Compiling for an Heterogeneous Vector Image Processor

Fabien Coelho, François Irigoin

Compiling for a Heterogeneous Vector Image Processor *

In Proceedings, Ninth Workshop on Optimizations for DSP and Embedded Systems (ODES-9) Chamonix, France, April 2011

Technical Report MINES ParisTech A/430/CRI

Fabien Coelho François Irigoin
CRI, Maths & Systems, MINES ParisTech, France
firstname.lastname@mines-paristech.fr

Abstract

We present a new compilation strategy, implemented at a small cost, to optimize image applications developed on top of a high level image processing library for an heterogeneous processor with a vector image processing accelerator. The library provides the semantics of the image computations. The pipelined structure of the accelerator allows to compute whole expressions with dozens of elementary image instructions, but is constrained as intermediate image values cannot be extracted. We adapted standard compilation techniques to perform this task 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 for the target. Experiments were run on the hardware functional simulator. We compile 1217 cases, from elementary tests to full applications. All are optimal but a few which are mostly within a mere accelerator call of optimality. 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) a specific heuristic to minimize execution time on the target vector accelerator; 3) numerous experiments that show the effectiveness of our strategy.

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. 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.

* This work is funded by the French ANR through the FREIA project [2].

We present a compilation strategy to map image processing applications developed on top of a high-level image library onto a heterogeneous processor with a vector image processing accelerator. This 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 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 dilatations, erosions, 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 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.

The application development relies on the FREIA image processing library API [2]. A software implementation on top of Fulguro [8], a portable open-source image processing library, is used for functional tests. The developer has no knowledge of the target accelerator hardware. Operators of the FREIA image library must be programmed specifically for the chosen target accelerator, either by simply calling basic hardware accelerated operators (basic hardware operator library implementation), or, better, with a specialized implementation (hardware optimized library implementation) that takes advantage of the hardware by composing basic operations. Although the library layer provides functional application portability over accelerators, it does not provide all the time, energy and cost performance expected from these pieces of 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 [1], 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 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, with no or few memory transfers for the intermediate values. To detect which calls to merge, techniques have been developed such as loop fusion or complex polyhedral transformations. 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. 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, such as the number of vector units, data paths, code size or local memory available, and split these expression DAGs into parts as large as possible, but meeting these constraints. Finally, using the expression DAGs as input, we generate the configuration code ands calls to a runtime library activating the accelerator, and replace the expressions by these calls.

The whole optimization strategy is automated and implemented in PIPS [4, 17], a source-to-source compiler, which let the user see the C source code that is generated. This greatly helps compiler debugging. We compile 1217 test cases, from elementary tests to full applications, all of which are optimal but a few. Experiments were run with the SPoC functional simulator. The results on the running example included in this paper show a speed-up of 16.5 over the most naïve use of the accelerator, and a speed-up of 3 over the use of the optimized library.

In the remainder of this paper, we first introduce our running example which is a short representative of the application domain (Section 2) and present the target architecture (Section 3). Then we show how the user source code is preprocessed to obtain basic blocks with optimization opportunities (Section 4). Next, compiler middle-end optimizations for locality are described (Section 5), and the back-end SPoC specific hardware configuration generation is detailed (Section 6). We finally present our implementation and experimental results obtained with a SPoC simulator (Section 7), and discuss the related work (Section 8).

2. Applications and Running Example

The FREIA project aims at mapping efficiently an image processing applications developed on top of a high-level API onto different hardware accelerators. The image applications use all kind of image processing operations, such as: `AND`-ing an image with a mask to select a subregion; `MAXLOC`-cating where is the hottest point; `THR`-esholding an image with values to select regions of interest; mathematical morphology (MM) [20] operators. The MM framework created in the 1960's provides a well-founded theory to image analysis, with algorithms described on top of basic image operators. The 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 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 targets. The project aims at reducing these costs through optimizing compilation and careful runtime designs. Typical applications extract informations 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), or whether there is some motion under a surveyance camera (VS, Figure 3).

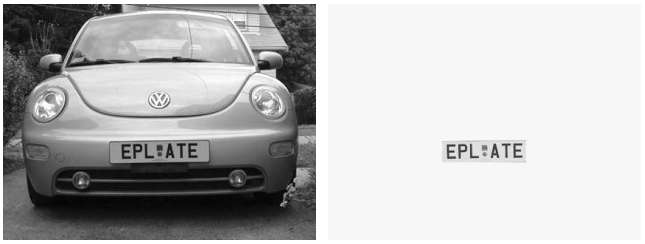

Figure 1. License plate *(LP)*: character extraction

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

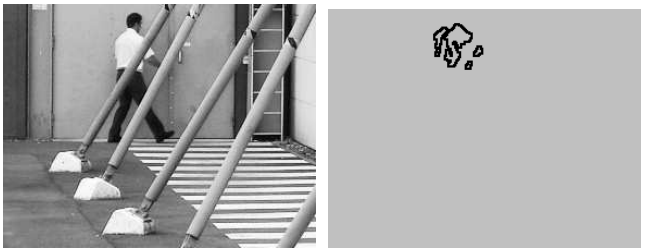

Figure 3. Video survey *(VS)*: motion detection

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 the applications. There are two implementations for the SPoC vector hardware accelerator (Section 3), 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 on going versions of the library are optimized for the Terapix [5] SIMD accelerator, and for OpenCL targeting graphics hardware (GPGPU).

The code in Figure 4 was defined as part of the FREIA project to provide a short test case significant both for the difficulties involved and for the optimization potential, with the two hardware accelerators in mind. The test case contains all the steps of a typi-

```c
#include <stdio.h>
#include <freia.h>

int main(void) {
  freia_dataio fin, fout;
  freia_data2d *in, *og, *od;
  int32_t min, vol;
  // initializations
  freia_common_open_input(&fin, 0);
  freia_common_open_output(&fout, 0, ...)
  in = freia_common_create_data(fin.bpp, ...);
  od = freia_common_create_data(fin.bpp, ...);
  og = freia_common_create_data(fin.bpp, ...);
  // get input image
  freia_common_rx_image(in, &fin);
  // perform some computations
  freia_global_min(in, &min);
  freia_global_vol(in, &vol);
  freia_dilate(od, in, 8, 10);
  freia_gradient(og, in, 8, 10);
  // output results
  printf("input global min = %d\n", min);
  printf("input global volume = %d\n", vol);
  freia_common_tx_image(od, &fout);
  freia_common_tx_image(og, &fout);
  // cleanup
  freia_common_destruct_data(in);
  freia_common_destruct_data(od);
  freia_common_destruct_data(og);
  freia_common_close_input(&fin);
  freia_common_close_output(&fout);
  return 0;
}
```

Figure 4. FREIA API running example

cal 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 a paper, we use it as running example, together with extracts from larger applications. Optimization opportunities at the `main` level of our test case are very limited. The `min` and `vol` function calls correspond to two 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 SPoC instruction set. They are implemented in the non-optimized SPoC version of the FREIA library, using calls to elementary functions. Since these calls are not visible in the main function, no optimization is possible in this case. With the naïve elementary function based implementation, 33 calls to the accelerator are used per frame, hidden in the callees. A hand-optimized SPoC implementation of the FREIA image library results in 6 accelerator calls only, because calls to elementary functions can be merged within the implementation of the FREIA functions.

3. SPoC Architecture

Figure 5 outlines the structure of the SPoC processor. It can be seen as a simplified version of the 30 year old CDC Cyber 205 [16], 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.

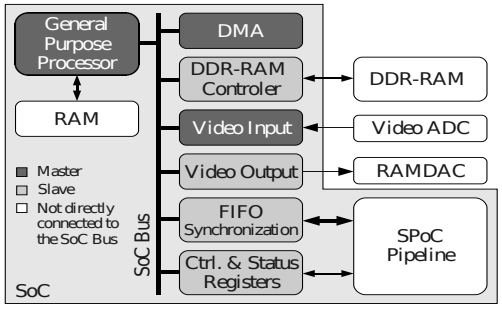

Figure 5. SPoC architecture

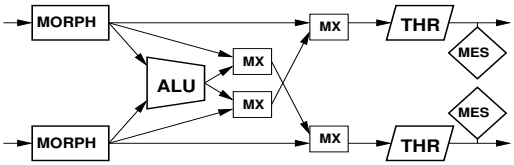

Figure 6. One SPoC vector unit, to be chained

Figure 6 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 apparent 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* compute reductions such as maximal 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` 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 76800 element 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 erosions and dilatations on few images 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 depth 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 image boundaries which 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 image size. This is another reason why low-level loop transformation-based approaches are likely to fail. Micro-instruction scheduling and compaction is easy once the order of operations is determined.

To sum up, the useful 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 microinstructions, 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.

4. Phase 1 – Application Preprocessing

The FREIA API [2] and its Fulguro [8] implementation are designed to be general with respect to the connectivity, the image sizes and the pixel representation. Standard or advanced loop transformations cannot take advantage of such source code because the loops are distributed into different functions and because elementary array accesses are hidden into function calls to preserve the abstraction over the pixel structure.

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 such as the image dilatation. Several source-to-source transformations help achieve this goal: 1) inlining to suppress functional boundaries, 2) partial evaluation to reduce the control complexity and 3) constant propagation to allow full loop unrolling, 4) dead code elimination to remove useless control, 5) 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 five transformations is chosen to maximize the available information so as to simplify the code and obtain larger basic blocks. Figure 7 shows the resulting code after automatic application of these transformations on the `main` function in Figure 4. It contains a sequence of elementary image operators mixed with scalar operations and temporary image allocations and deallocations.

5. Phase 2 – DAG Optimization

The basic blocks of the image application are analyzed to build an expression DAG as the one in Figure 8 (on next page), which is then optimized for locality. The vertices are the operations to perform, which may be image operations (*MORPH* 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

```
// perform some computations
freia_aipo_global_min(in, &min);
freia_aipo_global_vol(in, &vol);
freia_aipo_dilate_8c(od, in, k8c);
freia_aipo_dilate_8c(od, od, k8c);
// previous line repeated 10 times...
I_0 = 0;
tmp = freia_common_create_data(...);
freia_aipo_dilate_8c(tmp, in, k8c);
freia_aipo_dilate_8c(tmp, tmp, k8c);
// previous line repeated 10 times...
freia_aipo_erode_8c(og, in, k8c);
freia_aipo_erode_8c(og, og, k8c);
// previous line repeated 10 times...
freia_aipo_sub(og, tmp, og);
freia_common_destruct_data(tmp);
```

Figure 7. Excerpt of the `main` of Figure 4 after preprocessing

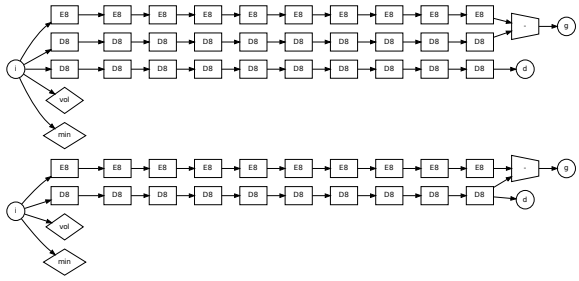

Figure 9. Initial and optimized expression DAG for Figure 4

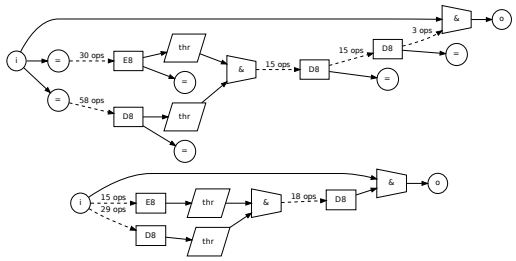

Figure 10. Extract of initial and optimized DAG for *License Plate*

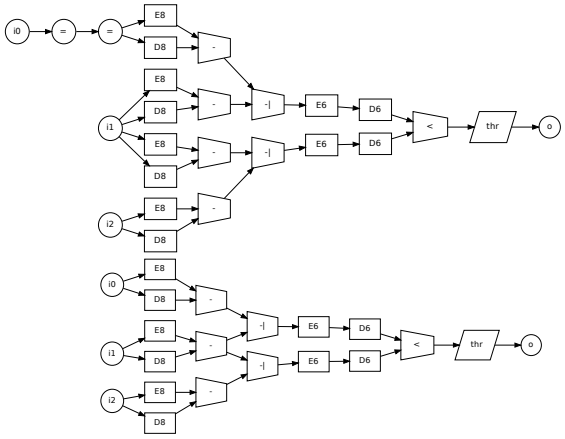

Figure 11. Initial and optimized DAG from application *OOP*

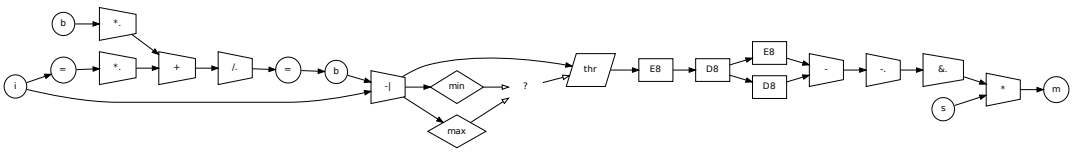

Figure 8. DAG example from *Video Survey* – the left-end updates background *b*, the remainder detects movements in hot regions

end is used at the sink node. Arcs shown as black arrows embed image dependencies, and white arrows represent 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 upper part of Figure 9. The third row of dilatations is the call to the `freia_dilate` function with connectivity 8 and size 10. This DAG is then optimized in a target independent manner, with standard compilation techniques: common sub-expressions elimination and copy propagation are applied 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. 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. 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 lower part of Figure 9, and the intermediate result is simply extracted.

Other challenges are found in the DAG for the *license plate* application in Figure 10, 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. Eliminating such diversions is important for our target vector accelerator because extractions break the computation pipeline, thus inducing more accelerator calls. The DAG in Figure 11 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 optimization of the DAG in Figure 8 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. Phase 3 – SPoC Hardware Configuration

Finally, an accelerator specific compilation phase generates the hardware configuration, i.e. the micro-instructions for evaluating the optimized DAG resulting from the previous phase. We have two code generators; one for SPoC described hereafter, and one under development for the Terapix SIMD accelerator.

Problem Description

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 6). 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, the 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 of they occur simultaneously. So although mapping scalar expression DAG onto a register machine [3, 6] 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.

Code Generation

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 to avoid a global combinatorial optimization because this simple heuristic, which satisfies each constraint one after the other, leads to excellent experimental results (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*, with a prioritized topological order which focuses on the critical resource, namely the small number of data paths. Scalar dependencies, 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 opera-

tions 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 microinstructions. 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. Microinstruction compaction is performed at the same time because the packing constraints are very easy to meet: structural, control and data pipeline hazards are avoided by the hardware, hence sophisticated microinstruction 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 improve 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 stage of the process is trivially optimal if the compaction is optimal.

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.

7. Discussion, implementation and experiments

There is a cost performance tradeoff in choosing the number of vector units, as longer pipeline 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.

Our optimization strategy is implemented in PIPS [10], for a small development cost measured hereafter with the KLOCs *(line of codes)* involved. Figure 12 summarizes the different phases presented in detail in the previous sections. Transformations of Phase 1 are standard in an advanced optimizing compiler. 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 uses about 2 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 1.6 KLOCs including DAG splitting, scheduling, wiring, and SPoC configuration. It produces acceleration functions to be called from the initial application. Each generated pipeline configuration function (see excerpt in Figure 13) is called from the `main` with the appropriate arguments (in Figure 14).

Phase 1 application preprocessing – enlarge basic blocs

1. inlining of FREIA library functions
2. partial evaluation
3. constant propagation and loop full unrolling
4. dead code elimination
5. block flattening

Phase 2 DAG optimization

1. DAG construction per sequence
2. common sub-expression elimination, with commutativity
3. forward or backward copy propagation
4. extraction of remaining copies
5. dead image operation removal

Phase 3 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

Figure 12. Outline of our compilation strategy: phases and stages

```
void helper_0(freia_data2d * o0, freia_data2d * o1,
  freia_data2d * i0, int32_t * red0, int32_t * red1,
  int32_t * kern2 /* ... up to kern16 */)
{
  // SKIPPED: declarations & initializations
  // - si & op: micro instructions
  // - sp & par: operation parameters
  // - redres & reduc: reduction results

  // set state of MUX stage 0 number 0
  si.mux[0][0].op = SPOC_MUX_IN0;
  // set state of POC stage 1 side 0
  si.poc[1][0].op = SPOC_POC_DILATE;
  si.poc[1][0].grid = SPOC_POC_8_CONNEX;
  // and its kernel
  for(i=0 ; i<9 ; i++)
    sp.poc[1][0].kernel[i] = kern2[i];
  // SKIPPED: more configurations...

  // actual call to the hardware accelerator
  // instructions, params, 2 images out, 2 images in
  freia_cg_process_2i_2o(op, par, o0, o1, i0, i0);
  // extract reductions results
  freia_cg_read_reduction_results(&redres);
  *red0 = (int32_t) reduc.measure[0][0].minimum;
  *red1 = (int32_t) reduc.measure[0][0].volume;
}
```

Figure 13. One stub source code (excerpt) for Figure 4

```
// perform some computations
helper_0(od, og, in, &min, &vol, k8c, /* 18 more k8c args */);
helper_1(od, og, od, og, k8c, k8c, k8c, k8c, k8c);
```

Figure 14. Code in Figure 4 is reduced to two stub calls

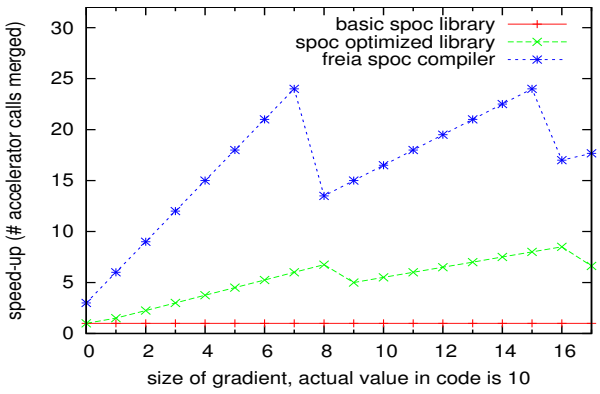

Figure 15. Speed-ups on SPoC with 8 vector units for Figure 4

Other applications may require more preprocessing phases, such as while loop unrolling or code hoisting, to obtain longer basic blocks.

Figure 15 compares speed-ups obtained on versions the running example Figure 2 with differing numbers of iterations (the source code selected executes 10 iterations). 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 optimized the whole application. It fares better than any other versions, thanks to the extracted common subexpression 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. Our compiler was tested on 1217 cases, comprising 1005 combinatorial tests (3 to 6 ops), 105 elementary tests (1 to 13 ops), 40 atomic tests (1 op for which we generate the hardware accelerated version) and finally 57 significant applications or functional blocks (5 to 135 ops) tested with various parameters. Only 15 results are not optimal for the target SPoC accelerator: 14 are one call from optimality, and one is non optimal by 3 calls. Most of these non optimality cases are linked to the greedy nature of the heuristic coupled with pipeline spilling effects.

8. Related Work

The related work is rather limited because people developing hardware accelerators in an academic environment usually do not have the resources required to develop a full programming tool chain. They either design a specialized language, or use pragmas to guide the optimizer, or build an optimized library, which may grow with each new application to include the application-specific API that leads to good performance, or they 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 recent standard aims at providing portability across accelerators, especially GPGPU targets, but it is quite large and pretty low-level: application developers should be able to ignore it. 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 [13] is an example of a domain-specific language designed for signal processing and for accelerator programming. On the one hand, Array-OL is not general enough to write a whole application, and on the other hand it 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 a standard language are used to preserve the portability of applications and allow their functional validation in a standard environment. OpenMP 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 [1] 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 that an API has a direct impact on the application structure, which may not lead to good 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. 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 [15] 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 [18] 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 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 [21], 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 obtained by different code transformations, including code hoisting and code sinking [14]. 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 [19], 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 a domain-specific hardware accelerator composed of multiple chained vector units. Applications can be developed in C by any programmer competent in the image processing field, but without knowledge of the hardware accelerator, 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 speedups.

Some experimental results are even better than expected. The PIPS automatically optimized version of the running example beats the hardware expert first-cut hand-optimized version, because common sub-expression elimination opportunities were not considered. It is up to three times faster than the version based on the hand optimized FREIA library implementation, and it is optimal, as most of our 1217 test cases. The compiler also generates, as a side effect, the basic hardware accelerated library version by considering elementary operations as a whole application. The full library hardware accelerated version is more subtle, as it dynamically adapts the generated configuration to the parametric number of iterated operations (see the dilatation) and the available hardware pipeline depth. These are known to the compiler when considering a full applications in context, but not when simply looking at a function library implementation.

What are the underlying reason of our success? Firstly, the application domain uses one large type of data, images, and a limited set of operators executed on whole images, with a lot of implicit locality and parallelism. Secondly, the architectural choices of the hardware with the high level instruction set provided by SPoC [9] takes advantage of these opportunities to provide a potentially high performance pipeline, which, although not as convenient as a crossbar which would enable any operator chaining, fits the kind of DAG found in applications, and is accessible through runtime calls which handle low-level details but enable all necessary configurations. Thirdly, the library API is reasonably small (about 40 basic operations and about 20 higher level combined operations), and 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. Thus the 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.

This does not preclude the implementation of the same approach on more traditional SIMD hardware accelerators or GPGPU targets, because the high level API provides all the semantical information needed to generate code and perform many classical compiler optimizations. However it may need to be combined with more traditional loop transformation techniques to produce optimized combined operation microcode for these targets, or to develop a specific runtime which takes advantage of the available hardware once high-level optimizations and choices are performed. Such work is already underway. A second direction is to test our approach on more real-life applications. We also have to look at the impact on our strategy on domains with multiple data types, such as signal processing applications. A third direction is to reuse the semantical loop fusion and emulate its the schedule of our target to benefit from the locality increase for general-purpose processors. The technique used by our accelerator to handle image boundaries by maintaining a constant image size could also be useful.

Acknowledgments

We are indebted to Christophe Clienti who designed the SPoC accelerator, its simulator, the Fulguro library and the FREIA interface, and who ported existing 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 application codes. Finally, we want to thank Alain Darte for a long discussion about NPC problems.

References

[1] VSIPL Std v1.3, Jan. 2008. Vector Signal Image Processing Library.

[2] FREIA: FRamework for Embedded Image Applications, 2008–2011. URL `freia.enstb.org`. French ANR-funded project.

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

[4] M. Amini, C. Ancourt, F. Coelho, F. Irigoin, P. Jouvelot, R. Keryell, P. Villalon, B. Creusillet, and S. Guelton. PIPS Is not (just) Polyhedral Software. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[5] P. Bonnot, F. Lemonnier, G. Edelin, G. Gaillat, O. Ruch, and P. Gauget. Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA. In *Design Automation and Test in Europe*, pages 610–615. IEEE, Dec. 2008.

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

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

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

[9] C. Clienti, S. Beucher, and M. Bilodeau. A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. In *EUSIPCO: European Signal Processing Conference*, Aug. 2008.

[10] CRI, MINES ParisTech. PIPS, 1989–2011. URL `pips4u.org`. Open source, under GPLv3.

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. 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. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981. ISSN 0018-9340.

[13] C. Glitia, P. Dumont, and P. Boulet. Array-ol with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 23(3), 2009.

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

[15] M. H. Hall, , S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. 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, Dec. 1995.

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

[17] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *1991 International Conference on Supercomputing, Cologne, June 1991*, 1991.

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

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

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

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