Performance Optimization of the HPCG Benchmark on the Sunway TaihuLight Supercomputer

YULONG AO, Chinese Academy of Sciences

CHAO YANG, Peking University & Chinese Academy of Sciences

FANGFANG LIU, Chinese Academy of Sciences

WANWANG YIN, National Research Center of Parallel Computer Engineering and Technology

LIJUAN JIANG and QIAO SUN, Chinese Academy of Sciences

In this article, we present some key techniques for optimizing HPCG on Sunway TaihuLight and demonstrate how to achieve high performance in memory-bound applications by exploiting specific characteristics of the hardware architecture. In particular, we utilize a block multicoloring approach for parallelization and propose methods such as requirement-based data mapping and customized gather collective to enhance the effective memory bandwidth. Experiments indicate that the optimized HPCG code can sustain 77% of the theoretical memory bandwidth and scale to the full system of more than 10 million cores, with an aggregated performance of 480.8 Tflop/s and a weak scaling efficiency of 87.3%.

CCS Concepts: • **Theory of computation** → **Massively parallel algorithms**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **General and reference** → Evaluation;

Additional Key Words and Phrases: HPCG, Sunway TaihuLight, heterogeneous many-core processor, performance optimization

ACM Reference format:

Yulong Ao, Chao Yang, Fangfang Liu, Wanwang Yin, Lijuan Jiang, and Qiao Sun. 2018. Performance Optimization of the HPCG Benchmark on the Sunway TaihuLight Supercomputer. *ACM Trans. Archit. Code Optim.* 15, 1, Article 11 (March 2018), 20 pages.

<https://doi.org/10.1145/3182177>

1 INTRODUCTION

As an important alternative to the current high-performance LINPACK (HPL) [11] benchmark for the TOP500 List, the high-performance conjugate gradient (HPCG) [10] benchmark exhibits computation patterns closer to real-world applications. Unlike HPL, which relies heavily on dense

This work was supported in part by the National Key R&D Plan of China (grant 2016YFB0200603), the Natural Science Foundation of China (grant 91530323), and the CAS Key Research Program of Frontier Sciences (grant QYZDB-SSW-SYS006).

Authors' addresses: Y. Ao, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China; email: aoyulong@outlook.com; C. Yang (corresponding author), CCSE, CAPT & NELVT, School of Mathematical Sciences, Peking University, Beijing 100871, China, and Institute of Software & State Key Laboratory of Computer Science, Chinese Academy of Sciences, Beijing 100190, China; email: chao_yang@pku.edu.cn; F. Liu, L. Jiang, and Q. Sun, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China, and University of Chinese Academy of Sciences, Beijing 100049, China; email: {lijuan2014, fangfang, sunqiao}@iscas.ac.cn; W. Yin, National Research Center of Parallel Computer Engineering and Technology, Beijing 100190, China; email: yinwanwang@gmail.com.

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

© 2018 ACM 1544-3566/2018/03-ART11 \$15.00

<https://doi.org/10.1145/3182177>

matrix computations, HPCG contains typical sparse matrix operations such as sparse matrix-vector multiplications (SpMV), symmetric Gauss-Seidel relaxations (SymGSs), and conjugate gradient iterations, which are all frequently found in many scientific computing applications. These operations are usually memory bound with low arithmetic intensities, irregular data access patterns, and neighboring and collective communications, and thus are challenging to optimize on modern supercomputing systems. Since its release, HPCG has been attracting rapidly increased research interests in the HPC community [20, 23, 24, 29–31].

Driven by the growing demand for extreme-scale computing, heterogeneous architectures with high computing throughputs and low energy costs play a key role in the design of supercomputers today. In the latest TOP500 list of June 2017 [26], half of the top 10 are based on heterogeneous architectures, including Sunway TaihuLight (rank 1) that uses the Chinese homegrown Shen Wei many-core processors, Tianhe-2 (rank 2) and Oakforest-PACS (rank 7) that employ the Intel Xeon Phi coprocessors, and Piz Daint (rank 3) and Titan (rank 4) that use NVIDIA GPUs. In particular, the Sunway TaihuLight supercomputer, hosted at the National Supercomputing Center in Wuxi, China, is designed with a ground-breaking peak performance of 125 Pflop/s and can achieve an outstanding HPL performance of 93 Pflop/s in double precision. It employs a heterogeneous architecture based on the new generation of homegrown SW26010 many-core processors in China and consists of more than 10 million lightweight heterogeneous cores. Because of the relatively high computing capability and the relatively low memory bandwidth, Sunway TaihuLight is especially powerful with computation-bound problems such as the dense matrix computations in HPL but may encounter troubles when dealing with memory-bound problems such as sparse matrix computations in HPCG. It is thus of interest to the community to have a systematic study of methodologies and techniques to improve the performance of HPCG on this leadership heterogeneous system.

In this work, we enable and scale the HPCG benchmark on the Sunway TaihuLight supercomputer to demonstrate how high performance can be achieved in the memory-bound HPCG benchmark by exploiting some advanced architecture features of the Sunway system. In particular, the following architecture-aware methods are applied to achieve significant performance improvement:

- *Block multicoloring parallelization*: To exploit the limited and fine-grain parallelism of the symmetric Gauss-Seidel kernel, we employ a block multicoloring reordering method to increase the degree of parallelism without severely degrading the convergence rate.
- *Locality-aware layout transformation*: To improve the data locality, the original data are rearranged in accordance with the parallelization scheme. An index conversion is further performed to limit the computation to the block level.
- *Requirement-based data access*: To efficiently utilize the limited fast scratch pad memory, we identify the data required by the computation and map them to the appropriate cores. In this way, the data access overhead can be greatly reduced and the memory bandwidth is improved dramatically.
- *Concurrent gather collective*: Data sharing through the main memory often leads to race conditions and incurs extra data-moving overheads among different cores on the many-core processor. Our proposed gather collective employs the low-latency on-chip register data communication mechanism provided by the processor to circumvent these obstacles and alleviate the bandwidth stress.
- *Fine-grain task overlapping*: We decompose the whole execution process within each computing core into smaller fine-grain tasks, and then the overlapping between the data access and other operations can be exploited as much as possible.

Fig. 1. General architecture of the SW26010 processor.

By applying the preceding methods together with other optimization techniques such as code transformation, SIMD vectorization, index compression, register message combination, and local data management, the optimized code is able to sustain up to 77% of the theoretical memory bandwidth and scale to the full system of more than 10 million heterogeneous cores, sustaining an aggregated HPCG performance of 480.8 Tflop/s and a parallel efficiency of around 87.3%.

The remainder of this article is organized as follows. In Section 2, we give an overview of the Sunway TaihuLight supercomputer, following which the basic algorithms of HPCG is introduced in Section 3. After that, we introduce in detail the proposed key optimization techniques of HPCG in Section 4. Test results of the optimized code, along with some further analysis, can be found in Section 5. Some related works are summarized in Section 6. The article concludes in Section 7.

2 BRIEF OVERVIEW OF SUNWAY TAIHULIGHT

The Sunway TaihuLight supercomputer is built based on a heterogeneous architecture and a customized integration approach with different levels. The computing nodes equipped with an SW26010 processor are the basic units of the computing system. Every 256 computing nodes are integrated into a tightly coupled super node using a fully connected crossing switch. The super nodes are connected to each other through the central switch network. Sunway TaihuLight has a very high aggregate peak performance of 125 Pflop/s in double precision and has been known as the world fastest supercomputer according to the recent TOP500 list of June 2017. The details of the system configuration and some related applications can be found in Fu et al. [12].

2.1 SW26010 Processor

The general architecture of the Chinese home-grown SW26010 processor is shown in Figure 1. The processor has four core groups (CGs), each of which includes one management processing element (MPE), one computing processing element (CPE) cluster with 64 CPEs, one protocol processing unit (PPU), and one DDR3 memory controller (MC). The processor works at a frequency of 1.45GHz and can deliver an aggregated peak performance of more than 3.06 Tflop/s in double precision. The aggregated memory bandwidth of all MPEs and CPEs of the four CGs

is 136.51GB/s. Each CG of the SW26010 processor accounts for a quarter of the aggregated performance and bandwidth, around 765.6 Gflop/s and 34.13GB/s, respectively. Therefore, the flop-to-byte ratio is around 22.43, which indicates that the processor is highly memory bound. All 64 CPEs in each CG are organized as an 8×8 mesh network equipped with low-latency register data communication channels for exchanging data with each other to compensate for the limited off-chip memory bandwidth. A mesh controller is also included to handle interrupts and synchronizations. These four CGs are connected via the network on chip (NoC). Each CG has its own memory space, which is connected to the MPE and CPE cluster through the MC. The PPU is responsible for handling the different types of request from the CPEs to the MPE. All processors are connected to each other through a system interface (SI).

The MPE is a fully functional 64-bit RISC core with a 256-bit vector unit, whereas the CPE is a reduced one with the same instruction set and vector width. Both of them work at the same frequency and support fused multiply-add (FMA) instructions. An MPE has a dual pipeline executing 8 flops/cycle, whereas a CPE has only one pipeline for float-point operations but another pipeline for integer operations and register communication. The theoretical peak performance of a MPE is $8 \text{ flops/cycle} \times 1.45 \text{ GHz} \times 2 = 23.2 \text{ Gflops/s}$, and the CPE cluster is $8 \text{ flops/cycle} \times 1.45 \text{ GHz} \times 64 = 742.4 \text{ Gflops/s}$. In terms of the memory hierarchy, each MPE has a 32KB four-way set-associative L1 instruction cache and a 32KB four-way set-associative L1 data cache, with a 256KB eight-way set-associative L2 cache shared by instruction and data. Each CPE has its own 16KB direct-mapped L1 instruction cache and 64KB Scratch Pad Memory (SPM). In addition, a 64KB direct-mapped L2 instruction cache is shared by the CPE cluster. The SPM of each CPE can be configured as either a user-programmable buffer or a software-emulated cache.

2.2 Parallel Programming Models

On the Sunway TaihuLight platform, both the MPE and CPE run a same Sunway Raise Linux system. Moreover, they share a unified address space, which can minimize the data movement when collaborating with each other. There are at least three possible programming models on this platform: MPI, MPI+OpenACC, and MPI+Athread. Here Athread, similar to the pthread library, is a lightweight effective thread library for users to fully exploit the parallelism of the CPE cluster. By using it, the sequential host program on the MPE can initiate parallel programs known as kernels, which run on the CPE cluster. The kernels will be executed with at most 64 lightweight effective threads (one thread per CPE), each of which runs the same sequential code. These threads can be synchronized with each other by using the fast hardware barrier on the CPE cluster. Note that in the Athread programming model, only one kernel can run on the CPE cluster at one time. The host program must make sure that the current kernel is finished before spawning another one.

To further improve the efficiency of memory bandwidth usage, we exploit three advanced architecture features provided by the Sunway processor: user-programmable SPM, direct memory access (DMA), and register data communication. The user-programmable SPM of each CPE, also called the *local device memory* (LDM), can be treated as a CPE's private memory and allows the user to explicitly manipulate it in the kernel. The DMA channel can be utilized in users' kernels to efficiently move data between the main memory and the LDM. Then instead of directly accessing data in the main memory from the CPE cluster, the kernels can access their data in the LDM with a much smaller cost. By using the low-latency on-chip register data communication mechanism, the data exchanging among different CPEs within a CG can be done without extra data movement between the CPE cluster and the main memory. Therefore, it provides users an opportunity to combine an individual small work set on each CPE together to form a much larger one on the whole CPE cluster.

Fig. 2. A 27-point stencil in HPCG.

Fig. 3. The four-level V-cycle of the MG in HPCG.

3 BASIC HPCG ALGORITHM

Proposed in 2013 [8], the HPCG benchmark focuses on measuring the capability of a supercomputer on solving a large sparse linear system. In particular, the linear system solved in HPCG arises from the discretization of an elliptic partial differential equation, $\nabla^2 u = f$, defined in a three-dimensional regular cubic domain with homogeneous Dirichlet boundary conditions. A second-order 27-point stencil is employed as illustrated in Figure 2, leading to a linear system with a symmetric and positive definite sparse matrix, which has 27 nonzero entries per row for the rows corresponding to the interior points and 7 to 18 nonzeros for the those on the boundary. Although the nonzero structure of the sparse matrix is simple and clear, it is prohibited in HPCG to take advantage of the regularity of the sparse matrix for avoiding the loss of generality.

In HPCG, the linear system $Ax = b$, resulting from a 27-point stencil discretization, is solved by the preconditioned conjugate gradient (PCG) method described in Algorithm 1. To accelerate the convergence of the PCG algorithm, a four-level V-cycle geometric multigrid (MG) method, as shown in Algorithm 2, works as the preconditioner M^{-1} .

ALGORITHM 1: Preconditioned Conjugate Gradient for $Ax = b$

```

input:  $A, b, x_0, it_{max}, \epsilon$ 
 $r_0 \leftarrow b - Ax_0$ 
for  $i = 1, 2, \text{ to } it_{max}$  do
     $z_i \leftarrow M^{-1}r_{i-1}$   $\triangleright$  MG
     $s_i \leftarrow (r_{i-1}, z_i)$   $\triangleright$  DOT
    if  $i = 0$  then  $p_i \leftarrow z_i$ 
    else  $p_i \leftarrow z_i + (s_i/s_{i-1})p_{i-1}$   $\triangleright$  WAXPBY
     $\alpha_i \leftarrow s_i / (p_i, Ap_i)$   $\triangleright$  SpMV, DOT
     $x_{i+1} \leftarrow x_i + \alpha_i p_i$   $\triangleright$  WAXPBY
     $r_{i+1} \leftarrow r_i - \alpha_i Ap_i$   $\triangleright$  WAXPBY
    if  $\|r_{i+1}\|_2 / \|r_0\|_2 \leq \epsilon$  then break  $\triangleright$  DOT
end
output:  $x_{i+1}$ 
  
```

ALGORITHM 2: V-cycle for $x^h = \text{MG}(A_h, x_0^h, b^h)$

```

if on the coarsest level then
  |  $x^h \leftarrow \text{SymGS}(A_h, 0, b^h)$   $\triangleright$  Bottom solver
else
  |  $x^h \leftarrow \text{SymGS}(A_h, 0, b^h)$   $\triangleright$  Presmoothing
  |  $r^h \leftarrow b^h - A_h x^h$   $\triangleright$  Residual
  |  $r^{2h} \leftarrow I_h^{2h} r^h$   $\triangleright$  Restriction
  |  $x^{2h} \leftarrow \text{MG}(A_{2h}, 0, r^{2h})$   $\triangleright$  Recursion
  |  $x^h \leftarrow x^h + I_{2h}^h x^{2h}$   $\triangleright$  Prolongation
  |  $x^h \leftarrow \text{SymGS}(A_h, x^h, b^h)$   $\triangleright$  Postsmoothing
end
return  $x^h$ 

```

In the MG preconditioner, the fine-to-coarse ratio is fixed to be two and the SymGS is used as the pre- and postsmoother, as well as the bottom solver. A simplified illustration of the MG preconditioner is presented in Figure 3, where only the XY-plane is depicted. Overall, the main operations performed in each iteration of the PCG loop include three vector dot-products (Dots), three BLAS-1 vector updates (WAXPYs), four SpMV, seven SymGSs, three MG restrictions, and three MG prolongations. HPCG uses a three-dimensional domain decomposition strategy for distributed parallel computing. The whole computational domain is partitioned into smaller subdomains of the same size, each of which is assigned to an MPI process. To better represent real-world applications, the size of each subdomain should be large enough so that the data accessed in the PCG iteration cannot fit into the local cache. There exist two types of MPI communications in HPCG: one is the global collective communication required in each vector dot-product; the other, required in SpMV and SymGS, is the neighboring communication for each subdomain to exchange halos.

In terms of computational cost, there are two major kernels, SpMV and SymGS, which play dominant roles in HPCG. They both suffer from the low arithmetic intensity and irregular memory access, and thus are bounded by the memory bandwidth according to the roofline model [37]. Specifically, SymGS can be written as

$$x \leftarrow (L + D)^{-1}(b - Ux), \quad \triangleright \text{forward-sweep}, \quad (1)$$

$$x \leftarrow (U + D)^{-1}(b - Lx), \quad \triangleright \text{backward-sweep}, \quad (2)$$

where D is the diagonal of the matrix, and L and U are the strict lower and upper triangular parts, respectively. From Equations (1) and (2), SymGS can be interpreted into one forward and one backward sparse triangular solver with another two SpMV-like operations that only involve the triangular parts. Therefore, considering the amount of the floating-point operations required, the ratio of SymGS to SpMV is around 2:1 (four triangular parts in SymGS as compared to a whole matrix used in SpMV). In addition to that, SymGS shares great similarity to SpMV with respect to the required data and the memory access pattern. Although SpMV is suitable for parallelization, it is particularly difficult to improve the parallelism of SymGS due to its inherent sequentiality among different matrix rows. Officially, as a benchmark, HPCG allows for using a user-defined ordering, such as the red-black reordering, to improve the degree of parallelism of SymGS. But the possibly increased number of PCG iterations caused by the reordering/coloring are counted as a penalty in the final performance results measured in terms of the sustained floating-point operations per second by HPCG.

4 OPTIMIZING HPCG ON SUNWAY TAIHULIGHT

For the Sunway TaihuLight system, the MPE usually conducts management, communication, and a small amount of computation tasks, whereas the CPEs are in charge of most of the computation tasks. Considering the heterogeneity between the MPE and the CPE, in this study we assign all computation to the CPE cluster of 64 CPEs, leaving the corresponding MPE to handle other noncomputational work. In HPCG, SymGS plays a centric role and may cause most difficulties on many-core based architectures because of its inherent data dependency and limited parallelism. To deal with these difficulties, we conduct systematic optimizations listed as follows. First, a block multicoloring parallelization scheme is applied to reduce the synchronization overhead and balance the trade-off of parallelism, convergence rate, and data locality. Then to achieve better locality, the data layout is rearranged in a continuous way in accordance with the parallelization scheme. Further, only the required blocks are identified and mapped to the corresponding CPEs, which will keep the effective work set stayed within the limited LDMs to avoid extra data movement. And a concurrent gather collective based on the register data communication on the CPE cluster is proposed to efficiently exchange and share data during the computation. Moreover, the possibility of hiding the memory access is fully exploited by the overlapping technique, after decomposing the processing within one CPE into fine-grain tasks. In addition to the techniques mentioned previously, some other techniques are presented for further performance tuning. It is worth pointing out that most optimization techniques we employ to SymGS, possibly after some minor modifications, are also applicable to SpMV and other kernels in HPCG.

4.1 Block Multicoloring Parallelization

Extensive studies have been done on the parallelization of sparse triangular solvers frequently found in preconditioned iterative methods; some related works are summarized in Section 6. Taking into account platform-specific factors such as the architecture features and programming models, we employ a block multicoloring strategy [15, 20, 30] for the parallelization of SymGS, as shown in Figure 4. Compared to other methods, this approach has several advantages. First of all, using the blocking technique can make the required data of each block reside in the limited LDM space, so as to avoid access to the main memory. Moreover, the computation within each block can be done in a continuous way to maintain a good locality. The block multicoloring method can improve the degree of parallelism significantly with moderate synchronization overhead, but at the cost of partially breaking the overall dependency among the unknowns. In addition, it can still achieve fast convergence under the condition that larger blocks instead of larger numbers of colors are utilized. More discussion can be found in the discussion [15].

The block multicoloring parallelization scheme consists of three modules, namely the block multicoloring module, the scheduling module, and the multithreading module. The separation of these modules makes the implementation and optimization more flexible without much interference with each other. In the block multicoloring module, a directed dependency graph derived from the sparse matrix is partitioned into blocks by using a graph partitioning method, such as the one in METIS [17] or Iwashita et al. [15]. Then a graph coloring process is applied to the generated blocks, during which the dependency among the blocks are ignored. In particular, there will be eight colors after the graph coloring process in HPCG. After that, the blocks labeled with the same color can be processed in parallel and the nodes within each block are processed sequentially in a natural order. In the scheduling module, a task scheduler is designed to distribute the colored blocks to the CPEs equally. Since the CPE cluster only has 64 cores, the blocks with the same color may need to be divided into several parallel groups, each of which will occupy 64 CPEs at most. It is easy to understand that load balance can be maintained among the CPEs as long as the CPEs

Fig. 4. The three modules of the block multicoloring method for the parallelization of SymGS.

process the same number of blocks with the same size. Therefore, given the limited capacity of LDM, each CPE will only accommodate one block. We make the block as large as possible in each parallel group to avoid possible performance degradation. In this work, the size of each block is fixed to be 64 unknowns. Finally, in the multithreading module, the processing of these blocks within each parallel group is accomplished on the CPE cluster by spawning multiple threads in parallel, each of which is responsible for one block per CPE. When processing each block on a CPE, further optimizations can be done to improve the performance of computation and data access.

4.2 Locality-Aware Layout Transformation

To achieve high performance with the block multicoloring scheme, the underlying data storage formats are transformed with respect to the computation process. The original sparse matrix storage scheme used in HPCG greatly resembles the popular compressed sparse row (CSR) format [36], in which only the nonzero values and their corresponding column indices are stored row by row with an additional array keeping the number of nonzeros of each row. As shown in Figure 5, HPCG originally allocates memory for the nonzero values (also the column indices) within each row separately and uses a pointer array to store their initial elements' addresses. To avoid the memory segmentation and improve the data locality, we instead employ the ELLPACK (ELL) format [18], analogous to the efforts on Intel processors [30] and the K computer [20]. First, we pad each row of the nonzero values and column indices to the length of the rows with the most nonzero elements and then allocate them in a continuous way, as seen in Figure 5. Then these matrix data are reordered to be consistent with the process of the block multicoloring

Fig. 5. Locality-aware layout transformations for sparse matrix A , solution vector x , and right-hand-side vector y .

parallelization as mentioned earlier. In addition to that, we only group the identifiers of the blocks with the same color together without reordering the actual data so that our approach can be easily adapted to different coloring schemes.

In addition to the layout transformation of the data related to the sparse matrix, the vectors such as the solution vector x and right-hand-side vector y are also blocked and reordered in a corresponding way, which is also shown in Figure 5. When limiting the computation of each block in their own ranges completely, the original column indices from the scope at the process level should be converted to the indices at the block level. In each block, the indices of the nonzeros on the diagonal of the matrix will fall into the range of their own block and can be directly mapped into $[0, \text{blk_size})$, where blk_size represents the number of rows in each block. However, the other indices corresponding to nonzeros off the diagonal will go beyond the range of the current block. To deal with these indices, a mapping structure with an additional counter, which is initialized to blk_size , is employed to record the mapping between the original process-level indices and the converted block-level ones. If an original index is found in the map, the corresponding value is directly used as its converted block index. Otherwise, the counter is incremented by one and used as the converted block index. Based on the preceding discussion, the values in vector x corresponding to the column indices of a block can be divided into two parts: the interior part belonging to the range of its own block and the exterior part from other blocks.

4.3 Requirement-Based Data Access

Data reuse of the sparse matrix A in HPCG is difficult on Sunway TaihuLight, because the matrix data are accessed only once in SpMV and the capacity of the LDM is too small to hold the necessary data for splitting-based data reuse [19] in SymGS. This observation also applies to the right-hand-side vector b . Therefore, for the two kernels, the solution vector x is the only one that can be potentially reused. For each block, the data of vector x consist of two parts, namely the interior part and the exterior part, corresponding to the data belonging to the interior of the subdomain and the exterior of it, respectively. For efficiency purposes, these two parts can be treated separately. The interior part can be simply dealt with by transferring the data to the LDM by DMA, so as to maximize the usage of fast LDM. As for the exterior part, we can either directly access the data from the main memory or reuse the data from other blocks on the CPE cluster through the register data communication. The former method uses the original process-level column indices without changing the data structure, whereas the latter requires accessing the block-level indices with

Fig. 6. Requirement-based data access method for the solution vector x .

the help of the conversion process presented in Section 4.2. The cost of the first method is high because of the much longer latency for accessing data from the main memory instead of the LDM. As compared to the first method, the second one is considerably promising, as it avoids the direct access to the main memory by taking advantage of the fast on-chip data exchange mechanism.

Even though the LDM can be treated as a user-controllable *cache*, its capacity is still too small to accommodate the whole x vector in HPCG. Considering that the computation done on a data block only depends on a few other blocks of x , it is possible to only keep the required data blocks of x in the LDMs. To that end, we propose a requirement-based data access method for managing x , as depicted in Figure 6. When processing the blocks in a same parallel group, dependent blocks of x are identified based on their nonzero positions in advance and mapped to the LDMs through DMA. After that, the concurrent on-chip gather collective, which will be introduced later, is used for all CPEs to gather their exterior parts of x from other blocks on different CPEs simultaneously. If there are too many required blocks of x to fit in the LDMs in a same parallel group of SymGS, the mapping and collective should be invoked multiple times. We remark that the semantic meanings of both the data partition and the customized gather collective, although only discussed for thread-level parallelism within the CPE cluster, share some similarities with the process-level domain decomposition and message reduction using MPI.

4.4 Concurrent All-to-All Gather Collective

Exchanging and sharing data among different CPEs through the main memory is a time-consuming process. Fortunately, the register data communication mechanism provided by the Sunway processor enables each CPE to send/receive data efficiently to/from its neighbors in the same row or column. The two-sided communication can be treated as a synchronous producer-consumer relationship in a peer-to-peer or broadcast mode. To be specific, after loading the data from the LDM into a register variable, users can use the command `putr` (or `putc`) to send the data to the other CPEs within the same row (or column) of the 8×8 mesh. Meanwhile, the CPEs can use the command `getr` (or `getc`) to retrieve the data into a register variable from the same row (or column) and then store them into the LDM. The producer CPE has to be blocked

Fig. 7. Pseudocodes and examples for the request stage of the customized concurrent gather collective.

after sending data through the register to the mesh until the corresponding consumer CPEs have received them. Based on the low-level instructions, we design a customized concurrent gather collective that can be called by each CPE to obtain the required data from other CPEs.

This collective routine is implemented in a two-stage manner, each of which further consists of two steps. In the request stage, each CPE sends its request messages to the target CPEs. In the response stage, the target CPEs send the response messages that contain the required data back to the source CPEs based on the collected request information from the first stage. The pseudocodes and examples for these two stages are shown in Figure 7 and Figure 8, respectively. It is worth noting that the pseudocodes are not the actual codes we implement and only used for

Fig. 8. Pseudocodes and examples for the response stage of the customized concurrent gather collective.

demonstration purposes. The format of the request message is $\langle \text{src_id}, \text{src_pos}, \text{dest_id}, \text{dest_pos} \rangle$, which means that the CPE src_id requires the value in the position dest_pos on the LDM of the target CPE dest_id , and it will be stored in the position src_pos on its own LDM after getting the value. The format of the response message is almost the same, except dest_pos is replaced by val to contain the actual value required by the source CPE. The dest_id and dest_pos information for every entry in the exterior part of the vector x are encapsulated in the ext_info list, which has been prepared along with the process of data layout transformation in Section 4.2. The coordinate information for each CPE of the 8×8 CPE cluster, such as my_id , my_row , and my_col , can be directly obtained through the Athread library interface. In addition, the receiver will set the not_finished variable to be false when receiving the finish message from the sender; we omit the detail here for brevity.

Fig. 9. Different ways to organize the fine-grain tasks in each parallel group.

4.5 Fine-Grain Task Overlapping

On the Sunway many-core processor, the computation of the SymGS kernel is broken into several parallel groups. According to the analysis in Section 4.1, every CPE processes a data block in each parallel group, and the major operations can be categorized into four types of fine-grain tasks, namely the CPE computation, the DMA read, the DMA write, and the concurrent gather collective. These tasks, as shown in the upper panel of Figure 9, are executed in a sequential way. First, both the values of the interior part of x and the location information of the exterior part of x (i.e., `ext_info`) required by the current block are loaded into the LDM from the main memory by DMA read. Next, all CPEs call the gather collective to gather the values of the exterior parts of x based on the location information. Then all other required data, such as the matrix data, including values and column indices, and vector data, including the matrix diagonal vector and the right-hand-side vector, are loaded into the LDM by DMA. After that, the computation can be started immediately when all data for the block are ready in the LDM. Finally, the results are written back to the main memory before the next parallel group starts. To achieve higher performance, we overlap the fine-grain tasks as much as possible. As illustrated in the lower panel of Figure 9, right after the interior part and the location information are read into the LDM, the gather collective based on them can be initiated immediately without waiting. At the same time, the DMA transfer of other matrix data and vectors can be started simultaneously. In this way, the cost of the gather collective can be hidden so that the overall performance can be further improved.

4.6 Miscellaneous Tuning Details

In addition to the optimizations mentioned previously, we briefly mention some other optimizations we use, such as code transformation, SIMD vectorization, index compression, register message combination, and LDM data management. For code transformation, we replace the recursive implementation of the V-cycle multigrid in HPCG with an equivalent iterative form to avoid the overhead caused by function invocations. For SIMD vectorization, the matrix data stored in the ELL format are sliced with respect to the vector width and transposed for efficient vectorization with aligned memory access. For index compression, we replace the integer data type for column indices with a shorter type when the indices are converted into the block range, so as to alleviate the memory traffic. For message combination, several messages in the concurrent gather collective can be combined together into a larger message since the 256-bit vector register can hold multiple messages. For LDM data management, we not only eliminate the unnecessary data movement for the consecutive steps of SymGS but also pin the data in the LDM when the dataset is small enough on the coarse levels.

Due to the similarity to SymGS, the SpMV kernel employs all of the preceding optimizations for SymGS, except it is no longer necessary to distinguish the data blocks by using the multicoloring approach. As for the vector operations that only involve continuous memory access, such

as WAXPBY and Dot, the parallelizations can be straightforwardly conducted on the CPE cluster: the blocked vectors are evenly distributed and assigned to each CPE and then independently processed until done. Other vector operations, such as the prolongation and restriction, are adjusted to be continuous and compact to avoid indirect or loose memory access that usually leads to poor performance. The subroutines in the problem optimization phase of HPCG, including blocking and reordering kernels, are also optimized for better performance. Since HPCG version 3.0, the time spent on the setup phase, which includes the generation of the sparse matrix and the construction of the data structure for halo exchange, are also counted as an overhead in the final performance calculation. We therefore also parallelize and optimize the setup phase on the CPE cluster. In addition, the packing and unpacking of MPI messages for the halo exchange are accelerated on the CPE cluster to reduce the cost of the MPI communication at large scales.

5 PERFORMANCE AND ANALYSIS

The Sunway TaihuLight supercomputer, running with a 64-bit Linux OS, is equipped with a customized compiler supporting C/C++ and Fortran. The message passing library supports the MPI 3.0 specification and has been tuned for large-scale parallel run. In practice, we assign one MPI process to each CG of the processor and assign the compute-intensive work to the CPE cluster with 64 threads spawned. According to the memory capacity of the system, we fix the size of the local subdomain owned by each CG to be $128 \times 128 \times 192$.

5.1 Single CG Results

To investigate the performance of the optimized code with different optimization techniques applied, we carry out several tests on a single CG with only one MPI process. Starting from the reference HPCG code (noted as MPE), which can only run on the MPE sequentially, we incrementally apply various optimizations to enable and exploit the computing capabilities of the CPE cluster of 64 CPEs. First, as mentioned earlier, the many-core parallelization based on the block multicoloring method slightly breaks the original dependency of the SymGS process and requires 56 iterations to achieve the same level of residual as the MPE version does with 50 iterations. Due to the rules of HPCG, the increased iterations have been counted here as a penalty of the final performance. Then to examine the effects of managing the data of vector x with different methods, we test the two approaches discussed in Section 4.3 (noted as MEM and REG, respectively). All other optimizations, including the fine-grain task overlapping presented in Section 4.5 (noted as +ASYN) and other miscellaneous tuning details in Section 4.6 (noted as +MISC), are applied to the REG version in an accumulative manner. The test results in terms of the achieved gigaflops per second and the memory bandwidth ratios are shown in Figure 10.

From Figure 10(a), we observe that the performance of SpMV is better than SymGS most of the time, which is because SymGS requires several sweeps depending on the multicoloring result. As expected, the final HPCG performance is lower than both SpMV and SymGS on the finest level, because of the inclusion of some vector operations and the penalty of convergence and optimization setup. Relatively, the performance of SymGS on the finest level is much closer to the final HPCG performance than that of SpMV due to the dominant role played by SymGS. Compared to the basic MPE version, the performance of the MEM version that relies on direct accessing of the main memory can deliver around 1.9X speedup. However, the REG version that shares the x vector on the CPE cluster based on the register communication can gain around 9.6X speedup. This substantial performance improvement is due to the greatly increased data locality and memory efficiency. Moreover, on top of the REG version, the asynchronous data transfer of the +ASYN version with fine-grain task overlapping provides another 1.3X performance increase. With all other miscellaneous tuning methods further applied, the final version of +MISC leads to around 3.45 Gflop/s

Fig. 10. Performance of SpMV, SymGS, and HPCG on different grid levels with various optimizations.

performance and gives an overall speedup of 14.4X compared to the MPE version. The improvement of SymGS in the +MISC version mainly comes from removing the conditional branches and vectorizing the data initializations on the CPE cluster, whereas SpMV of the +MISC version is the same as that of the +ASYNC version. In terms of the bandwidth utilization ratio, the results shown in Figure 10(b) strongly indicate the memory-bound nature of HPCG. It is therefore easy to understand that on a single CG of the Sunway TaihuLight platform, the final +MISC code achieves the best performance, which is around 77% of the theoretical bandwidth, among all tested versions. Based on the preceding observations and analysis, we employ the final +MISC version of HPCG in the large-scale tests.

5.2 Full-System Results

Next, we conduct the weak-scaling tests of HPCG on the Sunway TaihuLight supercomputer. As in the single CG case, at the full-system scale, the optimized HPCG code also requires 56 PCG iterations to achieve the same level of residual as the reference code does with 50 iterations, leading to a performance penalty of around 10.7%. In addition, the preprocessing work, which is used for generating the problem and user-controlled optimized data structure, results in another 1.26% drop in the final performance. The weak scaling results of the final HPCG performance, including the aggregate performance and the parallel efficiency, are shown in Figure 11. It is seen from the figure that the optimized code scales well up to 160,000 CGs (over 10.4 million cores) with an aggregate HPCG performance of 480.8 Tflop/s and a parallel efficiency of around 87.3%. This result has been accepted by the HPCG list of June 2017 [9].

To further examine the communication overhead, we measure the costs of both the halo exchange and the global collective, and record them in Figure 12. Observations are made from the figure that the communication overhead is kept to a low level at the extreme scale. For the full-system tests, the overheads of the halo exchange and the global collective with respect to the overall HPCG runtime are only around 7.3% and 5.0%, respectively.

For comparison purposes, Table 1 summarizes the HPCG results on several other systems, collected from both published results [20, 30, 31] and the official HPCG list of June 2017 [9]. It can be seen that although the HPCG-to-HPL ratio of the Sunway platform is relatively low because of the highly limited data-moving capability, the Flop/Byte efficiency [37], which measures the ratio of the HPCG performance to the total memory bandwidth, is comparable to other systems.

6 RELATED WORK

SpMV is one of the most frequently found kernels in scientific computing and large-scale data analysis. Extensive research has been done to enhance the performance of SpMV by proposing

Fig. 11. Weak scaling results of HPCG, from a single MPI process to 160,000 MPI processes.

Fig. 12. Communication overhead of the halo exchange and the global collective.

Table 1. HPCG Performance Comparison Among Various Platforms

HPCG Rank	System	HPCG (Gflop/s)	HPCG/HPL	Iterations	HPCG	Bandwidth	Flop/Byte
					per Proc (Gflop/s)	Per Proc (GB/s)	
1	K	602.7	5.7%	50	7.26	64	0.113
2	Tianhe-2	580.1	1.7%	57	12.59	320	0.039
3	Sunway	480.8	0.5%	56	3.01	34	0.088
8	Titan	322.3	1.8%	55	17.28	250	0.069

or studying different storage formats for sparse matrices. For example, a variety of well-known formats such as CSR, ELL, and COO were implemented and tested for GPUs [6]. Based on the basic formats, some new storage formats were derived, such as the CSR-based [21], ELL-based [22], and COO-based [38] formats. There are also many different optimization techniques suitable for sparse matrix computations, including the thread and cache blocking methods [36], the data compression approaches [16], and the automatic performance tuning techniques [34]. Based on the characteristics of both the algorithm of HPCG and the architecture of the Sunway processor, we utilize the sliced ELL sparse matrix format [2] and employ some popular optimization techniques, such as the blocking and vectorization approaches in HPCG.

Efficient parallel algorithms for solving sparse triangular linear systems in preconditioned iterative solvers have been studied by many researchers. Generally, these works can be classified into two categories: level scheduling [4, 25, 28, 29] and color ordering [1, 14, 15, 32, 33]. Both of them can be described based on a directed acyclic graph (DAG) constructed from the nonzero pattern of the sparse matrix. With the assistant of DAG, the procedure in the level scheduling method can be described by traversing the graph via level sets: those unknowns on the same level are independent and can be performed in parallel as long as their dependent levels have been accomplished. The multicoloring method can be considered as a graph coloring problem—that is, the unknowns of the DAG are divided into different groups after being labeled by different colors, and those unknowns of the same group can be solved in parallel. In particular, a synchronization sparsification technique [29] was proposed to eliminate the redundant dependency in the level scheduling method. To further exploit the parallelism of level scheduling, the matrix can be partitioned into blocks before being scheduled [25]. Recently, a work on the block multicolored reordering method was conducted for unstructured problems [15], which extended an early work [14] on the structured problems. The block multicoloring parallelization scheme that we employ on Sunway TaihuLight is largely based on these works.

Among many solution methods or preconditioning techniques, the multigrid methods have been known as a popular choice in the past two decades, due in large part to its optimal convergence property for a large variety of problems. An overview of the various approaches for the parallelization of multigrid algorithms can be found in Chow et al. [7] and Hülsemann et al. [13]. To optimize the communication of message passing, a hierarchical coarse grid aggregation was studied [27]. To circumvent the data movement obstacle on the GPU, a block-asynchronous iteration scheme was designed to reduce synchronization and increase floating-point operations [3]. A variety of optimization techniques, such as communication avoiding, SIMDization, and operator fusion were studied for the MG method on multicore processor systems and the Intel Xeon Phi coprocessor [35]. An autotuning compiler technology was employed to automate communication-avoiding optimizations for the smooth operator in the MG [5]. The preceding works play an important role in helping us choose the parallelization strategies and optimizations of the major operations in the multigrid algorithm of HPCG.

HPCG has drawn increasing attention from both academics and industry since its announcement in 2013. For example, a multicolor reordering technique was employed to improve the performance of HPCG on CPU-GPU heterogeneous clusters [31]. The optimization of HPCG on the K supercomputer was done in Kumahata et al. [20], where a block multicoloring method was employed for the parallelization of SymGS. In their new work, the performance has been improved dramatically by reusing the sparse matrix data in SymGS after splitting the matrix into two subtriangular matrices [19]. To achieve better performance on the finest multigrid level, a sparsification technique [29] was proposed to avoid the convergence penalty on systems built upon Intel Xeon Phi coprocessors, such as Stamped and Tianhe-2. To further improve from the work [30], in which only the Xeon Phi devices were used for running HPCG on Tianhe-2, a

hybrid CPU-MIC algorithm [23, 24] was proposed to enable the cooperative computing among heterogeneous resources. In our work, considering the hardware characteristics, we employ the block multicoloring strategy for parallelization and design new algorithms to alleviate the bandwidth requirement stress on Sunway TaihuLight.

7 CONCLUSION AND FUTURE WORK

As a newly proposed benchmark, HPCG aims to better correlate real-world applications in scientific and engineering computing that rely on memory-bound kernels such as sparse matrix computations. Optimization of the HPCG benchmark is challenging on modern supercomputing systems, especially on the heterogeneous many-core architectures, which have the gap between the irregular data access patterns and the limited memory bandwidth. In this article, we enable and scale the HPCG benchmark on the world's newest and largest heterogeneous supercomputer, Sunway TaihuLight, and demonstrate that a high level of memory-bound HPCG performance can be achieved despite the significant mismatch between the bandwidth requirement and the architecture characteristics. By applying architecture-aware optimization techniques (i.e., block multicoloring reordering, locality-aware layout transformation, requirement-based data access, concurrent gather collective, and task overlapping approaches), the most challenging and time-consuming SymGS kernel can be efficiently parallelized with a high utilization of the limited bandwidth. In addition to that, other major kernels in HPCG are also optimized to exploit the parallelism on the Sunway system. Although the current work focuses on HPCG, it may shed light on other typical scientific and engineering applications with memory-bound natures. Some of the key methodologies that we propose are also potentially generalizable to other heterogeneous platforms.

REFERENCES

- [1] Mark Adams. 2014. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems*. Technical Report LBNL-6630E. eScholarship.
- [2] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'10)*. 111–125. DOI: http://dx.doi.org/10.1007/978-3-642-11515-8_10
- [3] Hesham Ali, Yong Shi, Deepak Khazanchi, Michael Lees, G. Dick van Albada, Jack Dongarra, Peter M. A. Sloot, et al. 2012. Block-asynchronous multigrid smoothers for GPU-accelerated systems. *Procedia Computer Science* 9, 7–16. DOI: <http://dx.doi.org/10.1016/j.procs.2012.04.002>
- [4] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 01, 73–95. DOI: <http://dx.doi.org/10.1142/S0129053389000056>
- [5] Protonu Basu, Anand Venkat, Mary Hall, Samuel Williams, Brian Van Straalen, and Leonid Oliker. 2013. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *Proceedings of the 2013 20th International Conference on High Performance Computing (HiPC'13)*. IEEE, Los Alamitos, CA. DOI: <http://dx.doi.org/10.1109/HiPC.2013.6799131>
- [6] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis (SC'09)*. ACM, New York, NY, 18:1–18:11. DOI: <http://dx.doi.org/10.1145/1654059.1654078>
- [7] Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike Meier Yang. 2006. A survey of parallelization techniques for multigrid solvers. In *Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 179–201. DOI: <http://dx.doi.org/10.1137/1.9780898718133.ch10>
- [8] Jack Dongarra and Michael Heroux. 2013. *Toward a New Metric for Ranking High Performance Computing Systems*. Technical Report. Sandia.
- [9] Jack Dongarra, Michael Heroux, and Luszczek Piotr. 2017. HPCG Results: ISC'17. Available at <http://www.hpcg-benchmark.org/custom/index.html?id=155&slid=291>.
- [10] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. 2015. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications* 30, 1, 3–10. DOI: <http://dx.doi.org/10.1177/1094342015593158>

- [11] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience* 15, 803–820. DOI : <http://dx.doi.org/10.1002/cpe.728>
- [12] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, et al. 2016. The Sunway TaihuLight supercomputer: System and applications. *Science China Information Sciences* 59, 1–16. DOI : <http://dx.doi.org/10.1007/s11432-016-5588-7>
- [13] Frank Hülsemann, Markus Kowarschik, Marcus Mohr, and Ulrich Rüde. 2006. Parallel geometric multigrid. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, Berlin, Germany, 165–208. DOI : http://dx.doi.org/10.1007/3-540-31619-1_5
- [14] Takeshi Iwashita, Yuuichi Nakanishi, and Masaaki Shimasaki. 2005. Comparison criteria for parallel orderings in ILU preconditioning. *SIAM Journal on Scientific Computing* 26, 1234–1260. DOI : <http://dx.doi.org/10.1137/03060076X>
- [15] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhiro Takahashi. 2012. Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in ICCG method. In *Proceedings of the 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS'12)*. 474–483. DOI : <http://dx.doi.org/10.1109/IPDPS.2012.51>
- [16] Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2013. An extended compression format for the optimization of sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 24, 1930–1940. DOI : <http://dx.doi.org/10.1109/TPDS.2012.290>
- [17] George Karypis and Vipin Kumar. 1998. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing* 48, 1, 96–129. DOI : <http://dx.doi.org/10.1006/jpdc.1997.1404>
- [18] David R. Kincaid, John R. Respass, David M. Young, and Rober R. Grimes. 1982. ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Transactions on Mathematical Software* 8, 302–322. DOI : <http://dx.doi.org/10.1145/356004.356009>
- [19] Kiyoshi Kumahata, Kazuo Minami, Akira Hosoi, and Ikuo Miyoshi. 2016. HPCG Performance Improvement on the K computer. Retrieved February 14, 2018, from http://www.hpcg-benchmark.org/downloads/sc16/HPCG_on_the_K_Computer.pdf.
- [20] Kiyoshi Kumahata, Kazuo Minami, and Naoya Maruyama. 2016. High-performance conjugate gradient performance improvement on the K computer. *International Journal of High Performance Computing Applications* 30, 55–70. DOI : <http://dx.doi.org/10.1177/1094342015607950>
- [21] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS'15)*. ACM, New York, NY, 339–350. DOI : <http://dx.doi.org/10.1145/2751205.2751209>
- [22] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on Supercomputing (ICS'13)*. ACM, New York, NY, 273–282. DOI : <http://dx.doi.org/10.1145/2464996.2465013>
- [23] Yiqun Liu, Chao Yang, Fangfang Liu, Xianyi Zhang, Yutong Lu, Yunfei Du, Canqun Yang, Min Xie, and Xiangke Liao. 2015. 623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores. *International Journal of High Performance Computing Applications* 30, 1, 39–54. DOI : <http://dx.doi.org/10.1177/1094342015616267>
- [24] Yiqun Liu, Xianyi Zhang, Chao Yang, Fangfang Liu, and Yutong Lu. 2014. Accelerating HPCG on Tianhe-2: A hybrid CPU-MIC algorithm. In *Proceedings of the 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS'14)*. 542–551. DOI : <http://dx.doi.org/10.1109/PADS.2014.7097852>
- [25] Jan Mayer. 2009. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing* 86, 291–312. DOI : <http://dx.doi.org/10.1007/s00607-009-0066-3>
- [26] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Meuer Martin. 2017. Top 500 Supercomputer Lists. Retrieved February 14, 2018, from <http://www.top500.org>
- [27] Kengo Nakajima. 2014. Optimization of serial and parallel communications for parallel geometric multigrid method. In *Proceedings of the 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS'14)*. 25–32. DOI : <http://dx.doi.org/10.1109/PADS.2014.7097787>
- [28] Maxim Naumov. 2011. Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. Retrieved February 14, 2018, from <http://research.nvidia.com/sites/default/files/publications/nvr-2011-001.pdf>.
- [29] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *Supercomputing*. Springer International, 124–140. DOI : http://dx.doi.org/10.1007/978-3-319-07518-1_8
- [30] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D. Kalamkar, Xing Liu, Md. Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. 2014. Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'14)*. IEEE, Los Alamitos, CA, 945–955. DOI : <http://dx.doi.org/10.1109/SC.2014.82>

- [31] Everett Phillips and Massimiliano Fatica. 2014. A CUDA implementation of the high performance conjugate gradient benchmark. In *High Performance Computing Systems: Performance Modeling, Benchmarking, and Simulation*. Springer International, 68–84. DOI : http://dx.doi.org/10.1007/978-3-319-17248-4_4
- [32] Eugene L. Poole and James M. Ortega. 1987. Multicolor ICCG methods for vector computers. *SIAM Journal on Numerical Analysis* 24, 25. DOI : <http://dx.doi.org/10.1137/0724090>
- [33] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.
- [34] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16, 521. DOI : <http://dx.doi.org/10.1088/1742-6596/16/1/071>
- [35] Samuel Williams, Dhiraj D. Kalamkar, Amik Singh, Anand M. Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. 2012. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. IEEE, Los Alamitos, CA, 96:1–96:11. DOI : <http://dx.doi.org/10.1109/SC.2012.85>
- [36] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07)*. ACM, New York, NY, 38:1–38:12. DOI : <http://dx.doi.org/10.1145/1362622.1362674>
- [37] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 65–76. DOI : <http://dx.doi.org/10.1145/1498765.1498785>
- [38] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, NY, 107–118. DOI : <http://dx.doi.org/10.1145/2555243.2555255>

Received September 2017; revised November 2017; accepted December 2017