SCP: Shared Cache Partitioning for High-Performance GEMM

XING SU, XIANGKE LIAO, HAO JIANG, and CANQUN YANG, National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, China
JINGLING XUE, UNSW Sydney, Australia

GEneral Matrix Multiply (GEMM) is the most fundamental computational kernel routine in the BLAS library. To achieve high performance, in-memory data must be prefetched into fast on-chip caches before they are used. Two techniques, software prefetching and data packing, have been used to effectively exploit the capability of on-chip least recent used (LRU) caches, which are popular in traditional high-performance processors used in high-end servers and supercomputers. However, the market has recently witnessed a new diversity in processor design, resulting in high-performance processors equipped with shared caches with non-LRU replacement policies. This poses a challenge to the development of high-performance GEMM in a multithreaded context. As several threads try to load data into a shared cache simultaneously, interthread cache conflicts will increase significantly. We present a Shared Cache Partitioning (SCP) method to eliminate interthread cache conflicts in the GEMM routines, by partitioning a shared cache into physically disjoint sets and assigning different sets to different threads. We have implemented SCP in the OpenBLAS library and evaluated it on Phytium 2000+, a 64-core AArch64 processor with private LRU L1 caches and shared pseudo-random L2 caches (per four-core cluster). Our evaluation shows that SCP has effectively reduced the conflict misses in both L1 and L2 caches in a highly optimized GEMM implementation, resulting in an improvement of its performance by 2.75% to 6.91%.

CCS Concepts: • **Mathematics of computing** → **Computations on matrices**; • **Computer systems organization** → **Multicore architectures**;

Additional Key Words and Phrases: GEMM, BLAS, optimization, linear algebra, high-performance computing

ACM Reference format:

Xing Su, Xiangke Liao, Hao Jiang, Canqun Yang, and Jingling Xue. 2018. SCP: Shared Cache Partitioning for High-Performance GEMM. *ACM Trans. Archit. Code Optim.* 15, 4, Article 43 (October 2018), 21 pages.
<https://doi.org/10.1145/3274654>

New Article, Not an Extension of a Conference Paper.

This research is supported by the National Key Research and Development Program of China (NO.2017YFB0202003), Innovative Team Support Program of Hunan (2017RS3047), National Natural Science Foundation of Hunan (2018JJ3616), and Australian Research Council (DP170103956 and DP180104069).

Authors' addresses: X. Su, X. Liao, H. Jiang, and C. Yang, National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Yanwachi Main Street 109, Changsha, Hunan, 410073, China; emails: {xingsu, xkliao, haojiang, cqyang}@nudt.edu.cn; J. Xue, UNSW Sydney, NSW 2052, Australia; email: jingling@cse.unsw.edu.au.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2018/10-ART43

<https://doi.org/10.1145/3274654>

1 INTRODUCTION

Dense linear algebra libraries are the most fundamental software in scientific and engineering computing domains. Basic Linear Algebra Subprograms (BLAS) [17] define a collection of APIs that act as standard building blocks for dense matrix operations. BLAS routines are divided into three levels, level 1 for vector-vector operations, level 2 for matrix-vector operations, and level 3 for matrix-matrix operations. Processor vendors often provide BLAS implementations that are highly optimized for their processors, such as Intel MKL, AMD ACML, and NVIDIA cuBLAS. The HPC community has also contributed several high-quality BLAS implementations, e.g., ATLAS [31], GotoBLAS [8], OpenBLAS [39], and BLIS [27, 38].

Among the three levels of BLAS routines, level 3 provides the most opportunities for optimization because it has the highest computational complexity (of $O(n^3)$). Among all level 3 operations, GEMM (General Matrix Multiply) is of the most interest because the other level 3 operations can be defined in terms of GEMM and some level 1 and level 2 operations [10]. In the past, much effort has been spent on optimizing GEMM for different architectures [5, 19, 23, 28, 29].

An optimized GEMM implementation consists of two components: (1) a highly optimized kernel routine to accomplish its computation and (2) an overall strategy to partition the workload into small tasks and schedule these tasks to be executed effectively on the target processor. The kernel routine is a serial program performing matrix-multiply-accumulate on matrices from a single task. The overall strategy partitions the workload by tiling the underlying loop nest and determines a task schedule by choosing a specific traversal order in the loop nest's iteration space. In a multithreaded context, different tasks (loop tiles) are assigned to different threads to harness parallelism. The main objective of the GEMM optimization is to maximize the floating-point operation throughput, measured by Floating-point Operations Per Second (*flops*). For the serial kernel routine, the optimization applies instruction scheduling to improve instruction throughput. For the workload partitioning, the optimization reorganizes memory accesses to reduce long memory latencies for the kernel. In this article, we focus on optimizing the memory accesses for GEMM.

Two techniques, software prefetching and data packing, have been widely used in GEMM implementations to speed up memory accesses. Software prefetch instructions are utilized to load data into cache before they are used, in order to ensure that memory accesses will not incur long latencies and floating-point instructions can execute at peak throughput. To better exploit the capability of on-chip caches, matrices are blocked and then packed into continuous memory buffers that can fit into the caches under consideration before the kernel routine is called. Because GEMM is a computational intensive operation with an arithmetic complexity of $O(n^3)$ and a memory complexity of $O(n^2)$, the overhead caused by data packing is negligible for large input matrices [7].

However, software prefetching and data packing cannot be effectively applied to architectures with shared non-LRU caches. In traditional high-performance processors, e.g., the Intel Xeon series, both L1 and L2 caches are private to processor cores and an LRU replacement policy is adopted. Without considering the last-level cache (LLC), the threads running on different cores prefetch their data into different caches, and private data of different threads cannot cause conflict cache misses. This is no longer the case if a cache is shared by several processor cores. In this case, a cache line prefetched by one thread may be evicted before its lifetime ends because another thread prefetches another cache line into the same cache set. A simple-minded solution would be to reduce the size of a packed matrix so that the data used by several threads can simultaneously reside in a shared cache. Due to set-associativity, this solution is usually ineffective in reducing interthread cache conflicts, especially if the shared cache has a non-LRU replacement policy. Indeed, our evaluation shows that interthread cache conflicts in shared non-LRU caches can heavily hurt GEMM performance, even if packed matrices are reduced (in size) to fit into the shared cache.

Following the trend in recent decades, modern architecture design is introducing more and more cores on a single processor chip. To reduce the cost of on-chip cache memories and coherence networks, shared caches may become common in future many-core processors. In addition, non-LRU replacement policies, for example, pseudo-random, may also be used to further reduce the cache design complexity. Indeed, some high-performance processors based on the ARM technology, e.g., the Phytium processor series [1], have already adopted this design. As a result, developing a better solution to reduce the interthread cache conflicts for shared non-LRU caches is important.

In this article, we present a Shared Cache Partitioning (SCP) method to reduce interthread cache conflicts on architectures with shared non-LRU caches. The key insight is to partition a shared cache into physically disjoint sets and assign different sets to different threads. This can be achieved by exploiting the memory mapping mechanism in a set-associative cache.

To the best of our knowledge, SCP represents the first work to address the interthread cache conflicts problem for GEMM on architectures with shared non-LRU caches. As other level 3 routines, such as TRSM, TRMM, SYMM, and HEMM, call GEMM internally, they should also benefit from SCP; i.e., the SCP methodology can apply transparently to these BLAS-3 routines as well.

The main contributions of this article are as follows:

- We propose SCP, a method for reducing the interthread cache conflicts for GEMM on architectures with shared non-LRU caches.
- We present a quantitative analysis of the negative effect of interthread cache conflicts on the GEMM performance on architectures with shared non-LRU caches.
- We have implemented SCP in the OpenBLAS library and evaluated it on Phytium 2000+, an emerging high-performance processor based on ARM's AArch64 architecture. Phytium 2000+ has 64 cores, private LRU L1 caches, and shared L2 caches with a pseudo-random replacement policy. By reducing the interthread cache conflicts effectively, SCP can improve the performance of a highly optimized GEMM implementation consistently under various parallelization configurations, by 2.75% to 6.91%.

The rest of the article is organized as follows. Section 2 gives some background on GEMM and discusses the cache conflicts incurred. Section 3 introduces our SCP method. Section 4 presents and analyzes our experimental results. Section 5 reviews the related work. Finally, Section 6 concludes.

2 BACKGROUND

2.1 The Memory Hierarchy Review

Modern architectures take advantage of fast on-chip caches to overcome the “memory wall.” As a compromise between direct-mapped caches and full-associative caches, set-associative caches have been the dominant choice for real-world architectures. To understand how cache sharing causes interthread cache conflicts and hurts GEMM performance, we review some details about the working mechanism of set-associative caches and the memory hierarchy of modern processors.

A set-associative cache can be defined by a 4-tuple (c, l, n, nt) , where c is the cache capacity, l is the cache line size, n is the number of ways of associativity, and nt indicates the number of cores sharing the cache. The whole cache memory is divided into $ns = c/l/n$ sets, identified by indices in $\mathcal{S} = [0, ns)$, with each set containing n cache lines. Generally, ns is a power-of-two. An indexing function $\varphi : \mathcal{A} \mapsto \mathcal{S}$ is responsible for mapping the addresses in the address space $\mathcal{A}$ (either virtual or physical) to the cache sets in $\mathcal{S}$. The data at $addr \in \mathcal{A}$ must be fetched into the cache set $\varphi(addr) \in \mathcal{S}$ if requested. A classic implementation of the indexing function is shown below, with $\gg$ and $\&$ representing logical-shift-right and bitwise-and operations, respectively:

$$\varphi(addr) = (addr \gg \log_2(l)) \& (ns - 1),$$

Fig. 1. Memory hierarchy of the four-core processor (nodes labeled with $(layer, index)$).

Table 1. Architectural Parameters of the Four-Core Processor

Type	c	n	l	nt	ns	wc	Policy
L0 Registers	512B	32	16B	1	1	16B	Programmable
L1 Data	32KB	2	64B	1	256	16KB	LRU
L2 Unified	2MB	16	64B	4	2,048	128KB	Pseudo-random

which represents an extraction of the $\log_2(ns)$ bits starting from the $\log_2(l)$ -th position in $addr$'s binary form. It is easy to prove that any continuous memory block that is no larger than c/n cannot conflict with itself. We therefore define a quantity, *way-capacity*, denoted as $wc = c/n$, to represent this size bound for conflict-free memory blocks.

The main memory, NL levels of caches, and NT processor cores form a memory hierarchy and are generally organized in terms of an $NL + 2$ level regular tree structure. The processor cores at level 0 are leaves, the caches at levels $1 - NL$ are intermediate nodes, and the main memory at level $NL + 1$ is the tree root. Any node in the memory hierarchy can be identified by its layer and its unique index at the layer $(layer, index)$. The number of nodes at layer l is NT/nt_l .

Throughout this article, our example platform will be a four-core processor with $NL = 2$ levels of caches. Figure 1 shows its memory hierarchy and Table 1 lists its architectural parameters. Each core has thirty-two 128-bit (16B) vector registers, each capable of storing two double-precision floating-point numbers. For convenience, a processor core at level 0 is represented by its registers, which can be viewed as a special level 0 cache $L0 = (512B, 16B, 32, 1)$ with a programmable replacement policy.

2.2 Structure of GEMM

GEMM performs a matrix-multiply-accumulate operation $C = \beta C + \alpha AB$, where A, B , and C are matrices of sizes $M \times K$, $K \times N$, and $M \times N$, respectively, and α, β are scalars. While this operation is algorithmically simple, so that a three-deep loop nest suffices to accomplish it, a high-performance implementation can be quite sophisticated due to the presence of multilevel memory hierarchies on modern processors. Figure 2 shows the structure of GEMM from the OpenBLAS library [39]. Each loop (in the original three-deep loop nest) is tiled, resulting in a total of six loops (referred to as layers 1–6).

In this blocked algorithm, the jj , kk , and ii loops along 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. At layers 4 and 5, the j and i loops

Fig. 2. Structure of blocked GEMM.

along 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. In OpenBLAS, matrix C is scaled by β before the loops, so the multiply-accumulate operation at layer 7 does not have to scale C again. Tiling factors N_c , K_c , M_c , N_r , and M_r are carefully selected so that matrices at each level fit into a certain level in the memory hierarchy, with the following constraints:

$$es(M_r + N_r + M_r N_r) \leq c_0 \quad (1)$$

$$nt_1 \cdot es(N_r K_c + 2M_r K_c) \leq c_1 \quad (2)$$

$$nt_2 \cdot es(M_c K_c + 2N_r K_c) \leq c_2 \quad (3)$$

$$es(N_c K_c + nt_3 \cdot M_c K_c) \leq c_3, \quad (4)$$

where es denotes the size of a matrix element, e.g., 8B for a double-precision floating-point number, and c_l ($0 \leq l \leq 3$) denotes the capacity of the register file or cache at level l . By Equation (1), M_r and N_r are so constrained that M_r elements from A , N_r elements from B and $M_r \times N_r$ elements from C can fit into the registers (the pseudo level 0 cache). By Equation (2), K_c is so constrained that B_4 ($N_r \times K_c$), A_4 ($M_r \times K_c$), and A_4 's counterpart in the next iteration at layer 5 fit into the L1 cache. By Equation (3), M_c is so constrained that A_3 ($M_c \times K_c$), B_3 ($N_r \times K_c$), and B_3 's counterpart in the next iteration at layer 4 fit into the L2 cache. Finally, by Equation (4), N_c is so constrained that B_2 ($K_c \times N_c$) and A_2 ($M_c \times K_c$) fit into the L3 cache (if it exists). Larger matrices at higher layers 1 to 3 always reside in main memory.

Fig. 3. GEMM parallelization with four threads and data accessing patterns of thread T_1 .

Goto [7] factors out the innermost three loops at layers 4 to 6 for computing $C_2 = C_2 + \alpha A_2 B_2$ as an architecture-dependent kernel, known as GEBP (GEneral multiply of a Block of A and a Panel of B). It is worth noting that before GEBP is called (at layer 3), $A_1[ii : ii + M_c - 1][:]$ and B_1 are packed into A_2 and B_2 in a special continuous layout, so consecutive memory access is ensured within the GEBP kernel. GotoBLAS [8] and its successor, OpenBLAS [39], implement GEMM programs based on this factorization, with GEBP highly optimized for the target processor.

As stated in Section 1, a GEMM implementation is made up by a kernel routine and an overall workload-partitioning strategy. In Figure 2, GEBP serves as the kernel routine, and the workload-partitioning strategy is defined by the three outer loops at layers 1 to 3. Developers can decide how to partition the workload by choosing tile factors M_c , N_c , and K_c , and how to schedule the GEBP tasks by interchanging loop orders at layers 1 to 3. Either the jj loop at layer 1 or the ii loop at layer 3, or both, can be parallelized in a multithreaded context. Specifically, the implementation in Figure 2 chooses an jj - kk - ii loop order and parallelizes the ii loop at layer 3. Different threads work on different portions of A_1 but share the same B_1 .

As a result, there are multiple A_2 instances (one per thread) and a single B_2 instance (shared by all threads). To understand how the ii loop at layer 3 is parallelized on the four-core processor, Figure 3 shows the data accessing patterns for a single thread T_1 . While all the color-shaded submatrices are accessed by T_1 , only the submatrices highlighted by the z-curves (a subset of color-shaded ones) are packed by T_1 . In other words, each thread packs its own A_2 instance but all the threads pack collaboratively the unique B_2 instance.

2.3 Cache Conflicts in GEMM

For set-associative caches, cache misses can be categorized into capacity and conflict misses. While Equations (1) to (4) aim to avoid capacity misses, conflict misses cannot be eliminated completely.

In Figure 2, the packed matrix A_2 will be read N_c/N_r times within a single GEBP execution and is expected to reside in the L2 cache during all the N_c/N_r iterations at layer 4. The packed matrix B_4 is similar in that it is expected to reside in the L1 cache during all the M_c/M_r iterations at layer 5. In general, even with carefully selected tile factors, some matrix elements can still be evicted. For example, some data structures other than the packed matrices or program code (in case of a unified cache) may be fetched into the cache set in which these matrix elements reside. Such cache conflicts are known as *intrathread cache conflicts* because they occur within the execution of a single thread. To deal with intrathread cache conflicts, the GEBP kernel repeatedly prefetches data from A_4 and B_4 to the L1 cache during every iteration at layer 6 before it actually reads them

Table 2. Sizes of Packed Matrices in DGEMM ($es = 8B$)

Matrix	Size	Matrix	Size
A_2	$esM_cK_c = 384KB$	B_2	$esN_cK_c = 8MB$
A_3	$esM_cK_c = 384KB$	B_3	$esN_rK_c = 16KB$
A_4	$esM_rK_c = 8KB$	B_4	$esN_rK_c = 16KB$

just in case the data may have been evicted accidentally from the L1 cache. As GEBP accesses the matrix data much more frequently than other data structures, this repeated prefetching strategy can reduce the penalty caused by intrathread cache conflicts.

In contrary to intrathread cache conflicts, there exists another kind of cache conflicts, called *interthread cache conflicts*, on architectures with shared caches. Suppose two working threads, T_0 and T_1 , are running independently on the four-core processor in Figure 1 to accomplish a GEMM operation. At time t_s , T_0 prefetches some data at address $addr_0$ to the cache set $\varphi_2(addr_0)$ in the shared L2 cache. The data will be actually read at time t_e . Then at some time t_m ($t_s < t_m < t_e$), T_1 prefetches data at address $addr_1$ to the L2 cache. If $\varphi_2(addr_1) = \varphi_2(addr_0)$, then the data at $addr_1$ and $addr_0$ will be fetched into the same cache set. As the shared L2 cache has a non-LRU policy, the data at $addr_0$ may suffer an eviction before it is actually read; i.e., a conflict miss occurs.

In GEMM, every cache is utilized aggressively so that almost all of its cache space is expected to be occupied by packed matrices. Prefetch instructions from different threads are executed repeatedly, interleaved in an unpredictable manner. So interthread cache conflicts described above can occur frequently for shared non-LRU caches, thus making the technique dealing with intrathread cache conflicts ineffective. The situation worsens with more threads sharing the same cache.

3 SCP: THE SHARED CACHE PARTITIONING METHOD

This section introduces our SCP method for reducing interthread cache conflicts. The key insight is that interthread cache conflicts can be eliminated if packed matrices used by different threads are fetched to different cache sets in a shared L2 cache. In Section 3.1, we illustrate SCP by an example. In Section 3.2, we give the algorithms in our SCP method.

3.1 An Example

Suppose we want to implement a double-precision GEMM (DGEMM) on our four-core example processor, where $es = 8B$. We now determine the tiling factors used based on the constraints in Equations (1) through (4). Many solutions for (M_r, N_r) can satisfy Equation (1), including (4, 8), (8, 4), (6, 8), and (8, 6). A tuning process returns $M_r = 4$ and $N_r = 8$ as the best. By Equation (2), $K_c \leq \lfloor (c_{L1}/es/nt_{L1}/(N_r + 2M_r)) \rfloor = \lfloor 32KB/8B/1/16 \rfloor = 256$. Here, we choose $K_c = 256$. By Equation (3), $M_c \leq \lfloor (c_{L2}/es/nt_{L2} - 2N_rK_c)/K_c \rfloor = \lfloor (2MB/8B/4 - 2 * 8 * 256)/256 \rfloor = 240$. To leave space for other data structures and the program code, it is reasonable to shrink M_c slightly, leaving, for example, 192, 208, and 224 as good candidates. Again, a tuning process yields $M_c = 192$. As there is no L3 cache, Equation (4) can be ignored and N_c can be given a large value. Here, we choose $N_c = 1,024NT = 4,096$. Given these tiling factors, the sizes of packed matrices can be calculated. Table 2 lists the sizes of various packed matrices used.

Each A_2 is packed into a single continuous buffer. Figure 4(a) shows how the A_2 instances of four threads are distributed in the shared 16-way L2 cache. Every cache set $s \in [0, 2048)$ contains data from all four A_2 instances, potentially leading to interthread cache conflicts. What if the A_2 matrices are distributed in the way shown in Figure 4(b)? In this case, the different A_2 instances

Fig. 4. Partitioning styles for a shared L2 cache ($n = 16$ and $ns = 2,048$).

from different threads will reside in strictly disjoint cache sets, enabling the interthread cache conflicts caused by A_2 to be eliminated completely!

Figures 4(a) and 4(b) essentially represent two different partitioning styles for a shared cache, referred to as *way-partitioning* and *set-partitioning*, respectively. To obtain the data layout in set-partitioning, A_2 can no longer be stored in a single continuous buffer. Instead, A_2 will be distributed to 12 continuous memory segments, each of size 32KB. The distance from one segment to the next is $wc_2 = 128KB$. While avoiding interthread cache conflicts, set-partitioning introduces some complexity to GEMM implementations because the GEBP kernel and the matrix packing routines now must work with segmented memory buffers instead of continuous ones.

What about B_2 , the other packed matrix used in GEBP? Unlike A_2 , which is thread private, B_2 is shared among all the threads and the packing of B_2 is done collaboratively by all the threads. There are two choices (with different tradeoffs): (1) privatize B_2 and apply set-partitioning and (2) fall back to way-partitioning for B_2 (as done traditionally). B_2 can be made thread private if every thread makes its own pack of the whole B_1 matrix. The first choice achieves a full per-thread data isolation at the expense of redundant packing overhead. The second choice avoids the extra overhead in both time and space but potentially increases the interthread cache conflicts caused by the shared B_2 matrix. In this work, these two choices are evaluated and compared in Section 4.

3.2 Algorithms

Given the tiling factors M_r, N_r, K_c, M_c, N_c , and architectural details of the memory hierarchy, SCP will systematically determine the memory layout for the packed matrices A_2 and B_2 in GEMM. The memory layout of the other packed matrices at lower layers is determined once the memory layout of A_2 and B_2 is determined. We assume that the shared matrix B_2 is packed in the conventional way-partitioning style. If desired, B_2 can be privatized and handled in the same way as A_2 .

We make a few standard assumptions, which usually hold for the architecture considered:

- The architecture has two or three levels of caches.
- All caches are inclusive set-associative caches.
- The caches at the same level are homogeneous.
- A cache's way-capacity is a multiple of the sum of way-capacities of its children, i.e., $(\frac{nt_l}{nt_{l-1}} \cdot wc_{l-1}) \mid wc_l$.

We use a memory descriptor $\mathcal{D}_t^M$ to specify the memory space occupied by a packed matrix M used by a thread t . The memory descriptor is essentially a subspace of the whole address space, i.e., $\mathcal{D}_t^M \subset \mathcal{A}$. Generally, $\mathcal{D}_t^M$ consists of a sequence of N_t^M disjoint memory segments,

ALGORITHM 1: SCP (Phase 1): Allocating memory buffers for A_2 and B_2 **Require:** $L_l = (c_l, l_l, n_l, nt_l)$ for $l \in [0, NL]$, NT , M_r , N_r , K_c , M_c , N_c **Ensure:** $\mathcal{D}_p$ and $\mathcal{D}_s$

```

1:  $size_p \leftarrow esM_cK_c$ 
2:  $size_s \leftarrow esN_cK_c/NT$ 
3:  $align \leftarrow l_1$ 
4: for  $l = 1$  to  $NL$  do
5:   if  $nt_l/nt_{l-1} > 1$  and  $L_l$  is non-LRU then
6:      $align \leftarrow lcm(align, wc_l/nt_l)$ 
7:   end if
8: end for
9:  $size_p \leftarrow \lceil size_p/align \rceil \cdot align$ 
10:  $addr_p \leftarrow allocate(\mathcal{A}, size_p \cdot NT)$ 
11:  $addr_s \leftarrow allocate(\mathcal{A}, size_s \cdot NT)$ 
12:  $\mathcal{D}_p \leftarrow \{[addr_p, addr_p + size_p \cdot NT)\}$ 
13:  $\mathcal{D}_s \leftarrow \{[addr_s, addr_s + size_s \cdot NT)\}$ 

```

$\mathcal{D}_t^M = \{S_0, S_1, \dots, S_{N^M}\}$. A segment is represented by its start and end addresses, $S_i = [s_i, e_i)$, where $s_i < e_i \in \mathcal{A}$. With way-partitioning, the memory descriptor contains only one segment. With set-partitioning, the memory descriptor contains a sequence of equally strided segments.

SCP runs in three phases, as shown in Algorithms 1 to 3, respectively. The first phase allocates the memory buffers for all A_2 and B_2 instances used in GEMM. There are two buffers, $\mathcal{D}_p$ and $\mathcal{D}_s$, for storing private and shared matrices, respectively. The second phase computes the memory descriptors for A_2 instances by partitioning $\mathcal{D}_p$ into NT thread-private buffers $\mathcal{D}_t^{A_2}$, where $t \in [0, NT)$. The buffers of different threads are disjoint, i.e., $\mathcal{D}_i^{A_2} \cap \mathcal{D}_j^{A_2} = \emptyset$ if $i \neq j$. The third phase computes the memory descriptors for B_2 . Because B_2 is shared among all the threads, $\mathcal{D}_t^{B_2}$ does not represent the whole B_2 matrix, but the part (of size $\frac{esN_cK_c}{NT}$), which is packed by thread t .

In Algorithm 1 (Phase 1), the per-thread sizes for memory buffers $\mathcal{D}_p$ and $\mathcal{D}_s$ are computed (lines 1–2). Because the memory space in $\mathcal{D}_p$ will be set-partitioned, extra efforts are made to ensure a proper alignment for $\mathcal{D}_p$. Then $align$ is found (lines 3–8) and $size_p$ is enlarged to a multiple of $align$ (line 9). Specifically, $align$ is initialized with the L1 cache line size l_1 . Then for each level l (line 4) with shared non-LRU caches (line 5), $align$ is updated by computing the least common multiple of the earlier $align$ and wc_l/nt_l (line 6). The insight on using wc_l/nt_l is that each way of the level l cache should be equally partitioned among the nt_l threads in set-partitioning. Finally, memory is allocated (lines 10–11) and buffers $\mathcal{D}_p$ and $\mathcal{D}_s$ are created (lines 12–13).

In Algorithm 2 (Phase 2), the main component is a recursive procedure *subspace* (lines 5–26). This procedure traverses the memory hierarchy to compute for each node, including the processor cores and main memory, a subspace of $\mathcal{D}_p$. There are three input parameters, l and idx for identifying the node on which it is running, and a subspace $\mathcal{D}_l^{idx} \subset \mathcal{D}_p$ allocated to the node. The functionality of *subspace* is to partition $\mathcal{D}_l^{idx}$ among all the children of node (l, idx) . If *subspace* encounters a level 0 node, i.e., a processor core, then $\mathcal{D}_l^{idx}$ is exactly the thread-private memory buffer $\mathcal{D}_{idx}$ for thread idx , causing *subspace* to return early (lines 6–9). Otherwise, the number of children is computed (line 10) and *subspace* iterates over all the child nodes (lines 11–25). For each child node $cidx$ at layer $l-1$ (line 12), its memory space $\mathcal{D}_{l-1}^{cidx}$ is computed and *subspace* is called recursively on it (line 24). $\mathcal{D}_{l-1}^{cidx} \subset \mathcal{D}_l^{idx}$ is obtained by intersecting $\mathcal{D}_l^{idx}$ with a temporal memory space $\mathcal{D}_{temp}$ (line 23). The if-else branch (lines 13–22) represents two distinct ways

ALGORITHM 2: SCP (Phase 2): Computing the memory descriptors for A_2 **Require:** $L_l = (c_l, l_l, n_l, nt_l)$ for $l \in [0, NL]$, $\mathcal{D}_p$, NT **Ensure:** $\mathcal{D}_t^{A_2}$ for $t \in [0, NT]$

```

1:  $\mathcal{D}_{NL+1}^0 \leftarrow \mathcal{D}_p$ 
2:  $nt_{NL+1} \leftarrow NT$ 
3: call subspace( $NL + 1, 0, \mathcal{D}_{NL+1}^0$ )
4:
5: procedure subspace ( $l, idx, \mathcal{D}_l^{idx}$ ) do
6:   if  $l = 0$  then
7:      $\mathcal{D}_{idx}^{A_2} \leftarrow \mathcal{D}_l^{idx}$ 
8:     return
9:   end if
10:   $nchildren \leftarrow nt_l / nt_{l-1}$ 
11:  for  $i = 0$  to  $nchildren - 1$  do
12:     $cidx \leftarrow idx \cdot nchildren + i$ 
13:    if  $l = NL + 1$  or  $L_l$  is LRU then
14:       $lb \leftarrow \min(\mathcal{D}_l^{idx})$ 
15:       $ub \leftarrow \max(\mathcal{D}_l^{idx}) + 1$ 
16:       $len \leftarrow (ub - lb) / nchildren$ 
17:       $\mathcal{D}_{temp} \leftarrow [lb + i \cdot len, lb + (i + 1) \cdot len)$ 
18:    else
19:       $len \leftarrow ns_l / nchildren$ 
20:       $S_i \leftarrow [i \cdot len, (i + 1) \cdot len)$ 
21:       $\mathcal{D}_{temp} \leftarrow \varphi_l^{-1}(S_i)$ 
22:    end if
23:     $\mathcal{D}_{l-1}^{cidx} \leftarrow \mathcal{D}_l^{idx} \cap \mathcal{D}_{temp}$ 
24:    call subspace( $l - 1, cidx, \mathcal{D}_{l-1}^{cidx}$ )
25:  end for
26: end procedure

```

for computing $\mathcal{D}_{temp}$. If the node is main memory or an LRU cache (line 13), $\mathcal{D}_l^{idx}$ is partitioned in the conventional way-partitioning style. Otherwise, the else branch (line 18) performs a set-partitioning on $\mathcal{D}_l^{idx}$. Initially, the main memory is the whole $\mathcal{D}_p$ (line 1) and nt_{NL+1} is set to NT (line 2) because the main memory is shared by all the threads. Then the *subspace* procedure starts from the main memory (line 3) and traverses the memory hierarchy in a depth-first-search order.

In Algorithm 3 (Phase 3), the computation of $\mathcal{D}_t^{B_2}$ is quite simple. This is done by dividing $\mathcal{D}_s$ into NT partitions, one for each thread. First, the range of $\mathcal{D}_s$ is obtained (lines 1–2) and the capacity of $\mathcal{D}_s$ is equally divided among the NT threads (line 3). Then $\mathcal{D}_s$ is partitioned in a way-partitioning style and each thread t obtains the buffer space for its part in B_2 (lines 4–6).

4 PERFORMANCE EVALUATION

We have implemented SCP in the OpenBLAS [39] library and evaluated it on a Phytium 2000+ processor. The Phytium 2000+ processor is an emerging high-performance 64-core processor based on ARM's AArch64 architecture. The 64 cores are organized into 16 clusters with each cluster containing four cores. The structure of a cluster is almost the same as the four-core processor in Figure 1 except that the L2 caches of all 16 clusters connect to the main memory with hardware coherence. Table 1 lists the architectural parameters for the cores and caches in this processor and Table 3 shows the software environment used. The L2 caches in the Phytium 2000+ processor are

Table 3. Software Environment Used in Evaluation

Environment	Software	Version
Operating System	GNU/Linux	4.4.0 AArch64
Compiler	GNU/GCC	6.3.0
Thread Model	OpenMP	4.0
BLAS	OpenBLAS	0.3.0-dev

ALGORITHM 3: SCP (Phase 3): Computing the memory descriptors for B_2 **Require:** $L_l = (c_l, l_l, n_l, nt_l)$ for $l \in [0, NL]$, $\mathcal{D}_s$ **Ensure:** $\mathcal{D}_t^{B_2}$ for $t \in [0, NT)$

```

1:  $lb \leftarrow \min(\mathcal{D}_s)$ 
2:  $ub \leftarrow \max(\mathcal{D}_s) + 1$ 
3:  $len \leftarrow (ub - lb) / nchildren$ 
4: for  $t = 0$  to  $NT - 1$  do
5:    $\mathcal{D}_t^{B_2} \leftarrow \{[lb + t \cdot len, lb + (t + 1) \cdot len)\}$ 
6: end for

```

physically indexed. In our implementation, physically continuous memory buffers (allocated by the HUGETLBFS filesystem provided by the Linux kernel) are used for storing matrices in packed form. Therefore, for a given shared L2 cache, the matrix data accessed by different threads are guaranteed to be free of conflict cache misses. We restrict our evaluation to DGEMM, as in prior work [23, 30, 36], for two reasons. First, the basic idea behind SCP applies to other variants of GEMM such as SGEMM, CGEMM, and ZGEMM. Second, the LINPACK benchmark, which is used to build the TOP500 list of the world's most powerful supercomputers [2], relies on the DGEMM variant.

In our evaluation, we use the tiling factors computed as in Section 3.1: $M_r = 4$, $N_r = 8$, $K_c = 256$, $M_c = 192$, $N_c = 1,024NT$, as this combination causes GEMM to deliver the best performance when SCP is not used. The degree of parallelism is controlled by two parameters, the number of active clusters NC and the number of active threads per cluster NT_C . The total number of threads is $NT = NC \times NT_C$. For the Phytium 2000+ processor, we have $NC \in [1, 16]$ and $NT_C \in [1, 4]$. Generally, the DGEMM performance is measured in *flops*. As NT varies in our evaluation, we use another metric derived from *flops*, the average thread efficiency, to describe our performance results consistently. The average thread efficiency is a normalized value computed as $E_{avg} = \widehat{flops} / (NT \times \widehat{flops})$, where $\widehat{flops}$ stands for the theoretical performance peak of a single core.

The matrix sizes selected range from 3,072 to 4,096 when $NT \in \{4, 8\}$ and to 6,144 when $NT \in \{16, 32, 64\}$. These sizes are large enough to make GEMM run stably at peak performance given the total number of threads participating in the computation.

We show how cache sharing can hurt GEMM performance (Section 4.1), demonstrate how SCP can reduce the penalty caused by cache sharing (Section 4.2), conduct a quantitative analysis of the cache miss rates (Section 4.3), and discuss the privatization of the shared matrix B_2 (Section 4.4).

4.1 Penalty of Cache Sharing

In a cluster, the L2 cache is shared by its four cores. The impact of cache sharing can be demonstrated by comparing the DGEMM performance results achieved with the same NT but different NT_C s. Figure 5 shows the results. Each curve represents a parallelization configuration. Different subfigures are configured with different NT s, and different curves in the same subfigure are configured with different NT_C s. As there is only one configuration for $NT = 64$ ($NC = 16$ and

Fig. 5. Average thread efficiency with cache sharing.

Table 4. Average Thread Efficiency with Cache Sharing

	$NT = 4$	$NT = 8$	$NT = 16$	$NT = 32$	$NT = 64$
$NT_C = 1$	91.46	91.30	90.30	-	-
$NT_C = 2$	90.12	90.25	89.49	88.26	-
$NT_C = 4$	84.05	80.61	79.74	79.81	71.71

$NT_C = 4$), no comparison is available. Thus, the results for $NT = 64$ and $NT = 32$ are combined in Figure 5(d).

For all NT s, $NT_C = 1$ achieves the best performance, followed by $NT_C = 2$, and $NT_C = 4$ comes at the lowest efficiency. Table 4 summarizes the average value of E_{avg} across all the matrix sizes. Vertically, the results in the same column show how E_{avg} varies with NT_C . For all NT s, $NT_C = 1$ outperforms $NT_C = 4$ by a large margin. The performance gap is 7.4% when $NT = 4$ and roughly 10% when $NT > 4$. Horizontally, the results in the same row reflect the scalability of DGEMM with certain NT_C s. With $NT_C = 1$, the performance scales linearly with the number of threads. With $NT_C = 2$, E_{avg} suffers a 1.86% loss when NT grows from 4 to 32. The situation is much worse for $NT_C = 4$, in which E_{avg} drops by 14.19% from $NT = 4$ to $NT = 64$.

This shows clearly that cache sharing has a considerable impact on the DGEMM performance.

Fig. 6. Average thread efficiency under SCP.

4.2 Effectiveness of SCP

We evaluate SCP using the same method as in Section 4.1. SCP is not evaluated with $NT_C = 1$ because there is no need to partition the cache if $NT_C = 1$. Figure 6 shows the results. For comparison purposes, the results without SCP (denoted as Base) are also presented.

Table 5. Average Thread Efficiency Improvement under SCP

	$NT = 4$	$NT = 8$	$NT = 16$	$NT = 32$	$NT = 64$
$NT_C = 2$	1.77	1.69	1.82	1.78	-
$NT_C = 4$	2.75	4.62	4.41	3.11	6.91

Table 6. Hardware Events in GEMM

Name	Event	Description
L1M	PAPI_L1_DCM	L1 data cache misses
L1H	PAPI_L1_DCH	L1 data cache hits
L2M	PAPI_L2_DCM	L2 data cache misses
L2H	PAPI_L2_DCH	L2 data cache hits

Under all the parallelization configurations, SCP performs better than Base consistently across all the matrix sizes evaluated. Table 5 summarizes the performance improvement of SCP over Base. With $NT_C = 2$, SCP performs slightly better than Base by 1.7% to 1.8%. The performance gap becomes larger with $NT_C = 4$, ranging from 2.75% to 6.91%. The largest performance gain, 6.91%, which is a considerable improvement, is observed under the maximum parallelism with $NT = 64$.

In Figures 5 and 6, the performance drops slightly as the matrix size increases when $NT \geq 16$. Let us explain the reason behind this. There is a synchronization point at the end of each iteration of the loop at layer 2 (Figure 2). This synchronization is an all-to-all communication, whose overhead is proportional to NT^2 . This overhead is small when NT is 4 or 8 but increases quickly as NT increases. In addition, the performance also drops when matrix sizes are multiples of 1,024, as GEMM suffers from more memory contention at these matrix sizes. To understand this better, we have profiled the GEMM execution for the matrix sizes 3,968 and 4,096 under $NT_C = 4$ and $NC = 4$. The measured bandwidth for packing A (B) is 1.44 (0.89) GB/s for the matrix size 3,968 but drops significantly to 0.81 (0.44) GB/s for the matrix size 4,096. As the memory banks in the Phytium 2000+ processor are accessed in a nonbalanced manner, the tiling factors M_c , N_c , and K_c selected make it more likely for GEMM to suffer from memory contention at matrix sizes that are multiples of 1,024.

4.3 Cache Miss Rate Analysis

To understand why SCP outperforms Base and why SCP is effective in reducing interthread cache conflicts, we use the PAPI profiling tool [26] to analyze the cache behavior of GEMM. A total of four cache-related hardware events are counted, which are listed and briefly described in Table 6. Based on these hardware events, the miss rates of L1 and L2 caches can be estimated as $\frac{L1M}{L1M+L1H}$ and $\frac{L2M}{L2M+L2H}$, respectively. In this analysis, the hardware events are measured in a per-thread manner and all the results are the average measurements over all the threads.

Figure 7 shows the miss rates of L1 and L2 data caches. We only give the results with $NT_C = 4$ as SCP achieves the largest improvement with full cache sharing. Table 7 summarizes their average reductions across the matrix sizes used. Under all the parallelization configurations, both L1 and L2 cache miss rates are reduced under SCP, by roughly 13% for L1 and nearly 10% for L2.

For the Phytium 2000+ processor whose L1 caches are private, only L2 caches are specially handled by SCP. However, the miss rates for both L1 and L2 caches are reduced effectively by SCP.

Fig. 7. Cache miss rates with $NT_C = 4$.

While a reduction in the L2 miss rate is expected, why is the L1 miss rate also reduced? In this processor, every L2 cache is inclusive. The conflict misses for the L2 cache not only cause the data to be evicted from L2 but also invalidate the evicted data's copy in all its upper-level L1 caches, resulting in more L1 cache misses. In general, the conflict misses for an inclusive cache can affect

Table 7. Reduction of Cache Miss Rates with $NT_C = 4$

	$NT = 4$	$NT = 8$	$NT = 16$	$NT = 32$	$NT = 64$
L1	13.87	13.75	13.54	13.84	14.21
L2	10.00	10.00	8.27	8.56	7.62

Fig. 8. Packing workload (B_2) of thread T_1 with $NT_C = 4$ and $NC = 4$.

the performance of all its upper-level caches. As a result, improving the miss rate of a shared L2 cache also improves the miss rates of all its upper-level L1 caches.

Our results show that SCP can effectively reduce the conflict misses in shared L2 caches. In addition, the performance benefit thus obtained also propagates to their upper-level caches too.

4.4 Privatization of Shared Matrix

In the previous evaluations, SCP only set-partitions the shared caches for A_2 matrices, which are private to threads. The shared B_2 matrix is still packed in the conventional way-partitioning style. As mentioned in Section 3.1, set-partitioning can also be applied to B_2 after B_2 is privatized. In this section, we discuss and evaluate this privatization alternative.

Privatization incurs more packing overhead because the packing of B_2 must be done redundantly. Figure 8 shows three packing strategies for B_2 in a parallelization configuration specified by $NT_C = 4$ and $NC = 4$. In each case, the entire packing workload of B_2 is divided into 16 tasks. So only the packing workload of one thread, T_1 , which is highlighted by the z-curves (as shown), is illustrated. In the conventional packing approach (Figure 8(a)), each thread takes a single task. In full privatization (Figure 8(b)), all tasks are replicated to all the threads. As a result, the packing overhead grows in proportion to NT , which is unacceptable if GEMM is highly parallelized, e.g., when $NT = 64$. In partial privatization (Figure 8(c)), which is an alternative to full privatization, privatization only occurs inside a cluster so that the tasks for one cluster need not be replicated in the other clusters. In this case, the extra packing overhead is bounded by NT_C . For example, on Phytium 2000+, the overhead of packing B_2 in partial privatization is limited to 4 times of that in the conventional packing approach.

Fig. 9. Performance of partial privatization with $NT_C = 4$ and $NC = 4$.

Figure 9 shows the results for partial privatization with $NT_C = 4$ and $NC = 4$. The results for the other configurations are similar. For comparison purposes, SCP with and without privatization, denoted as SCP-P and SCP, respectively, are both given in Figure 9. Figure 9(a) shows the L1 and L2 cache miss rates, which are slightly lower with privatization (SCP-P) than without (SCP). Figure 9(b) shows the average thread efficiency. Despite its lower L1 and L2 cache miss rates, privatization (SCP-P) suffers a performance loss (relative to SCP) due to the extra packing overhead introduced by privatization. Figure 9(c) compares the three types of major overheads in percentage terms over the total execution time, including (1) packing of A_2 , (2) packing of B_2 , and (3) synchronization. The solid and dashed horizontal lines represent the average values of the total overheads across the matrix sizes under SCP and SCP-P, respectively. While behaving similarly as SCP in (1) and (3), SCP-P is worse than SCP in (2). To understand this further, Table 8 gives a breakdown of each's execution time in terms of (1) through (3) and the computation time of GEBP. As the overhead for packing B_2 is multiplied by a factor of $NT_C = 4$, the total overhead has thus increased from 2.35% (under SCP) to 4.72% (under SCP-P).

Our results demonstrate that privatizing the shared matrix B_2 can further reduce interthread cache conflicts. However, the improvement in cache performance is small. As B_2 only occupies a small portion of the shared L2 cache during GEMM execution, this marginal improvement is not sufficient to balance the extra packing overhead introduced by privatization.

Table 8. Execution Time Breakdown
with $NT_C = 4$ and $NC = 4$

	Sync	PackA	PackB	GEBP
SCP	1.01	0.53	0.80	97.65
SCP-P	1.00	0.52	3.20	95.28

5 RELATED WORK

The idea that all level 3 BLAS operations can be built on top of a high-performance GEMM implementation was first proposed in [10, 11]. The GotoBLAS library [8] and its successor OpenBLAS [39] are instantiated based on this insight. So optimizing GEMM has always been the central task in developing dense linear algebra software. The GEMM optimization has two aspects. One is developing fast computation kernels and the other is choosing a proper overall workload-partitioning strategy to optimize memory access.

There are several approaches for obtaining optimized GEMM kernels, yielding different tradeoffs between performance and portability. In GotoBLAS [8], OpenBLAS [39], and BLIS [27], the kernel component of GEMM is written by domain experts in assembly. ATLAS [31] adopts the auto-tuning method to automatically generate kernels with different parameters in C and find the best-performing one by running them on the actual computing system. POET [34–36] and AUGEM [30] use a directive-based programming approach. Given a GEMM kernel in C, POET inserts annotations into the C code to direct source-to-source compiler transformations. 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. Both the auto-tuning and directive-based programming approaches rely on the compiler to transform kernels in C to machine instructions. Kernel performance can be improved with compiler techniques, including SIMD vectorization [6, 16, 22, 40, 41], polyhedral optimization [3, 13], and loop tiling [15, 24, 32]. Recently, POCA [25] has leveraged a wide range of architecture-specific abstractions available in the LLVM compiler infrastructure and proposed a series of domain-specific yet architecture-independent optimizations to obtain high-performance kernels in a portable way.

The overall workload-partitioning strategy used in GEMM is mainly concerned with choosing tiling factors, loop orders, and parallelization techniques. The tiling factors M_r , N_r , M_c , N_c , and K_c are essential to GEMM performance. ATLAS [31] relies on auto-tuning to determine optimal values for these factors. Analytic techniques [12, 20, 37] can also be used instead of auto-tuning. These analytic methods generally take into consideration the cache capacity and associativity but assume an LRU replacement policy. As far as we know, this article is the first to address the problem of cache sharing with non-LRU replacement policies, and SCP serves as the first work to solve this problem. In [7], a detailed discussion on choosing different loop orders is given. GEMM is usually parallelized only at layer 3 (the ii loop), but all loops along the M and N matrix dimensions are potentially parallelizable. BLIS [23] allows developers to specify a sophisticated configuration so that more than one loop at layers 1, 3, 4, and 5 can be simultaneously parallelized. This nested parallelization is well suited to complex architecture features like multisojects and hyperthreading.

As GEMM is highly optimized and parallelized, other higher-level dense linear algebra operations, such as Cholesky, LU, and QR factorization, can enjoy its performance benefits by building themselves on top of GEMM. In other words, this programming pattern in dense linear algebra software harnesses a fork-and-join-style parallelism. To reduce synchronization overhead, the PLASMA project [4, 14, 33] makes use of a task-driven parallelization paradigm, in which dense

linear algorithms are formulated in terms of directed acyclic graphs (DAGs) of fine-grained tasks. The tasks are dynamically scheduled based on their data dependencies (represented by the graph edges). The basic idea behind SCP also applies to such DAG-based parallelization. Input matrices of different tasks can be packed into different sets of a shared cache. As a result, the tasks can be scheduled based on the cache sets in which the input matrices reside, so that the tasks whose input matrices reside in the same cache set are scheduled to the same core.

SCP is a cache partitioning technique. While SCP is specific to GEMM running on architectures with non-LRU caches, there are general cache partitioning techniques [9, 18, 21] that focus on estimating the interference across the multiple applications sharing the same cache and improving the overall system performance. In these research efforts, either hardware modifications are required or some sophisticated cache replacement policies are needed. In contrast, SCP is a pure software approach that works for commercial processors.

6 CONCLUSION

In this article, we present a shared cache partitioning method, SCP, to reduce interthread cache conflicts in GEMM programs on architectures with shared non-LRU caches. The basic idea is to partition a shared cache into physically disjoint sets and assign different sets to different threads. Our experimental results show that SCP can effectively reduce the interthread cache misses for a shared cache (and its upper-level caches), resulting in a considerable improvement in the GEMM performance in a multithreaded context.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Wanqing Chi and Prof. Ruibo Wang for providing a supported Linux kernel used in our SCP evaluation.

REFERENCES

- [1] 2018. The Phytium processor family. Retrieved from <http://www.phytium.com.cn>.
- [2] November 2017. Top500 supercomputer sites. Retrieved from <https://www.top500.org>.
- [3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08)*. 101–113.
- [4] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (2009), 38–53.
- [5] Huimin Cui, Lei Wang, Jingling Xue, Yang Yang, and Xiaobing Feng. 2011. Automatic library generation for BLAS3 on GPUs. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS'11)*. 255–265.
- [6] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'04)*. 82–93.
- [7] Kazushige Goto and Robert A. Van De Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Software* 34, 3 (2008), 1–25.
- [8] Kazushige Goto and Robert Van De Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.* 35, 1 (July 2008), 4:1–4:14.
- [9] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr., and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'08)*. 208–219.
- [10] Bo Kågström, Per Ling, and Charles van Loan. 1998. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.* 24, 3 (Sep. 1998), 268–302.
- [11] Bo Kågström and Charles van Loan. 1998. Algorithm 784: GEMM-based level 3 BLAS: Portability and optimization issues. *ACM Trans. Math. Softw.* 24, 3 (Sep. 1998), 303–316.
- [12] Vasilios Kelefouras, Angeliki Kritikakou, and Costas Goutis. 2014. A matrix-matrix multiplication methodology for single/multi-core architectures using SIMD. *J. Supercomput.* 68, 3 (Jan. 2014), 1418–1440.

- [13] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'13)*. 127–138.
- [14] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. 2010. Scheduling dense linear algebra operations on multicore processors. *Concurrency Comput. Pract. Exper.* 22, 1 (2010), 15–44.
- [15] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*. 63–74.
- [16] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*. 145–156.
- [17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.* 5, 3 (Sep. 1979), 308–323.
- [18] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the International Conference on High-Performance Computer Architecture (HPCA'08)*. 367–378.
- [19] Jie Liu, Lihua Chi, Chunye Gong, Han Xu, Jie Jiang, Yihui Yan, and Qingfeng Hu. 2012. High-performance matrix multiply on a massively multithreaded Fiteng1000 processor. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, 2012*. 166–176.
- [20] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Softw.* 43, 2 (Aug. 2016), 12:1–12:18.
- [21] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO'06)*. 423–432.
- [22] I. Rosen, D. Nuzman, and A. Zaks. 2007. Loop-aware SLP in GCC. In *GCC Developers' Summit*. 131–142.
- [23] Tyler M. Smith, Robert A. VandeGeijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'14)*. 1049–1059.
- [24] Daniele G. Spampinato and Markus Püschel. 2014. A basic linear algebra compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'14)*. 23:23–23:32.
- [25] Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic generation of fast BLAS3-GEMM: A portable compiler approach. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'17)*. 122–133.
- [26] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, Berlin, 157–173.
- [27] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.* 41, 3 (Jun. 2015), 14:1–14:33.
- [28] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the Conference on High Performance Computing (SC'08)*. 31:1–31:11.
- [29] Feng Wang, Hao Jiang, Ke Zuo, Xing Su, Jingling Xue, and Canqun Yang. 2015. Design and implementation of a highly efficient DGEMM for 64-Bit ARMv8 multi-core processors. In *Proceedings of the International Conference on Parallel Processing (ICPP'15)*. 200–209.
- [30] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, 25:1–25:12.
- [31] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the Conference on Supercomputing (SC'98)*. 1–27.
- [32] Jingling Xue. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Boston.
- [33] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. 2017. Porting the PLASMA numerical library to the OpenMP standard. *Int. J. Parallel Program.* 45, 3 (2017), 612–633.
- [34] Qing Yi. 2011. Automated programmable control and parameterization of compiler optimizations. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'11)*. 97–106.
- [35] Qing Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. 2007. POET: Parameterized optimizations for empirical tuning. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'07)*. 1–8.
- [36] Qing Yi, Qian Wang, and Huimin Cui. 2014. Specializing compiler optimizations through programmable composition for dense matrix computations. In *Proceedings of the International Symposium on Microarchitecture (MICRO'14)*. 596–608.
- [37] Kamen Yotov, Xiaoming Li, Gang Ren, María Jesús Garzarán, David A. Padua, Keshav Pingali, and Paul Stodghill. 2005. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE Special Issue on Program Generation, Optimization, and Adaptation* 93, 2 (2005), 358–386.

- [38] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. 2016. The BLIS framework: Experiments in portability. *ACM Trans. Math. Softw.* 42, 2 (Jun. 2016), 12:1–12:19.
- [39] Xianyi Zhang, Qian Wang, and Yunquan Zhang. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'12)*. 684–691.
- [40] Hao Zhou and Jingling Xue. 2016. A compiler approach for exploiting partial SIMD parallelism. *ACM Trans. Archit. Code Optim.* 13, 1, Article 11 (March 2016), 11:1–11:26 pages.
- [41] Hao Zhou and Jingling Xue. 2016. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'16)*. 59–69.

Received May 2018; revised August 2018; accepted August 2018