VexCL — a Vector Expression Template Library for OpenCL

Denis Demidov

Supercomputer Center of Russian Academy of Sciences
Kazan Federal University

November 2013, NSCF

Modern GPGPU frameworks

CUDA

- Proprietary architecture by NVIDIA
- Requires NVIDIA hardware
- More mature, many libraries

OpenCL

- Open standard
- Supports wide range of hardware
- Code is much more verbose

Modern GPGPU frameworks

CUDA

- Proprietary architecture by NVIDIA
- Requires NVIDIA hardware
- More mature, many libraries

- *Kernels are compiled to PTX together with host program*

OpenCL

- Open standard
- Supports wide range of hardware
- Code is much more verbose

- *Kernels are compiled at runtime, adding an initialization overhead*

- The latter distinction is usually considered to be an OpenCL drawback.
- But it also allows us to generate more efficient kernels at runtime!
  - VexCL takes care of this part.

VexCL — a vector expression template library for OpenCL

Fork me on GitHub

- https://github.com/ddemidov/vexcl
- Created for ease of C++ based OpenCL development.
  - Convenient notation for vector expressions.
  - OpenCL JIT code generation.
- The source code is publicly available under MIT license.
- *This is not a* C++ *bindings library!*
  - VexCL works on top of Khronos C++ bindings for OpenCL.

Hello OpenCL: vector sum

Vector sum

- A, B, and C are large vectors.
- Compute $C = A + B$.

Overview of OpenCL solution

1. Initialize OpenCL context on supported device.
2. Allocate memory on the device.
3. Transfer input data to device.
4. Run your computations on the device.
5. Get the results from the device.

Hello OpenCL: vector sum

1. Query platforms

```
std::vector<cl::Platform> platform;
cl::Platform::get(&platform);

if ( platform.empty() )
    throw std::runtime_error("OpenCL platforms not found.");
```

Hello OpenCL: vector sum

2. Get first available GPU device

```
cl::Context context;
std::vector<cl::Device> device;
for(auto p = platform.begin(); device.empty() && p != platform.end(); p++) {
    std::vector<cl::Device> dev;
    try {
        p->getDevices(CL_DEVICE_TYPE_GPU, &dev);
        for(auto d = dev.begin(); device.empty() && d != dev.end(); d++) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            device.push_back(*d);
            context = cl::Context(device);
        }
    } catch(...) {
        device.clear();
    }
}
if (device.empty()) throw std::runtime_error("GPUs not found");
```

Hello OpenCL: vector sum

3. Create kernel source

```
const char source[] =
    "kernel void add(\n"
    "        uint n,\n"
    "        global const float *a,\n"
    "        global const float *b,\n"
    "        global float *c\n"
    "        )\n"
    "{\n"
    "    uint i = get_global_id(0);\n"
    "    if (i < n) {\n"
    "        c[i] = a[i] + b[i];\n"
    "    }\n"
    "}\n";
```

Hello OpenCL: vector sum

4. Compile kernel

```
cl::Program program(context, cl::Program::Sources(
            1, std::make_pair(source, strlen(source))
            ));
try {
    program.build(device);
} catch (const cl::Error&) {
    std::cerr
        << "OpenCL compilation error" << std::endl
        << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device[0])
        << std::endl;
    return 1;
}
cl::Kernel add_kernel = cl::Kernel(program, "add");
```

5. Create command queue

```
cl::CommandQueue queue(context, device[0]);
```

Hello OpenCL: vector sum

6. Prepare input data, transfer it to device

```
const size_t N = 1 << 20;
std::vector<float> a(N, 1), b(N, 2), c(N);

cl::Buffer A(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        a.size() * sizeof(float), a.data());

cl::Buffer B(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        b.size() * sizeof(float), b.data());

cl::Buffer C(context, CL_MEM_READ_WRITE,
        c.size() * sizeof(float));
```

Hello OpenCL: vector sum

7. Set kernel arguments

```
add_kernel.setArg(0, N);
add_kernel.setArg(1, A);
add_kernel.setArg(2, B);
add_kernel.setArg(3, C);
```

8. Launch kernel

```
queue.enqueueNDRangeKernel(add_kernel, cl::NullRange, N, cl::NullRange);
```

9. Get result back to host

```
queue.enqueueReadBuffer(C, CL_TRUE, 0, c.size() * sizeof(float), c.data());
std::cout << c[42] << std::endl; // Should get '3' here.
```

Hello VexCL: vector sum

Much shorter!

```
std :: cout << 3 << std::endl;
```

Hello VexCL: vector sum

Get all available GPUs:

```
vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );
if ( !ctx ) throw std::runtime_error("GPUs not found");
```

Prepare input data, transfer it to device:

```
std::vector<float> a(N, 1), b(N, 2), c(N);
vex::vector<float> A(ctx, a);
vex::vector<float> B(ctx, b);
vex::vector<float> C(ctx, N);
```

Launch kernel, get result back to host:

```
C = A + B;
vex::copy(C, c);
std::cout << c[42] << std::endl;
```

Initialization

- Multi-device and multi-platform computations are supported.
- VexCL context is initialized from combination of device filters.
- Device filter is a boolean functor acting on **const** cl::Device&.

Initialize VexCL context on selected devices

```
vex::Context ctx( vex::Filter::All );
```


Initialization

- Multi-device and multi-platform computations are supported.
- VexCL context is initialized from combination of device filters.
- Device filter is a boolean functor acting on **const** cl::Device&.

Initialize VexCL context on selected devices

```
vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );
```

Initialization

- Multi-device and multi-platform computations are supported.
- VexCL context is initialized from combination of device filters.
- Device filter is a boolean functor acting on **const** cl::Device&.

Initialize VexCL context on selected devices

```
vex::Context ctx(
    vex::Filter::Type(CL_DEVICE_TYPE_GPU) &&
    vex::Filter::Platform("AMD")
    );
```

Initialization

- Multi-device and multi-platform computations are supported.
- VexCL context is initialized from combination of device filters.
- Device filter is a boolean functor acting on `const cl::Device&`.

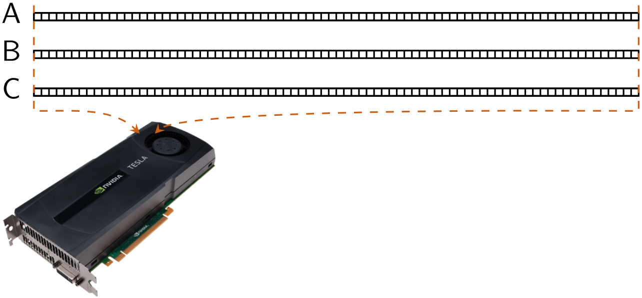
Initialize VexCL context on selected devices

```
vex::Context ctx(
    vex::Filter::Type(CL_DEVICE_TYPE_GPU) &&
    [](const cl::Device &d) {
        return d.getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() >= 4_GB;
    });
```

Memory and work splitting

Hello VexCL example

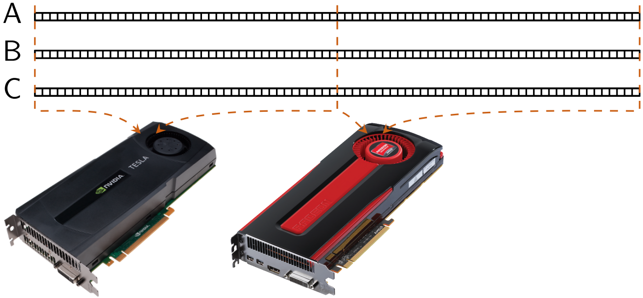
```
vex::Context ctx( vex::Filter::Name("Tesla") );

vex::vector<float> A(ctx, N); A = 1;
vex::vector<float> B(ctx, N); B = 2;
vex::vector<float> C(ctx, N);

C = A + B;
```



Memory and work splitting

Hello VexCL example

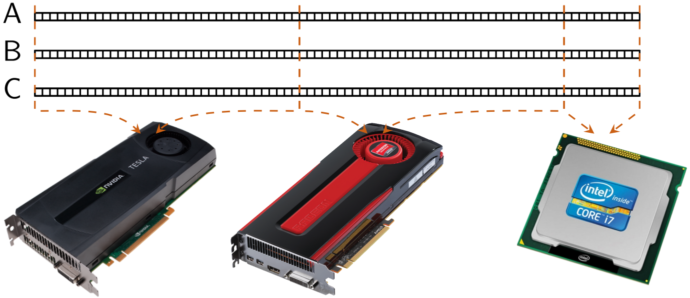
```
vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );

vex::vector<float> A(ctx, N); A = 1;
vex::vector<float> B(ctx, N); B = 2;
vex::vector<float> C(ctx, N);

C = A + B;
```



Memory and work splitting

Hello VexCL example

```
vex::Context ctx( vex::Filter::DoublePrecision );

vex::vector<float> A(ctx, N); A = 1;
vex::vector<float> B(ctx, N); B = 2;
vex::vector<float> C(ctx, N);

C = A + B;
```



Copies between host and device memory

```
vex::vector<double> X;
std::vector<double> x;
double c_array[100];
```

Simple copies

```
vex::copy(X, x); // From device to host.
vex::copy(x, X); // From host to device.
```

STL-like range copies

```
vex::copy(X.begin(), X.end(), x.begin());
vex::copy(X.begin(), X.begin() + 100, x.begin());
vex::copy(c_array, c_array + 100, X.begin());
```

Inspect or set single element (*slow*)

```
assert(x[42] == X[42]);
X[0] = 0;
```

What vector expressions are supported?

- All vectors in an expression have to be *compatible*:
  - Have same size
  - Located on same devices
- What may be used:
  - Vectors and scalars
  - Arithmetic, logical operators
  - Built-in OpenCL functions
  - User-defined functions
  - Random number generators
  - Slicing and permutations
  - Reduction (sum, min, max)
  - Stencil operations
  - Sparse matrix – vector products
  - Fast Fourier Transform

```
vex::vector<double> x(ctx, n), y(ctx, n);

x = (2 * M_PI / n) * vex::element_index();
y = pow(sin(x), 2.0) + pow(cos(x), 2.0);
```

Builtin operations and functions

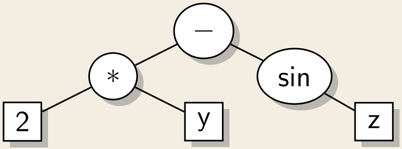
This expression:

```
x = 2 * y - sin(z);
```

- Define VEXCL_SHOW_KERNELS to see the generated code.

...results in this kernel:

```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * res,
    int prm1,
    global double * prm2,
    global double * prm3
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        res[idx] = ( ( prm1 * prm2[idx] ) - sin( prm3[idx] ) );
    }
}
```



Element indices

- vex::element_index(size_t offset = 0) returns index of an element inside a vector.
  - The numbering starts with offset and is continuous across devices.

Linear function:

```
vex::vector<double> X(ctx, N);
double x0 = 0, dx = 1e-3;
X = x0 + dx * vex::element_index();
```

Single period of sine function:

```
X = sin(2 * M_PI * vex::element_index() / N);
```

User-defined functions

- Users may define functions to be used in vector expressions.
- There are two options for doing this:
  - Provide function body in a string.
  - Provide generic C++ functor.
- Once defined, user functions are used in the same way as builtin functions.

1. Provide function body in a string

- Choose function name
- Specify function signature
- Provide function body

Defining a function:

```
VEX_FUNCTION( sqr, double(double, double), "return prm1 * prm1 + prm2 * prm2;" );
```

Using a function:

```
Z = sqrt( sqr(X, Y) );
```

2. Provide generic functor

Functor definition:

```
struct sqr_functor {
    template <class T>
    T operator(const T &x, const T &y) const {
        return x * x + y * y;
    }
};
```

2. Provide generic functor

Functor definition:

```
struct sqr_functor {
    template <class T>
    T operator(const T &x, const T &y) const {
        return x * x + y * y;
    }
};
```

Generate VexCL function:

```
using vex::generator::make_function;
auto sqr = make_function<double(double,double)>( sqr_functor() );
```

2. Provide generic functor

Functor definition:

```
struct sqr_functor {
    template <class T>
    T operator(const T &x, const T &y) const {
        return x * x + y * y;
    }
};
```

Generate VexCL function:

```
using vex::generator::make_function;
auto sqr = make_function<double(double,double)>( sqr_functor() );
```

Boost.Phoenix lambdas *are* generic functors:

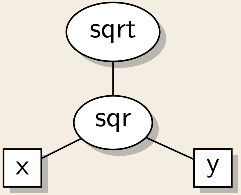
```
using namespace boost::phoenix::arg_names;
auto sqr = make_function<double(double,double)>( arg1 * arg1 + arg2 * arg2 );
```

User functions are translated to OpenCL functions

```
Z = sqrt( sqr(X, Y) );
```

...gets translated to:

```
double func1(double prm1, double prm2) {
    return prm1 * prm1 + prm2 * prm2;
}

kernel void vexcl_vector_kernel(
    ulong n,
    global double * res,
    global double * prm1,
    global double * prm2
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        res[idx] = sqrt( func1( prm1[idx], prm2[idx] ) );
    }
}
```



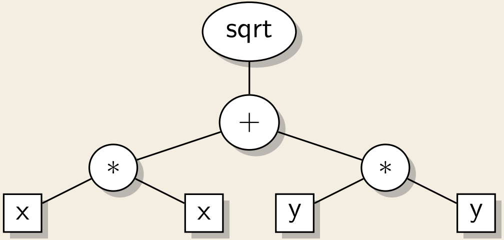
Functions may be not only convenient, but also effective

Same example without using a function:

```
Z = sqrt( X * X + Y * Y );
```

. . . gets translated to:

```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * res,
    global double * prm1,
    global double * prm2,
    global double * prm3,
    global double * prm4
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        res [idx] = sqrt( ( ( prm1[idx] * prm2[idx] ) + ( prm3[idx] * prm4[idx] ) ) );
    }
}
```



Tagged terminals

- Programmer may help VexCL to recognize same terminals by tagging them:

Like this:

```
using vex::tag;
Z = sqrt(tag<1>(X) * tag<1>(X) +
         tag<2>(Y) * tag<2>(Y));
```

or, equivalently:

```
auto x = tag<1>(X);
auto y = tag<2>(Y);
Z = sqrt(x * x + y * y);
```

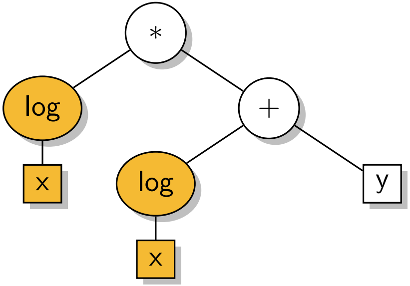
```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * res,
    global double * prm1,
    global double * prm2
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        res[idx] = sqrt( ( ( prm1[idx] * prm1[idx] ) + ( prm2[idx] * prm2[idx] ) ) );
    }
}
```

Reusing intermediate results

- Some expressions may have several inclusions of the same subexpression:

```
Z = log(X) * (log(X) + Y);
```

- $\log(X)$ will be computed twice here.
- One could tag X and hope that OpenCL compiler is smart enough…



Temporaries

- But it is also possible to introduce a temporary variable explicitly:

```
auto tmp = vex::make_temp<1>( log(X) );
Z = tmp * (tmp + Y);
```

```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * res,
    global double * prm1,
    global double * prm2
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        double temp1 = log( prm1[idx] );
        res[idx] = ( temp1 * ( temp1 + prm2[idx] ) );
    }
}
```

Slicing *(Single-device contexts)*

- When working with dense multidimensional matrices, it is general practice to store those in continuous arrays.
  - An instance of vex::slicer<NDIM> class allows to access sub-blocks of such matrix.

n-by-n matrix and a slicer:

```
vex::vector<double> x(ctx, n * n);
vex::slicer<2> slice(vex::extents[n][n]); // Can be used with any vector of appropriate size
```

Slicing *(Single-device contexts)*

- When working with dense multidimensional matrices, it is general practice to store those in continuous arrays.
  - An instance of vex::slicer<NDIM> class allows to access sub-blocks of such matrix.

n-by-n matrix and a slicer:

```
vex::vector<double> x(ctx, n * n);
vex::slicer<2> slice(vex::extents[n][n]); // Can be used with any vector of appropriate size
```

Access row or column of the matrix:

```
using vex::_;
y = slice[42](x);        // 42nd row
y = slice[_][42](x);     // 42nd column
slice[_][10](x) = y;     // Slices are writable
```

Slicing (*Single-device contexts*)

- When working with dense multidimensional matrices, it is general practice to store those in continuous arrays.
  - An instance of vex::slicer<NDIM> class allows to access sub-blocks of such matrix.

n-by-n matrix and a slicer:

```
vex::vector<double> x(ctx, n * n);
vex::slicer<2> slice(vex::extents[n][n]); // Can be used with any vector of appropriate size
```

Access row or column of the matrix:

```
using vex::_;
y = slice[42](x);       // 42nd row
y = slice[_][42](x);    // 42nd column
slice[_][10](x) = y;    // Slices are writable
```

Use ranges to select sub-blocks:

```
using vex::range;
z = slice[range(0, 2, n)][range(10, 20)](x);
```

Permutations (*Single-device contexts*)

- vex::permutation() function takes arbitrary (integral valued) vector expression and returns permutation functor:

Permutations (*Single-device contexts*)

- `vex::permutation()` function takes arbitrary (integral valued) vector expression and returns permutation functor:

Index-based permutation:

```
vex::vector<size_t> I(ctx, N);
I = N - 1 - vex::element_index();
auto reverse = vex::permutation(I);
y = reverse(x);
```

Permutations (*Single-device contexts*)

- `vex::permutation()` function takes arbitrary (integral valued) vector expression and returns permutation functor:

Index-based permutation:

```
vex::vector<size_t> I(ctx, N);
I = N - 1 - vex::element_index();
auto reverse = vex::permutation(I);
y = reverse(x);
```

Expression-based permutation:

```
auto reverse = vex::permutation(N - 1 - vex::element_index());
y = reverse(x);
```

Permutations (*Single-device contexts*)

- vex::permutation() function takes arbitrary (integral valued) vector expression and returns permutation functor:

Index-based permutation:

```
vex::vector<size_t> I(ctx, N);
I = N - 1 - vex::element_index();
auto reverse = vex::permutation(I);
y = reverse(x);
```

Expression-based permutation:

```
auto reverse = vex::permutation(N - 1 - vex::element_index());
y = reverse(x);
```

Permutations are writable:

```
reverse(y) = x;
```

Random number generation

- VexCL provides implementation[1] of *counter-based* random number generators from Random123[2] suite.
  - The generators are *stateless*; mixing functions are applied to element indices.
  - Implemented families: Threefry and Philox.
- vex::Random<T,G> — uniform distribution.
- vex::RandomNormal<T,G> — normal distribution.

Monte Carlo π:

```cpp
vex::Random<double, vex::random::threefry> rnd;

x = 2 * rnd(vex::element_index(), std::rand()) - 1;
y = 2 * rnd(vex::element_index(), std::rand()) - 1;

vex::Reductor<size_t, vex::SUM> sum(ctx);
double pi = 4.0 * sum( (x * x + y * y) < 1 ) / n;
```

[1] Contributed by Pascal Germroth ⟨pascal@ensieve.org⟩

[2] D E Shaw Research, http://www.deshawresearch.com/resources_random123.html

Reductions

- Class `vex::Reductor<T, kind>` allows to reduce arbitrary *vector expression* to a single value of type `T`.
- Supported reduction kinds: SUM, MIN, MAX

Inner product

```
vex::Reductor<double, vex::SUM> sum(ctx);
double s = sum(x * y);
```

Number of elements in x between 0 and 1

```
vex::Reductor<size_t, vex::SUM> sum(ctx);
size_t n = sum( (x > 0) && (x < 1) );
```

Maximum distance from origin

```
vex::Reductor<double, vex::MAX> max(ctx);
double d = max( sqrt(x * x + y * y) );
```

Sparse matrix – vector products *(Additive expressions only)*

- Class vex::SpMat<T> holds representation of a sparse matrix on compute devices.
- Constructor accepts matrix in common CRS format:
  - row indices, columns and values of nonzero entries.

Construct matrix

```cpp
vex::SpMat<double> A(ctx, n, n, row.data(), col.data(), val.data());
```

Compute residual value

```cpp
// vex::vector<double> u, f, r;
r = f - A * u;
double res = max( fabs(r) );
```

Inlining sparse matrix – vector products *(Single-device contexts)*

- SpMV may only be used in additive expressions:
  - Needs data exchange between compute devices.
  - Impossible to implement with single kernel.
- This restriction may be lifted for single-device contexts:

```
r = f - vex::make_inline(A * u);
double res = max( fabs(r) );
```

Inlining sparse matrix – vector products (*Single-device contexts*)

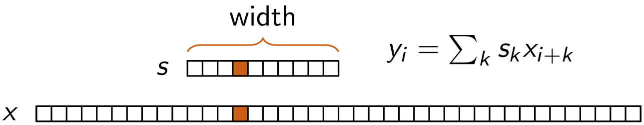
- SpMV may only be used in additive expressions:
  - Needs data exchange between compute devices.
  - Impossible to implement with single kernel.
- This restriction may be lifted for single-device contexts:

```cpp
r  = f - vex::make_inline(A * u);
double res = max( fabs(r) );
```

Do not store intermediate results:

```cpp
double res = max( fabs( f - vex::make_inline(A * u) ) );
```

Simple stencil convolutions *(Additive expressions only)*



$$y_i = \sum_k s_k x_{i+k}$$

- Simple stencil is based on a 1D array, and may be used for:
  - Signal filters (e.g. averaging)
  - Differential operators with constant coefficients
  - ...

Moving average with 5-points window

```
std::vector<double> sdata(5, 0.2);
vex::stencil<double> s(ctx, sdata, 2 /* center */);

y = x * s;
```

User-defined stencil operators (*Additive expressions only*)

- Define efficient arbitrary stencil operators:
  - Return type
  - Stencil dimensions (width and center)
  - Function body
  - Queue list

Example: nonlinear operator

$$y_i = x_i + (x_{i-1} + x_{i+1})^3$$

Implementation

```
VEX_STENCIL_OPERATOR(custom_op, double, 3/*width*/, 1/*center*/,
    "double t = X[-1] + X[1];\n"
    "return X[0] + t * t * t;",
    ctx);

y = custom_op(x);
```

Fast Fourier Transform (*Single-device contexts*)

- VexCL provides FFT implementation[3]:
  - Currently only single-device contexts are supported
  - Arbitrary vector expressions as input
  - Multidimensional transforms
  - Arbitrary sizes

Solve Poisson equation with FFT:

```
vex::FFT<double, cl_double2> fft(ctx, n);
vex::FFT<cl_double2, double> ifft(ctx, n, vex::inverse);

vex::vector<double> rhs(ctx, n), u(ctx, n), K(ctx, n);
// ... initialize vectors ...

u = ifft( K * fft(rhs) );
```

[3] Contributed by Pascal Germroth ⟨pascal@ensieve.org⟩

Multivectors

- vex::multivector<T,N> holds N instances of equally sized vex::vector<T>
- Supports all operations that are defined for vex::vector<>.
- Transparently dispatches the operations to the underlying components.
- vex::multivector::operator()(size_t k) returns k-th component.

```
vex::multivector<double, 2> X(ctx, N), Y(ctx, N);
vex::Reductor<double, vex::SUM> sum(ctx);
vex::SpMat<double> A(ctx, ... );
std::array<double, 2> v;

// ...

X = sin(v * Y + 1);              // X(k) = sin(v[k] * Y(k) + 1);
v = sum( between(0, X, Y) );     // v[k] = sum( between( 0, X(k), Y(k) ) );
X = A * Y;                       // X(k) = A * Y(k);
```

Multiexpressions

- Sometimes an operation cannot be expressed with simple multivector arithmetics.

Example: rotate 2D vector by an angle

$$
\begin{aligned}
y_0 &= x_0 \cos\alpha - x_1 \sin\alpha, \\
y_1 &= x_0 \sin\alpha + x_1 \cos\alpha.
\end{aligned}
$$

- Multiexpression is a tuple of normal vector expressions
- Its assignment to a multivector is functionally equivalent to component-wise assignment, but results in a single kernel launch.

Multiexpressions

- Multiexpressions may be used with multivectors:

```
// double alpha;
// vex::multivector<double,2> X, Y;

Y = std::tie( X(0) * cos(alpha) - X(1) * sin(alpha),
              X(0) * sin(alpha) + X(1) * cos(alpha)  );
```

- and with tied vectors:

```
// vex::vector<double> alpha;
// vex::vector<double> odlX, oldY, newX, newY;

vex::tie(newX, newY) = std::tie( oldX * cos(alpha) - oldY * sin(alpha),
                                 oldX * sin(alpha) + oldY * cos(alpha)  );
```

A multiexpression results in a single kernel

```
auto x0 = tag<0>( X(0) );
auto x1 = tag<1>( X(1) );
auto ca = tag<2>( cos(alpha) );
auto sa = tag<3>( sin(alpha) );

Y = std::tie(x0 * ca - x1 * sa, x0 * sa + x1 * ca);
```

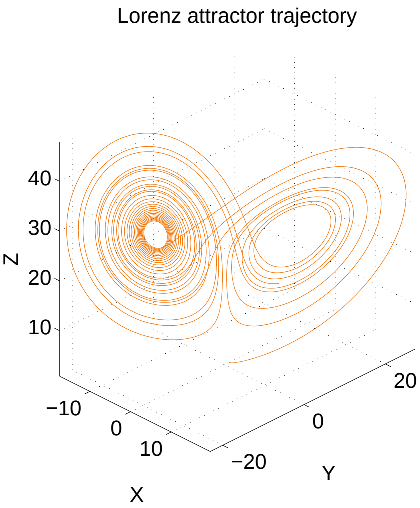
```
kernel void vexcl_multivector_kernel(ulong n,
    global double * res1, global double * res2,
    global double * prm1, double prm2,
    global double * prm3, double prm4
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        double buf1 = ( ( prm1[idx] * prm2 ) - ( prm3[idx] * prm4 ) );
        double buf2 = ( ( prm1[idx] * prm4 ) + ( prm3[idx] * prm2 ) );

        res1[idx] = buf1;
        res2[idx] = buf2;
    }
}
```

Parameter study for the Lorenz attractor system

Lorenz attractor system

$$
\begin{aligned}
\dot{x} &= -\sigma \left(x - y \right), \\
\dot{y} &= Rx - y - xz, \\
\dot{z} &= -bz + xy.
\end{aligned}
$$

- Let's solve large number of Lorenz systems, each for a different value of R.
- Let's use VexCL and Boost.odeint for that.



Using Boost.odeint

ODE in general:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \dot{x} = f(x, t), \qquad x(0) = x_0.$$

Using Boost.odeint:

1. Define state type (what is x?)
2. Provide system function (define f)
3. Choose integration method
4. Integrate over time

Naive implementation

1. State type

```cpp
typedef vex::multivector<double, 3> state_type;
```

2. System functor

```cpp
struct lorenz_system {
    const vex::vector<double> &R;
    lorenz_system(const vex::vector<double> &R ) : R(R) { }

    void operator()(const state_type &x, state_type &dxdt, double t) {
        dxdt = std::tie( sigma * ( x(1) - x(0) ),
                         R * x(0) - x(1) - x(0) * x(2),
                         x(0) * x(1) - b * x(2)   );
    }
};
```

Naive implementation

3. Stepper (4th order Runge-Kutta)

```
odeint::runge_kutta4<
            state_type /*state*/,      double /*value*/,
            state_type /*derivative*/, double /*time*/,
            odeint::vector_space_algebra, odeint::default_operations
            > stepper;
```

4. Integration

```
vex::multivector<double,3> X(ctx, n);
vex::vector<double> R(ctx, n);

X = 10;
R = Rmin + vex::element_index() * ((Rmax - Rmin) / (n - 1));

odeint::integrate_const(stepper, lorenz_system(R), X, 0.0, t_max, dt);
```

CUBLAS implementation

- CUBLAS is a highly optimized BLAS implementation from NVIDIA.
- Its disadvantage is that it has fixed number of kernels/functions.
- Hence, linear combinations (used internally by odeint):

$$x_0 = \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n$$

  are implemented as:

```
cublasDcopy(...);    // x_0 = x_1
cublasDscal (...);   // x_0 = α_1 * x_0
cublasDaxpy(...);    // x_0 = x_0 + α_2 * x_2
...
cublasDaxpy(...);    // x_0 = x_0 + α_n * x_n
```

Thrust implementation

- It is possible to fuse linear combination kernels with Thrust:

Thrust

```
struct scale_sum2 {
    const double a1, a2;
    scale_sum2(double a1, double a2) : a1(a1), a2(a2) { }
    template<class Tuple>
    __host__ __device__ void operator()(Tuple t) const {
        thrust::get<0>(t) = a1 * thrust::get<1>(t) + a2 * thrust::get<2>(t);
    }
};

thrust::for_each(
        thrust::make_zip_iterator(
            thrust::make_tuple( x0.begin(), x1.begin(), x2.begin() )
            ),
        thrust::make_zip_iterator(
            thrust::make_tuple( x0.end(), x1.end(), x2.end() )
            ),
        scale_sum2(a1, a2)
        );
```

Thrust implementation

- It is possible to fuse linear combination kernels with Thrust:

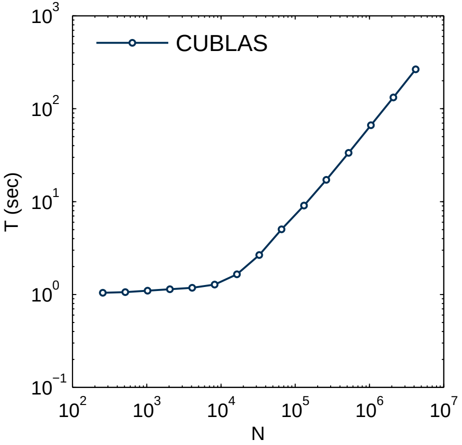
Thrust

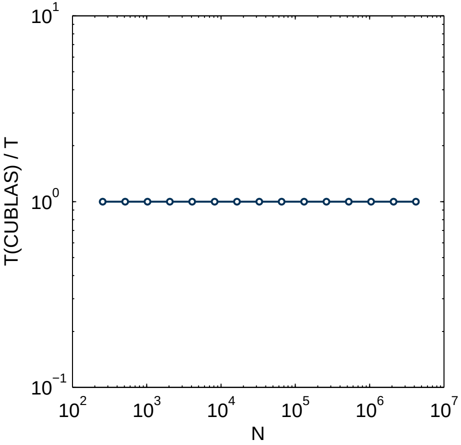
```
struct scale_sum2 {
    const double a1, a2;
    scale_sum2(double a1, double a2) : a1(a1), a2(a2) { }
    template<class Tuple>
    __host__ __device__ void operator()(Tuple t) const {
        thrust::get<0>(t) = a1 * thrust::get<1>(t) + a2 * thrust::get<2>(t);
    }
};

thrust::for_each(
        thrust::make_zip_iterator(
            thrust::make_tuple( x0.begin(), x1.begin(), x2.begin() )
            ),
        thrust::make_zip_iterator(
            thrust::make_tuple( x0.end(), x1.end(), x2.end() )
            ),
        scale_sum2(a1, a2)
        );
```

VexCL

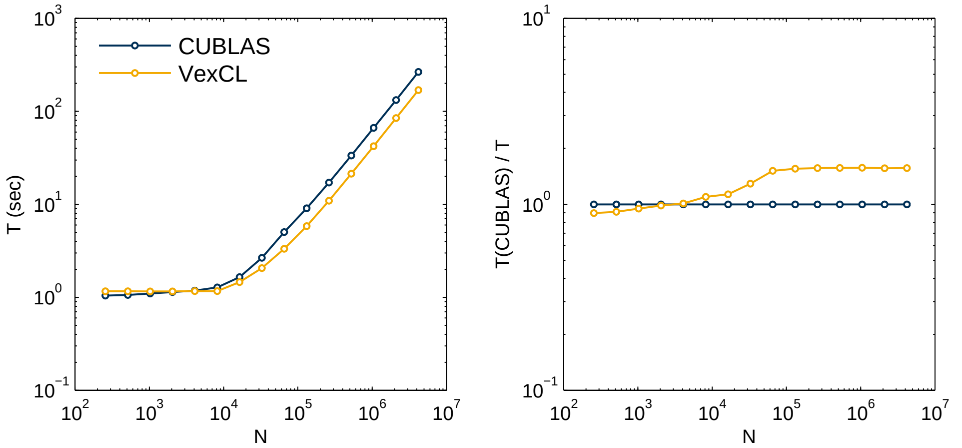
```
x0 = a1 * x1 + a2 * x2;
```

Performance (Tesla K20c)

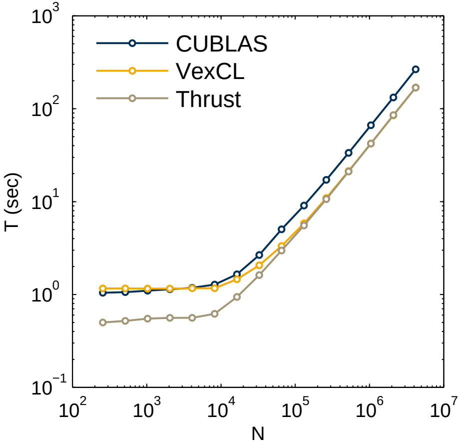


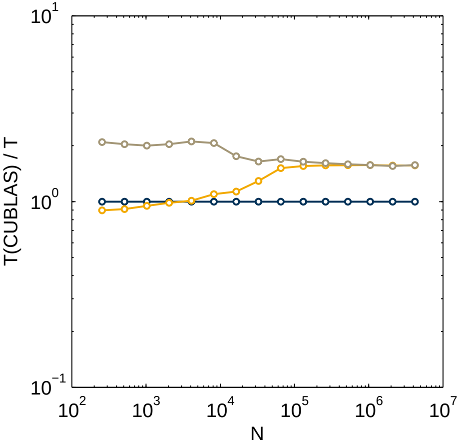


Performance (Tesla K20c)

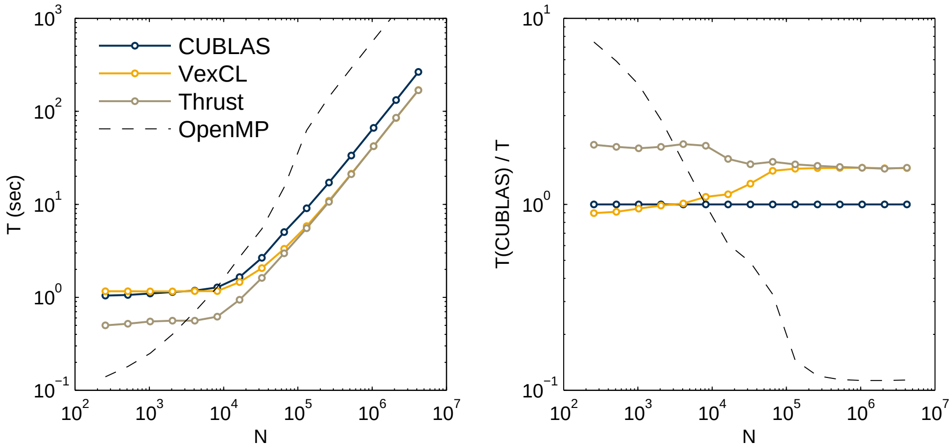


Performance (Tesla K20c)

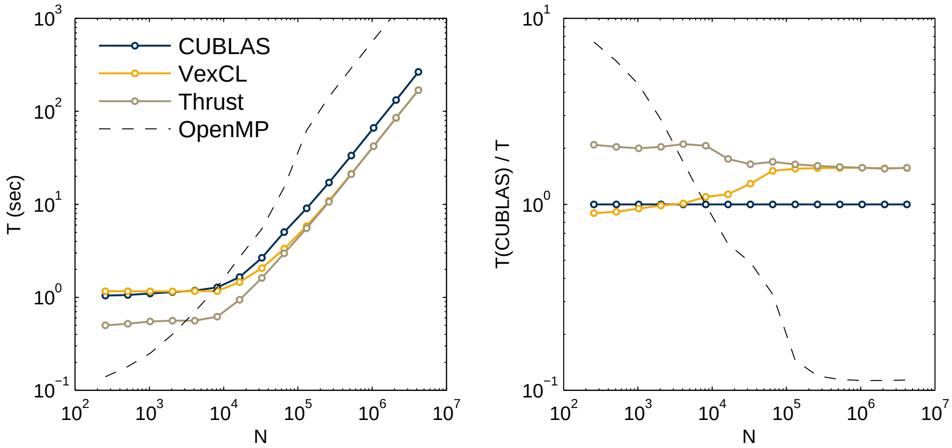




Performance (Tesla K20c)



Performance (Tesla K20c)



- Deficiencies of naive implementation:
  - Runge-Kutta method uses 4 temporary state variables (here stored on GPU).
  - Single Runge-Kutta step results in several kernel launches.

What if we did this manually?

- Create monolithic kernel for a single step of Runge-Kutta method.
- Launch the kernel in a loop.

- This would be 10x faster!

```
double3 lorenz_system(double r, double sigma, double b, double3 s) {
    return (double3)( sigma * (s.y - s.x),
                      r * s.x - s.y - s.x * s.z,
                      s.x * s.y - b * s.z);
}
kernel void lorenz_ensemble(
    ulong n, double dt, double sigma, double b,
    const global double *R,
    global double *X,
    global double *Y,
    global double *Z
    )
{
    for(size_t i = get_global_id(0); i < n; i += get_global_size(0)) {
        double r = R[i];
        double3 s = (double3)(X[i], Y[i], Z[i]);
        double3 k1, k2, k3, k4;

        k1 = dt * lorenz_system(r, sigma, b, s);
        k2 = dt * lorenz_system(r, sigma, b, s + 0.5 * k1);
        k3 = dt * lorenz_system(r, sigma, b, s + 0.5 * k2);
        k4 = dt * lorenz_system(r, sigma, b, s + k3);

        s += (k1 + 2 * k2 + 2 * k3 + k4) / 6;

        X[i] = s.x; Y[i] = s.y; Z[i] = s.z;
    }
}
```

What if we did this manually?

- Create monolithic kernel for a single step of Runge-Kutta method.
- Launch the kernel in a loop.

- This would be 10x faster! But,
  - We lost odeint's generality.

```
double3 lorenz_system(double r, double sigma, double b, double3 s) {
    return (double3)( sigma * (s.y - s.x),
                      r * s.x - s.y - s.x * s.z,
                      s.x * s.y - b * s.z);
}
kernel void lorenz_ensemble(
    ulong n, double dt, double sigma, double b,
    const global double *R,
    global double *X,
    global double *Y,
    global double *Z
    )
{
    for(size_t i = get_global_id(0); i < n; i += get_global_size(0)) {
        double r = R[i];
        double3 s = (double3)(X[i], Y[i], Z[i]);
        double3 k1, k2, k3, k4;

        k1 = dt * lorenz_system(r, sigma, b, s);
        k2 = dt * lorenz_system(r, sigma, b, s + 0.5 * k1);
        k3 = dt * lorenz_system(r, sigma, b, s + 0.5 * k2);
        k4 = dt * lorenz_system(r, sigma, b, s + k3);

        s += (k1 + 2 * k2 + 2 * k3 + k4) / 6;

        X[i] = s.x; Y[i] = s.y; Z[i] = s.z;
    }
}
```

Convert Boost.odeint stepper to a fused OpenCL kernel!

- VexCL provides vex::symbolic<T> type.
- An instance of the type dumps any arithmetic operations to output stream:

```
vex::symbolic<double> x = 6, y = 7;
x = sin(x * y);
```

```
double var1 = 6;
double var2 = 7;
var1 = sin( ( var1 * var2 ) );
```

Convert Boost.odeint stepper to a fused OpenCL kernel!

- VexCL provides vex::symbolic<T> type.
- An instance of the type dumps any arithmetic operations to output stream:

```cpp
vex::symbolic<double> x = 6, y = 7;
x = sin(x * y);
```

```
double var1 = 6;
double var2 = 7;
var1 = sin( ( var1 * var2 ) );
```

- The idea is very simple:
  - Record sequence of arithmetic expressions of an algorithm.
  - Generate OpenCL kernel from the captured sequence.

Record operations performed by Boost.odeint stepper

1. State type

```
typedef vex::symbolic< double > sym_vector;
typedef std::array<sym_vector, 3> sym_state;
```

2. System functor

```
struct lorenz_system {
    const sym_vector &R;
    lorenz_system(const sym_vector &R) : R(R) {}

    void operator()(const sym_state &x, sym_state &dxdt, double t) const {
        dxdt[0] = sigma * (x[1] - x[0]);
        dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
        dxdt[2] = x[0] * x[1] - b * x[2];
    }
};
```

Record operations performed by Boost.odeint stepper

3. Stepper

```
odeint::runge_kutta4<
        sym_state /*state*/,        double /*value*/,
        sym_state /*derivative*/, double /*time*/,
        odeint::range_algebra, odeint::default_operations
        > stepper;
```

4. Record one step of Runge-Kutta method

```
std::ostringstream lorenz_body;
vex::generator::set_recorder(lorenz_body);

sym_state sym_S = {{ sym_vector(sym_vector::VectorParameter),
                     sym_vector(sym_vector::VectorParameter),
                     sym_vector(sym_vector::VectorParameter) }};
sym_vector sym_R(sym_vector::VectorParameter, sym_vector::Const);

lorenz_system sys(sym_R);
stepper.do_step(std::ref(sys), sym_S, 0, dt);
```

Generate OpenCL kernel with the recorded sequence

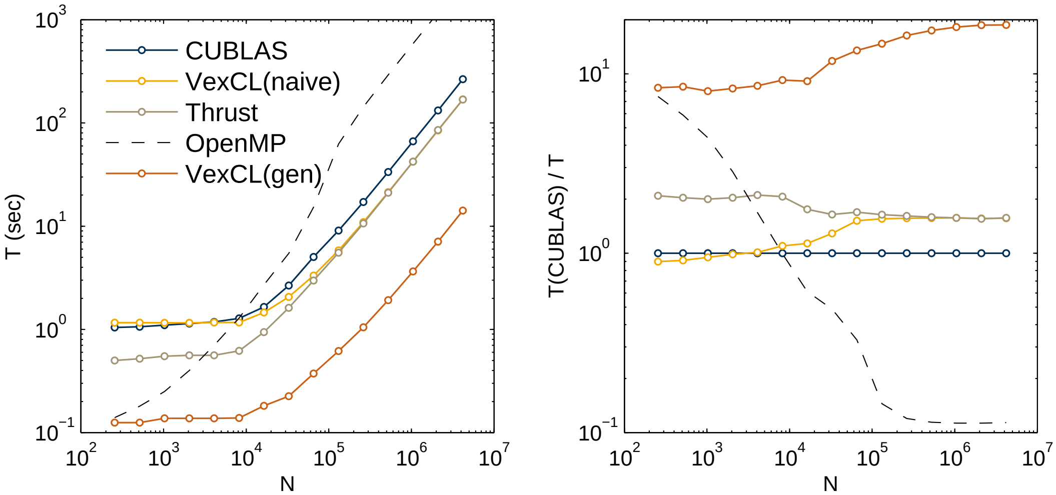
5. Generate and use OpenCL kernel

```
auto lorenz_kernel = vex::generator::build_kernel(ctx, "lorenz", lorenz_body.str(),
        sym_S[0], sym_S[1], sym_S[2], sym_R);

vex::vector<double> X(ctx, n), Y(ctx, n), Z(ctx, n), R(ctx, n);

X = Y = Z = 10;
R = Rmin + (Rmax - Rmin) * vex::element_index() / (n - 1);

for(double t = 0; t < t_max; t += dt) lorenz_kernel(X, Y, Z, R);
```

The restrictions

- Algorithms have to be embarrassingly parallel.
- Only linear flow is allowed (no conditionals or data-dependent loops).
- Some precision may be lost when converting constants to strings.

Performance of the generated kernel



Projects using VexCL

AMGCL — algebraic multigrid implementation:

- https://github.com/ddemidov/amgcl

Antioch — A New Templated Implementation Of Chemistry for Hydrodynamics:

- https://github.com/libantioch/antioch

Boost.odeint — numerical solution of Ordinary Differential Equations:

- http://odeint.com

Summary

- VexCL allows to write compact and readable code without sacrificing performance.
- Its code generator allows to convert generic C++ code to OpenCL at runtime:
  - Reduces global memory I/O
  - Reduces number of kernel launches
  - Facilitates code reuse
- Supported compilers (don't forget to enable C++11 features):
  - GCC v4.6
  - Clang v3.1
  - MS Visual C++ 2010
- https://github.com/ddemidov/vexcl