Design and Implementation of a Highly Efficient DGEMM for 64-bit ARMv8 Multi-Core Processors

Feng Wang*, Hao Jiang*, Ke Zuo*, Xing Su*, Jingling Xue† and Canqun Yang‡
*School of Computer Science, National University of Defense Technology, Changsha 410072, China
†School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia
‡Science and Technology on Parallel and Distributed Processing Laboratory,
National University of Defense Technology, Changsha 410072, China
Email: {fengwang,haojiang,zuoke,xingsu,canqun}@nudt.edu.cn, jingling@cse.unsw.edu.au

***Abstract*—This paper presents the design and implementation of a highly efficient Double-precision General Matrix Multiplication (DGEMM) based on OpenBLAS for 64-bit ARMv8 eight-core processors. We adopt a theory-guided approach by first developing a performance model for this architecture and then using it to guide our exploration. The key enabler for a highly efficient DGEMM is a highly-optimized inner kernel GEBP developed in assembly language. We have obtained GEBP by (1) maximizing its compute-to-memory access ratios across all levels of the memory hierarchy in the ARMv8 architecture with its performance-critical block sizes being determined analytically, and (2) optimizing its computations through exploiting loop unrolling, instruction scheduling and software-implemented register rotation and taking advantage of A64 instructions to support efficient FMA operations, data transfers and prefetching.**

We have compared our DGEMM implemented in OpenBLAS with another implemented in ATLAS (also in terms of a highly-optimized GEBP in assembly). Our implementation outperforms the one in ALTAS by improving the peak performance (efficiency) of DGEMM from 3.88 Gflops (80.9%) to 4.19 Gflops (87.2%) on one core and from 30.4 Gflops (79.2%) to 32.7 Gflops (85.3%) on eight cores. These results translate into substantial performance (efficiency) improvements by 7.79% on one core and 7.70% on eight cores. In addition, the efficiency of our implementation on one core is very close to the theoretical upper bound 91.5% obtained from micro-benchmarking. Our parallel implementation achieves good performance and scalability under varying thread counts across a range of matrix sizes evaluated.

***Keywords*—64-bit ARMv8 processor, register rotation, prefetching, blocking, DGEMM, BLAS, compute-to-memory access ratio**

I. Introduction

Recently, ARM-based SoCs have a rapid evolution. The promising qualities, such as competitive performance and energy efficiency, make ARM-based SoCs the candidates for the next generation HPC systems [1], [2]. For example, supported by the Mont-Blanc project, the Barcelona Supercomputing Center has built Tibidabo, the world's first ARM-based HPC cluster [3]. Recently, the new 64-bit ARMv8 instruction set architecture (ISA) has included a number of new features, including a greater addressing range, increased availability of larger registers, double-precision floating-point values supported by its NEON vector unit, and FMA (fused multiply-add) SIMD instructions. Therefore, there is now increasing interest in building HPC systems with ARMv8-based SoCs.

Meanwhile, dense matrix operations play an important role in scientific and engineering computing. Basic Linear Algebra Subprogram (BLAS) prescribes an application programming interface standard for publishing libraries, which is classified as Level-1, Level-2 and Level-3 BLAS for vector-vector, vector-matrix and matrix-matrix computations, respectively. To achieve high performance on a variety of hardware platforms, CPU vendors and some HPC researchers have developed a variety of BLAS libraries, including Intel's MKL, AMD's ACML, IBM's ESSL, ATLAS [4], GotoBLAS [5], [6], OpenBLAS [7] and BLIS [8]. NVIDIA has also provided CUBLAS on its own GPUs. In addition, there has also been a lot of work on optimizing matrix-related applications [9], [10]. For Level-3 BLAS, the most commonly used matrix-matrix computations can be implemented as a general matrix multiplication. In the HPC arena, as the core part of the LINPACK benchmark, Double-precision General Matrix Multiplication (DGEMM) has been an important kernel for measuring the potential performance of a HPC platform.

In this paper, we focus on designing and implementing a highly efficient DGEMM based on OpenBLAS for 64-bit ARMv8 eight-core processors. We adopt a theory-guided approach by first developing a performance model for this architecture and then using it to guide our exploration. Our model reveals clearly that optimizing the peak performance (efficiency) of DGEMM requires its compute-to-memory access ratios to be maximized across all levels of the memory hierarchy. In addition, our model also allows us to bound from below the performance of a DGEMM implementation.

Guided by our performance model, we obtain a highly efficient DGEMM for the ARMv8 architecture by developing systematically a highly-optimized inner kernel, GEBP, in assembly language. To boost the performance of GEBP, the main challenge lies in choosing the right register block size for its innermost register kernel. We make such a choice analytically with the goal of maximizing its compute-to-memory access ratio from the L1 data cache to registers. In order to realize the optimal ratio thus found, we optimize the operations in the register kernel by (1) exploiting loop unrolling, instruction scheduling and software-implemented register rotation and (2) taking advantage of A64 instructions to support efficient FMA operations, data transfer and data prefetching. Subsequently, we optimize GEBP to maximize its compute-to-memory access ratios across all three levels of cache memories by determining analytically the remaining performance-critical block sizes

used by GEBP. We perform this optimization by considering their set associativities and replacement policies.

Recently, supported by AMCC (Applied Micro Circuits Corporation), which produces X-Gene, the world's first ARMv8 64-bit server on a chip solution, Nuechterlein and Whaley [11] implemented DGEMM in ATLAS for the ARMv8 architecture by optimizing its inner kernel GEBP also in assembly. Our DGEMM implemented in OpenBLAS outperforms theirs in ALTAS by improving the peak performance (efficiency) of DGEMM from 3.88 Gflops (80.9%) to 4.19 Gflops (87.2%) on one core and from 30.4 Gflops (79.2%) to 32.7 Gflops (85.3%) on eight cores. These results, which translate into performance (efficiency) improvements by 7.79% on one core and 7.70% on eight cores, are significant as DGEMM is the core of the LINPACK benchmark. In addition, the efficiency of our DGEMM implementation on one core is very close to the theoretical upper bound 91.5% obtained from micro-benchmarking. Our parallel DGEMM implementation achieves good performance and scalability under varying thread counts across a range of matrix sizes evaluated.

The rest of this paper is organized as follows. Section II reviews the 64-bit ARMv8 architecture and introduces the blocking and packing algorithms used in implementing DGEMM by OpenBLAS. Section III introduces a performance model crucial for guiding the development of our DGEMM implementation. Section IV describes our DGEMM implementation by focusing on optimizing its inner kernel GEBP. Section V presents and analyzes our experimental results. Section VI concludes the paper and describes some future work.

II. Background

A. The 64-bit ARMv8 Multi-Core Processor

Figure 1 depicts a 64-bit ARMv8 eight-core processor. Each core has a 32 KB L1 instruction cache and a 32 KB L1 data cache. The two neighboring cores constitute a so-called dual-core module. The two cores in the same module share a 256 KB L2 cache, and four modules (with eight cores) share a 8 MB L3 cache. Each core has one floating-point computing pipeline supporting FMA and runs on 2.4 GHz, offering a peak performance of 4.8 Gflops.

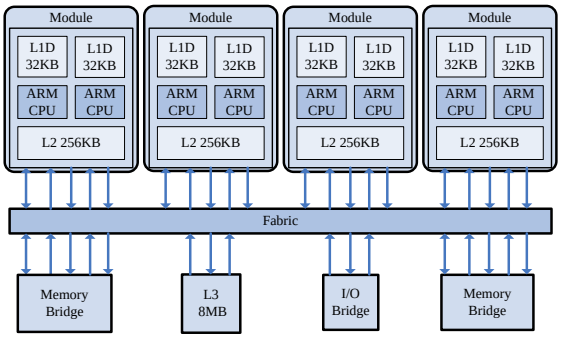

Fig. 1. Block diagram of the 64-bit ARMv8 eight-core processor.

Every core is a single-threaded four-issue superscalar design with out-of-order execution. In every core, the 64-bit ARMv8 architecture (aarch64) defines 32 128-bit floating-point registers, v0 – v31, which can be used for floating-point vector calculations. As shown in Figure 1, the four modules, the L3 cache, the two memory bridges and the I/O bridge are all connected to a cache-coherent fabric.

B. Open-Source BLAS implementations

Several well-known open-source BLAS implementations are netlib BLAS, ATLAS [4], GotoBLAS [5], [6], OpenBLAS [7] and BLIS [8]. The netlib BLAS is the original reference implementation, which does not take advantage of memory hierarchies and thus performs poorly on modern computer architectures. ATLAS relies on auto-tuning to improve cache performance, making it well-suited for generating BLAS libraries on a variety of platforms. OpenBLAS, an extension of the widely used but discontinued GotoBLAS, offers competitive performance on a range of architectures. In OpenBLAS, its inner kernel, known as GEBP (a basic unit of computation), is often implemented in assembly. BLIS, a new framework for producing rapid instantiations of BLAS, can be viewed as a re-implementation of GotoBLAS. BLIS also takes a layered approach as in GotoBLAS to increase code reuse, and breaks GEBP down into a double loop over a so-called micro-kernel, thereby facilitating optimization of level-3 BLAS.

C. Overview of DGEMM in OpenBLAS

DGEMM computes $C := \alpha \times AB + \beta \times C$, where C, A and B are matrices of sizes $M \times N$, $M \times K$ and $K \times N$, respectively. Without loss of generality, we assume that $\alpha = \beta = 1$ and thus simplify DGEMM to $C+ = AB$. We assume further that a matrix is always stored in column-major order.

Figure 2 illustrates the DGEMM algorithm by Goto [5], [6], including its multiple layers for blocking (to maximize cache performance) and packing (to enable data to be moved efficiently to the registers). In this work, we obtain a highly efficient DGEMM for the ARMv8 architecture through developing a highly-optimized GEBP in assembly systematically with its performance-critical block sizes selected analytically.

The outermost loop at layer 1 partitions C and B into column panels of sizes $M \times n_c$ and $K \times n_c$, respectively. The next loop at layer 2 partitions the $M \times K$ matrix A and a $K \times n_c$ submatrix of B into column panels of size $M \times k_c$ and row panels of size $k_c \times n_c$, respectively. C is updated as a sequence of rank-k_c updates, meaning that DGEMM consists of several general panel-panel multiplications (GEPP). Then each $M \times k_c$ panel of A is partitioned into $m_c \times k_c$ blocks, by the third loop at layer 3. In essence, GEPP is decomposed into multiple calls to block-panel multiplication (GEBP). Since there is a L3 cache in the 64-bit ARMv8 architecture, we assume that a $k_c \times n_c$ panel of B will always reside fully in the L3 cache [12].

GEBP, the *inner kernel* handled at layer 4, updates an $m_c \times n_c$ panel of C as a product of an $m_c \times k_c$ block of A and a $k_c \times n_c$ panel of B. To ensure consecutive accesses, OpenBLAS packs a block (panel) of A (B) into contiguous buffers.

As illustrated in Figure 3, packing A involves extracting a series of slivers (sub-blocks) of size $m_r \times k_c$ from an $m_c \times k_c$ block of A and organizing these slivers in the L2 cache. Similarly, packing B amounts to extracting a series of slivers of size $k_c \times n_r$ from a $k_c \times n_c$ panel of B and organizing these slivers in the L3 cache. To amortize the packing cost, each $k_c \times n_r$ sliver is moved into the L1 cache one by one.

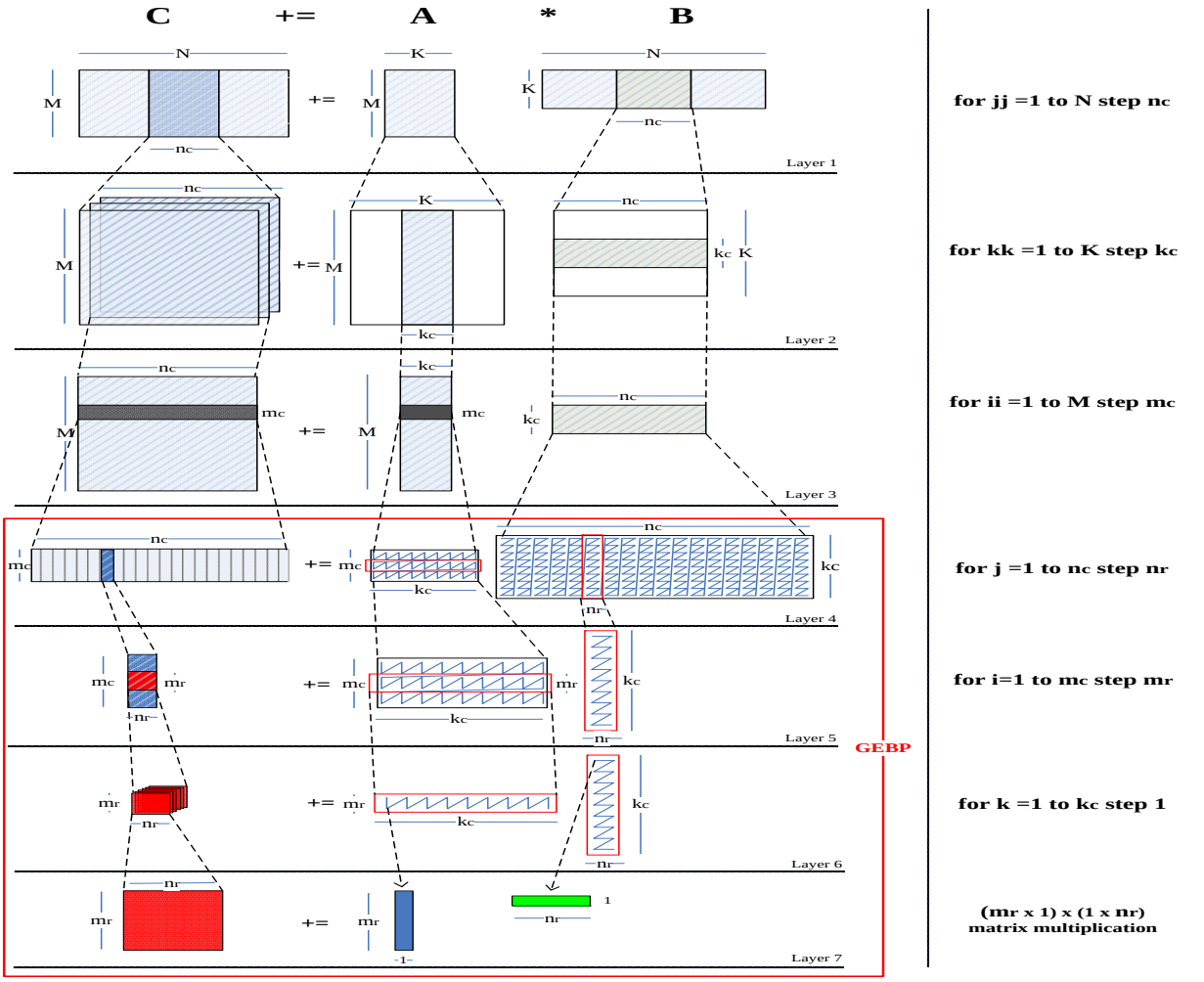

Fig. 2. Blocking and packing algorithms used in implementing DGEMM in GotoBLAS, where GEBP is the inner kernel highlighted inside a red box.

Let us return to GEBP in Figure 2. GEBP includes a double loop formed by the fifth loop at layer 5 and the sixth loop at layer 6. This double loop is responsible for implementing the packing process discussed above. In particular, the fifth loop partitions a $k_c \times n_c$ panel of B into $k_c \times n_r$ slivers and the sixth loop partitions an $m_c \times k_c$ block of A into $m_r \times k_c$ slivers. The computations at layers 5 and 6 are referred to as GEBS and GESS, respectively. GESS, also known as the *micro kernel* in BLIS, performs a sequence of rank-1 updates of an $m_r \times n_r$ sub-block of C using an $m_r \times 1$ column sub-sliver of A and a $1 \times n_r$ row sub-sliver of B, where m_r and n_r are selected to be the register block sizes for the micro kernel.

When performing each rank-1 update at layer 7, an $m_r \times n_r$ sub-block of C, two $m_r \times 1$ column sub-slivers of A, and two $1 \times n_r$ row sub-slivers of B will reside in the registers, as illustrated in Figure 3. Here, one sub-sliver of A (B) is for the current A (B) elements being used, and the other one is for the forthcoming A (B) elements required in the next rank-1 update. This last layer is referred to as the *register kernel*.

III. Performance Modeling

What is the theoretical basis that underpins the development of a highly efficient DGEMM for the 64-bit ARMv8 architecture? Our starting point is a general-purpose performance model with respect to the ratio of CPU speed to memory speed, i.e., the compute-to-memory access ratio. This model reveals clearly that optimizing the peak performance (efficiency) of DGEMM requires its compute-to-memory ratios to be maximized across all levels of the memory hierarchy for this architecture. In addition, our model also gives rise to a lower bound for the performance of a DGEMM implementation.

Given the memory hierarchy in the ARMv8 architecture shown in Figure 4, care must be taken to handle effectively the amount of data movements from memory to registers relative to the amount of computations performed on the data. We assume that a fixed cost μ is needed to perform a single operation, without differentiating among addition, subtraction and multiplication. For communication, we also assume a fixed cost ν_{ij} to move a word from level i to level j and a fixed cost η_{ij} to move a message from level i to level j in the memory hierarchy [13]. Here, a word is one floating-point value and a message denotes a cache line consisting of several consecutive words. Thus, ν_{ij} can be regarded as the inverse of bandwidth and η_{ij} as latency. If we ignore any overlap between computation and communication for now, then the execution

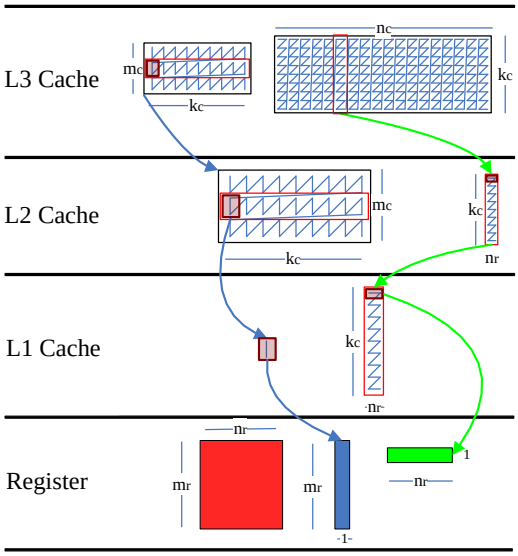

Fig. 3. Packed data storage for GEBP in GotoBLAS.

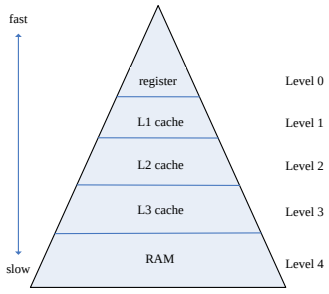

Fig. 4. The memory hierarchy in the ARMv8 architecture.

time, denoted T, of a program can be estimated as:

$$T = F\mu + \sum_i \sum_j W_{ij}\nu_{ij} + \sum_i \sum_j M_{ij}\eta_{ij} \quad (1)$$

where F, W_{ij}, and M_{ij} represent the number of operations, words, and messages, respectively. For example, W_{10} denotes the number of words loaded from the L1 cache to registers.

Given the packed data stored contiguously in slow memory, as shown at layer 4 in Figure 2, we assume that all the words in a message are needed in consecutive computations, i.e., that they can be read or written together as one message (one cache line). Hence, the ratio of the number of moved messages to that of moved words is nearly a constant: $\sum_i \sum_j M_{ij} \simeq \kappa \sum_i \sum_j W_{ij}$. Since $\nu_{ij} \geq 0$ and $\eta_{ij} \geq 0$, we have:

$$T \leq F\mu + (1+\kappa) \sum_i \sum_j W_{ij} \times (\sum_i \sum_j \nu_{ij} + \sum_i \sum_j \eta_{ij})$$

For convenience, we let $\pi = \sum_i \sum_j \nu_{ij} + \sum_i \sum_j \eta_{ij}$ and $W = \sum_i \sum_j W_{ij}$. Then the compute-to-memory access ratio, denoted γ, for the program can be expressed as:

$$\gamma = \frac{F}{W} = \frac{F}{\sum_i \sum_j W_{ij}} \quad (2)$$

Then we can obtain:

$$T \leq F\mu + (1+\kappa)W\pi \quad (3)$$

Since overlapping computation and communication is an important and necessary optimization for improving performance, we propose a so-called overlapping factor as a function $\psi(\gamma)$ of γ. Using this overlapping factor, we can refine (3) into:

$$T_{opt} \leq F\mu + (1+\kappa)W\pi\psi(\gamma) \quad (4)$$

Note that $\psi(\gamma) \to 1$ if $\gamma \to 0$ and $\psi(\gamma) \to 0$ if $\gamma \to +\infty$. In addition, $\psi(\gamma)$ is typically a monotonically decreasing function with respect to γ. By (2), we have:

$$T_{opt} \leq F(\mu + (1+\kappa)\pi\frac{\psi(\gamma)}{\gamma}) \quad (5)$$

Finally, we obtain the following lower bound on the performance of a DGEMM implementation:

$$Perf_{opt} = \frac{F}{T_{opt}} \geq \frac{1}{(\mu + (1+\kappa)\pi\frac{\psi(\gamma)}{\gamma})} \quad (6)$$

which indicates clearly that larger compute-to-memory ratios γ lead always to better peak performance (efficiency).

IV. Fast Implementation

Based on our performance model, we obtain a highly efficient implementation of DGEMM for the 64-bit ARMv8 architecture by developing systematically a highly-optimized GEBP kernel in assembly. As illustrated in Figure 2, GEBP comprises layers 4 – 7. Its development involves implementing each rank-1 update performed at layer 7 (referred to as the register kernel in Section II-C) and determining various block sizes used across the four layers. We will describe our GEBP implementation inside out from layer 7 (the register kernel) to layer 4, i.e., across the four levels of the memory hierarchy in the ARMv8 architecture, starting from the fastest to the slowest. Thus, some block sizes determined at a level will be used later to determine other block sizes at a lower level.

When developing a highly-optimized GEBP, the main challenge lies in choosing the right register block size for its register kernel. We make such a choice analytically with the goal of maximizing its compute-to-memory access ratio from the L1 data cache to registers. In order to realize the optimal ratio thus found, we optimize the operations in the register kernel by (1) exploiting loop unrolling, instruction scheduling and software-implemented register rotation and (2) taking advantage of A64 instructions to support efficient FMA operations, data transfer and data prefetching. Subsequently, we optimize GEBP by maximizing its compute-to-memory access ratios across all three levels of cache memories. We do so by determining analytically the other block sizes used, considering set associativities and replacement policies.

In Section IV-A, we describe how to determine the register block size $m_r \times n_r$ for the register kernel, together with associated optimizations. In Section IV-B, we find the block sizes k_c, m_c and n_c corresponding to layers 6, 5 and 4, respectively. As shown in Figure 3, k_c, m_c and n_c are determined by the L1, L2 and L3 caches used, respectively. In addition, we also determine how to insert prefetching instructions to prefetch data into the L1 data cache in order to accelerate further the operations in the register kernel. In Section IV-C, we adjust the block sizes m_c and n_c when moving from a serial to a parallel implementation due to cache sharing.

A. Register Blocking

Let us focus on the computation and data movement happening in the register kernel at layer 7 (Figures 2 and 3). We are concerned with maximizing the compute-to-memory access ratio from the L1 cache to registers during its execution. When the current $2m_r n_r$ flops are being performed, an $m_r \times 1$ column sub-sliver of A and a $1 \times n_r$ row sub-sliver of B are being loaded from L1 cache to registers. Here, a $1 \times n_r$ row sub-sliver of B always resides in the L1 cache and an $m_r \times 1$ column sub-sliver of A also fits into the L1 cache by prefetching data effectively. The $m_r n_r$ elements in a sub-block of C always reside in the registers to participate in consecutive operations. Hence, the compute-to-memory access ratio is:

$$\frac{2m_r n_r}{(m_r \times 1)_{L1\to R} + (n_r \times 1)_{L1\to R}} \tag{7}$$

where the subscript $L1 \to R$ indicates that the data are moved from the L1 cache to registers. So the optimization problem is:

$$\max \quad \gamma = \frac{2}{\frac{1}{n_r} + \frac{1}{m_r}} \tag{8}$$

subject to the following constraint:

$$(m_r n_r + 2m_r + 2n_r) \times element\_size \le (n_f + n_{rf}) \times p_f \tag{9}$$

Here, the register block size $m_r \times n_r$ is determined by the number of registers available, $element\_size$ is the size of an element of an matrix in bytes, e.g., 8 bytes in double-precision, n_f is the number of floating-point registers available, n_{rf} is the number of these registers that can be reused for register preloading, and finally, p_f is the size of a floating-point register in bytes, e.g., 16 bytes in the 64-bit ARMv8 architecture. Note that n_{rf} must satisfy the following constraint:

$$0 \le n_{rf} \times p_f \le (m_r + n_r) \times element\_size. \tag{10}$$

From this optimization formulation, it is obvious that the cost of loading the data into registers can be amortized most effectively when $m_r \approx n_r$. However, there are some practical issues that influence the choice of m_r and n_r as well. For example, in the 64-bit ARMv8 architecture, each floating-point register can hold two double-precision values. As a result, m_r and n_r are preferably to be set as multiples of 2:

$$m_r = 2i, \ n_r = 2j, \ i = 1, 2, \cdots, \ j = 1, 2, \cdots \tag{11}$$

In our setting, we have $n_f = 32$, $p_f = 16$ and $element\_size = 8$. Based on the objective function (8) and its three associated constraints (9) – (11), we plot the surface of the register kernel's compute-to-memory ratio in terms of m_r and n_{rf} in Figure 5. In order to obtain the highest ratio 6.857, it suffices to set the number of reused floating-point registers as $n_{rf} = 6$ with the register block size $m_r \times n_r$ being selected as either 8×6 or 6×8. Since a cache line has 64 bytes, it will be convenient to prefetch the elements of A when the register block size is selected as $m_r \times n_r = 8 \times 6$, as will be discussed in detail in Section IV-B. Hence, we use 24 registers, v8 – v31, to store the 48 elements of C, and 8 registers, v0 – v7, to store the 8 elements of A and the 6 elements of$B$.

It is easy to allocate 24 registers to the 48 elements of C as each register holds two elements. However, how do we allocate the remaining 8 registers to the 8 elements of A and the 6 elements of B. A simple-minded approach is to use just 7 registers, with one to spare. In this paper, we adopt a software-implemented register rotation scheme to obtain a more efficient register kernel for the 64-bit ARMv8 architecture, which has fewer physical registers than x86 for register renaming. As a result, we start with the following optimization problem:

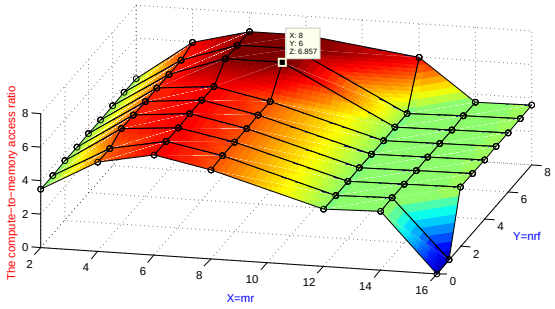

Fig. 5. Surface of the compute-to-memory access ratio for the register kernel.

$$\max_{s\in S_c} \min_{\mathrm{v}_i \in V} \quad Loc('\mathrm{R}','\mathrm{NF}', \mathrm{v}_i, s) - Loc('\mathrm{R}','\mathrm{CL}', \mathrm{v}_i, s) \tag{12}$$

where $V = \{\mathrm{v}_0, \mathrm{v}_1, \cdots, \mathrm{v}_7\}$ is the set of registers used by A and B, 'R' represents a read instruction (fmla) using v_i, 'CL' and 'NF' mean that the current value in v_i is read for the last time and the next value in the same register is read for the first time, respectively, S_c is the set of all execution orderings for the Read (fmla) instructions in the register kernel, and Loc denotes the location of an instruction in a particular ordering.

When formulating (12), we have purposedly ignored all the corresponding load instructions for defining the values used in all the registers. In an optimal solution, the distance between the instruction at $Loc('\mathrm{R}','\mathrm{CL}', \mathrm{v}_i, s)$ and the instruction at $Loc('\mathrm{R}','\mathrm{NF}', \mathrm{v}_i, s)$ is made as large as possible. In between the two instructions, there is a load instruction for writing the value used by the latter instruction into v_i. When $Loc('\mathrm{R}','\mathrm{NF}', \mathrm{v}_i, s) - Loc('\mathrm{R}','\mathrm{CL}', \mathrm{v}_i, s)$ is large, there is a good chance to position the load instruction to prevent from stalling the pipeline, so that the loaded value is already available when it is needed at $Loc('\mathrm{R}','\mathrm{NF}', \mathrm{v}_i, s)$.

TABLE I. SOFTWARE-IMPLEMENTED REGISTER-ROTATION (WITH THE REGISTERS ALLOCATED FROM $\{v_0, \ldots, v_7\}$ TO A AND B IN THE i-TH COPY OF THE LOOP BODY OF THE REGISTER KERNEL).

Array	Eight Copies of the Loop Body of the Register Kernel								
	#0	#1	#2	#3	#4	#5	#6	#7	#0
A	0	2	4	7	6	1	3	5	0
	1	3	5	0	2	4	7	6	1
	2	4	7	6	1	3	5	0	2
	3	5	0	2	4	7	6	1	3
B	4	7	6	1	3	5	0	2	4
	5	0	2	4	7	6	1	3	5
	6	1	3	5	0	2	4	7	6

To solve (12), we propose a software-implemented register rotation scheme as illustrated in Table I, which unrolls the loop governing the execution of the register kernel, i.e., the innermost loop depicted in Figure 2 by a factor of 8. Here, $\#i$ represents the i-th copy of the loop body in the original register kernel. To execute $\#i$, a total of 7 (128-bit) registers are needed to store the 8 elements of A and the 6 elements of B. To prefetch A and B used during the execution of $\#(i+1)\%8)$, another 7 registers are also needed. However, there are

only a total of 8 registers, $\{v_0, \ldots, v_7\}$, available. As a result, $n_{rf} = 6$, as discussed earlier. This means that 6 registers are reused between the two consecutive iterations, i.e., $\#i$ and $\#(i+1)\%8$, in the original register kernel. Figure 6 illustrates our register allocation scheme for #0 and #1, together with the order in which the computations are performed in each case. The optimal distance 7 from solving (12) has been found.

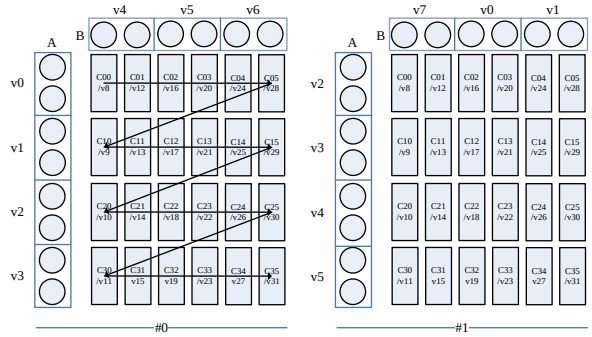

Fig. 6. 8×6 register blocking for #0 and #1.

When scheduling the instructions for each copy $\#i$ from the original register kernel, we need to consider also their WAR (write-after-read) and RAW (read-after-write) dependences. In each copy $\#i$, 24 floating-point FMA instructions (fmla) and 7 load instructions (ldr) are executed, together with one prefetching instruction (prfm). Due to register renaming, RAW must be considered carefully but WAR is not as important (as validated later). However, we must still strive to hide the latency of a load instruction to prevent it from stalling the pipeline so that the loaded value is available when it is needed. We do so by solving the following optimization problem:

$$\max_{s \in S} \min_{v_i \in V} \quad Loc('\mathrm{R}', v_i, s) - Loc('\mathrm{W}', v_i, s) \qquad (13)$$

where 'R' and 'W' represent a Read (fmla) and Write (ldr) instruction, respectively, $V = \{v_0, v_1, \cdots, v_7\}$, S is the set of all possible instruction execution orderings, and Loc denotes the location of an instruction in a particular ordering.

In each copy $\#i$, the order in which its 24 floating-point FMA instructions are executed is fixed, along the zig-zag path in Figure 6. Thus, the optimization problem (13) is reduced to one of searching for appropriate points to insert all the required load instructions. Figure 7 shows how these instructions are scheduled, with the execution order shown in row-major. The registers in green are loaded in $\#i$ while the registers in red are loaded in $\#(i-1)\%8$ (i.e., one iteration earlier in the original register kernel). We can observe that the optimal distance 9 from solving (13) has been found. Finally, Figure 8 gives a code snippet from the loop body of the unrolled register kernel with the register block size 8×6 in assembly language.

B. Cache Blocking for Locality

Let us consider GESS at layer 6 in Figure 2. The elements from a $k_c \times n_r$ silver of B are reused many times, and therefore remain in the L1 cache (Figure 3). At the same time, the elements from an $m_r \times k_c$ silver of A should be loaded from the L2 cache into the L1 cache. When the $2m_r n_r k_c$ flops are performed, the $m_r \times n_r$ elements in a sub-block of C need to be loaded to registers and the updated results will be stored

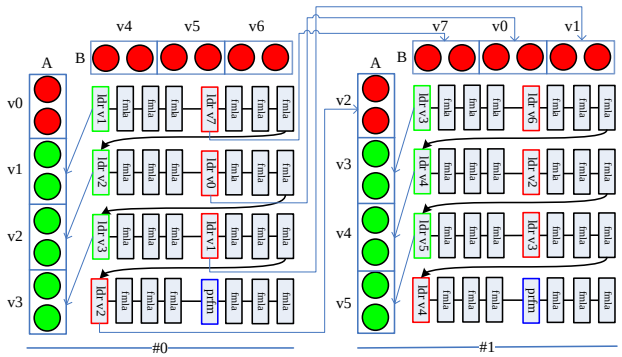

Fig. 7. Instruction scheduling with the optimal RAW dependence distance.

```
ldr     q1,[x14],#16            //ARMv8-64bit load instruction
fmla    v8.2d,  v0.2d, v4.d[0] //NEON FMA instruction
fmla    v9.2d,  v0.2d, v4.d[1]
fmla    v10.2d, v0.2d, v5.d[0]
ldr     q2,[x14], #16
fmla    v11.2d, v0.2d, v5.d[1]
fmla    v12.2d, v0.2d, v6.d[0]
fmla    v13.2d, v0.2d, v6.d[1]
ldr     q7,[x15], #16
......
prfm    PLDL1KEEP, [x14,#PREFA] //Prefetch A to L1 Cache
......
prfm    PLDL2KEEP, [x15,#PREFB] //Prefetch B to L2 Cache
......
```

Fig. 8. A code snippet from the 8×6 register kernel in assembly language.

from the registers to the memory hierarchy. Then the compute-to-memory access ratio of GESS can be estimated to be:

$$\frac{2m_r n_r k_c}{(m_r k_c)_{L2 \to L1} + (m_r k_c)_{L1 \to R} + (k_c n_r)_{L1 \to R} + (2m_r n_r)_{M \leftrightarrow R}}$$

By conducting a similar analysis, we can also express the compute-to-memory access ratio of GEBS at layer 5 as:

$$\frac{2m_c n_r k_c}{(m_c k_c)_{L2 \to L1} + (m_c k_c)_{L1 \to R} + (k_c n_r)_{L1 \to R} \lceil \frac{m_c}{m_r} \rceil + (2m_c n_r)_{M \leftrightarrow R}}$$

In practice, m_c is an integer multiple of m_r. Then the ratios of GESS and GEBS are very close and are thus simplified to:

$$\gamma = \frac{2}{\frac{2}{\mathbf{n_r}} + \frac{1}{\mathbf{m_r}} + \frac{2}{k_c}} \qquad (14)$$

where the bold-faced m_r and n_r are already fixed in Section IV-A. From (14), we can observe that if n_r is larger than m_r, then the cost of loading the elements of A from the L2 cache to the L1 cache can be more easily amortized. However, in our implementation, considering that a cache line has 64 Bytes, we have selected the register block size to be $m_r \times n_r = 8 \times 6$. As a result, each sub-sliver (8 elements) of A can be prefetched in one cache line from L2 to L1. Furthermore, it is impossible to overlap the loading of elements of C into the registers with computation. However, we can overlap the process for storing the elements of C back to memory with computation. Therefore, if m_r and n_r are known,

the ratio γ in (14) is maximized if k_c is the largest possible, in order to amortize the cost on updating the elements of C.

In our implementation, the L1 cache adopts a LRU replacement policy and a $k_c \times n_r$ silver of B is reused many times. Thus, it is easy to keep B in the L1 cache. Meanwhile, to optimize performance, at least two columns of a $m_r \times k_c$ silver of A and one $m_r \times n_r$ sub-block of C should also be loaded into the L1 cache without causing any conflict misses with the elements of B. To account for the set associativity of the L1 cache, we follow [14] by imposing the two constraints:

$$\begin{aligned} k_c \times n_r \times element\_size &\leq \frac{(assoc1 - k1) \times L1}{assoc1} \\ (m_r \times n_r + m_r \times 2) \times element\_size &\leq \frac{k1 \times L1}{assoc1} \end{aligned} \tag{15}$$

where $m_r = 8$ and $n_r = 6$ as established earlier, $element\_size = 8$, $L1 = 32$K is the size of the L1 data cache in bytes, $assoc1 = 4$ is the number of ways in the L1 cache, and $k1$ is an integer satisfying $0 < k1 < assoc1$. It is easy to see that the smaller $k1$ is, the larger k_c will be, which allows us to conclude that $k1 = 1$ and $k_c = 512$. This means that a $k_c \times n_r$ silver of B fills 3/4 of the L1 data cache.

Next, we maximize the compute-to-memory access ratio of GEBP at layer 4 in Figure 2. We assume that an $m_c \times k_c$ block of A already resides in the L2 cache and a $k_c \times n_c$ panel of B already resides in the L3 cache (Figure 3). Thus, its compute-to-memory access ratio is obtained by dividing $2m_ck_cn_c$ with $(m_ck_c)_{L2\to L1}\lceil\frac{n_c}{n_r}\rceil + (m_ck_c)_{L1\to R}\lceil\frac{n_c}{n_r}\rceil + (k_cn_c)_{L1\to R}\lceil\frac{m_c}{m_r}\rceil + (k_cn_c)_{L3\to L2} + (k_cn_c)_{L2\to L1} + (2m_cn_c)_{M\leftrightarrow R}$. Here, m_c and n_c are usually integer multiples of m_r and n_r, respectively. Proceeding similarly as before, we obtain:

$$\gamma = \frac{2}{\frac{2}{\mathbf{n_r}} + \frac{1}{\mathbf{m_r}} + \frac{2}{\mathbf{k_c}} + \frac{2}{m_c}} \tag{16}$$

where the bold-faced m_r, n_r and k_c have been determined previously. We observe that the amount of data movement $(k_cn_c)_{L3\to L2}$ can overlap with the $2m_ck_cn_c$ operations, but the data movement represented by $(k_cn_r)_{L2\to L1}$ can overlap only with the last $2m_rk_cn_r$ operations for the same $k_c \times n_r$ sliver of B involved. The larger m_c is, the better the amount of data movement $(k_cn_r)_{L3\to L2}$ can be hidden. Proceeding similarly as in the case of establishing (15), we obtain:

$$\begin{aligned} m_c \times k_c \times element\_size &\leq \frac{(assoc2 - k2) \times L2}{assoc2} \\ k_c \times n_r \times element\_size &\leq \frac{k2 \times L2}{assoc2} \end{aligned} \tag{17}$$

where $k_c = 512$ and $n_r = 6$ as obtained earlier, $L2 = 256$K is the size of L2 cache in bytes, $assoc2 = 16$ is the number of ways in the L2 cache, $k2$ is an integer satisfying $0 < k2 < assoc2$ and $element\_size = 8$. From (16), m_c is as large as possible. From (17), we can then obtain $k2 = 2$ and $m_c = 56$. Thus, an $m_c \times k_c$ block of A fills 7/8 of the L2 cache and a $k_c \times n_r$ sliver of B occupies under 1/8 of the L2 cache.

We are now ready to analyze the prefetch distances required by A and B. Since a $k_c \times n_r$ sliver of B always resides in the L1 cache when being multiplied with each $m_r \times k_c$ sliver of A, this sliver of B does not need to be prefetched. It is only necessary to prefetch the next $k_c \times n_r$ sliver of B to the L2 cache during the multiplication of the current sliver of B and the last sliver of A. In this case, the prefetch distance is set to be $\text{PRE}_B = k_c \times n_r \times element\_size = 24576$. In order for all accesses to a sub-sliver of A to hit in the L1 cache, we use a shorter distance for prefetching A: $\text{PRE}_A = \alpha_{prea} \times num\_unroll \times m_r \times element\_size = 2 \times 8 \times 8 \times 8 = 1024$.

Finally, we discuss how to choose n_c. From GEPP performed at layer 3 in Figure 2, we can see that n_c should be as large as possible in order to amortize the cost of data movement of an $m_c \times k_c$ block of A from the L3 cache to the L2 cache. Thus, we have an analogue of (15) and (17) for n_c:

$$\begin{aligned} k_c \times n_c \times element\_size &\leq \frac{(assoc3 - k3) \times L3}{assoc3} \\ m_c \times k_c \times element\_size &\leq \frac{k3 \times L3}{assoc3} \end{aligned} \tag{18}$$

where $m_c = 56$ and $k_c = 512$ as obtained earlier, $L3 = 8$M is the size of the L3 cache in bytes, $assoc3 = 16$ is the number of ways in the L3 cache and $k3$ is an integer satisfying $0 < k3 < assoc3$. It is easy to obtain that $k3 = 1$ and $n_c = 1920$, meaning that a $k_c \times n_c$ panel of B (an $m_c \times k_c$ sliver of A) occupies 15/16 (under 1/16) of the L3 cache.

C. Cache blocking for Parallelism

We discuss how to adjust the block sizes m_c and n_c in a multi-threaded setting, based on the block sizes m_r, n_r and k_c found earlier. As illustrated in Figure 9, the loop at layer 3 is parallelized, as this achieves better locality for the shared L3 cache, in which all the $k_c \times n_c$ row panels of B are stored. In this case, each thread will be assigned a different $m_c \times k_c$ block of A, and all threads share the same $k_c \times n_c$ row panel of B. Then each thread will multiply its own block of A with the shared row panel of B. Such a strategy for parallelizing this particular loop at layer 3 is discussed extensively in [15]. It is also adopted here for the 64-bit ARMv8 multi-core processor.

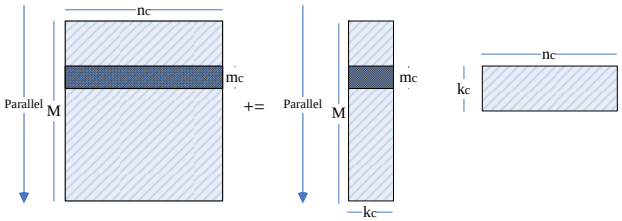

Fig. 9. Parallelization of the loop at layer 3.

Consider again the 64-bit ARMv8 architecture in Figure 1. Without loss of generality, we consider a parallel implementation of DGEMM with eight threads, with one thread per core. Since two cores in one module share a 256 KB L2 cache, the corresponding two threads will have their own block of A at the same L2 cache. Hence, (17) need to be modified to:

$$\begin{aligned} 2 \times m_c \times k_c \times element\_size &\leq \frac{(assoc2 - k2) \times L2}{assoc2} \\ 2 \times k_c \times n_r \times element\_size &\leq \frac{k2 \times L2}{assoc2} \end{aligned} \tag{19}$$

Given $n_r = 6$ and $k_c = 512$, solving both yields $m_c = 24$ and $k2 = 4$. Similarly, the multi-threaded version of (18) is:

$$\begin{aligned} k_c \times n_c \times element\_size &\leq \frac{(assoc3 - k3) \times L3}{assoc3} \\ 8 \times m_c \times k_c \times element\_size &\leq \frac{k3 \times L3}{assoc3} \end{aligned} \tag{20}$$

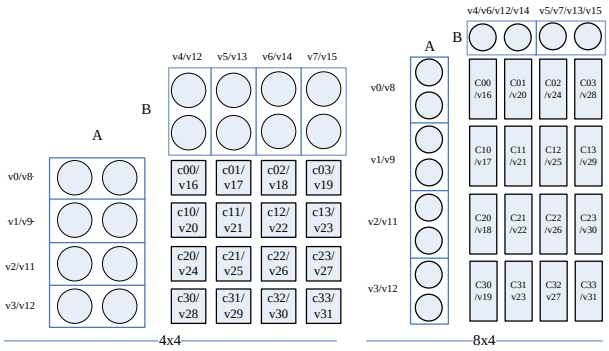

Fig. 10. 4×4 and 8×4 register blocking for the register kernel.

By solving these constraints, we obtain $n_c = 1792$ and $k3 = 2$. As a result, eight blocks of A will reside in the L3 cache together with a shared row panel of B.

V. Experimental results

We have conducted our evaluation on a 64-bit ARMv8 eight-core computing platform described in Table II. We show that our DGEMM implementation realized in terms of our highly-optimized GEBP in OpenBLAS is the fastest available for this architecture. Our serial implementation achieves a peak performance (efficiency) that is close to a theoretical upper bound obtained from micro-benchmarking. Our parallel implementation achieves good scalability across a range of different matrix sizes. In addition, our DGEMM implementation (in OpenBLAS) outperforms a DGEMM implementation in ATLAS [11] by 7.79% on one core and 7.70% on eight cores. Note that the DGEMM implementation in ATLAS is also accelerated by a highly-optimized GEBP in assembly (with its register block size being 5×5 only), representing the only one previously available for the 64-bit ARMv8 architecture.

TABLE II. The experimental platform.

CPU	64-bit ARMv8 eight-core processor
OS	Linux mustang-3.8.0
Compiler	gcc-4.8.2 -O2 -march=armv8-a
OpenBLAS	OpenBLAS_develop-r0.2.9
ATLAS	ATLAS 3.11.31

Based on our performance model, we have optimized DGEMM by producing a highly-optimized 8×6 GEBP with its compute-memory access ratio being maximized. To validate our performance model, we have also developed two more GEBP implementations, as illustrated in Figure 10: (a) 4×4 GEBP, with its register block size being 4×4, and (b) 8×4 GEBP, with its register block size being 8×4. In the 4×4 kernel, a product of a 4×2 matrix A and a 2×4 matrix B is performed. The 8×4 kernel can be viewed as a simplified version of our 8×6 kernel. By proceeding as described in Sections IV-B and IV-C, the block sizes for single- and multi-threaded settings can be found, as given in Table III.

A. Microbenchmarking for Register Block Sizes

We have done some micro-benchmarking to demonstrate that our register block size 8×6 gives rise to good performance and that our instruction scheduling optimization is effective.

TABLE III. Block sizes for three different implementations of GEBP with one thread or eight threads.

	One Thread	Eight Threads
Register Block Size	$m_r \times n_r \times k_c \times m_c \times n_c$	$m_r \times n_r \times k_c \times m_c \times n_c$
8×6	$8 \times 6 \times 512 \times 56 \times 1920$	$8 \times 6 \times 512 \times 24 \times 1792$
8×4	$8 \times 4 \times 768 \times 32 \times 1280$	$8 \times 4 \times 768 \times 16 \times 1192$
4×4	$4 \times 4 \times 768 \times 32 \times 1280$	$4 \times 4 \times 768 \times 16 \times 1192$

The register kernel at layer 7, which is the innermost loop shown in Figure 2, dominates the entire execution time. In each iteration, represented by $\#i$ in Table I, there are $(m_r + n_r)/2$ 128-bit memory instructions on loading data into registers and $m_r n_r/2$ 128-bit NEON floating-point FMA instructions, as depicted in Figure 8. The percentage of arithmetic instructions over the total is given by $(m_r n_r/2)/(m_r n_r/2 + (m_r + n_r)/2)$.

It is interesting to analyze the efficiencies achieved with different ratios of loads over FMA instructions, denoted $LDR : FMLA$. We have written a micro-benchmark, in which the instructions are independent and evenly distributed, to avoid any effect of instruction latency on our experiments. This micro-benchmark can always keep the data in the L1 cache. We report our findings in Table IV. Note that $1 : 2$, $6 : 16$ and $7 : 24$ are the $LDR : FMLA$ ratios roughly corresponding to the 4×4, 8×4 and 8×6 GEBP implementations, respectively. One key observation is that increasing the percentage of arithmetic instructions over the total, i.e., $(m_r n_r/2)/(m_r n_r/2 + (m_r + n_r)/2)$, can improve performance. The percentages for the 4×4, 8×4 and 8×6 GEBP implementations are 66.7%, 72.7% and 77.4%. Thus, 8×6 GEBP is expected to be the best performer.

TABLE IV. Efficiencies under varying $LDR : FMLA$ ratios.

$LDR : FMLA$	1:1	1:2	6:16	1:3	7:24	1:4	1:5
Efficiency (%)	63.0	80.9	87.7	88.7	91.5	94.2	95.2

In a DGEMM implementation, there are two types of memory latencies: WAR and RAW. We have modified our micro-benchmark by letting a memory instruction on loading data to a register to follow a FMA instruction on reading from the same register. The same efficiencies remain. Therefore, there does not appear to be a need to consider the WAR latency, due to possibly the register renaming mechanism used.

We have also tried different instruction execution orders to observe the efficiencies affected by the RAW latency. We find that registers used in load instructions can be free after at least 4 fmla instructions have been executed. Our instruction scheduling in Figure 7 satisfies this constraint. Therefore, in the presence of data dependency, it is possible to hide memory latency by applying instruction scheduling optimizations and to obtain the same efficiencies given in Table IV.

In addition, the results obtained in these micro-benchmarking experiments indicate that the larger the register block size is, the higher the efficiency can be achieved. Due to the absence of L1 cache misses in our experiments, the efficiencies presented in Table IV can be seen as the upper bounds for our DGEMM implementations.

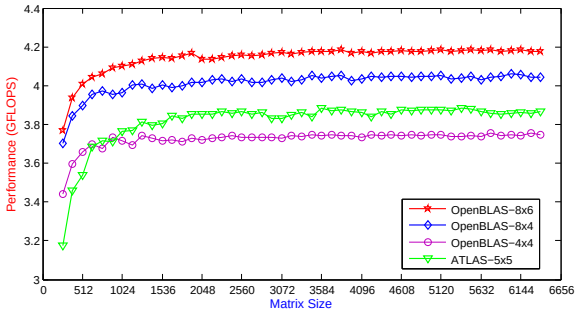

Fig. 11. Performance of four DGEMM implementations (one thread).

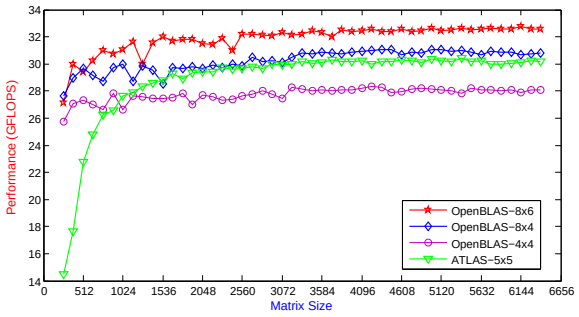

Fig. 12. Performance of four DGEMM implementations (eight threads).

B. Performance Analysis

We consider four DGEMM implementations: OpenBLAS-8×6, OpenBLAS-8×4, OpenBLAS-4×4 and ATLAS-5×5. The former three are developed by us in OpenBLAS with 8×6, 8×4 and 4×4 GEBP kernels, respectively. ATLAS-5×5 is developed with a 5×5 GEBP kernel in ATLAS [11]. In our experiments, square matrices are used with their sizes ranging from 256 to 6400, with a step of 128. For each matrix size, the execution time is measured as the average of five runs.

We present our performance results in Figure 11 (with one thread) and Figure 12 (with eight threads). OpenBLAS-8×6 stands out as the best performer in nearly all the input sizes tested. In particular, OpenBLAS-8×6 outperforms ATLAS-5×5 across all the input sizes, demonstrating that OpenBLAS-8×6 is the fastest DGEMM for the 64-bit ARMv8 architecture.

Table V summarizes these results in terms of peak and average efficiencies in both the serial and multi-threaded settings. OpenBLAS-8×6 is the best performer among all the four metrics evaluated. In particular, the single-thread peak efficiency of OpenBLAS-8×6 is 87.2%, which is very close to the theoretical upper efficiency 91.5% obtained from our micro-benchmarking experiment (Table IV). Compared with ATLAS-5×5, OpenBLAS-8×6 improves its peak performance (efficiency) from 3.88 Gflops (80.9%) to 4.19 Gflops (87.2%) on one core and from 30.4 Gflops (79.2%) to 32.7 Gflops (85.3%) on eight cores. These translate into peak performance (efficiency) improvements by 7.79% on one core and 7.70% on eight cores. In addition, OpenBLAS-8×6 also improves ATLAS-5×5's average efficiency by 8.55% on one core and 10.79% on eight cores.

For OpenBLAS-8×6, OpenBLAS-8×4, OpenBLAS-4×4 and ATLAS-5×5, the compute-to-memory-ratios of their register kernels are estimated by (8) as 6.86, 5.33, 4 and 5, respectively.

TABLE V. EFFICIENCIES OF FOUR DGEMM IMPLEMENTATIONS.

Efficiencies		OpenBLAS			ATLAS
		8×6	8×4	4×4	5×5
Peak	1 Thread	87.2%	84.6%	78.2%	80.9%
	8 Threads	85.3%	81.0%	73.7%	79.2%
Average	1 Thread	86.3%	83.6%	77.6%	79.5%
	8 Threads	83.2%	77.7%	72.3%	75.1%

These results show that our performance model is reasonable. The larger this compute-to-memory access ratio is, the higher the efficiency of a DGEMM implementation will be.

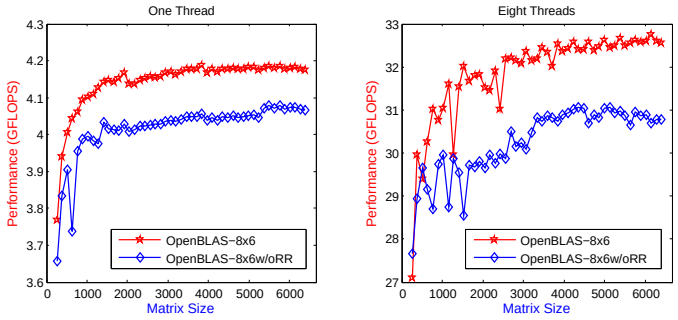

Fig. 13. Effectiveness of software-implemented register rotation.

For the 64-bit ARMv8 architecture with fewer physical registers than x86 for register renaming, Figure 13 demonstrates clearly the effectiveness of our register rotation scheme.

Figure 14 demonstrates the performance and scalability of OpenBLAS-8×4 under four thread configurations (with 1, 2, 4 and 8 threads), with their block sizes $m_r \times n_r \times k_c \times m_c \times n_c$ also shown. In all the cases, m_r, n_r and k_c are the same, obtained in Sections IV-A and IV-B. In the case of 1 and 8 threads, m_c and n_c are found in Sections IV-B and IV-C, respectively. For 2 threads, we find m_c by solving (17) and n_c by solving (20) with 8 replaced by 2. For 4 threads, we find m_c and n_c similarly as for 2 threads. In the case of 2 and 4 threads, different threads always run on different modules (Figure 1), so that a thread running in one module can use the entire L2 cache alone. For each thread configuration, all the threads share the same L3 cache. These results show that OpenBLAS-8×6 is not only efficient but also scalable.

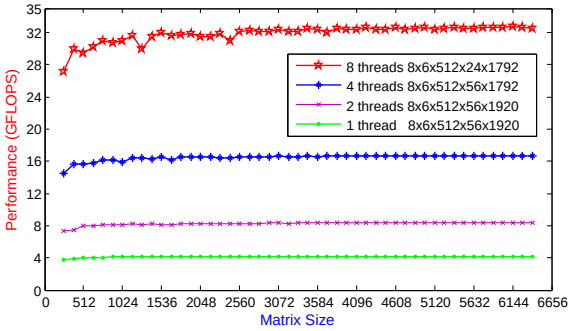

Fig. 14. Performance of OpenBLAS-8×6 under four thread counts.

Table VI compares the performance results of OpenBLAS-8×6 with a few different block sizes selected for $k_c \times m_c \times n_c$. According to [5], these block sizes are often used so that one $m_c \times k_c$ block of A occupies about half of the L2 cache and

TABLE VI. PERFORMANCE OF OPENBLAS-8×6 UNDER DIFFERENT BLOCK SIZES (WITH THE ONES IN BOLD DETERMINED IN THIS PAPER).

OpenBLAS-8×6	$k_c \times m_c \times n_c$	Peak efficiency (%)	Average efficiency (%)
Serial	$\mathbf{512 \times 56 \times 1920}$	87.2	86.3
	$320 \times 96 \times 1536$	86.4	85.4
Parallel (8 Threads)	$\mathbf{512 \times 24 \times 1792}$	85.3	83.2
	$512 \times 24 \times 1920$	85.2	82.9
	$512 \times 56 \times 1792$	80.4	75.5
	$512 \times 56 \times 1920$	80.1	75.4

one $k_c \times n_r$ sliver of B occupies about half of the L1 cache. In the serial setting, we have used $k_c \times m_c \times n_c = 320 \times 96 \times 1536$ obtained according to [5]. As we consider the set-associativity and replacement policy of a cache in selecting block sizes, our choices lead to higher efficiencies in both settings.

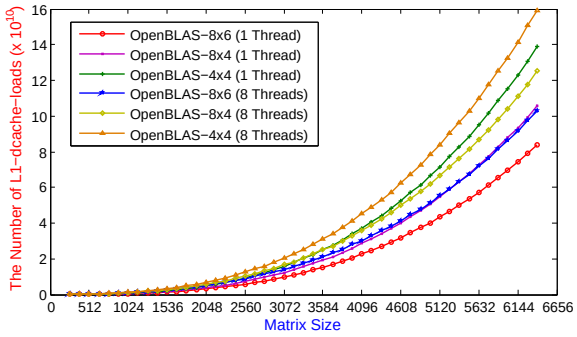

Fig. 15. The number of L1-dcache-loads performed by OpenBLAS-8×6, OpenBLAS-8×4 and OpenBLAS-4×4 under on one core and eight cores.

Finally, Figure 15 compares the number of L1-dcache-loads of OpenBLAS-8×6, OpenBLAS-8×4 and OpenBLAS-4×4. In both the serial and parallel settings, OpenBLAS-8×6 has the smallest number of L1-dcache-loads. It is reasonable to assume that all these DGEMM implementations exhibit the same number of floating-point operations. Therefore, OpenBLAS-8×6 is superior to the other two, since it can overlap the computation with the data movement (especially data loading from the L1 data cache to registers) more effectively. According to Table V, OpenBLAS-8×6 achieves slightly better peak and average efficiencies serially than in paralle, because the parallel execution suffers from more L1-dcache-loads.

TABLE VII. THE L1 CACHE MISS RATES OF OPENBLAS-8×6, OPENBLAS-8×4 AND OPENBLAS-4×4 ON ONE AND EIGHT CORES.

	OpenBLAS-8×6	OpenBLAS-8×4	OpenBLAS-4×4
One Thread	5.2%	4.3%	5.7%
Eight Threads	3.6%	3.2%	5.0%

Table VII compares the L1-dcache-load-miss rates of OpenBLAS-8×6, OpenBLAS-8×4 and OpenBLAS-4×4 in the serial and parallel settings. Note that OpenBLAS-8×6 does not yield the lowest L1 data cache miss rate in either case However, it ultimately attains the highest efficiency due to the smallest number of L1-dcache-loads performed. This shows that the L1 cache miss rate is not as performance-critical in the ARMv8 architecture as in the others [7].

VI. CONCLUSION

We have presented the design and implementation of a highly efficient DGEMM for the 64-bit ARMv8 multi-core processor, guided by a performance model. Our sequential implementation attains a peak performance that is very close to a theoretical upper bound established from micro-benchmarking. Our parallel implementation achieves good scalability across a range of different matrix sizes evaluated. In addition, our implementation outperforms the one in ATLAS quite substantially, making ours the fastest available for this architecture.

In this paper, we do not consider any issues related to the Translation Look-aside Buffer (TLB). In future work, we will analyze the TLB misses and improve our selection of block sizes [16], [17]. We also plan to apply auto-tuning [18] to generate a highly optimized GEBP in assembly language.

ACKNOWLEDGMENT

This research is partly supported by the National High Technology Research and Development Program of China (NO. 2012AA010903), National Natural Science Foundation of China (No. 61402495, No. 61303189, No. 61170049) and ARC Grants (DP110104628 and DP130101970).

REFERENCES

[1] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC?" in *SC*, 2013, pp. 40:1–40:12.
[2] N. Rajovic, N. Puzovic, L. Vilanova, C. Villavieja, and A. Ramirez, "The low-power architecture approach towards Exascale computing," in *ScalA*, 2011, pp. 1–2.
[3] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez, "Tibidabo: Making the case for an arm-based hpc system," *Future Generation Computer Systems*, vol. 36, no. 0, pp. 322 – 334, 2014.
[4] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC*, 1998, pp. 1–27.
[5] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008.
[6] K. Goto and R. Van De Geijn, "High-performance implementation of the Level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 4:1–4:14, Jul. 2008.
[7] X. Zhang, Q. Wang, and Y. Zhang, "Model-driven Level 3 BLAS performance optimization on Loongson 3A processor," in *ICPADS*, 2012, pp. 684–691.
[8] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for generating blas-like libraries," The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-12-30, 2012.
[9] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on fermi GPU," in *SC*, 2011, pp. 35:1–35:11.
[10] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster*, 2010, pp. 19–28.
[11] ATLAS, "Automatically tuned linear algebra software," http://math-atlas.sourceforge.net/.
[12] M. Ali, E. Stotzer, F. D. Igual, and R. A. van de Geijn, "Level-3 BLAS on the TI C6678 multi-core DSP," in *SBAC-PAD*, 2012, pp. 179–186.
[13] G. Ballard, "Avoiding communication in dense linear algebra," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug 2013.
[14] V. Kelefouras, A. Kritikakou, and C. Goutis, "A matrix-matrix multiplication methodology for single/multi-core architectures using SIMD," *The Journal of Supercomputing*, pp. 1–23, 2014.
[15] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *IPDPS*, 2014, pp. 1049–1059.
[16] J. Xue, *Loop Tiling for Parallelism*. Kluwer Academic, 2000.
[17] J. Xue and C. Huang, "Reuse-driven tiling for improving data locality," *International Journal of Parallel Programming*, vol. 26, no. 6, pp. 671–696, 1998.
[18] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng, "Automatic library generation for BLAS3 on GPUs," in *IPDPS*, 2011, pp. 255–265.