


Article

NUMA-Aware DGEMM Based on 64-Bit ARMv8 Multicore Processors Architecture

Wei Zhang ¹, Zihao Jiang ^{2,*}, Zhiguang Chen ², Nong Xiao ¹ and Yang Ou ¹

¹ College of Computer Science, National University of Defense Technology, Changsha 410073, China; zhangw19@nudt.edu.cn (W.Z.); nongxiao@nudt.edu.cn (N.X.); michaelouyang@163.com (Y.O.)

² College of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou 510006, China; chenzhg29@mail.sysu.edu.cn

* Correspondence: jiangzh57@mail2.sysu.edu.cn

Abstract: Double-precision general matrix multiplication (DGEMM) is an essential kernel for measuring the potential performance of an HPC platform. ARMv8-based system-on-chips (SoCs) have become the candidates for the next-generation HPC systems with their highly competitive performance and energy efficiency. Therefore, it is meaningful to design high-performance DGEMM for ARMv8-based SoCs. However, as ARMv8-based SoCs integrate increasing cores, modern CPU uses non-uniform memory access (NUMA). NUMA restricts the performance and scalability of DGEMM when many threads access remote NUMA domains. This poses a challenge to develop high-performance DGEMM on multi-NUMA architecture. We present a NUMA-aware method to reduce the number of cross-die and cross-chip memory access events. The critical enabler for NUMA-aware DGEMM is to leverage two levels of parallelism between and within nodes in a purely threaded implementation, which allows the task independence and data localization of NUMA nodes. We have implemented NUMA-aware DGEMM in the OpenBLAS and evaluated it on a dual-socket server with 48-core processors based on the Kunpeng920 architecture. The results show that NUMA-aware DGEMM has effectively reduced the number of cross-die and cross-chip memory access, resulting in enhancing the scalability of DGEMM significantly and increasing the performance of DGEMM by 17.1% on average, with the most remarkable improvement being 21.9%.

Keywords: BLAS; DGEMM; ARMv8; NUMA; high-performance computing (HPC)



Citation: Zhang, W.; Jiang, Z.; Chen, Z.; Xiao, N.; Ou, Y. NUMA-Aware DGEMM Based on 64-Bit ARMv8 Multicore Processors Architecture. *Electronics* **2021**, *10*, 1984. <https://doi.org/10.3390/electronics10161984>

Academic Editor: Paulo Ferreira

Received: 15 July 2021

Accepted: 15 August 2021

Published: 17 August 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

By considering the advantages of high performance and energy efficiency, ARM-based SoCs have stimulated the development of ARM-based servers [1]. This ARM-based server is not only widely used in data centers but is also applicable for supercomputers. For example, the Japanese supercomputer Fugaku, which won the champion of TOP500 in June 2021, uses Fujitsu's ARM architecture A64FX [2] in its processor. ARM architecture server chips for data center services have been vigorously developed in recent years, including HuaweiKunpeng920 [3], AWSGraviton2, AmpereQuickSilver2019, and MarvellThunderX3. Meanwhile, the 64-bit ARMv8 ISA supports a broader range of address searching, NEON vector unit, fused multiply-add (FMA) operation, and scalable vector extensions (SVE). Therefore, it is meaningful to create HPC based on 64-bit ARMv8 multicore processors architecture.

In the field of HPC, BLAS, as a principal component in many dense linear algebra operations, is widely used in scientific and engineering calculations. BLAS standardizes an application programming interface, which can be used for different implementations and is divided into three types of calculations: vector–vector (Level-1), matrix–vector (Level-2), and matrix–matrix (Level-3). The most critical operation is general matrix multiplication GEMM at Level-3. Processor vendors and HPC researchers usually provide BLAS implementations that are highly optimized for their respective processors, such as Intel's

MKL; AMD's ACML; IBM's ESSL; NVIDIA's cuBLAS. ARM provides the arm performance libraries (ARMPL) [4], a commercial math library that meets the needs of scientific computing and HPC community on arm architecture. The HPC community also offers excellent open source libraries, including ATLAS [5], GotoBLAS [6], OpenBLAS [7], and BLIS [8]. In these numerical libraries, the research community has devoted effort to optimizing GEMM for different architectures [9–12] based on the fact that the double-precision general matrix multiplication (DGEMM) is the core part of the LINPACK benchmark.

Recently, processor manufacturers have introduced an increasing amount of cores on a CPU and begun to integrate more CPUs on an SoC to improve the performance of ARMv8-based SoCs, resulting in dozens or even hundreds of cores on an SoC while using NUMA architecture to accelerate memory access. Theoretically, DGEMM adopts the most straightforward parallelization strategy, where all threads should be using roughly the same amount of time to execute a subtask. Regardless of the system's memory allocation strategy, threads on different cores may have different speeds in practice due to NUMA. Su et al. [13] proposed a hybrid-grained dynamic load-balancing method to reduce this drawback of the NUMA effect by allowing fast threads to steal work from slow ones. Wail et al. [14] proposed a novel user-level scheduling and a specific data alignment method on the NUMA architecture to solve the data locality problem in such systems and alleviate memory bottlenecks in problems with large input datasets. Although we have common goals, our implementation methods and platforms are very different. Unlike Su's and Wail's methods, we reduce the impact of NUMA effect on performance by reducing the number of cross-die and cross-chip memory access events when DGEMM runs on a multi-NUMA architecture compatible with ARMv8.

After analyzing the source of NUMA effects in DGEMM, we identify DGEMM's performance and scalability bottlenecks. This paper designs and implements an efficient NUMA-aware DGEMM based on OpenBLAS on a dual-socket server with 48-core processors to reduce the number of cross-die and cross-chip memory access events. The key insight is to obtain two levels of parallelism: node-level parallelism between NUMA nodes and thread-level parallelism within NUMA nodes. First, NUMA-aware DGEMM allocates node-level tasks based on the NUMA structure and wakes up the main thread on each NUMA node. Second, the remaining threads in each NUMA node are awakened and bound to separate cores. Then each NUMA node redeploys local data. Finally, DGEMM subtasks are executed in parallel by multiple threads in each node.

The main contributions of this research are as follows:

- After constructing a comprehensive empirical characterization design on Kunpeng920 about how NUMA impact DGEMM, we discover the scalability issues and settle the problem from the root;
- We propose a NUMA-aware DGEMM method and design details to reduce cross-die cross-chip memory access caused by NUMA architecture. NUMA-aware DGEMM is a two-level parallelized multi-solver design based on NUMA, used to accelerate DGEMM in 64-bit ARMv8 multicore processor architectures;
- We have implemented this method on dual-socket servers with 48-core processors. The results show that DGEMM performance is improved by 17.1% on average, with the highest rate being 21.9%. Furthermore, the scalability speed-up ratio is increased by 71% if expanding from one NUMA node to four NUMA nodes. The cross-die and cross-chip write operations are reduced by 62.6% and 29.4%, respectively. At the same time, the cross-die and cross-chip read operations are reduced by 24.8% and 22.7%, respectively.

The rest of the paper is organized as follows. Section 2 reviews the Kunpeng920 dual-chip processor architecture and introduces the blocking and packing algorithms in OpenBLAS. Section 3 analyzes the impact of NUMA on the performance and scalability of DGEMM on 64-bit ARMv8 multicore processors. Section 4 describes our DGEMM implementation by focusing on optimizing cross-die and cross-chip access under a NUMA architecture. Section 5 presents and analyzes our experimental results. Section 6 reviews

the related work. Section 7 discusses the future research prospects. Finally, Section 8 concludes the paper and describes some future work.

2. Background

This section provides a brief insight into the details of Kunpeng920 dual-chip processor architecture, non-uniform memory access (NUMA), and the implementation of DGEMM in BLAS3.

2.1. Details of Kunpeng920 Dual-Chip Processor Architecture

Figure 1 shows the memory hierarchy of the Kunpeng920 dual-chip processor system-on-chip [15]. There are two super core clusters (SCCLs) on each chip, with each SCCL having six core clusters (CCLs), which further contain four cores. Each SCCL is packaged in a CPU die. The Kunpeng920 core uses TaiShanV110, which is compatible with the ARMv8-A architecture. Therefore, the dual-chip system function is based on two 48-core processors.

Each processor core integrates private 64 KB 4-way set-associative L1 cache, 64 KB L1d cache and 512 KB 8-way set-associative L2 cache. Each SCCL is configured with a 24 M shared L3 cache which uses the write-back strategy and supports the random re-replacement algorithm. L3 cache is a three-level cache shared on a system-level chip.

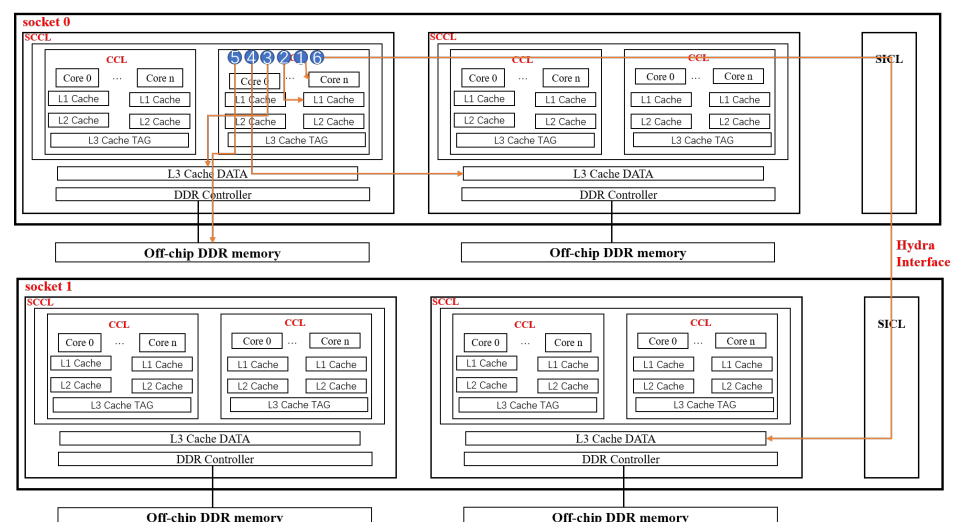


Figure 1. The hierarchical structure of the memory in the Kunpeng920 dual-chip processor system.

In the Kunpeng920 architecture, each core is a single-threaded four-issue superscalar pipeline that supports out-of-order execution, with each core running on 2.6 GHz, as well as having a 128-bit floating-point unit (FPU) capable of executing double-precision fused multiply-add instructions. In addition, the ARMv8 ISA defines 32 128-bit floating-point registers, v0-v31, which can be used for SIMD instructions.

L3 cache is a system-level on-chip three-level cache. The cache is shared by the SCCL where it is located and shared by CPU, various accelerators, and I/O devices. In Figure 1, the data access between the chips is through the hydra root agent (HHA) module, which is not only based on Huawei's custom-defined protocol standard HUAWEI cache coherency system (HCCS) [15], for maintaining data consistency between multiple clusters, but also a module used to maintain data consistency between chips and sockets. The hydra protocol can achieve MESI protocol consistency among multiple L3 caches. Therefore, the access mode for shared data in memory should first access the local L3 cache on a dual-chip system supported by HHA. If data on the local L3 cache is missing, the L3 cache of the adjacent NUMA node on the same chip will be accessed. If there is also a cache miss, it will be accessed in the local memory. If it is still not available, it will be queried in the L3 cache on another chip.

2.2. Non-Uniform Memory Access (NUMA)

The Kunpeng920 processor system-on-chip supports NUMA architecture. Multiple processor cores form a NUMA node, and each node contains local DRAM. The interconnection and communication between each node in the system-on-chip are realized through the network-on-chip, while the interconnection and communication between different chips with high bandwidths and low latency are realized through the hydra interface. Both form the abstraction of the global shared memory of the application program.

On a chip, the time delay for each processor core to access memory depends on the location of the memory relative to the processor. Therefore, the latency highly depends on the distance (i.e., hops) between nodes [16]. Generally speaking, access to the cache and memory in the local node exhibits the best performance.

In Kunpeng920, an SCCL, regarded as a CPU die, can be considered a NUMA node (24 cores). Each chip contains two NUMA nodes. Hence, the dual-chip Kunpeng920 system-on-chip is a four-NUMA node architecture. Each Kunpeng920 processor core accesses not only its private L1 cache and L2 cache but also the shared L3 cache in the same SCCL, as well as the L3 cache in other SCCLs or off-chip DDR memory. Obviously, the processor's memory latency is affected by memory bandwidth and physical distance.

As shown in Figure 1, numbers 1 to 6 represent each of the six typical memory access paths. Memory latency is depicted in the order of fast to slow:

- ① Access to the private L1 cache of the processor core;
- ② Access to the private L2 cache of the processor core;
- ③ Access to the L3 cache data shared within the same SCCL;
- ④ Access to the shared L3 cache DATA in other SCCLs on the same chip;
- ⑤ Access to off-chip DDR memory;
- ⑥ Access to the shared L3 cache DATA of an SCCL on other chips.

2.3. The Implementation of DGEMM in OpenBLAS

Several prevalent open-source BLAS implementations are ATLAS [5], GotoBLAS [6], BLIS [8], and OpenBLAS [7]. ATLAS relies on auto-tuning to generate the kernel, which makes it highly compatible with the platform. BLIS is a portable software framework for instantiating high-performance BLAS-like dense linear. OpenBLAS is an extension and optimization on GotoBLAS. It manually compiles the kernel GEBP (a basic computation unit) for a series of architectures to improve its performance on various platforms. As a result, we choose the OpenBLAS library with the best performance in the Kunpeng920 architecture for optimization. This section describes the DGEMM blocked algorithm in OpenBLAS and a parallel strategy in a multicore architecture.

DGEMM computation indicates that $C := \alpha AB + \beta C$, where A , B , and C are matrices of sizes $M \times K$, $K \times N$, and $M \times N$, respectively. For a better understanding, we assume that both α and β representing the scalars of a matrix are equal to 1, and a matrix is stored in column-major order. The simplified DGEMM can be expressed as $C := AB + C$. Figure 2 shows the DGEMM blocked algorithm in detail, which explains the original three-loop nest unrolling into six layers of high-performance implementation cycles.

The purpose of the DGEMM blocked algorithm in OpenBLAS is to determine the appropriate size for kernel GEBP to maximize cache performance so that the computing power of each core is fully utilized.

In the outermost layer, the $K \times M$ matrix B and the $M \times N$ matrix C are partitioned into column panels of sizes K and M , respectively. At layer 2, the $M \times K$ matrix A and a $K \times N_c$ submatrix of B are partitioned into column panels of size $M \times K_c$ and row panels of size $K_c \times N_c$, respectively. The result of the multiplication of these two panels is a component of matrix C , and it also shows that several GEPPs can form the GEMM. At layer 3, each panel of size $M \times K_c$ in matrix A is partitioned into blocks of $M_c \times K_c$. At this point, the kernel GEBP is generated, which is decomposed by GEPP. The result of kernel GEBP is a panel of size $M_c \times N_c$ in matrix C , which is obtained by multiplying a block of size $M_c \times K_c$

in matrix A and a panel of size $K_c \times N_c$ in matrix B. The GEBP includes layers 4–7 and also undertakes a packing operation.

As shown in Figure 2, the calculations for layer4, layer5, and layer6 are performed in the L3 cache, L2 cache, and L1 cache, respectively. Each $K_c \times N_c$ panel of B should not only reside in the L3 cache all the time [7] but, also, in order to further improve the access speed, OpenBLAS packages the blocks of matrix A and the panels of matrix B into A continuous buffers. Packing A is the process of extracting sub-blocks of size $M_r \times K_c$ from blocks of size $M_c \times K_c$ in matrix A and organizing them together into the L2 cache (layer5). Similarly, packing B is the process of extracting sub-blocks of size $K_c \times N_r$ from a $K_c \times N_c$ panel of matrix B and organizing them together into the L3 cache (layer6). To minimize packaging overhead, each sub-block of size $K_c \times N_r$ is moved to the L1 cache one by one. At layer 5, the multiplication of an $M_c \times K_c$ block of matrix A and a $K_c \times N_r$ sliver of the $K_c \times N_c$ panel of B is called GEBS (block-sliver). Similarly, the calculation in layer6 is called GESS, also known as the micro-kernel, which results in a series of C sub-blocks of $M_r \times N_r$ size. At the last layer, the calculation of an $M_r \times 1$ column sub-slivers of A and a $1 \times N_r$ row sub-slivers of B is called the register kernel, where all the data are stored.

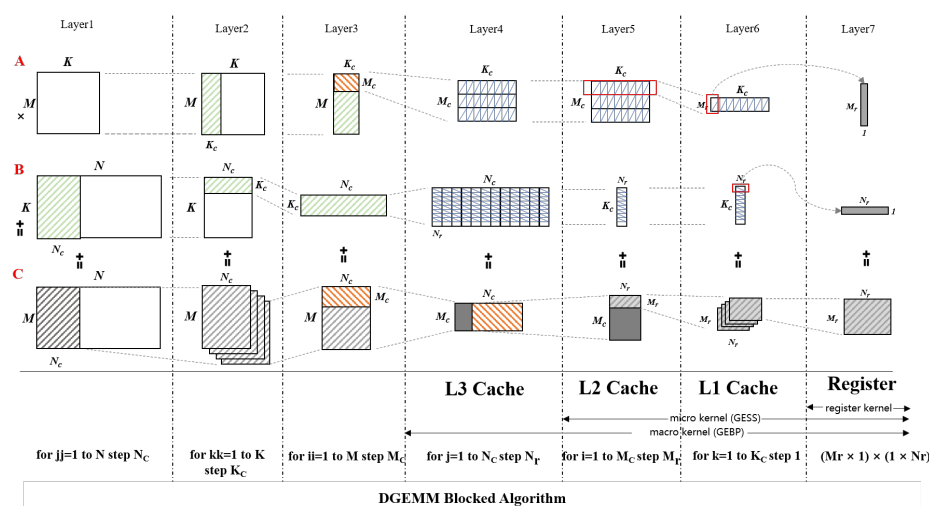


Figure 2. The implementation of DGEMM in OpenBLAS and the representation of the kernel GEBP.

3. Challenges and Issues

In order to analyze the impact of NUMA on the performance and scalability of DGEMM in the Kunpeng920 architecture, we design a series of comparative experiments to explore NUMA architecture. Generally, there are two ways to achieve high-performance DGEMM: (1) a highly optimized kernel GEBP for a specific architecture (2) an appropriate size of the block to make full use of the parallelism of the multicore. Here, we focus on the impact of NUMA events on the parallelism performance of DGEMM on multicore processors. OpenBLAS not only has two multithreading models, OpenMP and Pthreads, but also designed a high-performance kernel, GEBP, in assembly language. In addition, we believe that Pthreads is more suitable for the optimized implementation of DGEMM in NUMA architecture.

This paper evaluates the impact of NUMA architecture on DGEMM performance and scalability in a Kunpeng920 dual-chip processor equipped with four NUMA nodes. The detailed configuration of Kunpeng920 can be found in Section 2. This section first explains the reason why Pthreads was chosen. Second, we observed the scalability of DGEMM in NUMA architecture. Then, we designed six DGEMM computing scenarios using the numactl tool and libnuma API to identify the performance bottleneck of scalability. Finally, we explored the source of the NUMA effect in DGEMM. We analyzed the micro-architecture transactions and memory access events in six scenarios with the top-down tool [17] “Maluma” developed by HUAWEI. Micro-architecture analysis based on ARM performance

monitor unit (PMU) events reflects the operating status of instructions in the CPU pipeline, helping us to quickly locate hardware events that affect performance. Memory access events can determine performance bottlenecks in the memory access process, as well as give possible reasons for these performance problems and optimization suggestions.

3.1. Pthreads vs. OpenMP

The parallel implementation of DGEMM has two multithreading models in OpenBLAS, Pthreads, and OpenMP. However, different threading models perform differently on different platforms. The parallel implementations in Intel's MKL library and AMD's ACML library are based on the OpenMP model. Chen et al. [18] found that OpenMP has a higher implicit parallelism overhead in MKL and ACML. In order to determine which model is more suitable for the Kunpeng920 architecture, we carefully compared the differences and characteristics of the two threading models.

In the Kunpeng920 architecture, we measured DGEMM performance with different scales using Pthreads and OpenMP threading model, respectively. In Figure 3, the single-threaded performance of the two models is very similar, and the overall performance decreases as matrix size increases. However, when matrix size exceeds 5632 ($M = N = K$), the performance of OpenMP decreases faster than Pthreads. This is because the design of the original GEMM kernel is not ideally suitable for the Kunpeng920 processor, which also means that cache is underutilized. The larger the size of DGEMM, the greater the number of memory accesses. These memory access behaviors that are not specifically designed will result in lower cache utilization as the number of memory accesses increases, resulting in a decline in single-core processors performance. In Figure 4, the performance of Pthreads is 13.3% higher than that of OpenMP on average. Regardless of Pthreads or OpenMP is used in OpenBLAS, the performance of DGEMM will produce more considerable jitter with different matrix sizes, which is caused by cache thrashing in a multithreaded system.

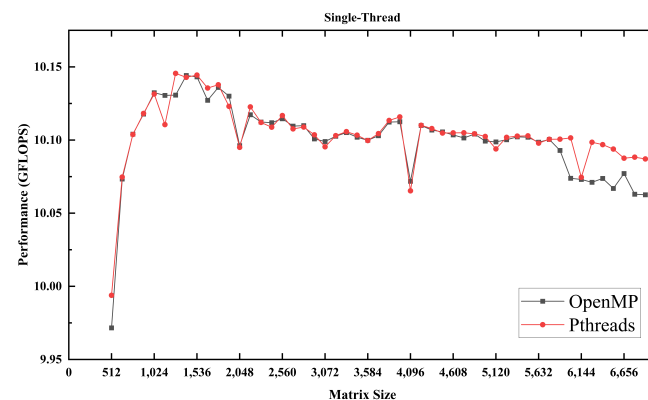


Figure 3. Performance of DGEMM implementations (single-thread).

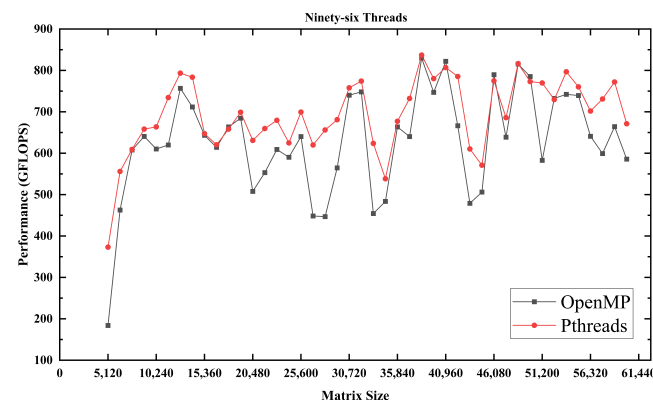


Figure 4. Performance of DGEMM implementations (ninety-six threads).

Pthreads and OpenMP represent two totally different multiprocessing paradigms. Pthreads is a very low-level API for generating threads and synchronization explicitly. Thus, we have highly fine-grained control over thread management, mutexes. On the other hand, OpenMP is much higher level and is more easily scaled than Pthreads. OpenMP should be used for loops that have to be computed on all the cores. Pthreads can do that too, but that is much work. However, as usual, we attain more flexibility with Pthreads. In the Kunpeng920 architecture, the Pthreads implementation of DGEMM in OpenBLAS is superior to the OpenMP implementation in performance. Of course, overlapping calculations and data transmission are critical. Considering the above points and DGEMM algorithm optimization must be implemented according to the hardware architecture, we utilized Pthreads to optimize DGEMM in the Kunpeng920 architecture and enable it to achieve better parallelism.

3.2. Observations on Scalability

In order to observe the scalability of DGEMM on NUMA architecture, we tested the scalability of six scales DGEMM on four NUMA nodes (24 cores per NUMA node). Regardless of the multithreaded distribution strategy, all thread groups were deployed in the global processor cores with the default strategy. Figure 5 shows the scalability of DGEMM in a strong expansion scenario across NUMA nodes. We made the following observations:

(1) All scales DGEMM have poor scalability on NUMA architecture. Meanwhile, by comparing the scalability of different scales DGEMM, it can be found that the smaller matrix size is, the worse the scalability will be. The reason for this phenomenon is simple. For small-scale DGEMM, multithreading creation and communication overheads have gradually concealed the benefits of multithreaded parallel computing. The scalability of DGEMM of all sizes on NUMA architecture usually begins to drop significantly with the addition of more than three nodes. Obviously, if NUMA overhead exceeds the computing power, performance will decrease;

(2) The multithreading performance of all scales DGEMM is almost the same at one NUMA node, with the peak performance at three NUMA nodes. However, the scalability begins to decline after more than 3 NUMA nodes. As the scale of matrix multiplication increases, the scalability declines more slowly. It is because when all 96 cores participate in the calculation, the amount of calculation of this scale cannot make full use of the multilevel cache. This illustrates that the Kunpeng920 architecture is more suitable for matrix multiplication with a scale of M, N, and K over 10,000;

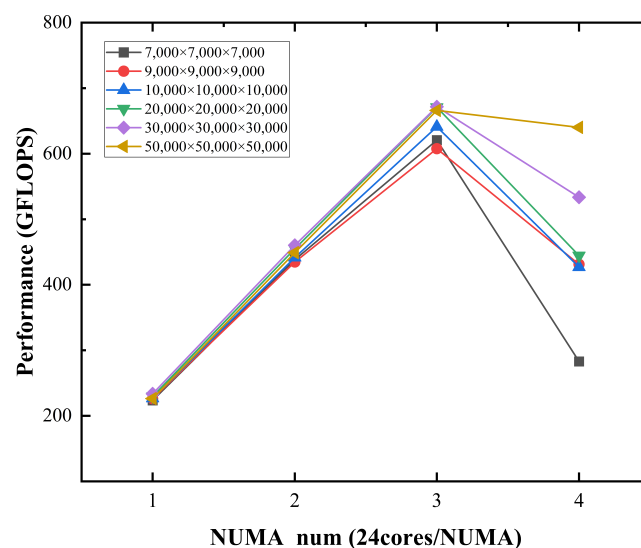


Figure 5. Scalability of DGEMM on NUMA architecture.

(3) All the above evidence indicates that when the number of NUMA nodes increases, the performance of all scales DGEMM on NUMA architecture suffers from serious scalability problems. Since the experiment was conducted in a strong expansion scenario, we focused on memory-level scalability. Next, we applied system-level analysis tools to quantify the bottleneck effect and explore the potential source of the scalability problem of DGEMM on a multicore architecture.

3.3. Cause of DGEMM's Scalability Bottleneck

By analyzing OpenBLAS source code and the Kunpeng920 processor architecture, as well as considering the impact of the memory hierarchy on the performance of DGEMM [19], the NUMA factor may be the most crucial factor affecting the scalability of DGEMM in the Kunpeng920 architecture. Therefore, we used hardware event counter tool "numastat" to explore whether remote NUMA access is frequent. Node-load-miss was used as an evaluation indicator, representing the load of acquiring data from DRAM and Cache in the remote NUMA domain.

Figure 6 shows the node-load-miss with the growing NUMA counts on Kunpeng920. We clearly observe that remote memory access increases. The figure indicates that the current DGEMM designs might not have taken NUMA scalability into consideration.

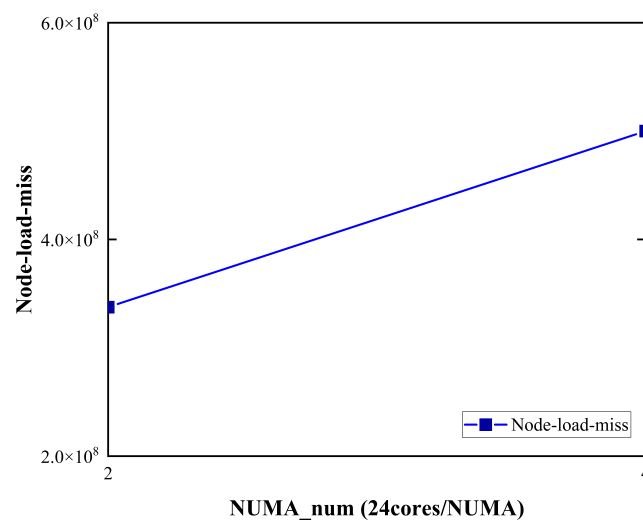


Figure 6. Node-load-miss when performing DGEMM on Kunpeng920.

In order to further quantify the impact of NUMA, we use numactl, which employs a specific NUMA scheduling or memory placement strategy to run the process, to design six DGEMM computing scenarios for the position of the thread on the NUMA node. Node x means that DGEMM runs on NUMA node x . Then, the hyper tuner [20] tool, Malluma, was used to measure event-accurate sampling [21] in six designed comparative experiments, including all_write_count, local_write_count, cross-die_write_count, cross-chip_write_count, all_read_count, local_read_count, cross-die_read_count, and cross-chip_read_count. These events represent the total number of write, the number of local write, the number of cross-die write, and the number of cross-chip write, respectively. The situation of reading data is similar. Table 1 shows the number of PMU events for a DGEMM with a size of 50,000 ($M = N = K$) in six experiments. The number of local accesses, the number of cross-die accesses, and the number of cross-chip accesses can be observed in Table 1. We observe that when the number of NUMA nodes is expanded from 1 to 4 (experiment ID 1, 2, 5, and 6), the number of cross-die and cross-chip increases significantly each time a NUMA node is expanded. The results show that the number of cross-die increased by 33.3%, and cross-chip increased by 14.09%. Due to the frequent remote memory access, the proportion of access time spent on DRAM increased significantly, which ultimately led to a decrease in the computing performance of DGEMM.

Table 1. The number of PMU events for a DGEMM with a size of 50,000 ($M = N = K$) in six experiments.

ID	1	2	3	4	5	6
Experiment Settings	Node 0	Node 0, 1	Node 0, 2	Node 0, 3	Node 0, 1, 2	Node 0, 1, 2, 3
local_write_count	4.63×10^9	7.17×10^7	7.25×10^7	7.33×10^7	7.06×10^7	6.88×10^7
cross-die_write_count	4.26×10^4	1.92×10^7	2.14×10^7	2.09×10^7	3.26×10^7	4.07×10^7
cross-chip_write_count	3.21×10^6	4.56×10^5	1.80×10^7	1.80×10^7	1.66×10^7	1.89×10^7
local_read_count	4.66×10^{13}	6.10×10^{10}	6.10×10^{10}	6.09×10^{10}	5.91×10^{10}	5.65×10^{10}
cross-die_read_count	1.23×10^7	4.03×10^{10}	4.13×10^{10}	4.17×10^{10}	4.96×10^{10}	5.80×10^{10}
cross-chip_read_count	4.57×10^7	5.24×10^7	9.89×10^{10}	1.09×10^{10}	1.20×10^{10}	1.37×10^{10}

3.4. Source of NUMA Effects in DGEMM

The reason why DGEMM frequently performs remote memory access on NUMA architecture must be explored theoretically. Remote memory access is caused by the absence of data required for the calculation task of the NUMA node in the DRAM and cache of the local NUMA itself. In order to analyze the reasons, the parallel process of DGEMM on the NUMA architecture should be carefully studied.

Section 2.3 has discussed in detail the detailed process of decomposing DGEMM into GEBP kernel units. In Figure 2, the kernel GEBP in layer4 is split from GEPP in layer3. The computing task of an $M \times K_c$ panel of A multiplied by a $K_c \times N_c$ panel of B is decomposed into several computing tasks of the $M_c \times K_c$ block and $K_c \times N_c$ panel. In this process, a panel of size $M \times K_c$ in GEPP is first divided into blocks of size $M_c \times K_c$, which are multiplied by the same panel of size $K_c \times N_c$, resulting in matrices of size $M_c \times N_c$ in C. Consequently, multiple blocks share a row panel in GEBP. For a single thread, a block of size $M_c \times K_c$ and a panel of size $K_c \times N_c$ are packaged into the L3 cache to produce a GEBP unit. After completing the calculation of this GEBP, the next block of size $M_c \times K_c$ is replaced into the L3 cache and calculated with the same panel of size $K_c \times N_c$. The next GEPP task will not start until the execution of this GEPP is completed. For the parallelism of multi-core processors, the execution process of a GEPP is a synchronization cycle. In a synchronization cycle, each core executes a GEBP task separately, and all the cores share a panel of $K_c \times M_c$ with the size of matrix B. Only when all the threads complete all the GEBP tasks in this cycle can the next GEPP task be executed.

Taking the execution process of DGEMM on the dual-chip Kunpeng920 processor as an example, the reason why DGEMM generates remote memory access on the multi-NUMA architecture is explained in detail. When the process is created, the OS default memory allocation policy is most likely the first-touch policy. As we all know, many applications must change their initialization in order to take advantage of NUMAness. However, regardless of the operating system's default memory allocation policy, the distribution of data and work on NUMA nodes is rough. For a computationally intensive program with rules, such as DGEMM, the data required for the calculation may still exist on other NUMA nodes, which will lead to a large number of remote memory accesses. Therefore, we use the first-touch policy as an example to analyze the detailed process of remote memory access. According to the first-touch policy, memory is allocated to the local memory of the NUMA node where the main thread is located. Therefore, matrix A, matrix B, and result matrix C are all in the local memory of the NUMA node where the main thread is located. Since the current DGEMM implementation does not consider NUMA boundaries, threads work across NUMA nodes.

Figure 7 illustrates the thread mapping and data access of the reading process of DGEMM parallelized on a multi-NUMA machine, where each thread is statically mapped to a core, with the private L1 cache and L2 cache of each core being omitted in the figure. Assuming that the main thread is created on NUMA node 0, the main thread divides

the task of DGEMM into several GEBP subtasks and wakes up a subthread on each core simultaneously. Each thread has a private block of matrix A whose size is $M_r \times N_r$, and 96 threads share a panel of matrix B with a size of $K_c \times N_c$. Since NUMA node0 is where the main thread is located, the matrices A, B, and C allocate memory in the DRAM corresponding to node0. The calculation task of 24 cores on node0 requires 24 blocks of size $M_r \times N_r$ in matrix A and a row panel of size $K_c \times N_c$ in matrix B, so these 24 cores directly read data from the local DRAM to the L3 cache. In NUMA nodes 1–3, 72 cores numbered core 24 to 95 must obtain data from the DRAM of NUMA node 0 through remote access. In a synchronization cycle, each core needs to fetch a block to execute the next GEBP task from the DRAM corresponding to NUMA Node0 whenever it completes a GEBP task. Nevertheless, the row panel remains unchanged in the L3 cache during this cycle until the GEBP implementation is complete. The implementation of fine-grained parallelism must frequently cross NUMA boundaries to read or write data from remote memory domains, which significantly increases the memory access latency under limited bandwidth.

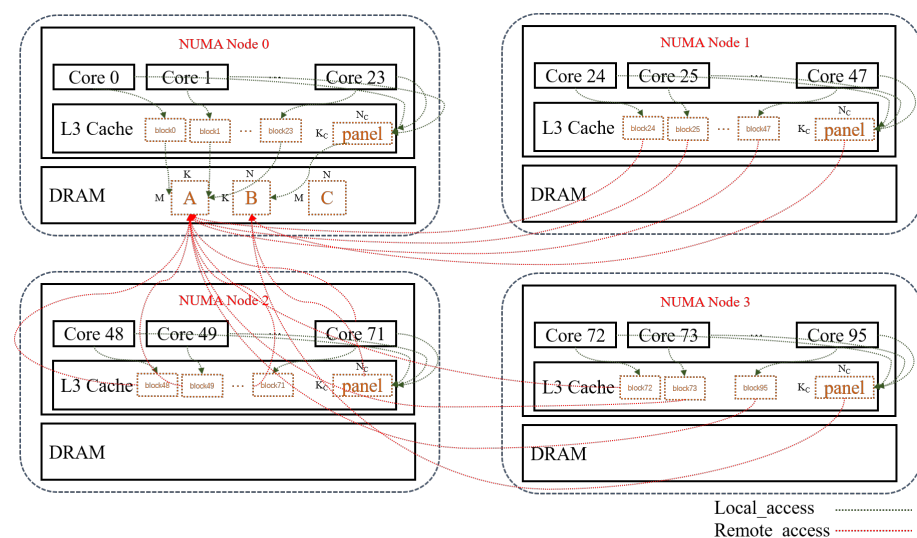


Figure 7. Parallel DGEMM thread mapping and data access paths.

4. Design Methodology

We propose a high-performance implementation method of NUMA-Aware DGEMM based on multicore and multi-CPU architecture to address the challenges and issues above. The core thought of this article is to reduce the number of cross-die and cross-chip memory accesses to improve computing performance. In order to avoid frequent remote NUMA access, the data should be coordinated with the calculation in the same place. It is natural to replace threads with processes to ensure the co-location of data and calculations. However, using processes instead of threads consumes more memory space [22]. It is unsuitable for computationally intensive applications to run a process on each core or NUMA node in machines with limited memory. Consequently, the implementation of NUMA-aware DGEMM should maintain a pure thread mode.

NUMA-aware DGEMM is different from various existing versions of DGEMM. It makes use of hierarchical parallelism on the multi-NUMA architecture to distribute tasks across threads. Figure 8 outlines the design of NUMA-aware DGEMM. At the algorithm level, NUMA-aware DGEMM decomposes the task into two parallel levels. The entire DGEMM task is first decomposed into N computing tasks with disjoint computing data according to the number N of NUMA nodes on the processor architecture. These computing tasks and computing data are deployed to different NUMA domains to benefit from NUMA node-level parallelism. Section 4.1 covers the details and verifies that the cost of data redeployment is reasonable. Then, NUMA-aware DGEMM further divides computing tasks for each processor core in each NUMA node to benefit from thread-level parallelism in the NUMA node. Section 4.2 describes the NUMA-aware algorithm in DGEMM.

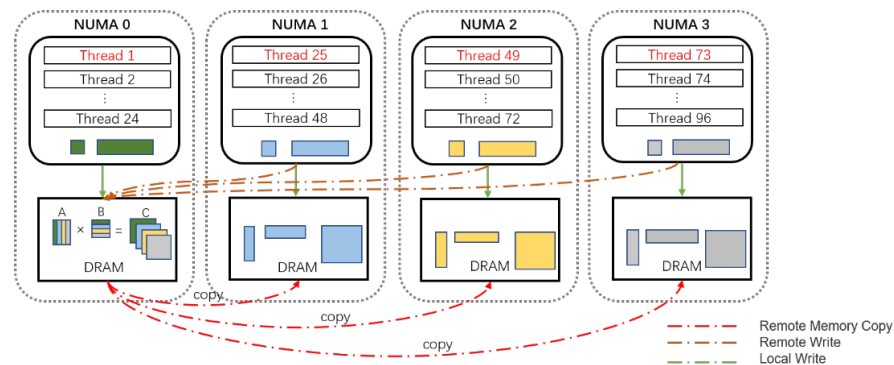


Figure 8. NUMA-aware DGEMM thread mapping and data access paths.

Then, NUMA-aware DGEMM maps the algorithm to a machine supporting NUMA architecture for calculation and data assignment. Compared with the implementation of DGEMM based on OpenMP, NUMA-Aware DGEMM adopts a more flexible multi-threading solution. Section 4.3 elaborates on the thread mapping method on the dual-chip Kunpeng920. Finally, Section 4.4 discusses the key issues in the implementation of NUMA-aware DGEMM.

4.1. Task Distribution and Data Redeployment

There are many ways to divide tasks in the parallel implementation of DGEMM. Among them, the above block algorithm used by DGEMM in OpenBLAS is efficient. In a synchronization cycle, all CPU cores share a panel of size $K_c \times N_c$ in matrix B and execute each GEBP independently and efficiently. In fact, for a server processor equipped with a large-scale core, the overhead of a synchronization cycle for all cores to execute a GEBP task is expensive. There is no data dependency on computing tasks in an asynchronous period. It can be seen from Section 3.4 that, since the initial data are stored in the local memory of the node where the main thread is located, a large number of remote memory accesses will be caused when all the cores in the remaining nodes participate in the calculation, which causes vast overhead. Given the nature of the DGEMM block algorithm, this paper divides the computation into data disjoint tasks to achieve independent node-level parallelism.

The scheme of task division and data redeployment on the four NUMA nodes architecture is shown in Figure 9. The four colors in matrix A, matrix B, and matrix C, respectively, correspond to the four calculation tasks that the four NUMA nodes are responsible for, and these four tasks are not related in any way. At this time, we call the NUMA node a solution module. Two points should be emphasized here: on the one hand, task division and data redeployment are only for matrix A and matrix B, and no improvements in the result matrix C. On the other hand, the processor cores in each solution module directly update the sub-blocks in the result matrix C after calculating the GEBP tasks they are responsible for. Therefore, there are no synchronizations between threads/processes while updating the result matrix C. NUMA-aware DGEMM performs thread-level fine-grained calculations based on local data in each solution module, giving full play to multicore parallelism. Each solution module acts as an independent machine to perform its DGEMM tasks. As the solution module after task division has its independent calculation tasks, in each solution module, we are only required to wait for the slowest core among the 24 processor cores to complete this round of tasks, and then the solution module starts the next local GEBP task. Therefore, the number of threads sharing a panel of size $K_c \times N_c$ in matrix B is reduced from the original 96 threads to 24 threads, which means that the synchronization range is reduced from the original 96 to 24 threads. Additionally, the method reduces the implementation of DGEMM multithreading synchronization overhead.

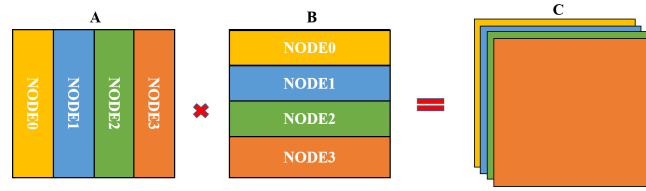


Figure 9. The DGEMM task division and data redeployment scheme on four NUMA nodes.

In order to reduce the number of cross-die and cross-chip memory accesses, it is vital to achieve the data localization of each solution module after the task allocation is completed. As the memory access overhead of a NUMA node is related to the memory location, the data deployment will seriously affect the application performance. On the one hand, the cache has an essential impact on performance in computationally intensive applications. When the data are required, the application must obtain them from memory to cache. If the data are stored in remote memory, the delay will be huge. After testing, the local memory access delay, cross-die access delay, cross-chip near-end access delay, and cross-chip remote access delay were 90, 156, 241, and 293 ns, respectively. On the other hand, non-local threads will access the data required by the computing task in the local memory of the NUMA node where the main thread is located, which will cause remote memory accesses to be much greater than the number of local memory accesses. In particular, when calculating large-scale DGEMM, the original algorithm will cause a more significant number of remote memory accesses, and data deployment is critical in this scenario.

Local memory access overhead is mainly related to the memory hierarchy and the frequency of cache replacement. The DGEMM in OpenBLAS optimizes the local cache utilization through the packing and blocking operations. Accordingly, each GEBP block and panel will not be read to the cache repeatedly, and there will only be one cache miss for each data block. It is known that initial data matrices A, B, and C are all stored in the memory of the NUMA node where the main thread is located, and the delay of reading and writing data are determined by the memory access delay and memory access bandwidth. For ease of description, assuming that the remote thread must read N data blocks, the total remote memory access overhead can be expressed as $T_{remoteaccess}$:

$$T_{remoteaccess} = \sum_i^N \left(\frac{block_i}{B_{rm}} + T_{r_delay} \right) \quad (1)$$

In Formula (1), $block_i$ represents the size of the i -th data, B_{rm} represents the bandwidth of remote access, and T_{r_delay} represents the delay of remote access. When data need be localized, the data will be redeployed based on the number and distance of nodes, and then the data from the remote NUMA node is copied to the local DRAM. After the operation is completed, the node will read data in the local DRAM. The time overhead of data redeployment comes from remote copies of data and multiple reads of local data. The total time cost is expressed by $T_{copy+localaccess}$

$$T_{copy+localaccess} = 2 \left(\frac{block_i}{B_{r-l}} + T_{r-l_delay} \right) + \sum_i^N \left(\frac{block_i}{B_{lm}} + T_{l_delay} \right) \quad (2)$$

In Formula (2), A and B represent the bandwidth and delay of the remote copy, respectively.

Regardless of whether the data are read from non-local memory to local registers or local memory, the overhead in the two cases is almost equal. On the contrary, the remote access bandwidth and latency are much greater than the local memory access bandwidth and latency, so we can assume that $T_{r_delay} \approx T_{r-l_delay}$, $T_{r_delay} > T_{l_delay}$, $B_{rm} \approx B_{r-l}$ and $B_{rm} < B_{lm}$. The difference between the overhead of the above two access methods is shown in Formula (3). When there are N data blocks and the value of N is large:

$$\Delta(i) = N \left(\frac{N-2}{N} \left(\frac{block_i}{B_{rm}} + T_{r_delay} \right) - \left(\frac{block_i}{B_{lm}} + T_{lm_delay} \right) \right) \quad (3)$$

According to the above reasoning, it can be concluded that $\Delta(i) > 0$, the overall overhead of data redeployment is less than the overhead of continuous access to remote memory.

4.2. NUMA-Aware DGEMM Algorithm

After discussing task division and data redeployment, this paper realizes efficient two-level parallelism on the dual-socket servers with 48-core processors, which combines node-level parallelism and thread-level parallelism. NUMA-aware DGEMM first divides the task into independent subtasks by node, forming an independent DGEMM solution module with NUMA nodes as the unit. Next, these modules are deployed on the nodes and then redeploy data according to the tasks of the solution module. Finally, multicore parallelism within the node is realized through fine-grained division of tasks on the node.

Algorithm 1 elaborates the parallel optimization algorithm of DGEMM in the Kunpeng920 architecture. The optimization algorithm is implemented based on OpenBLAS. OpenBLAS determines the number of threads at compile-time according to the number of processor cores. Therefore, each thread is only required to wake up to perform a specified task during the algorithm's running without creating a thread. In the algorithm, lines 1–3 detect the number of NUMA nodes in the architecture and node processor cores. Lines 4–15 describe a task division algorithm to determine matrix A and B's range of K dimensions. Lines 16–22 wake up the main thread on each NUMA node, also called a management thread, and each management thread is bound to the first processor core of the local NUMA node. The tasks between NUMA nodes are parallel. The tasks in the NUMA node are shown in lines 23–41. More specifically, line 26 indicates that the main thread on the NUMA node wakes up the remaining threads and binds to the kernel. Lines 27–31 allocate local memory space for the main thread on each NUMA node and copy into local memory some of the data from matrix A and B that will be used in subsequent calculations. The details inside the two nested loops in lines 33–40 indicate what the Pthreads thread is responsible for within the same NUMA node. First, a panel in matrix B shared by all threads in NUMA node is fetched to the shared L3 cache in line 33. Secondly, in line 34, each thread fetches a block of matrix A needed for its calculation to the private L2 cache. Then in line 35, each thread calculates the position of block C of the result of this task, and finally calculates with the kernel optimized for the core TSV110 in lines 36–37 and writes the result directly back to the position of result block C. The function pthread_barrier() is used for thread synchronization.

4.3. Two-Level Calculation and Data Mapping Method

The two-level parallelism of NUMA-aware DGEMM can be naturally mapped to threads running on the NUMA architecture. Binding these threads to the processor core is necessary but challenging. Because each thread has equal importance in the original DGEMM parallel implementation, these threads cannot only predict the creation order but are not bound to the core. Therefore, if we want to improve the performance of DGEMM, the two-level parallel NUMA-DGEMM will have to carefully design the specific behavior of Pthreads on the NUMA architecture.

NUMA-aware DGEMM can automatically map threads to the appropriate processor core for any NUMA architecture thanks to its thread scheduling strategy. Above all, NUMA-aware DGEMM uses functions in libnuma library to detect NUMA topology to determine the relationship between NUMA nodes and cores. Second, NUMA-aware DGEMM allocates and records a unique ID for each NUMA node while allocating a management thread for each NUMA node. Furthermore, all threads of the local node are awakened by the management thread. Finally, NUMA-aware DGEMM traverses all cores in the NUMA node and calls sched_setaffinity to bind all threads to the NUMA node where the management thread is located.

Algorithm 1: Parallel Optimization Algorithm of DGEMM on Kunpeng920

```

Input: A[MxN], B[NxK], C[]
Output: C[MxN]
1 Get information about NUMA architecture.;
2 Nodes = getNodes();
3 CPUsPerNode = getCPUsPerNode();
4 Task distribution: Calculate the partition range of matrix A and matrix B in k
  dimensions;
5 RangeK[0] = 0;
6 CPUK = 0;
7 while K > 0 do
8   width = (K + Nodes - CPUK - 1) / (Nodes - CPUK) ;
9   K = K - width;
10  if K < 0 then
11   | width = width + K;
12  end
13  RangeK[CPUK + 1] = RangeK[CPUK] + width;
14  CPUK = CPUK + 1;
15 end
16 Wake up the main thread on each NUMA node and bind it.;
17 j = 0 ;
18 for j < Nodes do
19   | WakeThread(j * CPUsPerNode, &PrimaryNodeThread, j);
20   | SetThreadAffinityMask(thread[j], j * CPUsPerNode);
21   | j++;
22 end
23 Tasks for the main thread on each NUMA node;
24 foreach Thread n in PrimaryNodeThread do
25   | /* Prepare and bind other threads on the local NUMA node */;
26   | n.PrepareOtherThread();
27   | data redeployment;
28   | allocOnNode(LocalA, n.nodeID);
29   | allocOnNode(LocalB, n.nodeID);
30   | memcpy(LocalA, A[:, RangeK[n.nodeID]:RangeK[n.nodeID+1]]);
31   | memcpy(LocalB, B[RangeK[n.nodeID]:RangeK[n.nodeID+1], :]);
32   | parallelism in node with Pthreads.;
33   | foreach Block panelB in LocalB do
34     | foreach Block blockA in LocalA do
35       | blockC = C.getCorespondentBlock();
36       | /* The NUMA main thread and the other threads on the node complete
37         | the GEMM calculation */;
38       | doBlockMultiply(blockA, blockB, blockC): kernel optimization with
39         | SIMD. Each thread calculate private Block A with common PanelB by
40         | vector instructions by vector instructions. The kernel is written in
41         | ARM64 assembly.
38     | end
39     | pthread_barrier();
40   | end
41 end

```

From an implementation perspective, after completing the task allocation and data redeployment stage according to NUMA features, the actual multithreaded parallel computing implementation mapping of NUMA-aware DGEMM is performed on three levels: (1) At the top level, NUMA-aware DGEMM is for each NUMA node wakes up a Pthreads-based management thread; (2) In the middle layer, the management thread of each NUMA

node executes independent matrix multiplication tasks by waking up the number of threads corresponding to the number of cores of the NUMA node. The management thread of each NUMA node maintains affinity with the NUMA node through binding; (3) At the bottom layer, each Pthreads thread binds the processor core and executes the single-threaded DGEMM' kernel GEBP operation to maintain affinity with this NUMA node.

NUMA-aware DGEMM should ensure that each CPU core can execute the DGEMM kernel GEBP correctly while binding threads to cores. In addition to ensuring that the data are adequately deployed in the local DRAM, the management thread within each NUMA node also applies the classic DGEMM partition algorithm to ensure all cores' efficient parallelism within the node.

4.4. Discussion

NUMA-aware DGEMM is designed for the NUMA architecture, with the following advantages: (1) It performs data localization for each NUMA node, eliminating the primary cross-die and cross-chip access in the parallel layer; (2) It is adaptive, as NUMA features of different architectures are different. It dynamically divides tasks and data according to the architectural features and keeps the data between NUMA nodes free of dependencies. This design is suitable for any NUMA architecture; (3) This design is accurate, which means that it calculates and writes back the result every time it is the same as the original DGEMM design.

5. Results

We implemented our methods through optimizing the DGEMM interface in OpenBLAS and evaluated its performance and scalability on Huawei's TaiShan2280(2 * Huawei Kunpeng920 5250 processor) server described in Table 2, which is a high-performance server with multicore processors based on the ARMv8 architecture. The NUMA-aware method we present is compatible with all variants of GEMM methods, such as SGEMM and ZGEMM. Since the Linpack benchmark used by TOP500 strongly relies on the DGEMM variant, we only evaluated the DGEMM method's performance.

Table 2. The experimental platform.

	Feature	Description
Hardware	Architecture	AArch64 (Arm64)
	Number of Core	96, no hyper-threading support
	Frequency	2600 MHz
	SIMD	AArch64 Neon instructions (128-bit)
	SIMD pipelines	1
	FLOPS/cycle	4
	Register File	32 128-bit vector registers
	L1 Data Cache	64 KB, 4-way set associative, 64 B cache line, LRU
	L2 Data Cache	512 KB, 4-way set associative, 64 B cache line, LRU
	L3 Data Cache	96 MB, 1 MB per core
	Memory	256 GB
	NUMA Nodes	4 NUMA Nodes
	Software	Operating System
Compiler		GNU/GCC 9.2.0
Thread Model		Pthreads
BLAS		OpenBLAS 0.3.13 version

This section presents the evaluation results. We proved NUMA-aware DGEMM's high performance and excellent scalability in the Kunpeng920 architecture. Additionally, we collected NUMA-related events through hardware performance counter APIs to prove that NUMA-aware DGEMM method can eliminate massive remote memory access, which leads to its high performance and scalability.

5.1. Overall Performance and Scalability

Among all open source libraries, DGEMM in OpenBLAS has the best performance on the Kunpeng920 processor. Therefore, our optimization is based on OpenBLAS and we use OpenBLAS as a major comparison object. The differences before and after DGEMM optimization are exhibited by how they divide and organize the computations, as shown in Figures 7 and 8. We evaluated eight DGEMM implementations: NUMA-aware DGEMM with 8×6 GEBP kernel, NUMA-aware DGEMM with 8×4 GEBP kernel, NUMA-aware DGEMM with 4×4 kernel, OpenBLAS 8×6 kernel, ATLAS, BLIS, ARMPL and NUMA-aware DGEMM without data localization. ARMPL is a commercial math library developed by ARM. NUMA-aware DGEMM without data localization can be used to prove the correctness of the performance model in Section 4.1. In this experiment, square matrices were used with their sizes ranging from 10,240 to 61,440, with a step of 1024, to demonstrate the superior performance of NUMA-aware DGEMM. For each matrix size, the execution time was measured as the average of ten runs.

For further comparison, we also evaluated the performance of BLIS and ATLAS on TaiShan 2280. We use thread average efficiency (Eavg) to describe the result; Eavg is normalized from GFlops, and computed by $Eavg = GFlops / (nt \times dGFlops)$, where dGFlops represents the theoretical peak performance of a core.

Figure 10 shows our performance results. NUMA-aware DGEMM with 8×6 GEBP kernel stands out as the outstanding performance in nearly all inputs. The performance of NUMA-Aware DGEMM-8x6 is 17.1% higher than OpenBLAS- 8×6 on average, with the most remarkable improvement being 21.9% ($M = N = K = 47,104$). ARMPL is the worst performer among all eight implementations. ATLAS uses an automatically-tuned technique unoptimized for a particular hardware platform, which leads to poor performance. BLIS performs better than ATLAS, closer to the results of NUMA-aware DGEMM with the 4×4 GEBP kernel, but not better than other NUMA-aware DGEMM versions. BLIS kernel is optimized by the compiler but not hand-crafted as with the OpenBLAS kernel. The performance of NUMA-aware DGEMM with the 8×6 GEBP kernel is the best within all inputs, which proves that our method is effective in 64-bit ARMv8 multicore processor architecture.

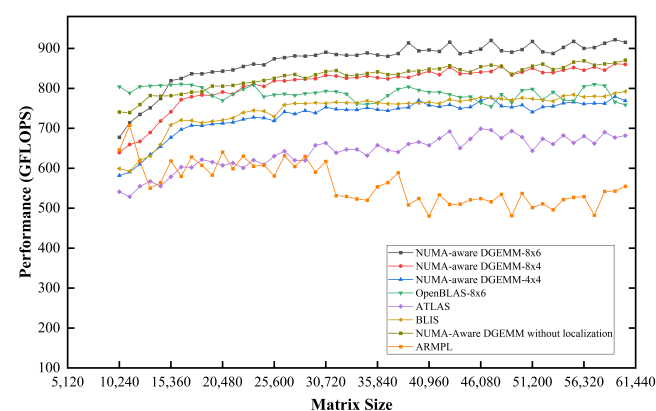


Figure 10. Performance of six DGEMM implementations.

For NUMA-aware DGEMM and NUMA-aware DGEMM without data localization, the former outperforms the latter across nearly all the input sizes with an average performance increase of 4.46%. These results show that our performance model in Section 4 model

is reasonable. The overall overhead of data redeployment is less than the overhead of continuous access to remote memory.

It is worth noting that when the square matrix size is smaller than 15,360, the performance of OpenBLAS is superior to NUMA-aware DGEMM, but with a larger matrix size, the advantage of OpenBLAS dwindle. When the matrix size is larger than 15,360, our method starts better than OpenBLAS. The reason for the phenomenon is that in the full-threaded running state, the calculation data cannot fully utilize the advantages of the three-level cache due to the too-small matrix size. At the same time, the creation cost of multithreading is far greater than the benefit of multithreading parallelism. Therefore, NUMA-aware DGEMM is more suitable for large-scale matrix multiplication operations.

Table 3 summarizes the results of the peak efficiency and average efficiency for different configurations. NUMA-aware DGEMM with the 8×6 GEBP kernel is the best among all six objects. The peak efficiency and average efficiency of NUMA-aware DGEMM reach 92.34% and 86.97%, respectively. Compared with OpenBLAS- 8×6 , NUMA-aware method improves DGEMM's peak efficiency from 80.70% to 92.34% and average efficiency from 78.12% to 86.97%. These translate into peak and average efficiency improvements by 11.64% and 8.85% on 96 cores, respectively.

Table 3. Efficiencies of four dgemm implementations.

Efficiencies	NUMA-Aware DGEMM			OpenBLAS	ATLAS	BLIS
	8×6	8×4	4×4	8×6		
Peak	92.34%	86.33%	78.11%	80.70%	70%	79.33%
Average	86.97%	81.20%	73.23%	78.12%	64%	74.61%

Figure 11 shows the scalability of NUMA-aware DGEMM for square matrices of different sizes on a multi-NUMA architecture. Compared to OpenBLAS DGEMM's scalability in Figure 5, the performance of NUMA-aware DGEMM improves significantly on multi-NUMA architecture. The acceleration ratio is 1.93 when expanded from one NUMA node to two nodes and 2.73 from one to three. In particular, when expanding the NUMA node from one to four, the parallel acceleration ratio is increased from 2.01 to 3.45, an improvement of approximately 71%. As the number of NUMA nodes increases, there is no obvious downward trend in scalability.

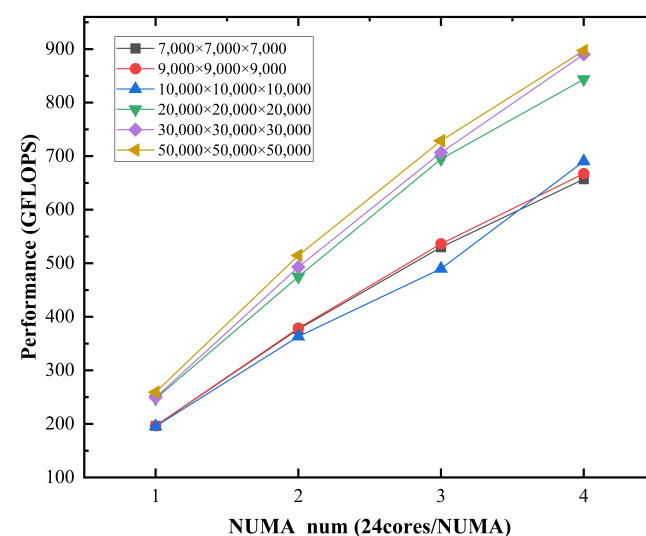


Figure 11. Scalability of NUMA-aware DGEMM.

Our work compares performance against codes that use the default OS memory allocation policy, which is most likely a first-touch policy. No matter what default memory

allocation strategy the OS uses, the distribution of data and work on NUMA nodes is rough, so our work focuses on how data and work are orchestrated across NUMA nodes. Assuming the first data touch is to be completed by the thread that will work with that data according to a classical owner computes rule.

It is well known that the initialization must be changed to exploit NUMAness. Therefore, we believe that comparing default initialization vs parallel NUMA-oriented initialization is very necessary. We tested the performance of DGEMM using three memory allocation strategies, first-touch, interleave, and bind, before and after optimization. Interleave refers to interleaving and allocating memory on all nodes or designated nodes. Bind refers to the forced allocation of memory to the specified node.

Figure 12a,b, respectively, show the performance results of DGEMM using different memory allocation strategies before and after optimization. Figure 12a shows that if matrices of the same size use different memory allocation strategies, they will have a more significant performance gap. Comparing the two curves of Origin_Interleave_All and Origin_Interleave_node1_node2, we can conclude that the more nodes that participate in memory allocation for large-scale matrix multiplication, the better the performance of DGEMM. At the same time, it can be observed that the performance of the first-touch strategy and the interleave strategy are comparable. Figure 12b shows that optimized DGEMM will not fluctuate with the default memory allocation of the operating system. No matter what memory allocation strategy is adopted, the larger the matrix size, the better the matrix multiplication performance. Therefore, the careful arrangement of data and work on NUMA nodes is beneficial to improving DGEMM performance.

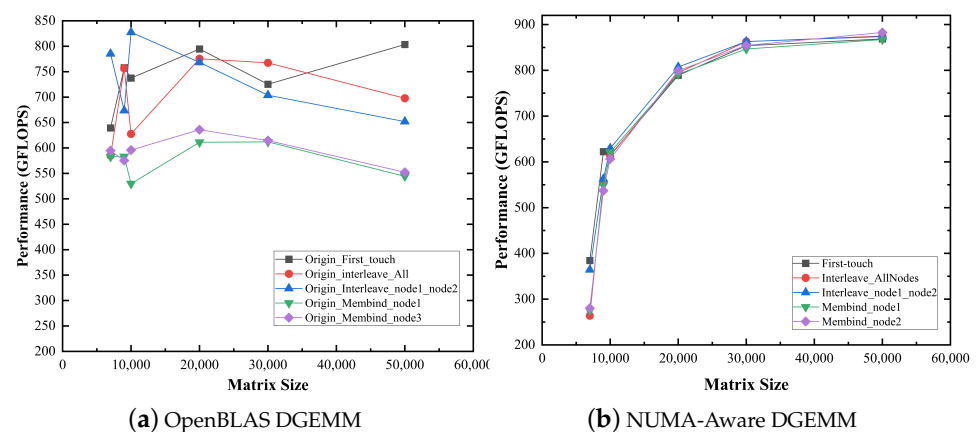


Figure 12. Performance of DGEMM using different memory allocation policies.

5.2. Reduced Remote and Random Accesses

To evaluate NUMA-aware DGEMM optimization effect on remote memory access, we used hardware performance counter APIs on TaiShan 2280 server to collect relative events. Except eight events mentioned in Section 3.3, four extra hardware events were included: instructions, IPC, L1-dcache-loads, and L1-dcache-load-misses, which represent the number of instructions to be executed, the number of instructions to be executed in each CPU cycle, the hit number of L1 data cache and the missing number of L1 data cache, respectively. We used a square matrix with a size of 50,000 ($M = N = K = 50,000$) as input below.

Figure 13a,b show that DGEMM NUMA-aware has less remote memory access operation than OpenBLAS DGEMM in four NUMA-nodes ARMv8 server. We analyzed them by dividing memory access operation in DGEMM into the reading and writing processes. In the reading process, DGEMM requires 5.079×10^{13} read operations when using origin OpenBLAS implementation. Our method reduces them to 3.818×10^{13} (24.8%). At the same time, the number of local memory access operations increases by approximately 20%. As a result, cross-die read operations are reduced by approximately 24.8%, and

cross-chip read operations are reduced by approximately 22.7%, respectively. In the writing process, our method reduces the number of remote write operations from 5.052×10^{11} to 3.712×10^{11} (26.5%), and local memory write operations, cross-die write operations and cross-chip write operations are reduced by 5.3%, 62.6%, and 29.4%, respectively. It proves that our method is accelerated by reducing the number of remote memory access.

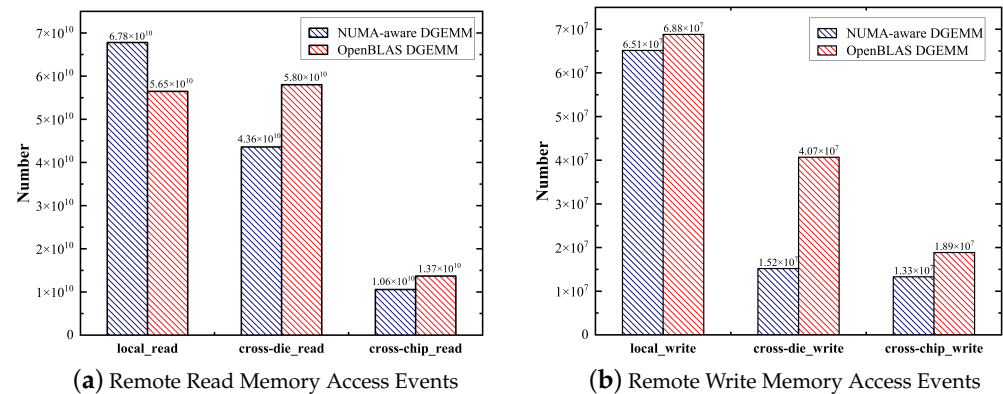


Figure 13. Remote read and write memory access events for both NUMA-aware DGEMM and OpenBLAS DGEMM.

NUMA-aware DGEMM reduces the number of remote memory access operations, the number of instructions after compiling, and the number of CPU cycles the program requires. Figure 14 shows that the program uses 36.54% fewer cycles and 58.1% fewer instructions. It proves our method can be an improvement on the architecture level.

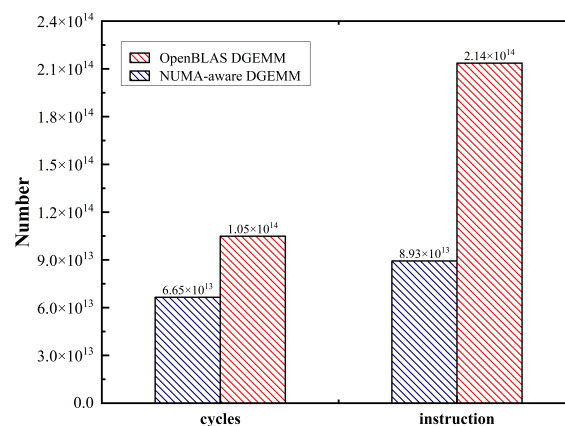


Figure 14. Number of cycles and instructions for both NUMA-aware DGEMM and OpenBLAS DGEMM.

Figure 15a,b compare the number of l1-dcache-load and l1-dcache-load-misses between NUMA-aware DGEMM and OpenBLAS DGEMM, respectively. Obviously, both methods will execute the same amount of floating-point operations. Our method is superior as it can overlap computation and data movement more effectively, especially moving data from the L1 data cache to register. At the same time, the number of l1-dcache-misses in our method is 16.3% less than that of OpenBLAS, with less l1-dcache-loads as mentioned before; finally, it shows better performance, which proves that l1-dcache-load-misses and l1-dcache-loads are key to high performance in the Kunpeng920 architecture.

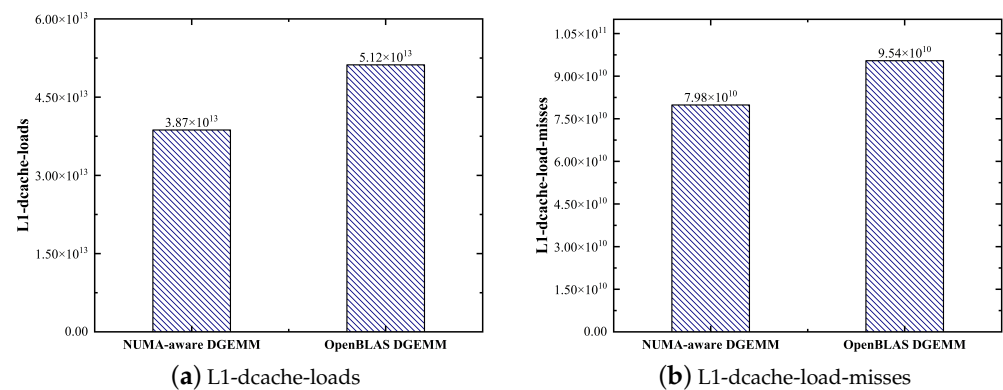


Figure 15. The number of L1-dcache-loads and L1-dcache-load-misses for both NUMA-aware DGEMM and OpenBLAS DGEMM.

6. Related Work

In dense linear algebra, high-performance GEMM is the core implementation and the base of BLAS3 [19,21]. Many state-of-the-art BLAS libraries, such as GotoBLAS [6] and its derivative version OpenBLAS [7], are implemented based on this idea. There are two possible ways to optimize GEMM operations. The one is to optimize the computation kernel of GEMM, and the other one is to optimize its parallel performance in multicore architecture. We introduced cross-die remote memory access and cross-chip memory access into optimizing GEMM as reducing them will achieve better performance.

Optimizing the GEMM kernel must consider a trade-off between performance and portability. For example, GotoBLAS [6] supports many CPU architectures, such as x86, ARM64, Itanium, and MIPS64, but not all are outstanding. OpenBLAS [7] is based on GotoBLAS by handcrafting the kernel but generating by a compiler for every architecture. OpenBLAS achieve excellent performance among lots of open-source BLAS implementation. In particular, Wang et al. [10] designed and implemented a highly efficient DGEMM kernel based on OpenBLAS for 64-bit ARMv8 eight-core processors, which allows OpenBLAS to perform better when it runs in the ARMv8 multicore platform, and we are still using this kernel within our method. ATLAS [5] adopts an automatically tuned technique to find the kernel which will achieve the best performance. Nuechterlein and Whaley [23] also optimized a GEBP kernel of DGEMM implementation in ATLAS. AUGEM [24] and POCA [25] also use automatically tuned techniques and kernel optimization. BLIS [8] reduced the necessary kernels to what we believe is the simplest set that still supports the high performance that the computational science community demands.

The two-layer parallel DGEMM design based on the NUMA-aware algorithm proposed by us is not the only method to design DGEMM for NUMA architecture. Su et al. [13] proposed a hybrid-grained dynamic load-balancing method to reduce this drawback of the NUMA effect by the improved work-stealing algorithm. Wail et al. [14] proposed a simple user-level thread scheduling and a specific data alignment method on the ccNUMA architecture to solve memory bottlenecks in problems with large input datasets. Smith et al. [12] added parallelism to the BLIS framework for matrix multiplication appears to support high performance.

The workload distribution methods on homogeneous and heterogeneous platforms to achieve performance gains and energy efficiency for DGEMM is a well-researched area. Li et al. [26] described some GPU GEMM auto-tuning optimization techniques that allow us to keep up with changing hardware by rapidly reusing. Nath et al. [27] presented GPU specific acceleration techniques to develop new kernels (e.g., syrkm, symv). Moss et al. [28] presented a customizable matrix multiplication framework for the Intel HARPv2 CPU+FPGA platform. Alonso et al. [29] investigated the balance between the time-to-solution and the energy consumption of dense linear algebra operations on a hybrid platform equipped with a multi-core processor and several GPUs.

7. Discussion

As we enter the era of petaflops, the demand for data processing and computing has grown exponentially. In order to meet this demand, researchers have begun to frequently use high-performance computing (HPC) platforms, such as multi-core processors, FPGA and GPU.

However, many high-performance CPUs achieve higher performance at the expense of large power consumption. Therefore, the power consumption of the CPU has increased significantly in recent years. The increase in CPU power consumption significantly impacts its deployment to a wide range of applications. It is estimated that exascale machines built with the technology used in today's supercomputers will consume hundreds of millions of watts of electricity [Miller 2013]. In order to continue to improve performance while expanding performance, low-power technology is indispensable for the CPU. In recent years, ARM-based SoCs have competitive performance and energy efficiency, making ARM-based SoCs the candidates for the next-generation HPC systems. The energy consumption of DGEMM on ARM has not been fully explored. For these reasons, the energy efficiency evaluation of DGEMM on the ARM architecture is worthy of study.

8. Conclusions

In this article, we presented the NUMA-aware DGEMM method based on the Kunpeng920 architecture to optimize cross-die and cross-chip access under the NUMA architecture. The experimental results show that the method is proven to be effective in the TaiShan 2280 server. Furthermore, our parallel implementation achieves good scalability across various matrix sizes evaluated, which is conducive to calculating large-scale matrix multiplications on architectures with larger-scale processor cores in the future. In addition, we did not design the high-performance kernel GEBP of this architecture in this article. In future work, we will establish a performance model design and implement an efficient GEMM core for this architecture.

Author Contributions: Conceptualization, W.Z. and Z.J.; methodology, W.Z.; software, Z.J.; validation, W.Z. and Z.J.; investigation, W.Z.; resources, Y.O.; data curation, Z.J.; writing—original draft preparation, W.Z.; writing—review and editing, W.Z. and Z.J.; visualization, W.Z.; theoretical and practical guidance, Z.C. and N.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key Research and Development Program of China grant NO.2018YFB0204303, NSFC61802418, NSFC61832020, NSFC61872392, Pearl River S & T Nova Program of Guangzhou under grant NO.201906010008, and the Major Program of Guangdong Basic and Applied Research under grant NO.2019B030302002.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Acknowledgments: The authors thank ZhiGuang Chen and Nong Xiao for their guidance and the server provided by Pengcheng Labs.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Azimi, R.; Fox, T.; Gonzalez, W.; Reda, S. Scale-out vs scale-up: A study of arm-based socs on server-class workloads. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2018**, *3*, 1–23. [CrossRef]
2. Top500 Supercomputer Sites. Available online: <https://www.top500.org/lists/2020/6/s/> (accessed on 25 May 2021).
3. Xia, J.; Cheng, C.; Zhou, X.; Hu, Y.; Chun, P. Kunpeng 920: The First 7nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* **2021**. [CrossRef]
4. Arm Performance Libraries. Available online: <https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries> (accessed on 25 May 2021).
5. Whaley, R.C.; Dongarra, J.J. Automatically tuned linear algebra software. In Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, San Jose, CA, USA, 7–13 November 1998; p. 38.
6. Goto, K.; Geijn, R.A.V.D. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* **2008**, *34*, 1–25. [CrossRef]

7. Zhang, X.; Wang, Q.; Zhang, Y. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems, Singapore, 17–19 December 2012; pp. 684–691.
8. Van Zee, F.G.; Van De Geijn, R.A. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.* **2015**, *41*, 1–33. [[CrossRef](#)]
9. Liu, J.; Chi, L.; Gong, C.; Xu, H.; Jiang, J.; Yan, Y.; Hu, Q. High-performance matrix multiply on a massively multithreaded Fiteng1000 processor. In Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, Fukuoka, Japan, 4–7 September 2012; pp. 166–176.
10. Wang, F.; Jiang, H.; Zuo, K.; Su, X.; Xue, J.; Yang, C. Design and implementation of a highly efficient DGEMM for 64-Bit ARMv8 multi-core processors. In Proceedings of the 2015 44th International Conference on Parallel Processing, Beijing, China, 1–4 September 2015; pp. 200–209.
11. Cui, H.; Wang, L.; Xue, J.; Yang, Y.; Feng, X. Automatic library generation for BLAS3 on GPUs. In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, Anchorage, AK, USA, 16–20 May 2011; pp. 255–265.
12. Smith, T.M.; Van De Geijn, R.; Smelyanskiy, M.; Hammond, J.R.; Van Zee, F.G. Anatomy of high-performance many-threaded matrix multiplication. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 19–23 May 2014; pp. 1049–1059.
13. Su, X.; Lei, F. Hybrid-Grained Dynamic Load Balanced GEMM on NUMA Architectures. *Electronics* **2018**, *7*, 359. [[CrossRef](#)]
14. Alkowiileet, W.Y.; Carrillo-Cisneros, D.; Lim, R.V.; Scherson, I.D. NUMA-aware multicore matrix multiplication. *Parallel Process. Lett.* **2014**, *24*, 1450006. [[CrossRef](#)]
15. Dai, Z.; Liu, J. *Architecture and Programming of Kunpeng Processor*; Tsinghua University Press: Beijing, China, 2020.
16. McCormick, P.S.; Braithwaite, R.K.; Feng, W.C. *Empirical Memory-Access Cost Models in Multicore Numa Architectures*; Technical Report; Los Alamos National Lab. (LANL): Los Alamos, NM, USA, 2011.
17. Yasin, A. A top-down method for performance analysis and counters architecture. In Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 23–25 March 2014; pp. 35–44.
18. Chen, S.; Zhang, Y.; Zhang, X.; Cheng, H. Performance Testing and Analysis of BLAS Libraries on Multi-Core CPUs. *J. Softw.* **2010**, *21*, 214–223.
19. Kågström, B.; Ling, P.; Van Loan, C. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.* **1998**, *24*, 268–302. [[CrossRef](#)]
20. Available online: <https://support.huawei.com/enterprise/zh/doc/EDOC1100172781/426cffd9> (accessed on 26 May 2021).
21. Kågström, B.; Van Loan, C. Algorithm 784: GEMM-based level 3 BLAS: Portability and optimization issues. *ACM Trans. Math. Softw.* **1998**, *24*, 303–316. [[CrossRef](#)]
22. Zhu, X.; Zhang, J.; Yoshii, K.; Li, S.; Zhang, Y.; Balaji, P. Analyzing MPI-3.0 process-level shared memory: A case study with stencil computations. In Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzhen, China, 4–7 May 2015; pp. 1099–1106.
23. ATLAS. Automatically Tuned Linear Algebra Software. Available online: <http://math-atlas.sourceforge.net/> (accessed on 26 May 2021).
24. Wang, Q.; Zhang, X.; Zhang, Y.; Yi, Q. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 17–22 November 2013; pp. 1–12.
25. Yi, Q. Automated programmable control and parameterization of compiler optimizations. In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2011), Seoul, Korea, 12–16 February 2011; pp. 97–106.
26. Li, Y.; Dongarra, J.; Tomov, S. A note on auto-tuning GEMM for GPUs. In *International Conference on Computational Science*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 884–892.
27. Nath, R.; Tomov, S.; Dongarra, J. Accelerating GPU kernels for dense linear algebra. In Proceedings of the International Conference on High Performance Computing for Computational Science, Berkeley, CA, USA, 22–25 June 2010; pp. 83–92.
28. Moss, D.J.; Krishnan, S.; Nurvitadhi, E.; Ratuszniak, P.; Johnson, C.; Sim, J.; Mishra, A.; Marr, D.; Subhaschandra, S.; Leong, P.H. A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 107–116.
29. Alonso, P.; Dolz, M.F.; Igual, F.D.; Mayo, R.; Quintana-Orti, E.S. Reducing energy consumption of dense linear algebra operations on hybrid CPU-GPU platforms. In Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, Madrid, Spain, 10–12 July 2012; pp. 56–62.