Code Generation for GPU Accelerators in the Domain of Image Preprocessing

Oliver Reiche, Richard Membarth, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, University of Erlangen-Nuremberg
Dagstuhl, April 4, 2013
Motivation: Medical Image Preprocessing

Keep X-ray dosage and contrast agent as low as possible → noisy images. Improve quality of medical images . . .

• remove noise
• detect edges
• compensate for detector defects
• . . .

• an implementation must be as efficient as possible.
• most algorithms are well known.

What for do we need code generation?
Challenge: How To Target Multiple Architectures?

Efficient code generation for different target architectures.

Domain-specific Languages

• **performance**
  • portable: high performance on different target hardware
  • competitive: comparable performance to hand-written code

• **productivity**
  • algorithm description at a high-level
  • hide low-level details from programmer

• **portability**
  • support different target architectures from the same algorithm description
  • support different target languages from the same algorithm description

Domain-specific languages offer both functional- and performance-portability.
Agenda

HIPA^{cc}

Results

Summary
HIPA$^{cc}$
HIPAcc: The Heterogeneous Image Processing Acceleration Framework

- C++ embedded DSL
- Source-to-Source Compiler: Clang/LLVM

Domain Knowledge
Architecture Knowledge

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

CUDA/OpenCL/Renderscript Runtime Library
Domain Analysis: Image Processing Kernel Categorization

Identified three groups of kernels:

• **Point** operators [HPPC’11]
  • each pixel is updated uninfluential of other pixels

• **Local** operators [IPDPS’12]
  • centered at the pixel it is applied to $[0, 0]$
  • bounded to the neighborhood $[-m, +m] \times [-n, +n]$
  • operator can be applied in parallel

• **Global** operators [ISPDC’12]
  • pixels of the whole image contribute to result
  • for instance, reduction operators

---


HIPA\textsuperscript{cc}: The Heterogeneous Image Processing Acceleration Framework

Domain-specific Extensions

\textbf{IterationSpace} defines ROI of the output image

\textbf{Accessor} input ROI with filtering (nearest, bilinear, bicubic, \ldots)

\textbf{BoundaryCondition} boundary handling modes

\textbf{Mask} convolution mask

Output image.  
Crop of output image.  
Crop of output image with offset.
HIPA\textsuperscript{cc}: The Heterogeneous Image Processing Acceleration Framework

Domain-specific Extensions

\texttt{IterationSpace} defines ROI of the output image

\texttt{Accessor} input ROI with filtering (nearest, bilinear, bicubic, \ldots)

\texttt{BoundaryCondition} boundary handling modes

\texttt{Mask} convolution mask

- Image and boundary.
- Image crop.
- Image crop with offset.
- Image offset.
**HIPA cc**: The Heterogeneous Image Processing Acceleration Framework

**Domain-specific Extensions**

IterationSpace defines ROI of the output image

Accessor input ROI with filtering (nearest, bilinear, bicubic, ...)

BoundaryCondition boundary handling modes

Mask convolution mask

---

Repeat

Clamp

Mirror

Constant
HIPA<sup>cc</sup>: The Heterogeneous Image Processing Acceleration Framework

Domain-specific Extensions

- **IterationSpace** defines ROI of the output image
- **Accessor** input ROI with filtering (nearest, bilinear, bicubic, ...)
- **BoundaryCondition** boundary handling modes

**Mask convolution mask**
HIPA\textsuperscript{cc} Example: Gaussian Blur

```cpp
/* ... */
Image<uchar> in(width, height);
Image<float> out(width, height);
Mask<float> mask(size, size);

in = in_image;
out = out_image;
mask = filter_mask;

BoundaryCondition bound(in, mask, BOUNDARY_CLAMP);

AccessorLF<uchar> acc(bound, width, height, 0, 0);

IterationSpace<float> iter(out, width/2, height/2, width/4, height/4);
GaussianBlur filter(iter, acc, mask, size/2);
filter.execute();
out_image = out;
```
HIPAcc Example: Gaussian Blur Kernel

```cpp
1 class GaussianBlur : public Kernel<float> {
2    Mask<float> mask;
3    Accessor<uchar> input;
4    size_t range;
5
6 public:
7    GaussianBlur(IterationSpace<float> iter, Accessor<uchar> acc,
8                   Mask<float> mask, size_t range)
9        : Kernel(iter), input(acc), mask(mask), range(range) {
10       addAccessor(acc);
11    }
12
13    void kernel() {
14       float sum = .0f;
15       for (int yf = -range; yf <= range; ++yf)
16          for (int xf = -range; xf <= range; ++xf)
17             sum += input(xf, yf) * mask(xf, yf);
18       output() = sum;
19    }
20  };
```
HIPA\textsuperscript{cc} Example: Gaussian Blur Kernel + Lambda Function

```cpp
1 class GaussianBlur : public Kernel<float> {
2    Mask<float> mask;
3    Accessor<uchar> input;
4    size_t range;
5
6 public:
7    GaussianBlur(IterationSpace<float> iter, Accessor<uchar> acc,
8                   Mask<float> mask, size_t range)
9        : Kernel(iter), input(acc), mask(mask), range(range) {
10       addAccessor(acc);
11   }
12
13   void kernel() {
14       output() = convolve(mask, HipaccSUM, [&]() {
15           return input(mask) * mask();
16       });
17   }
18};
```

Lambda function for convolution
Efficient Code Generation for Boundary Handling

- generates 10 different code variants
- minimize executed conditionals
- minimize divergence
- block index determines code variant
- limit necessary boundary handling
- with respect to mask size and image padding
Mapping of GPU Memory Accesses

Image Preprocessing
mostly load $\rightarrow$ compute $\rightarrow$ store $\Rightarrow$ memory bound

Memory Type
- global memory
- constant memory
- texture memory
- surface memory
- local memory

Optimizations
- memory access alignment
- unrolling
- target (e.g., Kepler35, SouthernIsland, Midgard, ...)
Results
## Results

### Gaussian Blur $5 \times 5$ (separated)

12 vs. 238 lines of CUDA code

<table>
<thead>
<tr>
<th></th>
<th>Undef.</th>
<th>Clamp</th>
<th>Repeat</th>
<th>Mirror</th>
<th>Const.</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve</td>
<td>crash</td>
<td>7.79</td>
<td>8.15</td>
<td>7.94</td>
<td>8.01</td>
</tr>
<tr>
<td>OpenCV</td>
<td>n/a</td>
<td>3.92</td>
<td>6.63</td>
<td>6.76</td>
<td>4.18</td>
</tr>
<tr>
<td>RapidMind</td>
<td>10.92</td>
<td>12.16</td>
<td>crash</td>
<td>n/a</td>
<td>14.58</td>
</tr>
<tr>
<td>Halide</td>
<td>n/a</td>
<td>8.93</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>NPP (8-bit)</td>
<td>6.86</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>HIPA$^\text{cc}$ CUDA</td>
<td>3.27</td>
<td>3.52</td>
<td>3.32</td>
<td>3.47</td>
<td>3.31</td>
</tr>
</tbody>
</table>

Image of $4096 \times 4096$ pixels on a Tesla C2050. Times in ms.

### Bilateral Grid Filter

62 vs. 386 lines of CUDA code

<table>
<thead>
<tr>
<th></th>
<th>GTX 680</th>
<th>i7-3770</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handtuned CUDA</td>
<td>11.26</td>
<td>n/a</td>
</tr>
<tr>
<td>Halide</td>
<td>37.10</td>
<td>261.36</td>
</tr>
<tr>
<td>HIPA$^\text{cc}$ CUDA</td>
<td>17.14</td>
<td>n/a</td>
</tr>
<tr>
<td>HIPA$^\text{cc}$ OpenCL</td>
<td>15.25</td>
<td>147.61</td>
</tr>
</tbody>
</table>

Image of $4096 \times 4096$ pixels. Times in ms.
Summary
Summary

HIPA$^{cc}$

- domain-specific language for image preprocessing
- optimizations tailored to application domain
- architecture model for GPU accelerators
- target-specific code generator for CUDA and OpenCL
- transformations based on
  - domain knowledge
  - architecture information
- provides: performance, productivity and portability

Recent Work

- three new target backends
  - Renderscript
  - Renderscript GPU
  - Filterscript
- support for Midgard architecture (ARM Mali T604)
Questions?

HIPA\textsuperscript{cc} framework sources released under \textit{Simplified BSD License}.

https://sourceforge.net/projects/hipacc
Results: Gaussian Blur, 5 × 5 window, 2048 × 2048 pixel

Samsung Exynos 5

Qualcomm Snapdragon S4 Pro