A Basic Linear Algebra Compiler for Embedded Processors

Nikolaos Kyrtatas Daniele G. Spampinato Markus Püschel

Department of Computer Science
ETH Zurich, Switzerland
nkyrtata@student.ethz.ch, {danieles, pueschel}@inf.ethz.ch

***Abstract*—Many applications in signal processing, control, and graphics on embedded devices require efficient linear algebra computations. On general-purpose computers, program generators have proven useful to produce such code, or important building blocks, automatically. An example is LGen, a compiler for basic linear algebra computations of fixed size. In this work, we extend LGen towards the embedded domain using as example targets Intel Atom, ARM Cortex-A8, ARM Cortex-A9, and ARM1176 (Raspberry Pi). To efficiently support these processors we introduce support for the NEON vector ISA and a methodology for domain-specific load/store optimizations. Our experimental evaluation shows that the new version of LGen produces code that performs in many cases considerably better than well-established, commercial and non-commercial libraries (Intel MKL and IPP), software generators (Eigen and ATLAS), and compilers (icc, gcc, and clang).**

I. Introduction

Dense linear algebra computations are common, and often performance-critical, in many domains in computer science and engineering. For this reason, excellent library support exists, typically structured around the BLAS (basic linear algebra subroutines) interface [1], [2], [3], [4]. However, these approaches mainly target large problem sizes, which are common in scientific computing and modeling; for small sizes, the performance is usually suboptimal. LGen [5], a program generator for basic linear algebra computations (BLACs) of fixed size operands, was designed to address this problem. These specialized computations are common in various domains including computer vision and graphics (e.g., geometric transformation and stereo vision algorithms), control systems (e.g., optimization algorithms and Kalman filters), and media processing (e.g., Viterbi algorithms for speech recognition). LGen is designed closely after Spiral (a generator for linear transforms [6], [7]) and can generate scalar as well as SIMD vector code. It was demonstrated that LGen-generated code can provide significant performance gains compared to existing library code (e.g., from [4], [8], [9], [10]) on Intel-based desktop and workstation computers.

Embedded processors. The above domains are of equal, if not higher importance in embedded computing. Embedded processors are used in automotive electronics, network devices, smartphones and tablets, and many other ubiquitous systems with a reduced power budget. However, their energy efficiency comes at a price: a reduced set of resources. Examples include in-order execution units, smaller numbers of instruction-issue ports, and less efficient access to unaligned memory locations. Because of these, additional optimizations are needed to obtain highest performance, and hand-tuning for these processors is even more common than on their desktop counterparts. A program generator that performs these automatically for a specific domain is thus an attractive solution to achieve highest performance at very low development cost.

Main contributions. The main contribution of this paper is an extension of LGen that generates efficient BLAC code for embedded processors, including support for the NEON instruction set. In doing so, we make the following contributions:

- We introduce a technique for load/store optimizations on vector ISAs, targeted specifically to our application domain, to reduce the number of data rearrangement instructions in the generated code.
- We show experimental results on four processors: one Intel-based (Atom) and three ARM-based (Cortex-A8, Cortex-A9, and ARM1176). The performance of LGen-generated code compares favorably against performance libraries (Intel MKL and IPP), other generators (ATLAS and Eigen), and naïve code optimized using icc, gcc, and clang.

II. Background

Next we will provide background knowledge on the basic linear algebra compiler LGen [5] and identify shortcomings for the embedded domain that are addressed in this paper. In the following, matrices will be denoted as $A, B, ...$, (column) vectors as $x, y, ...$, and scalars as $\alpha, \beta, ...$. A BLAC was defined in [5] as a computation on matrices, vectors, and scalars formed with four basic operations: matrix addition, matrix multiplication, matrix transposition, and scalar multiplication. Note that for these operations vectors are viewed as matrices.

LGen takes as input a BLAC including the sizes of the inputs and output. A valid example would be

$$B = x^T A + \alpha y^T, \tag{1}$$

where A and B are matrices of sizes 19×3 and 1×3, respectively, and x and y are vectors of length 19 and 3. The output of LGen is a C function (optionally using intrinsics for vector extensions) that implements the BLAC. Thus, LGen can be viewed as a program generator for small dense linear algebra.

Overview. LGen translates an input BLAC into C code using two intermediate compilation phases as shown in Fig. 1. During the first phase, the input BLAC is transformed at the mathematical level using a domain-specific language (DSL) called Σ-LL. At the code level, these transformations correspond to loop optimizations such as multi-level tiling, merging, and exchange (in Fig. 1 only the tiling decision is visualized). During the second phase, the mathematical expression

978-3-9815370-4-8/DATE15/©2015 EDAA

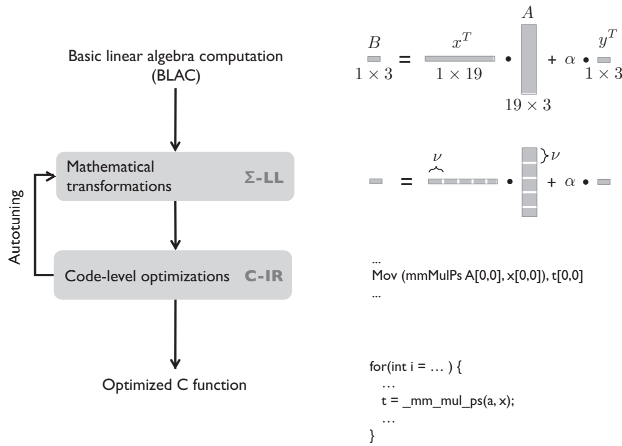


Fig. 1. LGen overview.

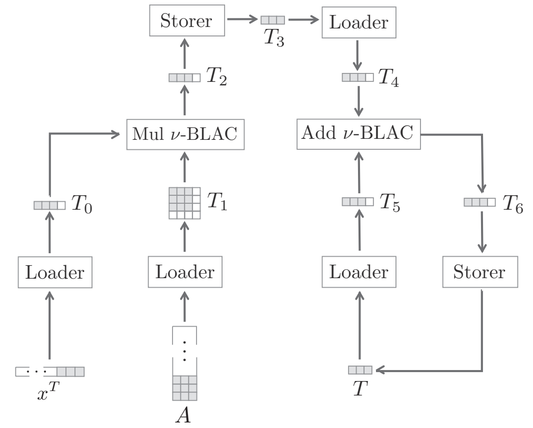


Fig. 3. C-IR codelets chain handling leftovers for $T = x^T A$. A is 19×3, $\nu = 4$. White elements in matrices T_0, ..., T_6 are zeros.

obtained from the first phase is converted to a C-intermediate representation (C-IR). At this level, LGen performs additional optimizations such as loop unrolling, scalar replacement, and conversion into SSA form. Finally, since different tiling decisions lead to different code versions of the same computation, LGen uses autotuning (search) to select the C function that achieves the best performance on the specified target device.

Vector code generation. LGen can generate functions that are vectorized using SIMD intrinsics. In this case LGen receives the vector length ν as additional input (e.g., $\nu = 4$ for NEON quadword float vectors). During the first compilation phase, the vector length ν is used for an additional tiling (ν-tiling) at the innermost level to create a structure for efficient mapping to vector instructions. As a consequence of the ν-tiling, the input BLAC is decomposed into ν-BLACs: the 18 basic operations on matrices of size $\nu \times \nu$ and vectors of length $\nu \times 1$ or $1 \times \nu$. In Fig. 2 we show the five ν-BLACs used to vectorize matrix multiplication. ν-BLACs are preimplemented once for every vector ISA and composed by LGen to obtain the final code for the BLAC. For embedded processors, LGen currently supports SSSE3 (Intel Atom) and NEON (ARM Cortex-A8 and A9), an extension included for the present paper. More details about vector code generation using ν-BLACs can be found in [5]. We now explain how we handle memory accesses when computing a ν-BLAC.

Accessing memory. All matrices involved in a ν-BLAC satisfy two assumptions: (a) every dimension has a length of either 1 or ν, and (b) the rows of matrices are contiguous in memory. As described in [5], the ν-tiling decision taken by LGen during the early compilation phase is made explicit using gather and scatter matrices. However, at the Σ-LL level there is no concept of loads and stores, but only of accesses to matrices. At the C-IR level such accesses are translated to code using two collections of pre-implemented codelets called

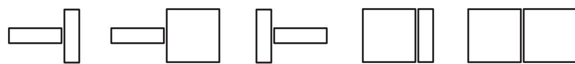


Fig. 2. The five ν-BLACs used to vectorize matrix multiplication. Matrices are $\nu \times \nu$ and vectors $\nu \times 1$ or $1 \times \nu$. The complete set of 18 ν-BLACs can be found in [5].

Loader and Storer that pack and unpack the data to make the ν-BLACs usable. Besides making the data contiguous, Loaders and Storers pack and unpack eventual leftovers (sizes $< \nu$) created by tiling. Using a packing-unpacking approach allows these leftover BLACs to be also treated as ν-BLACs, thus enabling a complete vectorization of the generated code.

As an example, consider the partial computation $T = x^T A$ from (1). Assume that A is 16-byte aligned and stored in row-major order, that our target ISA is SSSE3, and that we decide to vectorize choosing $\nu = 4$. In Fig. 3 we give a pictorial representation of the flow of computations required to take into account the leftover (the 1×3 matrix T) produced by ν-tiling. The 1×3 leftover tile of x^T and the 3×3 leftover tile of A are loaded to the temporary matrices T_0 and T_1 (of sizes $1 \times \nu$ and $\nu \times \nu$). A similar approach is followed with the outputs using Storers. Now, the code associated to the store-load chain $T_2 \to T_3 \to T_4$ should clearly be removed, holding the result from the multiplication in register for the second computation (Add ν-BLAC). However, as shown in Fig. 4, on embedded processors lacking mask-load/store instructions, the special size of the matrix result (1×3) requires additional data rearrangement overhead (e.g., vector shuffles). In a generic compiler such as clang, disposing of instructions such as shuffles is a more complicated task due to the large set of mask combinations. In our domain-specific LGen only a small subset of these actually occur, making the approach practical. In Section III we present a technique to remove data rearrangement overhead using generic C-IR vector loads and stores.

III. Store-Load Elimination with Generic C-IR Vector Loads and Stores

LGen generates C-IR code by combining codelets from the target ISA's Loader, Storer, and ν-BLACs. All of these codelets are implemented following a load-compute-store approach, meaning that they first load data from memory into registers, then they process the data, and finally they store the results back to memory. As a result, the generated C-IR code consists of chains of codelets, where data flow from one codelet to the next one, as shown in Fig. 3. Data between two consecutive codelets are stored in a local array allocated within the function. However, in the example shown in Fig. 3 the use of the

```
/* Begin nuBLAC mul */
// ... nuBLAC multiplication
__m128 mul_res = ...;
_mm_storeu_ps(T2, mul_res);
/* End nuBLAC mul */

/* Begin Storer 1x4 → 1x3 */
__m128 v0 = _mm_loadu_ps(T2);
_mm_storel_pi((__m64*)(T3), v0);
_mm_store_ss(T3 + 2,
_mm_shuffle_ps(v0, v0, _MM_SHUFFLE(3, 3, 3, 2)));
/* End Storer 1x4 → 1x3 */

/* Begin Loader 1x3 → 1x4 */
_mm_storeu_ps(T4, _mm_shuffle_ps(
_mm_loadl_pi(_mm_setzero_ps(), (__m64*)(T3)),
_mm_load_ss(T3 + 2),
_MM_SHUFFLE(1, 0, 1, 0)
));
/* End Loader 1x3 → 1x4 */
```

Fig. 4. SSSE3 code snippet for the store-load chain $T_2 \rightarrow T_3 \rightarrow T_4$ (Fig. 3).

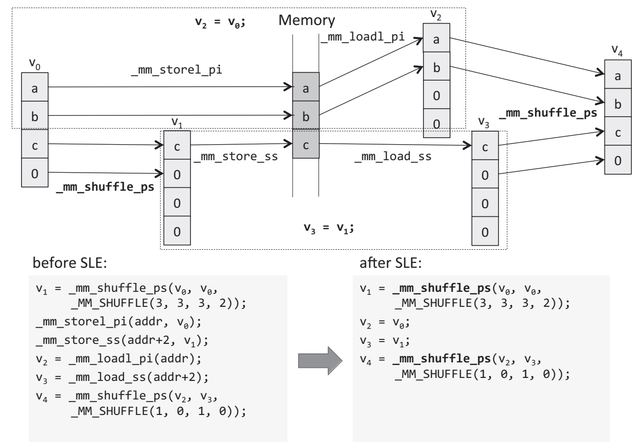


Fig. 5. Example of SLE with load and store instrinsics. Both v_0 and v_4 contain a leftover of length three.

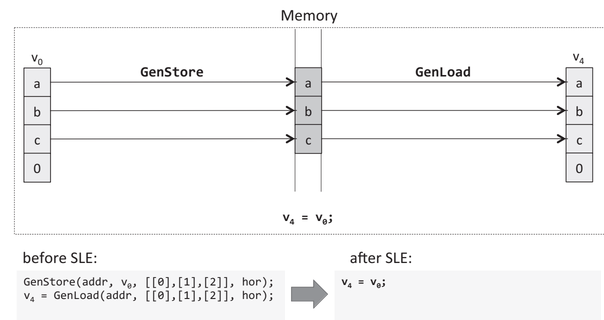


Fig. 6. Example of SLE with generic load and store C-IR instructions analogous to the one in Fig. 5.

temporary matrices T_0, ..., T_6 is superfluous since the result of each codelet could be passed directly to the next codelet through registers. Store-load elimination (SLE) is a compiler technique that can be used to eliminate redundant memory accesses. As we explain in the following, the challenge is in removing unnecessary shuffles.

Problems using SLE with intrinsics. Standard SLE works in the following way: Whenever a pair of load and store intrinsics with matching access patterns is found, it is replaced with an assignment between vector variables. By access pattern we refer to the mapping between memory locations and positions within a vector variable. Loads that do not follow any store with the same access pattern are left unchanged. The same holds for stores that are not followed by loads with the same access pattern. For example, Fig. 5 depicts the store-load chain from Fig. 4 with access patterns depicted with arrows. Finding store-load pairs to eliminate is thus equivalent to finding stores and loads whose outgoing and ingoing arrows can be "wired" together. Since storing the three values a, b, c to memory is implemented in the same way as loading them, we can safely wire up outgoing and ingoing connections. In other words, applying SLE to this piece of code replaces the wired store-load pairs with assignments between variables v_2 and v_0, and between v_3 and v_1 (bottom-right code snippet). However, shuffles are left untouched by the analysis.

Better SLE with generic C-IR vector load and store instructions. To facilitate the application of SLE and avoid unnecessary shuffle instructions like the ones shown before, we use in our C-IR load and store instructions that do not correspond to specific intrinsics, but are generic enough to represent all possible vector accesses to memory. These instructions are used during SLE and are translated to specific intrinsics only during unparsing C-IR into C code.

The full syntax of a generic load is `GenLoad(addr, poslist, orientation)` and the one of a generic store is `GenStore(addr, v, poslist, orientation)`. The parameter `addr` is a memory address, `v` is a vector variable, and `poslist` is a list that maps memory locations to positions within the vector `v`. More specifically, the ith element of `poslist` maps the ith element starting from `addr` to a list of positions within a vector. For example, `GenLoad(addr, [ [0],[1],[2],[3] ], hor)` loads four consecutive elements starting from `addr` to the four positions of the returned vector, while `GenLoad(addr, [ [0,1,2,3] ], hor)` loads one element at `addr` to all four positions of the returned vector. The parameter `orientation` can take the value `hor` or `vert` and determines whether the access refers to a row or a column of a mathematical matrix. At the C-IR level, LGen maintains a mapping between mathematical matrices and memory-allocated arrays. A generic load/store with `orientation` set to `vert` is interpreted as a strided memory access, and the stride is obtained from the size of the matrix that the related memory address is associated with.

Using these generic load/store instructions, for example, the code segment of Fig. 5 is transformed into the one shown in Fig. 6. Applying SLE on the latter will leave us with a single assignment, without any shuffle instructions. An example implementation of the generic load and store in Fig. 6 on NEON is shown in Fig. 7. Note that the "non-dual" mapping of generic loads and stores to code does not affect SLE.

Optimal alignment detection. Several embedded processors with vector architectures offer both aligned and (slower) unaligned loads and stores. In our case (Intel Atom, ARM Cortex-A8, and A9) aligned instructions are at least twice as fast. Furthermore, on Intel Atom, unaligned instructions require two out of two issue ports for execution, making it impossible to issue an unaligned load simultaneously with an unaligned store. In contrast the aligned ones require only one port [11].

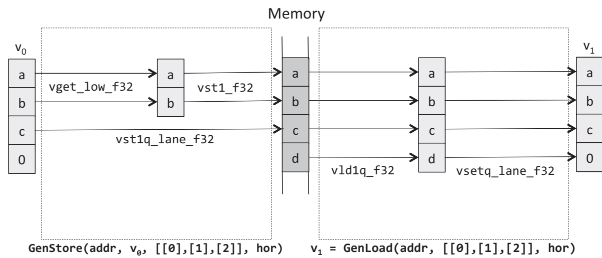


Fig. 7. Unpacking-packing of a leftover of length three using a non-dual NEON implementation of the generic store and load.

For these reasons we incorporated into LGen an alignment detection algorithm based on abstract interpretation [12] at the C-IR level. The idea behind the algorithm is as follows: First the C-IR code is analyzed by applying abstract interpretation using the abstract domain of congruences [13]. This analysis is similar to the congruence detection technique described in [14] and determines for each memory access, whether all the addresses used in this access during program execution are guaranteed to be divisible by the ISA-specific alignment length. In the affirmative case, an aligned load or store intrinsic is generated when unparsing to C code. Otherwise an unaligned instruction is generated.

Note that if the alignment of the input and output arrays is not known at compile time, LGen can generate various code versions, one for each of the possible alignments, including the control to select the proper one at runtime.

In our context of BLACs, all addresses are affine combinations of induction variables. Based on this restriction, we can prove that our alignment analysis is precise, i.e., each aligned memory access is detected and there are no false negatives. The proof is omitted due to space limitations and can be found in [15].

IV. Results

In this section we present performance experiments conducted to evaluate the code generated by LGen on four embedded processors: Intel Atom, ARM Cortex-A8, ARM-Cortex-A9, and ARM1176. First we explain the experimental setup; then we show and discuss the results for each target processor. A larger set of experiments is discussed in [15].

A. Experimental setup

Table I summarizes relevant information about the computing platforms. The peak performance values in this table were computed without considering the impact of loads and stores and assuming an ideal ratio of additions and multiplications. In the following we describe our tests, competitors, and provide details about the configuration of our hardware and software environment.

Chosen BLACs. We selected the following BLACs for our experiments:

- $\alpha = x^T Ay$: A memory-intensive bilinear form (*blinf*).
- $C = \alpha(A_0 + A_1)^T B + \beta C$: A compute-intensive matrix multiplication, where one of the operands is obtained by a matrix addition (*gemam*).

By memory-intensive we mean an operational intensity (operations/data movement) of $O(1)$. Both BLACs need more than one BLAS call. In the rest of this section we refer to the

TABLE I. Properties of the platforms used for the experiments. AI stands for arithmetic instruction, LS for load/store, f/c are flops/cycle.

CPU	**Intel Atom D2550**	**Cortex-A8**	**Cortex-A9**	**ARM1176**
Vector ISA	SSSE3	NEON	NEON	-
D-L1 [kB]	24	32	32	16
I-L1 [kB]	32	32	32	16
Peak [f/c]	6	4	4	1
Execution	in-order	in-order	OoO	in-order
Issues 2 AI	yes	yes (FMA)	yes (FMA)	no
Issues LS+AI	yes	yes	no	no
Board	Mini-PC	BeagleBone Black	Kayla DevKit	Raspberry Pi
OS kernel	Linux 3.8	Linux 3.8	Linux 3.1	Linux 3.6

BLACs above using the tags provided in parentheses. Unless stated otherwise, all matrices and vectors are 16-byte aligned.

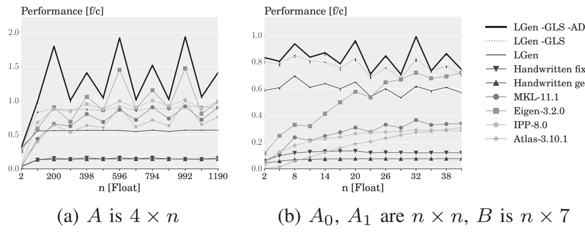
Competitors. Our selected competitors are: (a) Intel MKL 11.1 (Intel Atom only), (b) Intel IPP 8.0 (Intel Atom only), (c) Eigen 3.2.0 (all processors), (d) ATLAS 3.10.1 (all processors), and (e) compilers taking as input handwritten scalar code (all processors). Regarding the last case, we considered both code with fixed problem sizes that are known at compile time (labeled as *fixed* on plots) and code with unknown problem sizes that are passed as arguments (labeled as *gen* on plots).

Measuring process. All experiments involve single-precision code. For all plots, the y-axis shows performance in flops per cycle (f/c), and the x-axis shows the value of the input's varying dimensions as number of float elements.

The flop count is derived from the BLAC while cycles are explicitly measured. On Intel Atom, cycles are measured using the `rdtsc` instruction. On the ARM Cortex-A8 and ARM1176 we used the cycle counter of the PMU. On the ARM Cortex-A9 we used the Linux perf infrastructure.

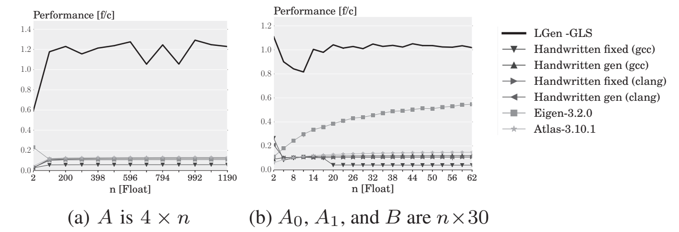
All experiments are run under warm cache conditions using the same measuring strategy as in [5]: The code is executed multiple times for at least 10^8 cycles. The reported measurement is the average number of cycles per execution. This process is repeated 15 times to compute median and quartile information. Each point in the plots is the median of 15 repetitions and it is accompanied by whiskers that show the most extreme data points falling into the range $[1.5q_1,\ 1.5q_3]$, where q_1 and q_3 are the first and third quartiles.

Hardware and software configuration. We disabled hyper-threading on Intel Atom and CPU throttling on the three ARM processors. LGen was configured to use a random search with sample size of 10. For both Intel MKL and ATLAS, we implemented blinf as a combination of `cblas_sgemv` and `cblas_sdot`. Gemam was implemented in MKL with a call to `MKL_Somatadd` followed by `cblas_sgemm` and in ATLAS with a call to `cblas_saxpy` followed by `cblas_sgemm`. For Eigen we used Map interfaces over existing arrays, no-alias assignments, and we enabled vector code generation by defining `EIGEN_VECTORIZE`. ATLAS was built natively using gcc 4.7 on



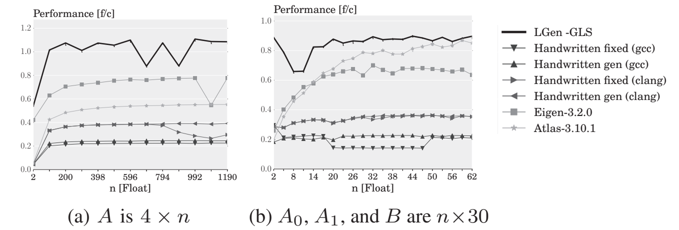
(a) A is $4 \times n$ (b) A_0, A_1 are $n \times n$, B is $n \times 7$

Fig. 8. (a) $\alpha = x^T Ay$ and (b) $C = \alpha(A_0 + A_1)^T B + \beta C$ on Intel Atom.



(a) A is $4 \times n$ (b) A_0, A_1, and B are $n \times 30$

Fig. 9. (a) $\alpha = x^T Ay$ and (b) $C = \alpha(A_0 + A_1)^T B + \beta C$ on Cortex-A8.



(a) A is $4 \times n$ (b) A_0, A_1, and B are $n \times 30$

Fig. 10. (a) $\alpha = x^T Ay$ and (b) $C = \alpha(A_0 + A_1)^T B + \beta C$ on Cortex-A9.

all platforms. On the Intel Atom we used the provided architectural defaults, while for the other three processors we executed a full search to find the best values for the ATLAS parameters. For all four processors, gemm was retuned after the installation to improve the performance of ATLAS for small matrix computations, as it is described in the errata section of the ATLAS website[1]. On the Intel Atom tests were compiled with icc 14 (flags `-O3 -xHost -fargument-noalias -fno-alias -no-ipo -no-ip -no-prec-div`); on the ARM processors with clang 3.4 (flags `-O3 -mcpu=<cpuname>`) and gcc 4.7 (flags `-O3 -ffast-math -fsingle-precision-constant -fstrict-aliasing -mcpu=<cpuname> -march=armv7-a -mtune=<cpuname> -mfpu=neon -mfloat-abi=hard`).

Labelling conventions. For plots we use the following labelling convention: *LGen* for the basic version of LGen, *LGen -GLS* for LGen using generic C-IR loads and stores, and *LGen -GLS -AD* for LGen using both generic C-IR loads and stores and alignment detection. Alignment detection only applies on Atom, since the ARM NEON intrinsics do not provide aligned loads and stores. Also, the NEON ν-BLACs were implemented directly with generic loads and stores; thus a comparison to the previous LGen is omitted.

B. Intel Atom

In Fig. 8a we show the results for the computation of blinf. *LGen -GLS -AD* performs better than all competitors, achieving speedups of up to 2.8× with respect to MKL. The presence of several downward spikes is due to the amount of unaligned instructions available in the code. The size of the panel matrix A strongly influences performance, bringing it down to 1 f/c whenever n mod $4 \in \{1, 3\}$ (which yields only 25% aligned accesses).

In Fig. 8b we show the performance results for gemam. The performance of *LGen -GLS* is around 30% higher than the one of LGen. Alignment detection adds another 30% of improvement over the performance of LGen for matrix sizes that favor this optimization (i.e. divisible by 4). Eigen, the best competitor, performs better than LGen for larger matrices, but never better than *LGen -GLS* and *LGen -GLS -AD*.

C. ARM Cortex Processors

The two Cortex-A processors present two critical microarchitectural differences: (a) Scalar floating point operations are more efficient on Cortex-A9 and (b) Cortex-A8 can issue a NEON load/store instruction together with a NEON arithmetic operation, while this is not possible on Cortex-A9.

Cortex-A8. In all experiments conducted on Cortex-A8 (Fig. 9) the competitors achieve lower performance than LGen (in most cases less than 0.2 f/c). The main reason is the mixing of scalar and vector instructions, which on Cortex-A8 leads to poor performance. This does not apply to LGen, as it generates completely vectorized code, even when handling leftovers. The performance of LGen is mostly in the range 1-1.3 f/c, being up to 9× faster than the best competitor.

Cortex-A9. Fig. 10 shows the experimental results for Cortex-A9. For blinf (Fig. 10a) Eigen is the best competitor, achieving 10–40% lower performance than LGen. For gemam (Fig. 10b) LGen is more than 2× faster than the optimizing compilers and 25% faster than Eigen, with a performance of between 0.8 and 1 f/c. For wider matrices ATLAS approaches LGen's performance to within 10%.

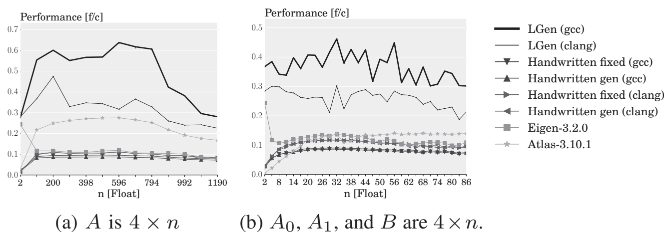
D. ARM1176

The ARM1176 is a scalar processor, for which optimizations such as tiling, loop unrolling, loop fusion, and loop exchange have a significant impact on the quality of the generated code. In all the experiments in Fig. 11 LGen is up to 4× faster than ATLAS, which is in all cases the best competitor. Drops in performance can be noticed for large values of n due to reaching the L1 data cache size (16 kB). Another general remark is that for all tested BLACs, LGen's generated code compiled with gcc is more efficient than the one compiled with clang. Finally, for large values of n the performance results of LGen are less stable because of the random search with a sample size that is relatively small compared to the large space of tiling options.

V. Related Work

Linear algebra libraries. Intel's Math Kernel Library (MKL) [4] and Integrated Performance Primitives (IPP) [8] are vendor libraries optimized for the Intel architectures. MKL implements the BLAS interface [1] and is optimized for large scale problems. IPP offers a subset of BLAS and other interfaces geared towards small-scale linear algebra computations. We used both as benchmarks. BLIS [16] is a framework for the instantiation of high-performance BLAS-like libraries, based on a set of micro-kernels that must be provided by the user. The studies presented in [17] show the results of using BLIS for a

[1]http://math-atlas.sourceforge.net/errata.html



(a) A is $4 \times n$ (b) A_0, A_1, and B are $4 \times n$.

Fig. 11. (a) $\alpha = x^T Ay$ and (b) $C = \alpha(A_0 + A_1)^T B + \beta C$ on ARM1176.

variety of architectures, including ARM Cortex-A9. Although the presented experiments did not involve vectorized code, BLIS appears to be competitive on this processor. LGen could be used for instantiating the needed micro-kernels.

Linear algebra generators. ATLAS [2] is a BLAS program generator that uses autotuning to tune optimization parameters (e.g., block sizes and unrolling factors) for large data. ATLAS provides support for both Intel and ARM processors. Eigen [10] is a linear algebra library based on C++ templates. Metaprogramming is used for compile-time optimizations like loop fusion and vectorization (SSE, NEON, and AltiVect ISAs). We used ATLAS and Eigen as benchmarks.

Optimization based on generic vector instructions. LLVM [18] provides generic vector instructions at the IR level. Our approach however, is closer in spirit to the work in [19] and [20], where generic vector instructions are defined at a higher level of abstraction than usual vector data types and are geared towards working with matrices. This enables the efficient handling of our domain of interest.

VI. Limitations

Here we list some of the current limitations of LGen.

Limited functionality. To date LGen supports only fixed-size BLACs on general matrices stored contiguously in memory. We are working on removing these restrictions.

Search strategies. The experimental results for ARM1176 showed that random search is far from optimal, since a relatively small sample size does not guarantee that good choices for the search parameters will be visited. We believe that there is significant potential in employing more sophisticated search strategies during the autotuning process, which is one possible direction for future research.

Aligned accesses. Although we proved that our alignment detection methodology is precise for our generated code, we could potentially achieve further improvement by (a) exposing more aligned memory accesses, e.g., introducing leftovers on both sides producing an effect similar to loop peeling in the resulting code, and (b) investigating techniques similar to [21] that replace unaligned accesses with aligned ones combined with shuffles.

VII. Conclusion

Efficient dense linear algebra code for small problem sizes is of crucial importance in various fields of computer science and engineering. With this paper we extend LGen, a domain-specific compiler that targets this type of functionalities, towards embedded processors. To do so, we extended LGen to support the NEON instruction set and then introduced a technique that eliminates unnecessary memory accesses and shuffle operations based on the use of generic load and store C-IR instructions. Furthermore, all aligned memory accesses in our generated code are guaranteed to be implemented using aligned intrinsics.

We evaluated LGen-generated code on four widely used embedded processors: Intel Atom, ARM Cortex-A8, ARM Cortex-A9, and ARM1176. The experimental results show that LGen performs in many cases better than well-established libraries, prior code generators, and general-purpose compilers.

References

[1] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, no. 1, pp. 1–17, 1988.

[2] C. R. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing (SC)*, 1998, pp. 1–27.

[3] E. Anderson *et al.*, *LAPACK Users' Guide (Third Ed.)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[4] Intel Corporation, "Intel Math Kernel Library," http://software.intel.com/en-us/intel-mkl, 2014.

[5] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler," in *International Symposium on Code Generation and Optimization (CGO)*, 2014, pp. 23–32.

[6] M. Püschel *et al.*, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[7] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*. Springer, 2011, ch. Spiral.

[8] Intel Corporation, "Intel Integrated Performance Primitives," http://software.intel.com/en-us/intel-ipp, 2014.

[9] J. G. Siek, I. Karlin, and E. R. Jessup, "Build to order linear algebra kernels," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[10] G. Guennebaud *et al.*, "Eigen," http://eigen.tuxfamily.org, 2014.

[11] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, July 2013.

[12] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *International Symposium on Programming*, 1976, pp. 106–130.

[13] P. Granger, "Static analysis of arithmetical congruences," *International Journal of Computer Mathematics*, vol. 30, no. 3-4, pp. 165–190, 1989.

[14] S. Larsen, E. Witchel, and S. P. Amarasinghe, "Increasing and detecting memory address congruence," in *Parallel Architectures and Compilation Techniques (PACT)*, 2002, pp. 18–29.

[15] N. Kyrtatas, "A Basic Linear Algebra Compiler for Embedded Processors," Master's thesis, ETH Zurich, 2014, http://spiral.ece.cmu.edu:8080/pub-spiral/abstract.jsp?id=180.

[16] F. G. V. Zee and R. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software (TOMS)*, 2013, accepted pending minor modifications.

[17] F. G. V. Zee *et al.*, "The BLIS framework: Experiments in portability," *ACM Transactions on Mathematical Software (TOMS)*, 2013, accepted pending modifications.

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Code Generation and Optimization (CGO)*, 2004, pp. 75–86.

[19] G. Ren, P. Wu, and D. Padua, "Optimizing data permutations for SIMD devices," in *Programming Language Design and Implementation (PLDI)*, 2006, pp. 118–131.

[20] F. Franchetti and M. Püschel, "Generating SIMD vectorized permutations," in *International Conference on Compiler Construction (CC)*, ser. Lecture Notes in Computer Science (LNCS), vol. 4959. Springer, 2008, pp. 116–131.

[21] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet SIMD code generation," in *Programming Language Design and Implementation (PLDI)*, 2013, pp. 127–138.