
VexCL Documentation

Release 1.4.1

Denis Demidov

May 04, 2017

Contents

1 Contents:	3
Bibliography	43

VexCL is a vector expression template library for OpenCL/CUDA. It has been created for ease of GPGPU development with C++. VexCL strives to reduce amount of boilerplate code needed to develop GPGPU applications. The library provides convenient and intuitive notation for vector arithmetic, reduction, sparse matrix-vector products, etc. Multi-device and even multi-platform computations are supported.

The library source code is available under MIT license at <https://github.com/ddemidov/vexcl>.

Initialization

Selecting backend

VexCL provides the following backends:

- **OpenCL**, built on top of [Khronos C++ API](#). The backend is selected when `VEXCL_BACKEND_OPENCL` macro is defined, or by default. Link with `libOpenCL.so` on unix-like systems or with `OpenCL.dll` on Windows.
- **Boost.Compute**. The backend is also based on OpenCL, but uses core functionality of the [Boost.Compute](#) library instead of somewhat outdated Khronos C++ API. The additional advantage is the increased interoperability between VexCL and [Boost.Compute](#). The backend is selected when `VEXCL_BACKEND_COMPUTE` macro is defined. Link with `libOpenCL.so/OpenCL.dll` and make sure that [Boost.Compute](#) headers are in the include path.
- **CUDA**, uses the NVIDIA CUDA technology. The backend is selected when `VEXCL_BACKEND_CUDA` macro is defined. Link with `libcuda.so/cuda.dll`. For the CUDA backend to work, CUDA Toolkit has to be installed, and NVIDIA CUDA compiler driver `nvcc` has to be in executable `PATH` and usable at runtime.

Whatever backend is selected, you will need to link to [Boost.System](#) and [Boost.Filesystem](#) libraries. Some systems may also require linking to [Boost.Thread](#) and [Boost.Date_Time](#). All of those are distributed with [Boost](#) libraries collection.

Context initialization

VexCL transparently works with multiple compute devices that are present in the system. A VexCL context is initialized with a device filter, which is just a functor that takes a `vex::backend::device` instance and returns a boolean value. Several standard filters are provided (see below), and one can easily add a custom functor. Filters may be combined with logical operators. All compute devices that satisfy the provided filter are added to the created context. In the example below all GPU devices that support double precision arithmetic are selected:

```
#include <iostream>
#include <stdexcept>
#include <vexcl/vexcl.hpp>

int main() {
    vex::Context ctx( vex::Filter::GPU && vex::Filter::DoublePrecision );

    if (!ctx) throw std::runtime_error("No devices available.");

    // Print out list of selected devices:
    std::cout << ctx << std::endl;
}
```

One of the most convenient filters is `vex::Filter::Env` which selects compute devices based on environment variables. It allows to switch the compute device without the need to recompile the program.

Each stateful object in VexCL, like `vex::vector<T>`, takes an STL vector of `vex::backend::command_queue` instances. The `vex::Context` class is just a convenient way to initialize and hold the command queues. Since it provides the corresponding type conversion operator, it also may be used directly for object initialization:

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

But the users are not required to actually create a `vex::Context` instance. They may just use the command queues initialized elsewhere. In the following example the `Boost.Compute` is used as a backend and takes care of initializing the OpenCL context:

```
#include <iostream>
#include <boost/compute.hpp>

#define VEXCL_BACKEND_COMPUTE
#include <vexcl/vexcl.hpp>

int main() {
    boost::compute::command_queue bcq = boost::compute::system::default_queue();

    // Use Boost.Compute queue to allocate VexCL vectors:
    vex::vector<int> x({bcq}, 16);
}
```

Device filters

Common filters

These filters are supported for all backends:

- `vex::Filter::Any`. Selects all available devices.
- `vex::Filter::DoublePrecision`. Selects devices that support double precision arithmetics.
- `vex::Filter::Count(n)`. Selects first `n` devices that are passed through the filter. This filter should be the last in the filter chain. This will assure that it will be applied only to devices which passed all other filters. Otherwise, you could get less devices than planned (every time this filter is applied, its internal counter is decremented).
- `vex::Filter::Position(n)`. Selects single device at the given position.

- `vex::Filter::Env`. Selects devices with respect to environment variables. Recognized variables are:

<code>OCL_DEVICE</code>	Name of the device or its substring.
<code>OCL_MAX_DEVICES</code>	Maximum number of devices to select. The effect is similar to the <code>vex::Filter::Count</code> filter above.
<code>OCL_POSITION</code>	Single device with the specified position in the list of available devices. The effect is similar to the <code>vex::Filter::Position</code> filter above.
<code>OCL_PLATFORM</code>	OpenCL platform name or its substring. Only supported for OpenCL-based backends.
<code>OCL_VENDOR</code>	OpenCL device vendor name or its substring. Only supported for OpenCL-based backends.
<code>OCL_TYPE</code>	OpenCL device type. Possible values are CPU, GPU, ACCELERATOR. Only supported for OpenCL-based backends.
<code>OCL_EXTENSIONS</code>	OpenCL device supporting the specified extension. Only supported for OpenCL-based backends.

- `vex::Filter::Exclusive(filter)`. This is a filter wrapper that allows to obtain exclusive access to compute devices. This may be helpful if several compute devices are present in the system and several processes are trying to grab a single device. The exclusivity is only guaranteed between processes that use the `Exclusive` filter wrapper.

OpenCL-specific filters

These filters are only available for OpenCL and `Boost.Compute` backends:

- `vex::Filter::CLVersion(major, minor)`. Selects devices that support the specified version of OpenCL standard.
- `vex::Filter::Extension(string)`. Selects devices that provide the specified extension.
- `vex::Filter::GLSharing`. Selects devices that support OpenGL sharing extension. This is a shortcut for `vex::Filter::Extension("cl_khr_gl_sharing")`.
- `vex::Filter::Type(cl_device_type)`. Selects devices with the specified device type. The device type is a bit mask.
- `vex::Filter::GPU`. Selects GPU devices. This is a shortcut for `vex::Filter::Type(CL_DEVICE_TYPE_GPU)`.
- `vex::Filter::CPU`. Selects CPU devices. This is a shortcut for `vex::Filter::Type(CL_DEVICE_TYPE_CPU)`.
- `vex::Filter::Accelerator`. Selects Accelerator devices. This is a shortcut for `vex::Filter::Type(CL_DEVICE_TYPE_ACCELERATOR)`.

Custom filters

In case more complex functionality is required than provided by the builtin filters, the users may introduce their own functors:

```
// Select a GPU with more than 4GiB of global memory:
vex::Context ctx(vex::Filter::GPU &&
    [](const vex::backend::device &d) {
        size_t GiB = 1024 * 1024 * 1024;
        return d.getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() >= 4 * GiB;
    });
```

Reference

class `vex::Context`

VexCL context.

Holds vectors of `vex::backend::context` and `vex::backend::command_queue` instances.

Public Functions

template `<class DevFilter>`

Context (`DevFilter &&filter`, `vex::backend::command_queue_properties properties = 0`)

Initializes context from the device filter.

Context (`std::vector<vex::backend::context> c`, `std::vector<vex::backend::command_queue> q`)

Initializes context from the user-supplied vectors of `vex::backend::context` and `vex::backend::command_queues` instances.

const `std::vector<vex::backend::context> &context () const`

Returns reference to the vector of initialized `vex::backend::context` instances.

`vex::backend::context &context` (`unsigned d`)

Returns reference to the specified `vex::backend::context` instance.

const `std::vector<vex::backend::command_queue> &queue () const`

Returns reference to the vector of initialized `vex::backend::command_queue` instances.

operator const `std::vector<vex::backend::command_queue>& () const`

Returns reference to the vector of initialized `vex::backend::command_queue` instances.

const `vex::backend::command_queue &queue` (`unsigned d`) **const**

Returns reference to the specified `vex::backend::command_queue` instance.

`vex::backend::device device` (`unsigned d`) **const**

Returns reference to the specified `vex::backend::device` instance.

`size_t size () const`

Returns number of initialized devices.

`bool empty () const`

Checks if the context is empty.

operator bool `() const`

Checks if the context is empty.

`void finish () const`

Waits for completion of all command queues in the context.

template `<>`

`std::vector<vex::backend::device> vex::backend::device_list<DevFilter>` (`DevFilter &&filter`)

Returns vector of compute devices satisfying the given criteria without trying to initialize the contexts on the devices.

Managing memory

Allocating

The `vex::vector<T>` class constructor accepts a const reference to `std::vector<vex::backend::command_queue>`. A `vex::Context` instance may be conveniently converted to this type, but it is also possible to initialize the command queues elsewhere (e.g. with the OpenCL backend `vex::backend::command_queue` is typedefed to `cl::CommandQueue`), thus completely eliminating the need to create a `vex::Context`. Each command queue in the list should uniquely identify a single compute device.

The contents of the created vector will be partitioned across all devices that were present in the queue list. The size of each partition will be proportional to the device bandwidth, which is measured the first time the device is used. All vectors of the same size are guaranteed to be partitioned consistently, which minimizes inter-device communication.

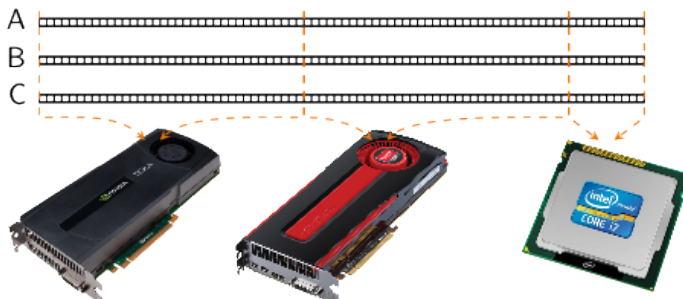
In the example below, three device vectors of the same size are allocated. Vector A is copied from the host vector a, and the other vectors are created uninitialized:

```
const size_t n = 1024 * 1024;
vex::Context ctx( vex::Filter::Any );

std::vector<double> a(n, 1.0);

vex::vector<double> A(ctx, a);
vex::vector<double> B(ctx, n);
vex::vector<double> C(ctx, n);
```

Assuming that the current system has an NVIDIA GPU, an AMD GPU, and an Intel CPU installed, possible partitioning may look like this:



```
template <typename T>
class vex::vector
    Device vector.

    Inherits from vex::vector_expression< Expr >
```

Public Functions

```
vector ()
    Empty constructor.
```

```
vector (const vector &v)
    Copy constructor.
```

```
vector (vector &&v)
    Move constructor.
```

vector (**const** backend::command_queue &*q*, **const** backend::device_vector<T> &*buffer*, size_t *size* = 0)

Wraps a native buffer without owning it.

May be used to apply VexCL functions to buffers allocated and managed outside of VexCL.

vector (**const** std::vector<backend::command_queue> &*queue*, size_t *size*, **const** T **host* = 0, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Creates vector of the given size and optionally copies host data.

vector (size_t *size*, **const** T **host* = 0, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Creates vector of the given size and optionally copies host data.

This version uses the most recently created VexCL context.

vector (**const** std::vector<backend::command_queue> &*queue*, **const** std::vector<T> &*host*, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Creates new device vector and copies the host vector.

vector (**const** std::vector<T> &*host*, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Creates new device vector and copies the host vector.

This version uses the most recently created VexCL context.

template <class Expr>

vector (**const** Expr &*expr*)

Constructs new vector from vector expression.

This will fail if VexCL is unable to automatically determine the expression size and the compute devices to use.

void **swap** (*vector* &*v*)

Swap function.

void **resize** (**const** *vector* &*v*, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Resizes the vector.

Borrows devices, size, and data from the given vector. Any data contained in the resized vector will be lost as a result.

void **resize** (**const** std::vector<backend::command_queue> &*queue*, size_t *size*, **const** T **host* = 0, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Resizes the vector with the given parameters.

This is equivalent to reconstructing the vector with the given parameters. Any data contained in the resized vector will be lost as a result.

void **resize** (**const** std::vector<backend::command_queue> &*queue*, **const** std::vector<T> &*host*, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Resizes the vector.

This is equivalent to reconstructing the vector with the given parameters. Any data contained in the resized vector will be lost as a result.

void **resize** (size_t *size*, **const** T **host* = 0, backend::mem_flags *flags* = backend::MEM_READ_WRITE)

Resizes the vector.

void **clear** ()

Fills vector with zeros.

This does not change the vector size!

const backend::device_vector<T> &**operator** () (unsigned *d* = 0) **const**
Returns memory buffer located on the given device.

backend::device_vector<T> &**operator** () (unsigned *d* = 0)
Returns memory buffer located on the given device.

const_iterator **begin** () **const**
Returns const iterator to the first element of the vector.

const_iterator **end** () **const**
Returns const iterator referring to the past-the-end element in the vector.

iterator **begin** ()
Returns iterator to the first element of the vector.

iterator **end** ()
Returns iterator referring to the past-the-end element in the vector.

const element **operator** [] (size_t *index*) **const**
Access vector element.

element **operator** [] (size_t *index*)
Access vector element.

size_t **size** () **const**
Returns vector size.

size_t **nparts** () **const**
Returns number of vector parts.
Each partition is located on single device.

size_t **part_size** (unsigned *d*) **const**
Returns vector part size on the given device.

size_t **part_start** (unsigned *d*) **const**
Returns index of the first element located on the given device.

const std::vector<backend::command_queue> &**queue_list** () **const**
Returns reference to the vector of command queues used to construct the vector.

backend::device_vector<T>::mapped_array **map** (unsigned *d* = 0)
Maps vector part located on the given device to a host array.
This returns a smart pointer that will be unmapped automatically upon destruction

backend::device_vector<T>::mapped_array **map** (unsigned *d* = 0) **const**
Maps vector part located on the given device to a host array.
This returns a smart pointer that will be unmapped automatically upon destruction

const *vector* &**operator**= (const *vector* &*x*)
Copy assignment.

const *vector* &**operator**= (*vector* &&*v*)
Move assignment.

template <class Expr>
auto **operator**= (const Expr &*expr*)
Expression assignment operator.

```
template <class Expr>
auto operator+=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator-=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator*=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator/=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator%=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator&=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator|=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator^=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator<<=(const Expr &expr)
    Expression assignment operator.

template <class Expr>
auto operator>>=(const Expr &expr)
    Expression assignment operator.

template <class vector_type, class element_type>
class iterator_type
    Inherits from boost::iterator_facade< iterator_type< vector_type, element_type >, T,
    std::random_access_iterator_tag, element_type >
```

Copying

The `vex::copy()` function allows to copy data between host and compute device memory spaces. There are two forms of the function – a simple one which accepts whole vectors, and an STL-like one, which accepts pairs of iterators:

```
std::vector<double> h(n);           // Host vector.
vex::vector<double> d(ctx, n);     // Device vector.

// Simple form:
vex::copy(h, d);                   // Copy data from host to device.
vex::copy(d, h);                   // Copy data from device to host.

// STL-like form:
vex::copy(h.begin(), h.end(), d.begin()); // Copy data from host to device.
vex::copy(d.begin(), d.end(), h.begin()); // Copy data from device to host.
```

The STL-like variant can copy sub-ranges of the vectors, or copy data from/to raw host pointers.

Vectors also overload the array subscript operator, `vex::vector::operator[]()`, so that users may directly read or write individual vector elements. This operation is highly ineffective and should be used with caution. Iterators allow for element access as well, so that STL algorithms may in principle be used with device vectors. This would be very slow but may be used as a temporary building block.

Another option for host-device data transfer is mapping device memory buffer to a host array. The mapped array then may be transparently read or written. The method `vex::vector::map()` maps the *d*-th partition of the vector and returns the mapped array:

```
vex::vector<double> X(ctx, N);
{
    auto mapped_ptr = X.map(0); // Unmapped automatically when goes out of scope
    for(size_t i = 0; i < X.part_size(0); ++i)
        mapped_ptr[i] = host_function(i);
}
```

Shared virtual memory

Both OpenCL 2.0 and CUDA 6.0 allow to share the same virtual address range between the host and the compute devices, so that there is no longer need to copy buffers between devices. In other words, no keeping track of buffers and explicitly copying them across devices! Just use shared pointers. OpenCL 2.0 calls this concept **Shared Virtual Memory (SVM)**, and CUDA 6.0 talks about **Unified Memory**. In VexCL, both of these are abstracted into `vex::svm_vector<T>` class.

The `vex::svm_vector<T>` constructor, as opposed to `vex::vector<T>`, takes single instance of `vex::backend::command_queue`. This is because the SVM vector has to be associated with a single device context. The SVM vectors in VexCL may be used in the same way normal vectors are used.

Example:

```
// Allocate SVM vector for the first device in context:
vex::svm_vector<int> x(ctx.queue(0), n);

// Fill the vector on the host.
{
    auto p = x.map(vex::backend::MAP_WRITE);
    for(int i = 0; i < n; ++i)
        p[i] = i * 2;
}
```

template <typename T>

class vex::svm_vector

Shared Virtual Memory wrapper class.

Inherits from `vex::vector_expression< Expr >`, `vex::vector_expression< Expr >`, `vex::vector_expression< Expr >`

Public Functions

svm_vector (const cl::CommandQueue &q, size_t n)

Allocates SVM vector on the given device.

size_t size () const

Returns size of the SVM vector.

const cl::CommandQueue &**queue** () **const**

Returns reference to the command queue associated with the SVM vector.

mapped_pointer **map** (cl_map_flags *map_flags* = CL_MAP_READ|CL_MAP_WRITE)

Returns host pointer ready to be either read or written by the host.

This returns a smart pointer that will be unmapped automatically upon destruction

const *svm_vector* &**operator=** (**const** *svm_vector* &*other*)

Copy assignment operator.

Vector expressions

VexCL allows the use of convenient and intuitive notation for vector operations. In order to be used in the same expression, all participating vectors have to be *compatible*:

- Have same size;
- Span same set of compute devices.

If these conditions are satisfied, then vectors may be combined with rich set of available expressions. Vector expressions are processed in parallel across all devices they were allocated on. Each vector expression results in the launch of a single compute kernel. The kernel is automatically generated and compiled the first time the expression is encountered in the program, and is submitted to command queues associated with the vector that is being assigned to.

VexCL will dump the sources of the generated kernels to stdout if either the `VEXCL_SHOW_KERNELS` preprocessor macro is defined, or there exists `VEXCL_SHOW_KERNELS` environment variable. For example, the expression:

```
X = 2 * Y - sin(Z);
```

will lead to the launch of the following compute kernel:

```
kernel void vexcl_vector_kernel(  
    ulong n,  
    global double * prm_1,  
    int prm_2,  
    global double * prm_3,  
    global double * prm_4  
)  
{  
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {  
        prm_1[idx] = ( ( prm_2 * prm_3[idx] ) - sin( prm_4[idx] ) );  
    }  
}
```

Here and in the rest of examples `X`, `Y`, and `Z` are compatible instances of `vex::vector<double>`; it is also assumed that OpenCL backend is selected.

VexCL is able to cache the compiled kernels offline. The compiled binaries are stored in `$HOME/.vexcl` on Linux and MacOSX, and in `%APPDATA%\vexcl` on Windows systems. In order to enable this functionality for OpenCL-based backends, the user has to define the `VEXCL_CACHE_KERNELS` preprocessor macro. NVIDIA OpenCL implementation does the caching already, but on AMD or Intel platforms this may lead to dramatic decrease of program initialization time (e.g. VexCL tests take around 20 seconds to complete without kernel caches, and 2 seconds when caches are available). In case of the CUDA backend the offline caching is always enabled.

VEXCL_SHOW_KERNELS

When defined, VexCL will dump source code of the generated kernels to stdout. Same effect may be achieved by exporting an environment variable with the same name.

VEXCL_CACHE_KERNELS

When defined, VexCL will use offline cache to store the compiled kernels. The first time a kernel is compiled on the system, its binaries are saved to the cache folder (\$HOME/.vexcl on Unix-like systems; %APPDATA%\vexcl on Windows). Next time the program is run, the binaries will be obtained from the cache, thus speeding up the program startup.

Builtin operations

VexCL expressions may combine device vectors and scalars with arithmetic, logic, or bitwise operators as well as with builtin OpenCL/CUDA functions. If some builtin operator or function is unavailable, it should be considered a bug. Please do not hesitate to open an issue in this case.

```
Z = sqrt(2 * X) + pow(cos(Y), 2.0);
```

Constants

As you have seen above, 2 in the expression $2 * Y - \sin(Z)$ is passed to the generated compute kernel as an `int` parameter (`prm_2`). Sometimes this is desired behaviour, because the same kernel will be reused for the expressions $42 * Z - \sin(Y)$ or $a * Y - \sin(Y)$ (where a is an integer variable). But this may lead to a slight overhead if an expression involves true constant that will always have same value. The `VEX_CONSTANT` macro allows one to define such constants for use in vector expressions. Compare the generated kernel for the following example with the kernel above:

```
VEX_CONSTANT(two, 2);
X = two() * Y - sin(Z);
```

```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * prm_1,
    global double * prm_3,
    global double * prm_4
)
{
    for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        prm_1[idx] = ( ( ( 2 ) * prm_3[idx] ) - sin( prm_4[idx] ) );
    }
}
```

VexCL provides some predefined constants in the `vex::constants` namespace that correspond to `boost::math::constants` (e.g. `vex::constants::pi()`).

```
VEX_CONSTANT(name, value) struct constant_##name { \
    typedef decltype(value) value_type; \
    static std::string get() { \
        static const value_type v = value; \
        std::ostringstream s; \
        s << "(" << std::scientific << std::setprecision(16) << v << ")"; \
        return s.str(); \
    } \
    decltype(boost::proto::as_expr<vex::vector_domain>( \
        vex::user_constant<constant_##name>())) \
    operator()() const { \
        return boost::proto::as_expr<vex::vector_domain>( \
            vex::user_constant<constant_##name>()); \
    } \
    operator value_type() const { \
        static const value_type v = value; \
        return v; \
    } \
    const constant_##name name = {} \
}
```

Creates user-defined constan functor for use in VexCL expressions. `value` will be copied verbatim into kernel source.

Element indices

The function `vex::element_index()` allows one to use the index of each vector element inside vector expressions. The numbering is continuous across all compute devices and starts with an optional `offset`.

```
// Linear function:
double x0 = 0.0, dx = 1.0 / (N - 1);
X = x0 + dx * vex::element_index();

// Single period of sine function:
Y = sin(vex::constants::two_pi() * vex::element_index() / N);
```

auto `vex::element_index` (size_t `offset` = 0, size_t `length` = 0)

Returns index of the current element index with optional `offset`. Optional `length` parameter may be used to provide the size information to the resulting expression. This could be useful when reducing stateless expressions.

User-defined functions

Users may define custom functions for use in vector expressions. One has to define the function signature and the function body. The body may contain any number of lines of valid OpenCL or CUDA code, depending on the selected backend. The most convenient way to define a function is via the `VEX_FUNCTION` macro:

```
VEX_FUNCTION(double, squared_radius, (double, x)(double, y),
    return x * x + y * y;
);
Z = sqrt(squared_radius(X, Y));
```

The first macro parameter here defines the function return type, the second parameter is the function name, the third parameter defines function arguments in form of a preprocessor sequence. Each element of the sequence is a tuple of argument type and name. The rest of the macro is the function body (compare this with how functions are defined in C/C++). The resulting `squared_radius` function object is stateless; only its type is used for kernel generation. Hence, it is safe to define commonly used functions at the global scope.

Note that any valid vector expression may be passed as a function parameter, including nested function calls:

```
Z = squared_radius(sin(X + Y), cos(X - Y));
```

Another version of the macro takes the function body directly as a string:

```
VEX_FUNCTION_S(double, squared_radius, (double, x)(double, y),
    "return x * x + y * y;"
);
Z = sqrt(squared_radius(X, Y));
```

In case the function that is being defined calls other custom function inside its body, one can use the version of the `VEX_FUNCTION` macro that takes sequence of parent function names as the fourth parameter. This way the kernel generator will know to include the function definitions into the kernel source:

```
VEX_FUNCTION(double, bar, (double, x),
    double s = sin(x);
    return s * s;
);
VEX_FUNCTION(double, baz, (double, x),
    double c = cos(x);
    return c * c;
```

```

    );
VEX_FUNCTION_D(double, foo, (double, x)(double, y), (bar)(baz),
    return bar(x - y) * baz(x + y);
);

```

Similarly to `VEX_FUNCTION_S`, there is a version called `VEX_FUNCTION_DS` (or symmetrical `VEX_FUNCTION_SD`) that takes the function body as a string parameter.

Custom functions may be used not only for convenience, but also for performance reasons. The above example with `squared_radius` could in principle be rewritten as:

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

The drawback of this version is that `X` and `Y` will be passed to the kernel and read *twice* (see the next section for an explanation).

VEX_FUNCTION (return_type, name, arguments, ...) `VEX_FUNCTION_S`(return_type, name, arguments, `VEX_STRINGIZE_SOURCE`(__VA_ARGS__))
Creates a user-defined function.

The body of the function is specified as unquoted C source at the end of the macro. The source will be stringized with `VEX_STRINGIZE_SOURCE` macro.

VEX_FUNCTION_S (return_type, name, arguments, body) `VEX_FUNCTION_SD`(return_type, name, arguments, , body)
Creates a user-defined function.

The body of the function is passed as a string literal or a static string expression.

VEX_FUNCTION_D (return_type, name, arguments, dependencies, ...) `VEX_FUNCTION_SD`(return_type, name, arguments, dependencies, `VEX_STRINGIZE_SOURCE`(__VA_ARGS__))
Creates a user-defined function with dependencies.

The body of the function is specified as unquoted C source at the end of the macro. The source will be stringized with `VEX_STRINGIZE_SOURCE` macro.

VEX_FUNCTION_SD (return_type, name, arguments, dependencies, body)
`VEX_FUNCTION_SINK`(return_type, name, \BOOST_PP_SEQ_SIZE(VEXCL_FUNCTION_MAKE_SEQ(a
\ VEXCL_FUNCTION_MAKE_SEQ(arguments), dependencies, body)
Creates a user-defined function with dependencies.

The body of the function is passed as a string literal or a static string expression.

VEX_STRINGIZE_SOURCE (...) #__VA_ARGS__
Converts an unquoted text into a string literal.

Tagged terminals

The last example in the previous section is ineffective because the compiler cannot tell if any two terminals in an expression tree are actually referring to the same data. But programmers often have this information. VexCL allows one to pass this knowledge to compiler by tagging terminals with unique tags. By doing this, the programmer guarantees that any two terminals with matching tags are referencing the same data.

Below is a more effective variant of the above example:

```

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

```

Here, the generated kernel will have one parameter for each of the vectors `X` and `Y`:

```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * prm_1,
    global double * prm_tag_1_1,
    global double * prm_tag_2_1
)
{
    for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        prm_1[idx] = sqrt( ( ( prm_tag_1_1[idx] * prm_tag_1_1[idx] )
                             + ( prm_tag_2_1[idx] * prm_tag_2_1[idx] ) ) );
    }
}
```

template <size_t Tag, class Expr>

auto vex::tag(const Expr &expr)

Tags terminal with a unique (in a single expression) tag.

By tagging terminals user guarantees that the terminals with same tags actually refer to the same data. VexCL is able to use this information in order to reduce number of kernel parameters and unnecessary global memory I/O operations.

Temporary values

Some expressions may have several occurrences of the same subexpression. Unfortunately, VexCL is not able to determine these cases without the programmer's help. For example, let us consider the following expression:

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

Here, $\log(X)$ would be computed twice. One could tag vector X as in:

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

and hope that the backend compiler is smart enough to reuse result of $\log(x)$. In fact, most modern compilers will in this simple case. But in harder cases it is possible to explicitly tell VexCL to store the result of a subexpression in a local variable and reuse it. The `vex::make_temp<size_t>()` function template serves this purpose:

```
auto tmp1 = vex::make_temp<1>( sin(X) );
auto tmp2 = vex::make_temp<2>( cos(X) );
Y = (tmp1 - tmp2) * (tmp1 + tmp2);
```

This will result in the following kernel:

```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * prm_1,
    global double * prm_2_1,
    global double * prm_3_1
)
{
    for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0))
    {
        double temp_1 = sin( prm_2_1[idx] );
        double temp_2 = cos( prm_3_1[idx] );
        prm_1[idx] = ( ( temp_1 - temp_2 ) * ( temp_1 + temp_2 ) );
    }
}
```

Any valid vector or multivector expression (but not additive expressions, such as sparse matrix-vector products) may be wrapped into a `vex::make_temp()` call.

template <size_t Tag, typename T, class Expr>

auto vex::make_temp(const Expr &expr)

Creates temporary expression that may be reused in a vector expression.

The type of the resulting temporary variable is automatically deduced from the expression, but may also be explicitly specified as a template parameter.

Raw pointers¹

Most of the expressions in VexCL are element-wise. That is, the user describes what needs to be done on an element-by-element basis, and has no access to neighboring elements. `vex::raw_pointer()` allows to use pointer arithmetic with either `vex::vector<T>` or `vex::svm_vector<T>`.

The N -body problem

Let us consider the N -body problem as an example. The N -body problem considers N point masses, m_i , $i = 1, 2, \dots, N$ in three dimensional space \mathbb{R}^3 moving under the influence of mutual gravitational attraction. Each mass m_i has a position vector \vec{q}_i . Newton's law of gravity says that the gravitational force felt on mass m_i by a single mass m_j is given by

$$\vec{F}_{ij} = \frac{Gm_im_j(\vec{q}_j - \vec{q}_i)}{\|\vec{q}_j - \vec{q}_i\|^3},$$

where G is the gravitational constant and $\|\vec{q}_j - \vec{q}_i\|$ is the distance between \vec{q}_i and \vec{q}_j .

We can find the total force acting on mass m_i by summing over all masses:

$$\vec{F}_i = \sum_{j=1, j \neq i}^N \frac{Gm_im_j(\vec{q}_j - \vec{q}_i)}{\|\vec{q}_j - \vec{q}_i\|^3}.$$

In VexCL, we can encode the formula above with the following custom function:

```
vex::vector<double>      m(ctx, n);
vex::vector<cl_double3> q(ctx, n), f(ctx, n);

VEX_FUNCTION(cl_double3, force, (size_t, n) (size_t, i) (double*, m) (cl_double3*, q),
  const double G = 6.674e-11;

  double3 sum = {0.0, 0.0, 0.0};
  double m_i = m[i];
  double3 q_i = q[i];

  for(size_t j = 0; j < n; ++j) {
    if (j == i) continue;

    double m_j = m[j];
    double3 d = q[j] - q_i;
    double r = length(d);

    sum += G * m_i * m_j * d / (r * r * r);
```

¹ This operation involves access to arbitrary elements of its subexpressions and may lead to unpredictable device-to-device communication. Hence, it is restricted to single-device expressions. That is, only vectors that are located on a single device are allowed to participate in this operation.

```
    }  
    return sum;  
};  
  
f = force(n, vex::element_index(), vex::raw_pointer(m), vex::raw_pointer(q));
```

The function takes number of elements `n`, index of the current element `i`, and pointers to arrays of point masses `m` and positions `q`. It returns the force acting on the current point. Note that we use host-side types (`cl_double3`) in declaration of function return type and parameter types, and we use OpenCL types (`double3`) inside the function body.

Constant address space

In the OpenCL-based backends VexCL allows one to use constant cache on GPUs in order to speed up the read-only access to small vectors. Usually around 64Kb of constant cache per compute unit is available. Vectors wrapped in `vex::constant()` will be decorated with the `constant` keyword instead of the usual `global` one. For example, the following expression:

```
x = 2 * vex::constant(y);
```

will result in the OpenCL kernel below:

```
kernel void vexcl_vector_kernel(  
    ulong n,  
    global int * prm_1,  
    int prm_2,  
    constant int * prm_3  
)  
{  
    for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0)) {  
        prm_1[idx] = ( prm_2 * prm_3[idx] );  
    }  
}
```

In cases where access to arbitrary vector elements is required, `vex::constant_pointer()` may be used similarly to `vex::raw_pointer()`. The extracted pointer will be decorated with the `constant` keyword.

template <typename T>

auto vex::raw_pointer(const vector<T> &v)

Cast `vex::vector` to a raw pointer.

template <typename T>

auto vex::raw_pointer(const svm_vector<T> &v)

Cast `vex::svm_vector` to a raw pointer.

template <class T>

constant_vector<T> vex::constant(const vector<T> &v)

Uses constant cache for access to the wrapped vector.

Note: Only available for OpenCL-based backends.

template <typename T>

auto vex::constant_pointer(const vector<T> &v)

Cast `vex::vector` to a constant pointer.

Note: Only available for OpenCL-based backends.

Random number generation

VexCL provides a counter-based random number generators from [Random123](#) suite, in which N-th random number is obtained by applying a stateless mixing function to N instead of the conventional approach of using N iterations of a stateful transformation. This technique is easily parallelizable and is well suited for use in GPGPU applications.

For integral types, the generated values span the complete range; for floating point types, the generated values lie in the interval [0,1].

In order to use a random number sequence in a vector expression, the user has to declare an instance of either `vex::Random` or `vex::RandomNormal` class template as in the following example:

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

// X will contain random numbers from [-1, 1]:
X = 2 * rnd(vex::element_index(), std::rand()) - 1;
```

Note that `vex::element_index()` function here provides the random number generator with a sequence position N, and `std::rand()` is used to obtain a seed for this specific sequence.

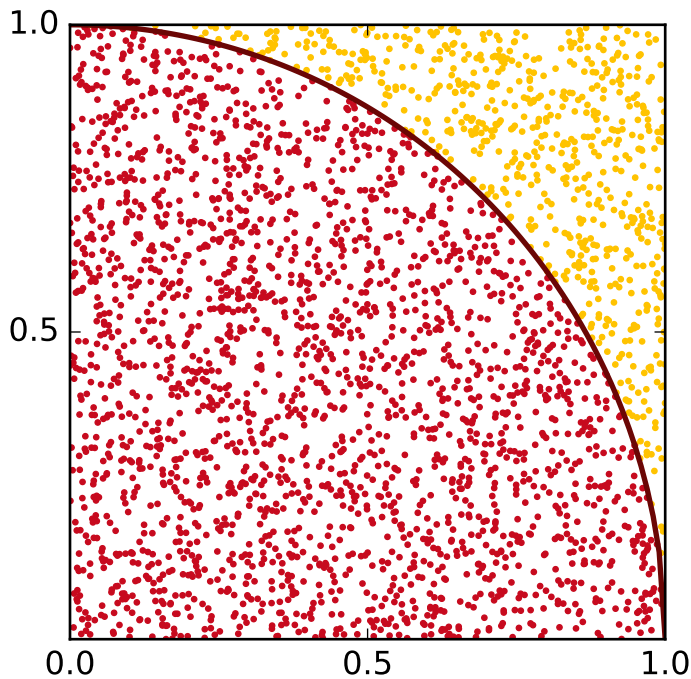
Monte Carlo π

Here is a more interesting example of using random numbers to estimate the value of π . In order to do this we remember that area of a circle with radius r is equal to πr^2 . A square of the same ‘radius’ has area of $(2r)^2$. Then we can write

$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4},$$

$$\pi = 4 \frac{\text{area of circle}}{\text{area of square}}$$

We can estimate the last fraction in the formula above with the Monte-Carlo method. If we generate a lot of random points in a square, then ratio of circle area over square area will be approximately equal to the ratio of points in the circle over all points. This is illustrated by the following figure:



In VexCL we can compute the estimate with a single expression, that will generate single compute-bound kernel:

```
vex::Random<cl_double2> rnd;
vex::Reductor<size_t> sum(ctx);

double pi = sum(length(rnd(vex::element_index(0, n), std::rand())) < 1) * 4.0 / n;
```

Here we generate n random 2D points and use the builtin OpenCL function `length` to see which points are located within the circle. Then we use the `sum` function to count the points within the circle and finally multiply the number with $4.0/n$ to get the estimated value of π .

template <class T, class Generator = random::philox>

struct `vex::Random`

Returns uniformly distributed random numbers.

For integral types, generated values span the complete range.

For floating point types, generated values are in $[0, 1]$.

Uses Random123 generators which provide 64(2x32), 128(4x32, 2x64) and 256(4x64) random bits, this limits the supported output types, which means `cl_double8` (512bit) is not supported, but `cl_uchar2` is.

Supported generator families are `random::philox` (based on integer multiplication, default) and `random::threefry` (based on the Threefish encryption function). Both satisfy rigorous statistical testing (passing BigCrush in TestU01), vectorize and parallelize well (each generator can produce at least 2^{64} independent streams), have long periods (the period of each stream is at least 2^{128}), require little or no memory or state, and have excellent performance (a few clock cycles per byte of random output).

Inherits from `vex::UserFunction< Random< T, Generator >, T(cl_ulong, cl_ulong)>`

template <class T, class Generator = random::philox>

struct `vex::RandomNormal`

Returns normally distributed random numbers.

Uses Box-Muller transform.

Inherits from `vex::UserFunction< RandomNormal< T, Generator >, T(cl_ulong, cl_ulong)>`

Permutations¹

`vex::permutation()` allows the use of a permuted vector in a vector expression. The function accepts a vector expression that returns integral values (indices). The following example reverses *X* and assigns it to *Y* in two different ways:

```
Y = vex::permutation(n - 1 - vex::element_index()) (X);

// Permutation expressions are writable!
vex::permutation(n - 1 - vex::element_index()) (Y) = X;
```

template <class Expr>

auto vex::permutation(const Expr &expr)

Returns permutation functor which is based on an integral expression.

Slicing¹

An instance of the `vex::slicer<NDim>` class allows one to conveniently access sub-blocks of multi-dimensional arrays that are stored in `vex::vector<T>` in row-major order. The constructor of the class accepts the dimensions of the array to be sliced. The following example extracts every other element from interval [100, 200) of a one-dimensional vector *X*:

```
vex::vector<double> X(ctx, n);
vex::vector<double> Y(ctx, 50);

vex::slicer<1> slice(vex::extents[n]);

Y = slice[vex::range(100, 2, 200)] (X);
```

And the example below shows how to work with a two-dimensional matrix:

```
using vex::range, vex::_; // vex::_ is a shortcut for an empty range

vex::vector<double> X(ctx, n * n); // n-by-n matrix stored in row-major order.
vex::vector<double> Y(ctx, n);

// vex::extents is a helper object similar to boost::multi_array::extents.
vex::slicer<2> slice(vex::extents[n][n]);

Y = slice[42] (X); // Put 42-nd row of X into Y.
Y = slice[_][42] (X); // Put 42-nd column of X into Y.

slice[_][10] (X) = Y; // Put Y into 10-th column of X.

// Assign sub-block [10,20)x[30,40) of X to Z:
vex::vector<double> Z = slice[range(10, 20)][range(30, 40)] (X);
assert(Z.size() == 100);
```

template <size_t NR>

struct vex::slicer

Slicing operator.

Provides information about shape of multidimensional vector expressions, allows to slice the expressions.

Public Functions

template <typename T>
slicer (const std::array<T, NR> &target_dimensions)
Creates slicer with the given dimensions.

template <typename T>
slicer (const T *target_dimensions)
Creates slicer with the given dimensions.

slicer (const extent_gen<NR> &ext)
Creates slicer with the given dimensions.

template <size_t C>
struct slice
Inherits from vex::gslice< NR >

const extent_gen<0> vex::extents
Helper object for specifying slicer dimensions.

struct vex::range
An index range for use with slicer class.

Public Functions

range ()
Unbounded range (all elements along the current dimension).

range (size_t i)
Range with a single element.

range (size_t start, ptrdiff_t stride, size_t stop)
Elements from open interval with given stride.

range (size_t start, size_t stop)
Every element from open interval.

const range vex::_
Placeholder for an unbounded range.

Reducing multidimensional expressions¹

`vex::reduce()` function allows one to reduce a multidimensional expression along its one or more dimensions. The result is again a vector expression, which may be used in other expressions. The supported reduction operations are `vex::SUM`, `vex::MIN`, and `vex::MAX`. The function takes three arguments: the shape of the expression to reduce (with the slowest changing dimension in the front), the expression to reduce, and the dimension(s) to reduce along. The latter are specified as indices into the shape array. Both the shape and indices are specified as static arrays of integers, but `vex::extents` object may be used for convenience.

In the following example we find maximum absolute value of each row in a two-dimensional matrix and assign the result to a vector:

```
vex::vector<double> A(ctx, N * M);  
vex::vector<double> x(ctx, N);  
  
x = vex::reduce<vex::MAX>(vex::extents[N][M], fabs(A), vex::extents[1]);
```

It is also possible to use `vex::slicer` instance to provide information about the expression shape:

```
vex::slicer<2> Adim(vex::extents[N][M]);
x = vex::reduce<vex::MAX>(Adim[_](fabs(A)), vex::extents[1]);
```

template <class Reducer, class SlicedExpr, class ReduceDims>

auto vex::reduce (const SlicedExpr &expr, const ReduceDims &reduce_dims)

Reduce multidimensional vector expression along specified dimensions.

template <class Reducer, class ExprShape, class Expr, class ReduceDims>

auto vex::reduce (const ExprShape &shape, const Expr &expr, const ReduceDims &reduce_dims)

Reduce multidimensional vector expression along specified dimensions.

Reshaping¹

`vex::reshape()` function is a powerful primitive that allows one to conveniently manipulate multidimensional data. It takes three arguments – an arbitrary vector expression to reshape, the dimensions `dst_dims` of the final result (with the slowest changing dimension in the front), and the native dimensions of the expression, which are specified as indices into `dst_dims`. The function returns a vector expression. The dimensions may be conveniently specified with the help of `vex::extents` object.

Here is an example that shows how a two-dimensional matrix of size $N \times M$ could be transposed:

```
vex::vector<double> A(ctx, N * M);
vex::vector<double> B = vex::reshape(A,
                                     vex::extents[M][N], // new shape
                                     vex::extents[1][0]  // A is shaped as [N][M]
                                     );
```

If the source expression lacks some of the destination dimensions, then those will be introduced by replicating the available data. For example, to make a two-dimensional matrix from a one-dimensional vector by copying the vector to each row of the matrix, one could do the following:

```
vex::vector<double> x(ctx, N);
vex::vector<double> y(ctx, M);
vex::vector<double> A(ctx, M * N);

// Copy x into each row of A:
A = vex::reshape(x, vex::extents[M][N], vex::extents[1]);
// Now, copy y into each column of A:
A = vex::reshape(y, vex::extents[M][N], vex::extents[0]);
```

Here is a more realistic example of a dense matrix-matrix multiplication. Elements of a matrix product $C = AB$ are defined as $C_{ij} = \sum_k A_{ik} B_{kj}$. Let's assume that matrix A has shape $N \times L$, and matrix B is shaped as $L \times M$. Then matrix C has dimensions $N \times M$. In order to implement the multiplication we extend matrices A and B to the shape of $N \times L \times M$, multiply the resulting expressions elementwise, and reduce the product along the middle dimension (L):

```
vex::vector<double> A(ctx, N * L);
vex::vector<double> B(ctx, L * M);
vex::vector<double> C(ctx, N * M);

C = vex::reduce<vex::SUM>(
    vex::extents[N][L][M],
    vex::reshape(A, vex::extents[N][L][M], vex::extents[0][1]) *
    vex::reshape(B, vex::extents[N][L][M], vex::extents[1][2]),
    1
);
```

This of course would not be as efficient as a carefully crafted custom implementation or a call to a vendor BLAS function. Also, this particular operation is more efficiently done with tensor product function described in the next section.

template <class Expr, class DstDims, class SrcDims>

auto vex::reshape (const Expr &expr, const DstDims &dst_dims, const SrcDims &src_dims)

Reshapes the expression.

Makes a multidimensional expression shaped as dst_dims from an input expression shaped as dst_dims[src_dims]. src_dims are specified as indices into dst_dims.

Tensor product¹

Given two tensors (arrays of dimension greater than or equal to one), A and B, and a list of axes pairs (where each pair represents corresponding axes from each of the two tensors), the tensor product operation sums the products of A's and B's elements over the given axes. In VexCL this is implemented as `vex::tensordot()` operation (compare with python's `numpy.tensordot`).

For example, the above matrix-matrix product may be implemented much more efficiently with `vex::tensordot()`:

```
using vex::_;

vex::slicer<2> Adim(vex::extents[N][M]);
vex::slicer<2> Bdim(vex::extents[M][L]);

C = vex::tensordot(Adim[_](A), Bdim[_](B), vex::axes_pairs(1, 0));
```

Here instances of `vex::slicer` class are used to provide shape information for the A and B vectors.

template <class SlicedExpr1, size_t SlicedExpr2, size_t CDIM>

auto vex::tensordot (const SlicedExpr1 &lhs, const SlicedExpr2 &rhs, const
std::array<std::array<size_t, 2>, CDIM> &common_axes)

Tensor dot product along specified axes for multidimensional arrays.

template <class... Args>

std::array<std::array<size_t, 2>, sizeof...(Args) / 2> vex::axes_pairs (Args... args)

Helper function for creating axes pairs.

Example:

```
auto axes = axes_pairs(a0, b0, a1, b1);
assert(axes[0][0] == a0 && axes[0][1] == b0);
assert(axes[1][0] == a1 && axes[1][1] == b1);
```

Scattered data interpolation with multilevel B-Splines

VexCL provides an implementation of the MBA algorithm based on paper by Lee, Wolberg, and Shin [LeWS97]. This is a fast algorithm for scattered N-dimensional data interpolation and approximation. Multilevel B-splines are used to compute a C2-continuously differentiable surface through a set of irregularly spaced points. The algorithm makes use of a coarse-to-fine hierarchy of control lattices to generate a sequence of bicubic B-spline functions whose sum approaches the desired interpolation function. Large performance gains are realized by using B-spline refinement to reduce the sum of these functions into one equivalent B-spline function. High-fidelity reconstruction is possible from a selected set of sparse and irregular samples.

The algorithm is setup on a CPU. After that, it may be used in vector expressions. Here is an example in 2D:

```

// Coordinates of data points:
std::vector< std::array<double, 2> > coords = {
    {0.0, 0.0},
    {0.0, 1.0},
    {1.0, 0.0},
    {1.0, 1.0},
    {0.4, 0.4},
    {0.6, 0.6}
};

// Data values:
std::vector<double> values = {
    0.2, 0.0, 0.0, -0.2, -1.0, 1.0
};

// Bounding box:
std::array<double, 2> xmin = {-0.01, -0.01};
std::array<double, 2> xmax = { 1.01, 1.01};

// Initial grid size:
std::array<size_t, 2> grid = {5, 5};

// Algorithm setup.
vex::mba<2> surf(ctx, xmin, xmax, coords, values, grid);

// x and y are coordinates of arbitrary 2D points
// (here the points are placed on a regular grid):
vex::vector<double> x(ctx, n*n), y(ctx, n*n), z(ctx, n*n);

auto I = vex::element_index() % n;
auto J = vex::element_index() / n;
vex::tie(x, y) = std::make_tuple(h * I, h * J);

// Get interpolated values:
z = surf(x, y);

```

template <size_t NDIM, typename real = double>

class vex::mba

Scattered data interpolation with multilevel B-Splines.

Public Functions

mba (const std::vector<backend::command_queue> &queue, const point &cmin, const point &cmax, const std::vector<point> &coo, std::vector<real> val, std::array<size_t, NDIM> grid, size_t levels = 8, real tol = 1e-8)

Creates the approximation functor. cmin and cmax specify the domain boundaries, coo and val contain coordinates and values of the data points. grid is the initial control grid size. The approximation hierarchy will have at most levels and will stop when the desired approximation precision tol will be reached.

template <class Cooler, class Vallter>

mba (const std::vector<backend::command_queue> &queue, const point &cmin, const point &cmax, Cooler coo_begin, Cooler coo_end, Vallter val_begin, std::array<size_t, NDIM> grid, size_t levels = 8, real tol = 1e-8)

Creates the approximation functor. cmin and cmax specify the domain boundaries. Coordinates and values of the data points are passed as iterator ranges. grid is the initial control grid size. The approximation hierarchy will have at most levels and will stop when the desired approximation precision tol will be reached.

```
template <class... Expr>
auto operator () (const Expr&... expr) const
    Provide interpolated values at given coordinates.
```

Fast Fourier Transform¹

VexCL provides an implementation of the Fast Fourier Transform (FFT) that accepts arbitrary vector expressions as input, allows one to perform multidimensional transforms (of any number of dimensions), and supports arbitrary sized vectors:

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

vex::vector<double> rhs(ctx, n), u(ctx, n), K(ctx, n);

// Solve Poisson equation with FFT:
u = ifft( K * fft(rhs) );
```

```
template <typename Tin, typename Tout = Tin, class Planner = fft::planner>
struct vex::FFT
```

Fast Fourier Transform.

FFT always works with complex types (`cl_double2` or `cl_float2`) internally. When the input is specified as real (float or double), it is extended to the complex plane (by setting the imaginary part to zero). When user asks the output to be real, the complex values are truncated by dropping the imaginary part.

Usage:

```
FFT<cl_double2> fft(ctx, length);
output = fft(input); // out-of-place transform
data = fft(data);    // in-place transform
FFT<cl_double2> ifft({width, height}, fft::inverse); // implicit context
input = ifft(output); // backward transform
```

To batch multiple transformations, use `fft::none` as the first kind:

```
FFT<cl_double2> fft({batch, n}, {fft::none, fft::forward});
output = fft(input);
```

Public Functions

FFT (**const** std::vector<backend::command_queue> &queues, size_t length, *fft::direction* dir = `fft::forward`, **const** Planner &planner = Planner())
1D constructor

FFT (**const** std::vector<backend::command_queue> &queues, **const** std::vector<size_t> &lengths, *fft::direction* dir = `fft::forward`, **const** Planner &planner = Planner())
N-dimensional constructor.

FFT (**const** std::vector<size_t> &lengths, *fft::direction* dir = `fft::forward`, **const** Planner &planner = Planner())
N-dimensional constructor.

FFT (**const** std::vector<backend::command_queue> &queues, **const** std::vector<size_t> &lengths, **const** std::vector<*fft::direction*> &dires, **const** Planner &planner = Planner())
N-dimensional constructor.

FFT (**const** std::vector<size_t> &lengths, **const** std::vector<fft::direction> &dirs, **const** Planner &planner = Planner())
N-dimensional constructor.

FFT (**const** std::vector<backend::command_queue> &queues, **const** std::initializer_list<size_t> &lengths, fft::direction dir = fft::forward, **const** Planner &planner = Planner())
N-dimensional constructor.

FFT (**const** std::initializer_list<size_t> &lengths, fft::direction dir = fft::forward, **const** Planner &planner = Planner())
N-dimensional constructor.

FFT (**const** std::vector<backend::command_queue> &queues, **const** std::initializer_list<size_t> &lengths, **const** std::initializer_list<fft::direction> &dirs, **const** Planner &planner = Planner())
N-dimensional constructor.

FFT (**const** std::initializer_list<size_t> &lengths, **const** std::initializer_list<fft::direction> &dirs, **const** Planner &planner = Planner())
N-dimensional constructor.

template <class Expr>
auto **operator** () (**const** Expr &x)
Performs the transform.

enum vex::fft::direction
FFT direction.

Values:

forward
Forward transform.

inverse
Inverse transform.

none
Specifies dimension(s) to do batch transform.

Parallel primitives and algorithms

Reductions

An instance of `vex::Reductor<T, ReduceOP=vex::SUM>` allows one to reduce an arbitrary vector expression to a single value of type T. Supported reduction operations are `vex::SUM`, `vex::MIN`, and `vex::MAX`. Reductor objects are stateful – they keep small temporary buffers on compute devices and receive a list of command queues at construction.

In the following example an inner product of two vectors is computed:

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

Also, see [Random number generation](#) for an example of estimating value of π with the Monte Carlo method.

Reduce operations may be combined with the `vex::CombineReducers` class. This way several reduction operations will be fused into single compute kernel. The operations should return the same scalar type, and the result of the combined reduction operation will be appropriately sized OpenCL/CUDA vector type.

In the following example minimum and maximum values of the vector are computed at the same time:

```
vex::Reductor<double, vex::CombineReducers<vex::MIN, vex::MAX>> minmax(ctx);
cl_double2 m = minmax(x);
std::cout << "min(x) = " << m.s[0] << std::endl;
std::cout << "max(x) = " << m.s[1] << std::endl;
```

In fact, the operation is so common, that VexCL provides a convenience typedef `vex::MIN_MAX`.

template <typename ScalarType, class RDC = *SUM*>

class vex::Reductor

Parallel reduction of arbitrary expression.

Reduction uses small temporary buffer on each device present in the queue parameter. One *Reductor* class for each reduction kind is enough per thread of execution.

Public Functions

Reductor (const std::vector<backend::command_queue> &queue = current_context().queue())

Constructor.

template <class Expr>

auto **operator**() (const Expr &expr) const

Compute reduction of a vector expression.

template <class Expr>

std::array<result_type, N> **operator**() (const Expr &expr) const

Compute reduction of a multivector expression.

struct vex::SUM

Summation.

Subclassed by vex::SUM_Kahan

struct vex::MIN

Minimum element.

struct vex::MAX

Maximum element.

template <class... R>

struct vex::CombineReducers

Combines several reduce operations.

typedef *CombineReducers*<MIN, MAX> vex::MIN_MAX

Combined MIN and MAX operation.

Sparse matrix-vector products

One of the most common operations in linear algebra is the matrix-vector product. An instance of `vex::SpMat` class holds a representation of a sparse matrix. Its constructor accepts a sparse matrix in common CRS format. In the example below a `vex::SpMat` is constructed from an *Eigen* sparse matrix:

```
Eigen::SparseMatrix<double, Eigen::RowMajor, int> E;
vex::SpMat<double, int> A(ctx, E.rows(), E.cols(),
    E.outerIndexPtr(), E.innerIndexPtr(), E.valuesPtr());
```

Matrix-vector products may be used in vector expressions. The only restriction is that the expressions have to be additive. This is due to the fact that the operation involves inter-device communication for multi-device contexts.


```
// Compute residual value for a system of linear equations:
Z = Y - A * X;
```

This restriction may be lifted for single-device contexts. In this case VexCL does not need to worry about inter-device communication. Hence, it is possible to inline matrix-vector product into a normal vector expression with the help of `vex::make_inline()`:

```
residual = sum(Y - vex::make_inline(A * X));
Z = sin(vex::make_inline(A * X));
```

template <typename val_t, typename col_t = size_t, typename idx_t = size_t>

class vex::SpMat

Sparse matrix in hybrid ELL-CSR format.

Public Functions

SpMat()

Empty constructor.

SpMat(const std::vector<backend::command_queue> &queue, size_t n, size_t m, const idx_t *row, const col_t *col, const val_t *val)

Constructor.

Constructs GPU representation of the $n \times m$ matrix. Input matrix is in CSR format. GPU matrix utilizes ELL format and is split equally across all compute devices.

size_t rows() const

Number of rows.

size_t cols() const

Number of columns.

size_t nonzeros() const

Number of non-zero entries.

template <class MVProdExpr>

auto vex::make_inline(const MVProdExpr &expr)

Inlines a sparse matrix - vector product.

When applied to a matrix-vector product, the product becomes inlineable. That is, it may be used in any vector expression (not just additive expressions). The user has to guarantee the function is only used in single-device expressions.

Example:

```
// Get maximum residual value:
eps = sum( fabs(f - vex::make_inline(A * x)) );
```

Sort, scan, reduce-by-key algorithms

VexCL provides several standalone parallel primitives that may not be used as part of a vector expression. These are `vex::inclusive_scan_by_key()`, `vex::exclusive_scan_by_key()`, `vex::sort()`, `vex::sort_by_key()`, `vex::reduce_by_key()`. All of these functions take VexCL vectors both as input and output parameters.

Sort and scan functions take an optional function object used for comparison and summing of elements. The functor should provide the same interface as, e.g. `std::less` for sorting or `std::plus` for summing; additionally, it should provide a VexCL function for device-side operations.

Here is an example of such an object comparing integer elements in such a way that even elements precede odd ones:

```
template <typename T>
struct even_first {
    #define BODY \
        char bit1 = 1 & a; \
        char bit2 = 1 & b; \
        if (bit1 == bit2) return a < b; \
        return bit1 < bit2;

    // Device version.
    VEX_FUNCTION(bool, device, (int, a)(int, b), BODY);

    // Host version.
    bool operator()(int a, int b) const { BODY }

    #undef BODY
};
```

Same functor could be created with the help of `VEX_DUAL_FUNCTOR` macro, which takes return type, sequence of arguments (similar to the `VEX_FUNCTION`), and the body of the functor:

```
template <typename T>
struct even_first {
    VEX_DUAL_FUNCTOR(bool, (T, a)(T, b),
        char bit1 = 1 & a;
        char bit2 = 1 & b;
        if (bit1 == bit2) return a < b;
        return bit1 < bit2;
    )
};
```

Note that VexCL already provides `vex::less<T>`, `vex::less_equal<T>`, `vex::greater<T>`, `vex::greater_equal<T>`, and `vex::plus<T>`.

The need to provide both host-side and device-side parts of the functor comes from the fact that multidimensional vectors are first sorted partially on each of the compute devices and then merged on the host.

Sorting algorithms may also take tuples of keys/values (in fact, any `Boost.Fusion` sequence will do). One will have to explicitly specify the comparison functor in this case. Both host and device variants of the comparison functor should take $2n$ arguments, where n is the number of keys. The first n arguments correspond to the left set of keys, and the second n arguments correspond to the right set of keys. Here is an example that sorts values by a tuple of two keys:

```
vex::vector<int>    keys1(ctx, n);
vex::vector<float>  keys2(ctx, n);
vex::vector<double> vals (ctx, n);

struct {
    VEX_FUNCTION(bool, device, (int, a1)(float, a2)(int, b1)(float, b2),
        return (a1 == b1) ? (a2 < b2) : (a1 < b1);
    );
    bool operator()(int a1, float a2, int b1, float b2) const {
        return std::make_tuple(a1, a2) < std::make_tuple(b1, b2);
    }
} comp;
```

```
vex::sort_by_key(std::tie(keys1, keys2), vals, comp);
```

template <typename T, class Oper>

void vex::**inclusive_scan** (vector<T> const &input, vector<T> &output, T init, Oper oper)

Inclusive scan.

template <typename T, class Oper>

void vex::**exclusive_scan** (vector<T> const &input, vector<T> &output, T init, Oper oper)

Exclusive scan.

template <class K, typename V, class Comp, class Oper>

void vex::**inclusive_scan_by_key** (K &&keys, const vector<V> &ivals, vector<V> &ovals, Comp comp, Oper oper, V init = V())

Inclusive scan by key.

template <class K, typename V, class Comp, class Oper>

void vex::**exclusive_scan_by_key** (K &&keys, const vector<V> &ivals, vector<V> &ovals, Comp comp, Oper oper, V init = V())

Exclusive scan by key.

template <class K, class Comp>

void vex::**sort** (K &&keys, Comp comp)

Sorts the vector into ascending order.

template <class K, class V, class Comp>

void vex::**sort_by_key** (K &&keys, V &&vals, Comp comp)

Sorts the elements in keys and values into ascending key order.

VEX_DUAL_FUNCTOR (type, args, ...) *VEX_FUNCTION*(type, device, args,
 __VA_ARGS__); \ *VEX_DUAL_FUNCTOR_SINK*(type, \
 BOOST_PP_SEQ_SIZE(VEXCL_FUNCTION_MAKE_SEQ(args)), \
 VEXCL_FUNCTION_MAKE_SEQ(args), __VA_ARGS__)

Defines both device and host versions of a function call operator.

The intended use is the creation of comparison and reduction functors for use with scan/sort/reduce algorithms.

Example:

```
template <typename T>
struct less {
    VEX_DUAL_FUNCTOR(bool, (T, a)(T, b),
        return a < b;
    )
};
```

template <typename T>

struct vex::less

Function object class for less-than inequality comparison.

The need for host-side and device-side parts comes from the fact that vectors are partially sorted on device and then final merge step is done on host.

Inherits from std::less< T >

template <typename T>

struct vex::less_equal

Function object class for less-than-or-equal inequality comparison.

Inherits from std::less_equal< T >

template <typename T>

struct vex::greater

Function object class for greater-than inequality comparison.

Inherits from std::greater< T >

template <typename T>

struct vex::greater_equal

Function object class for greater-than-or-equal inequality comparison.

Inherits from std::greater_equal< T >

template <typename T>

struct vex::plus

Binary function object class whose call returns the result of adding its two arguments.

Inherits from std::plus< T >

Multivectors and multiexpressions

The `vex::multivector<T,N>` class allows to store several equally sized device vectors and perform computations on each component in sync. Each operation is delegated to the underlying vectors, but usually results in the launch of a single fused kernel. Expressions may include values of `std::array<T,N>` where N is equal to the number of multivector components, or appropriately sized tuples. Each component gets the corresponding element of either the array or the tuple when the expression is applied. Similarly, `vex::multivector::operator[]()` or reduction of a multivector returns an instance of `std::array<T,N>`. `vex::multivector::operator()()` allows to access individual components of a multivector.

Some examples:

```
VEX_FUNCTION(bool, between, (double, a)(double, b)(double, c),
    return a <= b && b <= c;
);

vex::Reductor<double, vex::SUM> sum(ctx);
vex::SpMat<double> A(ctx, ... );
std::array<double, 2> v = {6.0, 7.0};

vex::multivector<double, 2> X(ctx, N), Y(ctx, N);

// ...

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);
```

Some operations can not be expressed with simple multivector arithmetic. For example, an operation of two dimensional rotation mixes components in the right hand side expressions:

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

This may in principle be implemented as:

```
double alpha;
vex::multivector<double, 2> X(ctx, N), Y(ctx, N);

Y(0) = X(0) * cos(alpha) - X(1) * sin(alpha);
Y(1) = X(0) * sin(alpha) + X(1) * cos(alpha);
```

But this would result in two kernel launches instead of single fused launch. VexCL allows one to assign a tuple of expressions to a multivector, which will lead to the launch of a single fused kernel:

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

`vex::tie()` function even allows to get rid of multivectors completely and fuse several vector expressions into a single kernel. We can rewrite the above examples with just `vex::vector<double>` instances:

```
vex::vector<double> x0(ctx, N), x1(ctx, N), y0(ctx, N), y1(ctx, N);

vex::tie(y0, y1) = std::make_tuple(
    x0 * cos(alpha) - x1 * sin(alpha),
    x0 * sin(alpha) + x1 * cos(alpha) );
```

```
template <typename T, size_t N>
```

```
class vex::multivector
```

Container for several equally sized instances of `vex::vector<T>`.

Inherits from `vex::multivector_expression< Expr >`

Public Functions

```
multivector (const std::vector<backend::command_queue> &queue, const std::vector<T> &host,
              backend::mem_flags flags = backend::MEM_READ_WRITE)
```

Constructor.

The host vector data is divided equally between the created multivector components. Each component gets continuous chunk of the source vector.

```
multivector (const std::vector<backend::command_queue> &queue, size_t size, const T *host = 0,
              backend::mem_flags flags = backend::MEM_READ_WRITE)
```

Constructor.

If host pointer is not NULL, it is copied to the underlying vector components of the multivector. Each component gets continuous chunk of the source vector.

```
multivector (size_t size)
```

Constructor.

Uses the most recently created VexCL context.

```
multivector (const multivector &mv)
```

Copy constructor.

```
multivector (multivector &&mv)
```

Move constructor.

```
void resize (const std::vector<backend::command_queue> &queue, size_t size)
```

Resizes the multivector.

This is equivalent to reconstructing the vector with the given parameters. Any data contained in the resized vector will be lost as a result.

```
void resize (size_t size)
```

Resizes the multivector.

Uses the most recently created VexCL context. This is equivalent to reconstructing the vector with the given parameters. Any data contained in the resized vector will be lost as a result.

void **clear** ()

Fills the multivector with zeros.

size_t **size** () const

Returns size of the multivector (equals size of individual components).

const vex::vector<T> &**operator** () (size_t i) const

Returns i-th multivector component.

vex::vector<T> &**operator** () (size_t i)

Returns i-th multivector component.

const_iterator **begin** () const

Returns const iterator to the first element of the multivector.

iterator **begin** ()

Returns const iterator to the first element of the multivector.

const_iterator **end** () const

Returns const iterator referring to the past-the-end element in the multivector.

iterator **end** ()

Returns iterator referring to the past-the-end element in the multivector.

const_element **operator** [] (size_t i) const

Returns i-th elements of all components packed in a std::array<T,N>.

element **operator** [] (size_t i)

Assigns values from std::array<T,N> to i-th elements of all components.

const std::vector<backend::command_queue> &**queue_list** () const

Returns reference to the multivector's queue list.

const multivector &**operator**= (const multivector &mv)

Assignment operator.

template <class Expr>

auto **operator**= (const Expr &expr)

Assignment operator

template <class Expr>

auto **operator**+= (const Expr &expr)

Assignment operator

template <class Expr>

auto **operator**--= (const Expr &expr)

Assignment operator

template <class Expr>

auto **operator***= (const Expr &expr)

Assignment operator

template <class Expr>

auto **operator**/= (const Expr &expr)

Assignment operator

template <class Expr>

auto **operator**%= (const Expr &expr)

Assignment operator

template <class Expr>

auto **operator**&= (const Expr &expr)

Assignment operator

```
template <class Expr>
```

```
auto operator|= (const Expr &expr)
```

Assignment operator

```
template <class Expr>
```

```
auto operator^= (const Expr &expr)
```

Assignment operator

```
template <class Expr>
```

```
auto operator<= (const Expr &expr)
```

Assignment operator

```
template <class Expr>
```

```
auto operator>= (const Expr &expr)
```

Assignment operator

```
template <class V, class E>
```

```
class iterator_type
```

Inherits from boost::iterator_facade< iterator_type< V, E >, sub_value_type, std::random_access_iterator_tag, E >

```
template <class... Expr>
```

```
auto vex::tie (const Expr&... expr)
```

Ties several vector expressions into a writeable tuple.

The following example results in a single kernel:

```
vex::vector<double> x(ctx, 1024);
vex::vector<double> y(ctx, 1024);

vex::tie(x,y) = std::make_tuple( x + y, y - x );
```

This is functionally equivalent to the following code, but does not use temporary vectors and is more efficient:

```
tmp_x = x + y;
tmp_y = y - x;
x = tmp_x;
y = tmp_y;
```

Converting generic C++ algorithms to OpenCL/CUDA

CUDA and OpenCL differ in their handling of compute kernels compilation. In NVIDIA's framework the compute kernels are compiled to PTX code together with the host program. In OpenCL the compute kernels are compiled at runtime from high-level C-like sources, adding an overhead which is particularly noticeable for smaller sized problems. This distinction leads to higher initialization cost of OpenCL programs, but at the same time it allows one to generate better optimized kernels for the problem at hand. VexCL exploits this possibility with help of its kernel generator mechanism. Moreover, VexCL's CUDA backend uses the same technique to generate and compile CUDA kernels at runtime.

An instance of `vex::symbolic<T>` dumps to an output stream any arithmetic operations it is being subjected to. For example, this code snippet:

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

results in the following output:

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

Kernel generator

The symbolic type allows one to record a sequence of arithmetic operations made by a generic C++ algorithm. To illustrate the idea, consider the generic implementation of a 4th order Runge-Kutta ODE stepper:

```
template <class state_type, class SysFunction>
void runge_kutta_4(SysFunction sys, state_type &x, double dt) {
    state_type k1 = dt * sys(x);
    state_type k2 = dt * sys(x + 0.5 * k1);
    state_type k3 = dt * sys(x + 0.5 * k2);
    state_type k4 = dt * sys(x + k3);

    x += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
}
```

This function takes a system function `sys`, state variable `x`, and advances `x` by the time step `dt`. For example, to model the equation $dx/dt = \sin(x)$, one has to provide the following system function:

```
template <class state_type>
state_type sys_func(const state_type &x) {
    return sin(x);
}
```

The following code snippet makes one hundred RK4 iterations for a single double value on a CPU:

```
double x = 1, dt = 0.01;

for(int step = 0; step < 100; ++step)
    runge_kutta_4(sys_func<double>, x, dt);
```

Let's now generate the kernel for a single RK4 step and apply the kernel to a `vex::vector<double>` (by doing this we essentially simultaneously solve a large number of identical ODEs with different initial conditions).

```
// Set recorder for expression sequence.
std::ostream body;
vex::generator::set_recorder(body);

// Create symbolic variable.
typedef vex::symbolic<double> sym_state;
sym_state sym_x(sym_state::VectorParameter);

// Record expression sequence for a single RK4 step.
double dt = 0.01;
runge_kutta_4(sys_func<sym_state>, sym_x, dt);

// Build kernel from the recorded sequence.
auto kernel = vex::generator::build_kernel(ctx, "rk4_stepper", body.str(), sym_x);

// Create initial state.
const size_t n = 1024 * 1024;
vex::vector<double> x(ctx, n);
x = 10.0 * vex::element_index() / n;
```



```
// Make 100 RK4 steps.
for(int i = 0; i < 100; i++) kernel(x);
```

This approach has some obvious restrictions. Namely, the C++ code has to be embarrassingly parallel and is not allowed to contain any branching or data-dependent loops. Nevertheless, the kernel generation facility may save a substantial amount of both human and machine time when applicable.

```
template <typename T>
```

```
class vex::symbolic
```

Symbolic variable.

Inherits from vex::generator::symbolic_expr< boost::proto::terminal< generator::variable >::type >

Public Types

```
enum scope_type
```

Scope/Type of the symbolic variable.

Values:

```
LocalVar = 0
```

Local variable.

```
VectorParameter = 1
```

Vector kernel parameter.

```
ScalarParameter = 2
```

Scalar kernel parameter.

```
enum constness_type
```

Constness of vector parameter.

Values:

```
NonConst = 0
```

Parameter should be written back at kernel exit.

```
Const = 1
```

Parameter is readonly.

Public Functions

```
symbolic()
```

Default constructor. Results in a local variable declaration.

```
symbolic(scope_type scope, constness_type constness = NonConst)
```

Constructor.

```
template <class Expr>
```

```
symbolic(const Expr &expr)
```

Expression constructor. Results in a local variable declaration initialized by the expression.

```
const symbolic &operator=(const symbolic &c) const
```

Assignment operator. Results in the assignment expression written to the recorder.

```
template <class Expr>
```

```
const symbolic &operator=(const Expr &expr)
```

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**+= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**-- (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator***= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**/= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**%= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**&= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**|= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**^= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**<<= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

template <class Expr>

const *symbolic* &**operator**>>= (const Expr &expr)

Assignment operator. Results in the assignment expression written to the recorder.

void vex::generator::set_recorder (std::ostream &os)

Set output stream for the kernel recorder.

template <class... Args>

kernel vex::generator::build_kernel (const std::vector<backend::command_queue> &queue,
const std::string &name, const std::string &body, const
Args&... args)

Builds kernel from the recorded expression sequence and the symbolic parameter list.

The symbolic variables passed to the function should have participated in the recorded algorithm and will be converted to the generated kernel arguments.

Function generator

VexCL also provides a user-defined function generator which takes a function signature and generic function object, and returns custom VexCL function ready to be used in vector expressions. Let's rewrite the above example using an autogenerated function for a Runge-Kutta stepper. First, we need to implement generic functor:

```
struct rk4_stepper {  
    double dt;  
  
    rk4_stepper(double dt) : dt(dt) {}  
};
```

```

template <class state_type>
state_type operator() (const state_type &x) const {
    state_type new_x = x;
    runge_kutta_4(sys_func<state_type>, new_x, dt);
    return new_x;
}
};

```

Now we can generate and apply the custom function:

```

double dt = 0.01;
rk4_stepper stepper(dt);

// Generate custom VexCL function:
auto rk4 = vex::generator::make_function<double(double)>(stepper);

// Create initial state.
const size_t n = 1024 * 1024;
vex::vector<double> x(ctx, n);
x = 10.0 * vex::element_index() / n;

// Use the function to advance initial state:
for(int i = 0; i < 100; i++) x = rk4(x);

```

Note that both `runge_kutta_4()` and `rk4_stepper` may be reused for the host-side computations.

It is very easy to generate a VexCL function from a `Boost.Phoenix` lambda expression (since `Boost.Phoenix` lambdas are themselves generic functors):

```

using namespace boost::phoenix::arg_names;
using vex::generator::make_function;

auto squared_radius = make_function<double(double, double)>(arg1 * arg1 + arg2 *
↳arg2);

Z = squared_radius(X, Y);

```

template <class Signature, class Functor>
auto vex::generator::make_function (Functor &&f)
 Generates a user-defined function from a generic functor.

Takes the function signature as template parameter and a generic functor as a single argument. Returns user-defined function ready to be used in vector expressions.

Custom kernels

As [Kozma Prutkov](#) repeatedly said, “One cannot embrace the unembraceable”. So in order to be usable, VexCL has to support custom kernels. `vex::backend::kernel` is a thin wrapper around a compute kernel for each of the contexts. Its constructor takes a command queue and the kernel source code, and its function call operator submits the kernel to the specified command queue. The following example builds and launches a custom kernel for the a context with a single device:

```

// Compile the kernel. This can be done once per program lifetime.
// If offline kernel cache is enabled, it will be used for custom kernels as well.
vex::backend::kernel dummy(ctx.queue(0), VEX_STRINGIZE_SOURCE(
    kernel void dummy(ulong n, global int *x) {

```

```
        for(size_t i = get_global_id(0); i < n; i += get_global_size(0))
            x[i] = 42;
    }},
    "dummy");

vex::vector<int> x(ctx, n);

// Apply the kernel to the vector partition located on the first device:
dummy(ctx.queue(0), static_cast<cl_ulong>(n), x(0));
```

In case there are several devices in the context, you will need to create an instance of the kernel for each of the devices. `vex::vector::operator()()` returns vector partition located on the given device. If the result depends on the neighboring points, one has to keep in mind that these points are possibly located on a different compute device. In this case the exchange of these halo points has to be addressed manually.

```
std::vector<vex::backend::kernel> kernel;

// Compile and store the kernels for the later use.
for(uint d = 0; d < ctx.size(); d++) {
    kernel.emplace_back(ctx.queue(d), VEX_STRINGIZE_SOURCE(
        kernel void dummy(ulong n, global float *x) {
            for(size_t i = get_global_id(0); i < n; i += get_global_size(0))
                x[i] = 4.2;
        }},
        "dummy");
}

// Apply the kernels to the vector partitions on each device.
for(uint d = 0; d < ctx.size(); d++)
    kernel[d](ctx.queue(d), static_cast<cl_ulong>(x.part_size(d)), x(d));
```

Interoperability with other libraries

VexCL does not try (too hard) to hide the implementation details from the user. For example, in case of the OpenCL backend VexCL is based on the [Khronos C++ API](#), and the underlying OpenCL types are easily accessible. Hence, it should be easy to interoperate with other OpenCL libraries. Similarly, in case of the CUDA backend, VexCL backend types are thin wrappers around [CUDA Driver API](#).

When [Boost.Compute](#) backend is used, VexCL is based on the core classes of the [Boost.Compute](#) library. It is very easy to apply [Boost.Compute](#) algorithms to VexCL vectors and to use [Boost.Compute](#) buffers within VexCL expressions.

Here is an example:

```
#include <iostream>
#include <boost/compute.hpp>

#define VEXCL_BACKEND_COMPUTE
#include <vexcl/vexcl.hpp>

namespace compute = boost::compute;

int main() {
    compute::command_queue bcq = compute::system::default_queue();

    const int n = 16;
```

```

// Use boost.compute queue to allocate VexCL vectors:
vex::vector<int> x({bcq}, n);
x = 2 * vex::element_index();

// Wrap boost.compute vectors into vexcl vectors (no data is copied):
compute::vector<int> bcv(n, bcq.get_context());
vex::vector<int> y({bcq}, bcv.get_buffer());
y = x * 2;

// Apply Boost.Compute algorithm to a vexcl vector:
compute::sort(
    compute::make_buffer_iterator<int>(x(0).raw_buffer(), 0),
    compute::make_buffer_iterator<int>(x(0).raw_buffer(), n)
);
}

```

Building VexCL programs with CMake

In order to build a VexCL program with the CMake build system you need just a couple of lines in your CmakeLists.txt:

```

cmake_minimum_required(VERSION 2.8)
project(example)

find_package(VexCL)

add_executable(example example.cpp)
target_link_libraries(example VexCL::OpenCL)

```

VexCL provides interface targets for the backends supported on the current system. Possible choices are VexCL::OpenCL for the OpenCL backend, VexCL::Compute for Boost.Compute, VexCL::CUDA for CUDA, and VexCL::JIT for the just-in-time compiled OpenMP kernels. The targets will take care of the appropriate compiler and linker flags for the selected backend.

find_package(VexCL) may be used when VexCL was installed system wide. If that is not the case, you can just copy the VexCL into a subdirectory of your project and replace the line with

```
add_subdirectory(vexcl)
```

Talks and publications

See also:

Slides for these and other talks may be found at <https://speakerdeck.com/ddemidov>.

University of Texas at Austin, 2013

An overview of VexCL interface.

Slides

Meeting C++, Berlin, 2014

Discussion of C++ techniques that VexCL uses to effectively generate OpenCL/CUDA compute kernels from the user expressions.

Slides

Video

Publications

- D. Demidov, K. Ahnert, K. Rupp, and P. Gottchling. “Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries.” SIAM Journal on Scientific Computing 35.5 (2013): C453-C472. DOI: [10.1137/120903683](https://doi.org/10.1137/120903683).
- K. Ahnert, D. Demidov, and M. Mulansky. “Solving Ordinary Differential Equations on GPUs.” Numerical Computations with GPUs. Springer International Publishing, 2014. 125-157. DOI: [10.1007/978-3-319-06548-9_7](https://doi.org/10.1007/978-3-319-06548-9_7).

Indices and tables

- [genindex](#)
- [search](#)

Bibliography

- [LeWS97] S. Lee, G. Wolberg, and S. Y. Shin. Scattered data interpolation with multilevel B-Splines. *IEEE Transactions on Visualization and Computer Graphics*, 3:228–244, 1997

V

- vex::_ (C++ member), 22
- vex::axes_pairs (C++ function), 24
- vex::backend::device_list<DevFilter> (C++ function), 6
- vex::CombineReducers (C++ class), 28
- vex::constant (C++ function), 18
- vex::constant_pointer (C++ function), 18
- vex::Context (C++ class), 6
- vex::Context::Context (C++ function), 6
- vex::Context::context (C++ function), 6
- vex::Context::device (C++ function), 6
- vex::Context::empty (C++ function), 6
- vex::Context::finish (C++ function), 6
- vex::Context::operator bool (C++ function), 6
- vex::Context::operator const
std::vector<vex::backend::command_queue>&
(C++ function), 6
- vex::Context::queue (C++ function), 6
- vex::Context::size (C++ function), 6
- vex::element_index (C++ function), 14
- vex::exclusive_scan (C++ function), 31
- vex::exclusive_scan_by_key (C++ function), 31
- vex::extents (C++ member), 22
- vex::FFT (C++ class), 26
- vex::fft::direction (C++ type), 27
- vex::FFT::FFT (C++ function), 26, 27
- vex::fft::forward (C++ class), 27
- vex::fft::inverse (C++ class), 27
- vex::fft::none (C++ class), 27
- vex::FFT::operator() (C++ function), 27
- vex::generator::build_kernel (C++ function), 38
- vex::generator::make_function (C++ function), 39
- vex::generator::set_recorder (C++ function), 38
- vex::greater (C++ class), 31
- vex::greater_equal (C++ class), 31
- vex::inclusive_scan (C++ function), 31
- vex::inclusive_scan_by_key (C++ function), 31
- vex::less (C++ class), 31
- vex::less_equal (C++ class), 31
- vex::make_inline (C++ function), 29
- vex::make_temp (C++ function), 17
- vex::MAX (C++ class), 28
- vex::mba (C++ class), 25
- vex::mba::mba (C++ function), 25
- vex::mba::operator() (C++ function), 25
- vex::MIN (C++ class), 28
- vex::MIN_MAX (C++ type), 28
- vex::multivector (C++ class), 33
- vex::multivector::begin (C++ function), 34
- vex::multivector::clear (C++ function), 33
- vex::multivector::end (C++ function), 34
- vex::multivector::iterator_type (C++ class), 35
- vex::multivector::multivector (C++ function), 33
- vex::multivector::operator() (C++ function), 34
- vex::multivector::operator*= (C++ function), 34
- vex::multivector::operator+= (C++ function), 34
- vex::multivector::operator-= (C++ function), 34
- vex::multivector::operator/= (C++ function), 34
- vex::multivector::operator= (C++ function), 34
- vex::multivector::operator%= (C++ function), 34
- vex::multivector::operator&= (C++ function), 34
- vex::multivector::operator^= (C++ function), 35
- vex::multivector::operator|= (C++ function), 35
- vex::multivector::operator>= (C++ function), 35
- vex::multivector::operator<= (C++ function), 35
- vex::multivector::operator[] (C++ function), 34
- vex::multivector::queue_list (C++ function), 34
- vex::multivector::resize (C++ function), 33
- vex::multivector::size (C++ function), 34
- vex::permutation (C++ function), 21
- vex::plus (C++ class), 32
- vex::Random (C++ class), 20
- vex::RandomNormal (C++ class), 20
- vex::range (C++ class), 22
- vex::range::range (C++ function), 22
- vex::raw_pointer (C++ function), 18
- vex::reduce (C++ function), 23
- vex::Reducer (C++ class), 28
- vex::Reducer::operator() (C++ function), 28

`vex::Reductor::Reductor` (C++ function), 28
`vex::reshape` (C++ function), 24
`vex::slicer` (C++ class), 21
`vex::slicer::slice` (C++ class), 22
`vex::slicer::slicer` (C++ function), 22
`vex::sort` (C++ function), 31
`vex::sort_by_key` (C++ function), 31
`vex::SpMat` (C++ class), 29
`vex::SpMat::cols` (C++ function), 29
`vex::SpMat::nonzeros` (C++ function), 29
`vex::SpMat::rows` (C++ function), 29
`vex::SpMat::SpMat` (C++ function), 29
`vex::SUM` (C++ class), 28
`vex::svm_vector` (C++ class), 11
`vex::svm_vector::map` (C++ function), 12
`vex::svm_vector::operator=` (C++ function), 12
`vex::svm_vector::queue` (C++ function), 11
`vex::svm_vector::size` (C++ function), 11
`vex::svm_vector::svm_vector` (C++ function), 11
`vex::symbolic` (C++ class), 37
`vex::symbolic::Const` (C++ class), 37
`vex::symbolic::constness_type` (C++ type), 37
`vex::symbolic::LocalVar` (C++ class), 37
`vex::symbolic::NonConst` (C++ class), 37
`vex::symbolic::operator*=` (C++ function), 38
`vex::symbolic::operator+=` (C++ function), 37
`vex::symbolic::operator-=` (C++ function), 38
`vex::symbolic::operator/=` (C++ function), 38
`vex::symbolic::operator=` (C++ function), 37
`vex::symbolic::operator%=` (C++ function), 38
`vex::symbolic::operator&=` (C++ function), 38
`vex::symbolic::operator^=` (C++ function), 38
`vex::symbolic::operator|=` (C++ function), 38
`vex::symbolic::operator>>=` (C++ function), 38
`vex::symbolic::operator<<=` (C++ function), 38
`vex::symbolic::ScalarParameter` (C++ class), 37
`vex::symbolic::scope_type` (C++ type), 37
`vex::symbolic::symbolic` (C++ function), 37
`vex::symbolic::VectorParameter` (C++ class), 37
`vex::tag` (C++ function), 16
`vex::tensordot` (C++ function), 24
`vex::tie` (C++ function), 35
`vex::vector` (C++ class), 7
`vex::vector::begin` (C++ function), 9
`vex::vector::clear` (C++ function), 8
`vex::vector::end` (C++ function), 9
`vex::vector::iterator_type` (C++ class), 10
`vex::vector::map` (C++ function), 9
`vex::vector::nparts` (C++ function), 9
`vex::vector::operator()` (C++ function), 8, 9
`vex::vector::operator*=` (C++ function), 10
`vex::vector::operator+=` (C++ function), 9
`vex::vector::operator-=` (C++ function), 10
`vex::vector::operator/=` (C++ function), 10
`vex::vector::operator=` (C++ function), 9
`vex::vector::operator%=` (C++ function), 10
`vex::vector::operator&=` (C++ function), 10
`vex::vector::operator^=` (C++ function), 10
`vex::vector::operator|=` (C++ function), 10
`vex::vector::operator>>=` (C++ function), 10
`vex::vector::operator<<=` (C++ function), 10
`vex::vector::operator[]` (C++ function), 9
`vex::vector::part_size` (C++ function), 9
`vex::vector::part_start` (C++ function), 9
`vex::vector::queue_list` (C++ function), 9
`vex::vector::resize` (C++ function), 8
`vex::vector::size` (C++ function), 9
`vex::vector::swap` (C++ function), 8
`vex::vector::vector` (C++ function), 7, 8
`VEX_CONSTANT` (C macro), 13
`VEX_DUAL_FUNCTOR` (C macro), 31
`VEX_FUNCTION` (C macro), 15
`VEX_FUNCTION_D` (C macro), 15
`VEX_FUNCTION_S` (C macro), 15
`VEX_FUNCTION_SD` (C macro), 15
`VEX_STRINGIZE_SOURCE` (C macro), 15
`VEXCL_CACHE_KERNELS` (C macro), 12
`VEXCL_SHOW_KERNELS` (C macro), 12