Neural Computing and Applications
https://doi.org/10.1007/s00521-019-04481-6

An experimental evaluation of extreme learning machines on several hardware devices

Liang Li[1] · Guoren Wang[2] · Gang Wu[1,3] · Qi Zhang[2]

Received: 31 December 2018 / Accepted: 29 August 2019

Abstract
As an important learning algorithm, extreme learning machine (ELM) is known for its excellent learning speed. With the expansion of ELM's applications in the field of classification and regression, the need for its real-time performance is increasing. Although the use of hardware acceleration is an obvious solution, how to select the appropriate acceleration hardware for ELM-based applications is a topic worthy of further discussion. For this purpose, we designed and evaluated the optimized ELM algorithms on three kinds of state-of-the-art acceleration hardware, i.e., multi-core CPU, Graphics Processing Unit (GPU), and Field-Programmable Gate Array (FPGA) which are all suitable for matrix multiplication optimization. The experimental results showed that the speedup ratio of these optimized algorithms on acceleration hardware achieved 10–800. Therefore, we suggest that (1) use GPU to accelerate ELM algorithms for large dataset, and (2) use FPGA for small dataset because of its lower power, especially for some embedded applications. We also opened our source code.

Keywords Extreme learning machine · Hardware · Multi-core · GPU · FPGA

1 Introduction

As machine learning technologies (e.g., Support Vector Machine [35], Neural Networks [6, 26], and Random Forest [21]), especially deep learning [2, 3, 27], are applied in a continuously wider range of scenarios, people are paying more and more attention to the performance of these algorithms. However, the learning speed of traditional machine learning algorithms is widely criticized [14]. The main reason for the slow speed is that parameters of machine learning algorithms usually need to be updated iteratively in a gradient method. Hence, traditional machine learning methods are difficult to meet the real-time learning needs for large-scale data applications.

Extreme learning machine (ELM) [9–12, 14–16] is a feedforward neural network whose design objective is to ensure a high accuracy, least user intervention, and real-time learning [14]. In practical applications, ELM is usually superior to the traditional machine learning algorithms, such as SVM and back propagation (BP) in both classification accuracy and learning speed when fed with adequate training samples [11]. Therefore, ELM has found application scenarios in medical healthcare [23, 28, 29, 31–33], query processing [4, 19], location-based social networks (LBSNs) [22, 42], Geographic Information System (GIS) [18], etc.

As a matter of fact, ELM is also hard to escape the mantra of performance scalability. As the dataset scale increases, the efficiency of learning gradually declines. Our preliminary investigation shows that for scales like classical UCI Forest CoverType dataset,[1] time cost on each step of the standard ELM single-core CPU implementation for

✉ Guoren Wang
wanggr@bit.edu.cn

Liang Li
liliang@stumail.neu.edu.cn

Gang Wu
wugang@mail.neu.edu.cn

Qi Zhang
2120161078@bit.edu.cn

[1] Computer Science and Engineering, Northeastern University, Shenyang, China

[2] Computer Science and Technology, Beijing Institute of Technology, Beijing, China

[3] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

[1] http://archive.ics.uci.edu/ml/datasets/Covertype.

Published online: 12 September 2019

supervised classification tasks becomes so significant that they cannot be ignored. Thus, ELM needs an efficient acceleration solution to adapt to AI applications for big data. Although the use of hardware acceleration is an obvious solution, there is no research on the applicability of various hardware acceleration platforms. To sum up, the motivation and contribution of this work are as follows.

1.1 Motivation

1. *Can ELM keep running well in large-scale application scenario*? As we know, real-time learning is one of the essential considerations of ELM. The learning speed under large-scale datasets becomes a severe test for the practicality of ELM.
2. *Is the ELM training phase suitable for hardware acceleration*? What is the key factor to affect the scalability of ELM learning speed? And how to rewrite the ELM algorithm based on this key factor to adapt it to various hardware acceleration platforms?
3. *If ELM training can be accelerated, then how to select appropriate hardware acceleration platforms for different scales of datasets*? There are several kinds of acceleration hardware available including multi-core CPU, GPU, FPGA, etc. The answer to the question will be instructive to the application of ELM in large-scale AI scenarios.

1.2 Contributions

In this work, we made the following contributions.

1. *We validated that the standard implementation of ELM has poor learning speed scalability on large-scale dataset.* We fully implemented the single-core CPU version of ELM algorithm as a baseline and evaluated the training time cost of each step in detail on a classical large-scale dataset as shown in Table 1. The evaluation shows that the training time cost of ELM and the size of dataset are linearly related.
2. *The study reveals that ELM is a hardware-friendly algorithm*. The algorithm and the evaluation of the baseline implementation both show that ELM training process is dominated by matrix multiplication operations which, as is known, are easily to be paralleled. For CoverType dataset, matrix multiplication accounts for roughly 97% of the training time (see Sect. 2). Hence, we were motivated to carry out substantial research on adapting the standard ELM algorithm to several types of acceleration hardware that support parallelization, such as Field-Programmable Gate Arrays (FPGAs) [5, 25, 40] and Graphic Processing Units (GPUs) [1, 17, 36], and specialized multi-core CPU with architecture optimized for multiple computational operations (see Sect. 3).
3. *GPU and FPGA are suitable devices for ELM.* We suggest that (1) use GPU to accelerate ELM algorithms for large dataset, and (2) use FPGA for small dataset considering its lower power consumption, especially for those embedded systems. To support these conclusions, the source code of the proposed hardware acceleration platforms related with ELM algorithms was implemented and opened. Experiments were performed on three datasets, namely Covertype, Ionosphere, and Sonar. Experimental results further confirmed our opinion that ELM training algorithm is suitable for hardware acceleration. Specifically, in our experiments, multi-core CPU with SIMD optimization can get about 10× speedup. GPU's speedup is larger than 800×, and FPGA's speedup is about 40×.

The rest of this paper is organized as follows: in Sect. 2, the basic ELM algorithm is discussed and its performance is evaluated to show the bottleneck. In Sect. 3, the optimization methods of ELM algorithm on different hardware devices are elaborated. In Sect. 4, the performance of those different ELM implementations is evaluated and compared. Section 5 elaborates related work, and the conclusion is shown in Sect. 6.

2 Preliminary

2.1 Extreme learning machine

Given N arbitrary distinct samples $(\mathbf{x}_i, \mathbf{t}_i) \in \mathbf{R}^{n\times m}$, where $\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{in}]^T \in \mathbf{R}^n$ and $\mathbf{t}_i = [t_{i1}, t_{i2}, \ldots, t_{im}]^T \in \mathbf{R}^m$. $\mathbf{x}_i$ is the sample vector; $\mathbf{t}_i$ is the target label. Standard **S**ingle-hidden **L**ayer **F**eedforward neural **N**etworks (SLFNs) are shown in Fig. 1 and can be modeled mathematically as Eq. 1:

$$\sum_{j=1}^{L} \beta_j g_j(\mathbf{x}_i) = \sum_{j=1}^{L} \beta_j g(\mathbf{w}_j \cdot \mathbf{x}_i + b_j) = \mathbf{o}_i, i = 1, \ldots, N \tag{1}$$

where L is the number of hidden layer nodes, $\mathbf{w}_j = [w_{j1}, w_{j2}, \ldots, w_{jn}]$ is the input weight vector, $\beta_j = [\beta_{j1}, \beta_{j2}, \ldots, \beta_{jm}]^T$ is the output weight vector, b_j is the bias of the j_{th} hidden node, and $\mathbf{o}_j$ is the output of the j_{th} node. Generally,

$$g(x) = \frac{1}{1 + e^x} \tag{2}$$

To approximate these samples with zero errors means that $\sum_{i=1}^{L} \|\mathbf{t}_i - \mathbf{o}_i\| = 0$, where β_j, $\mathbf{w}_j$ and b_j exist, satisfying that

Table 1 Baseline evaluation (single-core implementation): which gives the execution time distribution with different N and L.

N	L	rand(w) (s)	rand(bias) (s)	H (s)	$g(H)$ (s)	A (s)	b (s)	Solving (s)
250,000	200	0.000679	1.22e−05	47.4303	10.4087	146.5	6.7185	0.032932
300,000	200	0.000679	1.22e−05	56.7862	12.4688	176.063	8.27493	0.033314
350,000	200	0.000683	1.25e−05	66.4569	14.6103	204.562	9.42965	0.033097
400,000	200	0.000685	1.22e−05	75.6422	16.6199	234.503	10.7515	0.032972
450,000	200	0.000685	1.22e−05	85.3575	18.8	263.049	12.116	0.033029
500,000	100	0.000355	9.03e−06	47.6114	10.4667	73.9813	6.72915	0.00656
500,000	200	0.000721	1.31e−05	95.163	20.9783	296.22	13.5306	0.032887
500,000	300	0.001025	2.06e−05	142.003	32.5244	658.06	20.1207	0.093655
500,000	400	0.001337	2.77e−05	188.898	43.5225	1168.49	26.7123	0.205492
500,000	500	0.001682	3.16e−05	**236.566**	52.165	**1815.57**	33.8406	0.377687

The columns 3 to 9 record the running time of each step in Algorithm 1. Step 3 (column 5) and step 5 (column 7) in Algorithm 1 consume a large amount of execution time. Under the condition that $N = 500,000$ and $L = 500$, all the matrix multiplication operations take up about 97.5% of the total time cost in training algorithm

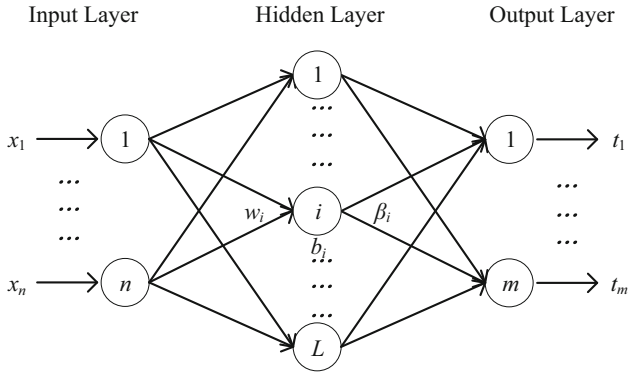


Fig. 1 An example of Standard **S**ingle-hidden **L**ayer **F**eedforward neural **N**etworks (SLFNs)

$$\sum_{j=1}^{L} \beta_j g(\mathbf{w}_j \mathbf{x}_i + b_j) = \mathbf{t}_i, i = 1, \ldots, N \tag{3}$$

which can be rewritten in terms of

$$\mathbf{H}\beta = \mathbf{T} \tag{4}$$

where

$$H(\mathbf{w}_1, \ldots, \mathbf{w}_L, \mathbf{b}_1, \ldots, \mathbf{b}_L, \mathbf{x}_1, \ldots, \mathbf{x}_N) = \begin{bmatrix} g(\mathbf{w}_1\mathbf{x}_1 + b_1) & \ldots & g(\mathbf{w}_L\mathbf{x}_1 + b_L) \\ \vdots & \ddots & \vdots \\ g(\mathbf{w}_1\mathbf{x}_N + b_1) & \ldots & g(\mathbf{w}_L\mathbf{x}_N + b_L) \end{bmatrix}_{N\times L} \tag{5}$$

$$\beta = \begin{bmatrix} \beta_1^T \\ \beta_2^T \\ \vdots \\ \beta_L^T \end{bmatrix}_{m\times L}^T, \text{ and } \quad \boldsymbol{T} = \begin{bmatrix} \boldsymbol{t}_1^T \\ \boldsymbol{t}_2^T \\ \vdots \\ \boldsymbol{t}_N^T \end{bmatrix}_{m\times N}^T \tag{6}$$

$\mathbf{H}$ is the ELM feature space to map the N-dimensional input data space into L-dimensional hidden nodes space. Unlike the most common algorithms that all parameters need to be iteratively adjusted to, once the input weight $\mathbf{w}_i$ and hidden layer bias b_i are randomly determined, the output matrix of hidden layer nodes will be ensured. Training SLFNs can be transformed into solving a linear system.

In most cases, H is a non-square matrix ($L \ll N$); the output weights β are computed by $\beta = H^{\dagger}T$, where $H^{\dagger}$ is the Moore–Penrose generalized inverse of matrix H. Equation 7 gives the steps about how to compute $H^{\dagger}$.

$$\begin{aligned} &\boldsymbol{H}\beta = \boldsymbol{T} \\ &\boldsymbol{H}^T\boldsymbol{H}\beta = \boldsymbol{H}^T\boldsymbol{T} \\ &\beta = (\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{T} \end{aligned} \tag{7}$$

The training phase of ELM algorithm is described in Algorithm 1, which includes 7 steps in all. Algorithm 2 is able to classify unknown data D based on trained W, B, and β. The matrix H_2 is computed in the same way as in the training phase. The output o represents predicted values.

Algorithm 1 ELM training phase

Input: Training data: X
Input: Target data: T
1: $W = rand()$
2: $B = rand()$
3: $\mathbf{H} = X * W + B$
4: $H = 1/(1 + e^{-H})$
5: $A = H^T H$
6: $b = H^T T$
7: $\boldsymbol{\beta} = A^{-1} b$

Algorithm 2 ELM predicting phase

Input: input predict data D

1: $H_2 = D * W + B$
2: $H_2 = 1/(1 + e^{H_2})$
3: $o = \boldsymbol{H_2\beta}$

2.2 Bottleneck analysis

In general, large datasets are provided in training phase in order to compute the parameters β of ELM, namely the weight vectors connecting the hidden nodes and the output nodes. Although the learning speed of ELM is much faster than most classic learning algorithms, it is still slow due to the large amount of the training dataset.

To further clarify the time cost of ELM algorithm, we focus on the training phase of ELM and perform a detailed measure of the execution time for each step in Algorithm 1. The algorithm is implemented on single-core CPU with Intel(R) Xeon(R) CPU E7-4820 v4 @ 2.00 GHz in Ubuntu 16.04 LTS. The dataset used in our experiment is Covertype,[2] which includes 581,012 samples, and each sample contains 54 data attributes. All samples are divided into 7 classes. That is to say, the number of input nodes of SLNFs in our ELM algorithm is 54, and the number of output nodes is 7. We change the number of nodes in the hidden layer L or the number of training samples N to measure the time cost.

Table 1 illustrates the execution time distribution with different N and L. Columns 3 to 9 record the running time of each step in Algorithm 1. Note that all of the time cost we measured has been tested 10 times, and it is the mean value that is recorded in the result. We can get the following 2 findings.

1. For each instance, step 3 and step 5 in Algorithm 1 consume a significant amount of execution time, and they are both corresponding to matrix multiplication operations. Under the condition that N or L is large, all the matrix multiplication operations will take up about 97.5% of the total time cost in training algorithm, which is the critical bottleneck of ELM algorithm.
2. The time cost of matrix multiplication is linearly related to N and L. With the increase of N or L, the execution time will be longer.

To sum up, matrix multiplication occupies a lot of execution time in ELM training phase, especially when the training dataset is large. In fact, the main cost of ELM predicting phase also incurs in the matrix multiplication on the verification dataset described in Algorithm 2. Hence, the efficiency of ELM algorithm is seriously affected by that of the matrix multiplication's implementation. How to efficiently calculate large-scale matrix multiplication is the key point of ELM acceleration. As we all know, matrix multiplication is very suitable to be implemented on several hardware devices; we will discuss the details in next section.

[2] http://archive.ics.uci.edu/ml/datasets/Covertype.

3 Implementations on hardware devices

The above baseline experiment shows that the critical point of ELM acceleration is how to calculate large-scale matrix multiplication quickly. With the rapid improvement in modern hardware's performance, parallel execution becomes prevalent, which can greatly reduce the execution time. Therefore, employing new hardware devices to accelerate ELM algorithm is a feasible solution.

In this section, we will discuss the performance of ELM algorithms in three modern hardware devices (multi-core/SIMD CPU, GPU, FPGA) for accelerating matrix multiplication operation which is the key bottleneck of ELM. The advantages and disadvantages of each computing device are shown in Table 2.

First of all, we will demonstrate the framework of what we did in this section. As we can see in Algorithm 1, the training phase of ELM algorithms has 7 steps in all. Figure 2 shows the framework of the training phase, in which we implement step 3 with multi-core (green box) and offload step 5 with several devices (yellow box), in short, offloading those time-consuming and hardware-friendly tasks into the new hardware devices.

3.1 CPU implementation

3.1.1 Baseline

Formally, given an $m \times k$ matrix A and an $k \times n$ matrix B, the matrix multiplication $C = A \times B$ is an $m \times n$ matrix; the element C_{ij} can be calculated by

$$C_{ij} = \sum_{l=1}^{k} A_{il}B_{lj} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{ik}B_{kj} \tag{8}$$

A straightforward implementation of the matrix multiplication which is based on a single core is shown in Algorithm 3. Obviously, the computing complexity is $O(m \times k \times n)$. We regard it as a baseline for other modern hardware implementations.

Table 2 Advantages and disadvantages of each computing device

Device	Advantage	Disadvantage
CPU	Programming-friendly	High power consumption
GPU	Almost real time in large data	High data transform cost
FPGA	Low power consumption	Difficult programming

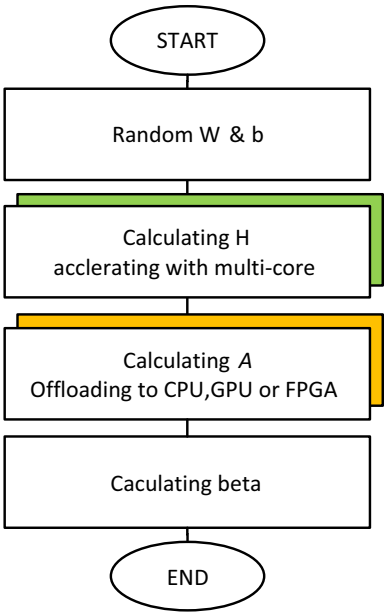


Fig. 2 Architecture of the ELM training phase. Calculating H by multi-core and calculating A with several devices, including CPU, GPU, and FPGA

Algorithm 3 Matrix multiplication with single core

```
for i = 0 to m do
    for j = 0 to n do
        C[i][j] = 0
        for l = 0 to k do
            C[i][j] = C[i][j] + A[i][l]*B[l][j]
        end for
    end for
end for
```

Definition 1 (*Speedup Ratio*) Suppose the running time of matrix multiplication on the baseline version is $T_{baseline}$, the time on device is T_{device}, and the speedup ratio is

$$speedup = \frac{T_{baseline}}{T_{device}}$$

3.1.2 Multi-core, thread-level parallelism

A multi-core processor, which consists of a single computing component with two or more independent processing units, called **cores**, can run multiple instructions on separate cores. Therefore, it can increase overall speed for programs amenable to parallel computing.

$$A_{m\times k} \times B_{k\times n} = \begin{bmatrix} A_{11}\ldots A_{1k} \\ \vdots \cdot \vdots \\ A_{a1}\ldots Aak \end{bmatrix} \times \begin{bmatrix} B_{11}\ldots B_{1b} \\ \vdots \cdot \vdots \\ B_{k1}\ldots Bkb \end{bmatrix} = \begin{bmatrix} C_{11}\ldots C_{1b} \\ \vdots \cdot \vdots \\ C_{a1}\ldots Cab \end{bmatrix} = C_{m\times n} \tag{9}$$

In order to accelerate ELM algorithm, we take full advantage of multi-core to accelerate its matrix multiplication which is the most time-consuming operation. As shown in Eq. 9, matrix C is divided into $a \times b$ subblocks, each of which is a $(m/a \times n/b)$ matrix and C will be derived by the calculation of these subblocks. Due to the independence of the calculation process between these submatrices, each one can be obtained by a single core. By computing all of the $a \times b$ in parallel, ELM algorithm can be greatly accelerated. Calculation algorithm for each submatrix is shown in Algorithm 4, where *id* represents the ID of threads, $id \in [0, a \times b)$. Obviously, the computing complexity of multi-core implementation algorithm for matrix multiplication is $O(\frac{m*n*k}{a\times b})$.

Algorithm 4 Matrix multiplication with multi core

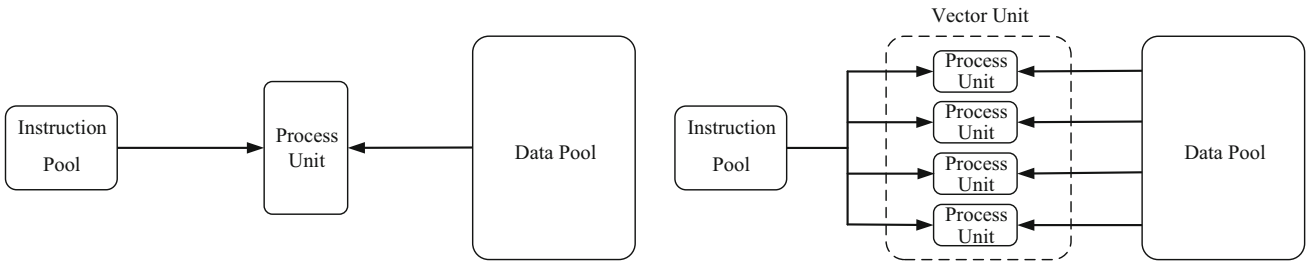
```
id = get_thread_id()
row = id/b
column = id%b
for i = m/a × row to m/a × (row + 1) do
    for j = n/b × column to n/b × (column + 1) do
        C[i][j] = 0
        for l = 0 to k do
            C[i][j] = C[i][j] + A[i][l]*B[l][j]
        end for
    end for
end for
```

Algorithm 5 Matrix multiplication with SIMD

```
for i = 0 to m do
    for j = 0 to n do
        l = 0
        C[i][j] = 0
        while l < k do
            regesiter r1 = load(A[i][l]), r9 = load(B[l][j])
            regesiter r2 = load(A[i][l+1]), r10 = load(B[l+1][j])
            regesiter r3 = load(A[i][l+2]), r11 = load(B[l+2][j])
            regesiter r4 = load(A[i][l+3]), r12 = load(B[l+3][j])
            regesiter r5 = load(A[i][i+4]), r13 = load(B[l+4][j])
            regesiter r6 = load(A[i][l+5]), r14 = load(B[l+5][j])
            regesiter r7 = load(A[i][l+6]), r15 = load(B[l+6][j])
            regesiter r8 = load(A[i][l+7]), r16 = load(B[l+7][j])
            mulps256(r1,r9), mulps256(r2,r10)
            mulps256(r3,r11), mulps256(r4,r12)
            mulps256(r5,r13), mulps256(r6,r14)
            mulps256(r7,r15), mulps256(r8,r16)
            addps256(r1,r2), addps256(r3,r4)
            addps256(r5,r6), addps256(r7,r8)
            addps256(r1,r3), addps256(r5,r7)
            addps256(r1,r5)
            C[i][j] += r1
            l = l + 8;
        end while
    end for
end for
```

3.1.3 SIMD, instruction-level parallelism

Single Instruction Multiple Data (SIMD) is an instruction unit that controls multiple duplicated processing units simultaneously to perform the same instructions on



(a) Single Instruction Single Data (SISD) **(b)** Single Instruction Multiple Data (SIMD)

Fig. 3 Comparison of SISD and SIMD. Single Instruction Multiple Data (SIMD) is an instruction unit that controls multiple duplicated processing units simultaneously to perform the same operations on multiple data

multiple data. Figure 3 illustrates the data processing of Single Instruction Single Data (SISD) and SIMD and reveals the excellent data parallelism of SIMD. At present, the prevalent SIMD instruction sets include MultiMedia eXtensions (MMX), Streaming SIMD Extensions (SSE), Intel Advanced Vector eXtensions (AVX), Fused Multiply Accumulate (FMA), and so on. SIMD is particularly suitable for processing continuous dense data. Most modern CPUs include SIMD instructions which can improve the performance of the games and multimedia's application. Hence, the excellent parallelism of SIMD can also be applied to accelerate ELM algorithm on matrix multiplication operation which involves mutually independent multiplication and addition.

GNU Compiler Collection (GCC) provides a native intrinsic to the above instruction sets. Algorithm 5 describes the implement of matrix multiplication by using AVX which contains eight duplicated processing units. The instruction unit manipulates these processing units to perform matrix multiplication simultaneously. We take the calculation of c_{ij} as an example to illustrate the workflow of this algorithm. c_{ij} is obtained by multiplying the i row of matrix A denoted A_i with the j column of matrix B denoted B_j. Both A_i and B_j have length k. In the calculation process, every eight pieces of data of A_i and B_j are sent to eight processing units for matrix multiplication in parallelism manner. After each of the 8 pairs of multiplication, the result is added to c_{ij}. In other words, the data of matrix A and matrix B are split into (k / 8) blocks each of which has a length of eight, as shown in Fig. 3b. The complexity of SIMD implementation is $O(\frac{m*n*k}{8})$.

3.2 GPU implementation

3.2.1 GPU

Compared with multi-core CPUs, the computation performance of GPUs which consist of thousands of cores to handle multiple tasks at the same time is more efficient, especially for the array-based workload. Architectural-

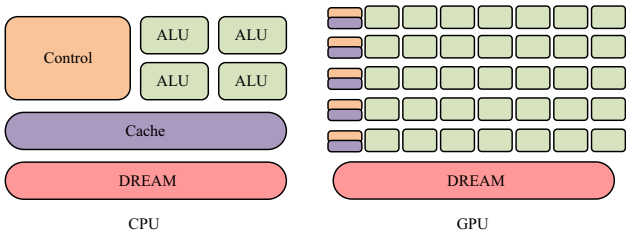


Fig. 4 CPU versus GPU. Compared with multi-core CPUs, the computation performance of GPUs is more efficient which consists of thousands of cores to handle multiple tasks at the same time, especially for the array-based workload

level comparisons of CPU and GPU are given in Fig. 4. GPUs are suitable for computation-intensive procedures whose running time is mainly spent on the operation of registers rather than data access. In addition, procedures that are easy to be parallelized are also applicable to GPUs, in order to make cores full loaded simultaneously.

3.2.2 Programming model

Compute Unified Device Architecture (CUDA)[3] is a parallel computing platform and programming model based on NVIDIA GPUs. GPU-CUDA hardware is built with grids, blocks, and threads as shown in Fig. 5. In the three-level hierarchical architecture, the execution is independent among the entities of the same level. Threads are the smallest execution unit, controlled by kernel functions to perform operations independently in parallel. Each thread has a unique thread ID (threadIdx). Blocks are organized as a 3D array of threads, and the total size of a block is limited to 1024 threads. Like thread, each block also has its unique block ID (blockIdx). Blocks that execute the same operations independently can form a grid. All the necessary data on host memory need to be transferred to the allocated device memory when kernel is executed on the device. Similarly, resultant data will be transferred back to the

[3] https://developer.nvidia.com.

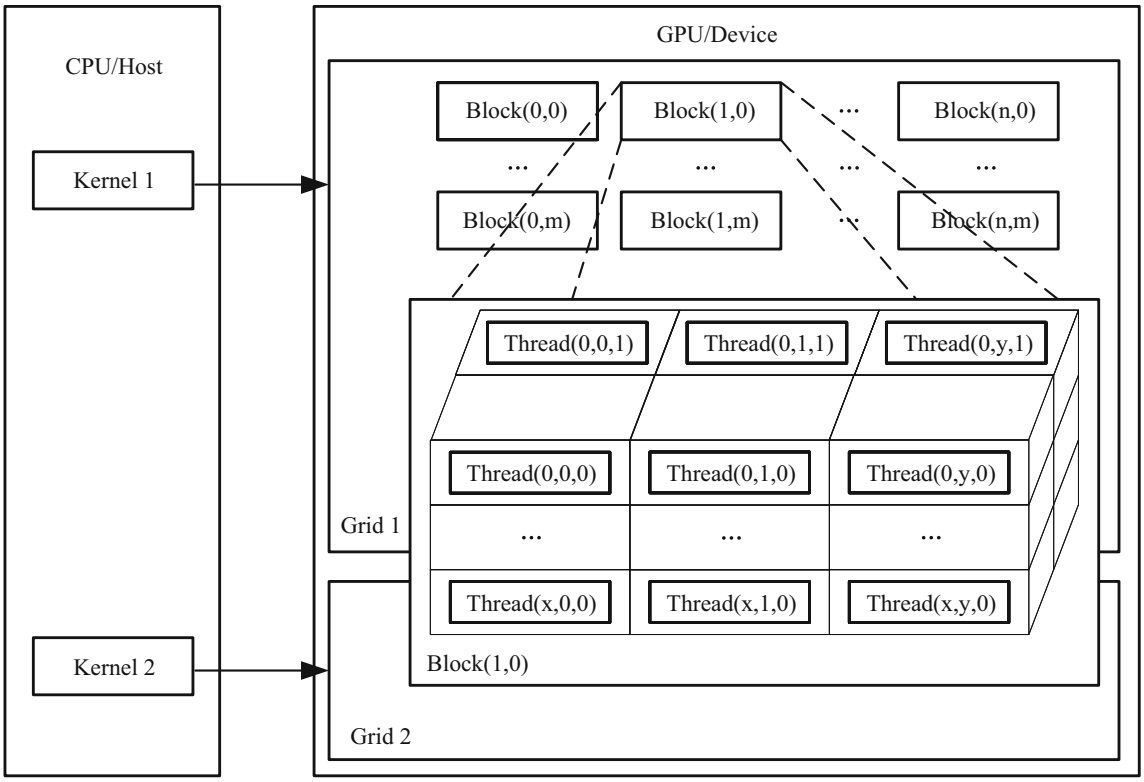


Fig. 5 CUDA model. Threads are the smallest execution unit controlled by kernel functions to perform operations independently in parallel

host, and the device memory will be released after device execution.

3.2.3 ELM implementation on GPU

In this part, we employ CUDA to design a matrix multiplication algorithm on GPU for ELM algorithm acceleration. Given an $m \times k$ matrix A and a $k \times n$ matrix B, the matrix multiplication $C = A \times B$ is an $m \times n$ matrix, each element C_{ij} can be calculated by a GPU thread, the progress of the thread is shown in Algorithm 6, and it has $m \times n$ CUDA threads in total. Obviously, the complexity of the algorithm is $O(k)$.

Algorithm 6 Matrix multiplication with GPU

```
1: threadID = (blockIdx.y * blockDim.y + threadIdx.y) * gridDim.x *
   blockDim.x + blockIdx.x * blockDim.x + threadIdx.x;
2: if threadID < m × n then
3:     row = threadID / n
4:     column = threadID % n
5:     C[row][column] = 0
6:     for l = 0 to k do
7:         C[row][column] = C[row][column] +
   A[row][l]*B[l][column]
8:     end for
9: end if
```

3.3 FPGA implementation

3.3.1 FPGA

Field-Programmable Gate Array (FPGA) is an integrated circuit custom designed by a customer or a designer after manufacturing to perform specific functions. An FPGA includes Configurable Logic Block (CLB), Input Output Block (IOB), and Interconnect. CLB module can not only be used to implement composition logic and timing logic, but also to configure distributed RAM and distributed ROM. The interconnect allows CLBs to be "wired together" or to connect to IOBs. The architecture of FPGA is shown in Fig. 6.

FPGA has the following advantages. First, FPGA has high flexibility. Compared with ASIC manufacturing process, FPGA does not need wiring and taping out. Therefore, the development process is greatly simplified and the development cycle is shortened. Second, FPGA is of high parallel computing efficiency, and it can execute multiple instructions per instruction cycle, while ASIC, DSP, and even CPU can only handle one instruction. Third, FPGA has lower power consumption, owing to its lower frequency.

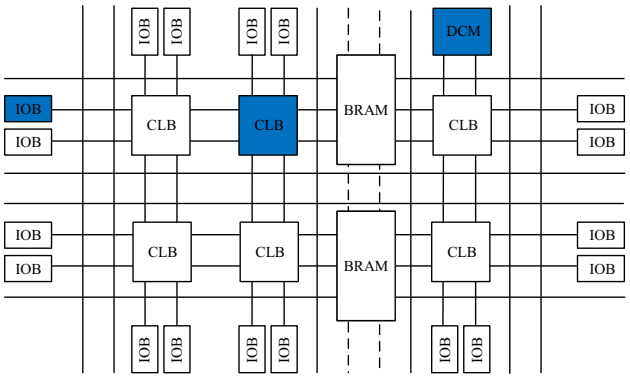


Fig. 6 FPGA chip. An FPGA includes Configurable Logic Block (CLB), Input Output Block (IOB), and Interconnect. CLB module can be used not only to implement composition logic and timing logic, but also to configure distributed RAM and distributed ROM

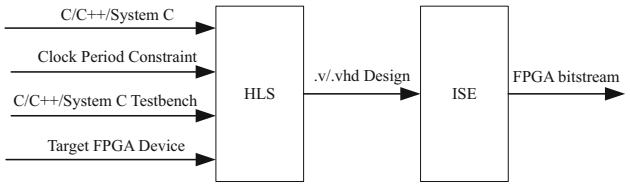


Fig. 7 HLS. The algorithm implemented with System C can be compiled and bitstreamed to the FPGA device

3.3.2 Programming model

High-Level Synthesis (HLS) not only allows engineers to develop at a high level of abstraction, but also makes it easy to generate multiple design solutions that can be deployed on FPGAs. Figure 7 shows the synthesis process. An algorithm for specific function is designed by C/C++/System C. Then, C/C++/System C testbench verifies the correctness of the design algorithm and also for the collaborative simulation of RTL and C. The collaborative simulation also includes verifying the design functionality of the generated RTL with the C/C++/System C testbench. The clock period constraint represents the target clock period which the design algorithm should run. Finally, the design will be mapped to the target FPGA device.

There are three methods, namely **loop pipelining**, **loop unrolling**, and **array partitioning**, to improve the performance of the hardware function. They all exploit the parallelism between loop iterations.

The example of **loop pipelining** is shown in Fig. 8. In the mode of serial execution, there are three clock periods between the two RD operations, and it requires six clock periods to complete an entire loop. However, with pipelining, there is only one clock period between the two RD operations and it needs four clock cycles to finish the

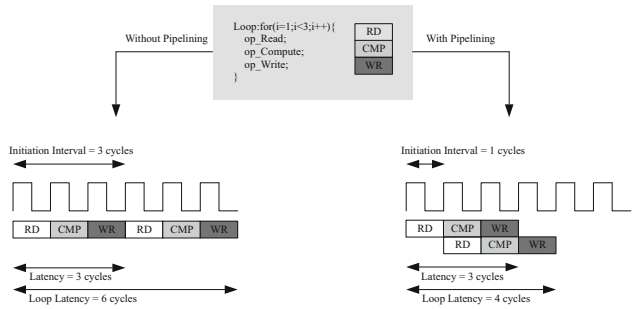


Fig. 8 Loop pipeline. The 3 operations in the loop can run with pipeline manner

entire loop. In other words, the next iteration of the loop can start finish of the current iteration; in this way, the performance of the loop can be improved.

Loop unrolling is another method to accelerate the performance of hardware. It creates multiple copies of the loop body and adjusts the loop iteration counter accordingly. For instance, unrolling a loop by creating two copies of the loop body, the loop variable referenced by each copy is updated accordingly, and the loop iteration counter is also updated. Obviously, there are more operations in each loop iteration, and more parallelism among these operations can be employed. More parallelism means more throughput and a higher system performance. As shown in Fig. 7, the main body of the loop operates 2 elements of array *arr*.

Algorithm 7 Loop with unroll optimized.

```
1: sum = 0
2: for i = 1 to 10 do
3:     sum+ = arr[i]
4:     sum+ = arr[i + 1]
5:     i+ = 2
6: end for
```

Array partitioning By default, there are only has two data ports for the arrays to read/write data to block RAM, and HLS carries out the array partition to improve the bandwidth by splitting the array into several smaller arrays, which can increase the data ports, as shown in Fig. 9. The three styles of partitioning are:

- *block* The original array is divided into same sized consecutive sub-arrays.
- *cyclic* The original array is divided into same sized interleaving sub-arrays.
- *complete* The original array is divided into individual elements.

3.3.3 FPGA for ELM

The detail is shown in Algorithm 8; the main architecture is similar to the baseline version, where the main difference is

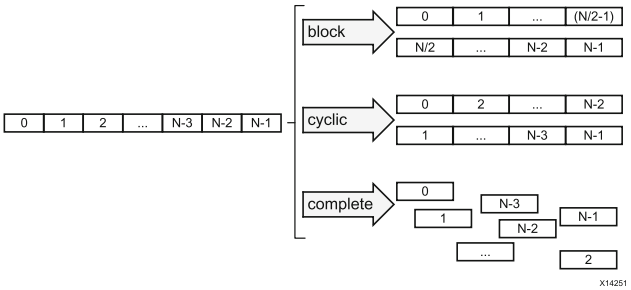


Fig. 9 FPGA array partition. HLS provides three-array partitioning method

that we can take full advantage of the parallelism of FPGA by using loop unrolling, pipelining, and array partitioning. Line 5 and line 6 are about the loop pipelining and loop unrolling preprocessor directives. The internal circuit logic of ELM training algorithm is shown in Fig. 10.

Algorithm 8 Intellectual Property kernel of Matrix Multiplication

```
1: # pragma HLS ARRAY_PARTITION variable= A block factor=Factor dim=2
2: # pragma HLS ARRAY_PARTITION variable= B block factor=Factor dim=1
3: for i = 0 to m do
4:     for j = 0 to n do
5:         C[i][j] = 0
6:         for l = 0 to k do
7:             # pragma HLS PIPELINE II = 1
8:             # pragma HLS UNROLL FACTOR = 3
9:             C[i][j] = C[i][j] + A[i][l]*B[l][j]
10:        end for
11:    end for
12: end for
```

4 Evaluation

4.1 Benchmark with CPU

Setup The ELM algorithms are evaluated by our high-end server, which is equipped with 4 Intel Xeon E7-4820 v4 CPUs and 1TB Main Memory. Ubuntu 16.04 LTS and GCC 5.4 are installed.

Several datasets We exploit the following 3 datasets to reveal their accelerating performance: CoverType,[4] Ionosphere,[5] and Sonar.[6] The source code can be found in Github.[7] The performance comparison of the three data sets is illustrated in Table 3. For the small training set, the performance is excellent, while for the large data set, steps 3 and 5 are the critical bottleneck as we have discussed in Sect. 2.2.

Multi-thread For the first group of evaluation, the dataset Covertype was used. The variable parameters used here are N and L. We set N = 500,000 and $L = 200$. In the experiment, we adjusted the number of threads to investigate the experimental features. We tested the experimental data in a similar way to that in Sect. 2.2. Table 4 shows the result of our multi-core performance under different thread counts with the same N and L. The result reveals the following findings:

1. The performance is linear with the thread count, *i.e.*,, the computing time of columns 3 and 5 decreases when the thread number increases.
2. As the number of threads increases, this approach will reach the upper bound of its performance, *i.e.*,, 32 threads and 64 threads have similar training time.

SIMD For the second group of evaluation, we integrate the SIMD techniques into the ELM algorithm. Figure 11a, b shows the training performance (single thread) under different N and L, with or without SIMD optimizations. The vertical axis represents the training time of the ELM. Figure11c shows the comparison result of the performance in multi-thread environment, with and without SIMD optimization. The following findings are obtained: The performance with SIMD optimization is better than that of baseline algorithm under the same experimental parameters. For example, the speedup ratio of SIMD is 1.67 under the parameters of N = 500,000 and $L = 200$.

4.2 Benchmark with GPU

Setup In this part, the experiment was on a workstation which is equipped with an Intel(R) Core(TM) i7 6500U CPU, a NVIDIA GeForce 940MX, and 8 GB Memory. Our development environment is Visual Studio 2010 and CUDA 9.1. To support GPU computing, we used the library cuBLAS[8] and CULA.[9] We also used the same dataset CoverType, Ionosphere, Sonar as in the above section.

Implementation First, we re-implement the baseline version of ELM on the Windows platform; then, we offloading the step 5 to step 7 in Algorithm 1 into the GPU computing card. In contrast to Sect. 4.1, we need to record the memory transforming time from host memory to device memory, because of the low speed of PCI-E. The source code of this part can be found in Github.[10]

[4] http://archive.ics.uci.edu/ml/datasets/Covertype.

[5] https://archive.ics.uci.edu/ml/machine-learning-databases/ionosphere/.

[6] https://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connectionist-bench/sonar/.

[7] https://github.com/bombe-org/ELM_exp.

[8] https://developer.nvidia.com/cublas.

[9] http://www.culatools.com/.

[10] https://github.com/bombe-org/ELM_GPU/.

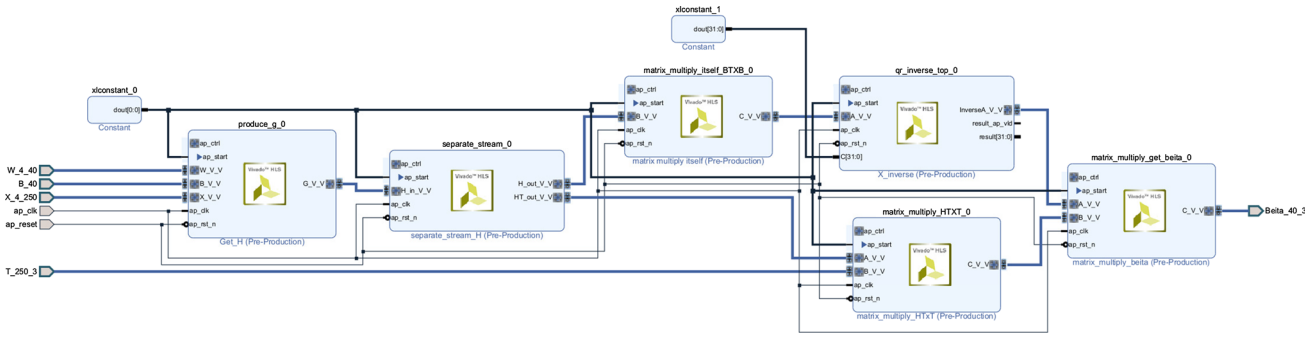


Fig. 10 FPGA logical implementation of ELM. Get_H is the IP kernel about calculating H in Algorithm 1 line 3. Matrix_multiply_itself is the IP kernel about calculating $H^T H$ in Algorithm 1 line 5

Results Table 5 shows the performance comparison based on 3 different datasets, and Table 6 shows the performance comparison of running ELM under different parameters. We can find that:

1. The performance of matrix multiplication on GPU is still proportional to the amount of data. However, the performance of matrix multiplication on GPU is much better than that on CPU (Table 5).
2. The performance is approximately 800× faster than that of the CPU only version. The training process for ELM can achieve real-time effect (Table 6).
3. GPU has a relatively high data transforming cost, especially for large datasets (Table 6 column 3).

4.3 Benchmark with FPGA

Setup Same as the previous experiments, the benchmark was run on a VCU118 FPGA chip, and parameters are shown in Table 7.

Results Table 8 shows the performance result of the implementation of step 5 in Algorithm 1 on different hardware devices, including single-core CPU, GPU, and FPGA. Since the BRAM on FPGA is small, this part of evaluation work is based on the small-scale dataset Ionosphere. We can find that the speedup is roughly 40×.

4.4 Summary

In this paper, we used several kinds of popular modern hardware devices to accelerate the original ELM algorithm. We get the follow findings:

1. Matrix multiplication is the most time-consuming operation in ELM algorithm; thus, all of the devices evaluated in this paper can accelerate the processing of ELM training phase.
2. The speedup order is: $Speedup_{GPU} > Speedup_{FPGA} > Speedup_{multi-core}$. Multi-core CPU with SIMD optimization is friendly for ELM and can get about 10× speedup. GPU's speedup is larger than 800× and is more suitable for ELM. FPGA's speedup is about 40× and is also suitable for ELM, due to its small energy consumption.
3. *Drawback* CPU will have a performance limitation when the number of threads is large, which was 32 threads in our benchmark. GPU may have a relatively high data transforming cost, because its chip has a lower power. FPGA is only suitable for relatively small datasets since the block RAM is small compared with host memory.

5 Related works

The extreme learning machine (ELM) [9–12, 14–16] was introduced by Huang as a classification algorithm with relatively fast learning speed and good generalization performance [14]. Because of these advantages, ELM can be applied in many fields and display significant application performance [13, 20, 24, 30, 34, 37, 38, 41, 43, 44].

ELM with parallelism There were several methods proposed to speed up the ELM algorithm from a parallel perspective. A parallel incremental extreme SVM classifier was proposed in [7]. The ELM algorithm for large-scale regression on GPU was proposed in the paper [1, 17, 36]. Besides, paper [8] described an algorithm that was designed and implemented on MapReduce framework. Research has shown that Field-Programmable Gate Array (FPGA) performs better than General-Purpose Processors in machine learning algorithms [39]. As the hardware implementation of ELM for classification is still at an early stage, and FPGA is a suitable device for its implementation, we are confronted with emerges the research question of what performance can be achieved by existing ELM computational methods and FPGA devices. At present, there are only a few studies touching upon this topic [5, 25, 40].

Table 3 Several dataset benchmark results

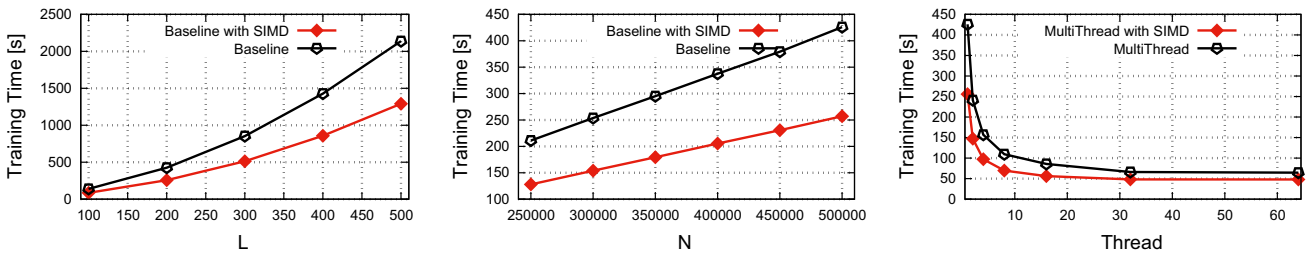
Dataset	N	L	rand(w) (s)	rand(bias) (s)	H (s)	$g(H)$ (s)	A (s)	b (s)	Solving (s)
CoverType	581,012	500	0.00170248	3.21e−05	**275.264**	63.3346	**2113.85**	39.475	0.378053
Ionosphere	351	500	0.00173075	4.79e−05	0.134948	0.036379	1.31487	0.008308	0.352429
Sonar	208	500	0.00312803	5.39e−05	0.127082	0.021495	0.797169	0.004891	0.352528

For the small training dataset, the performance is good, while for the large data set, step 3 and 5 are a critical bottleneck as discussed in Sect. 2.2

Table 4 Multi-thread performance details with CoverType dataset, which shows the result of our multi-core performance under different thread counts; we set $N = 500{,}000$ and $L = 200$

Thread	rand(w) (s)	rand(bias) (s)	H (s)	$g(H)$ (s)	A (s)	b (s)	Solving (s)	Speedup
1	0.000721	1.31e–05	**95.163**	20.9783	**296.22**	13.5306	0.032887	1
2	0.000692	1.20e–05	**56.0644**	20.9961	**149.538**	14.5034	0.033052	1.7
4	0.000699	1.27e–05	**35.8234**	21.5315	**84.6711**	14.7814	0.033429	2.7
8	0.000695	1.23e–05	**25.0924**	21.0244	**48.4229**	14.8554	0.033292	3.9
16	0.000695	1.24e–05	**20.1657**	21.1562	**29.7101**	14.6505	0.03289	4.9
32	0.000691	1.24e–05	**17.3037**	21.1233	**11.2662**	15.0423	0.033092	6.6
64	0.000691	1.26e–05	**17.0348**	21.1648	**11.1566**	15.1945	0.033096	6.6

The 1st result is the baseline performance, as we can see, the computing time of column 3 and 5 decreases when the thread increases



(a) Training Time influenced by L, with N=500000

(b) Training Time influenced by N, with L=500

(c) Training Time influenced by threads number

Fig. 11 CPU performance Analysis. The performance with SIMD optimization is better than that with baseline algorithm under the same experimental parameters. Besides, the performance is linear with the thread count

Table 5 CPU versus GPU performance on several datasets. ELM is much more suitable for GPU

Dataset	Device	N	L	Offload (ms)	A	b	Solving	Speedup
Covertype	CPU	581,012	200		**296.22** s	13.5306 s	0.032887s	
Covertype	GPU	581,012	200	72.3931	**399.949 ms**	51.1442 ms	10.0203 ms	740
Sonar	CPU	208	200		0.127548 s	0.00242006 s	0.0293581s	
Sonar	GPU	208	200	0.5825	0.3771 ms	0.1907 ms	4.086 ms	338.3
Ionosphere	CPU	351	200		0.209365 s	0.00408718 s	0.0293511 s	
Ionosphere	GPU	351	200	1.0777	0.4071 ms	0.2031 ms	4.1066 ms	522

6 Conclusion

In this work, we first evaluated the performance of ELM algorithm based on the single-core implementation and concluded that the main cost of ELM was on the matrix multiplication. Then, we designed and implemented optimization algorithms for different hardware devices (multi-core, SIMD, GPU, and FPGA). All of the above hardware can achieve performance improvement, and the best performance is obtained by using GPU, especially under large dataset. We strongly recommend that (1) use GPU to accelerate ELM algorithms for large dataset, and

Table 6 Influence of the change of N or L on GPU performance

N	L	Offload (ms)	A (ms)	b (ms)	Sovling (ms)	Speedup
250,000	200	32.8578	180.769	24.1119	3.7112	810
300,000	200	39.661	213.661	27.9726	5.4304	824
350,000	200	46.1074	252.592	32.4253	3.8824	810
400,000	200	51.1929	281.807	35.2628	4.0391	832
450,000	200	55.4978	315.737	39.8834	4.0164	833
500,000	100	33.0917	110.176	13.6463	27.7917	672
500,000	200	61.4055	358.525	44.443	302.81	826
500,000	300	92.369	776.261	63.8473	70.9121	848
500,000	400	123.684	1422.512	85.3677	73.5045	821

The performance of matrix multiplication on GPU is still proportional to the amount of data, but GPU has an extra transforming time to offload dataset (column 3)

Table 7 Parameters of the VCU118 XCVU9P-L2FLGA2104E FPGA

System Logic Cells (K)	2586
DSP Slices	6840
Memory (Mb)	345.9
GTY 32.75 Gb/s transceivers	120
I/0	832
Frequency	250 MHz

Table 8 CPU versus GPU performance on several datasets. $Speedup_{GPU} > Speedup_{FPGA} > Speedup_{multi-core}$

Dataset	Device	N	L	A (ms)	Speedup
Ionosphere	CPU	351	200	209.365	1
Ionosphere	GPU	351	200	0.4071	514.3
Ionosphere	FPGA	351	200	5.3003	**39.5**

(2) use FPGA for small dataset because of its lower power, especially for some embedded applications.

Acknowledgements Gang Wu is supported by the NSFC (Grant No.61872072) and the State Key Laboratory of Computer Software New Technology Open Project Fund (Grant No.KFKT2018B05). Guoren Wang is the corresponding author of this paper. Guoren Wang is supported by the NSFC (Grant No. U1401256, 61732003, 61332006 and 61729201).

Compliance with ethical standards

Conflict of interest The authors declared that they have no conflict of interest to this work.

References

1. Alia-Martinez M, Antoñanzas J, Antonanzas-Torres F, Pernía-Espinoza A, Urraca R (2015) A straightforward implementation of a gpu-accelerated ELM in R with NVIDIA graphic cards. In: International conference on hybrid artificial intelligence systems. Springer, Berlin, pp 656–667
2. Baldi P, Sadowski P, Whiteson D (2014) Searching for exotic particles in high-energy physics with deep learning. Nat Commun 5:4308
3. Deng L, Yu D, et al (2014) Deep learning: methods and applications. Found Trends® Signal Process 7(3–4):197–387
4. Ding L, Xin J, Wang G (2016) An efficient query processing optimization based on ELM in the cloud. Neural Comput Appl 27(1):35–44. https://doi.org/10.1007/s00521-013-1543-3
5. Frances-Villora JV, Rosado-Muñoz A, Martínez-Villena JM, Bataller-Mompean M, Guerrero JF, Wegrzyn M (2016) Hardware implementation of real-time extreme learning machine in FPGA: analysis of precision, resource occupation and performance. Comput Electr Eng 51:139–156
6. Hagan MT, Demuth HB, Beale MH, De Jesús O (1996) Neural network design, vol 20. Pws Pub, Boston
7. He Q, Du C, Wang Q, Zhuang F, Shi Z (2011) A parallel incremental extreme SVM classifier. Neurocomputing 74(16):2532–2540
8. He Q, Shang T, Zhuang F, Shi Z (2013) Parallel extreme learning machine for regression based on mapreduce. Neurocomputing 102:52–58
9. Huang GB, Chen L (2007) Convex incremental extreme learning machine. Neurocomputing 70(16–18):3056–3062
10. Huang GB, Chen L (2008) Enhanced random search based incremental extreme learning machine. Neurocomputing 71(16–18):3460–3468
11. Huang GB, Chen L, Siew CK et al (2006) Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Trans Neural Netw 17(4):879–892
12. Huang GB, Ding X, Zhou H (2010) Optimization method based extreme learning machine for classification. Neurocomputing 74(1–3):155–163
13. Huang GB, Liang NY, Rong HJ, Saratchandran P, Sundararajan N (2005) On-line sequential extreme learning machine. Comput Intell 2005:232–237
14. Huang GB, Wang DH, Lan Y (2011) Extreme learning machines: a survey. Int J Mach Learn Cybern 2(2):107–122
15. Huang GB, Zhou H, Ding X, Zhang R (2012) Extreme learning machine for regression and multiclass classification. IEEE Trans Syst Man Cybern Part B (Cybernetics) 42(2):513–529
16. Huang GB, Zhu QY, Siew CK (2006) Extreme learning machine: theory and applications. Neurocomputing 70(1–3):489–501
17. Jeowicz T, Gajdo P, Uher V, Snáel V (2015) Classification with extreme learning machine on GPU. In: 2015 international

conference on intelligent networking and collaborative systems (INCOS), pp 116–122. IEEE
18. Li H, Wu G (2014) Map matching for taxi GPS data with extreme learning machine. In: Advanced data mining and applications—10th international conference, ADMA 2014, Guilin, China, December 19–21, 2014. Proceedings, pp 447–460. https://doi.org/10.1007/978-3-319-14717-8_35
19. Li J, Wang B, Wang G, Zhang Y (2016) Probabilistic threshold query optimization based on threshold classification using ELM for uncertain data. Neurocomputing 174:211–219. https://doi.org/10.1016/j.neucom.2015.05.122
20. Liang NY, Huang GB, Saratchandran P, Sundararajan N (2006) A fast and accurate online sequential learning algorithm for feedforward networks. IEEE Trans Neural Netw 17(6):1411–1423
21. Liaw A, Wiener M et al (2002) Classification and regression by randomforest. R News 2(3):18–22
22. Ma Y, Yuan Y, Wang G, Bi X, Qin H (2018) Trust-aware personalized route query using extreme learning machine in location-based social networks. Cognit Comput 10(6):965–979. https://doi.org/10.1007/s12559-018-9600-y
23. Magsi H, Sodhro AH, Chachar FA, Abro SAK, Sodhro GH, Pirbhulal S (2018) Evolution of 5g in internet of medical things. In: 2018 international conference on computing, mathematics and engineering technologies (iCoMET), pp 1–7. IEEE
24. Rong HJ, Huang GB, Sundararajan N, Saratchandran P (2009) Online sequential fuzzy extreme learning machine for function approximation and classification problems. IEEE Trans Syst Man Cybern Part B (Cybernetics) 39(4):1067–1072
25. Safaei A, Wu QJ, Yang Y, Akılan T (2017) System-on-a-chip (soc)-based hardware acceleration for extreme learning machine. In: 2017 24th IEEE international conference on electronics, circuits and systems (ICECS), pp 470–473. IEEE
26. Schalkoff RJ (1997) Artificial neural networks, vol 1. McGraw-Hill, New York
27. Schmidhuber J (2015) Deep learning in neural networks: an overview. Neural Netw 61:85–117
28. Sodhro AH, Luo Z, Sangaiah AK, Baik SW (2019) Mobile edge computing based QOS optimization in medical healthcare applications. Int J Inf Manag 45:308–318
29. Sodhro AH, Malokani AS, Sodhro GH, Muzammal M, Zongwei L (2019) An adaptive QOS computation for medical data processing in intelligent healthcare applications. Neural Comput Appl, pp 1–12
30. Sodhro AH, Pirbhulal S, de Albuquerque VHC (2019) Artificial intelligence-driven mechanism for edge computing-based industrial applications. IEEE Trans Ind Inf 15(7):4235–4243. https://doi.org/10.1109/TII.2019.2902878
31. Sodhro AH, Pirbhulal S, Qaraqe M, Lohano S, Sodhro GH, Junejo NUR, Luo Z (2018) Power control algorithms for media transmission in remote healthcare systems. IEEE Access 6:42384–42393
32. Sodhro AH, Pirbhulal S, Sodhro GH, Gurtov A, Muzammal M, Luo Z (2018) A joint transmission power control and duty-cycle approach for smart healthcare system. IEEE Sens J 19(19):8479–8486
33. Sodhro AH, Shaikh FK, Pirbhulal S, Lodro MM, Shah MA (2017) Medical-QoS based telemedicine service selection using analytic hierarchy process. In: Handbook of large-scale distributed computing in smart healthcare. Springer, Berlin, pp 589–609
34. Sun Y, Yuan Y, Wang G (2011) An OS-ELM based distributed ensemble classification framework in p2p networks. Neurocomputing 74(16):2438–2443
35. Suykens JA, Vandewalle J (1999) Least squares support vector machine classifiers. Neural Process Lett 9(3):293–300
36. Van Heeswijk M, Miche Y, Oja E, Lendasse A (2011) Gpu-accelerated and parallelized elm ensembles for large-scale regression. Neurocomputing 74(16):2430–2437
37. Wang B, Wang G, Li J, Wang B (2012) Update strategy based on region classification using ELM for mobile object index. Soft Comput 16(9):1607–1615
38. Wang G, Zhao Y, Wang D (2008) A protein secondary structure prediction framework based on the extreme learning machine. Neurocomputing 72(1–3):262–268
39. Woods L, Teubner J, Alonso G (2011) Real-time pattern matching with FPGAD. In: 2011 IEEE 27th international conference on data engineering (ICDE), pp 1292–1295. IEEE
40. Yeam TC, Ismail N, Mashiko K, Matsuzaki T (2017) FPGA implementation of extreme learning machine system for classification. In: Region 10 conference, TENCON 2017-2017 IEEE, pp 1868–1873. IEEE
41. Zhang R, Huang GB, Sundararajan N, Saratchandran P (2007) Multicategory classification using an extreme learning machine for microarray gene expression cancer diagnosis. IEEE/ACM Trans Comput Biol Bioinform (TCBB) 4(3):485–495
42. Zhang Z, Zhao X, Wang G, Bi X (2018) A new point-of-interest classification model with an extreme learning machine. Cognit Comput 10(6):951–964. https://doi.org/10.1007/s12559-018-9599-0
43. Xg Zhao, Wang G, Bi X, Gong P, Zhao Y (2011) XML document classification based on ELM. Neurocomputing 74(16):2444–2451
44. Zhu QY, Qin AK, Suganthan PN, Huang GB (2005) Evolutionary extreme learning machine. Pattern Recognit 38(10):1759–1763