Code Generation for Embedded Heterogeneous Architectures on Android

Oliver Reiche, Richard Membarth, Frank Hannig, and Jürgen Teich

University of Erlangen-Nuremberg
Motivation

What do we need DSLs and code generation for?

3P: Performance, Productivity, and Portability

What’s the difference for embedded heterogeneous architectures?
Outline

1. Programming Models

2. Code Generation
   - HIPA$^c_c$ Framework
   - Renderscript Code Generation
   - Vector Support
   - HSA Memory Management

3. Results
Programming Models
Programming Models

Android NDK (Native Development Kit)
- no native support for GPUs
- low-level fine tuning:
  - implicit and explicit vectorization (SSE/AVX/NEON)
  - cache-aware programming

OpenCL (inofficial)
- support for CPUs, GPGPUs and others
- low-level fine tuning:
  - explicit mapping of threads
  - transparent memory hierarchy
  - supports unified CPU/GPU memory
Renderscript

Renderscript Compute

• code mapping to native threads
• targets CPUs and DSPs
• additionally targets GPUs (since Android 4.2)

Filterscript

• stricter limitations
  – relaxed precision
  – no scatter writes
  – pointers are illegal
• ensures wider compatibility
On first sight, much similarities to OpenCL but fundamentally different . . .

Philosophy behind Renderscript

- higher level of programming
- to widen support for different architectures
- dynamic execution on heterogeneous platforms
- uncouple developer from target hardware
- at the cost of performance

low-level optimizations are barely possible!
HIPA\textsuperscript{cc} Framework
HIPA\textsuperscript{cc} Framework Overview

Domain Knowledge
Architecture Knowledge
Source-to-Source Compiler
Clang/LLVM

C++
embedded DSL

CUDA (GPU)
OpenCL (x86/GPU)
C/C++ (x86)
Renderscript (x86/ARM/GPU)

CUDA/OpenCL/Renderscript Runtime Library
HIPA CC Example: Host Code

```cpp
/* ... */

Image<uchar> in(width, height);
Image<float> out(width, height);

in = in_image;
out = out_image;

const float filter_mask[size][size] = {...};
Mask<float> mask(filter_mask);

BoundaryCondition bound(in, mask, BOUNDARY_CLAMP);
Accessor<uchar> acc(bound);
IterationSpace<float> iter(out);

GaussianBlur filter(iter, acc, mask, size/2);
filter.execute();

out_image = out;
```
class GaussianBlur : public Kernel<float> {
  Mask<float> mask;
  Accessor<uchar> input;
  size_t range;

public:
  GaussianBlur(IterationSpace<float> iter, Accessor<uchar> acc,
                Mask<float> mask, size_t range)
    : Kernel(iter), input(acc), mask(mask), range(range) {
    addAccessor(acc);
  }

  void kernel() {
    float sum = .0f;
    for (int yf = -range; yf <= range; ++yf)
      for (int xf = -range; xf <= range; ++xf)
        sum += input(xf, yf) * mask(xf, yf);
    output() = sum;
  }
};
Renderscript Code Generation
Renderscript Memory Access

Memory Access Mapping

DSL Kernel:

```c
1 Accessor<uchar> input;
2 ...
3 void kernel() {
4     uchar val = input();
5     output() = val;
6 }
```

Filterscript:

```c
1 rs_allocation input;
2 ...
3 uchar __attribute__((kernel)) kernel(uint32_t x, uint32_t y) {
4     uchar val = rsGetElementAt_uchar(input, x, y);
5     return val;
6 }
```
Memory Access Mapping

DSL Kernel:

```c
1 Accessor<uchar> input;
2 ...;
3 void kernel() {
4  uchar val = input();
5  output() = val;
6 }
```

Renderscript:

```c
1 rs_allocation input;
2 ...;
3 void kernel(uchar *iter, uint32_t x, uint32_t y) {
4  uchar val = rsGetElementAt_uchar(input, x, y);
5  *iter = val;
6 }
```
Memory Access Mapping

**DSL Kernel:**

1. `Accessor<uchar> input;`
2. `...`
3. `void kernel() {
4.    uchar val = input();
5.    output() = val;
6. }

**Renderscript:**

(4 Pixels per Thread)

1. `rs_allocation input;`
2. `rs_allocation out;`
3. `...`
4. `void kernel(uchar *iter, uint32_t x, uint32_t y) {
5.    { uchar val = rsGetElementAt_uchar(input, x, y+0);  
6.      rsSetElementAt_uchar(out, x, y+0, val);  }
7.    { uchar val = rsGetElementAt_uchar(input, x, y+1);  
8.      rsSetElementAt_uchar(out, x, y+1, val);  }
9.    { uchar val = rsGetElementAt_uchar(input, x, y+2);  
10.   rsSetElementAt_uchar(out, x, y+2, val);  }
11.   { uchar val = rsGetElementAt_uchar(input, x, y+3);  
12.      rsSetElementAt_uchar(out, x, y+3, val);  }
13. }

25-Mar-14
Oliver Reiche / University of Erlangen-Nuremberg
Renderscript Iteration Space

- defined by output buffer size
- no custom launch configuration

When we need less threads, e.g., for
- processing multiple pixels per thread
- operating on a fraction of the buffer (ROI)

→ we need appropriate Iteration Space Mapping
Iteration Space Mapping (3 Approaches)

1. Temporary buffer
   - additional memory
   - copy overhead: $\text{width}_{\text{ROI}} \times \text{height}_{\text{ROI}}$
Iteration Space Mapping (3 Approaches)

1. Temporary buffer
   - additional memory
   - copy overhead: width_{ROI} \times height_{ROI}

2. Dummy buffer
   - allocation overhead for unused buffer
   - not suitable for Renderscript

Renderscript Iteration Space
Iteration Space Mapping (3 Approaches)

1. Temporary buffer
   - additional memory
   - copy overhead: \( \text{width}_{\text{ROI}} \times \text{height}_{\text{ROI}} \)

2. Dummy buffer
   - allocation overhead for unused buffer
   - not suitable for Filterscript

3. Add guards to the kernel
   - suitable for Filterscript
   - copy overhead:
     \( (\text{width}_{\text{IMG}} \times \text{height}_{\text{IMG}}) - (\text{width}_{\text{ROI}} \times \text{height}_{\text{ROI}}) \)
   - minor execution overhead
Vector Support
Mobile GPUs: SIMD Units

- vector support is crucial for performance

Vector Support

- added vector types \( T_n \) (e.g., \( \text{float}4 \))
- added conversion functions \( T_n \ convert\_Tn(\ldots) \)

Single Core of the ARM Mali-T604
HSA Memory Management
Support for unified CPU/GPU memory

- abstract memory from developer
- implicitly handle memory transfers
- manage `map()` and `unmap()` operations

→ avoid unnecessary memory copies
Results
Results: Productivity

Productivity

Lines of Code for implementing different image filters

<table>
<thead>
<tr>
<th></th>
<th>Sobel</th>
<th>Gaussian</th>
<th>Laplace</th>
<th>FIR</th>
<th>Harris</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCV</td>
<td>1681</td>
<td>1641</td>
<td>1712</td>
<td>982</td>
<td>2247</td>
</tr>
<tr>
<td>HIPA\textsuperscript{cc} DSL</td>
<td>16</td>
<td>22</td>
<td>11</td>
<td>11</td>
<td>68</td>
</tr>
<tr>
<td>Renderscript</td>
<td>1915</td>
<td>1951</td>
<td>8575</td>
<td>3680</td>
<td>4265</td>
</tr>
</tbody>
</table>

HIPA\textsuperscript{cc} is

- up to 156x more compact than OpenCV
- up to 780x more compact than generated Renderscript
Results: Performance

Speedup GPU

5x5 Gaussian Blur on an ARM Mali-T604

- Renderscript-GPU
- Filterscript-GPU
- OpenCL-GPU

Code Variants show

→ use of constant memory is almost negligible (≈5%) on embedded GPUs
Results: Performance

Execution Time HSA (GPU with OpenCL)

normalized execution time

<table>
<thead>
<tr>
<th>Function</th>
<th>Sobel</th>
<th>Gaussian</th>
<th>Laplace</th>
<th>FIR</th>
<th>Harris</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td></td>
<td></td>
<td></td>
<td>copy</td>
<td>copy HSA</td>
</tr>
<tr>
<td>write/unmap</td>
<td>HSA</td>
<td>HSA</td>
<td>HSA</td>
<td>HSA</td>
<td></td>
</tr>
<tr>
<td>execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read/map</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

25-Mar-14
Oliver Reiche / University of Erlangen-Nuremberg
39
Summary
Contributions: We showed

- what kind of optimizations are useful on eGPGPUs
- using DSLs for embedded devices is reasonable, high productivity in describing image filters
- implicit use of unified CPU/GPU memory
Contributions: We showed

• what kind of optimizations are useful on eGPGPUs
• using DSLs for embedded devices is reasonable, high productivity in describing image filters
• implicit use of unified CPU/GPU memory

**HIPA^cc Framework Features**
• ROI definition
• boundary handling modes
• interpolation modes
• image pyramids
• built-in architecture model
• automatic exploration
• target-specific optimizations

**HIPA^cc Compiler Features**
• exploit full GPU memory hierarchy
• loop unrolling
• constant propagation
• multiple pixels per thread
• forced use of textures
• vectorization (point operators)
• unified CPU/GPU memory support
Contributions: We showed

• what kind of optimizations are useful on eGPGPUs
• using DSLs for embedded devices is reasonable, high productivity in describing image filters
• implicit use of unified CPU/GPU memory

HIPA\textsuperscript{cc} Framework Features
• ROI definition
• boundary handling modes
• interpolation modes
• image pyramids
• built-in architecture model
• automatic exploration
• target-specific optimizations

HIPA\textsuperscript{cc} Compiler Features
• exploit full GPU memory hierarchy
• loop unrolling
• constant propagation
• multiple pixels per thread
• forced use of textures
• vectorization (point operators)
• unified CPU/GPU memory support
Contributions: We showed

• what kind of optimizations are useful on eGPGPUs
• using DSLs for embedded devices is reasonable, high productivity in describing image filters
• implicit use of unified CPU/GPU memory

HIPA\textsuperscript{cc} Framework Features

• ROI definition
• boundary handling modes
• interpolation modes
• image pyramids
• built-in architecture model
• automatic exploration
• target-specific optimizations

HIPA\textsuperscript{cc} Compiler Features

• exploit full GPU memory hierarchy
• loop unrolling
• constant propagation
• multiple pixels per thread
• forced use of textures
• vectorization (point operators)
• unified CPU/GPU memory support
Questions?

HIPA²⁷ framework sources released under *Simplified BSD License*.

University Booth Demonstration: Wednesday, 12 p. m. & 4 p. m.

http://hipacc-lang.org
Results: Performance

Speedup CPU

- **Sobel**
- **Gaussian**
- **Laplace**
- **FIR**
- **Harris**

Execution speedup

- **CV-CPU**
- **RS-CPU**
- **FS-CPU**