Accelerate Parallel Deep Learning Inferences with MCTS in the game of Go

Ching-Nung Lin[1,a)] Shi-Jim Yen[1,b)] Jr-Chang Chen[2,c)]

Abstract: The performance of Deep Learning Inference is a serious issue when combining with speed constraint Monte Carlo Tree Search(MCTS). Traditional hybrid CPU and Graphics processing unit solution is bounded because of frequently heavy data transferring. This research focuses on accelerating parallel synchronized Deep Convolution Neural Network(DCNN) prediction in MCTS. This paper proposes a method to accelerate parallel DCNN prediction and MCTS execution at GPU, Intel AVX-512 CPU and Xeon Phi Corner. It outperforms the original architecture using the GPU forwarding server. In some cases, GPU speeds up 7.2 times; AVX-512 CPU increase 15.7 times speed. Xeon Phi Corner accelerates 11.1 times performance. In addition, with 64 threads in Google Cloud Platform, maximal 53.8 times faster is achieved.

1. Introduction

The Computer Go programs combining Deep Convolutional Neural Network(DCNN) and Monte Carlo Tree Search(MCTS) make a big step ahead in the game of Go[1][2]. Achieving this level requires huge computation power. However, using devices such as the graphics processing unit(GPU) does not utilize all the resource in the hardware when doing deep learning inferences with Batch size = 1. Also, in the current computer architecture, the number of CPU threads is more than GPU devices(A server usually has more than 64 threads and has 8 GPUs or less, the ratio is 8 to 1 or more). This research provides a method to utilize all the threads in CPU and GPU to maximize the overall performance when DCNN forwarding with Batch size = 1.

Unlike deep learning in image recognition and other similar domains, throughput performance(How many forwarding can be done per second) in DCNN with searching is not so important. Simulating or searching with DCNN usually requires the batch size as small as better. In this kind of applications, if the state is not divergent enough, all the forwarding results are duplicated and useless. Some researches[3] try to use statistic to estimate the states in order to achieve the performance with the big batch size to accelerate forwarding performance. However the result is not significantly better than the small batch size. The statistic states might occur rarely in the real searching tree. Therefore, this research focuses on DCNN inferences with Batch size = 1.

The DCNN inference speed is usually slower than tradition pattern based evaluation when doing search or simulation. Usually, the researcher[2] uses the asynchronized method to utilize all hardware resources. There is a forwarding server. Because DCNN forwarding is usually 1000 time slower than traditional pattern based ones, when threads are waiting for the forwarding server to return the results, pattern based evaluations are used to do expansion and simulation at the same time. Synchronized DCNN expanding or simulating with MCTS is selected in this research because it is not domain specific. This architecture is straight and easy to be implemented. So, it is easy to port to other applications and possible to be efficiently.

This research introduces a new method to share single hardware(GPU,CPU and Xeon Phi(PHI)) with many threads simultaneously. On the other hand, For one GPU, it can be considered as many small virtual GPUs(vGPU) to maximize the forwarding speed in deep learning. For CPU and PHI, intuitively each thread is considered as a unit. In MCTS, a single tree with global lock is used. This paper is described as below: Firstly, this method requires threads to direct connect the hardware(GPU or CPU) to each thread.

[1] Dept. of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan
[2] Dept. of Computer Science and Information Engineering, National Taipei University, Taiwan
[a)] 810221001@gms.ndhu.edu.tw
[b)] sjyen@mail.ndhu.edu.tw
[c)] jcchen@cycu.edu.tw

	i7-7800X	Tesla P100
Cores	6	56 SMs
Threads	12	3584
Clock Speed	4.6GHz	1.33GHz
AVX-512 Speed	4.4GHz	
L1 Data(KB)	32	$64 + 48$
L2 Data(KB)	$1024 * 6$	4096
L3 Data(KB)	$1375 * 6$	
Independent unit	12 with some sharing	56
	Xeon Phi(31SP1)	GTX 1070
Cores	57	15 SMs
Threads	228	1920
Clock Speed	1.1GHz	1.797GHz
L1 Data(KB)	32	$64 + 48$
L2 Data(KB)	$512 * 57$	2048
Independent unit	228	15
	Google Cloud	
Cores	64 vCPU	
Threads	64	
Clock Speed	2.0GHz	
AVX-512 Speed	2.0GHz	
L1 Data(KB)	32	
L2 Data(KB)	$1024 * (32 or 64)$	
Total Cache(KB)	56320	
Independent unit	64	

Table 1 Table of Intel i7-7800X, Nvidia Tesla P100, Intel Xeon Phi Corner, Nvidia GTX 1070 and Google Cloud Platform architectures

Secondly, it shares the read-only trained model with all threads in the same GPU or CPU and keeps private individual necessity data only. For the computation part: In GPU. the cuDNN library is used; In CPU, AVX-512 and IMCI instructions are used. Then, the experimental results are presented. Finally a conclusion and future work are described.

2. Architecture

Table 1 shows all architectures used in the experimental tests. The on-board memory is excluded because of at least 10 times slowness. The main differences between GPU and CPU are that GPU has more computation power and CPU has more local cache. The GPU can execute more threads simultaneously which is efficient for deep learning training and forwarding. On the contrary, PHI has more Independent cores which can run different processes concurrently and each core has bigger cache size. For CPU, the clock is fixed as 4.6GHz, and clock for AVX-512 is 4.4GHz. With AVX-512 instructions, the computation performance increases dramatically. All CPU architectures support 512 bit Single instruction, multiple data(SIMD) instructions, Load, store and max(which is used in ReLU) 16 float or integer values simultaneously. In addition, SIMD can do 16 FMA($A*B+C$) at one time which is important at matrix multiplication. About Half Precision Floating-point Format(FP16) or 8 Bit Integer Format(INT8), store everything with the 16 bit float or the 8 bit integer type, which is half or 1/4 size and not sacrificing any inference quality[4]. However, there is no big advantage in our case(Test training on Tesla P100 and forwarding on PHI).

In addition, bigger cache size makes data locality easily. For accelerating DCNN inference, fit as much as data in the local cache memory is an important key point. Obviously, CPUs has more local memory per core. For example, each core having 4 threads in PHI can share 512KB memory comparing to 56 SMs in Telsa P100 sharing with 4096KB memory. In addition, AVX-512 CPUs have bigger cache size(1024KB/2). In addition, the latency accessing the local L2 cache: PHI is about 11 cycles; AVX-512 CPU is about 13 cycles; GPU is more than 100 cycles. Both GPU and CPU can scale up to 8 chips in a single hardware. However, CPU can scale up more with the Omni-Path architecture.

3. Method

There are two steps to accelerate parallel deep learning inferences with MCTS in the game of Go. First, use direct connection to replace a DCNN forwarding server. Second, sharing a single hardware with multiple threads to maximize the overall resources.

3.1 Direct connection

The darkforestGo[1] uses a forwarding server to handle the DCNN inference. Each thread sends the input feature planes to the server and waits for the prediction result returning from the server. It consumes 2 spare threads to keep the server running at each GPU. Instead of this, our method connects the input feature planes to GPU directly via NVIDIA cuDNN library. Therefore, there is no synchronizing and locking overhead in this direct connection architecture. It can minimize the communication overhead between CPU and GPU. For CPU and PHI, everything executes on the same hardware. There is no communication bottleneck.

3.2 Sharing a single hardware with multiple threads

In GPU, split the SMs to small groups as many individual vGPUs. For example, Tesla P100 can be grouped as 8 vGPUs, so each vGPU has 7 SMs. Also, GTX 1070 can be split to 5 vGPUs, each vGPU has 3SMs(See Table 1). When parallel DCNN forwarding in a single GPU, if the network model size fits into the L2 cache, it will gain the best performance. For GTX 1070, it can handle approximate 50000 weight parameters. For Tesla P100, it can handle 1 millions weight parameters in the L2. Since the Cache hit miss is reduced, parallel kernels on GPU have less penalty to read weight data from the GPU memory. So it can achieve the maximal utilization. Because cuDNN and CUDA are binary-only software, it is difficult to hack it to get better performance. Our method cannot scale up to over 16 threads whatever trying different configurations in the 10 GPU server.

In CPU, in order to maximize the performance, data locality and SIMD are required. AVX-512 and IMCI intrinsics are heavily used in the forwarding engine. few data prefetch instructions are used in AVX-512 CPU and more are used in PHI. There are two parts:

Table 2 a 7x7 Input matrix and a 3x3 Filter matrix

x11	x21	x31	x41	x51	x61	x71
x12	x22	x32	x42	x52	x62	x72
x13	x23	x33	x43	x53	x63	x73
x14	x24	x34	x44	x54	x64	x74
x15	x25	x35	x45	x55	x65	x75
x16	x26	x36	x46	x56	x66	x76
x17	x27	x37	x47	x57	x67	x77

y11	y21	y31
y12	y22	y32
y13	y23	y33

First, calculate the matrix multiplication and store the result in the cache. Table 2 is an example. The stride size is 1 and zero padding is 1, so the input size is 7x7 and the output size is 7x7. The first element in the output is calculated as $x11 * y22 + x21 * y32 + x12 * y23 + x22 * y33$; Then, stride one cell right, the second element in the output is $x11 * y12 + x21 * y22 + x31 * y32 + x12 * y13 + x22 * y23 + x32 * y33$. All elements in the input matrix mostly are required to multiply all elements in the filter matrix. So, calculate the multiplication of each element in the input matrix and every element in the filter matrix. By the benefit of vectorization, sixteen input elements can be processed at the same time. An example is in one instruction $x11, x21, x31, x41, ..., x13, x23$ can multiply $y11$ simultaneously. Then the result is save to a cache. In our application, the input size is 19x19 and the filter size is 3x3. an two dimension array [9][361] is necessary to cache all the result whose size is approximate 13KB.(We change 361 to 368 to maintain the alignment memory loading). It fits to the CPU L1 cache.

The Pseudo-code is below(See algorithm 1). Usually there are many channels in each layer, when calculating the second channel, the fused multiply-add (FMA) instruction can be applied as multiple input and weight and add previous result at the same time. In this algorithm, only inputs are repeatedly loading C times. Because it fits the L1 cache, the memory loading penalty remains small. For AVX-512 CPU, the _mm512_set1_ps can be replaced with _mm512_load_ps and _mm512_permutexvar_ps which gains a little better performance.

Algorithm 1 Matrix multiplication

```
Require: Exist array temp_store[9][368], Channel Size C
Ensure: 16input, 1weight, loadinput, loadweight
  for $i = 1$ to $C$ do
    for $j = 1$ to 23 do
      loadinput ← _mm512_load_ps(16input) {Load 16 data}
      for $k = 1$ to 9 do
        loadweight ← _mm512_set1_ps(1weight) {Load 1 data and broadcast}
        $temp\_store = loadinput * loadweight + temp\_store$ {FMA}
      end for
    end for
  end for
```

Algorithm 2 AVX-512 SIMD reduction

```
Require: Exist array temp_store[9][368]
Ensure: loadweight, index0_cache, nonempty_mask, final_result
  for $i = 1$ to 23 do
    loadweight ← _mm512_load_ps(temp_store) {Load aligned data}
    index0_cache ← _mm512_permutexvar_ps(loadweight) {Faster than load unaligned data, and the left (0,12),(0,13),(0,14),(0,15) can be used at the input 32-47}
    nonempty_mask ← _mm512_int2mask( (0,0), ...,(0,11) ) {add (0,0) to (0,11) only}
    final_result ← _mm512_mask_add_ps(fianl_result, nonempty, index0_cache, final_result) {SIMD add}
    ...
  end for
```

Second, Reducing with SIMD. Table 3 shows the relationship between temp_store and the final output. There are some patterns. With SIMD, 16 elements can be calculated at the same time. The Pseudo-codes are below(See algorithm 2 and 3). For AVX-512 CPU, using aligned data SIMD load and permutation instructions is faster than unaligned data load. For PHI, unaligned data SIMD load instructions are the only solution. With 2 masks and SIMD mask add, the 16 elements can be calculated in 9 cycles. This is faster than _mm512_reduce_add_ps.

Table 3 Input 16-31 elements reduction from temp_store

(0,-4)	(1,-3)	(2,-2)	(3,15)	(4,16)	(5,17)	...
(0,-3)	(1,-2)	(2,-1)	(3,16)	(4,17)	(5,18)	...
(0,-2)	(1,-1)	(2,0)	(3,17)	(4,18)		...
(0,-1)	(1,0)	(2,1)		(4,19)	(5,20)	...
(0,0)	(1,1)	(2,2)	(3,19)	(4,20)	(5,21)	...
(0,1)	(1,2)	(2,3)	(3,20)	(4,21)	(5,22)	...
...	...	...	...	...	...	...
(0,11)	(1,12)	(2,13)	(3,30)	(4,31)	(5,32)	...

Algorithm 3 IMCI SIMD reduction

```
Require: Exist array temp_store[9][368]
Ensure: loadweight1, loadweight2, index0_cache,
  nonempty_mask, final_result
  for $i = 1$ to 23 do
    loadweight1 ← _mm512_loadunpacklo_ps(temp_store)
    {Load unaligned data (5,17) to (5,31)}
    loadweight2 ← _mm512_loadunpackhi_ps(temp_store)
    {Load unaligned data (5,32)}
    nonempty_mask ← _mm512_int2mask( (5,17),(5,18)
    and (5,20),...,(5,32) ) {skip (5,19)}
    final_result ← _mm512_mask_add_ps(fianl_result,
    nonempty, index0_cache, final_result) {SIMD add}
    ...
  end for
```

Finally, it is important to train a smaller network which fits the cache in order to gain parallel DCNN forwarding performance. For example, the darkforestGo model has 57% accuracy, but its network has 14 million weight parameters which does not scale well with the sharing hardware method. Table 4 describes the DCNN models trained with smaller weight parameters and pretty good accuracy. It is about 1/40 size or less than the one in darkforestGo. It can run more than 4 threads concurrently in a Tesla P100. In face, it is not easy to get a smaller network with good accuracy. There are two methods: The first method is deep compression[5] which prunes some network weights out and reorganize the same data together. But it is time consuming and the retraining result is not guaranteed. The second method is using knowledge planes[6] which can reduce the network size. In Table 4, the original darkforestGo Torch training program is used. The ReLU activation function is used as same as the original one after each layer but the last one. Some networks use Batch Normalization layer and some do not. The Last Layer is always the same size 3x3x3. However, in the DCNN forwarding, only 3x3x1 is used. The accuracy is validated with the same validating data in the darkforestGo. The training data is different. 446832 9 Dan games from Tygen Go server are used. For the feature planes, it remains as same as darkforestGo instead of few modification. Since all the games have the same rank, so 9 rank feature planes are removed and one all 0 and one all 1 feature planes are added. The additional 2 and 4 planes are the 2 liberty Ladder and Snapback feature plane(LAD) and the 1 liberty Ladder escape feature plane(ESP). In the 20 feature plane network, the opponent side groups LAD and the self side groups ESP are considered. In the 22 feature plane network, both sides are considered. If some stones can be captured or escaped, the locations for those stones in the LAD or ESP plane are set to 1; Others are set to 0.

Usually, the DCNN forwarding benchmark only considers pure forwarding speed because the feature planes are simple. A good example is DCNN forwarding in the image recognition. But, in the domain such as games, the time to construct feature planes should be considered as a factor to affect overall performance. For example, AlphaGo uses 48 planes. To construct those feature planes costs extra computing power which is most efficiently on CPU. It is a trade off between more feature planes and smaller network size. When the network size is small enough, the multiple threads can share the same hardware and fit to the cache to achieve better performance. Finally, all threads use the same forwarding function with own private data.

Network	Features	First Layer	Inner Layers
A	20	20x5x5	3x3x32x17
B	22	22x5x5	3x3x24x18
C	22	22x5x5	3x3x16x18
NA	22	64x3x3	3x3x64x9
NB	22	32x3x3	3x3x32x14
NC	22	22x3x3	3x3x22x12
Network	**Acc. %**	**Batch Norm.**	**Weights**
A	49.1	Yes	175841
B	47.2	Yes	109009
C	44.8	Yes	51937
NA	50.7	No	345665
NB	48.4	No	136129
NC	45.8	No	57113

Table 4 The 6 architectures use on the experimental tests

4. Experimental Result

The test program is modified from Facebook darkforestGo which uses synchronized DCNN when expanding in the tree[1], and the simulation is pattern base which is 1000 times faster than DCNN forwarding. All the setting remains default. For direction connection: In GPU. a Torch forwarding server is replaced by the cuDNN library; In CPU, the cuDNN library is replaced with the AVX-512 or IMCI instruction engine. In DCNN simulation, the playout is replaced with previous GPU and CPU engines. More detail is described in each subsection. First, presenting synchronized DCNN with MCTS make a strong go program. Second, showing the method provides good performance improvement. Third, comparing the scalability with the previous method and the providing method. Forth, describing the speed gains on CPU and GPU in different network settings. Finally, presenting that it is practical to use DCNN forwarding on MCTS simulation.

4.1 Synchronized DCNN with MCTS

The darkforestGo from Facebook with synchronized DCNN whose tree is expanded after getting the result from DCNN evaluation is tested against Pachi. The simulation number of Pachi(11.00) is fixed as 400000(about Elo 1298(KGS 2D)) without using DCNN and loading any pattern. The pattern is not used because huge number of random simulations sometimes benefits for Semeai. It beats the darkforestGo even with DCNN rollout equal to 40000 by killing it. Table 5 shows that the winning rate increases significantly when the number of DCNN rollout is big enough. However, the rank is stopped around KGS 6D. The number of 20000 simulations is that the server with 8 GPUs can play when each move is constrained around 6 to 7 seconds. When increasing the rollout number to 40000, the Elo does not improve.

	Pachi	Elo
Rollout = 1	44.75% ± 2.49%	1261(KGS 2D)
Rollout = 256	85.75% ± 1.75%	1610
Rollout = 1024	89.5% ± 1.53%	1670
Rollout = 20000	97.5% ± 0.78%	1934(KGS 6D)
Rollout = 40000	96.75% ± 0.9%	1888(KGS 6D)

Table 5 Results (Winning rates) of various DCNN rollout against Pachi

4.2 Direct connection and a single GPU with multiple threads

The test system is dual CPU E5-2675 v3 at 1.8GHz with 1 Tesla P100 and 9 GTX 1070. The operating system is Cent OS 7.3 with default g++ compiler and the GPU driver uses Cuda 8.0 and cuDNN 5.1.10. Table 6 compares the time in seconds(averaging first 150 self-playing moves) with the number of 100000 rollout(simulation) for each move on a NVIDIA Tesla P100(The detail is in Table 1). The Network architecture is described in Table 4. For direct connection, the performance increases 2.01, 1.58 and 1.32 times in the network A,B and C. It seems that GPU prefers bigger channel size such as 32 than 24. It might be caused by removing many unnecessary communication overhead.

Network	A	B	C
Via server	1011.5±13.0	997.2±9.2	1046.0±11.6
Directly	**491.6s±2.4**	**631.0±15.3**	**794.2±16.7**

Table 6 Results of 100000 simulation time(seconds)

Then, test scalability in the Tesla P100. The darkforestGo is implemented as MCTS tree parallelization with global atomic lock. Here executing multiple threads running DCNN forwarding simultaneously in the same card is tested. Table 7 describes the performance of scalability at different size of network architectures. The performance is increasing at 3.5 times in the network A with 8 threads, 4.0 times in the network B with 10 threads and 4.6 times with 8 threads in the network C. However, It is bounded at 8 threads in the networks A and B. The result is really good. On the other hand, one GPU can be virtually treated as 4 vGPUs in this case.

Network	A	B	C
Thread = 1	491.6±2.4	631.0±15.3	794.2±16.7
Thread = 2	318.6±4.6	345.6±14.8	349.1±6.4
Thread = 4	182.6±1.0	214.2±2.8	227.9±2.9
Thread = 6	150.9±0.7	208.2±4.0	188.4±1.3
Thread = 8	**140.1±0.1**	171.6±0.8	**170.7±0.8**
Thread = 10	148.7±2.4	**156.4±1.4**	176.3±0.6

Table 7 Time(seconds) of multiple threads in a Tesla P100

4.3 Multiple threads with multiple GPUs

Our method works on multiple GPUs. But, the scalability is bounded by 16 threads. The reason is unknown and it is difficult to debug because of binary-only software. However, with CPU, this is not an issue. Table 8 shows the scalability in the original darkforestGo pipeline forwarding server with Network A. The hardware spec is as the same as previous section, the server has 32 cores(64 hyperthreads). The scalability is quite good. But comparing to our method. One single Tesla P100 is only 3.3 times slower than 10 GPUs. Furthermore, the CPU solutions is really good although the network A and network NB are slightly different.

Network	A	NB
Thread = 1	1011.5±13.0	Google Cloud
Thread = 2	467.2±1.4	Thread = 64
Thread = 4	264.6±1.3	**18.8±0.2**
Thread = 8	123.2±0.3	
Thread = 16	81.6±0.3	i7-7800X
Thread = 32	66.3±0.3	Thread = 6
Thread = 64	**42.7±0.2**	64.2±0.8

Table 8 Time(seconds) of multiple threads in the 10 GPU server

4.4 Single hardware performance on GPU and CPU

The time is measured by averaging first 150 self-playing moves with the number of 100000 rollout(simulation) for each move. The test is stopped when the current test time is not less than the previous test time minus 2 standard error. Otherwise, the thread number is increased one each time. Because the i7-7800X platform does not support Tesla P100, so GTX 1070 GPU is used as comparison. The hardware spec is at Table 1. The operating system is Ubuntu 16.04. The compiler is Intel C++ Compiler 2016 update 4 because Nvidia Nvcc does not support

Intel compiler 2017 yet. The GPU driver is Cuda 8.0 and cuDNN 5.1.5. The Network architecture is described in Table 4. In the network NC setting, Table 9 describes the performance of scalability. The performance is increasing at 8.0 times in CPU with 6 threads and 3.3 times in GPU with 3 threads. The CPU outperforms than GPU in this network architecture. Google Cloud is about 2.6 times faster than i7-7800X. When the model size is small, the hyper-threading works better because the two threads share the same resource.

	i7-7800X	GTX 1070	Google Cloud
Thread = 1	268.1±14.1	384.3±20.7	
Thread = 2	146.2±2.4	166.0±4.9	
Thread = 3	119.0±3.2	**115.5±2.0**	
Thread = 4	83.6±2.1	127.0±0.8	
Thread = 5	53.5±1.0		
Thread = 6	55.2±1.2		
Thread = 7	43.3±0.8		
Thread = 8	44.4±0.7		
Thread = 9	41.4±0.8		
Thread = 10	**33.7±0.4**		Thread = 64
Thread = 11	36.8±0.4		**13.0±0.1**

Table 9 Time(seconds) of multiple threads in the network NC

In the network NB setting, Table 10 describes the scalability. The performance is increasing at 5.3 times in CPU with 6 threads and 4.6 times in GPU with 3 threads. The CPU is faster than GPU in this network architecture. Google Cloud is about 3.3 times faster than i7-7800X. It shows the parallel DCNN forwarding with MCTS can scale up in the CPU architecture because Google Cloud has about 3 times more computation power.

	i7-7800X	GTX 1070	Google Cloud
Thread = 1	342.7±3.2	667.6±5.7	
Thread = 2	166.3±1.4	202.8±4.0	
Thread = 3	115.7±1.3	**146.2±2.5**	
Thread = 4	96.8±1.0	161.2±5.7	
Thread = 5	77.8±1.0	165.3±1.0	
Thread = 6	**64.2±0.8**		Thread = 64
Thread = 7	79.6.3±1.6		**18.8±0.2**

Table 10 Time(seconds) of multiple threads in the network NB

In the network NA setting, Table 11 shows the performance of scalability. The performance is increasing at 5.5 times in CPU with 6 threads and 3.1 times in GPU with 4 threads. The CPU is outperforms than GPU, but the gap is getting smaller. Google Cloud remains about 3.0 times faster than i7-7800X. The network parameter number is 345665 which fits in the cache.

PHI runs on Cent OS 7.3 with Intel C++ Compiler 2017 update 4 due to the compatible issues. Table 12 shows parallel scalability. It scales up well[7] and the speed is faster than GPU. It is possible DCNN forwarding utilizes more VPU than the traditional pattern based method.

	i7-7800X	GTX 1070	Google Cloud
Thread = 1	516.4±4.4	454.0±7.8	
Thread = 2	305.7±2.7	205.2±1.2	
Thread = 3	188.1±1.5	**145.7±0.9**	
Thread = 4	158.2±4.9	171.1±2.7	
Thread = 5	112.9±2.2		
Thread = 6	**93.4±0.8**		Thread = 64
Thread = 7	97.5±0.8		**31.1±0.2**

Table 11 Time(seconds) of multiple threads in the network NA

Network	NB	NC
Thread = 1		6833.4±331.6
Thread = 2		2848.4±57.2
Thread = 4		1461.5±23.3
Thread = 8		1345.2±95.9
Thread = 16	594.7±9.7	553.7±16.5
Thread = 32	398.2±7.6	194.4±5.2
Thread = 56	147.1±1.4	123.2±3.0
Thread = 112	**91.4±0.6**	76.6±1.6
Thread = 168	97.3±1.3	**65.0±1.0**
Thread = 224		76.9±1.8

Table 12 Time(seconds) of multiple threads in PHI

4.5 DCNN forwarding on expanding and simulating with MCTS

Since the speed for DCNN forwarding is so fast with small networks, doing simulation with DCNN inference is practical. For pattern based simulation method, the accuracy is lower comparing the DCNN models. However, it is usually 1000 times faster than DCNN inference. With a smaller network, it is possible to reduce the gap between both. The same network NC is used to do DCNN forwarding when MCTS is expanding and simulating. In addition, a random selection algorithm is used to prevent from states duplication when doing DCNN forwarding. Instead of choosing the move with the highest score in the softmax. Before the 300th move, if the softmax score is bigger or equal to 20%, the legal move with highest score is used. If the softmax score is smaller than 20%, there is a 1% chance that the legal move with second large softmax score is selected. After the 300th move, the legal move with highest score which is not filled its self eye is selected. After the 600th move, force to pass and end the game. Above algorithm prevents from that DCNN forwarding always plays the same action in the same state. The average number of moves for each game is around 420 moves. Table 13 shows the performance of DCNN inference at expanding and simulating. The result is very good. The previous research[3] states approximately 1000 rollout/s in a system consisting of 8x high-end GPUs with Batch size = 64. In the google cloud platform, 30.9 rollout/s is achieved with Batch size = 1. The i7-7800X and PHI can do 9.9 rollout/s and 5.0 rollout/s. But, the network accuracy is 45.8% which is far better than 34.8%. It is tested on GTX 1070 also. With 1 thread, it takes about 520 seconds for 640 rollout. However, it is crashed after first move when executing with 2 threads. The time is measured by averaging first 150 self-playing moves with the number of 640 rollout(simulation) for each move.

Hardware setting	Time(seconds)
PHI(Thread = 112)	128.6±1.8
i7-7800X(Thread = 12)	64.5±0.4
Google Cloud(Thread = 64)	20.7±0.3

Table 13 DCNN forwarding on expanding and simulating in the network NC

5. Conclusions

Sharing a single hardware with multiple threads in deep learning inference is efficiently. It improves overall performance when launching multiple threads concurrently. In the same time constraint, it can have more simulations. In addition, one single GPU or CPU can run many different programs at the same time. This is good at utlizing computing resource efficiently. For example, one single hardware can handle many DCNN go programs playing at the same time and use all resource to the maximum. For small DCNN models, CPU outperforms GPU in our method. Althought the value network[2] is useful for the computer Go, it is very difficult to train and get a good accuracy. Simulation with DCNN forwarding might be a good idea to develop for other applications when it is difficult to get a good value network.

In the future, the relationship between DCNN accuracy and real program strength will be tested. However, it requires some tuning in the MCTS parameters. Also, training a network with FP16 or INT16 is a good topic because computer Go is a good Testbed.

6. Acknowledgement

The authors would like to thank anonymous referees for their valuable comments in improving the overall quality of this paper, and Ministry of Science and Technology of Taiwan for financial support of this research under the contract numbers 105-2218-e-259-001-.

References

[1] Tian, Yuandong and Zhu, Yan, "Better Computer Go Player with Neural Network and Long-term Prediction", ICLR, 2016.

[2] David Silver and Aja Huang and Christopher J. Maddison and Arthur Guez and Laurent Sifre and George van den Driessche and Julian Schrittwieser and Ioannis Antonoglou and Veda Panneershelvam and Marc Lanctot and Sander Dieleman and Dominik Grewe and John Nham and Nal Kalchbrenner and Ilya Sutskever and Timothy Lillicrap and Madeleine Leach and Koray Kavukcuoglu and Thore Graepel and Demis Hassabis, "Mastering the game of Go with deep neural networks and tree search", Nature, 2016, Pages 484-503, Volume 529.

[3] Peter H. Jin and Kurt Keutzer, "Convolutional Monte Carlo Rollouts in Go", CoRR, 2016.

[4] Suyog Gupta and Ankur Agrawal and Kailash Gopalakrishnan and Pritish Narayanan, "Deep Learning with Limited Numerical Precision", CoRR, 2015.

[5] Song Han and Huizi Mao and William J. Dally, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding", CoRR, 2017.

[6] Ching-Nung Lin and Shi-Jim Yen, "Accelerate Deep Learning Inference with MCTS in the game of Go on the Intel Xeon Phi", Game Programming Workshop, 2016.

[7] S. Ali Mirsoleimani and Aske Plaat and H. Jaap van den Herik and Jos Vermaseren, "Scaling Monte Carlo Tree Search on Intel Xeon Phi", CoRR, 2015.