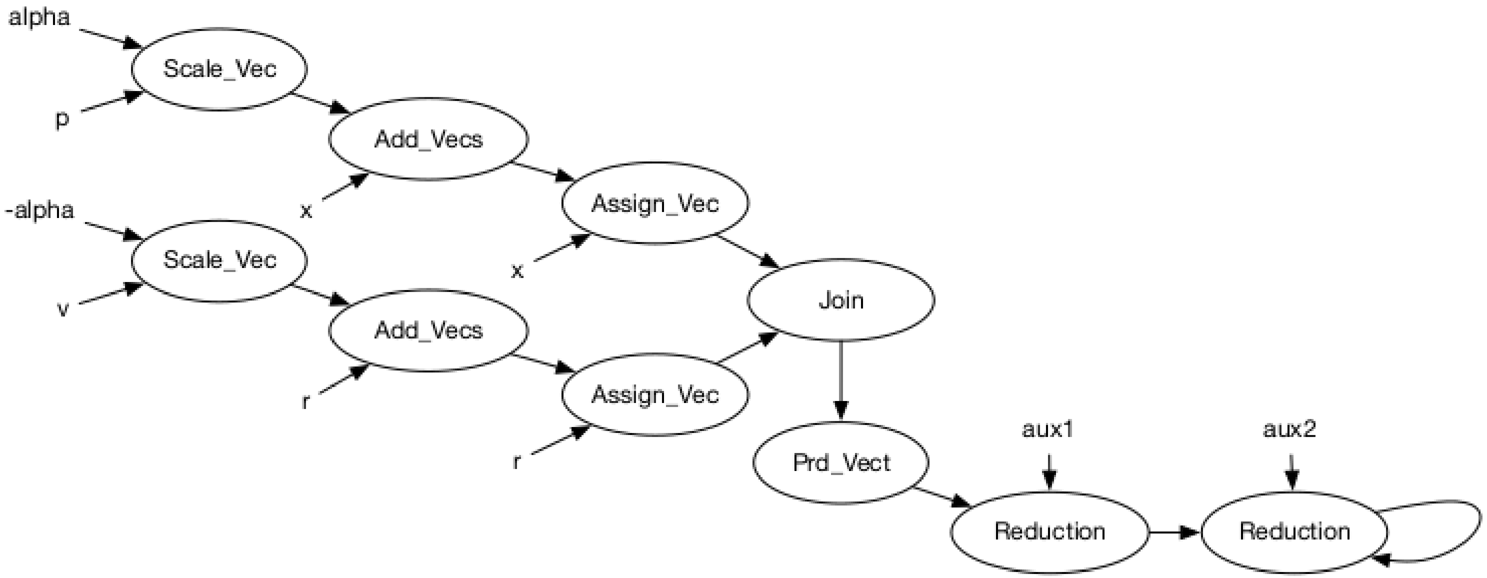
Kernel fusion

Nodes from different operations can be fused together in the same kernel.
E.g: Multiple AXPY operations can be combined on the same kernel dispatch if independent.
The **Join** node fuses multiple nodes into a single one
Using **kernel fusion** we reduce the number of data transfers and the overhead of the kernel launch.




```
Zi = a * Xi + Yi
Z`i = a` * X`i + Y`i
```

```
// concatenate both operations
auto assignOp1 = make_op<Assign>(my_vy1, addBinaryOp1);
auto assignOp2 = make_op<Assign>(my_vy2, addBinaryOp2);
auto doubleAssignOp = make_op<Join>(assignOp1, assignOp2);
```



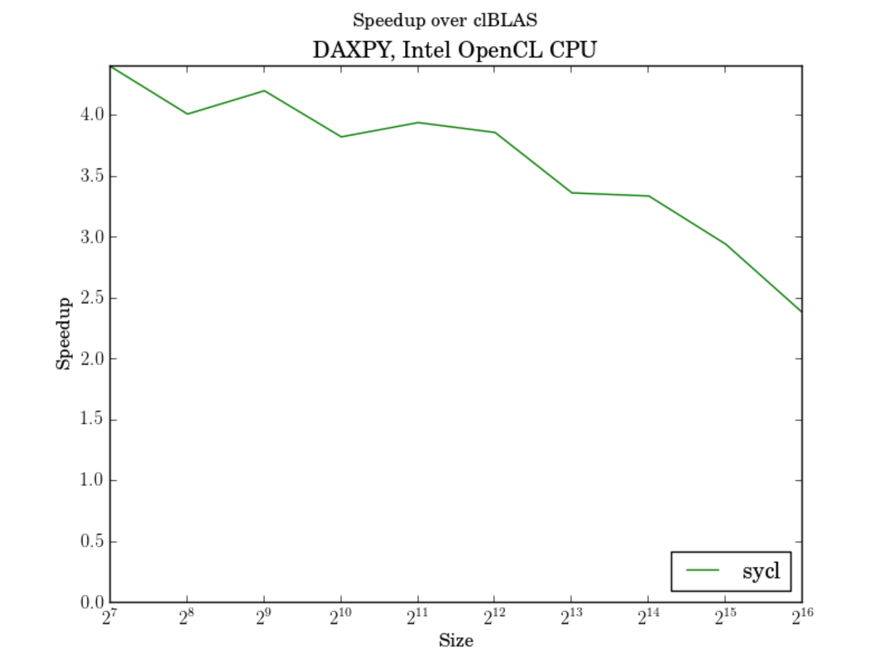
If a developer wants to fuse these expression tree..

```
template <typename ExecutorType, typename T, typename ContainerT>
void _two_axpy_dot(.. alpha1,.. _vx1,.. _vy1,.. alpha2,..._vx2,.. _vy2, .._rs,.. ) {
  // Creation of the operands and constants for 1st axpy
  auto my_vx1 = OperVectorView<T, ContainerT>(_vx1, _vx1.getDisp(), _incx1, _N);
  auto my_vy1 = OperVectorView<T, ContainerT>(_vy1, _vy1.getDisp(), _incy1, _N);
  // Definition of the expression tree for 1st axpy
  auto scalOp1 = make_prdScalar(_alpha1, my_vx1);
  auto addOp1 = make_addBinary(my_vy1, scalOp1);
  auto assignOp1 = make_assign(my_vy1, addOp1);
  // Creation of the operands and constants for 2nd axpy
  auto my_vx2 = OperVectorView<T, ContainerT>(_vx2, _vx2.getDisp(), _incx2, _N);
  auto my_vy2 = OperVectorView<T, ContainerT>(_vy2, _vy2.getDisp(), _incy2, _N);
  // Definition of the expression tree for 2nd axpy
  auto scalOp2 = make_prdScalar(_alpha2, my_vx2);
  auto addOp2 = make_addBinary(my_vy2, scalOp2);
  auto assignOp2 = make_assign(my_vy2, addOp2);
  // Join both axpy's
  auto joinOp = make_join (assignOp1, assignOp2);
  // Creation of the operands and constants for the reduction
  auto localSize = 256;
  auto nWG = 512;
  auto my_rs = OperVectorView<T, ContainerT>(_rs, _rs.getDisp(),     1,  1);
  // Definition of the expression tree for the dot
  auto prdOp  = make_prdBinary(my_vx, my_vy);
  auto assignOp3 = make_addReducAssignNewOp2(my_rs, prdOp, localSize, localSize*nWG);
  // Execution of the expression tree
  ex.reduce(assignOp3);
}
```

... can write the C++ code to fuse it

Performance

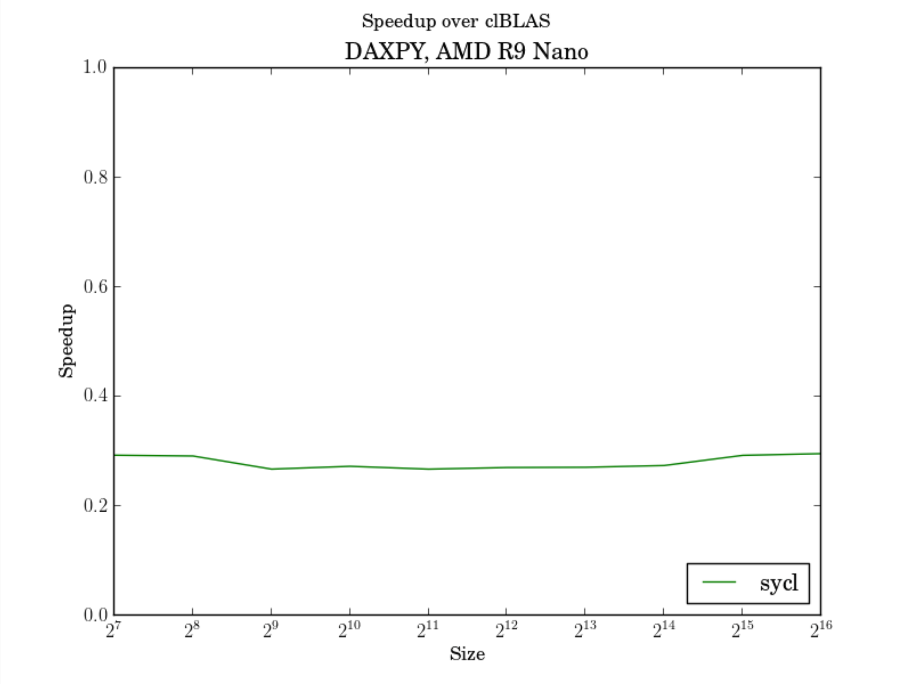
We obtain speedup over clBLAS on Intel CPU



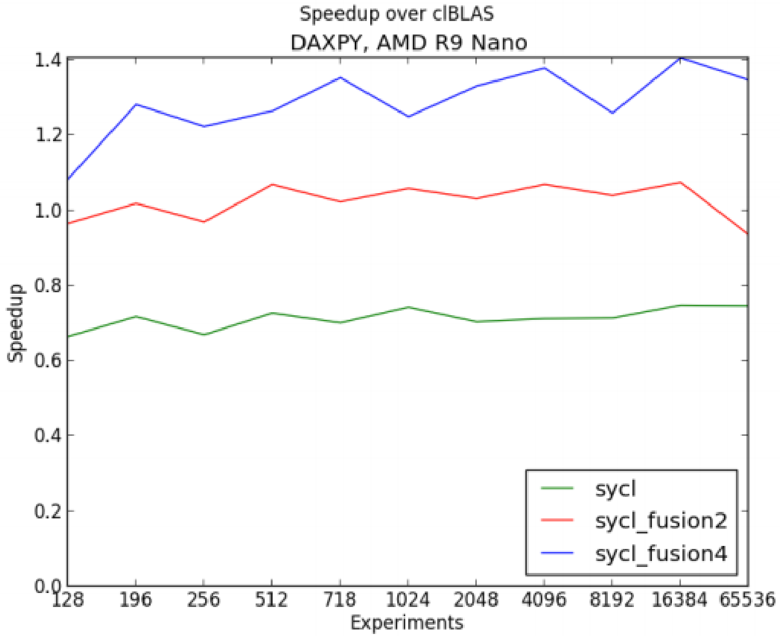
Performance

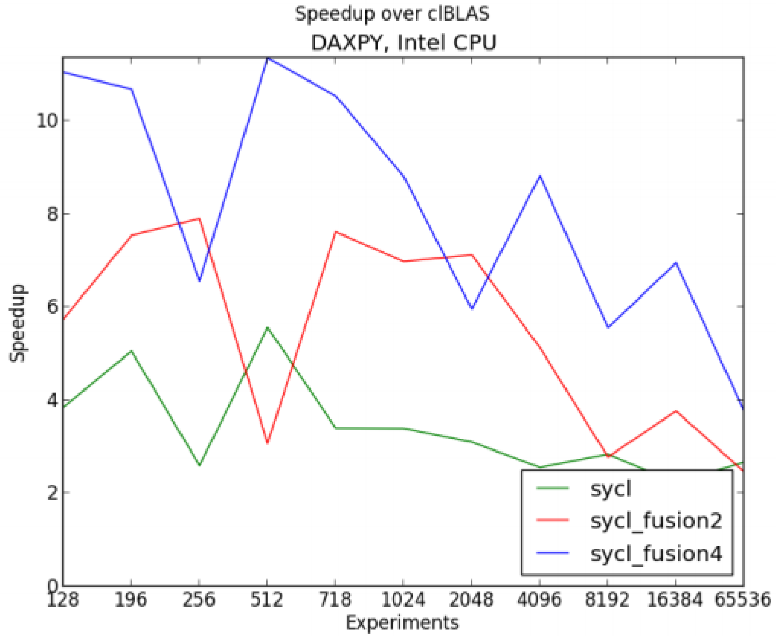
But not on the GPU

Possibly due to missing vector load/store nodes



Using fusion improves performance on all platforms





More fusion-oriented paper accepted in ParCo 2017!

Status and Future work

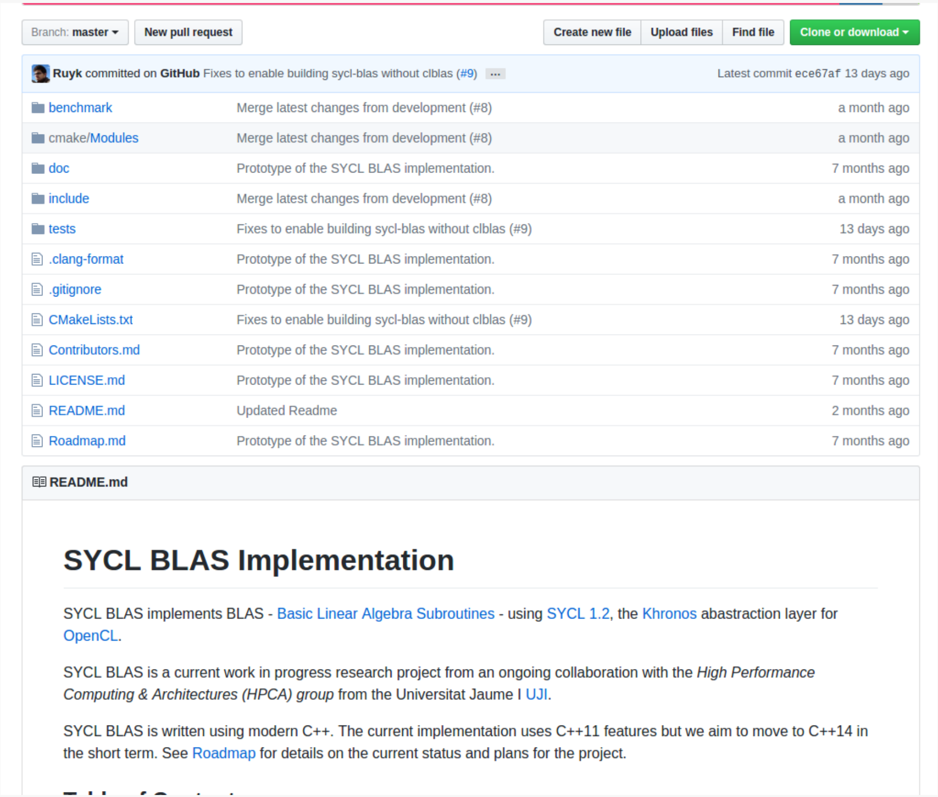
Status

- BLAS LVL1 and LVL2 implemented. GEMM from LVL 3 prototype implementation available.
- Currently analyzing performance of LVL1, identifying performance bottlenecks (e.g, missing vload/vstore functions).
- Working on a higher-level DSL using operator overloading to simplify re-using nodes and express kernel fusion.
- Planning to use multi-stage programming

How do we use SYCLBLAS:

- Ideas and experimental approaches are tested/designed in SYCL-BLAS, then ported to other frameworks (e.g, Eigen).
- Providing feedback to the committee and to the ComputeCpp implementation, e.g: missing vload/vstore from specification!
- Provide feedback to our Eigen/TF work

Help Wanted!



Interns and research visitors coming back to Edinburgh over the summer

https://github.com/codeplaysoftware/sycl-blas

codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Thanks for your attention

@codeplaysoft

info@codeplay.com

codeplay.com