Automatic Generation of Fast BLAS3-GEMM: A Portable Compiler Approach

Xing Su†  Xiangke Liao‡  Jingling Xue‡

†National Laboratory for Parallel and Distributed Processing, College of Computer, NUDT, Changsha, 410073, China
‡School of Computer Science and Engineering, UNSW, Sydney, NSW 2052, Australia

Abstract

GEMM is the main computational kernel in BLAS3. Its micro-kernel is either hand-crafted in assembly code or generated from C code by general-purpose compilers (guided by architecture-specific directives or auto-tuning). Therefore, either performance or portability suffers.

We present a Portable Compiler Approach, POCA, implemented in LLVM, to automatically generate and optimize this micro-kernel in an architecture-independent manner, without involving domain experts. The key insight is to leverage a wide range of architecture-specific abstractions already available in LLVM, by first generating a vectorized micro-kernel in the architecture-independent LLVM IR and then improving its performance by applying a series of domain-specific yet architecture-independent optimizations. The optimized micro-kernel drops easily in existing GEMM frameworks such as BLIS and OpenBLAS. Validation focuses on optimizing GEMM in double precision on two architectures. On Intel Sandybridge and AArch64 Cortex-A57, POCA’s micro-kernels outperform expert-crafted assembly code by 2.35% and 7.54%, respectively, and both BLIS and OpenBLAS achieve competitive or better performance once their micro-kernels are replaced by POCA’s.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers; G.1.3 [Numerical Analysis]: Numerical Linear Algebra

Keywords dense linear algebra, GEMM, code optimization

1. Introduction

Dense linear algebra libraries are fundamental in scientific computing. Basic Linear Algebra Subprograms (BLAS) are routines that provide standard building blocks for performing vector-vector operations (level 1), matrix-vector operations (level 2), and matrix-matrix operations (level 3). BLAS libraries are widely available. Vendor supplied implementations include Intel MKL, AMD ACML and NVIDIA cuBLAS. The HPC community have also contributed several high-performance BLAS implementations such as ATLAS [32], GotoBLAS [12], OpenBLAS [37] and BLIS [28].

For the level-3 BLAS (BLAS3), GEMM (GEneral Matrix Multiplication) is the main computational kernel, as the other level-3 BLAS routines can be defined in terms of GEMM and some level 1 and 2 computations [14]. In addition, GEMM is also the key library routine for deep learning. Finally, the LINPACK benchmarks rely critically on GEMM for its performance measurements. Therefore, optimizing GEMM is the core task in any high-performance BLAS implementation. However, given a C specification of GEMM, general-purpose compilers are still not able to generate machine code that achieves near peak performance.

Broadly speaking, there are three approaches for obtaining optimized loop-based GEMM kernels, (1) assembly programming, (2) auto-tuning, and (3) directive-based programming, yielding different tradeoffs between performance and portability. With assembly programming, domain experts write a few innermost loops in GEMM directly in assembly code. In the case of auto-tuning, ATLAS [32] generates different GEMM kernels (with different parameter values) in C and compile them to run on the actual computing system to find the best-performing one. Finally, directive-based programming is embraced by POET [35] and AUGEM [31]. Given a GEMM kernel in C, POET inserts annotations into the C code to direct source-to-source compiler transformations and AUGEM uses a template-based method to match predefined patterns in the C code and transforms the matched C code sequence into an optimized assembly code sequence.

These three approaches make different performance and portability tradeoffs. Coding GEMM in assembly by domain experts can achieve near peak performance but is tedious and non-portable. Auto-tuning makes the opposite tradeoff. For example, ATLAS relies on general-purpose compilers to generate optimized GEMM kernels for different architectures automatically, thus resulting in portable but sub-optimal performance. In the case of directive-based programming, POET and AUGEM resort to architecture-specific annotations and templates, respectively, written still by domain experts, to guide general-purpose compilers to produce optimized GEMM kernels. In particular, AUGEM is shown to generate optimized GEMM kernels for x86 only.

How do we obtain near peak yet portable performance for GEMM automatically for a wide range of architectures? Despite its algorithmic simplicity, this problem is very chal-
lenging to solve. Any promising solution is significant due to the continued proliferation of computer architectures. In this paper, we present a POrtable Compiler Approach, POCA, implemented in LLVM, to automatically generate and optimize GEMM in an architecture-independent manner. Due to the nature of GEMM, it suffices to focus on its innermost loop, known as a micro-kernel, \( \mu \text{kernel} \). The key insight is to leverage a wide range of architecture-specific abstractions (e.g., SIMD engines supported and instruction latencies) already available in LLVM, by first generating a vectorized \( \mu \text{kernel} \) in the architecture-independent LLVM IR and then boosting its performance by applying a series of domain-specific yet architecture-independent optimizations. The optimized \( \mu \text{kernel} \), obtained without any involvement of domain experts, drops easily in existing GEMM frameworks such as GotoBLAS [12], OpenBLAS [37] and BLIS [28].

We restrict our presentation to GEMM that works on double-precision real numbers, known as DGEMM, as in prior work [26, 31, 35], for two reasons. First, the basic idea behind POCA applies to other variants of GEMM such as SGEMM, CGEMM and ZGEMM (as discussed in Section 2). Second, the LINPACK benchmarks, which rely on GEMM as the performance-critical routine, must work on double-precision real numbers in order to build the TOP500 list, ranking the world’s most powerful supercomputers.

This paper makes the following contributions:

- We introduce a portable compiler approach, POCA for generating highly optimized GEMM fully automatically.
- We evaluate POCA in optimizing GEMM that works on double-precision real numbers on two architectures. On Intel Sandybridge and AArch64 Cortex-A57, POCA’s micro-kernels outperform expert-crafted assembly code by 2.35% and 7.54% respectively. In addition, both BLIS and OpenBLAS achieve competitive or better performance once their micro-kernels are replaced by POCA’s.
- We provide a comprehensive quantitative analysis to understand and evaluate the impact of POCA-specific compiler optimizations on kernel performance.

To the best of our knowledge, this is the first compiler approach for generating fast GEMM kernels portably, without involving domain experts. While the work presented here is for GEMM in BLAS3, the approach can be applied to the other kernels in BLAS3 such as TRSM and GEMV.

2. Structure of GEMM

GEMM comes with four main variants, SGEMM, DGEMM, CGEMM and ZGEMM, which operate on four different data types, single-precision floating-point (S), double-precision floating-point (D), complex single-precision floating-point (C), and complex double-precision floating-point (Z). We first describe a blocked algorithm for DGEMM and introduce several basic concepts used throughout the paper. We then look at SGEMM, CGEMM and ZGEMM briefly.

DGEMM performs a matrix-multiply-add operation on double-precision real numbers, \( C = \beta C + \alpha AB \), where \( A, B \) and \( C \) are matrices of sizes \( M \times K, K \times N \) and \( M \times N \), respectively, and \( \alpha \) and \( \beta \) are scalars. While this operation is algorithmically simple, so that a 3-deep loop nest suffices to accomplish it, a high-performance implementation is quite sophisticated due to the presence of multi-level memory hierarchies on modern processors. Figure 1 shows the program.
structure of a real-world DGEMM kernel implemented via a blocked algorithm. Each loop (in the original 3-deep loop nest) is tiled, resulting in a total of six loops (often referred to as layers 1 – 6). Loop tiling, together with data packing and prefetching, serves to improve data locality and overlap computation and memory access effectively.

In this blocked algorithm, the loops over the N, K and M matrix dimensions are tiled by sizes Ne, Ke and Mc at layers 1, 2 and 3, respectively. Ne, Ke and Mc are carefully selected so that matrix B2 (Ke × Ne) fits into the L3 cache (if it exists). A3 (Mc × Ke) fits into the L2 cache, and both A4 (Mc × Ke) and B4 (Ke × Ne) fit into the L1 cache. At layers 4 and 5, the N and M dimensions are further tiled by sizes Ne and Mc, respectively. As a result, the innermost loop at layer 6 goes over the K dimension for a total of Ks times, with each iteration performing a rank-1 update on the Mc × Ne submatrix of C, as shown at layer 7. Note that a new scalar β, where β = (kk = 0 ? α : β : 1), is introduced at layer 7, to ensure that βC is computed only once by loop kk at layer 2. Ne and Mc are selected so that Mc elements from A, Ne elements from B and Mc × Ne elements from C can fit into registers, with the Mc × Ne submatrix of C residing in registers throughout the innermost loop at layer 6. Finally, A3 [ii : ii + Mc – 1] and B1 are packed into continuous buffers at layer 3 in order to ensure their consecutive memory access at layers 4 – 7.

Goto [12] factors out the innermost three loops at layers 4 – 6 for computing C3 += A2 × B2 as an architecture-dependent kernel, known as a macro-kernel and abbreviated as GEBP (General multiply of a Block of A and a Panel of B). In GotoBLAS [12], and its successor, OpenBLAS [37], their GEMM kernels are developed based on this factorization, with GEBP coded in assembly. Thus, a hand-crafted GEBP kernel, together with a set of well-chosen tile sizes Me, Ne, Ke, Mc and Ne, will suffice to instantiate a highly optimized GEMM on a target processor. BLIS [28] factors out an even smaller kernel, the innermost loop at layer 6, known as a micro-kernel and denoted as £kernel, as the only architecture-dependent kernel to be coded in assembly.

While SGEMM shares exactly the same structure as DGEMM, CGEMM and ZGEMM are slightly different but are easily adapted from SGEMM and DGEMM, respectively. In general, a complex matrix-multiply-add operation β(Cr + Ci)i + α(Ar + Ai)i(Br + Bi)i is factorized as (βCr + αArBr − αAriBi) + (βCi + αAiBi) and thus implemented in terms of four real matrix-multiply-add operations. For [CZ]GEMM, A3 [ii : ii + Mc – 1] and B1 at layer 3 are each packed into two real submatrices of the same size, one for the real part and one for the imaginary part. As a result, a complex GEBP kernel is decomposed into four real GEBP kernels with each containing a real £kernel. All the tile sizes are determined similarly as in the case of DGEMM to improve cache locality. Therefore, for SGEMM, DGEMM, CGEMM and ZGEMM, it suffices to focus on optimizing GEBP or £kernel that works on single- and double-precision real numbers only.

3. POCA: A Portal Compiler Approach

To obtain a fast GEMM kernel portably, we focus on automatically generating a fast £kernel of size M × N that works on real numbers portably, where M and N are input parameters determined by the overall design of GEMM [12].

$$\begin{align*}
C &= (c_{pq}) \\
T &= (t_{pq}); \text{// initialized to 0} \\
A &= A'[i][k] = (a_{pq}) \\
B &= B'[k][i] = (b_{pq}) \\
\text{for } p &= 0 : M - 1 : 1 \\
\text{for } q &= 0 : N - 1 : 1 \\
t_{pq} &= a_{pq} \times b_{pq} \\
c_{pq} &= \beta \times c_{pq} + \alpha \times t_{pq}
\end{align*}$$

\[\text{Figure 2. £kernel with } M \times N = 4 \times 4 \text{ illustrated.}\]

To ease presentation, we will focus on £kernel given in Figure 2 abstracted from Figure 1. A’, B’ and C are matrices of sizes M × Ke, Ke × Ne, and Mc × Ne, respectively. A (B) is the k-th column (row) of A’ (B’). The rank-1 update $T += A \times B$, where $T = (t_{pq})$, $A = (a_{pq})$ and $B = (b_{pq})$, is first performed in the two innermost loops and a multiply-add operation is then performed at the end of loop k to update C, as illustrated for the case when $M = N = 4 \times 4$. Below we will use a double-precision £kernel of this particular size as a running example.

To generate a fast £kernel kernel portably even though it looks simple, we must address two major problems:

- £kernel Generation. How do we perform instruction selection to accomplish the £kernel computation, which is expected to be fully vectorized? This problem depends on the processor’s ISA. Different ISAs lead to different vectorized £kernels. Even for the same ISA, different £kernels can result for different tile sizes M × N given.

- £kernel Optimization. How do we perform instruction scheduling to achieve near peak performance? This problem is also architecture-dependent. General-purpose compilers are known to be ineffective, since they cannot exploit domain-specific knowledge well.

POCA represents the first compiler approach for obtaining near peak performance portably for £kernel, as illustrated in Figure 3. POCA is portable, because it operates on top of the architecture-specific abstractions already provided by general-purpose compilers such as LLVM for a wide range of computer architectures. POCA can deliver near peak performance, because it is structured into two sequential phases, £kernel generation and £kernel optimization. Our £kernel generator produces a vectorized £kernel in the architecture-independent LLVM IR. The main task performed here is instruction selection. Our £kernel optimizer then applies a series of domain-specific but architecture-independent optimizations to produce a highly optimized £kernel. The main task performed here is instruction scheduling.
3.1 The \( \mu \text{kernel} \) Generator

We take as input, the tile size \( M_r \times N_r \) for \( \mu \text{kernel} \), the data type of matrix elements (e.g., float or double), and the architecture targeted, as shown in Figure 3, and produce a vectorized \( \mu \text{kernel} \) in LLVM IR. To perform instruction selection, we carry out vectorization, data prefetching and addressing mode optimization, as described below. While seemingly standard, these passes are applied portably on top of LLVM’s architecture-specific abstractions.

3.1.1 Vectorization

To achieve near peak performance, vector instructions are utilized whenever possible. While SIMD accelerators are ubiquitous in microprocessors, their SIMD features can differ greatly. Despite its simplicity, \( \mu \text{kernel} \) must be vectorized carefully to achieve portability due to the ISA diversity.

To vectorize \( \mu \text{kernel} \) in Figure 2, we adopt the same design exercised in hand-crafted GEMM variants, such as GotoBLAS [12], OpenBLAS [37] and BLIS [28]. The two loops governing the rank-1 update are fully unrolled. \( C \) resides in registers as live-ins but the two vectors \( A \) and \( B \) are loaded from memory into registers across the iterations of loop \( k \). To ensure consecutive memory access, \( A'[i][k] \) and \( B'[k];; \) have already been packed into contiguous buffers. Finally, tile size \( M_r \times N_r \) has also been selected so that \( M_r \) is a multiple of the vector length, denoted \( VL \), expressed in terms of the maximum number of data elements residing in a vector register. Thus, \( A \) can always be loaded into \( M_r/VL \) vector registers with \( M_r/VL \) aligned vector loads and \( C \) occupies exactly \( M_r/N_r/VL \) vector registers.

Therefore, it suffices to consider how to load an \( N_r \)-sized vector \( B \) into vector registers. Currently, we make use of three strategies, VectorLoad-Broadcast, VectorLoad-Shuffle and ScalarLoad-Broadcast. To achieve high bandwidth, vector loads (VectorLoad-Broadcast and VectorLoad-Shuffle) are preferred over scalar loads (ScalarLoad-Broadcast). However, vector loads are considered only when they are all aligned, which happens when \( VL \) divides \( N_r \). Then there are two cases. VectorLoad-Broadcast is adopted on architectures that support “zero-cost scalar broadcasts” (e.g., fmla (vector, by element) on AArch64). VectorLoad-Shuffle is adopted on architectures such as x86 that support vector shuffling. Otherwise, ScalarLoad-Broadcast is selected, resulting in a scalar load for each element of \( B \) and its subsequent broadcast to all the lanes in a vector register.

After all the loads for \( A \) and \( B \) are generated, arithmetic instructions are selected to perform the accumulation operation in \( \mu \text{kernel} \). FMA instructions are used, if available.

Example Figure 4 illustrates how the \( 4 \times 4 \) double-precision \( \mu \text{kernel} \) is vectorized on Sandybridge and Cortex-A57 (with their architectural features listed in Table 5). On Sandybridge, where \( VL = 4 \), VectorLoad-Shuffle is adopted. After the vector loads for \( A \) and \( B \) (lines 1–2), the three vector shuffle instructions (lines 3–5) are executed to obtain three different permutations of \( B \). Then the outer product \( A \times B \) is computed (lines 6–13). On Sandybridge, FMA is not supported. In lines 14–15, the loop induction instructions are executed to enable new vectors \( A \) and \( B \) to be loaded in the next iteration. The prefetching instructions in lines 16–18 will be explained shortly in Section 3.1.2. Note that the shuffle instructions used for restoring \( C0 – C3 \) to the storage order required for \( C \), as shown in Figure 2, are omitted.

On Cortex-A57, where \( VL = 2 \), VectorLoad-Broadcast is adopted. The loads for \( A \) and \( B \) are vectorized (lines 1–4). Then, each scalar element of \( B \) is replicated in a distinct vector register (lines 5–8). These four scalar broadcasts have zero cost, as they will be later combined and performed together with their corresponding float instructions (lines 9–16). For this architecture, FMA instructions are selected.

3.1.2 Data Prefetching

Data prefetching is important for \( \mu \text{kernel} \). In Figure 2, \( A' \) and \( B' \) symbolize \( A_4 (M_r \times K_c) \) and \( B_4 (K_c \times N_r) \) in Figure 1, respectively. According to Section 2, these tile sizes are selected to fit \( A_4 \) and \( B_4 \) into the L1 cache.

From Figure 1, we can see that \( A_4 \) varies but \( B_4 \) stays unchanged at layer 5. Therefore, two different prefetching strategies are applied, following GotoBLAS [12], OpenBLAS [37] and BLIS [28]. For \( B_4 \), its rows are prefetched several iterations in advance. For \( A_4 \), the next \( M_r \times K_c \) row block of \( A_3 \), \( A_4^{\text{next}} \) := \( A_3[i + M_r : i + 2M_r - 1]\), is prefetched, as \( A_4^{\text{next}} \) will be used in the next call to \( \mu \text{kernel} \).

Consider the vectorized \( \mu \text{kernel} \) for Sandybridge given in Figure 4, where the instructions in lines 16–18 serve to prefetch \( A_4^{\text{next}} \) and \( B_4 \). The base addresses of \( A_4 \), \( B_4 \) and \( A_4^{\text{next}} \) are represented by \( AP, BP \) and \( AN \), respectively. Set \( S_r \) be the size of the data type of matrix elements and \( S_c \) the cacheline size for the underlying L1 data cache (in bytes). For each iteration of loop \( k \) in \( \mu \text{kernel} \) (Figure 2), a column of \( A_4^{\text{next}} \) and a row of \( B_4 \) are prefetched by issuing \( [M_r * S_r/S_c] \) and \( [N_r * S_r/S_c] \) prefetching instructions, respectively. When \( M_r \times N_r = 4 \times 4 \), only \( [4 * 8/64] = 1 \) prefetching instruction is needed each. The prefetch distance used for \( B_4 \) is empirical, ranging typically between 256 – 768 bytes. In POCA, 512 bytes are used, by default.
3.1.3 Addressing Mode Optimization

In Poca, loads with immediate offsets are preferred over those with pre- and post-indexed immediate offsets, which also increase (decrease) the base register by the number of loaded data elements as a side-effect. With a pre- and post-indexed load, we can increase code density, but at the expense of one integer $\mu$op issued for modifying the base register. This extra $\mu$op acts as pure overhead, as it consumes resources in the processor frontend (e.g., its $\mu$op buffer space and dispatcher issue slot). The situation is worse for processors with in-order issue or a narrow issue window.

Instead of the post-indexed loads “$A0 = vload \ AP, 2; A1 = vload \ AP, 2;$”, costing 4 $\mu$ops, we prefer the loads with immediate offset “$A0 = vload \ AP; A1 = vload \ AP+2; \ AP = add \ AP, 4;””, costing 3 $\mu$ops only.

3.2 The $\mu$kernel Optimizer

Given a vectorized $\mu$kernel, our optimizer obtains an optimized $\mu$kernel in assembly code. Its core functionality is to schedule the instructions in $\mu$kernel to achieve near peak performance, a task beyond general-purpose compilers.

When crafting $\mu$kernel by hand, domain experts achieve near peak performance by orchestrating carefully instruction scheduling and register allocation. However, general-purpose compilers must make tradeoffs in solving these two problems due to the lack of domain-specific knowledge, resulting in sub-optimal performance. Specifically, compilers usually schedule instructions too conservatively in order to keep enough registers available for later use.

To achieve near peak performance for $\mu$kernel portably, we take a radically different approach. Our $\mu$kernel optimizer can be viewed as a domain-specific compiler for $\mu$kernels, with its three passes given in Figure 3. In Single-Iteration Scheduling, we schedule the instructions in a single iteration of $\mu$kernel to minimize its execution time while keeping the number of live variables, $LiveReg$, as close as but never larger than the total number of (architectural) vector registers, $MaxReg$, available. In Two-Iteration Pipelining, we perform software pipelining on two consecutive iterations of $\mu$kernel to improve resource utilization while pushing $LiveReg$ nearly to the $MaxReg$ limit. In contrast, modulo scheduling [11, 21, 23] is too general to be effective for $\mu$kernel (as validated later). We consider only two consecutive loop iterations because two are sufficient to exhaust all the vector registers available. In Unroll-based Rotating Register Allocation, we perform a rotating register allocation on an unrolled loop to produce an optimized kernel, without register spills (since $LiveReg \leq MaxReg$ always). Therefore, no iteration is needed between the last two passes.

We will illustrate our three passes by using the vectorized $\mu$kernel on Sandybridge, with its relevant architectural features given in Tables 1 and 5. While there are 16 256-bit AVX registers, we assume $MaxReg = 8$ for this example.

3.2.1 Single-Iteration Scheduling

This pass schedules the instructions from one iteration of $\mu$kernel to minimize its execution time, by operating on its dependence graph. It applies list-scheduling with customized heuristics, including four new ones. The objective is to minimize the single-iteration latency while making $LiveReg$ as close as possible but never exceed $MaxReg$. This is practically feasible (without backtracking) since $M_r$ and $N_r$ are so selected that $A$, $B$ and $C$ can fit into registers.

Figure 5 shows the data dependence graph for the vectorized $\mu$kernel on Sandybridge. Each node represents an instruction, labeled with its output variable (except for prefetching instructions, PrfA for prefetch0 AN and PrfB for prefetch0 (BP+64)). All dependencies are depicted in solid arrows. The ENTRY- and EXIT-related artificial dependencies are depicted in dashed arrows. ENTRY
We compute \( \text{EarliestCycle}(I) \) by modeling precisely the underlying architecture, from its instruction decoding to \( \mu \)op execution. All relevant architectural features (e.g., out-of-order vs. in-order, issue width, function units and their latency and throughput), provided by LLVM’s architecture-specific abstraction, are considered.

**BalancedWKLD** We aim to balance the workload for the function units at the processor backend. Let \( FU(I_{\text{last}}) \) (\( FU(I) \)) be the set of function units required for executing \( I_{\text{last}}(I) \), where \( I_{\text{last}}(I) \) is the instruction most recently (being) scheduled. We set \( \text{BalancedWKLD}(I) \) to 1 (true) if \( FU(I_{\text{last}}) \cap FU(I) = \emptyset \), since their \( \mu \)ops will be dispatched to different function units, and 0 (false) otherwise.

**FPFirst** \( \text{FPFirst}(I) \) returns 1 (true) if \( I \) is a floating-point instruction, since \( I \) is highly likely to be on a critical path in the dependence graph, and 0 (false) otherwise.

**MaxHeight, MinDepth, MaxHeight and InstOrder** These four are standard, already available in LLVM. \( \text{MaxRelease} \) favors scheduling instructions such that more instructions will be ready to be scheduled afterwards. \( \text{MinDepth} \) and \( \text{MaxHeight} \) favor instructions with smaller depths and larger heights, respectively. The \( \text{depth} \) (height) of an instruction is measured as the longest distance from \( \text{ENTRY} \) (to \( \text{EXIT} \)) in the dependence graph. Finally, \( \text{InstOrder} \) is a fallback heuristic that respects the original instruction order.

**Example** Table 3 gives the schedule for Figure 4(a) on Sandybridge (based on the architectural information given in Tables 1 and 5). With \( \text{MaxReg} = 8 \) assumed, the instructions are listed in the order scheduled from left to right. For each instruction, the second row gives the cycle at which it is dispatched and the third row gives \( \text{LiveReg} \) immediately afterwards. The single-iteration latency is 12 cycles.

This schedule is unconventional. We have obtained it due to the sophisticated heuristics used for modeling Sandybridge’s out-of-order behavior (provided by LLVM’s machine abstraction). Note that the \( 4 \times 4 \) \( \mu \)kernel has 4 live-in registers, \( C_0, C_1, C_2 \) and \( C_3 \). Thus, \( \text{LiveReg} = 5 \) after the first instruction \( B_0 = \text{vload} \) \( BP \) has been scheduled. Throughout, \( \text{LiveReg} \) is made as close to \( \text{MaxReg} \) as possible, particularly in the middle of a schedule.

Let us select a particular scheduling point to highlight the capability of our cost model. Why is \( C_0 \) scheduled before \( B_3, M_2 \) and \( M_3 \) but executed only afterwards? After \( M_1 \) is scheduled, \( \text{LiveReg} = \text{MaxReg} = 8 \). Then the set of candidate instructions becomes \( \{C_0, C_1, M_2, B_3\} \). We have \( \text{RegPress}(C_0) = \text{RegPress}(C_1) = 1 \) and \( \text{RegPress}(B_3) = \text{RegPress}(M_2) = 0 \). Thus, \( C_0 \) and \( C_1 \) are preferred, as scheduling \( B_3 \) and \( M_2 \) would cause \( \text{LiveReg} > \text{MaxReg} \).

According to the next heuristic, \( \text{EarliestCycle}(C_0) = 9 \) and \( \text{EarliestCycle}(C_1) = 10 \). Thus, \( C_0 \) is finally selected.

Now, the set of candidate instructions is \( \{C_1, M_2, B_3\} \). We first compute \( \text{RegPress}(C_1) = \text{RegPress}(M_2) = \text{RegPress}(B_3) = 1 \). We then find that \( \text{EarliestCycle}(C_1) = 10 \) and \( \text{EarliestCycle}(M_2) = \text{EarliestCycle}(B_3) = 6 \). As \( M_2 \) and \( B_3 \) tie with respect to \( \text{BalancedWKLD} \), \( \text{FPFirst}, \text{MaxRelease} \) and \( \text{MinDepth} \), we finally obtain \( 9 = \text{MaxHeight}(B_3) > \text{MaxHeight}(M_2) = 8 \). Thus, \( B_3 \) is selected to be scheduled after \( C_0 \). However, \( B_3 \)
(at cycle 6) will be dispatched before $C0$ (at cycle 9) due to register renaming and out-of-order dispatching modeled in EarliestCycle. (Note that, for Intel Sandybridge, there are 16 256-bit AVX registers but 144 256-bit physical registers.)

### 3.2.2 Two-Iteration Pipelining

In this second pass, we apply software pipelining to only two consecutive iterations of $\mu$kernel that have been scheduled earlier. The objective is to maximize resource utilization while pushing LiveReg even closer to MaxReg.

#### Algorithm 1 Two-iteration pipelining.

**Require:** $S = [I_0, \ldots, I_{n-1}]$, $S' = [I'_0, \ldots, I'_{n-1}]$

**Ensure:** $P_\mu = [I_0, \ldots, I_{n-1}]$, where $I_l \in S \cup S'$

1: $P_\mu \leftarrow []$
2: $i \leftarrow 0$, $j \leftarrow 0$
3: while $i < n$ do
4:   if (inserting $I'_j$ immediately before $I_i$ ensures that (1) LiveReg $\leq$ MaxReg, and (2) EarliestCycle($I$) remains unchanged as in $S$, for every $I_k$, where $k \geq i$) then
5:     $I \leftarrow I'_j$, $j \leftarrow j + 1$
6:   else
7:     $I \leftarrow I_i$, $i \leftarrow i + 1$
8:   end if
9:   $P_\mu \leftarrow P_\mu \cup [I]$ \hspace{1cm} // append $I$ to $P_\mu$
10: end while
11: $P_\mu \leftarrow P_\mu \cup [I]$ \hspace{1cm} // pipelined kernel
12: return $P_\mu$ \hspace{1cm} // $j = 0 \implies P_\mu = S$

Let $S = [I_0, \ldots, I_{n-1}]$ be the schedule for one iteration and $S' = [I'_0, \ldots, I'_{n-1}]$ the schedule for the next iteration obtained in the first pass. We generate a two-iteration (modulo) schedule by applying Algorithm 1. The basic idea is to move $j$ instructions from $S'$ into $S$ while keeping the relative order of the instructions in each sequence unchanged, so that the latency of $S$, which contains now $n + j$ instructions, remains the same. Let $P_\mu = S[j : n + j - 1]$. Typically, LiveReg is close to MaxReg in the middle of a one-iteration schedule (Table 3). Thus, $P_\mu$ usually contains all the $j$ instructions moved from $S'$. In theory, this can always be enforced by overlapping $S'$ gradually with a later part of $S$. Therefore, $P_\mu$ is the pipelined kernel $S[0 : j - 1]$ is the prologue and $S'$(without the $j$ removed instructions) is the epilogue. In the special case when $j = 0$, which will never happen for $\mu$kernel of a well-chosen $M_r \times N_r$, $P_\mu = S$.

**Example** Continuing from the single-iteration schedule given in Table 3 with 18 instructions, Table 4 shows the pipelined kernel $P_\mu$ consisting of also 18 instructions, with half from $S$ (in normal font) and half from $S'$ (in bold). Previously, two iterations take 24 cycles to execute, with 12 cycles each. After pipelining, two iterations take only 20 cycles (with 4 cycles in the prologue consisting of the first nine instructions in Table 3, 8 cycles in $P_\mu$, and 8 cycles in the epilogue consisting of the last nine instructions in Table 3).

By comparing Tables 3 and 4, we note that we have pushed LiveReg to nearly the MaxReg = 8 limit.

#### 3.2.3 Unroll-based Rotating Register Allocation

In this last pass, we first unroll the pipelined $\mu$kernel with its loop body as $P_\mu$ and then perform a rotating register allocation. We will choose an unroll factor of $UF$ to ensure that every variable is allocated to the same register every $UF$ iterations, resulting in a rotating register allocation [10, 13, 17, 30]. As LiveReg stays near but never exceeds MaxReg in the pipelined schedule, as shown in Table 4, the final $\mu$kernel obtained is highly optimized, free also of register spills.

We compute $UF$ by using a directed graph created from the pipelined kernel $P_\mu$, called Register Transfer Graph (RTG). Each node represents a variable in $P_\mu$. Each edge $v \rightarrow w$ represents a possible flow of a physical register, indicating that a register assigned to $v$ can be reassigned to $w$ after $v$ is dead, since their live ranges do not interfere.

**Algorithm 2 RTG construction.**

**Require:** $P_\mu = [I_0, \ldots, I_{n-1}]$

**Ensure:** RTG
1: $k \leftarrow MaxReg - LiveIn$
2: $\Delta_{\text{psudo}} \leftarrow \{F_0, \ldots, F_{k-1}\}$
3: $\Delta_{\text{dead}} \leftarrow \Delta_{\text{psudo}}$
4: for $I$ in $[I_0, \ldots, I_{n-1}]$ in their sequential order do
5:   Let $LHS(I)$ be the variable defined by $I$
6:   $k \leftarrow k + |KillSet(I)|$
7:   $\Delta_{\text{dead}} \leftarrow \Delta_{\text{dead}} \cup KillSet(I)$
8:   for $d \in \Delta_{\text{dead}}$ do
9:     Add $d \rightarrow LHS(I)$
10: end for
11: $k \leftarrow k - 1$
12: if $k = 0$ then
13:   $\Delta_{\text{dead}} \leftarrow \phi$
14: end if
15: end for
16: for $d \in \Delta_{\text{dead}}$ and $v \in \Delta_{\text{psudo}}$ do
17:   Add $d \rightarrow v$
18: end for

Algorithm 2 constructs the RTG needed. LiveIn gives the set of live-in variables for $P_\mu$ (e.g., $C0 - C3, A0, B1, M0$ in our example). Thus, $\Delta_{\text{psudo}}$ represents the set of pseudo variables (i.e., registers) available initially. KillSet($I$) represents the set of variables that is dead after, i.e., killed by $I$. We use $\Delta_{\text{dead}}$ and $k$ to keep track of the set of dead variables and the number of free registers, respectively. Due to lines 8
10, the variable defined by \( I \), i.e., \( LHS(I) \) can reuse any register previously allocated to \( d \in \Delta_{\text{dead}} \). Note that \( \Delta_{\text{dead}} \) and \( k \) are not updated in sync, because when \( I \) is processed, one free register must be allocated to \( LHS(I) \), causing \( k \) to go down by 1 (line 11). However, this free register may be any previously assigned to any dead variable in \( \Delta_{\text{dead}} \). Thus, \( \Delta_{\text{dead}} \) remains unchanged and is cleared only when \( k = 0 \) (lines 12 – 14). This ensures that all possible flows of registers are considered. Even after \( \Delta_{\text{dead}} \) is cleared, every variable is guaranteed to have at least one (dead variable) predecessor, as \( \text{LiveReg} \leq \text{MaxReg} \) holds always. Finally, \( d \to v \) is added, where \( d \in \Delta_{\text{dead}} \) and \( v \in \Delta_{\text{pseudo}} \).

We can find \( UF \) from \( RTG \) easily to unroll the pipelined \( \mu \)kernel (with \( P_\mu \) as its loop body) to enable a rotating register allocation. As \( \text{LiveReg} \leq \text{MaxReg} \), every node in \( RTG \) lies in a cycle. For every cycle \( c \), which is not necessarily an elementary cycle, let \( T_c \) be the number of live-out variables on \( c \), called its period. \( T_c \) is an upper-bound for the number of live variables in \( c \), since any variable in \( c \) can only become live after it predecessor is dead.

Let \( P = \{c_0, \ldots, c_{m-1}\} \) be a partition of \( RTG \), every \( c_i \) is a cycle. Then \( UF = LCM(T_0, \ldots, T_{m-1}) \). A rotating register allocation can be readily obtained [10, 17, 30].

The problem of finding optimal unroll factors for instruction scheduling and register allocation remains open. In practice, \( RTG \) is small, with less than 50 nodes. We find \( P = \{c_0, \ldots, c_{m-1}\} \) with exhaustive search by preferring a \( UF = LCM(T_0, \ldots, T_{m-1}) \) within [2, 8] if it exists and the smallest otherwise, in several seconds. As \( P_\mu \) is pipelined with \( \text{LiveReg} \) close to \( \text{MaxReg} \), the impact of different unroll factors on performance is small (as evaluated later).

![Figure 6. RTG for \( P_\mu \) in Table 4, with one four-cycle partition shown in four different colors. The period of a cycle is the number of live-out variables drawn in a solid shade. Example For the pipelined kernel \( P_\mu \) in Table 4, Figure 6 shows its RTG and a partition with four cycles. From the last column in Table 4, we see that \( \text{LiveIN} = 7 \). Thus, \( F0 \) represents the only pseudo variable initially available. For the partition shown, its four circles are drawn in four different colors. For each cycle, its period is represented by the number of nodes in a solid shade. Thus, \( UF = LCM(T_{\text{blue}}, T_{\text{red}}, T_{\text{green}}, T_{\text{yellow}}) = LCM(4, 1, 2, 1) = 4 \).

4. Performance Evaluation

In our evaluation, we address three research questions:

**RQ1.** Can the \( \mu \)kernels generated automatically by POCA outperform expert-crafted assembly versions?

**RQ2.** Can representative GEMM frameworks, OpenBLAS and BLIS, achieve competitive or better performance once their expert-crafted \( \mu \)kernels are replaced by POCA’s?

**RQ3.** How effective are POCA’s optimizations?

We have implemented POCA in LLVM (3.9.0). Its \( \mu \)kernel generator is a stand-alone module generating vectorized \( \mu \)kernels in LLVM IR. Its \( \mu \)kernel optimizer is embedded into LLVM’s backend compiler. The “Single-Iteration Scheduling” and “Two-Iteration Pipelining” passes replace LLVM’s machine scheduler. The “Unroll-based rotating Register Allocation” pass is implemented in terms of LLVM’s register allocator by making hints on which physical registers should be allocated to which variables. To optimize \( \mu \)kernel with POCA, LLVM’s “-O3” is turned on.

POCA is implemented in about 9300 lines of C++, with about 1500 LOC in its \( \mu \)kernel generator and 7800 LOC in its \( \mu \)kernel optimizer (3800 LOC for scheduling and pipelining, 2500 for register allocation, and 1500 for others).

| Table 5. Configurations for Sandybridge and Cortex-A57. |
|---------------------------------|-----------------|-----------------|
| Processor | Sandybridge | Cortex-A57 |
| Hardware | Arch | X86,64 | AArch64 |
| | L1 cache | 32KB (8-way) | 32KB (+) |
| | L2 cache | 256KB (8-way) | 512KB (-) |
| | Dispatch | out-of-order (4-issue) | in-order (3-issue) |
| | Execution | out-of-order | out-of-order |
| | SIMD | AVX (256b) | Neon (128b) |
| | FMA | NO | YES |
| GEMM | BLIS | 0.1.8-29 | 0.1.8-29 |
| | OpenBLAS | 0.2.20-dev | 0.2.20-dev |
| | ATLAS | 3.10.3 | 3.10.3 |
| | MKL | 11.03.03 | - |
| Compilers | ICC | 16.0.3 | - |
| | GCC | 4.8.2 | 6.1.0 |

We evaluate POCA on two processors, Intel Sandybridge and AArch64 Cortex-A57, with their hardware and software configurations listed in Table 5. Sandybridge represents a series of dominant processors in the server and HPC market from the x86 camp, while Cortex-A57 is the first 64-bit AArch64 processor from the ARM camp. Both have their own micro-architectures, differing in their cache design, \( \mu \)op dispatchers, and SIMD engines supported. For GEMM frameworks, we have selected two representative hand-crafted implementations, OpenBLAS [37] and BLIS [28], one well-known auto-tuning-based implementation, ATLAS [32], and Intel MKL. All matrices are real double-precision matrices. For compiler-synthesized \( \mu \)kernels, ICC and GCC, are considered, with “-O3” turned on. LLVM is omitted since it generates poorer code for \( \mu \)kernel.

4.1 RQ1: The \( \mu \)kernel Performance

We compare POCA with assembly programming, ICC and GCC. As discussed in Section 2, OpenBLAS [37] and BLIS [28] are written by domain experts, with GEBP and \( \mu \)kernel depicted in Figure 1. In OpenBLAS, GEBP is coded in assembly. In BLIS, only \( \mu \)kernel is coded in assembly. Therefore, its expert-crafted \( \mu \)kernel, identified as BLIS-\( \mu \),
is selected in our evaluation. For BLIS-μ, \( M_r \times N_r = 8 \times 4 \) on Sandybridge and \( M_r \times N_r = 4 \times 8 \) on Cortex-A57.

For the \( \mu \text{kernels} \) generated by POCA for both processors (in several seconds in each case), \( UF = 4 \). For BLIS-μ, loop unrolling is applied identically. To compare POCA with each compiler \( C \in \{ \text{ICC, GCC} \} \), we consider two versions of \( \mu \text{kernel} \), \( C_1 \) and \( C_4 \), generated by \( C \), where \( i \) in \( C_i \) is the unroll factor used. In each case, we wrote a \( \mu \text{kernel} \) loop vectorized with low-level SIMD intrinsics in exactly the same way as the vectorized \( \mu \text{kernel} \) generated by POCA. This loop is then unrolled by a factor of \( i \) and compiled by \( C \) (under “-O3”) to produce \( C_i \). Neither GCC (with -fmodulo-sched) nor ICC emits software-pipelined loops.

![Figure 7. The \( \mu \text{kernel} \) performance results (with the floating-point efficiency shown on the top of each bar).](image)

Figure 7 gives our results. For the \( \mu \text{kernel} \) loop given in Figure 2, \( A' \) and \( B' \) are stored in column- and row-major, respectively, to simulate the effect that both are pre-packed into continuous buffers. We set \( K_c = 192 \) on both Sandybridge and Cortex-A57 to ensure that all the matrices involved fit into the L1 data cache (with \( A' \) being 12KB, \( B' \) being \( 6K \), and the next \( A' \) to be prefetched into the L1 cache (Section 3.1.2)). For each \( \mu \text{kernel} \) evaluated, we show its floating-point performance (measured as the average of 5000 runs), together with its floating-point efficiency.

POCA is the best performer on both processors. For Sandybridge, POCA’s \( \mu \text{kernel} \) generator selects the same instructions as BLIS-μ. For Cortex-A57, POCA’s \( \mu \text{kernel} \) generator also behaves identically as BLIS-μ except that POCA selects loads with immediate offsets while BLIS-μ selects loads with post-indexed immediate offsets (Section 3.1.3).

POCA’s \( \mu \text{kernel} \) optimizer is a key contributor to performance. As shown in Tables 1 and 5 for Sandybridge, POCA schedules and pipelines instructions to improve both execution time and resource utilization aggressively by keeping \( \text{LiveReg} \) as close to \( \text{MaxReg} \) as possible, since a rotating register allocation without register spills is guaranteed later. In contrast, general-purpose compilers and domain experts (to a lesser degree) are more conservative, by scheduling instructions with smaller \( \text{LiveReg} \) in order to reduce potential register spills later, resulting in poorer resource utilization. In fact, no spilling eventually happens for ICC and GCC.

### 4.2 RQ2: The GEMM Performance

The performance advantages of POCA-synthesized \( \mu \text{kernels} \) also translate into the performance competitiveness of POCA-synthesized GEMM routines. Figure 8 gives the results. BLIS-POCA is BLIS with its expert-written \( \mu \text{kernels} \) being replaced by POCA’s. For OpenBLAS, \( M_r \times N_r = 8 \times 4 \) on Sandybridge and \( M_r \times N_r = 4 \times 8 \) on Cortex-A57 are also used [37]. However, the loops at layers 4 – 6 forming GEBP (Figure 1) are all coded in assembly. OpenBLAS-POCA is OpenBLAS with only its \( \mu \text{kernels} \) loop being replaced by POCA’s and the other two loops still in C. For each GEMM routine, the same inputs with 32 matrix sizes are used. Its execution time is the average of three runs.

On Sandybridge, MKL is the best performer for large matrices but exhibits unstable behavior for smaller ones. In contrast, ATLAS is the worst performer on both processors due to its lacking knowledge about the underlying hardware during its auto-tuning process. Compiler-synthesized GEMM routines are not competitive as hand-written ones.

POCA achieves performance results that are competitive as or better than BLIS and OpenBLAS on both processors. Note that \( \mu \text{kernels} \) is an important factor affecting the overall GEMM performance, but not the only one. Other important factors include matrix blocking and submatrix scheduling. This explains why BLIS-POCA and OpenBLAS-POCA show different performance results even on the same architecture.

On Sandybridge, POCA achieves competitive performance as BLIS and OpenBLAS. BLIS-POCA is slightly faster than BLIS, by 1.002x on average. OpenBLAS-POCA is slightly slower than OpenBLAS, exhibiting an average slowdown of 0.994x, possibly because GEBP in OpenBLAS runs slightly faster, with two more loops (at layers 4 and 5) also coded in assembly than OpenBLAS-POCA.

On Cortex-A57, POCA is more impressive. OpenBLAS-POCA outperforms OpenBLAS by 1.065x on average due to a better schedule generated, and BLIS-POCA outperforms BLIS by 1.067x on average due to also the addressing mode optimization (as discussed earlier in Section 4.1).
4.3 RQ3: The Optimization Analysis

We analyze the impact of POCA’s instruction scheduling and loop unrolling on the optimized \( \mu \)kernels generated.

4.3.1 Instruction Scheduling

POCA schedules instructions in two passes, single-iteration scheduling and two-iteration pipelining. To assess their effectiveness individually, we start with a non-scheduled version of \( \mu \)kernel, base, and obtain a scheduled version, sched, by applying one-iteration scheduling only, and a pipelined version, piped, by applying two-iteration pipelining only. For each optimization strategy \( S \in \{ \text{base, sched, piped} \} \), we consider a total of four unrolled variants, \( S-1, S-2, S-4 \) and \( S-8 \), where \( i \) in \( S-i \) is the unroll factor being applied.

![Figure 9. Impact of instruction scheduling on performance (with the floating-point efficiency shown on each bar).](image)

Figure 9. Impact of instruction scheduling on performance (with the floating-point efficiency shown on each bar).

Figure 9 shows the results, obtained in the same setting as in Section 4.1. On Sandybridge, base, sched and piped exhibit similar performance, with a small variation of less than 3\%. Sandybridge is a high-end server processor with powerful out-of-order \( \mu \)op dispatch and execution, reducing the need for software-level instruction scheduling. While applying either sched or piped makes little improvement to base, applying both brings a great performance gain, with POCA outperforming base-1 by 12.5\%.

On Cortex-A57, the situation is a little different: sched is consistently worse than base. With sched, all the loads are scheduled before all the floating-point instructions for one iteration of \( \mu \)kernel, making both seriously imbalanced. Despite its out-of-order execution, Cortex-A57 has an in-order \( \mu \)op dispatcher with a narrow 3-issue window. Thus, the imbalanced instruction stream makes the \( \mu \)op dispatcher the bottleneck, resulting in poor utilization of its backend function units. However, piped is always better than base, because the loads and floating-point instructions from two iterations are interleaved (Tables 1 and 5), creating a better-balanced instruction stream. Finally, when sched and piped are combined, POCA outperforms base-1 by 16.7\%.

Therefore, POCA’s instruction scheduling is effective for not only low- and middle-end processors like Cortex-A57 but also high-end server processors like Sandybridge.

4.3.2 Loop Unrolling

Due to our aggressive one-iteration scheduling and two-iteration pipelining passes, our unroll-based rotating register allocation pass will select an unroll factor within \( [2, 8] \) arbitrarily, as described in Section 3.2.3. On both Sandybridge and Cortex-A57, \( UF = 8 \) happens to be selected.

![Figure 10. Impact of unroll factors on performance (with the floating-point efficiency shown on each bar).](image)

(a) Sandybridge (\( M_r \times N_r = 8 \times 4 \)) (b) Cortex-A57 (\( M_r \times N_r = 4 \times 8 \))

Figure 10. Impact of unroll factors on performance (with the floating-point efficiency shown on each bar).

Figure 10 compares the performance results of BLIS-\( \mu \) and the POCA-synthesized \( \mu \)kernels under all possible unroll factors in \( [2, 8] \), in the same setting as in Section 4.1. According to our register allocation pass, these unroll factors are 4, 5, 6, 7 and 8 for the \( 8 \times 4 \) \( \mu \)kernel on Sandybridge and 2, 4, 6 and 8 for the \( 4 \times 8 \) \( \mu \)kernel on Cortex-A57.

POCA outperforms BLIS-\( \mu \) on both processors under all the unroll factors. The performance advantages of POCA over BLIS-\( \mu \) are already analyzed in Section 4.3.1. Note that the effects of unroll factors on performance are small. In general, \( \mu \)kernel is backend-bound. So the instruction decoding in the frontend rarely becomes a bottleneck if the \( \mu \)op dispatcher and backend function units work at full capacity, which happens for all the unroll factors evaluated.

On Cortex-A57, POCA-8 does not exhibit a performance drop. On Sandybridge, however, POCA-7 and POCA-8 are slightly slower than POCA-4 due to more DSB-MITE switching (\( \mu \)op cache misses) as UF increases (with 17\% and 21\% more DSB-MITE switching cycles for POCA-7 and POCA-8 than POCA-4, obtained by Intel VTune Amplifier).

These results show that with instructions well scheduled, \( \mu \)kernels can achieve near peak performance under a range of unroll factors. The problem of finding a good unroll factor in our register allocation pass has thus been made simple.

5. Related Work

In addition to the prior work discussed on BLAS in Section 1, we review some additional work related to this paper.

The performance of DLA kernels can be improved with compiler techniques, including SIMD vectorization [9, 19, 24, 38, 39], polyhedral optimization [4, 16], and loop tiling [18, 27, 33]. However, general-purpose compilers lack the domain-specific knowledge to achieve near peak performance for DLA kernels. Source-to-source transformations [7, 8, 31], assisted with domain-specific directives, can be more effective, particularly when combined with an empirical tuning engine to ease the auto-tuning process [34, 35].

Empirical auto-tuning is widely used for DLA kernels. The idea starts from [11] and was later adopted by PHiPAC [3]
and ATLAS [32]. Their BLAS libraries automatically select the best from a set of optimized DLA kernels by actually running them on the actual computing system. A DLA kernel may be obtained from a general-purpose compiler, a domain-specific code generator [2], or a domain expert. Being compiler-dependent, both auto-tuning and source-to-source transformations trade performance for portability.

Instead of auto-tuning, analytic techniques [15, 22, 36] attempt to determine optimal values for various parameters in GEMM, such as block sizes, analytically. Techniques for accelerating BLAS on Intel Xeon Phi processors [26] and other DLA kernels on GPUs [6, 29] also exist.

POCA takes a different but complementary approach. With domain-specific optimizations performed on top of LLVM’s abstract machine descriptions, µkernels with near peak performance can be generated automatically for a processor and drop easily in existing GEMM frameworks.

POCA produces highly optimized µkernels by performing domain-specific scheduling and pipelining while guaranteeing a subsequent rotating register allocation without register spills. For general-purpose compilers, software pipelining [11, 17, 20, 21, 23] and register allocation [5, 25] techniques are mature. Techniques for adopting cyclic interval graphs as an alternative representation of interference graph for register allocation are introduced in [10, 13]. Our RTG used for producing a rotating register allocation can be regarded as a (live-range) interval graph for µkernels.

6. Conclusion

In this paper, we present a compiler approach, POCA, for automatically generating highly optimized GEMM kernels with competitive or better performance compared to expert-crafted GEMM implementations, portably for a wide range of computer architectures. The key novelty lies in its domain-specific yet architecture-independent optimizations, especially its aggressive instruction scheduling techniques for maintaining the number of live variables to a maximum. In future work, we plan to extend POCA to multi-threaded GEMM and other dense linear algebra kernels.

Acknowledgments

We thank all the reviewers for their constructive comments on an earlier draft of this paper. This research is partly supported by the National Key Research and Development Program of China (NO.2016YFB0200400), NSFC (NO.61272483, NO.61370018, NO.61402495), and Australian Research Council (DP150102109 and DP170103956).

References