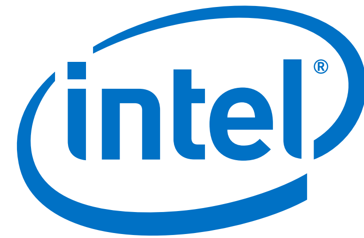

Softmax* Optimizations for Intel® Xeon® Processor-based Platforms

---

Jacek Czaja	Intel Corporation
Michal Gallus	Intel Corporation
Tomasz Patejko	Intel Corporation
Jian Tang	Baidu

Softmax*[5] is popular normalization method used in machine learning. Deep learning solutions like Transformer*[15] or BERT*[4] use the softmax function intensively, so it is worthwhile to optimize its performance. This article presents our methodology of optimization and its results applied to softmax. By presenting this methodology, we hope to increase an interest in deep learning optimizations for CPUs. We believe that the optimization process presented here could be transfered to other deep learning frameworks such as TensorFlow* or PyTorch*.

0.1 Introduction

Softmax is a function used in classification problems in machine learning models such as AlexNet*[10], GoogleNet*[14], or ResNet*[7], where its execution time is small compared to convolution functions. However, recently published natural language processing (NLP) models use softmax more intensively[17], and its high computation cost triggered research towards equivalent but computationally cheaper methods such as hierarchical softmax[11].

This article presents optimizations and performance improvements of the softmax operation for x86-64 architectures (in particular Intel® Xeon® processors). We focused on inference with a deep attention matching (DAM) model[17], and for our experiments we used Baidu's PaddlePaddle* deep learning platform[2]. Using the PaddlePaddle built-in profiler we measured that softmax takes 18 percent of the entire execution time of the DAM model[1]; that is the reason we optimized this functionality. We limited our efforts to single-thread execution, as the usual optimization process starts with exploiting all the capabilities of a single core. Multithread performance improvements are not the topic of the paper and were explored in[9].

The scope of this work[2] includes: algorithmic improvements (reducing general implementation so it is tailored for inference), profiling (identifing the most time consuming fragments of code), using efficient computational libraries (Intel® Math Kernel Library, or Intel® MKL and Intel® Math Kernel Library for Deep Neural Networks, or Intel® MKL-DNN) as well as improving vectorization (analysis of applicability of OpenMP* and manually crafted assembly code for vectorization improvement).

[1] Other time-consuming operators were also optimized, but their analysis is not part of this article

[2] See appendix: Notices and Disclaimers for links to our implementations.

0.1.1 Softmax* theory

Softmax function is an extension of logistic regression that works with multiple classification categories.

$$softmax(z_j) = \frac{e^{z_j}}{\sum_i^N e^{z_i}} \quad (1)$$

where:

z_j = element j of input vector **z**
N = number of elements in a vector **z**

It is often used as a tool to normalize data. Softmax calculates normalized exponential[3] of data, which is often interpreted as a probability vector **z** for classification tasks where values z_j of vector **z** are probability distribution over a set of categories.

0.1.2 Softmax* as implemented in PaddlePaddle*

Our starting point was PaddlePaddle's softmax implementation using the popular Eigen*[6] computation library presented in Figure 1:

PaddlePaddle does offer a functionality to check the time of execution of operators that were executed. The target CPU used for our work to get performance results of softmax execution was the Intel® Xeon® Platinum 8180 processor. We referred to PaddlePaddle profiling to get the performance status of both softmax and the overall DAM model, while optimizing softmax. The exemplary profiler report from PaddlePaddle for DAM model execution is presented at Figure 2 :

```
1   template <typename DeviceContext, typename T>
2   void SoftmaxFunctor<DeviceContext, T>::operator()(const DeviceContext& context,
3                                                     const framework::Tensor* X,
4                                                     framework::Tensor* Y) {
5     auto logits = EigenMatrix<T>::From(*X);
6     auto softmax = EigenMatrix<T>::From(*Y);
7
8     const int kBatchDim = 0;
9     const int kClassDim = 1;
10
11    const int batch_size = logits.dimension(kBatchDim);
12    const int num_classes = logits.dimension(kClassDim);
13
14    Eigen::DSizes<int, 1> along_class(kClassDim);
15    Eigen::DSizes<int, 2> batch_by_one(batch_size, 1);
16    Eigen::DSizes<int, 2> one_by_class(1, num_classes);
17
18    auto shifted_logits = (logits -
19                           logits.maximum(along_class)
20                               .eval()
21                               .reshape(batch_by_one)
22                               .broadcast(one_by_class))
23                               .unaryExpr(ValueClip<T>());
24
25    softmax.device(*context.eigen_device()) = shifted_logits.exp();
26    softmax.device(*context.eigen_device()) = (softmax *
27                                               softmax.sum(along_class)
28                                                   .inverse()
29                                                   .eval()
30                                                   .reshape(batch_by_one)
31                                                   .broadcast(one_by_class));
32  }
```

Figure 1 – Reference implementation (introduced in PaddlePaddle* PR#14337)

```
1   ------------------------>     Profiling Report     <----------------------
2
3   Place: CPU
4   Time unit: ms
5   Sorted by total time in descending order in the same thread
6
7   Event                  Calls    Total     Min.      Max.      Ave.       Ratio.
8   thread0::layer_norm    316000   68958     0.21599   18.1111   0.218222   0.396
9   thread0::softmax       158000   32188.1   0.193633  0.882732  0.203722   0.185
10  thread0::stack         19000    19002.9   0.926812  2.51014   1.00015    0.109
11  thread0::conv3d        2000     16806.6   1.25346   19.4991   8.40328    0.096
12  thread0::mul           317000   13812.1   0.009354  69.8051   0.0435712  0.079
13  ...
```

Figure 2 – Exemplary PaddlePaddle's DAM profiling report

0.2 Optimization process

0.2.1 Algorithmic modifications

By inspecting the implementation code, we can see that there is a functionality of ValueClip (Figure 1, line 23) . When ValueClip is used, softmax does not produce zero values by assigning a very small floating point constant ,e.g., 10^{-60} to a variable that holds zero. This is needed in a situation when there is a logarithmic operation following softmax ,e.g., the cross-entropy loss. A logarithm of 0 is -inf, which will later produce NAN in a training. However, as we are optimizing(speeding up) inference this threshold is not needed as there is no cost function. After removing the mentioned functionality **execution time is reduced by 5%[3].**

[3] See appendix for configuration details

0.2.2 Profiling

Once we analyzed softmax implementation and removed unnecessary elements, we could do profile operations in the softmax operator to find operations that were the most time-consuming. Hotspots were the operations on which we focused our attention because their successful optimization can potentially give the highest performance gains.

We used timestamp counter (TSC), which is a very precise time measuring device for CPUs. Instruction __rdtsc returns the current value of the CPU clock. Our idea was to measure the entire execution time of the operator as well as selected parts of it, so we would know which part took the most time. Having the absolute value when we began modification/optimization let us see whether we were making progress. One note was that profiling is done on more than one execution of function for more reliable results. Figure 3 presents previously introduced softmax implementation (Figure 1), but with added profiling code (lines: 1, 20, 27, 29, 36-44). After execution finished, profiling told us that over 50% of softmax execution time is spent in the exp part of the function, and sum&div took around 30%. Hence, optimization of e^x followed by summing and elementwise division would be our targets.

```
#include <x86intrin.h>
....
template <typename DeviceContext, typename T>
void SoftmaxFunctor<DeviceContext, T>::operator()(const DeviceContext& context,
                                                  const framework::Tensor* X,
                                                  framework::Tensor* Y) {
  auto logits = EigenMatrix<T>::From(*X);
  auto softmax = EigenMatrix<T>::From(*Y);

  const int kBatchDim = 0;
  const int kClassDim = 1;

  const int batch_size = logits.dimension(kBatchDim);
  const int num_classes = logits.dimension(kClassDim);

  Eigen::DSizes<int, 1> along_class(kClassDim);
  Eigen::DSizes<int, 2> batch_by_one(batch_size, 1);
  Eigen::DSizes<int, 2> one_by_class(1, num_classes);

  unsigned long long t0 = __rdtsc();
  auto shifted_logits = (logits -
                         logits.maximum(along_class)
                             .eval()
                             .reshape(batch_by_one)
                             .broadcast(one_by_class));

  unsigned long long t1 = __rdtsc();
  softmax.device(*context.eigen_device()) = shifted_logits.exp();
  unsigned long long t2 = __rdtsc();
  softmax.device(*context.eigen_device()) = (softmax *
                                             softmax.sum(along_class)
                                                 .inverse()
                                                 .eval()
                                                 .reshape(batch_by_one)
                                                 .broadcast(one_by_class));
  unsigned long long t3 = __rdtsc();

  std::cout << "softmax computing time is:" << (t3-t0) << std::endl;
  std::cout << "shifted_logits computing is:" << (t1-t0)/((float)(t3-t0))
       << " of softmax time" << std::endl;
  std::cout << "exp computing is:" << (t2-t1)/((float)(t3-t0))
       << " of softmax time" << std::endl;
  std::cout << "sum&div computing is:" << (t3-t2)/((float)(t3-t0))
       << " of softmax time" << std::endl;
}
```

Figure 3 – Example of __rdtsc based profiling

0.2.3 Performance improvements with Intel® Math Kernel Library (Intel® MKL)

To spare developers the effort of low-level optimizations for the most common mathematical algorithms, a number of libraries have been created that provide optimized implementations of such operations: OpenBLAS*[1], Eigen[6], and Intel MKL. PaddlePaddle baseline code uses Eigen which is a fast and elegant library. We used Intel MKL as it provides implementations optimized for x86-64 architectures (in particular Intel Xeon® processors). We replaced exponential computations and elementwise division with BLAS functions provided by Intel MKL, and the remaining Eigen code was replaced with a hand crafted implementation[see Figure 4]. **Performance improvement was around 2X.**

```
template <typename DeviceContext>
class SoftmaxFunctor<DeviceContext, float, true> {
  void operator()(const DeviceContext& context, const framework::Tensor* X,
                  framework::Tensor* Y) {

    auto in_dims = X->dims();
    auto out_dims = Y->dims();
    const float* in_data = X->data<float>();
    float* out_data = Y->data<float>();
    const int kBatchDim = 0;
    const int kClassDim = 1;
    // 2D data. Batch x C
    const int batch_size = in_dims[kBatchDim];
    const int num_classes = in_dims[kClassDim];
    for (int n=0; n < batch_size; ++n) {
      float max = in_data[n*num_classes];
      for (int c=1; c < num_classes; ++c) {
        max = in_data[n*num_classes + c] > max ? in_data[n*num_classes+c] : max;
      }
      for (int c=0; c < num_classes; ++c) {
        out_data[n*num_classes+c] = in_data[n*num_classes+c] - max;
      }
    }
    vsExp(num_classes*batch_size, out_data, out_data);

    for (int n=0; n < batch_size; ++n) {
      float sum = out_data[n*num_classes];
      for (int c=1; c < num_classes; ++c) {
        sum += out_data[n*num_classes + c];
      }
      cblas_sscal(num_classes, 1.0f/sum, &out_data[n*num_classes], 1);
    }
  }
};
}
```

Figure 4 – Intel MKL based implementation (introduced in PaddlePaddle* PR#14437)

0.2.4 Autovectorization with OpenMP

Our next step was to improve the code that was not replaced with Intel MKL. We optimized the following operations:

- Subtracting value from all elements of vector (Figure 4, lines 20-22)
- Summing up vector elements (Figure 4, lines 28-30)
- Finding the maximal value within elements of vector (Figure 4, lines 17-19)

We took advantage of the OpenMP simd reduction clause[12] that was introduced in OpenMP 4.0 and is available in ICC and GCC (from version 4.9). OpenMP simd is a compiler hint (directive instructing compiler how to process input source code) that can be viewed as a way to provide additional details on an implementation and reduction mechanism so a compiler can more effectively vectorize the code[8].

0.2.4.1 Elementwise subtraction

We inspected the generated assembly of the three operations[4] previously mentioned and found that elementwise subtraction is already vectorized. No further improvements to the subtraction code was needed.

[4] See Appendix for description of methodology used.

0.2.4.2 Summing up elements

We found that OpenMP simd[12] by itself (hints to loops vectorization) did not provide much of a performance boost, as compilers were able to vectorize code without additional information passed via openmp simd. It may result in code size reduction, as the compiler did not have to generate multiple implementations of code when some hints were provided. However, OpenMP simd followed by the reduction clause[12] asks the compiler to change sequential addition to vectorized computation of partial sums, followed by accumulation of partial sums.

Figure 5 presents the original assembly of the summing procedure, as generated by the compiler. It can appear that although the Intel® Advanced Vector Extensions (Intel® AVX) instruction set is used (e.g. vaddss is used) it does not operate on 128/256 bit words; it just sequentially adds 32-bit words.

```
1          BEGIN SEQUENCE SUM! <---
2   # 0 "" 2
3   #NO_APP
4          vxorps  xmm0, xmm0, xmm0
5          lea     eax, [rdx-1]
6          test    edx, edx
7          lea     rax, [rsi+4+rax*4]
8          vmovss  DWORD PTR [rdi], xmm0
9          jle     .L3
10         .p2align 4,,10
11         .p2align 3
12  .L4:
13         vaddss  xmm0, xmm0, DWORD PTR [rsi]
14         add     rsi, 4
15         cmp     rax, rsi
16         vmovss  DWORD PTR [rdi], xmm0
17         jne     .L4
18  .L3:
19  #APP
20  # 36 "/home/jacekc/test-openmp/main.cpp" 1
21         END SEQUENCE SUM! <---
```

Figure 5 – Assembly code of reduction (summing up) of vector

Code in Figure 6 (line 28) contains a modification we introduced. By marking the loop in the line below with pragma omp simd reduction, we gave a hint to the compiler that reduction on variable sum can be safely vectorized, and each partial sum can be computed in parallel using Intel AVX instructions. We inspected the generated assembly (see Figure 10) to check that introduced modification produced the expected vectorization.[5]

[5] Extract of generated assembly shows packed vector AVX instructions that implement sum reduction computation on 128-bit memory chunks

This optimization brought an additional 5% reduction in execution time.

Although there was a performance improvement, we did not use it in PaddlePaddle. Summing e^{z_i} is an operation that sums positive values, and Intel MKL already provides such an operation, cblas_sasum, which sums absolute values of elements. The advantage of using Intel MKL's sasum is that OpenMP pragmas support is present

```
template <typename DeviceContext>
class SoftmaxFunctor<DeviceContext, float, true> {
  void operator()(const DeviceContext& context, const framework::Tensor* X,
                  framework::Tensor* Y) {

    auto in_dims = X->dims();
    auto out_dims = Y->dims();
    const float* in_data = X->data<float>();
    float* out_data = Y->data<float>();
    const int kBatchDim = 0;
    const int kClassDim = 1;
    // 2D data. Batch x C
    const int batch_size = in_dims[kBatchDim];
    const int num_classes = in_dims[kClassDim];
    for (int n=0; n < batch_size; ++n) {
      float max = in_data[n*num_classes];
      for (int c=1; c < num_classes; ++c) {
        max = in_data[n*num_classes + c] > max ? in_data[n*num_classes+c] : max;
      }
      for (int c=0; c < num_classes; ++c) {
        out_data[n*num_classes+c] = in_data[n*num_classes+c] - max;
      }
    }
    vsExp(num_classes*batch_size, out_data, out_data);

    for (int n=0; n < batch_size; ++n) {
      float sum = out_data[n*num_classes];
      #pragma omp simd reduction(+ : sum)
      for (int c=1; c < num_classes; ++c) {
        sum += out_data[n*num_classes + c];
      }
      cblas_sscal(num_classes, 1.0f/sum, &out_data[n*num_classes], 1);
    }
  }
};
}
```

Figure 6 – Intel MKL and openmp simd based implementation

in the recent generation of compilers. In production environments some old compilers like MSVC and GCC 4.8 are still used, so when using Intel® MKL we sped up those older configurations as well. Our OpenMP vectorization effort was not discarded; instead, we upstreamed it into the Intel MKL-DNN project[6] softmax implementation.

[6] MKL-DNN can be built with Intel MKL as well as without.

```
..............
        vmovaps ymm0, YMMWORD PTR [rbp-48]
        xor     r8d, r8d
.L11:
        mov     r9, r8
        add     r8, 1
        sal     r9, 5
        cmp     ecx, r8d
        vaddps  ymm0, ymm0, YMMWORD PTR [rsi+r9]
        ja      .L11
        cmp     eax, edx
        vmovaps YMMWORD PTR [rbp-48], ymm0
.............
```

Figure 7 – Fragment of assembly code of vectorized reduction (summing up) of vector

0.2.4.3 Finding maximal value in an array of elements

We applied openmp simd reduction(max:) for searching maximal value in the softmax operator, but despite asking the compiler for max reduction the generated code was not vectorized [7]. This was because OpenMP simd support varies across compilers (we used GCC 5.4) and not all compilers fully support it. Hence, not having it autovectorized, as suggested by openmp simd reduction max, we implemented max value search directly, using assembly language (see 0.2.5).

[7] We inspected the generated assembly code with the simd max reduction clause applied, and no vectorization was found.

0.2.5 Vectorization with SIMD instructions

As we have shown here, compiler autovectorization capabilities, when used carefully, can bring visible performance improvements. However, that is not always the case because code generated by the compiler's autovectorizer can turn out to be suboptimal, and the only option is to implement an algorithm manually with SIMD instructions (Intel® AVX, Intel® Advanced Vector Extensions 2 -- Intel® AVX2, and Intel® Advanced Vector Extensions 512 -- Intel® AVX512). Performance-critical functionality may benefit significantly when implemented manually in assembly, in particular when vector instructions need to be used.

Assembly implementation of max value search was implemented with the help of Xbyak*[13] project. Xbyak is a JIT assembler that generates assembly code at runtime. JIT functionality suits deep learning use cases very well, as declared models (description of neural network) are usually not modified during their execution (inference, training). Hence, we could generate assembly after the model was defined and

we could have assembly code suited for a neural network model. In particular, we could have different assembly code for different batch sizes.

The manually crafted assembly code for finding maximal value in an array is presented at Figure 12. Due to the introduction of a manually crafted assembler, the max finding function is around 3X faster than our reference code. As the percentage of time spent executing max function is small compared to the computation of exponential function e^x, its performance impact on softmax is small. Softmax, after implementing max finding value in Xbyak, is on average 3% faster. That amount may seem small, but for data centers that are constantly executing deep learning workloads, even a 3% improvement can account for a significant savings in energy and time.

Portability and maintainability problems are the main disadvantages of using assembly language for performance optimizations. Assembly code is not portable among different architectures, and it is more difficult to maintain than the implementations written in higher-level languages. In general, if possible, we recommend using existing softmax implementations like those provided by Intel MKL-DNN, and implementing critical operations with assembly language only when they are not available in Intel MKL-DNN or other fast computational libraries.

0.2.6 Limits of optimizations

When working on optimizing the code we wanted to know whether there was any room for improvement in execution time. We needed a measure of how close the actual performance of softmax came to platform maximal capabilities. There are two limitations to improving performance on any given hardware platform:

- memory-bound limit
- computation-bound limit

Those two limitations and the kernel's operational intensity are the foundation for the Roofline model[16], which is often used for estimation of whether further performance optimizations are possible. The application of Roofline model is out of scope of this document and was not used during the work discussed here. When working on softmax optimization in the context of a DAM model, we experimented by replacing softmax computation with a memory copying routine, memcpy. Memcpy is usually well-optimized (often written manually using vector instructions) so the speed of execution of memcpy is limited by memory throughput. Softmax takes some input buffer and writes its result to output buffer. Both buffers are of the same size; hence, memcpy can be used to replace softmax computation. We knew that comparing both execution times (actual softmax implementation and memcpy) could give us an idea of whether investing more effort in optimizing the softmax implementation. If the softmax execution time were close to memcpy then the algorithm is bound by maximal memory throughput and we wouldn't get better performance in a given execution environment. Using memcpy we initially verified that baseline (not fully vectorized) implementation is far from being memory-bound (see Figure 8). On the basis of that result we concluded that performance could be increased by better utilizing the computing resources of the processor by introducing effective Intel MKL implementations and more effective vectorization.

0.3 Performance evaluation

Figure 8 shows that the softmax execution in the DAM model was 2X faster than the original implementation. This optimization impacted the performance of the entire DAM model and improved it by over 15% (Figure 9).

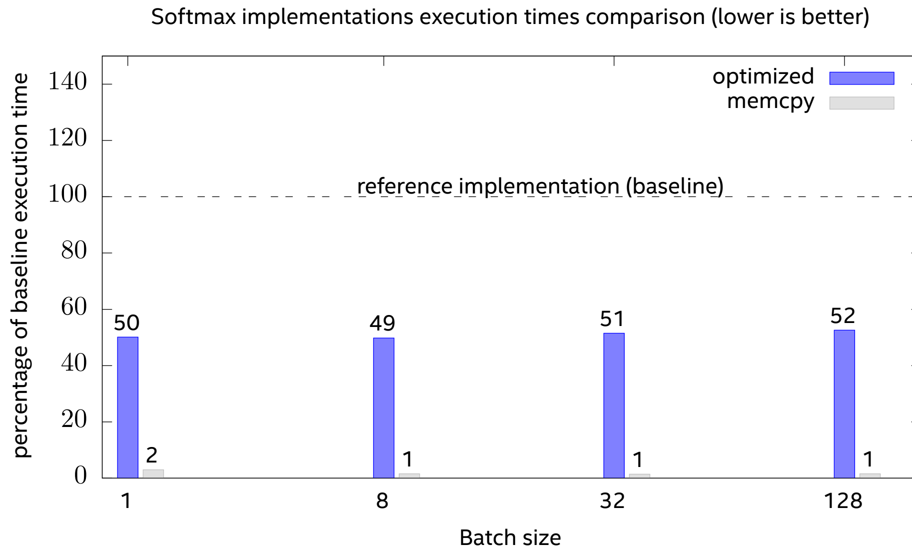

Figure 8 – Comparison of optimized softmax* and memcpy execution times to reference softmax*

0.4 Conclusion and Further work

We presented the methods we used to optimize softmax as well as the performance gained by applying this methodology.

From profiling information, we observed that exponential functions execution take up significant time. In exchange for some slight computational inaccuracy,performance can be improved through computationally cheaper execution approximation functions. Applying a Roofline model to softmax implementations is another way to estimate how much performance can potentially be improved. With optimized implementation far away from memory throughput limitation, it would be beneficial to use the Intel AVX-512 instruction set to manually implement the entire softmax operator. Knowing that softmax is a popular deep learning primitive, we upstreamed[8] our optimizations to the Intel MKL-DNN library.

[8] Sum reduction using both OpenMP vectorization and Intel MKL was merged to Intel MKL-DNN codebase.

0.5 Acknowledgments

The authors would like to express our gratitude to Krzysztof Badziak, Mateusz Ozga, and Evarist Fomenko from Intel Corporation for their advice on optimizations as well as to Andres Rodriguez, Ananth Sankaranarayanan, and Emily Hutson for reviewing this article, and to Luo Tao from Baidu Corporation for her helpful suggestions.

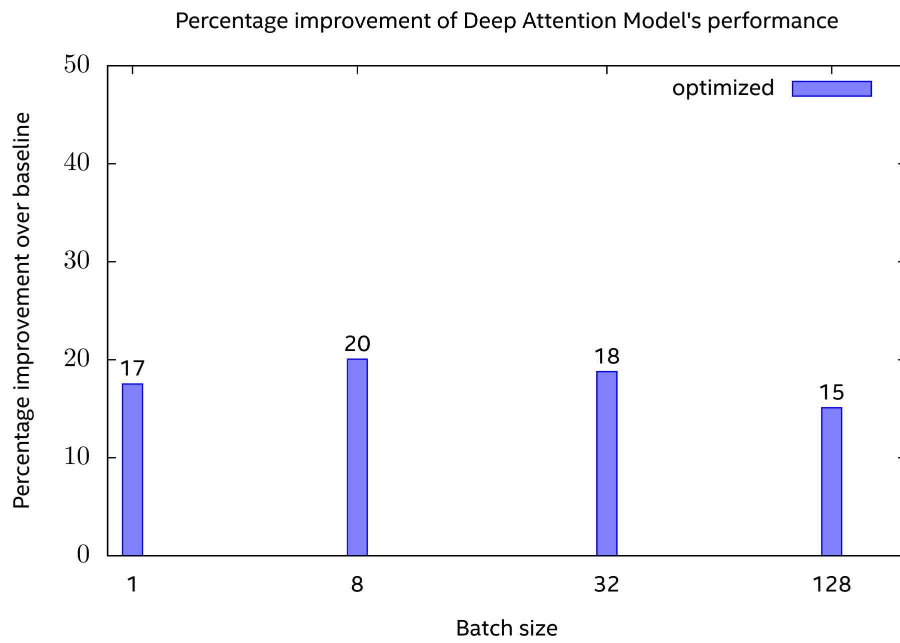

Figure 9 – DAM models overall performance comparison

Bibliography

[1] OpenBLAS. https://www.openblas.net.

[2] Paddlepaddle: An easy-to-use, easy-to-learn deep learning platform. http://www.paddlepaddle.org/.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2009.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[5] Kaibo Duan, S. Sathiya Keerthi, Wei Chu, Shirish Krishnaj Shevade, and Aun Neow Poo. Multi-category classification by soft-max combination of binary classifiers. In *Proceedings of the 4th International Conference on Multiple Classifier Systems*, MCS'03, pages 125--134, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770--778, 2016.

[8] Joseph Huber, Oscar Hernandez, and Graham Lopez. Effective vectorization with OpenMP 4.5. https://info.ornl.gov/sites/publications/files/Pub69214.pdf, 2017.

[9] Vadim Karpusenko, Andres Rodriguez, Jacek Czaja, and Mariusz Moczala. Caffe* optimized for intel® architecture: Applying modern code techniques. https://software.intel.com/en-us/articles/caffe-optimized-for-intel-architecture-applying-modern-code-techniques, 2016.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84--90, May 2017.

[11] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, AISTATS 2005, Bridgetown, Barbados, January 6-8, 2005*, 2005.

[12] OpenMP Architecture Review Board. OpenMP application programming interface 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, 2013.

[13] Mitsunari Shigeo. Xbyak 5.76; jit assembler for x86(ia32), x64(amd64, x86-64) by c++. https://github.com/herumi/xbyak.

[14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998--6008. Curran Associates, Inc., 2017.

[16] Samuel Williams, Lawrence Berkeley, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65--76.

[17] Xiangyang Zhou, Lu Li, Daxiang Dong, Yi Liu, Ying Chen, Wayne Xin Zhao, Dianhai Yu, and Hua Wu. Multi-turn response selection for chatbots with deep attention matching network. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1118--1127. Association for Computational Linguistics, 2018.

Appendix: Inspecting generated assembly code

For the purpose of the work presented here generated assembly code was inspected using compiler switches (relevant to GCC):

```
1 -S -masm=intel
```

and markers were injected into code to easily locate section of code that we are interested in. For example, following C++ code (Figure 10), after applying compiler switches resulted in generated assembly as presented in Figure 11.

```
1  #   ifdef GENERATE_ASSEMBLY
2      asm volatile ("BEGIN␣SIMD␣SOFTMAX␣SUM!␣<---");
3  #   endif
4      float* tmpptr = &out_data[n*num_classes];
5      #pragma omp simd reduction(+: result) aligned(tmpptr)
6      for (int c=0; c < num_classes; ++c) {
7        result += tmpptr[c];
8      }
9      entities[n] = result;
10 #   ifdef GENERATE_ASSEMBLY
11     asm volatile ("END␣SIMD␣SOFTMAX␣SUM!␣<---");
12 #   endif
```

Figure 10 – Assembly code of vectorized reduction (summing up) of vector

Appendix: Notices and Disclaimers

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit:

www.intel.com/benchmarks.

Configurations:

Most of our work was upstreamed into PaddlePaddle and MKL-DNN projects. They can be accessed respectively at:

https://github.com/PaddlePaddle/Paddle[9]

and

https://github.com/intel/mkl-dnn

[9] All quoted Pull Requests within this article, are related to this repository

Optimizations of softmax using direct implementation in assembly language (section 0.2.5) are not part of PaddlePaddle and Intel MKL-DNN repositories. For taking performance measures we created an integration branch located at:

https://github.com/tpatejko/Paddle/commits/tpatejko/jit-max-in-softmax

The Experiments were executed[10] using the following commands:

[10] using commit ID of the integration branch: 28bba75d9108026f236c312813caf5ba72a6aabe

```
OMP_NUM_THREADS=1 ./paddle/fluid/inference/tests/api/test_analyzer_dam \
    --infer_model=third_party/inference_demo/dam/model/ \
    --infer_data=third_party/inference_demo/dam/data.txt \
    --gtest_filter=Analyzer_dam.profile --batch_size=1 \
    --test_all_data=true --num_threads=1 --use_analysis=false --profile
echo "===> Batch 8"
OMP_NUM_THREADS=1 ./paddle/fluid/inference/tests/api/test_analyzer_dam \
    --infer_model=third_party/inference_demo/dam/model/ \
    --infer_data=third_party/inference_demo/dam/data.txt \
    --gtest_filter=Analyzer_dam.profile --batch_size=8 \
    --test_all_data=true --num_threads=1 --use_analysis=false --profile
echo "===> Batch 32"
OMP_NUM_THREADS=1 ./paddle/fluid/inference/tests/api/test_analyzer_dam \
    --infer_model=third_party/inference_demo/dam/model/ \
    --infer_data=third_party/inference_demo/dam/data.txt \
    --gtest_filter=Analyzer_dam.profile --batch_size=32 \
    --test_all_data=true --num_threads=1 --use_analysis=false --profile
echo "===> Batch 128"
OMP_NUM_THREADS=1 ./paddle/fluid/inference/tests/api/test_analyzer_dam \
    --infer_model=third_party/inference_demo/dam/model/ \
    --infer_data=third_party/inference_demo/dam/data.txt \
    --gtest_filter=Analyzer_dam.profile --batch_size=128 \
    --test_all_data=true --num_threads=1 --use_analysis=false --profile
```

All measures and performance evaluation as presented in this article were taken using Intel® Xeon® Platinum 8180 processor.

Performance results are based on testing as of 1st of Febuary 2019 and may not reflect all publicly available security updates. No product or component can be absolutely secure.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice Revision #20110804

Intel®, the Intel® logo, and Intel® Xeon® are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

©Intel Corporation

```
1          BEGIN SIMD SUM! <---
2   # 0 "" 2
3   #NO_APP
4          test    edx, edx
5          mov     DWORD PTR [rdi], 0x00000000
6          mov     QWORD PTR [rbp-48], 0
7          mov     QWORD PTR [rbp-40], 0
8          mov     QWORD PTR [rbp-32], 0
9          mov     QWORD PTR [rbp-24], 0
10         jle     .L9
11         lea     ecx, [rdx-8]
12         shr     ecx, 3
13         add     ecx, 1
14         cmp     edx, 7
15         lea     eax, [0+rcx*8]
16         jle     .L15
17         vmovaps ymm0, YMMWORD PTR [rbp-48]
18         xor     r8d, r8d
19  .L11:
20         mov     r9, r8
21         add     r8, 1
22         sal     r9, 5
23         cmp     ecx, r8d
24         vaddps  ymm0, ymm0, YMMWORD PTR [rsi+r9]
25         ja      .L11
26         cmp     eax, edx
27         vmovaps YMMWORD PTR [rbp-48], ymm0
28         je      .L21
29         vzeroupper
30  .L10:
31         movsx   rcx, eax
32         vmovss  xmm0, DWORD PTR [rsi+rcx*4]
33         lea     ecx, [rax+1]
34         vaddss  xmm0, xmm0, DWORD PTR [rbp-48]
35         cmp     edx, ecx
36         vmovss  DWORD PTR [rbp-48], xmm0
37         jle     .L9
38         movsx   rcx, ecx
39         vaddss  xmm0, xmm0, DWORD PTR [rsi+rcx*4]
40         lea     ecx, [rax+2]
41         cmp     edx, ecx
42         vmovss  DWORD PTR [rbp-48], xmm0
43         jle     .L9
44         movsx   rcx, ecx
45         vaddss  xmm0, xmm0, DWORD PTR [rsi+rcx*4]
46         lea     ecx, [rax+3]
47         cmp     edx, ecx
48         vmovss  DWORD PTR [rbp-48], xmm0
49         jle     .L9
50         movsx   rcx, ecx
51         vaddss  xmm0, xmm0, DWORD PTR [rsi+rcx*4]
52         lea     ecx, [rax+4]
53         cmp     edx, ecx
54         vmovss  DWORD PTR [rbp-48], xmm0
55         jle     .L9
56         movsx   rcx, ecx
57         vaddss  xmm0, xmm0, DWORD PTR [rsi+rcx*4]
58         lea     ecx, [rax+5]
59         cmp     edx, ecx
60         vmovss  DWORD PTR [rbp-48], xmm0
61         jle     .L9
62         movsx   rcx, ecx
63         add     eax, 6
64         vaddss  xmm0, xmm0, DWORD PTR [rsi+rcx*4]
65         cmp     edx, eax
66         vmovss  DWORD PTR [rbp-48], xmm0
67         jle     .L9
68         cdqe
69         vaddss  xmm0, xmm0, DWORD PTR [rsi+rax*4]
70         vmovss  DWORD PTR [rbp-48], xmm0
71  .L9:
72         vxorps  xmm0, xmm0, xmm0
73         vaddss  xmm0, xmm0, DWORD PTR [rbp-48]
74         vaddss  xmm0, xmm0, DWORD PTR [rbp-44]
75         vaddss  xmm0, xmm0, DWORD PTR [rbp-40]
76         vaddss  xmm0, xmm0, DWORD PTR [rbp-36]
77         vaddss  xmm0, xmm0, DWORD PTR [rbp-32]
78         vaddss  xmm0, xmm0, DWORD PTR [rbp-28]
79         vaddss  xmm0, xmm0, DWORD PTR [rbp-24]
80         vaddss  xmm0, xmm0, DWORD PTR [rbp-20]
81         vmovss  DWORD PTR [rdi], xmm0
82  #APP
83  # 52 "/home/jacekc/test-openmp/main.cpp" 1
84         END SIMD SUM! <---
```

Figure 11 – Assembly code of vectorized reduction (summing up) of vector

```
struct maxUFunc : public Xbyak::CodeGenerator {
    maxUFunc()
{
#if defined(__x86_64__)
// calling convention RDI, RSI, RDX, RCX, R8, R9
// XMM0-7 (ints are passed that way)
//      RDI - Reference to Result
//      RSI - PTR to Array
//      RDX - Num classes

// Regsters that need to be preserved: RBX,RBP, R12-R15

  Xbyak::util::Cpu current_cpu;
  if(current_cpu.has(Xbyak::util::Cpu::tAVX2)) {
    printf("AVX2␣supported!\n");
  } else {
    printf("AVX2␣not␣detected!\n");
  }

  mov (rcx,rdx);
  push(rbx);
  shr (rcx,3);  // Divide by 8 (eight floats)
  shl (rdx,2);  // num of Output elements * size of float (4)
  shl (rcx,5);  // Trunc to 32 bytes


        // Compute partial maximums
  vpbroadcastd(ymm0, ptr [rsi]);
  xor(rax,rax);
  L("for_i");
    cmp(rax,rcx);
    jz("tail");
    vmovups(ymm1, ptr [rsi + rax]);  // A
                add(rax,32);
                vmaxps(ymm0,ymm0,ymm1);
    jmp("for_i");
  // Tail execution
  L("tail");
    sub(rdx,rcx);
    cmp(rdx,16);
    jb("seq");
    vmovups(xmm2, ptr [rsi + rax]);  // A
                add(rax,16);
    sub(rdx,16);
                vperm2f128(ymm2,ymm2,ymm2,0);
                vmaxps(ymm0,ymm0,ymm2);  //partial maxes in ymm0
  L("seq");
          cmp(rdx,0);
    jz("done");
                vpbroadcastd(ymm2, ptr [rsi + rax]);
                vmaxps(ymm0,ymm0,ymm2);  //partial maxes in ymm0
    sub(rdx,4);
    add(rax,4);
    jmp("seq");
  L("done");
  // Get within shortlisted buffer maximum
        vperm2f128(ymm1,ymm0,ymm0,1);
  vmaxps(ymm0,ymm0,ymm1);  //partial maxes in ymm0
  vpermilps(xmm1,xmm0,0x1B);
  vmaxps(ymm0,ymm0,ymm1);  //partial maxes in ymm0
  vpermilps(xmm1,xmm0,1);
  vmaxps(ymm0,ymm0,ymm1);  //ymm0[0:31] contains global maximum
  vmovss(ptr[rdi],xmm0);  // Result <-Max(X[.])
  pop(rbx);

  printf("Generating␣Max␣Value␣code\n");
#else
        printf("32bit␣not␣supported\n");
#endif
  ret();
}
};
```

Figure 12 – JIT assembly code of maximal value finding in an array