EasyChair Preprint

№ 2817

Deep Learning Workload Performance Auto-Optimizer

Connie Y. Miao, Andrew Yang and Michael J. Anderson

February 29, 2020

Deep Learning Workload Performance Auto-Optimizer

Connie Y. Miao[1] Andrew Yang[1] Michael J. Anderson[1]

ABSTRACT

The industry has seen a wave of new domain-specific accelerators purpose-built for deep learning workloads. To obtain real-world performance close to the highest theoretical performance from the accelerators, the tensor layout and workload distribution need to be optimized along with the accelerator instruction set, communication fabric, and memory architecture.

In this paper, we introduce a general methodology for automating hardware architecture and software co-optimization for domain-specific accelerators. Applying this methodology to The Intel® Nervana™ Neural Network Processor for Training (Intel® Nervana™ NNP-T), it has achieved the state-of-the-art (SOTA) deep-learning microbenchmark performance on convolution benchmarks. A generic convolution context distribution algorithm developed based on auto-optimizer results for ResNet50 is also discussed in this paper.

Keywords

Deep learning; hardware-software co-optimization; domain-specific accelerator; locality; parallelism

1. INTRODUCTION

The recent wave of domain-specific accelerators for deep-learning workload has raised the challenge on how to generate instruction code with optimized deep learning performance on a domain-specific accelerator.

General-purpose compilers such as LLVM compile high-level language input programs to a computer device with traditional computer architectures, memory hierarchy and a single CPU operating on scalar or vector values [1, 2]. Without low-level accelerator architecture and instruction set knowledge, a generic compiler is not able to generate executable code with high performance on domain-specific accelerators.

In today's deep learning accelerators or domains such as dense linear algebra, it is still widely accepted and practiced to have handwritten and optimized code surpassing the performance of code output by compilers.

To extract the best deep learning workload performance from accelerators, there are two main aspects to consider when generating the programing codes: 1. Tensor layout and workload distributions, and 2. Accelerator microarchitecture dependent optimization. The first varies with accelerator architecture, affected by locality, parallelism, memory and compute throughput of the device [3, 4]. Different solutions apply for different accelerators. The second aspect is tightly related to accelerator micro-architecture design and fine-tuning the instruction sequence will provide a further performance improvement.

This paper focuses on the first issue which normally provides an order of magnitude performance improvement with a highly optimized solution compared with a naïve implementation. The additional improvement provided by the second aspect normally is less than 1x as it addresses corners cases where the instruction bound occurs.

Automated search guided by performance feedback is a common technique for dealing with complex tuning spaces in several applications. Such techniques have been applied to dense linear algebra [5], sparse linear algebra [6], image processing [7], and neural networks [8,9], to name a few. In this work, we apply these well-known open-source techniques to a novel deep learning training accelerator architecture - Intel Nervana NNP-T, and we derive actionable insights from the search itself to achieve SOTA deep learning microbenchmark performance.

2. AUTO-OPTIMIZER

One of the main operations on deep learning workloads is General Matrix Multiply (GEMM) and convolutions as dense linear algebra calculations. The pseudo-code for GEMM is below:

```
for m, n, k {
        C [m][n] += A [m][k] * B [k][n];
}
```

Where A and B are the two-input matrix and C is the output matrix. All three matrices are two-dimension arrays.

For convolution, here is the pseudo-code:

```
for n, p, q, k, r, s, c {
        ofm [n][p][q][k] += ifm [n][c][p:p+r][q:q+s] * filter [c][r][s][k]
}
```

Ofm is output feature map, and ifm is input feature map. Each matrix has 4 independent dimensions.

Note that the above is one of the tensor layouts for convolution computation. Notice that both GEMM and convolution have tensors with at least 2 dimensions.

The core of the deep learning accelerator are matrix multiply engines that speed up 2D tensor multiplication and accumulations. Accelerators with multiple MM engines, tensor layout in memory, locality and parallelism will have a big impact on the performance. We will elaborate on this using Intel® Nervana™ NNP-T as an example.

[1] Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA. Correspondence to: Connie Y. Miao <connie.y.miao@intel.com>.

Proceedings of the First International Workshop on Benchmarking Machine Learning Workloads on Emerging Hardware. To be held along with the Third Conference on Machine Learning and Systems (MLSys), 2020.

2.1 Intel® Nervana™ NNP-T SOC

Figure 1 shows the NNP-T chip diagram [10]. NNP-T consists of 4 HBM2 2400 with 2.5D packaging technology, providing 1.22 TBps raw bandwidth and a 32GB total device memory. PCIe Gen4 x16 supports communication with the host CPU. It also includes 64 28Gbps SerDes lanes for a chip to chip scale-out communication. The deep learning workload acceleration functionality resides in the 24 tensor processing clusters (TPC), shown in Figure 2. NNP-T can provide up to 119 TOPS compute. TPC consists of four main subsystems. The on-chip router is the green block on the top right, which directly passes data between the TPCs, as well as to and from HBM, PCIe and SerDes. It is a bidirectional 2-D mesh architecture with cut-through forwarding and multicast support. The on-chip router provides a total of 2.6TBps cross-sectional BW and 1.3TBps per-direction. The control block is for instruction decoding, scheduling and retiring as well as managing the dependency for computing and memory. The compute units are the two red blocks, each with 32x32 matrix multiplication arrays, and support for vector and deep learning specific operations. The TPC data pipeline allows compound operations to reduce memory access and increase throughput. Deep learning required operations such as activation functions, random number generator, reductions, and accumulations are supported. The local memory block in blue sources and sinks data to and from the compute units and HBM, as well as allowing sending or receiving data from other TPCs' memory. The total on-chip memory is 60 MB, which is software managed.

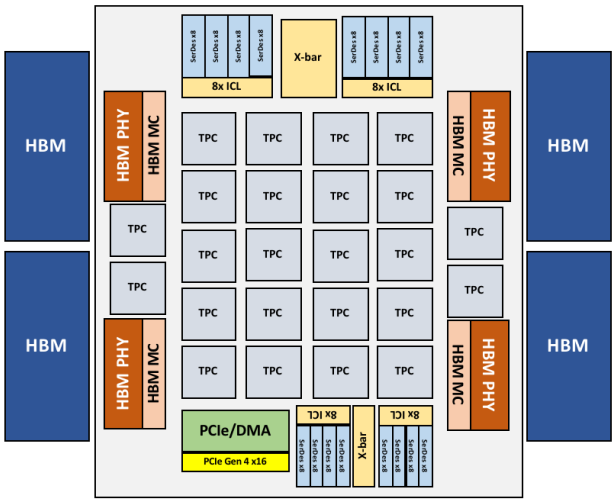

Figure 1. Intel® Nervana™ NNP-T chip diagram.

2.2 Auto-Optimizer Flow

Each GEMM or convolution operation requires a few operations:

1. load data from HBM to on-chip memory
2. send data from on-chip memory to compute unit and save MM data output to on-chip memory
3. optionally store data back to HBM.

To obtain high utilization from GEMM or convolution, an efficient way to layout and distribute the two or more dimensions tensor in HBM and to each TPCs is needed. The best distribution will allow the compute units within each TPC to be evenly fed with data. The industry still sees hand-generated instruction code outperforming the distribution generated by compilers for deep learning workloads. However, the disadvantage of hand-generated code is that it is time consuming to produce and not generic for all input dimensions.

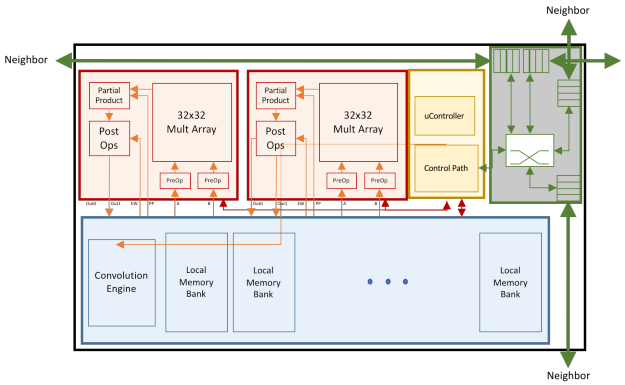

Figure 2. NNP-T Tensor Processing Cluster (TPC).

Figure 3 shows the generic deep learning performance auto-optimizer flow. We approach this problem as a search problem. It consists of two parts: the measurements and search. In the measurement, a GEMM or convolution test with specific set hyper-parameters runs on a hardware or hardware simulator, which generates the user-defined cost function. In our case, the cost function is the workload runtime. The search engine outputs configurations with a set of parameters. In our case, they are the hyper-parameters of workload and hardware supported parameter space. Search techniques are methods for exploring the search space and changing the configurations to the test in the measurements. We employ several different search techniques to explore the large available search space with several different test configurations. Multiple test configurations can be run simultaneously to provide faster measurements and feedback on the distribution strategy. The search and measurement communicate exclusively through a results database used to record all the configurations and cost function results during the optimization process.

For this work, we implemented OPENTURNER [11] framework as our search engine. The measurement components are developed in house. The main development is the hardware simulator used in this work. It is a transaction-level simulator [12], which models the entire NNP-T architecture including computing unit, on-chip router, on-chip memory and controller. It has good accuracy for latencies and bandwidth and can be configured to different network topologies (e.g. 1 ring, or 2 rings) with different number of TPCs. The convolution slice engine is modeled with accuracy on-chip memory capacity and compute unit pipeline latency and throughput. Loading data from HBM to on-chip memory and storing data from on-chip memory back to HBM is also modeled with good accuracy. One of the most important features of the simulator is the capability of launching parallel processes to each cluster and interacting with different subsystems as in the actual hardware. The simulator is capable of producing the runtime of the workload test.

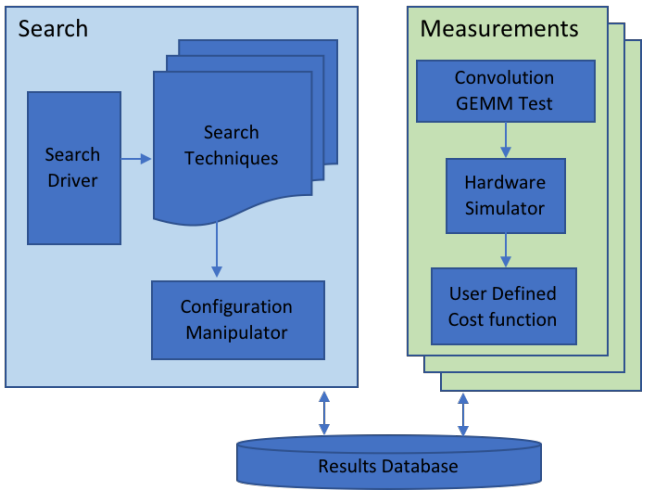

Figure 3. Deep Learning performance auto-optimizer flow.

In the GEMM or convolution tests, test interfaces are developed with hyper-parameters for either network layer descriptors or hardware parameters. The test runtime is defined as the cost function.

The search component is integrated into the simulator environment to allow closed-loop auto-search to find the optimized workload distribution with the shortest runtime.

2.3 Auto-Optimizer Setup for Convolution

The auto-optimizer setup flows for convolution layers are discussed in detail in this section. A few considerations are taken into account before setting up the auto-optimizer. The first one is data locality. Notice that for NNP-T, there are a total of 24 TPCs and 4 HBM to reduce the on-chip network conjunction and latency, as well as power for data movement, 24 TPCs are grouped into four clusters each grouped to the nearest. The data consumed or produced by one of the six TPC in the group is only loaded or stored from the local HBM. In the convolution case, the IFM, and OFM are split up into 4 portions and stored to one of the TPC.

The second consideration is the data layout. The NNP-T matrix multiply array is 32x32. So, it computes matrix dimensions which can be blocked into 32 for each dimension with the highest efficiency. Due to this consideration, the convolution input and filter matrix should be laid out in such a way that it can be blocked into sub-matrices with each dimension being 32. Using Resnet as an example, the IFM Height (H), Width (W) dimension is from 224 to 7, and the input Channel (C) dimension is from 3 to 2048, filter output filter Channel (K) is from 64 to 2048. Two tensor layouts are considered for IFM and OFM: HWCxN and CHWx N with N as the minibatch size. When C is < than 32, CHWxN format is selected, so that the convolution layer context can be split to multiple TPCs with each having a subset of the full layer height (H') and width (W'). The context allocated to each TPC is CH'W'xN. As for layers when C is small, the HxW value is normally larger than 32. Blocking on height and width to distribute to TPCs, the quantization impact to the performance is amortized.

For layer C is >=64, HWCxN data format can be used. Convolution for each OFM pixel is computed as:

```
for n, r, s, c, k {
        ofm [n][k] += ifm [n][p:p+r][q:q+s][c] * filter [r][s][c][k]
}
```

Here, if both C, K and N are integer multiples of 32, the efficiency of matrix multiple array does not suffer from quantization effects.

With these two considerations, we build the measurement for the convolution auto-optimizer with a convolution test, hardware simulator, and runtime output. The test input includes convolution hyper-parameters, H, W, C, R, S, K, P, Q, Sr, and N, where Sr is stride.

The search space is selected with many splits over H, W, C, and K to TPC. To have the best compute efficiency, C or K context split over each TPC should be multiple of 32, so this constraint is added to the search space for C and K split. For H and W context split, the search space is set from 1 to the number of TPCs. Other microarchitecture or ISA constraints can also be added to the search space for effective solution space.

In each iteration, the search engine configures the test input within the search space based on the runtime results. Multiple tests can be launched at the same time to reduce the overall simulation time. As the hardware simulator models compute units, memory access, on-chip router and on-chip memory with good accuracy, the best context distribution represents the overall best scheme considering locality, parallelism, memory and compute throughput.

3. GENERALIZED DISTRIBUTION

For scalable deep learning software, a generalized algorithm is required to support all the input dimensions for convolutions. One contribution to this auto-optimizer is to provide a base for a generalized distribution algorithm. Multiple convolutional layer dimensions are sent to auto-optimize. Based on the best output results, we summarize them into a few lines of codes to be integrated into instruction code generation software. Below is one example pseudo-code of the generalized convolution distribution:

```
For (C, K =< 64):
        Split on W, H
        (W_split * H_split) % active_TPC = 0
elif (C, K == 128):
        Split on C, K, W
        (C_split * K_split * W_split) % active_TPC = 0
        With K context = 64
        Minimize(C_split – 2) to make C is as close to 64 as
else:
        Split on C , K
        (C_split * K_split ) % active_TPC = 0
        K context = 64
        Rest Split from C_split
```

Figure 4 shows the auto-optimizer and deep learning training flow. The auto-optimizer for context distribution with various input dimensions can be run offline and generate the optimized context distribution results. The generalized distribution algorithm can be developed based on the results. The distribution algorithm can be

integrated into the training software stack and generate the optimized distribution at runtime based on input convolution layer dimension size.

For deep learning networks with dynamic shapes for convolution, it is expected this flow is instrumental to obtain the best performance.

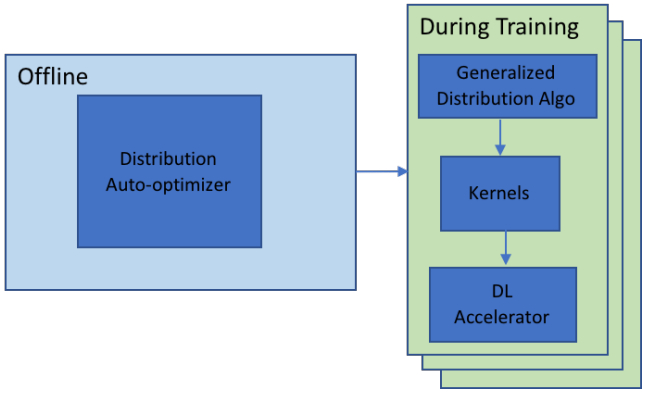

Figure 4. Auto-optimizer and DL training flow.

4. RESULTS

Table 1 lists selected measured results based on this methodology.

Table 1. Measured NNP-T convolution utilization*

Description	NNP-T Utilization
c64xh56xw56_k64xr3xs3_st1_n128	86%
c128xh28xw28_k128xr3xs3_st1_n128	71%
c32xh120xw120_k64xr5xs5_st1_n128	87%

*All products, computer systems, dates, and figures are preliminary based on current expectations and are subject to change without notice.

NNP-T Performance measured on pre-production NNP-T1000 silicon, using 900MHz core clock and 2GHz HBM clock, Host is an Intel® Xeon® Gold 6130T CPU @ 2.10GHz with 64 GB of system memory.

Comparing with the published benchmark from competitive devices, the above utilization represents SOTA [13].

5. SUMMARY

In this paper, a deep learning workload auto-optimizer methodology and application flow are discussed. The detailed implementation of Intel NNP-T has demonstrated SOTA utilization on convolution layers.

It should be noted that with this scalable methodology, it can be used to optimize other deep learning performance matrices, such as power or TOPS/W.

6. ACKNOWLEDGMENTS

Our thanks to Crest architecture and software team for their support during this work.

7. REFERENCES

[1] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 26(10):1367–1372, 2004.

[2] Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. arXiv preprint arXiv:1805.08166, 2018b.

[3] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Notices, 48 (6):519–530, 2013.

[4] Hennessy, J., Patterson, D. A., Computer Architecture – A Quantitative Approach, 6th Edition, 2019

[5] Whaley, R. Clinton, and Jack J. Dongarra. "Automatically tuned linear algebra software." SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing. IEEE, 1998.

[6] Vuduc, Richard, James W. Demmel, and Katherine A. Yelick. "OSKI: A library of automatically tuned sparse matrix kernels." Journal of Physics: Conference Series. Vol. 16. No. 1. IOP Publishing, 2005.

[7] Ragan-Kelley, Jonathan, et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." Acm Sigplan Notices 48.6 (2013): 519-530.

[8] Moreau, Thierry, et al. "VTA: an open hardware-software stack for deep learning." arXiv preprint arXiv:1807.04188 (2018).

[9] Chen, Tianqi, et al. "Learning to optimize tensor programs." Advances in Neural Information Processing Systems. 2018.

[10] Yang, A, Garegrat, N, Miao, C, Vaidyanathan, K, Deep Learning Training at Scale – Spring Crest Deep Learning Accelerator, Hotchips, Palo Alto, 2019

[11] Ansel, J., Kamil, S., Veeramachaneni, K, Ragan-Kelley, J, Bosboom, J, O'Reilly U, Amarasinghe, S, OpenTuner: An Extensible Framework for Program Autotuning, International Conference on Parallel Architecture and Compilation Techniques. Edmonton, Canada. August 2014

[12] Cai, Lukai, and Daniel Gajski. "Transaction level modeling: an overview." Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. ACM, 2003. APA

[13] Baidu. DeepBench: Benchmarking deep learning operations on different hardware, 2017. URL https://github. com/baidu-research/DeepBench