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-

978-1-5090-4931-8/17 © 2017 IEEE

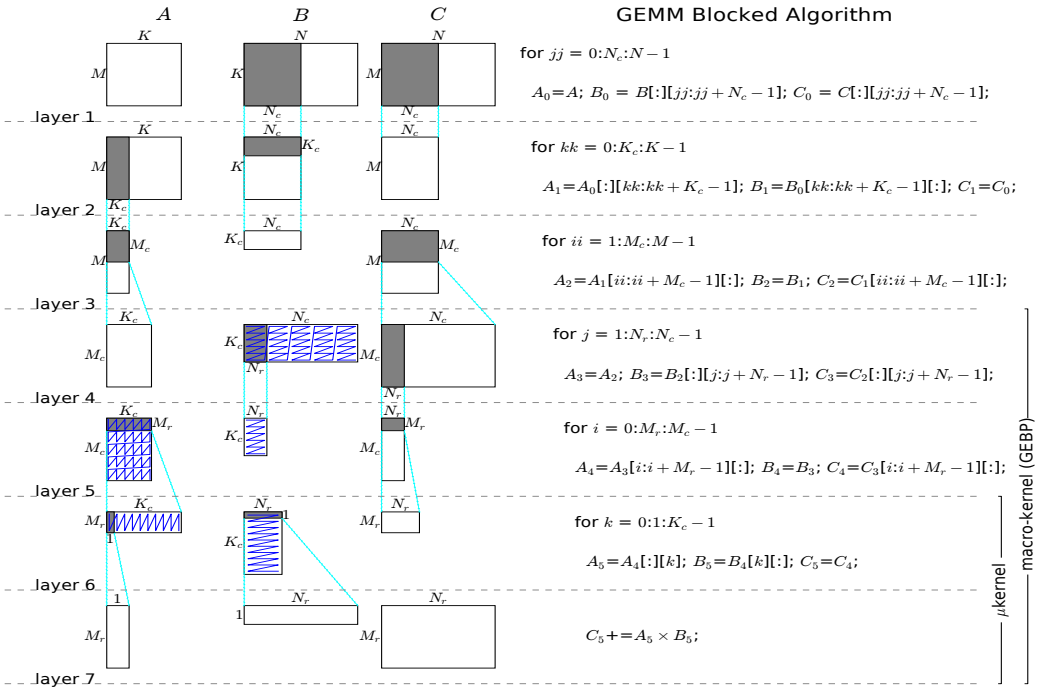


Figure 1. Structure of blocked DGEMM (the structures of SGEMM, CGEMM and ZGEMM are similar (Section 2)).

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, *μkernel*. 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 *μkernel* in the architecture-independent LLVM IR and then boosting its performance by applying a series of domain-specific yet architecture-independent optimizations. The optimized *μkernel*, 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 varaints, 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 α and β 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 N_c, K_c and M_c at layers 1, 2 and 3, respectively. N_c, K_c and M_c are carefully selected so that matrix B_2 ($K_c \times N_c$) fits into the L3 cache (if it exists), A_3 ($M_c \times K_c$) fits into the L2 cache, and both A_4 ($M_r \times K_c$) and B_4 ($K_c \times N_r$) fit into the L1 cache. At layers 4 and 5, the N and M dimensions are further tiled by sizes N_r and M_r, respectively. As a result, the innermost loop at layer 6 goes over the K dimension for a total of K_c times, with each iteration performing a rank-1 update on the $M_r \times N_r$ submatrix of C, as shown at layer 7. Note that a new scalar $\bar{\beta}$, where $\bar{\beta} = (kk == 0\ ?\ \beta : 1)$, is introduced at layer 7, to ensure that βC is computed only once by loop kk at layer 2. N_r and M_r are selected so that M_r elements from A, N_r elements from B and $M_r \times N_r$ elements from C can fit into registers, with the $M_r \times N_r$ submatrix of C residing in registers throughout the innermost loop at layer 6. Finally, $A_1[ii : ii + M_c - 1][:]$ and B_1 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 $C_2 \mathrel{+}= A_2 \times B_2$ 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 M_c, N_c, K_c, M_r and N_r, 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 *μkernel*, 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 $\beta(C_r + C_i i) + \alpha(A_r + A_i i)(B_r + B_i i)$ is factorized as $(\beta C_r + \alpha A_r B_r - \alpha A_i B_i) + (\beta C_i + \alpha A_r B_i + \alpha A_i B_r)i$ and thus implemented in terms of four real matrix-multiply-add operations. For [CZ]GEMM, $A_1[ii : ii + M_C - 1][:]$ and B_1 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 *μkernel*. 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 *μkernel* 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 *μkernel* of size $M_r \times N_r$ that works on real numbers portably, where M_r and N_r are input parameters determined by the overall design of GEMM [12].

```
C = (c_pq);
T = (t_pq); // initialized to 0
for k = 1 : K_c : 1
  A = A'[:][k] = (a_p);
  B = B'[k][:] = (b_q);
  for p = 0 : M_r−1 : 1
    for q = 0 : N_r−1 : 1
      t_pq += a_p * b_q
c_pq = β̄ * c_pq + α * t_pq
```

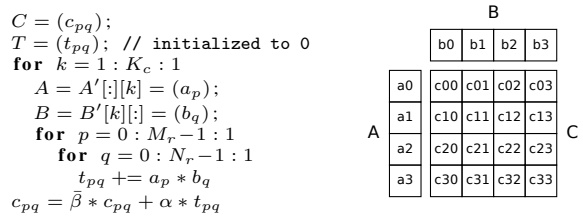


Figure 2. *μkernel* with $M_r \times N_r = 4 \times 4$ illustrated.

To ease presentation, we will focus on *μkernel* given in Figure 2 abstracted from Figure 1. A', B' and C are matrices of sizes $M_r \times K_c$, $K_c \times N_r$, and $M_r \times N_r$, respectively. A (B) is the k-th column (row) of A' (B'). The rank-1 update $T \mathrel{+}= A \times B$, where $T = (t_{pq})$, $A = (a_p)$ and $B = (b_q)$, 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 $M_r \times N_r = 4 \times 4$. Below we will use a double-precision *μkernel* of this particular size as a running example.

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

***μkernel* Generation.** How do we perform instruction selection to accomplish the *μkernel* computation, which is expected to be fully vectorized? This problem depends on the processor's ISA. Different ISAs lead to different vectorized *μkernel*s. Even for the same ISA, different *μkernel*s can result for different tile sizes $M_r \times N_r$ given.

***μkernel* 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 *μkernel*, 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, *μkernel* generation and *μkernel* optimization. Our *μkernel* generator produces a vectorized *μkernel* in the architecture-independent LLVM IR. The main task performed here is instruction selection. Our *μkernel* optimizer then applies a series of domain-specific but architecture-independent optimizations to produce a highly optimized *μkernel*. The main task performed here is instruction scheduling.

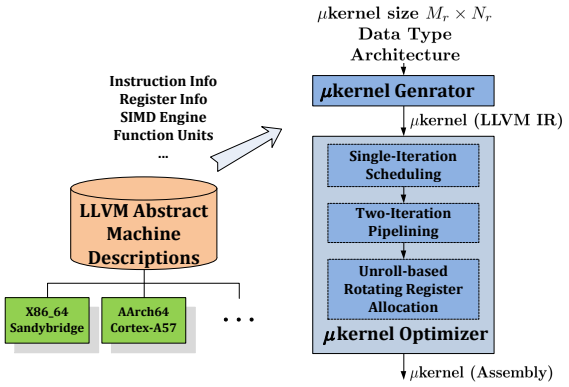


Figure 3. POCA's framework for generating fast *μkernel*s portably on top of LLVM's architecture-specific abstractions.

3.1 The *μkernel* Generator

We take as input, the tile size $M_r \times N_r$ for *μkernel*, the data type of matrix elements (e.g., `float` or `double`), and the architecture targeted, as shown in Figure 3, and produce a vectorized *μkernel* 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, *μkernel* must be vectorized carefully to achieve portability due to the ISA diversity.

To vectorize *μkernel* 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, $A'[:][k]$ 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 $M_r \times N_r/VL$ 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 *μkernel*. FMA instructions are used, if available.

Example Figure 4 illustrates how the 4×4 double-precision *μkernel* 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 `fmla` instructions (lines 9 – 16). For this architecture, FMA instructions are selected.

3.1.2 Data Prefetching

Data prefetching is important for *μkernel*. 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, $A_4^{next} := A_3[i + M_r : i + 2M_r - 1][:]$, is prefetched, as A_4^{next} will be used in the next call to *μkernel*.

Consider the vectorized *μkernel* for Sandybridge given in Figure 4, where the instructions in lines 16 – 18 serve to prefetch A_4^{next} and B_4. The base addresses of A_4, B_4 and A_4^{next} are represented by *AP*, *BP* and *AN*, respectively. Let S_t 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 *μkernel* (Figure 2), a column of A_4^{next} and a row of B_4 are prefetched by issuing $\lceil M_r * S_t/S_c \rceil$ and $\lceil N_r * S_t/S_c \rceil$ prefetching instructions, respectively. When $M_r \times N_r = 4 \times 4$, only $\lceil 4 * 8/64 \rceil = 1$ 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.

```
A0 = vload AP              // <a0, a1, a2, a3>
B0 = vload BP              // <b0, b1, b2, b3>
B1 = vshuffle B0 <1,0,3,2> // <b1, b0, b3, b2>
B2 = vshuffle B1 <2,3,0,1> // <b3, b2, b1, b0>
B3 = vshuffle B2 <1,0,3,2> // <b2, b3, b0, b1>
M0 = fmul A0, B0           //
C0 = fadd C0, M0           // <c00, c11, c22, c33>
M1 = fmul A0, B1
C1 = fadd C1, M1           // <c01, c10, c23, c32>
M2 = fmul A0, B2
C2 = fadd C2, M2           // <c03, c12, c21, c30>
M3 = fmul A0, B3
C3 = fadd C3, M3           // <c02, c13, c20, c31>
AP = add AP, 4             // AP += 4
BP = add BP, 4             // BP += 4
prefetch_0 (BP+64)         // 8 * 64 = 512(bytes)
prefetch_0 AN              // next A block
AN = add AN, 32            // Mr * 8 = 32(bytes)
```

(a) Sandybridge

```
A0 = vload AP              // <a0, a1>
A1 = vload (AP+2)          // <a2, a3>
B0 = vload BP              // <b0, b1>
B1 = vload (BP+2)          // <b2, b3>
B0_0 = vshuffle B0 <0,0>   // <b0, b0>
B0_1 = vshuffle B0 <1,1>   // <b1, b1>
B1_0 = vshuffle B1 <0,0>   // <b2, b2>
B1_1 = vshuffle B1 <1,1>   // <b3, b3>
C0 = fma C0, A0, B0_0      // <c00, c10>
C1 = fma C1, A1, B0_0      // <c20, c30>
C2 = fma C2, A0, B0_1      // <c01, c11>
C3 = fma C3, A1, B0_1      // <c21, c31>
C4 = fma C4, A0, B1_0      // <c02, c12>
C5 = fma C5, A1, B1_0      // <c22, c32>
C6 = fma C6, A0, B1_1      // <c03, c13>
C7 = fma C7, A1, B1_1      // <c23, c33>
AP = add AP, 4             // AP += 4
BP = add BP, 4             // BP += 4
...                        // prefetching insts omitted
```

(b) Cortex-A57

Figure 4. Vectorized *µkernel* in double precision on Sandybridge and Cortex-A57.

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 an side-effect. With a pre- and post-indexed load, we can increase code density, but at the expense of one integer μop issued for modifying the base register. This extra μop acts as pure overhead, as it consumes resources in the processor frontend (e.g., its μop 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 μops, we prefer the loads with immediate offset "`A0 = vload AP; A1 = vload AP+2; AP = add AP, 4;`", costing 3 μops only.

3.2 The *µkernel* Optimizer

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

When crafting *µkernel* 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 *µkernel* portably, we take a radically different approach. Our *µkernel* optimizer can be viewed as a domain-specific compiler for *µkernel*s, with its three passes given in Figure 3. In *Single-Iteration Scheduling*, we schedule the instructions in a single iteration of *µkernel* 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 *µkernel* 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 *µkernel* (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 $LiveReg \leqslant MaxReg$ always). Therefore, no iteration is needed between the last two passes.

Table 1. Instruction information for Sandybridge.

Unit	Total
LoadU	2
ShuffleU	3
FPMulU	1
FPAddU	1
IntegerU	3

Instruction	Latency	Unit
vload	4	*LoadU*
vshuffle	1	*ShuffleU*
fmul	5	*FPMulU*
fadd	3	*FPAddU*
add	1	*IntegerU*
prefetch_0	0	*LoadU*

We will illustrate our three passes by using the vectorized *µkernel* 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 *µkernel* 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 *µkernel* for Sandybridge. Each node represents an instruction, labeled with its output variable (except for prefetching instructions, `PrfA` for `prefetch_0 AN` and `PrfB` for `prefetch_0 (BP+64)`). All dependencies are depicted in solid arrows. The *ENTRY*- and *EXIT*-related artificial dependencies are depicted in dashed arrows. *ENTRY*

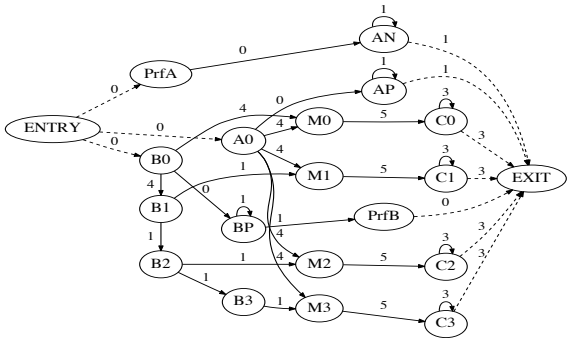


Figure 5. Data dependence graph for the 4×4 vectorized *μkernel* given in Figure 4(a) on Sandybridge.

(*EXIT*) is linked with the original source (sink) nodes. Each dependence edge is labeled by the latency of the instruction.

Formally, let $\mathbb{I}$ be the set of candidate instructions such that the instructions that they depend on have been scheduled. A *scheduling heuristic* is a function $h : \mathbb{I} \rightarrow \mathbf{X}_h$ that assigns a score $x \in \mathbf{X}_h$ to each instruction $I \in \mathbb{I}$. Here, $\mathbf{X}_h$ is a totally ordered set with a total order $\prec_{\mathbf{X}_h}$. For two instructions $I_1, I_2 \in \mathbb{I}$, h prioritizes I_1 if $h(I_1) \prec_{\mathbf{X}_h} h(I_2)$.

To break ties, list scheduling can be made more effective by applying several heuristics to select the best instruction to be scheduled next. If $h(I_1) =_{\mathbf{X_h}} h(I_2)$, where $I_1, I_2 \in \mathbb{I}$, for a particular heuristic, then the next heuristic is tried.

Table 2. Single-iteration scheduling heuristics.

Heuristic (h)	Range ($\mathbf{X}_h$)	Order	
		$\prec_{\mathbf{X}_h}$	$=_{\mathbf{X}_h}$
RegPress	$\mathbb{B}$	$>$	=
EarliestCycle	$\mathbb{N}$	$<$	=
BalancedWKLD	$\mathbb{B}$	$>$	=
FPFirst	$\mathbb{B}$	$>$	=
MaxRelease	$\mathbb{N}$	$>$	=
MinDepth	$\mathbb{N}$	$<$	=
MaxHeight	$\mathbb{N}$	$>$	=
InstOrder	$\mathbb{N}$	$<$	n/a

Table 2 lists eight heuristics tried by the *μkernel* optimizer in that order, with the top four being POCA-specific. For each heuristic, the "Range ($\mathbf{X}_h$)" and "Order ($\prec_{\mathbf{X}_h}$ and $=_{\mathbf{X}_h}$)" columns give its totally ordered set and total order, respectively. $\mathbb{N} = \{0, 1, \cdots\}$ is the set of natural numbers. $\mathbb{B} = \{0, 1\}$. $<$, $>$ and = have the traditional meanings.

Below we describe the eight heuristics used:

RegPress We keep track of $LiveReg$ and compute the set of variables, $KillSet(I)$, that are dead after instruction I is scheduled. We set $RegPress(I)$ to 0 (false) if $LiveReg > MaxReg$ immediately afterwards, i.e., if $LiveReg + 1 - \mid KillSet(I) \mid > MaxReg$ and 1 (true) otherwise.

EarliestCycle $EarliestCycle(I)$ gives the cycle at which I can be executed, based on $EarliestCycle(I')$ for all the instructions I' already scheduled, thereby favoring the earliest-starting instruction. We compute $EarliestCycle(I)$ by modeling precisely the underlying architecture, from its instruction decoding to μop 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 $FU(I_{last})$ ($FU(I)$) be the set of function units required for executing I_{last} (I), where I_{last} (I) is the instruction most recently (being) scheduled. We set $BalancedWKLD(I)$ to 1 (true) if $FU(I_{last}) \cap FU(I) = \varnothing$, since their μops will be dispatched to different function units, and 0 (false) otherwise

FPFirst $PFFirst(I)$ 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. $MaxRelease$ favors scheduling instructions such that more instructions will be ready to be scheduled afterwards. $MinDepth$ and $MaxHeight$ favor instructions with smaller depths and larger heights, respectively. The *depth* (*height*) of an instruction is measured as the longest distance from *ENTRY* (to *EXIT*) in the dependence graph. Finally, $InstOrder$ 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 $MaxReg = 8$ 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 $LiveReg$ 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 4×4 *μkernel* has 4 live-in registers, *C0*, *C1*, *C2* and *C3*. Thus, $LiveReg = 5$ after the first instruction ***B0 = vload BP*** has been scheduled. Throughout, $LiveReg$ is made as close to $MaxReg$ 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 *C0* scheduled before *B3*, *M2* and *M3* but executed only afterwards? After *M1* is scheduled, $LiveReg = MaxReg = 8$. Then the set of candidate instructions becomes $\{C0, C1, M2, B3\}$. We have $RegPress(C0) = RegPress(C1) = 1$ and $RegPress(B3) = RegPress(M2) = 0$. Thus, *C0* and *C1* are preferred, as scheduling *B3* and *M2* would cause $LiveReg > MaxReg$. According to the next heuristic, $EarliestCycle(C0) = 9$ and $EarliestCycle(C1) = 10$. Thus, *C0* is finally selected.

Now, the set of candidate instructions is $\{C1, M2, B3\}$. We first compute $RegPress(C1) = RegPress(M2) = RegPress(B3) = 1$. We then find that $EarliestCycle(C1) = 10$ and $EarliestCycle(M2) = EarliestCycle(B3) = 6$. As *M2* and *B3* tie with respect to $BalacedWKLD$, $FPFirst$, $MaxRelease$ and $MinDepth$, we finally obtain $9 = MaxHeight(B3) > MaxHeight(M2) = 8$. Thus, *B3* is selected to be scheduled after *C0*. However, *B3*

Table 3. Single-iteration scheduling for the double-precision *μkernel* in Figure 4(a) on Sandybridge ($MaxReg = 8$).

Instruction	B0	A0	BP	AP	PrfA	AN	PrfB	B1	M0	B2	M1	C0	B3	M2	M3	C1	C2	C3
Cycle	0	0	0	0	1	1	1	4	4	5	5	9	6	6	7	10	11	12
LiveReg	5	6	6	6	6	6	6	7	7	8	8	7	8	8	7	6	5	4

Table 4. The pipelined kernel P_μ obtained by two-iteration pipelining for the schedule in Table 3 on Sandybridge ($MaxReg = 8$), with the instructions from one iteration in S shown in normal font and the instructions from the next iteration in S' in bold.

Instruction	B2	M1	C0	B3	M2	M3	**B0'**	C1	**A0'**	**BP'**	**AP'**	**PrfA'**	**PrfB'**	**AN'**	C2	**B1'**	**M0'**	C3
Cycle	5	5	9	6	6	7	7	10	8	8	9	9	10	9	11	10	11	12
LiveReg	8	8	7	8	8	7	8	7	8	8	8	8	8	8	7	8	8	7

(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 *μkernel* 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: $P_\mu = [\bar{I}_0, \ldots, \bar{I}_{n-1}]$, where $\bar{I}_i \in S \cup S'$
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) $LiveReg \leqslant MaxReg$, and (2) $EarliestCycle(I_k)$ remains unchanged as in S, for every I_k, where $k \geqslant i$) **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: $P_\mu \leftarrow P_\mu \,\#\, [I]$ `// append I to` P_μ
10: **end while**
11: $P_\mu \leftarrow P_\mu[j : j + n - 1]$ `// pipelined kernel`
12: **return** P_μ `//` $j = 0 \Longrightarrow P_\mu = S$

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_μ 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_μ 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 *μkernel* 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_μ 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_μ, 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 *μkernel* with its loop body as P_μ 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 *μkernel* obtained is highly optimized, free also of register spills.

We compute *UF* by using a directed graph created from the pipelined kernel P_μ, called *Register Transfer Graph (RTG)*. Each node represents a variable in P_μ. 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: $P_\mu = [\bar{I}_0, \ldots, \bar{I}_{n-1}]$
Ensure: RTG
1: $k \leftarrow MaxReg - LiveIn$
2: $\Delta_{psudo} \leftarrow \{F_0, \ldots, F_{k-1}\}$
3: $\Delta_{dead} \leftarrow \Delta_{psudo}$
4: **for** I in $[\bar{I}_0, \ldots, \bar{I}_{n-1}]$ in their sequential order **do**
5: Let $LHS(I)$ be the variable defined by I
6: $k \leftarrow k + |KillSet(I)|$
7: $\Delta_{dead} \leftarrow \Delta_{dead} \cup KillSet(I)$
8: **for** $d \in \Delta_{dead}$ **do**
9: Add $d \rightarrow LHS(I)$
10: **end for**
11: $k \leftarrow k - 1$
12: **if** $k = 0$ **then**
13: $\Delta_{dead} \leftarrow \phi$
14: **end if**
15: **end for**
16: **for** $d \in \Delta_{dead}$ and $v \in \Delta_{psudo}$ **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_μ (e.g., *C0 – C3*, *A0*, *B1*, *M0* in our example). Thus, Δ_{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 Δ_{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 $d \in \Delta_{dead}$. Note that Δ_{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 Δ_{dead}. Thus, Δ_{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 Δ_{dead} is cleared, every variable is guaranteed to have at least one (dead variable) predecessor, as $LiveReg \leqslant MaxReg$ holds always. Finally, $d \rightarrow v$ is added, where $d \in \Delta_{dead}$ and $v \in \Delta_{psudo}$.

We can find UF from RTG easily to unroll the pipelined *µkernel* (with P_μ as its loop body) to enable a rotating register allocation. As $LiveReg \leqslant MaxReg$, 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_μ is pipelined with $LiveReg$ close to $MaxReg$, the impact of different unroll factors on performance is small (as evaluated later).

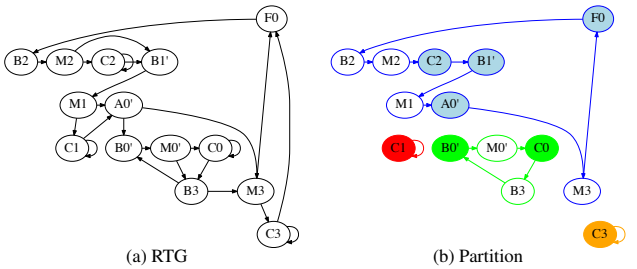


Figure 6. RTG for P_μ in Table 4, with one four-cycle partition shown in four different colors. The period of a cycle is the number of live-out variables drawn in a solid shade.

Example For the pipelined kernel P_μ in Table 4, Figure 6 shows its RTG and a partition with four cycles. From the last column in Table 4, we see that $Livein = 7$. 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, $UF = LCM(T_{blue}, T_{red}, T_{green}, T_{yellow}) = LCM(4, 1, 2, 1) = 4$.

4. Performance Evaluation

In our evaluation, we address three research questions:

RQ1. Can the *µkernel*s 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 *µkernel*s are replaced by POCA's?

RQ3. How effective are POCA's optimizations?

We have implemented POCA in LLVM (3.9.0). Its *µkernel* generator is a stand-alone module generating vectorized *µkernel*s in LLVM IR. Its *µkernel* 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 *µkernel* with POCA, LLVM's "-O3" is turned on.

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

Table 5. Configurations for Sandybridge and Cortext-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

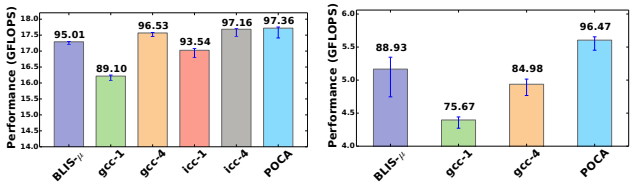
We evaluate POCA on two processors, Intel Sandybridge and AArch64 Cortex-A57, with their hardware and software configurations listed in Table 5. Sandybridge represents a series of dominant processors in the server and HPC market from the x86 camp, while Cortex-A57 is the first 64-bit AArch64 processor from the ARM camp. Both have their own micro-architectures, differing in their cache design, μop 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 *µkernel*s, ICC and GCC, are considered, with "-O3" turned on. LLVM is omitted since it generates poorer code for *µkernel*.

4.1 RQ1: The *µkernel* 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 *µkernel* depicted in Figure 1. In OpenBLAS, GEBP is coded in assembly. In BLIS, only *µkernel* is coded in assembly. Therefore, its expert-crafted *µkernel*, identified as BLIS-μ,

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 $\mu kernel$s 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 $C \in \{\text{ICC}, \text{GCC}\}$, we consider two versions of $\mu kernel$, C_1 and C_4, generated by C, where i in C_i is the unroll factor used. In each case, we wrote a $\mu kernel$ loop vectorized with low-level SIMD intrinsics in exactly the same way as the vectorized $\mu kernel$ 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.



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

Figure 7. The $\mu kernel$ performance results (with the floating-point efficiency shown on the top of each bar).

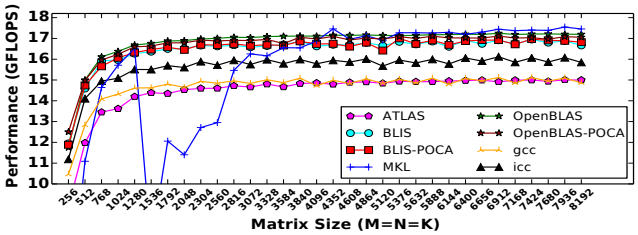
Figure 7 gives our results. For the $\mu kernel$ loop given in Figure 2, A' and B' are stored in column- and row-major, respectively, to simulate the effect that both are pre-packed into continuous buffers. We set $K_c = 192$ on both Sandybridge and Cortex-A57 to ensure that all the matrices involved 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 $\mu kernel$ 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 $\mu kernel$ generator selects the same instructions as BLIS-μ. For Cortex-A57, POCA's $\mu kernel$ 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).

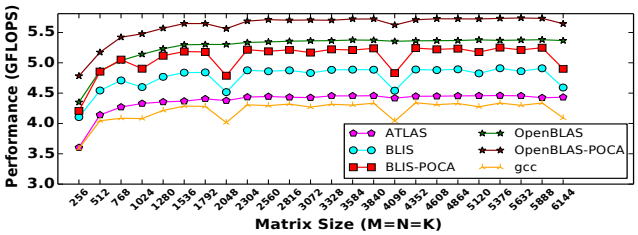
POCA's $\mu kernel$ optimizer is a key contributor to performance. As shown in Tables 1 and 5 for Sandybridge, POCA schedules and pipelines instructions to improve both execution time and resource utilization aggressively by keeping $LiveReg$ as close to $MaxReg$ as possible, since a rotating register 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 $LiveReg$ 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 $\mu kernel$s also translate into the performance competitiveness of POCA-



(a) Sandybridge



(b) Cortex-A57

Figure 8. The GEMM performance for large data sets.

synthesized GEMM routines. Figure 8 gives the results. BLIS-POCA is BLIS with its expert-written $\mu kernel$ 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 $\mu kernel$ 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 $\mu kernel$ 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.

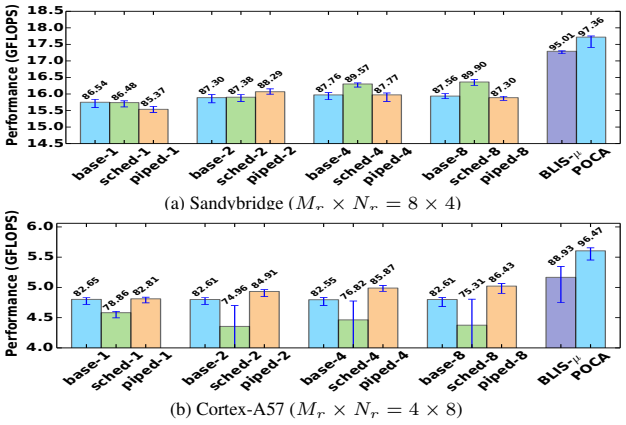
On Cortex-A57, POCA is more impressive. OpenBLAS-POCA outperforms OpenBLAS by 1.065x on average due to a better schedule generated, and BLIS-POCA outperforms BLIS by 1.067x on average due to also the addressing mode optimization (as discussed earlier in Section 4.1).

4.3 RQ3: The Optimization Analysis

We analyze the impact of POCA's instruction scheduling and loop unrolling on the optimized μ*kernel*s 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 μ*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 $S \in \{base, sched, piped\}$, 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.



(a) Sandybridge ($M_r \times N_r = 8 \times 4$)

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

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

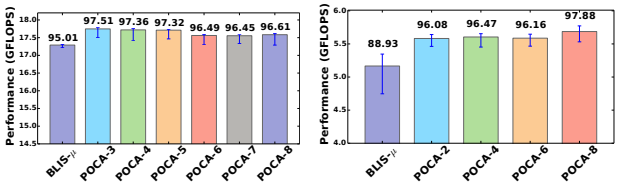
Figure 9 shows the results, obtained in the same setting as in Section 4.1. On Sandybridge, *base*, *sched* and *piped* exhibit similar performance, with a small variation of less than 3%. Sandybridge is a high-end server processor with powerful out-of-order μop dispatch and execution, reducing the need 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 μ*kernel*, making both seriously imbalanced. Despite its out-of-order execution, Cortex-A57 has a in-order μop dispatcher with a narrow 3-issue window. Thus, the imbalanced instruction stream makes the μop 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, $UF = 4$ happens to be selected.



(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-μ and the POCA-synthesized μ*kernel*s 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 8×4 μ*kernel* on Sandybridge and $2, 4, 6$ and 8 for the 4×8 μ*kernel* on Cortex-A57.

POCA outperforms BLIS-μ on both processors under all the unroll factors. The performance advantages of POCA over BLIS-μ are already analyzed in Section 4.3.1. Note that the effects of unroll factors on performance are small. In general, μ*kernel* is backend-bound. So the instruction decoding in the frontend rarely becomes a bottleneck if the μop 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 (μop 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, μ*kernel*s 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 [1] 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, μ*kernel*s with near peak performance can be generated automatically for a processor and drop easily in existing GEMM frameworks.

POCA produces highly optimized μ*kernel*s 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 μ*kernel*s.

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

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research & Development*, 38(5):563–576, 1994.

[2] G. Belter, J. G. Siek, I. Karlin, and E. R. Jessup. Automatic generation of tiled and parallel linear algebra routines. In *IWAPT' 10*, pages 1811–1820.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ansi c coding methodology. In *SC '97*, pages 253–260.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*, pages 101–113.

[5] G. J. Chaitin. Register allocation and spilling via graph coloring. In *CC '82*, pages 98–105.

[6] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for BLAS3 on GPUs. In *IPDPS '11*, pages 255–265.

[7] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. *ACM Trans. Archit. Code Optim.*, 9(3):14:1–14:37, Oct. 2012.

[8] H. Cui, Q. Yi, J. Xue, and X. Feng. Layout-oblivious compiler optimization for matrix computations. *ACM Trans. Archit. Code Optim.*, 9(4):35:1–35:20, Jan. 2013.

[9] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI '04*, pages 82–93.

[10] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: A new model for loop cyclic register allocation. In *PACT '95*, pages 264–267.

[11] L. Gao, Q. H. Nguyen, L. Li, J. Xue, and T. Ngai. Thread-sensitive modulo scheduling for multicore processors. In *ICPP '08*, pages 132–140.

[12] K. Goto and R. a. V. D. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008.

[13] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *CC '93*, pages 176–191.

[14] B. Kågström, P. Ling, and C. van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, Sept. 1998.

[15] V. Kelefouras, A. Kritikakou, and C. Goutis. A matrix-matrix multiplication methodology for single/multi-core architectures using SIMD. *The Journal of Supercomputing*, 68(3):1418–1440, jan 2014.

[16] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *PLDI '13*, pages 127–138.

[17] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *PLDI '88*, pages 318–328.

[18] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS '91*, pages 63–74.

[19] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00*, pages 145–156.

[20] D. M. Lavery and W.-M. W. Hwu. Unrolling-based optimizations for modulo scheduling. In *MICRO '95*, pages 327–337.

[21] J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96*, pages 80–.

[22] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Softw.*, 43(2):12:1–12:18, Aug. 2016.

[23] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO '94*, pages 63–74.

[24] I. Rosen, D. Nuzman, and A. Zaks. Loop-aware SLP in GCC. In *GCC Developers' Summit*, pages 131–142, 2007.

[25] M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04*, pages 277–288.

[26] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *IPDPS '14*, pages 1049–1059.

[27] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *CGO '14*, pages 23:23–23:32.

[28] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.*, 41(3):14:1–14:33, June 2015.

[29] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC'08*, pages 31:1–31:11.

[30] L. Wang, J. Xue, and X. Yang. Reuse-aware modulo scheduling for stream processors. In *DATE '10*, pages 1112–1117.

[31] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *SC '13*, pages 25:1–25:12.

[32] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC '98*, pages 1–27.

[33] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Boston, 2000.

[34] Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *CGO '11*, pages 97–106.

[35] Q. Yi, Q. Wang, and H. Cui. Specializing compiler optimizations through programmable composition for dense matrix computations. In *MICRO '14*, pages 596–608.

[36] K. Yotov, X. Li, G. Ren, M. J. S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, Feb 2005.

[37] X. Zhang, Q. Wang, and Y. Zhang. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *IPDPS '12*, pages 684–691.

[38] H. Zhou and J. Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *CGO '16*, pages 59–69.

[39] H. Zhou and J. Xue. A compiler approach for exploiting partial simd parallelism. *ACM Trans. Archit. Code Optim.*, 13(1):11:1–11:26, Mar. 2016.