Performance Tuning of Deep Learning Framework Chainer on the K computer

Akiyoshi KURODA[1], Kiyoshi KUMAHATA[1], Syuichi CHIBA[2], Katsutoshi TAKASHINA[2] and Kazuo MINAMI[1]
[1] R-CCS, Operations and Computer Technologies Division, Application Tuning Development Unit.
[2] FUJITSU LIMITED.

RIKEN R-CCS

1. Introduction

Recently GPUs has become a popular platform for executing deep learning (DL) workloads. We revisit the idea of doing **DL on CPUs**, especially massively parallel CPU clusters (supercomputers). In anticipation of deployment of the Supercomputer Fugaku with much more DL capable CPUs, we investigate which optimizations can be already done using the K computer, **current leadership computing facility and predecessor to the Supercomputer Fugaku**. We use **Chainer as a deep learning framework** of choice.

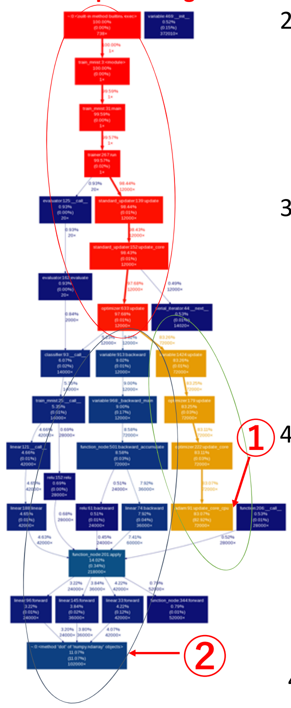

Fig.1 Call graph sample of Chainer on the K computer

2. Characteristic of Chainer

– Profiling results of Chainer using cProfile+gprof2dot are [Fig.1][Table.1] .
  - Total execution time of **10.311 s** breaks down as follow:
    – **Adam optimizer** [adam.py]: 84%①.
    – **numpy.dot** called from linear.py: 11%②.
    – Other parts (ReLU and Python control and so on.): 5%.

Table.1 ASIS Chainer Profiler (cProfile) result on the K computer.

・Original Chainer Ver.4.4.0 profile for MNIST sample (unit=1,000, epoch=20) on the K computer

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
① 72000	8,405.11	0.117	8,421.99	0.117	optimizers/adam.py:91(update_core_cpu)
② 102000	1,122.61	0.011	1,122.61	0.011	{method 'dot' of 'numpy.ndarray' objects}
28000	46.44	0.002	47.38	0.002	activation/relu.py:29(forward_cpu)
218000	34.60	0.000	1,421.38	0.007	function_node.py:201(apply)
24000	30.97	0.001	32.05	0.001	activation/relu.py:96(forward_cpu)

3. Adam optimizer

– Optimizer consumed **84% of execution time** and ran with **0.04% efficiency** as measured using Fujitsu hardware counters.
– The dominant operation is **square root of matrix elements** called from NumPy for filter update.
– This function in NumPy was implemented with **automatic C language code generation**, and thus difficult to tune directly..
– We rewrote all calculations in Adam using vectorized Fortran library and **SIMD conversion and software pipelining (SWPL)**.
– In filter update calculations many values happened to be close to zero(**denormalized number**) , raising underflow exceptions. We recompiled Python **with an option forcing to truncate such numbers to zero**.
– We have also applied **SWPL and masked SIMD** using Fortran Library to implement ReLU activation function.

```
Line# Hits    Time       Per Hit % Time  Line Contents
==============================================================
104    7200 4482481.0   622.6   21.2 m += (1 - hp.beta1) * (grad - m)
105    7200 4366618.0   606.5   20.6 v += (1 - hp.beta2) * (grad * grad - v)
       7200 7337882.0 1019.2   34.6 param.data -= hp.eta * (self.lr * m / (numpy.sqrt(vhat) + hp.eps) +
113    7200  4864633.0   675.6  23.0         hp.weight_decay_rate * param.data)
```

Fig.2 Line costs of Adam optimizer region.

4. numpy.dot GEMM convolution

– achieves only **7.76%** peak performance.
– NumPy was compiled against vectorized (using SSl II) but **single-threaded Fujitsu BLAS library**.
– We modified numpy.dot to call **multi-threaded version.**

Table.2 numPy.dot calculation.

subroutine	calculation	gemm size(M,N,K)	#call	elapse [s]	efficency %
linear.py:33(forward)	y=x.dot(W.T)	(100, 1000, 784), (100, 1000, 1000), (100, 10, 1000)	x42000	429.3[s]	9.20%
linear.py:96(forward)	gx=gy.dot(W)	(100, 1000, 10), (100, 1000, 1000)	x24000	283.2[s]	6.73%
linear.py:145(forward)	gW=gy.T.dot(x)	(10, 1000, 100) , (1000, 1000, 100), (1000, 784, 100)	x36000	380.6[s]	8.90%

4. other cost

– The ratio of the other costs such as **ReLU calculations** and **Python control regions** in ASIS Chainer is small (4.5%).
– These regions almost **not vectorized (SIMD) and parallelized**.
– We adopted **FlatMPI** on these regions for threads parallelization.

Table.3 Elapsed time of tuning stage.

name	tuning	GEMM	Adam	other	total
original		1,189.1	8,657.4	465.2	10,311.7
underflow	Kfast(Kns)	990.2	1,681.9	372.2	3,044.4
GEMM	thread	236.5	8,386.9	336.9	8,960.2
Adam	FortranLib.	1,052.6	780.7	439.6	2,272.9
all		194.9	41.3	399.0	635.2
FlatMPI (8proc.)		131.9	40.6	78.6	283.4

Table.4 Performance efficiency of GEMM and sqrt.

name	tuning	GEMM	sqrt
original		7.76%	0.04%
underflow	Kfast(Kns)	9.31%	0.19%
GEMM	thread	38.81%	0.48%
Adam	FortranLib.	8.76%	0.92%
all		47.03%	17.89%
FlatMPI (8proc.)		70.05%	18.64%

Table.5 Profiler result after all tuning.

・Tuned Chainer Ver.4.4.0 profile for MNIST sample (unit=1,000, epoch=20) on the K computer

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
102000	166.80	0.002	166.80	0.002	{method 'dot' of 'numpy.ndarray' objects}
72000	60.73	0.001	104.47	0.001	optimizers/adam.py:53(adam_kro2)
218000	38.49	0.000	440.19	0.002	function_node.py:201(apply)
12000	18.17	0.002	292.06	0.024	variable.py:968(_backward_main)
364000	16.73	0.000	21.16	0.000	numpy/ctypeslib.py:196(from_param)

5. Result

– Speedups by each tuning step [Fig.3]:
  - SSL II thread parallel BLAS effect in numpy.dot: **1.15x**
  - SWPL effect by Fortran library for Adam's sqrt: **4.54x**
  - floating point underflow control effect: **3.38x**
  - FlatMPI effect: **2.24x**
  - Total speedup: **36.4x**

– Efficiency improvements [Table 4]:
  - Performance efficiency of the numpy.dot: **7.76%→38.81%→47.03%→70.05% (9.03x)**
  - Performance efficiency of the Adam optimizer: **0.04%→ 0.92%→17.89%→18.64% (511x)**

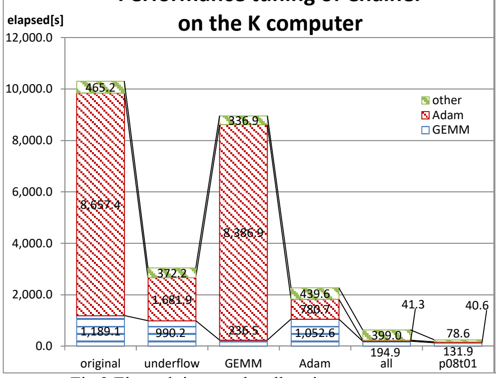

Fig.3 Elapsed time on the all tuning step.

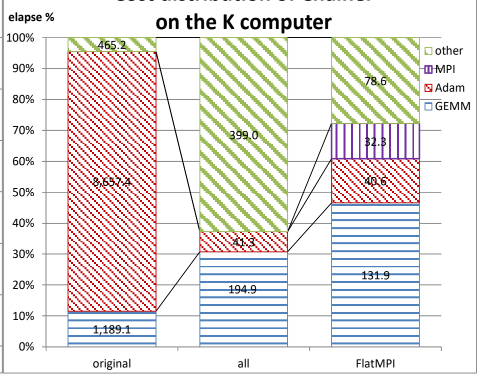

Fig.4 The change of cost distribution by the tuning effect.

6. Parallelization by the ChainerMN

We also tried to **evaluate ChainerMN-1.3.0 and released it to the K computer users** as **an R-CCS software.**

– Scalability of ChainerMN on the K computer is [Fig.5].
  - Measurement conditions: MNIST sample (unit=1000, epoch=20 → 1epoch=600iter).
  - Although it can be measured even using 600 proc. or more, we must take care of recurrence and consistency of results.
  - **It scales well up to about 200 processes** for this MNIST sample problem.
  - It became possible to calculate in **10 seconds for MNIST sample** on the K computer.

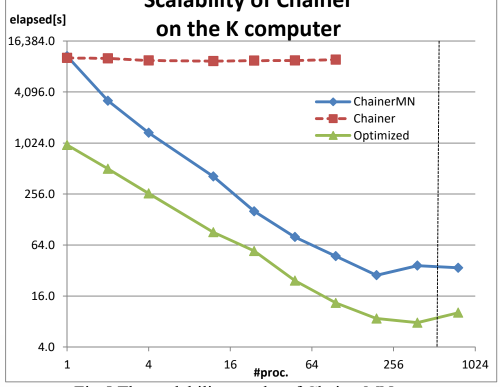

Fig.5 The scalability results of ChainerMN.

7. Conclusion

There are some limitations on the use of Chainer on the K computer. It is necessary to prepare the learning data beforehand and to **stage-in the data to an appropriate storage system**. Moreover, since **Python is in the shared storage**, it takes time to load the library. However, it was confirmed that **we can use the K computer for deep learning sufficiently as well as GPU. Now, we are preparing for the use of deep learning calculations on the Fugaku.**

HPC Asia 2020, ACROS Fukuoka, January 15-17, 2020