Программирование OpenCL

с использованием библиотек C++

Денис Демидов

Институт системных исследований РАН
Казанский Федеральный Университет

23.10.2015

Современные GPGPU платформы

NVIDIA CUDA

- Проприетарная архитектура
- Работает только на видеокартах NVIDIA
- Высокоуровневый интерфейс
- Ядра (C++) компилируются в псевдокод (PTX) вместе с основной программой

OpenCL

- Открытый стандарт
- Поддерживается многими вендорами
- Низкоуровневый интерфейс
- Ядра (C99) компилируются во время выполнения основной программы

Поддерживаемые устройства

- NVIDIA CUDA
  - Видеокарты NVIDIA
- OpenCL
  - Видеокарты NVIDIA, AMD, Intel
  - Центральные процессоры Intel, AMD, ARM
  - Мобильные системы, программируемые чипы, ...

Программный интерфейс OpenCL

Справочная информация

- Сайт группы компаний Khronos:
  - Спецификации: khronos.org/registry/cl
  - Ресурсы: khronos.org/opencl/resources
- google.com

Основные этапы при использовании OpenCL

- Инициализация
  - Выбор платформы
  - Выбор устройства
  - Создание контекста
  - Создание очереди команд
  - Выделение памяти на устройстве
  - Компиляция вычислительных ядер
- Работа
  - Перенос данных
  - Выполнение расчетов

Платформа OpenCL

Платформа — это реализация OpenCL от производителя аппаратного обеспечения (AMD, Intel, NVIDIA, и т.д.)

- С платформой связан список поддерживаемых устройств

C интерфейс

```
1  #include <iostream>
2  #include <vector>
3  #include <stdexcept>
4  #include <CL/cl.h>
5
6  void check(cl_int return_code) {
7      if (return_code != CL_SUCCESS) throw std::runtime_error("OpenCL error");
8  }
9  int main() {
10     cl_uint np;
11     check( clGetPlatformIDs(0, NULL, &np) );
12     std::vector<cl_platform_id> platforms(np);
13     check( clGetPlatformIDs(np, platforms.data(), &np) );
14     char name[256];
15     for (auto p : platforms) {
16         check( clGetPlatformInfo(p, CL_PLATFORM_NAME, 256, name, NULL) );
17         std::cout << name << std::endl;
18     }
19 }
```

C интерфейс

```
1  #include <iostream>
2  #include <vector>
3  #include <stdexcept>
4  #include <CL/cl.h>
5
6  void check(cl_int return_code) {
7      if (return_code != CL_SUCCESS) throw std::runtime_error("OpenCL error");
8  }
9  int main() {
10     cl_uint np;
11     check( clGetPlatformIDs(0, NULL, &np) );
12     std::vector<cl_platform_id> platforms(np);
13     check( clGetPlatformIDs(np, platforms.data(), &np) );
14     char name[256];
15     for (auto p : platforms) {
16         check( clGetPlatformInfo(p, CL_PLATFORM_NAME, 256, name, NULL) );
17         std::cout << name << std::endl;
18     }
19 }
```

C интерфейс

```
1  #include <iostream>
2  #include <vector>
3  #include <stdexcept>
4  #include <CL/cl.h>
5
6  void check(cl_int return_code) {
7      if (return_code != CL_SUCCESS) throw std::runtime_error("OpenCL error");
8  }
9  int main() {
10     cl_uint np;
11     check( clGetPlatformIDs(0, NULL, &np) );
12     std::vector<cl_platform_id> platforms(np);
13     check( clGetPlatformIDs(np, platforms.data(), &np) );
14     char name[256];
15     for (auto p : platforms) {
16         check( clGetPlatformInfo(p, CL_PLATFORM_NAME, 256, name, NULL) );
17         std::cout << name << std::endl;
18     }
19 }
```

C интерфейс

```
1  #include <iostream>
2  #include <vector>
3  #include <stdexcept>
4  #include <CL/cl.h>
5
6  void check(cl_int return_code) {
7      if (return_code != CL_SUCCESS) throw std::runtime_error("OpenCL error");
8  }
9  int main() {
10     cl_uint np;
11     check( clGetPlatformIDs(0, NULL, &np) );
12     std::vector<cl_platform_id> platforms(np);
13     check( clGetPlatformIDs(np, platforms.data(), &np) );
14     char name[256];
15     for (auto p : platforms) {
16         check( clGetPlatformInfo(p, CL_PLATFORM_NAME, 256, name, NULL) );
17         std::cout << name << std::endl;
18     }
19 }
```

C++ интерфейс

```
1  #include <iostream>
2  #include <vector>
3
4  #define __CL_ENABLE_EXCEPTIONS
5  #include <CL/cl.hpp>
6
7  int main() {
8      std::vector<cl::Platform> platforms;
9      cl::Platform::get(&platforms);
10
11     for (const auto &p : platforms)
12         std::cout << p.getInfo<CL_PLATFORM_NAME>() << std::endl;
13 }
```

C++ интерфейс

```
1  #include <iostream>
2  #include <vector>
3
4  #define __CL_ENABLE_EXCEPTIONS
5  #include <CL/cl.hpp>
6
7  int main() {
8      std::vector<cl::Platform> platforms;
9      cl::Platform::get(&platforms);
10
11     for (const auto &p : platforms)
12         std::cout << p.getInfo<CL_PLATFORM_NAME>() << std::endl;
13 }
```

C++ интерфейс

```
1  #include <iostream>
2  #include <vector>
3
4  #define __CL_ENABLE_EXCEPTIONS
5  #include <CL/cl.hpp>
6
7  int main() {
8      std::vector<cl::Platform> platforms;
9      cl::Platform::get(&platforms);
10
11     for (const auto &p : platforms)
12         std::cout << p.getInfo<CL_PLATFORM_NAME>() << std::endl;
13 }
```

Устройство OpenCL

Устройство — конкретное вычислительное устройство, поддерживаемое одной из установленных платформ.

Получение списка устройств для платформы

```
1 std::vector<cl::Device> devices;
2 p.getDevices(CL_DEVICE_TYPE_ALL, &devices);
3
4 for(const auto &d : devices)
5     std::cout << " " << d.getInfo<CL_DEVICE_NAME>() << std::endl;
```

Типы устройств:

- CL_DEVICE_TYPE_ALL
- CL_DEVICE_TYPE_DEFAULT
- CL_DEVICE_TYPE_CPU
- CL_DEVICE_TYPE_GPU
- CL_DEVICE_TYPE_ACCELERATOR

Контекст OpenCL

Контекст — служит для управления объектами и ресурсами OpenCL.

С контекстом связаны:

- программы и ядра
- буферы памяти
- очереди команд

Создание контекста

```
1 cl::Context context(devices);
```

Очередь команд

Очередь команд — позволяет отправить задание на выполнение на устройство.

- Очередь связана с единственным устройством.
- Постановка задания в очередь выполняется асинхронно.
- Задания выполняются в порядке их постановки в очередь.
  - Вычислительные ядра
  - Операции копирования памяти

Создание очереди

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

Выделение и копирование памяти

Буфер памяти — объект, владеющий некоторым объемом памяти в контексте.

- Все устройства в контексте могут получить доступ к буферам памяти.

Выделение и перенос памяти

```
1  std::vector<double> x(1024, 42.0);
2
3  cl::Buffer a(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
4           x.size() * sizeof(x[0]), x.data());
5
6  size_t nbytes = 1024 * sizeof(double);
7  cl::Buffer b(context, CL_MEM_READ_WRITE, nbytes);
8
9  queue.enqueueWriteBuffer(b, CL_FALSE, 0, nbytes, x.data());
10 queue.enqueueReadBuffer(b, CL_TRUE, 0, nbytes, x.data());
```

Создание вычислительного ядра

Ядро — функция, исполняющаяся на вычислительном устройстве.

Программа — содержит исходные тексты и/или скомпилированные ядра.

Создание программы и ядра

```
1  std::string source = R"(
2  kernel void add(ulong n, global const double *a, global double *b) {
3      ulong i = get_global_id(0);
4      if (i < n) b[i] += a[i];
5  }
6  )";
7
8  cl::Program program(context, source);
9  program.build(devices);
10
11 cl::Kernel add(program, "add");
```

Запуск ядра на выполнение

Постановка ядра в очередь

```
1 add.setArg(0, static_cast<cl_ulong>(n));
2 add.setArg(1, a);
3 add.setArg(2, b);
4
5 queue.enqueueNDRangeKernel(add, cl::NullRange, cl::NDRange(n), cl::NullRange);
```

Считывание результатов

```
1 queue.enqueueReadBuffer(b, CL_TRUE, 0, nbytes, x.data());
2 std::cout << x[0] << std::endl;
```

Полный пример

- Вычислить сумму двух векторов на видеокарте
  - A и B — векторы большой размерности
  - Вычислить поэлементную сумму $\mathrm{B} = \mathrm{A} + \mathrm{B}$.

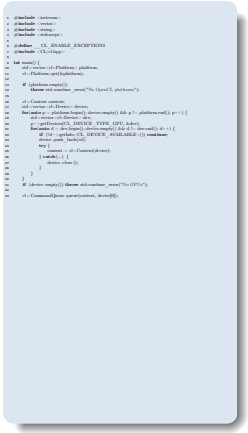
- Основные шаги
  1. Инициализируем контекст
  2. Выделяем память
  3. Переносим входные данные
  4. Проводим вычисления
  5. Забираем результаты

Hello OpenCL: Сумма двух векторов

Разрешаем выбрасывать исключения

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include <stdexcept>
5
6  #define __CL_ENABLE_EXCEPTIONS
7  #include <CL/cl.hpp>
```

Hello OpenCL: Сумма двух векторов



Инициализируем контекст

```
8
9   int main() {
10      std :: vector<cl::Platform> platform;
11      cl :: Platform::get(&platform);
12
13      if (platform.empty())
14          throw std::runtime_error("No OpenCL platforms");
15
16      cl :: Context context;
17      std :: vector<cl::Device> device;
18      for(auto p = platform.begin(); device.empty() && p != platform.end(); p++) {
19          std :: vector<cl::Device> dev;
20          p->getDevices(CL_DEVICE_TYPE_GPU, &dev);
21          for(auto d = dev.begin(); device.empty() && d != dev.end(); d++) {
22              if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
23              device.push_back(*d);
24              try {
25                  context = cl::Context(device);
26              } catch(...) {
27                  device.clear();
28              }
29          }
30      }
31      if (device.empty()) throw std::runtime_error("No GPUs");
32
33      cl :: CommandQueue queue(context, device[0]);
```

Hello OpenCL: Сумма двух векторов

Выделяем память

```
34
35      const size_t n = 1024 * 1024;
36      std::vector<double> a(n, 1.5), b(n, 2.7);
37
38      cl::Buffer A(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
39              a.size() * sizeof(a[0]), a.data());
40
41      cl::Buffer B(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
42              b.size() * sizeof(b[0]), b.data());
```

Hello OpenCL: Сумма двух векторов

Компилируем и выполняем ядро

```
43
44      std::string source = R"(
45          kernel void add(ulong n, global const double *a, global double *b) {
46            ulong i = get_global_id(0);
47            if (i < n) b[i] += a[i];
48          }
49          )";
50
51      cl::Program program(context, source);
52
53      try {
54          program.build(device);
55      } catch (const cl::Error&) {
56          std::cerr
57              << "OpenCL compilation error" << std::endl
58              << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device[0])
59              << std::endl;
60          throw std::runtime_error("OpenCL build error");
61      }
62      cl::Kernel add(program, "add");
63
64      add.setArg(0, static_cast<cl_ulong>(n));
65      add.setArg(1, A);
66      add.setArg(2, B);
67
68      queue.enqueueNDRangeKernel(add, cl::NullRange, n, cl::NullRange);
```

Hello OpenCL: Сумма двух векторов

```
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>

#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>

int main() {
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    if (platforms.empty()) {
        throw std::runtime_error("No OpenCL platforms");
    }

    cl::Context context;
    std::vector<cl::Device> devices;
    for(auto p : platforms.begin()) { devices.empty(); ++p ) {
        std::vector<cl::Device> devs;
        p->getDevices(CL_DEVICE_TYPE_GPU, &devs);
        for(auto d = devs.begin(); devs.empty(); ++d ) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            devices.push_back(*d);
            try {
                context = cl::Context(devices);
            } catch(...) {
                devices.clear();
            }
        }
    }
    if (devices.empty()) throw std::runtime_error("No GPUs");

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

    const size_t n = 1024 * 1024;
    std::vector<double> a(n, 1.5), b(n, 2.7);

    cl::Buffer A(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                 n * sizeof(a[0]), a.data());

    cl::Buffer B(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                 b.size() * sizeof(b[0]), b.data());

    std::string source = R"(
        kernel void add(double a, global const double *a, global double *b) {
            const int i = get_global_id(0);
            if (i < n) b[i] += a[i];
        }
    )";

    cl::Program program(context, source);

    try {
        program.build(devices);
    } catch (const cl::Error&) {
        std::cout
            << "OpenCL compilation error" << std::endl
            << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0])
            << std::endl;
        throw std::runtime_error("OpenCL build error");
    }
    cl::Kernel add(program, "add");

    add.setArg(0, sizeof(cl_double), &a[0]);
    add.setArg(1, A);
    add.setArg(2, B);

    queue.enqueueNDRangeKernel(add, cl::NullRange, n, cl::NullRange);

    queue.enqueueReadBuffer(B, CL_TRUE, 0, b.size() * sizeof(b[0]), b.data());
    std::cout << b[42] << std::endl;
}
```

Забираем результаты

```
69
70      queue.enqueueReadBuffer(B, CL_TRUE, 0, b.size() * sizeof(b[0]), b.data());
71      std::cout << b[42] << std::endl;
72  }
```

Терминология: CUDA vs OpenCL

Иерархия памяти

CUDA

- Глобальная память

  double *p

OpenCL

- Глобальная память

  global double *p

Иерархия памяти

CUDA

- Глобальная память
  double *p
- Разделяемая память
  __shared__ **double** *p

OpenCL

- Глобальная память
  global double *p
- Локальная память
  local **double** *p

Иерархия памяти

CUDA

- Глобальная память
  double *p
- Разделяемая память
  __shared__ **double** *p
- Константая память
  __constant__ **double** *p

OpenCL

- Глобальная память
  global double *p
- Локальная память
  local **double** *p
- Константая память
  constant **double** *p

Иерархия памяти

CUDA

- Глобальная память
  double *p
- Разделяемая память
  __shared__ **double** *p
- Константная память
  __constant__ **double** *p
- Локальная память

OpenCL

- Глобальная память
  global double *p
- Локальная память
  local **double** *p
- Константная память
  constant **double** *p
- Приватная память

Декораторы функций

CUDA

- Ядро

  `__global__`

OpenCL

- Ядро

  kernel

Декораторы функций

CUDA

- Ядро
  `__global__`
- Функция
  `__device__`

OpenCL

- Ядро
  kernel
- Функция
  не требуется

Индексация потоков

CUDA

- Размер блока (thread block)

  blockDim.x *//y, z*

OpenCL

- Размер рабочей группы (work-group)

  get_local_size(0) *// 1, 2*

Индексация потоков

CUDA

- Размер блока (thread block)
  blockDim.x *//y, z*
- Номер блока
  blockIdx.x

OpenCL

- Размер рабочей группы (work-group)
  get_local_size(0) *// 1, 2*
- Номер рабочей группы
  get_group_id(0)

Индексация потоков

CUDA

- Размер блока (thread block)
  blockDim.x *//y, z*
- Номер блока
  blockIdx.x
- Число блоков
  gridDim.x

OpenCL

- Размер рабочей группы (work-group)
  get_local_size(0) *// 1, 2*
- Номер рабочей группы
  get_group_id(0)
- Число рабочих групп
  get_num_groups(0)

Индексация потоков

CUDA

- Размер блока (thread block)
  blockDim.x *//y, z*
- Номер блока
  blockIdx.x
- Число блоков
  gridDim.x
- Номер потока в блоке (thread)
  threadIdx.x

OpenCL

- Размер рабочей группы (work-group)
  get_local_size(0) *// 1, 2*
- Номер рабочей группы
  get_group_id(0)
- Число рабочих групп
  get_num_groups(0)
- Номер элемента рабочей группы (work-item)
  get_local_id(0)

Индексация потоков

CUDA

- Размер блока (thread block)
  blockDim.x *//y, z*
- Номер блока
  blockIdx.x
- Число блоков
  gridDim.x
- Номер потока в блоке (thread)
  threadIdx.x
- Глобальный номер потока
  blockDim.x * blockIdx.x + threadIdx.x

OpenCL

- Размер рабочей группы (work-group)
  get_local_size(0) *// 1, 2*
- Номер рабочей группы
  get_group_id(0)
- Число рабочих групп
  get_num_groups(0)
- Номер элемента рабочей группы (work-item)
  get_local_id(0)
- Глобальный номер элемента
  get_global_id(0)

Индексация потоков

CUDA

- Размер блока (thread block)
  blockDim.x *//y, z*
- Номер блока
  blockIdx.x
- Число блоков
  gridDim.x
- Номер потока в блоке (thread)
  threadIdx.x
- Глобальный номер потока
  blockDim.x * blockIdx.x + threadIdx.x
- Глобальный размер
  blockDim.x * gridDim.x

OpenCL

- Размер рабочей группы (work-group)
  get_local_size(0) *// 1, 2*
- Номер рабочей группы
  get_group_id(0)
- Число рабочих групп
  get_num_groups(0)
- Номер элемента рабочей группы (work-item)
  get_local_id(0)
- Глобальный номер элемента
  get_global_id(0)
- Глобальный размер
  get_global_size(0)

Пример ядра

CUDA

```
1  __device__ float process(float a) {
2      return a * 2;
3  }
4  __global__ void do_stuff(
5          size_t n,
6          const float *a,
7          float *b
8          )
9  {
10     size_t i = threadIdx.x + blockIdx.x * blockDim.x;
11     if (i < n) b[i] = process(a[i]);
12 }
```

OpenCL

```
1  float process(float a) {
2      return a * 2;
3  }
4  kernel void do_stuff(
5          ulong n,
6          global const float *a,
7          global float *b
8          )
9  {
10     ulong i = get_global_id(0);
11     if (i < n) b[i] = process(a[i]);
12 }
```

Программирование OpenCL с использованием библиотек C++

GPGPU библиотеки C++

- Boost.Compute
  - github.com/boostorg/compute
  - OpenCL
- VexCL
  - github.com/ddemidov/vexcl
  - OpenCL, Boost.Compute, CUDA
- ViennaCL
  - github.com/viennacl/viennacl-dev
  - OpenCL, CUDA, OpenMP
- Bolt
  - github.com/HSA-Libraries/Bolt
  - OpenCL*, TBB, Microsoft C++ AMP
- ...

Boost.Compute

- Почти полная реализация STL на OpenCL
  - Контейнеры
    vector, string, flat_map, flat_set, array, ...
  - Алгоритмы
    fill , copy, transform, accumulate, count, partial_sum, sort, ...
  - Генераторы случайных чисел
  - ...
- Ядро библиотеки может использоваться как более качественная и удобная альтернатива C++ API от Khronos
- Исходный код доступен под лицезией Boost

Hello Boost.Compute: Сумма двух векторов

OpenCL

```
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

#define __CL_ENABLE_EXCEPTIONS
#include "CL/cl.hpp"
```

Boost.Compute

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

namespace compute = boost::compute;
```

Hello Boost.Compute: Сумма двух векторов

OpenCL

Boost.Compute

```
1  #include <iostream>
2  #include <vector>
3  #include <boost/compute.hpp>
4
5  namespace compute = boost::compute;
6
7  int main() {
8      compute::command_queue q = compute::system::default_queue();
9      std::cout << q.get_device().name() << std::endl;
```

Hello Boost.Compute: Сумма двух векторов

OpenCL

Boost.Compute

```
1  #include <iostream>
2  #include <vector>
3  #include <boost/compute.hpp>
4
5  namespace compute = boost::compute;
6
7  int main() {
8      compute::command_queue q = compute::system::default_queue();
9      std::cout << q.get_device().name() << std::endl;
10
11     size_t n = 1024 * 1024;
12     std::vector<float> a(n, 1), b(n, 2);
13
14     compute::vector<float> A(a, q);
15     compute::vector<float> B(b, q);
```

Hello Boost.Compute: Сумма двух векторов

OpenCL

```
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

#define __CL_ENABLE_EXCEPTIONS
#include "CL/cl.hpp"

int main() {
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    if (platforms.empty()) {
        throw std::runtime_error("No OpenCL platforms");
    }

    cl::Context context;
    std::vector<cl::Device> devices;
    for(auto p = platforms.begin(); devices.empty() && p != platforms.end(); p++) {
        std::vector<cl::Device> devs;
        p->getDevices(CL_DEVICE_TYPE_GPU, &devs);
        for(auto d = devs.begin(); devices.empty() && d != devs.end(); d++) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            devices.push_back(*d);
            try {
                context = cl::Context(devices);
            } catch(...) {
                devices.clear();
            }
        }
    }
    if (devices.empty()) throw std::runtime_error("No GPUs");

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

    const size_t n = 1024 * 1024;
    std::vector<double> a(n, 1.0), b(n, 2.7);

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

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

    std::string source = R"(
        kernel void add(ulong n, global const double *a, global double *b) {
            ulong i = get_global_id(0);
            if (i < n) b[i] += a[i];
        }
    )";

    cl::Program program(context, source);

    try {
        program.build(devices);
    } catch(cl::Error const&) {
        std::cout
            << "OpenCL compilation error" << std::endl
            << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0])
            << std::endl;
        throw std::runtime_error("OpenCL build error");
    }
    cl::Kernel add(program, "add");

    add.setArg(0, static_cast<cl_ulong>(n));
    add.setArg(1, A);
    add.setArg(2, B);

    queue.enqueueNDRangeKernel(add, cl::NullRange, n, cl::NullRange);
```

Boost.Compute

```
1  #include <iostream>
2  #include <vector>
3  #include <boost/compute.hpp>
4
5  namespace compute = boost::compute;
6
7  int main() {
8      compute::command_queue q = compute::system::default_queue();
9      std::cout << q.get_device().name() << std::endl;
10
11     size_t n = 1024 * 1024;
12     std::vector<float> a(n, 1), b(n, 2);
13
14     compute::vector<float> A(a, q);
15     compute::vector<float> B(b, q);
16
17     compute::transform(A.begin(), A.end(), B.begin(), B.begin(), compute::plus<float>(), q);
```

Hello Boost.Compute: Сумма двух векторов

OpenCL

Boost.Compute

```
1  #include <iostream>
2  #include <vector>
3  #include <boost/compute.hpp>
4
5  namespace compute = boost::compute;
6
7  int main() {
8      compute::command_queue q = compute::system::default_queue();
9      std::cout << q.get_device().name() << std::endl;
10
11     size_t n = 1024 * 1024;
12     std::vector<float> a(n, 1), b(n, 2);
13
14     compute::vector<float> A(a, q);
15     compute::vector<float> B(b, q);
16
17     compute::transform(A.begin(), A.end(), B.begin(), B.begin(), compute::plus<float>(), q);
18
19     compute::copy(B.begin(), B.end(), b.begin(), q);
20     std::cout << b[42] << std::endl;
21 }
```

Поддержка пользовательских операций

Лямбда-функции

```
1 using compute::_1;
2 using compute::_2;
3
4 compute::transform(A.begin(), A.end(), B.begin(), B.begin(), _1 + _2, q);
```

Пользовательские функции

```
1 BOOST_COMPUTE_FUNCTION(float, my_sum, (float x)(float y), {
2     return x + y;
3 });
4
5 compute::transform(A.begin(), A.end(), B.begin(), B.begin(), my_sum, q);
```

VexCL — библиотека векторных выражений для OpenCL/CUDA

- Создана для облегчения разработки GPGPU приложений на C++
  - Удобная нотация для векторных выражений
  - Автоматическая генерация ядер OpenCL/CUDA во время выполнения
- Поддерживаемые технологии
  - OpenCL (Khronos C++ API, Boost.Compute)
  - NVIDIA CUDA
- Исходный код доступен под лицензией MIT

Hello VexCL: vector sum

OpenCL

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

#define __CL_ENABLE_EXCEPTIONS
#include "CL/cl.hpp"
```

VexCL

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

Hello VexCL: vector sum

OpenCL

```
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

#define __CL_ENABLE_EXCEPTIONS
#include "CL/cl.hpp"

int main() {
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    if (platforms.empty()) {
        throw std::runtime_error("No OpenCL platforms");
    }

    cl::Context context;
    std::vector<cl::Device> devices;
    for(auto p = platforms.begin(); devices.empty() && p != platforms.end(); p++) {
        std::vector<cl::Device> devs;
        p->getDevices(CL_DEVICE_TYPE_GPU, &devs);
        for(auto d = devs.begin(); devices.empty() && d != devs.end(); d++) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            devices.push_back(*d);
            try {
                context = cl::Context(devices);
            } catch(...) {
                devices.clear();
            }
        }
    }
    if (devices.empty()) throw std::runtime_error("No GPUs");

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

VexCL

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

int main() {
    vex::Context ctx( vex::Filter::GPU );
    std::cout << ctx << std::endl;
```

Hello VexCL: vector sum

OpenCL

```
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

#define __CL_ENABLE_EXCEPTIONS
#include "CL/cl.hpp"

int main() {
    std::vector<cl::Platform> platform;
    cl::Platform::get(&platform);

    if (platform.empty()) {
        throw std::runtime_error("No OpenCL platforms");
    }

    cl::Context context;
    std::vector<cl::Device> device;
    for(auto p = platform.begin(); device.empty() && p != platform.end(); p++) {
        std::vector<cl::Device> dev;
        p->getDevices(CL_DEVICE_TYPE_GPU, &dev);
        for(auto d = dev.begin(); device.empty() && d != dev.end(); d++) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            device.push_back(*d);
            try {
                context = cl::Context(device);
            } catch(...) {
                device.clear();
            }
        }
    }
    if (device.empty()) throw std::runtime_error("No GPUs");

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

    const size_t n = 1024 * 1024;
    std::vector<double> a(n, 1.5), b(n, 2.7);

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

    cl::Buffer B(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
            b.size() * sizeof(double), b.data());
```

VexCL

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

int main() {
    vex::Context ctx( vex::Filter::GPU );
    std::cout << ctx << std::endl;

    size_t n = 1024 * 1024;
    std::vector<float> a(n, 1), b(n, 2);

    vex::vector<float> A(ctx, a);
    vex::vector<float> B(ctx, b);
```

Hello VexCL: vector sum

OpenCL

```
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

#define __CL_ENABLE_EXCEPTIONS
#include "CL/cl.hpp"

int main() {
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    if (platforms.empty()) {
        throw std::runtime_error("No OpenCL platforms");
    }

    cl::Context context;
    std::vector<cl::Device> devices;
    for(auto p = platforms.begin(); devices.empty() && p != platforms.end(); p++) {
        std::vector<cl::Device> devs;
        p->getDevices(CL_DEVICE_TYPE_GPU, &devs);
        for(auto d = devs.begin(); devices.empty() && d != devs.end(); d++) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            device pack back(*d);
            try {
                context = cl::Context(device);
            } catch(...) {
                device.clear();
            }
        }
    }
    if (device.empty()) throw std::runtime_error("No GPUs");

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

    const size_t n = 1024 * 1024;
    std::vector<double> a(n, 1.0), b(n, 2.7);

    cl::Buffer A(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
            n_size() * sizeof(double), &a.data());

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

    std::string source = R"(
        kernel void add(ulong n, global const double *a, global double *b) {
            ulong i = get_global_id(0);
            if (i < n) b[i] += a[i];
        }
    )";

    cl::Program program(context, source);

    try {
        program.build(device);
    } catch(cl::Error const&) {
        std::cout
            << "OpenCL compilation error" << std::endl
            << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device[0])
            << std::endl;
        throw std::runtime_error("OpenCL build error");
    }
    cl::Kernel add(program, "add");

    add.setArg(0, static_cast<cl_ulong>(n));
    add.setArg(1, A);
    add.setArg(2, B);

    queue.enqueueNDRangeKernel(add, cl::NullRange, n, cl::NullRange);
```

VexCL

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

int main() {
    vex::Context ctx( vex::Filter::GPU );
    std::cout << ctx << std::endl;

    size_t n = 1024 * 1024;
    std::vector<float> a(n, 1), b(n, 2);

    vex::vector<float> A(ctx, a);
    vex::vector<float> B(ctx, b);

    B += A;
```

Hello VexCL: vector sum

OpenCL

```
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

#define __CL_ENABLE_EXCEPTIONS
#include "CL/cl.hpp"

int main() {
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    if (platforms.empty()) {
        throw std::runtime_error("No OpenCL platforms");
    }

    cl::Context context;
    std::vector<cl::Device> devices;
    for(auto p = platforms.begin(); devices.empty() && p != platforms.end(); p++) {
        std::vector<cl::Device> devs;
        p->getDevices(CL_DEVICE_TYPE_GPU, &devs);
        for(auto d = devs.begin(); devices.empty() && d != devs.end(); d++) {
            if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;
            device.push_back(*d);
            try {
                context = cl::Context(device);
            } catch(...) {
                device.clear();
            }
        }
    }
    if (device.empty()) throw std::runtime_error("No GPUs");

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

    const size_t n = 1024 * 1024;
    std::vector<double> a(n, 1.0), b(n, 2.7);

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

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

    std::string source = R"(
        kernel void add(ulong n, global const double *a, global double *b) {
            ulong i = get_global_id(0);
            if (i < n) b[i] += a[i];
        }
    )";

    cl::Program program(context, source);

    try {
        program.build(device);
    } catch(cl::Error const&) {
        std::cout
            << "OpenCL compilation error" << std::endl
            << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device[0])
            << std::endl;
        throw std::runtime_error("OpenCL build error");
    }
    cl::Kernel add(program, "add");

    add.setArg(0, static_cast<cl_ulong>(n));
    add.setArg(1, A);
    add.setArg(2, B);

    queue.enqueueNDRangeKernel(add, cl::NullRange, n, cl::NullRange);

    queue.enqueueReadBuffer(B, CL_TRUE, 0, b.size() * sizeof(cl_double), b.data());
    std::cout << b[42] << std::endl;
}
```

VexCL

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

int main() {
    vex::Context ctx( vex::Filter::GPU );
    std::cout << ctx << std::endl;

    size_t n = 1024 * 1024;
    std::vector<float> a(n, 1), b(n, 2);

    vex::vector<float> A(ctx, a);
    vex::vector<float> B(ctx, b);

    B += A;

    vex::copy(B, b);
    std::cout << b[42] << std::endl;
}
```

Инициализация

- Поддерживается одновременная работа с несколькими устройствами.
- Контекст VexCL получает *фильтр устройств* при инициализации.
- Фильтр устройств — это булевский функтор, получающий ссылку на устройство.

Инициализируем контекст VexCL на выбранных устройствах

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

Инициализация

- Поддерживается одновременная работа с несколькими устройствами.
- Контекст VexCL получает *фильтр устройств* при инициализации.
- Фильтр устройств — это булевский функтор, получающий ссылку на устройство.

Инициализируем контекст VexCL на выбранных устройствах

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

Инициализация

- Поддерживается одновременная работа с несколькими устройствами.
- Контекст VexCL получает *фильтр устройств* при инициализации.
- Фильтр устройств — это булевский функтор, получающий ссылку на устройство.

Инициализируем контекст VexCL на выбранных устройствах

```
1 vex::Context ctx(vex::Filter::Accelerator && vex::Filter::Platform("Intel"));
```

Инициализация

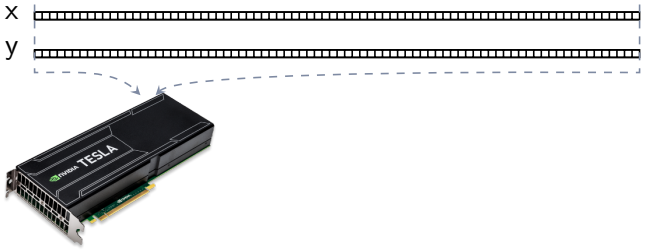
- Поддерживается одновременная работа с несколькими устройствами.
- Контекст VexCL получает *фильтр устройств* при инициализации.
- Фильтр устройств — это булевский функтор, получающий ссылку на устройство.

Инициализируем контекст VexCL на выбранных устройствах

```
1 vex::Context ctx(
2     vex::Filter::DoublePrecision &&
3     [](const vex::backend::device &d) {
4         return d.getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() >= 16_GB;
5     });
```

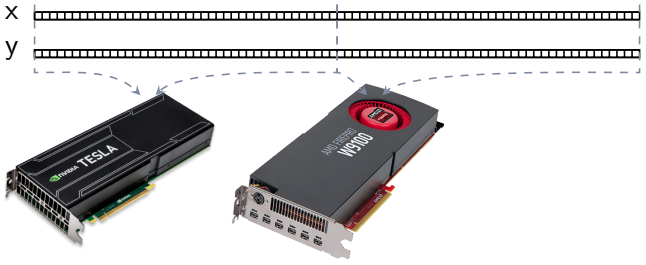
Распределение памяти и вычислений между картами

```
1  vex::Context ctx( vex::Filter::Name("Tesla") );
2
3  vex::vector<double> x(ctx, N);
4  vex::vector<double> y(ctx, N);
5
6  x = vex::element_index() * (1.0 / N);
7  y = sin(2 * x) + sqrt(1 - x * x);
```



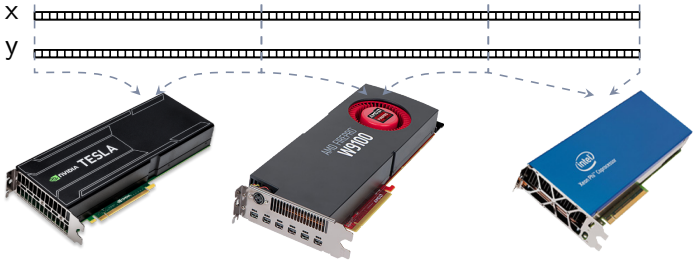
Распределение памяти и вычислений между картами

```
1  vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );
2
3  vex::vector<double> x(ctx, N);
4  vex::vector<double> y(ctx, N);
5
6  x = vex::element_index() * (1.0 / N);
7  y = sin(2 * x) + sqrt(1 - x * x);
```



Распределение памяти и вычислений между картами

```
1  vex::Context ctx( vex::Filter::DoublePrecision );
2
3  vex::vector<double> x(ctx, N);
4  vex::vector<double> y(ctx, N);
5
6  x = vex::element_index() * (1.0 / N);
7  y = sin(2 * x) + sqrt(1 - x * x);
```



Перенос данных

```
1 vex::vector<double> d(ctx, n);
2 std::vector<double> h(n);
3 double a[100];
```

Простые копии

```
1 vex::copy(d, h);
2 vex::copy(h, d);
```

Копирование диапазонов

```
1 vex::copy(d.begin(), d.end(), h.begin());
2 vex::copy(d.begin(), d.begin() + 100, a);
```

Поэлементный доступ (*медленно*)

```
1 double v = d[42];
2 d[0] = 0;
```

Отображение буффера OpenCL на хостовый указатель

```
1 auto p = d.map(devnum);
2 std::sort(&p[0], &p[d.part_size(devnum)]);
```

Язык векторных выражений VexCL

- Все векторы в выражении должны быть *совместимыми*:
  - Иметь один размер
  - Быть расположенными на одних и тех же устройствах
- Что можно использовать в выражениях:
  - Векторы, скаляры, константы
  - Арифм. и логич. операторы
  - Встроенные функции
  - Пользовательские функции
  - Генераторы случайных чисел
  - Сортировка, префиксные суммы
  - Временные значения
  - Срезы и перестановки
  - Редукция (сумма, экстремумы)
  - Произв. матрицы на вектор
  - Свертки
  - Быстрое преобразование Фурье

Встроенные операторы и функции

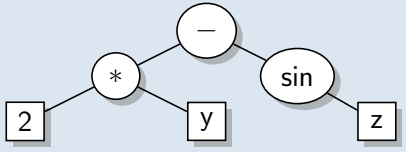
Выражение:

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

- **export** VEXCL_SHOW_KERNELS=1
  чтобы увидеть сгенерированный код.

. . . генерирует ядро:

```
1  kernel void vexcl_vector_kernel(
2      ulong n,
3      global double * prm_1,
4      int prm_2,
5      global double * prm_3,
6      global double * prm_4
7  )
8  {
9      for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
10         prm_1[idx] = ( ( prm_2 * prm_3[idx] ) - sin( prm_4[idx] ) );
11     }
12 }
```



Индексы элементов

- vex::element_index(**size_t** offset = 0, **size_t** size = 0)
  возвращает индекс текущего элемента вектора.
  - Нумерация начинается с offset, элементы на всех устройствах нумеруются последовательно.
  - Необязательный параметр size задает размер выражения.

Линейная функция:

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

Один период функции синуса:

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

Пользовательские функции

Определение функции:

```
1 VEX_FUNCTION( double, sqr, (double, x)(double, y),
2     return x * x + y * y;
3     );
```

Использование функции:

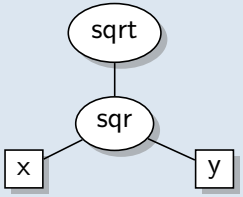
```
1 Z = sqrt( sqr(X, Y) );
```

Пользовательские функции транслируются в функции OpenCL

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

. . . ведет к генерации ядра:

```
1  double sqr(double x, double y) {
2      return x * x + y * y;
3  }
4
5  kernel void vexcl_vector_kernel(
6      ulong n,
7      global double * prm_1,
8      global double * prm_2,
9      global double * prm_3
10 )
11 {
12     for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
13         prm_1[idx] = sqrt( sqr( prm_2[idx], prm_3[idx] ) );
14     }
15 }
```



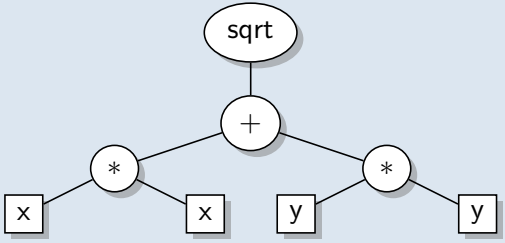
Функции часто не только удобны, но и эффективны

Тот же пример без использования функции:

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

. . . транслируется в:

```
1  kernel void vexcl_vector_kernel(
2    ulong n,
3    global double * prm_1,
4    global double * prm_2,
5    global double * prm_3,
6    global double * prm_4,
7    global double * prm_5
8  )
9  {
10   for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
11     prm_1[idx] = sqrt( ( ( prm_2[idx] * prm_3[idx] ) + ( prm_4[idx] * prm_5[idx] ) ) );
12   }
13 }
```



Выделение идентичных терминалов

- Программист может помочь VexCL узнать идентичные терминалы пометив их:

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

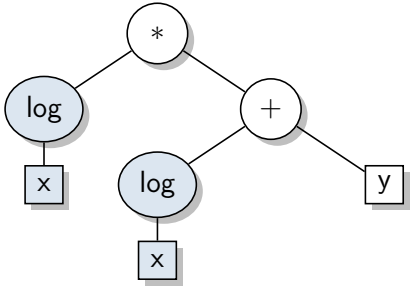
```
auto x = tag<1>(X);
auto y = tag<2>(Y);
Z = sqrt(x * x + y * y);
```

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

Повторное использование промежуточных результатов

```
1  z = log(x) * (log(x) + y);
```

- Некоторые выражения используют промежуточные результаты несколько раз.
- В простых случаях компилятор сможет избавиться от избыточного кода.



Промежуточные значения

- VexCL позволяет явно сохранить и использовать промежуточный результат:

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

```
kernel void vexcl_vector_kernel(
    ulong n,
    global double * prm_1,
    global double * prm_2,
    global double * prm_3
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        double temp_1 = log( prm_2[idx] );
        prm_1[idx] = ( temp_1 * ( temp_1 + prm_3[idx] ) );
    }
}
```

Перестановки

- vex::permutation(expr) принимает произвольное целочисленное выражение и возвращает функтор:

```
1 auto reverse = vex::permutation(N - 1 - vex::element_index(0, N));
2 y = reverse(x);
```

- Перестановки поддерживают как чтение, так и запись:

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

Пример: выделение вещественной и мнимой частей вектора

```
1 vex::vector<double> x(ctx, n * 2), y(ctx, n);
2
3 auto Re = vex::permutation(2 * vex::element_index(0, n));
4 auto Im = vex::permutation(2 * vex::element_index(0, n) + 1);
5
6 y = sqr(Re(x), Im(x));
```

Срезы многомерных массивов

- При работе с многомерными массивами данные эффективнее всего располагать в непрерывных одномерных массивах.
  - Класс vex::slicer <NDIM> позволяет работать со срезами таких массивов.

Матрица $n \times m$ и slicer:

```
1 vex::vector<double> x(ctx, n * m);
2 vex::slicer<2> slice(vex::extents[n][m]);
```

Срезы многомерных массивов

- При работе с многомерными массивами данные эффективнее всего располагать в непрерывных одномерных массивах.
  - Класс vex:: slicer <NDIM> позволяет работать со срезами таких массивов.

Матрица $n \times m$ и slicer:

```
1 vex::vector<double> x(ctx, n * m);
2 vex:: slicer <2> slice(vex::extents[n][m]);
```

Доступ к строке или столбцу матрицы:

```
3 using vex::_;
4 y = slice[42](x);          // строка
5 y = slice[_][42](x);       // столбец
6 slice[_][10](x) = y;
```

Срезы многомерных массивов

- При работе с многомерными массивами данные эффективнее всего располагать в непрерывных одномерных массивах.
  - Класс vex::slicer <NDIM> позволяет работать со срезами таких массивов.

Матрица $n \times m$ и slicer:

```
1 vex::vector<double> x(ctx, n * m);
2 vex::slicer<2> slice(vex::extents[n][m]);
```

Доступ к строке или столбцу матрицы:

```
3 using vex::_;
4 y = slice[42](x);          // строка
5 y = slice[_][42](x);       // столбец
6 slice[_][10](x) = y;
```

Использование диапазонов для выделения подблоков:

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

Тензорное произведение

Тензорное произведение — обобщенная версия скалярного произведения. Элементы двух многомерных массивов перемножаются и суммируются вдоль заданных осей.

Произведение матрицы на вектор

```
1 vex::vector<double> A(ctx, n * m), x(ctx, m), y(ctx, n);
2 vex:: slicer <2> Adim(vex::extents[n][m]);
3 vex:: slicer <1> xdim(vex::extents[m]);
4
5 y = vex::tensordot(Adim[_](A), xdim[_](x), vex::axes_pairs(1, 0));
```

Произведение двух матриц

```
1 vex::vector<double> A(ctx, n * m), B(ctx, m * k), C(ctx, n * k);
2 vex:: slicer <2> Adim(vex::extents[n][m]);
3 vex:: slicer <2> Bdim(vex::extents[m][k]);
4
5 C = vex::tensordot(Adim[_](A), Bdim[_](B), vex::axes_pairs(1, 0));
```

Генерация случайных чисел

- В VexCL реализованы позиционные генераторы случайных чисел[1] (counter-based random number generators).
  - Такие генераторы не имеют состояния и позволяют получить случайное число по его номеру (позиции) за O(1).
  - Реализованные семейства: threefry и philox.
  - Удовлетворяют тестам TestU01/BigCrush; позволяют получить до 2^{64} независимых последовательностей с периодом 2^{128}.
  - Производительность: $\approx 2 \times 10^{10}$ чисел/сек (Tesla K40c).
- vex::Random<T, G=philox> — равномерное распределение.
- vex::RandomNormal<T, G=philox> — нормальное распределение.

```
1 vex::Random<double> rnd;
2 vex::vector<double> x(ctx, n);
3
4 x = rnd(vex::element_index(), std::rand());
```

[1] Random123 suite, D. E. Shaw Research, deshawresearch.com/resources_random123.html

Редукция

- Класс vex::Reductor<T, kind=SUM> позволяет редуцировать произвольное *векторное выражение* и получить значение типа T.
- Виды редукции: SUM, SUM_Kahan, MIN, MAX

Скалярное произведение

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

Число элементов в интервале (0, 1)

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

Максимальное расстояние от центра

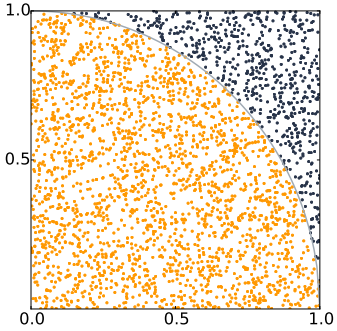
```
1 vex::Reductor<double, vex::MAX> max(ctx);
2 double d = max( sqrt(x * x + y * y) );
```

Пример: число π методом Монте-Карло

- Приближенная оценка π:

$$\frac{\text{площадь круга}}{\text{площадь квадрата}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4},$$

$$\pi = 4\frac{\text{площадь круга}}{\text{площадь квадрата}} \approx 4\frac{\text{точки в круге}}{\text{все точки}}$$




```
1 vex::Random<cl_double2> rnd;
2 vex::Reductor<size_t, vex::SUM> sum(ctx);
3
4 double pi = 4.0 * sum( length( rnd(vex::element_index(0, n), seed) ) < 1 ) / n;
```

Монте-Карло π: сгенерированное ядро

```
#if defined(cl_khr_fp64)
#pragma OPENCL EXTENSION cl_khr_fp64: enable
#elif defined(cl_amd_fp64)
#pragma OPENCL EXTENSION cl_amd_fp64: enable
#endif

void philox_uint_4_10
(
  uint * ctr,
  uint * key
)
{
  uint m[4];
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
  key[0] += 0x9E3779B9;
  key[1] += 0xBB67AE85;
  m[0] = mul_hi(0xD2511F53, ctr[0]);
  m[1] = 0xD2511F53 * ctr[0];
  m[2] = mul_hi(0xCD9E8D57, ctr[2]);
  m[3] = 0xCD9E8D57 * ctr[2];
  ctr[0] = m[2] ^ ctr[1] ^ key[0];
  ctr[1] = m[3];
  ctr[2] = m[0] ^ ctr[3] ^ key[1];
  ctr[3] = m[1];
}
double2 random_double2_philox
(
  ulong prm1,
  ulong prm2
)
{
  union
  {
    uint ctr[4];
    ulong res_i[2];
    double res_f[2];
    double2 res;
  } ctr;
  uint key[2];
  ctr.ctr[0] = prm1; ctr.ctr[1] = prm2;
  ctr.ctr[2] = prm1; ctr.ctr[3] = prm2;
  key[0] = 0x12345678;
  key[1] = 0x12345678;
  philox_uint_4_10(ctr.ctr, key);
  ctr.res_f[0] = ctr.res_i[0] / 18446744073709551615.0;
  ctr.res_f[1] = ctr.res_i[1] / 18446744073709551615.0;
  return ctr.res;
}
ulong SUM_ulong
(
  ulong prm1,
  ulong prm2
)
{
  return prm1 + prm2;
}
kernel void vexcl_reductor_kernel
(
  ulong n,
  ulong prm_1,
  ulong prm_2,
  double prm_3,
  global ulong * g_odata,
  local ulong * smem
)
{
  ulong mySum = (ulong)0;
  for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0))
  {
    mySum = SUM_ulong(mySum, ( length( random_double2_philox( (prm_1 + idx), prm_2 ) ) < prm_3 ));
  }
  local ulong * sdata = smem;
  size_t tid = get_local_id(0);
  size_t block_size = get_local_size(0);
  sdata[tid] = mySum;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (block_size >= 1024)
  {
    if (tid < 512) { sdata[tid] = mySum = SUM_ulong(mySum, sdata[tid + 512]); }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (block_size >= 512)
  {
    if (tid < 256) { sdata[tid] = mySum = SUM_ulong(mySum, sdata[tid + 256]); }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (block_size >= 256)
  {
    if (tid < 128) { sdata[tid] = mySum = SUM_ulong(mySum, sdata[tid + 128]); }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (block_size >= 128)
  {
    if (tid < 64) { sdata[tid] = mySum = SUM_ulong(mySum, sdata[tid + 64]); }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid < 32)
  {
    volatile local ulong * smem = sdata;
    if (block_size >= 64) { smem[tid] = mySum = SUM_ulong(mySum, smem[tid + 32]); }
    if (block_size >= 32) { smem[tid] = mySum = SUM_ulong(mySum, smem[tid + 16]); }
    if (block_size >= 16) { smem[tid] = mySum = SUM_ulong(mySum, smem[tid + 8]); }
    if (block_size >= 8) { smem[tid] = mySum = SUM_ulong(mySum, smem[tid + 4]); }
    if (block_size >= 4) { smem[tid] = mySum = SUM_ulong(mySum, smem[tid + 2]); }
    if (block_size >= 2) { smem[tid] = mySum = SUM_ulong(mySum, smem[tid + 1]); }
  }
  if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
}
```

Произведение разреженной матрицы на вектор

- Класс vex::SpMat<T> содержит представление разреженной матрицы на вычислительных устройствах.
- Конструктор принимает матрицу в формате CRS:
  - Номера столбцов и значения ненулевых элементов, указатели на начало каждой строки.

Конструирование матрицы

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

Расчет невязки СЛАУ

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

Арифметика указателей

- Функция raw_pointer(**const** vector<T>&) возвращает указатель на данные вектора.
  - Может использоваться для реализации случайного доступа

Одномерный оператор Лапласа:

```
1 VEX_FUNCTION(double, laplace, (size_t, i)(size_t, n)(double*, x),
2     if (i == 0 || i == n-1) return 0;
3     return 2 * x[i] - x[i-1] - x[i+1];
4     );
5
6 y = laplace(vex::element_index(), n, vex::raw_pointer(x));
```

Пример: взаимодействие N тел

$$y_i = \sum_{j \neq i} e^{-|x_i - x_j|}$$

```
1  VEX_FUNCTION(double, nbody, (size_t, i)(size_t, n)(double*, x),
2      double sum = 0, myval = x[i];
3      for(size_t j = 0; j < n; ++j)
4          if (j != i) sum += exp(-fabs(x[j] - myval));
5      return sum;
6      );
7
8  y = nbody(vex::element_index(), x.size(), raw_pointer(x));
```

Быстрое преобразование Фурье

- Произвольное выражение на входе
- Преобразование многомерных массивов
- Произвольные размеры массивов
- Пакетное преобразование (batch transform)
- В 2-3 медленнее NVIDIA CUFFT, сравнимо по скорости с AMD clFFT

Решение периодической задачи Пуассона:

```
1  vex::vector<double> rhs(ctx, n), u(ctx, n), K(ctx, n);
2
3  vex::FFT<double, cl_double2> fft(ctx, n);
4  vex::FFT<cl_double2, double> ifft(ctx, n, vex::inverse);
5
6  u = ifft( K * fft(rhs) );
```

Мультивекторы

- Класс vex::multivector<T,N> владеет N экземплярами класса vex::vector<T>.
- Поддерживает все операции, определенные для vex::vector<T>.
- Операции с мультивекторами выполняются в единственном ядре.
- vex::multivector::**operator**()(**size_t** k) возвращает k-ю компоненту.

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

Мультивыражения

- Некоторые операции не удается выразить через арифметику мультивекторов.

Пример: поворот 2D вектора на заданный угол

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

- Мультивыражение — это кортеж (tuple) векторых выражений.
- Присваивание мультивыражения мультивектору эквивалентно покомпонентному присваиванию, но выполняется в единственном ядре.

- Мультивыражения можно присваивать мультивекторам:

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

- и кортежу выражений:

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

Мультивыражение транслируется в единственное ядро

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

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

```
kernel void vexcl_multivector_kernel(ulong n,
    global double * lhs_1, global double * lhs_2,
    global double * rhs_1, double rhs_2,
    global double * rhs_3, double rhs_4
)
{
    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
        double buf_1 = ( ( rhs_1[idx] * rhs_2 ) - ( rhs_3[idx] * rhs_4 ) );
        double buf_2 = ( ( rhs_1[idx] * rhs_4 ) + ( rhs_3[idx] * rhs_2 ) );

        lhs_1[idx] = buf_1;
        lhs_2[idx] = buf_2;
    }
}
```

Пользовательские ядра

- При использовании собственных ядер программист сам управляет обменом между картами.

Создание ядра

```
1 vex::backend::kernel dummy(ctx.queue(0), R"(
2     int the_answer(ulong i) { return 42; }
3     kernel void dummy(ulong n, global int *x) {
4         for(ulong i = get_global_id(0); i < n; i += get_global_size(0))
5             x[i] = the_answer(i);
6     })",
7     "dummy"
8     );
```

Пользовательские ядра

- При использовании собственных ядер программист сам управляет обменом между картами.

Создание ядра

```
vex::backend::kernel dummy(ctx.queue(0), VEX_STRINGIZE_SOURCE(
    int the_answer(ulong i) { return 42; }
    kernel void dummy(ulong n, global int *x) {
        for(ulong i = get_global_id(0); i < n; i += get_global_size(0))
            x[i] = the_answer(i);
    }),
    "dummy"
    );
```

Пользовательские ядра

- При использовании собственных ядер программист сам управляет обменом между картами.

Создание ядра

```
vex::backend::kernel dummy(ctx.queue(0), VEX_STRINGIZE_SOURCE(
    int the_answer(ulong i) { return 42; }
    kernel void dummy(ulong n, global int *x) {
        for(ulong i = get_global_id(0); i < n; i += get_global_size(0))
            x[i] = the_answer(i);
    }),
    "dummy"
    );
```

Вызов ядра

```
vex::vector<int> x(ctx, n);
dummy(ctx.queue(0), static_cast<cl_ulong>(n), x(0));
```

- VexCL позволяет писать компактный и читаемый код
  - Хорошо подходит для быстрой разработки научных GPGPU приложений.
  - Производительность часто сравнима с ядрами, написанными вручную.

[1] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling.
Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries.
SIAM J. Sci. Comput., 35(5):C453 – C472, 2013.
doi:10.1137/120903683

[2] K. Ahnert, D. Demidov, and M. Mulansky.
Solving ordinary differential equations on GPUs.
In *Numerical Computations with GPUs* (pp. 125-157). Springer, 2014.
doi:10.1007/978-3-319-06548-9_7

Как это работает?

Шаблоны выражений

Шаблоны выражений

- Как *эффективно* реализовать предметно-ориентированный язык (DSL) в C++?

- Идея не нова:
  - *Todd Veldhuizen*, Expression templates, C++ Report, 1995
- Первая реализация:
  - "***Blitz++*** — *библиотека классов* C++ *для научных расчетов, имеющая производительность сравнимую с Фортраном 77/90* ".
- Сегодня:
  - Boost.uBLAS, Blaze, MTL, Eigen, Armadillo, и пр.

- *Как это работает?*

Простой пример: сложение векторов

Мы хотим иметь возможность написать:

```
1 x = y + z;
```

чтобы результат был так же эффективен как:

```
1 for(size_t i = 0; i < n; ++i)
2     x[i] = y[i] + z[i];
```

C++ допускает перегрузку операторов!

```
1  vector operator+(const vector &a, const vector &b) {
2      vector tmp( a.size() );
3      for(size_t i = 0; i < a.size(); ++i)
4          tmp[i] = a[i] + b[i];
5      return tmp;
6  }
```

C++ допускает перегрузку операторов!

```
1 vector operator+(const vector &a, const vector &b) {
2     vector tmp( a.size() );
3     for(size_t i = 0; i < a.size(); ++i)
4         tmp[i] = a[i] + b[i];
5     return tmp;
6 }
```

- Проблемы:
  - Дополнительное выделение памяти
  - Дополнительные операции чтения/записи

```
1 a = x + y + z;
```

- 2 временных вектора
- $8 \times n$ операций чтения/записи

```
1 for(size_t i = 0; i < n; ++i)
2     a[i] = x[i] + y[i] + z[i];
```

- нет временных векторов
- $4 \times n$ операций чтения/записи

Отложенный расчет v0.1

Идея: отложим расчет результата до операции присваивания.

Отложенный расчет v0.1

Идея: отложим расчет результата до операции присваивания.

```
1  struct vsum {
2      const vector &lhs;
3      const vector &rhs;
4  };
5
6  vsum operator+(const vector &a, const vector &b) {
7      return vsum{a, b};
8  }
```

Отложенный расчет v0.1

Идея: отложим расчет результата до операции присваивания.

```
1  struct vsum {
2      const vector &lhs;
3      const vector &rhs;
4  };
5
6  vsum operator+(const vector &a, const vector &b) {
7      return vsum{a, b};
8  }
9
10 const vector& vector::operator=(const vsum &s) {
11     for(size_t i = 0; i < data.size(); ++i)
12         data[i] = s.lhs[i] + s.rhs[i];
13     return *this;
14 }
```

Решение недостаточно универсально

Следующее выражение приведет к ошибке компиляции:

```
1 a = x + y + z;
```

```
lazy_v1.cpp:38:15: error: invalid operands to binary expression
      ('vsum' and 'vector')
    a = x + y + z;
        ~~~~~ ^ ~
lazy_v1.cpp:12:12: note: candidate function not viable:
      no known conversion from 'vsum' to 'const vector' for 1st argument
vsum operator+(const vector &a, const vector &b) {
     ^
1 error generated.
```

Отложенный расчет v0.2

```
1  template <class LHS, class RHS>
2  struct vsum {
3      const LHS &lhs;
4      const RHS &rhs;
5
6      double operator[ ](size_t i) const { return lhs[i] + rhs[i]; }
7  };
```

Отложенный расчет v0.2

```
1  template <class LHS, class RHS>
2  struct vsum {
3      const LHS &lhs;
4      const RHS &rhs;
5
6      double operator[ ](size_t i) const { return lhs[i] + rhs[i]; }
7  };
8
9  template <class LHS, class RHS>
10 vsum<LHS, RHS> operator+(const LHS &a, const RHS &b) {
11     return vsum<LHS, RHS>{a, b};
12 }
```

Отложенный расчет v0.2

```
1  template <class LHS, class RHS>
2  struct vsum {
3      const LHS &lhs;
4      const RHS &rhs;
5
6      double operator[ ](size_t i) const { return lhs[i] + rhs[i]; }
7  };
8
9  template <class LHS, class RHS>
10 vsum<LHS, RHS> operator+(const LHS &a, const RHS &b) {
11     return vsum<LHS, RHS>{a, b};
12 }
13
14 template<class Expr>
15 const vector& vector::operator=(const Expr &expr) {
16     for(int i = 0; i < data.size(); ++i) data[i] = expr[i];
17     return *this;
18 }
```

Добавим остальные операции

```
1 struct plus {
2     static double apply(double a, double b) { return a + b; }
3 };
```

Добавим остальные операции

```
1  struct plus {
2      static double apply(double a, double b) { return a + b; }
3  };
4
5  template <class LHS, class OP, class RHS>
6  struct binary_op {
7      const LHS &lhs;
8      const RHS &rhs;
9
10     double operator[ ](size_t i) const { return OP::apply(lhs[i], rhs[i]); }
11 };
```

Добавим остальные операции

```
1  struct plus {
2      static double apply(double a, double b) { return a + b; }
3  };
4
5  template <class LHS, class OP, class RHS>
6  struct binary_op {
7      const LHS &lhs;
8      const RHS &rhs;
9
10     double operator[ ](size_t i) const { return OP::apply(lhs[i], rhs[i]); }
11 };
12
13 template <class LHS, class RHS>
14 binary_op<LHS, plus, RHS> operator+(const LHS &a, const RHS &b) {
15     return binary_op<LHS, plus, RHS>{a, b};
16 }
```

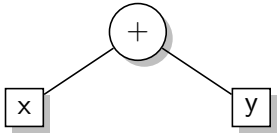
Шаблоны выражений — это деревья

Выражение в правой части:

```
1 a = x + y;
```

... имеет тип:

```
binary_op<
    vector,
    plus,
    vector
>
```



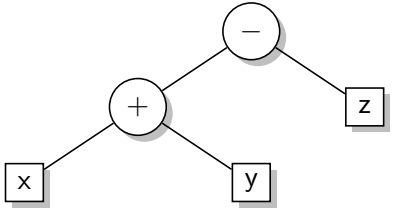
Шаблоны выражений — это деревья

Выражение в правой части:

```
1 a = x + y - z;
```

... имеет тип:

```
binary_op<
    binary_op<
        vector,
        plus,
        vector
    >
    , minus
    , vector
>
```



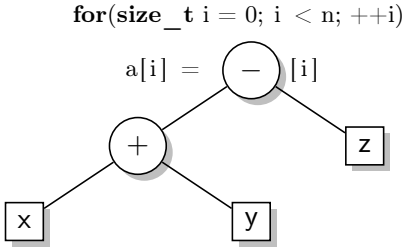
Шаблоны выражений — это деревья

Выражение в правой части:

```
1 a = x + y - z;
```

... имеет тип:

```
binary_op<
    binary_op<
        vector,
        plus,
        vector
    >
    , minus
    , vector
>
```



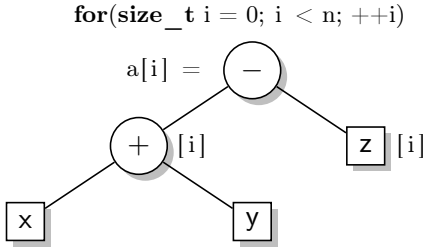
Шаблоны выражений — это деревья

Выражение в правой части:

```
1 a = x + y - z;
```

... имеет тип:

```
binary_op<
    binary_op<
        vector,
        plus,
        vector
    >
    , minus
    , vector
>
```



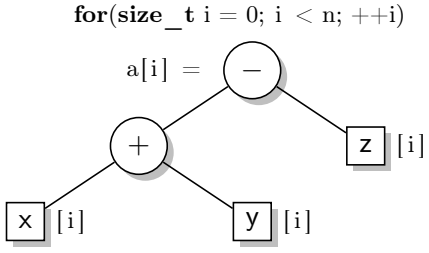
Шаблоны выражений — это деревья

Выражение в правой части:

```
1 a = x + y - z;
```

... имеет тип:

```
binary_op<
    binary_op<
        vector,
        plus,
        vector
    >
    , minus
    , vector
>
```



Шаблоны выражений — это деревья

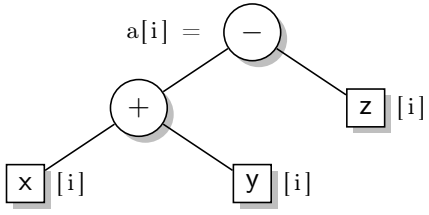
Выражение в правой части:

```
1 a = x + y - z;
```

... имеет тип:

```
binary_op<
    binary_op<
        vector,
        plus,
        vector
    >
    , minus
    , vector
>
```

```
#pragma omp parallel for
for(size_t i = 0; i < n; ++i)
```



Шаблоны выражений — это деревья

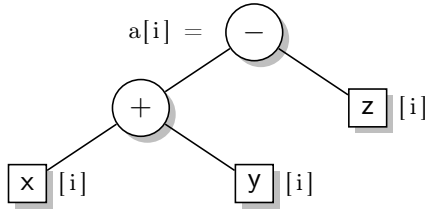
Выражение в правой части:

```
1 a = x + y - z;
```

... имеет тип:

```
binary_op<
    binary_op<
        vector,
        plus,
        vector
    >
    , minus
    , vector
>
```

```
#pragma omp parallel for
for(size_t i = 0; i < n; ++i)
```



- C++ компилятор обходит дерево.
- Вся работа выполняется в binary_op::**operator**[].

Промежуточный итог

Теперь мы можем записать:

```
1  v = a * x + b * y;
2
3  double c = (x + y)[42];
```

... и это будет так же эффективно, как:

```
1  for(size_t i = 0; i < n; ++i)
2      v[i] = a[i] * x[i] + b[i] * y[i];
3
4  double c = x[42] + y[42];
```

- Дополнительная память не требуется
- Накладные расходы пропадут при компиляции с оптимизацией

Промежуточный итог

Теперь мы можем записать:

```
1  v = a * x + b * y;
2
3  double c = (x + y)[42];
```

... и это будет так же эффективно, как:

```
1  for(size_t i = 0; i < n; ++i)
2      v[i] = a[i] * x[i] + b[i] * y[i];
3
4  double c = x[42] + y[42];
```

- Дополнительная память не требуется
- Накладные расходы пропадут при компиляции с оптимизацией

- *Но как это связано с OpenCL?*

Генерация кода OpenCL

Как работает OpenCL?

1. Вычислительное ядро компилируется во время выполнения из $\mathrm{C99}$ кода.
2. Параметры ядра задаются вызовами API.
3. Ядро выполняется на вычислительном устройстве.

Как работает OpenCL?

1. Вычислительное ядро компилируется во время выполнения из C99 кода.
2. Параметры ядра задаются вызовами API.
3. Ядро выполняется на вычислительном устройстве.

- Исходный код ядра можно считать из файла, из статической текстовой переменной, или *сгенерировать*.

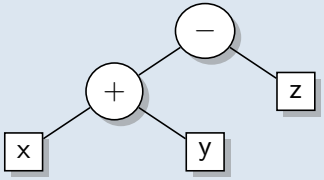
Генерация исходного кода ядра из выражений C++

Следующее выражение:

```
1  a = x + y - z;
```

. . . должно привести к генерации ядра:

```
1  kernel void vexcl_vector_kernel(
2      ulong n,
3      global double * res,
4      global double * prm1,
5      global double * prm2,
6      global double * prm3
7  )
8  {
9      for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {
10         res[idx] = ( ( prm1[idx] + prm2[idx] ) - prm3[idx] );
11     }
12 }
```



Объявление параметров

Каждый терминал знает, какие параметры ему нужны:

```
1  /*static*/ void vector::prm_decl(std::ostream &src, unsigned &pos) {
2      src << ",\n    global double * prm" << ++pos;
3  }
```

Выражение просто делегирует работу своим терминалам:

```
4  template <class LHS, class OP, class RHS>
5  /*static*/ void binary_op<LHS, OP, RHS>::declare_params(
6                                  std::ostream &src, unsigned &pos)
7  {
8      LHS::declare_params(src, pos);
9      RHS::declare_params(src, pos);
10 }
```

Построение строкового представления выражения

```
1 struct plus {
2     static void to_string(std::ostream &src) { src << " + "; }
3 };
```

Построение строкового представления выражения

```
1  struct plus {
2      static void to_string(std::ostream &src) { src << " + "; }
3  };
4
5  /*static*/ void vector::to_string(std::ostream &src, unsigned &pos) {
6      src << "prm" << ++pos << "[idx]";
7  }
```

Построение строкового представления выражения

```
1  struct plus {
2      static void to_string(std::ostream &src) { src << " + "; }
3  };
4
5  /*static*/ void vector::to_string(std::ostream &src, unsigned &pos) {
6      src << "prm" << ++pos << "[idx]";
7  }
8
9  template <class LHS, class OP, class RHS>
10 /*static*/ void binary_op<LHS, OP, RHS>::to_string(
11                                 std::ostream &src, unsigned &pos) const
12 {
13     src << "( ";
14     LHS::to_string(src, pos);
15     OP::to_string(src);
16     RHS::to_string(src, pos);
17     src << " )";
18 }
```

Генерация исходного кода ядра

```
1  template <class LHS, class RHS>
2  std::string kernel_source() {
3      std::ostringstream src;
4
5      src << "kernel void vexcl_vector_kernel(\n   ulong n";
6      unsigned pos = 0;
7      LHS::declare_params(src, pos);
8      RHS::declare_params(src, pos);
9      src << ")\n{\n"
10         "      for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {\n"
11         "          ";
12     pos = 0;
13     LHS::to_string(src, pos); src << " = ";
14     RHS::to_string(src, pos); src << ";\n";
15     src << "    }\n}\n";
16
17     return src.str();
18 }
```

Генерация исходного кода ядра

```
1  template <class LHS, class RHS>
2  std::string kernel_source() {
3      std::ostringstream src;
4
5      src << "kernel void vexcl_vector_kernel(\n    ulong n";
6      unsigned pos = 0;
7      LHS::declare_params(src, pos);
8      RHS::declare_params(src, pos);
9      src << ")\n{\n"
10         "    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {\n"
11         "        ";
12     pos = 0;
13     LHS::to_string(src, pos); src << " = ";
14     RHS::to_string(src, pos); src << ";\n";
15     src << "    }\n}\n";
16
17     return src.str();
18 }
```

Генерация исходного кода ядра

```
1  template <class LHS, class RHS>
2  std::string kernel_source() {
3      std::ostringstream src;
4
5      src << "kernel void vexcl_vector_kernel(\n   ulong n";
6      unsigned pos = 0;
7      LHS::declare_params(src, pos);
8      RHS::declare_params(src, pos);
9      src << ")\n{\n"
10         "     for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {\n"
11         "         ";
12     pos = 0;
13     LHS::to_string(src, pos); src << " = ";
14     RHS::to_string(src, pos); src << ";\n";
15     src << "    }\n}\n";
16
17     return src.str();
18 }
```

Задание параметров ядра

```
1  void vector::set_args(cl::Kernel &krn, unsigned &pos) {
2      krn.setArg(pos++, buffer);
3  }
4
5  template <class LHS, class OP, class RHS>
6  void binary_op<LHS, OP, RHS>::set_args(cl::Kernel &krn, unsigned &pos) {
7      lhs.set_args(krn, pos);
8      rhs.set_args(krn, pos);
9  }
```

- Методы уже не статические!

Объединяем все компоненты

```
1  template <class Expr>
2  const vector& vector::operator=(const Expr &expr) {
3      static cl::Kernel kernel = build_kernel(device, kernel_source<This, Expr>());
4
5      unsigned pos = 0;
6
7      kernel.setArg(pos++, size);      // размер
8      kernel.setArg(pos++, buffer);    // результат
9      expr.set_args(kernel, pos);      // параметры
10
11     queue.enqueueNDRangeKernel(kernel, cl::NullRange, buffer.size(), cl::NullRange);
12
13     return *this;
14 }
```

Объединяем все компоненты

```
1  template <class Expr>
2  const vector& vector::operator=(const Expr &expr) {
3      static cl::Kernel kernel = build_kernel(device, kernel_source<This, Expr>());
4
5      unsigned pos = 0;
6
7      kernel.setArg(pos++, size);      // размер
8      kernel.setArg(pos++, buffer);    // результат
9      expr.set_args(kernel, pos);      // параметры
10
11     queue.enqueueNDRangeKernel(kernel, cl::NullRange, buffer.size(), cl::NullRange);
12
13     return *this;
14 }
```

- Ядро генерируется и компилируется однажды, применяется множество раз:
  - Каждое ядро однозначно определяется типом выражения.
  - Можем использовать локальную статическую переменную для кеширования ядра.

На самом деле все не (совсем) так

- Фактическая реализация немного сложнее:
  - Кроме векторов, есть другие терминалы (скаляры, константы, ...)
  - Унарные, бинарные, *n*-арные выражения
  - Специальные терминалы, требующие задания преамбулы в коде ядра
  - Встроенные и пользовательские функции
  - ...
- Для упрощения работы с шаблонами выражений используется Boost.Proto.

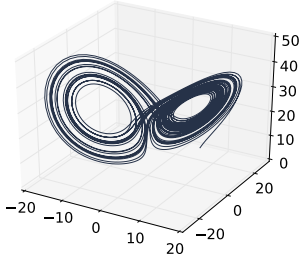
Оценка производительности

Параметрическое исследование системы Лоренца

Система Лоренца

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

- Необходимо решать большое число систем Лоренца для различных значений R.
- Будем использовать Boost.odeint.



Boost.odeint

Общий вид ОДУ

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

Применение Boost.odeint:

1. Определяем тип переменной состояния (что такое x?)
2. Определяем системную функцию (f)
3. Выбираем алгоритм интегрирования
4. Выполняем интегрирование по времени

Простейшая реализация с VexCL

1. Тип переменной состояния

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

2. Системная функция

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

3. Алгоритм (Рунге-Кутты 4-го порядка)

```
11 odeint :: runge_kutta4<
12         state_type /*state*/,        double /*value*/,
13         state_type /*derivative*/, double /*time*/,
14         odeint :: vector_space_algebra, odeint::default_operations
15         > stepper;
```

4. Интегрирование

```
16 vex :: multivector<double,3> X(ctx, n);
17 vex :: vector<double> R(ctx, n);
18
19 X = 10;
20 R = Rmin + vex::element_index() * ((Rmax - Rmin) / (n - 1));
21
22 odeint :: integrate_const(stepper, lorenz_system{R}, X, 0.0, t_max, dt);
```

Вариант с использованием CUBLAS

- CUBLAS — оптимизированная библиотека линейной алгебры от NVIDIA.
- CUBLAS имеет фиксированный программный интерфейс.
- Линейные комбинации (используемые в алгоритмах Boost.odeint):

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

реализованы следующим образом:

```
cublasDset (...);        // x0 = 0
cublasDaxpy(...);        // x0 = x0 + α1 * x1
 ...
cublasDaxpy(...);        // x0 = x0 + αn * xn
```

Вариант с использованием Thrust

- Библиотека Thrust позволяет получить монолитное ядро:

Thrust

```
1  struct scale_sum2 {
2      const double a1, a2;
3      scale_sum2(double a1, double a2) : a1(a1), a2(a2) { }
4      template<class Tuple>
5      __host__ __device__ void operator()(Tuple t) const {
6          thrust::get<0>(t) = a1 * thrust::get<1>(t) + a2 * thrust::get<2>(t);
7      }
8  };
9
10 thrust::for_each(
11         thrust::make_zip_iterator(
12             thrust::make_tuple( x0.begin(), x1.begin(), x2.begin() )
13             ),
14         thrust::make_zip_iterator(
15             thrust::make_tuple( x0.end(), x1.end(), x2.end() )
16             ),
17         scale_sum2(a1, a2)
18         );
```

Вариант с использованием Thrust

- Библиотека Thrust позволяет получить монолитное ядро:

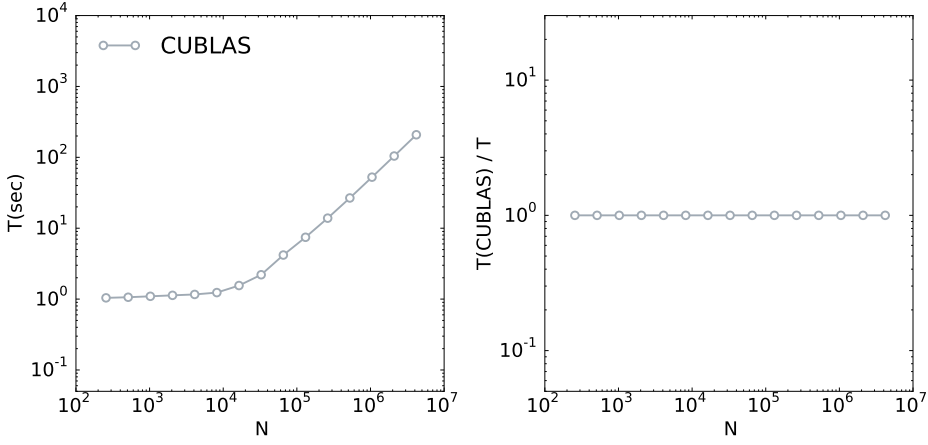
Thrust

```
1  struct scale_sum2 {
2      const double a1, a2;
3      scale_sum2(double a1, double a2) : a1(a1), a2(a2) { }
4      template<class Tuple>
5      __host__ __device__ void operator()(Tuple t) const {
6          thrust::get<0>(t) = a1 * thrust::get<1>(t) + a2 * thrust::get<2>(t);
7      }
8  };
9
10 thrust::for_each(
11         thrust::make_zip_iterator(
12             thrust::make_tuple( x0.begin(), x1.begin(), x2.begin() )
13             ),
14         thrust::make_zip_iterator(
15             thrust::make_tuple( x0.end(), x1.end(), x2.end() )
16             ),
17         scale_sum2(a1, a2)
18         );
```

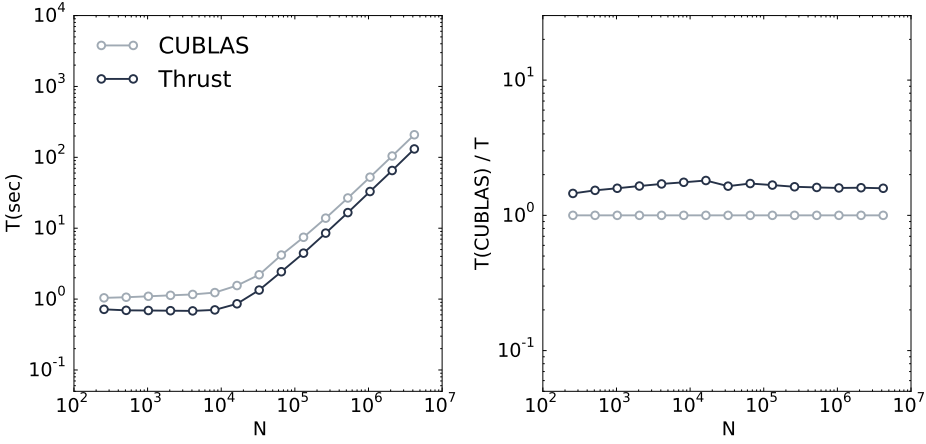
VexCL

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

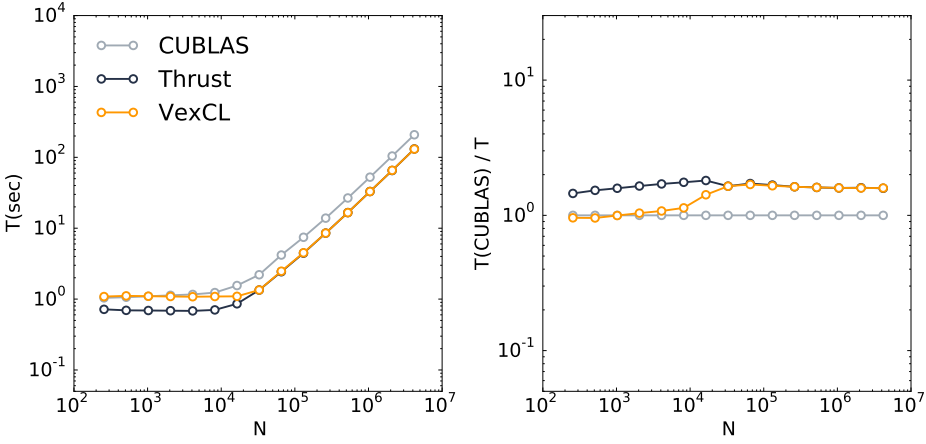
Производительность (Tesla K40c)



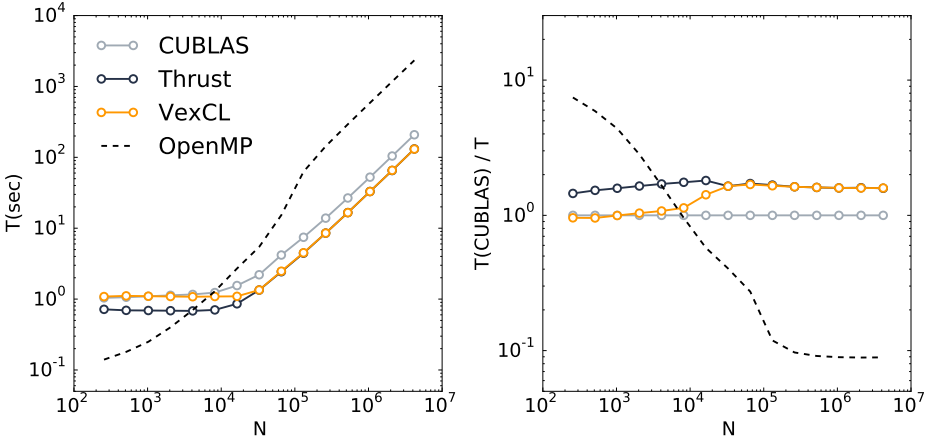
Производительность (Tesla K40c)



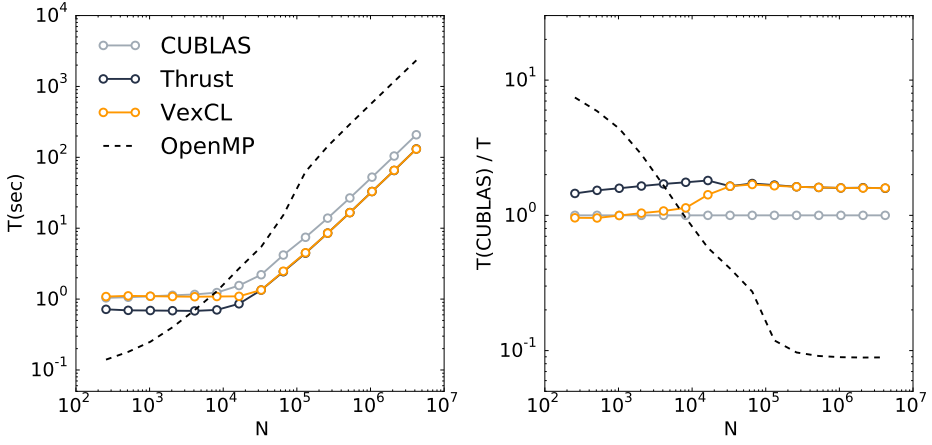
Производительность (Tesla K40c)



Производительность (Tesla K40c)



Производительность (Tesla K40c)



- Недостатки простейшей реализации:
  - Метод Рунге-Кутты использует 4 временных переменных состояния.
  - Одна итерация метода приводит к запуску нескольких вычислительных ядер.

Специально написанное ядро

- Создадим монолитное ядро, соответствующее одной итерации Рунге-Кутты.
- Будем вызывать это ядро в цикле по времени.
- Получим 10-кратное ускорение!

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

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

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

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

Специально написанное ядро

- Создадим монолитное ядро, соответствующее одной итерации Рунге-Кутты.
- Будем вызывать это ядро в цикле по времени.
- Получим 10-кратное ускорение!
  - *Потеряв универсальность odeint*

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

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

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

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

Генерация OpenCL кода из алгоритмов Boost.odeint

- VexCL определяет тип vex::symbolic<T>.
- Любые операции с переменными этого типа выводятся в текстовый поток:

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

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

Генерация OpenCL кода из алгоритмов Boost.odeint

- VexCL определяет тип vex::symbolic<T>.
- Любые операции с переменными этого типа выводятся в текстовый поток:

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

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

- Простая идея:
  - Запишем последовательность действий алгоритма.
  - Сгенерируем монолитное ядро OpenCL.

Запишем последовательность действий алгоритма Boost.odeint

1. Тип переменной состояния

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

2. Системная функция

```
3  struct lorenz_system {
4      const sym_vector &R;
5
6      void operator()(const sym_state &x, sym_state &dxdt, double t) const {
7          dxdt[0] = sigma * (x[1] - x[0]);
8          dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
9          dxdt[2] = x[0] * x[1] - b * x[2];
10     }
11 };
```

3. Алгоритм

```
12 odeint::runge_kutta4<
13         sym_state /*state*/,       double /*value*/,
14         sym_state /*derivative*/, double /*time*/,
15         odeint::range_algebra, odeint::default_operations
16         > stepper;
```

4. Запишем одну итерацию метода Рунге-Кутты

```
17 std::ostringstream lorenz_body;
18 vex::generator::set_recorder(lorenz_body);
19
20 sym_state sym_S = {{ sym_vector(sym_vector::VectorParameter),
21                       sym_vector(sym_vector::VectorParameter),
22                       sym_vector(sym_vector::VectorParameter) }};
23 sym_vector sym_R(sym_vector::VectorParameter, sym_vector::Const);
24
25 lorenz_system sys{sym_R};
26 stepper.do_step(std::ref(sys), sym_S, 0, dt);
```

Сгенерируем монолитное ядро

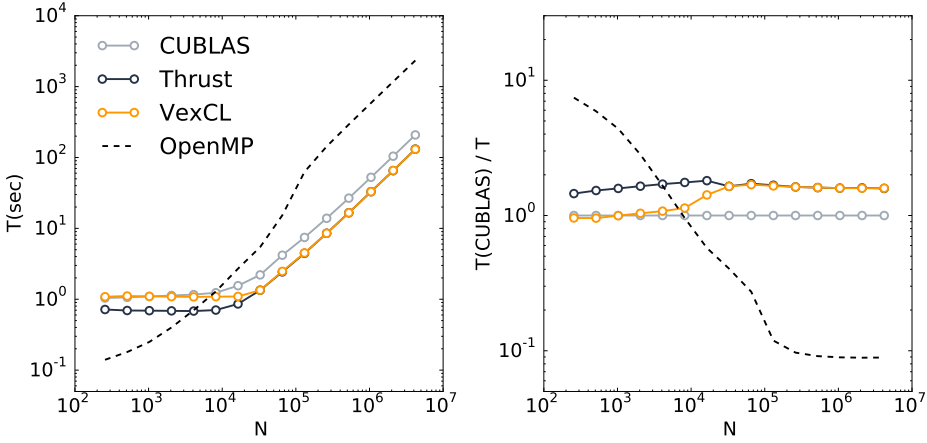
5. Сгенерируем и вызовем ядро OpenCL

```
27 auto lorenz_kernel = vex::generator::build_kernel(ctx, "lorenz", lorenz_body.str(),
28         sym_S[0], sym_S[1], sym_S[2], sym_R);
29
30 vex::vector<double> X(ctx, n), Y(ctx, n), Z(ctx, n), R(ctx, n);
31
32 X = Y = Z = 10;
33 R = Rmin + (Rmax - Rmin) * vex::element_index() / (n - 1);
34
35 for(double t = 0; t < t_max; t += dt) lorenz_kernel(X, Y, Z, R);
```

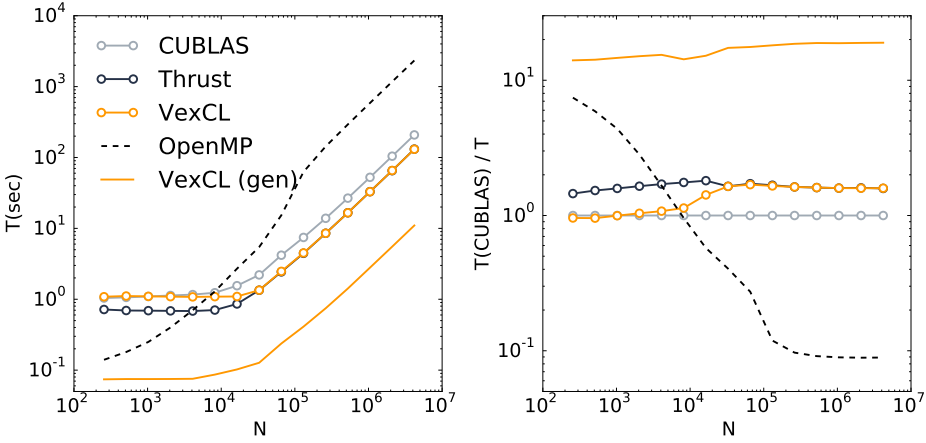
Ограничения

- Строго параллельные алгоритмы (нет взаимодействия между потоками).
- Не допускаются ветвления в зависимости от текущих значений.

Производительность сгенерированного ядра



Производительность сгенерированного ядра



Проекты, использующие VexCL

AMGCL — решение разреженных СЛАУ алгебраическим многосеточным методом:

- github.com/ddemidov/amgcl

Antioch — A New Templated Implementation Of Chemistry for Hydrodynamics:

- github.com/libantioch/antioch

Boost.odeint — численное решение обыкновенных дифуравнений:

- github.com/boostorg/odeint

Kratos Multiphysics — библиотека численных методов для решения прикладных инженерных задач:

- cimne.com/kratos