A case study: optimizing GCC on ARM for performance of `libevas` rasterization library

Dmitry Melnik[1], Andrey Belevantsev[1], Dmitry Plotnikov[1], and Semun Lee[2]

[1] Institute for System Programming, Russian Academy of Sciences
{dm,abel,dplotnikov}@ispras.ru
[2] Samsung Corp.
semun.lee@samsung.com

Abstract. This paper reports on the work for optimizing GCC on ARM to improve performance of `libevas` rasterization library. We used manual profiling and analysis as well as ACOVEA [3] compiler options tuning tool to identify weak places and tune GCC optimization parameters. We identified a number of deficiencies in GCC optimizations with `libevas` on ARM, including GCSE, register allocation, autovectorization and loop prefetching, and proposed solutions to them, that altogether brought 15.78% average performance increase, and with up to 119% increase on certain tests, as measured with customized `expedite` benchmark. These results show that tuning existing GCC optimizations for specific platform and application may provide significant performance boost, comparable to that of developing a new compiler optimization.

1 Introduction

GCC is designed to be a multiplatform compiler. It also contains dozens of optimization passes that are parameterized in such a way that any target platform code can influence their decisions. Given the complexity of GCC, usually there is a room for improvement if you need to tune GCC for one particular target, while not paying attention to performance on other targets. The improvement may come from several sources. First, since most of the tuning happens for `x86` and `x86-64` architectures, there may be code generation deficiencies for a less popular target. A fix for them usually requires adding new instruction patterns or peephole optimizations to the backend, or adjusting target-dependent costs. For example, GCC inlining pass has internal constants specifying costs of function calls, prefetching pass has parameters that control cache metrics, etc. Second, machine-independent optimizations may not be tuned for a given target, meaning that their default behavior should be changed. For example, register allocator and/or inlining may use different limits on embedded targets than on general targets. Finally, a new target-independent feature could be implemented to take into account certain specific features of the target. For example, to make advantage of speculation feature of Intel Itanium, we have implemented its support in the instruction scheduler. This kind of improvements may take much time and resources.

In this work, our primary optimization target was ARM Cortex-A8 [1] architecture, including its NEON vector unit. In the paper we show a number of deficiencies that we found in GCC optimizations, and that are specific either to Cortex-A8 architecture or ARM platform in general. These optimizations include GCSE, prefetching, and autovectorization for NEON. We have proposed solutions to the problems found which altogether brought 15.78% average performance increase, with up to 119% gain on certain tests (as measured with customized `expedite` benchmark described in Section 2). We also show that using automatic compiler option tuning tools like ACOVEA may facilitate identification of those optimizations that need improvement as well as determining optimal optimization parameter values.

Rest of the paper is organized as follows. In Section 2 we give overview on `libevas` rasterization library, `expedite` benchmark test suite, ACOVEA tool and the environment we have used. In Section 3 we describe GCC optimizations we have analyzed and the improvements we made to them. In Section 4 we present the performance results of our optimizations and tuning. Section 6 outlines areas for the future work, and Section 6 concludes.

2 Test Application and Environment

As our primary performance target for GCC optimizations we have chosen `libevas`. It is a part of EFL (Enlightment Foundation Libraries) [2], which are base libraries for `E` window manager as well as other applications. This multiplatform library contains various routines for fast rasterization and processing image data, such as blending, scaling, clipping images, drawing polygons, etc. To measure `libevas` performance, we used a modified version of `expedite` benchmark suite, available in EFL repository. It was customized to improve results precision and speed, so it can provide results with variance less than 0.5% in 1-2 minutes test run time. The improvements to precision include adding few iterations to each test that are executed before time measurement is started so to exclude time required to fill cache from the result; a median filter is applied to evaluate final *fps* value for each benchmark among several runs. Better precision allowed us to significantly decrease the minimal number of iterations required for each test to obtain stable results. Also we have excluded from the original test set benchmarks with similar profiles and those with volatile results. To compute composite benchmark result value geometric mean is used. These improvements altogether allowed us to use this benchmark suite with automatic tuning tools as a complement to manual tuning.

To aid manual tuning, we used ACOVEA [3] automatic tuning tool. It aims to find a combination of options and parameters that provide best performance on a given benchmark using genetic algorithm [4, 5]. We have adapted ACOVEA to work in a cross environment. The changes include added capability to cross-compile benchmarks on `x86` machine, transfer binaries to `ARM` testboard, execute them, and transfer the resulting *fps* value back to `x86` host. Then, ACOVEA core cross-breeds GCC options from different runs according to performance

these options provide: the better *fps* the certain options combination achieves, the greater chance it has to reproduce.

Initially we ran ACOVEA tuning with a set of GCC flags that are enabled by default by `-O3` optimization level, plus `-fprefetch-loop-arrays` flag. It took about 4 days to complete with the following ACOVEA parameters: 20 generations, 3 populations, and 60 species in a population. This tuning run have shown that the greatest effect on performance has `-fno-gcse` option, which disables Global Common Subexpression (GCSE) optimization. This positive effect was observed on 44 out of 45 tests, and it didn't depend from other options specified along with `-fno-gcse`, or an input data. We give analysis of the GCSE optimization problem in Section 3.1. Other option combinations found by ACOVEA we have analyzed didn't tend to show consistent performance improvement, e.g. simply enabling `-fprefetch-loop-arrays` without tuning its parameters affected tests controversially, improving some by up to 10-15% while slowing down others as much as 25%.

Since there are more than 100 numeric parameters in GCC, each with its own integer value range, it is impractical to tune them all at once, so we selected few optimizations for detailed tuning of their parameters with ACOVEA. These optimizations are inlining, loop unrolling, register allocator and loop prefetching. We discuss results of this tuning in Sections 3.1–3.4, dedicated to corresponding optimizations.

In our work we used GCC 4.4.1 release branch as the base compiler.

3 GCC optimizations

In this section we discuss problems found in GCC optimizations and propose solutions for them.

3.1 GCSE

We have analyzed assembly code of `libevas` and identified a common deficiency in the way GCC deals with long immediate constants on ARM. On ARM, due to architecture constraints, a constant can be used as an immediate instruction operand if and only if it can be represented in the form `CONST_32 = CONST_8 << (2 * N)`, where `CONST_8` is an 8-bit constant, and $0 \leq N < 16$. If a constant doesn't comply with this constraint, the instruction can't use it as an immediate value, but should either preload it into a register or split original operation. Figure 1 shows an example of such constant splitting. Let's say we need the 2nd and 4th byte components from 4-byte integer variable b (this access pattern is very common in `libevas` blend routines). The C code to access appropriate components with a bit mask (at the left) is translated into two `bic` instructions (at the right). Assuming that in original application these instructions are located inside loop, in this case better solution would be store this constant into a register outside a loop and then use just one `and` instruction with that register

```
int a, b;
a = b & 0x00ff00ff;
```

(a) C code

```
ldr r3, [fp, #-8]       ; load b
bic r3, r3, #-16777216  ; r3 = r3 & ~0xff000000
bic r3, r3, #65280      ; r3 = r3 & ~0x0000ff00
str r3, [fp, #-12]      ; store a
```

(b) ARM assembly code generated by GCSE pass

Fig. 1. Splitting long ARM constants

as an operand instead of two `bic` instructions with immediate constants inside the loop.

We found that the main reason for generating such inefficient code is that during global common subexpression elimination (GCSE [6, 7]) optimization pass GCC doesn't consider ARM architecture specifics regarding immediate value representation in instruction code, assuming that constant propagation is always profitable, which is not true when propagating constants inside loops on ARM. There are three GCC passes that are involved in this problem: `pass_gcse`, `pass_rtl_move_loop_invariants`, `pass_split_all_insns` (by default executed in this order). At GCSE pass, two instructions `reg1 = const` and `reg2 = reg2 & reg1` are merged into `reg2 = reg2 & const`. At this stage, the compiler doesn't know whether `const` is a valid immediate constant for ARM or it needs splitting into two separate instructions and proceeds with merge anyway. Then, at `move_loop_invariants` pass, it wouldn't have any invariant to move, since at this point it is already an immediate value in instruction. Then, `split_all_insns` is run, which adds an extra instruction into the loop body. Changing the order of passes (doing loop invariant motion after split) doesn't help since after split it isn't a loop invariant any more because of data dependencies.

The problem can be worked around by disabling GCSE completely, but this isn't an appropriate solution, since in this case optimization opportunities can be missed. So we developed a solution for "conservative" GCSE, which takes into account ARM immediate value representation specifics. In order for loop invariant to work, we moved loop invariant code motion pass before GCSE pass, where all constants that could be moved outside of a loop body still reside in separate pseudo-registers. Our tests have shown that such pass order change doesn't affect the performance of `expedite` test suite. Loop invariant code motion pass has its own heuristics that estimate register pressure and doesn't allow moving invariant if it will likely result in a register spill. After loop invariants have been moved, our restricted GCSE will only allow to move "short" constants (those which don't require several operations to load) if moved into the loop body from the outside, and will allow GCSE to proceed in its usual way on the same loop hierarchy level. More strictly, at GCSE pass, we deny the transformation if the following two conditions are met:

1. The expression moved is a "long" 32-bit ARM constant, i.e. the constant doesn't fit into 12-bit immediate value (8-bit number and 4-bit shift values);

2. The expression is moved to destination block with a deeper loop hierarchy level than the source block, e.g. from an outer loop into a nested loop body.

Our patch includes two options to control GCSE behaviour on ARM. First option, `-farm-fix-gcse` checks just the first of above conditions, only denying transformation for long constants, and retaining original pass order; second, `-farm-fix-gcse-loop-hierarchy`, checks both conditions before allowing GCSE transformation and swaps GCSE and invariant code motion passes.

Restricting GCSE and letting loop invariant code motion pass to do its job increases performance of `libevas` on average by 5.5%. Though it isn't better than simple `-fno-gcse` for this application, we believe that it's the right way to address the problem and that there are applications that benefit from this approach. While testing our optimization on Aburto's benchmark suite, we found that it brings performance gain up to 10% on several tests without any significant regressions on others, compared to completely disabling GCSE which actually causes performance loss on this test suite. For example, on `hanoi` benchmark, completely disabling GCSE causes "short" constant 1 to be put into separate register, which results in additional register save and restore instructions in function prologue and epilogue. Since the subject function is recursive, these excessive instructions result in performance degradation by 10%, which is fixed by our patch that lets GCSE to propagate this "short" constant. We still need to test this optimization on more applications to make sure loop invariant code motion heuristics can handle well an increased number of loop invariants, so it doesn't cause performance regressions in loops with high register pressure.

3.2 The Register Allocator

Another problem we have found is excessive memory loads generated inside loops. Figure 2(a) shows the original code generated by GCC for a simple loop from `evas_common_scale_rgba_in_to_out_clip_smooth_c`. Both load instructions could be placed outside the loop, if there were enough free hardware registers available.

The reason for these excessive loads in the loop is that the cost of corresponding pseudo registers was calculated using basic block frequency via integer math, and truncating rounding caused sub-optimal code generation. We have tried a patch to use rounding to nearest integer in the register allocator, and it fixed the test case, but it did not provide performance improvements. We believe that due to the *NP*-complete nature of register allocation problem [6] this case represents just the bad case for register allocator heuristics, and in general it can be possibly improved by providing better estimation for basic block frequencies, which means using profiling information. Indeed, we have confirmed that when the compiler has precise execution counts from profiling, it generates exactly the same code for the problem loop either with or without the fix.

We have also tuned with ACOVEA the following GCC register allocator [9] options and parameters: `-fira-coalesce`, `-fira-algorithm`, `-fira-region`, `-fno-ira-share-spill-slots`, `-fno-ira-share-save-slots`, and `ira-max-`

```
.L133:
ldr lr, [fp, #-84]
mov r3, r1, asr #16
add r1, r1, r0
str r3, [lr, r2, asl #2]
ldr r3, [fp, #24]
add r2, r2, #1
cmp r3, r2
bgt .L133
```

(a) original code

```
.L133:

mov r3, r1, asr #16
str r3, [lr, r2, asl #2]
add r2, r2, #1

cmp r9, r2
add r1, r1, r0
bgt .L133
```

(b) code after a fix for register frequency rounding was applied

Fig. 2. Removing excessive invariant loads inside loops

`loops-num`. There were different option combinations found, that showed up to 6.5% gain on certain tests, while causing sometimes even bigger regression on others. Though only one option, `-fira-coalesce` seemed to improve performance consistently for the majority of the tests, giving 1-1.5% average gain. However, after we enabled `-fprefetch-loop-arrays` option later and tuned its parameters, the positive effect from `-fira-coalesce` was not longer reproduced. This shows that option tuning should involve all GCC options of interest at once, since optimizations tend to influence each other. At the same time, increased optimization search space may result in too much time to complete the tuning to make this approach practical.

3.3 Function Inlining

While tuning GCC inlining with ACOVEA, we found that `libevas` doesn't respond much to tuning its parameters, so we examined its source code to find whether there is a potential for this optimizations or it's just a problem with automatic tuning strategy that can't find the right parameters.

In `libevas` most CPU cycles are spent in tight loops performing rasterization. These loops are manually optimized by EFL developers, so this application has little inlinable calls that may affect the performance. We found that out of 19 EFL hottest functions that are invoked by `expedite` test suite 11 don't have calls at all, 4 have indirect calls through pointers, 1 is just a stub for `memcpy`, so function calls that can be inlined by GCC present only in 3 of these functions, which performance impact is minor. This way, the compiler options and parameters controlling inlining don't affect significantly the performance of `expedite` test suite, as we have found with automatic tuning, so `libevas` just might be not the right candidate to tune these optimizations. A good candidate for such study might be an application written in C++ that contains many small class member functions.

3.4 Loop Unrolling and Prefetching

As a part of original `libevas` hand-optimization, most critical loops are unrolled using custom `UNROLL8_PLD_WHILE` macro, which duplicates given loop body 8 times. This leaves compiler with little options for loop unrolling optimization: further unrolling such pre-unrolled loops usually doesn't yield any additional improvement. That's why automatic tuning of the RTL unroller parameters, similarly to inlining, didn't show significant improvement.

Modern ARM architectures have a *prefetching* feature, which allows preloading values from memory into L2 cache. This mechanism is controlled explicitly by a programmer or a compiler by issuing `pld` instruction, which hints CPU that data referenced by its argument soon will be needed, so CPU may start fetching it into its cache.

The abovementioned `UNROLL8_PLD_WHILE` macro, besides performing unrolling, inserts one `pld` prefetch instruction per unrolled loop body, assuming that cache line size equals to 32 bytes, and prefetching next cache line ahead of one iteration. Though this configuration shows +2.5% performance increase on `ARMv6`, it was found to be not optimal for `Cortex-A8`. Technical documentation for this architecture specifies L2-cache line size equal to 64 bytes, so each `pld` instruction generated with the macro hits the same cache line twice on `Cortex-A8`. Also, prefetching just 32 bytes ahead may be too little to allow complete loading next cache line before a new loop iteration begins. On the other hand, if the unrolled loop iterates just 4 times, long prefetching distance would be fetching values that will never be used.

We tried different prefetching parameters (distances in range from 32 to 320) and unrolling factors (from 2 to 16) and found that the best performance with this macro on `Cortex-A8` is achieved with no prefetching instruction at all and with unrolling factor equal to 4. These changes together yield increase of `libevas` performance by 6.5%, and the result can be evenly attributed to removing prefetching and changing unroll factor from 8 to 4. These results partially can be explained by the results of value profiling of `UNROLL8_PLD_WHILE` parameter `size`: about 20% of executed loops that are unrolled using this macro iterate just 4 times, and for about half of these loops the number of iterations doesn't exceed 16.

Prefetching is an optimization feature that is hard to implement properly, if it's done manually at the source code level, especially when it comes to tuning for different architectures at the same time. To benefit from this optimization, hardware cache specification should be taken into the account, such as L1 and L2 cache sizes, cache line size, the number of memory operations that can be processed simultaneously, and a latency of loading data from main memory into cache. GCC has a prefetching optimization (`-fprefetch-loop-arrays`) combined with loop unrolling, which takes these parameters into account in effort to generate optimal prefetching code.

GCC ARM backend doesn't override hardware cache parameters, so with this optimization common default GCC values are used, which were never tuned specifically for this architecture. However, as we found with `libevas` tuning,

these parameters should be set distinctly even among different ARM architectures. Not only these architectures have different latencies, cache sizes and limits on number of parallelly executed loads, but also `prefetch-latency` parameter, measured in number of instructions executed before prefetch operation is completed, has different meaning depending on instruction latencies and an issue rate (e.g. `Cortex-A8` is a dual-issue, while `ARMv6` is single-issue architecture).

We tried to specify cache parameters found in ARM technical specification (`l1-cache-line-size=64`, `l1-cache-size=32`, `l2-cache-line-size=256`) as well as tuning them with ACOVEA. Specifying correct parameters from documentation does improve the performance, but with automatic tuning we have found other parameter sets that slightly differ from those in technical specs but give even greater improvement. Here are two best parameter strings found by ACOVEA, each of them is beneficial for distinct subset of benchmarks:

1. `l2-cache-size=256 l1-cache-size=16 simultaneous-prefetches=8 prefetch-latency=200 l1-cache-line-size=32`
2. `l2-cache-size=512 l1-cache-size=64 simultaneous-prefetches=6 prefetch-latency=400 l1-cache-line-size=64`

The second parameter set provides slightly better overall performance, so in final results table we use the latter. These two parameters have one common property: the `simultaneous-prefetches` parameter is set far above GCC default value of 3.

Some parameters found by ACOVEA differ from those in hardware specification, e.g. the best value found for `l1-cache-line-size` is 32, though ARM documentation specifies line size equal to 64. Smaller cache line value causes prefetch optimization to choose smaller unrolling factor (since it tries to issue one prefetching instruction per unrolled loop body). So the fact that ACOVEA has found cache line size to be less than that in specification shows that for some tests smaller unrolling factor is better than not have an extra preload instruction, which hits the same cache line twice.

Also, it can be noted that prefetching optimization with parameters properly adjusted is overall 4% better than without prefetching and with just 4 regressions in range 2.73%, opposed to default prefetching parameters which yield 9 regressions in that range and two 13% regressions, while gaining just 2% on average. Prefetching with parameters properly tuned speeds up certain tests by as much as 20%.

3.5 Autovectorization for NEON

GCC features autovectorization for many SIMD architectures [8], including NEON vfpu that is available in Cortex-A8. We studied how well this feature works with `libevas` code. After enabling autovectorizer (`-ftree-vectorize -mfpu=neon -mfloat-abi=softfp`), we were surprized to observe performance regression. Since the target application spends most of its runtime in tight rasterization loops, supposedly it should have respond well to vectorization.

```
int main() {
  int a[256], b[256];
  int i;

  for (i = 0; i<256; i++) {
    a[i] = b[i] >> 8;
  }
}
```

(a) original code

```
.L2:
  add     r2, r0, r3
  fldd    d16, [r2, #0]
  vmov.32 r2, d16[0]
  vmov.32 r1, d16[1]
  mov     r2, r2, asr #8
  str     r2, [r5, r3]
  add     r2, r5, r3
  add     r3, r3, #8
  mov     r1, r1, asr #8
  cmp     r3, #1024
  str     r1, [r2, #4]
  bne     .L2
```

(b) Original assembly code generated by GCC

```
.L2:
  add     r2, r5, r3
  add     r1, r0, r3
  add     r3, r3, #8
  cmp     r3, #1024
  fldd    d16, [r1, #0]
  vshl.s32  d16, d16, d17
  fstd    d16, [r2, #0]
  bne     .L2
```

(c) Assembly code after NEON backend was fixed

Fig. 3. Autovectorization of shift operation on NEON

First, we have analyzed why so few loops (just about 25%) were vectorized automatically by GCC. Most common causes of autovectorizer's failure were the following: function calls within the loop body (mostly indirect calls, so they can not be inlined), switch operator within a loop, and unsupported operations (e.g. there is no support for vector division on NEON). It's worth to note that switch operators within loops in `libevas` are used to specialize two cases for transparency values 0 and 1, so a multiplication by alpha-channel value could be replaced with simple copy of either source or destination color value. Though such specialization prevents the loop from being vectorized for NEON, pure ARM specialized code still significantly outperforms autovectorized NEON code on `expedite` tests.

We have found a problem with autovectorization of shift operations for NEON. If a loop being autovectorized contains shift operations (`>>`), autovectorizer is not able to find appropriate vector shift operation, so loop is vectorized partially: for all the rest operations (except shifts) vector instructions are generated, but for shifts data is moved first from NEON vector registers to ordinary ARM 32-bit ones, then ARM shifts for each vector component are issued, and finally data is moved back to NEON registers to store the vector into memory. Such transfers from NEON to ARM core and back cause severe performance degradation of the affected loops. Figure 3 gives an example of such poor loop auto-vectorization.

The cause for this problem was a bug in ARM NEON backend, which assigned shift operations to wrong operation table. Fixing this issue improved overall `expedite` performance by 8.82%, while certain tests ("Rect Blend" family), which suffered the most from poor shifts vectorization, grow as much as by 171%. There are few regressions, but most of them are within an error margin.

Also, we have found that specifying `-mvectorize-with-neon-quad` option gives slightly better overall results (about 1%) than default double-integer vectorization.

4 Experimental results

The performance results on reduced `expedite` test suite (as described in Section 2) are presented in Table 1. These results were obtained on `EBV Beagle` board with `vga` profile and using linux framebuffer. All the values presented are medians among 3 runs of the whole test suite. Due to space constraints, we omit those tests which performed similarly across all optimizations.

We reference each column with corresponding number. In the first row we specify the options used for benchmarks, or reference with square brackets another column where these options can be found, and specify just those options in which they differ from the referenced column, e.g. *"[A2] no prefetch, unroll=4"* means *"the same compile parameters as for A2, but with manual prefetching in macro turned off and unrolling factor set to 4"*.

The first column of Table 1 (`A1`) gives numbers for base GCC optimization level, `-O2`. We have chosen `-O2` as the base, since our tests have shown that `-O2` outperforms `-O3` on `expedite` by 0.5-1%. The second column, `A2`, shows results for base optimization level with GCSE turned off. It can be seen that this option is good for almost every test, since in `libevas` long constants, whose performance is highly affected by GCSE, are widely used in color component masks (like `0xFF00FF00`). The next column, `A3` shows our efforts on fixing GCSE to work properly on ARM rather than disabling it completely. Though currently with `libevas` it doesnt show additional gain relative to `-fno-gcse`, we still believe that for other applications this approach may be more profitable than disabling GCSE, as we have seen 10% gain on Aburto's `hanoi` benchmark. Still, this optimization delivers performance 5.5% better than `-O2`. The next column, `A5`, shows results for adjusting unrolling factor from 8 to 4 in `UNROLL8_PLD_WHILE` macro and disabling there manual prefetching (each of them contributed approximately by 3%). The next two columns, `A6` and `A7`, show numbers for prefetching/unrolling optimizations (`-fprefetch-loop-arrays`). This GCC optimization with default parameters gives 2% average improvement, but with few serious regressions up to -13%. If parameters are tuned properly (we used the second parameter string from Section 3.4), this optimization provides 4% improvement (`A7`), while growth is more evenly distributed among tests and without significant regressions. Total average gain on customized `expedite` from all ARM optimizations developed/tuned comparing to base `-O2` level is 15.78%.

Due to space constraints, we don't show separate results table for NEON autovectorization. The main results for NEON autovectorization are as follows. Fixing the autovectorization of shifts improved overall `expedite` performance by 8.82%, while certain tests ("Rect Blend" family), which suffered the most from poor shifts vectorization, grew as much as by 171%. These results are achieved with `-mvectorize-with-neon-quad` option, which gives about 1% overall gain, and manual unroll factor set to 4 in macro. The manual unrolling factor setting makes sense for those loops that don't get autovectorized (e.g. due to the presence of switch operator). Still, performance on pure ARM (without NEON) is 1.5% better than that with NEON autovectorization due to unaligned data accesses that autovectorizer currently doesn't handle.

	-O2 (A1, base)	-O2 -fno-gcse (A2)	to base, %	-O2 -farm-fix-gcse (A3)	to base, %	[A2] no prefetch, unroll =4 (in macro) (A5)	to [A2] -fno-gcse, %	[A5] -fprefetch-loop-arrays (A6)	to [A5], %	[A6] + prefetching params (A7)	to [A6] (default prefetch params), %	to [A5], %	to [base] -O2, %
1 - Widgets File Icons	11.53	11.79	**2.25**	11.9	**3.21**	12.35	**3.87**	13.26	**7.37**	13.07	**-1.43**	**5.83**	**13.36**
2 - Widgets File Icons 2	23.81	24.07	**1.09**	23.98	**0.71**	26.85	**12.02**	31.19	**16.16**	30.05	**-3.66**	**11.92**	**26.21**
5 - Image Blend Unscaled	14.89	14.9	**0.07**	15.47	**3.90**	16.46	**10.40**	16.63	**1.03**	16.89	**1.56**	**2.61**	**13.43**
6 - Image Blend Solid Middle Unscaled	10.92	11.06	**1.28**	11.12	**1.83**	11.65	**4.48**	11.78	**1.12**	11.75	**-0.25**	**0.86**	**7.60**
7 - Image Blend Fade Unscaled	7.3	7.92	**8.49**	7.81	**6.99**	8.3	**5.60**	8.15	**-1.81**	8.35	**2.45**	**0.60**	**14.38**
9 - Image Blend Solid Unscaled	50.71	52.01	**2.56**	51.58	**1.72**	50.37	**-3.25**	50.91	**1.07**	52.54	**3.20**	**4.31**	**3.61**
10 - Image Blend Solid Fade Unscaled	10.44	11.73	**12.36**	11.63	**11.40**	12.53	**6.10**	12.38	**-1.20**	12.54	**1.29**	**0.08**	**20.11**
11 - Image Blend Solid Fade Power 2 Unscaled	10.46	11.74	**12.24**	11.63	**11.19**	12.55	**6.18**	12.38	**-1.35**	12.54	**1.29**	**-0.08**	**19.89**
12 - Image Blend Nearest Scaled	6.41	6.48	**1.09**	6.73	**4.99**	7.09	**9.92**	7.4	**4.37**	7.45	**0.68**	**5.08**	**16.22**
14 - Image Blend Smooth Scaled	1.37	1.45	**5.84**	1.45	**5.84**	1.47	**7.30**	1.43	**-2.72**	1.43	**0.00**	**-2.72**	**4.38**
16 - Image Blend Nearest Same Scaled	22.81	23.91	**4.82**	23.86	**4.60**	24.62	**1.53**	25.12	**2.03**	25.05	**-0.28**	**1.75**	**9.82**
17 - Image Blend Nearest Solid Same Scaled	63.29	64.85	**2.46**	64.25	**1.52**	63.84	**-1.51**	65.43	**2.49**	66.1	**1.02**	**3.54**	**4.44**
18 - Image Blend Smooth Same Scaled	22.87	24.22	**5.90**	23.71	**3.67**	24.74	**2.02**	25.13	**1.58**	25.1	**-0.12**	**1.46**	**9.75**
19 - Image Blend Smooth Solid Same Scaled	69.94	71.61	**2.39**	71.31	**1.96**	69.36	**-4.01**	70.91	**2.23**	73.27	**3.33**	**5.64**	**4.76**
20 - Image Blend Border	1.47	1.56	**6.12**	1.56	**6.12**	1.62	**10.20**	1.58	**-2.47**	1.58	**0.00**	**-2.47**	**7.48**
21 - Image Blend Solid Middle Border	14.15	14.61	**3.25**	14.61	**3.25**	14.82	**4.15**	14.83	**0.07**	14.84	**0.07**	**0.13**	**4.88**
22 - Image Blend Solid Border	20.69	21.57	**4.25**	21.59	**4.35**	21.73	**3.97**	21.52	**-0.97**	21.74	**1.02**	**0.05**	**5.07**
23 - Image Blend Border Recolor	1.4	1.49	**6.43**	1.48	**5.71**	1.51	**9.42**	1.49	**-1.32**	1.49	**0.00**	**-1.32**	**6.43**
25 - Image Data ARGB	48.18	47.74	**-0.91**	47.07	**-2.30**	47.64	**1.21**	53.73	**12.78**	55.33	**2.98**	**16.14**	**14.84**
26 - Image Data ARGB Alpha	18.18	18.41	**1.27**	18.31	**0.72**	19.94	**8.90**	24.02	**20.46**	22.2	**-7.58**	**11.33**	**22.11**
27 - Image Data YCbCr 601 Pointer List	31.04	31.88	**2.71**	31.18	**0.45**	31.89	**2.08**	32.55	**2.07**	33.16	**1.87**	**3.98**	**6.83**
28 - Image Data YCbCr 601 Pointer List Wide Stride	25.95	27.51	**6.01**	27.13	**4.55**	27.57	**6.12**	27.13	**-1.60**	28.29	**4.28**	**2.61**	**9.02**
29 - Image Crossfade	33.21	34.6	**4.19**	34.36	**3.46**	34.85	**1.46**	45.6	**30.85**	41.99	**-7.92**	**20.49**	**26.44**
30 - Text Basic	38.4	38.89	**1.28**	38.74	**0.89**	39.74	**4.44**	40.68	**2.37**	41.31	**1.55**	**3.95**	**7.58**
31 - Text Styles	3.76	3.8	**1.06**	3.82	**1.60**	3.93	**3.69**	3.94	**0.25**	3.97	**0.76**	**1.02**	**5.59**
33 - Text Change	19.8	19.89	**0.45**	19.56	**-1.21**	20.93	**6.24**	21.58	**3.11**	21.79	**0.97**	**4.11**	**10.05**
34 - Rect Blend	5.76	8.24	**43.06**	8.19	**42.19**	10.73	**30.22**	9.32	**-13.14**	12.61	**35.30**	**17.52**	**118.9**
36 - Rect Solid	44.63	46.73	**4.71**	45.9	**2.85**	50.47	**9.55**	52.88	**4.78**	53.13	**0.47**	**5.27**	**19.05**
37 - Rect Blend Few	711.5	824.2	**15.84**	818.5	**15.04**	923.4	**14.51**	889.4	**-3.69**	943.8	**6.12**	**2.21**	**32.65**
39 - Rect Solid Few	1066	1096	**2.80**	1088	**2.05**	1204	**12.33**	1230	**2.14**	1176	**-4.37**	**-2.33**	**10.30**
41 - Image Blend Occlude 2 Few	131.1	133.8	**2.09**	136.1	**3.81**	139.6	**4.97**	143.2	**2.59**	146.8	**2.55**	**5.20**	**12.01**
43 - Image Blend Occlude 1 Many	64.13	67.64	**5.47**	67.86	**5.82**	68.92	**3.51**	68.31	**-0.89**	69.28	**1.42**	**0.52**	**8.03**
45 - Polygon Blend	11.1	13.44	**21.08**	12.67	**14.14**	13.65	**4.04**	13.33	**-2.34**	15.16	**13.73**	**11.06**	**36.58**
Geometric Mean	22.97	24.32	**5.88**	24.23	**5.48**	25.55	**6.47**	26.06	**1.97**	26.60	**2.09**	**4.09**	**15.78**

Table 1. The performance results (frames per second)

We have also verified the results against full original `expedite` test suite (as found in EFL repository) with 11.12% average performance optimization and with two different Cortex-A8 boards. Also, we verified the results with a different input data set, getting about 8% gain on original `expedite`.

5 Future work

Though we have fixed the vector shift instructions, there are still problems remaining with this optimization. First, autovectorizer currently is unable to produce operations involving vector and scalar arguments at the same time, e.g. for original operation `a[i] << CONST` it would first produce a vector containing four identical constants, and only then issue a vector operation `a_vec[j] << 4xCONST`, though NEON has a distinct operation for shifting vector by a single scalar. Second, the vectorizer can only handle aligned data, and if at runtime it finds out it's misaligned, it executes regular non-vectorized version of a loop, thus wasting time on alignment checks. We believe that this GCC optimization has more potential, and additional efforts should be done in improving autovectorization for NEON.

The results of tuning GCC loop prefetching/unrolling optimization prove that it is important for achieving good performance on ARM in applications with intensive memory usage and regular memory access pattern (like rasterization routines in `libevas`). Its performance results may vary among applications (as they vary among `expedite` tests), from input data (with small data sizes prefetching won't have time even to complete a load of first portion of data since prefetch distance may be greater than data size), and from the target architecture. We think that this optimization needs more detailed manual analysis so to find whether it has some implementation specifics that can be improved on ARM, as well as automatic tuning of its parameters with more applications.

Also, more general mechanism should be developed to provide GCSE optimization with a target-specific information on representation of constants (e.g. via a target hook), so to improve code on other architectures that may have similar constraints to those found on ARM.

6 Conclusions

We have identified a number of performance regressions in GCC optimizations with `libevas` on ARM, including GCSE, register allocation, autovectorization and loop prefetching, and suggested fixes to them. The solutions proposed altogether brought 15.78% average performance increase as measured with reduced `expedite` benchmark (as described in Section 2), with up to 119% increase on certain tests. We have verified the produced speedup on the full benchmark with 11.12% average performance optimization and with up to 119% performance increase on some tests.

Overall, we believe that a project on optimizing GCC compiler for certain applications and target architecture (both manually and automatically) makes

perfect sense, as such projects identify weak places of the compiler with regard to this target and application, and fixing these places may bring performance improvements. The results of such optimization should be made available to GCC community and developers of the target application so to avoid duplicate work on the same compiler optimizations and to encourage further development of the application using coding practices found to help compiler optimizations. After optimizations developed will be verified on wider range of applications, they could be enabled by default in GCC compiler for this target platform.

References

1. ARM Cortex-A8 Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344d/DDI0344D_cortex_a8_r2p1_trm.pdf`
2. The Enlightenment Foundation Libraries (EFL) web page. `http://www.enlightenment.org/p.php?p=about/efl`
3. ACOVEA home page. `http://www.coyotegulch.com/products/acovea/index.html`
4. Christopher G. Langton, ed. Artificial Life: An Overview. MIT, 1997.
5. David E. Goldberg. Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley, 1989.
6. A. Aho, R. Sethi, J. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1988.
7. E. Morel, C. Renvoise. Global Optimization by Suppression of Partial Redundancies. In Communications of the ACM, Vol. 22, Num. 2, Feb. 1979.
8. D. Nuzman, R. Henderson. Multi-platform Auto-vectorization. In Proceedings of the International Symposium on Code Generation and Optimization, 2006.
9. V. Makarov. The integrated register allocator for GCC. In Proceedings of the GCC Developers Summit, July 2007.