API-Compiling for Image Hardware Accelerators Technical Report – MINES ParisTech A/500/CRI

Fabien Coelho and François Irigoin
CRI, Maths & Systems, MINES ParisTech, France

June 19, 2012

Abstract

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. 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 a SIMD architecture; the last one runs both on GPGPU and multi-cores through OpenCL. We 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. Experiments were run on hardware functional simulators, covering 1276 cases, from elementary tests to full applications, with very good results. 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 demonstrate that our API-based compilation strategy is a viable approach for both development cost and overall performance, and discuss the conditions required to apply this approach to other application domains.

1 Introduction

Heterogeneous hardware accelerators, based on GPU, FPGA or ASIC, are used to reduce 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. They can also be used to embed the intellectual property of manufacturers or to ensure product perennity. Thanks to Moore's law, their potential advantage increases with respect to standard general-purpose processors which do not gain anymore from the increase in area and transistor number. But all 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 developed on top of a high-level image library onto several image processing accelerator, by generating optimized configurations and relying on high level runtimes. The first target is a heterogeneous processor with a vector image processing accelerator. The second accelerator offers a 128 SIMD processor array. The third backend targets OpenCL for GPGPU or multi-core processors. The approach is relatively inexpensive as mostly-standard and reusable compilation techniques are involved: only the last code generation phase is fine-tuned and target-specific.

Our first hardware target, the SPoC vector image processing accelerator [9], currently runs on a FPGA chip as part of a SoC. The hardware accelerator implements directly some basic image operators, possibly part of the developer visible API: this hardware-level API characterizes the accelerator instruction set. Dozens of elementary image operations such as convolutions, dilatations/erosions (max/min pixel on a stencil), ALUs, thresholds and measures, can be combined to compute whole image expressions per accelerator call. However these capabilities come with constraints: only two images can be fed into the accelerator internal pipeline structure, and two images can be extracted after various image operations are performed on the fly. The accelerator is a set of chained vector units. It does not hold a single image but only a few lines (2 lines per unit) which are streamed in and out of the main memory. There is no way to extract intermediate image values from the pipeline.

Our second hardware target, the Terapix 128 processing element SIMD array, provides elementary pixel operations, neighbor communications and limited memory per processor. It is currently implemented on a FPGA chip as well. The memory does not allow to handle full images, so a rectangular tiling is performed by the runtime. Tailored microcodes are available to perform all image operations on 128-pixel wide tiles, with a varrying height. The accelerator also allows to overlap communications and computations at the price of additional pressure on the available memory to perform the double buffering.

The last target is OpenCL [20], which provides a portable programming environment on top of GPGPU hardware from NVIDIA and other vendors, and allows to run on multi-core processors as well.

The application development relies on the FREIA image processing library API [4]. The API includes low-level basic image operations, and high-level composed image operations which are implemented using the lower level. The developer has no knowledge of the potential target hardware accelerators. Multiple implementations of FREIA are available. (1) A software implementation on top of Fulguro [8], a portable open-source image processing library, is used for functional tests. Then accelerated versions have been develop that take advantage of the hardware accelerator capabilities: (2) Basic operators are ported to the accelerators (SPoC, Terapix, OpenCL) when possible, and the high-level composed operators use a generic implementation which calls the basic operators. However, this approach does not allow to take advantage of specific optimization opportunities offered, especially by the SPoC architecture which can perform many basic operations per accelerator call. (3) Thus a second-level optimized version has been developed, where composed operators are implemented directly on top of the accelerator. Yet again, although the implementation provided by this specialized library version is better, optimization opportunities are still available across functional boundaries. Although the library layer provides functional application portability over accelerators, it does not provide the time, energy and cost performance expected from these hardware.

In order to reach better performance, library developers may be tempted to increase the sizes of API's to provide more opportunities for optimized code to be used, but this is an endless process leading to over-bloated libraries and possibly non-portable code: up to thousands of entries are defined in VSIPL [28], the Vector Signal Image Processing Library. In contrast to this library-restricted approach, we use the basic hardware operator library implementation, but the composition of operations needed to derive an efficient version of a large expression is performed by the compiler for the whole application. We see the image API as a domain specific programming language, and we compile this language for the low-level target architecture.

The keys to performance improvement are to lower the control overhead and to increase data locality at the accelerator level, so that larger numbers of operations are performed for each memory load. This is achieved by merging successive calls to the accelerator,

Figure 1: License plate *(LP)*: character extraction

with no or few memory transfers for the intermediate values. To detect calls to merge, techniques have been developed such as loop fusion [29] or complex polyhedral transformations [23]. Such techniques cannot be applied usefully on a well-designed, highly modular software library such as Fulguro: loops and memory accesses are placed in different modules and loop nests are not adjacent: size checks, type dispatch and dynamic allocations of intermediate values are performed between image processing steps.

Instead of studying the low-level source code and trying to guess its semantics with respect to the available hardware operators, we remain at the higher image operation level. We inline high-level API function calls not directly implemented in the accelerator, unroll loops, flatten the code, so as to increase the size of basic blocks containing image operators. These basic blocs are then analyzed to build expression DAGs using the instruction set of the accelerator. They are optimized by removing common sub-expressions and propagating copies. Up to here, the hardware accelerator is only known by the operations it implements. We then consider hardware constraints to map the optimized DAG onto the actual target, may it be a vector-oriented such as SPoC or SIMD-oriented such as Terapix or OpenCL. The mapping techniques differ depending on the target.

The whole optimization strategy is automated and implemented in PIPS [19, 3], a source-to-source compiler, which lets the user see the C source code that is generated. This greatly helps compiler debugging. We compile 1276 test cases, from elementary tests to full applications, with very good results for our three targets.

In the remainder of this paper, we first introduce our running example which is a short representative of the application domain, along other typical application examples (Section 2). Then we describe our three target architectures and their challenges (Section 3: SPoC, Terapix and OpenCL). We outline our compilation strategy (Section 4), before detailing the target-independent phases (Section 5) and hardware-specific back-end phases (Section 6). We finally present our implementation and experimental results (Section 7) obtained with hardware simulators, and discuss the related work (Section 8) before concluding.

2 Applications and Running Example

The FREIA project aims at mapping efficiently 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, such as a license plate in a picture (LP, Figure 1), whether a car passenger is out of position and could be harmed if the airbag is triggered (OOP, Figure 2), whether there is some motion under a surveyance camera (VS, Figure 3), what macular degeneration can be detected in a picture of a retina (Retina, Figure 4) or the efficiency of various antibiotics against a given bacteria (Antibio, Figure 5). These image applications use all kind of image processing operations, such

Figure 2: Out of position *(OOP)*: airbag ok or not

Figure 3: Video survey *(VS)*: motion detection

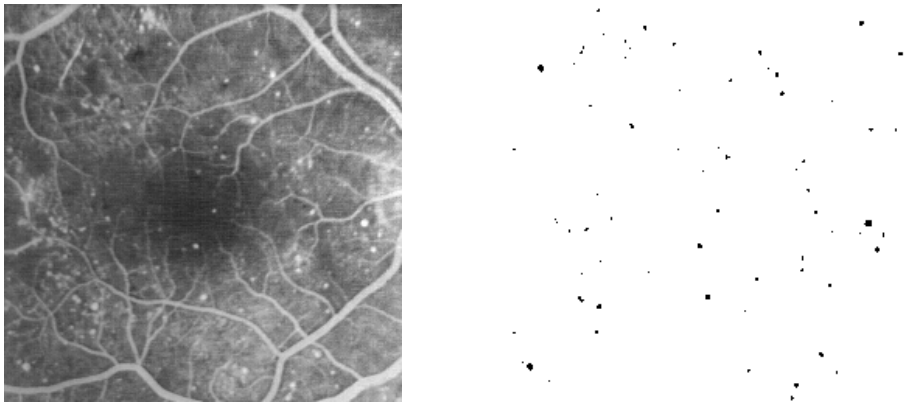

Figure 4: Retina: macular degeneration analysis

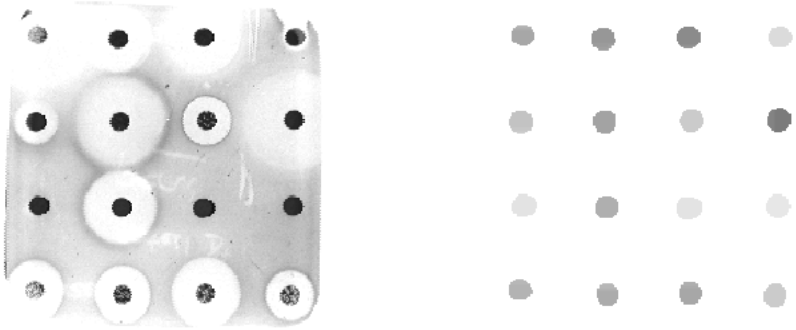

Figure 5: Antibio: antibiotics test plate analysis

```c
#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;
}
```

Figure 6: FREIA API running example (some initialization and cleanup removed)

as: `AND`-ing an image with a mask to select a subregion; `MAXLOC`-cating where the hottest point is; `THR`-esholding an image with values to select regions of interest; mathematical morphology [26] 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 is ready to make some efforts in order to reach the expected high performances for critical applications on selected hardware. Current development costs are high, as application must be optimized from the high-level algorithmic choices down to the low-level assembler code and memory transfers for every hardware target. The project aims at reducing these development costs through optimizing compilation and careful runtime designs.

The high-level FREIA image API has several implementations. The first one is pure C, based on the Fulguro [8] open-source image processing library, and is used for the functional validation of applications. There are two implementations for the SPoC vector hardware accelerator, which can run over a functional simulator or on top of the actual FPGA-based hardware: one uses SPoC for elementary functions, which are directly supported by the SPoC instruction set, one elementary operator at a time; the other is hand-optimized at the library call level by taking full advantage of the SPoC vector hardware capability to combine operations. Other versions of the library are optimized for the Terapix [5] SIMD accelerator, and for OpenCL [20] targeting graphics hardware (GPGPU).

The code in Figure 6 was written as part of the FREIA project to provide a short test case significant for both the difficulties involved and the optimization potential, with the two hardware accelerators in mind. The test case contains all the steps of a typical image processing code: an image is read, intermediate images are allocated and processed, and results are displayed. As it is short enough to fit in this paper, we use it as a running example, together with extracts from larger applications. Optimization opportunities for a compiler at the `main` level of our test case are very limited: the `min` and `vol` basic function calls correspond to SPoC instructions. Since they are next to each other and use the same input argument, they can be merged into a unique call to SPoC. The `dilate` and `gradient` functions are not part of the accelerator instruction set. With the naïve elementary function-based library implementation, 33 calls to the accelerator are used per frame,

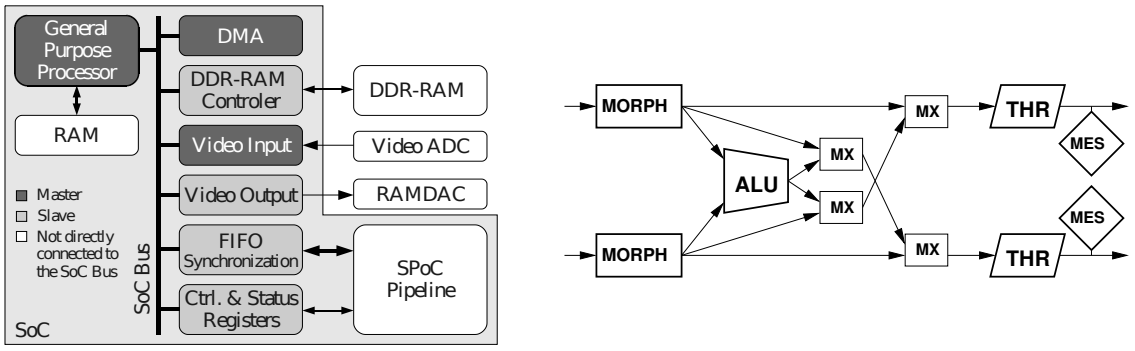

Figure 7: SPoC architecture and one vector unit Typically 8 vector units are chained in the SPoC pipeline

mostly hidden in the callees. The hand-optimized SPoC implementation of the FREIA image library results in 6 accelerator calls, a $\times 5.5$ speed-up, as calls to elementary 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 stategy will detect this redundancy and merge all image operations into only 2 accelerator calls, achieving an additional $\times 3$ speed-up over the hand-optimized library.

3 Hardware Targets

This section introduces our three hardware targets. The first one, SPoC, presents a vector architecture. The second one, Terapix, relies on a SIMD array of small processing elements. The third one is generic, based on OpenCL, so that applications can be run on GPGPU and multi-cores which support this environment.

SPoC Architecture

Figure 7 outlines the structure of the SPoC processor. It can be seen as a simplified version of the 30-year old CDC Cyber 205 [18], 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, constitutes the image processing accelerator. It also contains a DDR3 memory controller, DMA engines, FIFOs to synchronize memory transfers and vector computations and the host, a gigabit Ethernet interface and video converters for I/Os.

Figure 7 shows one vector unit of the SPoC pipeline, with two inputs and two outputs of 16-bit-per-pixel images. The units are chained linearly, directly one to the next, using their outputs and inputs: there is no vector image registers. The first inputs and last outputs are connected to the external memory by DMA engines. A vector unit is made of several operators, but the interconnection is not free: the data paths are quite rigid, with some control by multiplexers Mx. One morphological operator Morph can be applied to each input. Their results can be combined by an arithmetic and logic unit, alu. Two outputs are selected among those three results by the four multiplexers which control the stream of images. Then a threshold operator, Thr, can be applied to each selected output and the reduction engine Mes computes reductions such as maximum or sum of the passing pixels, the result of which can be extracted if needed after the accelerator call. To sum up, each micro-instruction can perform concurrently up to 5 full image operations and a number of reductions, equivalent to 29 pixel operations per tick. A nop no-operation

micro-instruction is available to copy the two inputs on the two outputs. It is useful when some vector units of the SPoC pipeline are unused.

The host processor controls the vector units by sending them one micro-instruction each and by configuring the four DMA engines for loading and storing pixels. The host processor can also retrieve the reduction results from the vector units. 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.

When considering FPGA implementations, the number of vector micro-instructions that can be executed concurrently, i.e. the number of vector units, ranges from 4 to 32. The limiting factor is the internal RAM available. Our reference target hardware includes 8 vector processing units, but the solution we suggest below is parametric with respect to this number. In practice, this vector depth provides a reasonable cost-performance trade-off as it fits patterns of iterated convolutions, erosions or dilatations that are often found in typical applications, but is yet not too expensive when these patterns are not found. With a specific set of application in mind, several vector depths can be tested to choose the best setting. The total number of image operations that can be executed at a given time is 5 times the number of units, not counting the reductions. So the compiler must chain 40 image operations of the proper kind and order to obtain the peak performance. Unlike the Cray vector register architecture, only two inputs are available. Unlike the CDC 205, no general interconnection is present between elementary functional unit. Chaining and register allocation are very much constrained as each vector processing unit is pipelined: delay lines help compute 3×3 morphological convolutions, including a transparent and accurate management of the image boundaries that are out of the stencil. Thus the size of the output image is equal to the input image size, contrary to repeated stencil computations [11] which usually reduce the available image size. Micro-instruction scheduling and compaction is easy once the order of operations is determined.

To sum up, the significant hardware constraints are: 1) the structure of the micro-instruction set and the structure of the vector unit data paths, 2) the maximal number of chained micro-instructions, i.e. the number of vector units, and 3) the number of image paths, two. Furthermore, the operations must be as packed as possible to reduce the number of micro-instructions. With 8 vector units, up to 40 full image operations can be performed for two loads and two stores, which leads to 10 SPoC operations per memory access, including high-level morphological convolutions which require more than 20 elementary operations each, and not counting the many reductions. So between 50 and 100 elementary operations can be executed per memory access.

Terapix Architecture and Runtime

The Terapix accelerator [5] is a rigid 128-processing elements (PE) SIMD array with a limited local memory of 1024 pixels per PE. It is developed by THALES as a SoC for image processing embedded systems, such as security surveillance cameras. The host/accelerator memory bandwidth allows to send or receive 4 pixels per tick, and these transfers can run in parallel with computations, overlapping computations and communications, at the price of manually managing double buffers and reducing the memory available for one computation. Each PE accesses its own memory and can send or receive values from its neighbors. Computation kernels have been developed, initially manually, to compute each image operator on up to two input imagelets and put the result in one output imagelet. Moreover, when stencils are computed on imagelets, the area of valid results shrinks as borders between imagelets result in bad values because of missing border pixels. However, the architecture and kernels ensure that computed values for the actual border of the full image (not inter-imagelet) are correct, based on the semantics of the image operators. The memory constraints imply that full images cannot be dealt with directly, requiring

the runtime to perform an overlapping tiling of the input. An additional shared 2D global memory allows to pass parameters such as stencil coefficients needed for computations.

The runtime performs an important task which greatly helps the compilation process: images are automatically tiled based on declared overlaps and imagelet size, and the resulting imagelets are pipelined, so that whole image operations are performed seamlessly. The hardware configuration must provide an explicit memory allocation schemes, including double buffers where needed, and a full schedule of communications and computations for both computations (one with one set of buffers, the other with the other set). The runtime inserts synchronizations so that computations wait for the availability of their input data, and ouput communications wait for the availability of the results.

There is a speed-up ceiling for the Terapix accelerator due to the intrinsic overlapping of communications and computation of stencils. For a typical morphological erosion or dilatation computation, there is one imagelet to send and one imagelet to receive, which requires 64 cycles per imagelet row, and the computation itself needs 15 cycles per row. These operations can be combined, but it is not interesting to do so once the computations amortize all the communications, especially as the area of valid results shrinks as the depth increases. This ceiling occurs when $64/15 \approx 4$ such operations are put together, so the practical maximum speed-up that can be expected when these morphological operations are used is about 4, due to the intrinsic communication-computation hardware balance.

OpenCL Target

OpenCL (Open Computing Language [20]) 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 multi-threaded programming model that executes simple kernels on many data sent to an external device. Moreover, dependencies can be declared between computations to expose task parallelism as well. Hardware portability is reached by compiling kernels on the fly so that it is optimized for the particular hardware at hand. The actual performance obtained with such codes depend 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 freshness of each image so as to minimize these communications. The memory allocation also checks dynamically for data that may be overwritten by operations while still needed, and allocate 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 intra-device memory bandwith allows to fully exploit the available processors.

4 API-Based Compilation Strategy

Our overall strategy is to rely on a Domain Specific Language (DSL) for image applications, the semantics of which is provided by well designed API calls to low-level basic operators and high-level operators implemented on top of the former level. From these surmises, we propose a compilation strategy which first builds image operation basic blocks that can be mapped onto our target accelerators, optimizes the expression DAGs derived from these basic blocks, and then generates target-specific code for the accelerator hardware in specific back-ends. The implementation is low-cost because of the reuse of standard phases, and the overall performance is very good (Section 7).

Phase 1 application preprocessing – enlarge basic blocs

- **1.** inlining of FREIA high-level library functions
- **2.** partial evaluation
- **3.** constant propagation and loop full unrolling
- **4.** convergence while unrolling (only for SPoC)
- **5.** dead code elimination
- **6.** block flattening

Phase 2 DAG optimizations

- **1.** DAG construction per sequence
- **2.** operator normalization *(improve CSE stage)*
- **3.** algebraic opt: 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

Phase 3 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

Phase 3.2 Terapix configuration: map DAG onto hardware

- **1.** DAG splitting and scheduling
- **2.** memory allocation, including double buffers
- **3.** instruction generation, chooses best imagelet size dynamically
- **4.** unused image cleanup

Phase 3.3 OpenCL code generation

- **1.** DAG splitting along scalar dependencies
- **2.** simple operator aggregation
- **3.** OpenCL kernel generation
- **4.** unused image cleanup

Figure 8: Outline of our compilation strategy: phases and stages

Figure 8 outlines the different phases of our compilation strategy, which are detailed hereafter. The first phase is a combination of standard transformations to build larger basic blocks; The second phase, optimizes the image operation DAG by detecting common sub-expressions and removing dead-code. As the third phase focuses on generating code for the hardware targets, there is a specific approach for each target.

5 Target-Independent Preprocessing Phases

The first compilation phases are (mostly) target-independent. They aim at retrieving from the source the underlying basic image operations, to aggregate them into basic blocks, and to build and optimize image operation DAGS.

Phase 1 – Application Preprocessing

The FREIA API [4] and its Fulguro [8] implementation are designed to be general with respect to connectivity, image sizes and pixel representation. Standard or advanced loop transformations cannot take advantage of such source code because the loops are distributed into different functions and elementary array accesses are hidden into function calls to preserve the abstraction over pixel structure.

```
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);
```

Figure 9: Excerpt of the `main` of our running example (Figure 6) after preprocessing

To build large basic blocks of elementary image operations, control flow breaks such as procedure call sites, local declarations, branches and loops must be removed by using key parameters such as connectivity and image size set up by the `main` and propagated to callees. Several source-to-source transformations help achieve this goal: 1) inlining to suppress functional boundaries; 2) partial evaluation to reduce control complexity; 3) constant propagation to allow full loop unrolling; 4) for the SPoC target only, a special case of convergence while-loop is detected and can be unrolled; 5) dead code elimination to remove useless control; 6) declaration flattening to suppress basic block breaks. Safety tests are automatically eliminated as the application is assumed correct before its optimization is started. The order of application of these transformations is chosen to make the best usage of the available information so as to simplify the code and obtain larger basic blocks. Figure 9 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 the expression DAG shown in Figure 10, which is then optimized for locality. We also rename images and generate new temporaries in order to avoid any conflict that may generate unexpected false dependencies when the code will be regenerated, as the overall operation order is not kept by the optimizations which only rely on flow dependencies. The many images introduced by this renaming phase will be reduced as much as possible later.

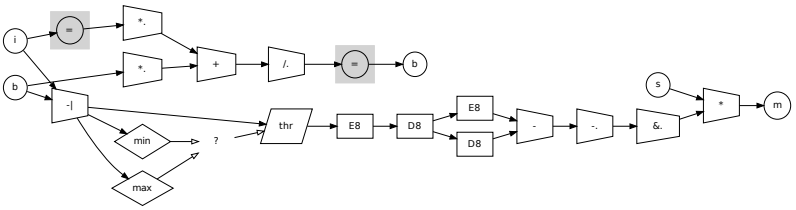

Figure 10: DAG sample from *Video Survey* (the upper parts updates background b, the lower parts detects movements in hot regions)

The vertices of the DAG are operations to be performed, which may be image operations (convolutions and MORPH dilations/erosions as rectangles, ALU as trapezium, THR as parallelogram, MES as diamond, copy and input/output images as circles) or intermediate scalar operations depicted as question marks. The arcs represent the dependencies between operations, when a piece of data defined at the source node end is used at the sink node. Arcs shown as black arrows embed image dependencies while the white arrows do scalar dependencies. For instance the result of reductions on an image is used after some computation for thresholding it. The DAG derived from our running example is shown in the left side of Figure 11, with the initial functional boundaries in dashed lines.

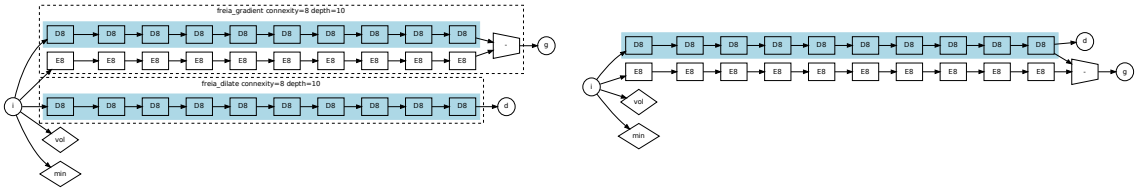

Figure 11: Initial and optimized expression DAG for our running example (Figure 6)

This 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 sub-expressions to be detected later. Second, algebraic simplifications are performed. In particular, constant image expressions, *i.e.* expressions than 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 12, developers often xor an image against itsef in order to generate a constant black image: although perfectly

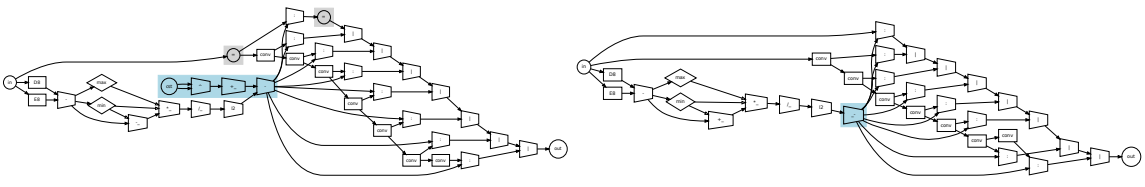

Figure 12: Initial and optimized DAG from application *Deblocking*

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 image paths available. Third, we apply standard compilation techniques: common sub-expressions elimination, dead code suppression and copy propagation [2] are performed at the image level. Operator commutativity is taken into account to perform CSE, thanks to the image operator semantics which is available by recognizing the calls. Simple information about parameters, image or scalar, input and/or output, are derived automatically from C source stubs. In our running example, the optimization detects that all operations of the *dilatation* are also performed within the *gradient*, so they are removed in the right hand side of Figure 11, and the intermediate result is simply extracted.

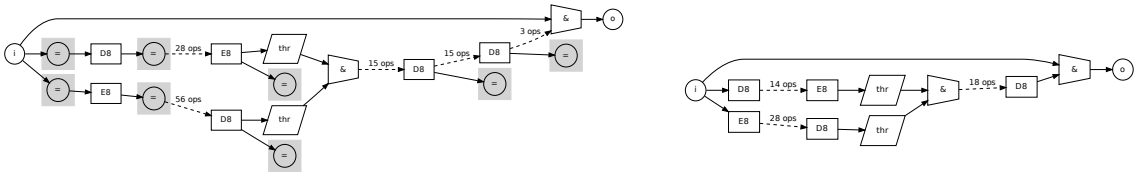

Figure 13: Extract of initial and optimized DAG for *License Plate*

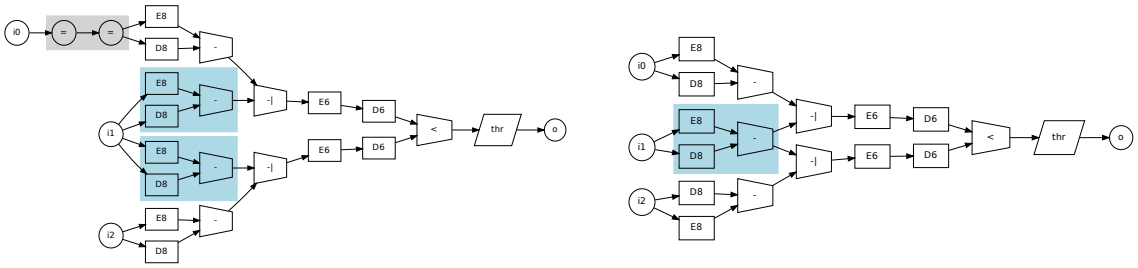

Figure 14: Initial and optimized DAG from application *OOP*

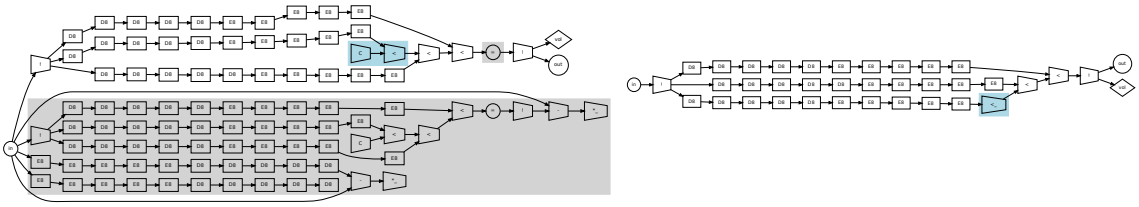

Figure 15: Initial and optimized DAG from application *Retina*

Other challenges are found in the DAG for the *license plate* application in Figure 13, where repeated operations are denoted as dashed arrows. In this debug version of the code, every two operations are image copies either inserted within the computations or diverted to extract intermediate images. All these useless copies are removed from the optimized DAG. Image copies are propagated forward toward their uses, with the exception of copies on output images which are propagated backward to their producer so that they are directly generated instead of using a temporary image. Remaining input and output image copies are extracted from the DAG to be performed outside of the accelerator.

The DAG in Figure 14 is extracted from the *OOP* application. The optimized version removed both copies and a common subexpression involving 3 operations. These redundancies are not obvious to spot in the source code. The DAG from Figure 15 displays a very large piece of dead code which comes from variants tested by the developer. The

optimization of the DAG in Figure 10 removes both copy operations on input and within the graph. The result of this phase is an optimized image expression DAG ready to be mapped onto the available hardware.

6 Target-Dependent Code Generation

An accelerator specific compilation phase finally generates the hardware configuration, i.e. the micro-instructions for evaluating the optimized DAG resulting from the previous phase. We have three code generators; one for SPoC vector pipeline, one for the Terapix SIMD accelerator, one for OpenCL.

Phase 3.1 – SPoC Hardware Configuration

The SPoC hardware accelerator [9] constraints discussed in Section 3 must be met: the computations in the hardware accelerator must only involve two live images at any single point because only two data paths are available (Figure 7). Actual computations must be scheduled on components so that live images can still reach their use or the end of the path. If all available vector units in the SPoC pipeline are used for a computation and more operations remain, pipeline spilling must be managed. The optimality criterion is to minimize the number of calls to the accelerator, taking into account its actual number of vector units, as one call lasts 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, with in our case only two registers. However, unlike register allocation problems, our spill code is to interrupt 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 [6, 1] is NP-Complete, these results do not apply directly to our case.

We conjecture nevertheless that our problem is NP-Complete, because of the close similarity with the code generation problem for register-machines. First, 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. Second, evaluating the cost of a proposed solution is reasonably easy: given an order 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 the 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, where each resulting sub-DAG operations are ordered by the 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 available vector units. We chose not to perform a global combinatorial optimization because this simple heuristic, which satisfies each constraint one after the other, leads to excellent experimental results on significant test cases (Section 7).

The optimized expression DAG is first split into sub-DAGS with no more than two live images and no internal scalar-carried dependencies. 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* [2], with a prioritized topological order that focuses on the critical resource, namely the small number of data paths. Scalar depen-

dencies cannot be handled within one hardware accelerator call as images are processed concurrently, so the needed result would not be available at the start of the dependent computation: they must be split across distinct sub-DAGs. 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 DAGs, each with an *ordered* list of operations that require no more than two live images if processed in that particular order along the pipe.

Each sub-DAG is then mapped onto a pipeline with a conceptually infinite number of vector units by compacting operations into micro-instructions. We do not allow much freedom at this stage because the order of operations cannot be modified without putting at risk the two live image constraint. It is kept unchanged. Micro-instruction compaction is performed at the same time because the packing constraints are very easy to meet: structural, control and data pipeline hazards [17] are avoided by the hardware, hence sophisticated micro-instruction scheduling and compaction are not required. The compaction is achieved by scheduling operations in the first available slot. When only one image is needed by the pipeline, it is sent on both input paths so as to help the compaction at the beginning of the pipe. Path selection implies the multiplexer configuration. It must ensure that computed images reach the operators that process them, which may shift a computation further down in the pipeline in some cases. Under these assumptions, this compaction stage could be proven optimal, that is the number of units used is minimal, by induction on the structure of the pipeline, as we choose the first available operator at each iteration. However this optimality is weak because it requires that there is no reordering of the operations, which could have improved the result if allowed. Moreover this optimality is local, and taking this constraint in the previous stage could help improve the overall solution.

The third stage of the code generation process is to map the open-ended pipeline onto the available vector units. This is simply achieved by cutting the micro-instructions sequence at the number of available vector units, and to perform another activation of the SPoC pipeline for the remainder, until all sub-DAG operations are performed.

This heuristic phase for the SPoC accelerator reuses standard compilation techniques to generate most of the time optimal results. It is followed by a quick cleanup of intermediate images which are not used anymore 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.

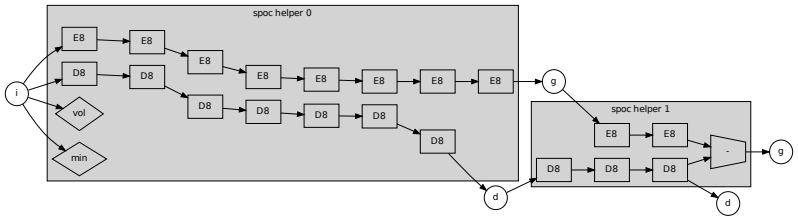

Figure 16: Running example DAG split on SPoC

The two DAGs corresponding to the splitting for the SPoC target of the optimized DAG of our running example is shown in Figure 16. The sequence of dilatations is delayed by one slot to allow for 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.

For the SPoC target, there is a cost performance tradeoff in choosing the number of vector units, as longer pipelines are less efficiently used when no operations can be scheduled and add to the overall latency of accelerator calls. The solution to this tradeoff depends on the actual applications and on the user ability to select optimal hardware. It is not taken into account here as we assume that the number of vector units is a given, with 8 a typical figure. However, testing the impact of different depths is easy thanks to the availability of our compiler.

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 that the valid area computed shrinks with stencil operations, and (2) allocate the little available memory to imagelets needed for a computation, including double buffers used to improve the hardware usage efficiency.

The first aim is reached by splitting DAG with *vertical* slices (assuming that inputs are on the left and outputs are on the right, as consistently used in the illustration of this paper), so that intermediate images valid areas are shrinked by the same amount of border. The number of strips is computed based on the available memory bandwidth, so that communication and computation costs are balanced:

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

where the comp/comm ratio is the number of images that can be transmitted during the computation of the DAG. Those that need to be transmitted are the input and output images, plus the intermediate images if the DAG is cut, this later number being approximated by the width of the DAG. The rational 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 pixel on each side) is computed, and the DAG is split at these depths. Although more accurate options could be thought of, this simplistic approach gives reasonable results on our test cases and was kept. For small DAGs, splitting does not bring any benefice.

The second aim is reached through a list-directed scheduling of operations in each splitted DAGs, similar to the one described earlier for SPoC, but with different priorities applied. The overall design of the priorities is to favor the consumption of data so as to make imagelet memory slots available for later computations. The chosen schedule induces a total number of imagelets to perform the whole computations: first, data input and output imagelets must be allocated twice for double buffering; then some additional imagelets may be required if some operations do not free their operands, or operations such as dilatations that cannot be performed in place. The schedule also allows to generate the computation code, which consists of calling operator kernels with parameters that designate the input and output imagelet operands. Two versions are generated for each set of double buffers, *i.e.* one for the *flip* and one for the *flop*. The schedule attempts to put result imagelets directly in the relevant double buffers; if a result imagelet is not stored in a double buffer, an additional copy operation is inserted at the end of the imagelet computation. The generated code also includes the input and output communications needed to feed the pipelined computations, and border erosion declarations that enable the runtime to compute the appropriate overlapping tiling.

Once the number of imagelets for a schedule is known, the maximum imagelet height can be computed dividing the available memory by the number of imagelets. The imagelet width is always 128, which is 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 imagelet tiles for an image is usually quite small, typically 6 to 10, and as the operator kernels perform their computations on whole imagelets whatever the live part in them is, significant side effects occur which can change the optimal size significantly. Thus, once the input image size is known, the generated code computes the best possible imagelet size based on the actual image size and the amount of overlapping borders, so as to minimize the amount of unused computations due to these side effects.

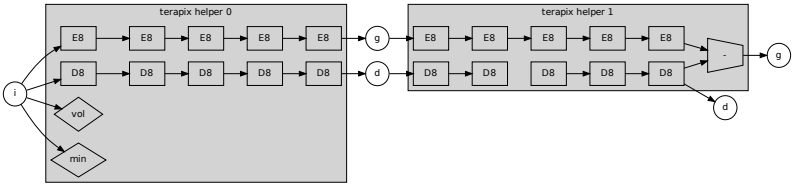

Figure 17: Running example DAG split on Terapix

The splitting result of the running example for Terapix is shown in Figure 17. In contrast to the splitting for SPoC, the sub-DAGs for Terapix are balanced and the border erosion is the same in both accelerator calls.

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 9300 GE, launched in June 2008, can perform about 16 integer operations per 8 bytes transfered, that is 4 operations per 2-byte pixels: the 9 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 time dominates the overall performance. Thus we aggregate such arithmetic operations when possible, and generate dedicated kernels to compute them.

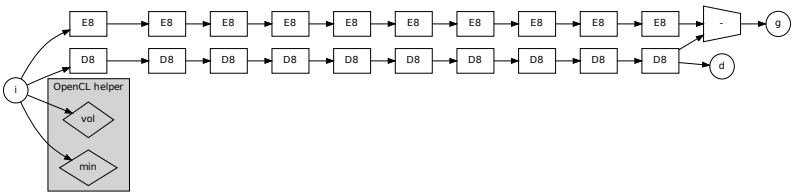

Figure 18: Running example operator aggregation for OpenCL

Figure 18 shows the simple arithmetic operation aggregation performed on our running example for OpenCL: the two reductions are computed together. No other aggregation is performed as all others but one of the image operations are stencil computations which do not need such transformations.

7 Implementation and experiments

Our optimization strategy is implemented in PIPS [10], a source-to-source research compiler, for a small development cost measured hereafter with the KLOCs *(thousand lines of code)* involved. Transformations of Phase 1 are more or less standard in an advanced

```
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_IN0;                // 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;
}
```

Figure 19: SPoC helper code excerpt for the running example (Figure 6)

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

Figure 20: SPoC helper (Figure 19) stub calls

```
void running_tpx_helper_1(f_data2d *o0, f_data2d *o1,
    f_data2d *i0, f_data2d *i1, int32_t *pi4) {
  // SKIPPED: declarations & initializations
  int imglet_1 = 0, imglet_2 = 204, ...;              // ... up to 5
  int imglet_io_1_0 = imglet_1;                       // double buffers
  int imglet_io_1_1 = imglet_4;                       // ... 2 more
  int vertical_border = 5, imglet_computed_size = 194;
  int imglet_size = ... ;   // compute best imagelet size depending on input image
  mcu_macro[0][0].xmin1 = imglet_io_1_0;        // E8(1) -> 3
  mcu_macro[0][0].xmin2 = imglet_3;             // more settings skipped
  mcu_macro[0][0].iter2 = imglet_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 imagelet erosion
  mcu_instr.imagelet_height = imglet_size;
  mcu_instr.nbin = 2;                           // I/O imagelet declarations...
  tile_in[0][0].x = imglet_io_1_0;              // more settings skipped
  freia_cg_process(&param, o0, o1, i0, i1);     // call accelerator
}
```

Figure 21: Terapix helper code excerpt for the running example (Figure 6)

```
KERNEL void oop_opencl_helper_3(GLOBAL PIXEL * o0, int ofs_o0,  // output image
  GLOBAL PIXEL * i0, int ofs_i0, GLOBAL PIXEL * i1, int ofs_i1,  // input image
  int width, int pitch, int c0, int c1, int c2) { // thread & scalar parameters
  GLOBAL PIXEL *p0 = o0 + ofs_o0;                 // ... IDEM j0/i0 & j1/i1
  int gid = pitch*get_global_id(0);
  for (int i=gid; i < (gid+width); i++) {          // some computations
    PIXEL in0 = j0[i], in1 = j1[i];                // read input pixels
    PIXEL t228 = PIXEL_INF(in1, in0);
    PIXEL t230 = PIXEL_THRESHOLD(t228, c0, c1, c2);
    p0[i] = t230;                                  // write output pixel
} }
```

Figure 22: OpenCL helper code excerpt for *OOP* (Figure 14)

optimizing compiler. They amount to 155 KLOCs in PIPS, including the parser, semantical analyses, 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 about 3 KLOCs for representing the FREIA elementary operator semantics, plus building and optimizing the DAG representation. Phase 3 is the back-end specific code generation. It uses about 1.7 KLOCs including DAG splitting, scheduling, wiring, and SPoC configuration, 1.3 KLOCs for mapping onto Terapix, and 0.6 KLOCs for OpenCL. Thus the implementation for phase 2 and 3 added about 4% to the code base of the compiler. 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 acceleration functions to be called from the initial application. Each generated configuration function (see excerpts in Figure 19 for SPoC, Figure 21 for Terapix and Figure 22 for OpenCL) is called from the `main` with the appropriate arguments (*e.g.* in Figure 20). Other applications may require more preprocessing phases, such as while loop unrolling or code hoisting, to obtain longer basic blocks.

Apps	dags	image ops			speedups		
		static		run.	SPoC	Tpx	OCL
		ini	opt	ini			
Anr999	1	33	23	33	16.5	3.1	1.6
Antibio	7	49	48	832	9.2	-	1.5
Burner	4	422	422	518	14.0	2.5	1.1
Deblocking	1	36	32	36	2.1	2.4	1.2
LP	1	135	69	135	16.9	2.6	2.0
OOP	1	22	18	4290	3.1	4.1	1.7
Retina	3	106	42	304	19.0	2.6	1.8
Toggle	1	15	15	15	2.1	-	2.1
VS (core)	1	17	16	95339	(?) 2.0	-	-
Overall	*2.2*	*93*	*76*	***370***	***6.5***	***2.8***	***1.6***

Figure 23: Whole application speed-ups **vs optimized library** on the three target architectures Averages are *arithmetic* for static, ***geometric*** for dynamic counts

Our prototype FREIA compiler was tested on 1276 cases, comprising 1005 combinatorial tests (3 to 6 ops), 169 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) tested with various parameters. Figure 24 shows speed-ups

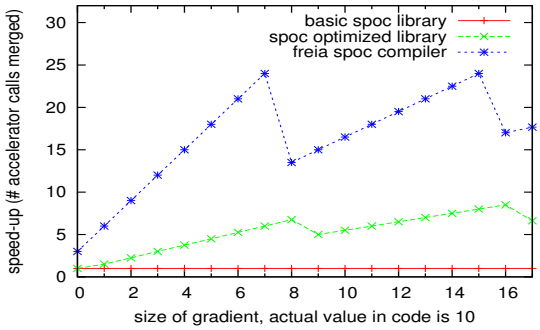

Figure 24: Speed-ups on SPoC with 8 vector units for the running example (Figure 6)

on actual applications, compared to the *optimized* FREIA library. For each application we display: the number of dags found, the number of image operation before and after DAG optimizations, the initial number of image operations at runtime, and finally the speed-ups on the three target architectures. On average, DAGs found in the applications comprise 42 vertices. A few figures are missing: bugs not yet fixed in the Terapix functional simulator prevented some runs to complete successfully, and there is an issue with the VS full application. The overall compilation time, with all preprocessing and code generation, is reasonable, especially as not much time has been spent on optimizing the implementation: most cases are processed in a few seconds, although the largest application (422 ops Burner) requires about 2 minutes.

Figure 24 compares speed-ups obtained on parametric versions of the running example in Figure 2 where the numbers of iterations is adjusted from 0 to 17 (the source code selected executes 10 iterations) on SPoC hardware. In the baseline version, each call to the accelerator performs only one operation. In the optimized library version, each call to a FREIA operator is optimized independently. Finally the PIPS version is generated with the techniques described in this paper to optimize the whole application. It fares better than any other version, thanks to the extracted common sub-expression and the optimal (in this case) hardware mapping which combines elementary operations whenever possible. Note that the optimized version runs out of vector units for a size 8 gradient operation, whereas the version using the SPoC optimized library goes down for 9: the first vector unit is used up by the optimized version for the volume and minimum measurements in the input image, hence the shift of the discontinuity between the two versions.

For the SPoC accelerator, only 30 cases on 1276 are not optimal: 28 are one call away from optimality, and 2 are not optimal by 2 and 3 calls. Most of these non optimality cases are linked to the greedy nature of the heuristic coupled with pipeline spilling effects. The average speed-up we obtained on SPoC for the available applications compared with the optimized library version is 6.46.

The optimality for the Terapix accelerator is simple to define: the computations take the minimal time to execute on the architecture with respect to *all* possible transformations and code generation techniques. However, it is much less obvious to check for the potential optimality of the results found by our compilation strategy. The average measured execution time speed-up we obtained on Terapix for the working applications is 2.81. Although this is less than the SPoC results, this must be compared to the intrinsic speed-up ceiling which is around 4 on this architecture for stencil computations, which compose an important part of image operations in our sample applications.

For OpenCL, it is not easy to discuss performance which may vary greatly depending on the device, the runtime and the compiler. In order to devise a device and compiler-neutral speed-up, we compute the speed-up based on the number of dynamic kernel calls of

the library version compared to the kernel called with our compiled version. The average speed-up we obtained on OpenCL for the available applications is 1.59. This is significant, especially as applications use a lot of stencil computations which do not need nor allow much optimization opportunities, and as the library version already performs quite well.

8 Related Work

The related work is rather limited because researchers looking at hardware accelerators 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 and do not advertise it. 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 which are not well served by general purpose languages. The OpenCL [20] recent standard aims at providing portability across accelerators, especially GPGPU targets, but it is quite large and rather low-level: application developers should be able to ignore it [29]. 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 [14] is an example domain-specific language designed for signal processing and for accelerator programming. However, Array-OL is not general enough to enable writing 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 graphical tool.

Pragma annotations on top of standard languages are used to preserve the portability of applications and allow their functional validation in a standard environment. OpenMP [22] allows the developer to hint 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 [7] 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, issue which is not addressed by our technique. However the set of directives is very specific and requires deep architectural insight from the developer to be exploited fully.

Another way of achieving high performance on specialized hardware and still retain portability is to use domain-specific libraries which can be implemented for various targets. VSIPL [28] 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 functions core lite profile, followed by the 513 functions 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 [27] 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 has been designed for vector-

based instruction set additions such as Altivec or Intel SSE extension family [24, 13]. To optimize its functions, application-level loops had to be moved down into the library to improve data re-use. 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 [16] to re-optimize 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 [21] has special instructions for computational intensive parts such as inter-block prediction but also uses co-processors 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.

Optimization of expression evaluation

We use commutativity to detect more common subexpressions, but we do not currently attempt to use advanced algebraic properties [30], mainly because none of our test cases would benefit from these complex combinatorial optimizations. However we would consider using them if we had a motivating example that would be really improved by such optimizations. Basic block enlargement is useful for trace scheduling [12] and may be obtained by different code transformations, including code hoisting and code sinking [15]. For image processing applications, code hoisting and sinking do not seem useful. Our technique is close to the optimization of expression evaluation and vector instruction chaining [25], although in our case we must preliminary meet the pipeline constraints of our target hardware.

9 Conclusion and Future Work

We have shown how standard compilation techniques can be efficiently reused and adapted to optimize applications based on an image processing library for two domain-specific hardware accelerators and a generic OpenCL target. Applications can be developed in C by any programmer competent in the image processing field, but without knowledge of the underlying hardware accelerators, and are automatically optimized for the specific target system without any of the traditional hurdles such as the procedure calls imposed by the different APIs used. The source code transformations and the high-level optimization strategy is simple. It properly combines and adapts existing techniques to perform a wide range of loop fusions based on semantical information. This simplicity is an asset, as it greatly reduces the development costs of the compiler and bring large speed-ups.

What are the conditions for our approach to apply 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 implicit locality and parallelism. (2) 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 implements directly most of the elementary operations. (3) The target hardware, both accelerators and OpenCL, are quite well fitted to perform image operations. (4) Functional simulators allow to test very simply the compiler, what greatly helps compiler development. (5) Hardware and software experts, who moved between them what is done by the hardware,

runtime and compiler, must trust each other and really work together.

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: (1) 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. (2) Performance: once available, the compiler provides a significant speedup with respect to the optimized library version, and with a portable source code. (3) Cost: as this speedup is achieved by the compiler, less knowledgeable application developers can reach accelerations which would require a hardware expert otherwise. (4) Perennity: the code can be retargeted by providing a new code generator, so these investments in applications do not need to be spent all over again if a new hardware is available;

As future work, we expect more applications to be ported to FREIA so as to extend the test bed for our optimizing compiler and possibly find new profitable transformations. We also want retarget our compiler to updates of our current accelerators or to new accelerators. We're planning to have a look at the impact on our strategy on domains with multiple data types, such as signal processing applications.

Acknowledgment

We are especially 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 on the FREIA interface. We also thank Serge Guelton, designer, implementer and maintainer 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 for FREIA. Pierre Jouvelot advices and proof-reading greatly helped to improve this paper. Finally, we want to thank Alain Darte for a long discussion about NP-Complete problems.

References

[1] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, 1977.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[3] Mehdi Amini, Corinne Ancourt, Fabien Coelho, François Irigoin, Pierre Jouvelot, Ronan Keryell, Pierre Villalon, Béatrice Creusillet, and Serge Guelton. PIPS Is not (just) Polyhedral Software. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[4] Michel Bilodeau, Christophe Clienti, Fabien Coelho, Serge Guelton, François Irigoin, Ronan Keryell, and Fabrice Lemonnier. FREIA: FRamework for Embedded Image Applications, 2008–2011. French ANR-funded project with ARMINES (CMM, CRI), THALES (TRT) and Télécom Bretagne.

[5] Philippe Bonnot, Fabrice Lemonnier, Gilbert Edelin, Gérard Gaillat, Olivier Ruch, and Pascal Gauget. Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA. In *Design Automation and Test in Europe*, pages 610–615. IEEE, December 2008.

[6] John L. Bruno and Ravi Sethi. Code generation for a one-register machine. *J. ACM*, 23(3):502–510, 1976.

[7] CAPS enterprise. HMPP, Manycore Portable Programming, 2008.

[8] Christophe Clienti. Fulguro image processing library. Source Forge, 2008.

[9] Christophe Clienti, Serge Beucher, and Michel Bilodeau. A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. In *EUSIPCO: European Signal Processing Conference*, August 2008.

[10] CRI, MINES ParisTech. PIPS, 1989–2012. Open source Research Compiler, under GPLv3.

[11] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08: Conference on Supercomputing*, pages 1–12. IEEE Press, 2008.

[12] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[13] Freescale. AltiVec Technology Programming Interface Manual. web site, 1999.

[14] Calin Glitia, Philippe Dumont, and Pierre Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 23(3), 2009.

[15] Rajiv Gupta. A code motion framework for global instruction scheduling. In *International Conference on Compiler Construction, LNCS 1383*, pages 219–233. Springer Verlag, 1998.

[16] Mary H. Hall, , Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Conference on High Performance Networking and Computing, Proceedings of the 1995 Conference on Supercomputing*. ACM, December 1995.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 2003.

[18] Roger W. Hockney and Chris R. Jesshope. *Parallel Computers 2: architecture, programming, and algorithms*. Adam Hilger, IOP Publishing Ltd, 1988.

[19] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *1991 International Conference on Supercomputing, Cologne, June 1991*, 1991.

[20] KHRONOS Group. OpenCL Computing Language v1.0, December 2008.

[21] S.D. Kim, C.J. Hyun, and M.H. Sunwoo. VSIP: Implementation of Video Specific Instruction-set Processor. In *APCCS: Asia Pacific Conference on Circuits and Systems*, pages 1075–1078, Singapore, December 2006. IEEE.

[22] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.0, May 2008.

[23] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Research Report 6962, INRIA, September 2009.

[24] R. M. Ramanathan et al. Extending the world's most popular processor architecture. White paper, Intel, 2006. *SSE3*.

[25] T. Rauber. Optimal evaluation of vector expression trees. In *JCIT: Proceedings of the fifth Jerusalem conference on Information technology*, pages 467–473, 1990.

[26] Pierre Soile. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag, 2003.

[27] THALES (TRT) and other institutions. TERAOPS Project, 2009.

[28] VSIPL Forum. VSIPL Std v1.3, January 2008. Vector Signal Image Processing Library.

[29] Michael Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley Longman, 1995.

[30] Julien Zory and Fabien Coelho. Using algebraic transformations to optimize expression evaluation in scientific codes. In *PACT: Parallel Architectures and Compilation Techniques*, pages 376–384, Paris, December 1998. IEEE.