OpenCL-backend for Accelerate

Martin Dybdal (dybb@di.tohber.dk)
Supervisor: Ken Friis Larsen

OpenCL example

```
// Create context, device and command queue objects
cl_context ctx = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, 0, 0, 0);
cl_device_id dev[1]; clGetContextInfo(ctx, CL_CONTEXT_DEVICES, 0, 0, &ndev);
cl_command_queue queue = clCreateCommandQueue(ctx, dev[0], 0);

// Load and compile kernel program
cl_program prog = clCreateProgramWithSource(ctx, 1, &programSource, 0, 0);
clBuildProgram(prog, 0, 0, 0, 0, 0);
cl_kernel kernel = clCreateKernel(prog, "vectorAdd", 0);

// Create memory objects
cl_mem devMemA = clCreateBuffer(ctx, 0, sizeof(cl_float) * dim, NULL, NULL);
cl_mem devMemB = clCreateBuffer(ctx, 0, sizeof(cl_float) * dim, NULL, NULL);
cl_mem devMemC = clCreateBuffer(ctx, 0, sizeof(cl_float) * dim, NULL, NULL);

// Load data into device memory
clEnqueueWriteBuffer(queue, devMemA, CL_TRUE, 0, sizeof(cl_float) * dim, devA, 0, 0, NULL);
clEnqueueWriteBuffer(queue, devMemB, CL_TRUE, 0, sizeof(cl_float) * dim, devB, 0, 0, NULL);

// Set kernel arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), devMemA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), devMemB);

// Copy data from host to device
clEnqueueWriteBuffer(queue, devMemC, CL_TRUE, 0, sizeof(cl_float) * dim, devC, 0, 0, NULL);

// Execute kernel
clEnqueueNDRangeKernel(queue, kernel, 1, 0, &dim, 0, 0, NULL);
```

General purpose computing on GPUs

OpenCL

OpenCL is an open standard for GPU programming, executing on hardware from many vendors. The main alternative to OpenCL is CUDA, which executes solely on NVIDIA hardware.

The smallest unit of work in OpenCL is called a work-item and represents a single thread of execution. GPU programming is done by writing kernel programs, which specifies the work done by single work-items. It is the task of the kernel itself to find the subset of the data it should work on, and it is first when the kernel is invoked that it is specified how many parallel instances of the kernel is executed (the number of work items).

OpenCL provides primitives for synchronization between work-items, transferring data to and from the GPU, and moving data between the different layers of GPU memory.

hopencil

In my project I have developed a Haskell interface to OpenCL. It is almost as low-level as the C-interface. It provides an interface using Haskell-native types, additional error handling and automatic memory management by attaching deallocation procedures to the Haskell garbage collector.

Back-end

The OpenCL and CUDA back-ends for Accelerate are organized into two phases. The first phase receives the final abstract syntax tree from the front-end, generates and compiles GPU-kernels. Simultaneously all needed arrays are transferred to GPU-memory. In the second phase the syntax tree is traversed bottom up executing one kernel at a time. Each higher-order function corresponds to specific kernel-skeletons. The OpenCL kernel-skeleton for map is shown in Figure 3. The skeleton is instantiated to a by fixing the Tyout and Ty0 variables, as well as for, set and apply-functions.

Surface Language

Accelerate is an array programming language embedded in Haskell, providing a purely functional and type safe interface to GPGPU programming. GPU-kernels are generated from Accelerate functions and are then scheduled on the GPU. Accelerate programs are considerably easier to comprehend and reason about than programs written directly for OpenCL/CUDA and most errors are guaranteed not to occur.

Front-end

It is remarkable that Accelerate programs can be written using Haskell λ-bindings and λ-expressions (e.g. (+) above), which are then translated into GPU code.

Accelerate programs specify abstract syntax, where λ-expressions are eliminated by inserting argument-indices (de Bruijn indices) in place of actual arguments. The arguments represents the number of binding-sites between their occurrence and the actual binder.

Similarly, λ-bindings can in some cases be recovered (to avoid recomputation in the generated code) by using functionality available in GHC, that lets the programmer identify values originating from the same binding-site.

The front-end is also responsible for doing certain optimizations and converting arrays into a memory layout where memory accesses can be properly aligned.

Data.Array.Accelerate

The following Accelerate-function computes a dot product on the GPU.

```haskell
data DotProduct = DotProduct Float
data DotProduct = DotProduct Float
let xs = use xs
  ys = use ys
in fold (+) 0 (zipWith (+) xs’ ys’)
```

Accelerate programs are executed in a pure manner with the function:

```haskell
run : Array a => Acc a -> a
```

Back-end

The OpenCL and CUDA back-ends for Accelerate are organized into two phases. The first phase receives the final abstract syntax tree from the front-end, generates and compiles GPU-kernels. Simultaneously all needed arrays are transferred to GPU-memory. In the second phase the syntax tree is traversed bottom up executing one kernel at a time. Each higher-order function corresponds to specific kernel-skeletons. The OpenCL kernel-skeleton for map is shown in Figure 3. The skeleton is instantiated to a by fixing the Tyout and Ty0 variables, as well as for, set and apply-functions.