J. Fang *et al.* Performance Evaluation of Memory-Centric ARMv8 Many-Core Architectures: A Case Study with Phytium 2000+. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(1): 1–13 January 2018. DOI 10.1007/s11390-015-0000-0

Performance Evaluation of Memory-Centric ARMv8 Many-Core Architectures: A Case Study with Phytium 2000+

Jian-Bin Fang[1], Xiang-Ke Liao[1], Chun Huang[1], and De-Zun Dong[1,*]

[1] *College of Computer Science, National University of Defense Technology, Changsha 410073, China*

E-mail: {j.fang, xkliao, chunhuang, dong}@nudt.edu.cn

Received July 15, 2018 [Month Day, Year]; revised October 14, 2018 [Month Day, Year].

Abstract This article presents a comprehensive performance evaluation of Phytium 2000+, an ARMv8-based 64-core architecture. We focus on the cache and memory subsystems, analyzing the characteristics that impact the high-performance computing applications. We provide insights into the memory-relevant performance behaviours of the Phytium 2000+ system through micro-benchmarking. With the help of the well-known roofline model, we analyze the Phytium 2000+ system, taking both memory accesses and computations into account. Based on the knowledge gained from these micro-benchmarks, we evaluate two applications and use them to assess the capabilities of the Phytium 2000+ system. The results show that the ARMv8-based many-core system is capable of delivering high performance for a wide range of scientific kernels.

Keywords many-core architectures, memory-centric design, performance evaluation

1 Introduction

The high-performance computing (HPC) hardware is firmly moving towards the many-core design, and the ARMv8-based processors are emerging as an interesting alternative building block for HPC systems [1–3]. This can be seen from the fact that the 64-core Phytium 2000+ architecture has been used to build the prototype of China's new-generation exascale supercomputer (Tianhe-3) [4], as well as the fact that the Fugaku supercomputer has been built upon a 48-core A64FX architecture [5]. Thus, it is important to understand such novel microarchitecture designs and their performance impacts on typical HPC applications. Having such knowledge is useful not only for better utilizing the computation resources, but also for justifying a further increase in the processor core provision and driving the innovations in memory system design.

In this article, we present a comprehensive performance evaluation of the Phytium 2000+ many-core architecture for HPC applications. We focus on the characteristics of Phytium 2000+ that have a direct performance impact on these applications – high memory capacity and bandwidth, the unique cache organization, the mesh-based on-chip interconnect, and a large number of available hardware cores.

We first describe and highlight the new design features of the Phytium 2000+ processor (Section 2). Most notably, Phytium 2000+ integrates 64 ARMv8-based hardware cores and uses 128 GB of memory, which can deliver 563.2 GFLOP/s of double-precision performance and 153.6 GB/s of memory throughput

Regular Paper

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB0204301, and the National Natural Science Foundation of China under Grant Nos. 61972408 and 61602501.

*Corresponding Author

when running at 2.2 GHz. This results in a good machine balance between calculations and memory accesses. Besides, Phytium 2000+ leverages a hierarchical heterogeneous on-chip network and cache organization distributed among 8 panels, which helps to achieve a good data locality and scalability [3].

We then use a set of microbenchmarks to characterize the low-level memory-relevant features of the Phytium 2000+ architecture (Section 3). We focus on the memory and cache subsystem in particular, including the core-to-core communication performance and the aggregated memory bandwidth with all hardware cores. We observe that a hierarchy of data locality has to be dealt with so as to achieve high performance. Special care has to be taken as to whether the data is located in the local cache, in the same core group, in the same panel, or across panels. Our results show that the measured `write` bandwidth can reach over 80% of the theoretical peak DRAM bandwidth, which is surprisingly larger than the `read` bandwidth. This is achieved by using all available hardware cores, and pinning each thread to a distinct hardware core. We also demonstrate that the non-contiguous memory access is detrimental to the bandwidth efficiency, with Phytium 2000+ showing more restrictions on the stanza length of data prefetching than the conventional `x86` processors.

In addition to our low-level evaluation on the memory subsystem, we analyze a Phytium 2000+ many-core architecture with the well-known roofline model (Section 4). Thanks to its balance between calculations and memory accesses, the Phytium 2000+ system can deliver promising performance for typical high-performance computing kernels, including sparse matrix-vector multiplication (SpMV), 3D stencil, and general matrix-matrix multiplication (GEMM).

We finally evaluate the Phytium 2000+ system using two typical high-performance computing kernels (Section 5). They are SpMV and GEMM, which are two essential linear algebra kernels and widely used in high-performance computing applications. We show that for both applications, the Phytium 2000+ system achieves excellent performance, comparable to the start-of-the-art results. Our work demonstrates that the Phytium 2000+ system can be used to efficiently handle much larger datasets, compared with the accelerator-centric design.

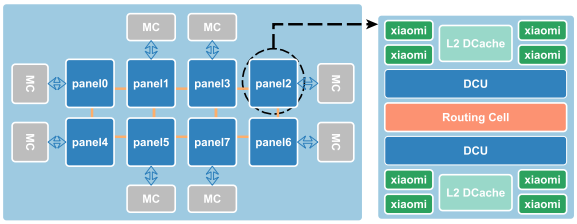


Fig.1. A high-level view of the Phytium 2000+ architecture. The 64 processor cores are groups into eight panels (a), where each panel contains eight ARMv8 based Xiaomi cores (b).

2 Overview of the Phytium 2000+ Architecture

Fig.1 gives a high-level view of the Phytium 2000+ processor. It uses the `Mars II` architecture [1], and features 64 high-performance ARMv8 compatible `xiaomi` cores running at 2.2 GHz. The entire chip offers a peak performance of 563.2 Gflops for double-precision operations, with a maximum power consumption of 96 Watts. The 64 hardware cores are organized into 8 `panels`, where each panel connects a memory control unit.

The panel architecture of Phytium 2000+ is shown in Fig.1(b). Each panel has eight `xiaomi` cores, and each core has a private L1 cache of 32KB for data and instructions, respectively. Every four cores form a `core group` and share a 2MB L2 cache. The L2 cache of Phytium 2000+ uses a inclusive policy, i.e., the cache-lines stored in L1 are also present in the L2 cache.

[1]Phytium Mars II Microarchitectures, https://en.wikichip.org/wiki/phytium/microarchitectures/mars_ii

Each panel contains two Directory Control Units (DCU) and one routing cell. The DCUs on each panel act as dictionary nodes of the entire on-chip network. Mars II uses a hierarchical on-chip network, with a local interconnect on each panel and a global connect for the entire chip. The former couples cores and L2 cache slices as a local cluster, and the latter is implemented with a configurable cell-network to connect panels. Phytium 2000+ uses a home-grown Hawk cache coherency protocol to implement a distributed directory-based global cache coherency across panels. The connected DDR memory modules are working at 2400MHz, giving a theoretical bandwidth of 153.6GB/s.

We run a customized Linux OS based on version 4.4 on the Phytium 2000+ system. We use gcc v8.2.0 compiler and the OpenMP/POSIX threading model.

3 MicroBenchmark Results

This section presents the core-to-core communication performance numbers, and the aggregated memory bandwidth for the Phytium 2000+ memory subsystem.

3.1 Core-to-Core Latency Results

3.1.1 Latency Overview

The accessing latency is referred to as the time of moving a cacheline within the local core or between two distinct cores. We use Molka's approach to measure the core-to-core latency numbers [6]. During the measurement, we use multiple threads to move data between cores. To ensure that the buffer allocated by a thread belongs to a fixed core, we pin each thread to a fixed core, i.e., thread n always runs on core n (Cn). Fig.2 shows the latency results when C0 loads data from its local cache, from C1 sharing a L2 cache slide with C0, from C4 on the same panel, and from C8 on a different panel. Note that, the performance numbers are measured when the cachelines are modified initially.

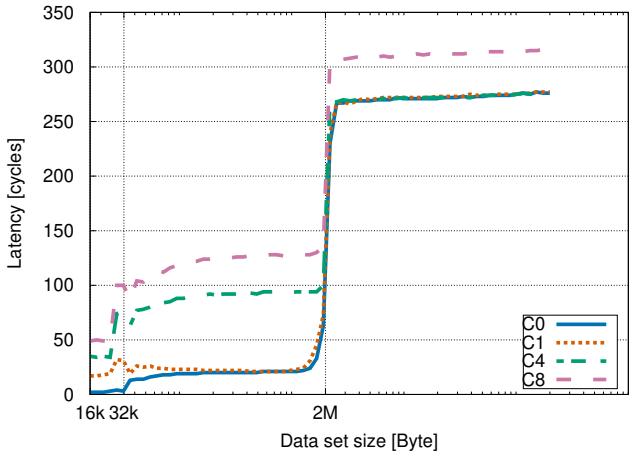


Fig.2. Read latency of C0 accessing the local (C0) cache or the cache of another core (C1, C4 or C8).

We see that, accessing the local L1 and L2 cache takes 3 cycles (1.4 ns) and 21 cycles (9.5 ns), respectively. The specification of the first generation Mars describes that accessing the local L1 and L2 takes 2 ns and 8 ns, respectively, which is in accordance with our measured numbers [3]. When C0 loads data from C1, the latency is the same as that accessing the local L2 cache. Fig.2 shows that, no matter which memory layer the data is suited in, loading cachelines across core groups or panels takes many more cycles than accessing the local cache slices. Thus, loading data within the local cache slice is the fastest.

3.1.2 Across-Panel Latency Results

We evaluate the performance impact of panel distance on latency when accessing cores fixed to different panels. Fig.3 shows the latency results when C0 accessing the cores on P1 (Panle 1)–P7 (Panel 7), respectively.

We see that the latency numbers vary over the panel distance, with a latency variance of up to 105 cycles. Besides, the latency numbers of C0 on P0 accessing C8 on P1 and C32 on P4 are the same. This is because P1 and P4 are at the same distance to P0. This result also agrees with our measured NUMA-aware stream bandwidth (Fig.6) and the theoretical latency results [3].

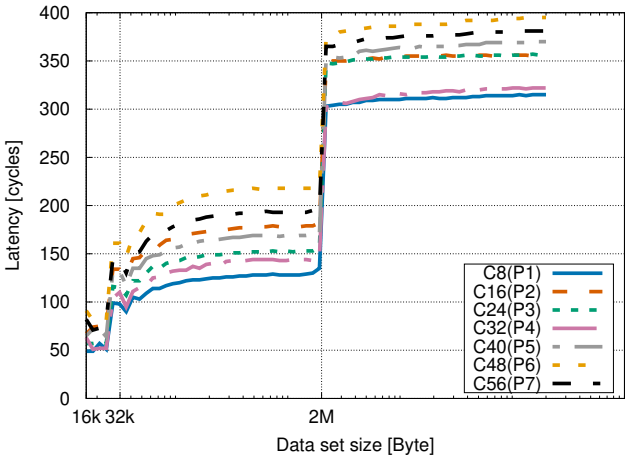


Fig.3. Read latency of C0 accessing the cache on the cores of different panels (P1–P7).

3.2 Core-to-Core Bandwidth

This subsection presents the read bandwidth on the Phytium 2000+ architecture. We measure the core-to-core sustainable bandwidth by continuously accessing a chunk of data elements [6]. Fig.4 shows the bandwidth of C0 loading cachelines which are modified, or shared in different cores and different cache levels. Note that, the exclusive and modified states are the same for the directory-based caching architectures. We measure the bandwidth of C0 loading data from its local cache, from C1 sharing a L2 cache with C0, from C4 on the same panel, and from C8 on a different panel.

In Fig.4, we find that the read bandwidth results show a clear phase change as the size of the dataset increases. Moreover, the size of the dataset when the staged change occurs is basically consistent with the size of various levels of cache. Compared with the first change point occurring exactly at 32KB (the size of L1 cache), the second change occurs earlier than 2M (the size of L2 cache). This is because the L1 cache is a pure data cache, while the L2 cache is a hybrid cache for both data and instructions.

3.2.1 *Local Cache Accesses*

Whatever the state of the cachelines, the data can be loaded from C0's local caches. The obtained bandwidth has nothing to do with the coherency state of the accessed data. The read bandwidth to its local L1 cache can reach 33.6 GB/s, while reading data from the local L2 cache can reach a bandwidth of 18.5 GB/s. Given that the L1 read port of Phytium 2000+ is 128 bits in width and runs at 2.2 GHz, we calculate the theoretical L1 read bandwidth as $2.2 \times 128 \div 8 = 35.2$ GB/s. We see that the measured bandwidth is quite close to its theoretical counterpart (33.6 GB/s vs. 35.2 GB/s). The measured write bandwidth stays about 17.4 GB/s for L1. We note that the write bandwidth is around a half of the read bandwidth. This is because storing data into L1 occurs at 64 bits per cycle, while loading data from L1 occurs at 128 bits per cycle.

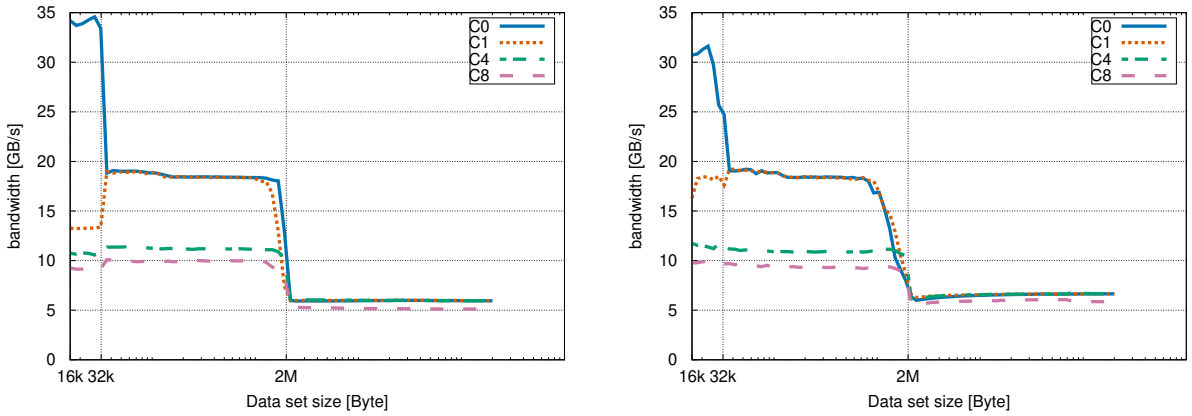
3.2.2 *Within a Core Group*

Differing from Intel's MESIF cache coherence protocol, Phytium 2000+ uses a home-grown Hawk cache coherence protocol to implement a distributed directory-based global cache coherency across panels. Note that the directory-based protocol cannot distinguish a data block cached in an exclusive or modified state.

Given that C1 and C0 shares the same L2 cache slice, data can be loaded from the local L2 cache when the cacheline is shared. And the memory bandwidth of accessing the local L2 can reach 18.5 GB/s. But the bandwidth is reduced to be around 13.3 GB/s when C0 loading exclusive or modified cachelines suited in C1's L1 cache. This is a notable difference from the x86 processor that the exclusive and modified states are treated the same on Phytium 2000+ .

3.2.3 *Within a Panel*

When C0 loads the data from C4 of the same panel, where the two cores share no common cache slices, the



(a) Modified (b) Shared

Fig.4. Read bandwidth of C0 accessing the local or another core (C1, C4 or C8).

bandwidth will be limited by the cross-group links. As can be seen from Fig.4, the bandwidth is significantly smaller (by around 40%) than the case when sharing the same L2 cache slice.

Similar to C1, when performing cross-group access to C4 for exclusive or modified cachelines, the bandwidth for reading the remote L1 cache is always smaller than that for accessing the remote L2. This is also because the data can be obtained directly from the L2 cache only when its state is shared initially.

3.2.4 Across Panel Accesses

C8 does not share a common L2 cache slice with C0, and the two cores have to be communicated via the cross-panel routing cells. The read bandwidth of C0 accessing C8 ranges from 9.2 GB/s to 9.7 GB/s, which is smaller than the bandwidth of accessing C1 or C4 within the same panel with C0.

3.2.5 NUMA Memory Accesses

Since C0, C1, C4 are within the same panel, they are connected directly to the same MCU and memory module. When accessing the data in the local memory module for C1 and C4, the bandwidth can reach around 6 GB/s. On the other hand, C8 is connected directly to another memory module. The bandwidth of C0 loading data from C8's memory module is around 5.1 GB/s.

To summarize, there is another difference between the Phytium 2000+ processor and the x86 processor when accessing the shared cachelines. The x86 processor uses an extension of the MESIF protocol, which requires the data to be fetched from the core with the latest copy (forward). Meanwhile, the Phytium 2000+ processor uses a MOSEI-like coherency protocol. There is no need to find the forward copy, but it can directly obtain the data with an arbitrary shared copy.

3.3 Overall Bandwidth

3.3.1 Aggregated Stream Bandwidth

We use the stream v5.9 benchmark suite (copy, scale, add, triad) to measure the aggregated memory bandwidth [7], where the array size is 200,000,000 (i.e., 4577.6 MB memory required) and each test is run 20 times. Also, we use separate benchmarks to measure the memory bandwidth for pure read and write operations. The read benchmark reads the data from an array A ($b = b + A[k]$). The write benchmark writes a constant value into an array A ($A[k] = C$). Note that A needs to be large enough (e.g., 1 GB) such that it cannot fit in the on-chip memory. To avoid the impact of "cold" TLBs, we start with two "warm-up" iterations

of the benchmarks, before we measure a third one. We use different numbers of running threads - from 1 to 64. Note that each thread is pinned to a hardware core in a sequential order with `GOMP_CPU_AFFINITY`.

The aggregated memory bandwidth of `read`, `write`, `copy`, `scale`, `add` and `triad` is shown in Fig.5, when using different numbers of cores. We see that the overall memory bandwidth increases over the number of used cores. It happens because when using more threads, we can generate more requests to memory controllers, thus making the memory channels busier. Thus, if aiming to achieve high memory bandwidth, programmers need to launch enough threads to saturate the interconnect and the memory channels. We also see that the maximum bandwidth for the six benchmarks is far below the theoretical peak of 153.6 GB/s. Specifically, the `read` bandwidth peaks at 100.42 GB/s and the `write` bandwidth peaks at 123.43 GB/s, achieved with 64 threads by pinning each thread to a fixed core.

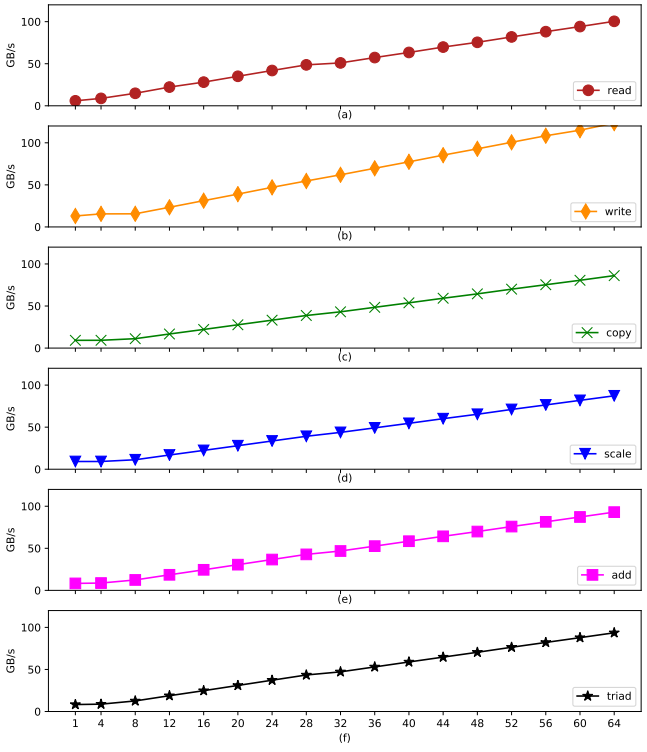


Fig.5. Memory bandwidth with the `stream` benchmark.

Fig.5 shows that, the achieved memory bandwidth increases very slightly, when using 1, 4, or 8 cores. This is because using less than 8 cores will load the data from the same memory channel and module. In such a case, the maximum `triad` bandwidth stays around 12.57 GB/s, which is 65.47% of the theoretical peak bandwidth. When using 64 cores, the achieved memory bandwidth reaches their maximum, with a bandwidth of 86.13 GB/s for `copy`, 87.27 GB/s for `scale`, 93.00 GB/s for `add`, and 93.57 GB/s for `triad`.

Fig.6. The NUMA-aware memory bandwidth with the `stream` benchmark (The ID of x-axis and y-axis denotes the ID of NUMA nodes, and the numbers on the heatmap represent the achieved memory bandwidth in GB/s).

3.3.2 NUMA-Aware Stream Bandwidth

We measure the NUMA-aware `stream` bandwidth on the `Phytium 2000+` processor with the `libnuma` library. This is achieved by running a `Pthread` version of the `stream v5.9` benchmark [7] and pinning 64 threads to eight NUMA nodes. Fig.6 shows the `triad` bandwidth of all pairs of NUMA nodes. We see that the bandwidth results obtained on the local NUMA node are the largest. The maximum bandwidth within the same nodes is 14.46 GB/s. With the help of the `libnuma` library, the aggregated `triad` memory bandwidth reaches around 103.96 GB/s (with 64 threads), which is around 10% larger than the measured number in subsection 3.3.1. On the other hand, the across-node bandwidth numbers are noticeably much smaller.

The cross-border accessing bandwidth can be reduced to only 10.16 GB/s on the `Phytium 2000+` processor.

3.3.3 Prefetching Effect

To evaluate the prefetching efficiency of `Phytium 2000+`, we use the Stanza Triad (`STriad`) [8] benchmark with a single thread. `STriad` works by performing a DAXPY (`Triad`) inner loop for a length L stanza, then jumps over k elements, and continues with the next L elements, until reaching the end of the array. We set the total array size to 1 GB, and set k (the jump length) to 2048 which is large enough to ensure no prefetch between stanzas, but small enough to avoid penalties from TLB misses and DDR precharge [8]. For each stanza, we run the experiment 10 times, with the L2 cache flushed each time, and calculate median value of the 10 runs to get the memory bandwidth for each stanza length.

Fig.7 shows the results of the `STriad` experiments on both `Phytium 2000+` and a regular Xeon processor (`Intel Xeon Gold 6130`). We see an increase in memory bandwidth over stanza length L, and it eventually approaches a peak of 8.4 GB/s on `Phytium 2000+` and 12.8 GB/s on Xeon. We conclude that the prefetching mechanism on both system does offer a significant benefit. Further, we see the transition point from the bandwidth-increasing state to the bandwidth-stable state appears at the same point when $L = 2^{10}$, but the Xeon processor can reach a much larger bandwidth. Therefore, we conclude that the non-contiguous access to memory is detrimental to memory bandwidth efficiency, with `Phytium 2000+` showing more restrictions on the stanza length of prefetching than the regular Xeons. To comply with this restriction, programmers have to create the longest possible stanzas of contiguous memory accesses, improving memory bandwidth.

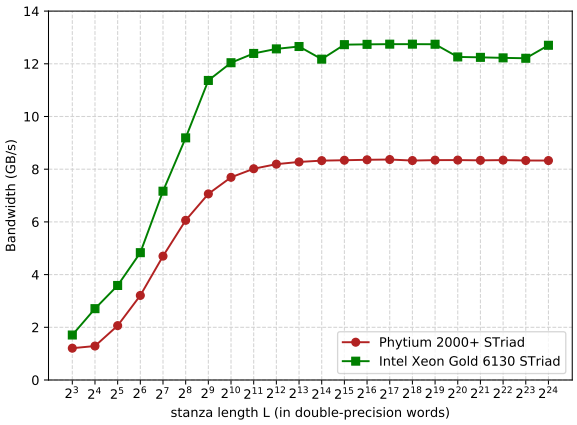


Fig.7. Performance of STriad on `Phytium 2000+` (the x-axis is in log scale and the results on `Intel Xeon Gold 6130` are normalized to those on `Phytium 2000+`).

4 Roofline Model Analysis

In this section, we use the well-known roofline model to estimate the `Phytium 2000+`'s performance for various scientific kernels. The roofline model is a visually-intuitive and throughput-oriented approach of characterizing a system's performance for various arithmetic intensities (floating-point operations per byte of DRAM traffic) of algorithms [9]. This visualization and mapping of performance to algorithms helps to identify and quantify the primary factors that limit the performance of a given application. Such type of "bound and bottleneck analysis" has the advantage of being more intuitive and user-friendly than traditional analytical performance models.

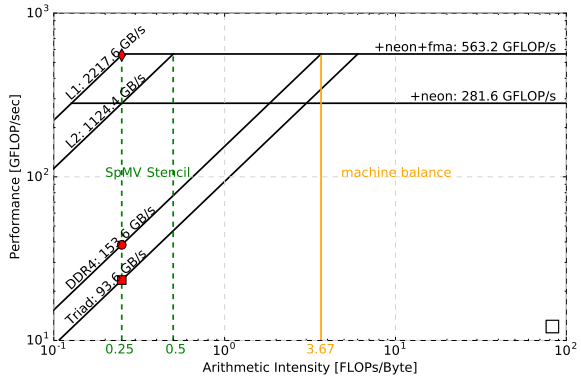


Fig.8. Roofline for the `Phytium 2000+` processor.

Fig.8 shows the roofline model specialized for the `Phytium 2000+` system. The system's double-precision performance is 563.2 GFLOP/s and and its peak DDR4 memory bandwidth is 153.6 GB/s. This gives a machine balance of 3.67 (denoted by the orange vertical line), which is calculated as the ratio of peak compute throughput to memory throughput. This ridge point, where the two lines meet, is the arithmetic intensity at which an algorithm goes from being memory bound to being compute bound on that system. To derive more practical performance numbers, we also use the sustainable `triad` memory bandwidth (93.6 GB/s).

`Phytium 2000+` can be represented as a set of cores with two-level on-chip caches connected to a DRAM memory. And using the original roofline modeling approach is not sufficient to fully capture the performance of `Phytium 2000+` which relies on the on-chip memory hierarchy. Ilic *et al.* further proposed a cache-aware roofline model [10], based on a core-centric concept where FP operations, data traffic and memory bandwidth at different levels are perceived. Based on the microbenchmarking results on the L1 and L2 caches (Section 3), we add two more cache-relevant rooflines to capture the memory accessing capabilities.

Using the roofline model, we can conduct a performance analysis for a variety of scientific kernels. As shown in Fig.8, for three kernels that are frequently used in scientific applications – sparse matrix-vector multiplication (SpMV), 3D stencil (Stencil), and general matrix-matrix multiplication (GEMM) – we can estimate a parallel performance upper-bound on the `Phytium 2000+` system. Note that, the arithmetic intensity of GEMM depends on the matrix size, and we do not show it in the figure. For example, for the SpMV kernel, whose operational intensity is around 0.25, we can expect a peak performance of 38.4 GFLOP/s (red circle) with 64 threads or cores. The above analysis, however, assumes the optimal memory bandwidth use of `Phytium 2000+` . Instead, if we use the sustainable `triad` memory bandwidth, then the expected performance would be 23.4 GFLOP/s (red square) instead. If the SpMV dataset is sufficiently small to be within the L1 cache (i.e., smaller than 64×32 KB), then the expected performance would be 554.4 GFLOP/s (red diamond) on the `Phytium 2000+` system.

Our results show that the `Phytium 2000+` system is optimized for data intensive applications with low operational intensities. `Phytium 2000+` has a system balance of 3.67, which suggests that it is well balanced. This is different from many other systems whose system balance typically ranges from 6 to 7 and which prefer increasing the number of processing units rather than the memory bandwidth [11]. By taking the difficult, but effective, strategy of keeping a "balance" between compute and memory throughputs, the `Phytium 2000+` system is capable of delivering excellent performance for a wide range of scientific kernels.

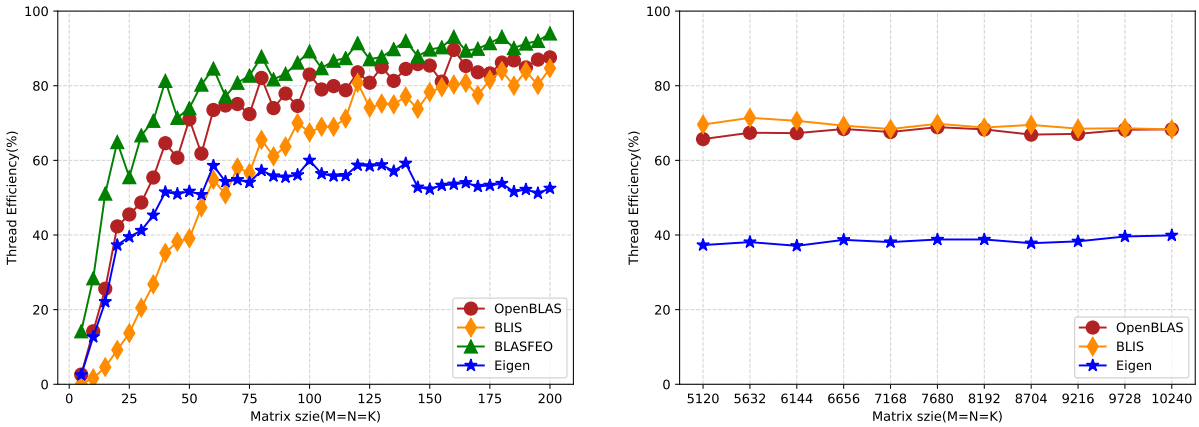
5 Applications

5.1 GEMM

General matrix-matrix multiplication (GEMM) is a common algorithm in linear algebra, machine learning, and many other domains. Given that large-scale GEMM and small-scale GEMM are used in different scenarios, this subsection demonstrates how well they perform on the `Phytium 2000+` architecture. We use four open-source GEMM libraries (OpenBLAS, BLIS [12], BLASFEO [13], and Eigen) for the evaluation.

5.1.1 Small-Scale GEMM

Fig.9(a) shows the performance results of small-scale SGEMM on the `Phytium 2000+` system. In our experiments, square matrices are used with their sizes



(a) Small SGEMM with a single thread

(b) Large SGEMM with 64 threads

Fig.9. The performance of small-scale and large-scale SGEMM.

ranging from 5 to 200, with a step of 5. Since the size of the matrix is small, we use a single thread to evaluate the performance of each BLAS library. The execution time is calculated as the average of 10 runs.

BLASFEO stands out as the best performer in nearly all the input sizes tested, reaching a maximum thread efficiency of 94%. This is because the matrix represented in BLASFEO is stored in a panel-major format, which has no data packing overhead, while the other libraries are stored in either row-major or column-major format. When the matrix dimension in BLASFEO is a multiple of 4, there occurs a ridge point for the GEMM performance, which is related to the index computation of elements in the panel-major format [13]. When the dimension of the matrix is not a multiple of 4, BLASFEO has to pad the data structure with zeros, resulting in an overhead.

OpenBLAS, BLIS and Eigen all suffer from the packing overhead. We see that OpenBLAS achieves better performance than the other two. This is due to the fact that it employs hand-crafted assembly-coded GEMM kernels. Eigen employs C++-coded GEMM kernels, which fails to schedule instructions, resulting in poor performance. Although both OpenBLAS and BLIS employ hand-crafted assembly-coded GEMM kernels, there exist differences in loop organization and register blocking strategies.

5.1.2 *Large-Scale GEMM*

Fig.9(b) shows the performance results of large-scale SGEMM on the `Phytium 2000+` system. Because BLASFEO does not have a multi-threaded version, we only measure the performance of OpenBLAS, BLIS, and Eigen. The matrix size ranges from 5120 to 10240, with a step of 512, and we use 64 threads.

The performance of OpenBLAS, BLIS and Eigen does not change significantly over the matrix size. BLIS runs slightly faster than OpenBLAS, reaching over 70% of `Phytium 2000+`'s theoretical peak. Eigen employs C++-coded GEMM kernel, leading to bad performance among the three BLAS libraries, achieving only 38% of the theoretical peak. According to the prior experimental results [14], the single-thread GEMM performance on `Phytium 2000+` can reach 91% of the theoretical peak. In terms of per thread GEMM performance, using 64 threads is 20% – 25% slower than using a single thread. This slowdown comes from the overhead of thread management and the shared L2 cache by every four cores on `Phytium 2000+` .

5.2 SpMV

Sparse matrix-vector multiplication (SpMV) is an essential kernel in linear algebra and is widely used in scientific and engineering applications. This subsection evaluates the performance of the SpMV kernel on the `Phytium 2000+` systems.

5.2.1 *Impact of NUMA Bindings*

As shown in Fig.8, SpMV is a memory-bound kernel with an arithmetic intensity of 0.25. Thus, memory accesses on `Phytium 2000+` have a direct impact on SpMV's performance. `Phytium 2000+` exposes eight NUMA nodes where a group of eight cores are directly connected to a local memory module. Indirect access to remote memory modules is possible but 1.5x slower than accessing the local module (Subsection 3.2.5). We use the Linux NUMA utility, `numactl`, to allocate the required data buffers from the local memory module for a thread that performs SpMV computation [15].

As can be seen from Fig.10, the NUMA-aware memory allocation significantly outperforms the non-NUMA-aware counterpart, giving an average speedup ranging from 1.5x to 6x across five storage formats. As such, we enable static NUMA bindings on the `Phytium 2000+` processor. We also observe that the ELL format consumes the largest memory buffers among the five storage formats, and thus can benefit the most from using manual NUMA bindings.

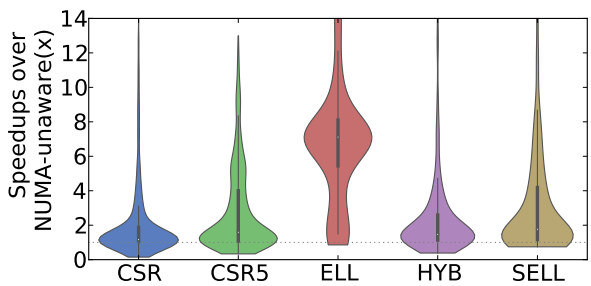


Fig.10. The diagram shows the speedup distribution of NUMA-aware memory allocation on `Phytium 2000+` (64 threads). The thick black line shows where 50% of the data locates.

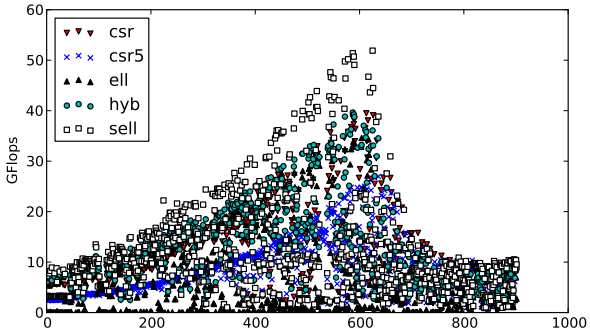


Fig.11. The overall SpMV performance on `Phytium 2000+` (64 threads). The x-axis labels different sparse matrices ordered by the number of nonzeros, and the y-axis denotes the achieved SpMV performance in GFlops.

5.2.2 *Overall SpMV Performance*

Fig.11 shows the overall SpMV performance on `Phytium 2000+` with five storage formats (CSR, CSR5 [16], ELL [17], SELL [18], and HYB [19]). Our estimation with the roofline model (Fig.8) indicates that the maximum SpMV performance with 64 threads is 23.4 Gflop/s. But the achieved performance is much larger than this estimated performance. This occurs when the input sparse matrix can be hold within the on-chip cache of `Phytium 2000+`.

We also see that there is no "one-size-fits-all" format across inputs. On the `Phytium 2000+` platform, SELL is the optimal format for around 50% of the sparse matrices and ELL gives the worst performance on most of the cases. As such we need an adaptive scheme to help developers to choose the optimal sparse matrix format [20–22].

6 Related Work

For the effective use of the memory systems on modern architectures, researchers have obtained their performance results and disclosed implementation details through measurements.

Babka *et al.* [23] proposed experiments that investigate detailed parameters of the `x86` processors. The experiment is built on a general benchmark framework and obtains the required memory parameters by per-

forming one or a combination of multiple open-source benchmarks. It focuses on detailed parameters including the address translation miss penalties, the parameters of the additional translation caches, the cacheline size, and the cache miss penalties.

McCalpin *et al.* [7] presented four benchmark kernels (Copy, Scale, Add, and Triad), `STREAM`, to assess memory bandwidth for a large variety current computers, including uniprocessors, vector processors, shared-memory systems, and distributed-memory systems. `STREAM` is one of the most commonly used memory bandwidth measurement tools in Fortran and C. But it focuses on throughput measurement without considering the latency metric.

Molka *et al.* [6] proposed a set of benchmarks, including to study the performance details of the `Nehalem` architecture. Based on these benchmarks, they obtained undocumented performance data and architectural properties. This is the first work to measure the core-to-core communication overhead, but it is only applicable to the `x86` architectures. Fang *et al.* extended the microkernels to Intel Xeon Phi [24]. Ramos *et al.* [25] proposed a state-based modelling approach for memory communication, allowing algorithm designers to abstract away from the architecture and the detailed cache coherency protocols. The model is built based on the measurement numbers of the cache-coherent memory hierarchy.

As for the `Phytium 2000+` architecture, there are a few related works on performance evaluation and optimization. You *et al.* [4] evaluated three linear algebra kernels such as matrix-matrix multiplication, matrix-vector multiplication and triangular solver with both sparse and dense datasets, aiming to provide performance indicators for the prototype Tianhe-3 cluster built from the `Phytium 2000+` architecture. Su *et al.* [14] presented a Shared Cache Partitioning (SCP) method to eliminate inter-thread cache conflicts in the GEMM routines on the `Phytium 2000+` architecture. This is achieved by partitioning a shared cache into physically disjoint sets and assigning different sets to different threads. Chen *et al.* evaluated and optimized the SpMV kernel from various angles [20–22].

Different from the previous work, we provide a systematic evaluation on the `Phytium 2000+` many-core architecture with a memory-centric perspective. This evaluation work is performed at both the microbenchmark level and the kernel level, guided by a `Phytium 2000+` specified roofline model.

7 Conclusion

This article presents a comprehensive performance evaluation of a 64-core ARMv8-based architecture. We focused on cache and memory subsystems, analyzing the characteristics that have an impact on high-performance computing applications. We provided insight into the relevant characteristics of the `Phytium 2000+` processor using a set of micro-benchmarks. We then analyzed the `Phytium 2000+` processor at the system level using the well-known roofline model. Using the knowledge gained from these micro-benchmarks, we optimized two applications and used them to assess the capabilities of the `Phytium 2000+` system. The results showed that the ARMv8-based many-core system is capable of delivering high performance for a wide range of scientific kernels. Our evaluation results also indicate that the shared L2 cache by four cores can be a performance bottleneck for GEMM and leveraging a private L2 per core is highly recommended for building future generations of `Phytium` processors.

Acknowledgement(s) We would like to thank the anonymous reviewers for their valuable and constructive comments. We thank Weiling Yang and Wanrong

Gao from National University of Defense Technology for the experiment support. This work is partially funded by the National Key Research and Development Program of China under Grant No. 2018YFB0204301, and the National Natural Science Foundation of China under Grant Nos. 61972408 and 61602501.

References

[1] M. A. Laurenzano, A. Tiwari, A. Cauble-Chantrenne, A. Jundt, W. A. W. Jr., R. L. Campbell, and L. Carrington, "Characterization and bottleneck analysis of a 64-bit armv8 platform," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*. IEEE Computer Society, 2016, pp. 36–45.

[2] N. Stephens, "Armv8-a next-generation vector architecture for HPC," in *2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, August 21-23, 2016*. IEEE, 2016, pp. 1–31.

[3] C. Zhang, "Mars: A 64-core armv8 processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015, pp. 1–23.

[4] X. You, H. Yang, Z. Luan, Y. Liu, and D. Qian, "Performance evaluation and analysis of linear algebra kernels in the prototype tianhe-3 cluster," in *Supercomputing Frontiers - 5th Asian Conference, SCFA 2019, Singapore, March 11-14, 2019, Proceedings*, ser. Lecture Notes in Computer Science, D. Abramson and B. R. de Supinski, Eds., vol. 11416. Springer, 2019, pp. 86–105.

[5] J. Dongarra, "Report on the fujitsu fugaku system," Tech. Rep. ICL-UT-20-06, June 2020.

[6] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*. IEEE Computer Society, 2009, pp. 261–270.

[7] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 12 1995.

[8] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. A. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *Proceedings of the 2005 workshop on Memory System Performance, Chicago, Illinois, USA, June 12, 2005*, B. Calder and B. G. Zorn, Eds. ACM, 2005, pp. 36–43.

[9] S. Williams, A. Waterman, and D. A. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[10] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Comput. Archit. Lett.*, vol. 13, no. 1, pp. 21–24, 2014.

[11] X. Liu, D. Buono, F. Checconi, J. W. Choi, X. Que, F. Petrini, J. A. Gunnels, and J. Stuecheli, "An early performance study of large-scale POWER8 SMP systems," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 263–272.

[12] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, 2008.

[13] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl, "BLASFEO: basic linear algebra subroutines for embedded optimization," *ACM Trans. Math. Softw.*, vol. 44, no. 4, pp. 42:1–42:30, 2018.

[14] X. Su, X. Liao, H. Jiang, C. Yang, and J. Xue, "SCP: shared cache partitioning for high-performance GEMM," *TACO*, vol. 15, no. 4, pp. 43:1–43:21, 2019.

[15] C. Hollowell, C. Caramarcu, W. Strecker-Kellogg, A. Wong, and A. Zaytsev, "The effect of NUMA tunings on CPU performance," *Journal of Physics: Conference Series*, vol. 664, no. 9, p. 092010, dec 2015.

[16] W. Liu and B. Vinter, "CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15*, L. N. Bhuyan, F. Chong, and V. Sarkar, Eds. ACM, 2015, pp. 339–350.

[17] R. Grimes, D. Kincaid, and D. Young, "Itpack 2.0 user's guide," Center for Numerical Analysis, University of Texas, Austin, Tech. Rep. CNA-150, 1979.

[18] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, 2014.

[19] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. ACM, 2009.

[20] D. Chen, J. Fang, C. Xu, S. Chen, and Z. Wang, "Characterizing scalability of sparse matrix-vector multiplications on phytium FT-2000+," *Int. J. Parallel Program.*, vol. 48, no. 1, pp. 80–97, 2020.

[21] D. Chen, J. Fang, S. Chen, C. Xu, and Z. Wang, "Optimizing sparse matrix-vector multiplications on an armv8-based many-core architecture," *Int. J. Parallel Program.*, vol. 47, no. 3, pp. 418–432, 2019.

[22] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, "Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures," in *20th IEEE International Conference on High Performance Computing, HPCC 2018*. IEEE, 2018, pp. 649–658.

[23] V. Babka and P. Tuma, "Investigating cache parameters of x86 family processors," in *Computer Performance Evaluation and Benchmarking, SPEC Benchmark Workshop 2009, Austin, TX, USA, January 25, 2009. Proceedings*, ser. Lecture Notes in Computer Science, D. R. Kaeli and K. Sachs, Eds., vol. 5419. Springer, 2009, pp. 77–96.

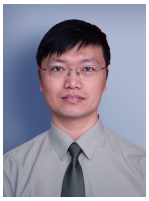
[24] J. Fang, H. J. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving intel xeon phi," in *ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014*, K. Lange, J. Murphy, W. Binder, and J. Merseguer, Eds. ACM, 2014, pp. 137–148.

[25] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent SMP systems: a case-study with xeon phi," in *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*, M. Parashar, J. B. Weissman, D. H. J. Epema, and R. J. O. Figueiredo, Eds. ACM, 2013, pp. 97–108.

Jian-Bin Fang is an assistant professor in computer science at National University of Defense Technology (NUDT). He obtained his Ph.D. from Delft University of Technology in 2014. His research interests include parallel programming for many-cores, parallel compilers, performance modeling, and scalable algorithms. He is a member of CCF.

Xiang-Ke Liao received his B.S. degree from Tsinghua University, Beijing, in 1985, and M.S. degree from National University of Defense Technology (NUDT), Changsha, in 1988, both in computer science. Currently he is a full professor of College of Computer at NUDT. His research interests include high performance computing systems, operating systems, and parallel and distributed computing. Prof. Liao is a fellow of CCF and an academician of Chinese Academy of Engineering.

Chun Huang is a full processor in computer science at National University of Defense Technology (NUDT). Her research interests are high-performance computing, system software, parallel compilers, parallel programming, performance optimization, and high-performance math libraries.



De-Zun Dong received his B.S., M.S., and Ph.D. degrees from the National University of Defense Technology (NUDT), Changsha, in 2002, 2004, and 2010, respectively. He is a professor in the College of Computer, NUDT, where he leads the research group of high-performance network and architecture (HiNA) and serves as the deputy director designer of Tianhe supercomputer. His research interests focus on high-performance network and architecture for supercomputer, datacenter and deep learning systems. He has published over 60 peer-reviewed papers in reputed international journals and conferences.