VexCL
Генерация ядер OpenCL/CUDA из выражений C++

Денис Демидов

МСЦ РАН, Казань

Октябрь 2014, Таруса

Современные GPGPU платформы

NVIDIA CUDA

- Проприетарная архитектура
- Необходимо аппаратное обеспечение NVIDIA
- Зрелое окружение, большое число библиотек

OpenCL

- Открытый стандарт
- Большой диапазон поддерживаемого железа
- Низкоуровневый программный интерфейс

Современные GPGPU платформы

NVIDIA CUDA

- Проприетарная архитектура
- Необходимо аппаратное обеспечение NVIDIA
- Зрелое окружение, большое число библиотек
- *Ядра (C++) компилируются в псевдо-ассемблер (PTX) вместе с основной программой*

OpenCL

- Открытый стандарт
- Большой диапазон поддерживаемого железа
- Низкоуровневый программный интерфейс
- *Ядра (C99) компилируются во время выполнения, увеличивается время инициализации*

- Последнее отличие обычно считается недостатком OpenCL.
- Однако, это позволяет генерировать во время выполнения более эффективные ядра под конкретную задачу!

VexCL

Библиотека шаблонов векторных выражений для OpenCL/CUDA

Fork me on GitHub

- Создана для облегчения разработки GPGPU приложений на C++.
  - Интуитивная нотация для записи векторных выражений.
  - Автоматическая генерация вычислительных ядер во время выполнения.
- Исходный код доступен под лицензией MIT.
  - https://github.com/ddemidov/vexcl

Программный интерфейс VexCL

Пример использования VexCL: сложение векторов

Инициализация конекста:

```
1 vex::Context ctx( vex::Filter::DoublePrecision );
2 if ( !ctx ) throw std::runtime_error("No compute devices available");
```

Подготовка входных данных, перенос на GPU:

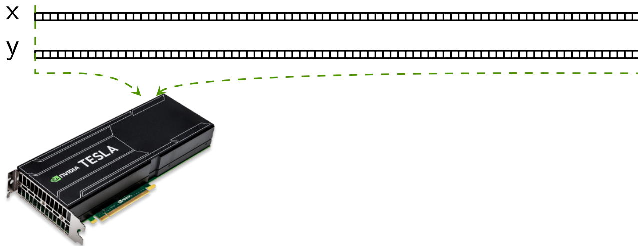
```
3 std::vector<double> a(N, 1), b(N, 2), c(N);
4 vex::vector<double> A(ctx, a);
5 vex::vector<double> B(ctx, b);
6 vex::vector<double> C(ctx, N);
```

Запуск вычислительного ядра, перенос результатов на CPU:

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

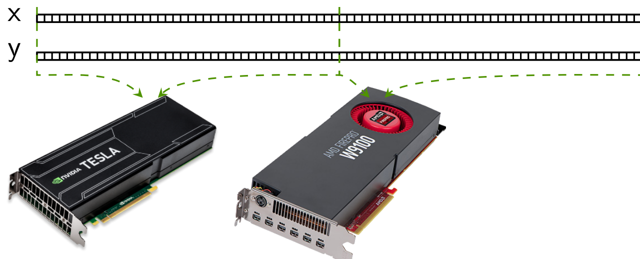
Использование нескольких ускорителей

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

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

x = vex::element_index() * (1.0 / N);
y = sin(2 * x) + sqrt(1 - x * x);
```



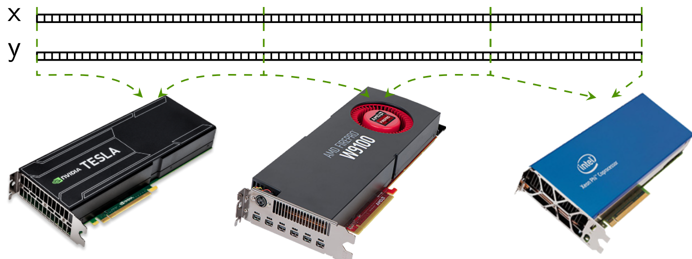
Использование нескольких ускорителей

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



Использование нескольких ускорителей

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



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

```
1 x = (2 * M_PI / n) * vex::element_index();
2 y = pow(sin(x), 2.0) + pow(cos(x), 2.0);
```

Ядра OpenCL/CUDA генерируются во время исполнения

Следующее выражение:

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

скомпилированное с ключами:

```
g++ -DVEXCL_BACKEND_OPENCL -lOpenCL ...
```

приводит к генерации и исполнению следующего ядра:

```
kernel void vexcl_vector_kernel
(
  ulong n,
  global double * prm_1,    // x
  int prm_2,                // 2
  global double * prm_3,    // y
  global double * prm_4     // z
)
{
  for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0))
  {
    prm_1[idx] = ( ( prm_2 * prm_3[idx] ) - sin( prm_4[idx] ) );
  }
}
```

Ядра OpenCL/CUDA генерируются во время исполнения

Следующее выражение:

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

скомпилированное с ключами:

```
g++ -DVEXCL_BACKEND_CUDA -lcuda ...
```

приводит к генерации и исполнению следующего ядра:

```
1  extern "C" __global__ void vexcl_vector_kernel (
2    ulong n,
3    double * prm_1,          // x
4    int prm_2,               // 2
5    double * prm_3,          // y
6    double * prm_4           // z
7  )
8  {
9    for(ulong idx = blockDim.x * blockIdx.x + threadIdx.x, grid_size = blockDim.x * gridDim.x;
10       idx < n; idx += grid_size)
11   {
12     prm_1[idx] = ( ( prm_2 * prm_3[idx] ) - sin( prm_4[idx] ) );
13   }
14 }
```

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++ — это 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      assert(a.size() == b.size());
3      vector tmp( a.size() );
4      for(size_t i = 0; i < a.size(); ++i)
5          tmp[i] = a[i] + b[i];
6      return tmp;
7  }
```

C++ допускает перегрузку операторов!

```
1  vector operator+(const vector &a, const vector &b) {
2      assert(a.size() == b.size());
3      vector tmp( a.size() );
4      for(size_t i = 0; i < a.size(); ++i)
5          tmp[i] = a[i] + b[i];
6      return tmp;
7  }
```

- 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      vsum(const vector &lhs, const vector &rhs) : lhs(lhs), rhs(rhs) {}
5  };
6
7  vsum operator+(const vector &a, const vector &b) {
8      return vsum(a, b);
9  }
```

Отложенный расчет v0.1

Идея: отложим расчет результата до операции присваивания.

```
1  struct vsum {
2      const vector &lhs;
3      const vector &rhs;
4      vsum(const vector &lhs, const vector &rhs) : lhs(lhs), rhs(rhs) {}
5  };
6
7  vsum operator+(const vector &a, const vector &b) {
8      return vsum(a, b);
9  }
10
11 const vector& vector::operator=(const vsum &s) {
12     for(size_t i = 0; i < data.size(); ++i)
13         data[i] = s.lhs[i] + s.rhs[i];
14     return *this;
15 }
```

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      vsum(const LHS &lhs, const RHS &rhs) : lhs(lhs), rhs(rhs) {}
6
7      double operator[](size_t i) const { return lhs[i] + rhs[i]; }
8  };
```

Отложенный расчет v0.2

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

Отложенный расчет v0.2

```
1  template <class LHS, class RHS>
2  struct vsum {
3      const LHS &lhs;
4      const RHS &rhs;
5      vsum(const LHS &lhs, const RHS &rhs) : lhs(lhs), rhs(rhs) {}
6
7      double operator[](size_t i) const { return lhs[i] + rhs[i]; }
8  };
9
10 template <class LHS, class RHS>
11 vsum<LHS, RHS> operator+(const LHS &a, const RHS &b) {
12     return vsum<LHS, RHS>(a, b);
13 }
14
15 template<class Expr>
16 const vector& vector::operator=(const Expr &expr) {
17     for(int i = 0; i < data.size(); ++i) data[i] = expr[i];
18     return *this;
19 }
```

Добавим остальные операции

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

Добавим остальные операции

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

template <class LHS, class OP, class RHS>
struct binary_op {
    const LHS &lhs;
    const RHS &rhs;
    binary_op(const LHS &lhs, const RHS &rhs) : lhs(lhs), rhs(rhs) {}

    double operator[ ](size_t i) const { return OP::apply(lhs[i], rhs[i]); }
};
```

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      binary_op(const LHS &lhs, const RHS &rhs) : lhs(lhs), rhs(rhs) {}
10
11     double operator[](size_t i) const { return OP::apply(lhs[i], rhs[i]); }
12 };
13
14 template <class LHS, class RHS>
15 binary_op<LHS, plus, RHS> operator+(const LHS &a, const RHS &b) {
16     return binary_op<LHS, plus, RHS>(a, b);
17 }
```

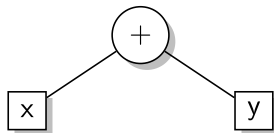
Шаблоны выражений — это деревья

Выражение в правой части:

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

... имеет тип:

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



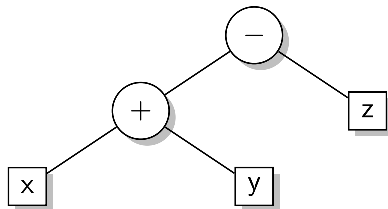
Шаблоны выражений — это деревья

Выражение в правой части:

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

... имеет тип:

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



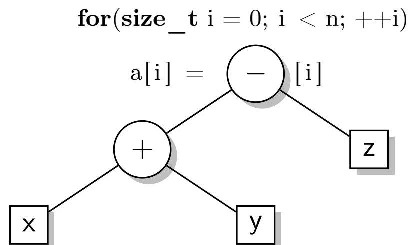
Шаблоны выражений — это деревья

Выражение в правой части:

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

... имеет тип:

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



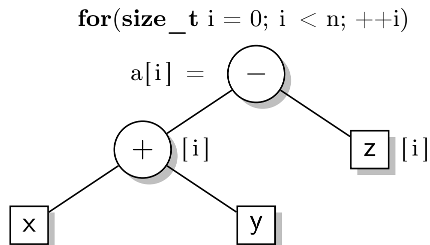
Шаблоны выражений — это деревья

Выражение в правой части:

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

… имеет тип:

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



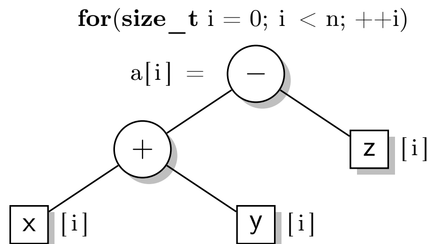
Шаблоны выражений — это деревья

Выражение в правой части:

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

... имеет тип:

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



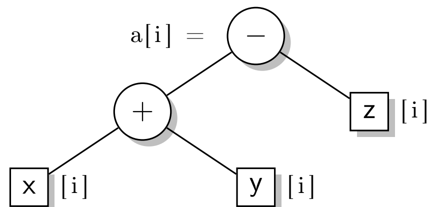
Шаблоны выражений — это деревья

Выражение в правой части:

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

... имеет тип:

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



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

Генерация кода OpenCL

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.

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.

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

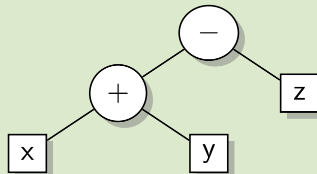
Генерация исходного кода ядра из выражений C++

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>::prm_decl(
6                                      std::ostream &src, unsigned &pos)
7  {
8      LHS::prm_decl(src, pos);
9      RHS::prm_decl(src, pos);
10 }
```

Построение строкового представления выражения

```
1  struct plus {
2      static std::string string() { return "+"; }
3  };
```

Построение строкового представления выражения

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

Построение строкового представления выражения

```
struct plus {
    static std::string string() { return "+"; }
};

/*static*/ void vector::make_expr(std::ostream &src, unsigned &pos) {
    src << "prm" << ++pos << "[idx]";
}

template <class LHS, class OP, class RHS>
/*static*/ void binary_op<LHS, OP, RHS>::make_expr(
                                        std::ostream &src, unsigned &pos) const
{
    src << "( ";
    LHS::make_expr(src, pos);
    src << " " << OP::string() << " ";;
    RHS::make_expr(src, pos);
    src << " )";
}
```

Генерация исходного кода ядра

```
template <class LHS, class RHS>
std::string kernel_source() {
    std::ostringstream src;

    src << "kernel void vexcl_vector_kernel(\n    ulong n";
    unsigned pos = 0;
    LHS::prm_decl(src, pos);
    RHS::prm_decl(src, pos);
    src << ")\n{\n"
           "    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {\n"
           "        ";
    pos = 0;
    LHS::make_expr(src, pos); src << " = ";
    RHS::make_expr(src, pos); src << ";\n";
    src << "    }\n}\n";

    return src.str();
}
```

Генерация исходного кода ядра

```
template <class LHS, class RHS>
std::string kernel_source() {
    std::ostringstream src;

    src << "kernel void vexcl_vector_kernel(\n    ulong n";
    unsigned pos = 0;
    LHS::prm_decl(src, pos);
    RHS::prm_decl(src, pos);
    src << ")\n{\n"
           "    for(size_t idx = get_global_id(0); idx < n; idx += get_global_size(0)) {\n"
           "        ";
    pos = 0;
    LHS::make_expr(src, pos); src << " = ";
    RHS::make_expr(src, pos); src << ";\n";
    src << "    }\n}\n";

    return src.str();
}
```

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::prm_decl(src, pos);
8       RHS::prm_decl(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::make_expr(src, pos); src << " = ";
14      RHS::make_expr(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  }
```

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.

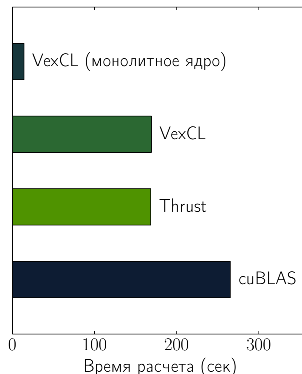
На самом деле все не (совсем) так

- 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.

Заключение

- Преимущества генерации кода во время выполнения:
  - Получаем ядро, сгенерированное под конкретную задачу.
  - Гибкость C++ несмотря на код C99 в ядрах.
  - Нет необходимости привязываться к одному поставщику.

Одновременное интегрирование большого числа ОДУ на GPU:



[1] Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries. *SIAM J. Sci. Comput.*, 35(5):C453 – C472, 2013.

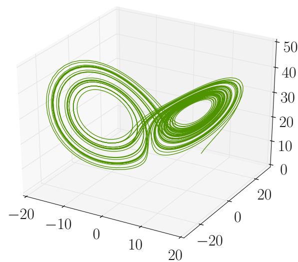
Оценка производительности

Параметрическое исследование системы Лоренца

Система Лоренца

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

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



Траектория аттрактора Лоренца

Вариант с использованием CUBLAS

- CUBLAS — оптимизированная библиотека линейной алгебры от NVIDIA.
- Линейные комбинации (используемые в алгоритмах 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

- 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

- Thrust позволяет получить монолитное ядро:

Thrust

```
struct scale_sum2 {
    const double a1, a2;
    scale_sum2(double a1, double a2) : a1(a1), a2(a2) { }
    template<class Tuple>
    __host__ __device__ void operator()(Tuple t) const {
        thrust::get<0>(t) = a1 * thrust::get<1>(t) + a2 * thrust::get<2>(t);
    }
};

thrust::for_each(
        thrust::make_zip_iterator(
            thrust::make_tuple( x0.begin(), x1.begin(), x2.begin() )
            ),
        thrust::make_zip_iterator(
            thrust::make_tuple( x0.end(), x1.end(), x2.end() )
            ),
        scale_sum2(a1, a2)
        );
```

VexCL

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

Использование символьных переменных

- Любые арифметические операции с экземплярами класса vex::symbolic<> выводятся в текстовый поток:

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

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

- Мы можем подать символьные переменные на вход шаблонной функции и получить запись алгоритма в виде кода C99.
  - Используем один шаг алгоритма Рунге-Кутты из Boost.odeint для получения монолитного ядра.