This is the author's version of the work. The definitive work was published in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE), Dresden, Germany, March 24-28, 2014.*

Code Generation for Embedded Heterogeneous Architectures on Android

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

Department of Computer Science,
University of Erlangen-Nuremberg, Germany.

***Abstract*—The success of Android is based on its unified Java programming model that allows to write platform-independent programs for a variety of different target platforms. However, this comes at the cost of performance. As a consequence, Google introduced APIs that allow to write native applications and to exploit multiple cores as well as embedded GPUs for compute-intensive parts. This paper proposes code generation techniques in order to target the Renderscript and Filterscript APIs. Renderscript harnesses multi-core CPUs and unified shader GPUs, while the more restricted Filterscript also supports GPUs with earlier shader models. Our techniques focus on image processing applications and allow to target these APIs and OpenCL from a common description. We further supersede memory transfers by sharing the same memory region among different processing elements on HSA platforms. As reference, we use an embedded platform hosting a multi-core ARM CPU and an ARM Mali GPU. We show that our generated source code is faster than native implementations in OpenCV as well as the pre-implemented script intrinsics provided by Google for acceleration on the embedded GPU.**

I. Introduction

The steady desire for new applications, augmented reality, and higher display resolutions drives the development of embedded platforms and the need for faster, more powerful processors at the same time. As a consequence, today's mobile platforms found in smartphones and tablets host multi-core Central Processing Units (CPUs) and even programmable embedded Graphics Processing Units (GPUs) to deliver the demanded performance. This raises the question how to harness the processing power of these parallel and heterogeneous platforms. As a remedy, Google proposed two new parallel programming concepts for Android that allow to target CPUs as well as GPUs: Renderscript and Filterscript. These programming models were designed for the predominant application domain of image processing with portability in mind.

While applications in Android use Java as programming language, Renderscript and Filterscript are based on C99. Hence, Java programmers have to write low-level C code in order to benefit from the high performance that is provided by these new programming models. As a remedy, this work proposes code generators that allow to automatically generate Renderscript and Filterscript code. The generated target code is derived from a Domain-Specific Language (DSL) for image processing algorithms as introduced in [1]. The proposed code generators for Renderscript and Filterscript are based on the existing compiler infrastructure, which provides back ends for CUDA and OpenCL on discrete, standalone GPU accelerators.

The focus of this work is on the code generation that allows to target the different components in today's heterogeneous embedded platforms:

- We present the first code generator for Renderscript and Filterscript on Android platforms starting from an abstract high-level representation. The generated implementations are even faster compared to the target-specific implementations in the Open Source Computer Vision (OpenCV) framework. At the same time, the algorithm description is compact and requires only a fraction compared to available highly optimized implementations.
- We generate target code for embedded Heterogeneous System Architecture (HSA) platforms. With HSA, CPU and GPU share the same physical memory (see Figure 1). This allows us to avoid extensive memory transfers and enables the employment of heterogeneous resources where the same data has to be accessed frequently from different compute resources.

978-3-9815370-2-4/DATE14/©2014 EDAA

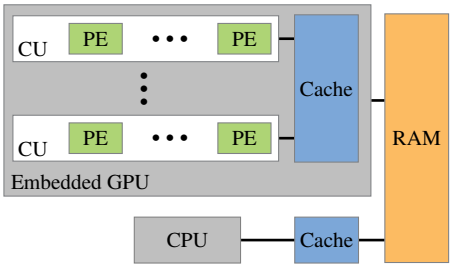

Figure 1: Structure of a typical HSA platform with an embedded GPU.

II. Programming Models on Embedded Devices

The Android operating system is widespread on end user mobile devices but it becomes more and more interesting to use it also on other embedded devices as found in the industrial automation or the automotive sector. For example, compute-intensive advanced driver assistance systems such as self- and convoy-driving technology, parking guidance, and better infotainment systems are becoming more and more important for automotive.

For such compute-intensive tasks, energy efficient embedded GPUs are of particular interest. However, there exists no common programming language that allows to harness also embedded GPUs. Android, for example, provides multiple programming models that allow to target a huge variety of devices with different usage scenarios in mind:

a) SDK: The Android Software Development Kit (SDK) is based on the Java programming language. The SDK source code is compiled to bytecode that is executed by the DalvikVM, which is either directly interpreting the bytecode or compiling it into machine-specific instructions and executing it. Hardware-specific characteristics are hidden and not exposed to the programmer. For that reason it is not possible to gain benefit from certain hardware-specific optimizations like vectorization. This dramatically limits the overall achievable performance.

b) NDK: The Android Native Development Kit (NDK) promises much better execution performance. Even though complete Android applications can be developed using C/C++ in the NDK, native code is usually executed by an application that is written using the SDK through the Java Native Interface (JNI).

Using the NDK, hardware-specific characteristics are transparent to the programmer and can be utilized, for instance, by compiler intrinsics. For efficiently supporting a wide range of different CPU architectures, native code needs to be rewritten beforehand in a way that exploits particular architecture features to achieve high performance. For that reason, usually only compute-intensive code segments are realized in native parts of an Android application.

c) Renderscript: Google first introduced the Renderscript programming language in 2011. The aim of this new language is to provide a programming model that avoids performance issues of the SDK without introducing portability problems the NDK suffers from. Renderscript is based on C99 and provides additional support for vector types. The Renderscript front end compiler generates an intermediate representation which is then further compiled to native code that is optimized for a specific available target architecture like CPUs, Digital Signal Processors (DSPs), and GPUs.

Unlike other parallel computing Application Programming Interfaces (APIs) such as OpenCL or CUDA, Renderscript was not designed with performance as the primary goal. Hardware-specific features like local memory are hidden from the programmer and, hence, target-specific optimizations like tiling may not be exploited. The number of threads is directly inferred from the output buffer where each element is processed by a single thread. Another limitation is that the actual execution target (CPU, GPU, DSP) cannot be specified and is automatically chosen by the runtime system. These limitations ensure portability at the expense of absolute performance.

d) Filterscript: Filterscript is a subset of Renderscript with certain limitations and stricter constraints to ensure a wider compatibility among CPUs, GPUs, and DSPs. Filterscript files are used and compiled the same way as Renderscript.

The major difference is that pointers are not allowed. Therefore, memory cannot be directly read using linear arrays. Instead, the provided access API functions must be used. Only gather reads are supported, which means that only one output value can be written per thread (in contrast to scatter writes). Instead of assigning the value to a buffer, it is returned by the kernel function and written to the global ID of the current thread. Further limitations are relaxed floating point precision and lack of 64-bit built-in types.

e) OpenCL: The Open Compute Language (OpenCL) programming model is not officially supported on Android devices. However, on recent devices, like the Nexus 4 smartphone and the Nexus 10 tablet, a working OpenCL driver can be found within the system libraries. Note that Google pulled the unofficial OpenCL support in Android 4.3.

Providing a well known common API like OpenCL cannot hide the fact that embedded GPU architectures vary considerably from their desktop counterparts. For instance, the local memory is not dedicated on-chip memory, but is instead mapped to global memory. Another major difference is that compute cores of most embedded GPUs are Very Long Instruction Word (VLIW) architectures and gain a huge benefit from vectorization or the use of vector types.

OpenCL supports the allocation of host accessible memory using `map()` and `unmap()`. On desktop GPUs, this feature can be used to define page-locked host memory, which may speed up memory transfers. On HSA platforms like the ARM Mali [2] and AMD Fusion [3], these operations are used to avoid copying buffers. Because the main memory is shared among CPU and GPU, both can access the same memory region without copies.

III. Target Code Generation

This section introduces the target code generation for the parallel computing APIs introduced in Section II. We use a DSL for image processing as basis for target code generation and employ source-to-source translation to target different parallel programming models.

A. The Heterogeneous Image Processing Acceleration (HIPA^{cc}) Framework

The DSL provided by the HIPA^{cc} framework [1] is based on C++ and provides built-in classes for two-dimensional images and other objects such as filter masks. An image in the DSL stores the image pixels and can be initialized from plain C data arrays:

```
uchar *image = readPGM(&width, &height, "lena.pgm");
Image<uchar> in(width, height);
in = image;
```

Similarly, a filter mask can store the stencil used for a convolution:

```
const float filter_mask[3][3] = {
  { 0.057118f, 0.124758f, 0.057118f },
  { 0.124758f, 0.272496f, 0.124758f },
  { 0.057118f, 0.124758f, 0.057118f }
};
Mask<float> mask(filter_mask);
```

Using these data abstraction classes, computations on multi-dimensional image objects can be defined. A computational kernel is defined as a C++ class that holds a `kernel()` method describing the computation on a single pixel. Within the kernel method, all memory accesses are relative to the current pixel and only members of the C++ class can be accessed. Considering the Gaussian blur filter as an example, its computation can be expressed using relative memory accesses to the `mask` filter mask and the `input` image. The result is written back using the `output()` method:

```
void kernel() {
  float sum = 0;
  int range = size/2;

  for (int yf = -range; yf <= range; ++yf)
    for (int xf = -range; xf <= range; ++xf)
      sum += mask(xf, yf) * input(xf, yf);

  output() = (uchar) sum;
}
```

A more concise and expressive syntax for common computational patterns such as convolutions are provided by the HIPA^{cc} DSL as well: the `convolve()` method describes the convolution of an image with a mask:

```
void kernel() {
  output() = convolve(mask, SUM, [&] () -> float {
               return mask() * input(mask);
             });
}
```

Using the convolve method is not only more compact, it gives the source-to-source compiler also more freedom to optimize the code.

The HIPA^{cc} compiler [1] generates target CUDA and OpenCL code from programs written in this DSL. Using source-to-source translation, instances of DSL C++ classes are replaced by corresponding API calls to the CUDA and OpenCL runtime library provided by HIPA^{cc}. Compute kernels, in contrast, are

not mapped one-to-one to corresponding CUDA and OpenCL. Instead, the Abstract Syntax Tree (AST) of a kernel is analyzed and optimizations are applied such as staging image pixels into local memory or mapping multiple iterations to one GPU thread (loop unrolling).

Memory accesses are then redirected to memory fetches from global memory, texture memory, or local memory—depending on the target device. Similarly, *Mask* accesses get mapped to constant memory or propagated as constants in case the operator is described using the `convolve()` function.

B. Renderscript and Filterscript Support for HIPA^{cc}

We have extended HIPA^{cc} for Renderscript and Filterscript support, adding a new back end for each API. Program parts of the DSL responsible for resource management are mapped to corresponding commands in the runtime library, which we provide. The compute-intensive kernels, however, are translated into Renderscript and Filterscript kernels. These get initialized at program start and can be executed afterwards.

1) Memory Access Mapping: As highlighted in Section II, memory accesses are handled differently in OpenCL, Renderscript, and Filterscript. Therefore, the introduced back ends map reads and writes to an *Image* to corresponding API calls and memory array accesses. To illustrate this, we consider a simple read from an *Image*, followed by a write in HIPA^{cc}:

```
Image<uchar> input;
...
void kernel() {
  uchar val = input();
  output() = val;
}
```

In OpenCL, these memory accesses are mapped to 1D memory arrays that are added to the signature of the kernel function with corresponding attributes indicating that the arrays reside in global CPU or GPU memory. In case neighboring pixels are read, the x and y index is adjusted accordingly:

```
__kernel void kernel(__global const uchar *input,
    __global uchar *output, ...) {
  uchar val = input[y*width + x];
  output[y*width + x] = val;
}
```

Renderscript provides data buffers for storing image data (`rs_allocation`) and `rsGetElementAt` API calls for reading/writing data elements. Rather than writing the result to the current iteration point (i. e., writing to the first kernel parameter), the result is stored to `_iter`, the currently processed pixel:

```
rs_allocation input;
...
void kernel(uchar *_iter, uint32_t x, uint32_t y) {
  uchar val = rsGetElementAt_uchar(input, x, y);
  *_iter = val;
}
```

In Filterscript, data buffers are defined and read as in Renderscript, but no API calls are provided for storing results. Instead, the result for the current thread (pixel) is returned by the kernel:

```
rs_allocation input;
...
uchar __attribute__((kernel)) kernel(uint32_t x,
    uint32_t y) {
  uchar val = rsGetElementAt_uchar(input, x, y);
  return val;
}
```

In order to map the execution of a kernel in Renderscript and Filterscript either to the CPU or to the GPU, environment variables are used.

In order to achieve high performance, the kernel has to be mapped to the memory hierarchy of the target architecture. GPUs provide multiple memory types apart from the *global memory* that are optimized for different access patterns such as *constant memory*, *texture memory*, or *local memory*. OpenCL allows to explicitly use these memory types in the source program. For example, filter masks are typically mapped to *constant memory* and read-only images with high spatial or temporal locality to *texture memory* or *local memory*. In contrast to this, Renderscript and Filterscript do not support any explicit mapping to the memory hierarchy.

2) Iteration Space Mapping: HIPA^{cc} allows the programmer to define a Region of Interest (ROI) in the output image to be computed. Similarly, only an ROI on input images can be read. This allows to work on images of different size in one kernel and to process only image regions of interest. The ROI on the output image defines also the iteration space and the number of *threads* required for kernel execution. This iteration space size is used as launch configuration in parallel compute APIs such as CUDA and OpenCL and offsets to the image are passed to the kernel in order to process only the ROI. However, the native API of Renderscript and Filterscript does not provide launch configurations up to Android 4.4. Instead, the *buffer* holding the image data defines also the launch configuration. That is, for each pixel in the image a thread is started resulting in an *index space* that is larger than the *iteration space* defined by the programmer.

To overcome this deficiency, there exist three approaches. First, we can define a buffer with the size of the iteration space. This temporary buffer stores the result of the kernel and is copied back to the ROI in the output image as specified by the programmer. This approach requires additional memory of $ROI_{width} \times ROI_{height}$ and requires one additional memory transfer of the same size. Second, we can define a dummy buffer with dimensions equal to the iteration space and use this buffer to provide the index space. Result pixels are not stored to this buffer, but to the buffer associated with the output image of the kernel. This approach requires in theory no additional memory and no additional memory transfers, but can only be used for Renderscript[1]. In Filterscript, the output pixel is not written to a buffer, but returned within the kernel. Third, we can use an index space with a size of the whole output image and add guards to the kernel so that only threads of the index space calculate pixels that are also part of the iteration space. This launches $(IMG_{width} \times IMG_{height}) - (ROI_{width} \times ROI_{height})$ additional threads that do not compute pixels. While this approach is valid for Renderscript, the behavior is undefined in Filterscript in case no return statement is executed. Hence, we read the corresponding pixel value from the output image for index points outside of the iteration space and return those. This requires $(IMG_{width} \times IMG_{height}) - (ROI_{width} \times ROI_{height})$ additional memory reads and writes for the Filterscript implementation, but has no memory allocation overhead. This approach is the only one that provides a valid iteration space mapping for Filterscript with only little overhead. Thus, this approach is followed by the code generator in this work.

[1]However, it turns out that the Renderscript runtime still allocates memory for the buffer although no data is associated with the buffer.

3) Vector Support: HIPA^{cc} supports scalar data types for discrete GPUs from AMD and NVIDIA. These GPUs schedule scalar instructions to single lanes of their Single Instruction, Multiple Data (SIMD)-like architecture. Adjacent threads are mapped to adjacent lanes. The embedded CPUs and GPUs considered here require vector instructions, though. Otherwise only a fraction of peak performance can be achieved. Therefore, we add vector type support to HIPA^{cc}, which is compatible with the syntax in OpenCL and Renderscript.

C. Support for HSA Memory Management

The HIPA^{cc} DSL was designed for desktop systems and therefore has no particular support for HSA platforms. Device memory is abstracted from the developer by the `Image` class in HIPA^{cc}. Memory transfers to the device are handled implicitly by the framework. On HSA targets, it is possible to share memory between CPU and GPU by using host accessible memory that must be allocated using OpenCL API calls.

We added support for HSA platforms to HIPA^{cc} by extending its memory management to abstract host memory as well and implicitly manage `map()` and `unmap()` operations. This additional abstraction has the benefit that a) faster page-locked memory can be utilized on desktop systems, and b) the same memory region can be used for the CPU and GPU on HSA platforms in order to avoid memory copies. In case memory is allocated by third party frameworks (e. g., OpenCV or FreeImage), the programmer can still manage host memory explicitly.

IV. Evaluation and Results

We evaluate our results on an Arndale Board with a Samsung Exynos 5250 running Android 4.2 [4]. To ensure a fair comparison of the generated code, we first analyze timing behavior for every target API. The portability and performance of the described Renderscript and Filterscript back ends are then evaluated by considering the Gaussian blur filter—the only filter available as script intrinsic[2] with execution enabled on the GPU. We further compare the performance of our implementations in OpenCL against script intrinsics and the vectorized implementations provided by the OpenCV library. Moreover, in order to also evaluate non-functional properties such as productivity, several applications written in HIPA^{cc} for execution on mobile platforms are evaluated in terms of Lines of Code (LoC).

A. Evaluation Environment

The Exynos 5250 MPSoC as found on the Arndale Board [4] is based on a dual-core ARM Cortex-A15 CPU with NEON SIMD extension (128-bit), an ARM Mali T604 GPU with four cores (each SIMD4, 128-bit), and 2 GB of DDR3 RAM. Similar hardware can also be found in the Nexus 10 tablet. While the GPU has twice as many cores, they run at only 533 MHz compared to 1.7 GHz for the CPU cores. This allows for many computationally intensive tasks a more energy efficient processing on the GPU.

To cross-compile the generated target code we use the standalone toolchain provided by the NDK. We compare the execution times of 100 runs of box blur implementations using different APIs (filter window size of 3×3; image size of 2048×2048). The corresponding box plot (see Figure 2) shows that all measurements have outliers with significantly higher execution time (up to $2\times$),

[2]Script intrinsics are optimized and pre-implemented filtering functions provided in Android.

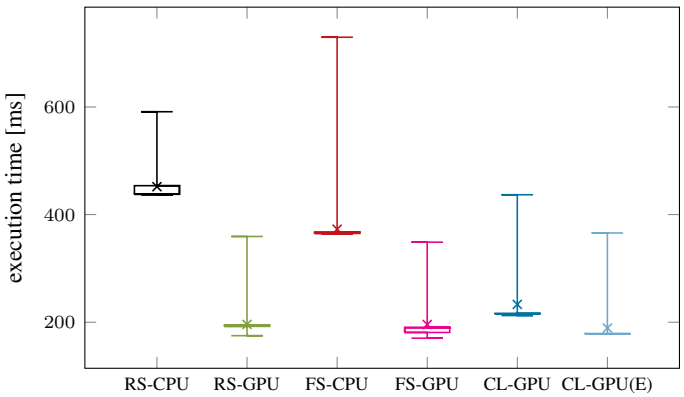

Figure 2: Box plot for 100 runs of the box filter when using the CPU and GPU back ends for Renderscript (RS), Filterscript (FS), and OpenCL (CL). CL-GPU(E) performs timing on the embedded GPU while all other variants perform timing on the CPU.

which we attribute to the power/thermal management. This is the case even for the execution times of the OpenCL implementation using events on the GPU for time measurement (CL-GPU(E)). Therefore, we use the median in the following when execution times are stated and time kernels on the CPU with the same time measurement overhead for all APIs.

B. Gaussian Blur Filter

The Gaussian blur filter implementation from Section III-A requires $(size_x \times size_y) + 1$ memory reads/writes to execute the kernel. Since a separated implementation requires only $size_x + 1 + size_y + 1$ reads/writes, we separate the filter into a row and column component for evaluation. This reduces the memory accesses for a filter window size of 5×5 from 26 to 12 per pixel. Also the other considered implementations in OpenCV and script intrinsics use this implementation variant.

1) Implementation Variants: In the following, we further consider three implementation variants based on different properties in the source code: the implementation as indicated in Section III-A using a) nested loops and a filter mask that is not constant, b) nested loops and a constant filter mask, and c) the `convolve()` method and a constant filter mask.

The generated target code variants employ optimizations that make use of the properties each source code variant features: a) the filter mask is allocated on the host side, b) the filter mask is defined as statically allocated constant array at global scope within the kernel source file, and c) the lambda-function is completely unrolled and the filter mask constants are inserted for each iteration (constant propagation).

2) Implementation Results: Table I shows the execution times for the different generated code variants. It can be seen that using the different GPU back ends leads to similar execution times with OpenCL being the fastest. GPU execution is faster than execution on the CPU, which emphasizes also additional power efficiency benefits of embedded GPUs. When looking at Filterscript results, it has to be considered that Filterscript uses relaxed floating point precision and leads, hence, for compute-bounded kernels using floating-point arithmetic to much faster execution times.

The different code variants reflect the optimization potential in particular on embedded GPUs: using constant memory improves the performance slightly ($\approx 5\,\%$ for Filterscript and OpenCL) and loop unrolling in combination with constant propagation

Table I: Execution times in ms for the Gaussian blur filter for the generated implementations (Renderscript, Filterscript, and OpenCL) as well as for hand-tuned implementations in OpenCV and script intrinsic implementations for an image of 2048×2048 pixels and a filter window size of 5×5.

	RS-CPU	RS-GPU	FS-CPU	FS-GPU	CL-GPU
Normal	393.04	263.06	278.79	279.72	214.66
ConstMask	323.83	265.86	287.34	270.63	203.27
ConstProp	285.53	193.42	223.95	182.67	153.05
			CV-CPU	SI-CPU	SI-GPU
Hand-tuned			847.48	110.12	343.55

improves performance significantly (up to 35 % for Filterscript). This highlights the need of device-specific optimizations: on discrete GPUs, the benefit of constant propagation compared to using constant memory is negligible since discrete GPUs feature special caches that are optimized for this memory access patterns. Similarly, using local memory in order to benefit from locality has the opposite effect on embedded GPUs: there is no dedicated local memory and the execution takes twice as long (not shown) using OpenCL.

Similar performance improvements can be observed on the CPU. For the second variant using FS-CPU and RS-GPU, the execution is slightly slower, which we attribute to problems in the compiler.

Comparing the results against script intrinsics and implementations in OpenCV (see also Table I) shows that our generated code is significantly faster ($2.25\times$) compared to script intrinsics on the GPU. However, script intrinsics on the CPU are twice as fast as our generated code. Compared to the highly optimized OpenCV implementation using NEON-specific optimizations, our generated Renderscript implementation on the CPU is $2.60\times$ faster. The current OpenCV implementation only supports vectorization, but no parallelization. Assuming no parallelization overhead in OpenCV, their implementation would still be slower.

3) Discussion: Although Renderscript does not provide any options for hardware-specific optimizations, we have shown that we get competitive performance, even without sacrificing portability requirements. If OpenCL were officially supported on Android, it would probably be number one choice for most developers.

It can be seen that our GPU implementation (RS/FS/CL-GPU) performs better than the pre-implemented and hand-tuned script intrinsics version of the filter (SI-GPU). On the CPU, it was quite astonishing to see that script intrinsics is by far the fastest implementation (SI-CPU). The reason for that is that script intrinsics heavily utilizes cache-aware interleaving of pixel rows, which is highly optimized for CPU targets. These optimizations cannot be provided in parallel programming models such as OpenCL or Renderscript. This is due to the fact that these models only allow to describe an algorithm (with respect to architectural characteristics, such as data layout or local memory), but not to manually define a fine-grained schedule.

C. Application Performance

We consider Sobel and Laplace operators, the Gaussian blur and FIR filters, as well as a Harris Corner detector [5] for performance and memory transfer overhead considerations. Corresponding highly optimized implementations in OpenCV serve as reference.

All implementations in OpenCV are vectorized while our algorithm description of the Harris Corner detection operates

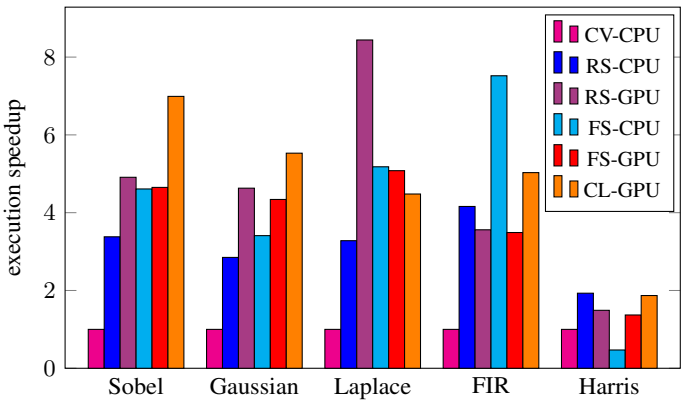

Figure 3: Execution speedup of Sobel and Laplace operators, the Gaussian blur and FIR filters, as well as a Harris Corner detector using different APIs. The graphs are normalized to OpenCV (CV-CPU) on the CPU.

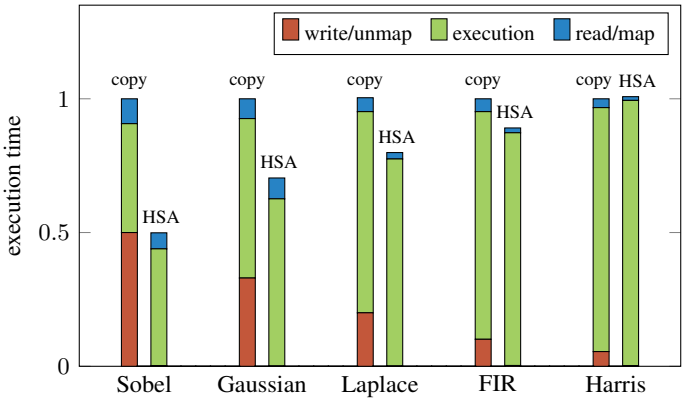

Figure 4: Execution times of the different applications in OpenCL. Timings include time spent on read/write (memory transfers) and map/unmap (HSA). The graphs are normalized to the execution without HSA (copy) on the CPU.

only on the luminance channel. Figure 3 shows the speedup of the applications using different APIs on the Arndale board compared to the hand-tuned implementations from OpenCV. The results affirm that our generated code is faster than the hand-tuned implementations in OpenCV. Only for the Harris Corner detector the lack of vectorization in our algorithm description is noticeable.

Figure 4 shows the benefit of exploiting the HSA feature to avoid memory copies between CPU and GPU. The benefit is inversely proportional to the time spent on the actual computation. For short-running applications, the savings are up to 50 % (Sobel), while for long-running applications only small improvements can be expected. HSA suffers from overheads for synchronization (waiting for writes to finish, flushing caches), instead of memory transfer overheads. Interestingly, the execution time itself with HSA enabled is slightly higher (3–8 %) for all applications, which might improve in the future. This leads to a marginally higher overall execution time for the more complex Harris Corner application, where memory transfers are negligible. This shows that HSA can reduce memory transfer overheads significantly for applications that share data frequently between CPU and GPU.

D. Productivity and Portability

The high efficiency of OpenCV comes at the cost of portability and productivity. For example, the manually vectorized implementations of the Gaussian blur filter for only the CPU requires 1641 LoC

in order to support different vector extensions. This includes neither the LoC for the available GPU back end using CUDA as target language nor the LoC for the currently developed GPU back end using OpenCL as target language. The LoC include only the hardware-specific implementation variants without any object-orient class abstraction. In contrast, the description of the separated Gaussian blur filter algorithm requires only 22 LoC in HIPA^{cc}. From this high-level algorithm description, our Renderscript back end (CPU) generates 1951 LoC tailored to the target architecture as well as to the algorithm at hand (i. e., considering characteristics such as the filter mask size, the memory access pattern, or data reuse). The large code length results from aggressive loop unrolling and constant propagation of local operators described as lambda-function in order to increase Instruction-Level Parallelism (ILP) and reduce memory accesses. Note that the iterations of the iteration space have not been *unrolled* in the generated code.

Table II summarizes the LoC for the considered applications, showing that HIPA^{cc} may improve productivity significantly. The implementation and optimization efforts associated with manual implementations lead to situations where an algorithm is only available for one back end (e. g., the CPU), but not for another (e. g., the GPU). Generating the implementation from a common description in a high-level language provides a remedy for this dilemma and provides portability across different target architectures without rewriting applications for each target.

V. Related Work

While there exist a wide range of frameworks and compilers that generate low-level assembly code for embedded architectures, there is only little related work on targeting the new compute APIs Renderscript and Filterscript.

For example, the Portland Group introduced the PGCL framework [6], which adds support for OpenCL to Android on the ST-Ericsson NovaThor platform. Their compiler supports the NEON instruction set and vectorization for ARM multi-core CPUs. Halide [7], a DSL for image processing provides a back end for ARM NEON, and the OpenCV [8] library utilizes also the ARM NEON instruction set. These frameworks have all in common to target only the CPU. More recently, a source-to-source translator for mapping annotated Java code (using pragmas similar to OpenMP within source code comments) to Renderscript and OpenCL was presented [9]. Hence, the developer requires profound knowledge of OpenMP, while our domain-specific approach abstracts from low-level architecture details.

Qian, Zhu, and Li [10] compare and analyze the programming model of the SDK, NDK, and Renderscript considering usability aspects such as programmability, but also performance. They conclude that Renderscript provides the best performance while preserving portability at the cost of manual memory management and difficult library extensibility. The solution presented in this work does not have these limitations since the Renderscript code and the supporting files are automatically generated. Moreover, we support also Filterscript and OpenCL as back ends, which allows to map program parts to the GPU and CPU.

Table II: Lines of code for the OpenCV implementation, HIPA^{cc} DSL code and generated Renderscript code.

	Sobel	Gaussian	Laplace	FIR	Harris
OpenCV	1681	1641	1712	982	2247
HIPA^{cc} DSL	16	22	11	11	68
Renderscript	1915	1951	8575	3680	4265

GMAC [11] provides similar memory management abstractions for discrete GPUs (CUDA, OpenCL), but do not consider HSA platforms. However, the proposed abstractions can also be integrated into GMAC.

VI. Conclusion

We presented a code generator for Renderscript and Filterscript in the domain of image processing by utilizing the HIPA^{cc} DSL. All code variants are automatically generated from a common description that is highly portable and performs well on both the CPU and the GPU. The code generator has been integrated into the HIPA^{cc} framework and is available as open-source under http://hipacc-lang.org.

Using this code generator, we have shown that we are able to produce efficient code for different parallel programming models on embedded devices. We were able to show that—when it comes to performance—our generated Renderscript and Filterscript code is actually faster than implementations using other APIs with comparable portability. We also perform better than the target-optimized OpenCV library on the CPU. On the GPU, it was even possible to surpass the heavily optimized proprietary script intrinsics implementation.

Acknowledgment

This work is supported by the German Research Foundation (DFG), as part of the Research Training Group 1773 “Heterogeneous Image Systems”.

References

[1] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert, “Generating Device-specific GPU Code for Local Operators in Medical Imaging”, in *International Parallel & Distributed Processing Symposium (IPDPS)*, IEEE, May 2012, pp. 569–581.

[2] ARM. (2013). Mali-T600 Series GPU OpenCL – Developer Guide.

[3] B. A. Hechtman and D. J. Sorin, “Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips”, in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Jun. 2013.

[4] ArndaleBoard, *Samsung Exynos 5 Dual Arndale Board*, http://www.arndaleboard.org, 2012–2013.

[5] C. Harris and M. Stephens, “A Combined Corner and Edge Detector”, in *Alvey Vision Conference*, 1988, pp. 147–151.

[6] The Portland Group, *PGI OpenCL Compiler for ARM*, http://www.pgroup.com/products/pgcl.htm, 2011–2013.

[7] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines”, *ACM Transactions on Graphics (TOG)*, vol. 31, no. 4, 32:1–32:12, Jul. 2012.

[8] Willow Garage, *Open Source Computer Vision (OpenCV)*, http://opencv.willowgarage.com/wiki, 1999–2013.

[9] A. Acosta and F. Almeida, “Towards an Unified Heterogeneous Development Model in Android”, in *International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar)*, Springer, Aug. 2013.

[10] X. Qian, G. Zhu, and X.-F. Li, “Comparison and Analysis of the Three Programming Models in Google Android”, in *Asia-Pacific Programming Languages and Compilers Workshop (APPLC)*, ACM, Beijing, China, Jun. 2012, pp. 1–9.

[11] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-M. W. Hwu, “An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems”, in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, Mar. 2010, pp. 347–358.