Optimizing Deep Learning Recommender Systems On CPU Architectures

Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke
Intel Corporation

ISC 2020: Research Poster — June 21 – 25, 2020

What Is DLRM?

Overview of DLRM:

- One of the most important Machine Learning / Deep Learning workload of the Super-7 (Facebook, Google, Microsoft, Amazon, Baidu, Alibaba and Tencent) are recommender systems (RecSys)
- They are used to make suggestions of various kinds to the user of various services offered by the Super-7
- There are various topologies used for RecSys, but the industry does not have a standard benchmark which can be scaled up and down to test and compare platforms
- DLRM [1,3] is an effort started by Facebook to define the "ResNet50" of RecSys
- DLRM is benchmark topology which is inspired by Facebook's production workloads and it can be adjusted for different problem sizes - perfect for benchmarking

The DLRM Topology

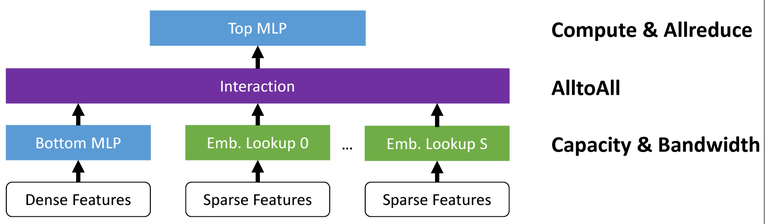

Figure 1: Schematic of the DLRM topology. It comprises of MLPs and Embedding table look-ups and the corresponding interaction operations. Thus, it stresses all important aspects of the underlying hardware platform at the same time: compute capabilities, network bandwidth, memory capacity and memory bandwidth. This is rather unusual for classic HPC applications.

- DLRM has 3 major components: a) collection of dense and sparse features, b) combination/interaction of the processed features c) a classic dense model (neural net)
- Sparse features are processed by a so called embedding table. We can think of an embedding as multi-hot encoded look-ups into an embedding table $W \in \mathcal{R}^{MxE}$, with M denoting the entries and E the length of each entry. This approximates the well-known matrix factorization. This kernel is heavily memory capacity and memory bandwidth bound for random but full cache line accesses.
- The interaction is a relatively simple copy and/or dot-product kernel which stresses the interconnect between cores/sockets and is therefore bandwidth bound.
- The bottom and top Multi-Layer-Perceptrons (MLP) are well-known deep learning constructs: fully-connected layers with ReLU as activation-functions. This kernel is in general compute-bound as it is matrix multiplication heavy.

Unlike classic HPC applications or benchmarks which are either compute-bound (HPL, quantum chemistry, higher0order finite elements) or bandwidth-bound (stencils, sparse matrix vector multiplication, graph analytics), DLRM stress all components of the computing platform in a single application:

- memory capacity
- memory bandwidth
- random cacheline accesses
- interconnect bandwidth
- floating point operation density

Why Are CPUs Preferred For DLRM?

In this poster we focus on CPU-based optimizations for DLRM for the following reasons:

- ✓ CPUs offer very high memory capacity per socket
- ✓ CPUs are a well-balanced platform which is needed for a workload which stresses all aspects of the platform
- ✓ Large SMPs (up to 8 sockets) and a lot of interconnect bandwidth allows efficient scaling
- ✓ Therefore, we can make sure that neither the embedding look-up nor the MLP become a bottleneck

Optimization of DLRM - Software Setup

- **pyTorch:** version 1.15 + Intel pyTorch Extension; compiled using gcc 8.3.0, [5]
- **LIBXSMM:** version 1.14; compiled with gcc version 8.3.0, [2,6]
- **MKL-DNN:** version 1.2; compiled with gcc version 8.3.0, [4]
- **oneCCL:** version 1.0: oneAPI collective communction library

Experimental Setup

All the experiments and measurements are conducted on following hardware platforms using FP32 as numeric format:

- A 32 node cluster, with dual-socket nodes featuring the Intel Xeon Platinum 8280 CPU with 28 cores@2.5GHz. Each socket has its private 100G Intel OPA network card and we use a 2:1 pruned tree as topology.

We use the following benchmark configuration in this work:

Configuration Parameter	Small	Large	MLPerf
Minibatch 1s (N)	2048	-	2048
Global Minibatch (GN)	8192	16384	16384
Local Minibatch (LN)	1024	512	2048
Avg. look-ups per Table (P)	50	100	1
Number of Tables (S)	8	64	26
Embedding Dimension (E)	64	256	128
Avg. #rows per table (M)	$1 \cdot 10^6$	$6 \cdot 10^6$	up to 40M
Length Inputs Bottom MLP	512	2048	13
#Layers Bottom MLP	2	8	3
Bottom MLP Size	512	2048	512-256-182
#Layers Top MLP	4	16	5
Bottom MLP Size	1024	4096	2x1024-512-256-1

Embedding Look-Up Optimizations

In general the sparse reads and writes from and to the embedding tables are random memory accesses. However, at each random position in memory we read several cachelines ($E \cdot$ sizeof(E)). Therefore, we achieve close to stream bandwidth, when applying these optimizations:

- EmbeddingBag forward: By parallelizing and vectorizing the lookup over the offsets we achieve about 8× of vanilla pyTorch
- Sparse EmbeddingBag weight update: We avoid doing sequential coalescing and parallelize this operation by using fine-grained locks, the CPU's restricted transactional memory (rtm) extensions or a race free implementation. This results in a 100× improvement over the baseline implementation.

MLP Optimizations

Compared to the embedding, the MLP is pretty straight forward:

- its main ingredient is matrix multiplication that is optimized by our small GEMM library
- Bias addition and activation function (ReLU) are fused.
- we integrated oneCCL to overlap the AllReduce of weights in the backward pass of the MLP, as the MLP is run in data parallel fashion when using multiple sockets

Single Socket Absolute Performance

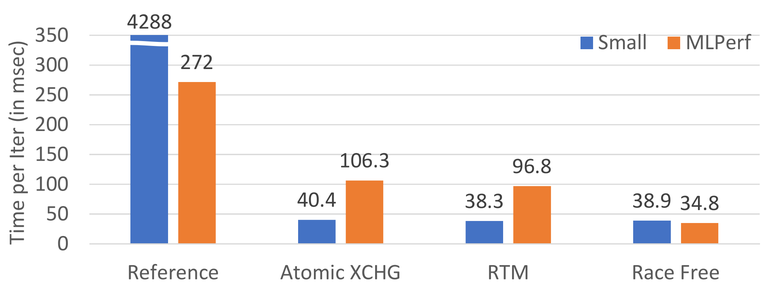

Figure 2: Detailed DLRM performance single socket improvements.

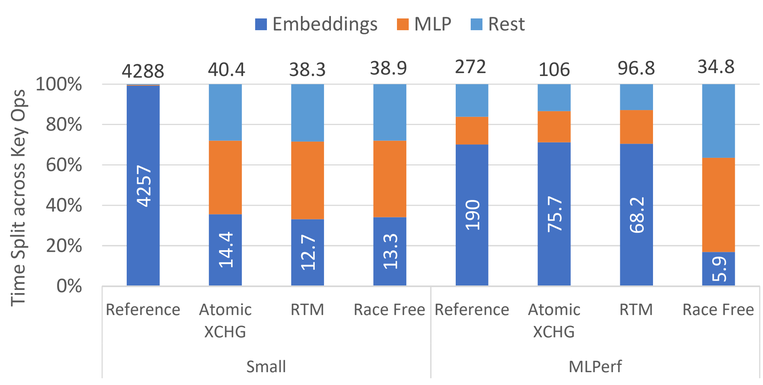

Figure 3: Detailed DLRM performance single socket timing breakdown.

Fig.2 and Fig. 3 depict that we can achieve between 8-110x speed-up over vanilla pyTorch using our pyTorch optimizations. MLP and Embedding phase are then fairly balanced for the small configuration while MLPerf is Embedding-bound.

Single Socket Efficiency – Scaling Baseline

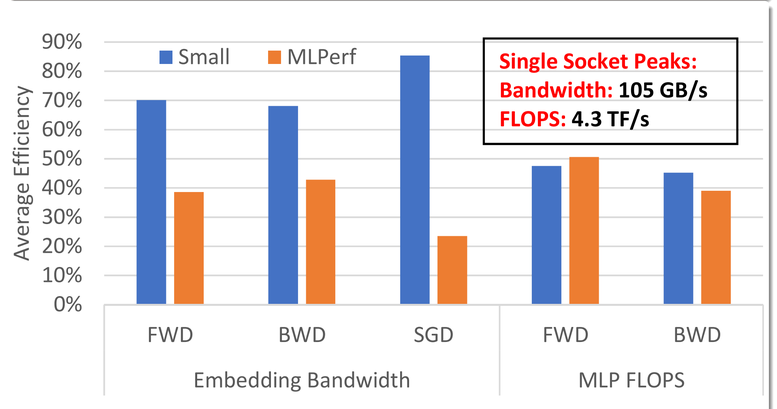

Figure 4: Single Socket FLOPS and Bandwidth Efficiency.

Fig. 4 depicts the hardware utilization:

- we achieve 70+% bandwidth peak and 50 % FLOPS peak for the small config (and therefore for large as well) .
- we achieve 40+% bandwdith peak (lower due to real-life skewed table sizes) and 50 % FLOPS peak for the MLPerf config.

Cluster Scaling Performance - 64 sockets

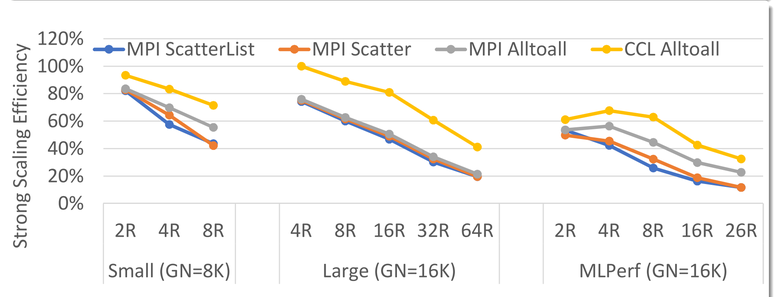

Figure 5: Strong-scaling efficiency of the three DLRM configurations.

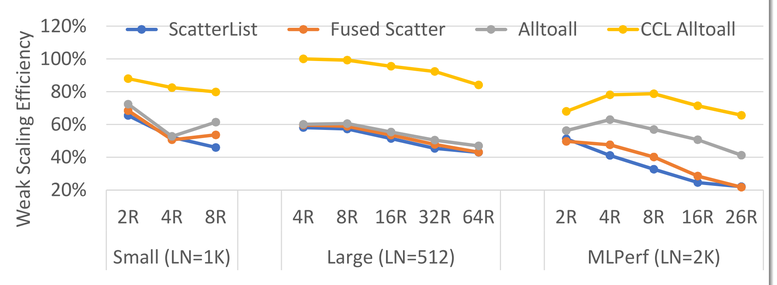

Figure 6: Weak-scaling efficiency of the three DLRM configurations.

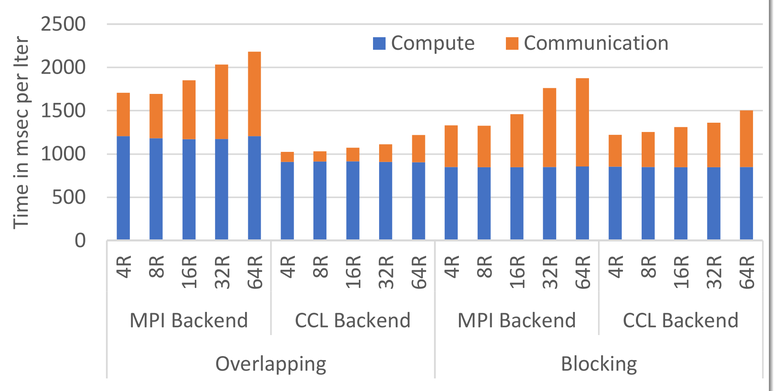

Figure 7: Weak scaling (local and per-socket minibatch is 512) of the large problem on the 64-socket cluster.

Fig. 5, Fig. 6 and Fig. 7 depict the scaling efficiency of Fig. 4 single socket performance on up to 64 sockets (one Intel OPA adapter per socket) using different communication back-ends (classic MPI and Intel oneCCL). We make following observations:

- oneCCL performance is clearly superior due to much better overlapping and therefore less exposed communication time.
- between 40%-60% strong-scaling efficiency when scaling to max. socket count per config.
- excellent weak-scaling efficiency (≈80%) when scaling to max. socket count per config.

Summary, Conclusion & Future Work

On this poster we demonstrated:

- how HPC methodologies can be used to significantly speed-up training different DLRM variants
- how Cascade Lake CPUs can achieve a performance boost of two orders of magnitudes on a single socket using pyTorch
- how DRLM training be efficiently strong- and weak-scaled on a 64-socket cluster using pyTorch

Future work is focused on BFLOAT16 optimizations on Intel Cooper Lake Xeon CPUs

References:

[1] Maxim Naumov, et.al. – Deep Learning Recommendation Model for Personalization and Recommendation Systems, `https://arxiv.org/abs/1906.00091`.
[2] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, Hans Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation, SC 2016: 981–991.
[3] DLRM – An implementation of a deep learning recommendation model (DLRM), `https://github.com/facebookresearch/dlrm`
[4] MKL-DNN – Intel Math Kernel Library for Deep Neural Networks, `https://github.com/intel/mkl-dnn`
[5] pyTorch – An open source machine learning framework `https://pytorch.org/` **and Intel Extension:** `https://github.com/intel/intel-extension-for-pytorch`
[6] LIBXSMM – Library targeting Intel Architecture for specialized dense and sparse matrix operations, and deep learning primitives, `https://github.com/hfp/libxsmm`