swGBDT: Efficient Gradient Boosted Decision Tree on Sunway Many-Core Processor

Bohong Yin¹, Yunchun Li^{1,2}, Ming Dun², Xin You¹, Hailong Yang^{1(✉)},
Zhongzhi Luan¹, and Depei Qian¹

¹ School of Computer Science and Engineering, Beihang University, Beijing, China
{15061139,lych,youxin2015,hailong.yang,07680,depei.q}@buaa.edu.cn

² School of Cyber Science and Technology, Beihang University, Beijing, China
dunming0301@buaa.edu.cn

Abstract. Gradient Boosted Decision Trees (GBDT) is a practical machine learning method, which has been widely used in various application fields such as recommendation system. Optimizing the performance of GBDT on heterogeneous many-core processors exposes several challenges such as designing efficient parallelization scheme and mitigating the latency of irregular memory access. In this paper, we propose *swGBDT*, an efficient GBDT implementation on Sunway processor. In *swGBDT*, we divide the 64 CPEs in a core group into multiple roles such as *loader*, *saver* and *worker* in order to hide the latency of irregular global memory access. In addition, we partition the data into two granularities such as *block* and *tile* to better utilize the LDM on each CPE for data caching. Moreover, we utilize register communication for collaboration among CPEs. Our evaluation with representative datasets shows that *swGBDT* achieves 4.6× and 2× performance speedup on average compared to the serial implementation on MPE and parallel XGBoost on CPEs respectively.

Keywords: Gradient Boosted Decision Tree · Many-core processor · Performance optimization

1 Introduction

In recent years machine learning has gained great popularity as a powerful technique in the field of big data analysis. Especially, Gradient Boosted Decision Tree (GBDT) [6] is a widely used machine learning technique for analyzing massive data with various features and sophisticated dependencies [17]. GBDT has already been applied in different application areas, such as drug discovery [24], particle identification [18], image labeling [16] and automatic detection [8].

The GBDT is an ensemble machine learning model that requires training of multiple decision trees sequentially. Decision trees are binary trees with dual

judgments on internal nodes and target values on leaves. GBDT trains the decision trees through fitting the residual errors during each iteration for predicting the hidden relationships or values. Thus the GBDT spends most of its time in learning decision trees and finding the best split points, which is the key hotspot of GBDT [9]. In addition, GBDT also faces the challenge of irregular memory access for achieving optimal performance on emerging many-core processors such as Sunway [7].

Equipped with Sunway SW26010 processors, Sunway TaihuLight is the first supercomputer that reaches the peak performance of over 125 PFLOPS [4]. The Sunway processor adopts a many-core architecture with 4 Core Groups (CGs), each of which consists of a Management Processing Element (MPE) and 64 Computation Processing Elements (CPEs) [7]. There is a 64 KB manually-controlled Local Device Memory (LDM) on each CPE. The Sunway many-core architecture also provides DMA and register communication for efficient memory access and communication on CPEs.

In this paper, we propose an efficient GBDT implementation for the Sunway many-core processor, which is an attractive target for accelerating the performance of GBDT with its unique architectural designs. The hotspot of GBDT can be further divided into two parts: 1) sorting all the feature values before computing the gains; 2) computing gain for every possible split. To speedup the hotspot, we partition the data into finer granularities such as *blocks* and *tiles* to enable efficient data access on CPEs. To improve the performance of sorting, we divide CPEs into multiple roles for pipelining the computation of segmenting, sorting, and merging with better parallelism. We evaluate the optimized GBDT implementation *swGBDT* with representative datasets and demonstrate its superior performance compared to other implementations on Sunway.

Specifically, this paper makes the following contributions:

- We propose a memory access optimization mechanism, that partitions the data into different granularities such as *blocks* and *tiles*, in order to leverage LDM and register communication for efficient data access on CPEs.
- We propose an efficient sorting algorithm on Sunway by segmenting and sorting the data in parallel and then merging the sorted sequences. During the sorting and merging, we divide the CPEs into multiple roles for pipelining the computation.
- We implement *swGBDT* and evaluate its performance by comparing with the serial implementation on MPE and parallel XGBoost on CPEs using representative datasets. The experiment results show $4.6\times$ and $2\times$ speedup, respectively.

This paper is organized as follows. We give the brief introduction of GBDT algorithm and Sunway architecture as background in Sect. 2. Section 3 describes our design methodology for *swGBDT*. Section 4 shows the implementation details of *swGBDT*. In Sect. 5, we compare *swGBDT* with both serial GBDT implementation and parallel XGBoost on synthesized and real-world datasets in terms of performance. Related works are presented in Sect. 6 and we conclude our work in Sect. 7.

2 Background

2.1 Sunway Many-Core Processor

The Sunway SW26010 many-core processor is the primary unit in Sunway TaihuLight supercomputer. The illustration of the many-core architecture within a single Core Group (CG) of SW26010 is in Fig. 1. There are four CGs in a single SW26010 processor. The peak double-precision performance of a CG can be up to 765 GFLOPS, while the theoretical memory bandwidth of that is 34.1 GB/s. Moreover, the CG is comprised of a Management Processing Element (MPE), 64 Computation Processing Elements (CPEs) in a 8×8 array and a main memory of 8 GB. The MPE is in charge of task scheduling whose structure is similar to mainstream processors, while CPEs are designed specifically for high computing output with 16 KB L1 instruction caches and 64 KB programmable Local Device Memories (LDMs). There are two methods for memory access from main memory in the CG to a LDM in the CPE: DMA and global load/store (*gld/gst*). DMA is of much higher bandwidth compared to *gld/gst* for contiguous memory access. The SW26010 architecture introduces efficient and reliable register communication mechanism for communication between CPEs within the same row or column which has even higher bandwidth than DMA.

Fig. 1. The many-core architecture of a Sunway core group.

2.2 Gradient Boosted Decision Tree

The Gradient Boosted Decision Tree is developed by Friedman [6]. The pseudocode of GBDT algorithm is presented in Algorithm 1 [20]. The training of GBDT involves values from multiple instances under different attributes and there are

several hyperparameters in GBDT: the number of trees N , the maximum depth of tree d_{max} and the validation threshold of split points β . To store the dataset of GBDT algorithm, the sparse format [20] is developed to reduce the memory cost which only stores the non-zero values instead of values of all attributes in all instances as the dense format. We use the sparse format for *swGBDT*.

Moreover, as shown in Algorithm 1, GBDT trains the decision trees iteratively using the residual errors when the loss function is set to mean squared error. During each iteration, in order to find the best split points which is the bottleneck of GBDT, the algorithm needs to search for the maximum gain in one attribute, which will generate a preliminary split point that is appended to set P , and finally the best split point will be extracted from the set P with a validation threshold constant β . Thus the primary process in searching for best split points are gain computation and sorting. The *gain* among all instances of one attribute can be derived from Eq. 1, where G_L and G_R are the sum of first-order derivatives of loss function in left or right node respectively, while H_L and H_R are the sum of second-order derivatives of loss function in left or right node, respectively. The first-order and second-order derivatives can be computed from Eq. 2 and Eq. 3 respectively, where E is the loss function and it is set to be mean squared error in *swGBDT*.

$$gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] \quad (1)$$

$$g_i = \frac{\partial E}{\partial y_i} \quad (2)$$

$$h_i = \frac{\partial^2 E}{\partial y_i^2} \quad (3)$$

2.3 Challenges for Efficient GBDT Implementation on Sunway Processor

In order to implement GBDT algorithm efficiently on Sunway, there are two challenges to be addressed:

1. How to leverage the unique many-core architecture of Sunway to achieve effective acceleration. Unlike random forest that each tree is independent of each other, the computation of each tree in GBDT depends on the result of the previous tree, which prevents the tree-level parallelism. Therefore, we need to design a more fine-grained parallel scheme to fully utilize the CPEs for acceleration.
2. How to improve the efficiency of memory access during GBDT training. The large number of random memory accesses during GBDT training lead to massive *gld/gst* operations with high latency. The poor locality with random memory access deteriorates the performance of GBDT. Therefore, we need to design a better way to improve memory access efficiency.

Algorithm 1. GBDT Algorithm

```

1: Input:  $\mathbf{I}$ ,  $d_{max}$ ,  $\beta$ ,  $N$ 
2: Output:  $\mathbf{T}$ 
3:  $\mathbf{T} \leftarrow \phi$ 
4: for  $i = 1 \rightarrow N$  do
5:    $T_i \leftarrow \text{Tree\_Init}(\mathbf{T})$ 
6:    $\mathbf{P} \leftarrow \phi$ 
7:    $\mathbf{N} \leftarrow \text{RootNode}(T_i)$ 
8:    $\mathbf{A} \leftarrow \text{GetAttribute}(\mathbf{I})$ 
9:   for  $n \in \mathbf{N}$  do
10:    if  $d > \text{GetDepth}(n)$  then
11:       $\mathbf{I}_n \leftarrow \text{GetInstance}(n)$ 
12:      for each  $A \in \mathbf{A}$  do
13:         $g_m \leftarrow 0$ 
14:         $\mathbf{V}_n \leftarrow \text{GetAttributeValue}(A, n)$ 
15:         $(g_m, p) \leftarrow \text{MaxGain}(\mathbf{I}_n, \mathbf{V}_n, A)$ 
16:         $P \leftarrow \text{GetNewSplit}(P, (A, g_m, p))$ 
17:      end for
18:       $(A^*, g^*_m, p^*) \leftarrow 0$ 
19:      for each  $(A, g_m, p) \in P$  do
20:        if  $(g^* < g \text{ and } g > \beta)$  then
21:           $(A^*, g^*_m, p^*) \leftarrow (A, g_m, p)$ 
22:        end if
23:      end for
24:      if  $g^*_m = 0$  then
25:         $\text{RemoveLeafNode}(n, \mathbf{N})$ 
26:      else
27:         $(n_1, n_2) \leftarrow \text{SplitNode}(n, A^*, p^*)$ 
28:         $\text{UpdateLeaf}(\mathbf{N}, n_1, n_2)$ 
29:      end if
30:    end if
31:  end for
32: end for

```

3 Methodology

3.1 Design Overview

In this paper, data partitioning and CPE division are used to reduce the time of memory access through prefetching. For data partition, as shown in Fig. 2, firstly, we divide the data into *blocks* evenly according to the number of CPEs participating in the computation. Then we divide the *blocks* into *tiles* according to the available space of every CPE's LDM. When calculating the data from a *tile*, the DMA is used to prefetch the next *tile*. That's how we use the double buffering to hide the data access delay. When multi-step memory access or multi-array access are simultaneously needed (such as computing $A[i] = B[i] + C[i]$ needs access array A, B and C simultaneously), we divide the CPEs into data cores called *loaders* and computing cores called *savers*. *Loaders* prefetch data and

then send it to *savers* for calculating by register communication. Meanwhile, for the reason that sorting operation is the most time-consuming, we propose an efficient sorting method. Firstly, we divide the data to be sorted evenly into 64 segments and sort them separately by 64 CPEs to achieve the maximum speedup ratio. Then we merge the 64 segments by dividing the CPEs into different roles and levels and register communication. Every 128 bits transferred by register communication is divided into four 32-bit length part as shown in Fig. 3.

Fig. 2. The illustration of bistratal array blocking.

3.2 Data Prefetching on CPE

When there is a N -length big array named ARR participating in calculating, we firstly partition it into K blocks evenly (normally 64 when the CPE division in Sect. 3.1 is not needed, otherwise 32), so that every CPE processes a *block*. Because normally the processing time of every element in ARR is the same, so the static partition can achieve load balance. Then every CPE divides its *block* into *tiles* according to its usable LDM size. If the size of a *tile* is too small, more DMA transactions will be needed, whereas the size is too large, the *tile* will not be able to fit the limit of LDM. As a result, we use the equation $T = \frac{M}{\sum_{i=0}^{n-1} P_i}$ to calculate the number of *tiles*, in which T denotes the number of *tiles*, M denotes the LDM usable space size, n denotes the number of arrays that participate in the task, P_i denotes the element size of every array. Because the DMA is an asynchronous operation, it needs no more computation after sending the request, so it can be paralleled with computations. Thus we use the double buffering to hide the DMA time. In the beginning, the CPE loads the first *tile* and sends the DMA request for prefetching the next *tile*, then begins calculating. Every time it finish the calculating of one *tile*, the next *tile* has been prefetching by DMA, so the CPE sends a new DMA transaction for the next uncached *tile* and begins calculating the cached *tile*. That's how our double-buffering works.

In the training process of GBDT, we will face computing tasks like $C[i] = func(A[B[i]])$ which need multi-step memory access. Due to the *gld* operation in the second step, the memory access is of high latency and low bandwidth. Meanwhile, all CPEs access memory at the same time will cause the load of memory controller too heavy. So, we use the CPE division mode at these times, set half of CPEs as data cores called *loader*, another half of CPEs as computing cores called *saver* because they also need to save the final result to the main memory. There is a one-to-one relationship between *saver* and *loader*. The multiple roles

Fig. 3. The message formats used in register communication.

of CPEs and data communication are shown in Fig. 4. The *loader* firstly uses the data partitioning to prefetch *tiles* from array B , then uses the *gld* to get the value of $A[B[i]]$, finally sends it to its *saver* by register communication. We use the communication format in Fig. 3(a). The *saver* computes the $C[i]$ and saves $C[i]$ into the buffer and saves the result to the main memory by a DMA request when fills a buffer.

3.3 Sort and Merge

The sorting of large array is the main hotspot. To make full use of the 64 CPEs and to maximize parallelism, we firstly divide the array evenly into 64 segments, every CPE sorts a segment so that we can get 64 sorted sequences $A_0, A_1, A_2, \dots, A_{63}$, then we merge them to get the final result. As shown in Fig. 5, each round carries out two combined merge, 32 sorted sequences are got after the first round, 16 after the second round and so on, 6 rounds are needed to get the final result. For the reason that unmerged data may be replaced, as shown in Fig. 6, during every merging round, the data source and destination must be different. This means that at least two times of memory reading and writing is needed, reading from source and writing to a temporary location then reading from the temporary location and writing to the source. If we do not implement the data reuse through register communication, each round of merging requires a time of memory reading and memory writing, that are reading data for merging and writing the result into memory. 6 times of data reading and writing is needed for 6 rounds of merge. This will lead to a large amount of data movements which will cause unnecessary time consumption. In this paper, we divide the CPEs into different roles and levels and use register communication to reduce the times of memory reading and writing from six to two. To achieve this goal, the 6 rounds of merge is divided into two steps. The first step only includes the first round of merge, writes the merged data into a temporary location. The second step includes all the last 5 rounds of merge and writes the final result back to source location. Because the CPEs

Fig. 4. The multiple roles of CPEs and data communication.

are divided into different roles and levels and compose a pipeline, the merging results of intermediate rounds are all transmitted by register communication to CPEs that doing the next rounds' mergence instead of writing back to memory, thus only one round of memory reading and writing is needed. The formats of Fig. 3(b) and (c) are used for register communication.

In the first step, the CPEs are divided into two types, *loaders* and *savers*. Each *loader* corresponds to a *saver*, they are in the same row so that register communication can be directly performed. Every *loader* reads two sequences with prefetching method mentioned in Sect. 3.1. Then *loaders* send the merged data to its *saver* through register communication. The roles and data stream are similar to the left part of Fig. 4, the difference is that in mergence, no *gld* is needed and all the data is got by DMA. For the fact that the key of all the data we will sort is non-negative integers, *loaders* send a message to its *saver* with the key field set to -1 as the flag of data transmitting ended after all the data is got and merged and sent to its *saver*. Each *saver* holds a double buffer, it saves the data into its buffer every time it receives data from its *loader*. When one part of its double buffer is full, it will write the data back to memory by a DMA request and use another part of the double buffer for data receiving. When *savers* receive the data with the value of the key field is -1 which means the data transmitting is ended, they will write all the remaining data in their double buffer back to memory and end working.

In the second step, the CPEs are divided into three types, *loaders*, *savers* and *workers*. A part of CPEs' hierarchical topology, division and data stream are shown in the right part of Fig. 4. *Workers* are set to be in different levels

Fig. 5. The illustration of sequence merging.

Fig. 6. The illustration of read-write conflict. (OD denotes the original data and T denotes the temporary memory)

according to the flow of the data stream, the *workers* that directly receive data from *loaders* are in the lowest level, the *worker* that directly sends data to the *saver* is in the highest level and the level of the others are sequentially increased according to the flow of the data stream. In Fig. 4, W_n means the level n worker. *Loaders* read data of two different sequences from memory and merge them and then send to the *workers* in the lowest level through register communication. The *workers* of every level receive data from two different lower-level *worker* and send the data to a higher-level *worker* after merge through register communication. There is only one highest level *worker*. It sends the merged data to the *saver* instead of other *workers*. The *saver* saves the result back to memory. Also we set key of communication to -1 as the end flag.

4 Implementation

In this section, we present the implementation details of *swGBDT*, especially focusing on the gain computation and sorting process which are the major

hotspots of GBDT. Moreover, we also describe the communication scheme for CPEs in detail.

4.1 Processing Logic of Gain Computation

As shown in Algorithm 2, to find the best gains, we need to consider every possible split and compute the gain of every split according to Eq. 1 to find the best gains. As the G_L and G_R in the equation are the sum of the g of instances on the left side and the right side, respectively. H_L and H_R are the same but the sum of h . The computation of g and h are shown in Eq. 2 and 3. Naturally, every feature value is a possible split point, but not all the instances have all the features. So, there are two ways to handle those instances that do not have the feature that the split uses. One is to divide the instances without this feature into the left side, that is assuming the feature values of these instances are smaller than the split point. Another is to divide them to the right side, that is assuming the feature values of these instances are bigger than the split point.

Algorithm 2. The Processing Logic of Gain Computation

```

1: Input:  $\mathbf{S}$ ,  $fatherGH$ ,  $missingGH$ ,  $prefixSumGH$ ,  $\lambda$ 
2: Output:  $Gain$ 
3: if  $thread\_id \% 2 == 0$  then
4:   for each  $split \in \mathbf{S}$  do
5:      $REG\_GET(reg)$ 
6:      $(f\_gh, m\_gh) \leftarrow reg$ 
7:      $r\_gh \leftarrow prefixSumGH[split]$ 
8:      $(l\_gain, r\_gain) \leftarrow ComputeTwoGain(f\_gh, r\_gh, m\_gh, \lambda)$ 
9:     if  $l\_gain > r\_gain$  then
10:       $Gain[split] \leftarrow l\_gain$ 
11:     else
12:       $Gain[split] \leftarrow -r\_gain$ 
13:     end if
14:   end for
15: else
16:   for each  $split \in \mathbf{S}$  do
17:      $reg \leftarrow (fatherGH[split], missingGH[split])$ 
18:      $REG\_SEND(reg, thread\_id - 1)$ 
19:   end for
20: end if

```

Through the prefix sum operation, we already know the sum of g/h of the instance with the feature the split uses for all the possible split. The sum of g/h of the missing instances has been calculated, too. Thus we can easily calculate the G_L and G_R of two kinds of division by simple addition. The sum of g/h of the father node which is the sum of g/h of all the instances is also known as the result of the previous iteration. Thus we can get the G_L and H_L with a simple

subtraction. So, they are all used as the input of the algorithm. In the algorithm, we obtain the index of the node that is to be split at first and get *father_gh* value with the index. Then we calculate the *gain* of missing instances on the left and right side, respectively. We only need the larger one and keep the original value if they are on the left or take the opposite if they are on the right. Since getting the *fatherGH* and *missingGH* are both two-step memory access, we cannot predict the memory access location of the second step because it depends on the result of the first step, so the data cannot be load into LDM by DMA easily. This means *gld* with high latency is needed. To reduce the performance loss, we divide the CPEs into *loaders* and *savers*. *Loaders* load the possible split into LDM using DMA and then get the *fatherGH* and *missingGH* with *gld*. Finally they send the data to their *saver* with register communication. *Savers* receive data, then compute the gains and write back to memory using DMA.

4.2 Processing Logic of Sorting

For sorting, we need to split the whole sequence to be sorted evenly into $A_0, A_1, A_2, \dots, A_{63}$, each element in the sequence consists of two parts: key and value. It's a key based radix sort. As the key is a 32-bit integer, the time complexity is $O(\lceil 32/r \rceil \times n)$, where r is the number of the bits of the base, that is, r bits are used for calculation at each round. It can be seen that the larger the r is, the lower the time complexity is. However, the rounding up operation leads to the result that when r is set to r_0 and is not the factor of 32, the time complexity is the same with r using a factor of 32 that is the closest to r_0 but smaller than r_0 . The factors of 32 are 1, 2, 4, 8, 16, 32 and the capacity of LDM is 64 KB which can only accommodate up to 16386 32-bit integers. When r takes 16, $2^{16} = 65536$ buckets are needed, the LDM will be exceeded even if the capacity of each bucket is 1. That is to say, the r can only take 8, so four rounds are needed to finish sorting. Because every CPE sorts independently, only 64 internally ordered sequences, $B_0, B_1, \dots, B_{63}$, are obtained after sorting. 64 sequences are unordered with each other. We need the merging operation to get the final result.

For merging, we use the loader-saver mode to divide the CPEs in the first step. For stable sorting, as shown in Algorithm 3, the i^{th} *loader* reads the data from B_{2i} and B_{2i+1} and merges them. We can consider the two sequences as two queues, the queue with data from B_{2i} calls q_0 , the queue with data from B_{2i+1} calls q_1 . Reading a *tile* means the data in the *tile* enqueue the corresponding queue. Comparing the key of the elements of the two queues continually, only when the key of the head element of q_1 is smaller than that of q_0 or q_0 is empty with no more data to be enqueued, q_1 can dequeue the head element, otherwise q_0 dequeue the head element. The dequeued element is sent to the corresponding *saver* by register communication. *Saver* saves the received data into buffer and writes the data into the main memory every the buffer is filled.

Algorithm 3. The Processing Logic of the Loaders of Mergence

```

1: Input: sequences, sendto, seg_id
2: readbuffer1  $\leftarrow$  sequences[seg_id[0]].tiles
3: readbuffer2  $\leftarrow$  sequences[seg_id[1]].tiles
   /*refill from memory when readbuff is empty, the refill operation is skipped here*/
4: while Not (readbuffer1.empty And readbuffer2.empty) do
5:   send = min(readbuffer1, readbuffer2)
   /*get the min item from two buffers and remove it*/
   /*if any buffer is empty then directly get from one the other*/
6:   REG_SEND(send, sendto)
7: end while
8: REG_SEND(END, sendto)

```

In the second step, we use the loader-worker-saver mode to divide the CPEs. Because the receiver of register communication cannot know the sender, a send-flag in the message that indicates the sender is needed if there are more than one sender in the same row or the same column with receiver. But the message length is only 128 bits, the sum length of key and value is 64 bits. If sender-flag is added, we can only send a pair of data a time which lower the efficiency. Thus, we propose a method that each CPE receives data from only one same-row CPE and one same-column CPE. And for stable sort, we ensure that the data received from the same-column CPE is in the former sequence than which from the same-row CPE by a carefully designed communication method. More specifically, the i^{th} worker in a level receives last-level's $(2i)^{th}$ intermediate result by register communication from the same-column CPE and the $(2i + 1)^{th}$ from the same-row CPE and sends the merged data (the i^{th} intermediate result of this level) to the $\lfloor i/2 \rfloor^{th}$ CPE of the next level as shown in Algorithm 4. According to the

Algorithm 4. The Processing Logic of the Workers for Mergence

```

1: Input: sendto
2: No Input and Output
3: REG_GETC(col_recv)
4: REG_GETR(row_recv)
5: while row_recv! = END And col_recv! = END do
6:   (send, whichmin) = Min(col_recv, row_recv)
   /*readcache contains two queues, get the smaller item from the front*/
   /*if one queue is empty then get the head item from the other one*/
7:   REG_SEND(send, sendto)
8:   if whichmin == 0 then
9:     REG_GETC(col_recv)
10:  else
11:    REG_GETR(row_recv)
12:  end if
13: end while
14: REG_SEND(END, sendto)

```

design, when $i \bmod 2 = 0$, the data is sent to the same-column CPE, otherwise to the same-row CPE. Meanwhile, the read buffer of register communication is a queue with clear-after-reading, we do not need queues for merging. *Loaders* and *savers* work similar to the first step.

4.3 Synchronization Among CPEs

Since the CPE senders and receivers perform register communications according to the index of the array to be written or to be read and all the communications are one-to-one communications, no explicit synchronization mechanism is required. In other words, the whole 128 bits of the message are usable data. Therefore, our method can make full use of the bandwidth of register communication and thus improve communication performance.

5 Evaluation

5.1 Experiment Setup

Datasets. To evaluate the performance of our *swGBDT*, we use 6 datasets from LIBSVM Data [2] and 4 synthesized datasets named dataset1–4. The details of the datasets are shown in Table 1.

Table 1. The datasets for evaluation.

DataSet	Instances	Features	NNZ
dataset1	779,412	17,293	1,339,361
real-sim	72,309	20,958	3,709,083
news20	19,996	155,191	9,097,916
dataset2	7,245,157	179,232	12,445,475
dataset3	9,206,499	54,357	15,821,051
e2006	16,087	150,360	19,971,015
YearPredictionMSD	463,715	90	41,734,350
rcv1.test	677,399	47,236	49,556,258
dataset4	31,354,632	447,882	53,868,233
SUSY	5,000,000	18	88,938,127

Evaluation Criteria. We conduct our experiments on a CG of Sunway SW26010 processor. We compare the performance of our *swGBDT* with serial implementation on MPE and parallel XGBoost [3] on CPEs. The serial implementation is the naive implementation of our GBDT algorithm without using CPEs. We port the popular open source implementation¹ of XGBoost for parallel

¹ <https://github.com/dmlc/xgboost>.

execution on CPEs (with LDM used for better performance). In our experiments, we set the parameter depth to 6 and the number of trees to 40. All experiments run in single precision.

5.2 Performance Analysis

We use the average training time of a tree for comparison and use the MPE version as baseline. The results are shown in Fig. 7, Fig. 8 and Table 2, we can see clearly that *swGBDT* is the best one on all datasets. Compared to the MPE version, *swGBDT* can reach an average speedup of $4.6\times$ and $6.07\times$ for maximum on SUSY. Meanwhile, compared to XGBoost, we can achieve $2\times$ speedup for average, $2.7\times$ speedup for maximum. The advantage of *swGBDT* comes from the CPEs division that reduces the memory access time.

Fig. 7. The performance of *swGBDT* and XGBoost on real-world datasets

Fig. 8. The performance of *swGBDT* and XGBoost on synthesized datasets

Table 2. The training results for *swGBDT* and XGBoost.

DataSet	Training RMSE	
Name	<i>swGBDT</i>	XGBoost
dataset1	576	577
real-sim	0.47	0.5
news20	0.48	0.52
dataset2	575	577
dataset3	575	577
e2006	0.23	0.24
YearPredictionMSD	8.8	8.9
rcv1.test	0.43	0.43
dataset4	564	577
SUSY	0.37	0.37

5.3 Roofline Model

In order to analyse the efficiency of our implementation, we apply the roofline model [22] to *swGBDT* on a CG of Sunway processor. Giving a dataset with m instances and n features, assuming the non-zero numbers is nnz , we store it in CSC format. N_split is the number of possible splits during every training round. Let Q , W and I represent the amount of data accessed from memory, the number of floating point operations and the arithmetic intensity [23] respectively. The calculation of Q , W , I is shown in Eq. 5, 4, 6 respectively.

$$W = 19.5 * nnz + 37 * n_split \quad (4)$$

$$Q = 22 * nnz + 32.5 * n_split \quad (5)$$

$$I = \frac{W}{Q * 8bytes} = 0.125 + \frac{1.8 * n_split - nnz}{70.4 * nnz + 104 * n_split} \quad (6)$$

In our experiments, in most of the dataset, the n_split is about 0.9 of nnz . In this situation, $I = 0.1288$, the ridge point of Sunway processor is 8.46, we can see that the bottleneck of GBDT is memory access. The version without memory access optimization (the MPE version) gets the $I = 0.108$. Our optimization increases the arithmetic intensity for about 20%.

5.4 Scalability

To achieve better scalability, we divide the features into n segments evenly when the number of CGs is n . The i^{th} CG only stores and processes the i^{th} feature segment. Each CG computes its 2^{depth} splits and then determines the 2^{depth} best splits for all, where $depth$ is the depth of the tree currently. As shown in Fig. 9, we use up to 4 CGs on a processor for evaluating the scalability of *swGBDT*. Comparing to one CG, we can reach an average of 8×, 11.5× and 13× speedup when scaling to 2, 3 and 4 CGs, respectively.

6 Related Work

6.1 Acceleration for Gradient Boosted Decision Tree

To improve the performance of the GBDT algorithm, on one hand, some of the recent researches have been making efforts to modify the GBDT algorithm for acceleration. LightGBM [9] accelerate the time-consuming gain estimation process by eliminating instances with small gradients and wrapping commonly exclusive features, which can reduce computation. Later, Biau *et al.* [1] optimize the GBDT through combining Nesterov’s accelerated descent [15] for parameter update. On the other hand, researches have been trying to transfer the GBDT to novel accelerators like GPU. Mitchell and Frank [14] implement the tree construction within GBDT algorithm in XGBoost [3] to GPU entirely to reach higher performance. Besides, Wen *et al.* [20] develop *GPU-GBDT* which enhance the performance of GBDT through dynamic allocation, data reuse and Run-length Encoding compression. The *GPU-GBDT* is further optimized to ThunderGBM [21] on multiple GPUs which incorporates new techniques like efficient search for attribute ID and approximate split points. However, those implementations do not target at Sunway architecture and there have not been any efficient GBDT algorithm designed to leverage the unique architecture features on Sunway to achieve better performance.

6.2 Machine Learning on Sunway Architecture

There have been many machine learning applications designed for Sunway architecture since its appearance. Most of the previous researches focus on optimizing neural networks on Sunway. Fang *et al.* [5] implement convolutional neural networks (CNNs) on SW26010 which is named swDNN through systematic optimization on loop organization, blocking mechanism, communication and instruction pipelines. Later, Li *et al.* [10] introduce swCaffe which is based on the popular CNN framework Caffe and develop topology-aware optimization for synchronization and I/O. Liu *et al.* [13] propose an end-to-end deep learning compiler on Sunway that supports ahead-of-time code generation and optimizes the tensor computation automatically.

Moreover, researchers have paid attention to optimize the numerical algorithms which are kernels in machine learning applications on Sunway architecture. Liu *et al.* [12] adopt multi-role assignment scheme on CPEs, hierarchical partitioning strategy on matrices as well as CPE cooperation scheme through register communication to optimize the Sparse Matrix-Vector Multiplication (SpMV) algorithm. The multi-role assignment and CPE communication schemes are also utilized by Li *et al.* [11] who develop an efficient Sparse triangular solver (SpTRSV) for Sunway. What’s more, Wang *et al.* [19] improve the performance of SpTRSV on Sunway architecture through Producer-Consumer pairing strategy and novel Sparse Level Tile layout. Those researches provide us the inspiration of accelerating GBDT algorithm for Sunway architecture.

Fig. 9. The scalability of *swGBDT*.

7 Conclusion and Future Work

In this paper, we present an efficient GBDT implementation *swGBDT* on Sunway processor. We propose a partitioning method that partitions CPEs into multiple roles and partitions input data into different granularities such as *blocks* and *tiles* for achieving better parallelism on Sunway. The above partitioning scheme can also mitigate the high latency of random memory access through data prefetching on CPEs by utilizing DMA and register communication. The experiment results on both synthesized and real-world datasets demonstrate *swGBDT* achieves better performance compared to the serial implementation on MPE and parallel XGBoost on CPEs, with the average speedup of 4.6× and 2× respectively. In the future, we would like to extend *swGBDT* to run on CGs across multiple Sunway nodes in order to support the computation demand of GBDT at even larger scales.

Acknowledgments. This work is supported by National Key R&D Program of China (Grant No. 2016YFB1000304), National Natural Science Foundation of China (Grant No. 61502019, 91746119, 61672312), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (Grant No. 2019A12) and Center for High Performance Computing and System Simulation, Pilot National Laboratory for Marine Science and Technology (Qingdao).

References

1. Biau, G., Cadre, B., Rouvière, L.: Accelerated gradient boosting. *Mach. Learn.* **108**(6), 971–992 (2019). <https://doi.org/10.1007/s10994-019-05787-1>
2. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. *ACM Trans. Intell. Syst. Technol.* **2**, 27:1–27:27 (2011). Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>
3. Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794. ACM (2016)
4. Dongarra, J.: Sunway TaihuLight supercomputer makes its appearance. *Nat. Sci. Rev.* **3**(3), 265–266 (2016)
5. Fang, J., Fu, H., Zhao, W., Chen, B., Zheng, W., Yang, G.: swDNN: a library for accelerating deep learning applications on sunway TaihuLight. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 615–624. IEEE (2017)
6. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. *Ann. Stat.* **29**, 1189–1232 (2001)
7. Fu, H., et al.: The sunway TaihuLight supercomputer: system and applications. *Sci. China Inf. Sci.* **59**(7), 072001 (2016)
8. Hu, J., Min, J.: Automated detection of driver fatigue based on eeg signals using gradient boosting decision tree model. *Cogn. Neurodyn.* **12**(4), 431–440 (2018)
9. Ke, G., et al.: LightGBM: a highly efficient gradient boosting decision tree. In: *Advances in Neural Information Processing Systems*, pp. 3146–3154 (2017)
10. Li, L., et al.: swCaffe: a parallel framework for accelerating deep learning applications on sunway TaihuLight. In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 413–422. IEEE (2018)

11. Li, M., Liu, Y., Yang, H., Luan, Z., Qian, D.: Multi-role SpTRSV on sunway many-core architecture. In: 2018 IEEE 20th International Conference on High Performance Computing and Communications, IEEE 16th International Conference on Smart City, IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 594–601. IEEE (2018)
12. Liu, C., Xie, B., Liu, X., Xue, W., Yang, H., Liu, X.: Towards efficient SpMV on sunway manycore architectures. In: Proceedings of the 2018 International Conference on Supercomputing, pp. 363–373. ACM (2018)
13. Liu, C., Yang, H., Sun, R., Luan, Z., Qian, D.: swTVM: exploring the automated compilation for deep learning on sunway architecture. arXiv preprint [arXiv:1904.07404](https://arxiv.org/abs/1904.07404) (2019)
14. Mitchell, R., Frank, E.: Accelerating the XGBoost algorithm using GPU computing. *PeerJ Comput. Sci.* **3**, e127 (2017)
15. Nesterov, Y.: A method of solving a convex programming problem with convergence rate $o(\frac{1}{k^2})$. *Soviet Math. Dokl.* **27**, 372–376 (1983)
16. Nowozin, S., Rother, C., Bagon, S., Sharp, T., Yao, B., Kohli, P.: Decision tree fields: an efficient non-parametric random field model for image labeling. In: Criminisi, A., Shotton, J. (eds.) *Decision Forests for Computer Vision and Medical Image Analysis*. ACVPR, pp. 295–309. Springer, London (2013). https://doi.org/10.1007/978-1-4471-4929-3_20
17. Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A.V., Gulin, A.: CatBoost: unbiased boosting with categorical features. In: *Advances in Neural Information Processing Systems*, pp. 6638–6648 (2018)
18. Roe, B.P., Yang, H.J., Zhu, J., Liu, Y., Stancu, I., McGregor, G.: Boosted decision trees as an alternative to artificial neural networks for particle identification. *Nucl. Instrum. Methods Phys. Res. Sect. A: Accel. Spectrom. Detectors Assoc. Equip.* **543**(2–3), 577–584 (2005)
19. Wang, X., Liu, W., Xue, W., Wu, L.: swSpTRSV: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In: *ACM SIGPLAN Notices*, vol. 53, pp. 338–353. ACM (2018)
20. Wen, Z., He, B., Kotagiri, R., Lu, S., Shi, J.: Efficient gradient boosted decision tree training on GPUs. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 234–243. IEEE (2018)
21. Wen, Z., Shi, J., He, B., Chen, J., Ramamohanarao, K., Li, Q.: Exploiting GPUs for efficient gradient boosting decision tree training. *IEEE Trans. Parallel Distrib. Syst.* **30**, 2706–2717 (2019)
22. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **52**(4), 65–76 (2009)
23. Xu, Z., Lin, J., Matsuoka, S.: Benchmarking sw26010 many-core processor. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 743–752. IEEE (2017)
24. Xuan, P., Sun, C., Zhang, T., Ye, Y., Shen, T., Dong, Y.: Gradient boosting decision tree-based method for predicting interactions between target genes and drugs. *Front. Genet.* **10**, 459 (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.