Programmatically Reaching the Roof: Automated BLIS Kernel Generator for SVE and RVV

Stepan Nassyr, Kaveh Haghighi Mood, and Andreas Herten*

Novel System Architecture Design, Jülich Supercomputing Centre, Forschungszentrum Jülich

Abstract

A novel generator is used to generate highly optimized, architecture-specific microkernels for the BLIS library. The performance of potential GEMM microkernels is evaluated on Allwinner D1, EUPILOT VEC accelerator FPGA SDV and Fujitsu A64FX.

Introduction

In recent years, there has been a growing diversity of Instruction Set Architectures (ISAs) in the computing industry. x86 dominance is challenged by other ISAs such as ARM and RISC-V. The demand for energy-efficient computing and specialized accelerators provides opportunities for innovation at the ISA level.

This prospect also brings challenges for software developers, especially for the HPC community. Adapting software to a new ISA can be an elaborate and costly process. Performance-critical parts of libraries and HPC software must be ported, verified, optimized, and maintained for different ISAs with diverse extensions.

In this work, we introduce a tool that can generate compute kernels for different ISAs. The focus is on two Vector-Length-Agnostic (VLA) ISAs, ARM SVE and RISC-V RVV, in which the vector size is not fixed at compile time. The generator was applied to generate highly-optimized ARM SVE kernels for the A64FX processor. We use the same approach to generate RISC-V RVV 0.7.1 kernels for the Allwinner D1, a commercially available non-HPC RISC-V processor with support for a draft version of the RVV extension, as well as the FPGA SDV (RVV 0.7.1) of the in-development EUPILOT VEC accelerator and evaluate the performance.

In the following section, we introduce the BLIS library [1, 2] and the microkernel generator. In the subsequent section, we present a method to exploit the hardware and show the results for various kernel sizes.

Design and Implementation

BLAS (Basic Linear Algebra Subroutines) is a fundamental building block of any high-performing software stack, in HPC and beyond. The performance of a multitude of applications and numerous higher-level libraries, such as LAPACK, depends on this library. BLIS [1] is an open-source, superset implementation of BLAS with a modern design and a modern interface. BLIS can reach performance on par with or better than vendor BLAS implementation. The approach of the library to accelerating individual routines allows accelerating virtually all BLAS3 methods by implementing only a handful of optimized microkernels (GEMM, TRSM, GEMMTRSM, etc.) – see the extensive presentation in [1].

In BLIS, GEMM is implemented as a blocked algorithm with 6 loops. The microkernel implements the innermost loop and multiplies a $m_r \times k_c$ micropanel from the A matrix with a $k_c \times n_r$ micropanel from the B matrix. The microkernel keeps a $m_r \times n_r$ element microtile in registers and performs a series of k_c rank-1 updates into it. The result is scaled with a scalar α and added into to a microtile from the matrix C, scaled with β.

The microkernel size is $m_r \times n_r$. In our approach, m_r elements are stored in vector registers and the vector register size is not fixed. Because of this, we give a kernel size in vector registers for m_r and elements for n_r, for example 2Vx10 denotes a kernel with two vector registers (of given precision) and ten elements.

The code generated by our tool includes the $O(n^3)$ k_c rank-1 updates and accumulation, but not the $O(n^2)$ update of the C microtile. We call these partial microkernels *nanokernels*. The generator code is written in Python and generates C files with inline-assembly blocks. The generator produces well-optimized routines based on input parameters:

- Instruction set
- Size of the kernel
- Specific tuning parameters (i.e. order of loading operations, prefetching, k_c unroll factor, etc.)

For this work, we only focus on performance of different kernel sizes. The basic unit of computation

*Corresponding author: mailto:s.nassyr@fz-juelich.de

utilized by the generator is an FMA (Fused Multiply-Add) instruction. Modern architectures have started implementing other approaches that allow higher compute throughput, such as instructions that perform inner-products (Intel AMX, Apple AMX, ARM SME, newer NVIDIA and AMD GPU ISAs, ...). The development to utilize these types of instructions is planned for the future.
Standalone benchmarks are created to evaluate the performance of different kernel sizes with nanokernels. Measuring the performance of these nanokernels allows identifying microkernel sizes and structures that are likely to perform well on a given micro architecture.

Results and Discussion

We empirically measure the throughput of FMA instructions by creating a series of benchmarks that contain an assembly block with an increasing number of independent FMA instructions and measuring the clock cycles to execute the block. The empirical maximum for FP32 is 3.63 $\frac{\text{FLOP}}{cycle}$ on Allwinner D1, 31.08 $\frac{\text{FLOP}}{cycle}$ on the VEC SDV and 63.93 $\frac{\text{FLOP}}{cycle}$ on the A64FX. In the subsequent figures, the respective value is given as a horizontal line. We compare our throughput to these numbers instead of the theoretical values to be more robust against platform side-effects such as OS interference. The theoretical values are 4, 32 and 64 $\frac{\text{FLOP}}{cycle}$ respectively.

Figures 1, 2, and 3 show the performance of different kernel sizes on the Allwinner D1, VEC SDV, and A64FX, respectively, for single-precision GEMM microkernels. The respective best kernel reaches 85%, 94%, and almost 100% of each measured maximum throughput.

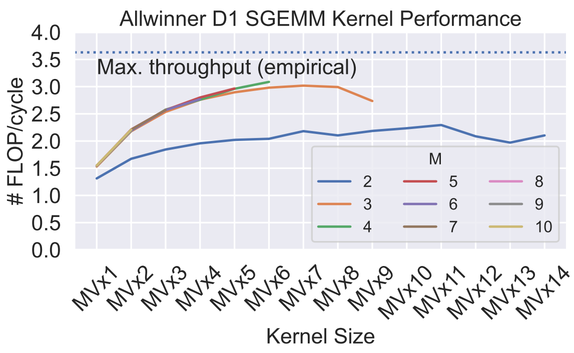

Figure 1: *Nanokernel performance on Allwinner D1*

Apart from the kernel size, further optimization opportunities relate to the internal structure of the code, such as the order of instructions, number of preloaded vectors, and more. The generator incorporates features to fine-tune these internal structures to match

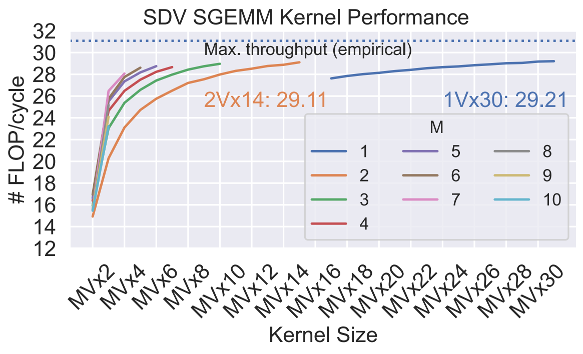

Figure 2: *Nanokernel performance on VEC FPGA SDV*

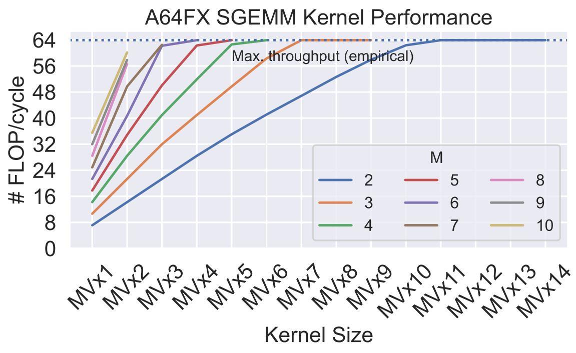

Figure 3: *Nanokernel performance on Fujitsu A64FX*

best the microarchitecture and its quirks.

We see excellent efficiency with the code generation for the A64FX, which is based on previous work done in [3] and is very mature. For the SDV, we expect a similar level of efficiency after incorporating further optimizations respecting architectural details. For the D1 we observed changes in performance depending on the order in which vector registers are accessed, suggesting that further optimization is also possible, however this is is a non-HPC grade processor as is evident by the max. throughput.

References

[1] Field G. Van Zee and Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Transactions on Mathematical Software* 41.3 (June 2015), 14:1–14:33. URL: `https://doi.acm.org/10.1145/2764454`.

[2] Field G. Van Zee et al. "The BLIS Framework: Experiments in Portability". In: *ACM Transactions on Mathematical Software* 42.2 (June 2016), 12:1–12:19. URL: `https://doi.acm.org/10.1145/2755561`.

[3] Stepan Nassyr. "BLIS on A64FX - Optimization Steps and Results". In: ISC High Performance 2021 - AHUG (ARM HPC User Group) workshop, Online (Germany), 24 Jun 2021 - 2 Jul 2021. June 24, 2021. URL: `https://juser.fz-juelich.de/record/905609`.