API Compilation for Image Hardware Accelerators

FABIEN COELHO and FRANÇOIS IRIGOIN, MINES ParisTech

We present an API-based compilation strategy to optimize image applications, developed using a high-level image processing library, onto three different image processing hardware accelerators. We demonstrate that such a strategy is profitable for both development cost and overall performance, especially as it takes advantage of optimization opportunities across library calls otherwise beyond reach. The library API provides the semantics of the image computations. The three image accelerator targets are quite distinct: the first one uses a vector architecture; the second one presents an SIMD architecture; the last one runs both on GPGPU and multicores through OpenCL. We have adapted standard compilation techniques to perform these compilation and code generation tasks automatically. Our strategy is implemented in PIPS, a source-to-source compiler which greatly reduces the development cost as standard phases are reused and parameterized. We carried out experiments with applications on hardware functional simulators and GPUs. Our contributions include: (1) a general low-cost compilation strategy for image processing applications, based on the semantics provided by library calls, which improves locality by an order of magnitude; (2) specific heuristics to minimize execution time on the target accelerators; (3) numerous experiments that show the effectiveness of our strategies. We also discuss the conditions required to extend this approach to other application domains.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Retargetable compilers; D.3.2 [Programming Languages]: Language Classifications—Specialized application languages; I.4.0 [Image Processing and Computer Vision]: General—Image processing software

General Terms: Languages, Performance, Design, Algorithms

ACM Reference Format:

Coelho, F. and Irigoin, F. 2013. API compilation for image hardware accelerators. *ACM Trans. Architect. Code Optim.* 9, 4, Article 49 (January 2013), 25 pages.

DOI = 10.1145/2400682.2400708 <http://doi.acm.org/10.1145/2400682.2400708>

1. INTRODUCTION

Heterogeneous hardware accelerators, based on GPU, FPGA, or ASIC, are used to reduce the execution time, the energy used, and/or the cost of a small set of application-specific computations, or even the cost of a whole embedded system. Thanks to Moore's law, their potential advantage increases with respect to general-purpose cores which do not gain much from the increase in area and transistor number. However, these gains are often undermined by large software development cost increases, as programmers knowledgeable in the target hardware must be employed, and as this investment is lost when the next hardware generation appears.

We present a compilation strategy to map image processing applications, initially developed on top of a high-level image library onto several accelerators, by generating optimized configurations and relying on image-level runtimes. Our approach, although

This work is funded by the French ANR through the FREIA project [Bilodeau et al. 2011].

Authors' addresses: F. Coelho (corresponding author), F. Irigoin, CRI, Math. & Systems, MINES ParisTech, France; email: fabien.coelho@mines-paristech.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART49 \$15.00

DOI 10.1145/2400682.2400708 <http://doi.acm.org/10.1145/2400682.2400708>

original because it operates at function level and not at loop level, is relatively easy to implement as many phases are known and reusable compilation techniques. Only the last code generation phase is fine-tuned and target-specific. To demonstrate the efficiency of our approach, we have implemented our strategy for three different targets. The first and second ones are dedicated image processing devices implemented as a SoC on an FPGA chip. The third and last target is the range of multicore GP and GPGPU processors that are targeted with OpenCL.

The first target is a heterogeneous processor fitted with a vector image processing accelerator, SPoC [Clienti et al. 2008]. Dozens of elementary image operations (additions and other arithmetic pixel operations, simple stencil operations like convolutions, erosions or dilatations, thresholdings, finding the min/max pixels and their coordinates. . .) can be chained together. The SPoC accelerator is fed pixel by pixel with two input images, and two resulting images can be stored back at the end of the pipeline. The second accelerator, Terapix [Bonnot et al. 2008], is a 128 SIMD processor array. Each processing element performs the very same operation on its local pixels at each clock tick. As images are usually too big to fit the available accelerator memory, overlapping 128-pixel wide tiles must be sent to and retrieved from the accelerator in order to perform a sequence of operations on a whole image. The third back-end targets OpenCL [Khronos Group 2008] which provides a portable programming environment for GPGPU hardware and for multicore processors.

The application development environment relies on the FREIA image processing API [Bilodeau et al. 2011], which includes basic image operations and high-level composed image operations implemented on top of the basic ones. The developer has no knowledge of the potential target hardware accelerators. Multiple implementations of FREIA are available: An implementation based on a software library is used for functional tests. Then accelerated versions have been developed that take advantage of hardware accelerator capabilities, either at basic image operator level or even specialized at composed image operation level. The specialized libraries offer better performance, but optimization opportunities are still available across functional boundaries. Although the library layer provides application portability over accelerators, it does not provide the time, energy, and cost performance expected from them.

In order to reach better performance, library developers may be tempted to increase the sizes of their APIs to provide more opportunities for optimized code, but this is an endless process leading to over-bloated libraries and portability issues: thousands of entries are defined in the Vector Signal Image Processing Library [VSIPL Forum 2008]. Unlike the pure library approach, we use only the most basic hardware operator library implementation, and our compiler composes these operations as needed to derive efficient versions of large image expressions extracted interprocedurally from the whole application. We thus see the image API as a domain-specific programming language, and compile this language for the low-level target architectures.

In the remainder of this article, we first introduce our running example, which is a sample of the application domain, along with other typical application examples (Section 2). Then we outline our compilation strategy and detail the target-independent phases (Section 3). The next three sections introduce each target and describe the corresponding back-end phases for SPoC (Section 4), Terapix (Section 5), and OpenCL (Section 6). Then we present our implementation (Section 7), report extensive experimental results (Section 8) obtained with hardware simulators, multicores, and GPUs, and finally discuss the related work (Section 9) before concluding.

2. APPLICATIONS AND RUNNING EXAMPLE

We first discuss our application domain and introduce a short but significant example.

Fig. 1. Out of position (*OOP*): airbag OK or not. (© MINES ParisTech)

Fig. 2. Video survey (*VS*): motion detection. (© THALES)

The FREIA project aimed at efficiently mapping image processing applications developed on top of a high-level API onto different hardware accelerators. Typical target applications extract information from one image or from a stream of images to detect such things as a license plate (LP), a car passenger out of position who could be harmed if the airbag is triggered (*OOP*, Figure 1), motion under a surveillance camera (*VS*, Figure 2), or macular degeneration in a picture of a retina (Retina). These image applications use all kinds of image processing operations, such as: AND-ing an image with a mask to select a subregion; MAXLOC-ating where the hottest point is; THR-esholding an image with values to select regions of interest; mathematical morphology [Soile 2003] operators. This framework created in the 1960's provides a well-founded theory to image analysis, with algorithms described on top of basic image operators. The FREIA project targets high-performance, possibly hardware-accelerated, very often embedded, high-throughput image processing. For this purpose, the software developer must invest time to reach the expected high performances for critical applications on selected hardware. Current development costs are high, as applications must be optimized from the high-level algorithmic choices down to the low-level assembly code and memory transfers for each hardware target. The project aims at reducing these development costs using optimizing compilers and dedicated runtimes.

The FREIA image API has been implemented in several ways. The first one is pure C, based on the Fulguro [Clienti 2008] open-source image processing library. It is used for the functional validation of applications. Two implementations are available for the SPoC accelerator: one uses SPoC only for the image operations of the FREIA API that are directly supported by the its instruction set, one operation at a time; the other is hand-optimized at library call level by taking full advantage of the SPoC capability for chained operations. Other versions of the library are hand-optimized for the Terapix SIMD accelerator and for OpenCL, mostly targeting GPGPUs.

```

#include <freia.h> // use FREIA user API
int main(void) {
    int32_t min, vol; // declarations...
    freia_data2d *in = freia_common_create_data(...); // AND od, og
    freia_dataio fin, fout; // input & output descriptors
    freia_common_open_input(&fin, 0); // AND fdout
    freia_common_rx_image(in, &fin); // get input image
    freia_aipo_global_min(in, &min); // some computations
    freia_aipo_global_vol(in, &vol);
    freia_cipo_dilate(od, in, 8, 10); // dilate with 8 neighbors at depth 10
    freia_cipo_gradient(og, in, 8, 10); // gradient with 8 neighbors at depth 10
    printf("input min=%d vol=%d\n", min, vol); // show results
    freia_common_tx_image(od, &fout); // AND og
    freia_common_destruct_data(in); // AND od, og...
    freia_common_close_input(&fin); // AND fdout
    return 0;
}

```

Fig. 3. FREIA API running example (some initialization and cleanup removed).

The code in Figure 3 was written as part of the FREIA project to provide a short test case significant for both the difficulties involved and the optimization potential of SPoC and Terapix. The test case contains all the steps of a typical image processing code: an image is read, intermediate images are allocated and processed, results are displayed. As it is short enough to fit in this article, we use it as a running example, together with excerpts from larger applications. There are few optimization opportunities for the SPoC accelerator at main level. The `min` and `vol` function calls belong are SPoC instructions. As they are next to each other and use the same argument, they may be merged into a unique call to the accelerator. The `dilate` and `gradient` functions are not part of the accelerator instruction set. With the basic library implementation, 33 calls to the accelerator are used per frame, most of them indirectly from the callees. The use of a hand-optimized SPoC implementation of the FREIA image library could result in only 6 accelerator calls, a $\times 5.5$ speedup¹ as calls to basic functions can be merged within the implementation of the FREIA composed functions. However, there are still cross-functional optimization opportunities: for instance the `dilate` operation is already computed within the `gradient` function. Our compiler strategy detects this redundancy and merges all image operations into only 2 accelerator calls, achieving an additional $\times 3$ speedup with respect to the hand-optimized library.

3. API-BASED COMPILATION STRATEGY

This section presents an overview of our compilation strategy, and details the preprocessing and DAG optimization phases. The 3 target-specific code generation phases will be described in the next 3 sections, together with their corresponding hardware.

Our overall strategy is based on a library-based Domain-Specific Language (DSL) for image applications, the FREIA API. Its semantics is provided by a combination of low-level basic operators and high-level operators implemented in C on top of the formers. Assuming the application is based on FREIA, we first build large image operation basic blocks, optimize the expression DAGs (Directed Acyclic Graphs) derived from these basic blocks, and then generate code for the target with accelerator-specific back-

¹The execution time of any call only depends on the input and output image sizes (see Section 4).

Phase 1. application preprocessing – enlarge basic blocks

- 1. simplification of safety checks
- 2. inlining of FREIA high-level library functions
- 3. constant propagation
- 4. loop full unrolling
- 5. only for SPoC: convergence while unrolling
- 6. control simplification
- 7. code flattening

Phase 2. DAG optimizations

- 1. DAG construction per basic block
- 2. operator normalization (*improve CSE stage*)
- 3. algebraic optimizations: constant image detection and propagation
- 4. common sub-expression elimination (CSE), with commutativity and reductions
- 5. dead image operation removal
- 6. forward and backward copy propagation
- 7. extraction of remaining copies

Target-specific back-ends for SPoC, Terapix and OpenCL.

Phase 3.1. SPoC configuration: map DAG onto hardware

- 1. DAG splitting and scheduling of sub-DAGs
- 2. instruction compaction and path selection
- 3. pipeline overflow management
- 4. unused image cleanup and image reuse

Phase 3.2. Terapix configuration: map DAG onto hardware

- 1. DAG splitting along scalar dependencies and connected components
- 2. scheduling and memory allocation, including double buffers
- 3. instruction generation, best tile size selected at runtime
- 4. unused image cleanup and image reuse

Phase 3.3. OpenCL code generation

- 1. DAG splitting along scalar dependencies and connected components
- 2. operator aggregation
- 3. OpenCL kernel generation
- 4. unused image cleanup and image reuse

Fig. 4. Outline of our compilation strategy: phases and stages.

ends. The implementation is easy because standard phases are reused, and attractive because significant speedups are obtained; see Section 8.

Figure 4 outlines the different phases of our compilation strategy. The first phase is a combination of mostly standard transformations to build larger basic blocks, and is target independent. It retrieves the basic image operations in the source-code. They are then aggregated into basic blocks. The second phase builds and optimizes the image expression DAGs by detecting common subexpressions or removing dead code. The third phase is specific to each target. The DAGs are partitioned according to hardware constraints, and then the code is generated and cleaned up.

Phase 1 – Application Preprocessing. The FREIA API [Bilodeau et al. 2011] and its Fulguro [Clienti 2008] implementation are designed for genericity over connectivity, image sizes, and pixel representations. Standard or advanced loop transformations cannot take advantage easily of such source-code because loops are distributed into different functions and elementary array accesses are hidden in function calls to preserve the abstraction over pixel structures.

To build large basic blocks with elementary image operations, control flow breaks, such as procedure call sites, local declarations, branches, and loops, must be removed, using key parameters set up by the main and propagated to callees. Several source-to-source transformations are used to that purpose: (1) Safety tests are simplified as the

```

freia_aipo_global_min(in, &min);      // some computations
freia_aipo_global_vol(in, &vol);
freia_aipo_dilate_8c(od, in, k8c);
freia_aipo_dilate_8c(od, od, k8c);   // repeated 8 more times
I_0 = 0;                             // scalar stuff..
tmp = freia_common_create_data(...); // image allocations...
freia_aipo_dilate_8c(tmp, in, k8c);
freia_aipo_dilate_8c(tmp, tmp, k8c); // repeated 8 more times
freia_aipo_erode_8c(og, in, k8c);
freia_aipo_erode_8c(og, og, k8c);    // repeated 8 more times
freia_aipo_sub(og, tmp, og);

```

Fig. 5. Excerpt of the main of our running example (Figure 3) after preprocessing.

application is assumed bug free before its optimization is started. This is done simply by replacing all check function calls by *true* at preprocessor level, so that the subsequent constant propagation and control simplification eliminates the corresponding error handling code. (2) Inlining suppresses functional boundaries around basic API functions. (3) Constant propagation reduces control complexity, as many tests can be simplified. (4) Loops are fully unrolled when the loop bounds are known. Note that only applicative loops *over* image operations are unrolled. Implicit loops over pixels stay hidden in the basic API functions and are not unrolled. (5) For SPoC, a particular optimization can be applied on while loops that encode idempotent image transformations. It is useful to detect and unroll these particular while loops for SPoC in order to fill up the accelerator pipeline which would stay mostly unused otherwise. Pattern matching is used to identify the loops that apply an image transformation until the resulting image is constant, and the found loops are unrolled. The useless operations added have almost no impact on the execution time as they are performed within an accelerator call, while the unrolling factor is a speedup. (6) Control simplification helps remove useless control and unreachable code. (7) Code flattening suppresses basic block breaks. This is mostly a C pretty-print issue for a source-to-source compiler: variables with identical names in different scopes must be renamed to merge the corresponding blocks. Block nesting is generated by inlining. The application order of these transformations is chosen to make the best use of the available information so as to simplify the code and obtain larger basic blocks. This overall approach is effective because the structural parameters of operations, such as the input image connectivity, are known statically. However, the actual image size is usually not needed at this stage, and, although the back-end phases can sometimes benefit from this information, it is not essential and is rather dealt with at runtime. Figure 5 shows the resulting code after automatic application of these transformations on the running example. It contains a sequence of elementary image operators mixed with scalar operations and temporary image allocations and deallocations.

Phase 2 – DAG Optimization. The basic blocks of the image application are analyzed to build expression DAGs over images (see Figure 6). Simple information about parameters, image or scalar, input and/or output, are derived automatically from C source stubs. The vertices of the DAG are operations to perform: they may be image operations or intermediate scalar operations depicted as question marks. Stencil operations are depicted by rectangles, arithmetic operations by trapeziums, thresholding by parallelograms, reductions by diamonds, image copy and input/output by circles. The arcs represent the def-use dependencies between operations. Arcs shown as black arrows embed image dependencies. White arrows stand for scalar dependencies. For instance,

Fig. 6. DAG sample from application Video Survey (VS – Figure 2)
(the upper part updates background b , the lower part detects movements in bright regions).

Fig. 7. Initial and optimized expression DAG for our running example (Figure 3)
(library function boundaries shown in dotted lines, common subexpression outlined in blue).

the result of a reduction on an image is used, after some computation, to threshold it. The DAG derived from our running example appears in the upper left side of Figure 7.

As the initial order of operation is not preserved, we also rename image variables and generate new temporaries in order to avoid conflicts that may create false dependencies when the code is regenerated. The many images introduced by this renaming phase, which amounts to generating a single-assignment form within the sequence, are mostly removed by a cleanup phase at the very end of the compilation process.

The image expression DAG is then optimized in a target-independent manner. First, a normalization phase is applied to replace some operators with equivalent ones, *e.g.*, “image sub constant pixel” is substituted by “image plus inverted constant pixel”, enabling more common subexpressions to be detected later. Second, algebraic simplifications are performed. In particular, constant image expressions, *i.e.*, expressions that lead to a fully monochromatic image, are detected and propagated as much as possible through expressions by reducing operator strength. For instance, as shown outlined in Figure 8, developers often xor an image against itself in order to generate a constant black image. Although perfectly functional, this technique is less efficient than the available hardware ALU operators to generate a black image, especially with the SPoC accelerator where the xoring technique directly saturates the two

Fig. 8. Initial and optimized DAG from application *Deblocking* (algebraic simplification outlined in blue, dead operations in gray are removed).

Fig. 9. Initial and optimized DAG from application *OOP* (Figure 1) (common subexpression outlined in blue, dead operations in gray are removed).

image paths available. Third, we apply standard compilation techniques: common subexpressions elimination, dead code suppression, and copy propagation [Aho et al. 2006] are performed at image level. Operator commutativity is taken into account to apply CSE, thanks to the image operator semantics which is carried by its name. Then image copies are forward propagated toward their uses, with the exception of copies on output images, which are backward propagated to their producers and generated directly. No temporary images are used. Remaining input and output image copies are extracted from the DAG to be performed by the host.

The optimization of the DAG in Figure 6 removes copy operations on input and within the graph. In our running example, the optimization detects that all operations of the *dilatation* are also performed within the *gradient*, so they have been removed to obtain the right-hand side of Figure 7, and the intermediate result is simply extracted. The optimized version of the DAG in Figure 9 removes both copies and a common subexpression involving 3 operations. These redundancies are not obvious to spot in the source-code. The result of this phase is an optimized image expression DAG ready to be mapped onto the available hardware.

Fig. 10. SPoC architecture, one vector unit, and stencil buffer.

4. SPOC ARCHITECTURE AND BACK-END

This section introduces SPoC, our first hardware target, focusing on the key constraints for the code generator. Then it details the algorithm which maps the optimized image operation DAG resulting from phase 2 onto the SPoC pipeline.

The structure of the SPoC processor [Clienti et al. 2008] (Figure 10 left) is similar to a simplified version of the 30-year old CDC Cyber 205 [Hockney and Jesshope 1988], specialized for image processing instead of floating-point computation. A MicroBlaze provides a general-purpose scalar host processor and a streaming unit, the SPoC pipeline, made of several image processing vector units, is the image processing accelerator. The SoC also contains a DDR3 memory controller, DMA engines, FIFOs to synchronize memory transfers and vector computations with the host, a gigabit Ethernet interface, and video converters for I/Os.

One vector unit of the SPoC pipeline (Figure 10, right top) has two input and two output 16-bit-per-pixel images. Typically 8 such units are chained. The first inputs and last outputs are connected to the external memory by DMA engines. Within the pipeline, pixels move from operator to operator at each clock tick, with operators doing either their work or a no-op. A vector unit is made of several operators, with a constrained interconnection: the datapaths are controlled by multiplexers M_x . The MORPH box can apply simple 3×3 stencil operations, such as erosions, dilatations, and basic convolutions. Because sliding windows are used to hold the two rows and a few pixels that provide the necessary neighborhood needed by the stencil operations (Figure 10, right bottom), a significant latency is induced at this stage. The stencil operations are also adapted automatically at the image borders and corners. Then the results can be combined by an Arithmetic and Logic Unit ALU, which performs unary or binary pixel operations such as *adding* or *subtracting* pixels. Two outputs are selected among those three results by the four multiplexers that control the stream of images. A threshold operator THR can be applied to each selected output and the reduction engine MES computes reductions such as max or sum of passing pixels.

The host processor controls the vector units by sending them one micro-instruction each, and by instructing the DMA engines to queue and dequeue pixels. The host processor can also retrieve the reduction results from the vector units once the images have been processed. The control overhead remains small because images are always large enough to generate very long pixel vectors. A low-resolution image, for instance 320×240 , is equivalent to a 76,800-elements vector. To sum up, each micro-instruction can concurrently perform up to 5 full image operations and a number of reductions, equivalent to 29 elementary pixel operations per tick.

Up to 32 chained vector units can be crammed into our FPGA chip. Our reference target hardware includes 8 vector processing units, but the solution we suggest shortly is parametric with respect to this number. In practice, the vector of depth 8 provides a good cost-performance trade-off as it fits patterns of iterated convolutions, erosions, or dilations that are often found in typical applications, but is yet not too expensive when these patterns are not found. With a specific set of applications in mind, several vector depths can be tested to choose the best setting.

The chaining of operations is very much constrained, but the micro-instruction detailed scheduling and compaction are easy once the order of operations is chosen. Three hardware constraints are mostly relevant for the compiler. (1) The structure of the micro-instruction set and of the vector unit datapaths must be matched by the application DAG. (2) The maximal number of vector units provides a cost-performance trade-off. (3) The number of image paths, here two, restrains the number of images sent and received by the accelerator. With 8 vector units containing ideally compacted operations, up to 40 full image operations can be performed for two loads and two stores, which leads to 10 image operations per memory access, including high-level convolutions which require about 10 elementary operations each, and not counting the many reductions. A SPoC pipeline stage can execute more than 60 elementary pixel operations per memory access.

Phase 3.1 – SPoC Hardware Configuration. The SPoC hardware accelerator constraints discussed before must be met: the computations in the hardware accelerator must only involve two live images at any single point because only two datapaths are available. Actual computations must be scheduled on components so that live images can still reach their use or the end of the path. When all available vector units in the SPoC pipeline are used and more operations remain, pipeline spilling must be managed. The optimality criterion is to minimize the number of calls to the accelerator, as one call takes about the same time whatever the operations performed in the pipeline.

The problem of mapping an image expression DAG onto the SPoC accelerator is very close to the pebble game problems used in register allocation [Cook and Sethi 1976], with, in our case, only two registers. However, unlike register allocation problems, our spill code interrupts our computation pipeline, resulting in *both* registers to be spilled at the same cost as one, as they occur simultaneously. So, although mapping scalar expression DAG onto a register machine [Bruno and Sethi 1976; Aho et al. 1977] is NP-complete, this result does not apply to SPoC. We conjecture nevertheless that our problem is NP-complete, because of the close similarity with the code generation problem for register machines. Firstly, the setting is highly combinatorial if one enumerates all possible evaluation orders compatible with the dependencies when there is a high degree of parallelism available in the DAG. Secondly, evaluating the cost of a proposed solution is reasonably easy: given a schedule of operations, one can detect in one pass over the vertices when an infinite pipeline should be cut because an operation would create more than two live images; if a finite number of vector units is considered, instruction compaction can tell when the pipeline is full.

Given the combinatorial nature of the problem, our heuristic consists in breaking down the problem into three successive stages. Each stage satisfies one of the constraints independently, and there is no guarantee of global optimality. First, we meet the two-live-image constraint with a decomposition of the expression DAG into sub-DAGs. The sub-DAG operations are ordered by this decomposition process so that their evaluation in *that* order only requires two live images. Then, instructions are compacted in a conceptually infinite pipeline, which is finally cut according to the number of avail-

Fig. 11. SPoC operation schedule for the running example (Figure 3 and Figure 7).

Fig. 12. Running example (Figure 3) DAG mapping on SPoC.

able vector units. We chose not to perform a global combinatorial optimization because our simple heuristic leads to optimal results in most cases; see Section 8.

The optimized expression DAG is first split into sub-DAGs with no more than two live images and no internal scalar-carried dependencies. This would require that the previous operation is completed before starting the next, so they cannot fit in a pipeline. As noted above, this is very similar to evaluating an expression with only two registers. We use the simple *list scheduling of basic blocks* technique described in the *Dragon book* [Aho et al. 2006], with a prioritized topological order that focuses on the critical resource, namely the small number of datapaths. The greedy list scheduling heuristic expands a subgraph as much as possible, and never backtracks. The priority choices favor the immediate use of computed images in the pipeline: reductions that do not update their source are performed first, then operations that use up an image and define another one, ordered by the number of uses, then other operations. The result of the first pass is a list of scheduled DAGs that do not require more than two intermediate images to be evaluated. The resulting one sub-DAG schedule found for our running example is presented in Figure 11. At any point along the graph, only two images are alive, thus the computation can be performed by the SPoC pipeline.

Each sub-DAG is then mapped onto a pipeline with a conceptually infinite number of vector units by compacting operations. The previous schedule is kept unchanged at this stage as modifying it would put at risk the two-live-image constraint. Micro-instruction compaction is performed at the same time because the packing constraints are easy to meet. Structural, control, and data pipeline hazards [Hennessy and Patterson 2003] are avoided by the hardware, hence sophisticated micro-instruction scheduling and compaction are not needed. The compaction is achieved simply by scheduling operations in the first available slot starting from the beginning of the pipe with a greedy heuristic. When only one input image is needed by the computation, it is sent on both input paths so as to favor compaction at the beginning of the pipe. Path selection contrives multiplexer configurations: computed images must reach the operators that process them, shifting a computation further down the pipeline in some cases.

The third stage of the code generation process is to map the open-ended pipeline onto the N available vector units. This is achieved by cutting the previous sequence into segments of length N and calling the SPoC pipeline for each segment.

The mapping of the optimized DAG of our running example onto SPoC results in two sub-DAGs shown in Figure 12. Dashed boxes show intra-pipeline vector-unit boundaries. One helper code encompasses one call to the accelerator. In the first helper, the

Fig. 13. Terapix accelerator architecture.

sequence of dilatations is delayed by one slot to allow the computation of the *vol* and *min* reductions at the start of the pipeline. The first call is filled as much as possible because of the greedy heuristics. Image variables are reused to transfer the intermediate images between the two accelerator calls.

This heuristic phase for the SPoC accelerator reuses standard compilation techniques to generate optimal results most of the time. It is followed by a quick cleanup of intermediate images which are no longer used by the function. The techniques are applied on very long vector flows of pixels from images, whereas they were originally designed for scalars in registers. This works well because the SPoC architecture takes care of pipeline hazards and performs stencil computations without reducing the image size: images are equivalent to scalar variables.

5. TERAPIX ARCHITECTURE, RUNTIME, AND BACK-END

This section presents Terapix, our second hardware target. Then it describes the algorithm to schedule the optimized DAG onto this SIMD array and its runtime.

Our second target, the Terapix accelerator [Bonnot et al. 2008], is developed by THALES as a SoC for image processing embedded systems, such as security surveillance cameras. Its structure is outlined in Figure 13. It is a 128-processing element (PE) SIMD array with a limited local memory (local RAM in the figure) of 1024 pixels per PE, and a small shared memory for parameters (global RAM). Each PE accesses its own memory and can send or receive values from its two neighbors. The host micro-processor loads into the accelerator a set of microcodes (μ -codes) for a number of different computations, usually only once for a given application. These microcodes have originally been developed manually, one for each image operator, designed to be run on a 128-pixel wide tile of varying height. The host (μP) can also load constants needed by these microcodes into the shared memory. The runtime controls two queues, one for memory transfers (Data Moving Unit) and one for computations (sequencer and addressing), so that communications and computations can be overlapped: both queues run in parallel, and can be synchronized so that a computation waits for its input data or a transfer waits for the availability of the result. The instruction queue specifies the function to run, as well as a set of other parameters to locate input and output tiles, provide the tile height, or the location of other parameters in the shared memory.

Full images cannot be processed because of the small internal memory size. The runtime tiles the input image and schedules the tile executions. When stencils are computed on a tile, the area of valid results shrinks as tile boundaries hold bad values because of missing pixels, so overlapping tiles must be sent to the accelerator. However, the architecture and functions ensure that computed values for the actual border of the

full image (not inter-tile) are correct, based on the image operator semantics. The tiling is performed automatically by the runtime based on image sizes and declared overlaps, so that operations are performed on whole images seamlessly. The task of the compiler is to generate the schedule of function calls and memory transfers for both flip and flop computations and to allocate double buffers and intermediate tiles explicitly. The runtime takes this schedule and fills in the instruction and transfer queues to apply the overlapping tiling on the image and to perform the whole computation.

The host/accelerator can send or receive about 5 pixels per tick. This induces a speedup ceiling for the accelerator when stencil operations are used. For a typical stencil computation, one tile is sent and one is received, which cost each 24 cycles per 128-pixel rows, and the computation itself needs about 15 cycles per row. Several stencil operations can be combined, but it is not interesting to do so once the computations amortize communications, as the area of valid results within a tile shrinks with depth. The balance occurs when $48/15 = 3.2$ such operations are put together, fused so the practical maximum speedup that can be expected *when* stencil operations are used is about 3, due to the intrinsic communication-to-computation balance of the accelerator. If other kinds of operations, such as point-to-point computations, dominate an application and can be combined, the speedup is larger. However, this is seldom the case in practice (see Terapix performances in Section 8).

Phase 3.2 – Terapix Hardware Configuration. The code generation for the Terapix hardware must perform two important tasks: (1) balance computations and communications so that the hardware is efficiently used, taking into account the shrinking of the valid area due to stencil operations, and (2) allocate the little available memory to tiles needed for a computation, and double buffers necessary for communication and computation overlap.

The first aim is reached by splitting DAGs in *vertical* slices (with inputs on the left and outputs on the right), so that valid results within a tile are shrunked by the same amount in all computed images. The number of strips is computed based on the memory access bandwidth, so that communication and computation costs are balanced. We have

$$\#cuts = \left\lfloor \frac{\left(\frac{\text{comp}}{\text{comm}} \right) - \#ins - \#outs}{2 \times \text{width}} \right\rfloor,$$

where the comp/comm ratio is the number of images that can be transmitted during the computation of the DAG. The input and output images must be transmitted, plus the intermediate images if the DAG is cut. Their number is about the width of the DAG. The rationale for the factor 2 is that if the DAG computation is interrupted, the intermediate results must be both extracted and resent to the accelerator. Once the number of DAG cuts is decided, the corresponding erosion factor (number of lost pixels on tile borders) is computed, and the DAG is split at these depths. Although more accurate formulas could be thought of, this simple approach gives good results on our test cases. For small DAGs, splitting does not bring any benefit.

The mapping of the running example on Terapix is depicted in Figure 14, with 4 accelerator calls. Unlike the mapping for SPoC, the sub-DAGs for Terapix are balanced so that the border erosion is similar between accelerator calls, and they are reduced to connected components to lower the memory requirements of each call.

The second task is to schedule operations and allocate memory on the Terapix. This is done through a list-directed scheduling of operations in each split DAG, similar to the one described earlier for SPoC, but with different priorities applied. The overall design of the priorities is to favor the use of data so as to make tile memory slots available for later computations. The chosen schedule induces a total number of tiles

Fig. 14. Running example (Figure 3) DAG mapping on Terapix.

Fig. 15. Terapix schedule and tile allocation for running example helper 3 (Figure 14).

to perform all computations: first, data input and output tiles must be allocated twice for double buffering; then some additional tiles may be required if some operations do not free their operands, or because of operations such as convolutions that cannot be performed in place. The schedule is also used to generate the computation code, which consists of calling operator functions with parameters that designate the input and output tile operands. Two versions are generated for each set of double buffers: one for the *flip* and one for the *flop*.

The scheduling attempts to put the overall computation result tiles directly in the relevant double buffers; if it does not succeed, an additional copy operation is inserted at the end of the computations. The generated code also includes the input and output communications needed to feed the pipelined computations, and tile border erosion declarations that enable the runtime to perform the appropriate overlapping tiling.

The resulting schedule and allocation for the fourth helper of our running example is outlined in Figure 15. The two input tiles on the left (numbers 1 and 2 above arrows) are filled in with a tile of the input images. Then the computations start using a third tile (number 3) which is necessary as the convolutions cannot be executed in place. Following the schedule, each computation uses the tile just freed by the previous computation to store its own result. At the end of the computation, outputs are available in tiles 1 and 2, which are the same as the initial input tiles. Thus three tiles are needed by this schedule for one computation. Then 2 double buffers, twins of tiles 1 and 2, must be allocated to be filled in and emptied out in parallel to the ongoing computation: 5 tiles are needed for the whole flip-flop.

Once the number of tiles for a schedule is known, the maximum tile height can be computed by dividing the available memory by the number of vertical tiles. The tile width is always 128, the number of processing elements of the accelerator. The computed maximum height is not necessarily the optimal size. Indeed, it is the best possible for the steady state, but as the number of tiles for an image is usually quite small, typically 6 to 10, and as the operator functions perform their computations on whole tiles whatever the live part in them, useless computations are performed on the remainder of the last tile. This is illustrated in Figure 16 for a hypothetical 6×8 image: on the left, the size-4 tile is called twice over the image and computes 2 useless rows, while on the right the size-3 tile is called twice and does not include

Fig. 16. Optimal tile size computation for Terapix.

redundant computations. This can change the optimal size significantly. The generated code computes at runtime the best possible tile size based on the actual image size and the amount of necessary overlap, so as to minimize the amount of redundant computations due to these effects. If the image size is available at compile time, the actual best tile size is directly computed and used in the generated code.

6. OPENCL TARGET, RUNTIME, AND BACK-END

This section presents our last target, which is any device with an OpenCL compiler and runtime. We split the optimized DAG into a set of OpenCL kernels.

The third target is OpenCL (Open Computing Language [Khronos Group 2008]), so that applications can be run on GPGPU and multicores which support this environment. OpenCL is an open standard for parallel programming of heterogeneous platforms proposed by the Kronos group, where vendors such as AMD, Apple, IBM, Intel, and Nvidia collaborate. It is especially well suited to GPGPU (General-Purpose Graphical Processing Unit) hardware, with a data parallel multithreaded programming model that executes simple kernels on data sent to an external device. Dependencies can be declared between computations to expose task parallelism as well. Hardware portability is obtained by compiling and optimizing kernels on-the-fly for the particular hardware at hand. The actual performance obtained with such codes depends heavily on the compiler implementation, runtime characteristics, and hardware details of the target.

The basic OpenCL implementation of the FREIA library manages the device memory allocation for each image and issues the necessary host-to-device or device-to-host communications when data is needed. It keeps track of the availability of each image so as to minimize these communications. The memory allocator also checks dynamically for data that may be overwritten by operations while still needed, and allocates additional buffers to avoid these interactions. A key factor in getting good accelerations with OpenCL is to improve data locality by combining operations so that the available internal memory bandwidth allows to fully exploit the available processors.

Phase 3.3 – OpenCL Code Generation. The FREIA OpenCL runtime provides a computation kernel for each image operator. For stencil operations, the amount of computation per pixel loaded is enough to use efficiently the device internal bandwidth on typical GPGPU devices. For instance, the Nvidia GeForce GT 640, launched in April 2012, can perform about 2 integer operations per byte transfered, that is about 4 operations per 2-byte pixels: the nine integer arithmetic operations of an erosion are about enough to cover the corresponding pixel load and store, once the steady state is reached. However, for simple pixel arithmetic operations, memory transfers are not amortized and processor-memory communication times dominate the overall performance.

Our compilation strategy is to aggregate image operations into specialized kernels that group together arithmetic operators and possibly one level of 3×3 stencil on input images. Only connected components are put together, so that unrelated computations

Fig. 17. Running example (Figure 3) operator aggregation for OpenCL.

do not become entangled. We also generate specialized kernels for lone convolutions that directly embed stencil coefficients.

Figure 17 shows aggregations performed on our running example for OpenCL: two reductions are merged with two stencil operations on the same input; the final subtraction is merged with the preceding stencil computations; two specialized versions of the remaining erosion and dilatation stencil operations are generated.

7. IMPLEMENTATION

This section presents and illustrates the implementation of the compilation and code generation algorithms described in the previous four sections, with an emphasis on the relative low software-engineering effort involved.

Our optimization strategy is implemented in PIPS [CRI, MINES ParisTech 2012], a source-to-source research compiler, for a limited development effort, measured hereafter in KLOCs (*thousand lines of code*). Transformations of phase 1 are more or less standard in an advanced optimizing compiler. They amount to 155 already existing KLOCs in PIPS, including the parser, semantical analyses, and optimizations phases. Phase 2 operations are also standard, but are used here for full image processing calls although the usual scope is on elementary scalar processor operations. Its implementation requires 4.1 KLOCs to represent the FREIA elementary operator semantics, plus building and optimizing the DAG representation and the generated code. Each basic function is described by its expected inputs and outputs, its parameters, whether it is commutative or not, and how it can be mapped onto the different targets by specifying the component and configuration for SPoC, the microcode to call for Terapix and whether it can work in place, or the underlying elementary pixel operator for OpenCL. Algebraic transformations on the DAG simplify some expressions, especially by detecting and propagating constant images which result in image operator strength reduction. Phase 3 is the back-end specific to each target. It uses 1.9 KLOCs for SPoC, 1.4 KLOCs for Terapix, and 1.1 KLOCs for OpenCL. The DAG splitting, scheduling, and wiring decisions implemented for SPoC are quite complex, as they have to deal with the structure and heterogeneity of the vector unit. The DAG splitting implementation for OpenCL and Terapix is simpler and mostly shared between the two back-ends, with parametric functions to allow specific behaviors. Dumps of the intermediate DAGs in dot format help visualize, understand, and possibly debug the compiler behavior. The implementation for phases 2 and 3 added about 5% to the compiler code base. As the generated code is in C, it is simple to generate, check, and debug, thanks to the examples in the optimized library implementation. It produces accelerated functions that are called from the initial application. Each generated configuration function (excerpts in Figure 18 for SPoC, Figure 19 for Terapix, and Figure 20 for OpenCL) is called from the `main` with the appropriate arguments (e.g., in Figure 21 for generated SPoC helpers). A standard C compiler then generates the executable for the host target.


```

void running_spoc_helper_0(f_data2d *o0, f_data2d *o1, f_data2d *i0,
    int32_t *r0, int32_t *r1, int32_t *k2 /* ... up to 16 */) {
    // INIT SKIPPED si op sp par red*: micro instructions, operation parameters, reductions
    si.mux[0][0].op = SPOC_MUX_INO; // set state of MUX stage 0 number 0
    si.poc[1][0].op = SPOC_POC_DILATE; // set state of POC stage 1 side 0
    si.poc[1][0].grid = SPOC_POC_8_CONNEX;
    // SKIPPED: more configurations...
    f_cg_process_2i_2o(op, par, o0, o1, i0, i0); // call hardware accelerator, 2 ins & 2 outs
    f_cg_read_reduction_results(&redres); // extract reduction results
    *r0 = (int32_t)reduc.measure[0][0].minimum;
    *r1 = (int32_t)reduc.measure[0][0].volume;
}

```

Fig. 18. SPoC helper code excerpt for the running example (Figure 3).

```

void running_tpx_helper_1(f_data2d *o0, f_data2d *o1,
    f_data2d *i0, f_data2d *i1, int32_t *pi4) {
    // SKIPPED: declarations & initializations
    int tile_1 = 0, tile_2 = 204, ...; // ... up to 5
    int tile_io_1_0 = tile_1; // double buffers
    int tile_io_1_1 = tile_4; // ... 2 more
    int vertical_border = 5, tile_computed_size = 194;
    int tile_size = ... ; // compute best tile size depending on input image
    mcu_macro[0][0].xmin1 = tile_io_1_0; // E8(1) -> 3
    mcu_macro[0][0].xmin2 = tile_3; // more settings skipped
    mcu_macro[0][0].iter2 = tile_size;
    mcu_macro[0][0].addrStart = UCODE_ERODE_3_3; // microcode for operation
    for(i=0; i<9; i++) p_0[i] = pi4[i]; // copy needed kernel
    // SKIPPED other operations definitions, including flip-flop
    mcu_instr.borderN = 5; // AND also S W E tile erosion
    mcu_instr.imagelet_height = tile_size;
    mcu_instr.nbin = 2; // I/O tile declarations...
    tile_in[0][0].x = tile_io_1_0; // more settings skipped...
    freia_cg_process(&param, o0, o1, i0, i1); // call accelerator
}

```

Fig. 19. Terapix helper code excerpt for the running example (Figure 3).

8. EXPERIMENTS

Thanks to the prototype compiler described in the previous section, we were able to perform detailed experiments on various hardware for all three back-ends. This section describes these experiments and analyzes the results.

Our prototype FREIA compiler has been tested on 1295 cases: 1005 combinatorial tests (3 to 6 ops), 172 elementary tests (1 to 13 ops), 50 atomic tests (1 op for which we generate the hardware accelerated versions) and finally 68 significant applications or functional blocks (5 to 422 ops) configured with various parameters. The overall compilation time, with all preprocessing and code generation, is reasonable, especially as not much time has been spent optimizing the implementation: most cases are processed in a few seconds, although Burner, our largest 422 ops application, requires about a minute. For the SPoC accelerator, only 30 cases out of 1295 tests are not optimally compiled by our approach: 28 are one call away from optimality; 2 are not optimal by

```

KERNEL void oop_opencl_helper_0(PIXEL * o0, PIXEL * o1, // 2 output images
    PIXEL * i0, PIXEL * i1, PIXEL * i2, // 3 input images
    int width, int height, int pitch) { // size parameters...
    int Wtile = (width+get_global_size(1)-1)/get_global_size(1);
    int Wlast = Wtile*(get_global_id(1)+1);
    int is_N = (get_global_id(0)==0), is_S = ... ; // detect N & S boundaries
    // SKIPPED: mome initializations...
    PIXEL * p0 = o0 + shift, * j0 = i0 + shift; // AND p1, j1 & j2
    for (int i=Wtile*get_global_id(1); i<Wlast; i++) { // width loop
        int is_W = (i==0), is_E = (i==(width-1)); // detect E & W boundaries
        PIXEL t220 = PIXEL_MIN;
        PIXEL in2NW = (is_N|is_W)? 0: j2[i-pitch-1];
        // SKIPPED: preloads for stencils on in0*/in1*/in2* from j0/j1/j2
        t220 = PIXEL_SUP(t220, (is_N|is_W)? PIXEL_MIN: in2NW);
        // SKIPPED: many more stencil computations on temporaries 220/1 197/8 203/4
        PIXEL t199 = PIXEL_SUB(t197, t198); // arithmetic operations
        PIXEL t205 = PIXEL_SUB(t203, t204);
        PIXEL t222 = PIXEL_SUB(t220, t221);
        PIXEL t208 = PIXEL_ABSDIFF(t199, t205);
        PIXEL t225 = PIXEL_ABSDIFF(t205, t222);
        p0[i] = t208, p1[i] = t225; // store result
    } }

```

Fig. 20. OpenCL helper code excerpt for OOP (Figure 9), parameterized by the 8- or 16-bit PIXEL type.

```

running_spoc_helper_0(od, og, in, &min, &vol, k8c, /* 18 * k8c */);
running_spoc_helper_1(od, og, od, og, k8c, k8c, k8c, k8c);

```

Fig. 21. SPoC helper (Figure 18) stub calls.

2 and 3 calls. Most of these nonoptimal cases are linked to the greedy nature of our heuristic coupled with pipeline spilling effects. Optimality for Terapix and OpenCL is simple to define: computations take a minimal time to execute with respect to *all* possible transformations and code generation techniques. However, it is less obvious to check the potential optimality of the results found by our compilation strategy.

Figure 22 lists our test applications and shows results obtained with our compiler with respect to the basic library implementation. The first column is the application name, and the next three columns show the number of DAGs built and the static number of image operations before (ini) and after (opt) optimizations. Then for each target platform, we show the number of accelerator calls performed by each application in the library and compiled versions, and compute the resulting *theoretical* acceleration ratio based on these counts. For application VS, the numbers shown are in thousands of calls. The numbers of runtime calls for both SPoC and OpenCL are exactly the same for the basic library version, and may slightly differ for Terapix because the *copy* operation is not implemented with the accelerator. The overall geometrical mean of the count ratios is computed on the last row for each platform, to sum up the overall *theoretical* benefit of our optimized version. The *optimized* library figures are not shown because it was only partially implemented for the different targets. If fully available, for the kind of statically parameterized applications of our domain, it should provide intermediate performances between the basic library and the compiled versions. Indeed, the optimized library can combine some calls, so it is better than the basic library. The combined calls could be at best the same as the

App.	dags & ops			SPoC			Terapix			OpenCL		
	#	ini	opt	lib	acc	lib acc	lib	acc	lib acc	lib	acc	lib acc
anr999	1	33	23	33	2	16.5	33	4	8.3	33	18	1.8
antibio	7	49	48	832	90	9.2	830	216	3.8	832	293	2.8
burner	4	422	422	518	37	14.0	517	82	6.3	518	437	1.2
deblocking	1	36	32	36	17	2.1	34	2	17.0	36	9	4.0
LP	1	135	69	135	8	16.9	69	10	6.9	135	63	2.1
OOP	1	22	18	4290	1365	3.1	3900	195	20.0	4290	975	4.4
retina	3	106	42	304	16	19.0	302	73	4.1	304	98	3.1
toggle	1	15	15	15	7	2.1	15	1	15.0	15	4	3.8
VS	6	28	26	201	100	2.0	201	100	2.0	201	100	2.0
<i>Overall</i>	—	—	—	—	—	6.5	—	—	7.3	—	—	2.6

Fig. 22. Accelerator call counts for SPoC, Terapix, and OpenCL targets on our test applications.

App.	CPU on SoC	SPoC accelerator						Terapix accelerator					
		lib	acc	Speedups			ms	lib	acc	Speedups			ms
		ms	ms	cpu lib	cpu acc	lib acc		ms	ms	cpu lib	cpu acc	lib acc	
anr999	455.5	15.2	0.9	29.9	493.5	16.5		8.3	3.5	55.2	129.3	2.3	
antibio	5117.5	386.9	41.9	13.2	122.3	9.2		198.1	88.5	25.8	57.8	2.2	
burner	6764.2	240.9	17.2	28.1	393.1	14.0		177.7	83.8	38.1	80.8	2.1	
deblocking	681.6	64.9	30.7	10.5	22.2	2.1		30.9	11.0	22.1	61.7	2.8	
LP	3743.8	229.9	13.6	16.3	274.8	16.9		67.2	36.8	55.7	101.7	1.8	
OOP	7514.2	391.7	124.6	19.2	60.3	3.1		213.4	63.3	35.2	118.7	3.4	
retina	2453.7	141.4	7.4	17.4	329.8	19.0		81.5	32.4	30.1	75.7	2.5	
toggle	417.7	27.0	12.6	15.4	33.1	2.1		14.0	4.3	29.8	97.8	3.3	
VS	451.6	192.9	95.6	2.3	4.7	2.0		1470.8	1203.9	0.3	0.4	1.2	
<i>Overall</i>	—	—	—	14.2	91.5	6.5		—	—	20.5	47.6	2.3	

Fig. 23. Application performance on the SPoC (pipeline depth 8) and Terapix embedded systems.

compiled version which is not constrained by any functional boundaries. Moreover, optimizations that take advantage of the overall graph, such as suppressing dead code or factorizing out common subexpressions, cannot be replicated simply by a library. For instance, *anr999* with an optimized library for SPoC results in 6 accelerator calls, and about 10–12 calls are necessary for Terapix, to be compared to the respective 2 and 4 calls of the compiled versions and the 33 calls of the basic library version.

Figure 23 presents actual execution times for the test applications on our target embedded system with the SPoC and Terapix accelerators. These times are derived from functional simulators for both platforms. They only take into account the computation part of the applications, ignoring I/O and other overheads. The *cpu* time is an evaluation of the time needed by the host processor on the SoC to compute all the image operations without delegating any work to the attached accelerator. It is the same for both accelerators as they use the soft core. The *lib* execution times are obtained with the basic library version, which calls the accelerators one operation at a time. The *acc* execution times are related to our compiled version. It is always significantly better than the *lib* version on both targets.

With the basic library version, the accelerators provide on average a significant cpu-to-library speedup, with a 44% advantage to Terapix. With the compiled version, the accelerator comparison is reversed: it fares on average about twice better on SPoC

App.	OpenCL Intel on dual-core						OpenCL AMD on quad-core					
	cpu <i>ms</i>	lib <i>ms</i>	acc <i>ms</i>	Speedups			cpu <i>ms</i>	lib <i>ms</i>	acc <i>ms</i>	Speedups		
				cpu lib	cpu acc	lib acc				cpu lib	cpu acc	lib acc
anr999	9.1	24.9	12.2	0.4	0.7	2.0	7.8	20.8	7.2	0.4	1.1	2.9
antibio	119.7	297.5	254.1	0.4	0.5	1.2	142.7	268.4	135.3	0.5	1.1	2.0
burner	1329.5	267.9	162.2	5.0	8.2	1.7	1619.1	204.5	124.4	7.9	13.0	1.6
deblocking	231.3	217.5	25.1	1.1	9.2	8.7	268.6	242.8	16.7	1.1	16.0	14.5
LP	869.5	137.9	32.2	6.3	27.0	4.3	1027.1	108.6	21.9	9.5	46.8	4.9
OOP	6.0	20.8	12.3	0.3	0.5	1.7	6.3	15.8	8.3	0.4	0.8	1.9
retina	244.0	136.5	93.5	1.8	2.6	1.5	281.7	133.4	55.4	2.1	5.1	2.4
toggle	11.0	21.9	15.0	0.5	0.7	1.5	11.0	17.3	9.0	0.6	1.2	1.9
VS	10.6	29.4	25.9	0.4	0.4	1.1	11.7	10.6	8.1	1.1	1.5	1.3
<i>Overall</i>	–	–	–	0.9	1.9	2.0	–	–	–	1.3	3.5	2.7

Fig. 24. OpenCL performance on CPU – VS times is in seconds.

than on Terapix. The library-to-accelerator speedups obtained with SPoC are precisely those announced from call counts in Figure 22 because, as stated earlier, calling this accelerator for a given image size costs the same whatever the computations performed. The 2.3 average speedup obtained on the applications with Terapix is smaller than the 3.2 speedup ceiling computed in Section 5 because stencil operations dominate the applications. It is much smaller than the accelerator call count ratio in Figure 22. The VS (VideoSurvey) application performs quite poorly, especially on the Terapix where we have a slowdown. The first phase of the application performs many computations on very small subimages to detect and cancel camera jitter, so that stabilized images can be compared. These little computations are slower on Terapix because the width of the subimage is much smaller than the number of processing elements, which thus are mostly unused. The largest speedups on SPoC are obtained on applications (*retina*, *burner*, *LP*, *anr999*, *antibio*) with very long chains of stencil operations which are especially well suited to the hardware pipeline.

Figure 24 presents measurements obtained on multicores with Intel (SDK OCL 1.1 13786) and AMD (FGLRX 8.960) OpenCL implementations. The execution times only include image computations, ignoring I/Os, kernel compilations, and other overheads. These figures must be interpreted with care. Indeed, the *cpu* time reflects the single-thread Fulgoro software implementation with SSE2 vectorization, which cannot be directly compared to *lib* and *acc* which rely on the OpenCL environment. The overall *cpu*-to-compiled speedup (second speedup columns) for both targets is close to the number of cores, but this does not seem significant. Applications that perform few computations per possibly small (sub-)images (*anr999*, *antibio*, *OOP*, *toggle*, *VS*) show the lowest performance, with slowdowns on the dual-core and just a little better on the quad-core: kernel launches are not really amortized on these. Our actual contribution is on the third *lib*-to-compiled speedup column, 2.0 and 2.7 for the dual- and quad-core.

Figure 25 reports performance on actual Nvidia GPUs with driver version 304.37. The *acc* time is the best performance obtained by varying the number and shape of threads over the image, and by testing loops and thread dimension exchanges in the kernels. The *lib* time is the basic library version. Contrary to the compiled version performance, and due to limitations in the current version of the runtime, the image to thread mapping strategy is fixed for the library version: there is always one thread for each image row. Yet again, these figures must be interpreted with care. Over all applications, the more recent or larger the GPU, the better the performance, with some

App.	GeForce 8800 GTX			Quadro 600			Tesla C 2050		
	lib	acc	$\frac{\text{lib}}{\text{acc}}$	lib	acc	$\frac{\text{lib}}{\text{acc}}$	lib	acc	$\frac{\text{lib}}{\text{acc}}$
	<i>ms</i>	<i>ms</i>		<i>ms</i>	<i>ms</i>		<i>ms</i>	<i>ms</i>	
anr999	119.9	9.8	12.2	103.4	1.5	68.8	17.0	0.9	18.3
antibio	1243.1	204.3	6.1	1332.5	57.4	23.2	306.6	93.4	3.3
burner	657.5	321.0	2.0	1296.7	576.2	2.3	280.7	142.0	2.0
deblocking	363.6	11.1	32.7	324.1	3.5	93.9	240.7	1.3	186.1
LP	436.4	36.6	11.9	854.8	7.3	116.4	145.9	2.3	62.3
OOP	80.6	5.8	13.9	73.4	1.8	40.0	12.5	1.0	12.8
retina	492.9	75.5	6.5	563.8	60.8	9.3	123.5	33.9	3.6
toggle	94.0	6.3	14.8	101.9	1.4	71.9	15.6	0.7	21.4
VS	30.8	24.1	1.3	43.9	36.4	1.2	67.3	77.4	0.9
<i>Overall</i>	–	–	7.8	–	–	22.1	–	–	10.2

Fig. 25. OpenCL performance on three Nvidia GPU – VS time is in seconds.

exceptions such as *VS*, which is faster on the old GeForce, and *antibio*, which runs better on the Quadro. However the times are disappointing when compared to typical OpenCL CPU performance in Figure 24. The Tesla runs only 1.5 to 15 times faster than the quad-core but for *VS*, where it is about 10 times *slower*. Our compilation contribution is illustrated by the speedups over the basic library version. They are excellent, especially on the Quadro. However, if we had restricted ourselves to keeping the same threading strategy as the basic library version, the overall speedup would be significantly lower, and closer to the theoretical call-count-based acceleration ratio in Figure 22. There is one slowdown for the *VS* application on the Tesla, which seems due to its particular image operation pattern discussed above.

The OpenCL times for both CPU and GPU are very sensitive to details in the kernel and runtime. For instance, if pixels are preloaded in the kernel code, the performance is significantly lower on CPU, but it improves a bit on GPU. Moreover, it is better that threads scan image rows to help CPU cache effects, but it is more or less the opposite on GPU, where performance is degraded in this case. Providing implementation variants to address these conflicting requirements is quite easy when generating code for a compiled version, but would be much more difficult when implementing the library, which is expected to run on different targets. The OpenCL runtime does a good job of using event-driven asynchronous kernel calls over image operations. If there are no dependencies between jobs, there is a good chance that they can run in parallel on high-end hardware such as the Tesla. However, this ability cannot be used simply when scalars are produced by reductions. In such cases, the kernel calls are in practice fully synchronous as they wait for the reduction result to return. The performance impact is obvious on Application *VS* where such synchronous kernel calls on very small subimages dominate. Another runtime issue is that the best thread setting is not the same from application to application and from chip to chip. For instance, the best per-thread tile size found for *anr999* is 8×3 on the GeForce, 1×12 on the Quadro, and 11×4 on the Tesla. A bad choice for thread number or shape on some GPUs increases the execution times by several orders of magnitude.

9. RELATED WORK

This section discusses the related work about improving both software engineering and hardware performance handling of domain-specific applications. Preliminary work about the SPoC target was presented at a workshop [Coelho and Irigoien 2011].

The related work is rather limited because researchers looking at hardware accelerator issues in an academic environment usually do not have the resources required to develop a full programming tool chain. They either design a specialized language, use pragmas to guide the optimizer, build an optimized library, which may grow with each new application to include the application-specific API that leads to good performance, or develop applications with target-knowledgeable people. We break the related work in two parts, software development for accelerators and optimization of expression evaluation.

Software Development for Accelerators. Specialized languages have been designed to address various needs of application domains and target architectures that are not well served by general-purpose languages. The OpenCL [Khronos Group 2008] recent standard aims at providing portability across accelerators, especially GPGPU targets, but it is quite extended and rather low level: application developers should be able to ignore it [Wolfe 1995]. It remains an interesting output language for a source-to-source tool like PIPS, or for implementing an efficient runtime to be called by the generated code. Array-OL [Glittia et al. 2009] is a domain-specific language designed for signal processing and for accelerator programming. However, Array-OL is not general enough to write a whole application, and is still hard to compile efficiently for a given target: parts of the application must be isolated and coded in Array-OL, and the Array-OL optimization process be performed under human supervision using a graphic tool. Specialized image processing languages include Halide [Ragan-Kelley et al. 2012] and Diderot [Chiw et al. 2012].

Pragma annotations added to standard languages are used to preserve the portability of applications and allow their functional validation in a development environment. OpenMP [OpenMP Architecture Review Board 2008] directives are hints about the program semantics, say loop parallelism or critical sections, but does not yet address all the requirements of hardware accelerators, especially when the hardware accelerator must be programmed. HMPP [CAPS enterprise 2008] is another pragma set designed by CAPS Enterprise to provide higher-level pragmas. It can be used to program an accelerator such as Nvidia Tesla or AMD FireStream, including the use of several accelerators linked to a unique host, an issue not addressed by our technique. However the set of directives is very specific and requires deep architectural insight from the developer to be exploited fruitfully.

Another way of achieving high performance on specialized hardware and still retain portability is to use domain-specific libraries: VSIPL [VSIPL Forum 2008] in the signal processing field was developed as an open standard by an industry, government, and research consortium. It contains thousands of functions, and various level of partial implementations are defined in the standard, starting from the 127-function core lite profile, followed by the 513-function core profile, but implementations do not necessarily implement these profiles in full. As the functions are not independent and orthogonal, the developer must choose an implementation strategy which may result in different performances with differing library implementations, and may impair the portability when all functions are not available. Moreover, we observed in the Ter@ops project [THALES (TRT) and other institutions 2009] that an API has a direct impact on the application structure, which may lead to poor performance on a new piece of hardware. A library was designed for vector-based instruction set additions such as AltiVec or Intel SSE extension family [Ramanathan et al. 2006; Freescale 1999]. To optimize its functions, application-level loops had to be moved down into the library to improve data reuse. When the application was ported on a new MPSoC, without any vector operation support but with multiple processors, loops were moved back up across functional boundaries [Hall et al. 1995] to reoptimize the application differently.

When performance is a concern, a fixed API cannot really remain target independent. Although our approach relies on an API, it is used to provide the underlying application semantics, and the generated code does not have to respect the API; the compiler restructures the computations to fit the target hardware.

Application-specific instructions can be added to an existing general-purpose instruction set. For instance, the Video-Specific Instruction Set Processor [Kim et al. 2006] has special instructions for computational-intensive parts such as inter-block prediction but also uses coprocessors for specific tasks such as entropy encoding. This is close to our case, although these instructions are very algorithm-specific, while we have mostly generic elementary operators.

A particular approach is to use software as a base to develop accelerators, starting from C [Cardoso and Diniz 2008] or Java [Auerbach et al. 2010]. Our compiler could possibly generate specialized accelerators, if hardware descriptions are provided for basic functions. However, this does not seem relevant to our domain, as the same hardware often needs to be used for several applications or functions, or that would require heavy but very fast hardware reconfigurations at runtime.

Optimization of Expression Evaluation. We use commutativity to detect more common subexpressions, but we do not attempt to use advanced algebraic properties [Zory and Coelho 1998], mainly because none of our test cases would benefit from these complex combinatorial optimizations. However we would consider using them if we had an application that would be really improved by such optimizations. Basic block enlargement is useful for trace scheduling [Fisher 1981] and may be obtained by different code transformations, including code hoisting and code sinking [Gupta 1998]. For image processing applications, code hoisting and sinking do not seem useful. Our technique for SPoC is close to the optimization of expression evaluation and vector instruction chaining [Rauber 1990], although in our case we must first meet the pipeline constraints of our target hardware.

10. CONCLUSION AND FUTURE WORK

We have shown how standard compilation techniques can be reused and adapted in an innovative way to fuse tasks and increase locality of image processing applications on multiple accelerator targets. Applications are developed in C using a domain-specific API by any programmer competent in the image processing field, but without knowledge of the underlying accelerators. The implementation of our strategy on top of a source-to-source compiler is simple and greatly reduces development costs. Speedups between 1.2 and 19 are obtained on embedded accelerators for nine applications.

First, what are the conditions to apply our approach to other domains? (1) Our application domain uses one type of large data, images, and a limited set of operators executed on whole images, with a lot of parallelism. (2) Applications have statically decidable behaviors, for instance with a known connectivity and number of iterations, so that the underlying computations can be statically derived and then compiled. However, the image size does not need to be known in advance and is dealt with efficiently by runtimes. (3) The target hardware, both accelerators and OpenCL, are quite well fitted to perform image operations.

Second, what are the conditions necessary for the harmonious codevelopment of the compiler, runtime, and even hardware? (1) The library API is reasonably small, about 40 basic operations and about 20 higher-level combined operations. It is both relevant to the application developers who can find high-level operations and develop functional blocks, and still easily mapped onto the hardware which directly implements most elementary operations. (2) Cooperation between hardware and software experts is key

to deciding if a feature is implemented in the compiler, the runtime, or the hardware. (3) Functional simulators greatly help testing.

Once these conditions are fulfilled, there are several benefits to our API-based compilation approach. The application-hardware gap is small enough to be compatible with a simple compilation strategy, allowing a low-cost fast development and integration in an existing source-to-source compiler. Key benefits include: *portability*—the API works as soon as the library is implemented, and with reasonable performance with the library optimized version, so application development does not need to wait for the availability of the compiler. *Performance*—once available, the compiler provides significant speedups with respect to the library version, and with a portable source-code. *Cost*—as this speedup is achieved by the compiler, less knowledgeable application developers can reach accelerations which would require a hardware expert otherwise; moreover, the software engineering effort invested in the compiler is quite low, thanks to the source-to-source approach and the careful work sharing with the runtime. *Reusability*—application investments are not lost, as the compiler can be retargeted by providing only a new code generator and a runtime. Applications based on the API can thus be simply ported to new targets.

As future work, we expect to port more applications to FREIA so as to extend the test bed for our optimizing compiler and find new profitable transformations. We would like also to port the compiler to new hardware targets. We wish to investigate whether extra code transformations or API enhancements could help improve performance by taking better advantage of event-driven asynchronous calls such as are available with OpenCL. We want to look at the impact on our strategy on domains with multiple data types, such as signal processing applications.

ACKNOWLEDGMENTS

We are indebted to Christophe Clienti who designed the SPoC accelerator, both SPoC and Terapix functional simulators, the Fulguro image library and the FREIA interface, and who ported some applications to FREIA. We also thank Serge Guelton, designer and implementer of the source-to-source inlining phase in PIPS, Laurent Daverio who contributed the code flattening transformation, Michel Bilodeau who provided other application codes and the OpenCL runtime, Pierre Jouvelot whose pieces of advice and proof-reading greatly helped to improve this article, and Alain Darté who discussed at length the NP-completeness issues with us. Finally, we thank anonymous reviewers and the editors for their corrections and suggestions.

REFERENCES

- AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. 1977. Code generation for expressions with common subexpressions. *J. ACM* 24, 1, 146–160.
- AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers Principles, Techniques, and Tools* 2nd Ed. Addison-Wesley.
- AUERBACH, J., BACON, D. F., CHENG, P., AND RABBAH, R. 2010. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *ACM OOPSLA – Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*.
- BILDEAU, M., CLIENTI, C., COELHO, F., GUELTON, S., IRIGOIN, F., KERYELL, R., AND LEMONNIER, F. 2008–2011. FREIA: Framework for embedded image applications. French ANR-funded project with ARMINES (CMM, CRI), THALES (TRT) and Télécom Bretagne.
- BONNOT, P., LEMONNIER, F., EDELIN, G., GAILLAT, G., RUCH, O., AND GAUGET, P. 2008. Definition and SIMD implementation of a multi-processing architecture approach on FPGA. In *Proceedings of Design Automation and Test in Europe*. IEEE, 610–615.
- BRUNO, J. L. AND SETHI, R. 1976. Code generation for a one-register machine. *J. ACM* 23, 3, 502–510.
- CAPS ENTERPRISE. 2008. HMPP, Manycore portable programming. <http://www.caps-enterprise.com/>
- CARDOSO, J. M. P. AND DINIZ, P. C. 2008. *Compilation Techniques for Reconfigurable Architectures*. Springer.

- CHIW, C., KINDLMANN, G., JOHN, R., SAMUELS, L., AND SELTZER, N. 2012. Diderot: A parallel DSL for image analysis and visualization. In *PLDI – Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- CLIENTI, C. 2008. *Fulguro Image Processing Library*. Source Forge.
- CLIENTI, C., BEUCHER, S., AND BILODEAU, M. 2008. A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. In *EUSIPCO: In Proceedings of the European Signal Processing Conference*.
- COELHO, F. AND IRIGOIN, F. 2011. Compiling for a heterogeneous vector image processor. In *Proceedings of the Workshop on Optimizations for DSP and Embedded Systems (ODES'9)*.
- COOK, S. A. AND SETHI, R. 1976. Storage requirements for deterministic polynomial finite recognizable languages. *J. Comput. Syst. Sci.* 13, 1, 25–37.
- CRI, MINES PARISTECH. 1989–2012. PIPS. Open source research compiler, under GPLv3. <http://www.cri.ensmp.fr/pips>
- FISHER, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.* 30, 7, 478–490.
- FREESCALE. 1999. AltiVec technology programming interface manual. <http://www.freescale.com/files/.32bit/doc/ref-manual/ALTVCVECPIM.pdf>
- GLITIA, C., DUMONT, P., AND BOULET, P. 2009. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimen. Syst. Signal Process.* 23, 3.
- GUPTA, R. 1998. A code motion framework for global instruction scheduling. In *Proceedings of the International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 1383. Springer, 219–233.
- HALL, M. H., AMARASINGHE, S. P., MURPHY, B. R., LIAO, S.-W., AND LAM, M. S. 1995. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of the International Conference on Supercomputing*. ACM.
- HENNESSY, J. L. AND PATTERSON, D. A. 2003. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.
- HOCKNEY, R. W. AND JESSHOPE, C. R. 1988. *Parallel Computers 2: Architecture, Programming, and Algorithms*. Adam Hilger, IOP Publishing Ltd.
- KHRONOS GROUP. 2008. OpenCL computing language v1.0. <http://www.khronos.org/opencl/>
- KIM, S., HYUN, C., AND SUNWOO, M. 2006. VSIP: Implementation of video specific instruction-set processor. In *APCCS: Proceedings of the Asia Pacific Conference on Circuits and Systems*. IEEE, 1075–1078.
- OPENMP ARCHITECTURE REVIEW BOARD. 2008. OpenMP application program interface, version 3.0. <http://www.openmp.org/mp-documents/spec30.pdf>
- RAMANATHAN, R. M., CURRY, R., CHENNUPATY, S., CROSS, R. L., KVO, S., AND BUXTON, M. J. 2006. Extending the world's most popular processor architecture. White paper, Intel. *SSE3*.
- RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S., AND DURAND, F. 2012. Decoupling algorithms from schedules for easy optimizations of image processing pipelines. *ACM Trans. Graph.* 31.
- RAUBER, T. 1990. Optimal evaluation of vector expression trees. In *JCIT: Proceedings of the 5th Jerusalem Conference on Information Technology*. 467–473.
- SOILE, P. 2003. *Morphological Image Analysis: Principles and Applications*. Springer.
- THALES (TRT) AND OTHER INSTITUTIONS. 2009. TERAOPS project. <http://www.systematic-paris-region.org/en/projects/teraops>
- VSIPL FORUM. 2008. *VSIPL Std v1.3. Vector Signal Image Processing Library*. VSIPL.
- WOLFE, M. 1995. *High-Performance Compilers for Parallel Computing*. Addison-Wesley Longman.
- ZORY, J. AND COELHO, F. 1998. Using algebraic transformations to optimize expression evaluation in scientific codes. In *PACT: Proceedings of the Conference on Parallel Architectures and Compilation Techniques*. IEEE, Paris, 376–384.

Received June 2012; revised September 2012; accepted November 2012