DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN

OpenCL-backend for Accelerate

Martin Dybdal (dybber@dybber.dk)
Supervisor: Ken Friis Larsen

General purpose computing on GPUs

General purpose computing on graphics processing units (GPGPU) is the technique of using GPUs (graphical processing units) for applications commonly executed on CPUs.

Modern GPUs are massively parallel multi-core processors and are made for data-parallel tasks.

Fermi is the architecture of NVIDIAs latest GPU-series. A Fermi-device consist of up to 512 cores grouped into *streaming multiprocessors* (SMs) that contains 32 cores each (see Figure 2). All cores in a streaming multiprocessor executes the same operation simultaneously. This is known as *SIMD* (single instruction, multiple data). All cores in a SM share the same set of registers and local cache, managed manually by the programmer.

The programmer needs to be aware of intimate details of the architecture to use it efficiently. He must be careful to align memory accesses and partition work-items into well-sized groups that can be scheduled efficiently on the streaming multiprocessors.

OpenCL example

```
const char* programSource =
  "__kernel void vectorAdd(__global const float *a, "
                          "__global const float *b, "
                          "__global float *c) {"
  "    int nIndex = get_global_id(0);"
  "    c[nIndex] = a[nIndex] + b[nIndex]; }";

const size_t dim = 100; size_t ndev;
float pA[dim], pB[dim];

// Create context, device and command queue objects
cl_context ctx = clCreateContextFromType(0,
                   CL_DEVICE_TYPE_GPU, 0, 0, 0);
clGetContextInfo(ctx, CL_CONTEXT_DEVICES, 0, 0, &ndev);
cl_device_id * devs = malloc(ndev);
clGetContextInfo(ctx, CL_CONTEXT_DEVICES, ndev, devs, 0);
cl_command_queue queue =
        clCreateCommandQueue(ctx, devs[0], 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 device arrays (pA is copied to the device)
cl_mem devMemA = clCreateBuffer(ctx,
                   CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
                   dim * sizeof(cl_float), pA, 0);
cl_mem devMemB = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY,
                   dim * sizeof(cl_float), 0, 0);

// Execute Kernel
clSetKernelArg(kernel, 0, sizeof(cl_mem), devMemA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), devMemA);
clSetKernelArg(kernel, 2, sizeof(cl_mem), devMemB);
clEnqueueNDRangeKernel(queue, kernel, 1, 0, &dim,
                       0, 0, 0, 0);

// Copy from device to host memory
clEnqueueReadBuffer(queue, devMemB, CL_TRUE, 0,
            dim * sizeof(cl_float), pB, 0, 0, NULL);
```

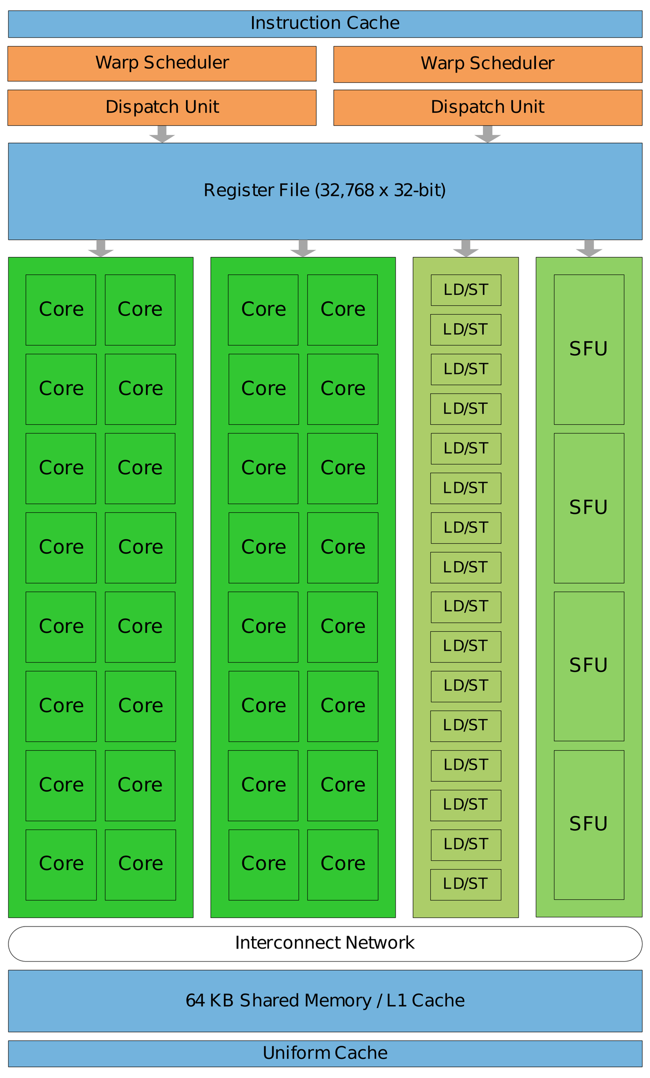

Figure 1: Fermi streaming multiprocessor. The illustration is borrowed from NVIDIAs Fermi whitepaper.

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.

hopencl

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.

Data.Array.Accelerate

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.

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

```
dotp :: Vector Float
     -> Vector Float
     -> Acc (Scalar Float)
dotp xs ys =
  let xs' = use xs
      ys' = use ys
  in fold (+) 0 (zipWith (*) xs' ys')
```

Accelerate programs are executed in a pure manner with the function: `run :: Arrays a => Acc a -> a`

Front-end

It is remarkable that Accelerate programs can be written using Haskell `let`-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 indices represents the number of binding-sites between their occurrence and the actual binder.

Similarly, `let`-bindings can in some cases be recovered (to avoid recomputation in the generated code) by a 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.

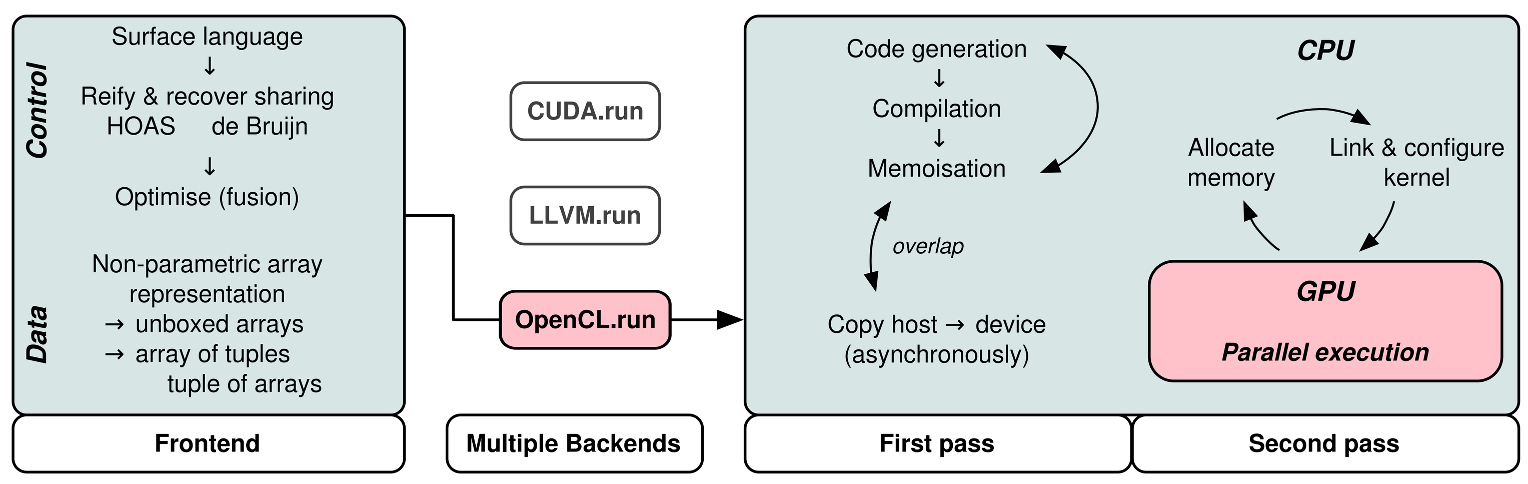

Figure 2: Accelerate overview. The illustration is borrowed from *Accelerating Haskell array codes with multicore GPUs* by M.M.T. Chakravarty et al. (2011)

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 fixating the `TyOut` and `TyIn` variables, as well as `get0`, `set` and `apply`-functions.

```
__kernel void map(__global TyOut* d_out, __global const TyIn0* d_in0, const Ix shape) {
    Ix idx;  const Ix gridSize = get_global_size(0);
    for(idx = get_global_id(0); idx < shape; idx += gridSize)
      set(d_out, idx, apply(get0(d_in0, idx)));
}
```

Figure 3: OpenCL skeleton kernel for the `map` operation.