Optimization of a DNN Program on the CPU+MIC Platform

Competition proposal of ASC

2016-09-13

University of Electronic Secience and Technology of China

Abstract

This article is a part of competition proposal of Asia Supercomputer Student Challenge. We analysis the `DNN` program, discuss different design methods, and show optimization methods with their efforts.

Contents

Item	Name	Configuration	Hosts
Server	Inspur NF5280M4 x 4	CPU：Intel Xeon E5-2680v3 x 2, 2.5Ghz, 12 cores	**hostname:** mic1, mic2, mic3, mic4
		Memory: 16G x8, DDR4, 2133Mhz	
		Hard disk: 1T SATA x 1	
		Accelerator card: Intel XEON PHI-31S1P (57 cores, 1.1GHz, 1003GFlops, 8GB GDDR5 Memory)	
Network		Infiniband+Ethernet	

Figure 1. Hardware configuration

Classification	Description	Installation path	Version
OS	GNU/Linux		RHEL 7.1
Compiler	Intel Composer XE Suites	/opt/intel/composer_xe_2015.0.090	2015.0.090
MKL	Intel MKL	/opt/intel/mkl/lib/intel64	
MPI	Intel MPI	/opt/intel/impi/5.0.1.035	5.0.1.035
PBS	Torque	/opt/tsce	3.0.5

Figure 2. Software configuration

1. Introduction

There is a program based on a standalone hybrid CPU+MIC platform called `DNN(deep neural network)` needed to be parallelized for obtain better performance. We could use any parallel programming mode supported by `MIC` to achieve this goal. Here is some detailed information about hardware in Figure 1, software configuration in Figure 2.

After optimization, the final program is tested on one computing server on the CPU+MIC hybrid cluster. Performance analysis in this proposal is based on the results of this test.

2. Analysis of the Serial Program

2.1. Coarse Grain Analysis

At first, we generate a call graph(Figure 3) by using `Google perfools`, a open source performance profiler, to have a glance though it. Every square represents a function, and the bigger square is, the more time corresponding function costs.

Obviously, the hot spot is something about `MKL`. After googling and searching Intel document we know that MKL provides `BLAS routinues`, which includes a serial function named `cblas_?gemm` to compute a product of general matrices.

But giving that MKL function is well-optimized[4], we search for all position where `cblas_*gemm` is called. Results show the usage of `cblas_*gemm` appear

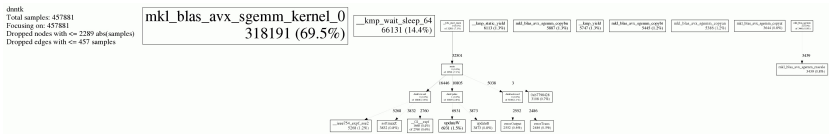


Figure 3. Google Perfools results

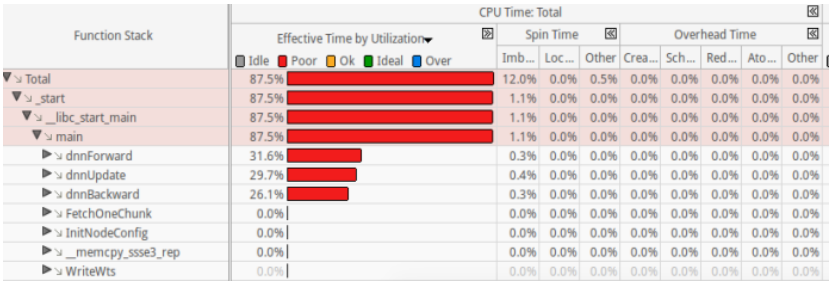


Figure 4. Intel VTune top-down tree

in file `dnn_func.cpp`, more specifically, in three functions:

- `extern "C" int dnnForward(NodeArg &nodeArg)`
- `extern "C" int dnnBackward(NodeArg &nodeArg)`
- `extern "C" int dnnUpdate(NodeArg &nodeArg)`

They call `MKL` function `cblas_sgemm` many times in their `for loop` and cost almost 90% of all CPU time. So we guess that those function is what we may optimize, aka, hotspots. The report(see Figure 4.) showed by`Intel VTune`, another profiler, proves our guess.

According to a skim through the source code, we could establish a clear structure about this program. To simplify code, original program could be rewritten in pseudocode:

```
GetInitFileConfig(cpuArg)
While FetchOneChunk(cpuArg, onChunk) do:
        While FetchOneBunch(oneChunk, nodeArg) do:
             dnnForward(nodeArg)
             dnnBackward(nodeArg)
             dnnUpate(nodeArg)
 WriteWts(nodeArg, cpuArg)
 UninitProgramConfig(cpuArg)
```

There are two nested loop surrounding `dnn*()` series, and in each of those processing function some matrix-matrix product are calculated. Whether those hotspots could be parallelized or not depends on data scale, dependency and so on. Before we discuss some methods and weighed their pros and cons the

implementation of DNN should be carefully checked.

2.2. Fine Grain Analysis

2.2.1. Matrix Size

Here is an example of how DNN use cblas_sgemm:

```
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,\
      numN, numA[i], numA[i-1], \
      one, d_Y[i-1], numA[i-1], d_W[i], numA[i],
      one, d_Y[i], numA[i]);
```

In this way we could calculate d_Y[i-1] * d_W[i] + d_Y[i] and assign the result to d_Y[i]. The arguments numN, numA[i], numA[i-1] indicating the size of the matrices:

- d_Y[i-1] is a numN row by numA[i] column matrix;
- d_W[i] is a numN row by numA[i-1] column matrix;
- d_Y[i] is a numA[i-1] row by numA[i] column matrix.

As we known the bigger matrix size is, the higher degree of MKL parallelism is[5]. But in the DNN program, the size of matrix is decided by bunchSize, a constant integer (≈1024), and a element (≈1024) of dnnLayerArr, which is a constant integer array. The two integers are set by configuration file, and we are not allowed to modify it. For this reason there are no sufficiently large matrix to enable auto offload model of MIC to speed up DNN.[2]

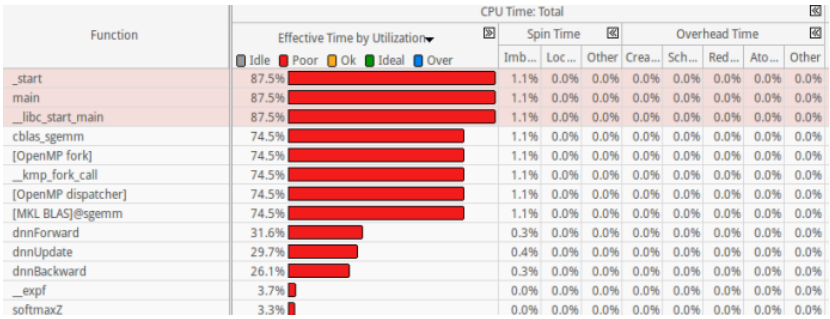
2.2.2. Cycles Index

In the dnn* series every loop call cblas_sgemm numN(≈7) times, which indicates the length of dnnLayerArr. It's regretful that the value also cannot be modified by us. Giving the number of core(≈24 on CPU or ≈60 on MIC) and constant numN, we couldn't improve performance a lot by parallelizing those loops.

3. Parallelization Design Methods

3.1. Fine Grain Parallelism

From the figure above it's easy to find that dnn* series, which invoke cblas_sgemm many times in for loop, nearly cost all CPU time. A rough thoughts occurred to us that we could parallelize those for loop using multi-threads. But data dependency in for loop and relatively small cycle index make it inefficient. So we decide to reduce the size of parallelization region. Taking the feature of MIC in to account we hope to execute highly parallel and compute intensive code on the MIC.[3] Focusing on dnn* series we could find an abstract structure to generalize them:



Function	CPU Time: Total								
	Effective Time by Utilization	Spin Time			Overhead Time				
	Idle Poor Ok Ideal Over	Imb...	Loc...	Other	Crea...	Sch...	Red...	Ato...	Other
_start	87.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
main	87.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
__libc_start_main	87.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
cblas_sgemm	74.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
[OpenMP fork]	74.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
__kmp_fork_call	74.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
[OpenMP dispatcher]	74.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
[MKL BLAS]@sgemm	74.5%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dnnForward	31.6%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dnnUpdate	29.7%	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dnnBackward	26.1%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
__expf	3.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
softmaxZ	3.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Figure 5. Overview of consumed time

```
extern "C" int dnn***(NodeArg &nodeArg)
{
   /* Variables definition */
   float  *d_X = nodeArg.d_X;
     ...
    /* Preprcess function func1 */
    func1(...);

  /* a for loop where a function, a MKL \
  and another function are invoked oderly */
  for (int i = num; ...) {
        func2(...);
        cblas_sgemm(CblasRowMajor, CblasNoTrans, ...);
        func3(...);
        }
    }
    return 0;
}
```

Data processed by `func1`, `func2`, `func3` are matrices or vectors, which are easy to parallelized. Taking what we discuss above into consideration, we have two choose: a) use serial `MKL` then parallelize the whole `for loop`, b) use multi-thread `MKL` and parallelize `func1`, `func 2` and `func 3`. The two measures all support `MIC`, but we prefer the b) because we could benefit from the parallel optimization of `MKL` and cooperation among `MKL` and `MIC`.[2, 6]

3.2. Coarse Grain Parallelism

To implenment coarse grain parallelism we hope that each thread/process finish large subcomponents. To achieve this goal `DNN` program should be divided into (mostly) independent and similar proportions, and every proportion should be as large as possible. But default ordering of `dnn*` series couldn't be changed,

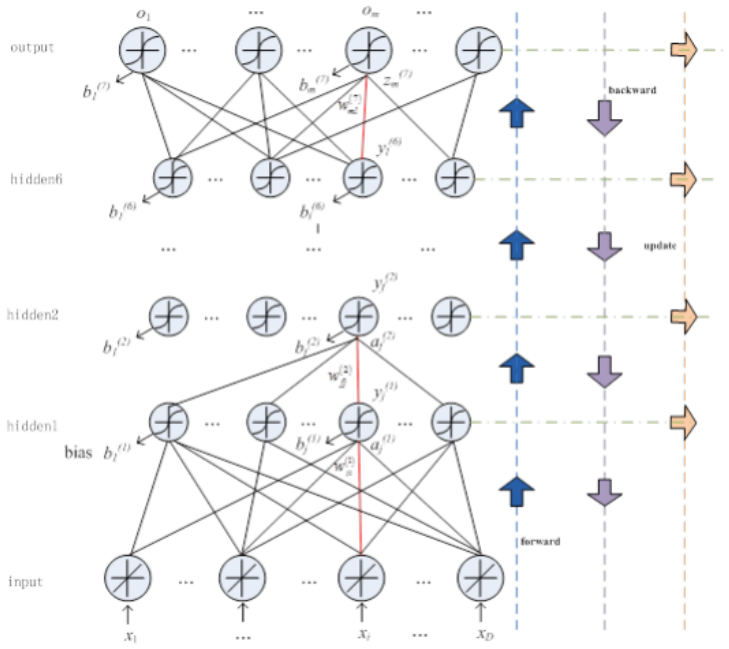


Figure 6. DNN structure

neither does the processing of file-reading. So in our opinion it's difficult to implement coarse grain parallelism without any change to DNN structure and its dataset. So we don't plan to implement it.

4. Performance Optimization Methods

4.1. Compiler Assisted Offload

4.1.1. Scope of Offloaded Sections

We have talked about the common structure of DNN series. In their for loop there are processing function and MKL function appear alternately, and the former modifies data which is used as input in cblas_sgemmfunction. Data change between MIC and CPU is slow because of the limitation of PCI-E so the best known methods[3] is to make the whole section of code an offload unit, which could reduce most of data transfer. In addition, if all three dnn* function are offload units, data transfer between them is less. Then all we only need to transfer input data to MIC before loop and fetch results. This is pseudocode of dnn* function (issues about array transfer discussed in 'Dimensionality Reduction'):

```
extern "C" int dnn***(NodeArg &nodeArg)
{
  /* Variables definition */
  float  *d_X = nodeArg.d_X;
    ...

  #pragma offload target(mic)\
  in(pd_Y:length(0) REUSE)\
  in(pd_W:length(0) REUSE)\
  ...
  {
    /* Preprcess function func1 */
    func1(...);

    /* a for loop where a function, a MKL \
    and another function are invoked oderly */

    for (int i = num; ...) {
        func2(...);
        cblas_sgemm(CblasRowMajor, CblasNoTrans, ...);
        func3(...);
    }
  }
  return 0;
}
```

4.1.2. Dimensionality Reduction

As we know, MIC disallow specifying array of pointer, aka, multi-dimensional array, to be used in `in` or `out` clauses. To cap it all, in this DNN program the real data in two dimensional arrays aren't stored consecutively in memory – every pointer points to a one dimensional array, which stores a matrix. So in order to allocate contiguous block of memory we should calculate offset of element of new one dimensional array before allocating. Here is an example:

```
W_len = 0;
/* calculate element size */
for (int i = 1; i < dnnLayerNum; i++) {
    W_pos[i - 1] = W_len;
    W_len += /* i-st element size */;
}
/* allocate memory */
pd_W = (float *)mkl_malloc(W_len * sizeof(float), 64);
/* build two-dimension array */
for (int i = 0; i < dnnLayerNum - 1; i++) {
    d_W[i] = pd_W + W_pos[i];
}
```

4.2. OpenMP Optimization

Though a function from the MKL could perform runtime check to choose a code path to maximize use of parallelism[1, 5], performance of other functions still need to improve. To maximize the utilization it's better to parallelize them [3]. In pseudocode we discuss in 'Analysis of the Serial Program', `fun1`, `fun2` and `fun3` are bottleneck of this program. Those function accept one or more matrices as arguments, make calculation then re-assign those matrices. Without data dependency it is easy to parallelize them using OpenMp:

```
extern "C" int errorOutput(float *E, float *Z, ...)
{
   float tmp;
   int idx, j;
  //OpenMp block
  #pragma omp parallel for private(tmp, idx, j)
   for(int i=0; i<row; i++)
   {
        for(j=0; j<col; j++)
        {
             idx = i*col + j;
             tmp = (j == T[i]) ? 1.0f:0.0f;
             E[idx] = Z[idx] - tmp;
        }
```

```
    }
    return 0;
}
```

4.3. Alignment

Alignment is important for MKL function to optimize code path and for compiler to assist vectorization. For example, allocating memory for matrices aligned on 64-byte boundary(or more) allows `cblas_sgemm` to have better performance. Using `mkl_alloc` and `mkl_free` is easy to allocate or deallocate an aligned 2MB(0x200000B) memory buffer :

```
float *d_X = (float *)mkl_malloc(N * sizeof(float), 0x200000);
...
mkl_free(d_x);
```

What's more, align CPU data on a 64B boundary or higher could improve data transfer rate[3]. Align at 2MB for maximum transfer rate, so we use `align modifier` improve rate and help compiler produce vectorized code on MIC:

```
#pragma offload target(mic) in(pd_X: length(X_len) align(0x200000))
...
```

4.4. Environmental Variables Settings

4.4.1. MKL_DYNAMIC=true:

This variable leads to a dynamic reduction of number of OpenMPI threads based on analysis of system workload, so it may reduce possible oversubscription from MKL threading.

4.4.2. MKL_NUM_THREADS=57 OMP_NUM_THREADS=57:

Those variables set maximum value of threads allowed to create. Since parallelization regions of this program run in the MIC which has 57 cores in test platform, we set those variables equal to 57.

4.4.3. MIC_USE_2MB_BUFFERS=64K

2MB pages are needed to improve transfer rate between MIC and CPU.[6]

5. Testing Process and Results on the CPU+MIC Platform

Our final program is tested with the workload2 on CPU+MIC hybrid cluster for many times($\geq$5) and our analysis is based on those results. After averaging the

time our DNN program costs and verifying the correctness of results we tabulate a form. The source code folder dnntk_src and log folder exp are packed in the file DNN.zip.

Methods			
Description	OpenMP Position	Time (ms)	Speedup
CPU	\	1,533,962.250	1.00
MIC	\	7,368,252.500	0.28
OpenMP	CPU	366,702.218	4.18
MIC with OpenMP	MIC	358,021.062	4.28
Final Result[1]	MIC	333,205.375	4.60

References

[1] Konstantin Arturov. *Recommended Settings for Calling Intel MKL Routines from Multi-Threaded Applications*. Intel, https://software.intel.com/en-us/articles/recommended-settings-for-calling-intel-mkl-routines-from-multi-threaded-applications.

[2] Noah Clemons. *Recommendations to Choose the Right MKL Usage Model for Xeon Phi*. Intel, https://software.intel.com/en-us/articles/recommendations-to-choose-the-right-mkl-usage-model-for-xeon-phi. Mar. 2013.

[3] Kevin Davis. *Effective Use of the Intel Compiler's Offload Features*. Intel, https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features.

[4] *Developer Reference for Intel Math Kernel Library, Cblas_sgemm*. Intel, https://software.intel.com/en-us/node/520775.

[5] *Parallelism in the Intel® Math Kernel Library*. Intel, https://software.intel.com/en-us/articles/parallelism-in-the-intel-math-kernel-library.

[6] Zhang Z. *Performance Tips of Using Intel® MKL on Intel® Xeon Phi™ Coprocessor*. Intel, https://software.intel.com/en-us/articles/performance-tips-of-using-intel-mkl-on-intel-xeon-phi-coprocessor.