Benchmarking Hardware Accelerating Techniques for Extreme Learning Machine

Liang Li[1], Guoren Wang[2](✉), Gang Wu[1,3], and Qi Zhang[2]

[1] School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China
liliang@stumail.neu.edu.cn

[2] School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
wanggr@bit.edu.cn

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

Abstract. As a popular classification algorithm for machine learning, Extreme Learning Machine (ELM) has been widely used. However, its performance on various hardware devices is unclear. According to the baseline implementation of single core ELM, we find that the main time cost of ELM is matrix multiplication. Then, this paper designs various optimized hardware algorithms for several computing devices (Multi-Core, GPU, and FPGA). According to the experiment of each platform, we can see that the speedup ratio of the new hardware platform to ELM is 4˜100+, we open our source code and strongly recommend that the later researchers design the application of ELM algorithm based on appropriate hardware platform.

Keywords: Extreme Learning Machine · Hardware · Multi core · GPU · FPGA

1 Introduction

Hardware implementation of ELM is one of the current research trends. This has motivated substantial research to develop a system by utilizing parallel computing devices, such as Field Programmable Gate Arrays (FPGAs) [1–3], Graphic Processing Units (GPUs) [4–6] and specialized multi-core microprocessors with architecture optimized for multiple computational operations.

As far as we know, there is almost no work to fully demonstrate the performance evaluation of ELM algorithm on different computing devices. **What is the performance of the ELM algorithm on the CPU, GPU and FPGA, respectively?**

To answer this question, we evaluate the performance of ELM algorithm on different hardware platforms. First, in order to compare the experimental

© Springer Nature Switzerland AG 2020

performance, we implement a baseline version of ELM training algorithm on single core CPU. Then, we proposed several types of hardware optimized ELM algorithm, including Multi-Core, GPU and FPGA.

In this work, we get the following contributions:

1. We fully implement the single-core version of ELM algorithm, and evaluate the time cost of each step in detail. The result shows that the main time cost of ELM is matrix multiplication, *i.e.,* In CoverType dataset, Matrix multiplication accounts for roughly 97.5% of the time. (Sect. 2.2).
2. Based on the problem of excessive cost of matrix multiplication, we design the optimized version of ELM algorithm on Multi-Core CPU, GPU and FPGA, respectively (Sect. 3).
3. We implement and open the source code of the above algorithms. Experiments are performed on three data sets, namely, Covertype, Inovophere, and Sonar. Results of the experiments show that the new hardware is very suitable for accelerating ELM algorithm. Specifically, in our experiments, the Multi-Core CPU (64 threads) combined with SIMD technology can get about 9× speedup. GPU can get about 100× speedup, and FPGA can get 4× of speedup (Sect. 4).

2 Preliminary

2.1 Extreme Learning Machine

The training phase of ELM algorithm is described in Algorithm 1, which include 7 steps.

Algorithm 1. ELM training phase

```
Require: Training data: X
Require: 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: $\boldsymbol{\beta} = A^{-1} b$
```

2.2 Time-Consuming Analysis

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 dataset used in our experiment is Covertype[1].

Table 1 illustrates the execution time distribution with different N and L. Three conclusions can be drawn as follows.

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

Table 1. Baseline: single core evaluation.

N	L	rand(w) [s]	rand(bias) [s]	H [s]	g(H) [s]	A [s]	b [s]	solving [s]
500000	100	0.000355	9.03e−06	47.6114	10.4667	73.9813	6.72915	0.00656
500000	200	0.000721	1.31e−05	95.163	20.9783	296.22	13.5306	0.032887
500000	300	0.001025	2.06e−05	142.003	32.5244	658.06	20.1207	0.093655
500000	400	0.001337	2.77e−05	188.898	43.5225	1168.49	26.7123	0.205492
500000	500	0.001682	3.16e−05	236.566	52.165	1815.57	33.8406	0.377687
250000	200	0.000679	1.22e−05	47.4303	10.4087	146.5	6.7185	0.032932
300000	200	0.000679	1.22e−05	56.7862	12.4688	176.063	8.27493	0.033314
350000	200	0.000683	1.25e−05	66.4569	14.6103	204.562	9.42965	0.033097
400000	200	0.000685	1.22e−05	75.6422	16.6199	234.503	10.7515	0.032972
450000	200	0.000685	1.22e−05	85.3575	18.8	263.049	12.116	0.033029

1. First, for each instance, line 3, line 5 and line 6 in Algorithm 1 consume a lot of execution time, and they are all correspond to matrix multiplication operations.
2. Second, 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.
3. Third, under the condition that N or L is huge, all the matrix multiplication operations take up about 97.5% of the total time cost in training algorithm, which greatly affects the efficiency of ELM algorithm.

3 Hardware Accelerating

3.1 Accelerating Methods with CPU

Baseline. Now, we give an algorithm of matrix multiplication operations in single core environment to provide a baseline for other modern hardware implementation. Given an $m \times k$ matrix A and an $k \times n$ matrix B, the matrix multiplication $C = AB$ is an $m \times n$ matrix. Intuitively, the single core implementation algorithm of matrix multiplication is shown as Algorithm 2.

Algorithm 2. Matrix Muplication with single core

```
1: for $i = 1$ to $m$ do
2:     for $j = 1$ to $n$ do
3:         C[i][j] = 0
4:         for $l = 1$ to $k$ do
5:             C[i][j] = C[i][j] + A[i][l]*B[l][j]
6:         end for
7:     end for
8: end for
```

Multi-core. To accelerate ELM algorithm, we employ the performance advantage of multi-core to implement its matrix multiplication which is the most time-cost operation. Due to the independence of calculation process between sub-matrices, each can be obtained with a single core. With the parallel computation of the $a \times b$ sub-matrices, ELM algorithm can be greatly accelerated. Calculation algorithm for each sub-matrix is shown in Algorithm 3, where id represents the ID of threads.

SIMD. Algorithm 4 describes the implement of matrix multiplication by using SIMD which contains an instruction unit and eight duplicated processing units. The instruction unit controls these processing units to perform matrix multiplication simultaneously.

Algorithm 3. Matrix Muplication with multi core

```
1: $id = get\_thread\_id()$
2: row = $id/a$
3: column = $id\%a$
4: for $i = \frac{m}{a} \times row$ to $\frac{m}{a} \times (row + 1)$ do
5:     for $j = \frac{n}{b} \times column$ to $\frac{n}{b} \times (column + 1)$ do
6:         C[i][j] = 0
7:         for $l = 1$ to $k$ do
8:             C[i][j] = C[i][j] + A[i][l]*B[l][j]
9:         end for
10:     end for
11: end for
```

3.2 Accelerating Methods with GPU

ELM Implementation on GPU. Since the running time of the ELM algorithm largely consists of matrix multiplication which contains the same operation for multiple sets of data. If this operation can be accelerated, then the running time of ELM training algorithm and ELM test algorithm can be greatly reduced. In this work, we employ CUDA to design a matrix multiplication algorithm (Algorithm 5) on GPU for ELM algorithm acceleration.

3.3 Accelerating Methods with FPGA

FPGA for ELM. As discussed above, we can use the HLS to develop the main IP core of Matrix Multiplication. The detail is shown in Algorithm 6, the main skeleton is similar to baseline version, what the main different is that we can take full advantage of the parallelism of FPGA by using loop unroll and pipeline. As shown in line 5 and line 6.

4 Evaluation

4.1 Benchmark with CPU

Setup. This part of experiments, we evaluate the ELM algorithm with our high end server, which equipped with 4 Intel Xeon E7-4820 v4 CPUs and 1 TB Memory. Ubuntu 16.04 LTS and GCC 5.4 was installed. we use following 3 datasets to reveal our accelerating performance: CoverType, Ionosphere, Sonar. The source code and dataset can be found in Github[2].

Multi-thread. First group evaluation, we use the dataset Covertype. The variable parameters we used in this paper is thread numbers. We set $N = 500000$ and $L = 200$. In the experiment, we adjusted the number of threads to detect the experimental features in the experiment. First, we test the experimental data in a similar way to the experimental Sect. 2.2. Table 2 shows the result of our multicore performance under different thread counts with the same N and L.

Algorithm 4. Matrix Muplication with SIMD

```
1: for $i = 1$ to $m$ do
2:     for $j = 1$ to $n$ do
3:         temp = 0
4:         C[i][j] = 0
5:         while $l < k$ do
6:             regesiter r1 = load(A[i][l]), r9 = load(B[l][j])
7:             regesiter r2 = load(A[i][l+1]), r10 = load(B[l+1][j])
8:             regesiter r3 = load(A[i][l+2]), r11 = load(B[l+2][j])
9:             regesiter r4 = load(A[i][l+3]), r12 = load(B[l+3][j])
10:            regesiter r5 = load(A[i][i+4]), r13 = load(B[l+4][j])
11:            regesiter r6 = load(A[i][l+5]), r14 = load(B[l+5][j])
12:            regesiter r7 = load(A[i][l+6]), r15 = load(B[l+6][j])
13:            regesiter r8 = load(A[i][l+7]), r16 = load(B[l+7][j])
14:            mulps256(r1,r9), mulps256(r2,r10)
15:            mulps256(r3,r11), mulps256(r4,r12)
16:            mulps256(r5,r13), mulps256(r6,r14)
17:            mulps256(r7,r15), mulps256(r8,r16)
18:            addps256(r1,r2), addps256(r3,r4)
19:            addps256(r5,r6), addps256(r7,r8)
20:            addps256(r1,r3), addps256(r5,r7)
21:            addps256(r1,r5)
22:            C[i][j] += r1
23:            l = l + 8;
24:        end while
25:    end for
26: end for
```

[2] https://github.com/bombehub/ELM_exp.

SIMD. The second group evaluation, we take advantage of the SIMD techniques into our ELM algorithm. First, In Table 3 we evaluate the single thread performance with SIMD. Contrast experiment under different data sets. In Table 5, we show the performance comparison of three data sets. Under the small training set, the optimization is not obvious, in the big data set, the optimization is obvious.

4.2 Benchmark with GPU

Setup. This part of experiments, we run on a workstation which equipped with a Intel(R) Core(TM) i7 6500U CPU and a NVIDIA GeForce 940MX, it has 16 GB Memory. Our development environment is Visual Studio 2010 and CUDA 9.1. To Support GPU Computing, we used the library cuBLAS[3] and CULA[4]. We also use the same dataset CoverType as in the above section.

Algorithm 5. Matrix Muplication with GPU

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

Algorithm 6. Matrix Multiplication with FPGA

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

[3] https://developer.nvidia.com/cublas.
[4] http://www.culatools.com/.

Result. First, we re-implement the baseline version of ELM on the Windows platform, then we offloading the Matrix multiplication work into the GPU computing card. Compared with 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. This part of the source code can be found in Github[5].

Table 6 shows the performance comparison of running ELM with CPU and GPU, Table 7 shows the performance comparison of 3 different dataset (Table 5).

Table 2. Multi thread performance details.

Thread	rand(w) [s]	rand(bias) [s]	H [s]	g(H) [s]	A [s]	b [s]	solving [s]
1	0.000721	1.31e−05	95.163	20.9783	296.22	13.5306	0.032887
2	0.000692	1.20e−05	56.0644	20.9961	149.538	14.5034	0.033052
4	0.000699	1.27e−05	35.8234	21.5315	84.6711	14.7814	0.033429
8	0.000695	1.23e−05	25.0924	21.0244	48.4229	14.8554	0.033292
16	0.000695	1.24e−05	20.1657	21.1562	29.7101	14.6505	0.03289
32	0.000691	1.24e−05	17.3037	21.1233	112.662	15.0423	0.033092
64	0.000691	1.26e−05	17.0348	21.1648	11.1566	15.1945	0.033096

Table 3. SIMD optimized single thread.

N	L	rand(w) [s]	rand(bias) [s]	H [s]	g(H) [s]	A [s]	b [s]	solving [s]
500000	100	0.000353508	9.80e−06	31.0448	6.45206	44.4624	3.89321	0.0046852
500000	200	0.000700064	1.28e−05	61.9189	11.9778	175.412	7.72489	0.0217625
500000	300	0.00103638	2.11e−05	92.2009	19.1485 s	389.961	11.6363	0.0593751
500000	400	0.00137603	3.15e−05	123.076	25.6241 s	693.677	15.7125	0.128346
500000	500	0.00170685	3.43e−05	153.732	29.806 s	1086.93	20.1913	0.233711
250000	200	0.000690002	1.28e−05	30.855	5.97438	87.2581	3.85043	0.0217484
300000	200	0.000708409	1.30e−05	37.2234	7.17052	104.899	4.62688	0.0217016
350000	200	0.000701056	1.30e−05	43.3128	8.35351	122.316	5.39334	0.0218693
400000	200	0.00070533	1.29e−05	49.4528	9.56823	140.149	6.1774	0.0217606
450000	200	0.000705564	1.29e−05	55.5154	10.715	157.332	6.97696	0.0218365

4.3 Benchmark with FPGA

As the same as the before experiments, the benchmark work runs in a VCU118 FPGA chip. Table 8 shows the performance result of the implementation of FPGA version of ELM algorithm. Compared with Table 1, we can find that the speedup ratio of FPGA to Single Core CPU is roughly 4×.

[5] https://github.com/bombehub/ELM_GPU/.

Table 4. Multi thread with SIMD.

Thread	rand(w) [s]	rand(bias) [s]	H [s]	g(H) [s]	A [s]	b [s]	solving [s]
1	0.000707	1.31e−05	61.5375	12.5624	173.9	7.68666	0.021873
2	0.000698	1.25e−05	39.0998	12.1115	87.8618	8.18397	0.021833
4	0.000705	1.32e−05	26.7559	12.0438	50.1302	8.31437	0.021899
8	0.000705	1.25e−05	20.4529	12.0952	28.823	8.29327	0.021929
16	0.000707	1.30e−05	17.5984	12.1333	17.9509	8.35382	0.021935
32	0.000727	1.35e−05	15.693	12.2235	67.8766	8.3262	0.021904
64	0.000708	1.79e−05	15.9949	12.1418	11.4846	8.37525	0.021959

Table 5. Several dataset benchmark result.

Dataset	N	L	Thread	isSIMD	rand(w) [s]	rand(bias) [s]	H [s]	g(H) [s]	A [s]	b [s]	solving [s]
CoverType	581012	500	1	0	0.00170248	3.21e−05	275.264	63.3346	2113.85	39.475	0.378053
Ionosphere	351	500	1	0	0.00173075	4.79e−05	0.134948	0.036379	1.31487	0.008308	0.352429
Sonar	208	500	1	0	0.00312803	5.39e−05	0.127082	0.021495	0.797169	0.004891	0.352528

Table 6. The influence of the change of N or L on GPU performance.

N	L	Offload [ms]	A [ms]	b [ms]	sovling [ms]
25000	100	3.05067	5.86667	1.6482	34.7678
30000	100	3.32879	6.96455	1.90855	2.05906
35000	100	3.78035	7.98342	2.1282	13.9374
40000	100	4.26075	9.14806	2.42331	2.02232
45000	100	4.25482	10.3581	2.68524	36.3825
50000	100	5.02914	11.4359	3.04751	2.58489
25000	10	0.873878	0.708742	0.608001	0.44405
25000	20	1.25946	0.881384	0.704791	1.49808
25000	30	1.65926	0.970273	0.792495	1.67151
25000	40	1.89156	1.82005	0.904297	86.4325
25000	50	2.00376	1.89432	1.00148	28.2963

Table 7. CPU vs. GPU performance on several dataset.

Device	Dataset	N	L	Offload	A	b	sovling
CPU	Covertype	25000	100		9.876 s	0.865 s	0.008 s
CPU	Sonar	208	100		0.169 s	0.005 s	0.013 s
CPU	Ionosphere	351	100		0.072 s	0.005 s	0.014 s
GPU	Covertype	25000	100	2.81324 ms	5.78569 ms	1.62252 ms	81.6803 ms
GPU	Sonar	208	100	0.757334 ms	0.32 ms	0.170667 ms	19.1771 ms
GPU	Ionosphere	351	100	0.896396 ms	0.773137 ms	0.28405 ms	20.8624 ms

Table 8. Evaluation on FPGA.

Thread	N	L	H [s]	A [s]	b [s]
1	500000	100	12.90285	17.49533	1.582288
1	500000	200	22.79075	73.055	3.18265
1	500000	300	34.50075	161.515	5.930175
1	500000	400	45.2245	297.1225	6.478075
1	500000	500	56.1415	458.8925	7.46015
1	250000	200	12.85758	35.625	1.379625
1	300000	200	15.19655	43.01575	2.568733
1	350000	200	17.61423	52.1405	2.757413
1	400000	200	17.91055	59.62575	2.987875
1	450000	200	20.33938	64.76225	3.629
2	500000	200	15.0161	38.3845	3.72585
4	500000	200	9.95585	22.16778	3.89535
8	500000	200	6.9731	11.10573	3.31385
16	500000	200	5.841425	7.527525	3.062625
32	500000	200	4.525925	3.3655	3.560575
64	500000	200	4.6587	2.28915	3.298625

Table 9. Summary of several device characteristics.

Device	Speedup	Advantage	Disadvantage
Multi-Core with SIMD (E7-4820 v4)	1–10	Programming friendly	High power consumption
GPU (NVIDIA GeForce 940MX)	100+	Almost real time in large data	High data transform cost
FPGA (VCU118)	3–5	Low power and low latency	Programming difficult

4.4 Summary

In this paper, we widely used several popular kind of computing device to accelerating ELM algorithm, we get the follow findings:

1. ELM algorithm's time consuming cost is in the matrix multiplication.
2. Multi-Core CPU with SIMD optimization is good for ELM, can get about 9× speedup, but it will have a performance limit when thread number is huge, *i.e.,* 32 threads in our benchmark.
3. GPU's speedup is large than 100×, and is very suitable for ELM, but may have a relatively high data transforming cost.
4. FPGA's speedup is 4× and is also very good at ELM, because this kind of chip has a low latency and lower power.

All the characteristics are shown in Table 9.

5 Conclusion

In this work, we first based on the single core implement and evaluate the performance of ELM algorithm, concluding that the main cost of ELM is on the matrix multiplication. Then, we design and implement optimization algorithms for different hardware devices (Multi Core, Multi Core with SIMD, GPU, FPGA). both of the above device can get performance improvement, the best performance is using GPU, especially under large dataset. We strongly recommended that (1) using GPU to accelerate the ELM algorithms for large dataset; (2) using FPGA for small dataset since its lower power, especially for some embedded applications.

Acknowledgments. Guoren Wang is supported by the NSFC (Grant No. U1401256, 61732003, 61332006 and 61729201). 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).

References

1. Yeam, T.C., Ismail, N., Mashiko, K., Matsuzaki, T.: FPGA implementation of extreme learning machine system for classification. In: 2017 IEEE Region 10 Conference, TENCON 2017, pp. 1868–1873. IEEE (2017)
2. Frances-Villora, J.V., Rosado-Muñoz, A., Martínez-Villena, J.M., Bataller-Mompean, M., Guerrero, J.F., Wegrzyn, M.: Hardware implementation of real-time extreme learning machine in FPGA: analysis of precision, resource occupation and performance. Comput. Electr. Eng. **51**, 139–156 (2016)
3. Safaei, A., Wu, Q.J., Yang, Y., Akılan, T.: 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 (2017)
4. Van Heeswijk, M., Miche, Y., Oja, E., Lendasse, A.: GPU-accelerated and parallelized ELM ensembles for large-scale regression. Neurocomputing **74**(16), 2430–2437 (2011)
5. Alia-Martinez, M., Antoñanzas, J., Antonanzas-Torres, F., Pernía-Espinoza, A., Urraca, R.: A straightforward implementation of a GPU-accelerated ELM in R with NVIDIA graphic cards. In: International Conference on Hybrid Artificial Intelligence Systems, pp. 656–667. Springer (2015)
6. Jeowicz, T., Gajdo, P., Uher, V., Snáel, V.: Classification with extreme learning machine on GPU. In: 2015 International Conference on Intelligent Networking and Collaborative Systems (INCOS), pp. 116–122. IEEE (2015)