This is the post peer-review accepted manuscript of:


The published version is available online at: https://doi.org/10.23919/DATE.2019.8714897

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works
Design and Evaluation of SmallFloat SIMD extensions to the RISC-V ISA

Giuseppe Tagliavini*, Stefan Mach†, Davide Rossi*, Andrea Marongiu‡, and Luca Benini‡

* DEI, University of Bologna, Italy / Email: {giuseppe.tagliavini, davide.rossi, luca.benini}@unibo.it
† IIS, ETH Zurich, Switzerland / Email: {smach, luca.benini}@iis.ee.ethz.ch
‡ DISI, University of Bologna, Italy / Email: {a.marongiu}@unibo.it

Abstract—RISC-V is an open-source instruction set architecture (ISA) with a modular design consisting of a mandatory base part plus optional extensions. The RISC-V 32IMFC ISA configuration has been widely adopted for the design of new-generation, low-power processors. Motivated by the important energy savings that smaller-than-32-bit FP types have enabled in several application domains and related compute platforms, some recent studies have published encouraging early results for their adoption in RISC-V processors. In this paper we introduce a set of 16-bit SIMD (small FP) FPTuner extensions to the RISC-V 32IMFC, supporting scalar and SIMD operations (fitting the 32-bit register size) for 8-bit and two 16-bit FP types. The proposed extensions are enabled by exposing the new FP types to the standard C/C++ type system and an implementation for the RISC-V GCC compiler is presented. As a further, novel contribution, we extensively characterize the performance and energy savings achievable with the proposed extensions. On average, experimental results show that their adoption provide benefits in terms of performance (1.64× speedup for 16-bit and 2.18× for 8-bit types) and energy consumption (30% saving for 16-bit and 50% for 8-bit types). We also illustrate an approach based on automatic precision tuning to make effective use of the new FP types.

I. INTRODUCTION

Due to the widespread adoption of Internet of things (IoT) and smart devices, an increasing amount of embedded applications must deal with complex data analytics algorithms. While most embedded applications involving computations with high dynamic range are performed using binary64 (double-precision) or binary32 (single-precision) floating-point (FP) formats [21], an emerging trend focuses into adapting the FP arithmetic precision of applications according to the specific constraints of the applications or their domains [13].

To trade-off the energy per operation with dynamic range and precision, the IEEE 754 specification includes a 16-bit format referred to as binary16 (half-precision). In recent years significant advances in research have been made to exploit approximation even more aggressively, aiming at relaxing the “always maximum precision” abstraction [19][14]. The most promising approaches are moving beyond the concept of approximation alone, toward a novel paradigm called transprecision computing [12], which leverages computing architectures and applications that operate with a smooth and wide range of precision vs. QoR. The adoption of smallFloat types has been demonstrated to be particularly beneficial in the context of transprecision computing [17].

Adopting a mixed-precision type system is paramount to provide methodologies to perform precision tuning, i.e. to associate the minimum bit-width to program variables without violating the QoR constraints. Available tools for precision tuning are based on static (e.g., FPTuner [7] and PRECISA [15]) or dynamic techniques (e.g. Precimonious [16] and fptuningJaquard [9]). The adoption of these tools is totally complementary to our approach, since they can be used to associate the program variables to the smallFloat types.

In this paper we propose a set of extensions for the RISC-V instruction set architecture (ISA) to provide support for smaller-than-32-bit FP formats on embedded processors. As a baseline for our experimental setup we consider the RV32IMFC configuration, which has been widely adopted for the design of embedded processors [18][3][4][2][8]. The proposed FP types include binary16 and two non-standard formats, namely binary16alt and binary8, and are collectively referred to as smallFloat formats. Scalar operations are supported by a set of ISA extensions corresponding to the new formats, namely “Xf16”, “Xf16alt” and “Xf8”. Moreover the complementary “Xfvec” extension defines SIMD sub-word parallelism for all operations in the scalar FP extensions. Auxiliary operations have been added in an additional extension set “Xfvec”. As a further contribution we provide an extension to RISC-V GCC compiler to support smallFloat types.

We present a set of experimental results to evaluate the impact of our proposal in terms of performance and energy consumption. On average, automatic vectorization enables a 1.64× speedup for 16-bit types and a 2.18× speedup for binary8, with a further margin of ≈ 10% that can be obtained by the adoption of manual vectorization techniques. In terms of energy consumption, 16-bit types achieve on average 30% savings compared to single-precision when data is placed in a low-latency memory, whereas the savings are on average 50% for the binary8 format. Finally we present a case study in which automatic precision tuning is used to provide associations among program variables and FP types in accordance with application requirements.

The rest of the paper is organized as follows. Section II discusses the related work. Section III describes the smallFloat extensions. Section IV illustrates the modifications to the RISC-V GCC compiler. Section V presents an experimental evaluation of our work. Section VI discusses conclusive remarks and future work.

II. RELATED WORK

In the area of approximate computing [19][14] researchers have proposed a wide range of techniques which aim at increasing performance and energy efficiency of computing systems trading off with the quality of results (QoR). In recent years an evolution of this paradigm has been introduced, known as transprecision computing [12], which leverages computing architectures and applications that operate with a smooth and wide range of precision vs. QoR. The adoption of smallFloat types has been demonstrated to be particularly beneficial in the context of transprecision computing [17].

Adopting a mixed-precision type system is paramount to provide methodologies to perform precision tuning, i.e. to associate the minimum bit-width to program variables without violating the QoR constraints. Available tools for precision tuning are based on static (e.g., FPTuner [7] and PRECISA [15]) or dynamic techniques (e.g. Precimonious [16] and fptuningJaquard [9]). The adoption of these tools is totally complementary to our approach, since they can be used to associate the program variables to the smallFloat types.

In terms of energy consumption, 16-bit types achieve on average 30% savings compared to single-precision when data is placed in a low-latency memory, whereas the savings are on average 50% for the binary8 format. Finally we present a case study in which automatic precision tuning is used to provide associations among program variables and FP types in accordance with application requirements.
TABLE I

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Instruction</th>
<th>Semantics</th>
<th>ISA Ext.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>fadd.h</td>
<td>rd = rs1 + rs2</td>
<td>Xf16</td>
</tr>
<tr>
<td>Conversions</td>
<td>vfcvt.x.h</td>
<td>rd = (f32)rs1</td>
<td>Xf16</td>
</tr>
<tr>
<td>Vector Arith.</td>
<td>vfadd.h</td>
<td>rd[1] = [rs1] + rs2[1]</td>
<td>Xfvec</td>
</tr>
<tr>
<td>Vector Conv.</td>
<td>eg. vfcvt.x.h</td>
<td>rd[1] = [int16v[rs1][1]]</td>
<td>Xfvec</td>
</tr>
<tr>
<td>Expanding</td>
<td>fmacex.s.h</td>
<td>rd = (fp32)(rs1</td>
<td>Xfaux</td>
</tr>
<tr>
<td>Other</td>
<td>eg. vfdoex.h</td>
<td>rd[1] = (fp32)(op[rs1][rs2])</td>
<td>Xfaux</td>
</tr>
</tbody>
</table>

However, this extension is based on the style of vector register architecture introduced by Seymour Cray in the 1970s tailored to high-performance architectures, as opposed to the packed SIMD approach proposed in this work targeting low-power embedded processors.

III. SMALLFLOAT EXTENSIONS

As a base, scalar extensions are provided that match the operations available in “F” and “D” standard extensions. Furthermore, optional vectorial extensions are specified which make use of SIMD sub-word parallelism on the FP register file. Lastly, there is an optional extension for auxiliary operations. The smallFloat extensions have no collisions with reasonable RISC-V implementations, and can thus be included in any implementation without loss of compliance with the standard. Table I gives a summary of available operation types with smallFloat extensions active. More details are provided in the smallFloat ISA manual [5].

A. Scalar Extensions

The scalar extensions provide support for the IEEE binary16 and custom binary16alt FP formats (both 16-bit wide), as well as the custom binary8 format (8-bit wide) [17]. The adoption of an alternative 16-bit format has been proven to be highly beneficial for those applications that require the dynamic range of binary32 but can tolerate a lower precision [17]. Each format is contained in its own respective ISA extension “Xf16”, “Xf16alt” and “Xfs”. The operations on smallFloat formats are equivalent to their single-precision counterparts, thus their encoding closely matches the standard FP operations (e.g., fadd.h in Table I). An unused configuration of the FP format field in the instruction word has been chosen to signify 16-bit FP types, while the patterns representing quad-precision FP operations (128-bit) have been repurposed to now denote the 8-bit FP type. While this poses a collision with the “Q” RISC-V standard extension, it is highly unlikely embedded implementations targeted towards low precision FP will also implement 128-bit floats. The two 16-bit formats (binary16 and binary16alt) are differentiated using unused states of the rounding-mode fields in the instruction word.

B. Vectorial Extension

The vector extension “Xfvec” is encoded in its own encoding space, which utilizes a previously unused prefix in the RISC-V “OP” opcode. This extension defines SIMD sub-word parallelism for all operations in the scalar FP extensions, such as the vfcvt.x.h operation in Table I. If “Xfvec” is supported, vectorial FP operations are added for all supported FP formats that are narrower than the width of the FP register file (FLEN) as shown in Table II.

“Xfvec” adds vector-specific conversion operations (e.g., the vfcvt.x.h operation in Table I). In addition, cast-and-pack instructions were added that convert two scalar single- or double-precision operands and insert them into two adjacent entries of a packed vector (e.g., the vfvecpk.h.s operation in Table I). These operations were added since “convert scalars and assemble vectors” operations emerged as a main bottleneck of transprecision computing [11].

C. Auxiliary Operations Extension

The extension set “Xfaux” includes additional operations that have been encoded in unused regions of either scalar or vectorial extensions. It includes the so-called expanding operations that take smallFloat type operands and return a single-precision result, making explicit conversion instruction cycles unnecessary where the dynamic range of operands increases over the execution. These instructions include expanding multiplication, multiple-accumulate of smallFloats on a binary32 accumulator (the fmacex.s.h operation in Table I), as well as expanding dot-products.

IV. COMPILER SUPPORT

To provide support to the smallFloat types in the GCC compiler, we have extended the real interface – used as an internal representation for all the FP types supported by the programming language – with callback functions that enable to convert data from the internal format to the smallFloat ones. Then we have augmented the RISC-V back-end to include new machine modes and corresponding machine description rules. At the higher level of abstraction, we have extended the standard C/C++ type system by introducing a new set of keywords (float8, float16 and float16alt) and extending the conversion rules to guarantee a correct behavior. GCC includes an automatic vectorization pass that operates on the middle-end intermediate representation [1]. In our work we have extended the GCC auto-vectorizer to enable the adoption of smallFloat types. Moreover, programmers can manually vectorize their code using the vector support provided by GCC. To complement this support we have provided a set of compiler intrinsics which provide access to the operations included in the “Xfvec” and “Xfaux” ISA extensions (see Table I). An example of manual vectorization using vectorial types and intrinsics is provided in Section V-C (Figure 5).

V. EXPERIMENTAL RESULTS

A. Setup

In our experiments we have considered a set of computational intensive kernels from the Polybench/C benchmark suite [20] and a support vector machine (SVM) used in the context of an embedded application [6]. Our target platform is the RISCY core of the open-source PULP project. We have added the smallFloat extensions to the PULP virtual platform and we have implemented the compiler support on its official compilation toolchain. A smallFloat unit implementing the proposed extensions was synthesized for the UMC 65 nm technology and the energy costs of FP operations have been obtained through simulation of the post-layout design set to 350 MHz using worst-case conditions (1.08 V, 125°C).

2Hardware design and software tools: https://github.com/pulp-platform/
Figure 1 shows the speedup achieved when different small-Float types are replaced to standard float variables, comparing automatic and manual vectorization. The solid part of the bars shows the measured results, whereas the dashed segments indicate the ideal ones. On average, float16 types allow for 3.34× faster execution than native float. The maximum speedup achievable by automatic vectorization is 1.64×; manual vectorization enables an additional ≈12% faster execution, with average 1.5× and peak 1.91× speedups. When float8 types are concerned, automatic vectorization enables 2.18× speedups on average float and up to 3.08×. The same speedup figures increase with manual vectorization to average and peak speedups of 2.35× and 3.58×, respectively. In many cases the speedups are very close to the ideal ones. In those cases when there is a significant difference, this is due to the fact that the computation happens inside nested loops, with the innermost using the iterator of the outermost as an upper bound; this condition creates significant additional overhead to handle the prologue/epilogue loops to the vectorized one.

Figures 2 and 3 depicts an experiment where speedup and energy consumption have been calculated for different memory latencies. Specifically, we indicate with L1 the setup where memory operations have 1-cycle latencies (representative of a load/store from a level-1 cache). L2 stands for 10-cycle latency operations and L3 for 100-cycle operations. For this experiment (and from now on) we only consider the manually vectorized version of each benchmark. Also, we only consider float16 as a 16-bit type, as there is no difference in speedup (or energy) with the float16alt type. Focusing on speedup, float16 types on average experience 7.4% higher values when data is read/written from L2, as compared to L1, and 10.65% higher values when data is read/written from L3, as compared to L1. For float8 types, these numbers get down to 4.75% and 8.01%. Concerning energy, float16 types achieve on average 30% savings compared to float, when data is placed in L1 memories. For float8 type the savings are on average 50%. The savings for float8 are less than its ideal maximum (twice the savings of 16-bit FP types); this effect is mainly due to the major impact of pack/unpack operations compared to 16-bit types.

Table III reports the QoR expressed in terms of SQNR, programmers can choose the minimum configuration among the available ones. This is a bottom-up approach that we have adopted to perform extensive benchmarking of the ISA extensions; in the following section we describe a top-down approach driven by the application constraints.

### C. A case study of mixed precision

In this section we explore the implication on performance and energy of vectorized codes under mixed-precision. We have considered a gesture recognition application using SVM as a classifier [6] and we have imposed a strict constraint on the QoR, i.e. to avoid classification errors on our data set. To find the minimum size for program variables we have used a tool for precision tuning [9]. The variable-to-type associations resulting from the tuning process include a float variable for the final accumulation and float16 for other variables (i.e., inputs, weights, intermediate results). By tolerating a minimum amount of classification errors (around 5%), the tuning tools would assign the accumulation variable to the float16alt type. As already mentioned, the adoption of this format is beneficial whenever the dynamic range of a variable is more critical than its precision.

Figure 4 shows the instruction count breakdown for both the original version and its two vectorized variants (automatic and manual). Focusing on automatic vectorization, we can see that many of the calculations on float scalar variables are converted into calculations on scalar and vectorial float16, which also has the merit of significantly reducing the number...
memory instructions. The main drawback of the automatic vectorization scheme resides in some inefficiencies on how the vectorial code is generated. Here, some optimizations that are applied on the baseline code are not automatically applied to the vectorized code, which leads to a significant number of additional ALU instructions, which end up eating all the margin for savings (and beyond). With manual vectorization it is possible to (i) convert more float operations in float16 vectorial ones (note that the scalar float16 operations have also disappeared); (ii) reduce the overhead instructions (ALU or conversion) generated for the vectorial ones. It is important to highlight that the mixed-precision types are used.

This effect is further explained in Figure 5, which shows a code snippet and its manually vectorized version using the widening multiply-and-add operation in the “Xfaux” extension. Manual vectorization enables to remove the conversion instructions, reducing by $25\%$ the instruction count.

Figure 6 summarizes the results achieved for the gesture recognition application when mixed precision is used as compared to fully replacing float variables with float16 or float8 ones. It is important to highlight that the mixed-precision scheme allows speedup and energy savings comparable to those achievable with float16, but achieves the same accuracy of the original float version. This outcome is fully aligned with the principles of transprecision computing: this technique enables a very fine-grained control of approximation at the intermediate steps of computation, nevertheless the accuracy of the results is not compromised at all.

VI. CONCLUSION

This paper introduces a set of extensions for the RISC-V ISA to support a set of smaller-than-32-bit FP formats. We present (i) a full specification for the proposed smallFloat extensions, (ii) design and implementation of the compiler support and (iii) a full experimental evaluation highlighting benefits and limits of this proposal. Experimental results show benefits in terms of performance ($1.64 \times$ speedup for 16-bit and $2.18 \times$ for 8-bit types) and energy consumption (30% saving for 16-bit and 50% for 8-bit types), and include a case study for mixed-precision computing in which the accuracy of the results is not compromised at all by the adoption of multiple formats.

Our future work will be focused on three main aspects, (i) improvement of the auto-vectorization, (ii) extension of small-Float support to RV64-based systems, and (iii) full integration in a transprecision computing toolchain.

ACKNOWLEDGMENT

This work has been partially supported by the European H2020 FET project OPRECOMP (g.a. 732631).

REFERENCES