Proceedings of the Federated Conference on Computer Science and Information Systems pp. 663–672 DOI: 10.15439/2020F145 ISSN 2300-5963 ACSIS, Vol. 21

Evaluation of Open-Source Linear Algebra Libraries targeting ARM and RISC-V Architectures

Christian Fibich, Stefan Tauner, Peter Rössler, Martin Horauer
Dept. of Electronic Engineering, University of Applied Sciences Technikum Wien
Höchstädtpl. 6, 1200 Vienna, Austria
{fibich, tauner, roessler, horauer}@technikum-wien.at

***Abstract*—Basic Linear Algebra Subprograms (BLAS) has emerged as a de-facto standard interface for libraries providing linear algebra functionality. The advent of powerful devices for Internet of Things (IoT) nodes enables the reuse of existing BLAS implementations in these systems. This calls for a discerning evaluation of the properties of these libraries on embedded processors.**

This work benchmarks and discusses the performance and memory consumption of a wide range of unmodified open-source BLAS libraries. In comparison to related (but partly outdated) publications this evaluation covers the largest set of open-source BLAS libraries, considers memory consumption as well and distinctively focuses on Linux-capable embedded platforms (an ARM-based SoC that contains an SIMD accelerator and one of the first commercial embedded systems based on the emerging RISC-V architecture). Results show that especially for matrix operations and larger problem sizes, optimized BLAS implementations allow for significant performance gains when compared to pure C implementations. Furthermore, the ARM platform outperforms the RISC-V incarnation in our selection of tests.

***Index Terms*—Embedded Systems, Basic Linear Algebra Subprograms, BLAS, Benchmarks, ARM, RISC-V**

I. Introduction

EDGE computing – processing sensor data as close to their origin as possible – is a paradigm to de-centralize processing and storage in order to improve the scalability of IoT applications. More potent embedded CPUs allow more complex processing tasks, for example signal and image processing operations as well as inference (and even training) of neural networks. Vector or matrix operations are essential building blocks of the digital algorithms that are at the core of these applications. This raises the question whether it is viable to re-use and adapt proven mathematical software libraries written for server and desktop computers for embedded target platforms.

Netlib, a repository for open-source mathematical software run by AT&T, Bell Labs, University of Tennessee, and Oak Ridge National Laboratory, both maintains the BLAS specification document [1] and provides a reference implementation of this specification[1]. BLAS essentially describes a programming interface for three levels of algorithms: level 1 contains vector-vector operations, level 2 defines vector-matrix operations, and level 3 specifies matrix-matrix operations all for single- and double-precision real numbers, as well as for single- and double-precision complex numbers. The reference implementation itself does not contain optimized code – for example, matrix-multiplication is implemented using a simple iterative algorithm with three nested loops – it is, however, widely used as a performance baseline for implementers of optimized BLAS libraries. For example, implementations taking advantage of Single Instruction Multiple Data (SIMD) hardware extensions provided by modern CPUs as well as GPUs via interfaces such as CUDA and OpenCL are common.

Since many modern embedded applications rely on algorithms that mandate complex computations an evaluation whether some BLAS implementations can be re-used under resource constraints imposed by typical embedded platforms seems in demand. Unlike the usual target systems for BLAS libraries this work evaluates them on comparably small embedded systems that display significantly divergent characteristics due to architectural differences. For example, modern x86 CPUs can retire over 10 instructions per cycle per core under ideal circumstances while running at a multi-GHz clock rate and dozens of MB of caches. This is in stark contrast to embedded CPUs, which are often still non-speculative in-order architectures with sub-GHz frequencies. In particular, in this paper we evaluate different implementations of the BLAS specification targeting the ARM Cortex-A9 and RISC-V RV64GCSU architectures, respectively. Both share some similarities such as having RISC architectures, comparably low clock frequencies and other attributes often found in embedded systems. Both Instruction Set Architectures (ISAs) support extensions leading to a wide variety of implementations. We chose two representative examples for our tests that are described in more detail in Section IV.

Furthermore, it was the intention of the authors to present a more comprehensive overview and evaluation of existing open-source BLAS libraries, when compared to related work that is discussed in Section II. In Section III, the scope of this work – the evaluated libraries as well as the benchmark applications and CPU architectures that constitute the basis of

This work has been supported by the Austrian Federal Ministry for Digital and Economic Affairs (BM:DW) and the National Foundation for Research, Technology and Development as related to the Josef Ressel Center "Innovative Platforms for Electronic-Based Systems" (INES) managed by the Christian Doppler Society (CDG).
We thank Kapsch TrafficCom AG for their support of this research.

[1]https://www.netlib.org/blas/, last visited on 2020-06-27

these evaluations – is described. Section IV provides details on the specific build and run-time environments as well as employed metrics and measurement methodologies. Finally, the results of the evaluations are discussed in Section V before the paper concludes.

II. Related Work

Due to its widespread use, libraries implementing the BLAS specification have been compared in the past. For example, [2] presents a wrapper for multiple BLAS library implementations facilitating their interchangeable use along with an evaluation by way of several benchmarks. Similar evaluations are provided by the *BLIS* framework presented in [3] targeting the performance of several BLAS level 2 and 3 algorithms and comparing them with an optimized ISO C implementation.

The code generator *LGen* generates computational kernels for some BLAS operations with fixed-sized operands that are optimized for different target architectures and thus improve over generic implementations that employ varying operand sizes, see [4]. This approach was refined in [5] where the targets have been shifted to some high-end embedded processors.

Finally, *BLASFEO* (see also Section III-B) is a BLAS implementation that is intended to enable executing optimization algorithms on embedded targets. It is especially optimized for widely used embedded application processors (e.g., ARM Cortex-A7, -A9, -A15) as well as modern Intel and AMD architectures (e.g., Intel Haswell, Sandy Bridge). In [6], the authors compare their implementation to multiple other BLAS libraries. BLASFEO provides its own API that is better suited for small input sizes than standard BLAS as it reduces the overhead of aligning the matrices for internal processing. However, the standard BLAS API has been implemented coequally to the native API and evaluated on Intel and ARM microarchitectures [7].

TABLE I: An overview of BLAS Benchmarks

	FlexiBLAS	BLIS	LGen	BLASFEO		*This work*
	2013	2015	2015	2018	2020	2020
	[2]	[3]	[4], [5]	[6]	[7]	
ATLAS	✓	✓	✓			✓
BLASFEO				✓	✓	✓
BLIS		✓			✓	✓
Eigen			✓	✓		✓
Intel MKL	✓	✓	✓	✓	✓	
Ne10						✓
Netlib	✓					✓
OpenBLAS	✓	✓		✓	✓	✓
Performance	✓	✓	✓	✓	✓	✓
RAM Footprint		✓				✓
x86	✓	✓	✓	✓	✓	
ARM			✓	✓	✓	✓
RISC-V						✓

An overview of related BLAS evaluations is shown in Table I listing evaluated libraries, their scope, and the respective target architectures. The last column depicts the contribution of this work that focuses on embedded platforms, in particular targeting ARM Cortex-A9 and RISC-V. The focus on embedded systems ruled out Intel's Math Kernel Library (MKL) implementation that specifically targets x86 architectures. Instead, the performance of ARM's Ne10 library was included. Although this is not an implementation of the BLAS specification, it has an overlapping scope. The treatment of memory consumption of the libraries and the inclusion of RISC-V sets this work apart from past publications.

In addition to academic literature, several authors of the libraries evaluated in this work have published their own benchmark results. ATLAS provides timing data for the `DGEMM` (double precision generic matrix multiplication) BLAS function (dimensions between 1000 and 2000) in various library versions and CPU architectures (e.g., Intel, MIPS, SPARC)[2]. The results, however, are provided for up to ATLAS version 3.9.5, which is about 8 years old. The Eigen project provides benchmark results of various level-2 and level-3 BLAS functions, as well as more complex functions such as lower-upper factorization of matrices[3]. The results date from 2011 and are compared to GotoBLAS (an OpenBLAS predecessor), Intel MKL, and ATLAS 3.8.3. BLIS provides the most recent results, dating from 2020. Results are provided for server- and workstation-class CPUs (i.e., 10s of cores, ARM ThunderX2, Intel SkylakeX, Intel Haswell, AMD EPYC). BLAS level 3 functions are evaluated against Eigen, OpenBLAS, BLASFEO, and MKL. Different sets of results are available for large[4] and small matrices[5].

Unlike the publications discussed above this work examines BLAS libraries on comparably small embedded systems. Preliminary results of this evaluation were published in [8]. This work widens the scope of the analysis both by inclusion of the RISC-V architecture and the very recent BLASFEO linear algebra library. It is, to the best of our knowledge, the first work that evaluates the performance of general-purpose linear algebra libraries on the RISC-V architecture. Furthermore, it extends previous work with a more in-depth analysis of performance differences between the different libraries, especially in the dot product, matrix multiplication, and neural network benchmarks.

In the past 10+ years the use of linear algebra algorithms on heterogeneous systems with potent GPUs have been investigated for High-Performance Computing (HPC). Due to the limitations of the communication between host CPUs and GPUs these implementations (e.g., ViennaCL, cuBLAS) often use custom APIs.[6] Similar considerations affect the use of custom accelerators on programmable logic, which is often

[2]http://math-atlas.sourceforge.net/timing/, 2020-06-27

[3]https://eigen.tuxfamily.org/index.php?title=Benchmark, 2020-06-27

[4]https://github.com/flame/blis/blob/master/docs/Performance.md, 2020-06-27

[5]https://github.com/flame/blis/blob/master/docs/PerformanceSmall.md, 2020-06-27

[6]ViennaCL's API is derived from the C++ `Boost.uBLAS` library that is object-oriented and uses operator overloading. cuBLAS supports a BLAS-compatible interface but it is deprecated and requires additional manually initiated transfers, cf. https://docs.nvidia.com/cuda/cublas/index.html

facilitated by High-Level Synthesis (HLS).[7] For these reasons directly comparing the performance of BLAS libraries running on other hardware than general-purpose CPUs is out of scope of this paper.

III. Scope of the Evaluation

A. Target Hardware

Unlike in high-performance computing the market for embedded processors is less concentrated due to the wide variety of applications. Nevertheless (or possibly because of that) ARM has been the leading provider for embedded RISC IP cores in the past few decades offering a wide range of processors from tiny cores for microcontrollers to the base for high-end smartphones with multiple gigabytes of memory. Almost all of its recent CPUs (at least optionally) comprise an FPU capable of SIMD operations called *NEON*. We focus our work on the mid-range Cortex-A9 implementation of the well-established and widely used ARMv7 architecture as one of our targets.

In stark contrast to ARM the RISC-V architecture is quite young but has gained substantial interest from academia and industry alike.[8] One of the obvious reasons is that the use of its ISA is royalty-free and thus a wide range of implementation exists. However, as of summer 2020 the standardization of the ISA has only frozen the most significant parts but some specifications have not been finalized relevant to embedded systems in general (e.g., the Bit Manipulation Extension[9]) and this paper in particular (e.g., the Vector Extension[10]).

The ISA allows for a modular design by offering base specifications for unprivileged and privileged execution environments as well as standard (i.e., defined by the foundation) and custom ISA extensions. Designers can thus build cores tailored to the specific use cases. For example, a RISC-V core developed in the PULP project contains a dot-product accelerator as a non-standard vector extension [9]. This accelerator can calculate the dot product of two vectors comprising two 16-bit or four 8-bit integers.

B. Evaluated Libraries

This section provides a brief description of the distinctive features of each library under investigation. The libraries were evaluated in unmodified form, with the respective versions and release dates indicated in Table II.

The aim of the ATLAS (Automatically Tuned Linear Algebra Subprograms) project[11] is to provide implementations that are highly optimized for the particular target platform they will be used on. For this purpose, ATLAS contains many different variants of its kernels that suite best for particular properties of the target (e.g., cache line size, vector accelerators). ATLAS' build process is carried out on the actual target, requiring a C compiler on the target platform. During the build process, the kernel variants delivering the highest performance are determined empirically. Further details on the basic principles of this optimization and the ATLAS project can be found in [10]. In the results section of this work, the version of ATLAS built with default settings is referred to as "ATLAS". The "ATLAS-Neon" version was built with disabled IEEE-754 compatibility options, potentially allowing better tuning to the NEON vector processor in ARM CPUs. While newer *unstable* releases of ATLAS are available, the most recent *stable* release of ATLAS was evaluated in this work.

TABLE II: Summary of evaluated libraries

Library	Version / Commit	Release Date
ATLAS	3.10.3	2016-07-28
BLASFEO	Commit `d404e3471dbb`	2019-10-23
BLIS	Commit `bc16ec7d1e2a`	2019-09-23
Eigen	Commit `8e409c71423f`	2019-09-27
Ne10	Commit `1f059a764d0e`	2018-11-15
Netlib BLAS	Commit `b5dd8d4016f7`	2019-09-12
OpenBLAS	Commit `2beaa82c0508`	2019-10-09

OpenBLAS[12] is widely-used optimized implementation of the BLAS specification. Details on the especially optimized matrix-multiplication kernels part of OpenBLAS and its predecessor GotoBLAS can be found in [11]. OpenBLAS is implemented in C, but provides assembly implementations of performance-critical kernels for several CPU architectures and accelerators. Architectures for which such kernels exist include MIPS, ARMv6/v7/v8, x86, and Power8/9. However, the low-level kernels for ARMv7 relevant to this work only make use of the VFPv3 instruction set, and do not include specific support for the NEON SIMD engine.

BLASFEO [6] is a fairly new linear algebra library[13]. It aims at improving performance at small vector and matrix sizes (between 10s and 100s per dimension). The authors motivate this goal with optimization algorithms calculated on embedded devices themselves. BLASFEO consists of three independent implementations of linear algebra functionality: *Wrapper* which is a custom interface to standard BLAS libraries such as Netlib BLAS, *Reference* which is implemented in ANSI C and serves as a reference that is easily portable to new architectures, and *High-Performance*. The *High-Performance* version provides kernel implementations in assembly optimized for various architectures and common accelerators (e.g., SSE3, AVX, AVX2 available in x86 CPUs and VFPv3/VFPv4, NEON, and NEON2 found in ARM CPUs). To allow for maximum efficiency many assembler routines are inlined, which can be further enforced at compile time. BLASFEO uses an internal, aligned format for storing matrices and vectors, and an API that differs from BLAS. The latter difference is especially relevant to matrix multiplication: While BLAS provides the `xGEMM` functions that allow transposing

[7]For example FBLAS (https://github.com/spcl/FBLAS) or Xilinx' Vitis BLAS Library (https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-blas.htm

[8]https://riscv.org/members-at-a-glance/, 2020-06-27

[9]https://github.com/riscv/riscv-bitmanip, 2020-06-27

[10]https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc, 2020-06-27

[11]http://math-atlas.sourceforge.net, 2020-06-27

[12]https://www.openblas.net, 2020-06-27

[13]https://github.com/giaf/blasfeo, 2020-06-27

either of the two input matrices to be multiplied via function parameters, BLASFEO offers different functions for each of these cases called `blasfeo_sgemm_{nn|nt|tn|tt}()`. However, the authors have also implemented a standard-conforming interface (denoted *CBLAS* hereinafter) that covers most but not all BLAS functions [7]. As a consequence, BLASFEO was evaluated only in the SDOT and SGEMM benchmark by using its native and CBLAS API, respectively.

Another BLAS library investigated in this work is BLIS [3][14]. A main motivation for the originators of BLIS was to provide portability to new architectures but also to accelerators. This is done by using a set of target-specific kernels which implement the BLAS routines. A generic implementation exists by relying on compiler optimizations. Furthermore, ports are available for ARMv7 or ARMv8 architectures, Intel or Power7 as well as others. The ARMv7a port provides kernels for both single-precision as well as double-precision floating-point matrix multiplications utilizing NEON SIMD intrinsics. As a special feature, checks for errors related to, e.g., buffer sizes or matrix and vector dimensions are performed by BLIS. However, according to the description of the test suite[15], these checks may result in performance degradation, and for this reason all the error checks have been disabled for our benchmark evaluations.

The last library that has been considered for our benchmarks is Ne10[16] which includes functions for generic linear algebra but also for signal processing or image processing. Three versions of the Ne10 library are provided: a portable implementation using plain C, another implementation that makes use of NEON intrinsics, and finally, an implementation based on ARM assembly code. However, no BLAS interface is provided by Ne10. Furthermore, operations like generic matrix multiplication or generic dot product are not provided and therefore, the Ne10 vector multiplication kernel has been used to implement this kind of calculations for our work. Since the optimized version of Ne10 (based on ARM assembly implementations) was benchmarked, only results for ARM are available.

C. Benchmark Applications

Our first synthetic benchmark application is the dot product of two vectors which is an example for an often used linear algebra operation. It can be expressed by a BLAS level 1 operation called `[DS]DOT`. Since the dot product of two vectors can be implemented as a Multiply and Accumulate (MAC) operation it can be assumed that this operation is perfectly suited to transformations such as automatic vectorization or utilization of SIMD facilities. Our implementation performs a calculation of the dot product of two randomly selected values for a given number of iterations. The time required for each calculation is measured and at the end the mean value per `SDOT` function call is calculated. The vector length varies from 4 to 1500[17]. Note, that a generic dot product operation does not exist in the ARM Ne10 library and thus, vector multiplication plus iterative addition of the results are used in our Ne10 implementation.

The next synthetic benchmark application is matrix multiplication which is also an example for an often used linear algebra operation. It can be implemented by the BLAS level 3 operation `[DS]GEMM`. Our implementation performs a calculation of the product of two randomly selected square matrices for a given number of iterations and measures an average of the runtime. The dimensions of the square matrix used for our benchmark range from 4x4 to 1500x1500. Since a matrix multiplication operation does not exist in the ARM Ne10 library, vector operations have been used instead, for the Ne10 implementation.

Our final synthetic benchmark application is LINPACK-PC which is a C implementation of benchmarks for linear algebra operations provided by Netlib[18]. A number of BLAS operations such as `[DS]DOT`, `[DS]SCAL` (scaling a vector by a constant) or `I[DS]AMAX` (calculating the maximum element of a vector) are used by the LINPACK-PC benchmarks, operating on matrices or vectors with a maximum dimension of 200 to 201 elements. The performance of a series of operations is measured and are reported in units of MFLOPS.

To verify that our results hold in real-world applications too we employ a benchmark in the field of Artificial Neural Networks (ANNs). In detail, we made use of KANN[19], which is a C-based framework for constructing and training of artificial neural networks. Through the CBLAS interface KANN makes use of the BLAS operations `SAXPY` (vector addition) and `SGEMM`. Two out of the existing sample applications provided with KANN are used for our benchmarks: (1) *RNN* is basically a Recurrent Neural Network (RNN) using 64 neurons trained to add integer numbers, and (2) *MNIST-MLP* implements a MLP (Multi-Layer Perceptron) with 64 neurons to process data from the MNIST (Modified National Institute of Standards and Technology) database of handwritten digits. For both ANN benchmarks the training time as well as the interference time has been evaluated.

IV. Experimental Setup

A. ARM Cortex-A9 on Cyclone V SoC Development Board

An Intel/Altera Cyclone V SoC Development Kit[20] was used as an ARM-based embedded platform. This board contains a Cyclone V SoC FPGA that implements a hard-wired dual-core ARM Cortex-A9 CPU besides the programmable logic fabric. The usage of an FPGA board was motivated by the fact to generate BLAS accelerators by means of HLS as

[14]https://github.com/flame/blis, 2020-06-27

[15]https://github.com/flame/blis/blob/c665eb9b888ec7e41bd0a28c4c8ac4094d0a01b5/docs/Testsuite.md, 2020-06-27

[16]https://github.com/projectNe10/Ne10, 2020-06-27

[17]200k iterations are performed for vector dimensions 4–64, 100k iterations for 100–500 and 50k for dimensions 1000–1500

[18]https://www.netlib.org/benchmark/linpack-pc.c, 2020-06-27

[19]https://github.com/attractivechaos/kann, 2020-06-27

[20]https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-soc.html, 2020-06-27

future research work while for this work the device manufacturer's *Golden Hardware Reference Design*[21] was used. The ARM core implements a 32 KiB L1 data cache, a 32 KiB L1 instruction cache as well as a shared L2 cache (512 KiB). A VFPv3 floating-point unit and a NEON SIMD engine are also integrated on-chip [12] while the board comes with peripherals such as 1 GiB of external DDR3-1600 memory, a GBit Ethernet PHY and an SD card slot to boot a Yocto Linux (using a 3.9.0 kernel).

B. RV64GCSU on SiFive Unleashed Board

The second target is one of the first commercially available 64-bit RISC-V development boards featuring four `RV64GCSU` cores in the SiFive Freedom U540 SoC and has 8 GiB DDR4-2400 RAM on board. The ISA specification `RV64GCSU` is shorthand for `RV64IMAFDCSU` which stands for 64-bit registers, compressed multiplication, atomic, single- and double-precision floating-point instructions with support for supervisor and user mode in the RISC-V ISA standard. The HiFive Unleashed[22] does also feature Ethernet and boots from a (micro) SD card like the Cyclone V board. The cores in the U540 are based on SiFive's U54 with 32 KiB L1 cache for data and instructions, respectively, and share a common 2 MiB L2 cache. The manufacturer-provided buildroot Linux with a 4.15 kernel was used.

C. Build Environment

Our build environment for the libraries under investigation, Section III-B, was as follows (see Section V for additional notes):

- As compilers, `gcc` and `gfortran` from the GNU Compiler Collection (GCC) version 8.3.0 (from Debian 10) for ARM (`armhf` Hard-Float Application Binary Interface (ABI)) and 64-bit RISC-V, were used
- GCC's `-O3` flag was used to specify the desired level of optimization. This flag enables, for example, automatic vectorization and loop peeling.
- Moreover, `-ffast-math` has been used to allow specific non-IEEE754-compliant optimizations in order to fully exploit the ARM Neon engine
- All libraries, including libc, have been linked statically.
- Since some of the libraries described in Section III-B provide multi-code/multi-thread support while others do not (NETLIB or Ne10, for instance) only single-threaded versions were benchmarked to allow a fair comparison between the libraries. Results may be of interest for single-core systems.

During the ATLAS build flow, the build system adapts the kernels used to the target platform. This requires the build flow to be executed on the target platform itself, requiring a full toolchain installation. To enable benchmarking ATLAS on ARM, the GNU C compiler (Version 7.2.0) was built using crosstool-NG[23] with hard-float ABI and support for ARM NEON.

D. Measurement Mechanics

Both, the ARM and RISC-V based CPU were configured to run at 800 MHz to allow for direct comparability. However, the raw memory bandwidth available to the RISC-V SoC is 50% higher in theory. Performance and memory measurements were taken using facilities provided by the target system:

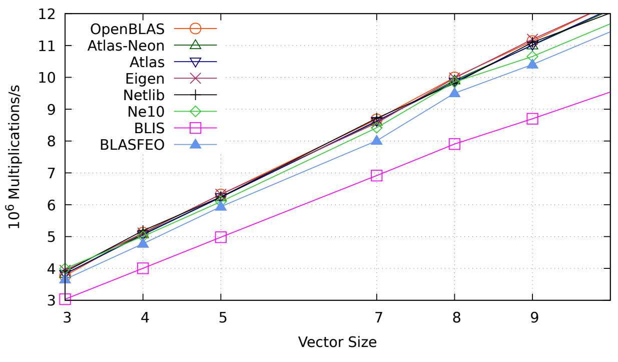
- The POSIX facility `clock_gettime()` was used to measure the time spent in the respective core BLAS functions of the synthetic dot product benchmark and the matrix multiplication benchmark. A timestamp was taken from the system-wide clock (via untampered `CLOCK_REALTIME`) before and after each call of the respective BLAS function. These times were accumulated and divided by the number of invocations in order to determine the average time per call.
- The LINPACK-PC library has a built-in mechanism to determine its performance in terms of Million Floating Point Operations per Second (MFLOPS) utilizing the ISO C `clock()` facility.
- The total execution time of the respective binaries was used as a performance metric in the KANN benchmarks. In the RNN inference and training case, as well as the MLP training case, the runtimes lie in the range of several minutes. Execution time of the executables was measured using GNU `time`.
- The memory footprint of the libraries has been determined over a snapshot of the application's Resident Set Size (RSS). This was done with Linux' `/proc/<PID>/smaps` interface that provides information about all memory mappings of a process. To capture realistic values including potential internal memories within the libraries these snapshots are taken with swapping disabled between the two middle iterations of the whole benchmark.
  Since the `smaps` interface is not provided by the RISC-V Linux kernel, only results for the ARM platform can be shown here (anyway, the CPU architecture should only have minor influence on the memory footprint). In order to avoid interference between the necessary instrumentation with the performance results, this was done using purpose-built executables for every library.

V. Results & Discussion

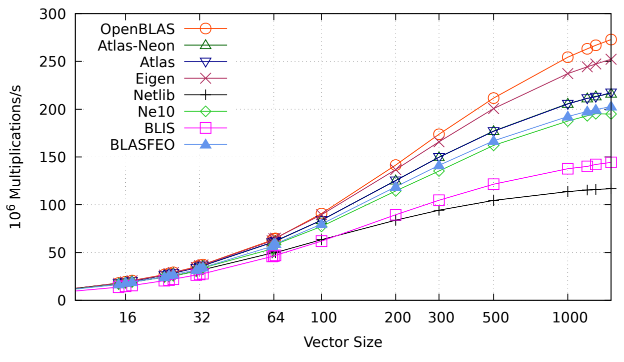
A. Fundamental Metrics: SDOT & SGEMM

In Figures 1 and 2 the performance results obtained from the single-precision dot product are shown. The results are split into smaller vector sizes (3-10, Figures 1a and 2a) and larger vector sizes (10-1500, Figures 1b and 2b) to retain readability at both ends of the evaluated spectrum of vector sizes. Figure 3 depicts the results of the matrix multiplication benchmarks. The y-axes of these figures are scaled to reflect the number

[21]https://rocketboards.org/foswiki/Documentation/GSRDGhrd, 2020-06-27
[22]https://www.sifive.com/boards/hifive-unleashed, 2020-06-27
[23]https://crosstool-ng.github.io, 2020-06-27

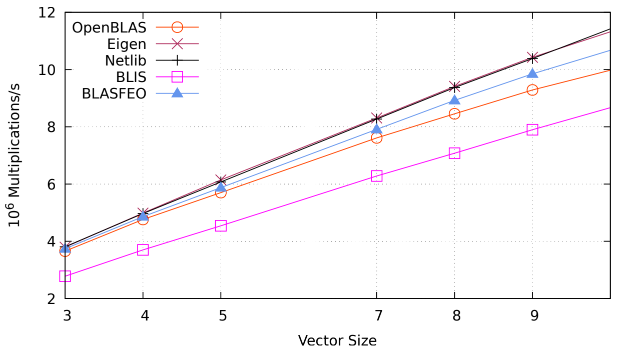


(a) Vector Sizes 3–10

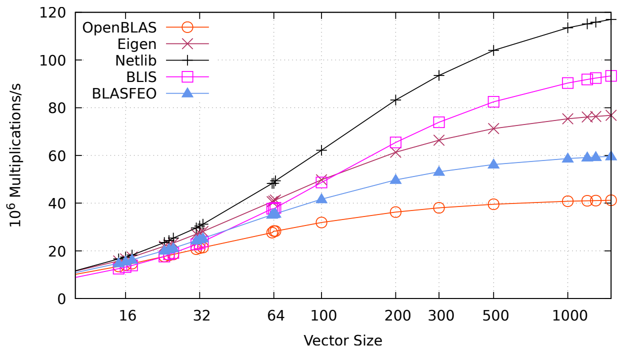


(b) Vector Sizes 10–1500

Fig. 1: `SDOT` performance in multiplications calculated per second on ARM platform



(a) Vector Sizes 3–10



(b) Vector Sizes 10–1500

Fig. 2: `SDOT` performance in multiplications calculated per second on RISC-V platform

of floating-point multiplications processed per second (i.e., a higher number corresponds to better performance). Since the number of multiplications is the determining factor this scaling allows for sharp discrimination between results. This metric was obtained by multiplying the number of `SDOT`/`SGEMM` calls per second with the respective multiplications per call (n for `SDOT` and $n^3 + 2n^2$ for `SGEMM` where n is the dimension of the input data).

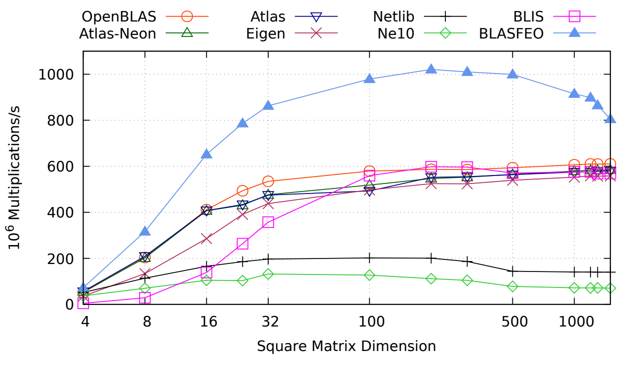
On ARM the widely-used OpenBLAS library performs well in both test cases. In the `SDOT` benchmark (Figure 1), OpenBLAS and Eigen are the fastest libraries, with effects getting increasingly noticeable with vectors longer than 100 elements. Especially at small vector lengths, BLIS falls far behind the other evaluated libraries. In the `SGEMM` benchmark (Figure 3a), OpenBLAS and ATLAS are clearly outperformed by BLASFEO, especially by small to medium dimensions but also at bigger ones (by about 25% at dimension 1500). The dip of BLASFEO's performance relative to the other libraries' that is visible for the largest matrices is probably because not all the data fits into the last level of cache and BLASFEO's lack of cache blocking. Similar results have been shown by its authors [6]. The performance of Eigen and BLIS is once again worse at smaller matrix sizes.

On RISC-V the situation for `SDOT` (Figure 2) is very different – almost inverse to that on ARM. The Netlib implementation shows the best results for all vector sizes, beating OpenBLAS and BLASFEO by almost a factor of 3 and 2 respectively, although the latter are the best-performing libraries in most of our other benchmarks no matter the architecture. BLIS is trailing the field at small vector sizes but performs relatively well for bigger vectors with multiple hundreds of elements where it becomes the second fastest library unlike in other benchmarks.

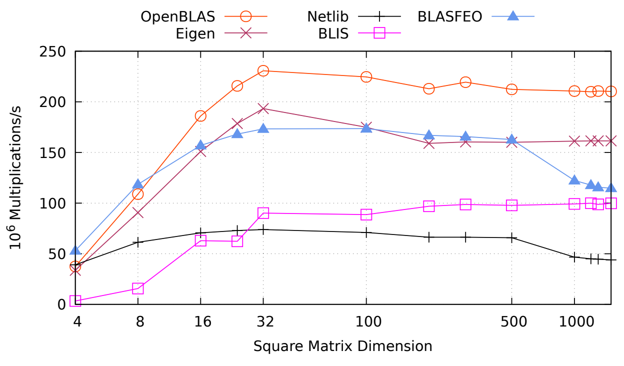
`SGEMM` on RISC-V (Figure 3b) looks similar to ARM at a much lower absolute performance level. Even the decreasing performance of BLASFEO starting at about dimension 500 is clearly visible. The most notable difference is that OpenBLAS is faster than BLASFEO for all matrices greater than 16×16. In addition to the absolute performance, different efficiency peaks of the evaluated libraries can be observed in the `SGEMM` benchmark results. The implementations of Netlib and Ne10 are most efficient at small matrix sizes, while the more optimized libraries OpenBLAS, ATLAS, Eigen, BLIS and BLASFEO are most efficient at sizes between 16 and 32.

B. Memory Usage

Figure 4 shows the memory consumption of the benchmark executables (statically linked with each BLAS library, respec-



(a) ARM platform



(b) RISC-V platform

Fig. 3: `SGEMM` performance in multiplications per second

tively) on the ARM platform. This data is provided for the smallest input size of the dot product and matrix multiplication benchmarks in order to assess the base memory consumption of each library. It subsumes both memory allocated for the input/output vectors/matrices and the library's own memory consumption (e.g., for buffers and other internal variables). As matrix dimensions grow larger, their size begins to dominate the application's memory consumption. This has been observed in the `SGEMM` benchmark starting with matrices of dimensions 200×200 and larger[24]. It can be seen that the memory consumption of the optimized ATLAS, BLASFEO and OpenBLAS libraries lies close to the NETLIB reference implementation, while Eigen and BLIS consumption is considerably higher.

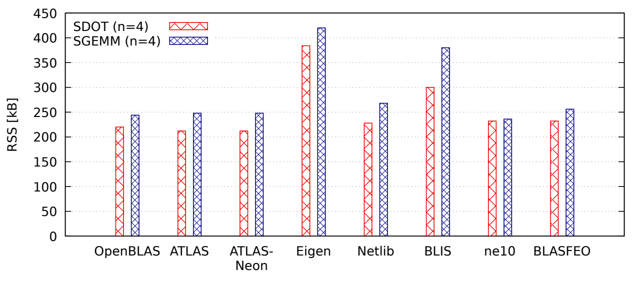


Fig. 4: Total application memory usage (RSS) on ARM

C. LINPACK

The results obtained by integrating the evaluated libraries into the LINPACK-PC benchmark are shown in Figure 5. LINPACK-PC calculates an MFLOPS metric from the number of iterations of its core algorithm completed in a set period of time. As Ne10 and ATLAS could not be benchmarked on the RISC-V platform, no bars for RISC-V are shown here. The results denoted by *LINPACK-** show the performance of the plain-C implementations of the used linear algebra operations that are provided by LINPACK-PC itself. The *"Unroll"* results were obtained from the same C source code that is provided in unrolled form by LINPACK. The former, however, was compiled with GCC's `-O3` optimizations turned on while the latter was compiled using GCC's `-ffast` compiler flag. *LINPACK-ffast* denotes a variant of the C code that has not been unrolled, but compiled with `-ffast` optimizations turned on.

These optimizations improve the performance of the standard C implementation even beyond the evaluated BLAS libraries on the ARM platform. In these cases, the entire program is available to the compiler as C source, and does not contain calls to an external library, which might enable more extensive optimization. On the RISC-V platform in contrast, there seems to be no benefit attached to these optimizations. Furthermore, the BLAS libraries cannot even improve on the performance of the basic LINPACK implementation, reaching only about 72–97% of its throughput.

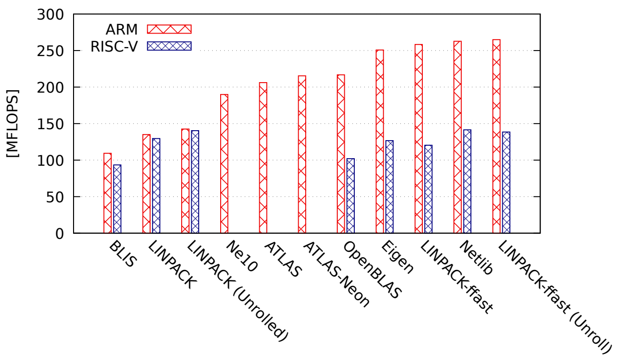


Fig. 5: Averaged LINPACK performance

D. Neural Networks

Figure 6 shows the speedup of the inference and training of two types of neural networks (*MLP* and *RNN*) relative to the C implementation provided by the KANN project itself (denoted as *"KANN"* in the graphs). For the training cases on the ARM platform, all evaluated libraries lead to a performance gain,

[24]Three single-precision 200×200 matrices (A, B, and C for SGEMM) consume $3 \times 200 \times 200 \times 4$ Byte = 480 kB, more than each libraries' base memory consumption.

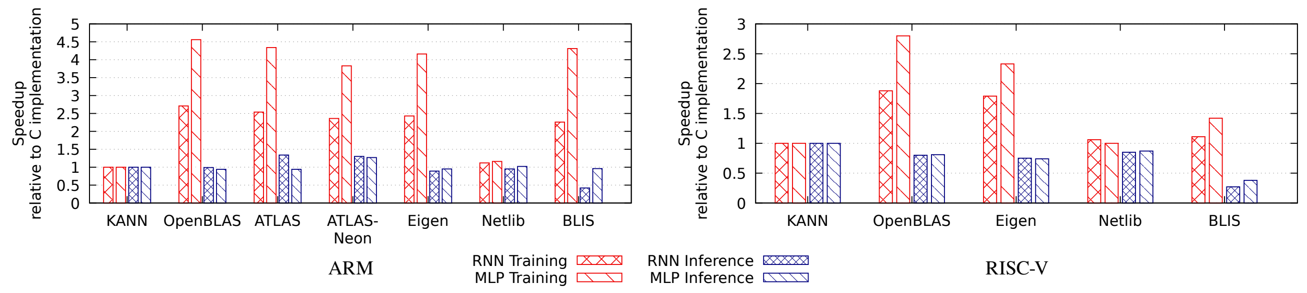


Fig. 6: Speedup of KANN benchmark relative to plain C implementation

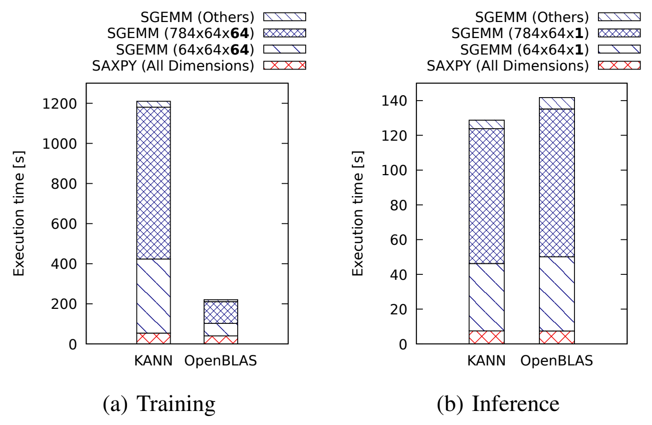


Fig. 7: Distribution of execution times of BLAS functions in KANN benchmark on ARM platform

even Netlib's reference implementation. To a lesser extent this is even true when training the neural networks on RISC-V although all libraries show significantly worse relative results.

The inference case behaves quite differently to the training as some libraries even lead to a slowdown of the application. Figure 7 depicts a plausible explanation. It compares the plain C implementation (*"KANN"*) to the fastest BLAS library implementation (OpenBLAS) regarding their cumulative execution time spent in BLAS functions when running on the ARM platform. For this purpose, the BLAS wrapper used by KANN was instrumented. The execution times of the MLP and RNN benchmarks for each BLAS function and dimension were summed up. This execution time is subdivided into calls to `SAXPY` of all dimensions, the two most prominent dimensions of `SGEMM`, and all other dimensions of `SGEMM`. In both training and inference, `SGEMM` accounts for the overwhelming majority of execution time spent in BLAS functions. In the inference case (Figure 7b), the two most prominent `SGEMM` operations act upon matrices where at least one dimension is 1. In these cases, OpenBLAS takes about 10% longer than the C implementation (supposably due to the overhead of the API). In contrast, the most prominent `SGEMM` operations in the training case (Figure 7a) act on significantly larger matrices, where OpenBLAS clearly outperforms the C implementation. BLIS is slower than all other libraries in the inference benchmarks, which may correlate with the lower performance at smaller matrix dimensions in the matrix multiplication benchmarks (see Figure 3a), and overall lower vector performance (see Figure 1).

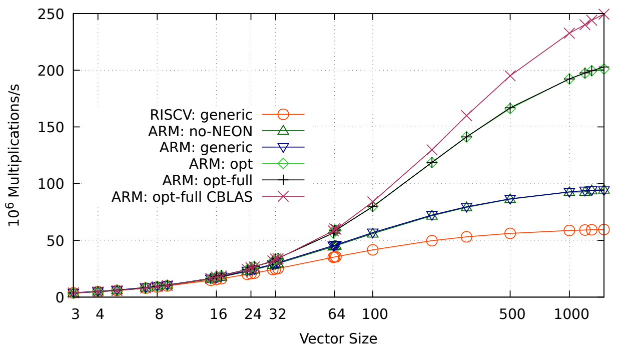
E. *BLASFEO*

Since BLASFEO is still in active development only the basic `SDOT` and `SGEMM` benchmarks were tested as explained in Section III-B. Instead of a broad application field we evaluated different build and execution options provided by BLASFEO:

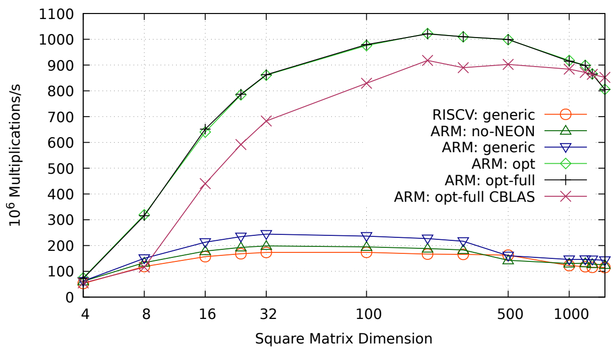
- A *generic* C implementation allows using BLASFEO even if no architecture-specific optimizations are available. We used this option to build the library for RISC-V where this is the case. For comparison two versions based on this option were created on ARM: One uses all available compiler optimizations (notably the use of NEON for SIMD instructions) and one uses the VFPU but without exploiting NEON (*no-NEON* in the respective figures).
- Two *optimized* builds exploiting BLASFEO's assembler implementations and native API were generated. While one (*opt*) reflects the default configuration of the library for supported architectures, the other one (*opt-full*) increases the amount of inlined assembly routines thus avoiding some function calls.
- Additionally, we evaluate the use of BLASFEO via its CBLAS-compatible API.

The results depicted in Figure 8 clearly show that the effort of manual optimization despite advancements in compilers still pays off. In the `SDOT` benchmark all versions including the one running on the RISC-V platform are within 10% for very small vector lengths (i.e. ≤ 10). At vector sizes of about 16–32 four groups begin to form. The generic implementation on RISC-V is clearly the slowest contender reaching only about 25% of the speed of the fastest solution for the largest vector size of 1500 elements. The second group consisting of the two compiler-optimized ARM implementations are able to get about twice as fast as the RISC-V library. Their almost equal result shows that GCC was not able to exploit NEON in the DOT application.

The two assembler implementations with the native BLASFEO API that form the third group calculate slightly over



(a) Million multiplications calculated per second by `SDOT`



(b) Million multiplications calculated per second by `SGEMM`

Fig. 8: Comparison of different build options of the BLASFEO library

200 million multiplications per second for very large vectors. These two variants perform almost the same in this benchmark as well as in `SGEMM` (cf. Figure 8b). Most interestingly though is the result of the CBLAS-enabled implementation that beats all others by well over 20%.

The results for `SGEMM` in Figure 8b are more refined. For 4×4 matrices the performance differences are relatively small showing well-working compiler optimizations. For bigger matrices, however, the hand-optimized implementations are by far superior (up to a factor of 6 for very large matrices). The CBLAS curve might be attributable to the API overhead that diminishes with bigger dimensions (and becomes negligible at about dimension 1000). In this benchmark the RISC-V implementation is able to almost match the compiler-optimized ARM versions. With smaller matrices the ARM versions are slightly better, also showing a small drawback when NEON is not used.

VI. Conclusion

In this work, a selection of linear algebra libraries providing the BLAS interface is evaluated. Compared to related work (which is, as explained in Section II, at least partly outdated) our evaluation covers the largest set of open-source BLAS libraries (see Table I), considers (unlike existing evaluations) memory consumption as well focuses on embedded platforms (to the best of our knowledge, it is the first work that evaluates the performance of general-purpose linear algebra libraries on the RISC-V architecture). Based on the results of these evaluations, as presented in Section V, we draw the following conclusions:

It can be seen from the LINPACK and the inference case in the KANN benchmark, that in some applications a plain C implementation may perform better (or at least as good) as an optimized linear algebra library. The compiler may be able to automatically optimize these applications to an extent that no significant additional performance can be gained by a hand-optimized implementation of the core mathematical operations. If, however, the application is more complex or uses core algorithms that cannot yet be automatically vectorized by the compiler, using optimized libraries can increase throughput. This can be seen in the matrix multiplication benchmark and in the training case in the KANN benchmark, both profiting from fast matrix-matrix multiplication offered by optimized BLAS libraries.

While in some cases a simple C implementation in combination with an optimizing compiler might suffice, BLAS offers a standardized interface to linear algebra functionality. This facilitates prototyping and modularization, allowing to replace the performance-critical parts of an application if required. In most of the benchmarks, the optimized BLAS libraries perform either better or comparable to a C implementation, thereby outweighing the slight performance advantage of non-standard C implementations.

From the results on the relatively new RISC-V target one can easily motivate further work on ISA extensions (similar to PULP's [9] and the proposed standard Vector Extension) since absolute performance levels are only a fraction of ARM's. They also show that generic improvements in libraries are possibly nullified by increased overhead or unanticipated working conditions and that architecture-specific optimizations are necessary to consistently improve results over straightforward implementations.

While performance is in most cases the distinctive metric to select computing libraries such as the ones described here, the aspect of power consumption is of similar importance to many embedded systems that often run on battery power and/or have to comply to strict thermal limits. In future work these potential contradictory properties should be commonly evaluated.

References

[1] BLAST Forum, "Basic Linear Algebra Subprograms Technical Forum Standard," https://netlib.org/blas/blast-forum/blas-report.pdf, 2020-06-27, University of Tennessee, Knoxville, Tennessee, Tech. Rep., 2001.

[2] M. Koehler and J. Saak, "FlexiBLAS - a flexible BLAS library with runtime exchangeable backends," https://www.netlib.org/lapack/lawnspdf/lawn284.pdf, 2020-06-27, LAPACK Working Notes, Tech. Rep., 2013.

[3] F. G. Van Zee and R. A. Van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 1–33, 2015. doi: 10.1145/2764454

[4] D. G. Spampinato and M. Püschel, “A basic linear algebra compiler,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’14. New York, NY, USA: Association for Computing Machinery, 2014. doi: 10.1145/2544137.2544155 p. 23–32.

[5] N. Kyrtatas and D. G. Spampinato, “A Basic Linear Algebra Compiler for Embedded Processors,” *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1054–1059, 2015. doi: 10.3929/ethz-a-010144458

[6] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl, “BLASFEO: Basic linear algebra subroutines for embedded optimization,” *ACM Trans. Math. Softw.*, vol. 44, no. 4, pp. 42:1–42:30, Jul. 2018. doi: 10.1145/3210754

[7] G. Frison, T. Sartor, A. Zanelli, and M. Diehl, “The BLAS API of BLASFEO: Optimizing performance for small matrices,” *ACM Transactions on Mathematical Software*, vol. 46, no. 2, May 2020. doi: 10.1145/3378671

[8] C. Fibich, S. Tauner, P. Rössler, M. Horauer, M. Krapfenbauer, M. Linauer, M. Matschnig, and H. Taucher, “Evaluation of open-source linear algebra libraries in embedded applications,” in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, June 2019. doi: 10.1109/MECO.2019.8760041 pp. 1–6.

[9] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, “Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017. doi: 10.1109/TVLSI.2017.2654506

[10] R. C. Whaley, A. Petitet, and J. J. Dongarra, “Automated empirical optimizations of software and the ATLAS project,” *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001. doi: 10.1016/S0167-8191(00)00087-9

[11] K. Goto and R. A. v. d. Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Transactions on Mathematical Software*, vol. 34, no. 3, pp. 12:1–12:25, May 2008. doi: 10.1145/1356052.1356053

[12] Altera Corporation, “cv_5v4: Cyclone V Hard Processor System Technical Reference Manual,” https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v4.pdf, 2020-06-27, July 2018.