# Unified, Cross-Platform, Open-Source Library Package for High-Performance Computing

Phase 2

Final Report

May 15, 2017

DE-SC0009514

EM Photonics, Inc.

Newark, DE 19711

## Table of Contents

## 1 SUMMARY

The purpose of this project is to expand the usability of supercomputers and desktop computers by the creation of a wide-ranging library for mathematical and scientific computations. The current state of the art presents a fragmented space across math/science domains and across different hardware types. As such, considerable programmer effort is expended on repetitive details and the results are sometimes suboptimal performance, and sub-optimally performing systems are wasteful of time, energy, and money.

The work in our project unifies science and math domains (in terms of the coding required) and provides very highly performing library instances of those functions. By leveraging OpenCL, the most widely supported programming interface for parallel computation, our codes will be portable to a wide range of vendors and hardware architectures.

This work can be applied to most scientific and mathematical codes that use canonical algorithms. The result to the user is simpler code, less code to maintain, and improved performance.

We are releasing our code under an open source license to encourage adoption and coding participation by the community. We are offering implementation services (i.e., creation of new routines or use of new hardware platforms) and alternative licensing options to help fund the work beyond Phase II.

## 2 TECHNICAL PROGRESS

### 2.1 BLAS SUPPORT

During the Phase II effort, we extended our Affinimath library BLAS routines, which includes full support for real-number routines as well as partial complex support. The latter proved to be challenging due to the lack of native complex support in OpenCL 1.2. With slight source modification, complex support can be enabled, but would break compatibility with Nvidia devices. To address this, we have begun to leverage an internally-developed OpenCL kernel development framework, discussed later in this project. This framework has been used to successfully enable partial complex GEMM support, as well as FFT algorithms.

The list of supported BLAS1 routines includes:

- (S/D)ASUM
- (S/D)AXPY
- (S/D)COPY
- (S/D/DS/SDS)DOT
- I(S/D)AMAX
- (S/D)NRM2
- (S/D)ROT
- (S/D)ROTM
- (S/D)SCAL
- (S/D)SWAP

The list of supported BLAS2 routines includes:

- (S/D)GBMV
- (S/D)GEMV
- (S/D)GER
- (S/D)SBMV
- (S/D)SPMV
- (S/D)SPR
- (S/D)SPR2
- (S/D)SYMV
- (S/D)SYR
- (S/D)SYR2
- (S/D)TBMV
- (S/D)TBSV
- (S/D)TPMV
- (S/D)TPSV
- (S/D)TRMV
- (S/D)TRSV

The list of supported BLAS3 routines includes:

- (S/D)GEMM, with support for non-transposed single (C) and double (Z) precision complex
- (S/D)SYMM
- (S/D)SYR2K
- (S/D)SYRK
- (S/D)TRMM
- (S/D)TRSM

We anticipate being able to quickly build out support for the remainder of the data types now that we have the facilities to generate kernels that use complex floating point math, enabled by the use of our internal kernel development library.

## 2.2 LAPACK Routine Support

In addition to BLAS algorithms, we also developed a subset of widely used LAPACK routines.

- (S/D)GETRF – LU Factorization
- (S/D)GETRS – LU Solver
- (S/D)GEQRF – QR Factorization
- (S/D)POTRF – Cholesky
- (S/D)TRTRI – Matrix inverse

These routines are implemented as blocked algorithms that rely on Level 3 BLAS to do most of the problem and an unblocked version of itself to do the small remaining portion; however, because the iterative strategy of blocking algorithms requires offsets, this introduces an issue with creating the sub-buffers that these internal functions will operate on.

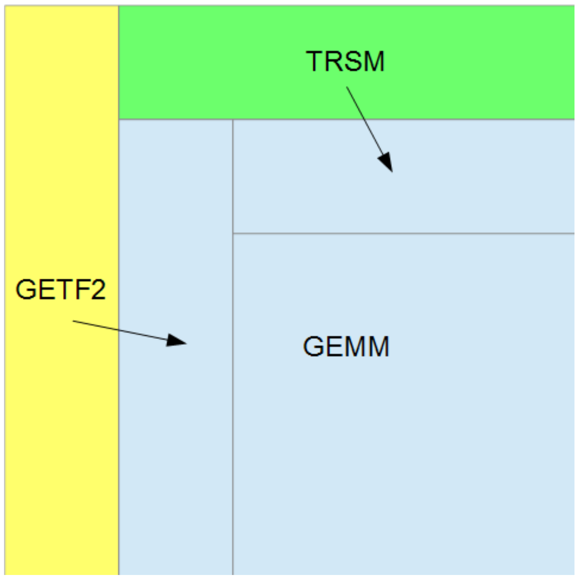


*Figure 1 - Blocking scheme of GETRF*

OpenCL's *clCreateSubBuffer* function will fail if the offset that the sub-buffer will start at is not aligned to the device's bit alignment. This would mean that the offset of each column would need to be aligned to this value. This affects the leading dimension argument that accompany matrices in the parameter list. The leading dimension must be a multiple of the number of values that can fit within the device's bit alignment.

```
device.getInfo<CL_DEVICE_MEM_BASE_ADDR_ALIGN>() / (8 * sizeof(T));
```

*Figure 2 - Formula to determine the required number of items in a leading dimension*

We added the functions *affinimathLDMultiple[TYPE]* for floating point and complex types to the host interface to retrieve this value. This only is necessary for the interface that takes *cl*_mem objects as parameters. The interface that takes host buffers internally accounts for this issue. In addition to the aforementioned LAPACK algorithms, this is also a requirement for the TRMM and TRSM Level 3 BLAS algorithms since they also use a blocking scheme.

## 2.3 FFT Functions

In addition to linear algebra functions, we have added a suite of FFT functions to our library. These include support for the following transforms:

- 1D complex-to-complex
- 1D real-to-complex
- 1D complex-to-real
- Simultaneous execution of multiple 1D transforms
- 2D complex-to-complex
- 2D real-to-complex
- 2D complex-to-real

The implementation of our FFT uses our kernel EDSL to simplify the use of complex data types. Additional kernels were written to facilitate the handling of real-valued inputs and outputs.

The initial version of this library was designed to facilitate ease of use. Many FFT software packages expose functions for FFT planning, which allows for more-efficient execution of transforms, if the original plan can be reused. Our current version does not expose this feature, but is anticipated as future work.

To summarize, we envision the following improvements to this feature:

- Support for planned FFTs
- Support for batched 2D FFTs
- Interface refinement and general improvement

## 2.4 Sparse Linear Algebra

We have also added sparse algorithms to the library. We currently have a verified Sparse GEMV implementation, which has been tested on large data sets. Additionally, we are nearing the completion of helpful, accelerated routines to convert between popular sparse matrix representations (e.g. CSR, COO, etc.). Furthermore, we are nearing completion of sparse CG and MINRES solvers, which will allow users to solve large systems of equations using our OpenCL library. We have also added matrix input routines supporting the commonly used matrix market format[1].

### 2.4.1 Sparse GEMV

Multiplication of vectors by sparse matrices is important is several fields. This primitive makes up the most fundamental operation of many equation solvers such as CG and MINRES. These solvers have widespread use in machine learning, fluid dynamics, computational electromagnetism and many other computational science disciplines. Interestingly, this primitive can also be used for graph operations, and we have used this to develop high performance graph analytics based on CUDA[1]. By creating a high-performance OpenCL implementation of this algorithm, we make this technology available to users of OpenCL based hardware platforms such as the Xeon Phi or AMD GPUs.

To create this implementation, we surveyed the literature for high performance GPU MV implementations on matrices stored in the CSR format. CSR based matrix vector product has a very simple naïve implementation that maps well to GPUs. Interestingly for many matrices, this is often the fastest method[2]. A more advanced method is based on merge paths[3]. Currently, we have implemented the naïve method, but may implement the merge path based method in the future as we have implemented merge path based techniques to perform vectorized sorted searches that are used in our conversion algorithms.

[1] S. Kozacik, A. L. Paolini, P. Fox, and E. Kelmelis, "Many-core graph analytics using accelerated sparse linear algebra routines," 2016, vol. 9848, pp. 984808-984808–6.
[2] M. Steinberger, A. Derlery, R. Zayer, and H. P. Seidel, "How naive is naive SpMV on the GPU?," in 2016 IEEE High Performance Extreme Computing Conference (HPEC), 2016, pp. 1–8.
[3] D. Merrill and M. Garland, "Merge-based Sparse Matrix-vector Multiplication (SpMV) Using the CSR Storage Format," in Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, 2016, p. 43:1–43:2.

We have created both transposed and non-transposed versions of this algorithm. The transposed versions require atomically accumulating the result vector values as it is not possible to assign each value to a thread due to the CSR format. While OpenCL does not have floating point or double precision atomic addition we implemented these operations using atomic compare and swap. Unfortunately, 64-bit atomic compare and swap is an extension not supported on all platforms so double precision transposed CSR GEMV cannot be supported on these platforms.

### 2.4.2 Format Conversions

As mentioned in the previous section our GEMV operates on sparse matrices stored in the CSR format. To support matrices built and imported from common formats we added conversion operators from the COO or coordinate format. COO is a simple format comprised of 3 arrays of the row indices, column indices, and values. We have built a conversion routine that runs on the CPU for converting this to the CSR format. In this case, the conversion routine is mostly composed of C++ standard template library algorithms. The main issue that we encountered was that it is not possible in C++ to sort one array using another as a key without making a temporary array of the pairs of values. To combat this, we developed a custom “proxy” iterator that iterates over both arrays return a tuple of references to elements when dereferenced. This currently works with both standard sort and a high performance radix based sort from the *boost.Sort* library[4].

We currently have a working conversion routine implemented for devices that operates in the special case when row and column indices are sorted. In this case, the main operation is converting row indices to row pointers. While on the CPU this is done with a for loop that repeatedly calls *std::upper_bound* on the COO row index array with the value being the currently searched for row, on the GPU it is possible to perform searches for many row indices in parallel. This is known as a vectorized sorted search. We have converted a high-performance CUDA implementation from an early version of the Modern GPU library[5] to OpenCL. This uses a merge path[6] technique, which consists of performing a rough version of the upper bound where instead of finding the upper bound for each row it finds the upper bound that each processor will be working on to distribute work evenly between processors to maintain high performance. The Modern GPU library is heavily templated, which made porting this to OpenCL difficult. We modified the algorithms to remove the use of functors, instead replacing these with global functions that performed only the operations we needed. Additionally, we created implementations of the functions for each template parameter that was required by our algorithm.

We are currently working on implementing this in the general case on the device. This will require among other algorithms, a segmented sort by key. This is also contained in the Modern GPU library and we will port this to OpenCL in the future.

### 2.4.3 CG and MINRES Solvers

We are currently in the process of implementing sparse solvers that use the conjugate gradient (CG) method and “Generalized minimal residual method” (MINRES). These are based on the solvers in our

[4] http://www.boost.org/doc/libs/1_64_0/libs/sort/doc/html/index.html
[5] https://moderngpu.github.io/sortedsearch.html
[6] O. Green, R. McColl, and D. A. Bader, “GPU merge path: a GPU merging algorithm,” in Proceedings of the 26th ACM international conference on Supercomputing, 2012, pp. 331–340.

CULA Sparse library. The solvers build upon both our sparse CSRMV kernels and host code, as well as dense BLAS kernels and host code such as AXPY.

## 2.5 Library Infrastructure and Interfaces

We have continued to improve upon our library infrastructure and interfaces through the addition kernel source embedding and generation; simplifying the dispatch procedure; library context management; and the implementation of a C user interface.

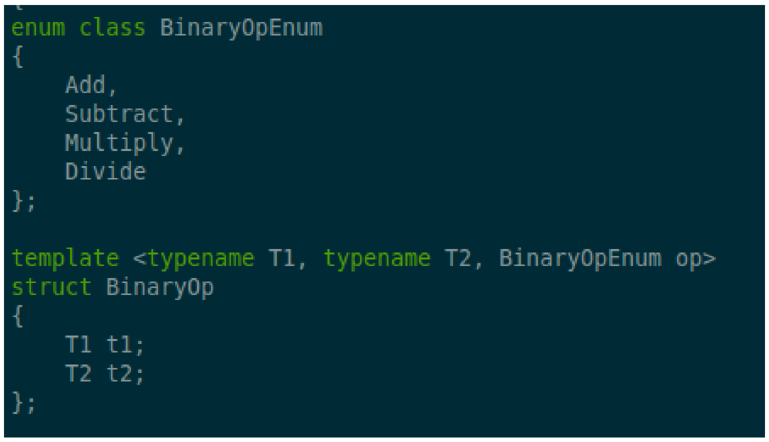
### 2.5.1 Kernel Source Embedding

Currently, we have a program that will run during the CMake build step to convert a native OpenCL file into a C++ source file that has a char array of hex values and a length, representing a raw version of the kernel source (e.g. ASCII values). This C++ source file will be built with our library, effectively embedding the raw kernel source into the library. When our *cl_programs* are being built, they will use the symbols representing this char array and size to create the binary.

### 2.5.2 EDSL for Source-to-Source Translation of Kernel Codes

An EDSL (embedded domain specific language) is a novel language, limited to a narrow domain, implemented using the infrastructure of an existing programming language. Typically, this means creating a language without parsing it as text from a file or a string. In many cases, custom data types such as C++ or Java classes are required. These data types are then used to define and select between behaviors. Operator overloading, the semantic redefinition of an operator such as the plus sign, is often used to allow a more traditional look and feel.

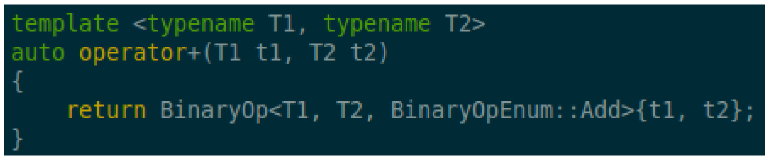
EDSLs in C++ make use of expression templates[7] to build "parse" trees representing the code. Expression templates are function and class templates that encode the result of an expression. These expressions are built up recursively, when a new operation occurs its type encodes is based on the types of its operands. Below is an example expression template and the addition operator that can be used to create expressions.



```
enum class BinaryOpEnum
{
    Add,
    Subtract,
    Multiply,
    Divide
};

template <typename T1, typename T2, BinaryOpEnum op>
struct BinaryOp
{
    T1 t1;
    T2 t2;
};
```

[7] T. Veldhuizen, "C++ Gems," S. B. Lippman, Ed. New York, NY, USA: SIGS Publications, Inc., 1996, pp. 475–487



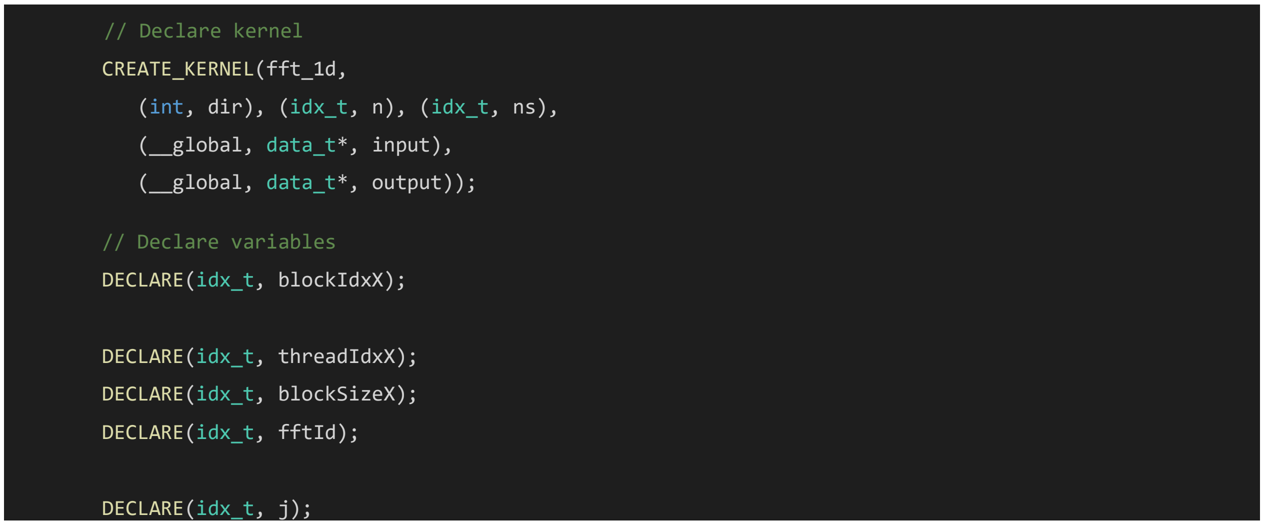
```
template <typename T1, typename T2>
auto operator+(T1 t1, T2 t2)
{
    return BinaryOp<T1, T2, BinaryOpEnum::Add>{t1, t2};
}
```

*Figure 3 - Example Expression template*

EDSLs can be useful in several situations. A common case is when solving a problem that seems more like a language than a library, such as creating an implementation of linear algebra or image processing algorithms. In these circumstances, a natural mathematical language exists for describing these problems and, while a solution can be represented as a library using function calls or classes, it can be described more intuitively using a language that closely mirrors the mathematical language.

Another example arises when the optimal computational structure for a problem is different than the optimal programming structure. Specifically, domain experts would generally prefer to program in an intuitive fashion without taking into consideration performance concerns[8]. A representative project is the Halide image processing library[9], where algorithms are decoupled from their schedule (order) and locality (storage). The ability to cleanly separate a problem definition from its underlying implementation details is a strength of EDSLs, and makes it possible for Halide to compile a single piece of code for multiple hardware platforms.

Our goal in this project is performance portability across architecturally different platforms, requiring conditional code generation and execution. The development of this library is complicated by the lack of OpenCL 2.0 support for NVIDIA platforms, which is now experimental in recent driver versions. Because of this, C++ usage is limited and complex data types are not supported natively. To overcome these difficulties, we developed an EDSL that automatically generates kernel code and is designed to look like C/C++ where possible, making it familiar to C, C++ and OpenCL developers. **At runtime, our EDSL source code is transformed into valid OpenCL source code.**



```
// Declare kernel
CREATE_KERNEL(fft_1d,
   (int, dir), (idx_t, n), (idx_t, ns),
   (__global, data_t*, input),
   (__global, data_t*, output));

// Declare variables
DECLARE(idx_t, blockIdxX);

DECLARE(idx_t, threadIdxX);
DECLARE(idx_t, blockSizeX);
DECLARE(idx_t, fftId);

DECLARE(idx_t, j);
```

[8] J. M. Ragan-Kelley, "Decoupling algorithms from the organization of computation for high performance image processing," Massachusetts Institute of Technology, 2014.
[9] "Automatically Scheduling Halide Image Processing Pipelines." [Online]. Available: http://graphics.cs.cmu.edu/projects/halidesched/. [Accessed: 04-May-2017]

```
    DECLARE(idx_t, r);
    DECLARE(idx_t, radix);
```
*Figure 4 - EM Photonics's OpenCL Kernel language EDSL showing kernel and variable declaration*

The above code shows the kernel declaration and declaration of local variables for a general matrix-matrix multiply kernel. This shows how the kernel can be created for different data types where necessary and is an example of using macros to hide unsightly template syntax. Note that the usage of these macros is optional and the underlying types are directly accessible. With this language, it is possible to declare both global and local memory arguments as well as scalars.

The syntax for kernel definition is shown in the code below. This code shows how standard accelerator functions such as thread ID and barriers can be accessed and is a good example of the similarities and differences between C and the EDSL syntaxes. C structures such as for loops and if statements are present, but different characters and capitalization are necessary to prevent name collisions.

```
// Enable kernel EDSL within this scope
INIT_KERNEL

// Enable support for double precision, complex types, etc. (if necessary)
ENABLE_DATA_TYPE_SUPPORT(data_t)

// Create a kernel definition
CREATE_KERNEL(gemm_gpu_notransa_notransb,
    (idx_t, M), (idx_t, N), (idx_t, K), (data_t, alpha),
    (__global, data_t*, A), (idx_t, a_offset), (idx_t, lda),
    (__global, data_t*, B), (idx_t, b_offset), (idx_t, ldb), (data_t, beta),
    (__global, data_t*, C), (idx_t, c_offset), (idx_t, ldc),
    (idx_t, Kwg), (__local, data_t*, scratchA), (__local, data_t*, scratchB));

// Declare variables
DECLARE(idx_t, lid0);
DECLARE(idx_t, lid1);
DECLARE(idx_t, gid0);
DECLARE(idx_t, gid1);
DECLARE(idx_t, Mwg);
DECLARE(idx_t, Nwg);

DECLARE(idx_t, pwg);
DECLARE(idx_t, kend);
DECLARE(idx_t, kind);
DECLARE(idx_t, pwi);
DECLARE(idx_t, c_ind);
DECLARE(idx_t, one);

DECLARE(data_t, zero);
DECLARE(data_t, temp);
DECLARE(data_t, ctemp);

// Begin kernel logic
KERNEL_SECTION_START

lid0 = ThreadId(0),
lid1 = ThreadId(1),
gid0 = GlobalThreadId(0),
gid1 = GlobalThreadId(1),
Mwg = BlockSize(0),
Nwg = BlockSize(1),

zero = 0,
one = 1,
c_ind = gid0 + gid1 * ldc + c_offset,
temp = 0,

If(alpha != zero)
(
    For(pwg = 0, pwg < K, pwg = pwg + Kwg)
    (
        kend = Min(pwg + Kwg, K) - pwg,

        If(gid0 < M)
        (
            For(kind = lid1, kind < kend, kind = kind + Nwg)
            (
                scratchA[kind * Mwg + lid0] = A[gid0 + (kind + pwg) * lda + a_offset]
            )
        ),

        Else()
        (
            For(kind = lid1, kind < kend, kind = kind + Nwg)
            (
                scratchA[kind * Mwg + lid0] = zero
            )
        ),

        If(gid1 < N)
        (
            For(kind = lid0, kind < kend, kind = kind + Mwg)
            (
                scratchB[kind * Nwg + lid1] = B[(pwg + kind) + gid1 * ldb + b_offset]
            )
        ),

        Else()
        (
            For(kind = lid0, kind < kend, kind = kind + Mwg)
            (
                scratchB[kind * Nwg + lid1] = zero
            )
        ),

        Barrier(),

        For(pwi = 0, pwi < kend, pwi = pwi + one)
        (
            temp = temp + scratchA[pwi * Mwg + lid0] * scratchB[pwi * Nwg + lid1]
        ),

        Barrier()
    )
),

If(gid0 < M)
(
    If(gid1 < N)
    (
        If(beta != zero)
        (
            ctemp = beta * C[c_ind]
        ),
        Else()
        (
            ctemp = zero
        ),

        ctemp = ctemp + alpha * temp,
        C[c_ind] = ctemp
    )
),

// End kernel logic
KERNEL_SECTION_END
KERNEL_FINALIZE
```

*Figure 5 - EM Photonics OpenCL kernel language showing kernel definition for generalized matrix-matrix multiply*

The above system makes heavy use of modern C++ language features such as automatic return types, variadic templates, and advanced template metaprogramming functionality. This use of new language features serves to create clean-looking EDSL code, hiding implementation complexity from the end user.

Currently, our EDSL supports the following operations:

- Kernel creation and finalization
- Support for multiple-kernel creation
- Variable creation and assignment
- Creation of fixed-size arrays
- Use of local and global OpenCL memory arguments
- Arithmetic operations (+, -, *, /, %)
- Array access for read and write operations
- Complex number support, including creation, conjugation, and part-wise access
- Access of OpenCL workgroup indices
- Cosine, sine, min, and max operations
- Looping
- Control flow

The current EDSL allows for the cross-platform definition of a broad variety of OpenCL kernels; however, this EDSL may also be easily extended to support new operations, as well as new, non-OpenCL backends (or backends to support different OpenCL standards). The latter ability is enabled due to the decoupling of our EDSL frontend and the OpenCL-creating backend. For instance, the following EDSL structure shows the underlying implementation of workgroup ID lookup:

```
template <typename T>
struct BlockIdx : public Value<BlockIdx<T>>
{
    using Value<BlockIdx<T>>::operator=;

    BlockIdx(int dim) : Value<BlockIdx<T>>(""), dim(dim)
    {
    }

    std::string dump(GenModifiers mod = GenModifiers::NONE) override
    {
        return "get_group_id(" + std::to_string(dim) + ")";
    }

    int dim;
};
```

*Figure 6 - Implementation of BlockIdx class*

In this simple case, the "dump" method is responsible for returning the OpenCL code that fetches the kernel's workgroup ID value; however, this code can easily be swapped out to create CUDA-compatible code.

Currently, this EDSL has been employed to enable a non-transposed complex GEMM, as well as the full extent of our FFT support. Future work on this project will focus on transitioning our existing OpenCL kernels into this new EDSL to further enhance the “write-once” goal of this project, as this effort will effectively decouple the remainder of our kernel code from the underlying language (e.g. OpenCL).

Our EDSL also inherently captures AST-like dependency information. In other words, at the moment of assignment, we can obtain a limited view of data dependency information associated with statement. This characteristic could prove useful going forward for in-EDSL optimization techniques.

### 2.5.3 Kernel Program Class

We created a new program class because the concepts between the plans and functors became blurred. For example, our matrix-vector BLAS functions are implemented in two ways, a GPU version that requires a multiply and reduce kernel and a CPU version that requires just a multiply. The GPU version would also require a workspace that would need to be allocated. All of this would be contained within the functor since it would be responsible launching these tightly-related kernels; however, that is nearly the same job the plan has with regards to resource allocation and subdivision. For example, as we mention later in this section, an algorithm (such as GEMM) can be subdivided into smaller kernel launches. It became unclear as to which class was supposed to do this task. Thus, we decided to combine the concepts into a single program class with the idea that it should know how to dispatch, size, allocate resources for its kernels.

The program class inherits the responsibilities of the functor, encapsulating the state of the GPU subroutine, dispatching to the appropriate kernel implementation, and setting up the device parameters necessary for the kernel launch. For example:

1. Selecting an optimized kernel version appropriate to the problem parameters.
2. Computing work-group and global sizes.
3. Launching multiple tightly-related kernels (for example, a multi-stage reduction).

The program also inherits some responsibilities of the plan which are problem decomposition and resource allocation. Programs can call other programs if they depend on other algorithms to compute some of their work. For example, the BLAS 3 functions can be decomposed to a number of GEMM function calls and a call to a specific kernel to do work on the diagonal. The GEMM function itself can be decomposed into many smaller GEMM kernel calls. Because the outputs of most BLAS 3 functions are independent of one another (TRSM is similar to LAPACK), there can be numerous small independent BLAS 3 kernels running in parallel. This problem decomposition is also important for several reasons.

1. Many devices are non-preemptive and have watchdog timers, so if an individual kernel runs too long, the device will be reset and any data on the device will be lost.
2. Many devices support parallel data transfer and execution. With a decomposed problem, data can be streamed to the device and back to the host on-demand in parallel with execution by using multiple OpenCL command queues. This technique can be used to hide most of the data transfer time for data originating on the host.
3. By selecting fixed problem block sizes, specialized kernels can be used that avoid costly bounds-checking and benefit from aggressive loop unrolling.
4. This provides the foundation for multi-device and hybrid solvers, where a plan can determine how to distribute work among available devices.

While we changed the resource allocation of host data to a higher level, the program may still need to allocate additional workspace memory to do extra computations. An example previously mentioned would be the multi-stage reductions. A temporary buffer would have needed to be allocated to store the accumulations before performing the final reduction that would be copied to the buffer that would return to host memory.

We plan to implement this problem decomposition in a future release as we continue to optimize our code.

### 2.5.4 Library Context

To encapsulate the state of the library and deal with initialization and teardown cleanly, we implemented a context object is that stores the library state. This class contains the current OpenCL device and context, as well as a list of programs associated with that context (kernels built for that OpenCL context/device pair).

The library creates one context object for each OpenCL device that it finds on the system and for each OpenCL platform that it finds, it creates an OpenCL context that will be shared between the appropriate context objects. These contexts are stored in a static vector. The context that is default to the current thread will be referenced through a thread-local pointer.

It is possible for there to be more context objects stored in the vector than devices on the system. If through the device interface the user passes a *cl_command_queue* that was created with an unrecognized OpenCL device/context pair, the library will create a new context object for that OpenCL device/context pair and set it as the default.

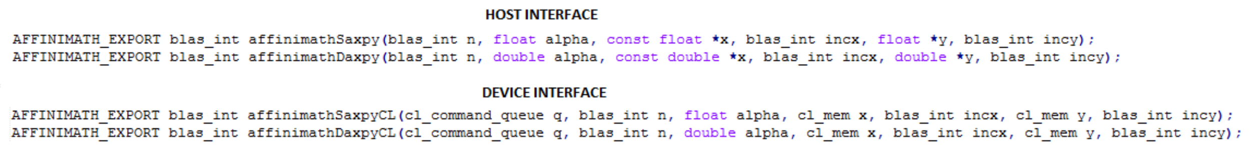
The context object contains the programs for each implemented BLAS/LAPACK function and their different floating point and complex types. These programs are normally lazily initialized by calling the corresponding BLAS/LAPACK function. The function will call the context object to retrieve the program. If the context object hasn't attempted to build it, it will create it, store it so it does not have to be built a second time, and return it to the function. If the context fails to build it, it will return an error to the function that will be propagated to the user. A known issue we have seen that would cause a program to fail to build is attempting to use OpenCL pragmas that a platform or device doesn't support. For example, one of our test devices Intel(R) HD Graphics 4400 doesn't support the cl_khr_fp64 pragma. This pragma enables the use of double precision in OpenCL kernels. Because of this, none of the double precision and complex double precision kernels that we have were able to build for that device.

The user interface exposes functions that allow for the creating, destroying, and changing the context object:

- *AffinimathInitialize* initializes the contexts and all of their programs
- *AffinimathSetDevice* initializes the contexts if they haven't been and sets a specified context to the default
- *AffinimathShutdown* cleans up the resources allocated for each context

### 2.5.5 C Interface

The interface to the library will be a standard C interface for compatibility for multiple compilers and runtimes. This will allow for the library to be dynamically linked or loaded. The interface will dispatch to the library context object for setup and destruction as well as our BLAS and LAPACK implementation. Also, it will give the user a interface to set the default device and context within the current thread. The BLAS/LAPACK interface will be exposed in two forms, a host interface and a device interface. The host interface will be in the standard form of BLAS/LAPACK. The device interface will take OpenCL objects as arguments, specifically *cl_command_queue* that will allow a kernel to be run on a specified device and *cl_mem*'s that are representative of vectors and matrices in device memory.



**HOST INTERFACE**

```
AFFINIMATH_EXPORT blas_int affinimathSaxpy(blas_int n, float alpha, const float *x, blas_int incx, float *y, blas_int incy);
AFFINIMATH_EXPORT blas_int affinimathDaxpy(blas_int n, double alpha, const double *x, blas_int incx, double *y, blas_int incy);
```

**DEVICE INTERFACE**

```
AFFINIMATH_EXPORT blas_int affinimathSaxpyCL(cl_command_queue q, blas_int n, float alpha, cl_mem x, blas_int incx, cl_mem y, blas_int incy);
AFFINIMATH_EXPORT blas_int affinimathDaxpyCL(cl_command_queue q, blas_int n, double alpha, cl_mem x, blas_int incx, cl_mem y, blas_int incy);
```

*Figure 7 - Host and device interface of AXPY function*

An issue that the host interface may have for BLAS 1 & 2 functions is that the memory transfer time from host to device memory, and vice-versa, may be significantly higher than the actual kernel execution time, especially when using large problem sizes.

### 2.5.6 Device Enumeration

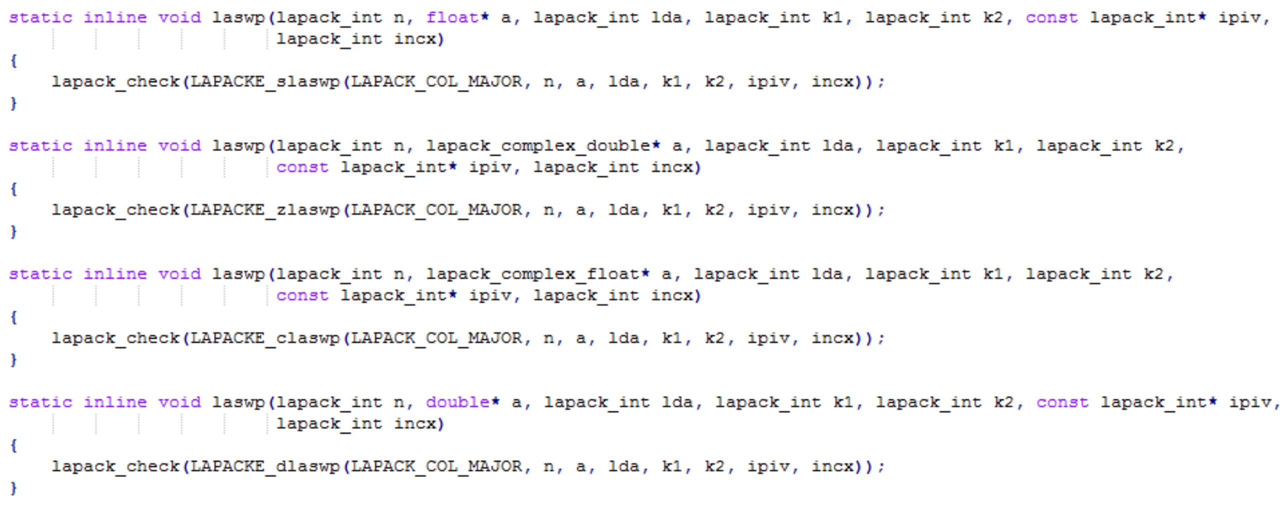
A system may have multiple OpenCL-capable devices on which kernels can be run. As such, we required a method to identify and track the devices on the system that can be targeted at run time. To address this need, we implemented a device enumeration class. On construction, this class finds OpenCL-capable devices on the system. Devices are segmented into three high-level types based on their reported OpenCL device type (a rough description of their capabilities). The first type is CPUs, which have good single-threaded performance, but limited capacity for multi-threaded execution. The second class is GPUs, which have inferior single-threaded performance, but are capable of simultaneously executions hundreds or thousands of threads simultaneously. The last type is “accelerators,” which are devices such as the GPU-like Xeon Phi.

This device enumeration class also manages device command queues, which can be retrieved via reference using the device’s integer id. Work is issued to these command queues (kernel execution, memory operations, etc.) by our library.

## 2.6 Third-Party BLAS/LAPACK Support

There are other BLAS/LAPACK libraries that have been created by other vendors, Netlib, OpenBLAS, ACML, and MKL to name a few. The standard BLAS C interface, however, is cumbersome to use, as it takes all parameters by pointer; this is a holdover from the FORTRAN roots of the BLAS and LAPACK libraries. In addition, building the library with different FORTRAN compilers such as gfortran and ifort can result in the argument order being different. In response to this, several friendlier, higher-level interfaces have emerged. In particular, CBLAS and LAPACKE interfaces are commonly offered. The CBLAS C interface eschews the pointer-only interface seen in the standard BLAS C interface, while the LAPACKE C interface handles aspects such as workspace management for the user. Nvidia also provides support for BLAS via its CUBLAS library which mostly follows the FORTRAN interface. CUBLAS requires the use of an NVIDIA GPU.

We have created header functions that wrap around the CBLAS/LAPACKE functions that provide them as a set of overloads for C++ users that would dispatch to the appropriate function depending on the floating-point type.



```
static inline void laswp(lapack_int n, float* a, lapack_int lda, lapack_int k1, lapack_int k2, const lapack_int* ipiv,
                         lapack_int incx)
{
    lapack_check(LAPACKE_slaswp(LAPACK_COL_MAJOR, n, a, lda, k1, k2, ipiv, incx));
}

static inline void laswp(lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int k1, lapack_int k2,
                         const lapack_int* ipiv, lapack_int incx)
{
    lapack_check(LAPACKE_zlaswp(LAPACK_COL_MAJOR, n, a, lda, k1, k2, ipiv, incx));
}

static inline void laswp(lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int k1, lapack_int k2,
                         const lapack_int* ipiv, lapack_int incx)
{
    lapack_check(LAPACKE_claswp(LAPACK_COL_MAJOR, n, a, lda, k1, k2, ipiv, incx));
}

static inline void laswp(lapack_int n, double* a, lapack_int lda, lapack_int k1, lapack_int k2, const lapack_int* ipiv,
                         lapack_int incx)
{
    lapack_check(LAPACKE_dlaswp(LAPACK_COL_MAJOR, n, a, lda, k1, k2, ipiv, incx));
}
```

*Figure 8 - Overloads of LASWP function*

The idea behind this header interface would be to allow the user to provide a third-party library that follow the CBLAS/LAPACKE interface and use their functions in place of ours. Since our internal classes are templated, using the header functions as wrappers would allow us to easily dispatch to their functions. This would only be applicable to the host interface as these libraries require host pointers. At the moment, we haven't added this functionality but we intend to do so in a future release.

What we have accomplished is adding an option to hook NVIDIA'S CUBLAS library. We added similar header functions to provide a set of overloads to CUBLAS functions. We would dispatch to the appropriate CUBLAS function from the host interface after copying it to device memory since CUBLAS operates on the GPU. Not all BLAS functions are supported by CUBLAS; if the user attempts to run DSDOT or SDSDOT, the function will return an unsupported status.

The option for linking to a third-party library will be a build-time option. Our CUBLAS hooks can serve as an example to how we plan to add hooks for other third-party libraries.

## 2.7 Migration to Boost Test

During the project, we identified the need to be able to auto-generate tests based on the various input (tuning) parameters. To accomplish this, we have migrated to a recent version of the Boost C++ unit testing framework. This unit testing framework allows us to effortlessly generate thousands of valid parameter combinations with which to test our developed routines, thereby ensuring good test coverage.

Essentially, parameters such as data layout, options (e.g. upper of lower triangular), and dimensions are each represented by arrays. These arrays can be concatenated (+), or combined via a cartesian product (*), the latter of which can greatly expand the testing space. Below is an example of how this generation is performed.

```
namespace bdata = boost::unit_test::data;
const storev_t layoutVals[] = {{column}};//, {row}};
const uplo_t uploVals[] = {{ upper }, {lower}};
const int nvals[] = {3, 16, 44, 140, 1024};

BOOST_DATA_TEST_CASE_F(PotrfFixture<float>, SPotrfDeviceTest,
                        bdata::make(layoutVals) * bdata::make(uploVals) * bdata::make(nvals) *
                        bdata::xrange(System::numDevices()),
                        layout, uplo, n, deviceId)
{
    DoDeviceTest(layout, uplo, n, deviceId);
}
```

*Figure 9 - Example of test case generation using Boost C++'s unit testing framework*

An issue that we had with the Boost testing framework was that there is a limit to the number of inputs to the BOOST_DATA_TEST_CASE_F macro that it uses to create the Cartesian product. Our GEMM test initially failed to build because it requires a lot of inputs for its Cartesian product. We had to patch a Boost header file to fix this issue so our tests could build. This patch is currently planned to ship with our software.

## 2.8 Polyhedral Optimization Research

At the outset of this project, we had anticipated using polyhedral optimization to perform tuning of our kernels. Polyhedral optimization is an approach to analyze dependencies within nested-loop structures to perform optimizations. Essentially, these methods work by representing the loop structure and included statements as nodes and edges in mathematical construct known as a polytope (essentially represented as a graph). After analysis, an affine transform is created to transform this structure, and the resultant polytope is converted back into new nested loop structure.

One optimization that is common within polyhedral frameworks is referred to as “skewing,” which reorders loop bounds and alters affected index calculations to achieve a more-optimized calculation of the loop without affecting the result. Another optimization that this model enables is “tiling,” which is breaks up the loop structure to improve aspects such as caching.

Recent developments in the literature have shown that such optimization can also be performed using LLVM IR[10]. In this case, programs are transformed from a high-level language into LLVM IR using any of a wide variety of frontends. This LLVM is transformed into a polytope model, and then transformed again into optimized LLVM IR.

[10] https://polly.llvm.org/

Polyhedral optimization has been used to automatically determine both decomposition (tiling) and the subsequent generation of GPU kernels[11,12]; however, much of this work is primarily concerned with conversion of serial CPU code into a GPU-compatible format.

During the project, we had hoped to leverage the dependency-preserving nature of our OpenCL kernel EDSL to allow for polyhedral analysis and optimization; however, the use of our EDSL involved explicit use of kernel indices (e.g. local thread ID), which essentially encapsulates much of the loop decomposition that can inferred via polyhedral optimization. The use of these indices effectively represents loops whose iterations have been separated for parallel execution. By removing this loop nesting, we are limiting the applicability of this type of optimization. Additionally, our kernels are already specifically designed to achieve good memory access patterns, thereby limiting another dimension of possible optimization.

It was concluded that while our kernel EDSL does indeed enable the application of polyhedral optimization techniques, the use of our EDSL is such that the effort of application would likely not justify the eventual performance improvement compared to more traditional approaches (e.g. automated search of parameter space).

As such, we are now planning to re-implement a new tuning system that seeks to intelligently search through the parameter space. This method is anticipated to carry out the following procedure:

1. Identify n-dimensional parameter space (e.g. kernel launch conditions, shared memory allocation, etc.)
2. Generate valid combinations that sparsely cover the said n-dimensional parameter space
3. Run the kernel under the launch conditions for each previously-generated combination and collect performance information
4. Identify one or more best-performing combinations, regenerate finer-grained parameter space around these values, and re-run performance
5. Iteratively perform this process for some number of passes or until improvement delta falls below a certain threshold

We now believe that this traditional approach, although requiring the user to run tuning passes on the code, is the most robust way to achieve the best performance. Additionally, these results will be shareable with others, and sane defaults are currently provided in our software.

## 3 Future Work and Transition

Upon completion of this project, EM Photonics intends to continue pursuing this technology with work in several ways. First, we will be making the core libraries available as one of the charter pieces of our Celerity Tools initiative. Second, we will be extending portions of the libraries themselves using new

[11] http://polly.llvm.org/documentation/gpgpucodegen.html

[12] Baskaran M.M., Ramanujam J., Sadayappan P. (2010) Automatic C-to-CUDA Code Generation for Affine Programs. In: Gupta R. (eds) Compiler Construction. CC 2010. Lecture Notes in Computer Science, vol 6011. Springer, Berlin, Heidelberg

hybrid and cross-platform development tools we are creating. And third, we will be incorporating the results of this work into other projects.

## 3.1 Celerity Tools

For more than 15 years, EM Photonics has developed novel applications using a variety of commodity processing hardware. Most of this work has been in the building specific software solutions by leveraging hybrid computing systems. More recently, we have also undertaken work in building tools to assist developers in doing the same for their own software. We have approached this in three ways: math libraries for hybrid systems, a cross-platform kernel development toolchain, and a hybrid task-scheduling framework (see Figure 10). While the latter two are still under development, the former was addressed in previous projects and broadened in this one. The result of this project is Affinimath, a foundational set of math libraries that provide a consistent feel across a variety of processing devices allowing users to take advantage of exotic hardware without the need to program it directly. Collectively, we are referring to these and potential subsequent offerings as Celerity Tools.

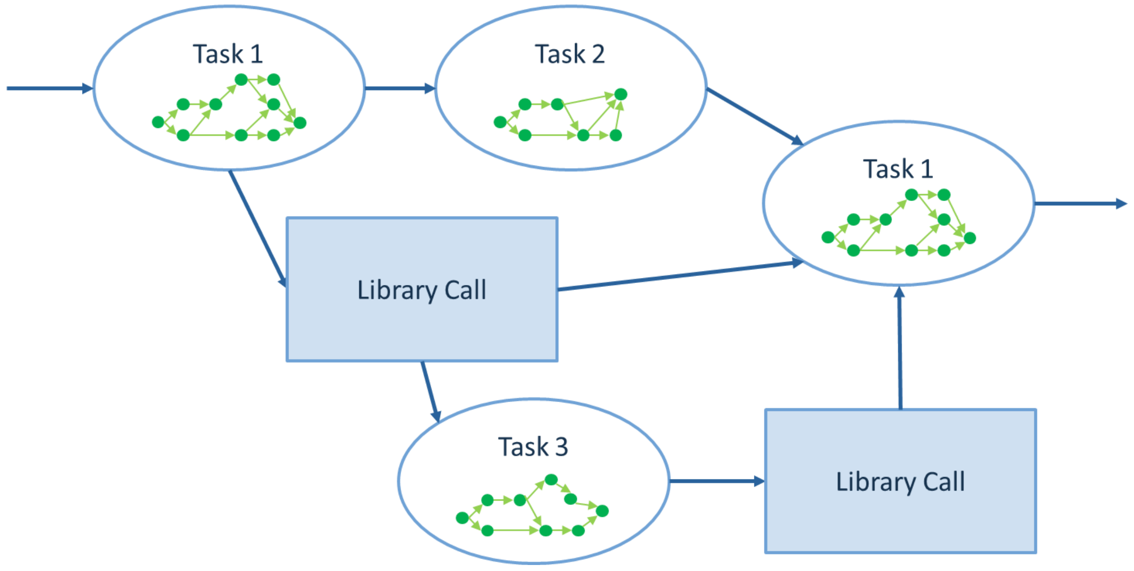
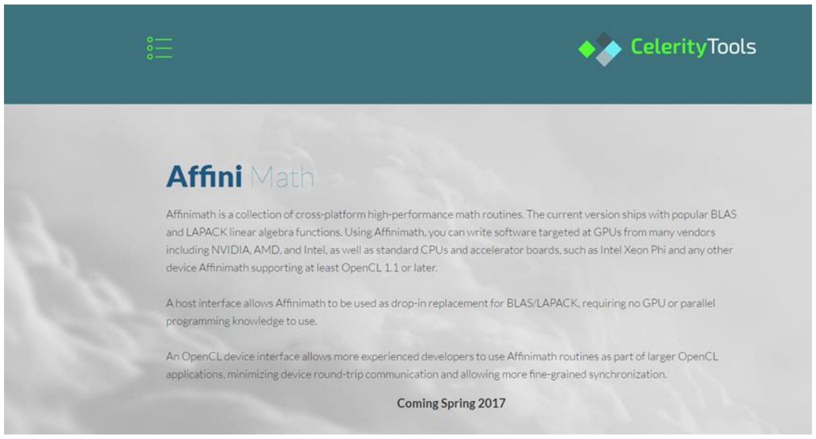


*Figure 10 – A notational program consisting of several subtasks that a developer would like run on various or mixed hardware devices. Standard math functionality can be replaced with the appropriate library call taken from the Affinimath package developed in this project. Other tasks can be written using our cross-platform kernel builder and all the tasks can collectively be managed with our hybrid task scheduler. The latter two tools are currently under development on other projects.*

The collection of Celerity Tools is meant to assist developers in building software for hybrid systems and/or varied deployment scenarios. Our goal is approaches for using mixed and varied hardware systems without the developer having to be intimately aware of each platform. The tools use various approaches for abstracting away the hardware's details. Affinimath, the libraries developed in this project are the first piece of our Celerity Tools initiative that will be released.

### 3.1.1 Celerity Tools Website

To distribute first Affinimath and subsequently other tools developed under our Celerity Tools initiative, EM Photonics is building a standalone website. From here, registered users will be able to download free software packages, purchase packages from our premium tier, initiate support requests, and access forum discussions (see Figure 11). We expect this site to launch in Spring of 2017.



Affini Math

Affinimath is a collection of cross-platform high-performance math routines. The current version ships with popular BLAS and LAPACK linear algebra functions. Using Affinimath, you can write software targeted at GPUs from many vendors including NVIDIA, AMD, and Intel, as well as standard CPUs and accelerator boards, such as Intel Xeon Phi and any other device Affinimath supporting at least OpenCL 1.1 or later.

A host interface allows Affinimath to be used as drop-in replacement for BLAS/LAPACK, requiring no GPU or parallel programming knowledge to use.

An OpenCL device interface allows more experienced developers to use Affinimath routines as part of larger OpenCL applications, minimizing device round-trip communication and allowing more fine-grained synchronization.

**Coming Spring 2017**

*Figure 11 – Celerity Tools website expected to launch Spring 2017.*

## 3.2 Extending Library Functionality

As mentioned above, EM Photonics is currently developing tools to assist in developing software to take advantage of the multitude of available processing devices. As these tools mature, they can be used to enhance the Affinimath libraries developed in this project. Specifically, our cross-platform kernel builder could be used to create optimized solvers for various devices of interest and our hybrid task scheduler could be used to scale them to larger systems. Incorporating these technologies will be a way to continue to progress the work of Affinimath outside this project.

## 3.3 Incorporating Affinimath into Other Projects

In addition to distributing and enhancing Affinimath, EM Photonics is also looking at applications of it in other projects. This includes both internal EM Photonics projects as well as those for our clients. For instance, we are adding a feature to our cross-platform kernel development toolchain that allows for swapping of user library calls with a cross-platform equivalent pulled from the Affinimath toolbox. In addition, we work with clients to develop optimized software, particularly as they are moving from CPU implementations to those targeted at more exotic hardware. We have observed the use of common math routines in fields ranging from scientific computing to biomedical to finance. Sometimes the initial codebase will have explicit library calls in it and in others the developer will have written their own version of a common routine (either knowingly or unknowingly). In these cases, we can use Affinimath to augment the custom development we are providing.

## 4 CONCLUSION

In the initial release, our library includes real BLAS functions implemented along with real Cholesky, QR, LU, LU solve, and inverse functions form LAPACK. The library also includes a FFT, Sparse MV, and complex GEMM functions. Our library presently targets CPUs, NVIDIA GPUs, AMD GPUs and Intel GPUs utilizing OpenCL to execute our algorithms.

We found in our benchmarks that certain functions of our library do not perform ideally, and require additional optimization; increasing our performance, especially on GPUs, is a priority tasks going forward. We also plan to increase the number of languages backends our library can utilize (such as CUDA). Our EDSL code will prove useful in that matter.

In future releases, we will increase the number of functions that our library provides; adding the complex flavors of our current algorithms, and additional LAPACK, FFT, and Sparse algorithms.