VexCL

Vector Expression Template Library for OpenCL

Denis Demidov

Kazan Federal University
Supercomputer Center of Russian Academy of Sciences

CSE13, Boston, February 26

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 VexCL: vector sum

Get all available GPUs:

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

Prepare input data, transfer it to device:

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

Launch kernel, get result back to host:

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

1. Motivating example

2. Interface
   - Device selection
   - Vector arithmetic
   - Reductions
   - User-defined functions
   - Using element indices
   - Random number generation
   - Sparse matrix – vector products
   - Stencil convolutions
   - Fast Fourier Transform
   - Multivectors & multiexpressions

3. Performance

4. Implementation details

5. 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 in 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 && vex::Filter::Env
3     ) );
```

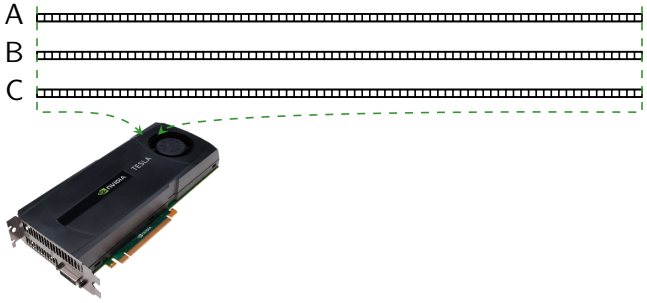
Using several contexts

- Different VexCL objects may be initialized with different VexCL contexts.
  - Manual work splitting across devices
  - Doing things in parallel on devices that support it
- Operations are submitted to the queues of the vector that is being assigned to.

Vector allocation and arithmetic

Hello VexCL example

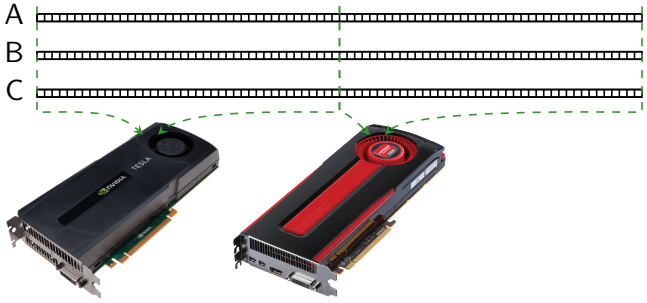
```
1  vex::Context ctx( vex::Filter::Name("Tesla") );
2
3  vex::vector<float> A(ctx, N); A = 1;
4  vex::vector<float> B(ctx, N); B = 2;
5  vex::vector<float> C(ctx, N);
6
7  C = A + B;
```



Vector allocation and arithmetic

Hello VexCL example

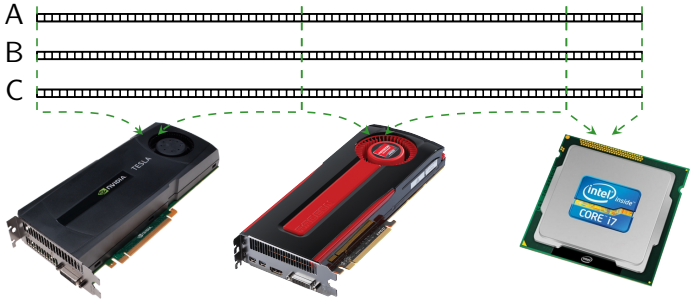
```
1  vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );
2
3  vex::vector<float> A(ctx, N); A = 1;
4  vex::vector<float> B(ctx, N); B = 2;
5  vex::vector<float> C(ctx, N);
6
7  C = A + B;
```



Vector allocation and arithmetic

Hello VexCL example

```
1  vex::Context ctx( vex::Filter::DoublePrecision );
2
3  vex::vector<float> A(ctx, N); A = 1;
4  vex::vector<float> B(ctx, N); B = 2;
5  vex::vector<float> C(ctx, 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
  - Built-in 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, x);
5  vex::vector<float> Y(ctx, n);
6  vex::vector<float> Z(ctx, 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);
2 double s = sum(x * y);
```

Number of elements in x between 0 and 1

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

Maximum distance from origin

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

User-defined functions

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

Defining a function

```
1 VEX_FUNCTION( between, bool(double, double, double),
2     "return prm1 <= prm2 && prm2 <= prm3;" );
```

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 }
```

Using element indices in expressions

- 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:

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

Single period of sine function:

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

Random number generation

- VexCL provides implementation[2] of *counter-based* random number generators from Random123[3] suite.
  - The generators are *stateless*; mixing functions are applied to element indices.
  - Implemented families: Threefry and Philox.

Monte Carlo π:

```
1  vex::Random<double, vex::random::threefry> rnd;        // RandomNormal<> is also available
2  vex::Reductor<size_t, vex::SUM> sum(ctx);
3  vex::vector<double> x(ctx, n), y(ctx, n);
4
5  x = 2 * rnd(vex::element_index(), std::rand()) - 1;
6  y = 2 * rnd(vex::element_index(), std::rand()) - 1;
7
8  double pi = 4.0 * sum(x * x + y * y < 1) / n;
```

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

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

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

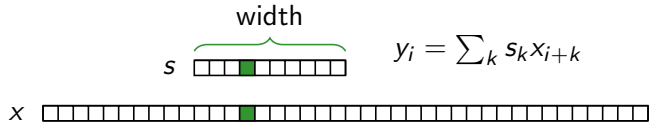
Construct matrix

```
1 vex::SpMat<double> A(ctx, 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 (*Additive expressions only*)



- 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, sdata, 2 /* center */);
3
4  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

```
1 VEX_STENCIL_OPERATOR(custom_op, double, 3/*width*/, 1/*center*/,
2     "double t = X[-1] + X[1];\n"
3     "return X[0] + t * t * t;",
4     ctx);
5
6 y = custom_op(x);
```

Fast Fourier Transform *(Additive expressions only)*

- VexCL provides FFT implementation[4]:
  - Currently only single-device contexts are supported
  - Arbitrary vector expressions as input
  - Multidimensional transforms
  - Arbitrary sizes

```
1 vex::FFT<double, cl_double2> fft(ctx, n);
2 vex::FFT<cl_double2, double> ifft(ctx, n, vex::inverse);
3
4 vex::vector<double>  in(ctx, n), back(ctx, n);
5 vex::vector<cl_double2> out(ctx, n);
6 // ... initialize 'in' ...
7
8 out  = fft(in);
9 back = ifft(out);
```

[4] 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**(uint k) returns k-th component.

```
1  vex::multivector<double, 2> X(ctx, N), Y(ctx, N);
2  vex::Reductor<double, vex::SUM> sum(ctx);
3  vex::SpMat<double> A(ctx, ... );
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] = sum( 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 component-wise 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( X(0) * cos(alpha) - X(1) * sin(alpha),
5                X(0) * sin(alpha) + X(1) * cos(alpha) );
```

- and with tied vectors:

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

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*)

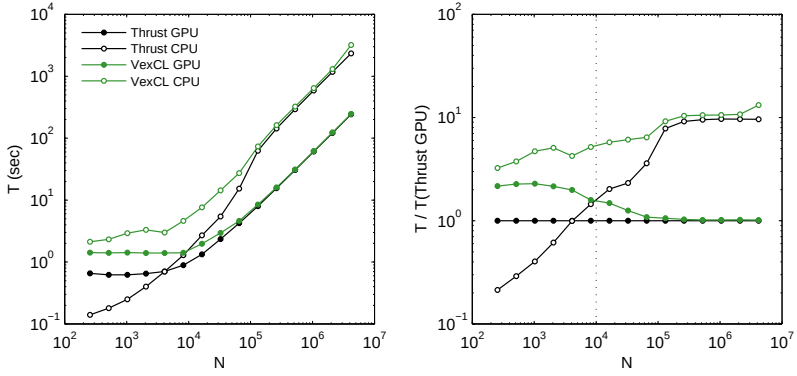
```
assert(x[42] == X[42]);
X[0] = 0;
```

Performance

- Solving ODE (Lorenz attractor ensemble) with Boost.odeint, Thrust, and VexCL[5]

  GPU: NVIDIA Tesla C2070

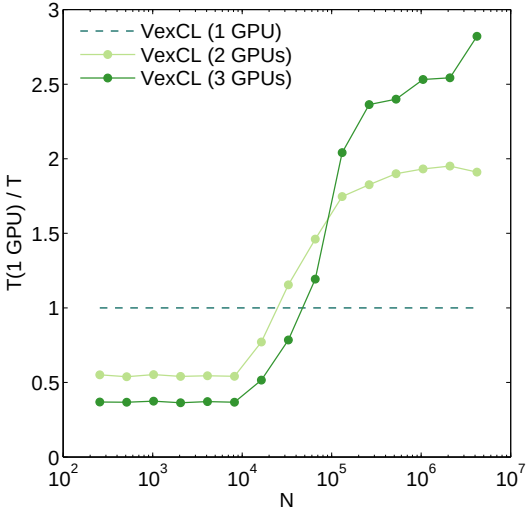
  CPU: Intel Core i7 930



[5] *Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries*. Denis Demidov, Karsten Ahnert, Karl Rupp, Peter Gottschling. arXiv:1212.6326

Multigpu scalability

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

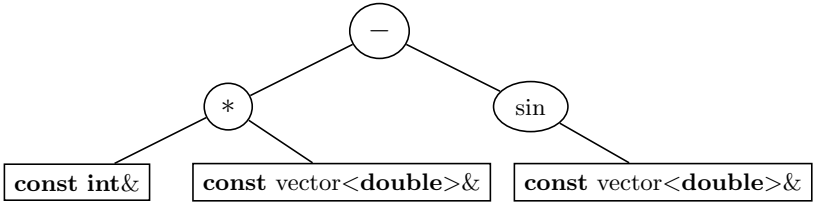


Expression trees

- VexCL is an *expression template* library.
- Boost.Proto is used as an expression template engine.
- 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);
```

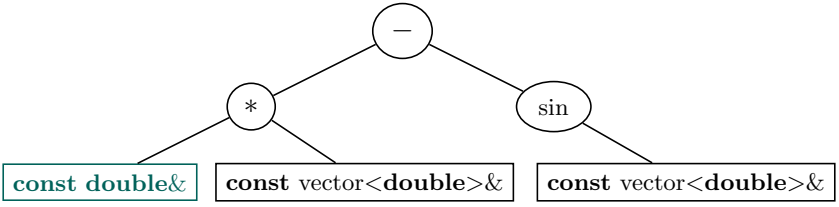


Expression trees

- VexCL is an *expression template* library.
- Boost.Proto is used as an expression template engine.
- 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 }
```

Conclusion and Questions

- VexCL allows to write compact and readable code without sacrificing 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

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