ETH zürich

A Basic Linear Algebra Compiler for Embedded Processors

Master Thesis

Nikolaos Kyrtatas

April 15, 2014

Advisors: Prof. Dr. Markus Püschel, Daniele Spampinato

Department of Computer Science, ETH Zürich

To my grandfather,
who understands computers better than me.

Abstract

In many fields, such as signal processing, control, and graphics, there is a significant demand for efficient Dense Linear Algebra (DLA) code for embedded devices. At the same time, code generators have proved to be useful for generating fast DLA code for general-purpose computers. An example is LGen, a research compiler designed after Spiral for basic linear algebra computations of fixed size. In this thesis, we extend LGen towards four processors that are widely used in the embedded ecosystem: Intel Atom, ARM Cortex-A8, ARM Cortex-A9, and ARM1176. For this purpose, we introduce into the LGen methodology a set of optimizations that take into account the specific limitations and capabilities of these processors, aiming at generating highly optimized code for them. An extensive set of experiments run on our target platforms shows that the new version of LGen produces code that performs better than well-established, commercial and non-commercial libraries (Intel MKL and Intel IPP), software generators (Eigen and ATLAS), and compilers (icc, gcc, and clang).

The large number of experiments that we conducted on the four investigated processors were executed using Mediator, a web-based middleware that was developed as part of this thesis. Mediator offers a RESTful interface that makes possible the simultaneous execution of experiments on multiple SSH-accessible devices by multiple users. More specifically, it guarantees mutual exclusion of the experiments running on a specific core, while at the same time applies load balancing over the cores of a device. On top of that, it also provides an easy-to-use mechanism for retrieving performance metrics on a variety of architectures with minimal user involvement.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

To begin with, I would like to deeply thank Prof. Markus Püschel, whose insightful guidance combined with his constant enthusiasm and good mood made this thesis a most rewarding experience for me. From the very first lecture of his that I attended until the last days of my thesis, he has been a source of knowledge and inspiration.

Furthermore, I am immensely grateful to Daniele Spampinato, my direct supervisor and at the same time my close teammate throughout my thesis project. I deeply appreciate the way he generously and patiently shared his knowledge and work with me, while at the same time being always open to my comments and ideas. His constant support and sincere willingness to help me, even at times when his busy schedule was making it very hard, were remarkable. I am very lucky to have worked with him and I hope I have given him back at least a small part of what I have received from him.

My warmest thanks also go to the remaining members of the Advanced Computing Laboratory, who made me feel part of the team from the very first moment. In particular, I would like to thank Georg Ofenbeck for his remarks about the Mediator chapter of this thesis and Alen Stojanov for all the fruitful discussions about Mediator, which largely determined its current form.

Leaving the ETH environment, many special thanks go to my father for his painstaking efforts in proofreading every draft I was sending to him, always providing me with very useful comments and advice. Finally, I would like to thank my two good friends Panos and Paul for their support, encouragement, and understanding during the last months.

Chapter 1

Introduction

1.1 Overview

The optimization of Dense Linear Algebra (DLA) code is one of the most critical tasks necessary to achieve high performance on modern computers. DLA is at the heart of many algorithms across a variety of domains, such as signal processing, computer graphics, control, and scientific computing. Decades of research have led to mature understanding of the problem, resulting in the performance-oriented design principles adopted by high performance libraries, code generators, and compilers. However, most of these tools are geared towards large matrix sizes (in the order of hundreds or more) and fail to support applications that require efficient small scale computations that are ubiquitous in many fields outside high performance computing.

LGen [34] is a research compiler that targets small Basic Linear Algebra Computations (BLACs) with fixed size on x86 architectures, optionally with SSE3, SSE4.1 or AVX vectorization extensions. However, LGen currently lacks support for embedded processors, which are gaining increasing importance due to their use in devices such as smartphones, tablets, nettops, and media centres.

1.2 Goal of the Thesis

The main goal of this thesis is to extend the backend of LGen with support for embedded processors. In this direction, we will investigate four processors that are widely used in embedded and mobile devices, namely Intel Atom, ARM Cortex-A8, ARM Cortex-A9, and ARM1176. Besides the extension of LGen's backend, another goal of this thesis is the development of a software tool that facilitates cross-compilation in the context of LGen. LGen is based on autotuning, which requires the execution of generated code versions on the target device throughout the compilation process. This tool should make possible the decoupling of the location of LGen from the location of the target device, allowing the remote execution of experiments on the latter. On top of that, it should allow the simultaneous execution of performance experiments on multiple devices by multiple users in a way that is not tied to LGen.

1.3 Contributions of the Thesis

The main contributions of this thesis can be summarized as follows:

- Modification of the LGen backend for Intel Atom, which supports the SSSE3 SIMD extension of the x86 ISA.
- Extension of LGen for the ARM Cortex-A8 and Cortex-A9 processors, which support the NEON SIMD extension of the ARMv7 ISA.
- Addition of a set of optimizations to LGen. More specifically:
    1. Generic memory access instructions (relevant for all processors).
    2. Detection of alignment of memory accesses (relevant for Intel Atom).
    3. A new matrix-vector multiplication approach that is more efficient than the existing one for most x86 architectures (Intel Atom included).
    4. Implementation of new computational building blocks (called ν-BLACs) for the NEON-extended ARM architectures that diverge from the approach followed by the existing building blocks for x86 in ways that are tailored to the characteristics of the Cortex-A8 and Cortex-A9 microarchitectures.
- Experimental evaluation of the effectiveness of the aforementioned extensions and optimizations.
- Development of Mediator, a middleware that facilitates the execution of performance experiments on multiple devices by multiple users.

1.4 Related Work

We can identify at least three approaches currently considered for the production of efficient DLA code. A first approach is the development of handwritten linear algebra libraries that are highly optimized for specific architectures. A second approach is the development of software generators for the automatic production of optimized DLA functions and libraries. Finally, a third approach is the use of optimizing compilers that apply a set of code transformation and optimization techniques in order to translate a (possibly naive) source code implementation into efficient binary code. In particular, auto-vectorization and the way it handles memory alignment are often critical for achieving high performance. Despite the large amount of progress that has been done in the development of optimizing compilers, [28] and the results of this thesis show that the auto-vectorization capabilities of general-purpose compilers are limited compared to what can be achieved through a domain-specific approach.

Linear algebra libraries. Various commercial and non-commercial libraries exist that target a wide range of architectures. Intel's Math Kernel Library (MKL) [8] and Integrated Performance Primitives (IPP) [7] are two of them, specific for the Intel x86 architecture. MKL implements the Basic Linear Algebra Subroutines (BLAS) interface [12] and is optimized for large scale problems, providing little support for small sizes (see [25] and results of this thesis). IPP offers functionalities covering a wide spectrum of domains, among which small-scale linear algebra computations that are not restricted to a BLAS interface. In the experiments that we conducted, we compete against both MKL and IPP.

BLIS [39] is a framework for the instantiation of high-performance BLAS-like libraries. All functionalities supported by BLIS are based on a set of micro-kernels that must be provided by the user. High performance can by achieved on a specific platform only if the micro-kernels have been properly optimized for it. The interface of BLIS is a superset of the BLAS functionality. The studies presented in [38] show the results of using BLIS for a variety of architectures, including ARM Cortex-A9. Although the presented experiments did not involve vectorized code at all, BLIS appears to be competitive on this processor, outperforming ATLAS for some of the tested computations.

Linear algebra generators. Autotuning is a well-established methodology for automatic code generation [5, 31, 34, 36], that involves the generation and execution of several code versions. The executions are monitored according to some predefined metrics (e.g., runtime), which are used to finally select the best code version. PHiPAC [5] and ATLAS [36] use this approach in order to select appropriate values for parameters like block sizes, unrolling factors, and loop order. Both of them focus on BLAS for large-size data. ATLAS provides support for both Intel and NEON-enabled ARM processors and it is included in our set of competitors.

Spiral [31, 30] is a software generator that strongly influenced the design of LGen. Spiral was initially devoted to the generation of high performance code for the domain of linear transforms. Later, with the introduction of the Operator Language (OL) [16], the system was extended to support functionalities outside the transform domain. OL is a point-free mathematical language for expressing algorithms, such as matrix-matrix multiplication, at a high abstraction level. Algorithms expressed using OL are restructured through the recursive application of a set of rules. This process results in the creation of a search space of algorithms that are candidates for the implementation of the desired kernel. The role of OL in Spiral is very similar to the role of LL in LGen, which is described in Section 2.1.2.

FLAME provides a framework for the automatic derivation of DLA algorithms [21, 4]. Code generation with FLAME assumes the existence of an efficient BLAS library.

Eigen [20] is a linear algebra library based on C++ templates. It uses metaprogramming to apply compile-time code optimizations like loop fusion, loop peeling, loop unrolling, and vectorization supporting the SSE 2/3/4, NEON, and AltiVect ISA extensions. Eigen is one of the libraries that LGen competes against in our experiments. Other libraries following a template-based approach for linear algebra code generation are MTL [33], uBLAS [35] (part of the Boost C++ library suite), and Armadillo [32].

Alignment for SIMD instructions. One important issue when vectorizing code for SIMD architectures is handling memory alignment, since for some vector architectures unaligned memory accesses are considerably slower than aligned ones. The authors of [27] present a two-step methodology for optimizing the performance of a program in terms of memory alignment: At first they apply a set of transformations (mainly loop peeling) in order to increase the amount of aligned memory accesses. Then they apply an abstract interpretation analysis similar to the one described in Section 3.2 of this thesis, in order to detect as many aligned accesses as possible. However, the benefit of loop peeling is low when a loop contains various memory accesses with different alignments.

A more sophisticated approach that handles situations where loop peeling is inadequate for improving alignment is presented in [14]. The authors propose a methodology specific for loops containing unit-stride accesses, based on the representation of memory accesses as streams. The manipulation of these streams is performed using a set of operators, resulting in what the authors call a *data reorganization graph*, which can then be translated into efficient SIMD code that involves aligned-only memory accesses combined with a limited amount of shuffle operations for reorganizing data in registers. The approach of [14] is extended in [37] so that it can handle conversions between datatypes of different size and arrays whose alignment is unknown at compile time.

The recent work of [26] introduces a SIMD code generation methodology, which combines the polyhedral compilation approach [29] with the codelet compiler from Spiral [31, 30]. Part of this methodology is dedicated to the elimination of unaligned memory accesses, which is mainly done by loop unrolling, followed by the replacement of unaligned accesses with aligned ones and vector shifts. On top of that, strength reduction and common subexpression elimination are applied in order to limit the overhead introduced by the shifts. Finally, whenever alignment is not known at compile time, multiple code versions are generated depending on different alignment assumptions. The version that is going to be executed is chosen at runtime using dynamic alignment checks.

1.5 Organization of the Thesis

The remainder of this document is organised as follows: Chapter 2 presents the background knowledge that the main contribution of this thesis is based on. Chapter 3 describes the four optimizations introduced in LGen. In Chapter 4 we present Mediator. In Chapter 5 we show and discuss the results of the most important experiments that we executed on the four target processors. Finally, in Chapter 6 we summarize the main contributions of this thesis, we discuss its limitations, and we list some possible future extensions.

Chapter 2

Background

2.1 LGen

This section gives a brief overview of LGen based on the work presented in [34]. The information contained in this section corresponds to the state of LGen at the commencement of this thesis and can be used as background knowledge for Chapter 3.

2.1.1 Overview

LGen is a compiler for small-scale, fixed-size, basic linear algebra computations (BLACs). By BLACs we refer to computations on matrices, vectors, and scalars that consist of matrix addition, matrix multiplication, matrix transposition, and scalar multiplication. For the rest of this thesis, matrices will be denoted as $A, B, \ldots$, vectors as $x, y, \ldots$, and scalars as $\alpha, \beta, \ldots$.

The input to LGen is a BLAC expressed as an equation of the form

$$y = \alpha A x + \beta y, \tag{2.1}$$

together with a specification of the sizes of all entities involved in it. For example, in this case we could have that A is a 10×20 matrix, x, y are vectors of length 20, and α, β are scalars. The output of LGen is a C function (also referred to as kernel) that implements this computation, optionally using SIMD instructions.

Before being converted into C code, an input BLAC like the one in (2.1) is processed at three different levels of abstraction, each one characterized by a language and a set of transformations. This process is depicted in Fig. 2.1. At first, the input BLAC is translated into a Linear algebra Language (LL) expression. LL is a domain-specific language (DSL) appropriate for tiling transformations. At the immediately lower level, the computation is translated into another DSL called Σ-LL, which is a generalization of Σ-SPL

[17] that has been used in Spiral for signal transforms. Σ-LL is a purely mathematical DSL that makes access patterns and loops explicit and is appropriate for the application of optimizations such as loop fusion and loop exchange. Next, the computation is converted into a C-like intermediate representation (C-IR). At this level, optimizations like loop unrolling, scalar replacement, and conversion into static single assignment (SSA) form are applied. Finally, the C-IR code is unparsed to C code that can be executed on the target platform. The LGen compilation process also includes a feedback loop over these three levels, which is used for autotuning.

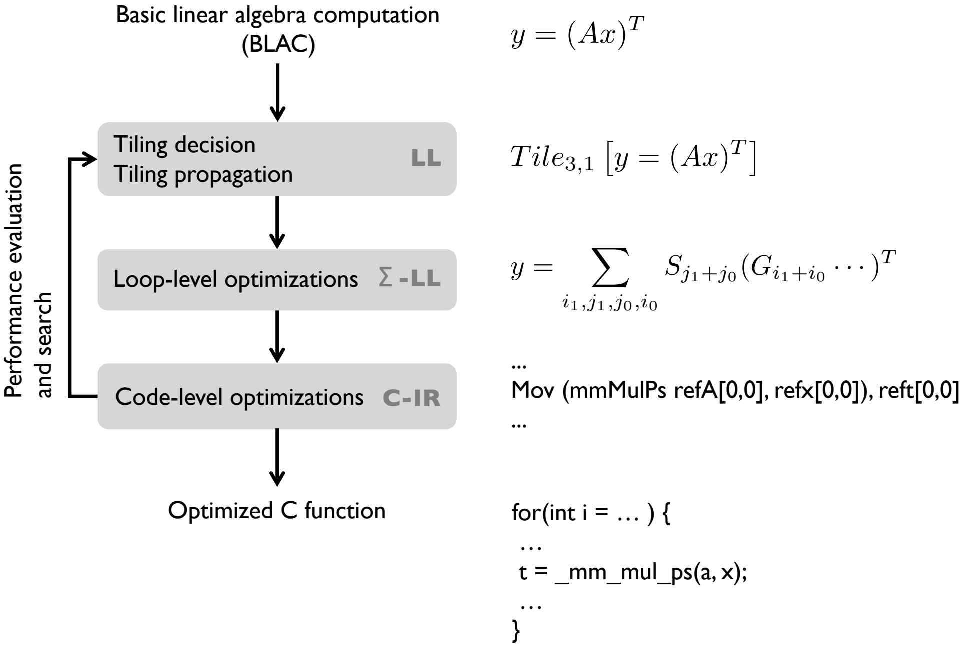

Basic linear algebra computation (BLAC)
$y = (Ax)^T$
Tiling decision
Tiling propagation
LL
$Tile_{3,1}\left[y = (Ax)^T\right]$
Performance evaluation and search
Loop-level optimizations
Σ -LL
$y = \sum_{i_1,j_1,j_0,i_0} S_{j_1+j_0}(G_{i_1+i_0}\cdots)^T$
Code-level optimizations
C-IR
...
Mov (mmMulPs refA[0,0], refx[0,0]), reft[0,0]
...
Optimized C function
for(int i = …) {
…
t = _mm_mul_ps(a, x);
…
}

Figure 2.1: LGen architecture (source: [34]).

Although LGen can generate code for any specified sizes of matrices, the optimizations applied by the LGen methodology are geared towards small-size BLACs that involve cache-resident matrices.

2.1.2 LL

Let's consider the following input BLAC:

$$C = AB, \tag{2.2}$$

with A having size 4×16 and B having size 16×4.

The input is already expressed using LL operators, so the transition to the LL level is trivial. Then, various tiling transformations are applied. The application of these transformations is decided by a predefined set of rules that take into account the operations involved in the expression (e.g., assignment and matrix multiplication), the hardware characteristics of the target platform (e.g., number of registers and size of instruction cache), and various configuration settings (e.g., vectorization) in order to decide on the appropriate tiling sizes. For example, a tiled version of (2.2) can be expressed as:

$$[C]_{2,4} = [A]_{2,8}[B]_{8,4}, \tag{2.3}$$

where the subscripts next to a matrix show the tile size that corresponds to it. For example, after applying this tiling, A consists of two rows of tiles, each one containing two 2×8 tiles.

In case the vectorization setting is enabled, the first level of tiling targets vectorization. Possible tile sizes at this level are $\nu \times \nu$, $\nu \times 1$ and $1 \times \nu$, where ν is the vector register length (e.g., 4 floats for SSE and NEON and 8 floats for AVX). Tiles of such sizes enable the introduction of predefined handwritten vector code snippets (called ν-BLACs) during the translation from Σ-LL to C-IR (see Section 2.1.4). Outer levels of tiling are also allowed and typically aim at reusing the contents of registers. Moreover, unrolling the innermost loops may help in exposing more instruction-level parallelism and thus improve performance. In a possible future extension of LGen that supports BLACs of larger sizes, outer levels of tiling may be used for reusing blocks of higher levels of the memory hierarchy (e.g., L1 cache and L2 cache).

An important restriction of LGen regarding multilevel tiling is that we can introduce leftovers in at most one level of tiling. For example, if we have a 30×4 matrix A and $\nu = 4$, after applying the first level of tiling we will get seven 4×4 tiles and one 2×4 leftover tile. Since seven is a prime number, we cannot further tile without introducing more leftovers, which results to an outer level of (pseudo-)tiling with tile size 1×1.

2.1.3 Σ-LL

An LL expression, such as (2.3), is translated into Σ-LL by converting operations on tiled matrices into sums over matrices. A Σ-LL expression makes access patterns and loops explicit by means of *gather* and *scatter* matrices, and summations, in order to facilitate the application of further transformations like loop merging and loop exchange.

Gather and scatter matrices are used to extract or embed submatrices from or to larger matrices, respectively. For example, given a 4×4 matrix A, we can extract its upper left 2×2 submatrix by multiplying A with the gather matrices G_L and G_R from left and right, respectively:

$$A(0:1,0:1) = G_L A G_R, \qquad G_L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, G_R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Similarly, we can insert a 2 × 2 matrix B into a 4 × 4 matrix A by using two scatter matrices $S_L = G_R$, $S_R = G_L$ as

$$A = S_L B S_R.$$

Using gather and scatter matrices, the LL expression (2.3) can be translated to Σ-LL as

$$C = \sum_{i=0,2}^{3} \sum_{j=0,4}^{3} \sum_{k=0,8}^{15} S_i(G_i A G_k) S_k S_k (G_k B G_j) S_j, \tag{2.4}$$

where G_x, $x \in \{i, j, k\}$ is a gather matrix targeting the submatrix starting from row or column x, depending on whether it is multiplied by another matrix from left or right, respectively. The same holds for the scatter matrices S_x. For the sake of simplicity, the number of rows or columns that such a matrix extracts or embeds is implied by the step of the corresponding summation (i.e. the number after the comma in the subscript of the $\sum$ symbol) and therefore it is omitted from the notation. Similarly, the number of rows or columns of the matrix that such a matrix extracts from or embeds to is implied by the upper bounds of the corresponding summations. Expression (2.4) is graphically depicted in Fig. 2.2.

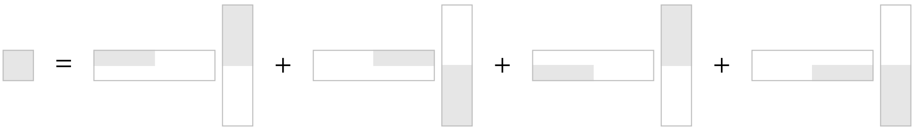


Figure 2.2: Illustration of the Σ-LL expression (2.4). The white regions contain zeros and were created using scatters.

2.1.4 C-IR

A Σ-LL expression is translated into C-IR code for the application of further optimizations, such as loop unrolling, scalar replacement, and conversion to SSA form. The translation from Σ-LL to C-IR is straightforward in the case of scalar code generation, while it is more complex in the case of vector code generation. Since all the optimizations introduced by this thesis apply to vector code generation, we will discuss only this scenario in the remaining of this section.

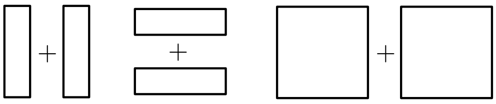
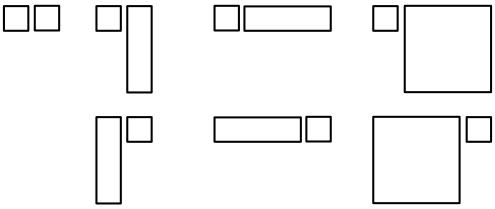
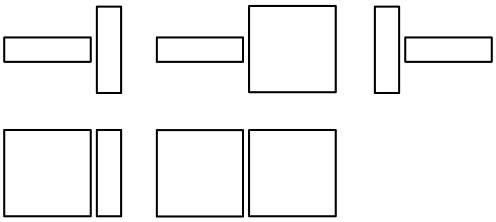
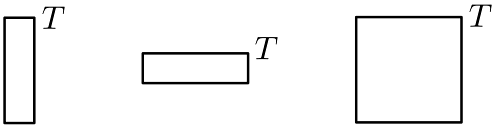
Operator	Required ν-BLACs
Addition (3 ν-BLACs)	
Scalar Multiplication (7 ν-BLACs)	
Matrix Multiplication (5 ν-BLACs)	
Transposition (3 ν-BLACs)	

Table 2.1: 18 required ν-BLACs for vectorization (source: [34]).

As it was mentioned in Section 2.1.2, if the vectorization setting is enabled, the inner level of tiling targets vectorization. During the translation from Σ-LL to C-IR, the loop that corresponds to the computation within a tile is fully unrolled and replaced by handwritten C-IR codelets, called ν-BLACs. Each ν-BLAC contains the implementation of a simple linear algebra operation for a specific vector ISA. Based on the four linear algebra operations that are supported by LGen, there are 18 different ν-BLACs, presented in Table 2.1. For each vector ISA that LGen supports, a special version of these 18 ν-BLACs has to be implemented.

Apart from the ν-BLACs, there are two more sets of handwritten codelets, called *Loader* and *Storer*. The Loader is responsible for moving a tile of a matrix into a ν-sized matrix that can be given as input to a ν-BLAC. Similarly, the Storer is responsible for embedding the result of a ν-BLAC into a tile of a matrix. In case the size of a matrix dimension is not divisible by ν, after applying tiling there will be some leftover tiles, with size smaller than ν, along the edges of the matrix. Operations on these tiles do not conform to any ν-BLAC. In order to generate vectorized code for these smaller BLACs, the leftover tiles are packed into appropriate ν-sized matrices by the Loader before being processed by the corresponding ν-BLAC. After the ν-BLAC computation is finished, the results are unpacked into a tile with the appropriate size by the Storer. In other words, the Loader and Storer codelets can be regarded as wrappers of ν-BLACs in the case of computations involving leftover tiles.

All codelets of the Loader, Storer, and ν-BLACs 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. The computation consists of chains of codelets, where each codelet within a chain loads data from the memory location that the previous codelet in that chain stored its results. At the one end of a chain there are the codelets that load data from the input arrays passed as arguments to the generated kernel and at the other end of a chain there are the codelets that store data to the output arrays of the kernel. Data between two consecutive codelets of a chain are stored in a local array that has been allocated from within the kernel. Fig. 2.3 shows one of the computation chains that are part of the computation $D = (A + B) + C$, with A, B, C, D being 8×6 matrices and $\nu = 4$. More specifically, this computation chain adds three leftover tiles of the matrices A, B, C and stores the result in the corresponding tile of matrix D.

In the example shown in Fig. 2.3 the use of the temporary arrays t_0, .., t_4 is superfluous, since the results of each codelet could be passed directly to the next codelet through registers. Additionally, memory accesses are much slower than moves between registers and, therefore, should be avoided whenever possible. In order to eliminate such unnecessary memory accesses, scalar replacement was introduced. Scalar replacement is a C-IR optimization that aims at substituting store-load sequences within a codelet chain with assignments of local variables. Such assignments will later be compiled to register moves, which are much more efficient than memory accesses. Fig. 2.4 shows the computation chain depicted in Fig. 2.3 after the application of scalar replacement. The load-compute-store approach followed at every step of the LGen methodology renders scalar replacement a crucial optimization, due to the great impact it has on the performance of the generated code.

2.2 Target microarchitectures

The main goal of this thesis was extending LGen towards four processors that are widely used in mobile and embedded devices: Intel Atom, ARM Cortex-A8, ARM Cortex-A9, and ARM1176. In the next sections we present each one of these processors listing their most important technical character-

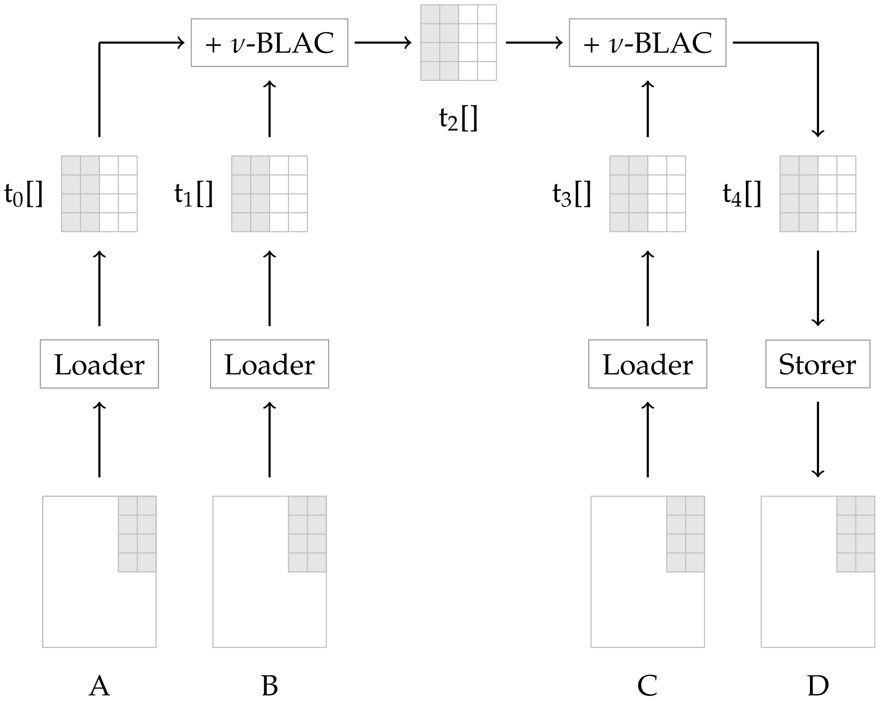


Figure 2.3: Computation chain handling leftovers for $D = (A + B) + C$. A, B, C, D are 8×6 matrices and $\nu = 4$.

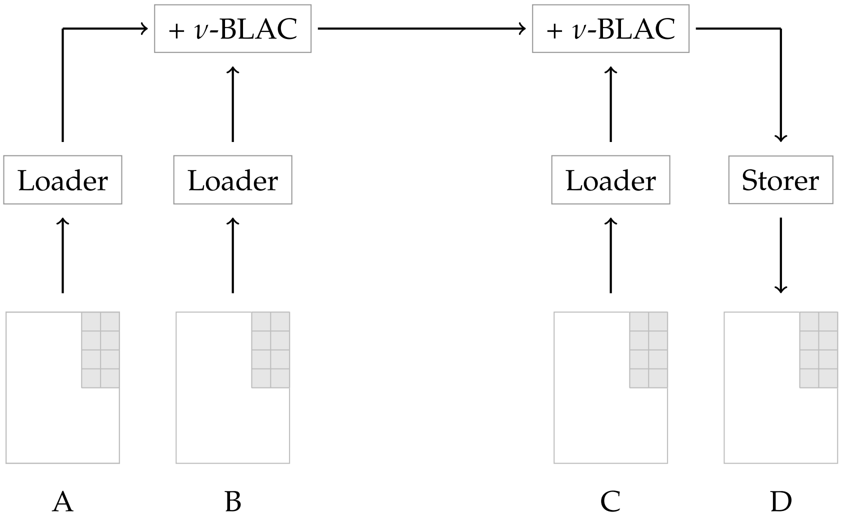


Figure 2.4: Computation chain of Fig. 2.3 after scalar replacement.

istics and limitations.

2.2.1 Intel Atom

Intel Atom (released in 2011) is a low-power processor used in a variety of devices, such as netbooks, mini PCs, mobile internet devices (MIDs), and tablets. The device that we used was a Zotac ZBox ID86 Plus containing an Intel Atom D2550 processor, whose main technical characteristics are listed in Table 2.2.

Attribute	Value
Number of CPUs	2
CPU frequency	1.86 GHz
L1 data cache	24 KB
L1 instruction cache	32 KB
ISA	x86-64
SIMD extension	SSSE3
Performance peak	6 flops/cycle

Table 2.2: Intel Atom specifications.

Atom implements the SSSE3 SIMD extension of the x86-64 ISA. SSSE3, like all the other extensions of the SSE family, supports 128-bit vectors that may contain four 32-bit floats or two 64-bit doubles. SSSE3 and its preceding SSE extensions that it extends, offer a large variety of load/store, data processing, and shuffling instructions that are extensively used from within the ν-BLAC, Loader, and Storer codelets of LGen. A detailed description of these instructions can be found in [23].

The Atom pipeline is capable of processing at most two instructions per cycle, given that they are issued in different ports. According to the SSE instructions specifications for Atom [22], the maximum number of data processing instructions (i.e. vector additions and multiplications) that can be processed per cycle is 1.5, under the assumption that there is a ratio of additions over multiplications equal to 2:1. Since a single SSE instruction applies to four single-precision elements at once, this leads to a performance peak of 6 flops/cycle. Of course this is a loose peak, mainly due to the fact that loads and stores share the same issue ports with additions and multiplications.

Another important characteristic of Atom that has a critical impact on performance is the fact that its pipeline is in-order. LGen does not apply any instruction scheduling optimizations on the generated code. Instead, it relies completely on the instruction reordering done by the underlying compiler.

2.2.2 ARM Cortex-A8

Cortex-A8 (released in 2006) is a very popular ARM processor, designed for power-optimized mobile devices and mainly used in smartphones, tablets, netbooks, digital TV, and printers. The device that we used was a BeagleBone Black, which includes a Sitara AM335x System on Chip (SoC). The main technical characteristics of Cortex-A8 are summarized in Table 2.3.

Attribute	Value
Number of CPUs	1
CPU frequency	1 GHz
L1 data cache	32 KB
L1 instruction cache	32 KB
ISA	ARMv7-A
SIMD extension	NEON
Performance peak	4 flops/cycle

Table 2.3: ARM Cortex-A8 specifications.

Cortex-A8 implements the NEON SIMD extension of the ARMv7-A ISA. The data types that NEON uses are 64-bit or 128-bit vectors. A 64-bit vector can contain two 32-bit floats and an 128-bit vector can contain four 32-bit floats. NEON offers a wide range of vector instructions for loads/stores, data-processing, and shuffling, applied on either 64-bit vectors (doubleword instructions) or 128-bit vectors (quadword instructions). Regarding data-processing, apart from the functionalities that are commonly found in other vector ISA extensions, NEON additionally offers a set of fused multiply-accumulate (FMA) instructions that are efficiently implemented in hardware and can be very useful in matrix multiplication implementations. Moreover, NEON offers a set of instructions that apply an arithmetic operation (e.g., multiplication) between all elements of a vector and a scalar coming from an arbitrary position of another vector. These instructions are very useful in matrix multiplication and by using them we can avoid unnecessary shuffling. A thorough description of all NEON instructions can be found in [3].

Cortex-A8 has two co-processors capable of executing floating point operations, the VFP and the NEON unit. The first is an IEEE-compliant execution engine that can run scalar floating point instructions. A serious limitation of the VFP unit is that it is non-pipelined, meaning that each instruction has to run to completion before the next instruction can be issued. Scalar floating point instructions can also run in the non-IEEE-compliant NEON unit, which is pipelined and performs a lot better than VFP for these instructions, although it requires a minimum amount of 7 cycles per instruction. SIMD instructions can run only on the NEON unit, which is capable of issuing one load/store and one data-processing instruction in parallel. A special microarchitectural characteristic of Cortex-A8 (as well as of Cortex-A9) that should be considered when designing a high performance implementation is that doubleword data processing instructions perform two times better than quadword instructions (in terms of both latency and throughput), while quadword memory access instructions perform roughly the same as doubleword ones.

Cortex-A8 can execute one doubleword FMA per cycle [2], so under the assumption that the computation consists of FMAs only and loads/stores are always issued in parallel with them, the peak performance for Cortex-A8 is $2 \cdot 2 = 4$ flops/cycle. Like Intel Atom, Cortex-A8 is also an in-order processor.

2.2.3 ARM Cortex-A9

Cortex-A9 (released in 2008) is a widely used, high-end processor of the ARM family that combines power efficiency and high performance. It is used in numerous mobile devices, such as smartphones and tablets, and it is available either as a single core processor or as a multicore processor that combines up to four cores. The latter is known as Cortex-A9 MPCore. The device we used was a Kayla Development Kit containing an NVIDIA Tegra 3 SoC, which is based on a quad core Cortex-A9 MPCore. The main technical specifications of a single Cortex-A9 core are presented in Table 2.4.

Attribute	Value
CPU frequency	1.4 GHz
L1 data cache	32 KB
L1 instruction cache	32 KB
ISA	ARMv7-A
SIMD extension	NEON
Performance peak	4 flops/cycle

Table 2.4: ARM Cortex-A9 core specifications.

Cortex-A9 implements the NEON vector extension of the ARMv7-A ISA, a short description of which can be found in Section 2.2.2.

Both the VFP and the NEON unit coexist in the part of the Cortex-A9 processor that is called NEON Media Processing Engine. The improvements over its predecessor Cortex-A8 include a pipelined VFP unit and out-of-order execution of instructions. A drawback compared to Cortex-A8 is the fact that the NEON pipeline of Cortex-A9 can issue only one instruction per cycle, while the corresponding pipeline of Cortex-A8 can issue a load/store and a data processing instruction at the same cycle. Again, the doubleword data-processing NEON instructions on this processor are twice as fast as their quadword equivalents. Similarly to Cortex-A8, the peak performance for Cortex-A9 is $2 \times 2 = 4$ flops/cycle, if we assume that there are only FMA instructions and we ignore memory accesses (memory access instructions are issued through the same port as data processing instructions).

2.2.4 ARM1176

ARM1176 is an older processor of the ARM family that has been used in a broad variety of devices, ranging from smartphones and game consoles to eReaders and digital picture frames. For the purpose of this thesis we worked on the ARM1176JZF-S variant of it that is included in the Raspberry Pi (Broadcom BCM2835 SoC). The main specifications of this processor are shown in Table 2.5.

Attribute	Value
Number of CPUs	1
CPU frequency	700 MHz
L1 data cache	16 KB
L1 instruction cache	16 KB
ISA	ARMv6
SIMD extension	-
Performance peak	1 flop/cycle

Table 2.5: ARM1176JZF-S specifications.

Unlike the Cortex-A processors, ARM1176 implements an older ARM ISA (ARMv6), which supports no SIMD extensions. Scalar floating point operations are executed by a VFP coprocessor that has three different in-order pipelines: FMAC for mainstream arithmetic operations, DS for division and square root operations and LS for loads and stores. All three pipelines share their first two stages, resulting to a peak performance of 1 flop/cycle, under the unrealistic assumption that there are only arithmetic operations in the code and no loads/stores.

2.3 Abstract Interpretation

In this section we present a brief overview of the abstract interpretation framework, as it was proposed by Patrick and Radhia Cousot [9, 10]. A thorough formal description of abstract interpretation is out of the scope of this thesis and the interested reader is redirected to the related literature [9, 10, 24]. Moreover, we assume that the reader has some basic knowledge of set theory, number theory, abstract algebra, and related mathematical concepts.

2.3.1 Motivation

The obvious way of reasoning about the behavior of a program is by considering all the possible executions of it. This kind of analysis would be both sound (correct) and complete (precise), but its increased complexity makes it intractable in practice. However, in most cases we are not interested in all properties of a program, but in a small subset of them. For example, if we have a square root calculation in our program, we are only interested in whether the radicand is non-negative or not. Correspondingly, if we access an element of an array we are concerned only about whether the expression that gives us the offset from the array's base address lies within the array bounds. In both of these cases tracking a specific abstract property, i.e. the sign of expressions or the interval within which their values lie, is enough to reason about some interesting aspects of the program execution, i.e. absence of divisions by zero and absence of out-of-bounds array accesses, respectively.

Abstract interpretation is a static analysis technique focused on tracking abstract properties like the ones mentioned above. It aims at overapproximating the semantics of a program in a way that is coarse enough to make the analysis computable and at the same time precise enough to retain the information that is needed to reason about some interesting aspects of the program.

2.3.2 The Abstract Interpretation Framework

This section presents the main components of the abstract interpretation framework from a high-level perspective, starting from introducing the notion of concrete and abstract domains, then moving on to giving abstract semantics to operators and statements, and finally explaining the way that the abstract interpretation analysis is applied on a program.

Concrete and abstract domains

The formalization of abstract interpretation is based on two complete lattices, one representing the real values that expressions can take and one representing the abstract values of them. The first lattice is defined as $\mathcal{L}_L := < L; \sqsubseteq_L, \sqcap_L, \sqcup_L >$, where L is a partially ordered set (poset) called the *concrete domain*, $\sqsubseteq_L$ is the partial order relation over L, $\sqcap_L$ is the greatest lower bound (meet) operator for L, and $\sqcup_l$ is the least upper bound (join) operator for L. The second lattice is similarly defined as $\mathcal{L}_{L'} := < L'; \sqsubseteq_{L'}, \sqcap_{L'}, \sqcup_{L'} >$ and L' is called the *abstract domain*. For example, for integer variables, the concrete domain is the power set of the set of integers $\mathcal{P}(\mathbb{Z})$ corresponding to the lattice $\mathcal{L}_{\mathcal{P}(\mathbb{Z})} :=< \mathcal{P}(\mathbb{Z})\,; \subseteq, \cap, \cup >$ and the abstract domain represents the property that we are interested in, such as sign, parity, or interval within which the value of an expression ranges. The elements of the concrete and the abstract domain are connected by two functions: the abstraction function $\alpha \colon L \mapsto L'$ and the concretization function $\gamma \colon L' \mapsto L$. We say that L′ is a sound abstraction of L if and only if the pair of functions α, γ forms a *Galois connection*, which is equivalent to the following set of conditions:

1. α and γ are monotone

2. $\alpha \circ \gamma(v) \sqsubseteq_{L'} v, \forall v \in L'$

3. $s \sqsubseteq_{L} \gamma \circ \alpha(s), \forall s \in L$

The conditions listed above are necessary for the soundness of an abstract interpretation analysis.

For example, in Fig. 2.5 we can see the lattice $\mathcal{L}_{\mathcal{P}(\mathbb{Z})}$, corresponding to the concrete domain for integers, and the lattice $\mathcal{L}_{Sign} :=< Sign, \sqsubseteq_{Sign}, \sqcap_{Sign}, \sqcup_{Sign} >$, corresponding to the Sign abstract domain, where $Sign := \{\top_{Sign}, \bot_{Sign}, +, -, 0\}$

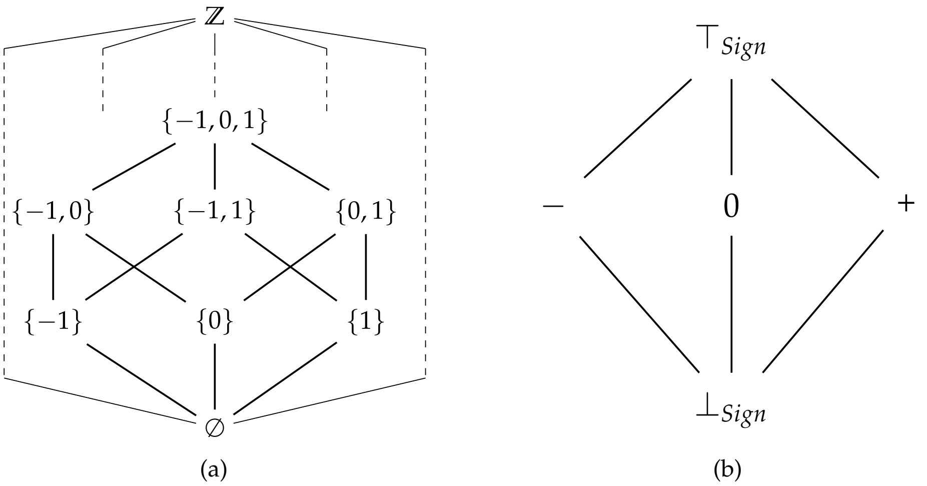


Figure 2.5: (a) The concrete domain lattice $\mathcal{L}_{\mathcal{P}(\mathbb{Z})}$ and (b) the abstract domain lattice $\mathcal{L}_{Sign}$.

The functions α and γ that connect these two domains are defined as follows:

$$\alpha(s) = \begin{cases} \bot_{Sign} & \text{if } s = \emptyset \\ - & \text{if } \forall n \in s\colon n < 0 \\ 0 & \text{if } s = \{0\} \\ + & \text{if } \forall n \in s\colon n > 0 \\ \top_{Sign} & \text{otherwise} \end{cases} \qquad \gamma(v) = \begin{cases} \emptyset & \text{if } v = \bot_{Sign} \\ \{-1, -2, -3, ...\} & \text{if } v = - \\ \{0\} & \text{if } v = 0 \\ \{1, 2, 3, ...\} & \text{if } v = + \\ \mathbb{Z} & \text{if } v = \top_{Sign} \end{cases}$$

Abstract semantics of operators

When analyzing a program using the abstract interpretation technique, we often need to combine the abstract values of variables in order to calculate the abstract values of expressions. For example, when we assign an expression to a variable, we have to evaluate the expression in the abstract domain and then assign the resulting abstract value to the variable. The result of an expression in the abstract domain is calculated based on the semantics of the involved abstract operators in a similar way as the result of this expression in the concrete domain would be computed based on the semantics of the involved concrete operators. For example, evaluating $(0 + 1)$ according to the semantics of the concrete operator + gives us 1, while evaluating $(0 +_{Sign} +)$ according to the semantics of the abstract operator $+_{Sign}$ gives us +.

The semantics of the abstract operators have to be defined in a way that over-approximates the semantics of the corresponding concrete operators, and this is a necessary condition for the soundness of an abstract interpretation analysis. For example, the semantics of the addition operator $+_{Sign}$ in the Sign domain is shown in Table 2.6.

$+_{Sign}$	$\bot_{Sign}$	-	0	+	$\top_{Sign}$
$\bot_{Sign}$	$\bot_{Sign}$	$\bot_{Sign}$	$\bot_{Sign}$	$\bot_{Sign}$	$\bot_{Sign}$
-	$\bot_{Sign}$	-	-	$\top_{Sign}$	$\top_{Sign}$
0	$\bot_{Sign}$	-	0	+	$\top_{Sign}$
+	$\bot_{Sign}$	$\top_{Sign}$	+	+	$\top_{Sign}$
$\top_{Sign}$	$\bot_{Sign}$	$\top_{Sign}$	$\top_{Sign}$	$\top_{Sign}$	$\top_{Sign}$

Table 2.6: Semantics of the $+_{Sign}$ operator.

Abstract semantics of statements

The abstract interpretation framework specifies the way that abstract interpretation is applied on a program represented by its control flow graph (CFG). Each edge of the CFG is related to an environment defined as a function $env\colon Var \mapsto L'$ that maps program variables to abstract values. The environment associated with an edge of the CFG represents the execution state at this point of the program. The abstract semantics of a statement is defined as a function $\mathbb{S}\colon Env \mapsto Env$ that specifies the environment related to the outgoing edge of the corresponding CFG node, given the combined environments of the ingoing edges of this node. If a node has more than one predecessors, the environments of its ingoing edges are combined by applying pointwise the $\sqcup_{L'}$ operator on them.

The abstract semantics of statements have to be defined in a way that over-approximates their concrete semantics and this is a necessary condition for the soundness of the analysis. For example, the semantics of the assign statement in the Sign domain is defined as follows:

$$assign_{Sign}(x, e, env) = env[x \mapsto E_{Sign}[\![e, env]\!]],$$

where x is the variable that is assigned, e is the expression that is assigned to x, env is the environment before the assignment, and $E_{Sign}[\![e, env]\!]$ is the evaluation of expression e in the Sign domain using the environment env. The notation $env[x \mapsto v]$ represents an environment that is identical to env with the only difference that the variable x is mapped to the value v.

Applying abstract interpretation on the control flow graph

Initially the environment of all edges of the CFG is set to a mapping from the program variables to the bottom element of the abstract domain, since we have no information at this point. Then we start applying the semantics of statements and information starts propagating throughout the program. This process goes on, iteratively in the case of loops, until a fixpoint is reached. A fixpoint is reached when the application of the statement semantics on all statements of the program leaves the environments of all edges of the control flow graph unchanged. The existence of a fixpoint is guaranteed by the Knaster-Tarski theorem.

2.3.3 Combining Abstract Domains

Different abstract domains are often combined into composite domains that lead to more accurate information. One way of combining abstract domains is the reduced product [11]. The resulting composite domain is a cartesian product of the individual domains and each operator on this domain is defined as its pointwise application on the individual domains. The main idea behind the reduced product is that information from some of the individual domains can be used to make more precise the information in other domains, whenever this is possible during the analysis. This refinement of information is done by the reduction function, which takes as input a value of the reduced product domain and returns a new value in the same domain that is equally or more precise, while at the same time abstracts the same concrete values. More formally, the reduction function is defined as:

$$red\colon A \mapsto A,$$

where A is the reduced product abstract domain. For a valid reduction function, the following properties must hold:

$$\begin{aligned} &red(a) \sqsubseteq_A a,\ \forall a \in A \\ &\gamma(red(a)) = \gamma(a),\ \forall a \in A. \end{aligned}$$

The first property guarantees that the reduction function has a positive effect on the analysis by making it more precise. The second property guarantees that there is no loss of information after applying the reduction function on an element of the abstract domain.

An example of reduction function is presented in Section 2.3.4.

2.3.4 Abstract Domains

In this section we present the Interval and the Congruence abstract domains, as well as the reduced product of them using an appropriate reduction function. The latter is the abstract domain used by the alignment detection optimization described in Section 3.2.

Interval domain

The Interval domain [10] I is used to approximate a set of integer values by an interval that contains them. The lower bound of the interval takes values from $\mathbb{Z} \cup \{-\infty\}$ and the upper bound takes values from $\mathbb{Z} \cup \{+\infty\}$. The values $-\infty, +\infty$ are used in cases of uncertainty about the lower or upper bound of a set of values, respectively. The lattice of the Interval domain is shown in Fig. 2.6 and the most important operators are defined in Table 2.7.

Operator	Definition
$\sqsubseteq_I$	$[a_1, a_2] \sqsubseteq_I [b_1, b_2] \Leftrightarrow a_1 \geq b_1 \wedge a_2 \leq b_2$
$\sqcup_I$	$[a_1, a_2] \sqcup_I [b_1, b_2] = [min(a_1, b_1), max(a_2, b_2)]$
$\sqcap_I$	$[a_1, a_2] \sqcap_I [b_1, b_2] =$ $[max(a_1, b_1), min(a_2, b_2)],$ if $max(a_1, b_1) \leq min(a_2, b_2)$ $\bot_I,$ otherwise
$+_I$	$[a_1, a_2] +_I [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$
$*_I$	$[a_1, a_2] *_I [b_1, b_2] = [min(a_1 * b_1, a_1 * b_2, a_2 * b_1, a_2 * b_2), \max(a_1 * b_1, a_1 * b_2, a_2 * b_1, a_2 * b_2)]$

Table 2.7: Operators in the Interval domain.

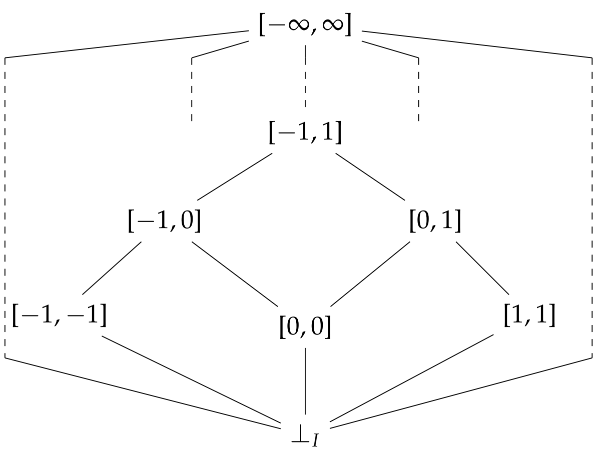


Figure 2.6: The lattice of the Interval domain.

Congruence domain

The Congruence domain [19] C is used to approximate a set of integer values by their congruence class. A congruence class is represented as $c + m\mathbb{Z} := \{v \in \mathbb{Z} \mid \exists k \in \mathbb{Z}\colon v = c + km\}$. An equivalent notation of $v \in c + m\mathbb{Z}$ is $v \equiv c[m]$. A congruence class $v + 0\mathbb{Z}$ is equal to the singleton $\{v\}$. A congruence class $c + m\mathbb{Z}$ is normalized if $0 \leq c < m$. For the rest of this thesis, we can assume that whenever a congruence class is not normalized, it is substituted by its normalized equivalent one. The lattice of the Congruence domain is shown in Fig. 2.7 and the most important operators are defined in Table 2.8.

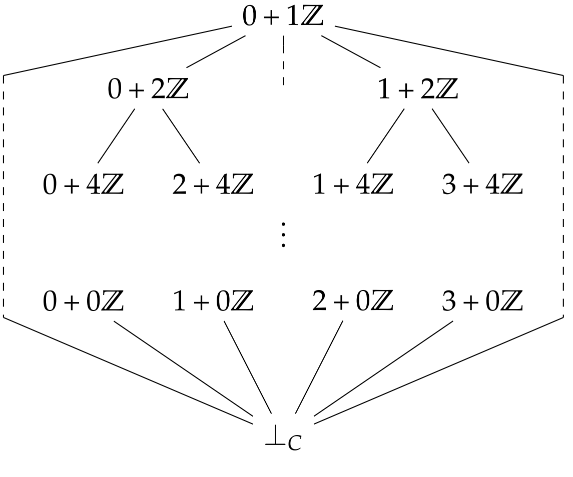


Figure 2.7: The lattice of the Congruence domain.

Operator	Definition
$\sqsubseteq_C$	$(c_1 + m_1\mathbb{Z}) \sqsubseteq_C (c_2 + m_2\mathbb{Z}) \Leftrightarrow m_2 \mid c_1 - c_2 \ \wedge \ m_2 \mid m_1$
$\sqcup_C$	$(c_1 + m_1\mathbb{Z}) \sqcup_C (c_2 + m_2\mathbb{Z}) = c_1 + gcd(m_1, m_2, c_1 - c_2)\mathbb{Z}$
$\sqcap_C$	$(c_1 + m_1\mathbb{Z}) \sqcap_C (c_2 + m_2\mathbb{Z}) =$ $\bot_C$, if $gcd(m_1, m_2) \nmid (c_1 - c_2)$ $x + lcm(m_1, m_2)\mathbb{Z}$, where $x \in c_1 + m_1\mathbb{Z} \cap c_2 + m_2\mathbb{Z}$, otherwise
$+_C$	$(c_1 + m_1\mathbb{Z}) +_C (c_2 + m_2\mathbb{Z}) = (c_1 + c_2) + gcd(m_1, m_2)\mathbb{Z}$
$*_C$	$(c_1 + m_1\mathbb{Z}) *_C (c_2 + m_2\mathbb{Z}) = c_1c_2 + gcd(c_1m_2, m_1c_2, m_1m_2)\mathbb{Z}$

Table 2.8: Operators in the Congruence domain. The function gcd returns the greatest common divisor of its arguments and the function lcm returns the least common multiplier of its arguments.

Reduced product of the Interval and Congruence domain

A common combination of abstract domains is the reduced product of the Interval and Congruence domain. This domain is capable of tracking the bounds of a set of integer values together with its density (the greater the congruence class in the Congruence lattice, the greater the density). A reduction function for this domain that was proposed in [19] is defined as follows (we assume that the conditions of the different cases are evaluated top-down and the result of $red(i, con)$ is determined by the first condition that is found to hold):

$$red(i, con) = \begin{cases} (\bot_I, \bot_C) & \text{if } i = \bot_I \text{ or } con = \bot_C \\ (\bot_I, \bot_C) & \text{if } con = c + 0\mathbb{Z} \text{ and } c \notin i \\ ([c, c], c + 0\mathbb{Z}) & \text{if } con = c + 0\mathbb{Z} \\ (\bot_I, \bot_C) & \text{if } i = [a, b] \\ & \text{ and } R(con, a) > L(con, b) \\ ([R(con, a), R(con, a)], R(con, a) + 0\mathbb{Z}) & \text{if } i = [a, b] \\ & \text{ and } R(con, a) = L(con, b) \\ ([R(con, a), L(con, b)], con) & \text{if } i = [a, b] \\ ([R(con, a), +\infty], con) & \text{if } i = [a, +\infty] \\ ([-\infty, L(con, b)], con) & \text{if } i = [-\infty, b] \\ (i, con) & \text{otherwise} \end{cases},$$

where the functions $R\colon C \setminus \{\bot_C\} \times \mathbb{Z} \mapsto \mathbb{Z}$ and $L\colon C \setminus \{\bot_C\} \times \mathbb{Z} \mapsto \mathbb{Z}$ are defined as:

$$R(c + m\mathbb{Z}, a) = a + ((c - a) \mod |m|)$$
$$L(c + m\mathbb{Z}, a) = a - ((a - c) \mod |m|).$$

The result of $R(c + m\mathbb{Z}, a)$ is the smallest integer n, such that $n \geq a \wedge n \in c + m\mathbb{Z}$. Correspondingly, the result of $L(c + m\mathbb{Z}, a)$ is the greatest integer n, such that $n \leq a \wedge n \in c + m\mathbb{Z}$. These two functions are used in order to tighten the bounds of i, in case they do not belong in con.

Some examples of the application of the reduction function are the following:

$red([0, 3], 4 + 0\mathbb{Z}) = (\perp_I, \perp_C)$ (information combined from both domains)
$red([0, 3], 4 + 5\mathbb{Z}) = (\perp_I, \perp_C)$ (information combined from both domains)
$red([0, 0], 0 + 8\mathbb{Z}) = ([0, 0], 0 + 0\mathbb{Z})$ (information passed from the Interval domain to the Congruence domain)
$red([-1, 1], 0 + 0\mathbb{Z}) = ([0, 0], 0 + 0\mathbb{Z})$ (information passed from the Congruence domain to the Interval domain)
$red([1, 5], 0 + 2\mathbb{Z}) = ([2, 4], 0 + 2\mathbb{Z})$ (information passed from the Congruence domain to the Interval domain)

Chapter 3

Optimizations

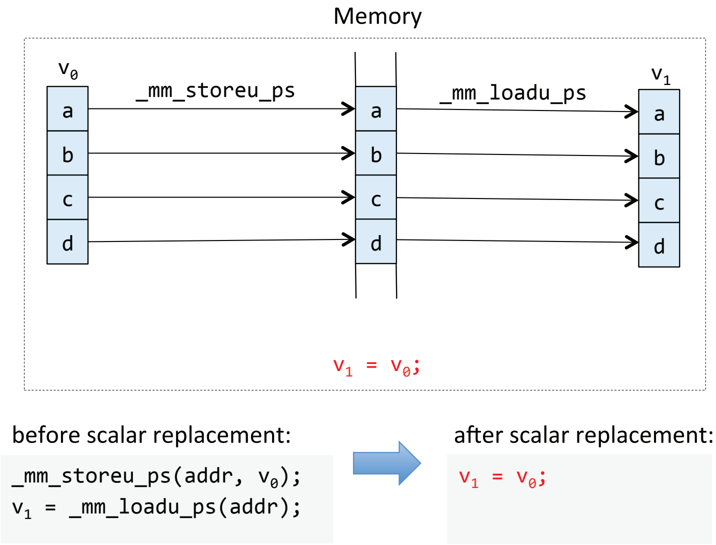
In this chapter we present the optimizations that were introduced in LGen in the context of this thesis. Each of the sections that follow is devoted to a single optimization, presenting the problem that it addresses, the intuition behind it, and the necessary details about the way it works. The effectiveness of the optimizations is evaluated through appropriate experiments, which we present and discuss in Chapter 5.

3.1 Generic Load/Store C-IR Instructions

In Section 2.1 we described how LGen applies scalar replacement to blocks of C-IR instructions when they appear in a load-compute-store sequence. The application of scalar replacement results in replacing store-load sequences within these blocks with assignments of local variables. In this way, unnecessary memory accesses are avoided, which has a significant positive impact on performance, as memory accesses are typically much more expensive than register moves. Scalar replacement is applied at the C-IR level and can be used for both scalar code (variables of type float or double) and vector code (variables of type vector).

Scalar replacement in LGen works in the following way: Whenever a store instruction is found that has the same memory footprint with a subsequent load instruction, both instructions are replaced by a single assignment which sets the variable that the load was loading to the value that the store was storing. Loads that don't follow any store with the same memory footprint are left in the code and their results are stored in local variables, so that they can be reused by subsequent instructions. Similarly, stores that are not followed by loads with the same footprint are left intact, as well.

The memory footprint of a load/store instruction is specified using a structure named memory map, which maps memory locations to positions within a vector. More specifically, each load/store instruction is associated with a position within a matrix and a memory map that relates horizontal offsets from this position to elements of the vector to be loaded/stored. Checking the matrix position and the memory map of a store and of a subsequent load is enough to determine whether these two instructions can be replaced by an assignment during scalar replacement. Fig. 3.1 shows an example of scalar replacement applied to a 4-way SSE store-load sequence.



before scalar replacement:

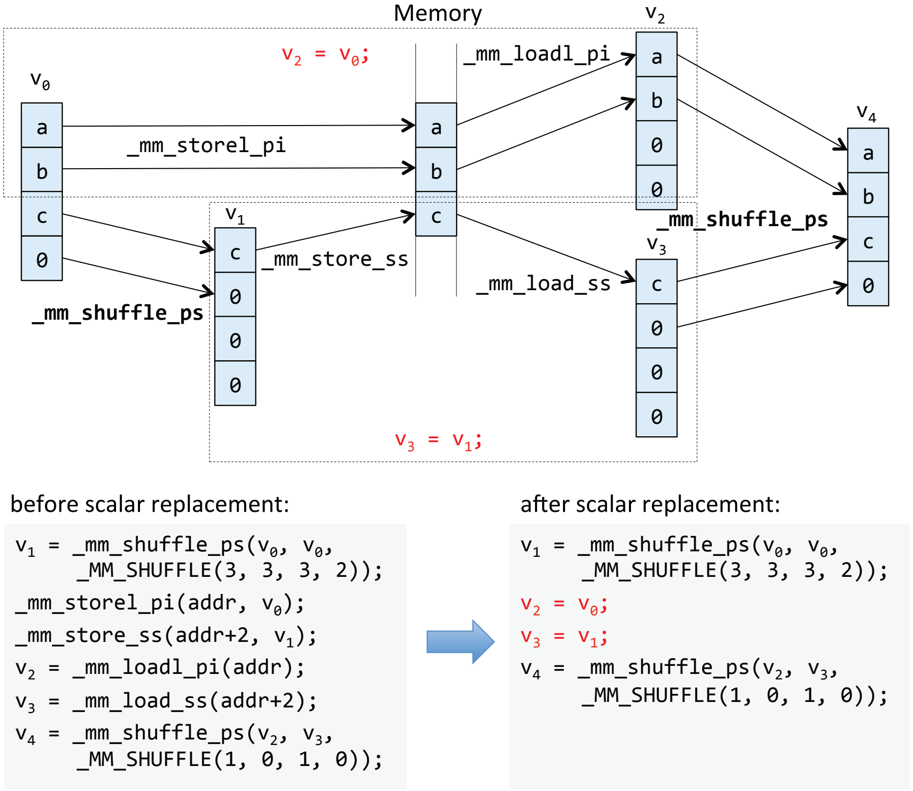
```
_mm_storeu_ps(addr, v0);
v1 = _mm_loadu_ps(addr);
```

after scalar replacement:

```
v1 = v0;
```

Figure 3.1: Simple case of scalar replacement.

However, data is not always stored/loaded in such a straightforward way. For example, let's consider the case where we want to store the first three elements of a vector into memory and then load them again into another vector. Since storing the first three elements of a vector is not possible with a single SSE instruction, we have to break this operation into two steps, one for storing the first two elements and another one for storing the third element, as it is shown in Fig. 3.2. The same holds for the load. As we can see in Fig. 3.2, apart from load/store instructions, we also have to use two shuffle instructions, one for moving the third element of the source vector v_0 into the first element of the auxiliary vector v_1 and one for moving the first element of the auxiliary vector v_3 into the third element of the target vector v_4. If we apply scalar replacement on this code, we will manage to eliminate the memory accesses, but the shuffle instructions will remain, although they are redundant. The compiler will most probably not eliminate the shuffles either (at least this is the case for Intel's icc), so we will finally have to pay the overhead of these unnecessary instructions. The accumulated effect of many redundant shuffle instructions may deteriorate the performance of our generated code significantly, especially in computations that have a high percentage of leftover code.



before scalar replacement:

```
v1 = _mm_shuffle_ps(v0, v0,
     _MM_SHUFFLE(3, 3, 3, 2));
_mm_storel_pi(addr, v0);
_mm_store_ss(addr+2, v1);
v2 = _mm_loadl_pi(addr);
v3 = _mm_load_ss(addr+2);
v4 = _mm_shuffle_ps(v2, v3,
     _MM_SHUFFLE(1, 0, 1, 0));
```

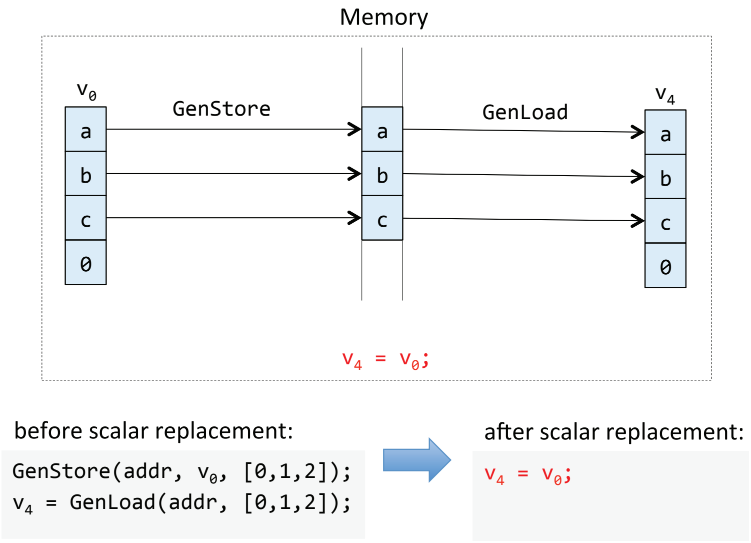
after scalar replacement:

```
v1 = _mm_shuffle_ps(v0, v0,
     _MM_SHUFFLE(3, 3, 3, 2));
v2 = v0;
v3 = v1;
v4 = _mm_shuffle_ps(v2, v3,
     _MM_SHUFFLE(1, 0, 1, 0));
```

Figure 3.2: Scalar replacement with normal loads/stores.

As a solution to this problem, we introduce the generic load and store C-IR instructions, which are not tied to specific C instructions, but instead they are generic enough to represent all possible interactions between vectors and memory. These instructions are translated to normal C-IR instructions (shuffles and normal loads/stores) according to their associated memory maps only one step before unparsing the C-IR code into C code. One generic load-/store can be translated into one or more normal instructions, depending on its memory map.

If we come back to our motivating example, the store of the first three elements of the source vector can now be represented by a single generic store with the corresponding memory map, as it is shown in Fig. 3.3. The same holds for loading these three elements to the destination vector. When scalar replacement is applied, the matching of these two instructions will be revealed, resulting in triggering their replacement by a single assignment of the source vector to the destination vector.



before scalar replacement:

```
GenStore(addr, v_0, [0,1,2]);
v_4 = GenLoad(addr, [0,1,2]);
```

after scalar replacement:

```
v_4 = v_0;
```

Figure 3.3: Scalar replacement with generic loads/stores.

In linear algebra computations like matrix multiplication or matrix transposition, we often want to access a part of a column of a matrix, which corresponds to a strided access of the physical layout of the matrix in memory (we always store matrices in row-major order). Up to now, the concept of memory map allowed us to match elements within a row of a matrix to elements of a vector. The introduction of generic loads/stores offered an opportunity to extend the concept of the memory map, in order to support the representation of vertical segments of a matrix as well. Using this extended version of memory map, we can now represent the load of vertically consecutive elements of a matrix into a vector using a single generic load instruction that is associated with a vertical memory map. The same can also be done for the equivalent store operations. This extension allows for easy and effective scalar replacement without any redundant shuffle operations, no matter whether we are accessing a horizontal or a vertical segment of a matrix.

Another benefit of using generic load/store instructions comes from the fact that they decouple (possibly complicated) memory accesses from their actual implementation. The normal instructions that a generic load/store is translated to, have no effect on scalar replacement, as they appear only at the last step before unparsing to C code. This allows us to implement a generic load in a different way than its equivalent generic store and still be confident that the matching of these two memory accesses will be deduced during scalar replacement. The absence of the restriction for implementing a non-trivial load in the same way as its equivalent store can make a difference in cases that the most performant implementation of the load is different than the most performant implementation of the store. An example of this, taken from our NEON ν-BLACs, can be seen in Fig. 3.4. Although we would like to store the first three elements of a vector by first storing the first two elements (vst1_f32) and then the third one (vst1q_lane_f32), while at the same time we would like to load three elements from memory into a vector by loading all four elements (vld1q_f32) and then setting the fourth element to zero (setq_lane_f32), scalar replacement would not be able to match and replace these memory accesses. Instead, the use of one generic store for storing and one generic load for loading provides scalar replacement with all the necessary information to match the memory accesses, while at the same time we will get the desired C code implementation when we unparse the C-IR code.

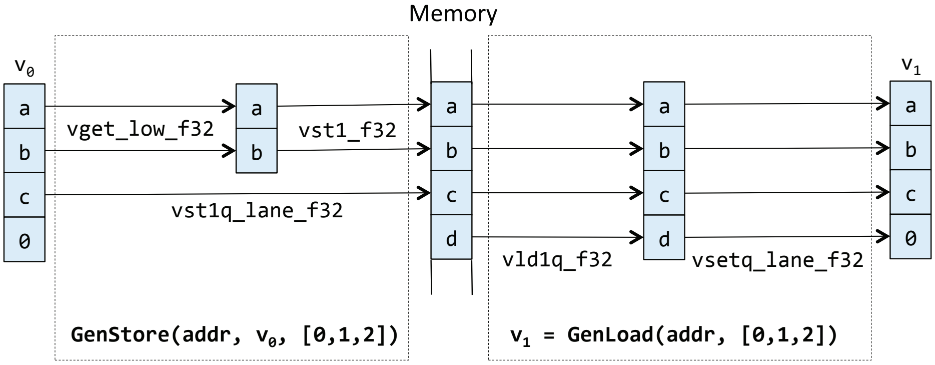


Figure 3.4: Mismatched implementations of generic load-store for NEON.

A last point worth mentioning about the generic load/store instructions is the fact that we can easily try different implementations of any of these instructions, e.g., trying to find the one that leads to the highest performance, without having to worry about how our changes affect their interaction with other instructions during our C-IR code processing. The representation of these instructions during the application of the various optimizations at the C-IR level is independent of the way we choose to translate them into C code.

Although we did not evaluate the positive effects of introducing generic memory access instructions through experiments, because the newly added ν-BLACs (i.e. the ones for SSSE3 and NEON) were implemented directly using these instructions and the already existent ν-BLACs (i.e. the ones for SSE3, SSE4.1, and AVX) were left unchanged, we could strongly argue that LGen benefited from this optimization, since the removal of redundant instructions from the generated code can only have a positive effect on performance.

3.2 Alignment Detection

Alignment detection is one of the most beneficial optimizations introduced for Intel Atom, due to the great performance difference between aligned and unaligned memory accesses for this processor. This section presents this optimization by first giving an overview of the problem and the solution, then providing the details of the applied methodology, next arguing about the soundness and preciseness of the analysis, and finally explaining how we handle arbitrary alignment of the array arguments of the generated kernels.

3.2.1 Overview

A vector load or store instruction involves moving long chunks of data between memory and a vector register. Such a memory access is characterized as aligned or unaligned, depending on whether the corresponding memory address is a multiple of a specific number of bytes, which we will refer to as alignment length. The alignment length is typically equal to the vector size. Aligned memory accesses perform better than unaligned ones, due to the fact that in most cases the hardware is designed to handle more efficiently memory segments that lie on specific boundaries. Memory is typically accessed in blocks and if the data chunk to be accessed spans two blocks (i.e. the data chunk is unaligned), the hardware has to read or write both of them during a load or store, respectively.

Although in many modern microarchitectures unaligned memory accesses have roughly the same performance as aligned ones [22], this is not the case for Intel Atom, one of the processors that were investigated in the context of this thesis. After adapting the already existent SSE4.1 ν-BLACs to the SSSE3 instruction set that Atom supports, experiments showed that the performance of our generated code was significantly inferior to the one of the competitor libraries and compilers, especially for memory-intensive computations like matrix addition or matrix-vector multiplication. The reason behind this issue was the fact that LGen was generating unaligned instructions only, while the competitors were using aligned instructions in cases that this was applicable. An aligned memory access is much more efficient on Atom, in terms of both latency and throughput, than its unaligned equivalent, therefore the generation of high performance code for this processor renders the use of aligned instructions a necessity.

Aligned SSE instructions can be used only in cases where the referenced memory address is 16-byte aligned, while the use of an aligned SSE instruction to access unaligned data produces a runtime error. In other words, we can use aligned instructions only if we are sure that the referenced data are aligned and for all the other cases we have to use unaligned instructions. Based on this observation, an additional optimization was introduced to

LGen, named alignment detection. Alignment detection is applied at the C-IR level and is divided in two steps: At the first step, the C-IR code is analyzed using the well-established technique of abstract interpretation [9, 10] combined with an abstract domain that is appropriate for tracking alignment (see Section 3.2.2). At the second step, the results of the analysis are used in order to reason about the alignment of all memory accesses contained in the C-IR code and those that are found to be certainly aligned are replaced by their aligned equivalents.

3.2.2 Details of the Analysis

The abstract domain that is used for the abstract interpretation analysis applied during alignment detection is a reduced product of the Interval and Congruence domains, the details of which can be found in Section 2.3.4. The Interval domain tracks the intervals within which the values of variables range. The Congruence domain tracks the congruence classes that variables belong to. The reduction function uses information from the one domain to make the information in the other domain more precise, which in our case is useful mainly in cases of loops that are taken only once, as we will see later on.

The format of the code generated by LGen with respect to memory accesses is shown in Listing 3.1, where start$_i$, end$_i$, step$_i$, a, a$_i$, $L \in \mathbb{N}$, $i \in [0, L-1]$. The address of any memory access has the format a$_0$ind$_0$ + a$_1$ind$_1$ + ... + a$_{L-1}$ind$_{L-1}$ + a, where ind$_i$ is the index of the i-th nested for-loop that encloses the corresponding load or store. Therefore, we apply our analysis only on the indexes introduced by the for-loops, since these are the only variables that may participate in the calculation of the memory address that a memory access refers to.

```
for( size_t ind_0 = start_0; ind_0 < end_0; ind_0 += step_0 ) {
  for( size_t ind_1 = start_1; ind_1 < end_1; ind_1 += step_1 ) {
    ...
      for(size_t ind_{L-1}=start_{L-1}; ind_{L-1}<end_{L-1}; ind_{L-1}+=step_{L-1}) {
        ...
        mem_access(a_0ind_0 + a_1ind_1 + ... + a_{L-1}ind_{L-1} + a);
        ...
      }
    ...
  }
}
```

Listing 3.1: Format of generated code with respect to memory accesses.

After applying the analysis, we go through all memory accesses that exist in the C-IR code and for each one we decide whether it refers to aligned data or not by evaluating the abstract value of the corresponding memory address using the abstract environment associated with this instruction. If the Congruence part of the abstract value of the memory address is $\sqsubseteq_C$ $0 + N\mathbb{Z}$, where N is the alignment length, the access is aligned and it is safe to replace this instruction with its aligned equivalent. Otherwise, we leave the instruction as is (unaligned).

In practice, since the offset of an array access in the generated code is expressed in number of elements (floats or doubles) and not in number of bytes, the criterion for characterizing a memory access as aligned is that the abstract value of the corresponding memory address is $\sqsubseteq_C$ $0 + (N/l)\mathbb{Z}$, where l is the size of an array element, in bytes. For example, in x86 $l = 4$ for floats or 8 for doubles.

The intuition behind using the Interval domain originates from the fact that it is capable of detecting loops that are taken only once. Propagating this information from the Interval domain to the Congruence domain through the reduction function can potentially make the value in the Congruence domain more precise and help us detect more cases of aligned memory accesses. Listing 3.2 shows an example where the use of the Interval domain makes the analysis more precise. In this example, if we considered only the Congruence domain, the abstract value of index k after reaching the fixpoint would be $0 + 13\mathbb{Z}$ and the memory access _mm_load_ps(A + k) would be decided to be unaligned, although in reality the loop is taken only for k = 0 and, therefore, _mm_load_ps(A + k) is aligned (we assume that the base address of A is aligned). The reduced product that we use as the abstract domain of our analysis handles this special case as follows: At the first iteration, the value of k is $([0, 0],\ 0 + 0\mathbb{Z})$. At the second iteration, right after we apply the semantics of the for-instruction, this value will become $([0, 0],\ 0 + 13\mathbb{Z})$. The value in the Interval domain remains $[0, 0]$, due to the fact that $[0, 0] \sqcup_I ([13, 13] \sqcap_I [0, 7]) = [0, 0] \sqcup_I \perp_I = [0, 0]$ (the $\sqcap_I$ comes from the semantics of the implicit *assume* statement that is associated with the true branch of the for-loop). Applying the reduction function on $([0, 0],\ 0 + 13\mathbb{Z})$ will give us $([0, 0],\ 0 + 0\mathbb{Z})$, the fixpoint will thus be reached already at the end of the second iteration and the memory access will eventually be characterized as aligned, since $0 + 0\mathbb{Z} \sqsubseteq_C 0 + 4\mathbb{Z}$.

```
for ( size_t k = 0; k < 8; k += 13 ) {
  ...
  v = _mm_load_ps(A + k);
  ...
}
```

Listing 3.2: Case where the Interval domain makes the analysis more precise.

3.2.3 Soundness and Preciseness

Theorem 3.1 *Our analysis overapproximates all real executions (soundness).*

The soundness of our analysis is a direct result of the soundness of the Interval and Congruence analyses separately, combined with the correctness of the reduction function. Hints for the corresponding proofs can be found in the related literature [10, 19] and, therefore, these proofs are omitted from this thesis.

Next we present three lemmas expressing some basic properties of the mod operator and the greatest common divisor (gcd) that will be used in the proof of Theorem 3.5. The proofs for these lemmas can be found in any major number theory textbook and, therefore, are omitted from this thesis.

Lemma 3.2 *Let* $a, n \in \mathbb{Z}, n \neq 0$.
Then: $(a \bmod n) \bmod n = a \bmod n$

Lemma 3.3 *Let* $a, b, n \in \mathbb{Z}, n \neq 0$.
Then: $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$

Lemma 3.4 *Let* $a, b \in \mathbb{Z}$*, such that not both of a and b are zero.*
Then: $c \mid a \wedge c \mid b \Rightarrow c \mid gcd(a, b)$

Theorem 3.5 *Our analysis detects every aligned memory access that is contained in any kernel generated by LGen (preciseness).*

Proof Equivalently, we have to prove that:
Let ind_i *be the index variable introduced in the i-th nested loop that encloses an access of the memory address* $A = a_0 ind_0 + \ldots + a_{L-1} ind_{L-1} + a$*, where* $a, a_i \in \mathbb{Z}$.
$\forall N \in \mathbb{Z}$: $A \bmod N = 0$ *every time the execution reaches* $A \Rightarrow E_C[\![A]\!] \sqsubseteq_C 0 + N\mathbb{Z}$*, where* E_C *is the evaluation function of our analysis for the Congruence domain.*

Let us assume that there exists $N \in \mathbb{Z}$, such that:

$$A \bmod N = 0 \text{ every time the execution reaches A} \tag{3.1}$$

Let $S = \{i \mid start_i + step_i < end_i\} = \{i \mid \text{the } i\text{-th loop is taken more than once}\}$

a. At the first iteration of the analysis on the i-th loop:

$$\begin{aligned} env_C^0(ind_i) &= E_C[\![start_i]\!] = start_i + 0\mathbb{Z} \\ env_I^0(ind_i) &= E_I[\![start_i]\!] = [start_i, start_i] \end{aligned}$$

b. At the second iteration of the analysis on the i-th loop:

$$\begin{aligned} env_C^1(ind_i) &= env_C^0(ind_i) \sqcup_C (env_C^0(ind_i) +_C E_C[\![step_i]\!]) \\ &= (start_i + 0\mathbb{Z}) \sqcup_C ((start_i + 0\mathbb{Z}) +_C (step_i + 0\mathbb{Z})) \\ &= start_i + step_i\mathbb{Z} \end{aligned}$$

- if $i \notin S$:

$$start_i + step_i \geq end_i \tag{3.2}$$

$$\begin{aligned} env_I^1(ind_i) &= env_I^0(ind_i) \sqcup_I ((env_I^0(ind_i) +_I E_I[\![step_i]\!]) \sqcap_I [start_i, end_i - 1]) \\ &= [start_i, start_i] \sqcup_I (([start_i, start_i] +_I [step_i, step_i]) \sqcap_I [start_i, end_i - 1]) \\ &= [start_i, start_i] \sqcup_I ([start_i + step_i, start_i + step_i] \sqcap_I [start_i, end_i - 1]) \\ &\overset{(3.2)}{=} [start_i, start_i] \sqcup_I \bot_I = [start_i, start_i] \end{aligned}$$

Applying the reduction function on $env_C^1(ind_i)$, $env_I^1(ind_i)$ will refine the value of $env_C^1(ind_i)$, giving us:

$$env_C^1(ind_i) = R(env_C^1(ind_i), start_i) + 0\mathbb{Z} = start_i + 0\mathbb{Z}$$

Thus, the fixpoint is reached.

- if $i \in S$:

$$start_i + step_i < end_i \tag{3.3}$$

$$\begin{aligned} env_I^1(ind_i) &= env_I^0(ind_i) \sqcup_I ((env_I^0(ind_i) +_I E_I[\![step_i]\!]) \sqcap_I [start_i, end_i - 1]) \\ &= [start_i, start_i] \sqcup_I (([start_i, start_i] +_I [step_i, step_i]) \sqcap_I [start_i, end_i - 1]) \\ &= [start_i, start_i] \sqcup_I ([start_i + step_i, start_i + step_i] \sqcap_I [start_i, end_i - 1]) \\ &\overset{(3.3)}{=} [start_i, start_i] \sqcup_I [start_i + step_i, start_i + step_i] \\ &= [start_i, start_i + step_i] \end{aligned}$$

Applying the reduction function on $env_C^1(ind_i)$, $env_I^1(ind_i)$ will leave both values unchanged. No matter what the relation among $start_i$, $step_i$, end_i is, in the next iterations only $env_I(ind_i)$ may change, while $env_C(ind_i)$ will retain the value $start_i + step_i\mathbb{Z}$ until the fixpoint is reached.

After reaching the fixpoint we will have:

$$\begin{aligned} E_C[\![A]\!] &= E_C[\![a_0]\!] *_C env_C(ind_0) +_C ... +_C E_C[\![a_{L-1}]\!] *_C env_C(ind_{L-1}) +_C E_C[\![a]\!] \\ &= \sum_{i=0}^{L-1} a_i start_i + a + gcd(a_i step_i \mid i \in S)\mathbb{Z} \end{aligned} \tag{3.4}$$

When the execution reaches A in the first iteration of all loops:

$$(3.1) \Rightarrow N \mid (\sum_{i=0}^{L-1} a_i start_i + a) \tag{3.5}$$

When the execution reaches A in the second iteration of loop $k \in S$ and the first iteration of all other loops:

$$
\begin{aligned}
(3.1) \Rightarrow & (\sum_{i=0}^{L-1} a_i start_i + a_k step_k + a) \bmod N = 0 \\
\overset{Lemma\ 3.3}{\Rightarrow} & ((\sum_{i=0}^{L-1} a_i start_i + a) \bmod N + (a_k step_k) \bmod N) \bmod N = 0 \\
\overset{(3.5)}{\Rightarrow} & ((a_k step_k) \bmod N) \bmod N = 0 \\
\overset{Lemma\ 3.2}{\Rightarrow} & (a_k step_k) \bmod N = 0 \\
\overset{Lemma\ 3.4}{\Rightarrow} & N \mid gcd(\{a_i step_i \mid i \in S\}) \qquad (3.6)
\end{aligned}
$$

By definition of $\sqsubseteq_C$ (Table 2.8), we have:

$$(3.4), (3.5), (3.6) \Rightarrow E_C[\![A]\!] \sqsubseteq_C 0 + N\mathbb{Z}$$

□

As it was mentioned in Section 3.2.1, aligned instructions can only be used if we are certain that the accessed data are aligned. Theorem 3.1 guarantees that this is the case for all memory accesses that our analysis detects as aligned. At the same time, Theorem 3.5 guarantees that our analysis is most precise for the code that LGen generates, meaning that each aligned memory access in the generated code is detected by our analysis and is handled by the appropriate aligned instruction. In other words, our analysis can be characterized as optimal or complete [18] with respect to the type of code that LGen generates.

3.2.4 Handling Arbitrary Alignment

In all the examples of the previous sections, our analysis was applied based on the assumption that the base addresses of all arrays involved were aligned. However, this assumption is not always valid, since in the general case the caller of our generated kernels may pass as arguments arbitrarily aligned arrays. In order to handle the various argument alignment combinations, we generate a different version of C-IR code for each one of them, with the corresponding load/store instructions turned into aligned ones. The code version that will be executed is chosen at runtime, according to the result of appropriate alignment checks.

More specifically, the process of handing arbitrary alignment is the following: First we generate the C-IR code with unaligned instructions only. Then we generate the various alignment combinations for the array parameters of the kernel and for each combination: a) we make a copy of the C-IR code, b) we apply abstract interpretation on it based on the corresponding alignment assumptions, c) we detect the accesses that are always aligned, and d) we replace them with their aligned equivalents. Finally, when we finish with all combinations, we combine the resulting C-IR code versions using conditional statements, where the conditions correspond to runtime checks of all possible alignment combinations.

If we consider a byte-addressable memory, each array argument of a kernel can be aligned in as many ways as the alignment length N, resulting to a total number of possible alignment combinations equal to N^a, where a is the number of parameters of the kernel that are arrays. Since an array containing elements of a specific datatype is accessed in offsets that are multiples of this datatype's length l, if its base address is not l-byte aligned, all accesses of this array will inevitably be unaligned. In this respect, all alignment cases that are not multiples of l can be grouped together and handled in the same way, reducing the number of combinations to $(N/l+1)^a$. In order to simplify things even more, we group all cases where at least one array is not l-byte aligned into one and we handle all memory accesses as unaligned for this case. After this simplification, the total number of alignment combinations is further reduced to $(N/l)^a+1$.

An example of a matrix multiplication kernel that was generated in order to handle arbitrary alignment is presented in Listing 3.3.

```
/*
 * mmm_kernel.h
 * Tile((1, 2), Tile((4, 4), C[18,81])) =
 * Tile((1, 4), Tile((4, 4), A[18,35])) *
 * Tile((4, 2), Tile((4, 4), B[35,81]))
 */

static __attribute__((noinline)) void kernel(float const * A, float const * B, float * C)
{
  if(((uintptr_t) C) % (4 * sizeof(float)) == 0 * sizeof(float)
  && ((uintptr_t) A) % (4 * sizeof(float)) == 0 * sizeof(float)
  && ((uintptr_t) B) % (4 * sizeof(float)) == 0 * sizeof(float))
  {
    /* Code version 1 */
  }
  else if(((uintptr_t) C) % (4 * sizeof(float)) == 0 * sizeof(float)
  && ((uintptr_t) A) % (4 * sizeof(float)) == 0 * sizeof(float)
  && ((uintptr_t) B) % (4 * sizeof(float)) == 1 * sizeof(float))
  {
    /* Code version 2 */
  }

  ...

  else if(((uintptr_t) C) % (4 * sizeof(float)) == 3 * sizeof(float)
  && ((uintptr_t) A) % (4 * sizeof(float)) == 3 * sizeof(float)
  && ((uintptr_t) B) % (4 * sizeof(float)) == 3 * sizeof(float))
  {
    /* Code version 4^3 */
  }
  else
  {
    /* Unaligned code version */
  }
}
```

Listing 3.3: Matrix multiplication kernel that handles arbitrary alignment.

3.3 Matrix-Vector Multiplication

Matrix-vector multiplication is a very common linear algebra operation and consists the main building block of most level-2 BLAS, as well as several more complicated BLACs. Therefore, an efficient implementation of it is critical for a high-performance linear algebra code generator, like LGen. However, matrix-vector multiplication experiments that we initially executed on x86 machines ranked LGen lower than most competitors. The investigation of this issue revealed that its source was that LGen had been treating matrix-vector multiplication as a special case of matrix-matrix multiplication and, as it will be shown later in this section, such an approach is suboptimal for most x86 microarchitectures. As part of this thesis we present a new matrix-vector multiplication approach that takes into consideration the x86 microarchitectural characteristics and limitations aiming at the generation of code that performs better on these machines. As it is shown in the experimental results section for Atom (Section 5.2), the new approach ranks LGen higher than the competitors.

Let's consider the matrix-vector multiplication $y = Ax$, where A is a $M \times N$ matrix and x, y are vectors of length N. Following the original approach of LGen for matrix-vector multiplication with tiles of size ν, the result is calculated according to the Σ-LL expression:

$$y = \sum_{i=0,\nu}^{M-1} S_i \sum_{j=0,\nu}^{N-1} (G_i A G_j)(G_j x) \tag{3.7}$$

For x86, the matrix-vector ν-BLAC, which is the main component of (3.7), is implemented using horizontal addition instructions (`_mm_hadd_ps` and `_mm_hadd_pd` for SSE, `_mm256_hadd_ps` and `_mm256_hadd_pd` for AVX), which are considerably less efficient than normal vector addition instructions (`_mm_add_ps` and `_mm_add_pd` for SSE, `_mm256_add_ps` and `_mm256_add_pd` for AVX) for most x86 microarchitectures [22]. Listing 3.4 shows the C-IR implementation of this ν-BLAC for SSSE3 and floats ($\nu = 4$), while Table 3.1 shows the performance of `_mm_add_ps` and `_mm_hadd_ps` for some common x86 microarchitectures. The difference between normal and horizontal vector addition is particularly large for Atom, an observation that is further amplified by the fact that horizontal addition occupies both of the issue ports of Atom, while normal addition occupies only one of them.

Microarchitecture	**_mm_add_ps**	**_mm_hadd_ps**
Haswell	3 / 1	5 / 2
Ivy Bridge	3 / 1	5 / 2
Sandy Bridge	3 / 1	5 / 2
Westmere	3 / 1	5 / 2
Nehalem	3 / 1	5 / 2
Atom	5 / 1	8 / 7

Table 3.1: Performance of vector addition for x86 microarchitectures. The numbers are in the form *latency / throughput*, where *latency* is the number of cycles that are required by the execution core to complete the execution of the instruction and *throughput* is the number of clock cycles required to wait before the issue ports are free to accept the same instruction again.

The inner summation of equation (3.7) consists of $\lceil N/\nu \rceil$ matrix-vector multiplications and $\lceil N/\nu \rceil - 1$ vector additions. Due to the outer summation, the whole computation involves in total $\lceil M/\nu \rceil \lceil N/\nu \rceil$ matrix-vector multiplications and $\lceil M/\nu \rceil (\lceil N/\nu \rceil - 1)$ vector additions. Since the matrices that participate in these multiplications and additions are ν-sized, these computations are implemented by the respective ν-BLACs for the corresponding target architecture. The implementation of these ν-BLACs for x86 with SSSE3 is shown in Listings 3.4 and 3.5.

```
blac_nu4_mvm(B, refA, refx, out):
  B ← Mov (mmLoaduPs refA[0,0]), va0
  B ← Mov (mmLoaduPs refA[1,0]), va1
  B ← Mov (mmLoaduPs refA[2,0]), va2
  B ← Mov (mmLoaduPs refA[3,0]), va3
  B ← Mov (mmLoaduPs refx[0,0]), vx
  B ← Mov (mmMulPs va0, vx), mul0
  B ← Mov (mmMulPs va1, vx), mul1
  B ← Mov (mmMulPs va2, vx), mul2
  B ← Mov (mmMulPs va3, vx), mul3
  B ← Mov (mmHaddPs mul0, mul1), hadd0
  B ← Mov (mmHaddPs mul2, mul3), hadd1
  B ← mmStoreuPs (mmHaddPs hadd0, hadd1), out[0,0]
```

Listing 3.4: SSSE3 ν-BLAC for Ax.

```
blac_nu4_vadd(B, refx, refy, out):
  B ← Mov (mmLoaduPs refx[0,0]), vx
  B ← Mov (mmLoaduPs refy[0,0]), vy
  B ← mmStoreuPs (mmAddPs vx, vy), out[0,0]
```

Listing 3.5: SSSE3 ν-BLAC for $x + y$.

In order to overcome this limitation resulting from the weak performance of horizontal add for x86, a new approach for matrix-vector multiplication was introduced that involves only a limited number of horizontal vector additions. This new approach is based on two new operators that were introduced into LL and Σ-LL: $\boxtimes$, called *matrix-vector Hadamard product* (*MVH*), and $\boxplus$, called *Row Reduction* (*RR*).

MVH takes as operands a $M \times N$ matrix A and a vector x of length N and returns a $M \times N$ matrix C, each row of which is the result of the Hadamard product between the corresponding row of A and x^T. More formally:

$$C = A \boxtimes x \Leftrightarrow C_{ij} = A_{ij} x_j$$

RR is a unary operator that transforms a $M \times N$ matrix A into a vector x of length M by applying addition reduction to each row of A. More formally:

$$x = \boxplus A \Leftrightarrow x_i = \sum_{j=0}^{N-1} A_{ij}$$

The semantics of these two operators are the same in the two DSLs and for translating them from Σ-LL into IR-code, two new types of ν-BLACs were introduced.

If we revisit the implementation of the matrix-vector multiplication ν-BLAC in Listing 3.4, we can easily see that the computation can be split into two parts, one involving a MVH between A and x (lines 7-10) and one involving a RR of the result of the MVH (lines 11-13), which is the main source of inefficiency due to the horizontal additions that it consists of. Based on this observation, the problem of having too many horizontal additions can be alleviated by splitting the ν-BLAC into these two parts and moving the inner summation of equation 3.7 one level deeper, between these two parts. Since both the RR and the summation aim at adding the results of the MVHs per row, applying such a transformation leaves the result of the computation unaffected, due to the commutative property of addition. Therefore, using the two newly introduced operators, matrix-vector multiplication can be computed according to the following Σ-LL expression:

$$y = \sum_{i=0,\nu}^{M-1} S_i \left[\boxplus \sum_{j=0,\nu}^{N-1} (G_i A G_j) \boxtimes (G_j x) \right] \tag{3.8}$$

Equation (3.8) is equivalent to equation (3.7), but its main computation block is the MVH $(G_i A G_j) \boxtimes (G_j x)$ instead of the matrix-vector multiplication $(G_i A G_j)(G_j x)$. The inner summation involves $\lceil N/\nu \rceil$ $\nu \times \nu$ MVHs and $(\lceil N/\nu \rceil - 1)$ $\nu \times \nu$ matrix additions, while the outer summation involves $\lceil M/\nu \rceil$ $\nu \times \nu$ RRs. In total, the computation as a whole consists of $\lceil M/\nu \rceil \lceil N/\nu \rceil$ MVHs, $\lceil M/\nu \rceil \lceil (N/\nu \rceil - 1)$ matrix additions and $\lceil M/\nu \rceil$ RRs. The implementation of the corresponding ν-BLACs for x86 with SSSE3 is shown in Listings 3.6, 3.7 and 3.8.

```
blac_nu4_pmul(B, refA, refx, out):
  B ← Mov (mmLoaduPs refA[0,0]), va0
  B ← Mov (mmLoaduPs refA[1,0]), va1
  B ← Mov (mmLoaduPs refA[2,0]), va2
  B ← Mov (mmLoaduPs refA[3,0]), va3
  B ← Mov (mmLoaduPs refx[0,0]), vx
  B ← mmStoreuPs (mmMulPs va0, vx), out[0,0]
  B ← mmStoreuPs (mmMulPs va1, vx), out[1,0]
  B ← mmStoreuPs (mmMulPs va2, vx), out[2,0]
  B ← mmStoreuPs (mmMulPs va3, vx), out[3,0]
```

Listing 3.6: SSSE3 ν-BLAC for $A \boxtimes x$.

```
blac_nu4_hred(B, refA, out):
  B ← Mov (mmLoaduPs refA[0,0]), va0
  B ← Mov (mmLoaduPs refA[1,0]), va1
  B ← Mov (mmLoaduPs refA[2,0]), va2
  B ← Mov (mmLoaduPs refA[3,0]), va3
  B ← Mov (mmHaddPs va0, va1), hadd0
  B ← Mov (mmHaddPs va2, va3), hadd1
  B ← mmStoreuPs (mmHaddPs hadd0, hadd1), out[0,0]
```

Listing 3.7: SSSE3 ν-BLAC for $\boxplus A$.

```
blac_nu4_madd(B, refA, refB, out):
  B ← Mov (mmLoaduPs refA[0,0]), va0
  B ← Mov (mmLoaduPs refA[1,0]), va1
  B ← Mov (mmLoaduPs refA[2,0]), va2
  B ← Mov (mmLoaduPs refA[3,0]), va3
  B ← Mov (mmLoaduPs refB[0,0]), vb0
  B ← Mov (mmLoaduPs refB[1,0]), vb1
  B ← Mov (mmLoaduPs refB[2,0]), vb2
  B ← Mov (mmLoaduPs refB[3,0]), vb3
  B ← mmStoreuPs (mmAddPs va0, vb0), out[0,0]
  B ← mmStoreuPs (mmAddPs va1, vb1), out[1,0]
  B ← mmStoreuPs (mmAddPs va2, vb2), out[2,0]
  B ← mmStoreuPs (mmAddPs va3, vb3), out[3,0]
```

Listing 3.8: SSSE3 ν-BLAC for $A + B$.

In the analysis that follows, we assume for the sake of simplification that both M and N are multiples of ν. The amount of the different types of arithmetic operations required by the old and the new MVM approach for x86 with SSSE3 and $\nu = 4$ is shown in Table 3.2.

Operation	**Old MVM**	**New MVM**
mmMulPs	$MN/4$	$MN/4$
mmAddPs	$(M/4)(N/4-1)$	$M(N/4-1)$
mmHaddPs	$3MN/16$	$3M/4$
Total	$(M/4)(2N-1)$	$(M/4)(2N-1)$

Table 3.2: Number of arithmetic operations involved in the old (equation 3.7) and the new (equation 3.8) matrix-vector multiplication approach for x86 with SSSE3 and $\nu = 4$.

As expected, the two approaches result to the same total number of arithmetic operations, since their only difference is the order in which the additions are performed. However, the new approach involves less mmHaddPs and more mmAddPs operations than the old one. Since mmAddPs is much more efficient than mmHaddPs, we expect that the new approach leads to considerably more performant generated code than the old one. Our expectations are confirmed by the related experiments of Section 5.2.

3.4 Specialized ν-BLACs

According to the basic approach of LGen, a BLAC is split into smaller computations, which are implemented by handwritten ν-BLACs for the corresponding architecture. These computations are applied on submatrices of the original matrices with possible sizes $1 \times \nu$, $\nu \times 1$ and $\nu \times \nu$. In case we need to apply a computation on submatrices with sizes smaller than ν (i.e. leftover tiles), these small submatrices are embedded into ν-sized matrices before being processed by the appropriate ν-BLACs. Handling leftovers like this typically involves generating additional code compared to a straightforward handwritten implementation that computes the same result. This additional code mainly consists of two parts:

1. Loads of zeros: The parts of the involved vectors that do not contain useful data have to explicitly be filled with zeros, so that they do not affect the result of the computation.

2. Unnecessary arithmetic operations: Since a ν-BLAC has no knowledge of which parts of the matrices contain useful data, some of the executed operations are applied on zero elements, although they have no effect in the computation.

According to the existing LGen approach, the elimination of the redundant part of the ν-BLAC code is left to the compiler. However, in most cases the compiler can eliminate only a part of the redundant code, while the rest of it remains, having a negative impact on the performance of the generated kernel (ν-BLACs usually exist inside tight loops).

For example, let's consider the case of a $2 \times 2 \times 2$ matrix-matrix multiplication with $\nu = 4$, depicted in Fig. 3.5. The two 2×2 operands are embedded into two 4×4 matrices A and B, then these two matrices are multiplied through the corresponding ν-BLAC and finally the upper left 2×2 submatrix of the result C is extracted and written to the 2×2 result matrix. The two 2×2 matrices are located in memory, while each of the 4×4 matrices is stored in four vectors, each of size 4. The first two rows of both 4×4 operands have to be filled by combining useful data and zeros through appropriate shuffle instructions, while the two last rows of them have to be fully filled with zeros. The typical implementation of a matrix-matrix multiplication ν-BLAC like this works as follows: The i row of the result is computed by multiplying the ij element of the first operand with the j row of the second operand and accumulating over j. In this respect, all computations involving zero elements of the first operand do not affect the result and, therefore, are redundant and can be completely eliminated.

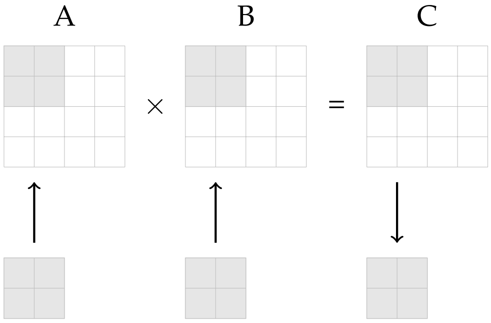


Figure 3.5: $2 \times 2 \times 2$ matrix-matrix multiplication.

Listing 3.9 shows the assembly code that is generated by clang 3.4 for this computation on ARM Cortex-A9. The lines 2, 5, 6, 7, 8 correspond to loading zeros in vectors. The instructions that calculate the two last rows of the result matrix have been eliminated by the compiler, since they correspond to dead code. However, the instructions that multiply a zero element of the first two rows of the first operand with a row of the second operand are left in the code (lines 15, 17, 18, 20) and the same holds for the instructions that add the zero results of these multiplications to the corresponding accumulators (lines 19, 21, 22, 23).

```
1 _ZL6kernelPKfS0_Pf:            ;kernel computing C=AB
2 vmov.i32      d3, #0x0
```

```
vld1.32     {d5, d6}, [r1]
vld1.32     {d1, d2}, [r0]
vmov.i32     q11, #0x0
vmov.f64     d16, d5
vorr         d17, d3, d3
vorr         d7, d3, d3
vmul.f32     q9, q8, d2[0]    ;P0=A[1][0]*B[0][]
vmul.f32     q10, q3, d2[1]   ;P1=A[1][1]*B[1][]
vorr         d2, d1, d1
vmul.f32     q8, q8, d2[0]    ;P2=A[0][0]*B[0][]
vmul.f32     q12, q3, d2[1]   ;P3=A[0][1]*B[1][]
vadd.f32     q9, q10, q9      ;S0=P0+P1
vmul.f32     q13, q11, d3[0]  ;P4=A[0][2]*B[2][]
vadd.f32     q8, q12, q8      ;S1=P2+P3
vmul.f32     q10, q11, d3[0]  ;P5=A[1][2]*B[2][]
vmul.f32     q12, q11, d3[1]  ;P6=A[0][3]*B[3][]
vadd.f32     q9, q13, q9      ;S0=S0+P4
vmul.f32     q11, q11, d3[1]  ;P7=A[1][3]*B[3][]
vadd.f32     q8, q10, q8      ;S1=S1+P5
vadd.f32     q9, q12, q9      ;S0=S0+P6
vadd.f32     q8, q11, q8      ;S1=S1+P7
vorr         d17, d18, d18
vst1.32     {d16, d17}, [r2]  ;C[0][]=S0[0:1]
                              ;C[1][]=S1[0:1]
bx           lr
```

Listing 3.9: Assembly for $2 \times 2 \times 2$ matrix multiplication on Cortex-A9 using the traditional approach.

Another pitfall of the current approach for leftover handling on Cortex-A processors stems from the fact that the corresponding ν-BLACs use only quadword vector instructions, even when the desired result can be computed using doubleword ones, which are twice as efficient (in terms of both latency and throughput) for these processors. The performance mismatch between doubleword and quadword instructions adds up to the unnecessary overhead that we have to pay following the current approach and was the main factor that determined our decision to implement specialized ν-BLACs for these microarchitectures.

The specialized ν-BLACs are similar to the traditional ν-BLACs, with the only difference that they are applied to matrices with sizes smaller than ν (leftovers). Whenever we come across such matrices, instead of embedding them into ν-sized ones and passing them to traditional ν-BLACs, we directly pass them to the appropriate specialized ν-BLACs, which are responsible for handling these small-sized computations in a more efficient way. The positive effect of using this new approach is visible in cases where we have a high percentage of leftovers, which happens in computations involving small matrices whose dimensions are not multiples of ν, or narrow panels whose small dimension is not a multiple of ν.

Listing 3.10 shows the equivalent of listing 3.9 using the new approach of specialized ν-BLACs. The generated assembly code in this case is much more compact, includes no zero loads at all and makes use of the faster doubleword NEON instructions. Executing the code of listing 3.9 on a Cortex-A9 processor takes 68 cycles, which in terms of performance is equivalent to 0.17 flops/cycle. On the other hand, the new approach gives us 23 cycles and 0.52 flops/cycle on the same machine, which corresponds to a speedup of about 3. More experiments that evaluate the usefulness of the specialized ν-BLACs are presented in Sections 5.3.5 and 5.4.5.

```
_ZL6kernelPKfS0_Pf:               ;kernel computing C=AB
vld1.32       {d0, d1}, [r0]
vld1.32       {d16, d17}, [r1]
vmul.f32       d18, d16, d1[0]    ;P0=A[1][0]*B[0]
vmul.f32       d19, d17, d1[1]    ;P1=A[1][1]*B[1]
vmul.f32       d20, d16, d0[0]    ;P2=A[0][0]*B[0]
vadd.f32       d19, d19, d18      ;S0=P0+P1
vmul.f32       d16, d17, d0[1]    ;P3=A[0][1]*B[1]
vadd.f32       d18, d16, d20      ;S1=P2+P3
vst1.32       {d18, d19}, [r2]    ;C[0]=S1
                                  ;C[1]=S0
bx             lr
```

Listing 3.10: Assembly for 2 × 2 × 2 matrix multiplication on Cortex-A9 using a specialised ν-BLAC.

Chapter 4

Mediator

In this chapter we present Mediator, a middleware developed as part of this thesis in order to facilitate and coordinate the execution of experiments on arbitrary SSH-accessible devices. In the next sections we motivate the development of Mediator and give an overview of its most important features. Next, we present the architectural skeleton of the software from a high-level perspective, we demonstrate the communication API, and, finally, we present the process of retrieving performance metrics using the infrastructure offered by Mediator.

4.1 Motivation

Code generation and execution are not necessarily carried out on the same hardware platform. Similarly to a cross-compiler, a code generator can generate code optimized for a target platform while not running on it. On top of that, an autotuning approach like the one followed by LGen instructs that many code variants have to be generated and tested on the target device before obtaining a final result. In such cases, the code has to be transferred from the source device to the target device, compiled and executed there, and finally the measurements have to be transferred back to the source device where they will potentially be further processed. A variation of this scenario involves cross-compilation on the source device and then transfer of a binary file to the target device, instead of transferring the source code to the target device and then doing a native compilation there. This variation is very common in cases where the target device is less powerful than the source device or in cases where native compilation is not possible.

Assuming we are able to execute experiments remotely, a basic prerequisite for reliable performance measurements is the elimination of interference from other programs running at the same time on the target device. If the target device is shared among a number of users, a mechanism that coordinates the execution of experiments is inevitable. Assuming that we are only interested in single-threaded programs, this mechanism has to ensure that only one experiment at a time is executed on a specific core of the target device. Although the problem of running experiments on a remote target device can be addressed by the development of an ad-hoc solution, such as a shell script, the problem of mutually exclusive experiment executions remains. Addressing this issue requires a centralized coordination unit over the target device that guarantees that the execution of the experiments is performed in the desirable mutually exclusive way.

Another important aspect of executing performance experiments is the measuring process. Measuring performance accurately is not always trivial, since it demands interacting with the hardware of the machine where the program runs, usually through accessing performance counters. Since this process is specific to the target device and at the same time remains identical for all experiments executed on a specific target device, it could be separated from the experiment code and put into reusable modules (one per target device) that implement a predefined interface, universal for all devices. In this respect, a user would be able to retrieve performance metrics for the execution of their programs without worrying about the way the performance metrics are extracted.

4.2 Overview

Mediator is a software tool that attempts to satisfy the requirements presented in the previous section. More specifically, it is a web application that functions as a middleware between the source device and the target device. It receives from the source device (also referred to as client) all the appropriate data for the execution of the experiments (both source code/binaries and experiment metadata) and is responsible for running the specified experiments on the specified target device(s) and sending the acquired measurements back to the source device when the experiments are completed. Additionally, if multiple users are trying to execute experiments at the same time, Mediator makes sure that only one experiment is running at any moment per core per device. More information about the internals of Mediator covering this issue can be found in Section 4.3. Finally, as a solution to the performance measuring issue described in the previous section, Mediator contains modules that implement the extraction of performance metrics for a set of microarchitectures. These modules can be optionally used from within the source code of the experiment according to the interface presented in Section 4.5.

Mediator is written in Python 2.7 (approximately 2K lines of code), while the performance measuring modules are written in C. The web interface of

Mediator was developed using the open-source Flask microframework [15].

4.3 Architecture

A high-level view of Mediator's architecture can be seen in Fig. 4.1. The continuous lines show the workflow between receiving a request and obtaining the results of the experiments that are contained in this request. The dashed lines show the workflow between obtaining the experiment results and returning them to the client.

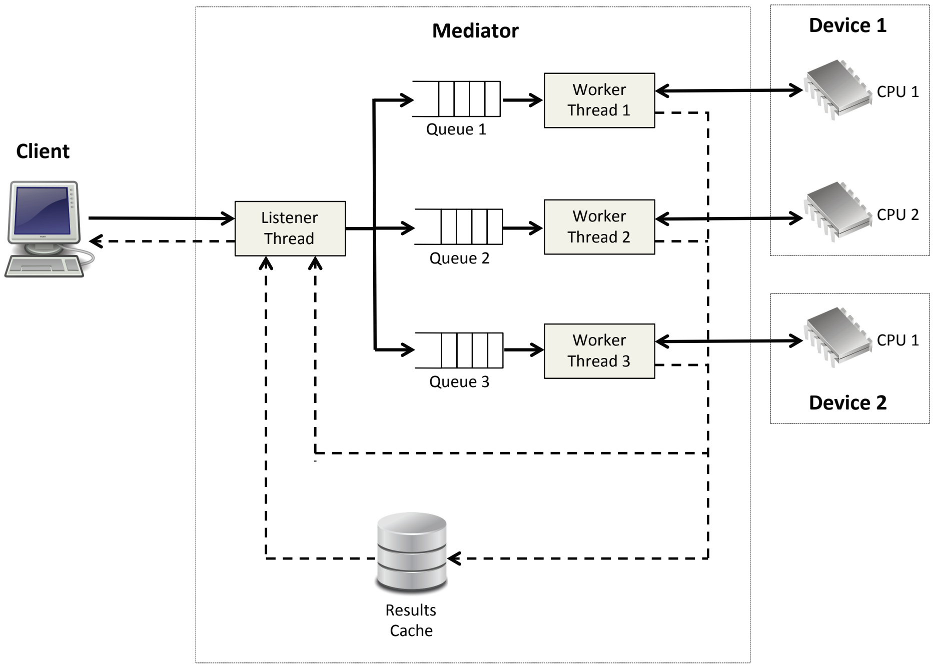


Figure 4.1: Mediator architecture.

The entry point of the workflow is a Listener Thread, which keeps listening at a specific port waiting for incoming HTTP requests from clients. This thread is currently part of the server where Mediator is deployed, that is the Flask built-in development server. In the current implementation of Mediator there is only one Listener Thread due to a restriction of the server, but in general we could (and should) have more than one. Mediator is appropriately designed to support multiple Listener Threads, since all shared data structures are either thread-safe or accessed in a thread-safe way using locks.

When the Listener Thread receives a new job request from a client, it extracts the experiments specified in it and enqueues them in the queues that correspond to the cores that the experiments are designated to run on. If the specifications of an experiment dictate that it can run on more than one cores of a device, the Listener Thread assigns the experiment to the core that has the least number of pending experiments (i.e. the queue that has the least number of entries), thus favoring a load-balance scheme. The experiments will then be dequeued by the Worker Threads and executed on the target devices. Since there is only one Queue and one Worker Thread associated with each core of each device, it is guaranteed that the experiments will be executed one at a time per core per device. Of course experiments targeting different devices or different cores of the same device can be executed simultaneously, since they are handled by different Worker Threads.

After the completion of the experiments, there are two possible workflow paths, depending on whether the request is processed synchronously or asynchronously, which is determined by one of the parameters of the initial request sent to Mediator. In the first case, the results are assembled by the Listener Thread and they are sent back to the client as a response to its initial request. In the second case, the connection with the client is already closed and the results can be sent back only as a response to a new results request from the client. For this reason, as soon as the experiments finish, the Worker Threads store the experiment results in a temporary in-memory storage called Results Cache (see Fig. 4.1). When the client sends a results request, the Listener Thread retrieves the results from the Results Cache and sends them back to the client. Results that stay in the Results Cache for more than a specific amount of time expire and are automatically deleted. The results expiration time is set in the configuration settings of Mediator.

4.4 Communication

The communication between the clients and Mediator is done through an HTTP RESTful interface based on the JavaScript Object Notation (JSON) data-interchange format [13]. The communication between Mediator and the target devices is done through the Secure Shell (SSH) protocol.

The client initially sends to Mediator a new job request, which defines a set of experiments that the client orders to be executed on some devices (one device per experiment). A thorough description of the JSON properties of a new job request is listed in Table A.1. If the request is not well-formatted, Mediator returns a job results response with an error at the top level of the contained JSON representation (see Table A.2). Otherwise, it connects through SSH to the target devices specified in the experiments description and attempts to run the experiments on them. If the experiment executions terminate successfully, Mediator collects the performance measurements and returns them to the client within a job results response (see Table A.2). If some of the experiments terminated in error, the entries of the job results response that correspond to these experiments contain descriptive error messages. A list of all possible errors can be found in Table A.5.

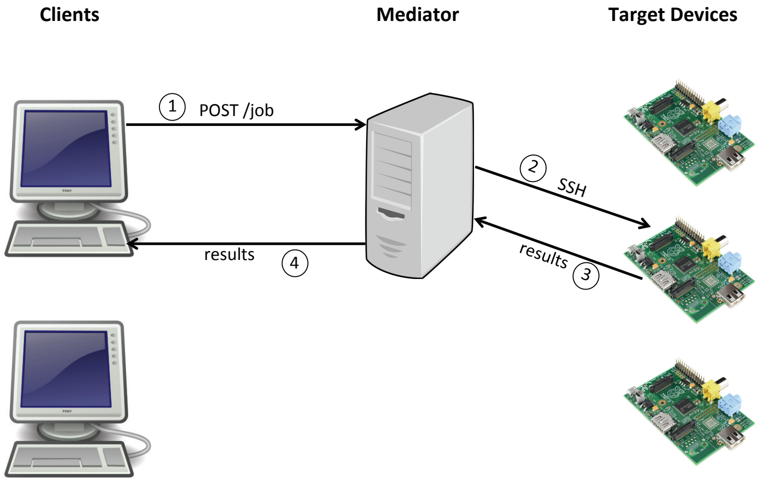


Figure 4.2: Synchronous communication.

If the property `async` of the new job request is not set to `True`, this signifies that the client requests for synchronous processing of the new job from Mediator and the procedure followed is the one depicted in Fig. 4.2. Between receiving the new job request and sending back the results response, Mediator keeps the HTTP connection with the client open.

If the property `async` is set to `True`, or not set at all, this signifies that the client requests for asynchronous processing of the new job from Mediator. When Mediator receives such a request, it does some preliminary checks on it and responds to the client before starting processing the new job. If the preliminary checks failed, Mediator responds with a job results response with an error at the top level of the contained JSON representation (see Table A.2). Otherwise, it responds with a job status response with the property `jobState` set to `SUBMITTED` (see Table A.4). The value of the `jobID` property of the job status response uniquely identifies the new job and can be used later by the client to poll for the job results. Polling is done by sending consecutive job results requests to Mediator that have the format specified in Table A.3. For as long as the job is not fully processed, Mediator responds to polling with a job status response with the `jobState` property set

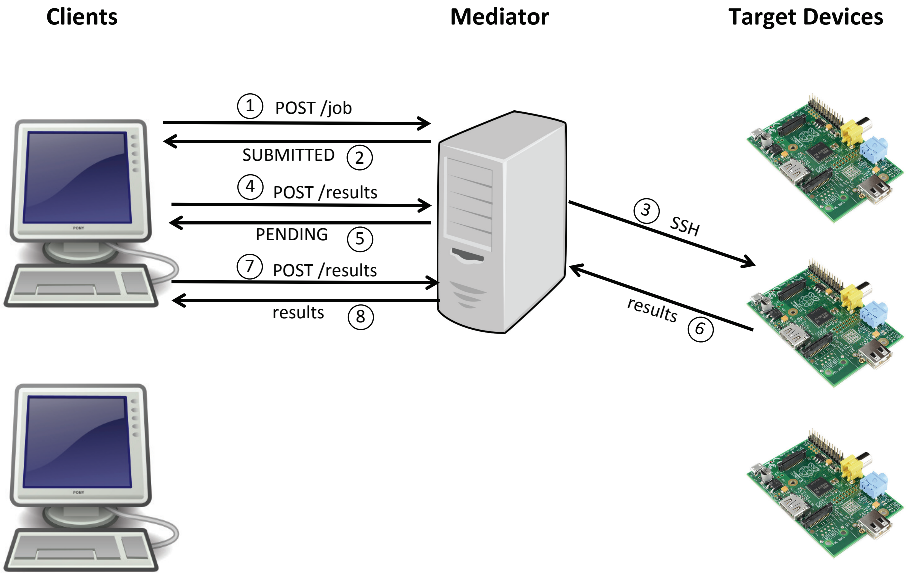


Figure 4.3: Asynchronous communication with polling.

to PENDING. After the job processing finishes, Mediator responds with a job status response with jobState set to FINISHED that contains the results of the job under the property data (see Table A.4).

4.5 Retrieving Performance Metrics

As mentioned before, Mediator provides some already implemented C modules that can be used to extract performance metrics. This feature is enabled if the new job request sets the Experiment property measureArch (see Table A.1) to one of the supported microarchitectures. If this happens, Mediator copies the relevant module to the experimentRootFolder (see Table A.1) on the target device as a file measure.h, so that it can be used from within the experiment code by including it. The interface that a performance measuring module has to implement is shown in Listing 4.1. To date the only performance metric that Mediator provides is CPU cycles and there is support for Cortex-A8, Cortex-A9, ARM1176, and microarchitectures of the Intel x86 family. Extending this feature for more microarchitectures is by design very simple, since it only requires the addition of one module per new microarchitecture that implements the required interface.

```
#ifndef WIN32
    #define myInt64 unsigned long long
#else
```

```
    #define myInt64 signed __int64
#endif

/************************************************************/
/* The functions below can be used if we just want to       */
/* trigger the measuring of performance metrics from        */
/* within our code and let Mediator gather the measured     */
/* values and return them to us in the HTTP response.       */
/************************************************************/

/**
* Initialize the measuring process.
* This function should be called by the client code at the
* beginning of the experiment, before the first call to
* measurement_start().
*/
__attribute__((noinline)) void measurement_init(void);

/**
* Start counting.
*/
__attribute__((noinline)) void measurement_start(void);

/**
* Stop counting.
*/
__attribute__((noinline)) void measurement_stop(void);

/**
* Finalize the measuring process.
* This function should be called by the client code at the
* end of an experiment, after the last call to measurement_stop().
*/
__attribute__((noinline)) void measurement_finish(void);


/************************************************************/
/* The functions below can be used if we want to explicitly */
/* access the performance metrics from within our code.     */
/************************************************************/

/**
* Initialize the cycles counter.
* This function should be called by the client code at the
* beginning of the experiment, before the first call to
* start_tsc().
*/
__attribute__((noinline)) void init_tsc(void);

/**
* Start counting CPU cycles.
* The return value should be given as an argument to
* the subsequent call to stop_tsc().
*/
```

```
__attribute__((noinline)) myInt64 start_tsc(void);

/**
* Stop counting CPU cycles.
* The return value is the number of CPU cycles passed
* since start.
*/
__attribute__((noinline)) myInt64 stop_tsc(myInt64 start);

/**
* Get the overhead of calling start_tsc(); stop_tsc();
*/
__attribute__((noinline)) myInt64 get_tsc_overhead(void)
```

Listing 4.1: Interface of performance measuring modules.

Chapter 5

Experimental Results

In this chapter we evaluate the effectiveness of the optimizations presented in Chapter 3 through the execution of appropriate experiments. In particular, at first we present the details of the experimental setup that we used and then we discuss the most important experiments on each of the four processors investigated. The results of a larger set of experiments can be found in Appendix B.

5.1 Experimental Setup

The experimental setup that we used is heavily based on the one presented in [34], both in terms of chosen BLACs and experimental procedure.

5.1.1 Chosen BLACs

In an attempt to get an as clear as possible view of the improvements added by the introduced optimizations, we tested LGen in a variety of computations. A possible classification of the chosen BLACs, based on [34], is the following:

1. Simple BLACs: $y = Ax$ and $C = AB$.
2. BLACs that closely match BLAS: $y = \alpha x + y$, $y = \alpha Ax + \beta y$, and $C = \alpha AB + \beta C$.
3. BLACs that consist of more than one BLAS: $y = \alpha Ax + \beta Bx$, $\alpha = x^T Ay$, and $C = \alpha(A_0 + A_1)^T B + \beta C$.
4. Micro-BLACs: $y = Ax, C = AB$ and $\alpha = x^T Ay$ for very small matrices and vectors.
5. Special BLACs: BLACs geared towards specific optimizations. The details of these BLACs as well as the intuition behind choosing them are given in the description of the corresponding experiments.

For the categories 1-3 we make use of three types of matrices:

a) Panels: Narrow rectangular matrices with sizes $4 \times n$ or $n \times 4$.

b) Blocks: Small 4×4 matrices.

c) Matrices with varying shape: $30 \times n$ and $n \times 30$ matrices whose shape varies between a horizontal panel and a vertical panel. This type of matrices is used in order to get a general view of LGen's performance for a BLAC.

For the category 4 we use small square matrices with sizes between 2 and 10.

All experiments involve single-precision code. For all plots, the y-axis shows performance, measured in flops per cycle (f/c), and the x-axis shows the value of the input's varying dimensions. This parameter is always represented by the symbol n in the plot captions.

5.1.2 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, naively implemented 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 to the generated kernel (labeled as *gen* on plots). The compilers that we used to compete against are presented in the next section.

5.1.3 Compilers

The compilers and the compiler flags that we used for each processor are:

- Intel Atom: icc 14.0.0 for all cases (flags: `-O3 -xHost -fargument-noalias -fno-alias -no-ipo -no-ip -no-prec-div`).
- ARM Cortex-A8: gcc 4.7.3 for handwritten code and ATLAS (flags: `-O3 -ffast-math -fsingle-precision-constant -fstrict-aliasing -mcpu=cortex-a8 -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=hard`), clang 3.4 for all cases except from ATLAS (flags: `-mcpu=cortex-a8 -O3`).
- ARM Cortex-A9: gcc 4.7.3 for handwritten code and ATLAS (flags: `-O3 -ffast-math -fsingle-precision-constant -fstrict-aliasing -mcpu=cortex-a9 -march=armv7-a -mtune=cortex-a9 -mfpu=neon -mfloat-abi=hard`), clang 3.4 for all cases except from ATLAS (flags: `-mcpu=cortex-a9 -O3`).

- ARM1176: gcc 4.7.2 for all cases (flags: `-O3 -fsingle-precision-constant -fstrict-aliasing -mcpu=arm1176jzf-s`), clang 3.4 for LGen and handwritten code (flags: `-mcpu=arm1176jzf-s -O3`).

5.1.4 Measuring Process

The metric that we measure in all experiments is flops per cycle (f/c). More specifically, flops are deduced from the BLAC that a generated kernel implements and the size of the matrices involved, while cycles are explicitly measured by the caller (tester) of the kernel. For Intel Atom, we are using the RDTSC instruction that reads the contents of the 64-bit time stamp counter (TSC) and writes them in the registers EDX:EAX. For ARM Cortex-A8 and ARM1176 we are accessing directly the cycle count register that is part of the system control coprocessor of each processor. In order to do that, we first had to enable user-mode access of this register through a loadable kernel module that does the necessary configuration modifications, as this is described in the related manuals [1, 3]. For ARM Cortex-A9 we didn't manage to enable user-mode access of the cycle count register and instead we used the perf infrastructure of Linux.

The correctness of all the experiments presented in this chapter was validated by comparing their calculated results with the corresponding results of equivalent naive implementations and verifying that their absolute difference is at all times less than a small threshold ε.

All experiments are done under warm cache conditions, meaning that the generated kernel is executed a few times before starting measuring its performance. Due to the short execution times of the generated kernels, we adopted the measuring strategy followed in [34]: The code is executed multiple times so that the total cycles are at least 10^8 and the reported measurement is the average number of cycles per execution. This process is repeated 15 times in order to retrieve median and quartile information. Each point in the plots is the median of these 15 repetitions and it is accompanied by whiskers that show the most extreme data points falling in the range $[1.5q_1,\ 1.5q_3]$, where q_1 and q_3 are the first and third quartiles.

The execution of all experiments was done using Mediator. This allowed us to easily execute experiments simultaneously on multiple cores of multiple devices, under the guarantee that only one experiment runs at a time per core per device.

5.1.5 Miscellaneous Hardware and Software Configuration Settings

The architectural characteristics of the four processors that we used for the experiments can be found in Section 2.2. Before executing the tests we disabled hyper-threading on Intel Atom and cpu throttling on the three ARM processors.

LGen was configured to use a random search over the search space with sample size 10. In both MKL and ATLAS, we implemented $\alpha Ax + \beta Bx$ with two calls to `cblas_sgemv` and $\alpha = x^T Ay$ as a combination of `cblas_sgemv` and `cblas_sdot`. $\alpha(A_0 + A_1)^T B + \beta C$ 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 4.7.3 for Atom and the Cortex-A processors and gcc 4.7.2 for ARM1176. For 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, since for ARM1176 there were no architectural defaults at the time that this thesis was written and for the Cortex-A processors the architectural defaults made use of the slow VFP unit instead of the much faster, non-IEEE-compliant NEON unit. For all four processors, gemm was re-tuned after the installation in order to improve the performance of ATLAS for small matrix computations, as it is described in the errata section of the ATLAS website[1]. The exact configuration flags that were used for building ATLAS on the target platforms are shown in Table 5.1.

Processor	**Configuration flags**
Intel Atom	`-D c -DWALL -b 64 --nof77 -t 0`
Cortex-A8	`-D c -DWALL -D c -DATL_NONIEEE=1 -D c -DATL_ARM_HARDFP=1 -Si archdef 0 -Fa alg -mfloat-abi=hard -m 1000 --nof77 -t 0`
Cortex-A9	`-D c -DWALL -D c -DATL_NONIEEE=1 -D c -DATL_ARM_HARDFP=1 -Si archdef 0 -Fa alg -mfloat-abi=hard --nof77 -m 1400 -t 0`
ARM1176	`-D c -DWALL -D c -DATL_ARM_HARDFP=1 -Si archdef 0 --nof77 -m 700`

Table 5.1: Configuration flags used for building ATLAS on the target platforms.

5.2 Intel Atom

Three out of the four introduced optimizations apply on Atom, namely the generic loads/stores, the alignment detection, and the new matrix-vector multiplication approach. The positive effect of the generic loads/stores can

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

not be explicitly evaluated, since the SSSE3 ν-BLACs were directly implemented using generic memory access instructions. On the other hand, the remaining two optizations can be turned on/off and we can easily estimate their effect on performance.

In the plots of this section we use the following labelling conventions: *LGen* for the basic version of LGen without any optimizations applied, *LGen-Align* for LGen with the alignment detection optimization enabled, *LGen-MVM* for LGen using the new matrix-vector multiplication approach, and *LGen-Full* for LGen with all optimizations enabled (*LGen-Align* and *LGen-MVM* combined). In the plots associated with BLACs that involve at least one matrix-vector product, we show all four versions of LGen, while in the remaining plots we show *LGen* and *LGen-Full*, since *LGen-MVM* has no effect in these cases and *LGen-Full* contains *LGen-Align*.

A general remark for all experiments is that *LGen-Full* performs better than all competitors, achieving in most cases speedups of 2-4× with respect to the best of them. In several experiments the basic version (*LGen*) performs worse than some or even the majority of the competitors and it is the contribution of the optimizations that boosts the performance of *LGen-Full* to higher values than the ones of the competitors. The relative ranking of the competitors varies a lot depending on the experiment, with MKL and Eigen standing out in most of the cases.

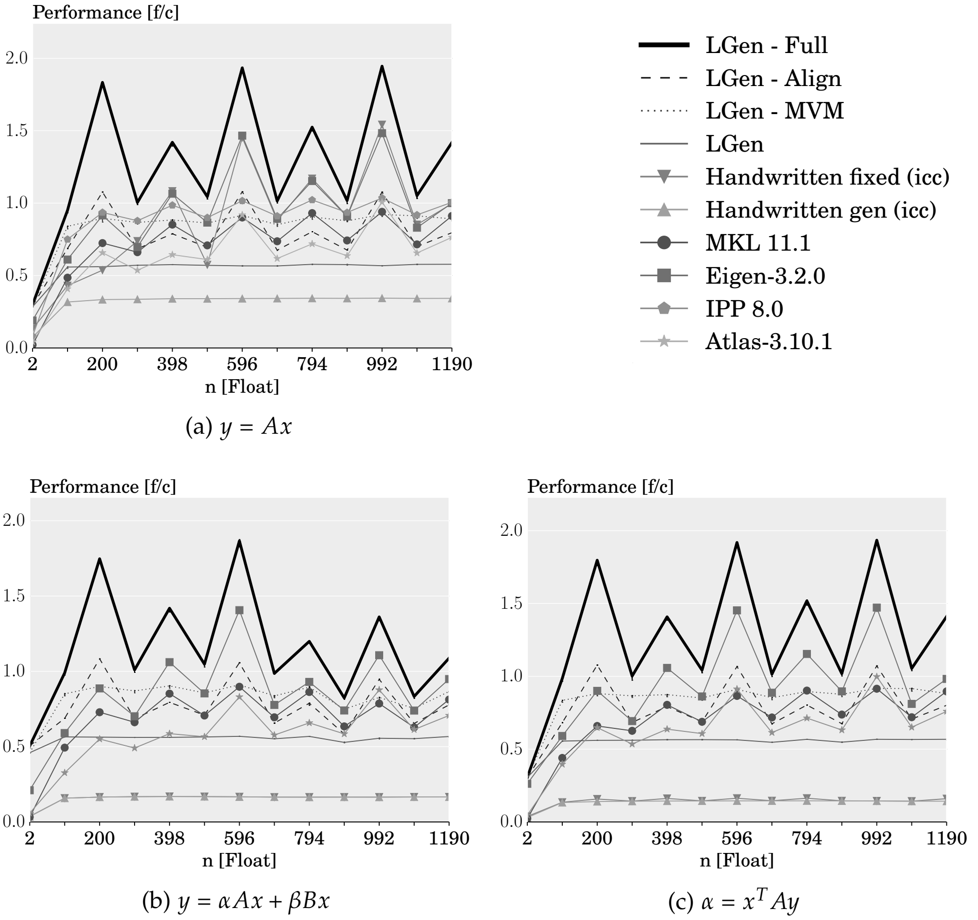
Unless otherwise stated, all the arrays involved in the experiments presented in the following sections were 16-byte aligned.

5.2.1 Matrix-Vector Multiplication

The plots presented in this section show the experimental results for BLACs that contain at least one matrix-vector product. In all experiments LGen performs better than the competitors, achieving speedups of even 5× with respect to the best of them. Fig. 5.1 shows the results for BLACs on $4 \times n$ horizontal panels, while Fig. 5.2 presents similar experiments, but this time involving $n \times 4$ vertical panels. Finally, Fig. 5.3 presents two micro-BLACs on square matrices of sizes between 2 and 10.

The long horizontal panels of Fig. 5.1 are ideal for demonstrating the superiority of the new matrix-vector multiplication approach compared to the old one, since the number of horizontal additions that are replaced by normal additions by the new approach is proportional to the number of columns of the matrix involved in the matrix-vector product (see related analysis in Section 3.3). The results are very similar in all these experiments, with *LGen-MVM* and *LGen-Align* being around 1.5× and 1.2-2× faster than the base version, respectively. The two optimizations are independent of each other and this is why their combination boosts the performance of the generated code, giving a total speedup of 1.7-3× compared to the base version, that clearly ranks *LGen-Full* above all competitors.

The very unstable lines that we observe in the plots of Fig. 5.1 are due to the effect that the shape of the matrices has on alignment. For the matrices whose number of columns N is divisible by 4 (alignment length), all rows are aligned, which results to 100% aligned memory accesses. If $N \bmod 4 = 2$, half of the rows are aligned and if $N \bmod 4 \in \{1, 3\}$, only one forth of them is aligned, leading to 25% of aligned accesses. The drop of performance after the value $n = 695$ in Fig. 5.1(b) is due to cache misses, because the matrices are too large to fit in the L1 data cache.



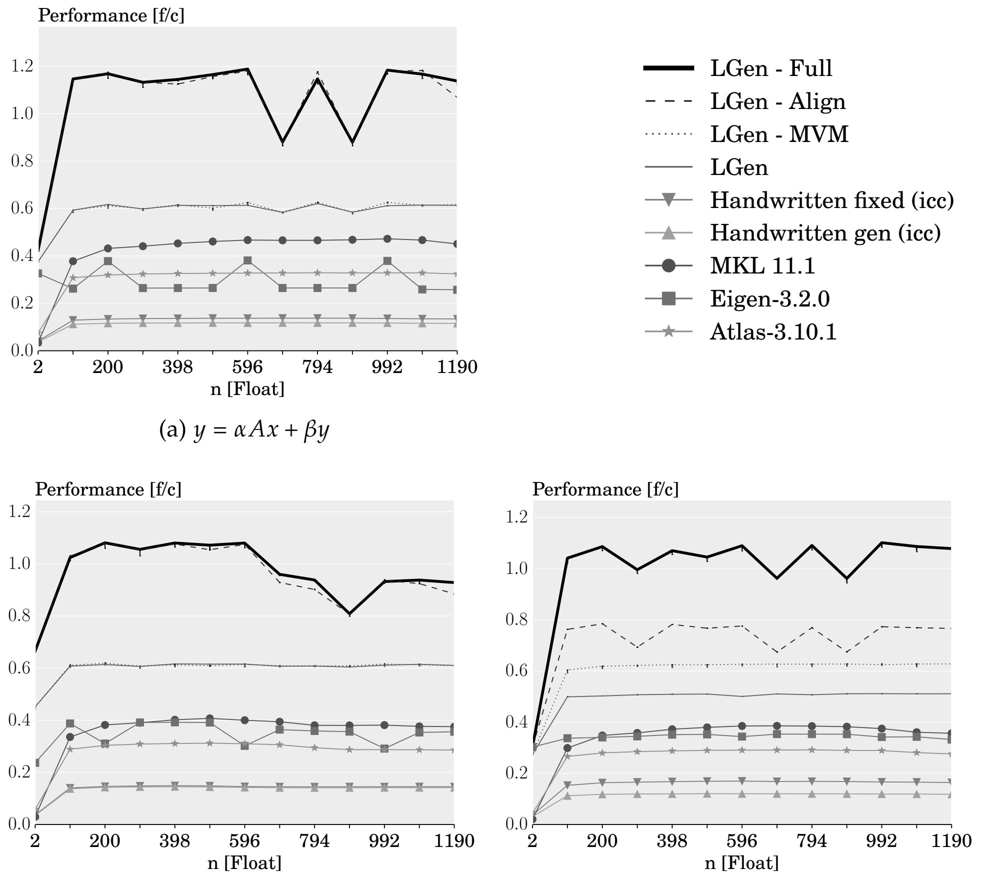
(a) $y = Ax$

(b) $y = \alpha Ax + \beta Bx$

(c) $\alpha = x^T Ay$

Figure 5.1: BLACs containing matrix-vector multiplications, where the matrices have size 4 × n.

Eigen stands out in all experiments of Fig. 5.1 with performance that is considerably higher than the remaining competitors, but always lower than the one of *LGen-Full*. icc on handwritten fixed-size code is also competitive for the simple BLAC $y = Ax$ (Fig. 5.1(a)), but its performance decreases radically as the computations become more complicated.



(a) $y = \alpha Ax + \beta y$

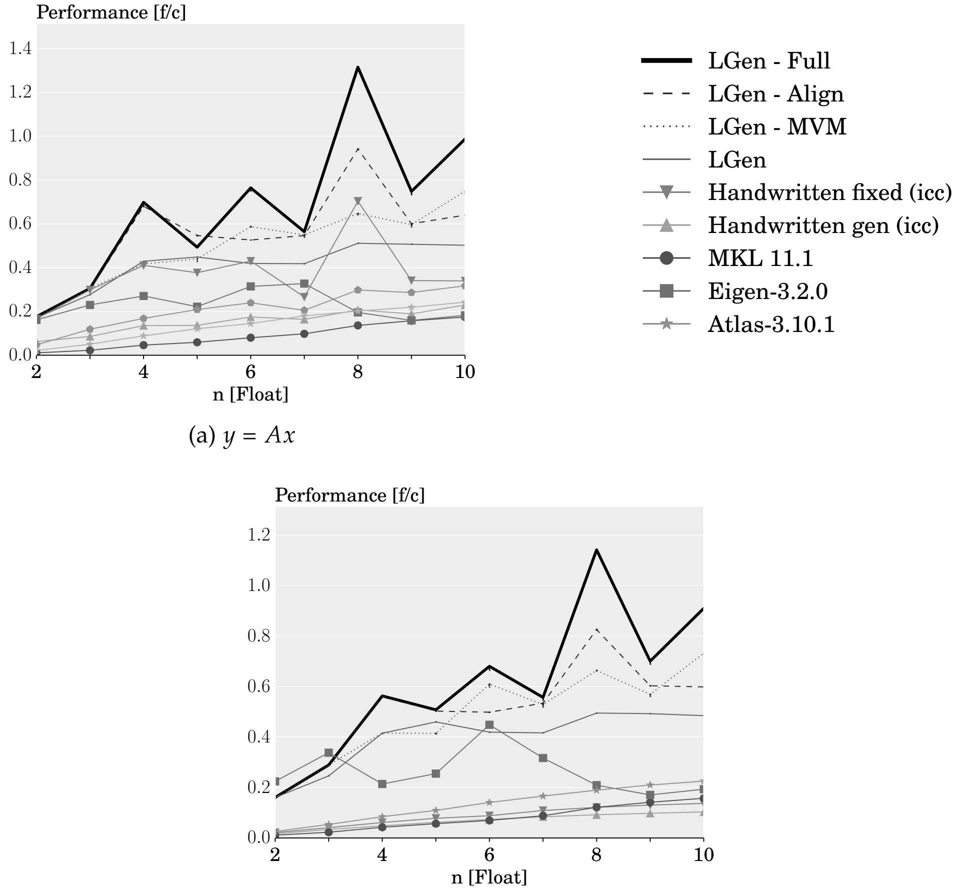
(b) $y = \alpha Ax + \beta Bx$

(c) $\alpha = x^T Ay$

Figure 5.2: BLACs containing matrix-vector multiplications, where the matrices have size $n \times 4$.

Applying the new matrix-multiplication methodology on computations involving vertical panels has no significant effect on performance, as we can see in Fig. 5.2(a)-(b). After applying the 4×4 inner tiling, the resulting matrices have only one tile per row. This leads to a degenerate case of the new matrix-vector multiplication approach that is identical to the old approach: For each 4×4 tile we apply MVH with the 4×1 vector and then RR on the resulting 4×4 matrix, which is exactly the same as simply multiplying each tile with the vector. The difference between the performance of the old and the new matrix-vector multiplication approach in Fig. 5.3(c) is explained by the fact that the result of Ay is a $n \times 1$ vector which is then multiplied from the left with the $4 \times n$ vector x^T and the new approach has a significant impact in this computation, since the number of horizontal additions that are avoided through its application is proportional to n.

The performance of LGen in Fig. 5.2 is much more stable than in Fig. 5.1. This is because the matrices in this case are $n \times 4$ vertical panels and, therefore, all their rows are aligned. The steep drops that we notice in most plots for $n = 695$ and $n = 893$ are explained by the fact that $\lfloor n/4 \rfloor$ for these values of n is a prime number, which makes impossible to further tile these matrices without introducing more leftovers. Since one limitation of LL is that at most one level of tiling may introduce leftovers, no further tiling is allowed in these cases, which has a noticeable impact on performance.



(a) $y = Ax$

(b) $\alpha = x^T Ay$

Figure 5.3: Micro-BLACs containing matrix-vector multiplications, where the matrices have size $n \times n$.

For the micro-BLACs of Fig. 5.3, the fully unrolled generated code by *LGen-Full* achieves up to 5.5× better performance (Fig. 5.3(b)) than the best competitor. The peaks at $n = 4, 6, 8$ are explained by the better alignment of the matrix rows and the low amount of leftover code for these sizes.

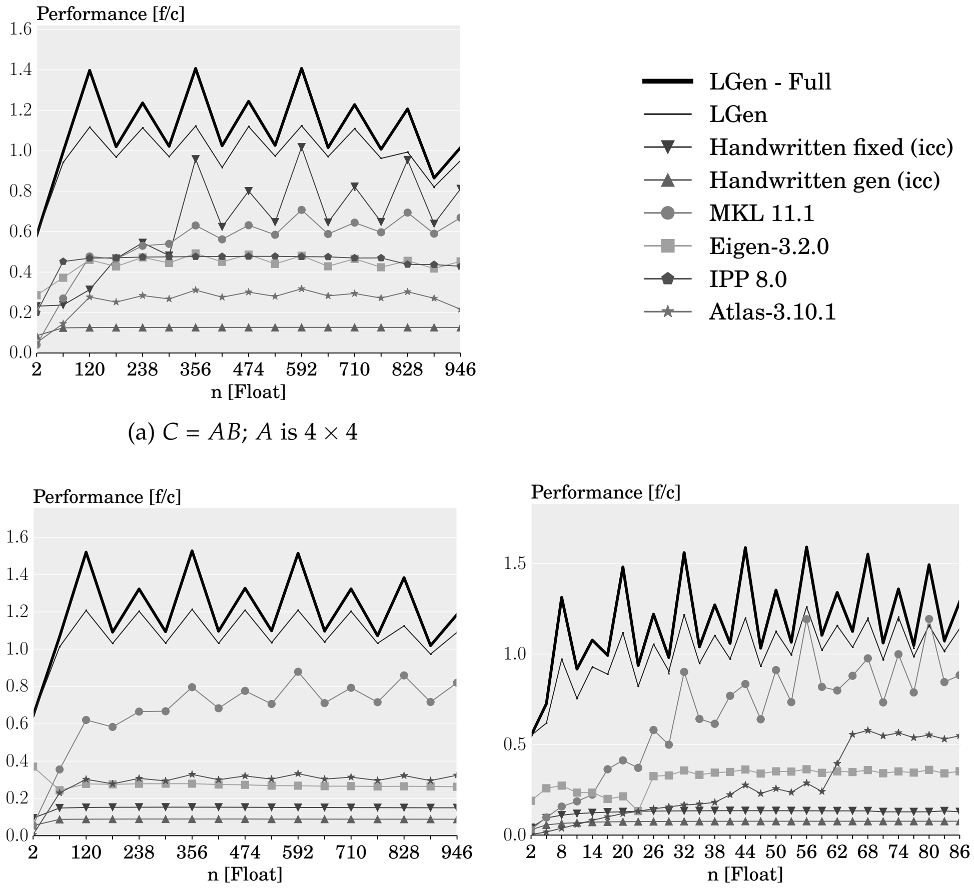
5.2.2 Matrix-Matrix Multiplication

In this section we present experiments with BLACs that are based on matrix-matrix multiplication. The $4 \times 4 \times 4$ matrix multiplication ν-BLAC for SSSE3 is implemented as follows: The i row of the result is computed by multiplying the ij element of the first operand with the j row of the second operand and accumulating over j. Therefore, the elements of the left operand are loaded one by one using the `_mm_load1_ps` instruction, while each 1×4 row of the right operand is loaded into an SSE vector using the instruction `_mm_loadu_ps` or `_mm_load_ps`, depending on its alignment. In this respect, the alignment of the right operand has a significant effect on performance, while the alignment of the left operand has no effect on it.

Fig. 5.4 shows some experiments based on matrix-matrix multiplication where the right operand is a $4 \times n$ horizontal panel, while the left operand is either a 4×4 block or a $n \times 4$ vertical panel. The congruence mod 4 of the number of columns of the right operand determines the percentage of aligned memory accesses in a way similar to the one discussed in Section 5.2.1 for Fig. 5.1. In the simple $4 \times 4 \times n$ matrix multiplication experiment shown in Fig. 5.4(a), icc produces more efficient vectorized code than the other competitors, although its performance is significantly lower than the one of *LGen-Full* for all values of n. In the remaining plots of Fig. 5.4 MKL is clearly the best competitor, producing code that is even 3× faster than the other competitors but still 10-80% slower than *LGen-Full*.

The experiments presented in Fig. 5.5 test the same BLACs as the ones of Fig. 5.4, with the difference that this time the right operand has always 4 columns and, therefore, all accesses of this matrix are aligned. This is the reason behind the straight lines that we observe in Fig. 5.5(a),(b) for *LGen-Full*. The difference between the performance of *LGen-Full* and *LGen* is much smaller than the one observed in the matrix-vector multiplication experiments of Section 5.2.1, which is explained by the fact that matrix-matrix multiplication involves a higher ratio of computation to memory accesses than matrix-vector multiplication. Thus, the use of aligned loads/stores affects much more the performance of the former than the one of the latter.

Finally, in Fig. 5.6 we can see the results for small matrix-matrix multiplications involving square matrices. The performance of *LGen-Full* reaches 1.3 flops/cycle, while IPP, which ranks second, achieves a peak performance of 0.75 flops/cycle for $n = 6$.



(a) $C = AB$; A is 4×4

(b) $C = \alpha AB + \beta C$; A is 4×4

(c) $C = \alpha(A_0 + A_1)^T B + \beta C$; A_0, A_1 are $4 \times n$

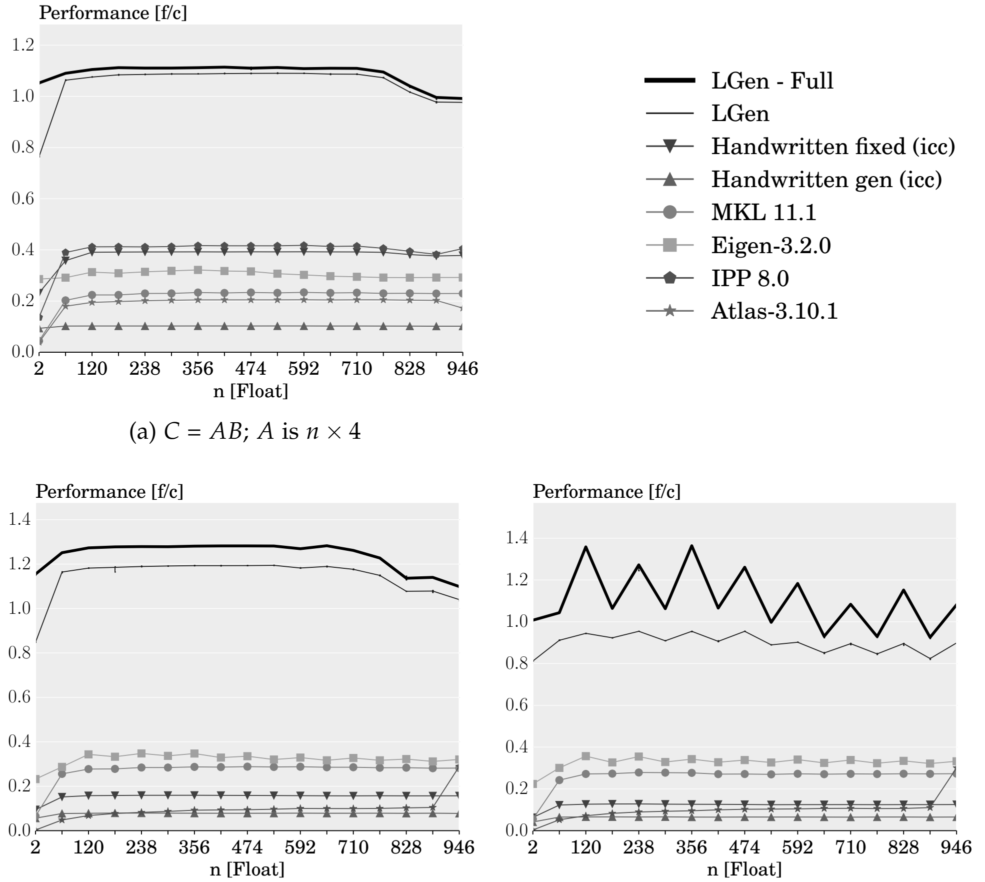
Figure 5.4: BLACs containing matrix-matrix multiplications, where the right operand has size $4 \times n$.

5.2.3 BLACs on matrices with varying shapes

Fig. 5.7 presents some experiments on matrices whose shape varies between vertical and horizontal panels. In all experiments LGen achieves the best performance, with Eigen and MKL following. For the BLACs based on matrix-matrix multiplication shown in Fig. 5.7(b),(c) the performance of Eigen and MKL approaches the one of LGen as the matrices become wider, which is an indication that these libraries may not be optimized for computations on narrow panels.

5.2.4 Alignment Detection

This section contains experiments that are targeted to the alignment detection optimization. Of course, apart from the experiments presented in this



(a) $C = AB$; A is $n \times 4$

(b) $C = \alpha AB + \beta C$; A is $n \times 4$

(c) $C = \alpha(A_0 + A_1)^T B + \beta C$; A_0, A_1 are $4 \times n$

Figure 5.5: BLACs containing matrix-matrix multiplications, where the right operand has 4 columns.

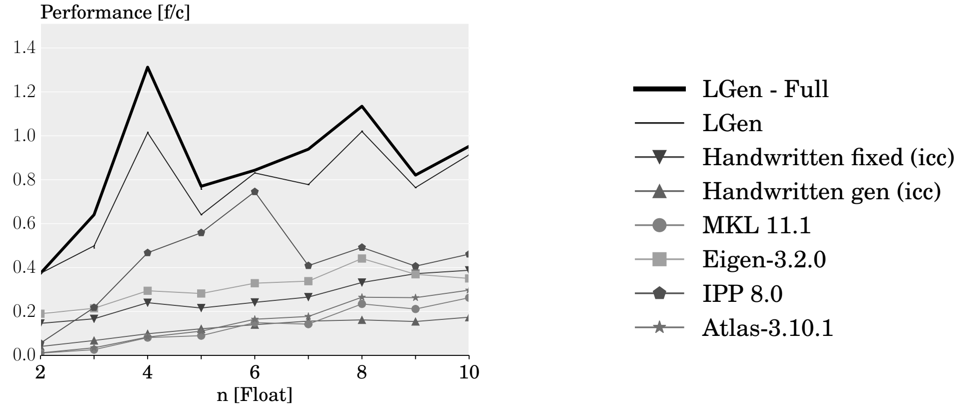
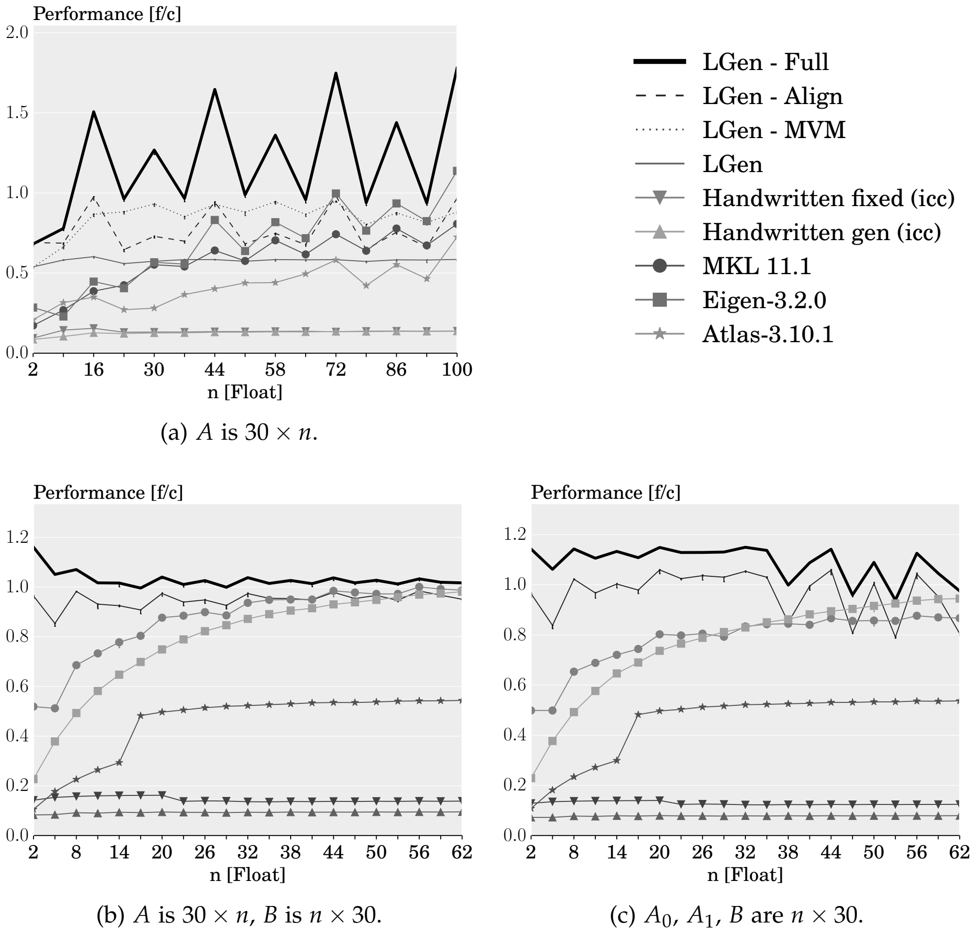


Figure 5.6: $C = AB$ micro-BLAC, where both A and B are small $n \times n$ matrices.



(a) A is $30 \times n$.

(b) A is $30 \times n$, B is $n \times 30$.

(c) A_0, A_1, B are $n \times 30$.

Figure 5.7: BLACs on matrices with varying shapes. (a): $y = \alpha Ax + \beta y$; (b): $C = \alpha AB + \beta C$; (c): $\mathrm{C} = \alpha(A_0 + A_1)^T B + \beta C$.

section, the impact of alignment detection on performance can be seen in all the experiments included in the previous sections, especially the ones involving matrix additions and matrix-vector multiplications, which are much more memory-intensive computations than matrix-matrix multiplications.

In Fig. 5.8 we can see the results for $y = \alpha x + y$, a computation that heavily depends on efficient memory accesses, since the ratio of memory accesses to arithmetic operations is 3:2. *LGen-Align* achieves a speedup of over 4× over *LGen*, showing the great importance of using aligned memory access instructions on Atom. The performance drop for $n > 3000$ is due to the fact that vectors of this size do not completely fit in the L1 data cache. For this simple BLAC, icc is better than all competitors and, especially for handwritten code with fixed sizes, it produces code that is more than twice as fast as MKL and the remaining competitors.

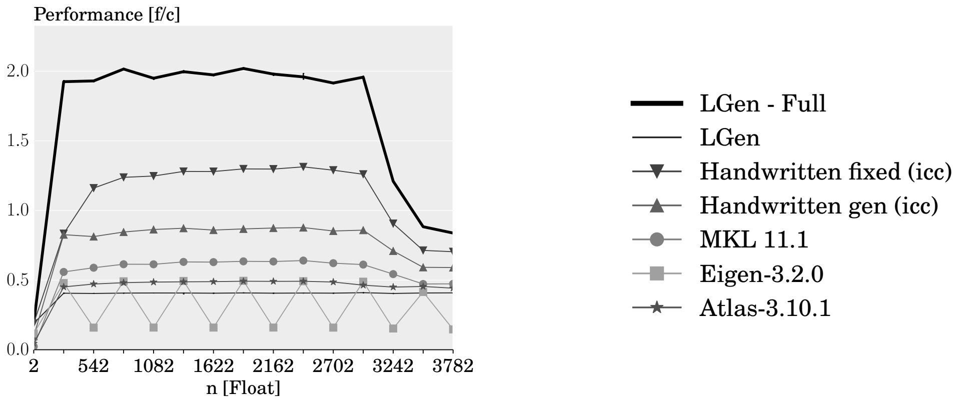
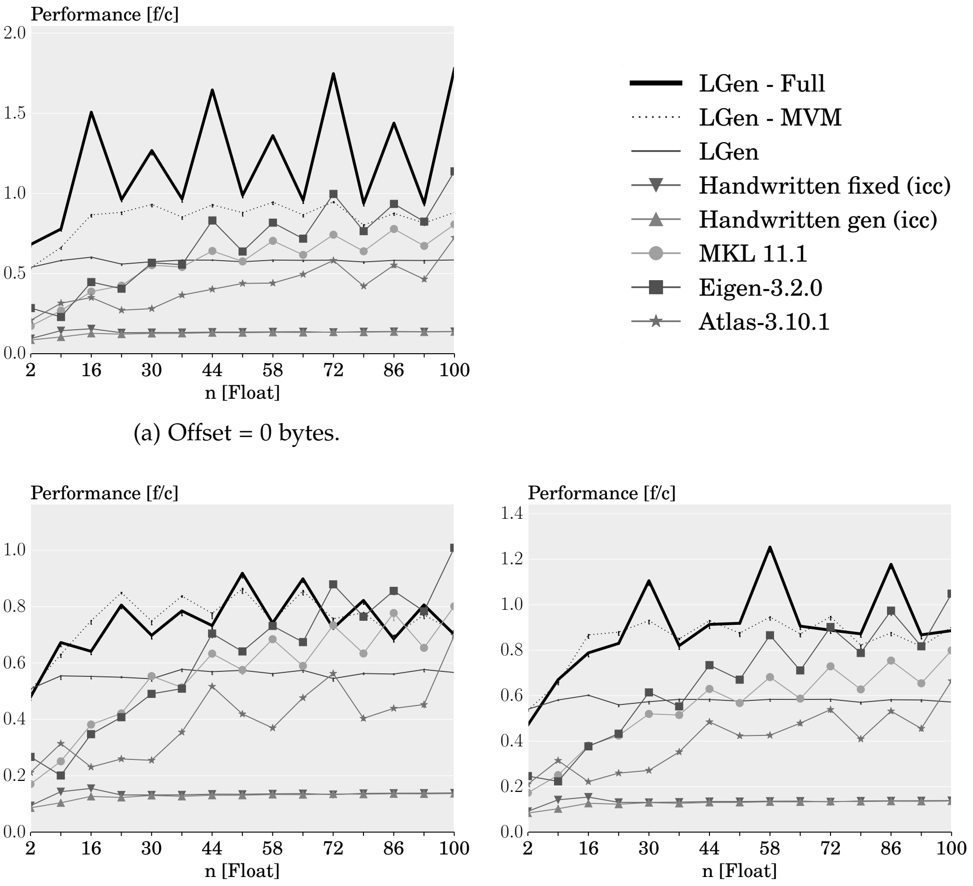


Figure 5.8: $y = \alpha x + y$, where x, y are $1 \times n$ vectors.



(a) Offset = 0 bytes.

(b) Offset = 4 bytes.

(c) Offset = 8 bytes.

Figure 5.9: $y = \alpha Ax + \beta y$, where A is $30 \times n$. All arrays are allocated at an aligned memory address plus an offset.

To test the way LGen handles arbitrary alignment of the input arrays, we executed an experiment using the BLAC $y = \alpha Ax + \beta y$ together with a modified caller of the kernel that allocates all arrays at an aligned memory location plus a given offset. We chose this specific BLAC because it is based on matrix-vector multiplication, which relies greatly on the alignment detection optimization since it involves a high ratio of memory accesses to computation. The values that we used as offset were 0 bytes, 4 bytes (i.e. one float), and 8 bytes (i.e. two floats). The results of the experiment are presented in Fig. 5.9.

Although for aligned arrays LGen achieves much better performance than the competitors (Fig. 5.9(a)), for unaligned arrays (Fig. 5.9(b),(c)) Eigen's and MKL's performance is comparable and, for large values of n, even better than LGen's. After investigating the assembly code for these libraries, we realized that they apply loop peeling, which can significantly increase the amount of aligned accesses in these experiments. For example, for an offset equal to 4 bytes (Fig. 5.9(b)) and even values of n, the vector x and all rows of A are unaligned, which leads to 100% unaligned memory accesses for LGen. On the other hand, Eigen peels the part of the loop that corresponds to the first 3 columns of A (and the first 3 elements of x) and uses aligned accesses for the remaining of the computation. This is the reason why for even values of n LGen obtains its worst performance, while Eigen and MKL reach their performance peaks. For odd values of n, one out of four rows of A is aligned, which leads to 25% aligned accesses of A for LGen and this is the explanation why LGen reaches its peak performance for these values.

For offsets of 8 bytes (Fig. 5.9(c)) the picture is slightly different. For values of n such that $n \bmod 4 = 0$ all rows of A are unaligned, for odd values of n 25% of the rows of A are aligned and for values of n such that $n \bmod 4 = 2$ this percentage rises to 50%. The large amount of aligned accesses in the last case is reflected in the high peaks that we notice in Fig. 5.9(c). Similarly to Fig. 5.9(b), Eigen handles the unaligned arrays better than all the other competitors, with performance that continuously rises as n grows.

Although *LGen-Full* is a more optimized version of *LGen-MVM*, since it additionally applies the alignment detection optimization, we notice that the former often obtains lower performance than the latter. Investigating the generated assembly code revealed that icc is a lot more conservative when handling the very large code generated by *LGen-Full* (alignment versioning for $y = \alpha Ax + \beta y$ results to the generation of 65 versions of the same computation, leading to a kernel with a total size in the order of hundreds of thousands lines). As an example, we mention that for one of our generated kernels icc unrolled some loops of the code generated by *LGen-MVM*, but left untouched the same loops of the code generated by *LGen-Full*. As an indication of the difference in the way icc handles these two versions of code, the ratio of the sizes of the two assembly files in terms of lines of code was 1:9, while the same ratio for the source files was 1:65 (due to the 65 versions). In this respect, although the source code that is executed in these two cases is almost identical, the generated assembly after the compilation differs considerably, making *LGen-Full* and *LGen-MVM* difficult to compare.

5.3 ARM Cortex-A8

In the following sections we present the experiments that we did on Cortex-A8, listed according to the BLAC categorization presented in section 5.1.1. The optimizations that apply on this processor are the generic load/store instructions and the specialized ν-BLACs. As we have already mentioned before for Atom, it is not possible to explicitly evaluate the effects of the generic instructions, because the NEON ν-BLACs were directly implemented using them. Moreover, the effect of the specialized ν-BLACs is visible only in computations involving small matrices with leftovers or narrow panels whose small dimension is not divisible by ν. Therefore, we will compare the versions of LGen with and without specialized ν-BLACs (*LGen - Full* and *LGen*, respectively) only in the experiments that fulfil at least one of these two conditions. In all the remaining experiments we will present only the version *LGen - Full*.

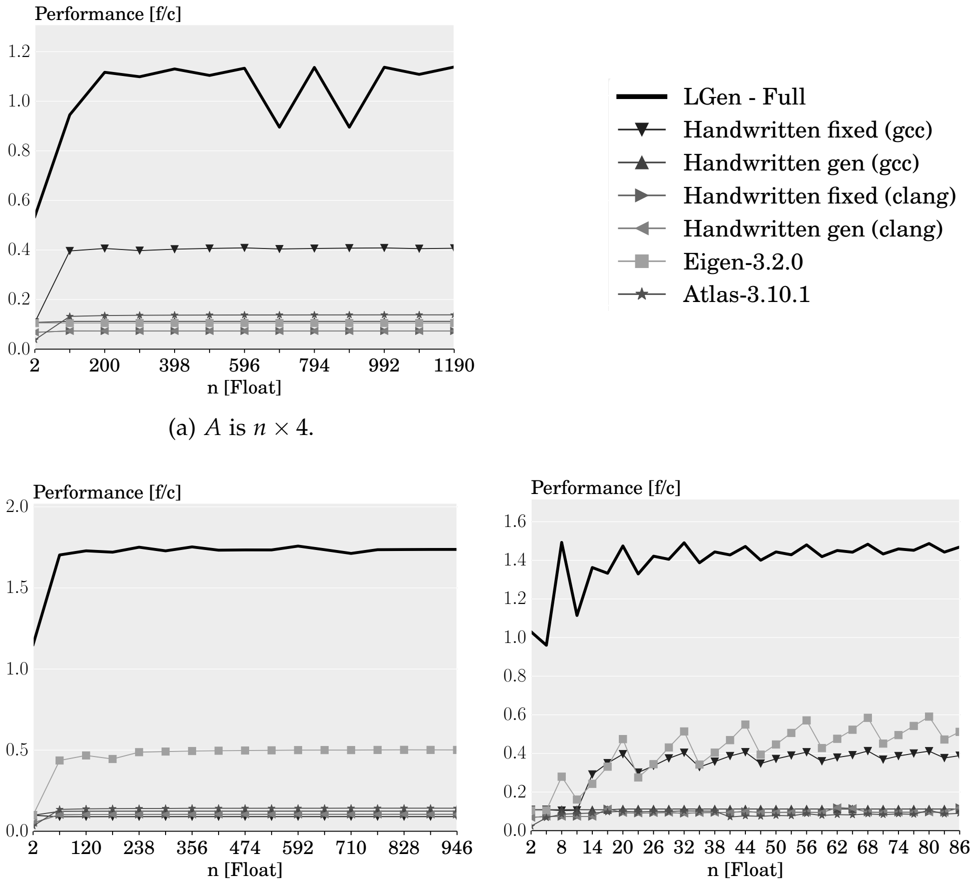
Regarding the compilers used, a general observation coming from the majority of the experiments that we did is that gcc is better in auto-vectorization for Cortex-A8 than clang, while clang does better instruction scheduling and register allocation than gcc. For this reason, we include experiments with handwritten code compiled with both clang and gcc, while we choose clang for compiling the already vectorized code generated by LGen and the code using the Eigen library. For the experiments with ATLAS we used gcc, since this is the recommended compiler for configuring and using ATLAS.

5.3.1 Simple BLACs

As it is shown in Fig. 5.10, in all experiments with simple BLACs LGen achieves 2-9× higher performance than the best competitor. The main source of inefficiency of the competitors is the mixing of scalar and vectorized code, which due to microarchitectural limitations of Cortex-A8 (see section 2.2.2) leads to very poor performance. This also applies to most of the experiments on Cortex-A8 that are presented in the following sections.

5.3.2 BLACs that closely match BLAS

Fig. 5.11 presents a set of experiments on BLACs that closely match BLAS. In general, the results of these experiments are very similar to the ones



(a) A is $n \times 4$.

(b) A is $4 \times n$, B is $n \times 4$.

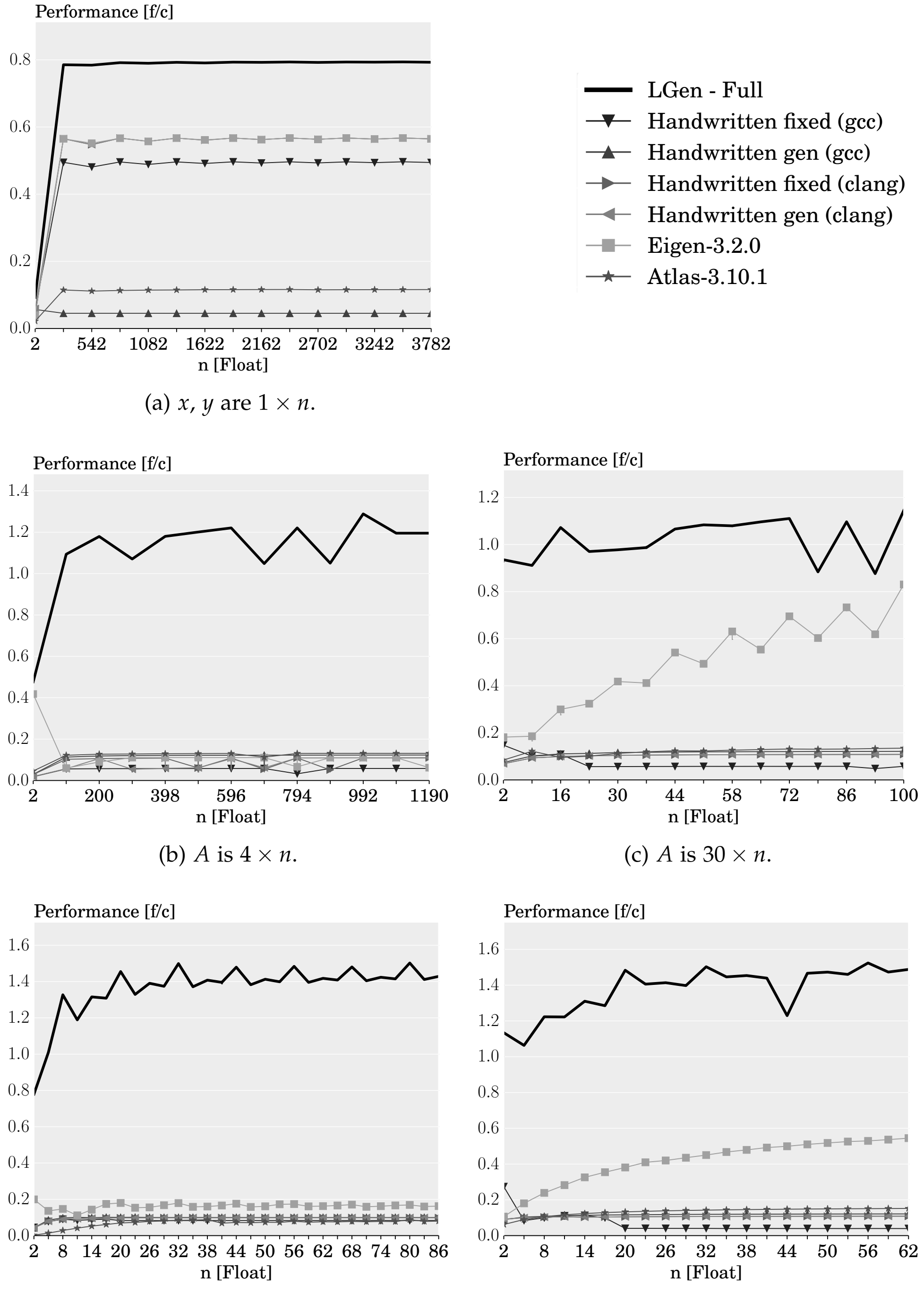
(c) A is $n \times 4$, B is $4 \times n$.

Figure 5.10: Simple BLACs. (a): $y = Ax$; (b)-(c): $C = AB$.

shown in the previous section. *LGen - Full* performs always better than all competitors, achieving speedups of even more than 7× with respect to them. For the easily vectorizable BLAC $y = \alpha x + y$ (Fig. 5.11(a)), both Eigen and gcc on fixed-size handwritten code obtain a reasonably good performance of 0.5-0.6 flops/cycle, while for the remaining experiments the performance of the competitors is very poor, in most cases less than 0.2 flops/cycle, for the reasons that we described in the previous section.

5.3.3 BLACs that require more than one BLAS call

The results for these experiments can be found in section B.2 of the Appendix.



(a) x, y are $1 \times n$.

(b) A is $4 \times n$.

(c) A is $30 \times n$.

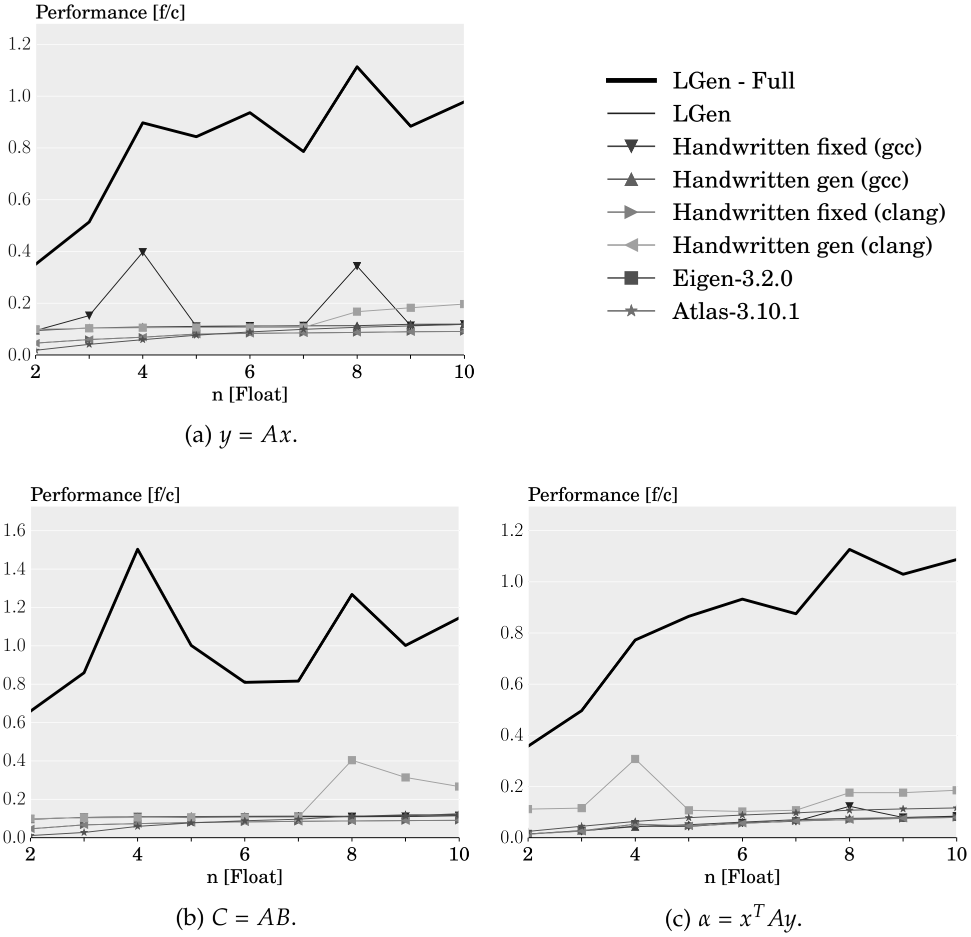
(d) A is $n \times 4$, B is $4 \times n$.

(e) A is $30 \times n$, B is $n \times 30$.

Figure 5.11: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b)-(c): $y = \alpha Ax + \beta y$; (d)-(e): $C = \alpha AB + \beta C$.

5.3.4 Micro-BLACs

Fig. 5.12 presents three experiments involving simple computations on small square matrices. The only cases that the competitors obtain considerably good performance are for matrix sizes equal to 4 or 8, since in these cases vectorization is straightforward. In the other cases combining scalar and vector operations keeps the performance of the competitors in low levels. On the other hand, LGen obtains high performance even for matrix sizes that are not 4 or 8. The small leftover computations benefit a lot from LGen's approach, since embedding the leftover tiles in vectors and applying vector arithmetic operations on them is much more efficient than applying scalar arithmetic operations on single elements in the Cortex-A8 microarchitecture.



(a) $y = Ax$.

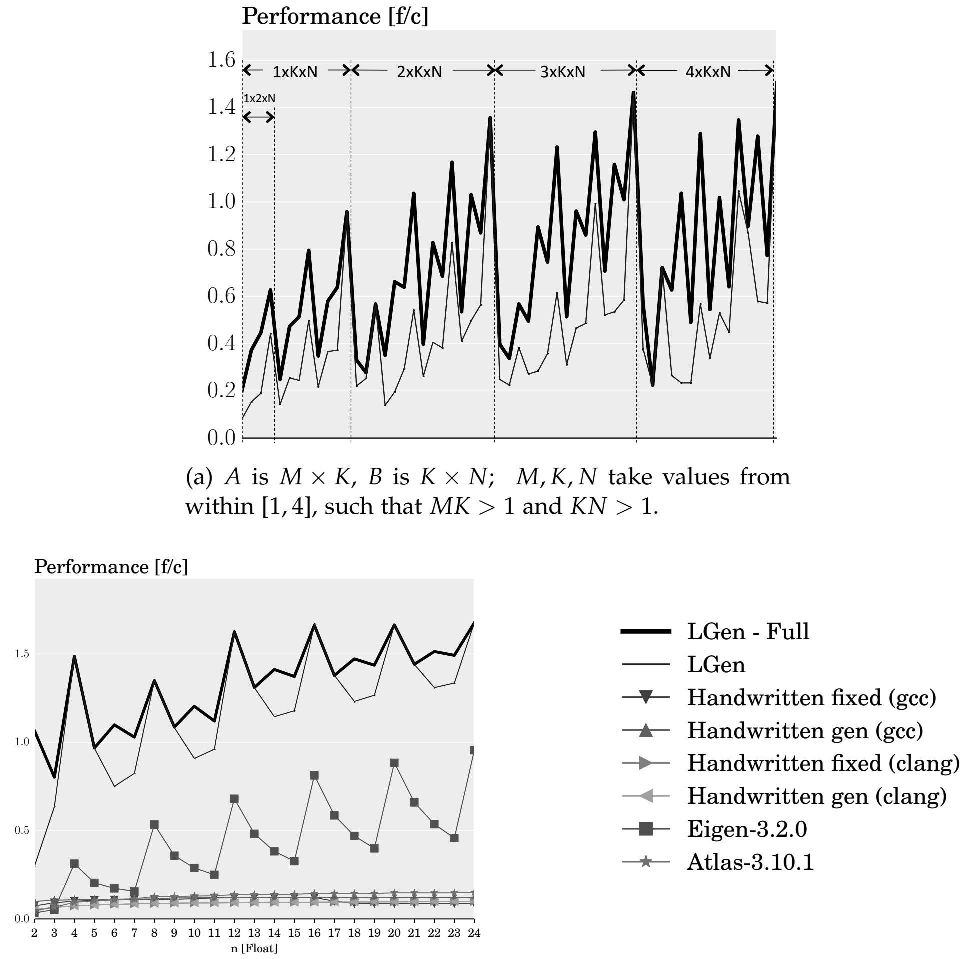
(b) $C = AB$.

(c) $\alpha = x^T Ay$.

Figure 5.12: Micro-BLACs. All matrices have size $n \times n$.

5.3.5 Specialized ν-BLACs

In order to reveal the impact of the specialized ν-BLACs, we executed the two experiments shown in Fig. 5.13. The first experiment (Fig. 5.13(a)) compares the old ν-BLACs implementation with the new one on matrix-matrix multiplications involving matrices of sizes in the range $[1, 4]$. Each point on the x-axis represents a different selection of matrix sizes. Comparing the two approaches using such computations can give us an indication of the efficiency of the specialized ν-BLACs.



(a) A is $M \times K$, B is $K \times N$; M, K, N take values from within $[1, 4]$, such that $MK > 1$ and $KN > 1$.

(b) A is $100 \times n$, B is $n \times n$.

Figure 5.13: $C = AB$ with a large percentage of leftovers.

In order to evaluate the impact of the specialized ν-BLACs in a more realistic setting, we chose a $100 \times n \times n$ matrix multiplication $C = AB$, with n taking values in the interval $[2, 24]$ (Fig. 5.13(b)). For values of n that are not divisible by 4, A has a large percentage of leftover tiles, the efficient handling of which has a significant impact on performance.

As it is shown in Fig. 5.13(a), there are a few cases that the two implementations have the same performance, which is due to the fact that for some matrix sizes the specialized ν-BLACs have exactly the same implementation as the old ν-BLACs. For the remaining cases, the specialized ν-BLACs are considerably more efficient than the old ones, with speedups reaching 4×. The performance gain for handling leftovers using the specialized ν-BLACs is also obvious in Fig. 5.13(b), since for the values of n such that $n \bmod 4 \in \{2, 3\}$ the performance of the old ν-BLACs deteriorates much more than the one of the specialized ν-BLACs. The difference between the performance of the two implementations becomes smaller as n increases, since the percentage of leftover computations decreases for higher values of n and computations on ν-sized tiles are processed in the same way by the two approaches.

5.4 ARM Cortex-A9

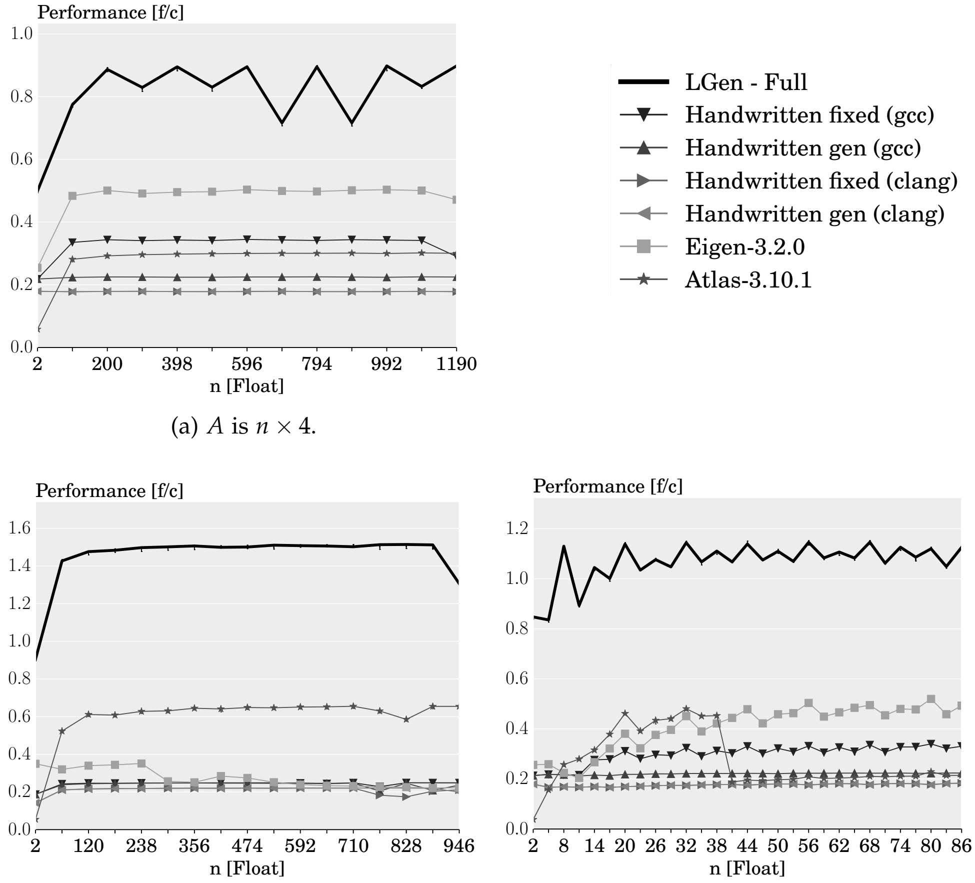
Cortex-A9 implements the same ISA as Cortex-A8, therefore the same optimizations that we used for Cortex-A8 also apply to it. The differences between these two Cortex-A processors appear at a microarchitectural level. Two critical differences are the following: (a) Scalar floating point operations are executed much more efficiently on Cortex-A9 and (b) a memory access and an arithmetic NEON instruction can be issued at the same cycle on Cortex-A8, while this is not possible on Cortex-A9. The former is the main reason behind the relatively better performance of the competitors on Cortex-A9 and the latter explains the slightly lower performance values that we obtain on Cortex-A9 compared to Cortex-A8.

5.4.1 Simple BLACs

Fig. 5.14 shows the experimental results for simple BLACs on Cortex-A9. The two steep drops of LGen's performance that we observe for $n = 695$ and $n = 893$ are explained by its inability to apply a second level of tiling over these matrices, as we have already seen in similar experiments for other processors.

For matrix-vector multiplication (Fig. 5.14(a)), Eigen is the best competitor achieving performance that is always 30-40% lower than LGen's. For matrix-matrix multiplication (Fig. 5.14(b),(c)) the difference between LGen and the competition is even larger, with LGen being more than 2× faster, mostly obtaining performance in the range 1-1.5 flops/cycle. For the $4 \times n \times 4$ multiplication (Fig. 5.14(b)) ATLAS gets the highest performance among the competitors, reaching 0.6 flops/cycle. For the rank-4 update (Fig. 5.14(c))

ATLAS is slightly better than Eigen for low values of n. From $n = 41$ onwards, ATLAS loses more than half of its performance while Eigen continues obtaining values in the range 0.4-0.5 flops/cycle, which is still less than half of LGen's performance.



(a) A is $n \times 4$.

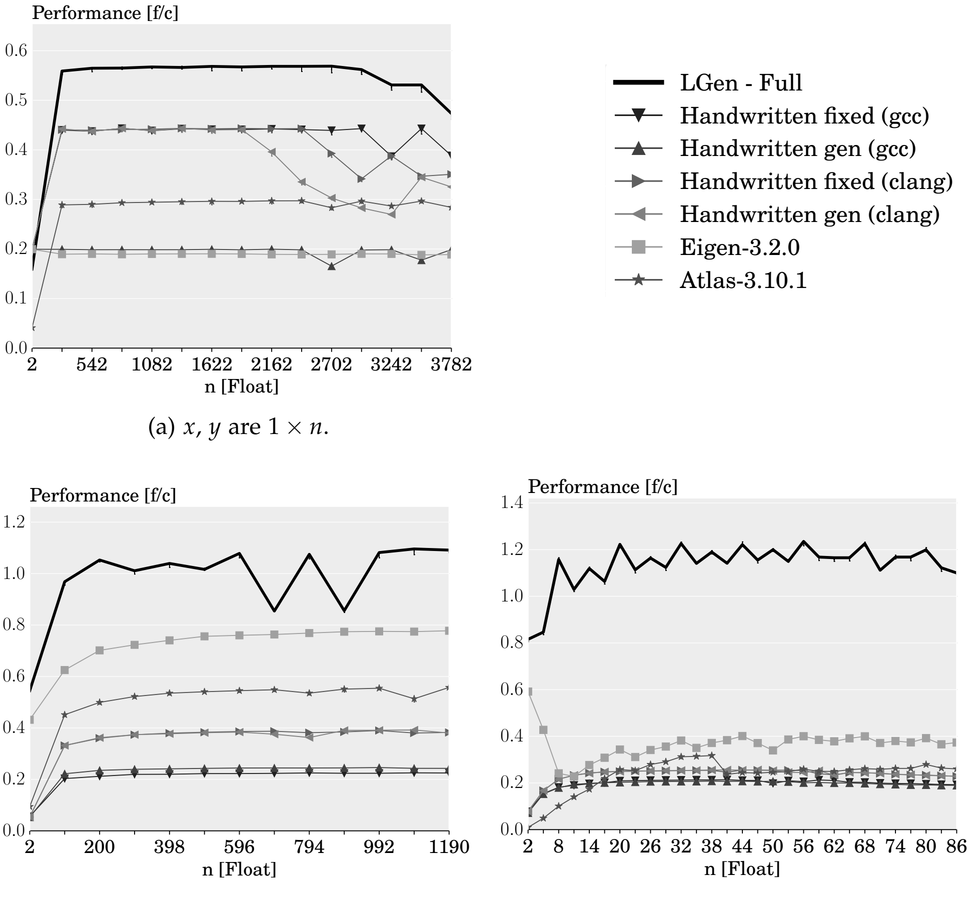
(b) A is $4 \times n$, B is $n \times 4$.

(c) A is $n \times 4$, B is $4 \times n$.

Figure 5.14: Simple BLACs. (a): $y = Ax$; (b)-(c): $C = AB$.

5.4.2 BLACs that closely match BLAS

Fig. 5.15 presents a set of experiments on BLACs that closely match BLAS. For the memory-intensive computation $y = \alpha x + y$ (Fig. 5.15(a)) LGen's performance (0.6 flops/cycle) is considerably lower than in the remaining experiments, which is explained by the single instruction issue restriction of the Cortex-A9 NEON pipeline. Both gcc and clang manage to auto-vectorize the fixed-sized handwritten code efficiently, following LGen in the ranking with 0.45 flops/cycle. The results for the computations $y = \alpha Ax + \beta y$, $C = \alpha AB + \beta C$ are very similar to the ones for $y = Ax$, $C = AB$, presented in the previous section.



(a) x, y are $1 \times n$.

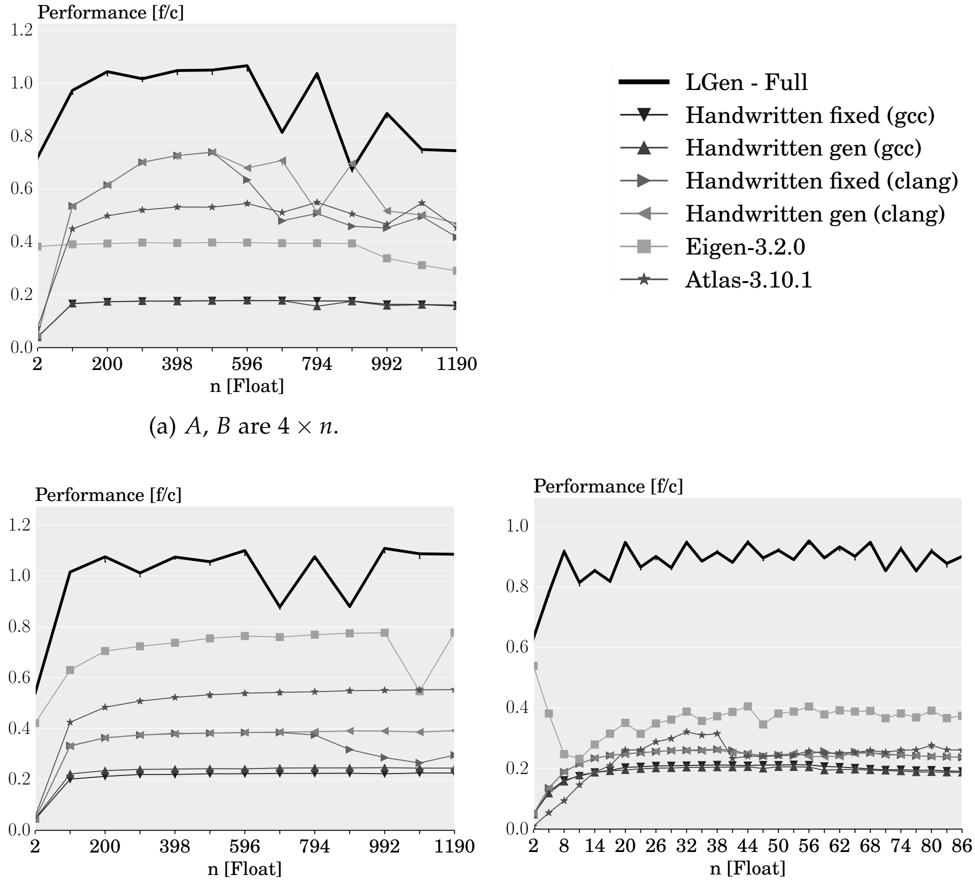
(b) A is $4 \times n$.

(c) A is $n \times 4$, B is $4 \times n$.

Figure 5.15: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b): $y = \alpha Ax + \beta y$; (c): $C = \alpha AB + \beta C$.

5.4.3 BLACs that require more than one BLAS call

Fig. 5.16 shows the results for BLACs that are translated to more than one BLAS call. The decreasing performance that we observe in Fig. 5.16(a) is a result of reaching the L1 data cache size limit. For the BLACs based on matrix-vector multiplication (Fig. 5.16(a)-(b)) LGen is around 1.5× faster than the best competitor, which is clang for $y = \alpha A x + \beta B x$ and Eigen for $\alpha = x^T A y$. For the more computation-intensive BLAC $C = \alpha(A_0 + A_1)^T B + \beta C$ (Fig. 5.16(c)) the difference between LGen and the competitors is much greater, with LGen being up to 3× faster than the best of them.



(a) A, B are $4 \times n$.

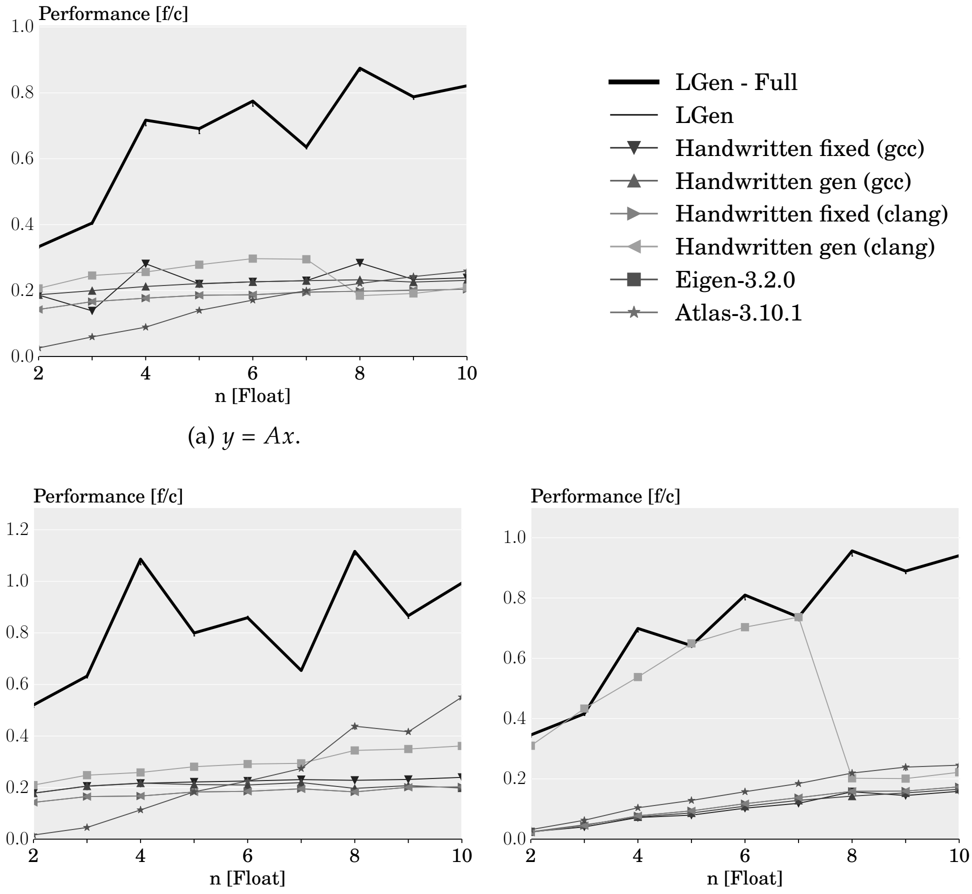
(b) A is $4 \times n$.

(c) A_0, A_1 are $4 \times n$, B is $4 \times n$.

Figure 5.16: BLACs that require more than one BLAS call. (a): $y = \alpha A x + \beta B x$; (b): $\alpha = x^T A y$; (c): $C = \alpha(A_0 + A_1)^T B + \beta C$.

5.4.4 Micro-BLACs

As we can see in Fig. 5.17, LGen performs significantly better than the competitors for the micro-BLACs $y = Ax$ (Fig. 5.17(a)) and $C = AB$ (Fig. 5.17(b)), even in cases where there is a large percentage of leftovers. For $\alpha = x^T Ay$ (Fig. 5.17(c)) Eigen's performance is comparable to LGen's for sizes up to $n = 7$, while it decreases radically for $n = 8$ and for $n > 8$ it remains at values that are less than one fourth of LGen's performance.



(a) $y = Ax$.

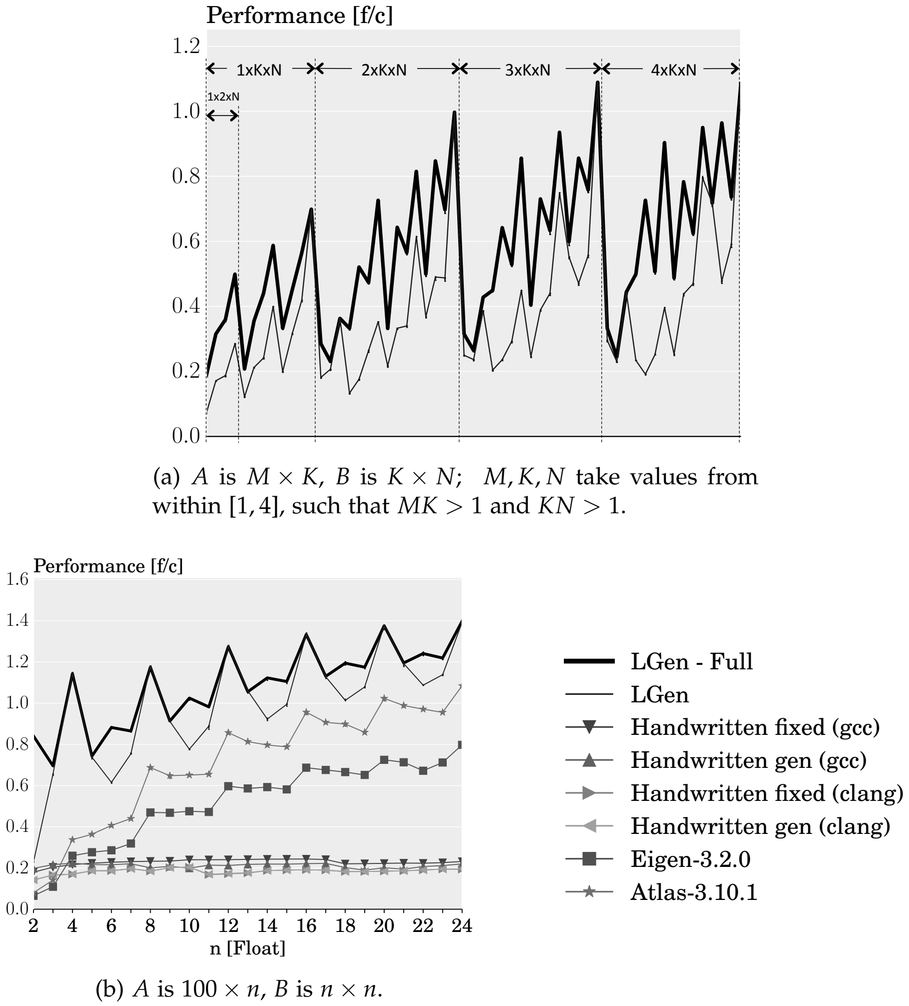
(b) $C = AB$.

(c) $\alpha = x^T Ay$.

Figure 5.17: Micro-BLACs. All matrices have size $n \times n$.

5.4.5 Specialized ν-BLACs

Similarly to what is described in section 5.3.5 for Cortex-A8, we executed the two experiments presented in Fig. 5.18 in order to evaluate the performance gain of using the specialized ν-BLACs on Cortex-A9. For both of them, the pattern of LGen's performance is the same as the one observed in the plots of section 5.3.5, with the new ν-BLACs being considerably more efficient than the old ones for computations with a large percentage of leftovers. The values that we obtain for Cortex-A9 are slightly lower than the ones for Cortex-A8, because of the different characteristics of the two microarchitectures regarding instruction issuing (see section 2.2).



(a) A is $M \times K$, B is $K \times N$; M, K, N take values from within $[1, 4]$, such that $MK > 1$ and $KN > 1$.

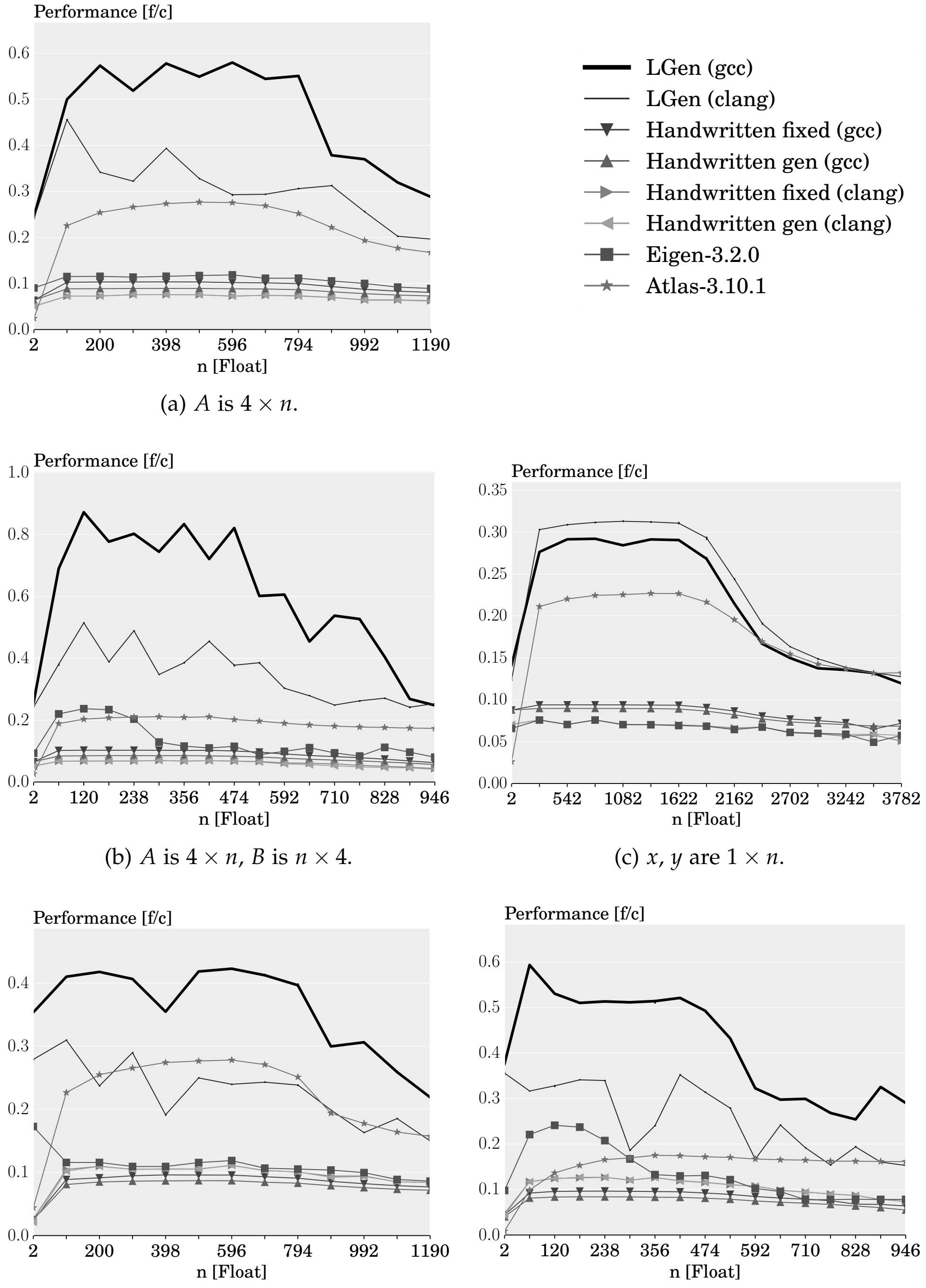
(b) A is $100 \times n$, B is $n \times n$.

Figure 5.18: $C = AB$ with a large percentage of leftovers.

5.5 ARM1176

ARM1176 implements the ARMv6 ISA, which supports no vectorization extension. Consequently, all the experiments that we conducted on this processor involve scalar code. Since all the optimizations that were introduced in this thesis apply to the process of generating vector code, the performance of LGen in these experiments is a result of the basic generation methodology that already existed before the commencement of this thesis. Optimizations like tiling, loop unrolling, loop fusion, and loop exchange have a significant effect on the quality of the generated code, while at the same time we heavily depend on the compiler for efficient instruction scheduling (which is particularly important for an in-order processor like ARM1176) and register allocation.

In Fig. 5.19 we show a representative subset of the experiments that we executed on ARM1176 involving a variety of BLACs. Except for the computation $\alpha = x^T Ay$ (Fig. 5.19(g)), LGen outperforms the competitors in all the other experiments, achieving speedups up to 4× with respect to ATLAS, which is in all cases the best of them. The drop in performance that we notice in all plots for large values of n is due to the reach of the L1 data cache limit (16 KB). Another general remark is that for all BLACs except $y = \alpha x + y$ (Fig. 5.19(c)) gcc produces considerably more efficient code than clang. Finally, the great variation of LGen's performance for different values of n is explained by the fact that for ARM1176 we have a very large amount of tiling options compared to the other three processors. Applying an inner level of ν-tiling for the other three processors considerably limits the options of outer tiling, especially in cases where the dimensions of the matrices are not divisible by ν (see restriction of tiling described in Section 2.1.2). Applying random search with a small sample size (10 in our case) for ARM1176 does not necessarily lead us to good tilings, since only a small percentage of the space of tiling choices is visited.



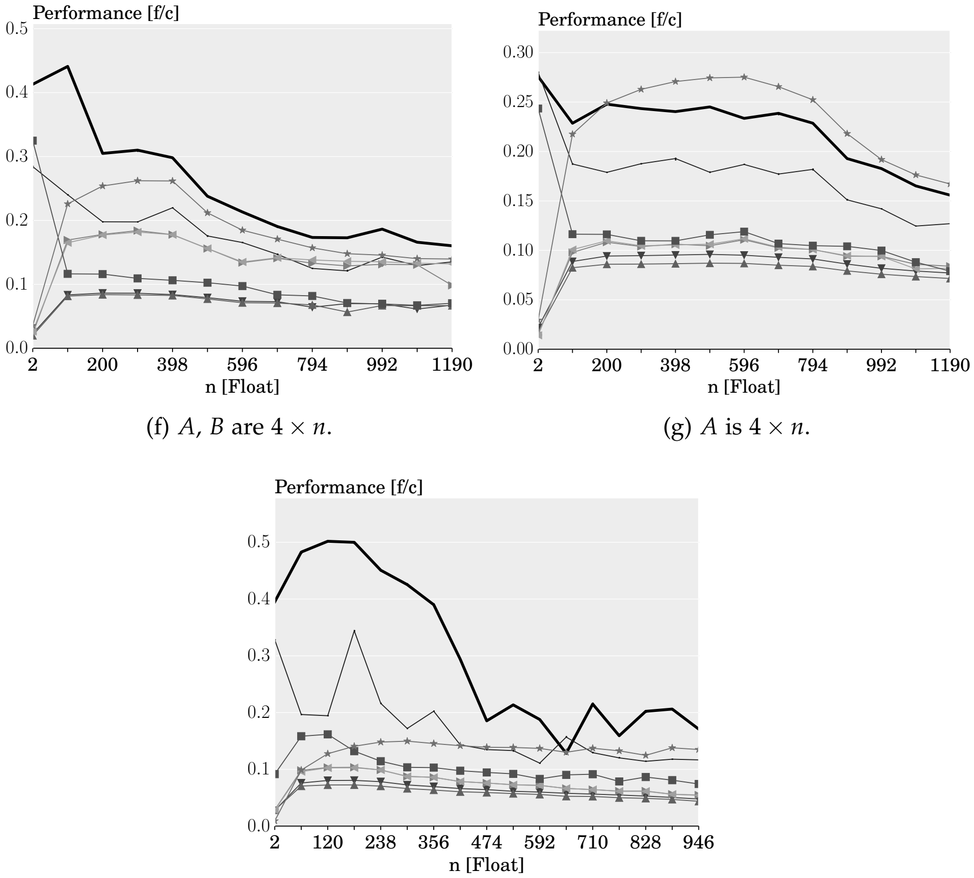
(a) A is $4 \times n$.

(b) A is $4 \times n$, B is $n \times 4$.

(c) x, y are $1 \times n$.

(d) A is $4 \times n$.

(e) A is $4 \times n$, B is $n \times 4$.



(f) A, B are $4 \times n$.

(g) A is $4 \times n$.

(h) A_0, A_1 are $n \times 4$, B is $n \times 4$.

Figure 5.19: Various BLACs. (a): $y = Ax$; (b): $C = AB$; (c): $y = \alpha x + y$; (d): $y = \alpha Ax + \beta y$; (e): $C = \alpha AB + \beta C$; (f): $y = \alpha Ax + \beta Bx$; (g): $\alpha = x^T Ay$; (h): $C = \alpha(A_0 + A_1)^T B + \beta C$.

Chapter 6

Conclusion

Contribution. The work presented in this thesis aims at the automatic generation of efficient, fixed-size, basic linear algebra code for mobile and embedded processors, starting from a high level mathematical specification of the computations. For this purpose, we extended the backend of the LGen linear algebra compiler to support code generation for four additional processors, namely Intel Atom, ARM Cortex-A8, ARM Cortex-A9, and ARM1176. To take full advantage of these processors, we added a set of optimizations to the existent LGen methodology: generic loads/stores, alignment detection, a new matrix-vector multiplication approach, and specialized codelets (ν-BLACs) for the NEON-extended ARM processors. The performance impact of such optimizations was assessed through an extensive set of experiments, which showed that the code generated by LGen performs better than commercial and non-commercial linear algebra libraries (i.e. Intel MKL and Intel IPP), linear algebra code generators (i.e. Eigen and ATLAS) and compilers (i.e. icc, gcc, and clang).

A second contribution of this thesis was the implementation of a web-based middleware, Mediator, in order to facilitate the simultaneous execution of experiments on multiple devices by multiple users, under the guarantee that only one experiment runs at a time per core per device. The features that Mediator offers include load-balancing over the cores of a device and a mechanism for retrieving performance metrics with minimal user involvement.

Limitations and Possible Extensions. An important limitation of the current version of LGen is that we can introduce leftovers in at most one level of tiling (see Section 2.1.3). As we saw in a number of experiments (e.g., Fig. 5.2(a)), this restriction has a direct impact on performance when handling matrices whose dimensions are of the form $a\nu + b$, where a is a prime number and $0 < b < \nu$. After applying a first level of ν-tiling on such a matrix, we cannot apply additional outer levels of tiling, since the choice of any tile sizes larger than one would require the introduction of further leftovers.

Since LGen typically unrolls inner loops, the absence of such an outer level of tiling potentially deteriorates the performance of the code generated by LGen. Our expectation is that the removal of this inherent limitation of LGen will smoothen the performance curves that we obtain for these experiments.

As we noticed in the experimental results for ARM1176 (see Section 5.5), applying random search over the tile sizes is not optimal when the search space is large compared to the sample size. Especially for processors that do not support any vectorization extension (like ARM1176), the tile sizes that may possibly be chosen are many and it is highly probable that random search will not visit the ones that give the best performance. In order to overcome this issue, LGen could possibly make use of heuristics in order to prune the search space and/or direct the search towards better choices.

An additional limitation of the current version of the LGen generation approach is that it tiles under the assumption that all data reside in the L1 cache. In this respect, it applies only two levels of tiling, with the inner one targeting vectorization and the outer one mainly targeting the reuse of the contents of registers. This methodology could be extended for larger matrices by adding more levels of tiling, with sizes that are appropriate for the higher cache levels. Such an optimization would also require the introduction of intermediate temporary arrays at the boundaries of the different levels of cache, which could be done by means of the gather and scatter matrices.

The current approach of LGen for handling memory alignment is the following: Code is generated using only unaligned instructions, and only when the code has taken its final form the alignment detection mechanism is applied and replaces unaligned instructions with their aligned equivalents. The side-effects of such an approach are visible in the experiments presented in Fig. 5.9 (see also related discussion in Section 5.2.4). A possible way to face this limitation would be the introduction of a loop peeling optimization, similar to the one used by Eigen, aiming at exposing a larger amount of aligned accesses. Taking arbitrary alignment handling one step further, we could additionally use a technique similar to the one described in [14], in order to combine the instructions of consecutive loop iterations and replace unaligned accesses with aligned ones together with a minimal number of shuffle operations.

Due to the importance of energy efficiency for embedded processors, another possible extension of LGen could be the introduction of energy-related metrics in the autotuning feedback loop. Such an endeavour is not trivial, since it would require the use of additional hardware for measuring these metrics. Additionally, a methodology similar to the one used in [6] would be required for isolating the processor power from the power consumed by the remaining parts of the board (e.g., memory).

Appendix A

Mediator API

Property Name	Property Value	Required	Default Value	Description
Top-level properties				
apiVersion	string	No	`"1.0"`	The API version number according to which the request is expected to be handled.
async	string	No	`"True"`	If `"True"` then the request will be processed asynchronously and the client will have to poll Mediator to get the job results. Otherwise, the request will be processed synchronously (the HTTP connection will be kept open until the results are readily available) and Mediator's response will have the format specified in Table A.2.
experiments	array: Experiment	Yes	-	The list of experiments to be executed.
Experiment properties				
compileCommands	array: string	No	`[]`	The list of compilation commands. They will be executed in a single session in the *experimentRootFolder* folder of the target device.
device	Device	Yes	-	The target device that corresponds to this experiment.
execCommands	array: string	No	`[]`	The list of execution commands. They will be executed each one in a separate session after the successful execution of the compilation commands.
files	array: File	No	`[]`	The list of files that have to be copied to the target device.
measureArch	string	No	-	The microarchitecture code of the target device, in case the client intends to use the Mediator's performance measuring modules from within the experiment code. If this property is specified, Mediator will copy the needed `"measure.h"` header file in *experimentRootFolder*. For more details about this functionality, see section 4.5. Presently, the possible values for this property are: `"x86"`, `"cortex_a8"`, `"cortex_a9"`, `and` `"arm1176"`.
measureMetrics	array: string	No	-	The list of performance metrics that Mediator should measure and return to the client (see section 4.5 for more details). *measureArch* should also be defined containing a valid value.
outputFiles	array: string	No	`[]`	The list of files (full relative path with respect to *experimentRootFolder* required) the contents of which have to be sent back to the client after the successful execution of the experiments.

repetitions	number	No	`1`	The number of times that each command in *execCommands* will be executed. The contents of *outputFiles* will be collected for each repetition separately and finally returned to the client.
Device properties				
affinity	array: number	No	`[0]`	A list of cpu affinities that could be used for the execution of the experiments on the target device. In case more than one affinities are specified, Mediator selects the affinity that has the least number of pending experiments associated to it (load balancing). For more details, see section 4.3.
experimentRootFolder	string	No	`"/home/<username>/ performance"`	The location in the filesystem of the target device where the experiment will take place.
hostname	string	Yes	-	The hostname of the target device.
os	string	No	`"LINUX"`	The operating system of the target device. Presently the only supported value for this property is `"LINUX"`.
password	string	No	-	The password related to *username* that will be used by Mediator to establish the SSH connection with the target device. If neither *rsaKey* nor *password* is defined, Mediator will try to connect to the target device using its own private RSA key. If both *rsaKey* and *password* are defined, Mediator will try to connect using *rsaKey*.
port	number	No	22	The port of the target device that will be used for the SSH connection.
rsaKey	string	No	-	The private RSA key of the client, that will be used by Mediator to establish the SSH connection with the target device. If neither *rsaKey* nor *password* is defined, Mediator will try to connect to the target device using its own private RSA key. If both *rsaKey* and *password* are defined, Mediator will try to connect using *rsaKey*.
rsaKeyPass	string	No	`""`	The passphrase that is needed to decrypt the RSA key.
type	string	No	`"SSH_DEVICE"`	The type of the target device (only the value `"SSH_DEVICE"` is presently supported).
username	string	No	`spirals`	The username that Mediator will use to connect to the target device through SSH.
File properties				
binary	string	No	`"False"`	Signifies whether the file is binary or not.
contents	string	Yes	-	The contents of the file to be created on the target device.
encoding	string	No	`"utf8"`	The encoding of *contents*. Presently the supported encodings are `"utf8"` for text files (e.g., source code) and `"base64"` for binary files (e.g., executables).
executable	string	No	`"False"`	Signifies whether the file is executable or not.
path	string	Yes	-	The path of the file that is going to be created on the target device, relative to *experimentRootFolder* (filename included).

Table A.1: Format of a new job request sent to Mediator.

Property Name	Property Value	Description
Top-level properties		
apiVersion	string	The API version number according to which the corresponding new job request was processed.
data	array: ExperimentResults	The list of experiment results. Each element of the list corresponds to a specific experiment. The order of experiments in the corresponding new job request is preserved here. The existence of this property signifies that there was no error related to the request as a whole (e.g., parsing errors due to a badly formatted request). The existence of this property is mutually exclusive with the existence of the property *error*.
error	Error	The existence of this property signifies that there was an error processing the request as a whole. The existence of this property is mutually exclusive with the existence of the property *data*.
ExperimentResults properties		
deviceHostname	string	The value of this property is the same as the value of the homonymous property in the corresponding new job request.
error	Error	The existence of this property signifies that there was some error during the execution of this experiment. More information about the error can be found by inspecting the values of the subproperties of this property. The existence of this property is mutually exclusive with the existence of the property *results*.
output	string	The output of compilation and execution of this experiment (useful for debugging in case of errors).
repetitions	number	The value of this property is the same as the value of the homonymous property in the corresponding new job request.
results	array: execCommandResults	The list of results for each execution command specified for this experiment. The existence of this property implies that there was no error during the execution of this experiment. The existence of this property is mutually exclusive with the existence of the property *error*.
execCommandResults properties		
execCommand	string	The command that this execCommandResults entity is associated with.
execResults	array: FileResults	The list of output file contents for this experiment and execution command. Each element in this list corresponds to a single repetition.
exitCodes	number	The list of exit codes for this experiment and execution command. Each element of this list corresponds to a single repetition.
FileResults properties		
fileContents	array: string	The contents of *outputFile* for each repetition of this execution command of this experiment.
outputFile	string	The name of the output file.
Error properties		
code	number	The error code. A list of all error codes can be found in Table A.5.
message	string	A textual description of the error, possibly providing information about how to solve the related issue.
reason	string	The name of the error. A list of all error reasons can be found in Table A.5.

Table A.2: Format of a job results response received from Mediator.

Property Name	Property Value	Required	Default Value	Description
Top-level properties				
apiVersion	string	No	`"1.0"`	The API version number according to which the request is expected to be handled.
jobID	string	Yes	-	The id of the job, whose results the client is querying for.

Table A.3: Format of a job results request sent to Mediator.

Property Name	Property Value	Description
Top-level properties		
apiVersion	string	The API version number.
data	array: ExperimentResults	Same as in Table A.2. This property exists only if the value of *jobState* is `FINISHED`.
jobID	string	The unique identifier of the job. Currently it is a long hexadecimal number that Mediator generates after receiving an asynchronous new job request, by applying the SHA-1 hash function on a randomly generated 256-bit number.
jobState	string	The current state of the job. Possible values for this property are: `SUBMITTED`, `NOT_FOUND`, `PENDING`, and `FINISHED`, each with the obvious semantics.

Table A.4: Format of job status response received from Mediator.

Code	Reason	Description
400	BadRequest	Badly formatted request.
401	SSHAuthenticationError	Invalid SSH credentials.
405	InstructionExecutionError	The execution of an instruction through SSH produced an error (see accompanying message for more details).
406	SSHError	General SSH Error (see accompanying message for more details).
408	InstructionTimeoutError	The execution of an instruction over SSH took a very long time.
500	InternalError	A server error that the user shouldn't know anything specific about.

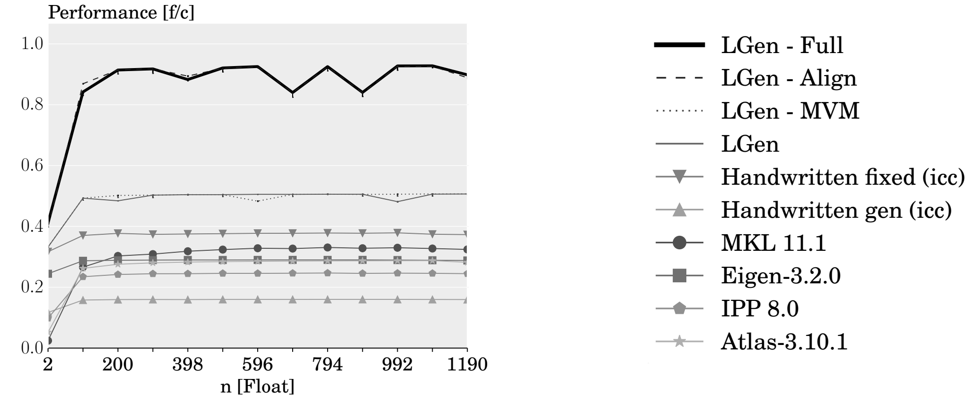
Table A.5: List of Mediator's API errors.

Appendix B

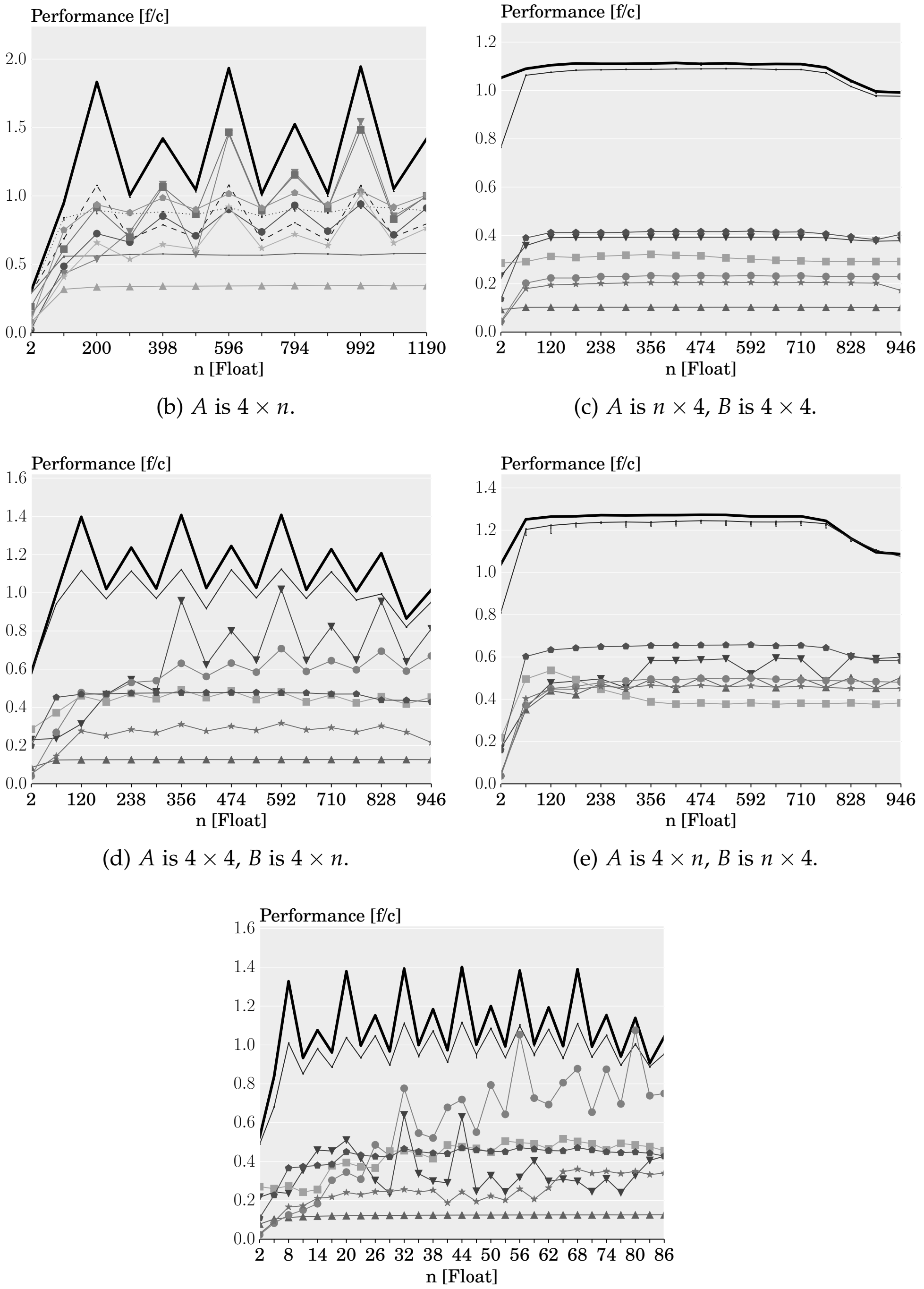
Complete Set of Experimental Results

In the following sections we present the results of the complete set of experiments that we executed on each of the four processors investigated for the purpose of this thesis. Some of them have already been presented in chapter 5, but they are also included here for the sake of completeness.

B.1 Intel Atom



(a) A is $n \times 4$.



(b) A is $4 \times n$.

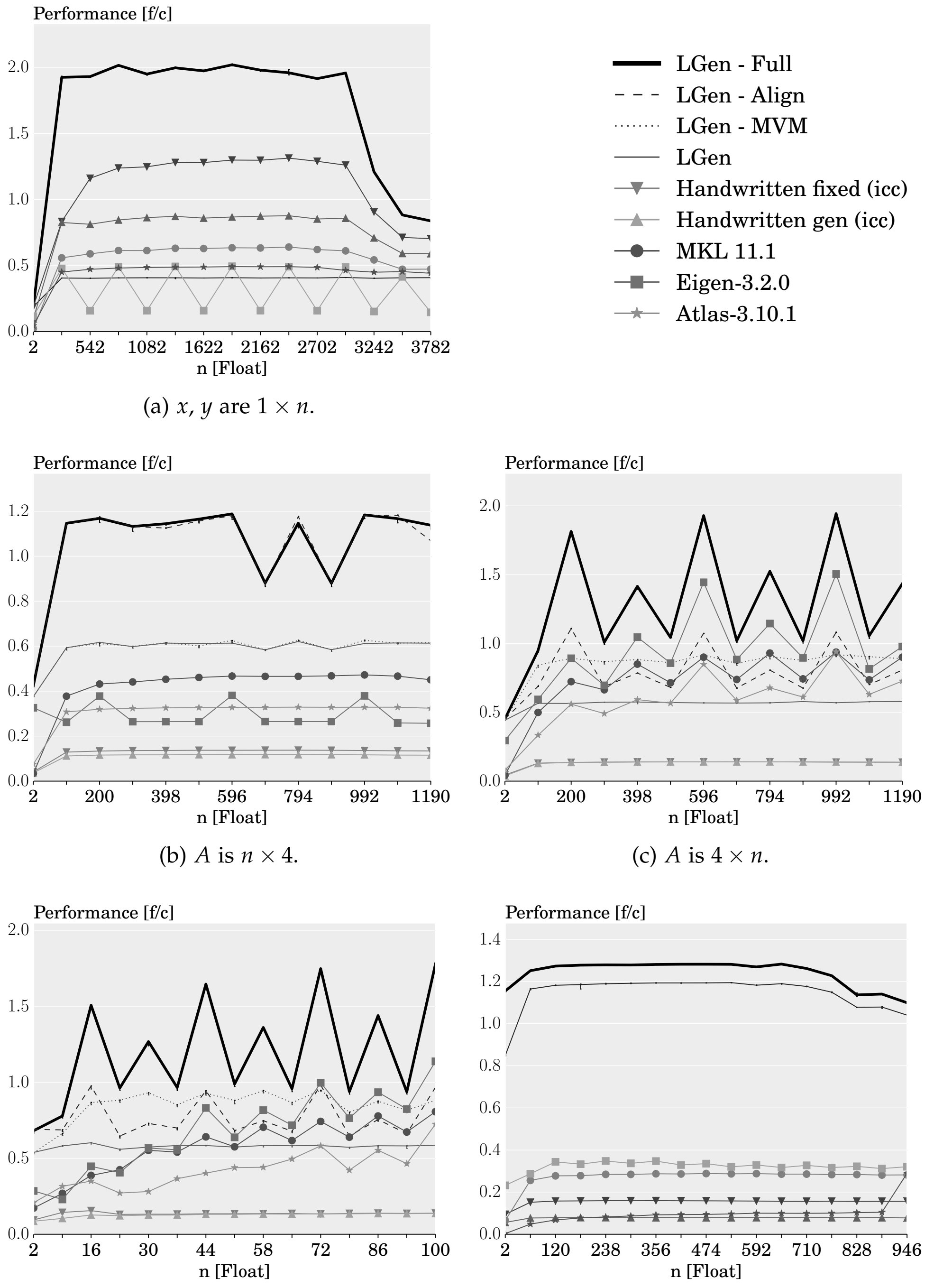
(c) A is $n \times 4$, B is 4×4.

(d) A is 4×4, B is $4 \times n$.

(e) A is $4 \times n$, B is $n \times 4$.

(f) A is $n \times 4$, B is $4 \times n$.

Figure B.1: Simple BLACs. (a)-(b): $y = Ax$; (c)-(f): $C = AB$.



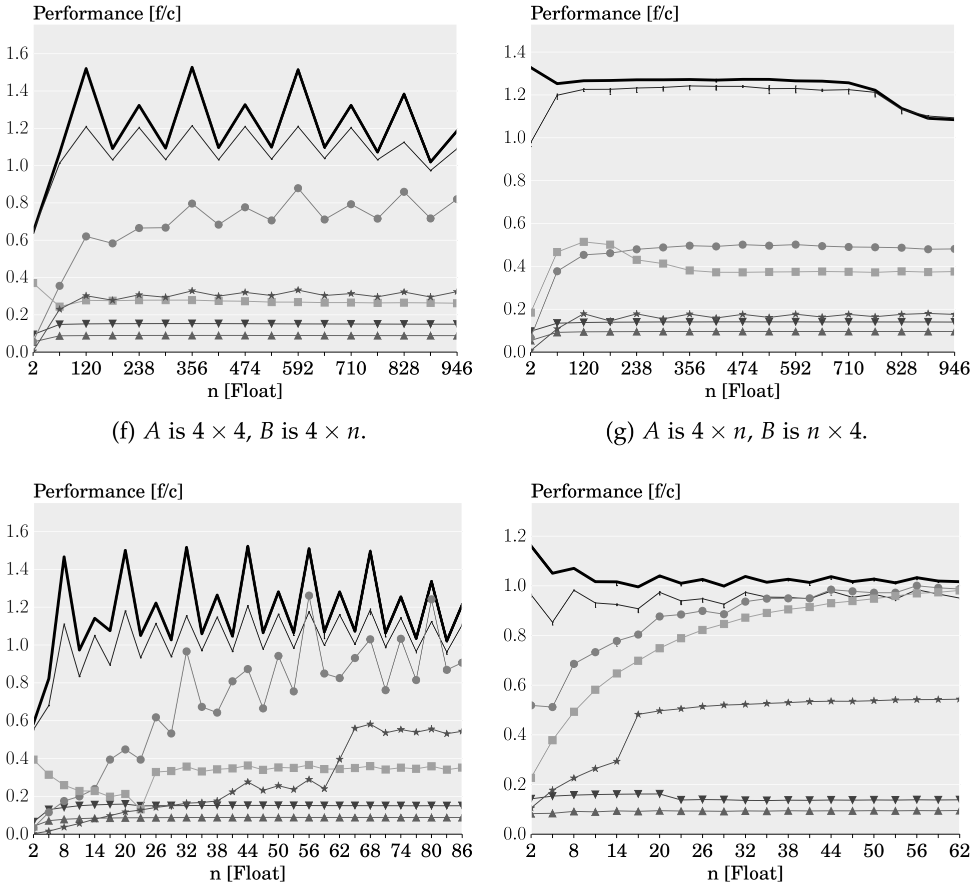
(a) x, y are $1 \times n$.

(b) A is $n \times 4$.

(c) A is $4 \times n$.

(d) A is $30 \times n$.

(e) A is $n \times 4$, B is 4×4.



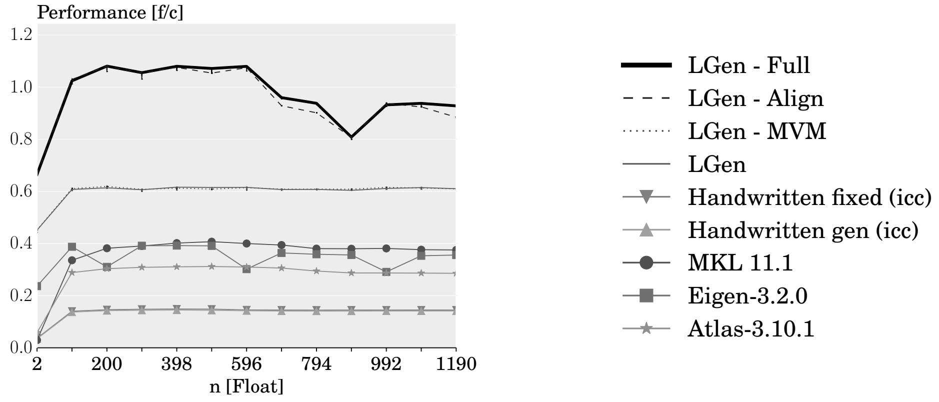
(f) A is 4×4, B is $4 \times n$.

(g) A is $4 \times n$, B is $n \times 4$.

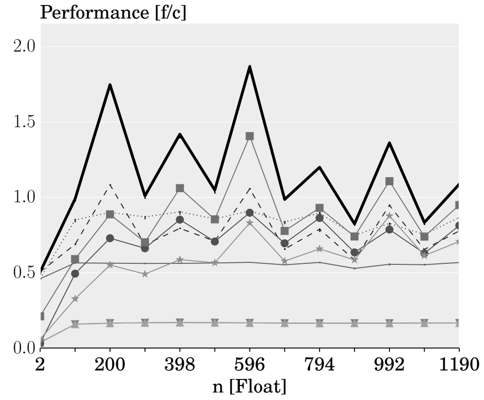
(h) A is $n \times 4$, B is $4 \times n$.

(i) A is $30 \times n$, B is $n \times 30$.

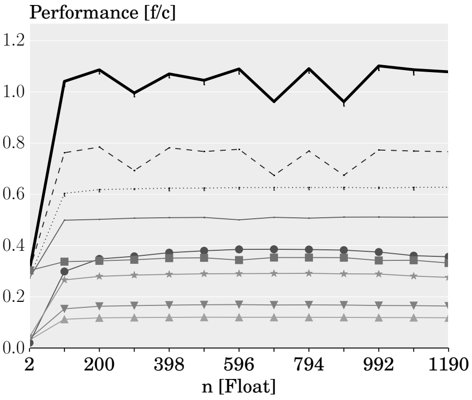
Figure B.2: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b)-(d): $y = \alpha Ax + \beta y$; (e)-(h): $C = \alpha AB + \beta C$.



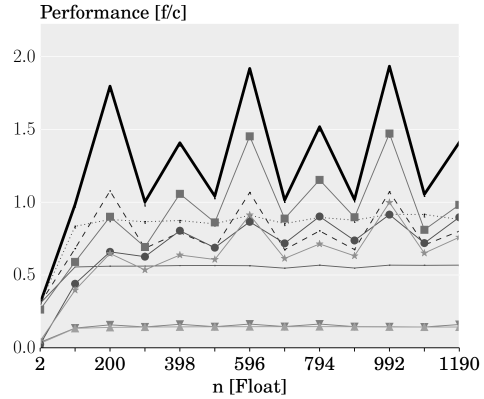
(a) A, B are $n \times 4$.



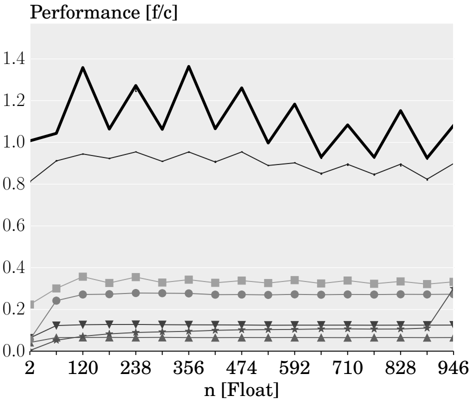
(b) A, B are $4 \times n$.



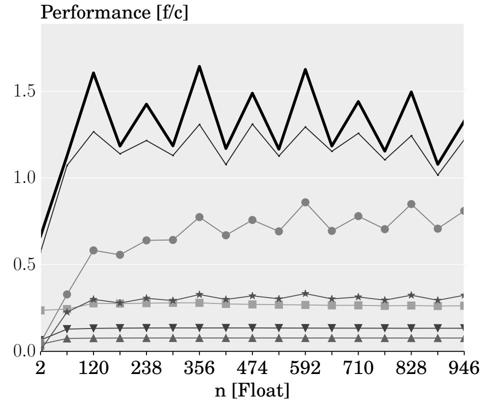
(c) A is $n \times 4$.



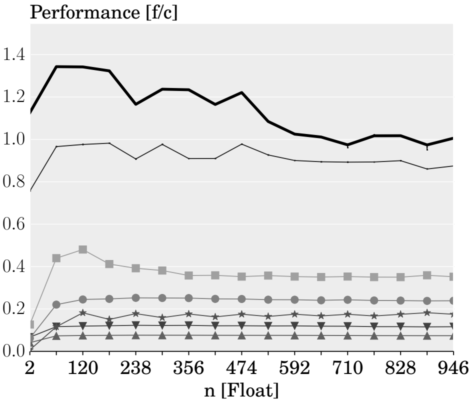
(d) A is $4 \times n$.



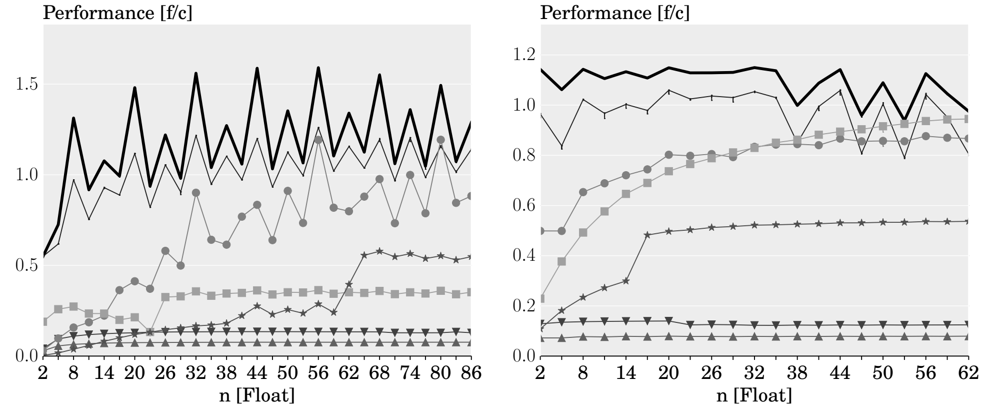
(e) A_0, A_1 are $4 \times n$, B is 4×4.



(f) A_0, A_1 are 4×4, B is $4 \times n$.



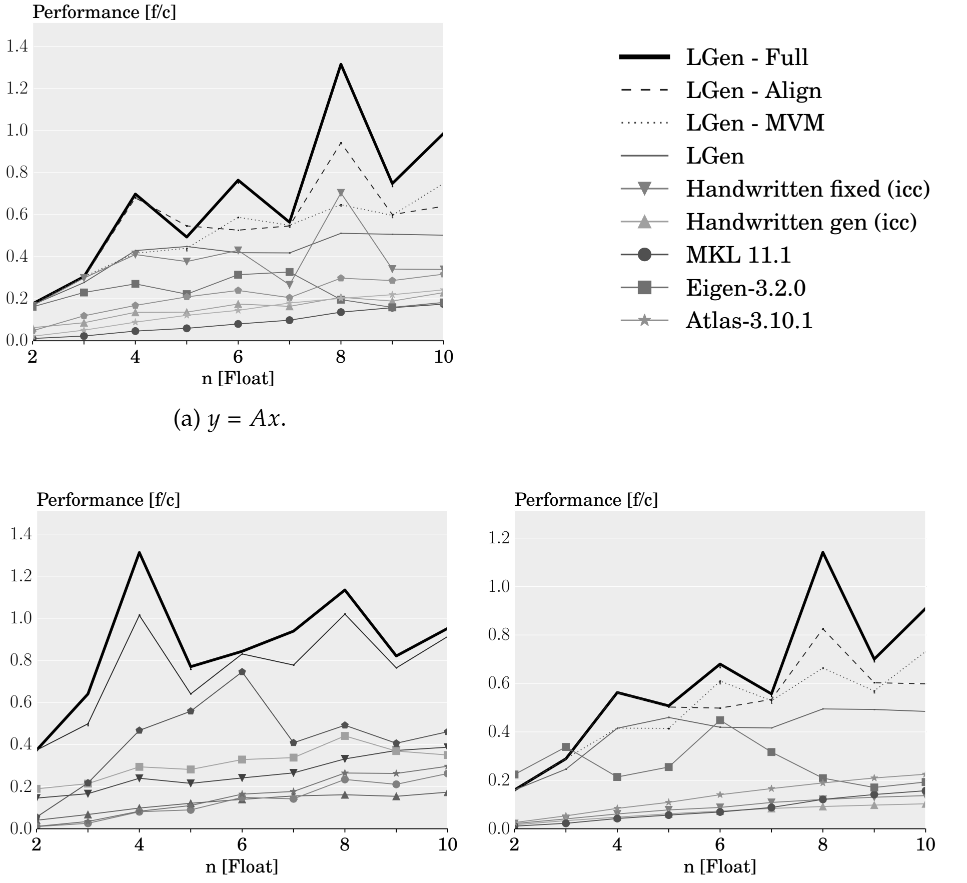
(g) A_0, A_1 are $n \times 4$, B is $n \times 4$.



(h) A_0, A_1 are $4 \times n$, B is $4 \times n$.

(i) A_0, A_1 are $n \times 30$, B is $n \times 30$.

Figure B.3: BLACs that require more than one BLAS call. (a)-(b): $y = \alpha Ax + \beta Bx$; (c)-(d): $\alpha = x^T Ay$; (e)-(i): $C = \alpha(A_0 + A_1)^T B + \beta C$.



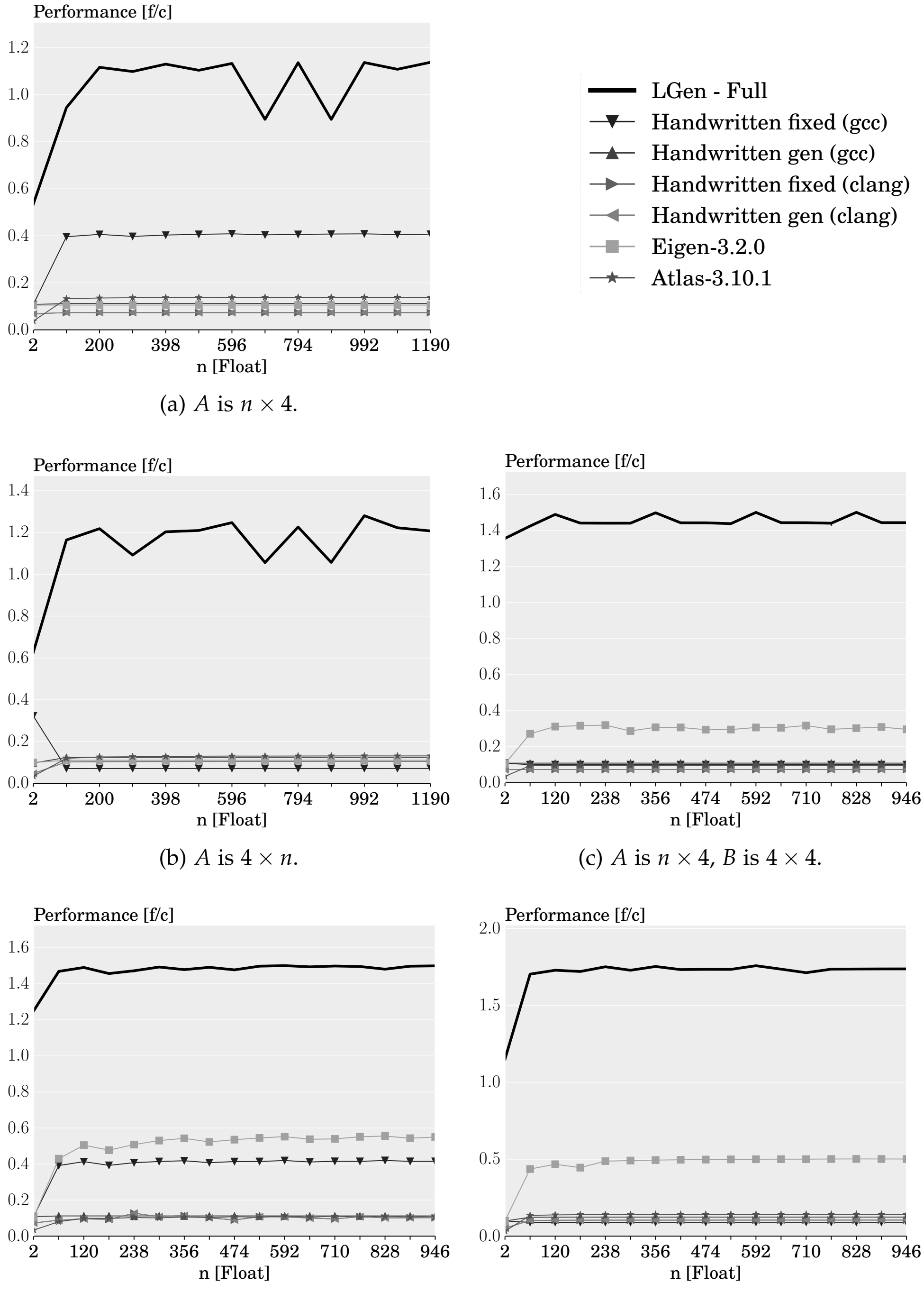
(a) $y = Ax$.

(b) $C = AB$.

(c) $\alpha = x^T Ay$.

Figure B.4: Micro-BLACs. All matrices have size $n \times n$.

B.2 ARM Cortex-A8



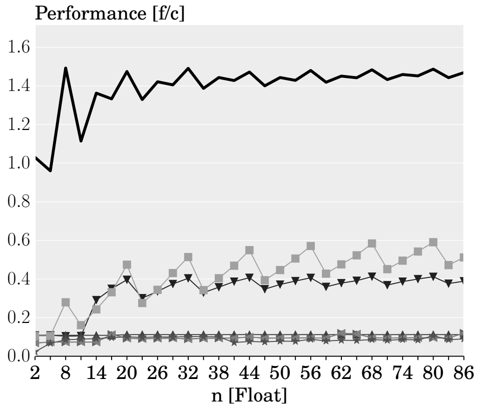
(a) A is $n \times 4$.

(b) A is $4 \times n$.

(c) A is $n \times 4$, B is 4×4.

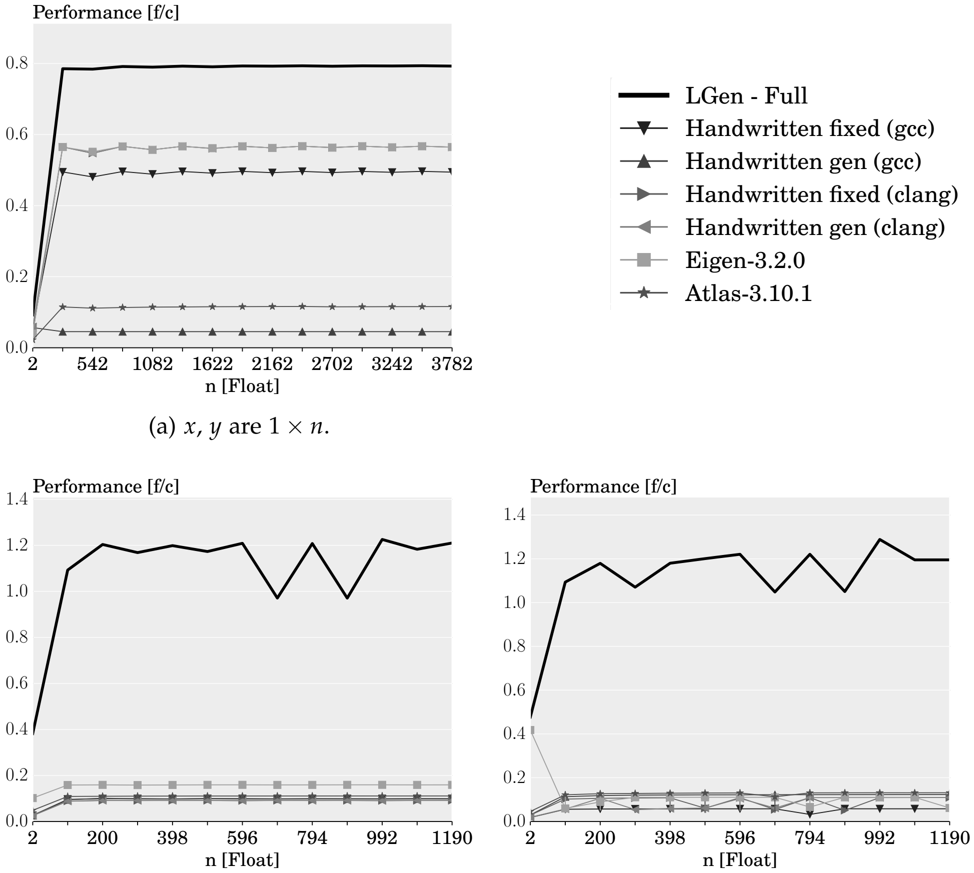
(d) A is 4×4, B is $4 \times n$.

(e) A is $4 \times n$, B is $n \times 4$.



(f) A is $n \times 4$, B is $4 \times n$.

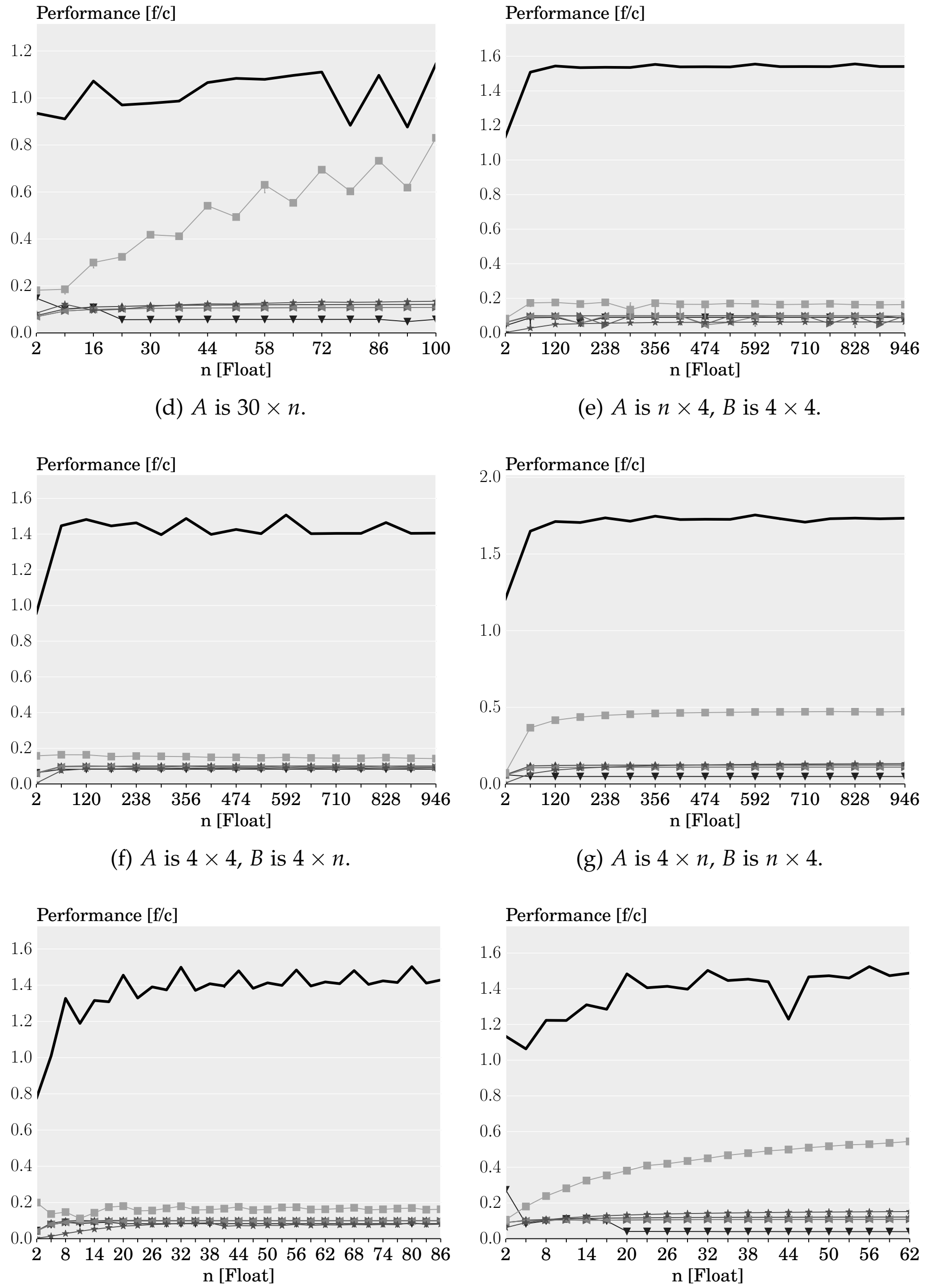
Figure B.5: Simple BLACs. (a)-(b): $y = Ax$; (c)-(f): $C = AB$.



(a) x, y are $1 \times n$.

(b) A is $n \times 4$.

(c) A is $4 \times n$.



(d) A is $30 \times n$.

(e) A is $n \times 4$, B is 4×4.

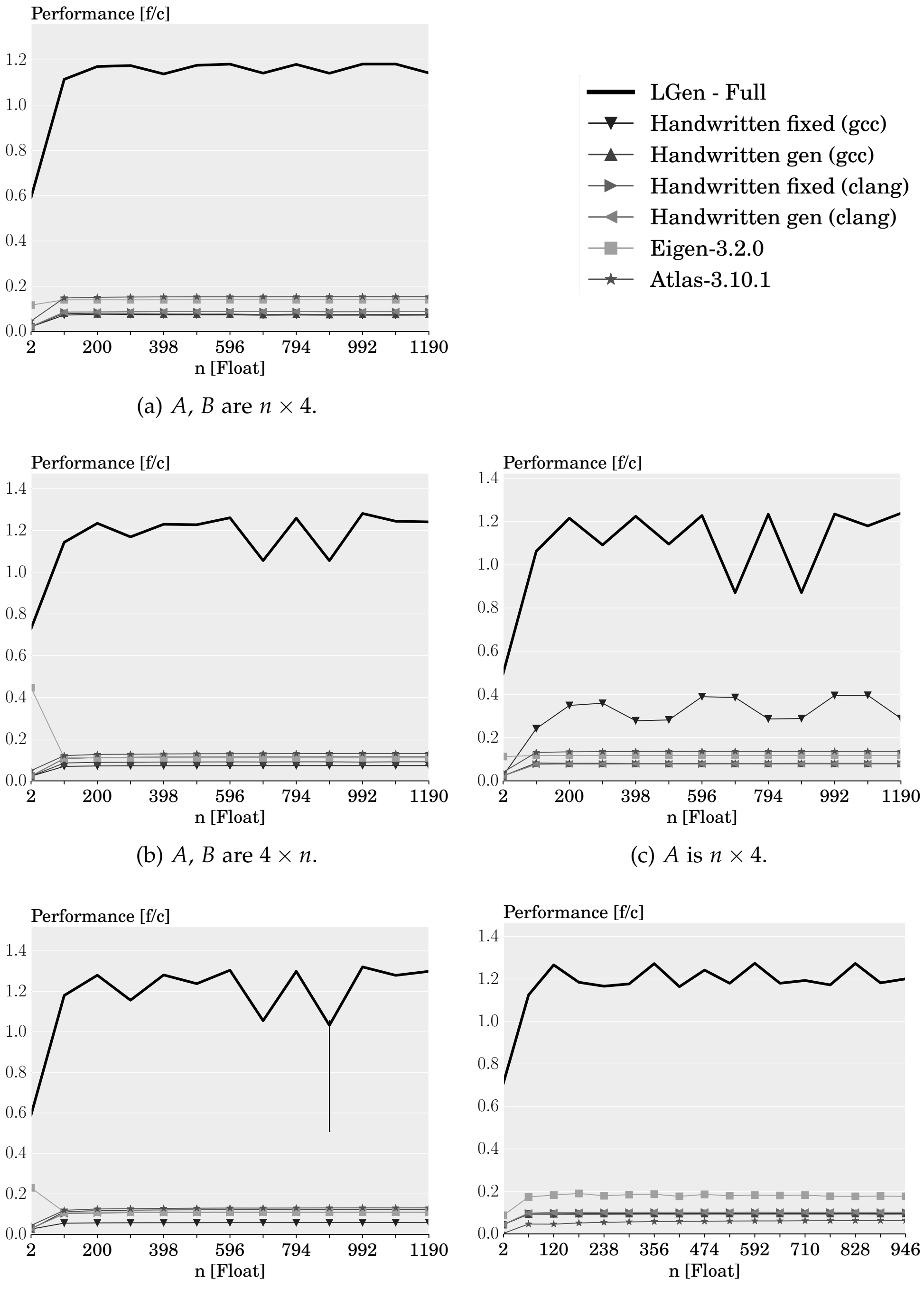
(f) A is 4×4, B is $4 \times n$.

(g) A is $4 \times n$, B is $n \times 4$.

(h) A is $n \times 4$, B is $4 \times n$.

(i) A is $30 \times n$, B is $n \times 30$.

Figure B.6: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b)-(d): $y = \alpha Ax + \beta y$; (e)-(h): $C = \alpha AB + \beta C$.



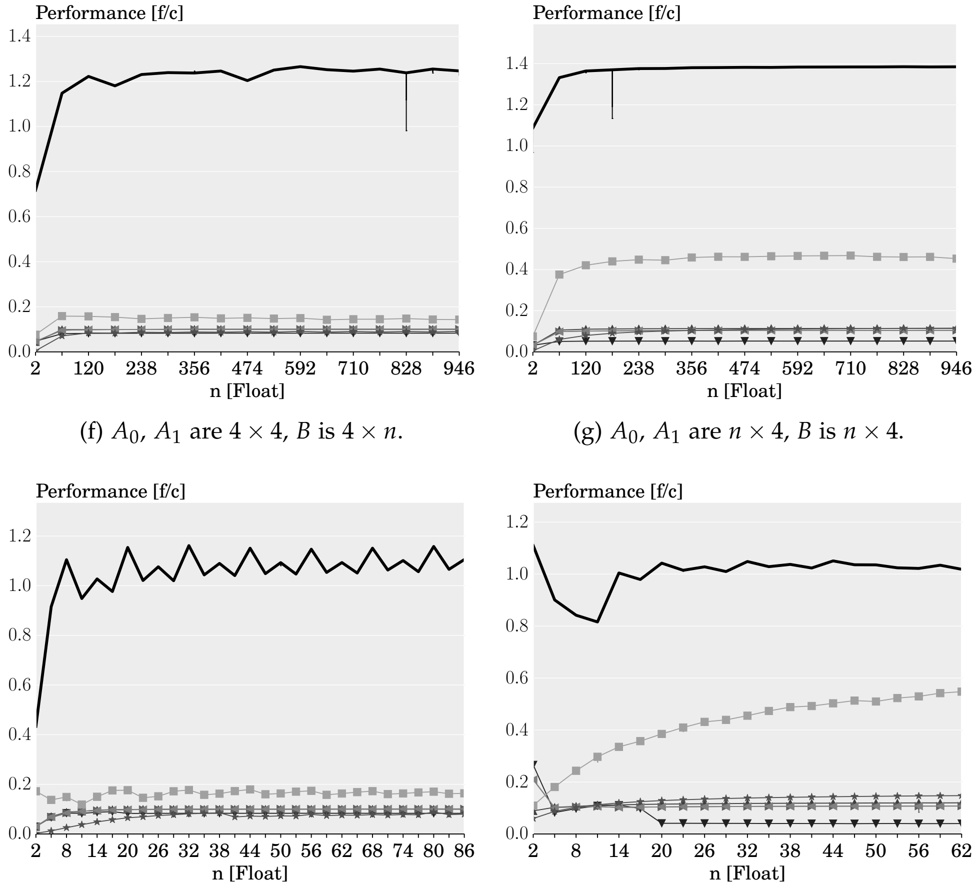
(a) A, B are $n \times 4$.

(b) A, B are $4 \times n$.

(c) A is $n \times 4$.

(d) A is $4 \times n$.

(e) A_0, A_1 are $4 \times n$, B is 4×4.



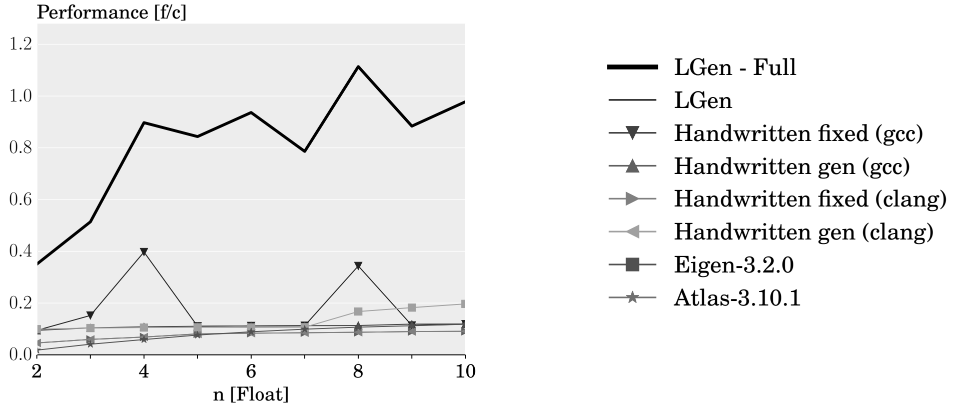
(f) A_0, A_1 are 4×4, B is $4 \times n$.

(g) A_0, A_1 are $n \times 4$, B is $n \times 4$.

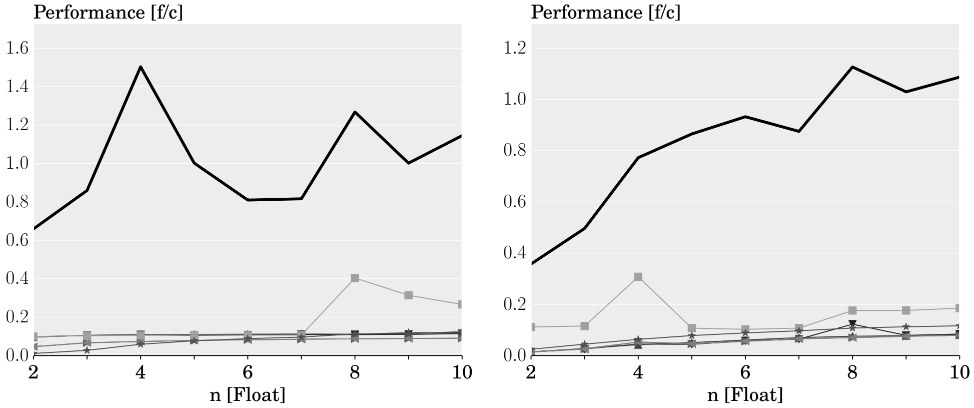
(h) A_0, A_1 are $4 \times n$, B is $4 \times n$.

(i) A_0, A_1 are $n \times 30$, B is $n \times 30$.

Figure B.7: BLACs that require more than one BLAS call. (a)-(b): $y = \alpha Ax + \beta Bx$; (c)-(d): $\alpha = x^T Ay$; (e)-(i): $C = \alpha(A_0 + A_1)^T B + \beta C$.



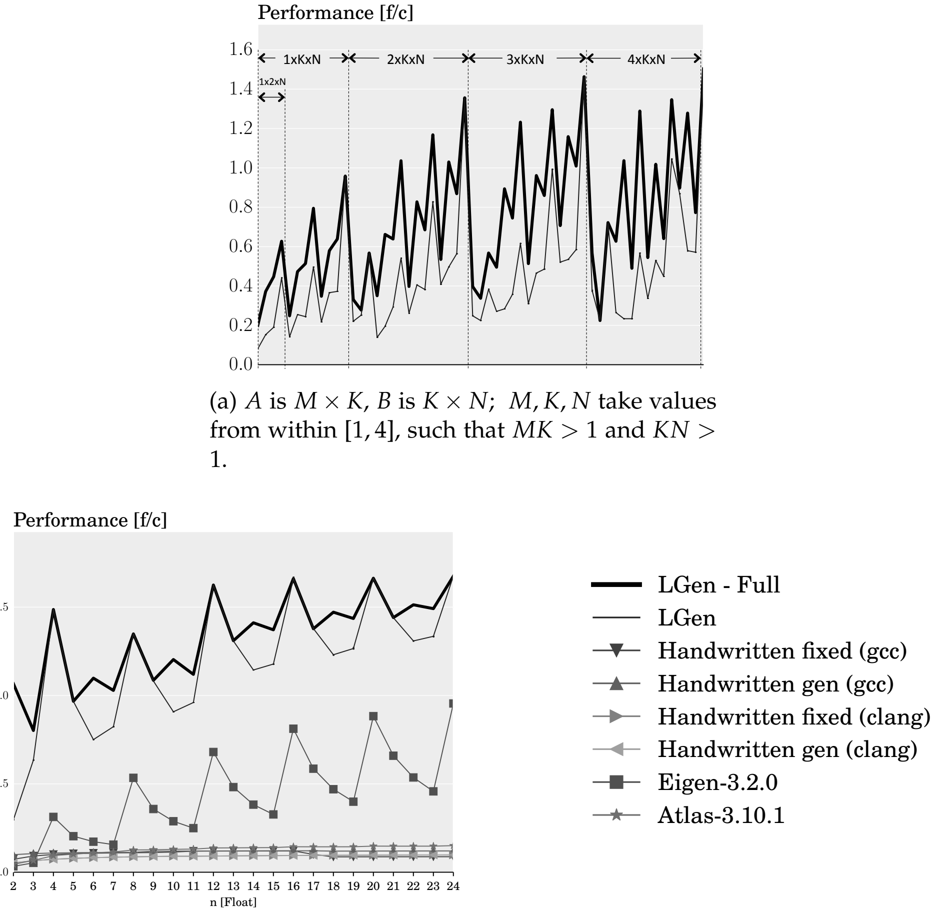
(a) $y = Ax$.



(b) $C = AB$.

(c) $\alpha = x^T Ay$.

Figure B.8: Micro-BLACs. All matrices have size $n \times n$.

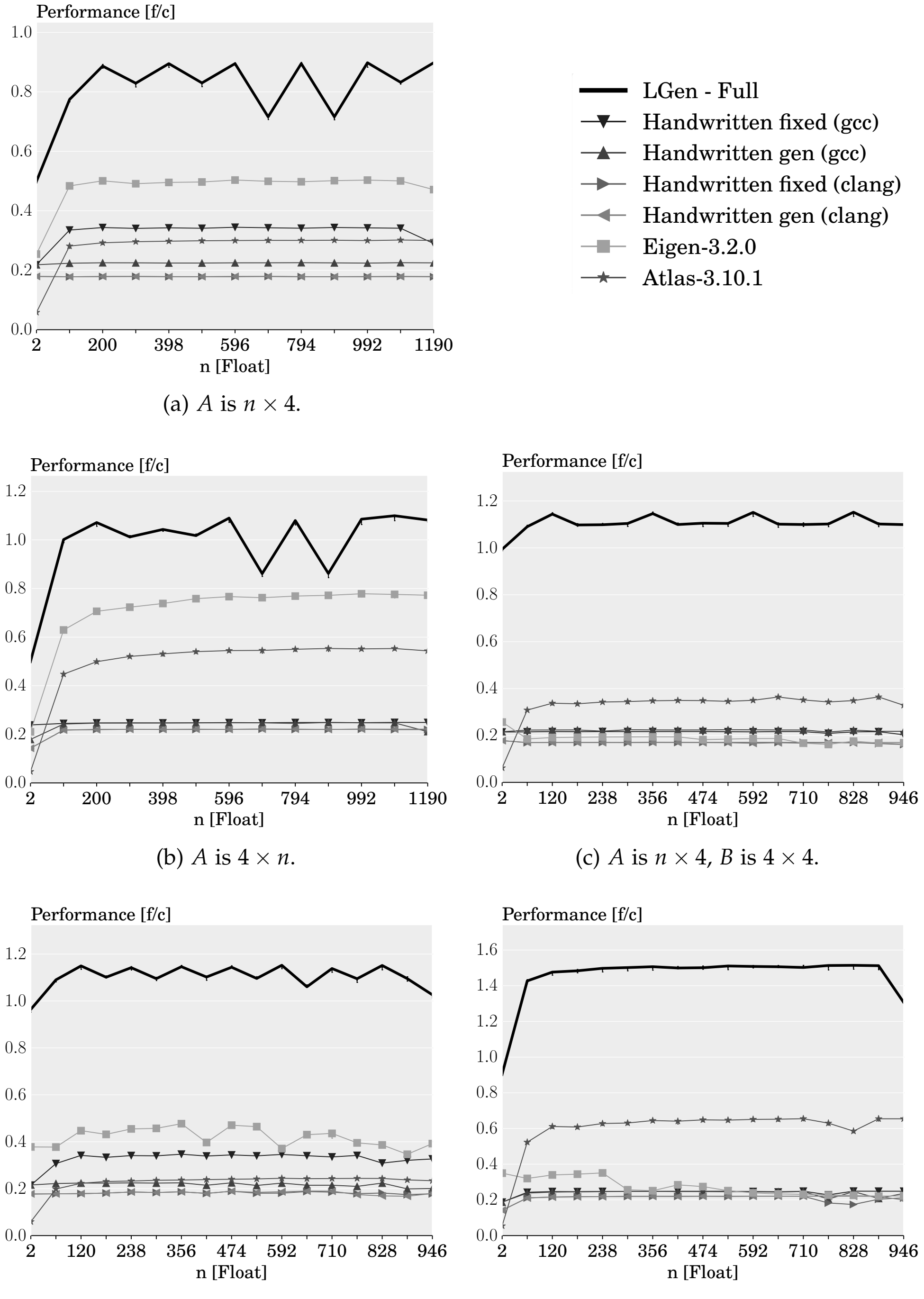


(a) A is $M \times K$, B is $K \times N$; M, K, N take values from within $[1, 4]$, such that $MK > 1$ and $KN > 1$.

(b) A is $100 \times n$, B is $n \times n$.

Figure B.9: $C = AB$ with a large percentage of leftovers.

B.3 ARM Cortex-A9



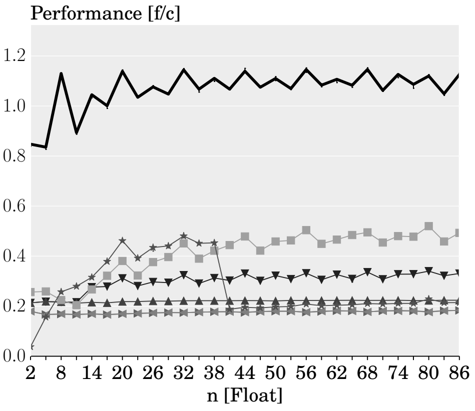
(a) A is $n \times 4$.

(b) A is $4 \times n$.

(c) A is $n \times 4$, B is 4×4.

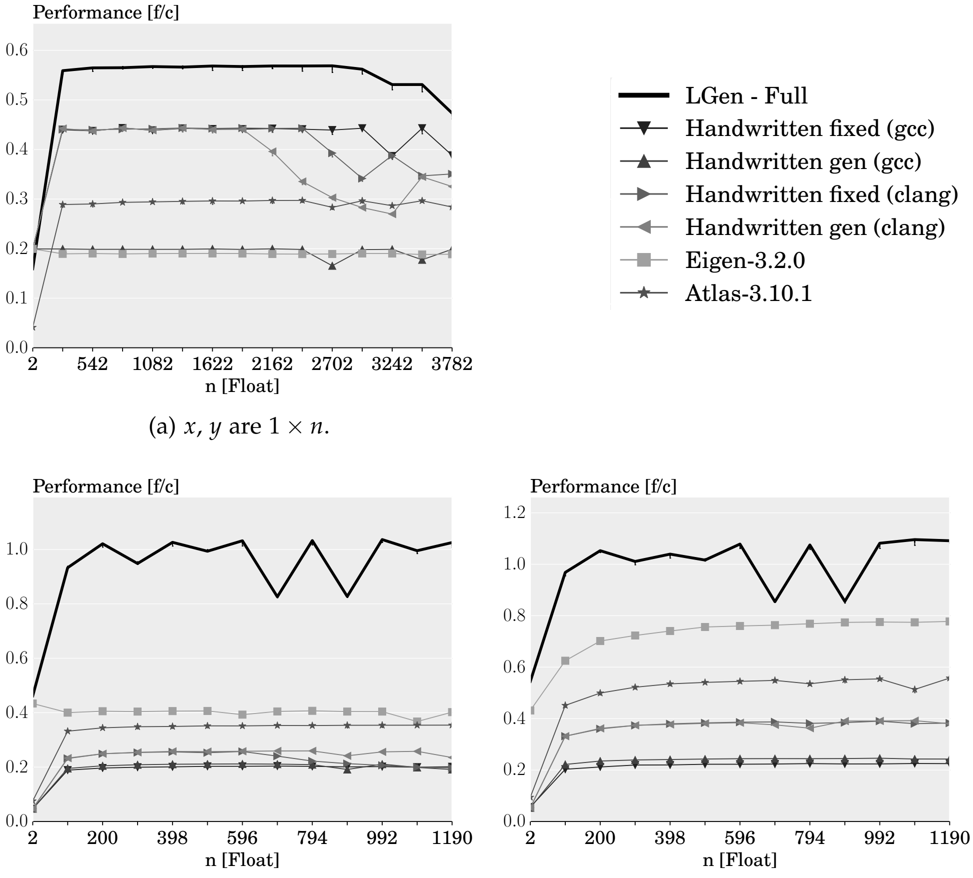
(d) A is 4×4, B is $4 \times n$.

(e) A is $4 \times n$, B is $n \times 4$.



(f) A is $n \times 4$, B is $4 \times n$.

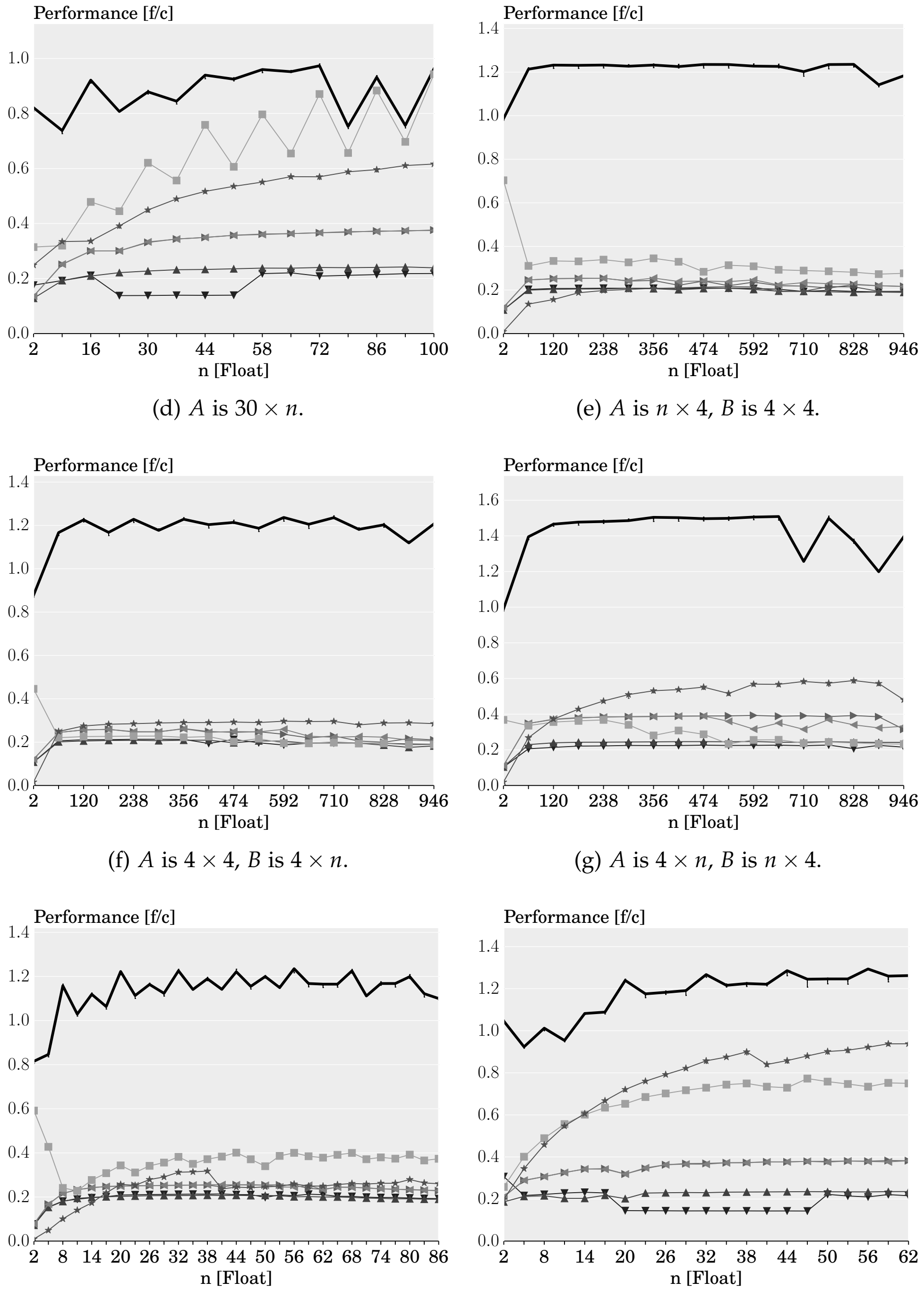
Figure B.10: Simple BLACs. (a)-(b): $y = Ax$; (c)-(f): $C = AB$.



(a) x, y are $1 \times n$.

(b) A is $n \times 4$.

(c) A is $4 \times n$.



(d) A is $30 \times n$.

(e) A is $n \times 4$, B is 4×4.

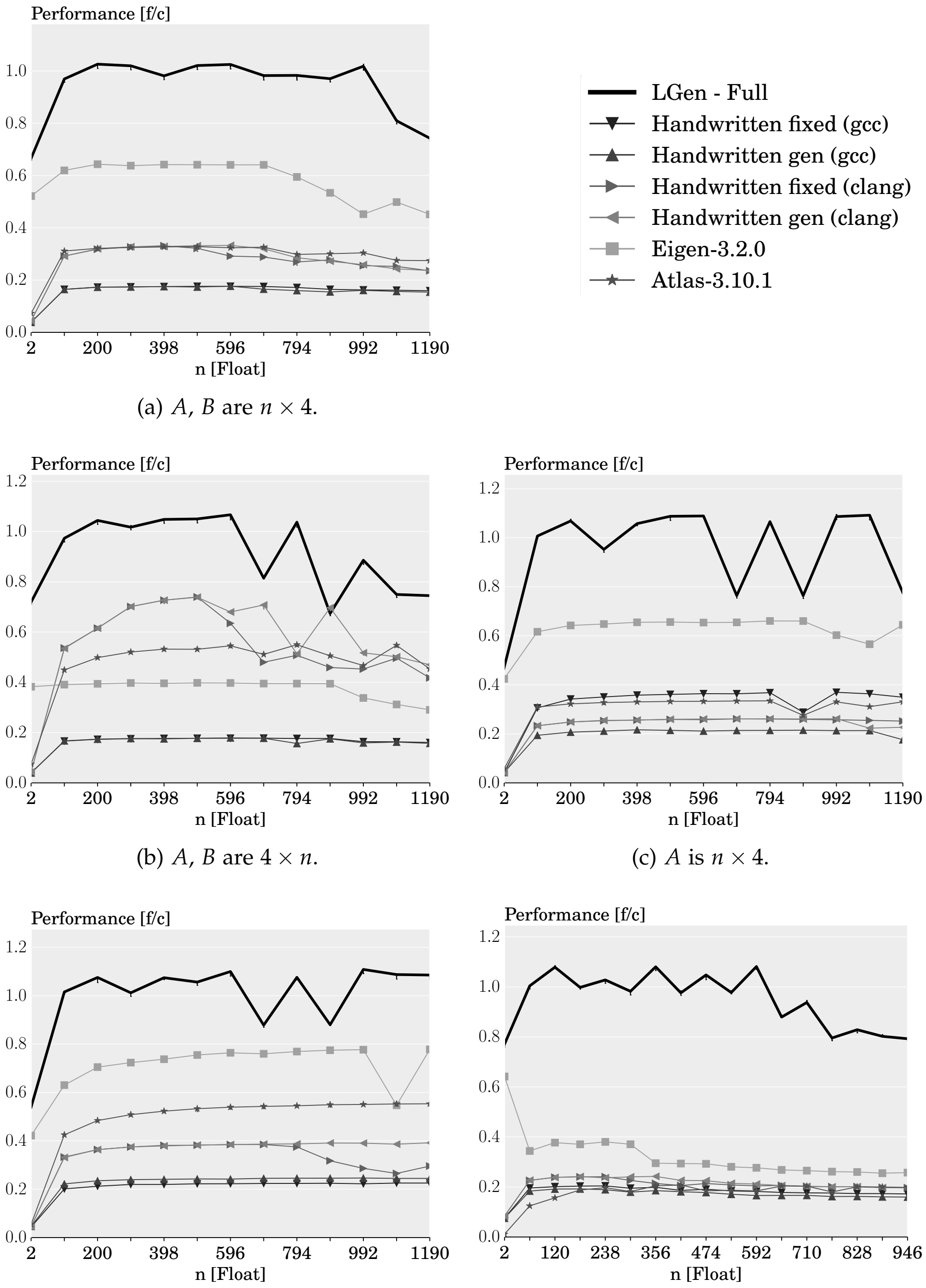
(f) A is 4×4, B is $4 \times n$.

(g) A is $4 \times n$, B is $n \times 4$.

(h) A is $n \times 4$, B is $4 \times n$.

(i) A is $30 \times n$, B is $n \times 30$.

Figure B.11: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b)-(d): $y = \alpha Ax + \beta y$; (e)-(h): $C = \alpha AB + \beta C$.



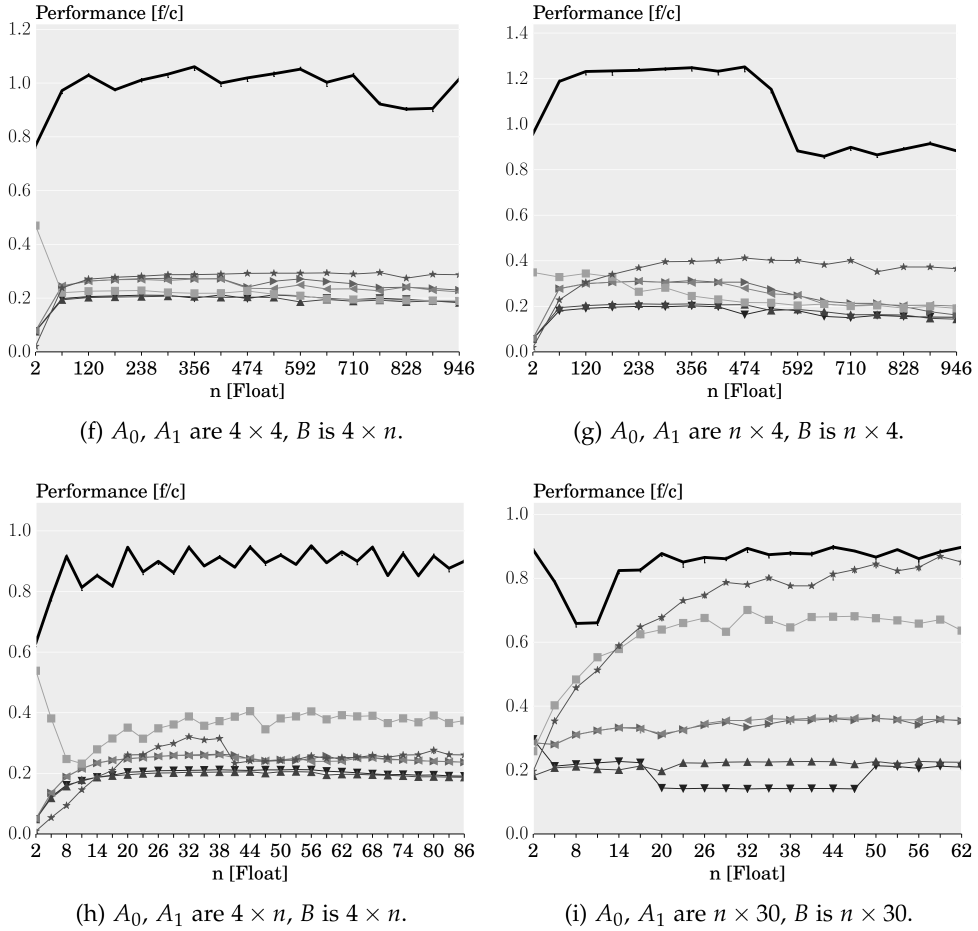
(a) A, B are $n \times 4$.

(b) A, B are $4 \times n$.

(c) A is $n \times 4$.

(d) A is $4 \times n$.

(e) A_0, A_1 are $4 \times n$, B is 4×4.



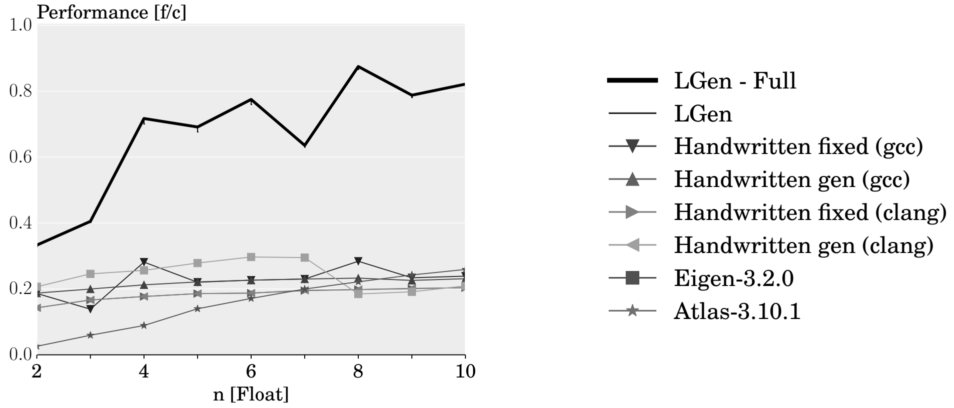
(f) A_0, A_1 are 4×4, B is $4 \times n$.

(g) A_0, A_1 are $n \times 4$, B is $n \times 4$.

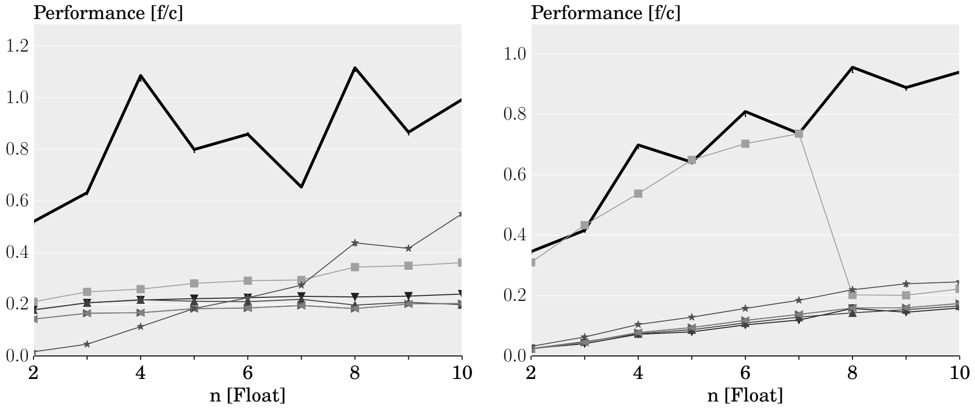
(h) A_0, A_1 are $4 \times n$, B is $4 \times n$.

(i) A_0, A_1 are $n \times 30$, B is $n \times 30$.

Figure B.12: BLACs that require more than one BLAS call. (a)-(b): $y = \alpha Ax + \beta Bx$; (c)-(d): $\alpha = x^T Ay$; (e)-(i): $C = \alpha(A_0 + A_1)^T B + \beta C$.



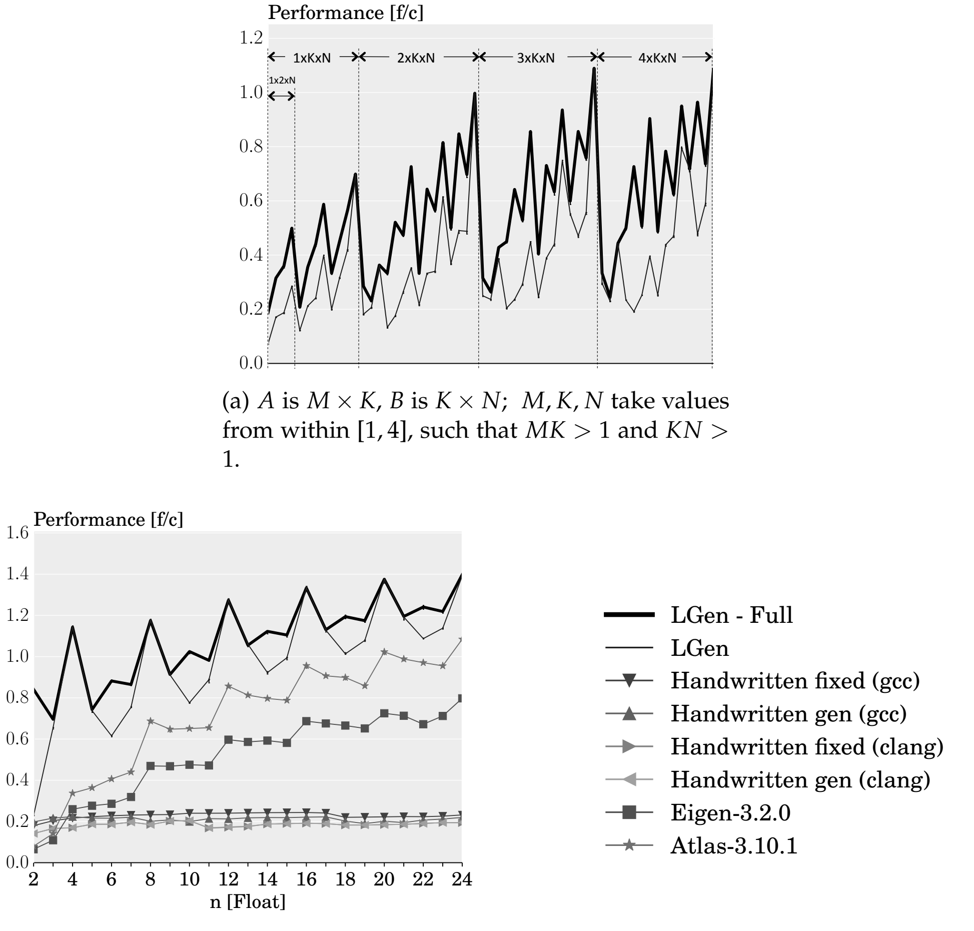
(a) $y = Ax$.



(b) $C = AB$.

(c) $\alpha = x^T Ay$.

Figure B.13: Micro-BLACs. All matrices have size $n \times n$.

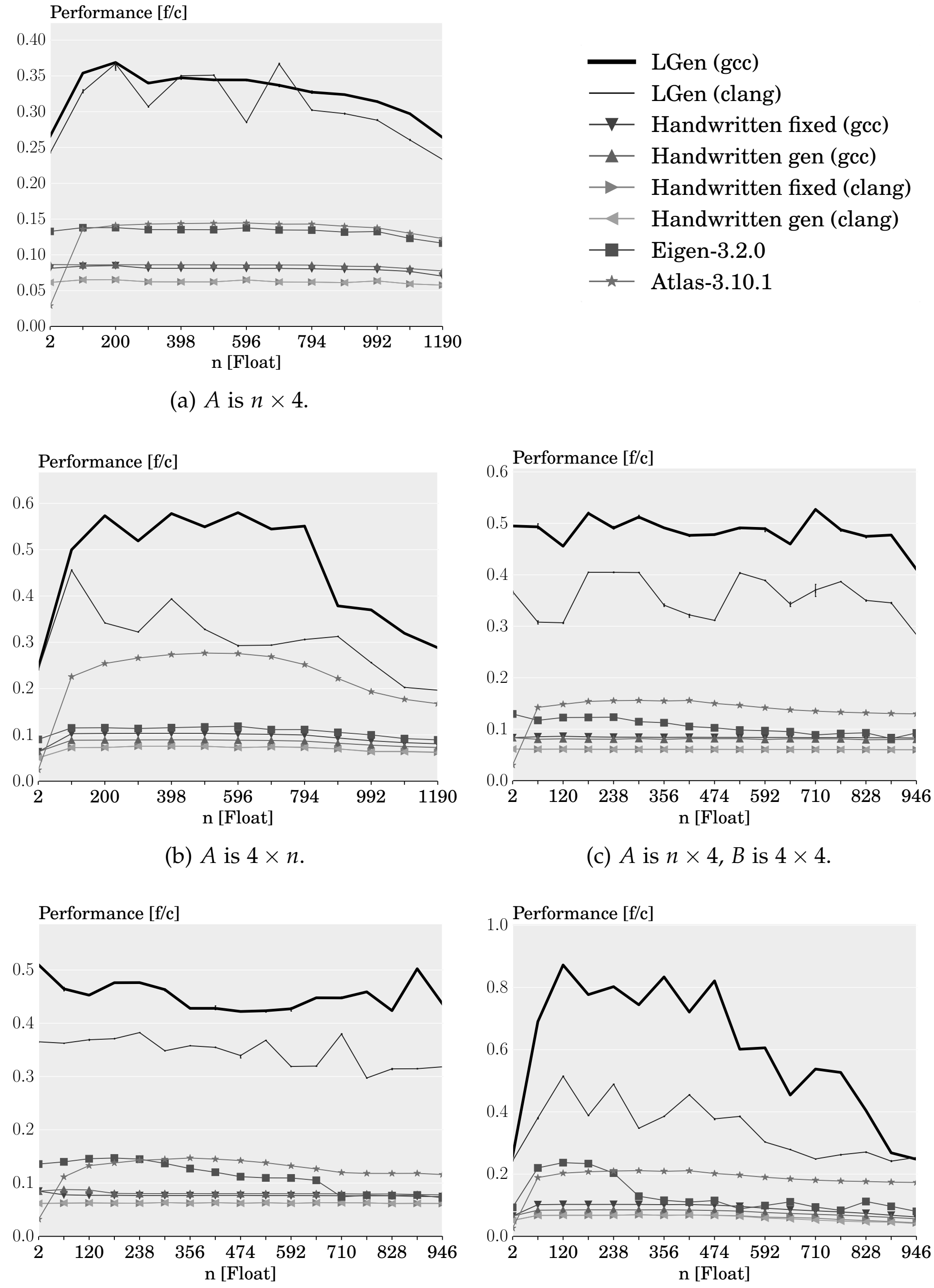


(a) A is $M \times K$, B is $K \times N$; M, K, N take values from within $[1, 4]$, such that $MK > 1$ and $KN > 1$.

(b) A is $100 \times n$, B is $n \times n$.

Figure B.14: $C = AB$ with a large percentage of leftovers.

B.4 ARM1176



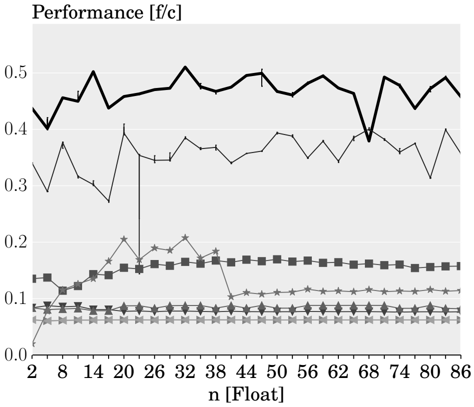
(a) A is $n \times 4$.

(b) A is $4 \times n$.

(c) A is $n \times 4$, B is 4×4.

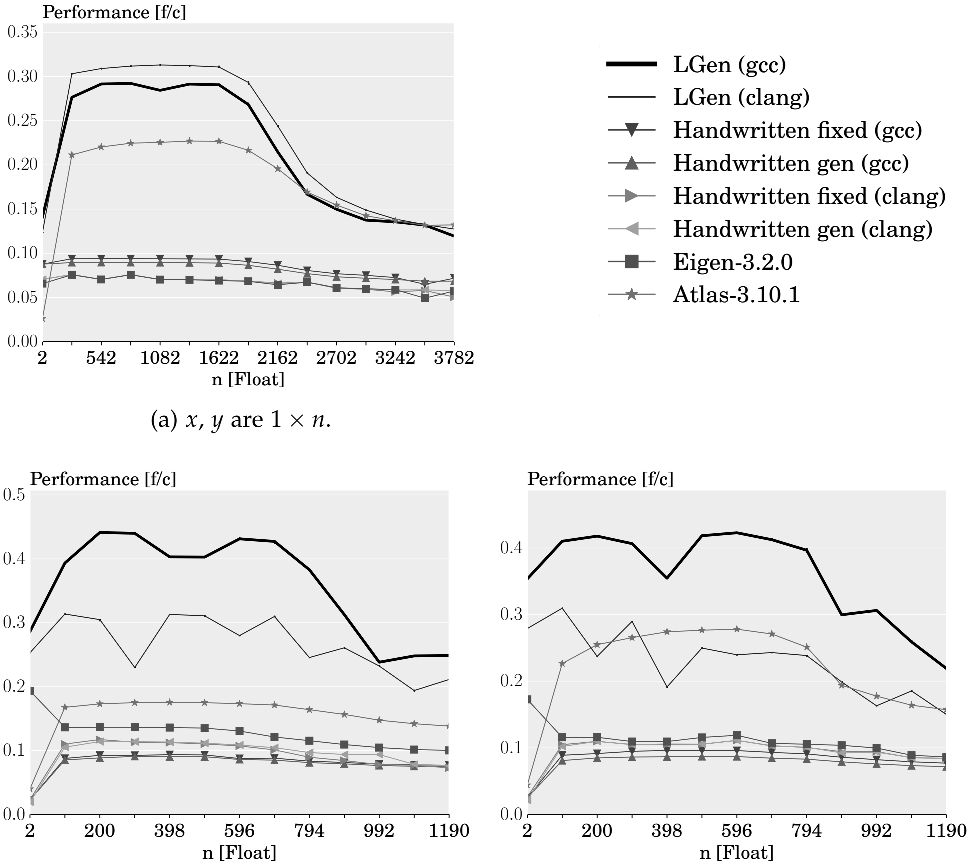
(d) A is 4×4, B is $4 \times n$.

(e) A is $4 \times n$, B is $n \times 4$.



(f) A is $n \times 4$, B is $4 \times n$.

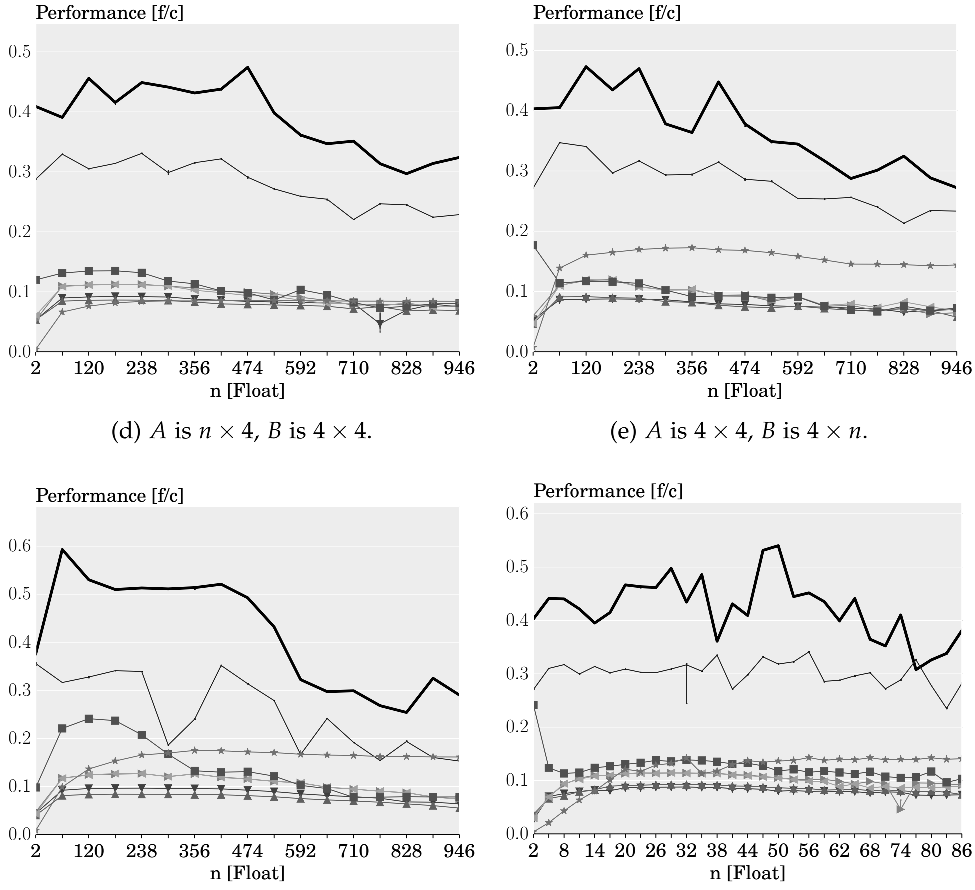
Figure B.15: Simple BLACs. (a)-(b): $y = Ax$; (c)-(f): $C = AB$.



(a) x, y are $1 \times n$.

(b) A is $n \times 4$.

(c) A is $4 \times n$.



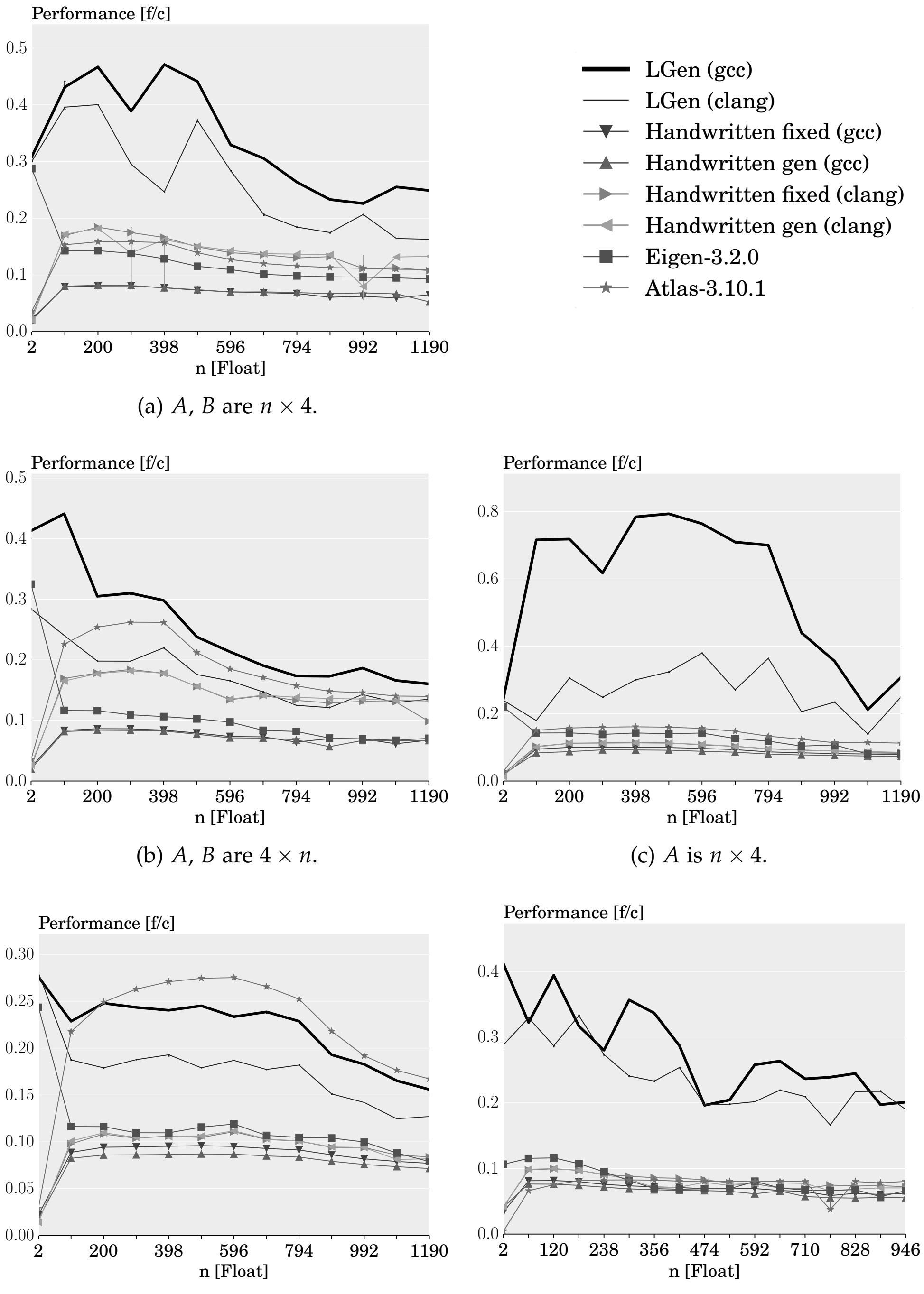
(d) A is $n \times 4$, B is 4×4.

(e) A is 4×4, B is $4 \times n$.

(f) A is $4 \times n$, B is $n \times 4$.

(g) A is $n \times 4$, B is $4 \times n$.

Figure B.16: BLACs that closely match BLAS. (a): $y = \alpha x + y$; (b)-(c): $y = \alpha Ax + \beta y$; (d)-(g): $C = \alpha AB + \beta C$.



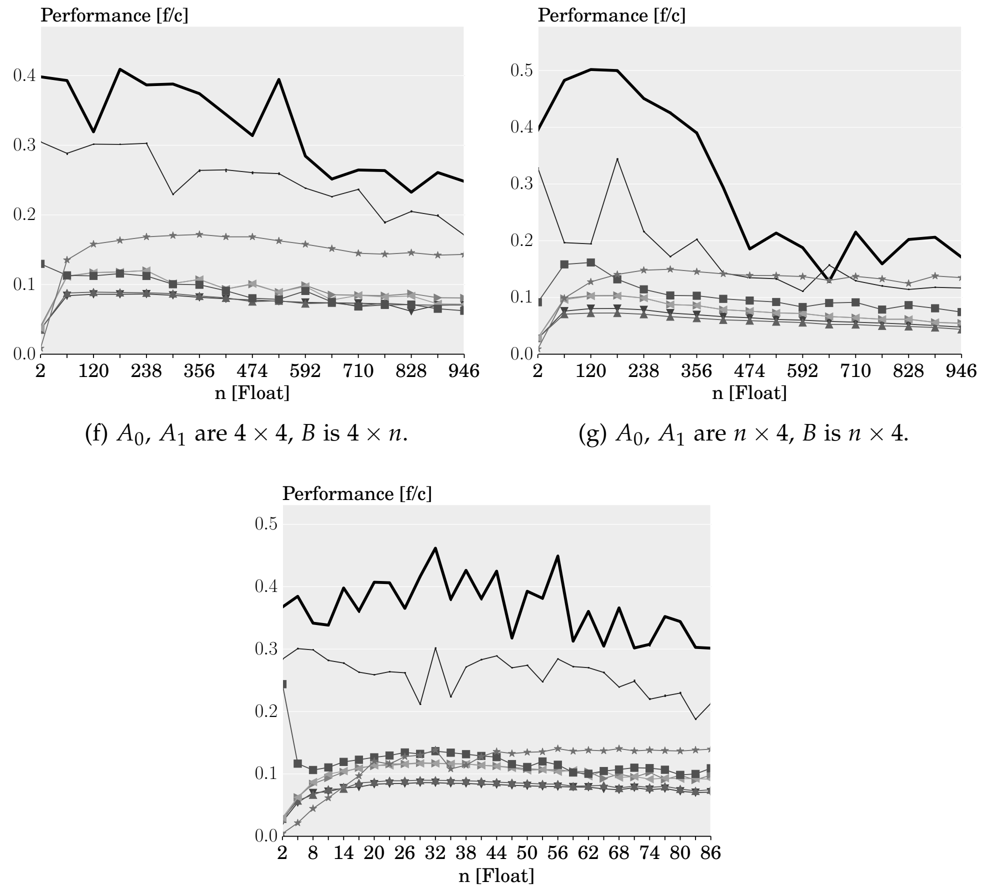
(a) A, B are $n \times 4$.

(b) A, B are $4 \times n$.

(c) A is $n \times 4$.

(d) A is $4 \times n$.

(e) A_0, A_1 are $4 \times n$, B is 4×4.

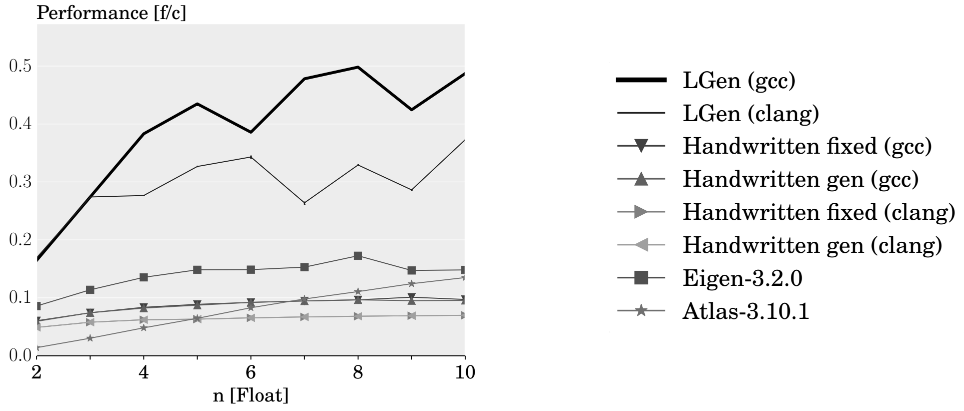


(f) A_0, A_1 are 4 × 4, B is 4 × n.

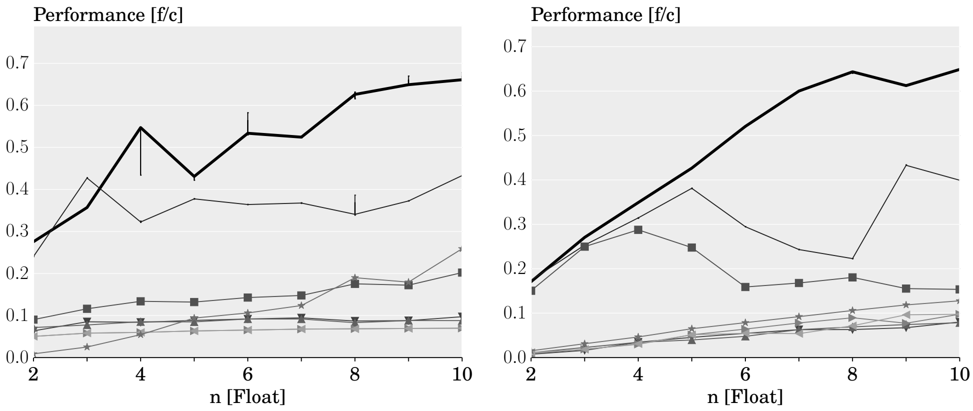
(g) A_0, A_1 are n × 4, B is n × 4.

(h) A_0, A_1 are 4 × n, B is 4 × n.

Figure B.17: BLACs that require more than one BLAS call. (a)-(b): $y = \alpha Ax + \beta Bx$; (c)-(d): $\alpha = x^T Ay$; (e)-(h): $C = \alpha(A_0 + A_1)^T B + \beta C$.



(a) $y = Ax$.



(b) $C = AB$.

(c) $\alpha = x^T Ay$.

Figure B.18: Micro-BLACs. All matrices have size $n \times n$.

Bibliography

[1] ARM. *ARM1176JZF-S® Technical Reference Manual*. November 2009.

[2] ARM. *ARM® Cortex-A8 Technical Reference Manual*. May 2010.

[3] ARM. *ARM® Architecture Reference Manual - ARMv7-A and ARMv7-R edition*. July 2012.

[4] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ort, and Robert van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 31(1):1–26, 2005.

[5] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Supercomputing (SC)*, pages 340–347, 1997.

[6] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *High Performance Computer Architecture (HPCA)*, HPCA '13, pages 1–12, 2013.

[7] Intel Corporation. Intel® Integrated Performance Primitives. `http://software.intel.com/en-us/intel-ipp`.

[8] Intel Corporation. Intel® Math Kernel Library. `http://software.intel.com/en-us/intel-mkl`.

[9] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming*, pages 106–130, 1976.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[11] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[12] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):1–17, March 1988.

[13] ECMA-404 Standard: The JSON Data Interchange Format. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, October 2013.

[14] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Programming Language Design and Implementation (PLDI)*, pages 82–93, 2004.

[15] Flask: A Python microframeword. http://flask.pocoo.org.

[16] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC)*, volume 5658 of *Lecture Notes in Computer Science*, pages 385–410. Springer, 2009.

[17] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005.

[18] Roberto Giacobazzi and Francesco Ranzato. Completeness in abstract interpretation: A domain perspective. In *Algebraic Methodology and Software Technology*, pages 231–245. Springer, 1997.

[19] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[20] Gaël Guennebaud et al. Eigen. http://eigen.tuxfamily.org.

[21] John A. Gunnels, Fred G. Gustavson, Greg Henry, and Robert van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001.

[22] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-028. July 2013.

[23] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. September 2013.

[24] Neil D. Jones and Flemming Nielson. *Abstract interpretation: a semantics-based tool for program analysis*. Oxford University Press, 1995.

[25] Dinesh Kaushik, William Gropp, Michael Minkoff, and Barry Smith. Improving the performance of tensor matrix vector multiplication in cumulative reaction probability based quantum chemistry codes. In *High Performance Computing (HiPC)*, pages 120–130. Springer, 2008.

[26] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *Programming Language Design and Implementation (PLDI)*, pages 127–138, 2013.

[27] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and detecting memory address congruence. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 18–29, 2002.

[28] Saeed Maleki, Yaoqing Gao, María Jesús Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 372–382, 2011.

[29] Polyhedral Compilation. http://polyhedral.info/.

[30] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[31] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[32] Conrad Sanderson et al. Armadillo. http://arma.sourceforge.net/.

[33] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *Computing in Object-Oriented Parallel Environments*, pages 59–70. Springer, 1998.

[34] Daniele G. Spampinato and Markus Püschel. A basic linear algebra compiler. In *International Symposium on Code Generation and Optimization (CGO)*, pages 23–32, 2014.

[35] Joerg Walter and Mathias Koch. uBLAS. http://www.boost.org/libs/numeric.

[36] Clint R. Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing (SC)*, pages 1–27, 1998.

[37] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *Code Generation and Optimization (CGO)*, pages 153–164, 2005.

[38] Field G. Van Zee, Tyler Smith, Francisco D Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, et al. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software (TOMS)*, Submitted.

[39] Field G. Van Zee and Robert van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)*, To appear.