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 available[1] under MIT license.
- *This is not a C++ bindings library!*

[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

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

if ( platform.empty() ) {
    std::cerr << "OpenCL platforms not found." << std::endl;
    return 1;
}
```

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> pldev;
    try {
        p->getDevices(CL_DEVICE_TYPE_GPU, &pldev);
        for(auto d = pldev.begin(); device.empty() && d != pldev.end(); d++) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            device.push_back(*d);
            context = cl::Context(device);
        }
    } catch(...) {
        device.clear();
    }
}
if (device.empty()) {
    std::cerr << "GPUs not found." << std::endl;
    return 1;
}
```

Hello OpenCL: vector sum

3. Create kernel source

```c
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 unsigned int 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

This is 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.size() ) {
    std::cerr << "GPUs not found." << std::endl;
    return 1;
}
```

Prepare input data, transfer it to device

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

Launch kernel, get result back to host

```
C = A + B;
vex::copy(C, c);
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

```
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

```
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

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

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

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

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.

```
vex::Context ctx( vex::Filter::Exclusive (
        vex::Filter::DoublePrecision &&
        vex::Filter::Env
        ) );
```

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*.

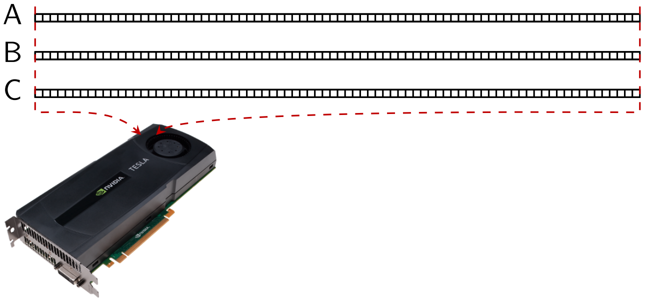
```
std::vector<cl::CommandQueue> my_own_vector_of_opencl_command_queues;
// ...
vex::vector<double> x(my_own_vector_of_opencl_command_queues, n);
```

- 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

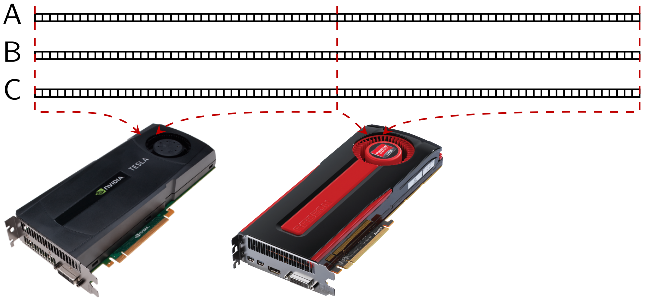
```
vex::Context ctx( vex::Filter::Name("Tesla") );

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

C = A + B;
```



Vector allocation and arithmetic

Hello VexCL example

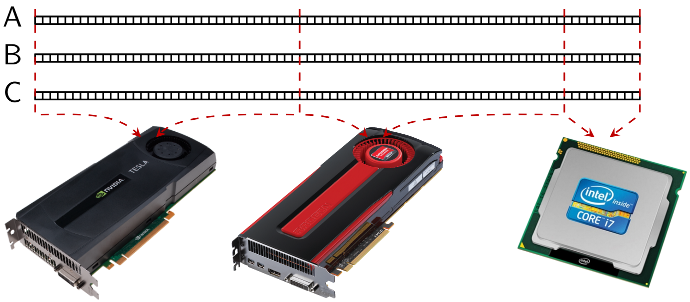
```
vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );

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

C = A + B;
```



Vector allocation and arithmetic

Hello VexCL example

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

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

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

```
std::vector<float> x(n);
std::generate(x.begin(), x.end(), rand);

vex::vector<float> X(ctx.queue(), x);
vex::vector<float> Y(ctx.queue(), n);
vex::vector<float> Z(ctx.queue(), n);

Y = 42;
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

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

Number of elements in x between 0 and 1

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

Maximum distance from origin

```
vex::Reductor<double, vex::MAX> max(ctx.queue());
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

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

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

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

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.

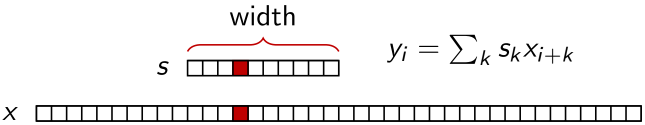
Construct matrix

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

Compute residual value

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

Simple stencil convolutions



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

- 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.queue(), sdata, 2 /* center */);

y = x * s;
```

User-defined stencil operators

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

Example: nonlinear operator

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

Implementation

```
extern const char custom_op_body[] =
    "double t = X[-1] + X[1];\n"
    "return X[0] + t * t * t;"

vex::StencilOperator<double, 3 /*width*/, 1 /*center*/, custom_op_body>
    custom_op(ctx.queue());

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.

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

$$y_0 = x_0 \cos\alpha - x_1 \sin\alpha,$$
$$y_1 = x_0 \sin\alpha + x_1 \cos\alpha.$$

- 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:

```
// 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),
        odlX * sin(alpha) + oldY * cos(alpha) );
```

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

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*)

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

Converting generic C++ algorithms to OpenCL kernels*

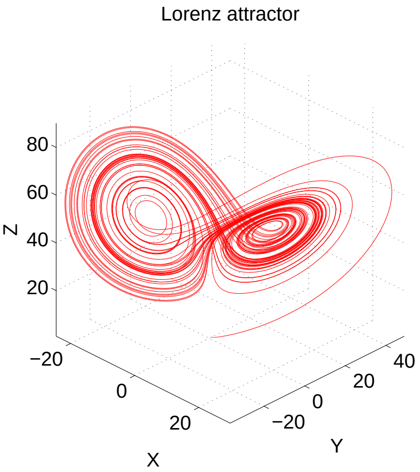
*Restrictions applied

Motivating example

- Let's solve an ODE!
- Let's do it with Boost.odeint!
- Lorenz attractor system:

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

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



odeint setup

1. System functor

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

struct lorenz_system {
    const vector_type &R;
    lorenz_system(const vector_type &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)
                );
    }
};
```

odeint setup

2. Integration

```
state_type  X( ctx.queue(), n );
vector_type R( ctx.queue(), r );

// ... initialize X and R here ...

odeint::runge_kutta4<
        state_type, double, state_type, double,
        odeint::vector_space_algebra, odeint::default_operations
        > stepper;

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

- That was easy!

odeint setup

2. Integration

```
state_type  X( ctx.queue(), n );
vector_type R( ctx.queue(), r );

// ... initialize X and R here ...

odeint::runge_kutta4<
        state_type, double, state_type, double,
        odeint::vector_space_algebra, odeint::default_operations
        > stepper;

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

- That was easy! And fast!

odeint setup

2. Integration

```
state_type  X( ctx.queue(), n );
vector_type R( ctx.queue(), r );

// ... initialize X and R here ...

odeint::runge_kutta4<
        state_type, double, state_type, double,
        odeint::vector_space_algebra, odeint::default_operations
        > stepper;

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

- That was easy! And fast! But,

odeint setup

2. Integration

```
state_type  X( ctx.queue(), n );
vector_type R( ctx.queue(), r );

// ... initialize X and R here ...

odeint::runge_kutta4<
        state_type, double, state_type, double,
        odeint::vector_space_algebra, odeint::default_operations
        > stepper;

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?

1. Create single monolithic kernel that does one step of Runge-Kutta method.
2. Launch the kernel in a loop.
3. This is $\approx$ 10 times 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 sigma, double b,
    const global double *R,
    global double *X,
    global double *Y,
    global double *Z
    )
{
    double r;
    double3 s, dsdt, k1, k2, k3, k4;

    for(size_t gid = get_global_id(0); gid < n; gid += get_global_size(0)) {
        r = R[gid];
        s = (double3)(X[gid], Y[gid], Z[gid]);

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

What if we did this manually?

1. Create single monolithic kernel that does one step of Runge-Kutta method.
2. Launch the kernel in a loop.

3. This is $\approx$ 10 times faster! But,

```
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 sigma, double b,
    const global double *R,
    global double *X,
    global double *Y,
    global double *Z
    )
{
    double r;
    double3 s, dsdt, k1, k2, k3, k4;

    for( size_t gid = get_global_id(0); gid < n; gid += get_global_size(0)) {
        r = R[gid];
        s = (double3)(X[gid], Y[gid], Z[gid]);

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

What if we did this manually?

1. Create single monolithic kernel that does one step of Runge-Kutta method.
2. Launch the kernel in a loop.

3. This is $\approx$ 10 times faster! But,
4. We lost the generality odeint offers!

```
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 sigma, double b,
    const global double *R,
    global double *X,
    global double *Y,
    global double *Z
    )
{
    double r;
    double3 s, dsdt, k1, k2, k3, k4;

    for( size_t gid = get_global_id(0); gid < n; gid += get_global_size(0)) {
        r = R[gid];
        s = (double3)(X[gid], Y[gid], Z[gid]);

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

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

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

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];
    }
};
```

Convert generic C++ algorithms to OpenCL kernels

2. Record one step of Runge-Kutta method

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

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

odeint::runge_kutta4<
        sym_state, double, sym_state, double,
        odeint::range_algebra, odeint::default_operations
        > stepper;

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

Convert generic C++ algorithms to OpenCL kernels

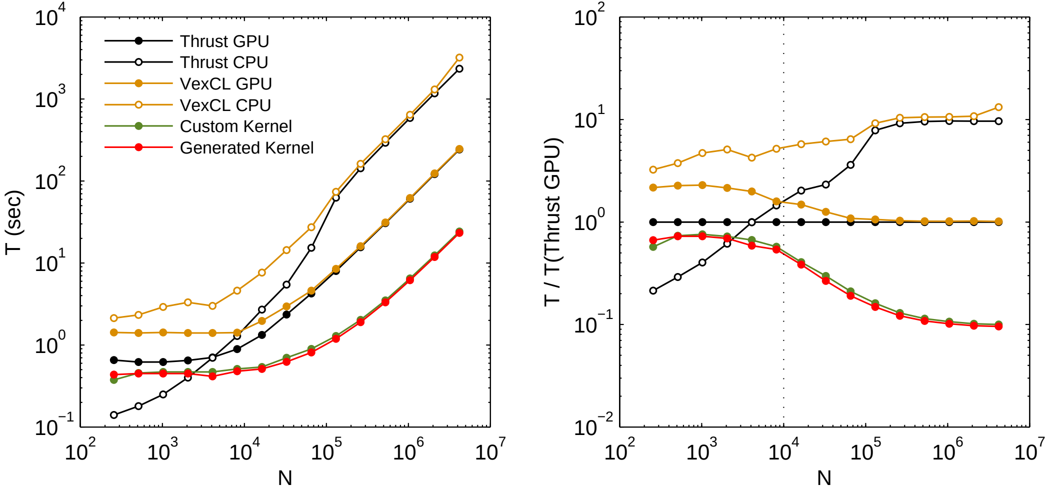
3. Generate and use OpenCL kernel

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

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

// ... initialize X, Y, Z, and R here ...

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

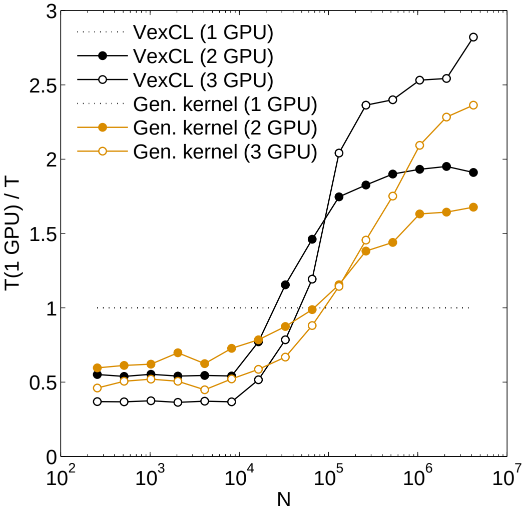


GPU: NVIDIA Tesla C2070

CPU: Intel Core i7 930

Multigpu scalability

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

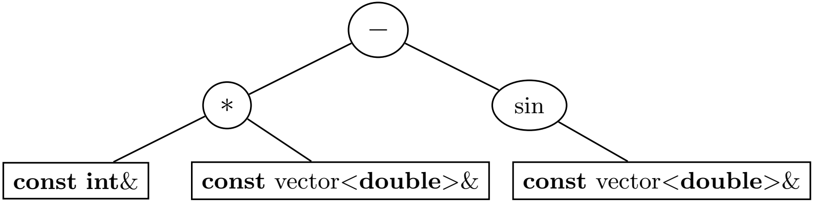


Expression trees

- 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

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

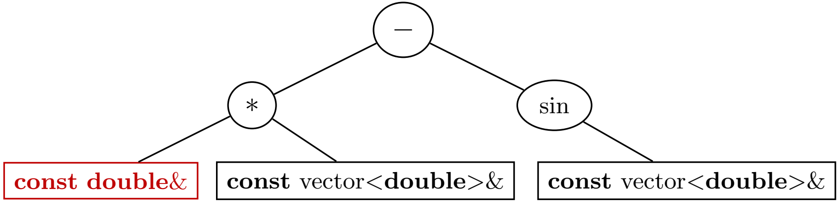


Expression trees

- 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

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



Kernel generation

The expression

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

Define VEXCL_SHOW_KERNELS to see the generated code.

. . . results in this kernel:

```
kernel void minus_multiplies_term_term_sin_term(
    ulong n,
    global double *res,
    int prm_1,
    global double *prm_2,
    global double *prm_3
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        res[idx] = ( ( prm_1 * prm_2[idx] ) - sin( prm_3[idx] ) );
    }
}
```

Performance tip

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

Naive

```
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

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

Custom kernels

It is possible to use custom kernels with VexCL vectors

```
vex::vector<float> x(ctx.queue(), n);

for(uint d = 0; d < ctx.size(); d++) {
    cl::Program program = build_sources(ctx.context(d),
        "kernel void dummy(ulong size, global float *x) {\n"
        "    x[get_global_id(0)] = 4.2;\n"
        "}\n");

    cl::Kernel dummy(program, "dummy");

    dummy.setArg(0, static_cast<cl_ulong>(x.part_size(d)));
    dummy.setArg(1, x(d));

    ctx.queue(d).enqueueNDRangeKernel(dummy, cl::NullRange, x.part_size(d), cl::NullRange);
}
```

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 $Au = f$

```
void cg::solve(const vex::SpMat<double> &A, const vex::vector<double> &f, vex::vector<double> &u) {
    // Member fields:
    // vex::vector<double> r, p, q;
    // Reductor<double,MAX> max; Reductor<double,SUM> sum;

    double rho1 = 0, rho2 = 1;
    r = f - A * u;

    for(int iter = 0; max( fabs(r) ) > 1e-8 && iter < n; iter++) {
        rho1 = sum(r * r);

        if (iter == 0) {
            p = r;
        } else {
            double beta = rho1 / rho2;
            p = r + beta * p;
        }

        q = A * p;

        double alpha = rho1 / sum(p * q);

        u += alpha * p;
        r -= alpha * q;

        rho2 = rho1;
    }
}
```

The generated kernel (is ugly)

```
kernel void lorenz(
ulong n,
global double* p_var0,
global double* p_var1,
global double* p_var2,
global const double* p_var3
)
{
 size_t idx = get_global_id(0);
 if (idx < n) {
double var0 = p_var0[idx];
double var1 = p_var1[idx];
double var2 = p_var2[idx];
double var3 = p_var3[idx];
double var4;
double var5;
double var6;
double var7;
double var8;
double var9;
double var10;
double var11;
double var12;
double var13;
double var14;
double var15;
double var16;
double var17;
double var18;
var4 = (1.000000000000e+01 * (var1 - var0));
var5 = (((var3 * var0) - var1) - (var0 * var2));
var6 = ((var0 * var1) - (2.666666666666e+00 * var2));
var7 = ((1.000000000000e+00 * var0) + (5.000000000000e-03 * var4));
var8 = ((1.000000000000e+00 * var1) + (5.000000000000e-03 * var5));
var9 = ((1.000000000000e+00 * var2) + (5.000000000000e-03 * var6));
var10 = (1.000000000000e+01 * (var8 - var7));
var11 = (((var3 * var7) - var8) - (var7 * var9));
var12 = ((var7 * var8) - (2.666666666666e+00 * var9));
var7 = (((1.000000000000e+00 * var0) + (0.000000000000e+00 * var4)) + (5.000000000000e-03 * var10));
var8 = (((1.000000000000e+00 * var1) + (0.000000000000e+00 * var5)) + (5.000000000000e-03 * var11));
var9 = (((1.000000000000e+00 * var2) + (0.000000000000e+00 * var6)) + (5.000000000000e-03 * var12));
var13 = (1.000000000000e+01 * (var8 - var7));
var14 = (((var3 * var7) - var8) - (var7 * var9));
var15 = ((var7 * var8) - (2.666666666666e+00 * var9));
var7 = ((((1.000000000000e+00 * var0) + (0.000000000000e+00 * var4)) + (0.000000000000e+00 * var10)) + (1.000000000000e-02 * var13));
var8 = ((((1.000000000000e+00 * var1) + (0.000000000000e+00 * var5)) + (0.000000000000e+00 * var11)) + (1.000000000000e-02 * var14));
var9 = ((((1.000000000000e+00 * var2) + (0.000000000000e+00 * var6)) + (0.000000000000e+00 * var12)) + (1.000000000000e-02 * var15));
var16 = (1.000000000000e+01 * (var8 - var7));
var17 = (((var3 * var7) - var8) - (var7 * var9));
var18 = ((var7 * var8) - (2.666666666666e+00 * var9));
var0 = (((((1.000000000000e+00 * var0) + (1.666666666666e-03 * var4)) + (3.333333333333e-03 * var10)) + (3.333333333333e-03 * var13)) + (1.666666666666e-03 * var16));
var1 = (((((1.000000000000e+00 * var1) + (1.666666666666e-03 * var5)) + (3.333333333333e-03 * var11)) + (3.333333333333e-03 * var14)) + (1.666666666666e-03 * var17));
var2 = (((((1.000000000000e+00 * var2) + (1.666666666666e-03 * var6)) + (3.333333333333e-03 * var12)) + (3.333333333333e-03 * var15)) + (1.666666666666e-03 * var18));
p_var0[idx] = var0;
p_var1[idx] = var1;
p_var2[idx] = var2;
}
}
```