Journal of Parallel and Distributed Computing 138 (2020) 178–189

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Efficient AES implementation on Sunway TaihuLight supercomputer: A systematic approach☆

Liandeng Li [a,b], Jiarui Fang [a,b], Jinlei Jiang [a,*], Lin Gan [a,b], Weijie Zheng [a,b], Haohuan Fu [b,c], Guangwen Yang [a,b,c,**]

[a] Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology, Tsinghua University, Beijing 100084, China
[b] National Supercomputing Center in Wuxi, Wuxi 214072, China
[c] Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science, Tsinghua University, Beijing 100084, China

ARTICLE INFO

*Article history:*
Received 4 February 2019
Received in revised form 20 July 2019
Accepted 21 December 2019
Available online 2 January 2020

*Keywords:*
High-performance computing
Supercomputer
AES algorithm
Vectorization
Parallelism

ABSTRACT

Encryption is an important technique to improve information security for many real-world applications. The Advanced Encryption Standard (AES) is a widely-used efficient cryptographic algorithm. Although AES is fast both in software and hardware, it is time-consuming to do data encryption especially for large amount of data. Therefore, it is a lasting effort to accelerate AES operations. This paper presents SW-AES, a parallel AES implementation on the Sunway TaihuLight, one of the fastest supercomputers in the world that takes the SW26010 processor as the basic building block. According to the architectural features of SW26010, SW-AES exploits parallelism from different levels, including (1) inter-CPE (Computing Processing Element) data parallelism that distributes tasks among the 256 on-chip CPEs, (2) intra-CPE data parallelism enabled by the Single-Instruction Multiple-Data (SIMD) instructions inside each CPE, and (3) instruction-level parallelism that pipelines memory access and the computation. In addition, corresponding to the two application scenarios, SW-AES presents scalable ways to efficiently run AES on many nodes. As a result, SW-AES can gain a maximum throughput of 13.50 GB/s on a single SW26010 node, which is 216.23× higher than the latest parallel AES implementation on the Sunway TaihuLight, and about 37.3% higher than the latest AES implementation on the GTX 480 GPU. When running on 1024 computing nodes with each one processing 1 GB data, SW-AES can achieve a throughput of 13819.25 GB/s. On the contrast, only a throughput of 63.91 GB/s can be achieved by the latest related work on the Sunway TaihuLight.



## 1. Introduction

The rapid development of information and communication technology (ICT) has greatly benefited human activities. The ICT-enabled big data era where large amount of data is generated, recorded, and analyzed by modern computers provides insightful information and guidance in various application domains. However, ICT technology is also a double-edged sword — along with the continuous surge of data, severe threats and challenges also arise in terms of data protection. People nowadays want more secure approaches to protect important digital assets such as enterprise secrets and personal information. Encryption is a most widely-used solution to achieve the goal.

Supercomputers are powerful facilities widely-used in various key fields such as national defense [2], scientific and industrial computing [9,30], and machine learning [7]. For some fields, encryption is also needed to protect sensitive data. Two typical scenarios are: (1) users upload some data to the supercomputer in a way secure or not, encrypt it and store the encrypted result for future processing (termed online mode in this paper), and (2) users keep the data generated by applications secure by encrypting it (termed offline mode here).

Since data encryption is time-consuming, it is a lasting effort to boost it either by ASIC (Application Specific Integrated Circuit) implementation [11,24,25,27] or by using more powerful devices such as FPGA (Field-Programmable Gate Array) [21,26] and GPU (Graphics Processing Unit) [10,12,13,15,18,23]. Besides, supercomputers provide a good choice for performing compute-intensive encryption operations. To make full use of the available computing resources, it is at the core to design highly efficient and parallel encryption algorithms that fit well with the state-of-the-art supercomputing systems. Unfortunately, it is not an

☆ An earlier version of this work appeared as Li et al., (2017).

* Correspondence to: FIT 3-113, Tsinghua University, Beijing 100084, China.
** Correspondence to: FIT 3-112, Tsinghua University, Beijing 100084, China.
*E-mail addresses:* jjlei@tsinghua.edu.cn (J. Jiang), ygw@tsinghua.edu.cn (G. Yang).

https://doi.org/10.1016/j.jpdc.2019.12.013
0743-7315/

easy task to achieve the purpose, since many factors are usually involved, such as the features of the algorithm itself and the characteristics of the underlying system.

The Sunway TaihuLight system [9] is one of the most powerful supercomputers in the world. With the SW26010 many-core processor as the basic building block, the Sunway TaihuLight presents a peak performance of 125 PFlops as well as many other interesting features. Since its debut in June, 2016, over one hundred large-scale applications have been deployed on it, including climate modeling [1,8,29], material science [6,20], big data [7,19], and so on. In this paper, we take a step further to show how to efficiently implement the widely-adopted Advanced Encryption Standard (AES) [5] algorithm on Sunway TaihuLight to get the best throughput so as to meet the need of data protection.

Unlike the previous work [4] that tried to improve AES performance by utilizing more nodes whereas resources within a single node were underutilized, our work focuses on accelerating AES algorithm in a systematic approach by extending our previous work [17], which shows how to make full use of such features of the SW26010 processor as heterogeneous-core architecture, multi-hierarchy memory, direct memory access (DMA), two instruction pipelines, and so on to boost AES operations on a single node. The paper extends our previous work in [17] with: (1) ways to scale AES execution to many computing nodes, (2) more details about the reasons behind the design decisions in [17], and (3) additional experimental data that can help people to develop a deep understanding of the impact of various optimization techniques.

The main contributions of our work are as follows:

- We present a SIMD-friendly data layout as well as an S-Box lookup strategy that enables efficient AES operation on a single Computing Processor Element (CPE).
- We propose new parallel mechanisms to fully utilize all CPEs on a chip and the two instruction pipelines within one CPE, and to scale AES operations to many computing nodes.
- We implement SW-AES, a parallel version of AES algorithm on the Sunway TaihuLight and evaluate it thoroughly. The result shows that SW-AES can gain a maximum throughput of 13.50 GB/s on a single SW26010 node and 13819.25 GB/s on 1024 nodes, which is two orders of magnitude higher than that of the latest related work in [4].

The remainder of this paper is organized as follows. Section 2 briefly reviews background and related work. Sections 3 and 4 show how to efficiently implement AES on a single chip and on the whole system respectively. After extensive experimental results and discussions given in Section 5, the paper ends in Section 6 with some concluding remarks.

## 2. Background and related work

### 2.1. AES algorithm

Fig. 1 shows the workflow of AES algorithm. It takes a 128-bit data block as input and performs several rounds of transformations to generate output cipher text. Each 128-bit data block is processed as a 4-by-4 byte array called the *state*. The length of the cipher key can be 128, 192, or 256 bits. The number of rounds repeated in the AES, *Nr*, depends on the length of the key: $Nr = 10$ for 128-bit keys, $Nr = 12$ for 192-bit keys and $Nr = 14$ for 256-bit keys. Each round uses a different 128-bit *roundkey* derived from the original cipher key. The *roundkey* can also be viewed as a 4-by-4 byte array. Four basic operations in AES are explained below.

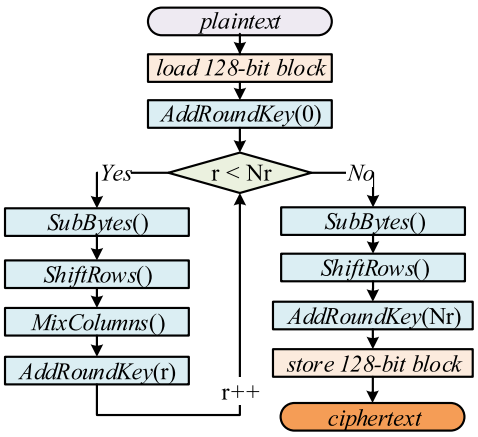



**Fig. 1.** The workflow of AES algorithm.

#### 2.1.1. SubBytes

Each byte $state[i, j]$ in the *state* matrix is replaced by the value of $S\text{-}Box(state[i, j])$, where *S-Box* is a 16-by-16 array of bytes called substitution box. The *S-box* is computed in advance before the AES encryption by the multiplicative inverse over $GF(2^8)$, which is a finite field known of good non-linearity properties. One can just view *S-box* as a fixed table.

#### 2.1.2. ShiftRows

In this step, we circularly shift row $i$ of the *state* matrix to the left by $i$ bytes, $0 \leq i \leq 3$.

#### 2.1.3. MixColumns

In this step, the four bytes of each column of the *state* array are combined using an invertible linear transformation. Then we multiply each column of the *state*, taken as a polynomial of degree below 4 with coefficient in $GF(2^8)$, by a fixed polynomial modulo $x^4 + 1$.

#### 2.1.4. AddRoundKey

In this step, the $r$th *roundkey* is added by combining each byte of the *state* with the corresponding byte of the $r$th *roundkey* using bitwise XOR. There are total $Nr + 1$ *roundkeys* needed as shown in the AES workflow. Please note, the *roundkeys* are usually calculated before the encryption process, and kept constant during encryption time. The process to generate all *roundkeys* according to the original AES key is termed key expansion.

In this paper, we only consider the encryption phase, because AES is a symmetric encryption algorithm and the decryption phase undergoes the same operations but in a reverse order. For the sake of simplicity, we only discuss the case of 128-bit keys here. Since the length of keys only determines the number of AES rounds and the *roundkeys* can be generated beforehand according to the original key, the method presented can be easily extended to the cases of 192-bit and 256-bit keys.

### 2.2. The SW26010 many-core processor

As shown in Fig. 2, each SW26010 processor consists of 4 *core groups (CGs)*. Each CG includes 65 cores: one *management processing element (MPE)*, and 64 *CPEs*, organized as an $8 \times 8$ mesh.

Both MPE and CPE are 64-bit RISC, single-threaded cores working at 1.45 GHz and supporting 256-bit vector (holding 4 single/double precision floating-point numbers or eight integers) instructions (including fused multiply-add, FMA) with 32 vector registers (extended from 32 64-bit general purpose registers),

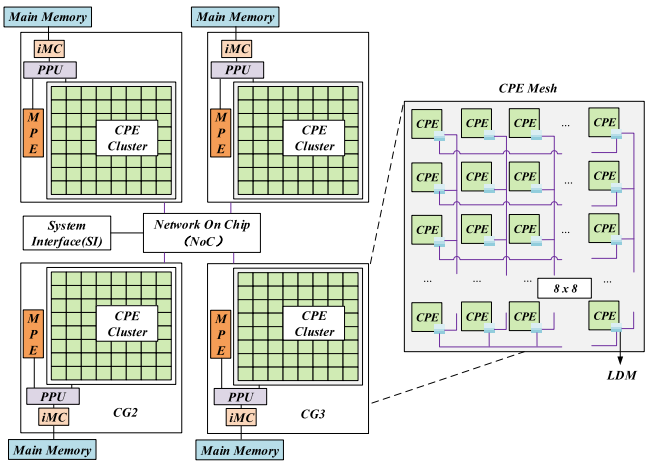



**Fig. 2.** The general architecture of the SW26010 many-core processor.

but play different roles in computation. The MPE, which supports the complete interrupt functions, memory management, superscalar, and out-of-order issue/execution and can perform 16 floating-point operations per cycle (implying a peak performance of 23.2 GFlops), is good at handling the management, task scheduling, and data communications. The CPE, which does not support interrupt functions and can only perform 8 floating-point operations per cycle (implying a peak performance of 11.6 GFlops), is designed for the purpose of maximizing the aggregated computing throughput while minimizing the complexity of the micro-architecture.

Each CG connects to its own 8 GB DDR3 memory through a 128-bit DDR3-2133 memory controller (*iMC* in the figure), which implies a theoretical DMA bandwidth of 34.128 GB/s. The Network on Chip (NoC) connects 4 CGs with *System Interface (SI)*. Users can explicitly set the size of each CG's private memory space, and the size of the memory space shared among 4 CGs. While the MPE adopts a more traditional cache hierarchy (32-KB L1 instruction cache, 32-KB L1 data cache, and a 256-KB L2 cache for both instruction and data), each CPE only provides a 16-KB L1 instruction cache, and relies on a 64 KB *Local Directive Memory (LDM)* (also known as *Scratch Pad Memory (SPM)*) as a user-controlling fast buffer. This user-controlling "cache", while increasing the programming difficulty for efficient utilization of the fast buffer, provides an option to implement a customized buffering scheme that can improve the overall performance significantly in certain cases. Inside each CPE mesh, we have a control network, a data transfer network (connecting the CPEs to the memory interface), 8 column communication buses, and 8 row communication buses. The 8 column and row communication buses enable fast register communication channels across the $8 \times 8$ CPE mesh, providing an important data sharing capability at the CPE level [28].

Each CPE has two pipelines ($P0$, and $P1$) for decoding, issuing, and executing instructions. $P0$ is for floating-point operations, and vector operations of both floating-point and integer. $P1$ is for memory-related operations. Both $P0$ and $P1$ support integer scalar operations. Therefore, identifying the right form of instruction-level parallelism can potentially resolve the dependencies in the instruction sequences, and further improve the computation throughput.

### 2.3. Nodes interconnection

Compute nodes of the Sunway TaihuLight supercomputer are connected via a customized network. The network is divided into 2 levels — a fat tree at the top and a supernode network at the bottom. While the fat tree network is used for communicating different supernodes, the bottom network is used to connect the 256 nodes within a supernode. TaihuLight uses FDR (Fourteen Data Rate) 56 Gbps network interface cards (NICs) for connection and the theoretical bandwidth between any two nodes is 14 GB/s. The real network speed is 12 GB/s with a latency at the level of micro-second when nodes are communicating via the Message Passing Interface (MPI).

### 2.4. Related work

#### 2.4.1. ASIC AES implementation

ASIC has the advantage of high efficiency and low power consumption. Therefore, it is also adopted to implement AES-specific devices. Wolkerstorfer et al. [27] presented a way to implement AES *S-Boxes* in hardware with combinational logic. The result circuit is of low transistor count and die-size, with a delay below 15 ns. Gürkaynak et al. [11] presented an ASIC implementation of the full AES algorithm, with a focus on balancing the decryption and encryption path through some optimizations. The result circuit can achieve a throughput of 0.265 GB/s for 128-bit keys. Shastry et al. [25] presented a low power ASIC implementation of AES based on the so-called rolled architecture that supports all key sizes. The implementation can achieve a throughput of 0.2 GB/s for 128-bit keys and a power consumption of as low as 22.58 mW. Schilling et al. [24] provided a 2PRG-based hardware implementation of AES-128 in their custom System-on-Chip (SoC). It has two AES-128 instances and can achieve a maximum throughput of about 0.674 GB/s at a frequency of 256 MHz. Nevertheless, ASIC is not suitable for the data center environment in that: (1) a single device cannot afford the high throughput required, and (2) using more devices means extra cost and complicate operations.

#### 2.4.2. AES on GPU

Harrison et al. [12] presented the first implementation of AES on GPU in 2007, achieving a throughput of 0.85 GB/s on Geforce 7900GT GPU using DirectX 9. Nishikawa et al. [22] and Iwai et al. [13] conducted the studies that try to figure out the influence of parallel granularity and memory usage scheme for GPU-based AES encryption. Guo et al. [10] implemented the encryption of AES-ECB algorithm and achieved a throughput of 3.96 GB/s on NVIDIA GT200 GPU. The best AES performance they can achieve is 4.375 GB/s on Geforce GTX285 GPU by adopting 16 Bytes per thread as the parallel granularity and storing *S-Box*es in shared memory. Nishikawa et al. [23] further evaluated AES based on previously reported insights and achieved 6.33 GB/s using NVIDIA Tesla C2050 GPU. In [15], the authors studied the AES encryption on Tesla c20 with innovative Read-Only cache to store *S-Box*es. However, the achieved performance was extremely poor so long as the input plaintext pattern became more random and less repetitive. Recently, a new approach proposed by Lim et al. [18] showed a scheme of restructuring the CPU-based bit-sliced implementation of the AES [14], to process four 16-byte blocks at a time and achieved 9.83 GB/s throughput on GTX 480 GPU.

#### 2.4.3. AES on FPGA

Wang et al. [26] proposed an optimization solution for AES used in storage area network applications on FPGA XC6VLX240T. A throughput of 9.78 GB/s was achieved as a result. Liu et al. [21] proposed a single pipeline design for the AES encryption algorithm based on FPGA, and managed to achieve a throughput of 8.26 GB/s. However, even if FPGAs can achieve higher power efficiency, their overall performance is limited by the total amount of hardware computing resources.

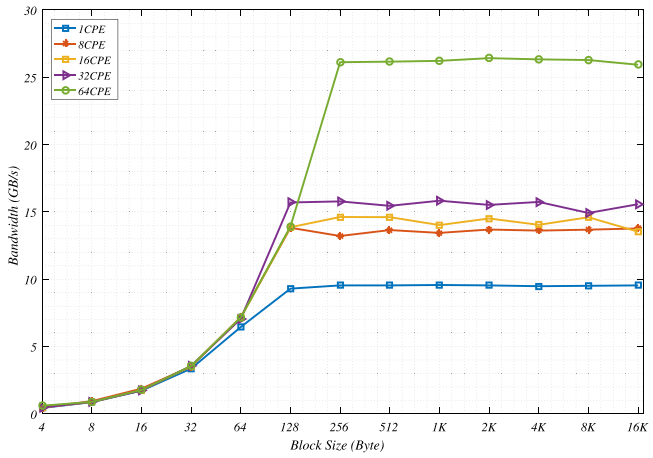



**Fig. 3.** DMA *get* bandwidth with different numbers of cores by different block sizes.

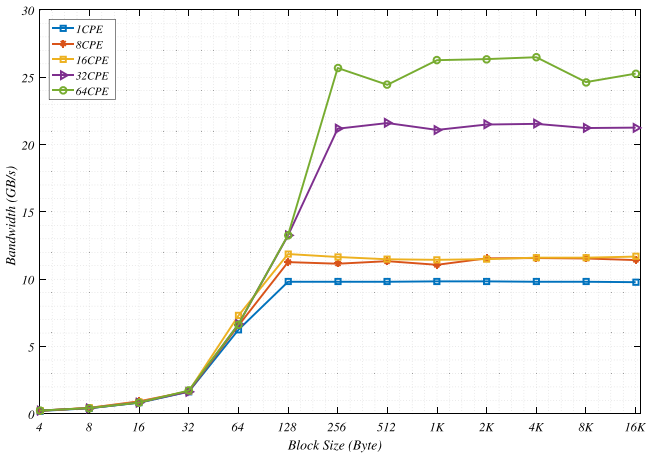



**Fig. 4.** DMA *put* bandwidth with different numbers of cores by different block sizes.

#### 2.4.4. AES on Sunway TaihuLight

Till now, few efforts are seen on designing parallel AES-ECB algorithm based on the latest SW26010 processor except the work in [4]. The authors in [4] showed a distributed implementation of AES algorithm on the Sunway TaihuLight system. Although a good scalability was achieved, that is, 998× speedup with MPI on 1024 nodes compared with the case on one node, their implementation performs poorly on a single SW26010 node, with a throughput of only 0.064 GB/s. In their work, plaintext data was processed by computing cores in parallel and no fine-grained vectorization optimization was exploited inside computing cores.

## 3. AES implementation on a single SW26010 chip

To map AES workflow to the SW26010 many-core architecture to get high throughput, the key is to exploit all possible parallelism provided by the architecture.

- Inter-CPE Data Parallelism. Whenever possible, we should try to utilize 256 CPEs to conduct AES encryption of independent plaintext blocks in parallel.
- Intra-CPE Data Parallelism. This can be achieved by using the 256-bit SIMD instructions to load more data at once and process them in parallel.
- Instruction-level Parallelism. Since there are two pipelines in a CPE and each can perform (different stages of) several instructions at once, we can adjust AES workflow and fulfill more efficient instruction scheduling to overlap instruction executions on two instruction pipelines to get more benefit.

### 3.1. Inter-CPE data parallelism

In AES, the plaintext to be encrypted can be divided into several independent fixed 16-byte plaintext blocks and processed in parallel. Therefore, it is natural to exploit data parallelism by assigning computations of plaintext blocks to 256 CPEs. A two-level parallel model is implemented for SW-AES method. Four processes are launched on 4 CGs by their MPEs to encrypt 1/4 of the whole plaintext. Inside each CG, each process launches 64 threads on 64 CPEs to encrypt a block of plaintext simultaneously.

To distribute plaintext data blocks to 256 CPEs, data movement strategy should be carefully investigated. The SW26010 architecture provides two memory access patterns for data transfer between the main memory and registers of CPEs. The CPE mesh can access data items either directly from the main memory, or from the three-level (REG–LDM–MEM) memory hierarchy. In the first case, the CPE mesh can directly access data items from memory by *gload*/*gstore* instructions. Such a direct memory access pattern does not take advantage of any possible data locality in cache. Moreover, the actual interface of *gload*/*gstore* only provides a physical bandwidth of 8 GB/s, leading to an extremely low utilization of the computing capability because it wastes most of time on data movement. In the second case, the CPE mesh accesses the data items through the MEM–LDM–REG hierarchy. We apply DMA operations to load plaintext into LDM first, and then load data into the register file for the computation. After computation, we store the obtained cipher text into LDM and transfer them back to the main memory by DMA operations. In this case, the LDM serves as a cache for each CPE. As shown in Figs. 3 and 4, the effective bandwidth for DMA load and store ranges from 4 GB/s to 28 GB/s. In general, a higher bandwidth over 24 GB/s is achieved with a block size larger than 256 bytes when all 64 cores are involved, accounting for about 70% of the theoretical DMA bandwidth. We adopt the second way for plaintext transfer between memory and registers and each CPE uses DMA to fetch data blocks as large as possible. In addition, double buffering technique is adopted to overlap DMA with computing. That is, while the data is computed in one LDM buffer, the next plaintext block is loaded into another LDM buffer by DMA.

SW26010 adopts a heterogeneous multi-core architecture combining on-chip computing array clustering and distributed shared storage, with a uniformly addressed on-chip main memory space that can be accessed by both the operation control core and the computing core. The way to run tasks on multiple CPEs in parallel is closely related to the memory operation modes. According to the structural characteristics of SW26010, two modes allowed are as follows in the Sunway TaihuLight supercomputer system:

1. Full-chip sharing mode: each running process can access the entire 32 GB memory. When 4 CGs perform data access at the same time, each CG can at most get 1/4 on-chip network bandwidth.
2. Core group private mode: there is no memory sharing between any 2 CGs and each CG can only access its own 8 GB private memory.

The parallel model with the full-chip sharing mode is shown in Fig. 5, where 4 *pthread* threads are created and allocated to 4 CGs automatically by the system. In each CG, the system uses its own slave thread acceleration *athread* library to accelerate program performance. The parallel model with the core group

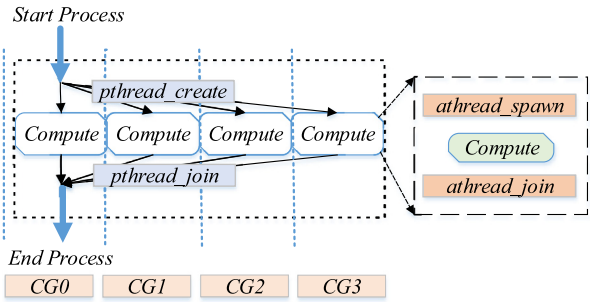



**Fig. 5.** The parallel model with full-chip sharing mode.

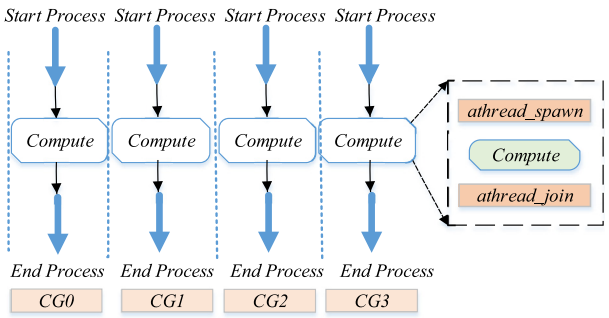



**Fig. 6.** The parallel model with CG private mode.

private mode is shown in Fig. 6, where each CG initiates a process for data processing.

### 3.2. Intra-CPE data parallelism

Based on our two-level parallel model, we intend to exploit the SIMD and instruction-level parallelism inside one thread of each CPE. However, implementing a parallel AES algorithm in one CPE is not a straightforward task due to the following reasons. First, the input data to be processed during the AES workflow is only 128 bit, making it difficult to exploit the parallelism provided by the 256-bit SIMD instructions. Second, operations critical to cryptography such as `vsrlw` and `seleqw`, which are used in already known solution [3], are not supported in CPEs of SW26010. Furthermore, the instruction set of CPE does not support SIMD operation on 1-byte elementary data type. Unfortunately, the random lookup of the fixed *S-Box* table in the AES workflow works in a bytewise fashion, that is, it can only process one byte at a time. Third, different from other popular parallel architectures such as GPU and Xeon Phi, which allows another group of threads to be issued to instruction execution pipeline when one group of threads is stalled, SW26010 requires programmers to explicitly control the instructions execution sequence. Finally, since the compilers on SW26010 provide no support of automatic vectorization to exploit SIMD capability, programming in such an environment poses a significant challenge.

To resolve the above difficulties, we propose three optimization techniques to exploit intra-CPE parallelism, namely a SIMD-friendly data layout to easily exploit data parallelism inside a CPE, a semi-SIMD style *S-Box* lookup approach to eliminate the performance degradation from non-vectorized LDM access and a Pipelined Step Execution workflow to fully utilize the two instruction pipelines. For the sake of clarity, we list the data structures used in Table 1 before introducing the above techniques in detail.

#### 3.2.1. SIMD-friendly data layout

As aforementioned in Section 2.1, AES works on the basis of a 4-by-4 byte array called *state*. Merely conducting AES operations on one *state* array is insufficient to exploit the SIMD capability of 256-bit vector registers. Even if we can load 2 *states* into a single vector register at the same time, the SIMD capability will not be unlocked naturally because the byte elements, which appear in the register in a fixed order, are not contiguous either in the view of *ShiftRows* operation or in the view of *MixColumns* operation — the former works in row-major order and the latter works in column-major order. To deal with the problem and make full use of the SIMD capability to boost performance, a SIMD-friendly data layout is proposed.

A SIMD-friendly data layout is possible because: (1) there is no dependency between different *states* so that they can be manipulated at the same time, and (2) there are enough (32) vector registers in one CPE. The idea of our SIMD-friendly data layout is straightforward. Since a 256-bit vector register can hold 32 bytes and each row of a *states* array has 4 bytes, we treat 8 *states* as a batch and deploy 4 registers to make the corresponding row in one batch to appear in one register as shown in Fig. 7, where the input byte stream is also shown. Such a data layout is SIMD-friendly because both *ShiftRows* and *MixColumns* can now perform directly on the bytes in one register. Please note that, although our idea of SIMD-friendly data layout is straightforward, deriving such a layout efficiently in the real world is not as easy as shown in Fig. 7, for the input data block (i.e., input stream in the figure) appears in column-major order from the perspective of the *state* array and at most two data blocks can be processed at the same time even with the help of 256-bit vector register. Indeed, a carefully designed transformation scheme is needed.

The transformation scheme we present is illustrated in Fig. 8. It consists of 4 steps and works entirely with SIMD instructions. At the very beginning, we use 4 *vldd* instructions to load 8 *states* from LDM to four 256-bit vector registers, each maintaining two *states*. Treating every 4 bytes in the input stream (corresponding to one column of a *state* array as shown in Fig. 7) as an integer, we then get 32 integers, namely *A0*, *A1*, …, *H3* as shown in Fig. 8. The first three steps work on integers. Each step executes a shuffle instruction with a different mask. Different colors in the figure indicate different columns of a *state* array. After the first three steps, the integers belonging to the same column of different *state* arrays are in a single vector register. Step 4 works on bytes, where each integer is viewed as 4 bytes with different colors indicating different positions (i.e., row numbers) in a *state* array. Vectorized *SHIFT* and *AND* instructions are used in this step to move the bytes to form the final SIMD-friendly data layout. Please note that all instructions here just change the positions of elements (either integers or bytes) rather than their contents.

In the end, we would like to make the following points clear. First, transformation is also needed to flush the produced cipher text back to the main memory. It is not discussed here because it can be easily implemented by reversing the steps and instructions mentioned above. Second, with the help of SIMD instructions, the transformation is very efficient. All transformations plus data loading and storing only take up 2%–3% of the whole AES workflow. Third, since the SIMD-friendly data layout changes the positions of bytes in a *state* array, the corresponding *roundkey* should be adjusted accordingly to facilitate vectorized operations and to ensure correct results. This is easy to do, for the *roundkey* is also a 4-by-4 byte array and keeps unchanged in one round. The only action needed is just to duplicate every byte of the *roundkey* 8 times and place them in the corresponding positions of the vector register. We call the key obtained in this way *ExtRoundKey*.

**Table 1**
Data structures used during the SW-AES workflow.

| Name | Size | Location | Description |
|---|---|---|---|
| *state* | 128 bits | LDM | Mentioned in Section 2.1.1 |
| *state batch* | 128 × 8 bits | 4 registers | 8 *state* arrays perform transformation together |
| *plaintext block* | 128 × bytes | LDM | A block of plaintext fetched by DMA |
| *S-Box* | 256 bytes | LDM | Mentioned in Section 2.1.1 |
| *ExtS-Box* | 1024 bytes | LDM | Extend each Byte in *S-Box* into integer |
| *roundkey* | 128 × 11 bits | LDM | Mentioned in Section 2.1.4, there are 11 *roundkeys* in total for 128-bit cipher keys |
| *ExtRoundKey* | 128 × 11 bytes | LDM | Duplicate each byte in a *roundkey* 8 times, refer to Section 3.2.1 for more details |

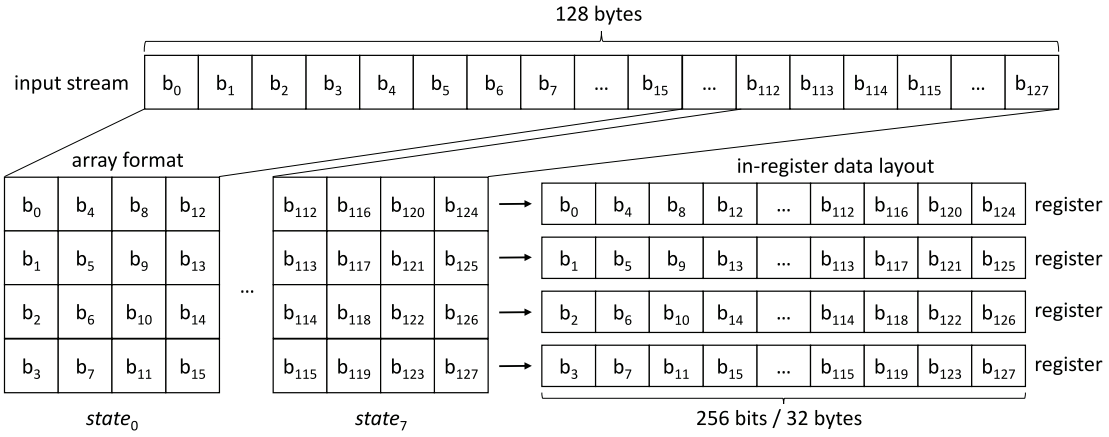



**Fig. 7.** An illustration of our SIMD-friendly data layout and an intuitive way to get it. Here one column of a *state* array corresponds to 4 continuous bytes in the input stream.

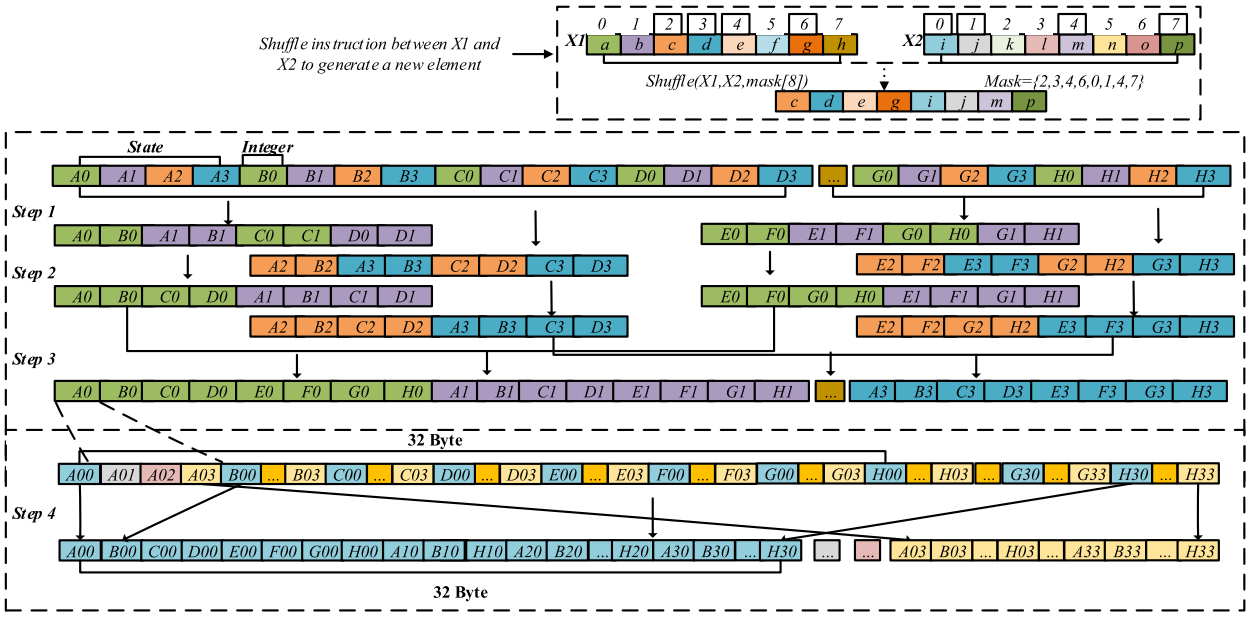



**Fig. 8.** A scheme to transform eight 128-bit *state* arrays into a SIMD-friendly data layout, where different characters indicate different *state* arrays, with *Ai* indicating the 4 bytes (forming an integer) in column *i* of the corresponding *state* array and *Aij* indicating the byte at [j, i] (namely $Aij = state[j, i]$). In the top of the figure, we illustrate how the instruction *Shuffle(X1, X2, mask)* works. *X1* and *X2* are two 256-bit registers each of which holds 8 integers. The *mask* (i.e., *{2,3,4,6,0,1,4,7}*) is used to indicate which 4 integers in a register will be taken to the new register. In this case, the first 4 integers (i.e., *c*, *d*, *e*, *g*) come from *X1* and the rest 4 integers (i.e., *i*, *j*, *m*, *p*) are from *X2*.

### 3.2.2. Semi-SIMD style S-Box lookup

The data layout proposed previously makes it possible to fully exploit the SIMD parallelism of such operations as *ShiftRows*, *MixColumns* and *AddRoundKey*. However, the operation *SubBytes* cannot make full use of SIMD parallelism because *S-Box* lookup according to $state[i, j]$ incurs a lot of random access. A naïve and intuitive solution is to convert *S-Box* into a vector, extract the desired bytes with vector operations, and transform the extracted bytes back into scalars. However, scalar operations to load elements of *S-Box* from LDM and insert them into the original vector consume a lot of CPU cycles, which is non-negligible.

To eliminate the overhead of vector and scalar transformation, a three-stage semi-SIMD style *S-Box* lookup strategy is proposed in accordance with the SIMD-friendly data layout, as illustrated in Fig. 9. The first stage is called extend stage, for we in this stage extend each byte of the input vector into an integer. With 3 `vsrlw`

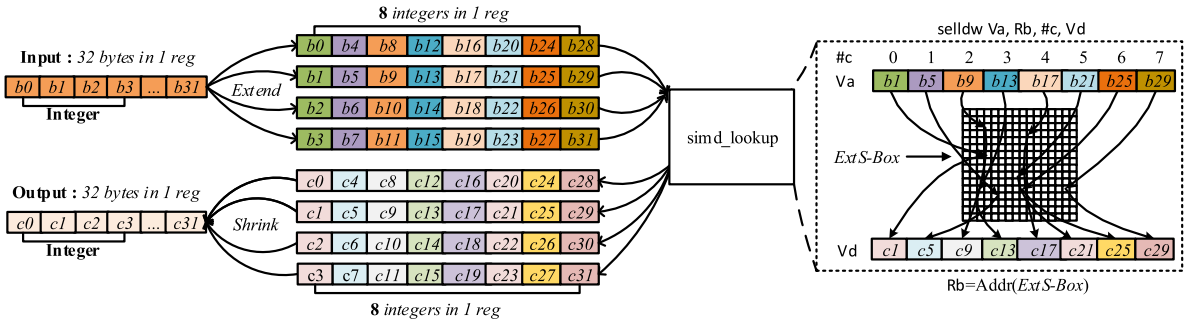



**Fig. 9.** Vectorized random table lookup operation. The input vector supplies the text (in SIMD-friendly layout) to be encrypted.

(vectorized SHIFT) instructions and 4 `vandw` (vectorized AND) instructions, the original input vector is extended into 4 *int*v8 vectors. Please note that *S-Box* should be extended in the same way in advance in order to facilitate the following operations. We use *ExtS-Box* to denote the table after extension. The second stage is table lookup stage. We finish the task with `selldw`, a unique instruction provided by SW26010 for selecting and loading data in word. The full format of the instruction is `selldw Va, Rb, #c, Vd`, where `Rb` is a 32-bit operand indicating the starting address of the memory (namely *ExtS-Box* here) to be looked up in LDM, `Va` is a 256-bit extended *int*v8 vector with 8 integers each of which specifies an offset to `Rb`, `#c` is an operand with the valid value in [0, 7] to indicate which integer in `Va` is selected, and `Vd` is the register for the return values. Eight `selldw` instructions are executed with `#c` decreasing from 7 to 0 to finish the processing of `Va` and store the final result in `Vd`. With the help of `selldw` instruction, all data are manipulated in vectors, thus avoiding the overhead of frequent vector–scalar transformation. The last stage is shrink stage, where we shrink integers back into bytes and combine the elements in 4 integer vectors into one byte vector.

### 3.3. Instruction-level parallelism

Each CPE of the SW26010 processor has two instruction pipelines, namely *P0* and *P1*. Floating-point operations and vector operations can only be handled on *P0*. Control transfer operations, *load*/*store* and register communication operations for both scalar and vector can only be handled on *P1*. In each cycle, if the next two instructions in the front of the instruction queue can be issued into two instruction pipelines separately, we can exploit the instruction-level parallelism (ILP) inside one CPE.

We propose a Pipelined Step Execution technique to balance the utilization of two instruction pipelines to increase ILP. We notice that the execution steps of the AES workflow can be categorized into two different types as *P*0-pipeline bounded and *P*1-pipeline bounded kernels, according to their computing patterns. Step *SubBytes*, as mentioned in the previous subsection, mainly involves LDM access operations on *P*1-pipeline. Step *ShiftRows*, which includes vectorized shift operations, and *MixColumns*, which includes a set of logical instructions, can be executed on *P*0-pipeline. Step *AddRoundKey* is divided into two parts. The part which requires to access a specific part of 128 bytes *ExtRoundKey* each round can be implemented with vectorized load instruction and executed on *P*1 pipeline. The part that performs vectorized *XOR* operations is *P*0 pipeline bounded. Without affecting the dependencies of the AES workflow, we rearrange the workflow of AES into pipeline-fashion as depicted in Fig. 10. By overlapping execution of *P*0-bounded and *P*1-bounded steps on independent *state* data, we can fully utilize the two pipelines.

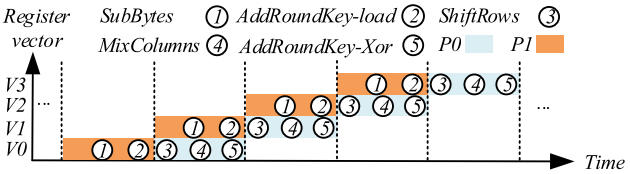



**Fig. 10.** Pipelined workflow to increase Instruction-Level-Parallelism (ILP).

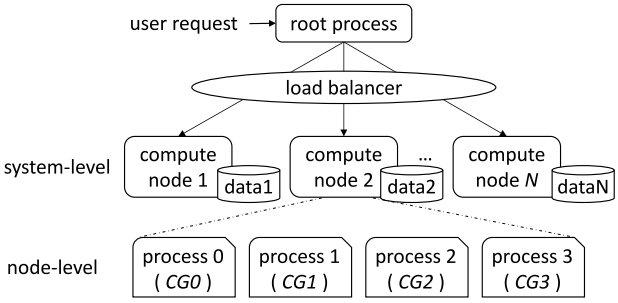



**Fig. 11.** Parallel scheme for offline encryption.

## 4. Running AES on the whole system

Since the processing capability of a single SW26010 processor is high enough (please refer to Section 5.1.1 for details), running AES on the whole system is only needed when the input data volume is very large. Corresponding to the scenarios mentioned in Section 1, two parallel schemes are developed as follows.

The parallel scheme for the offline scenario is illustrated in Fig. 11, where data is distributed in multiple nodes with the volume at each node known in advance. First, a root process receives a request that specifies the cipher key, the number of nodes to be used as well as the location of data to be encrypted. After that, *roundkeys* are generated and broadcast to all the nodes involved along with *ExtS-Box*. Next, parallel execution at the node level starts, which has been discussed previously in Section 3 and will not be repeated here. Here the core group private mode is suggested because, as shown in Fig. 13, it can gain better performance at most time. Since the volume of data at each node may vary greatly sometime in the real world, a load balancer is introduced to balance the load of different nodes. As different parts of data are independent during encryption and the data volume at each node is known in advance, the balancing policy we use – to make every node process the same amount of data while with the data transfer time taken into consideration – is simple and will not be further explained in detail here. Indeed, when each process deals with a large amount of data, the data transfer time can even be hidden (i.e., overlapped with computation) by processing the incoming data first.

**Table 2**
System configurations.

| Item | Value |
|---|---|
| Processor | SW26010 |
| Operation system | Sunway Raise OS 2.0.5 (based on Linux) |
| Instruction set | Sunway-64 Instruction Set |
| C compiler | sw5cc.new Version 5.421-sw-496 |
| MPI compiler | mpiCC Version 5.421-sw-496 |

The parallel scheme for the online scenario is illustrated in Fig. 12, where an input data stream is supplied along with other request parameters such as the cipher key. The detailed workflow is as follows. First, a root process generates *roundkeys* and broadcasts them to all the nodes involved along with *ExtS-Box* as does in the offline scenario. Then, the root process continuously splits the incoming data stream into fixed-sized data blocks and sends them to various nodes for processing. To balance the load, the root process tries to send each node the same amount of data. After that, data blocks are processed at each node independently. Here the full-chip sharing memory mode is used in order to hold in memory as many data blocks as possible. In addition, a double-buffer mechanism is used to overlap computation and communication so that no on-chip mesh network resource is wasted during cipher calculation. After a data processing task is complete, the root process collects the output. The procedure stops until no more data is coming and all data processing tasks are complete. In the end, we should point out that the data block size has great impact on the system performance. As can be seen from Fig. 13, the amount of data allocated to each core group each time must be no less than 64 MB in order to maintain system performance because it is only under this condition that the *pthread* startup overhead can be ignored.

## 5. Performance evaluation and analysis

We implement SW-AES using a two-level multi-threading programming model, with MPI to launch 1 (full-chip sharing mode) or 4 (core group private mode) processes on 1 or 4 MPEs of a single chip and Sunway *OpenACC*|*Athread* to launch 64 light-weight threads to the 64 CPEs within one CG. It is Sunway *OpenACC* that conducts data transfer between main memory and LDM and uses the *Athread* threading library to manage threads on CPE and to distribute the kernel workload to them. Except that the *SubBytes* Step is implemented with assembly codes, the other steps are implemented with C programming language. We perform the experiments on a single SW26010 processor as well as the whole system of the Sunway TaihuLight, with the configurations listed in Table 2.

### 5.1. SW-AES performance on a single node

#### 5.1.1. Overall performance

Fig. 13 shows SW-AES throughput with various input data sizes and memory operation modes. It is easy to see that, for the core group private mode, SW-AES throughput increases almost linearly with the input data size increasing from 1 kB to 8 MB. When the input data size is larger than 16 MB, it can achieve a throughput over 12.5 GB/s. The throughput increase drops rapidly after the input size is greater than 4 MB and stops at 256 MB because the computing resources (i.e., CPEs) are insufficient, for the DMA bandwidth is still underutilized now. In all, a maximal throughput of 13.50 GB/s can be achieved. The throughput of the full-chip sharing mode is lower and increases more slowly at the beginning, but it approaches that of the core group private mode when the input data size is larger than 64 MB.

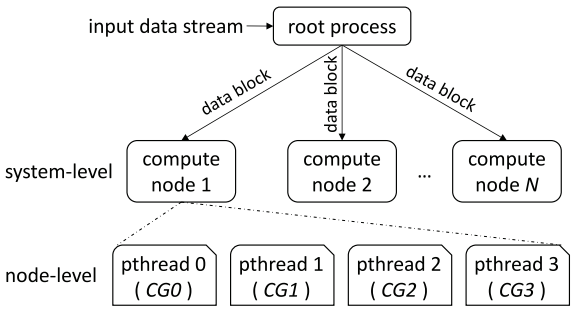



**Fig. 12.** Parallel scheme for online encryption.

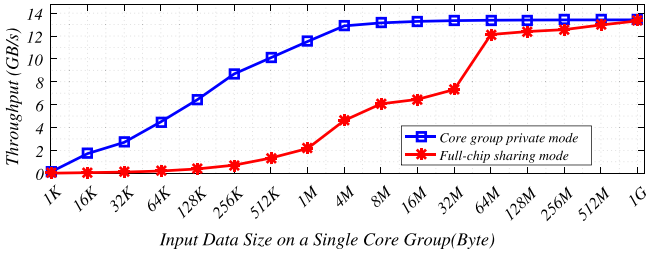



**Fig. 13.** The SW-AES overall performance with various plaintext block sizes in different memory operation modes.

To make the performance gains clearer, Fig. 14 illustrates the proportion of calculation time, DMA data transfer time, thread start-up time and the calculation time overlapped by double buffer in the core group private mode. The case of full-chip sharing mode is not given here because it works in the same way except that it introduces additional *pthread* startup time, which is more than 60 times of the *athread* startup time. We can see that, when the input data size is less than 1 MB, the thread start-up time on CPE is non-negligible. An interesting fact is that double-buffer technique, which is also used in [4], has limited benefit to the overall performance. The reason is due to the special DMA mechanism of the SW26010 processor: CPE mesh conducts asynchronous DMA operations across 4 groups of CPEs within one CG, whereas sequential DMA operations are performed within each group. Thus, computation on one CPE group has already been overlapped with DMA on another CPE group even if a single buffer is used for DMA operation. As a result, only limited improvement is achieved with the double buffer technique on SW26010.

According to Fig. 14, we can conclude that when the input data size is less than 4 MB, the startup time (*athread* startup time) of the CPE cannot be ignored. Similarly, in the full-chip sharing mode, the *pthread* startup time (overhead) is also non-negligible. The performance depends on the input data size greatly. When the input data size of each CG is less than 64 MB, the performance of full-chip sharing mode is limited due to the *pthread* startup time, which has a large impact on the performance of the system. When the input data size of each CG is greater than 64 MB, the percentage of computing time gradually increases, and the impact of *pthread* startup time on the system decreases accordingly. The figure also shows that only when the processed data block is large, the full-chip sharing mode has the advantage; otherwise, its startup time becomes the bottleneck of system performance acceleration.

#### 5.1.2. The effect of various optimizations

We also do some experiments to show the benefit of various optimization techniques proposed in this paper to the overall

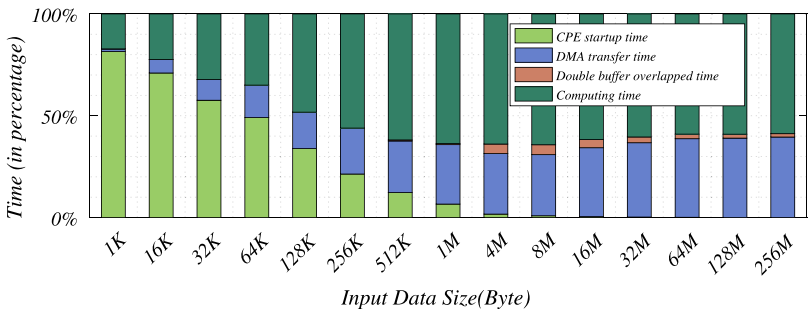



**Fig. 14.** Percentage of calculation time, double buffer overlap time, data transfer time and thread startup time of SW-AES workflow under various plaintext block sizes in the core group private mode.

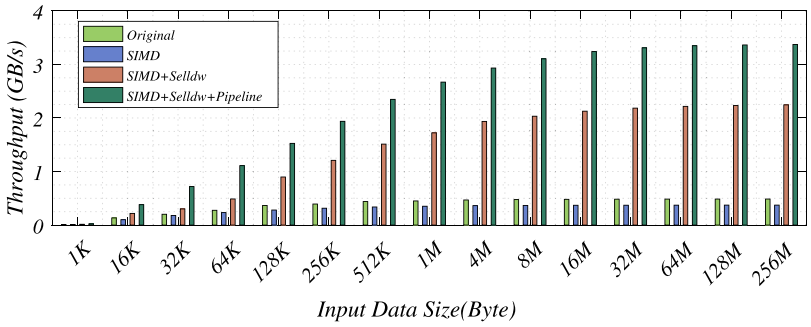



**Fig. 15.** The effect of different optimization techniques on the SW-AES overall performance under various plaintext block sizes. The throughput here is obtained on one core group.

performance. Fig. 15 shows the results on one CG with respect to different input sizes (from 1 kB to 256 MB). Different colors in the figure refer to the original non-vectorized implementation (denoted by Original parallel), implementation with the vectorization based on SIMD-friendly data layout method (denoted by SIMD), implementation with SIMD and a semi-SIMD fashion lookup table method (denoted by SIMD+selldw), and implementation incorporating all the optimization techniques (denoted by SIMD+selldw+Pipeline), respectively.

It is easy to see from the figure that directly using SIMD cannot improve performance. On the contrary, the performance is somewhat decreased. The reason is that the non-vectorized *S-Box* loop-up operation produces a huge amount of overhead when extracting elements from and inserting scalar elements into vector registers. However, it provides a good basis for the following operations. In detail, the proposed semi-SIMD fashion lookup table scheme is able to produce obvious benefits for all input sizes, with a speedup ranging from 1.4× to 4.6×. The maximum throughput for all input sizes is achieved with further using the pipelined workflow approaches, obtaining speedups of 2.3× to 6.9× over the corresponding original implementation.

### 5.2. SW-AES scalability

We evaluate the scalability of SW-AES on multiple nodes, with the experimental results shown in Fig. 16 (for weak scaling) and Fig. 17 (for strong scaling) respectively. Since the throughput of a single SW26010 processor can reach as high as 13.50 GB/s and the real network speed is only 12 GB/s when compute nodes are communicating via MPI, the network is obviously a bottleneck for data distribution in the online scenario. For operations with larger keys, more nodes would do help to the performance of on-line mode, but the benefit is limited because only 20% (for 192-bit keys) or 40% (for 256-bit keys) more computation is needed and network is still a bottleneck. Therefore, our experiment here omits the online scenario and focuses only on the offline scenario with different memory operation modes. Also, the result of related work [4] is listed as a comparison.

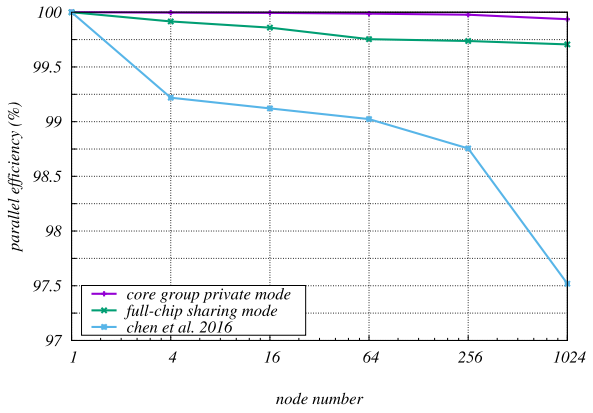



**Fig. 16.** Weak scaling of SW-AES on multiple nodes, where each node processes 1 GB data.

For weak scaling, all three cases can achieve a parallel efficiency over 97.5%. It means near-linear scalability, that is, using $N$ nodes means nearly $N$ times faster. This is because data parallelism scheme is adopted among multiple compute nodes and the information exchanged between nodes is relatively small so that the cost is negligible when each node processes a large amount of data. For strong scaling, near-linear scalability can also be achieved for all three cases when the number of nodes is less than 256. But, except for the core group private mode, the speedup rates drop greatly as the number of nodes increases. This is because, with more nodes, the data to be processed at each node gets less, making the once negligible overhead (i.e., CPE start time as shown in Fig. 14) obvious.

Please note that the result reported here is an ideal case where each node processes the same amount of data. For applications in

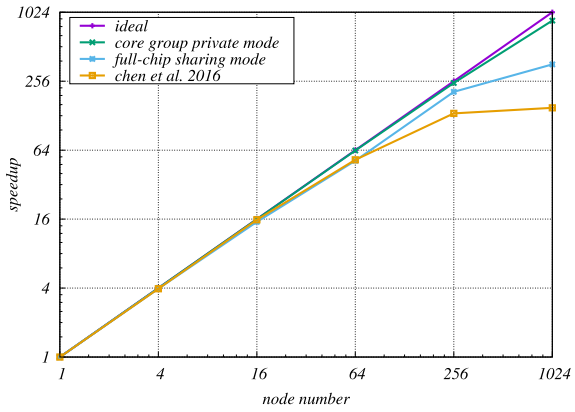



**Fig. 17.** Strong scaling of SW-AES on multiple nodes, where the data to be processed is fixed to 4 GB.

**Table 3**
A comparison of SW-AES with other work on a single device.

| Hardware device | Work | Throughput | Peak performance |
|---|---|---|---|
| FPGA XC7VX690T | [21] | 8.26 GB/s | Inapplicable |
| FPGA XC6VLX240T | [26] | 9.78 GB/s | Inapplicable |
| GeForce GTX 285 | [13] | 4.4 GB/s | 0.709 TFlops |
| Tesla C2050 | [23] | 6.33 GB/s | 1.03 TFlops |
| NVIDIA GT200 | [10] | 3.96 GB/s | 0.293 TFlops |
| GeForce GTX 480 | [18] | 9.83 GB/s | 1.345 TFlops |
| SW26010 | This paper | 13.50 GB/s | 3.06 TFlops |

the real world, the data generated at different nodes might not be that even. Therefore, it is hard to achieve near-linear scalability. Depending on the data distribution, the speedup that can be obtained might be quite different. Since applications are diverse and it is difficult to define a typical one to cover all the conditions, we do not show the scalability of real world applications here. But one worst case we would like to point out here is that there is only one node containing the data generated. Since network is a bottleneck as aforementioned for the online scenario, the maximal speedup that can be achieved in this case would be no more than 2×. That is, besides the local node, the data can only be sent to another node for processing without resources being wasted.

### 5.3. Comparison with related work

#### 5.3.1. Comparison with the work on the Sunway TaihuLight

The work reported in [4] is the only related work done on the Sunway TaihuLight. The comparison between it and SW-AES (core group private mode) is shown in Fig. 18. In accordance with the settings in [4], only the block sizes between 1 MB and 1 GB are used at each node, which means a total input data size between 1 GB and 1024 GB as shown in Fig. 18. It is easy to see from the figure that SW-AES performs much better than the work in [4], with a throughput improvement about 216.23× when the block size is 1 GB. Indeed, more throughput improvement can be gained with smaller block size. The reason is that the work in [4] did not take into consideration the fine-grained vectorization and pipelined execution as we do.

From the experimental results shown in Fig. 18, we can draw the conclusion that SW-AES has an aggregated throughput of 13819.25 GB/s (i.e., about 13.495 GB/s each node) on 1024 computing nodes with each processing 1 GB input data, while the work by Chen et al. [4] has an aggregated throughput of only 63.91 GB/s (i.e., about 0.062 GB/s each node). This is consistent with the result shown in Fig. 16.

#### 5.3.2. Comparison with other related work

As aforementioned in Section 2.4, there are also other ways to boost AES algorithm besides the work on the SW26010 processor. Table 3 lists the comparison results between SW-AES and

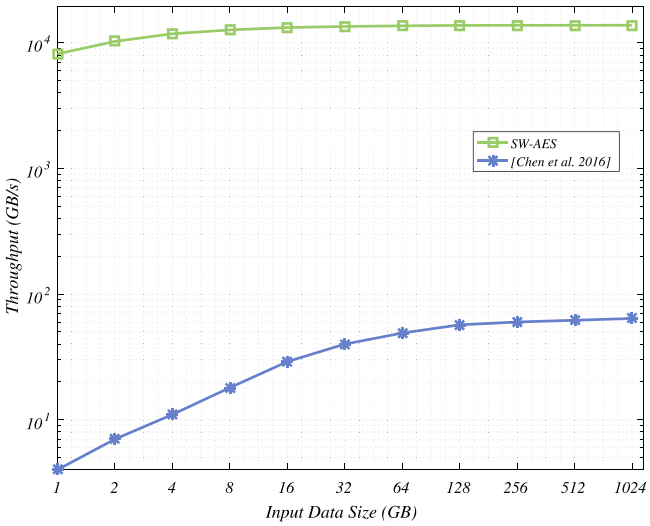



**Fig. 18.** Throughput comparison between SW-AES (core group private mode) and the work by *Chen et al.* [4] under various plaintext block sizes. The input data is evenly distributed on 1024 nodes.

other works on FPGA [21,26] and GPU [10,13,18,23], where peak performance means the maximum performance in theory. For FPGA, the peak performance is marked with inapplicable because it depends on how circuits of the FPGA card are used. Please note that the data presented here are taken from the corresponding papers, and only the maximum throughputs are listed. For SW-AES, a maximum throughput of 13.50 GB/s can be achieved on a single processor due to the novel techniques proposed. This value exceeds those on the mainstream HPC (high-performance computing) accelerators such as FPGA and GPU. Since different hardware devices are used and only operations of integer are involved in AES, the peak performance and the comparison result can only be used for reference.[1] Anyway, it indicates SW-AES is a promising and efficient solution for data encryption/decryption on the Sunway TaihuLight supercomputer.

## 6. Conclusion

In this paper, we reported our effort on accelerating AES encryption/decryption on the Sunway TaihuLight, one of the fastest supercomputers in the world that is homegrown in China. We presented SW-AES, a parallel version of AES algorithm on the Sunway TaihuLight. With a set of optimization techniques proposed, namely, task-level parallelism among many CPEs, data-level parallelism via SIMD, and instruction-level parallelism via pipelined execution, SW-AES managed to achieve a throughput of 13.50 GB/s on a single SW26010 processor. When running on 1024 computing nodes with each processing 1 GB data block, SW-AES can achieve a throughput as high as 13819.25 GB/s, compared with 63.91 GB/s of the latest work done on the Sunway TaihuLight [4]. SW-AES throughput on a single SW26010 processor also excels over that of the work on FPGA and GPU. As FPGA and GPU are nowadays widely used for data processing and protection, the great throughput gain achieved by SW-AES implies the SW26010 processor is a promising candidate for the AES algorithm.

In the end, we would like to point out that, though the techniques presented here are specially designed for AES, the idea of boosting performance systematically with full architectural features taken into consideration is applicable to other applications

[1] Since both GPU and SW26010 are optimized for floating-point operations, roughly speaking, the performance of integer operations is about 1/4∼1/3 of the floating-point performance.

including traditional HPC and/or MPI applications like scientific computing and the emerging ones like deep learning. Of course, different steps should be taken when applying the idea to other applications due to the difference between applications. One example can be found in [16]. It showed how to accelerate deep learning applications in a systematic way. Besides such mechanisms as utilizing inter-CPE parallelism as much as possible, large data block transfer between main memory and LDM via DMA, and SIMD-friendly data layout mentioned in this paper, it developed other optimizations such as improved all-reduce communication and parallel I/O operation. In the end, since SIMD instructions are widely supported by modern processors, the idea of making SIMD-friendly data layout can also benefit applications on the other supercomputers.

## Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.12.013.

## Acknowledgments

This work is co-supported by National Key R&D Program of China (2018YFB0204102), and National Natural Science Foundation of China (61572280, 61672312, 61702491).

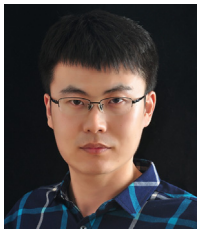
## References

[1] Y. Ao, C. Yang, X. Wang, W. Xue, H. Fu, F. Liu, L. Gan, P. Xu, W. Ma, 26 pflops stencil computations for atmospheric modeling on sunway taihulight, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 535–544.
[2] B.J. Archer, Seventy Years of Computing in the Nuclear Weapons Program, Tech. Rep., Los Alamos National Laboratory (LANL), 2015.
[3] J.W. Bos, D.A. Osvik, D. Stefan, Fast implementations of aes on various platforms, 2009, IACR Cryptol. ePrint Archive, 2009, 501.
[4] Y. Chen, K. Li, X. Fei, Z. Quan, K. Li, Implementation and optimization of AES algorithm on the sunway taihulight, in: 2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), IEEE, 2016, pp. 256–261.
[5] J. Daemen, V. Rijmen, Specification for the advanced encryption standard (aes), Fed. Inf. Process. Stand. Publ. 197 (2001).
[6] W. Dong, L. Kang, Z. Quan, K. Li, K. Li, Z. Hao, X.-H. Xie, Implementing molecular dynamics simulation on sunway taihulight system, in: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE, 2016, pp. 443–450.
[7] J. Fang, H. Fu, W. Zhao, B. Chen, W. Zheng, G. Yang, Swdnn: A library for accelerating deep learning applications on sunway taihulight, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 615–624.
[8] H. Fu, J. Liao, W. Xue, L. Wang, D. Chen, L. Gu, J. Xu, N. Ding, X. Wang, C. He, et al., Refactoring and optimizing the community atmosphere model (cam) on the sunway taihulight supercomputer, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, 2016, p. 83.
[9] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, et al., The sunway taihulight supercomputer: system and applications, Sci. China Inf. Sci. 59 (7) (2016) 072001.
[10] G.-l. Guo, Q. Qian, R. Zhang, Different implementations of aes cryptographic algorithm, in: 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), IEEE, 2015, pp. 1848–1853.
[11] F. Guürkaynak, A. Burg, N. Felber, W. Fichtner, D. Gasser, F. Hug, H. Kaeslin, A 2 gb/s balanced aes crypto-chip implementation, in: Proceedings of the 14th ACM Great Lakes Symposium on VLSI, ACM, 2004, pp. 39–44.
[12] O. Harrison, J. Waldron, Aes encryption implementation and analysis on commodity graphics processing units, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2007, pp. 209–226.
[13] K. Iwai, T. Kurokawa, N. Nisikawa, Aes encryption implementation on cuda gpu and its analysis, in: 2010 First International Conference on Networking and Computing (ICNC), IEEE, 2010, pp. 209–214.
[14] E. Käsper, P. Schwabe, Faster and timing-attack resistant aes-gcm, in: Cryptographic Hardware and Embedded Systems-CHES 2009, Springer, 2009, pp. 1–17.
[15] A. Khan, M. Al-Mouhamed, A. Almousa, A. Fatayar, A. Ibrahim, A. Siddiqui, Aes-128 ecb encryption on gpus and effects of input plaintext patterns on performance, in: 2014 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), IEEE, 2014, pp. 1–6.
[16] L. Li, J. Fang, H. Fu, J. Jiang, W. Zhao, C. He, X. You, G. Yang, Swcaffe: A parallel framework for accelerating deep learning applications on sunway taihulight, in: IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018, IEEE, 2018, pp. 413–422.
[17] L. Li, J. Fang, J. Jiang, L. Gan, W. Zheng, H. Fu, G. Yang, SW-AES: accelerating AES algorithm on the sunway taihulight, in: 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou, China, December 12-15, 2017, IEEE, 2017, pp. 1204–1211.
[18] R.K. Lim, L.R. Petzold, Ç.K. Koç, Bitsliced high-performance aes-ecb on gpus, in: The New Codebreakers, Springer, 2016, pp. 125–133.
[19] H. Lin, X. Tang, B. Yu, Y. Zhuo, W. Chen, J. Zhai, W. Yin, W. Zheng, Scalable graph traversal on sunway taihulight with ten million cores, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 635–645.
[20] J. Liu, H. Qin, Y. Wang, G. Yang, J. Zheng, Y. Yao, Y. Zheng, Z. Liu, X. Liu, Largest particle simulations downgrade the runaway electron risk for ITER, 2016, arXiv preprint arXiv:1611.02362.
[21] Q. Liu, Z. Xu, Y. Yuan, A 66.1 gbps single-pipeline aes on fpga, in: 2013 International Conference on Field-Programmable Technology (FPT), IEEE, 2013, pp. 378–381.
[22] N. Nishikawa, K. Iwai, T. Kurokawa, Granularity optimization method for aes encryption implementation on cuda, IEICE Tech. Rep. VLSI Des. Technol. 109 (393) (2010) 107–112.
[23] N. Nishikawa, K. Iwai, T. Kurokawa, High-performance symmetric block ciphers on cuda, in: 2011 Second International Conference on Networking and Computing (ICNC), IEEE, 2011, pp. 221–227.
[24] R. Schilling, T. Unterluggauer, S. Mangard, F.K. Gürkaynak, M. Muehlberghuber, L. Benini, High speed asic implementations of leakage-resilient cryptography, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018, IEEE, 2018, pp. 1259–1264.
[25] P. Shastry, A. Kulkarni, M.S. Sutaone, Asic implementation of aes, in: 2012 Annual IEEE India Conference (INDICON), IEEE, 2012, pp. 1255–1259.
[26] Y. Wang, Y. Ha, High throughput and resource efficient aes encryption/decryption for sans, in: 2016 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2016, pp. 1166–1169.
[27] J. Wolkerstorfer, E. Oswald, M. Lamberger, An asic implementation of the aes sboxes, in: Cryptographers' Track at the RSA Conference, Springer, 2002, pp. 67–78.
[28] Z. Xu, J. Lin, S. Matsuoka, Benchmarking sw26010 many-core processor, in: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2017, pp. 743–752.
[29] C. Yang, W. Xue, H. Fu, H. You, X. Wang, Y. Ao, F. Liu, L. Gan, P. Xu, L. Wang, et al., 10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, 2016, p. 6.
[30] J. Zhang, C. Zhou, Y. Wang, L. Ju, Q. Du, X. Chi, D. Xu, D. Chen, Y. Liu, Z. Liu, Extreme-scale phase field simulations of coarsening dynamics on the sunway taihulight supercomputer, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, 2016, p. 4.

**Liandeng Li** is a PhD candidate with Department of Computer Science and Technology, Tsinghua University. His research interests include high-performance computing and big data. His research work has appeared in SC, IEEE Cluster Conference, and IEEE ISPA.

**Jiarui Fang** is a PhD candidate with Department of Computer Science and Technology, Tsinghua University. His research interests include high-performance computing and deep learning. His research work has appeared in SC, ACM TACO, IPDPS, and ICPADS.

**Jinlei Jiang** is an associate professor with Department of Computer Science and Technology, Tsinghua University. His research interests include distributed computing and systems, cloud computing, and big data. He is currently on the editorial boards of KSII Transactions on Internet and Information Systems and EAI Endorsed Transactions on Industrial Networks and Intelligent Systems. He has published more than 50 research publications in refereed conferences and journals such as INFOCOM, IEEE Cluster, TPDS, and IEEE TBD.



**Lin Gan** earned his PhD in computer science in 2016 from Tsinghua University. His research interests include high-performance solutions to scientific applications based on state-of-the-art platforms such as CPU, FPGAs, and GPUs. He has more than 30 high-quality publications, and is the recipient of the 2016 Gordon Bell Prize, the finalist of the 2017 Gordon Bell Prize, the Most Significant Paper Award in 25 Years awarded by FPL 2015, the 2017 Tsinghua-Inspur Computational Earth Science Young Researcher Award, and the 2018 IEEE-CS Technical Consortium on High Performance Computing (TCHPC) Early Career Researchers Award for Excellence in High Performance Computing.

**Weijie Zheng** is a PhD candidate with Department of Computer Science and Technology, Tsinghua University. He is interested in the optimization methods, with a special focus on differential evolution to figure out the effect of each element in the whole process. He won the Best Paper Award of IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2015.

**Haohuan Fu** is a full professor with Department of Earth System Science, Tsinghua University and the Deputy Director of National Supercomputing Center in Wuxi, China. His research interests include extreme-scale computing on heterogeneous supercomputers, data mining methods for analyzing scientific data sets, and programming tools. He is the recipient of the 2016 and 2017 Gordon Bell Prize, the 2015 Tsinghua-Inspur Computational Earth Science Young Researcher Award, and People of the Year Award 2016 by the Scientific Chinese magazine.

**Guangwen Yang** is a full professor with Department of Computer Science and Technology, Tsinghua University and the director of National Supercomputing Center in Wuxi, China. His research interests include parallel and distributed algorithms, cloud computing, and the earth system model. He has published more than 150 research publications in refereed conferences and journals such as SC, USENIX ATC, IPDPS, IEEE TC, and ACM TACO.