

VexCL

Vector Expression Template Library for OpenCL

Denis Demidov

Supercomputer Center of Russian Academy of Sciences
Kazan Federal University

Meeting C++, 9./10.11.12

VexCL: Vector expression template library for OpenCL

- Created for ease of C++ based OpenCL developement.
- The source code is publicly available¹ under MIT license.
- *This is not a C++ bindings library!*

1 Motivation

2 Basic interface

3 Kernel builder

4 Performance

5 Implementation details

6 Conclusion

¹<https://github.com/ddemidov/vexcl>

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

```
1 std::vector<cl::Platform> platform;  
2 cl::Platform::get(&platform);  
3  
4 if ( platform.empty() ) {  
5     std::cerr << "OpenCL platforms not found." << std::endl;  
6     return 1;  
7 }
```

Hello OpenCL: vector sum

2. Get first available GPU device

```
8  cl::Context context;
9  std::vector<cl::Device> device;
10 for(auto p = platform.begin(); device.empty() && p != platform.end(); p++) {
11     std::vector<cl::Device> pldev;
12     try {
13         p->getDevices(CL_DEVICE_TYPE_GPU, &pldev);
14         for(auto d = pldev.begin(); device.empty() && d != pldev.end(); d++) {
15             if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
16             device.push_back(*d);
17             context = cl::Context(device);
18         }
19     } catch(...) {
20         device.clear();
21     }
22 }
23 if (device.empty()) {
24     std::cerr << "GPUs not found." << std::endl;
25     return 1;
26 }
```

Hello OpenCL: vector sum

3. Create kernel source

```
27 const char source[] =  
28     "kernel void add(\n"  
29     "     uint n,\n"  
30     "     global const float *a,\n"  
31     "     global const float *b,\n"  
32     "     global float *c\n"  
33     "     )\n"  
34     "{\n"  
35     "     uint i = get_global_id(0);\n"  
36     "     if (i < n) {\n"  
37     "         c[i] = a[i] + b[i];\n"  
38     "     }\n"  
39     "}\n";
```

Hello OpenCL: vector sum

4. Compile kernel

```
40 cl::Program program(context, cl::Program::Sources(  
41     1, std::make_pair(source, strlen(source))  
42 ));  
43 try {  
44     program.build(device);  
45 } catch (const cl::Error&) {  
46     std::cerr  
47         << "OpenCL compilation error" << std::endl  
48         << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device[0])  
49         << std::endl;  
50     return 1;  
51 }  
52 cl::Kernel add_kernel = cl::Kernel(program, "add");
```

5. Create command queue

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

Hello OpenCL: vector sum

6. Prepare input data, transfer it to device

```
54 const unsigned int N = 1 << 20;
55 std::vector<float> a(N, 1), b(N, 2), c(N);
56
57 cl::Buffer A(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
58     a.size() * sizeof(float), a.data());
59
60 cl::Buffer B(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
61     b.size() * sizeof(float), b.data());
62
63 cl::Buffer C(context, CL_MEM_READ_WRITE,
64     c.size() * sizeof(float));
```


Hello OpenCL: vector sum

7. Set kernel arguments

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

8. Launch kernel

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

9. Get result back to host

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

Hello VexCL: vector sum

This is much shorter!

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

Hello VexCL: vector sum

Get all available GPUs

```
1 vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );
2 if ( !ctx.size() ) {
3     std::cerr << "GPUs not found." << std::endl;
4     return 1;
5 }
```

Prepare input data, transfer it to device

```
6 std::vector<float> a(N, 1), b(N, 2), c(N);
7 vex::vector<float> A(ctx.queue(), a);
8 vex::vector<float> B(ctx.queue(), b);
9 vex::vector<float> C(ctx.queue(), N);
```

Launch kernel, get result back to host

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

1 Motivation

2 Basic interface

- Device selection
- Vector arithmetic
- Reductions
- User-defined functions
- Sparse matrix – vector products
- Stencil convolutions
- Multivectors & multiexpressions

3 Kernel builder

4 Performance

5 Implementation details

6 Conclusion

Device selection

- 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

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

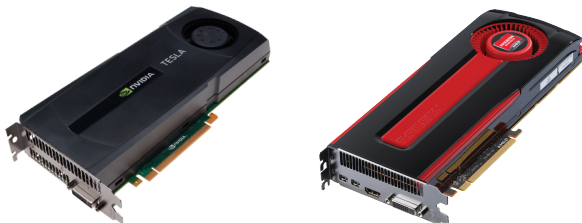


Device selection

- 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

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

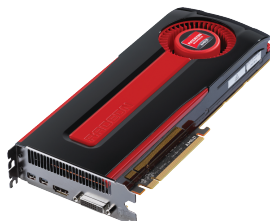


Device selection

- 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

```
1 vex::Context ctx(  
2     vex::Filter::Type(CL_DEVICE_TYPE_GPU) &&  
3     vex::Filter::Platform("AMD")  
4 );
```



Device selection

- 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

```
1 vex::Context ctx(  
2     vex::Filter::Type(CL_DEVICE_TYPE_GPU) &&  
3     [](const cl::Device &d) {  
4         return d.getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() >= 4.GB;  
5     });
```



Exclusive device access

- `vex::Filter::Exclusive()` wraps normal filters to allow exclusive access to devices.
- Useful for cluster environments.
- An alternative to NVIDIA's exclusive compute mode for other vendors hardware.
- Based on Boost.Interprocess file locks in temp directory.

```
1 vex::Context ctx( vex::Filter::Exclusive (
2     vex::Filter::DoublePrecision &&
3     vex::Filter::Env
4     ) );
```

What if OpenCL context is initialized elsewhere?

- You don't *have to* initialize `vex::Context`.
- `vex::Context` is just a convenient container that holds OpenCL contexts and queues.
- `vex::Context::queue()` returns `std::vector<cl::CommandQueue>`.
This may come from *elsewhere*.

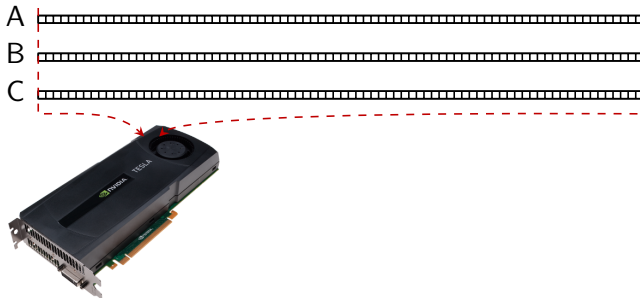
```
1 std::vector<cl::CommandQueue> my_own_vector_of_opengl_command_queues;  
2 // ...  
3 vex::vector<double> x(my_own_vector_of_opengl_command_queues, n);
```

- Each queue should correspond to a separate device.
- Different VexCL objects may be initialized with different queue lists.
- Operations are submitted to the queues of the vector that is being assigned to.

Vector allocation and arithmetic

Hello VexCL example

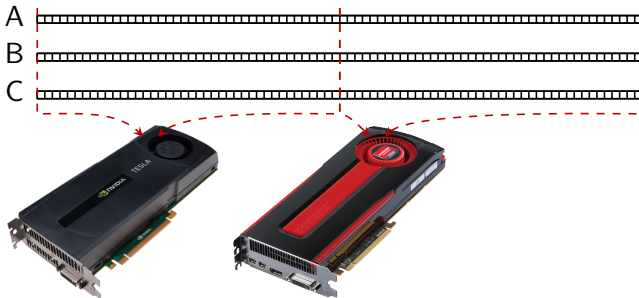
```
1 vex::Context ctx( vex::Filter::Name("Tesla") );  
2  
3 vex::vector<float> A(ctx.queue(), N); A = 1;  
4 vex::vector<float> B(ctx.queue(), N); B = 2;  
5 vex::vector<float> C(ctx.queue(), N);  
6  
7 C = A + B;
```



Vector allocation and arithmetic

Hello VexCL example

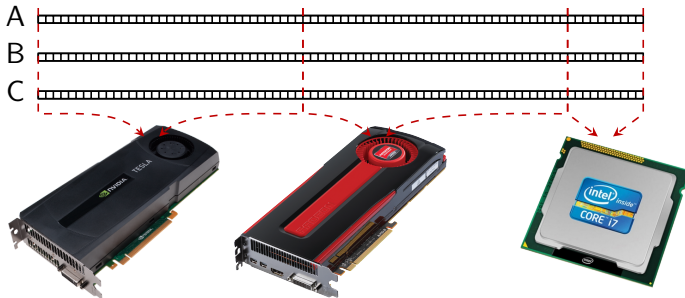
```
1 vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );  
2  
3 vex::vector<float> A(ctx.queue(), N); A = 1;  
4 vex::vector<float> B(ctx.queue(), N); B = 2;  
5 vex::vector<float> C(ctx.queue(), N);  
6  
7 C = A + B;
```



Vector allocation and arithmetic

Hello VexCL example

```
1 vex::Context ctx( vex::Filter::DoublePrecision );
2
3 vex::vector<float> A(ctx.queue(), N); A = 1;
4 vex::vector<float> B(ctx.queue(), N); B = 2;
5 vex::vector<float> C(ctx.queue(), N);
6
7 C = A + B;
```



What may be used in vector expressions?

- All vectors in expression have to be *compatible*:
 - Have same size
 - Located on same devices
- What may be used:
 - Scalar values
 - Arithmetic, bitwise, logical operators
 - Builtin OpenCL functions
 - User-defined functions

```
1 std::vector<float> x(n);
2 std::generate(x.begin(), x.end(), rand);
3
4 vex::vector<float> X(ctx.queue(), x);
5 vex::vector<float> Y(ctx.queue(), n);
6 vex::vector<float> Z(ctx.queue(), n);
7
8 Y = 42;
9 Z = sqrt(2 * X) + pow(cos(Y), 2.0);
```

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

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

Number of elements in x between 0 and 1

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

Maximum distance from origin

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

User-defined functions

- Users may define functions to be used in vector expressions:
 - Provide function body
 - Define return type and argument types

Defining a function

```
1 extern const char between_body[] = "return prm1 <= prm2 && prm2 <= prm3;";  
2 UserFunction<between_body, bool(double, double, double)> between;
```

Using a function: number of 2D points in first quadrant

```
1 size_t points_in_1q( const vex::Reductor<size_t, vex::SUM> &sum,  
2   const vex::vector<double> &x, const vex::vector<double> &y )  
3 {  
4   return sum( between(0.0, atan2(y, x), M_PI/2) );  
5 }
```


Sparse matrix – vector products

- 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).
- SpMV may only be used in additive expressions.

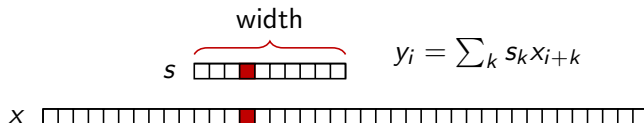
Construct matrix

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

Compute residual value

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

Simple stencil convolutions



- 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

```
1 std::vector<double> sdata(5, 0.2);  
2 vex::stencil<double> s(ctx.queue(), sdata, 2 /* center */);  
3  
4 y = x * s;
```

User-defined stencil operators

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

Example: nonlinear operator

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

Implementation

```
1 extern const char custom_op_body[] =  
2     "double t = X[-1] + X[1];\n"  
3     "return X[0] + t * t * t;"  
4  
5 vex::StencilOperator<double, 3 /*width*/, 1 /*center*/, custom_op_body>  
6     custom_op(ctx.queue());  
7  
8 y = custom_op(x);
```

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(uint k)` returns k -th component.

```
1 vex::multivector<double, 2> X( ctx.queue(), N), Y( ctx.queue(), N);
2 vex::Reductor<double, vex::SUM> sum(ctx.queue());
3 vex::SpMat<double> A( ctx.queue(), ... );
4 std::array<double, 2> v;
5
6 // ...
7
8 X = sin(v * Y + 1);           //  $X(k) = \sin(v[k] * Y(k) + 1)$ ;
9 v = sum( between(0, X, Y) ); //  $v[k] = \text{sum}( \text{between}( 0, X(k), Y(k) ) )$ ;
10 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

$$y_0 = x_0 \cos \alpha - x_1 \sin \alpha,$$

$$y_1 = x_0 \sin \alpha + x_1 \cos \alpha.$$

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

Multiexpressions

- Multiexpressions may be used with multivectors:

```
1 // double alpha;  
2 // vex::multivector<double,2> X, Y;  
3  
4 Y = std::tie(  
5     X(0) * cos(alpha) - X(1) * sin(alpha),  
6     X(0) * sin(alpha) + X(1) * cos(alpha) );
```

- and with tied vectors:

```
1 // vex::vector<double> alpha;  
2 // vex::vector<double> oldX, oldY, newX, newY;  
3  
4 vex::tie(newX, newY) = std::tie(  
5     oldX * cos(alpha) - oldY * sin(alpha),  
6     oldX * sin(alpha) + oldY * cos(alpha) );
```

- Any expression that is assignable to a vector is valid in a multiexpression.

Copies between host and device memory

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

Simple copies

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

STL-like range copies

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

Inspect or set single element (*slow*)

```
1 vex::copy(X, x);  
2 assert(x[42] == X[42]);  
3 X[0] = 0;
```

- 1 Motivation
- 2 Basic interface
- 3 Kernel builder
- 4 Performance
- 5 Implementation details
- 6 Conclusion

Converting generic C++ algorithms to OpenCL kernels*

*Restrictions applied

Motivating example

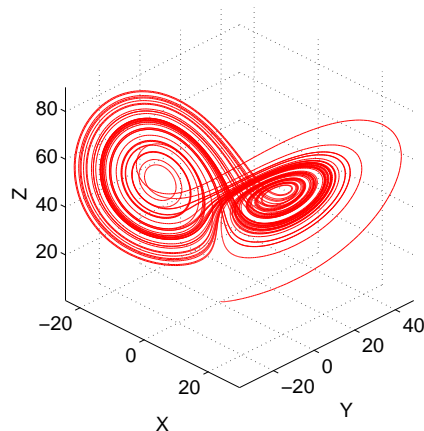
- Let's solve an ODE!
- Let's do it with Boost.odeint!

- Lorenz attractor system:

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

- We want to solve large number of Lorenz systems, each for a different value of R .

Lorenz attractor



odeint setup

1. System functor

```
1 typedef vex::vector<double>      vector_type;
2 typedef vex::multivector<double, 3> state_type;
3
4 struct lorenz_system {
5     const vector_type &R;
6     lorenz_system(const vector_type &R ) : R(R) { }
7
8     void operator()(const state_type &x, state_type &dxdt, double t) {
9         dxdt = std::tie(
10             sigma * ( x(1) - x(0) ),
11             R * x(0) - x(1) - x(0) * x(2),
12             x(0) * x(1) - b * x(2)
13         );
14     }
15 };
```

odeint setup

2. Integration

```
1 state_type X( ctx.queue(), n );
2 vector_type R( ctx.queue(), r );
3
4 // ... initialize X and R here ...
5
6 odeint::runge_kutta4<
7     state_type, double, state_type, double,
8     odeint::vector_space_algebra, odeint::default_operations
9     > stepper;
10
11 odeint::integrate_const( stepper, lorenz_system(R), X, 0.0, t_max, dt);
```

- That was easy!

odeint setup

2. Integration

```
1 state_type X( ctx.queue(), n );
2 vector_type R( ctx.queue(), r );
3
4 // ... initialize X and R here ...
5
6 odeint::runge_kutta4<
7     state_type, double, state_type, double,
8     odeint::vector_space_algebra, odeint::default_operations
9     > stepper;
10
11 odeint::integrate_const(stepper, lorenz_system(R), X, 0.0, t_max, dt);
```

- That was easy! And fast!

odeint setup

2. Integration

```
1 state_type X( ctx.queue(), n );
2 vector_type R( ctx.queue(), r );
3
4 // ... initialize X and R here ...
5
6 odeint::runge_kutta4<
7     state_type, double, state_type, double,
8     odeint::vector_space_algebra, odeint::default_operations
9     > stepper;
10
11 odeint::integrate_const(stepper, lorenz_system(R), X, 0.0, t_max, dt);
```

- That was easy! And fast! But,

odeint setup

2. Integration

```
1 state_type X( ctx.queue(), n );
2 vector_type R( ctx.queue(), r );
3
4 // ... initialize X and R here ...
5
6 odeint::runge_kutta4<
7     state_type, double, state_type, double,
8     odeint::vector_space_algebra, odeint::default_operations
9     > stepper;
10
11 odeint::integrate_const(stepper, lorenz_system(R), X, 0.0, t_max, dt);
```

- That was easy! And fast! But,
 - 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 single monolithic kernel that does one step of Runge-Kutta method.
- Launch the kernel in a loop.
- This is ≈ 10 times faster!

```
1 double3 lorenz_system(double r, double sigma, double b, double3 s) {
2     return (double3)(
3         sigma * (s.y - s.x),
4         r * s.x - s.y - s.x * s.z,
5         s.x * s.y - b * s.z
6     );
7 }
8
9 kernel void lorenz_ensemble(
10     ulong n, double sigma, double b,
11     const global double *R,
12     global double *X,
13     global double *Y,
14     global double *Z
15 )
16 {
17     double r;
18     double3 s, dsdt, k1, k2, k3, k4;
19
20     for(size_t gid = get_global_id(0); gid < n; gid += get_global_size(0)) {
21         r = R[gid];
22         s = (double3)(X[gid], Y[gid], Z[gid]);
23
24         k1 = dt * lorenz_system(r, sigma, b, s);
25         k2 = dt * lorenz_system(r, sigma, b, s + 0.5 * k1);
26         k3 = dt * lorenz_system(r, sigma, b, s + 0.5 * k2);
27         k4 = dt * lorenz_system(r, sigma, b, s + k3);
28
29         s += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
30
31         X[gid] = s.x; Y[gid] = s.y; Z[gid] = s.z;
32     }
33 }
```

What if we did this manually?

- Create single monolithic kernel that does one step of Runge-Kutta method.
- Launch the kernel in a loop.
- This is ≈ 10 times faster! But,

```

1 double3 lorenz_system(double r, double sigma, double b, double3 s) {
2     return (double3)(
3         sigma * (s.y - s.x),
4         r * s.x - s.y - s.x * s.z,
5         s.x * s.y - b * s.z
6     );
7 }
8
9 kernel void lorenz_ensemble(
10     ulong n, double sigma, double b,
11     const global double *R,
12     global double *X,
13     global double *Y,
14     global double *Z
15 )
16 {
17     double r;
18     double3 s, dsdt, k1, k2, k3, k4;
19
20     for(size_t gid = get_global_id(0); gid < n; gid += get_global_size(0)) {
21         r = R[gid];
22         s = (double3)(X[gid], Y[gid], Z[gid]);
23
24         k1 = dt * lorenz_system(r, sigma, b, s);
25         k2 = dt * lorenz_system(r, sigma, b, s + 0.5 * k1);
26         k3 = dt * lorenz_system(r, sigma, b, s + 0.5 * k2);
27         k4 = dt * lorenz_system(r, sigma, b, s + k3);
28
29         s += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
30
31         X[gid] = s.x; Y[gid] = s.y; Z[gid] = s.z;
32     }
33 }

```


What if we did this manually?

- Create single monolithic kernel that does one step of Runge-Kutta method.
- Launch the kernel in a loop.
- This is ≈ 10 times faster! But,
- We lost the generality odeint offers!

```
1 double3 lorenz_system(double r, double sigma, double b, double3 s) {
2     return (double3)(
3         sigma * (s.y - s.x),
4         r * s.x - s.y - s.x * s.z,
5         s.x * s.y - b * s.z
6     );
7 }
8
9 kernel void lorenz_ensemble(
10     ulong n, double sigma, double b,
11     const global double *R,
12     global double *X,
13     global double *Y,
14     global double *Z
15 )
16 {
17     double r;
18     double3 s, dsdt, k1, k2, k3, k4;
19
20     for(size_t gid = get_global_id(0); gid < n; gid += get_global_size(0)) {
21         r = R[gid];
22         s = (double3)(X[gid], Y[gid], Z[gid]);
23
24         k1 = dt * lorenz_system(r, sigma, b, s);
25         k2 = dt * lorenz_system(r, sigma, b, s + 0.5 * k1);
26         k3 = dt * lorenz_system(r, sigma, b, s + 0.5 * k2);
27         k4 = dt * lorenz_system(r, sigma, b, s + k3);
28
29         s += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
30
31         X[gid] = s.x; Y[gid] = s.y; Z[gid] = s.z;
32     }
33 }
```

Convert generic C++ algorithms to OpenCL kernels

- 1 Capture the sequence of arithmetic expressions of an algorithm.
- 2 Construct OpenCL kernel from the captured sequence.
- 3 ???
- 4 Use the kernel!

Convert generic C++ algorithms to OpenCL kernels

1. Declare functor operating on `vex::generator::symbolic<>` values

```
1 typedef vex::generator::symbolic< double > sym_vector;  
2 typedef std::array<sym_vector, 3> sym_state;  
3  
4 struct lorenz_system {  
5     const sym_vector &R;  
6     lorenz_system(const sym_vector &R) : R(R) {}  
7     void operator()(const sym_state &x, sym_state &dxdt, double t) const {  
8         dxdt[0] = sigma * (x[1] - x[0]);  
9         dxdt[1] = R * x[0] - x[1] - x[0] * x[2];  
10        dxdt[2] = x[0] * x[1] - b * x[2];  
11    }  
12 };
```

Convert generic C++ algorithms to OpenCL kernels

2. Record one step of Runge-Kutta method

```
1  std::ostream lorenz_body;
2  vex::generator::set_recorder(lorenz_body);
3
4  sym_state sym_S = {{
5      sym_vector::VectorParameter,
6      sym_vector::VectorParameter,
7      sym_vector::VectorParameter }};
8  sym_vector sym_R(sym_vector::VectorParameter, sym_vector::Const);
9
10 odeint::runge_kutta4<
11     sym_state, double, sym_state, double,
12     odeint::range_algebra, odeint::default_operations
13     > stepper;
14
15 lorenz_system sys(sym_R);
16 stepper.do_step(std::ref(sys), sym_S, 0, dt);
```

Convert generic C++ algorithms to OpenCL kernels

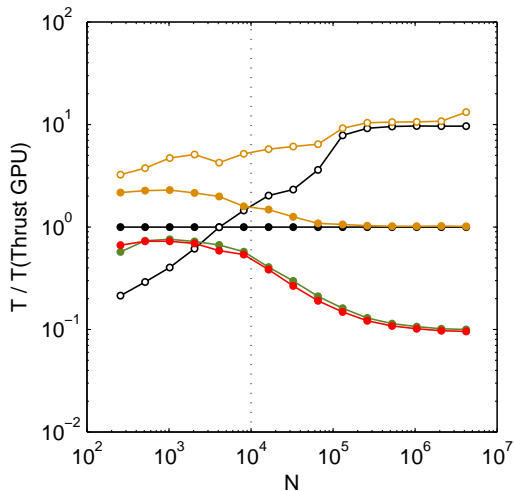
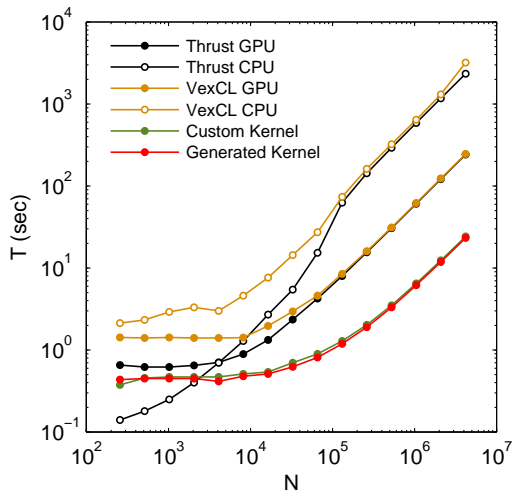
3. Generate and use OpenCL kernel

```
1  auto lorenz_kernel = vex::generator::build_kernel(ctx.queue(), "lorenz", lorenz_body.str(),
2      sym_S[0], sym_S[1], sym_S[2], sym_R);
3
4  vex::vector<double> X(ctx.queue(), n);
5  vex::vector<double> Y(ctx.queue(), n);
6  vex::vector<double> Z(ctx.queue(), n);
7  vex::vector<double> R(ctx.queue(), r);
8
9  // ... initialize X, Y, Z, and R here ...
10
11 for(double t = 0; t < t_max; t += dt) lorenz_kernel(X, Y, Z, R);
```

The restrictions

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

The performance results

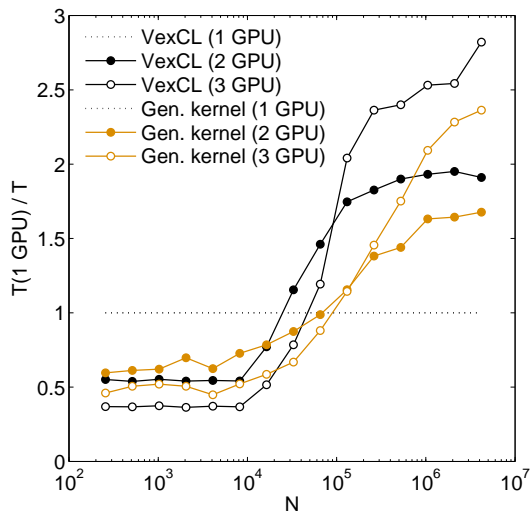


GPU: NVIDIA Tesla C2070

CPU: Intel Core i7 930

Multigpu scalability

- *Larger problems may be solved on the same system.*
- *Large problems may be solved faster.*



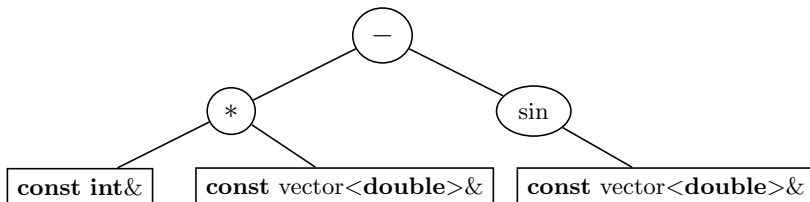
- 1 Motivation
- 2 Basic interface
- 3 Kernel builder
- 4 Performance
- 5 Implementation details**
- 6 Conclusion

Expression trees

- VexCL is an *expression template* library
- Each expression in the code results in an expression tree evaluated at time of assignment.
 - No temporaries are created
 - Single kernel is generated and executed

Example expression

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

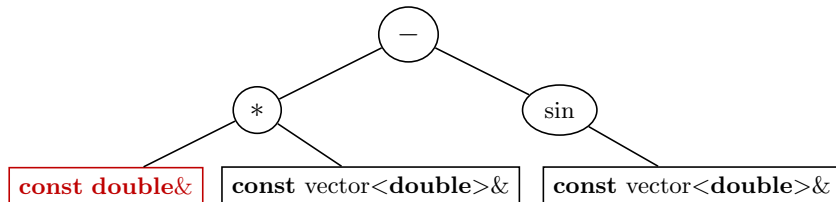


Expression trees

- VexCL is an *expression template* library
- Each expression in the code results in an expression tree evaluated at time of assignment.
 - No temporaries are created
 - Single kernel is generated and executed

Example expression

1 `x = 2.0 * y - sin(z);`



Kernel generation

The expression

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

Define `VEXCL_SHOW_KERNELS` to see the generated code.

... results in this kernel:

```
1 kernel void minus_multiplies_term_term_sin_term(  
2     ulong n,  
3     global double *res,  
4     int prm_1,  
5     global double *prm_2,  
6     global double *prm_3  
7 )  
8 {  
9     for( size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {  
10         res[idx] = ( ( prm_1 * prm_2[idx] ) - sin( prm_3[idx] ) );  
11     }  
12 }
```

Performance tip

- No way to tell if two terminals refer to the same data!
- Example: finding number of points in 1st quadrant

Naive

```
1 return sum( 0.0 <= atan2(y, x) && atan2(y, x) <= M_PI/2 );
```

- **x and y are read twice**
- **atan2 is computed twice**

Using custom function

```
1 return sum( between(0.0, atan2(y, x), M_PI/2) );
```

Custom kernels

It is possible to use custom kernels with VexCL vectors

```
1  vex::vector<float> x(ctx.queue(), n);
2
3  for(uint d = 0; d < ctx.size(); d++) {
4      cl::Program program = build_sources(ctx.context(d),
5          "kernel void dummy(ulong size, global float *x) {\n"
6          "    x[ get_global_id (0)] = 4.2;\n"
7          "}\n");
8
9      cl::Kernel dummy(program, "dummy");
10
11     dummy.setArg(0, static_cast<cl_ulong>(x.part_size(d)));
12     dummy.setArg(1, x(d));
13
14     ctx.queue(d).enqueueNDRangeKernel(dummy, cl::NullRange, x.part_size(d), cl::NullRange);
15 }
```

Conclusion and Questions

- VexCL allows to write compact and readable code without sacrificing too much performance.
- Multiple compute devices are employed transparently.
- Supported compilers (don't forget to enable C++11 features):
 - GCC v4.6
 - Clang v3.1
 - MS Visual C++ 2010 (partially)

- <https://github.com/ddemidov/vexcl>



Conjugate gradients method

Solve linear equations system $Au = f$

```

1 void cg::solve(const vex::SpMat<double> &A, const vex::vector<double> &f, vex::vector<double> &u) {
2     // Member fields:
3     // vex::vector<double> r, p, q;
4     // Reductor<double,MAX> max; Reductor<double,SUM> sum;
5
6     double rho1 = 0, rho2 = 1;
7     r = f - A * u;
8
9     for(int iter = 0; max( fabs(r) ) > 1e-8 && iter < n; iter++) {
10         rho1 = sum(r * r);
11
12         if (iter == 0) {
13             p = r;
14         } else {
15             double beta = rho1 / rho2;
16             p = r + beta * p;
17         }
18
19         q = A * p;
20
21         double alpha = rho1 / sum(p * q);
22
23         u += alpha * p;
24         r -= alpha * q;
25
26         rho2 = rho1;
27     }
28 }
```


The generated kernel (is ugly)

```

1 kernel void lorenz(
2   ulong n,
3   global doubles p,var0,
4   global doubles p,var1,
5   global doubles p,var2,
6   global const doubles p,var3
7 )
8 {
9   size_t idx = get_global_id(0);
10  if (idx < n) {
11    double var0 = p.var0[idx];
12    double var1 = p.var1[idx];
13    double var2 = p.var2[idx];
14    double var3 = p.var3[idx];
15    double var4;
16    double var5;
17    double var6;
18    double var7;
19    double var8;
20    double var9;
21    double var10;
22    double var11;
23    double var12;
24    double var13;
25    double var14;
26    double var15;
27    double var16;
28    double var17;
29    double var18;
30    var4 = (1.000000000000e+01 * (var1 - var0));
31    var5 = (((var3 * var0) - var1) - (var0 * var2));
32    var6 = ((var0 * var1) - (2.666666666666e+00 * var2));
33    var7 = (((1.000000000000e+00 * var0) + (5.000000000000e-03 * var4));
34    var8 = (((1.000000000000e+00 * var1) + (5.000000000000e-03 * var5));
35    var9 = (((1.000000000000e+00 * var2) + (5.000000000000e-03 * var6));
36    var10 = (1.000000000000e+01 * (var8 - var7));
37    var11 = (((var3 * var7) - var8) - (var7 * var9));
38    var12 = (((var7 * var8) - (2.666666666666e+00 * var9));
39    var7 = (((1.000000000000e+00 * var0) + (0.000000000000e+00 * var4)) + (5.000000000000e-03 * var10);
40    var8 = (((1.000000000000e+00 * var1) + (0.000000000000e+00 * var5)) + (5.000000000000e-03 * var11);
41    var9 = (((1.000000000000e+00 * var2) + (0.000000000000e+00 * var6)) + (5.000000000000e-03 * var12);
42    var13 = (1.000000000000e+01 * (var8 - var7));
43    var14 = (((var3 * var7) - var8) - (var7 * var9));
44    var15 = (((var7 * var8) - (2.666666666666e+00 * var9));
45    var7 = (((1.000000000000e+00 * var0) + (0.000000000000e+00 * var4)) + (0.000000000000e+00 * var10)) + (1.000000000000e-02 * var13);
46    var8 = (((1.000000000000e+00 * var1) + (0.000000000000e+00 * var5)) + (0.000000000000e+00 * var11)) + (1.000000000000e-02 * var14);
47    var9 = (((1.000000000000e+00 * var2) + (0.000000000000e+00 * var6)) + (0.000000000000e+00 * var12)) + (1.000000000000e-02 * var15);
48    var16 = (1.000000000000e+01 * (var8 - var7));
49    var17 = (((var3 * var7) - var8) - (var7 * var9));
50    var18 = (((var7 * var8) - (2.666666666666e+00 * var9));
51    var0 = (((1.000000000000e+00 * var0) + (1.666666666666e-03 * var4)) + (3.333333333333e-03 * var10)) + (3.333333333333e-03 * var13)) + (1.666666666666e-03 * var16);
52    var1 = (((1.000000000000e+00 * var1) + (1.666666666666e-03 * var5)) + (3.333333333333e-03 * var11)) + (3.333333333333e-03 * var14)) + (1.666666666666e-03 * var17);
53    var2 = (((1.000000000000e+00 * var2) + (1.666666666666e-03 * var6)) + (3.333333333333e-03 * var12)) + (3.333333333333e-03 * var15)) + (1.666666666666e-03 * var18);
54    p.var0[idx] = var0;
55    p.var1[idx] = var1;
56    p.var2[idx] = var2;
57  }
58 }

```