Fusing GPU kernels within a novel single-source C++ API

HiPEAC Compiler, Architecture and Tools Conference
Intel Development Center, Haifa, Israel

Ralph Potter ¹, Paul Keir ¹, Jan Lucas ², Mauricio Alvarez-Mesa ², Ben Juurlink ², Andrew Richards ¹

¹Codeplay Software Ltd., Edinburgh

²Technische Universität Berlin

Nov 18th, 2013
Presentation Overview

- The Low-Power GPU (LPGPU) project
- Codeplay Software Ltd.
- Why fusion is relevant for low-power computing
- The Offload3 C++ parallel programming model
- Fusion of image processing primitives within Offload3
FP7 Low-Power GPU (LPGPU)

- A 3 year EU-funded FP7 STREP project (Call 7)
- 6 Consortium Members: 4 Companies; 2 Universities
  - TU Berlin; Uppsala University
  - AiGameDev; Codeplay; Geomerics; ThinkSilicon
- Researching low-power GPU hardware, tools and applications
  - While GPUs increase in performance, visual innovation flatlines
  - A pending GPU power-wall calls for architectural innovation
  - Few GPU tools for analysis and program transformation
Incorporated in 1999
Based in Edinburgh, Scotland
33 full-time employees
Compilers, optimisation and language development
  - GPU and Heterogeneous Architectures
  - Increasingly Mobile and Embedded CPU/GPU SoCs
Commercial partners:
  - Qualcomm, Movidius, AGEIA, Fixstars
  - Many other partners remain confidential
Member of three 3-year EU FP7 research projects:
  - Peppher, CARP and LPGPU
Sony-licensed PS3™ middleware provider
Contributing member of Khronos group since 2006
A member of the HSA Foundation since 2013
Kernel Fusion

- Data transfers are a major source of power consumption
- Host-device reads and writes exchange data between kernels
- Loads and stores can be combined into a single *fused* kernel
- Hand-fused kernels possible, but reusability is poor
- For now we focus on graphical image operations
Preparatory Benchmarking of Hand-Fused Kernels

- Hand-fused several kernels to estimate performance gains
- Gamma correction of YUV encoded images
- 5 kernels fused into 2 or 3
- Timings taken from an NVIDIA GeForce GTS 450
Principles of the Offload3 Programming Model

- Simplified software porting for existing parallel applications
- Code reuse, through sharing of host and device code
- Generic algorithms through C++ template meta-programming
- A foundation for higher-level programming models
Offload3 : a Single-Source Model

- Existing C++ compiler processes the host sections of the code
- *Extended* C++ device compiler processes the device sections
  - Currently outputs OpenCL SPIR bitcode
- Call graph is *duplicated* for all devices targeted
  - So, a single object or datatype may be used in both contexts
- Assuming OpenCL 1.x devices, so no:
  - function pointers; virtual methods; recursion;
  - exception handling; or run-time type information
- OpenCL code and C++ code may be used together
- The C++ preprocessor can be used to select the best code
  - e.g. to guard between sections compiled for device; or host
#include <offload3.hpp>

int main(int argc, char *argv[])
{
    ol3::queue q;
    return 0;
}

- The Offload3 command queue contains a list of tasks.
- These tasks are administered by the command thread.
- Tasks run on either host or device.
- The queue will select default OpenCL devices; and error handling.
A Single Asynchronous Task

```cpp
#include <offload3.hpp>
#include "stub.h"

using namespace ol3;

int main(int argc, char *argv[]) {
    queue q;
    command_group(&q, [&](){
        single_task(kernel_lambda<class MyTask>([=](){}}));
    });
    return 0;
}
```

- A single task will execute on the default OpenCL device
- The stub header file is generated by the device compiler
  - Allows the kernel to be linked with the host code
Wrapping C++11 Lambdas for Portability

```cpp
void example1()
{
    queue q;
    command_group(&q, [&]() {
        single_task(kernel_lambda<class MyTask>([=](){})�);
    });
}
```

- Each lambda function has a unique type
- C++11 lambda function is 1st argument of kernel_lambda
- Naming class need only be declared; in C++11 can be inline
void example1()
{
    queue q;
    command_group(&q, [&]()
        {
            single_task(kernel_lambda<class MyTask>([=]() {
                // kernel code
            }));
        });
}

- Each lambda function has a unique type
- C++11 lambda function is 1st argument of kernel_lambda
- Naming class need only be declared; in C++11 can be inline
The Offload3 Command Group

```cpp
void example1()
{
    queue q;
    command_group(&q, [&](){
        single_task(kernel_lambda<class MyTask>([=](){
            // kernel code
        }));
    });
}
```

- A command group defines multiple task/data parallel kernels
- Command group captures variables within scope by reference
- The kernel captures closure variables by value
  - Allowing the runtime to pass variables to and from the device
An Offload3 buffer object allows for data reuse
A single buffer object can serve multiple accessor objects
Also possible to express data dependencies through sub-buffers
Execution Modes in Offload3

- Offload3 supports both task and data-parallelism
- Data-parallel operational modes:
  - Basic data-parallel; obtained via `parallel_for`
  - Workgroup data-parallel; obtained via `parallel_for_ndrange`
- One task-parallel mode:
  - Task parallelism obtained via `single_task`; shown earlier
- Can also provide a kernel as an OpenCL C string
void example3()
{
    queue q;
    float f[64];
    buffer<float> data(f, 64);
    command_group(&q, []()
    {
        parallel_for(range<3>(4, 4, 4),
                        kernel_lambda<class MyTask>([=] (item id)
                        {
                            accessor<int, WRITE_ONLY> kernel_data(data);
                            kernel_data[id] = id.x + id.y + id.z;
                        }
                        ));
    });
}

▶ A simple range executes over a range of n dimensions
```cpp
void example4()
{
    queue q;
    float f[64];
    buffer<float> data(f, 64);
    struct shared { float data[8]; };  
    command_group(&q, []()
    {
        parallel_for_ndrange(
            nd_range(size(4, 4, 4), size(2, 2, 2)),
            kernel_lambda<class MyTask>(
                [=] (item id, _local shared &tile)
                {
                    accessor<int, WRITE_ONLY> kernel_data(data);
                    kernel_data[id] = id.x + id.y + id.z;
                    local_barrier();
                }
            ));
    });
} 
```
Image Processing Primitives

```cpp
struct blend
{
    rgb operator()(const rgb &a, const rgb &b, float alpha) {
        return a * alpha + b * (1 - alpha);
    }
};
```

- C++ interface fusing image processing primitives into kernels
- Image processing primitives implemented as function objects
- Reducing the need for global memory accesses
- A weighted blend operation is shown above
```cpp
#include <functional>

using namespace std :: placeholders; // for _1, _2, etc.

char id(char c)
{
    auto f = [] (int x, double d, char c) { return c; };
    auto g = std::bind(f, 0, 3.142, _1);
    return g(c);
}
```

- C++11 `std::bind` function allows partial application
- Our device compiler is built on LLVM and Clang
- Clang cannot parse the header files from MS Visual Studio
- We implemented portions of the C++11 utilities library
Lazy and Partial Application

```c++
void example5()
{
    rgb a, b;
    float alpha;

    auto f = bind<blend>(a, b, alpha);

    // ...

    f(); // Evaluation here
}
```

- Using the earlier `blend` function object class
- A bind expression facilitates delayed/partial application
- Current version omits the first argument of `std::bind`
void example6()
{
    rgb colour;
    float saturation;

    auto g = bind<hsv_to_rgb>(
        bind<desaturate>(
            bind<rgb_to_hsv>(colour),
            saturation
        )
    );

    // ...
    g(); // Evaluation here
}

- Nested bind expressions can construct complex filter pipelines
Placeholder Variables in Action

```cpp
void example7()
{
    rgb colour;
    float saturation;
    placeholder<0> _1;

    auto h = bind<hsv_to_rgb>(
        bind<desaturate>(
            bind<rgb_to_hsv>(_1),
            saturation
        ),
    );

    // ...
    h(colour); // Evaluation with colour's current value
}
```

- Placeholder variables then allow custom pipelines with holes
- These can be declared on the host or the device
- So possible to write a parallel_image template function
  - Simply takes a function object, such as our pipeline
Iterators to Manage Shared Memory

```cpp
template<
size_t Size,
typename T>
struct iterator {
    iterator(__local void *s, const item &id,
             const T &obj) : id(id), store(s) {
        auto range = id.get_local_range();
        int i = id.get_local(0)*range[1]+id.get_local(1);
        store[i] = obj;
        local_barrier();
    }

    __local T* store;
    item id;
};
```

- Iterators act as stencils for convolution operators
- Automatically move data from private to local storage
  - ...through their constructors.
template<
    size_t Size
>
struct gaussian :
    requires_halo<
        Size / 2
    >
{
    cie_lab operator() (const iterator<
            Size, cie_lab>
        & iter) {
        auto f = [=](int2 offset,
            const __local cie_lab & lab) -> cie_lab {
            return lab *
                e_[offset.y+(Size/2)][offset.x+(Size/2)];
        };

        return iter.foreach(f);
    }

    T e_[Size][Size];
};

- Define function objects which take an iterator
- f applied to each element of the iterator
Image Processing Primitives and Iterators

```cpp
rgb a;

auto pipeline =
bind<gaussian<GAUSS_SIZE>>(
    bind<create_iterator<GAUSS_SIZE, rgb>>(local_mem, item, a)
);
```

- Now possible to integrate convolution into our pipelines
- Ideally we could get rid of the `local_mem` and `item` parameters
Halo Size Inference

```c++
template<size_t Size>
struct gaussian : requires_halo<Size / 2>;
```

- Function class `gaussian` was defined earlier
- `gaussian` inherits from `requires_halo`
- Bind expressions able to calculate the cumulative halo size
  - Halo size of entire pipeline calculated at compile-time
- For multiple convolutions, halo is the sum of individual halos
- Becomes a constant `static` member of final pipeline object
Initial Benchmarking

Unsharp Masking - SPIR vs Emulator on Windows

Execution time in seconds (log scale)

Image Resolution

256x256  512x512  768x768  1024x1024  1536x1536  2048x2048
In Conclusion...

- The Offload3 C++ parallel programming model
- An API capable of a high-level of abstraction
- An application to the fusion of image processing operations
- A fusion interface aligned with the developer experience
- Further benchmarks will measure the effect on power consumption