Создание реальных программ для видеокарт

Кэширование ядер

Генерация кода

Многомерные массивы и солверы

Несколько видеокарт

Создание программы из бинарников

```
std::string src = ...;
try {
  // пытаемся использовать скомпилированные ранее бинарники
  std::vector<const unsigned char> binaries = read_binaries(src);
  clCreateProgramWithBinary(..., binaries.data(), ...);
} catch (...) {
  // пытаемся скомпилировать бинарники из исходного кода
  clCreateProgramWithSource(..., src.data(), ...);
  clGetProgramInfo(..., CL_PROGRAM_BINARIES, ...);
  // сохраняем бинарники
  write_binaries(src, binaries);
}
```

Создание бинарников

```
std::string part1;
part1 += platform.getInfo<CL_PLATFORM_NAME>();
part1 += platform.getInfo<CL_PLATFORM_VENDOR>();
part1 += platform.getInfo<CL_PLATFORM_VERSION>();
part1 += device.getInfo<CL_DEVICE_NAME>();
part1 += device.getInfo<CL_DEVICE_VENDOR>();
part1 += device.getInfo<CL_DEVICE_VERSION>();
std::string part2;
part2 += src; // без учета include
part2 += flags;
std::hash<std::string> hash;
auto number = hash(part1 + part2);
std::stringstream filename;
filename << std::setfill('0') << std::setw(2*sizeof(number))
         << std::hex << number;
```

Генерация кода

Варианты:

- Использование опций препроцессора `-D`.
- Создание программы по шаблону.
- Создание библиотеки для работы с многомерными массивами для OpenCL.

Опции препроцессора

```
const char* flags = "-DREAL=float -DREAL_FLOAT";
cl::Program prog(...);
prog.build(devices, flags);
```

types.h (код OpenCL):

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

#define DO_CONCAT(x,y) x##y
#define CONCAT(x,y) VTB_DO_CONCAT2(x, y)

typedef REAL T;
typedef CONCAT(REAL,2) vec2; // float2 или double2
typedef CONCAT(REAL,3) vec3;
typedef CONCAT(REAL,4) vec4;
typedef CONCAT(REAL,16) vec16;
```

Опции препроцессора (2)

```
#include "types.h"
T interpolate_3d(
  global const T* func,
  vec3 x,
  const int3 size,
  const vec3 lbound,
  const vec3 ubound
) { ... }
```

Генерация по шаблону

```
std::stringstream out;
...
if (Nprev > 1) {
    out << "    angf = (float) (("
        << (z*numWorkItemsPerXForm) << " + ii) >>"
        << (logNPrev) << ");\n";
} else {
    out << "    angf = (float) ("
        << (z*numWorkItemsPerXForm)
        << " + ii);\n";
}
...
```

- Любой скалярный аргумент можно превратить в константу OpenCL.
- Чем больше констант, тем больше бинарных файлов в кэше.

Пример: для двухмерного быстрого преобразования Фурье от 2 до 1024 нужно 100 различных ядер.

Многомерные массивы на OpenCL

VexCL (код на C++):

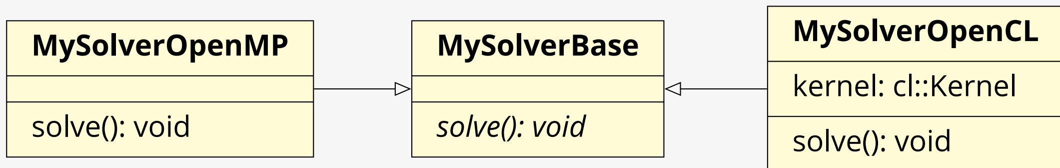
```
X = 2 * Y - sin(Z);
```

Сгенерированный код на OpenCL:

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

Библиотеки с многомерными массивами для OpenCL: VexCL, ClojureCL, ArrayFire.

Солверы на OpenCL



`My_solver.hh`:

```
// базовый класс для солверов с виртуальной функцией
struct My_solver_base { virtual void solve(...) = 0; };

enum class Engine { OpenMP, OpenCL };

// функция-конструктор
std::unique_ptr<My_solver_base> make_my_solver(Engine eng);
```

`My_solver_openmp.cc`:

```
// реализация солвера на OpenMP
struct My_solver_openmp: public My_solver_base { ... };
```

`My_solver_opencl.cc`:

```
// реализация солвера на OpenCL
struct My_solver_opencl: public My_solver_base { ... };
```

`My_solver.cc`:

```
std::unique_ptr<My_solver_base>
make_my_solver(Engine eng) {
  using ptr = std::unique_ptr<My_solver_base>;
  if (eng == Engine::OpenMP) { return ptr(new My_solver_openmp); }
  if (eng == Engine::OpenCL) { return ptr(new My_solver_opencl); }
  return nullptr;
}
```

Балансировка нагрузки

load_balancer.hh:

```
struct Queue_pair { cl::CommandQueue kernel_queue, data_queue; };
struct Load_balancer {
  std::vector<Queue_pair> queues; // очередь для каждого ускорителя
  size_t current = 0;             // текущая очередь
  std::vector<cl_event> events;
  void kernel(cl::Kernel k);      // запуск ядра
  void copy(...,cl::Buffer,...);  // копирование данных
  void wait();
  // round-robin
  void next() { if (++current >= queues.size()) current = 0; }
  size_t num_gpus() const { return queues.size(); }
  ...
};
```

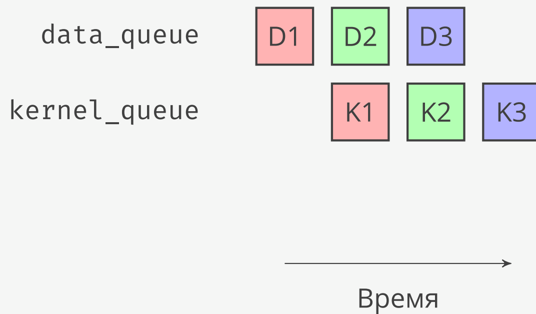
load_balancer.cc:

```
void Load_balancer::kernel(cl::Kernel k) {
  cl_event event;
  clEnqueueNDRangeKernel(queues[current].kernel_queue.get(),
    k.get(), ..., &event);
  events.emplace_back(event);
}

void Load_balancer::copy(..., cl::Buffer b, ...) {
  clEnqueueWriteBuffer(queues[current].data_queue.get(), ...);
}

void Load_balancer::wait() {
  clWaitForEvents(events.size(), events.data());
  events.clear();
}
```

Параллельная передача данных и вычисления



Несколько видеокарт

Было:

```
class My_solver_opencl {
  cl::Kernel kernel;
  cl::Buffer buffer;
public:
  void solve() override { queue.enqueueNDRangeKernel(kernel); }
};
```

Стало:

```
class My_solver_opencl {
  std::vector<cl::Kernel> kernel; // для каждой видеокарты
  std::vector<cl::Buffer> buffer; // для каждой видеокарты
public:
  void solve() override {
    Load_balancer lb;
    for (size_t i=0; i<lb.num_gpus(); ++i) {
      lb.kernel(kernel[i]);
    } } };
```

© 2019–2021 Ivan Gankevich i.gankevich@spbu.ru

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. The copy of the license is available at https://creativecommons.org/licenses/by-sa/4.0/.