SW-AES: Accelerating AES Algorithm on the Sunway TaihuLight

Liandeng Li[†‡§], Jiarui Fang[†‡§], Jinlei Jiang[†], Lin Gan[†‡§], Weijie Zheng[†], Haohuan Fu,[‡§], and Guanwen Yang[†‡§]
[†]*Department of Computer Science & Technology, Tsinghua University*
[‡]*Ministry of Education Key Lab. for Earth System Modeling, Department of Earth System Science, Tsinghua University*
[§]*National Supercomputing Center in Wuxi, Jiangsu, China*

***Abstract*—The Advanced Encryption Standard (AES) is a widely-used efficient cryptographic algorithm. Though 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 algorithms. This paper presents SW-AES, a parallel AES implementation on the Sunway TaihuLight, the fastest supercomputer in the world that takes the SW26010 many-core processor as the basic building block. According to the architectural features of SW26010, SW-AES exploits parallelism from different levels, including 1) inter-CPE (Compute-Processing Element) 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. As a result, SW-AES can gain a maximum throughput of 107.9 Gbps on a single SW26010 node, which is 53× 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.**

***Keywords*-AES encryption, Sunway supercomputer, information security, SIMD, vectorization**

I. Introduction

The rapid development of information science has resulted in a lot of novel technologies that benefit human activities. One good result is the dawn of the big data era where large amount of data is generated, recorded, or analyzed by modern computers to provide essential information and guidance in various application domains. However, along with the continuous surge of data, severe threats and challenges also arise in terms of data protection. People nowadays want more secure approaches for protecting various important information such as classified data, enterprise secrets, or personal privacies. Encryption is one of the most popular solutions to achieve this goal, and has been widely applied in many fields to protect information.

Supercomputers are powerful facilities widely-used in various key fields such as national defence [1], scientific and industrial computing [2][4], and machine learning [3]. Since it is time-consuming to do data encryption, 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 easy task to achieve the purpose, for many factors are usually involved, including not only the features of the algorithm itself but also the characteristics of the underlying system.

The Sunway TaihuLight system [4] is the world's most powerful supercomputer. 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 a hundred large-scale applications have been deployed on it, including climate modeling [5][6][7], material science [8][9], big data [3][10], and so on. In spite of the fact, it is challenging for applications to make full use of the system to gain the best performance. Taking the widely-adopted Advanced Encryption Standard (AES)[11] algorithm as an example, we in this paper show how to fit it with the Sunway TaihuLight to get the best throughput.

Unlike the previous work [12] that tries to improve AES performance by utilizing more nodes whereas resources within a single node are underutilized, our work focuses on accelerating AES algorithm by making full use of such features of the SW26010 processor as heterogeneous-core architecture, multi-hierarchy memory, direct memory access (DMA), two instruction pipelines available, and so on. Our main contributions are as follows.

1) We present a way to vectorize AES operations inside 256-bit registers.
2) We propose new parallel mechanism to fully utilize the on-chip Computing Processor Elements (CPEs) and the two instruction pipelines within each CPE.
3) 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 107.9 Gbps on a single SW26010 node, higher than that of the related work.

II. Background

A. AES Algorithm

As shown in Fig. 1, AES 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 in a 4-by-4 array of bytes, called the $state$. The $roundkey$ size can be 128, 192 or 256 bits. The number of rounds repeated in the AES, Nr, is defined by the length of

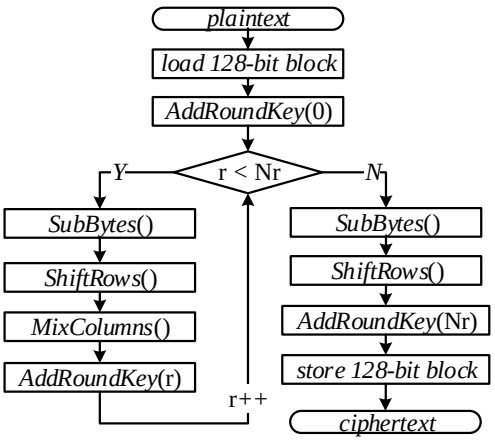

Figure 1: The workflow of AES algorithm.

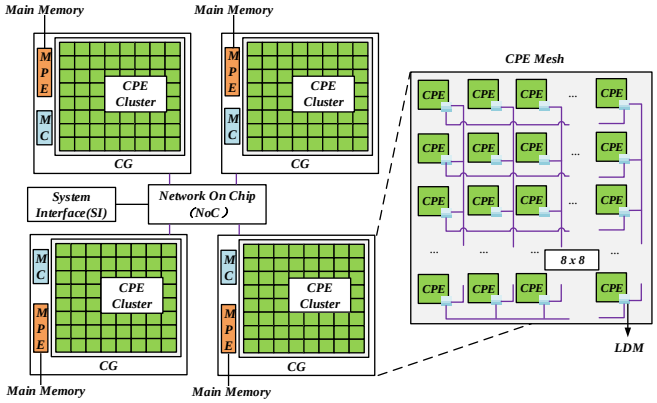

Figure 2: The general architecture of the SW26010 many-core processor.

the key, which is 10, 12 or 14 for key lengths of 128, 192 or 256 bits, respectively. Below are four basic transformations in AES.

1) SubBytes: Each byte $state[i, j]$ in the $state$ matrix is replaced with value of $S\text{-}Box(state[i, j])$ using 16-by-16 array of bytes substitution box, called $S\text{-}box$. The $S\text{-}box$ is computed before the AES encryption by deriving from the multiplicative inverse over $GF(2^8)$, which is a finite field known to have good non-linearity properties.

2) ShiftRows : In step, we cyclically left shifts every row i of the state matrix by i, $0 \leq i \leq 3$.

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

4) AddRoundKey: In this step, the the r-th $roundkey$ is added by combining each byte of the $state$ with the corresponding byte of the the r-th $roundkey$ using bitwise XOR. As a result, $roundkey$s can be calculated before the encryption process, and kept constant during encryption time.

In this paper, we only consider the encryption phase, because, during the decryption phase, the ciphertext can be transformed back into the original plaintext by applying a set of reverse rounds using the same encryption key. Although 128-bit, 192-bit, or 256-bit key size can be selected, we discuss only 128-bit in this paper. Our work can be easily extended to different key sizes.

B. The SW26010 Many-Core Processor

As shown in Fig. 2, each SW26010 processor consists of four *core groups (CGs)*. Each CG includes 65 cores: one *management processing element (MPE)*, and 64 *Computing Processor Element (CPEs)*, organized as an 8 by 8 mesh. The MPE and CPE are both 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 integer numbers) instructions (including fused multiply-add, FMA) with 32 vector registers (extended from 32 64-bit general purpose registers) [24], but play different roles in the computation. The MPE, supporting the complete interrupt functions, memory management, superscalar, and out-of-order issue/execution, is good at handling the management, task schedule, and data communications. The CPE is designed for the purpose of maximizing the aggregated computing throughput while minimizing the complexity of the micro-architecture.

As for the memory hierarchy, each CG connects to its own 8GB DDR3 memory through the *Memory Controller (MC)*, shared by the MPE and the CPE mesh. The on-chip network (NoC) connects four CGs with *System Interface (SI)*. Memory of four CGs are also connected through the NoC. Users can explicitly set the size of each CG's private memory space, and the size of the memory space shared among the four 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 64KB *Local directive Memory (LDM)* (also known as *Scratch Pad Memory (SPM)*) as a user-controlled fast buffer. This user-controlled "cache", while increases the programming challenges for an efficient utilization of the fast buffer, provides the 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 by 8 CPE

mesh, providing an important data sharing capability at the CPE level.

Each CPE includes two pipelines ($P0$, and $P1$) for the instruction decoding, issuing, and execution. $P0$ is for floating-point operations, and both floating-point and integer vector operations. $P1$ 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.

C. Related Works

As an important option to provide high data security, the advanced AES algorithm has been studied on different HPC accelerators, in order to guarantee a satisfactory performance. GPU has long become a popular platform to accelerate the AES algorithm. Harrison et. al [13] presented the first implementation of AES on GPU in 2007. They achieved a throughput of 870 Mbps on Geforce 7900GT GPU using Direct X9. Nishikawa et. al [14] and Iwai et. al [15] 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 [16] implemented the encryption of AES-ECB algorithm and achieved a throughput of 31.7 Gbps on NVIDIA GT200 GPU. The best AES performance they can achieve is 35 Gbps on Geforce GTX285 GPU by adopting 16Bytes per thread as the parallel granularity and storing $S\text{-}Box$es in shared memory. Nishikawa et. al [17] further evaluated AES based on previously reported insights and achieved 50.6 Gbps using NVIDIA Tesla C2050 GPU. In [18], the authors studied the AES encryption on Tesla c20 with innovative Read-Only cache to store $S\text{-}Box$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 [19] has showed a scheme of restructured the CPU-based bitsliced implementation of the AES [20], to process four 16-byte blocks at a time and achieved 78.6 Gbps throughput on GTX 480 GPU.

On the other hand, parallel designs of AES based on reconfigurable platforms such as FPGAs also attracted a lot of attentions. Wang et. al [21] proposed an optimization solution for AES used in storage area network applications based on FPGA XC6VLX240T. A throughput of 78.2 Gbps is achieved as a result. In [22],Liu et. al proposed a single pipeline design for the AES encryption algorithm based on FPGA, and managed to achieve a throughput of 66.1 Gbps. However, even if FPGAs can achieve higher power efficiency, their overall performance is limited by the total amount of hardware computing resources.

Till now, we only see a few efforts that design the parallel AES-ECB algorithm based on the latest SW26010 processor. The work in [12] has showed a distributed implementation of AES algorithm on the Sunway TaihuLight system. Although a good scalability is achieved, i.e. 998x speedup with MPI on 1024 CGs compared with one CG, the performance of their implementation in a single SW26010 node is very poor, which is only 2.0 Gbps. In their work, plaintext data was processed by computing cores in parallel and no fine-grain vectorization optimization was exploited inside computing cores.

III. SW-AES Optimization Framework

To map AES workflow to the innovative SW26010 many-core architecture, our SW-AES is focused on exploiting possible parallelism provided by the architecture from three perspectives.

1) Inter CPEs, we utilize 256 CPEs to conduct AES encryption of independent plaintext blocks in parallel to exploit data parallelism between CPEs.
2) Intra CPE, SIMD instructions can operate on several parts of vector registers in parallel. We can exploit the data parallel parallelism by SIMD operations supported by 256-bit width register.
3) Intra CPE, two instruction pipeline can perform (different stages of) several instructions at once. In addition, the two pipelines in an CPE can both execute instructions in parallel. Instruction-level parallelism can be exploited by workflow adjustment and instruction scheduling to overlap instruction executions on two instruction pipelines.

A. Inter-CPE Parallelism

In AES, the plaintext can be divided into several independent fixed 16-byte plaintext blocks to be processed in parallel. Therefore, it is naturally 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 four CGs by their MPEs to encrypt one-fourth 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 to transfer the plaintext blocks before computing and ciphertext blocks after computing between the main memory and registers of CPEs. The CPE mesh can access the 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 the data items from memory by using 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 and wasting 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 ciphertext 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. The effective bandwidth for DMA load and store ranges from 4 GB/s to 28 GB/s. In general, a higher bandwidth over 22 GB/s is achieved when using a block size larger than 256B. 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. While the data is computed in one LDM buffer, the next plaintext block is loaded into another LDM buffer by DMA.

B. Intra-CPE 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 algorithms in one CPE is not a straightforward task. The data structure to be processed during AES workflow is only 128 bit, which makes it difficult to exploit the 256-bit SIMD parallelism provided. In addition, operations critical to cryptography like `vsrlw`, `seleqw`, which are used to already known solution [23], are not supported in CPEs of SW26010. To make matter worse, the instruction set does not support SIMD operation on 1-byte elementary data type. While the AES workflow includes $S\text{-}Box$ table random look-up operations, which work in bytewise and bitwise fashion. Furthermore, different from other popular parallel architectures, like GPU and Xeon Phi, that adopt an occupancy-oriented parallel model, where another group of threads can be issued to instruction execution pipeline when one group of thread is stalled, SW26010 requires programmer to explicitly control the instruction execution sequence. Finally, the compilers on SW26010 provide no automatic vectorization optimizations to exploit SIMD capacity, therefore programming in such an environment poses a significant challenge.

To resolve the above difficulties, we propose three optimization techniques to exploit intra-CPE parallelism, including a SIMD-friendly data layout to easily explore data parallelism inside CPE, a semi-SIMD random $S\text{-}Box$ look-up to eliminate the performance damage from unvectorized LDM access and a Pipelined Step Execution workflow to fully utilize instruction pipelines. For clarity, before introducing above methods in detail, we present the data structures used in Table. I.

*1) **SIMD-friendly Data Layout**:* A SIMD-friendly data layout poses the data elements that can be executed with the same instruction operation together during AES workflow in one 256-bit register. It is essential to enable AES workflow perform in a SIMD-fashion. After DMA fetch operation,

Table I: Data structure used in AES workflow

Name	Size	Location	Description
$state$	128 bit		Mentioned in Sec. II-A1
$state\ batch$	128×8 bit	4 registers	8 $state$ arrays perform transformation together
$plaintext\ block$	128x B	LDM	A block of plaintext fetched by DMA
$S\text{-}Box$	256 B		Mentioned in Sec. II-A1
$ExtS\text{-}Box$	1 KB	LDM	Extend each Byte in $S\text{-}Box$ into integer
$Round\ Key$	240 B		Mentioned in Sec. II-A4
$ExtRound\ Key$	2 KB	LDM	Extend each Byte in $Round\ Key$ into integer, and aligned in 32B

a $plaintext\ block$ is maintained in the local LDM of each CPE. A 128-bit piece of $plaintext\ block$ called $state$, which can be viewed as a 4-by-4 byte array, should be loaded from LDM to registers, updated by encryption steps mentioned in II-A iteratively to form a piece of cipher text and then stored back to the main memory. Merely conducting AES on one $state$ array for computing is insufficient to exploit the SIMD capacity of 256-bit vector registers. The fact that operations to update different 128-bit $state$ arrays can be the same brings data parallelism by applying one SIMD instruction to multiple independent $state$ array updates. We conduct AES workflow on eight $state$ data as a batch together, for the reason that a SIMD instruction is able to operate on at most eight integers in parallel. However, the byte elements that can be used to perform the same operation are not contiguous, therefore we can not directly load them into a single vector register. To form a SIMD-friendly data layout, the element of eight $state$s should be stored together in a SIMD-friendly layout by a serial of transformation operations.

To guarantee no signification overhead introduced during transformations between the layout of plain-text data and the SIMD-friendly layout, we introduce a low-cost transformation scheme, which also works with SIMD instructions. To move eight continuous $state$ arrays from LDM to four registers, we use four *vldd* instructions, each of which can load 256 bit aligned data from LDM to vector registers. Therefore, each register maintains two $state$ arrays inside. Fig. 3 illustrates our SIMD-fashion transformation scheme to transform $state$ arrays into the SIMD-friendly data layout inside registers. We only describe the transformation of plaintext loading phase, since the ciphertext storing phase can be easily implemented with the corresponding inverse instructions. In the first three steps, four byte elements in a $state$ can be viewed as a integer element. Transformation works in granularity of integer with shuffle instructions of different masks. Different colors in the figure distinguish row numbers of arrays. After three steps, we can observe that 32 byte elements, which belong to the same row of different $state$ arrays, are collected together in a single register vector.

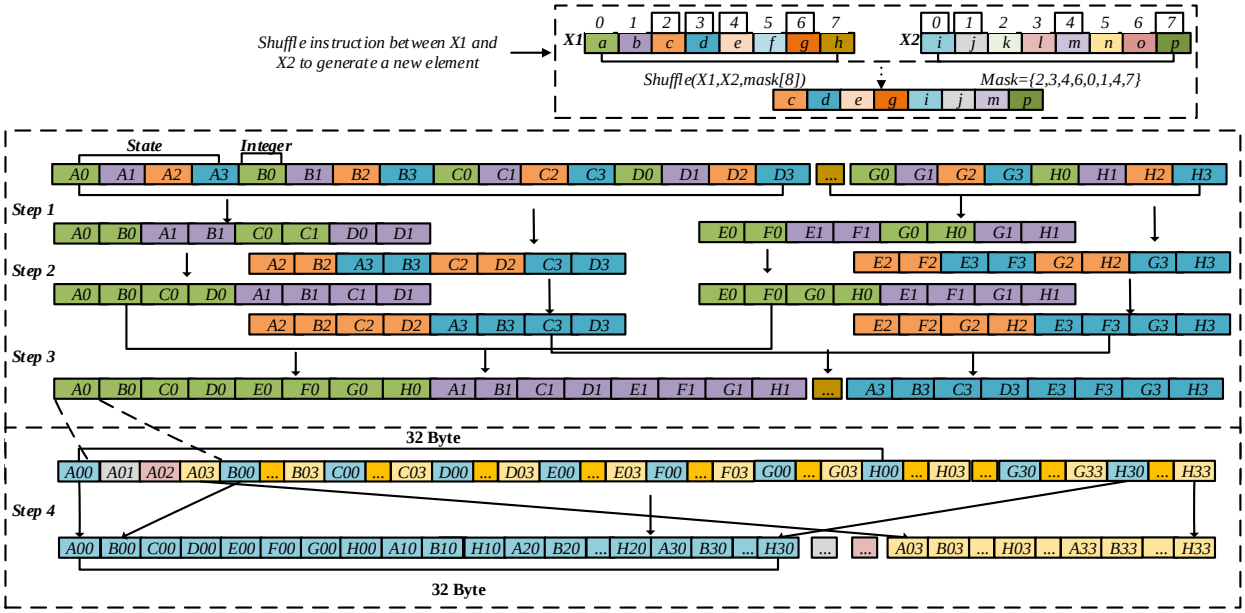

Figure 3: Transformation of eight 128-bit $state$ arrays into a SIMD-friendly data layout. In the top of the figure, we illustrate how the instruction *Shuffle(X1,X2, mask)* works. X1 and X2 are two 256-bit registers which contains 8 integer numbers. The array mask, which is *0x2340147* in this case, provides the information of which four numbers from each register are being taken to the new register. In this case, the first four numbers comes from X1, while the other four numbers are from X2.

In Step 4, we extract each byte inside the 4-byte integer elements and collect them together in a vector. The colors in the figure indicate the positions of the byte element in a row of the 4-by-4 byte array. Vectorized SHIFT and AND instructions can be used to move byte elements in fours vectors to form our SIMD-friendly data layout at last. The transformation is very efficient, which only takes up 2%-3% of the whole workflow.

To adapt to our SIMD-friendly data layout, we also make customized transformation to the $RoundKey$ used in our batch workflow. During each iteration, we use a 16-byte piece of the $RoundKey$ for each round. As shown in Fig. 3, the SIMD-friendly data layout stores a batch of $state$ in column-major format. As a result, we transpose the 4-by-4 byte $RoundKey$ piece from row-major into column-major before encryption. To facilitate the vectorized operations, we also duplicate every byte element eight times in $RoundKey$ to form the $ExtRoundKey$.

2) ***Semi-SIMD fashion S-Box Look up****:* The data layout design proposed previously has already been able to fully exploit the SIMD parallelism of operations in Step ShitRows, MixColumns and AddRoundKey. However, the Step SubBytes still poses a major hurdle to fully utilize SIMD parallelism. This step randomly looks up $S\text{-}Box$ table with every byte element of the $state$ array as index and update $state$ with elements found from $S\text{-}Box$. Therefore, its random LDM access pattern is hard to work in a SIMD fashion. A naïve implementation is to extract the byte elements to be updated from vectors and convert them into scalars. Scalar load operations are used to access the LDM table and then insert fetched elements back to the original vector sequentially. It will result in a lot of CPU cycles wasted on scalar extraction and insertion operations on vector elements, which can not be conducted in SIMD fashion.

To eliminate the overhand from vector and scalar transformation, based on our SIMD-friendly data layout design, a three-stage Semi-SIMD-fashion $S\text{-}Box$ look-up strategy is proposed, as illustrated in Fig. 4. In extend stage, we extend each byte of the input vectors into integers and therefore split original vector into four splitted $intv8$ vectors. It can be implemented with vectorized SHIFT and AND operations, i.e., three `vsrlw` instructions and four `vandw`. In look-up table stage, by taking advantage of the unique instruction `selldw` provided by SW26010 instruction set, we can avoid extraction and insertion operations to vectors and keep updated data always in vectors. Instruction `selldw Va,`

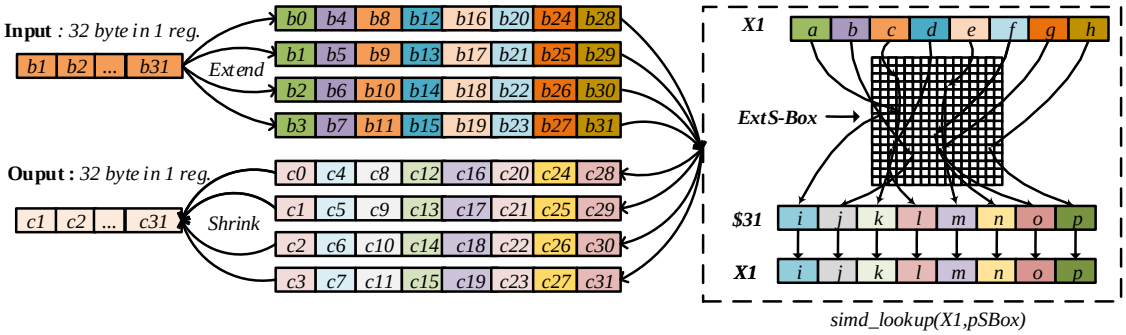

Figure 4: Vectorized random lookup table operation.

`Rb, #c, Vd` is designed for `sel`ecting and `l`oading `w`ord data from LDM. The 32 bit operand `Rb` indicates the starting address of the memory space to be looked up in LDM. The 256 bit operand `Va` in the form of $intv8$ contains eight offsets in format of integer indicating the target positions to `Rb`. Operand `#c` is an immediate operand, in the range 0 to 7, indicating the value of which integer of `Va` is used as the offset. Considering the `selldw` works in unit of integer, we replace $S\text{-}Box$ as $ExtS\text{-}Box$. Eight `selldw` instructions work sequentially to finish the $ExtS\text{-}Box$ table look-up with one $intv8$ vector. It caches the return values in a temporary register and move them to `Vd` registers afterwards. Finally, in shrinking stage, we shrink integers to bytes and combine elements in four integer vectors into one byte vector.

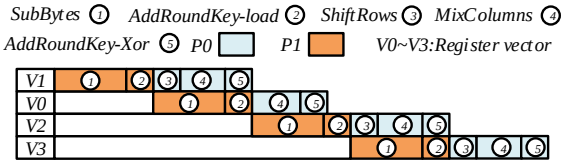

Figure 5: Pipelined workflow to increase Instruction-Level-Parallelsim (ILP).

3) ***Pipelined Step Execution Workflow****:* Each CPE of SW26010 processor has two types of instruction execution pipelines, i.e., *P0* and *P1* pipeline. 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 instructions execution pipelines separately, we can exploit the Instruction-Level-Parallelsim (ILP) inside computing unites.

We proposed a Pipelined Step Execution technique to balance the utilization of two instruction execution pipelines to increase ILP. We noticed that the execution steps of the AES workflow can be categorized into two different types as $P0$-pipeline bounded and $P1$-pipeline bounded kernels, according to their computing patterns. Step SubBytes, as mentioned by previous subsection, is mainly LDM access operations on $P1$-pipeline. Step ShiftRows, which includes vectorized shift operations, and MixColumns, which includes a set of logical instructions, can be executed on $P0$-pipeline. Step AddRoundKey is divided into two parts. The part which requires to access a distinguish part of 128 Byte $ExtRound\ Key$ each round can be implemented with vectorized load instruction executed on $P1$-pipeline. The part that performs vectorized XOR operations is $P0$-pipeline bounded. Without affecting the dependencies of the AES workflow, we rearrange the workflow of AES into pipeline-fashion as depicted in Fig. 5. By overlapping execution of $P0$-bounded and $P1$-bounded steps of independent $state$ data, we can fully utilize two pipelines.

IV. Performance Evaluation and Analysis

We implement SW-AES using a two-level multi-threading programming model, that is, we use MPI to launch four processes to the four CGs and Athread to launch 64 light-weight threads to the 64 CPEs within one CG. 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 of the Sunway TaihuLight, with the configurations listed in Table II.

A. SW-AES Performance

1) Overall performance: Fig. 6 shows the throughput of SW-AES with different input data sizes. It is easy to see that

Table II: System Comfigurations

Item	Value
Processor	SW26010
OS	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 throughput of SW-AES increases almost linearly with the input data size increasing from 1KB to 8MB. When the input data size is larger than 16 MB, it is able to achieve a throughput over 100 Gbps. The throughput increase drops after the input size is greater than 16MB and stops at 256MB because the computing resources (i.e., CPEs) are insufficient.

To make the performance gains clearer, Fig. 7 illustrates the proportion of calculation time, DMA data transfer time, thread start-up time and the calculation time overlapped by double buffer. We can see that the poor performance when the input size is less than 1MB is mainly caused by the thread start-up time on CPE. An interesting fact is that double-buffer technique, which is also used in [12], has limited the benefit to the overall performance. The reason comes from the special DMA mechanism of the SW26010 processor. CPE mesh conducts DMA operations in group of 4 CPEs. Sequential DMA operations are performed between groups. Computation of one CPE group has already been overlapped with DMA of another CPE group even if we use single-buffer for DMA operation. As a result, it is a very common phenomenon to observe only limited improvement with the double buffer techniques on SW26010.

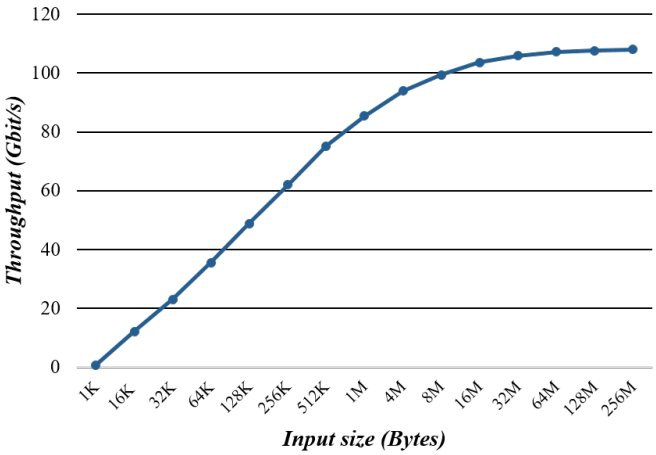

Figure 6: The SW-AES overall performance under various plaintext block sizes.

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 performance. Fig. 8 shows the results on one CG with respect to different input sizes (from 1K to 256M). Different colors in the figure

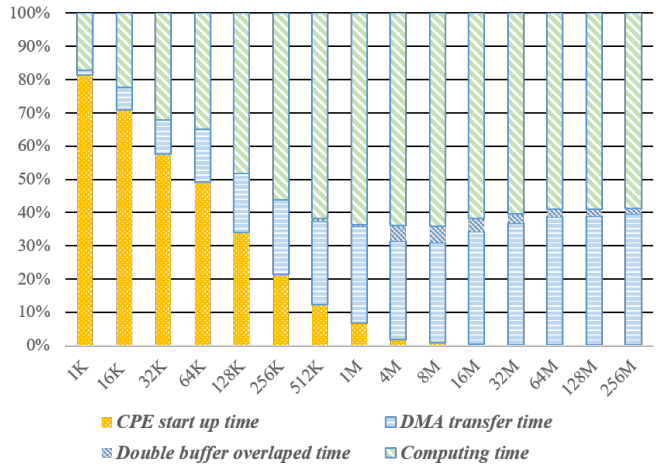

Figure 7: Percentage of calculation time, double buffering overlap time, data transfer time and thread startup time of AES workflow under various plaintext block sizes.

refer to the original no-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 look-up 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 result in straightforward performance improvements in spite of the input size. The reason is that the non-vectorized $S\text{-}Box$ loop-up operation produces a huge amount of overhead when extracting elements from and inserting scalar elements into vector registers. However, the proposed semi-SIMD fashion look-up table scheme is able to produce obvious benefits for all input sizes, with speedups 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.

B. Comparison with Related Work

1) Comparison with the work on the Sunway TaihuLight: The work reported in [12] is the only related work done on the Sunway TaihuLight. The comparison between SW-AES and it is shown in Fig. 9. In accordance with the settings in [12], only the block sizes between 1MB and 256MB are used. It is easy to see from the figure that SW-AES outperforms greatly than the work in [12], with a throughput improvement about 53.95× when the block size is 256MB. Indeed, more throughput improvement can be gained with smaller block size. The reason is that the work in [12] did not take into consideration the fine-grained vectorization and pipelined execution as we do.

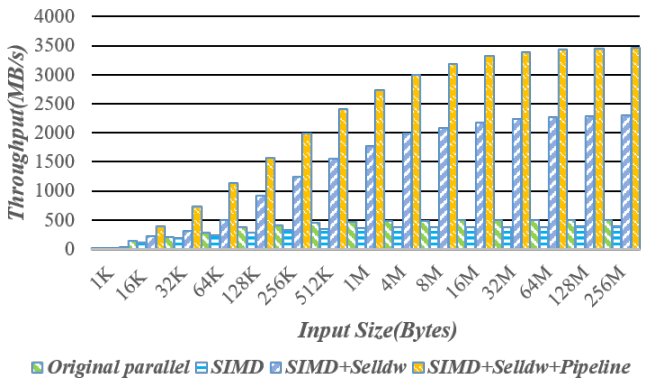

Figure 8: The effect of different optimization techniques on the SW-AES overall performance under various plaintext block sizes.

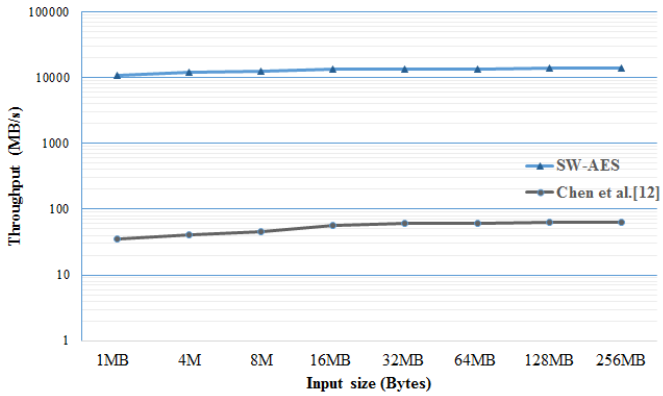

Figure 9: Throughput comparison between SW-AES and the work in [12] under various plaintext block sizes.

2) Comparison with other related work: As aforementioned in Section II-C, there are also other ways to boost AES algorithm besides the work on the SW26010 processor. Table III listed the comparison result between SW-AES and other work on FPGA [22][21], GPU [15][17][16][19]. 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 107.9 Gbps can be achieved due to the novel techniques proposed. It can be seen from the table that SW-AES outperforms the work on the mainstream HPC accelerators such as FPGA and GPU, with a throughput improvement ranging from 1.37× to 3.40×. Since different hardware devices are used, the comparison can only be used for reference. Anyway, it is a promising and efficient solution to adopt SW-AES for data processing on the Sunway TaihuLight supercomputer.

Table III: Performance compared with other work

Hardware Device	Throughput	Speedup	Work
SW26010	107.9 Gbps	-	this paper
FPGA XC7VX690T	66.1 Gbps	1.63×	[22], 2013
FPGA XC6VLX240T	78.2 Gbps	1.38×	[21], 2016
GeForce GTX 285	35.2 Gbps	3.07×	[15], 2010
Tesla C2050	50.6 Gbps	2.13×	[17], 2011
NVIDIA GT200	31.7 Gbps	3.40×	[16], 2015
GeForce GTX 480	78.6 Gbps	1.37×	[19], 2016

V. Conclusion

In this paper, we reported our effort on accelerating AES encryption algorithm on the Sunway TaihuLight, the fastest supercomputer 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 107.9 Gbps on a single SW26010 processor. Such a throughput is 53.95× higher than that of the latest work done on the Sunway TaihuLight [12]. It 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 should point out that the throughput of 107.9 Gbps is achieved on a single node of the Sunway TaihuLight. Much higher throughput can be obtained by using more computing nodes as done in [12]. In the future, we will do it. Also, we will investigate ways to further improve AES operations on a single node.

References

[1] Archer B J. Seventy Years of Computing in the Nuclear Weapons Program. *Los Alamos National Laboratory (LANL)*, 2015.

[2] Zhang J, Zhou C, Wang Y, et al. 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*,page 4,2016.

[3] Fang J, Fu H, Zhao W, et al. swDNN: A Library for Accelerating Deep Learning Applications on Sunway TaihuLight. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*,pages 615-624,2017 .

[4] Haohuan Fu, Junfeng Liao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, pages 1–16, 2016.

[5] Chao Yang, Wei Xue, Haohuan Fu, Hongtao You, Xinliang Wang, Yulong Ao, Fangfang Liu, Lin Gan, Ping Xu, Lanning 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*, page 6,2016.

[6] Fu H, Liao J, Xue W, et al. Refactoring and optimizing the community atmosphere model (CAM) on the sunway taihulight supercomputer. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for IEEE*,pages 969-980,2016.

[7] Ao Y, Yang C, Wang X, et al. 26 PFLOPS Stencil Computations for Atmospheric Modeling on Sunway TaihuLight. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 535-544,2017.

[8] Dong W, Kang L, Quan Z, et al. Implementing Molecular Dynamics Simulation on Sunway TaihuLight System.High Performance Computing and Communications; In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on. IEEE* ,pages 443-450,2016.

[9] Liu J, Qin H, Wang Y, et al. Largest Particle Simulations Downgrade the Runaway Electron Risk for ITER. *arXiv preprint arXiv:1611.02362*,2016.

[10] Lin H, Tang X, Yu B, et al. Scalable Graph Traversal on Sunway TaihuLight with Ten Million Cores. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*,pages 635-645,2017.

[11] National Institute of Standards and Technology,Specification for the Advanced Encryption Standard (AES), *Federal Information Processing Standards Publication 197*, November 26, 2001.

[12] Chen Y, Li K, Fei X, et al. Implementation and Optimization of AES Algorithm on the Sunway TaihuLight. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2016 17th International Conference on.IEEE*,pages 256-261,2016.

[13] Harrison.Owen,Waldron.J, AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. *Proc.of the 9th Workshop on Cryptographic Hardware and Embedded Systems(CHES 2007),Vienna,Austria,September 10-13,2007*. pages 209-226,2007.

[14] Nishikawa.N,Iwai.K,and Kurokawa.T,Granularity optimization method for AES encryption implementation on CUDA . *IEICE Technical Report. VLSI Design Technologies (VLD2009-69) (2010)*.pages 107-112,2010.

[15] Iwai.K,Kurokawa.T,and Nishikawa.N, AES encryption implementation on CUDA GPU and its analysis. In *Networking and Computing (ICNC), 2010 First International Conference on. IEEE*,pages 209-214,2010.

[16] Guo G, Qian Q, Zhang R. Different implementations of AES cryptographic algorithm. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE*,pages 1848-1853.2015.

[17] Nishikawa.N,Iwai.K,and Kurokawa.T, High-Performance Sysmmetric Block Ciphers on CUDA. In *Networking and Computing (ICNC), 2011 Second International Conference on. IEEE*, pages 221-227,2011.

[18] Khan A H, Al-Mouhamed M A, Almousa A, et al. AES-128 ECB encryption on GPUs and effects of input plaintext patterns on performance. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on. IEEE*,pages 1-6,2014.

[19] Lim R K, Petzold L R, et. al. Bitsliced High-Performance AES-ECB on GPUs. *The New Codebreakers. Springer Berlin Heidelberg*, pages 125-133,2016.

[20] Kaşper, Emilia, and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *CHES 2009,11th International Workshop,Lausanne,Switzerland*,pages 1-17,2009.

[21] Wang Y, Ha Y. High throughput and resource efficient AES encryption/decryption for SANs. In *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on. IEEE*, pages 1166-1169,2016.

[22] Liu Q, Xu Z, Yuan Y. A 66.1 Gbps single-pipeline AES on FPGA. In *Field-Programmable Technology (FPT), 2013 International Conference on. IEEE*,pages 378-381,2013.

[23] Bos J W, Osvik D A, Stefan D. Fast Implementations of AES on Various Platforms. *IACR Cryptology ePrint Archive 2009/501*, 2009.

[24] NSCCWX, Sunway taihulight compiler user guide, *Online. Available: http://www.nsccwx.cn/.*