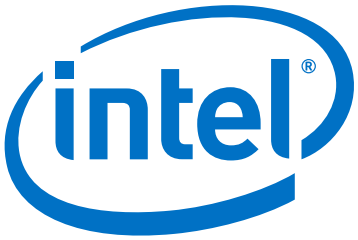


Applying the Roofline Model for Deep Learning performance optimizations

Jacek Czaja	Intel Corporation
Michal Gallus	Intel Corporation
Adam Grygielski	Intel Corporation
Joanna Wozna	Intel Corporation
Tao Luo	Baidu

In this paper We present a methodology for creating Roofline models automatically for Non-Unified Memory Access (NUMA*[8]) using Intel Xeon as an example. Finally, we present an evaluation of highly efficient deep learning primitives as implemented in the Intel (oneDNN) Library.

0.1 Introduction

Deep Learning is widely adopted for tasks, such as computer vision*[6] and natural language processing*[2]. Deep learning tasks often require significant computational resources to operate on large datasets of labeled data. With those requirements in mind, existing hardware platforms are now being optimized for efficient deep learning execution.

One example is the development of the Intel® oneAPI Deep Neural Network (oneDNN) Library, which automatically implements operators, including convolution, matrix multiplication, pooling, batch normalization, activation functions, recurrent neural network (RNN) cells, and long short-term memory (LSTM) cells on x86 architectures, and accelerates inference performance using Intel Deep Learning Boost technology found on Intel Xeon Scalable processors. In this work we evaluated the Intel oneDNN library as on Intel Xeon processors using Roofline models.

The Roofline model is a methodology*[17] for visual representation of platforms that can be used to:

- Estimate boundaries for performance gain from introducing new features e.g. multithreading and vectorization
- Estimate limitations for improvement of a kernel's implementation
- Explain efficiency of an existing kernel
- Compare performance of computing platforms

The Roofline model ties a kernel's representation with platform capabilities (represented by roof), so evaluated kernel maximal performance is bounded by the roof at a corresponding arithmetic intensity of kernel:

$$P = \min \begin{cases} \pi \\ I * \beta \end{cases}$$

A simplified example is presented in Figure 1.

This Roofline model relates the performance of the computer and memory traffic between the caches and DRAM. The model uses arithmetic intensity, (operations per byte of DRAM traffic), defining total bytes transferred to main memory after they have been filtered by the cache hierarchy. Thus, we obtained DRAM bandwidth needed by a kernel, what can discover bottleneck parts on the tested machine. The Roofline model is a 2D graph based on floating-point performance, memory performance and arithmetic intensity.

Initially, measurements needed for constructing the Roofline model were manually calculated. Offenbeck et al.*[13] proposed a methodology for automatically obtaining needed measurements, based on Performance Monitoring units (PMU) of x86_64 based computer architectures. This work is built on the above-mentioned article; we created a program to benchmark computing platforms and evaluate Deep Learning operators using a plot of the Roofline model for each evaluated platform and deep learning operator. We present our methodology on how to draw a Roofline model for Intel

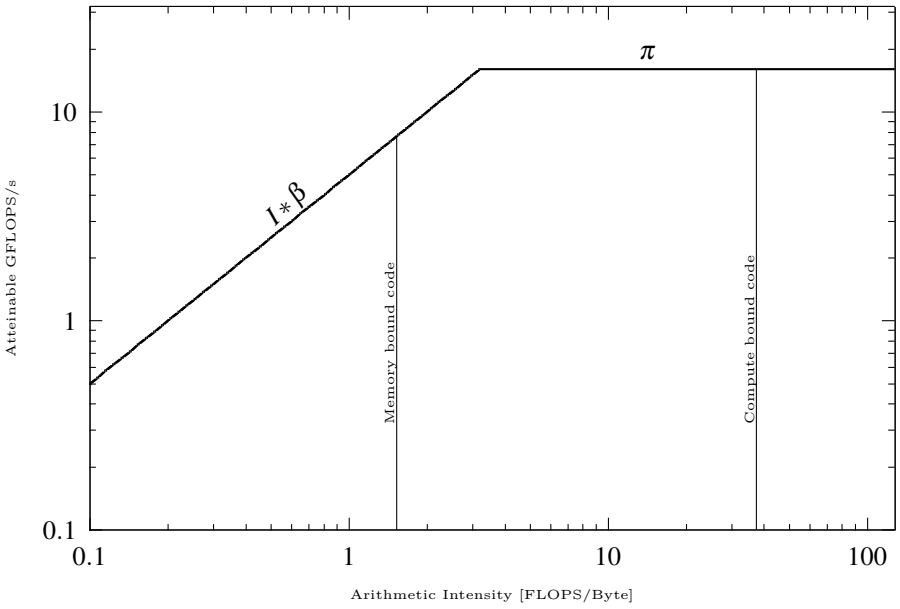


Figure 1 – simplified example of roofline

Xeon processors with limited usage of resources: single core, single socket and two-sockets exection.

We chose to implement benchmarking code on our own both for better control over platform resources and educational purposes. We plotted roofline models for the oneDNN Library's deep learning primitives:

- activation (GELU)
- convolution
- inner product
- layer normalization
- pooling (average)

We then presented our observations from performed experiments.

A conceptual description of the Roofline model is covered in detail in this article*[17]. Practical meaning of roofline is that this shows (depending on the evaluated kernel's arithmetic intensity):

- Attainable compute utilization
- Possible gains from updating the kernel's implementation to use features like: multithreading or vectorization
- Room for improvement of kernel's implementation for the same arithmetic intensity

To plot the Roofline model, we needed to gather characteristics of the computing platform and algorithm implementation (referred to as kernel) executed on that device, namely:

- Peak computational efficiency: π
- Peak memory throughput: T
- Amount of Floating point operations of kernel (Work) : W

- Memory traffic of kernel: Q
- Time of execution of kernel (Runtime): R

In section 0.2, we describe how those attributes were measured in a context of our target CPU, the Intel Xeon Gold 6248 processor.

Once we agreed on methodology, we ran experiments on various oneDNN Library kernels; we present the results and observations of those activities in section 0.3

0.2 Description of methodology

All experiments were conducted on Intel Xeon Scalable processors with Intel Turbo Boost technology disabled, as suggested in the work*[13] that we built upon.

0.2.1 Measuring peak computational performance

We chose to implement our own benchmark for checking peak computational capabilities so that we could better control resource usage for testing (threads, sockets). As well, we wanted our benchmarking to be independent from compiler optimizations, yet achieve maximum performance. Our peak performance checking routine consists of independent execution of runtime-generated assembly code2 on each of the available processor threads. When implementing benchmark it is often a problem that compilers remove dead-/unused code, but when benchmarking code is generated in runtime[1] we do not encounter that problem and overall performance is compiler-agnostic.

[1] using Xbyak*[14] project

```
...
vfmadd132ps zmm0,zmm1,zmm2
vfmadd132ps zmm3,zmm1,zmm2
vfmadd132ps zmm4,zmm1,zmm2
vfmadd132ps zmm5,zmm1,zmm2
vfmadd132ps zmm6,zmm1,zmm2
vfmadd132ps zmm7,zmm1,zmm2
....
```

Figure 2 – peak performance code snippet

Assembly code is a sequence of FMA instructions from Intel Advanced Vector Extensions (AVX, AVX2, AVX512) and designed to avoid chain dependency (read after write) so we can reach close to maximum performance. Using this benchmark we measured peak compute capabilities for the following processor scenarios:

- Single thread execution
- Single socket execution
- Two socket execution

0.2.2 Measuring peak memory throughput

Measuring peak memory bandwidth is complicated, as results may vary depending on the operation we are measuring*[12]. For that reason, we decided to determine maximal throughput value from independent checks:

- memset function from C standard library
- memcpy functional from C standard library

- Hand-crafted memset implemented in assembly language using non-temporal instructions

Both types of benchmark were run single-threaded as well as multi-threaded and were processing .5 Gigabyte of memory. Our own implementation using non-temporal instructions was the fastest method when we ran experiments using for scenarios of single socket and two sockets. On the other hand, memcpy and memset reported higher memory throughput in the single-threaded scenario, which we attribute to the memory prefetching mechanism.

We encountered an issue with our test of memory bandwidth in single-threaded situation as potentialy we could achieve higher bandwidth if we better utilized a memory prefetcher to benchmark memory bandwidth. This problem was present in some of Roofline plots with memory bound kernels. It may be that highly optimized, memory bound kernels executed in a single-threaded environment will have their actual runtime closer to (or beyond) the actual roof than necessary. For single socket or two socket based execution, this is not an important factor as the highest values of memory transferred are obtained using stream instructions (non-temporal stores).

One important thing to mention is that when running a bandwidth check on Intel Xeon processors, we bound memory allocations and threads of single-threaded and single-socket experiments to one chosen socket. It was needed as when full set of threads is used, there was not enough memory bandwidth available on one socket. We observed that threads and memory allocations were migrating to the second socket to take advantage of that socket's memory channels. This is generally efficient-wise practice, it was unwatned behaviour in our experiments as we wanted to limit execution to single socket.

Another important element is that to maximaize throughput when using two sockets (all available sockets of our target platform) we checked memory bandwidth by running two copies of our benchmarking program in parallel. The threads and memory allocation of one running benchmark were bound to one node and second benchmark was bound to the second node. The sum of both throughputs was reported as the peak platform memory throughput. Our justification for this merthod is that when threads are allocated on one node and memory is allocated on another node, it takes more time to access memory than when both resources are alloated on the same node.

0.2.3 Counting Work

Counting FLOPS was done in a similar way as described in this paper*[13]. The below, perf tool was used to read PMU counters:

```
FP_ARITH_INST_RETIRED:SCALAR_SINGLE
FP_ARITH_INST_RETIRED:128B_PACKED_SINGLE
FP_ARITH_INST_RETIRED:256B_PACKED_SINGLE
FP_ARITH_INST_RETIRED:512B_PACKED_SINGLE
```

We used perf externally to count work which made us to conduct two measurements per evaluated kernel:

1. Run our testing program to perform single execution of kernel (overall counted)

2. Run our testing program to initialize all data, but do not perform actual execution (framework overhead counted)

Using PMU counter values from the above runs, we could subtract framework overhead from overall measurement to get the value of counter for actual execution of the kernel. Next, we multiplied the counter value accordingly by 8 (for AVX2) and 16 (for AVX-512) to get actual FLOPS.

During this process, we had a concern if FLOPS were accuratly counted for FMA instructions, since the single FMA instruction for Intel AVX2 is actually performing 16 FLOPS, and for Intel AVX-512 it is performing 32 FLOPS. Therefore, we implemented the assembly code of vfma132ps (FMA instruction) and vfaddps (vector instruction of adding) and observed values of the PMU counter. We discovered that a single retirement of FMA instruction was increasing the counter by a factor of two as opposed to regular vector instructions where the counter was increased by one. This proved that FLOPS are counted precisely. As well, we implemented a more complex assembly code and compared its actuall FLOPS[2] with the FLOPS derived by te PMU counters-based method. Both results matched, so we concluded that this way of counting work is accurate.

[2] Having code implemented in assembly made is easy to count executed FLOPS

0.2.4 Counting memory traffic

Determining memory traffic (Q) was the most challenging element of the Roofline model to produce. Similar to work*[13] we started by counting the memory transfer from last level cache to memory. This approach produced much lower values than expected, due to memory prefetcher mechanisms. Next we disabled the hardware memory prefetcher as described here*[16]. For simple evaluated kernels[3] it provides accurate results, but for more complex algorithms like those implemented in Intel oneDNN Library, results were still much lower than expected. This is because the Intel oneDNN Library implementation is explicitly using software memory prefetcher instructions for GEMM and Winograd implementations which cannot be disabled by the methodology described in *[16]. Hence, we ended up in checking raw memory transfer as it goes through IMC (Internal memory controller) in a similar way as described in*[13] . Since the modern Linux profiler perf was equipped*[3] with support of PMU counters of IMC, we did not have to add PMU counters on our own.

[3] For testing purposes of software solution created at that work we implemented sum reduction kernel

As IMCs' PMUs are counting memory transfer of the whole platform, not only CPU cores where execution of the evaluated algorithm takes place, counted traffic is not just related to the execution of the tested algorithm. Checking IMC uncore counters is available from command-line interface of perf, so to limit the measure of traffic only to the execution of our evaluated kernel, we inspected the source code of the perf tool to get parameters values of syscall communicating perf with Linux kernel. With this knowledge we could call the same syscall in our code.

This method gives satisfying results for processed data greater than a megabyte. Analysis presented in this work is limited to algorithms that process bigger data (throughput) rather than a single chunk of data (latency).

0.2.5 Measuring runtime performance

We measured the time to conduct a number executions and reported an average value as runtime. We were interested in measuring performance in three use cases:

- single-threaded execution
- single-socket execution

- two-sockets execution

We found that it was needed to control threads and memory allocations with numactl utility for the single-socket execution scenario. It proved to be a crucial element, as when all threads from the same socket are heavily accessing memory then there is a shortage in memory bandwidth. The operating system may then migrate threads and allocation into another socket to use some of memory bandwidth of the other socket. This is the same situation as described in section 0.2.2. Not having this restriction (e.g. controlling placement of resources with NUMA tools), will result in a runtime performance that is higher than the actual roof for the analyzed kernel's arithmetic intensity.

0.2.5.1 Cold caches measurements

We decided to clear caches for each iteration before measuring the execution time of the kernel. It was reported[13] to invalidate measures when data size small, but for our experiments the buffer size was quite large [4], so we did not see a problem with unstable measurements. The only problem was that overwriting caches is time consuming, which the running time our experiments.

[4] based on actually used sizes in Deep Learning workloads

0.2.5.2 Warm caches measurements

Before conducting actual measurements, we executed the actual kernel a number oftimes to have caches warmed and then performed the executions to be measured. Modern architectures have advanced memory prefetching mechanisms built-in, so from that point of view the difference between cold and warm caches may not always be noticeable, in particular in some of oneDNN kernels that use software prefetching instructions.

0.3 Analysis of Deep Learning Kernels

0.3.1 Analysis of Convolution

In convolutional neural networks (CNNs), the majority of execution time is often spent in the convolution operation itself. The Intel oneDNN Library provides efficient implementations of convolutions for various x86_64 architectures. Roofline plots were generated by a program created for the purpose of this work. Our target processor for which we ran analysis in this work is the Intel Xeon Gold 6248 CPU. This processor has 44 cores, spread evenly between two sockets and is of NUMA architecture, as access time to the same memory location from each core may differ. We ran analysis for three scenarios:

- single threaded execution (Figure 3)
- one socket execution (Figure 4)
- two socket execution (Figure 5)

0.3.1.1 Single-threaded execution analysis

We started our analysis of the convolution operation using only single-threaded execution. This is an applicable use case for the PaddlePaddle*[1, 11] deep learning framework which is optimized for single-threaded execution. Figure 3 presents roofline plots[5]. We plotted the Roofline model of convolution operations using a fixed size of data to process in three subcases (vertical dashed lines from left to right in the Figure 4):

[5] For the purpose of this article, absolute benchmarked values were turned into relative percentage measure

- Execution of convolution using Winograd*[9] algorithm with cold caches
- Execution of convolution using NCHW data arrangement with cold caches
- Execution of convolution using NCHW16C (blocked) data arrangement with cold caches

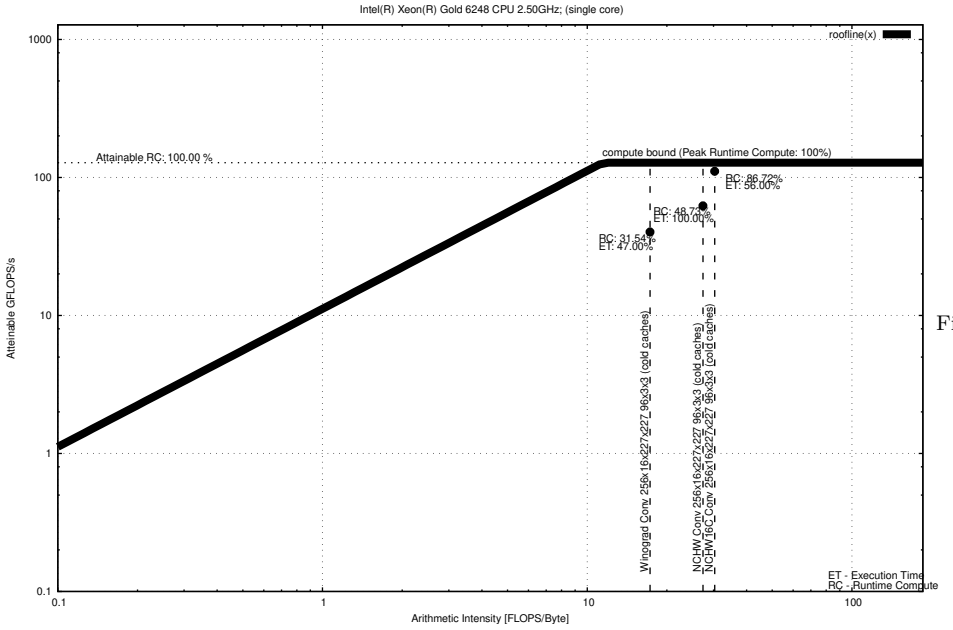


Figure 3 – Single-threaded convolutions

First, we had three different convolutional kernels on the Roofline plot. Apart from the relative utilization of compute capabilities (runtime compute) we also measured relative execution time (ET). NCHW convolution is the slowest so we denoted its ET as 100%. We can see that the NCHW16C convolutional kernel is slightly more efficiently implemented as it utilizes 86% of peak compute, as opposed to the NCHW convolutional kernel which uses only 48% of available computational resources. This is quite intuitive; we compare two different implementations, conceptually the same kind of algorithm is performing same mathematical operations using roughly the same amount of FLOPS. Winograd convolution on the other hand, is a totally different algorithm, which ultimately produces the same results using a different calculation method. Hence, comparing kernels when implementing totally different algorithms has very limited sense. It is more on how well a given kernel will utilize computing platform resources. We can see that Winograd convolution utilization is much lower (31%), yet it is the fastest one among the three presented. What we can see is that the implementation of Winograd has a room for improvement as its runtime compute is far from roof. Although Winograd is the fastest, its applications are limited to specific sizes of convolutional kernels , so direct convolution algorithm is of much wider use.

Next we looked to compare two implementations of direct convolution NCHW versus cache and vectorization-friendly NCHW16C. The Intel oneDNN Library is implementing the idea of layout propagation*[4] in a way that convolutional models input is converted from its original data arrangement to a blocked data arrangement (for example NCHW8C or NCHW16C). Then all subsequent deep learning operations (convolutions, normalization, nonlinearities) work on this data arrangement. Blocked data arrangements

help to ensure that all data used by vector instruction[6] comes from the same single cacheline thus reducing memory latency and helping to achieve higher computational utilization.

We can see that the percentage of total compute utilization is much higher for NCHW16C than for NCHW data arrangement. Most compute friendly scenarios, such as convolution executed using NCHW16C data layout, achieve over 86.0% of maximal FLOPS available on the processor. Such a high compute utilization rate indicates that further optimization of this implementation (without conceptual redesigning or changing the convolutional algorithm) will be difficult. It may be easier to change algorithm to more efficient if one exists. One option may be to replace direct convolution with Winograd*[9] convolution (if applicable) as discussed at the beginning of this section.

[6] AVX,AVX2, AVX512..

0.3.1.2 Single socket execution analysis

In Figure 4, when comparing to single core execution (previous section), we can see that the respective compute resources utilization is slightly lower:

- Winograd convolution: from 31.54% to 29.30%
- Direct NCHW convolution: from 48.73% to 45.68%
- Direct NCHW16C convolution: from 86.72% to 78.01%

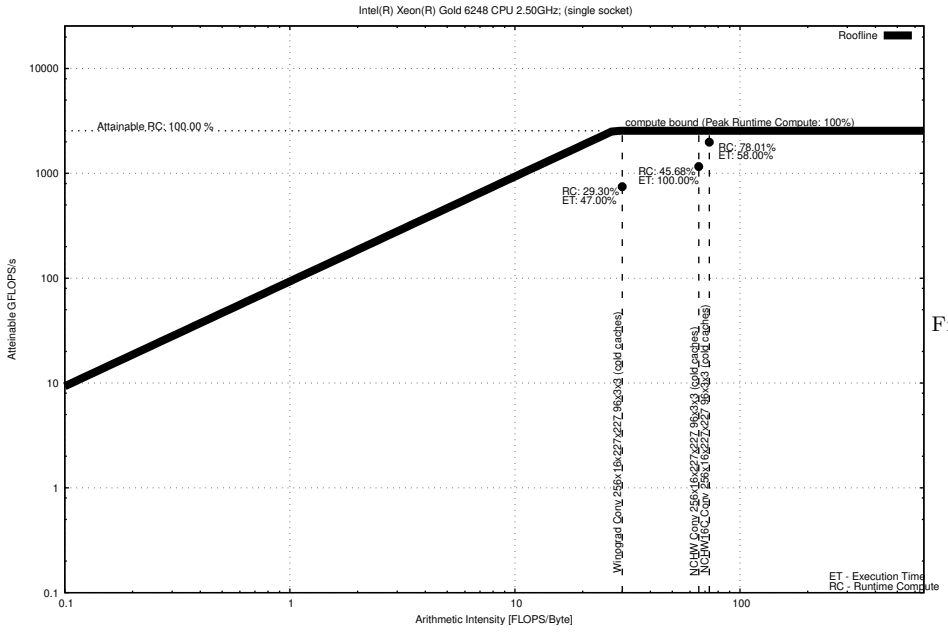


Figure 4 – Convolutions using one socket

We attribute it partially to multi threads handling and partially to memory prefetcher / cache limitations. Without more deeper analysis it is difficult to draw a different conclusion other than that it is easier to implement an efficient single-threaded kernel than a multi-threaded one.

Another observation drawn from the presented Roofline model is that as we migrate execution of evaluated convolutions from a single thread to one socket or to two sockets execution, we can see that less efficient implementations are starting to become memory bound. The explanation for this is not related to the algorithms, it is that the rigid point of the Roofline model was moved further right.This is because memory bandwidth available per

thread when using all hardware threads are available is lower than in the case of single thread execution.

0.3.1.3 Two socket execution analysis

As mentioned earlier, the Intel Xeon Gold 6248 has NUMA architecture. In this experiment we ran analysis on all available computing and memory resources to check utilization and compare it with single socket execution (subsection 0.3.1.2)

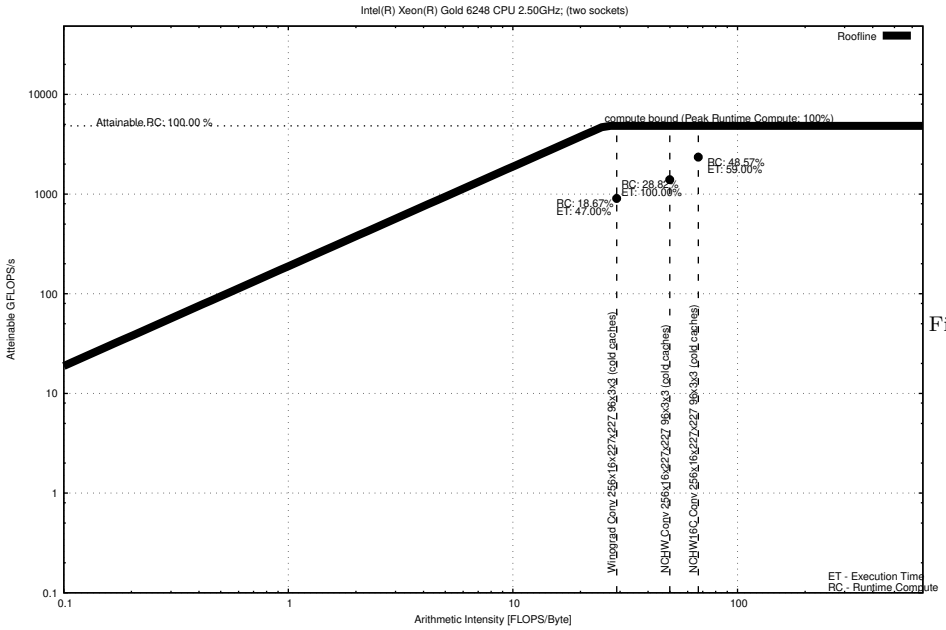


Figure 5 – convolutions using two sockets

Figure 5 presented our results using the full capabilities of the evaluated processor. We can see that the percentage of total compute utilization is relatively lower(48%) to single socket execution (78%) in cache friendly use case (NCHW16C) as well for the other two kernels' executions. We checked that for both execution scenarios the same implementation is being executed, hence we are looking at how well the Intel oneDNN's convolution execution scales from one socket to two sockets. Lower utilization of computing resources in a two-socket scenario is caused by the difficulty in harnessing the full computing resources of NUMA architecture with single kernel execution.

0.3.2 Analysis of Inner Product

In this section we will look at Inner Product which is the base of neural networks . In particular, in modern natural language processing (NLP) solutions like transformer based models *[15], the inner product takes majority of the execution time. The size of processed data of Inner Product as presented (Figure 6) does fit into the L3 cache of processor[7] that was used. Hence it should be possible to observe a difference between execution with cold caches vs execution with caches warmed up.

[7] Intel Xeon 6248

Looking at the Roofline model, we can conclude that in the case of warmed caches, memory traffic is much smaller than when caches are cold. Although we execute the same code so the work is the same, the arithmetic intensity is much higher in case of warmed caches execution, so memory traffic in that case has to be much smaller. Modern processors are using

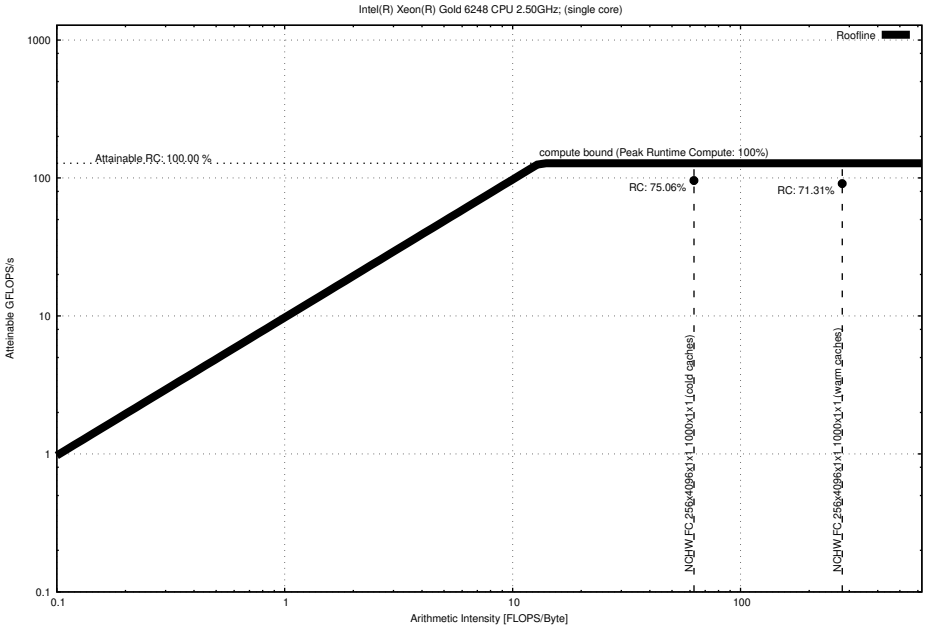


Figure 6 – single-threaded Inner product

a memory prefetcher for reading data, which makes it difficult to predict memory traffic*[10]

The other conclusion we can draw is that the Intel oneDNN Library's inner product is well optimized for that particular shape of input signal as runtime efficiency reaches over 71% of peak computational capacity for what is available on single-threaded execution. Roofline plots for other scenarios (e.g. execution using single socket and two-sockets) are in the appendix.

0.3.3 Analysis of Pooling

We attempted to analyze the pooling primitive using the Roofline model using two most popular pooling algorithms:

- max pooling
- average pooling

For max pooling, the methodology used in this work is not applicable to this operation as max pooling consists of data movement and max operation which are not recognized as FLOPS and not traced by relevant FLOPS PMU counters. Therefore the work value will be counted will not be representative and useful. In this paper, we present only the Roofline plots for average pooling.

Figure 7 shows that arithmetic intensity for NCHW and blocked layout data arrangement (NCHW16C) in a situation with cold caches is almost the same. The same observation applies to the warmed caches scenario. This is not very surprising in itself, but an interesting observation is that there is a huge difference in the percentage of CPU compute utilization. Implementations using NCHW data arrangement achieved 0.35% of compute utilization and NCHW16C implementation are utilizing around 14.8 % which is over 42 x better utilization. We found this interesting and searched for an explanation.

The Intel oneDNN library can work in verbose mode to provide details of internal execution as presented below:

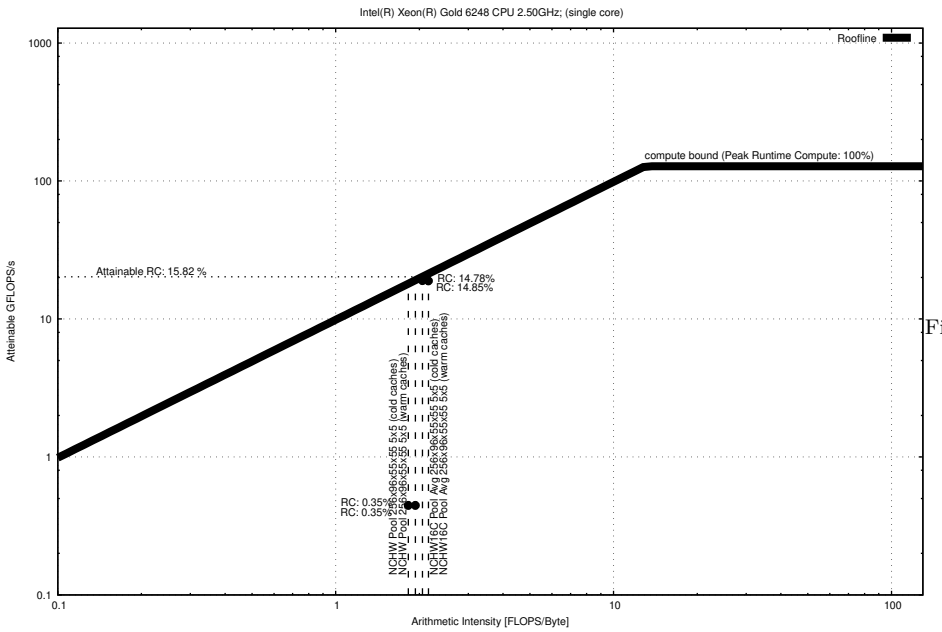


Figure 7 – single-threaded average pooling

- NCHW:

  dnnl_verbose,exec,cpu,pooling,simple_nchw:any,forward_inference,...

- NCHW16C:

  dnnl_verbose,exec,cpu,pooling,jit:avx512_common,forward_inference,...

Based on those outputs we can see that NCHW is using an average pooling implementation named : simple_nchw and the blocked data arrangement is using jit::avx512_common implementation. The former is a C++ based naive implementation and the latter one is a runtime generated assembly code that was implemented using the Xbyak*[14] project. NCHW pooling requires doing operations with-in simd register (as spatial has stride 1), while NHWC and NCHW16C pooling could directly operate on registers. This is the primary reason for NCHW being that low on compute utilization.

0.3.4 Analysis of GELU activation

Another oneDNN primitive we analyzed was recently introduced into oneDNN Gaussian Error Linear Units*[5] (GELU) activation. The reason why we chose to analyze that one is that GELU is an element-wise operation so data arrangement should not have an impact on performance of execution. Moreover activations are of lesser arithmetic intensity compared to convolutions as they are memory bound and we wanted to check if our work is applicable to memory bound primitives as well.

The presented roofline model (Figure 8) shows GELU operation executed via Intel oneDNN library. To our surprise We observed that execution using block data arrangement (NCHW16C) is of lower arithmetic intensity that NCHW implementation. We expected the same performance on both data arrangements. But when looking into actual values of work and traffic we saw that NCHW16C consumed four times as much memory and twice as much FLOPS than NCHW implementation. Our roofline model plots as such does not show W and T values and seeing lower Operation Intensity for NCHW16C made as to check underneath gathered data.

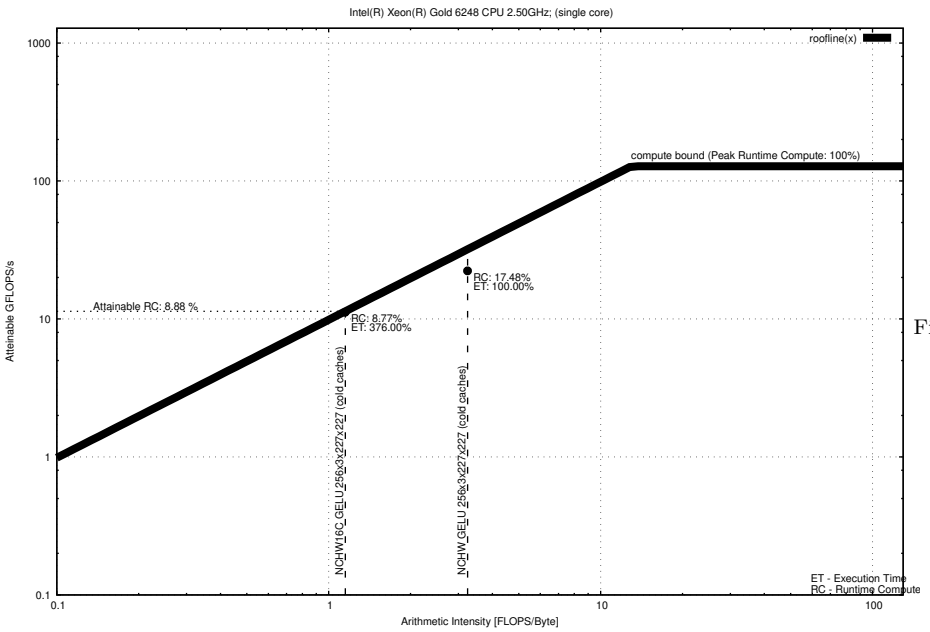


Figure 8 – Single-threaded GELU activations

Explanation of why twice as much of resources is consumed is that dimensionality of input signal [256,3,227,227] is having a second dimension (channel) equal to 3. Efficient implementation of NCHW16C, NCHW8C as provided by oneDNN, require that channel value is multiplication of 8. Hence oneDNN when forced to use blocked data arrangement and as a consequence to extend input tensor to have a shape of [256,8,227,227]. In that situation it is less efficient that using NCHW data arrangement. Does it mean that user has to understand details of implementation of oneDNN kernels to use them efficiently? The answer is No as Intel oneDNN library usage model is that computational primitives are choosing on their own which implementation to use. For the purpose of analysis we chose both unfavorable dimensionality for blocked processing and enforced GELU to work on blocked processing. In other words oneDNN internal logic will trigger most efficient implementation , which at that situation GELU working on blocked layout is not. GELU roofline plots far more often encountered dimensionality (more favorable to oneDNN) as presented in Appendix are confirming that arithmetic intensity for blocked and NCHW data arrangements are very similar as well over all efficiency.

0.3.5 Applicability of methodology

Work(W) is measured via FLOPS PMU events which counts floating point operations like subtractions, additions and multiplications, but it does not count data movements as well as getting maximum values which may be implemented using vmaxps instructions of AVX instruction set. This implies that measuring Work using PMU events (as used throughout this article) is not suitable methodology when to analyze Deep Learning algorithms like Rectified Linear Units (ReLU), Max-pooling and reorders and other primitives where majority of work is performed by operations not considered when counting FLOPS e.g. min, max and data movements.

0.4 Conclusion and Further work

During this work we found Roofline models to be an effective tool to help in understanding a complex deep learning library as Intel oneDNN Library, both from user perspective and oneDNN developer's as well. We expected that this project we developed may be very helpful in validation of oneDNN's kernels

A natural extension of this work would be applying roofline model to integer based computations as it is supported by oneDNN and widely used in modern Deep learning workloads*[7]. It would be also good to evaluate others operation like: max and min and data movements to have a full evaluation of oneDNN's operations. Both mentioned directions depend on the availability of existence of relevant PMU events.

Another direction to consider would be to improve the methodology of creating the Roofline models for single core scenarios. In this work for single core roofline we just benchmarked peak computation and bandwidth as available using single thread. It is fine for peak computation as this will scale with number of cores used. But situation is different for checking bandwidth, due to memory prefetcher working in background. Regardless of the number of cores of single socket used we always have the same number of memory prefetcher streams. So Memory bandwidth will not scale linearly as we increase number of cores used. Perhaps it would make more sense to run benchmarking in parallel for each core independently and report the average value as a bandwidth available for a single core.

Acknowledgment

The authors would like to express our gratitude to Krzysztof Badziak and Mateusz Ozga from Intel Corporation for their advice on optimizations and to Andres Rodriguez, Evarist Fomenko and Emily Hutson for reviewing this article and to Michal Lukaszewski and Michal Chruscinski for providing and preparing platform to run experiments on.

Bibliography

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

[2] 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.

[3] Stephane Eranian. perf/x86/uncore: add support for snb/ivb/hsw integrated memory controller pmu. https://lwn.net/Articles/585372/, 2014.

[4] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on simd architectures, 2018.

[5] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. CoRR, abs/1606.08415, 2016.

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

[7] ZHENLIN L., Ling Yan G., Gomathi R., Kola, Sneha, Pallavi G., Denis Samoilov, Karan Puttannaiah, Rajendrakumar C., Evarist Fomenko, Rajesh P., Niveditha S., and Andres Rodriguez and. Accelerate int8 inference performance for recommender systems with intel® deep learning boost (intel® dl boost). 2019.

[8] Christoph Lameter. An overview of non-uniform memory access. Communications of the ACM, 56:59–54, 09 2013.

[9] Andrew Lavin. Fast algorithms for convolutional neural networks. CoRR, abs/1509.09308, 2015.

[10] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. ACM Transactions on Architecture and Code Optimization - TACO, 9:1–29, 03 2012.

[11] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. Frontiers of Data and Computing, 1(1):105, 2019.

[12] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pages 19–25, December 1995.

[13] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel. Applying the roofline model. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 76–85, March 2014.

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

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. CoRR, abs/1706.03762, 2017.

[16] Vish Viswanathan. Disclosure of hardware prefetcher control on some intel® processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, 2014.

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

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:

Project we developed as part of this is work is currently publicly not-available.
We evaluated oneDNN project version tagged as: GIT_TAG v1.2, publicly available at:
https://github.com/intel/mkl-dnn.

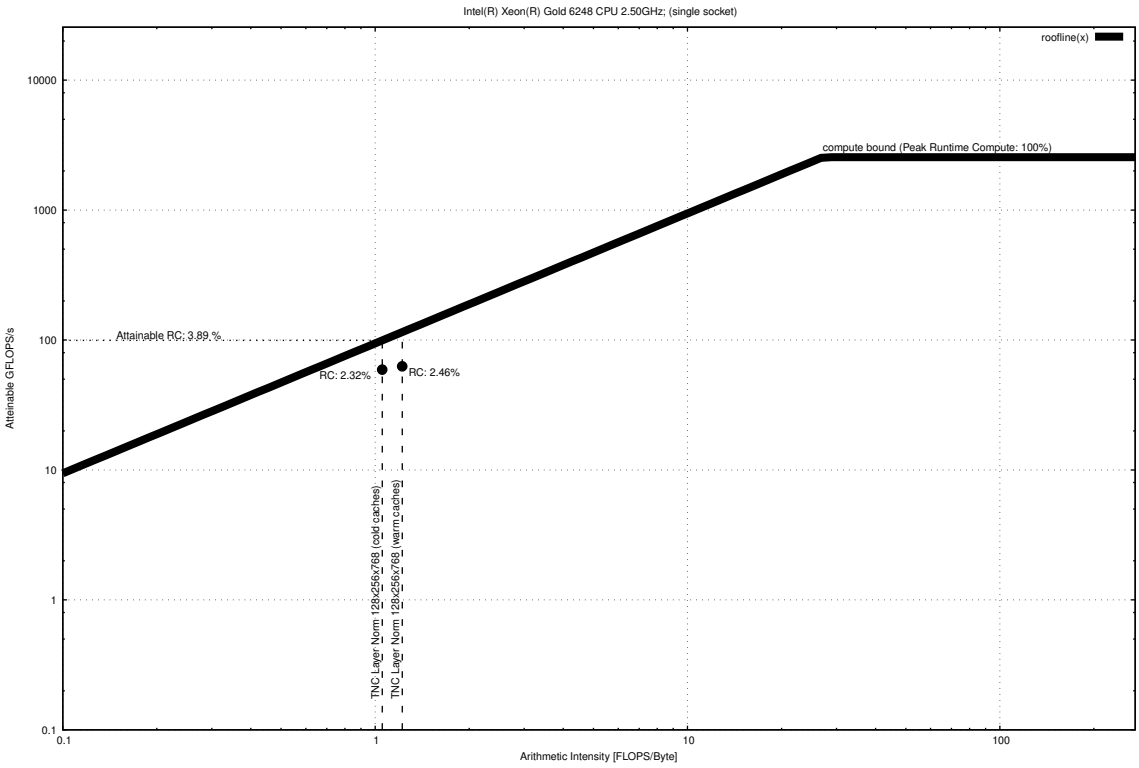
All measures and performance evaluation as presented in this article were taken using Intel Xeon 6248 processor running Ubuntu 18.04 Linux

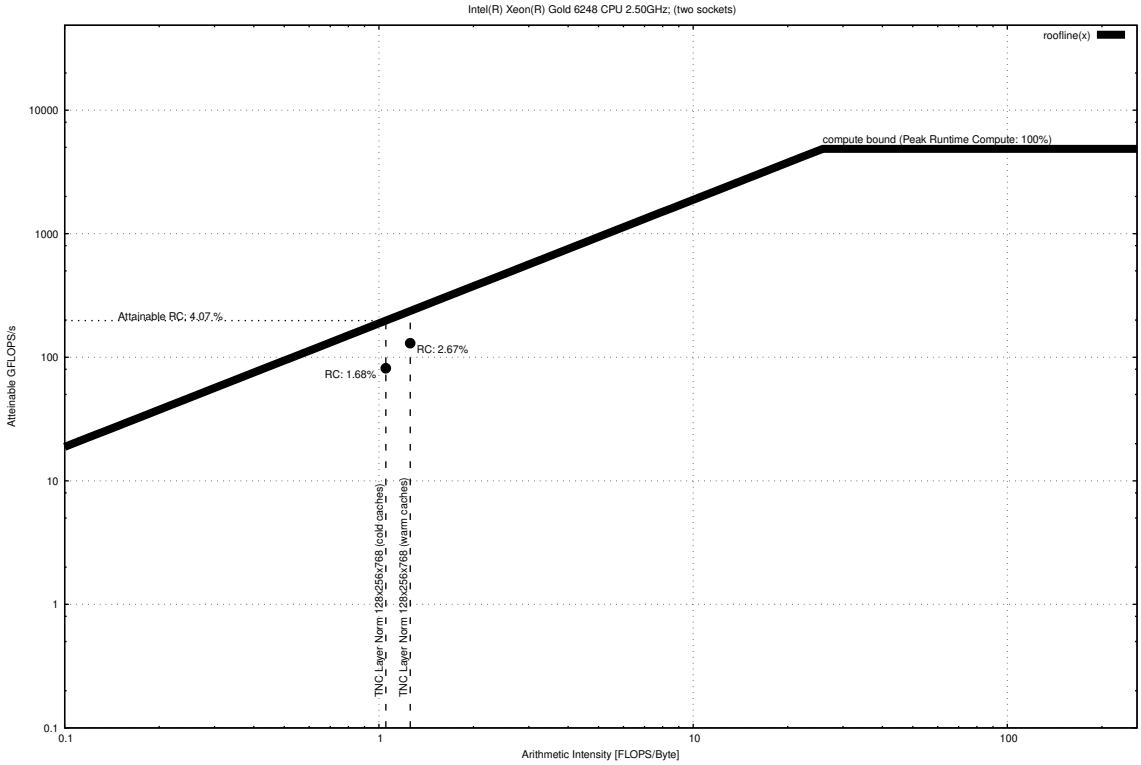
Performance results are based on testing as of 8th of July 2020 and may not reflect all publicly available security updates. No product or component can be absolutely secure.

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.

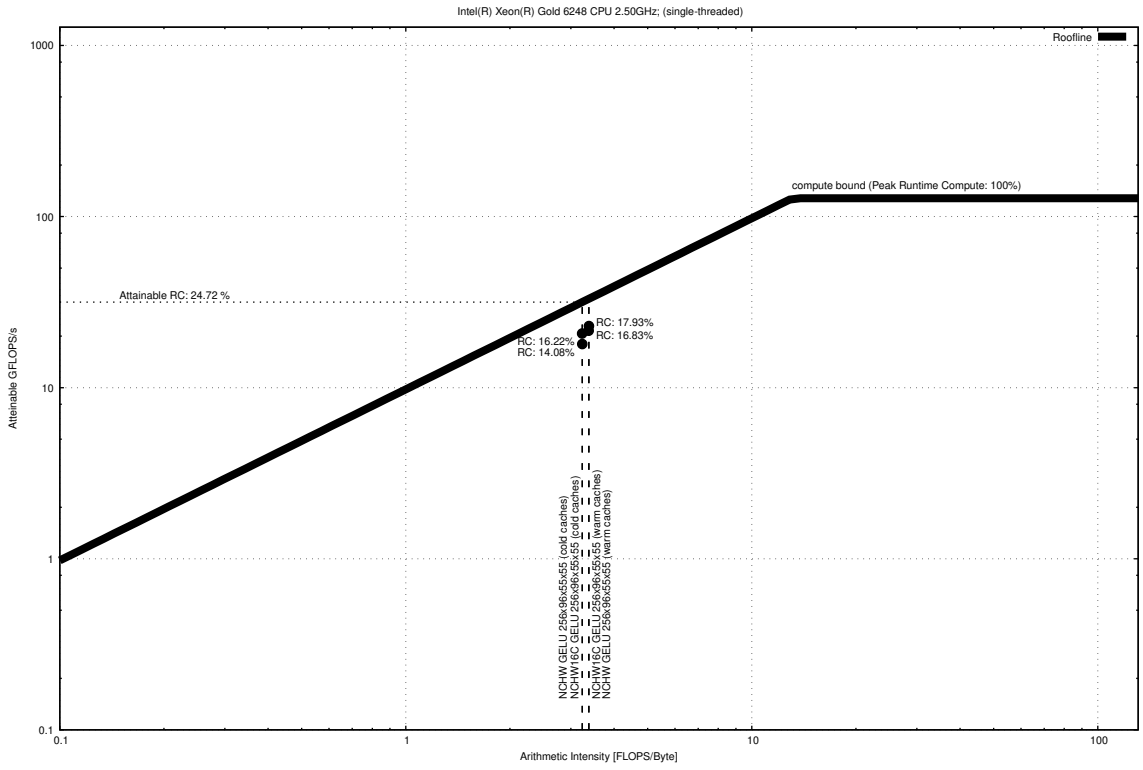
Appendix: Roofline plots of oneDNN kernels

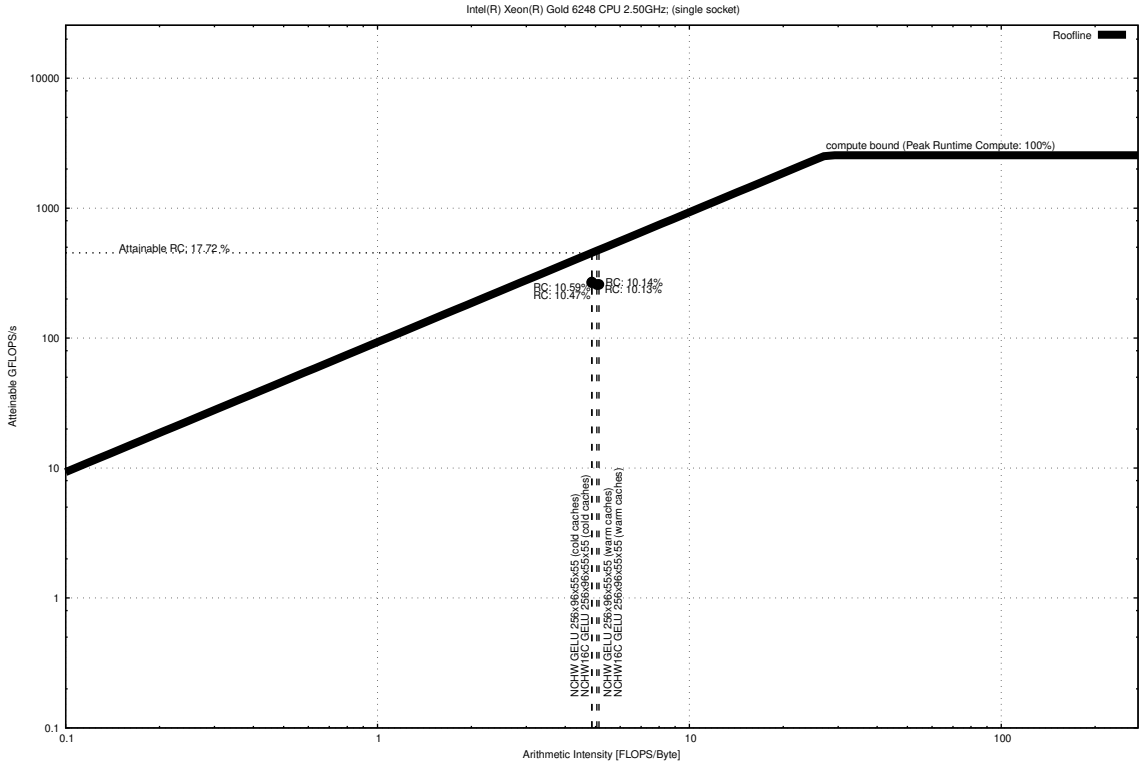
Layer Normalization

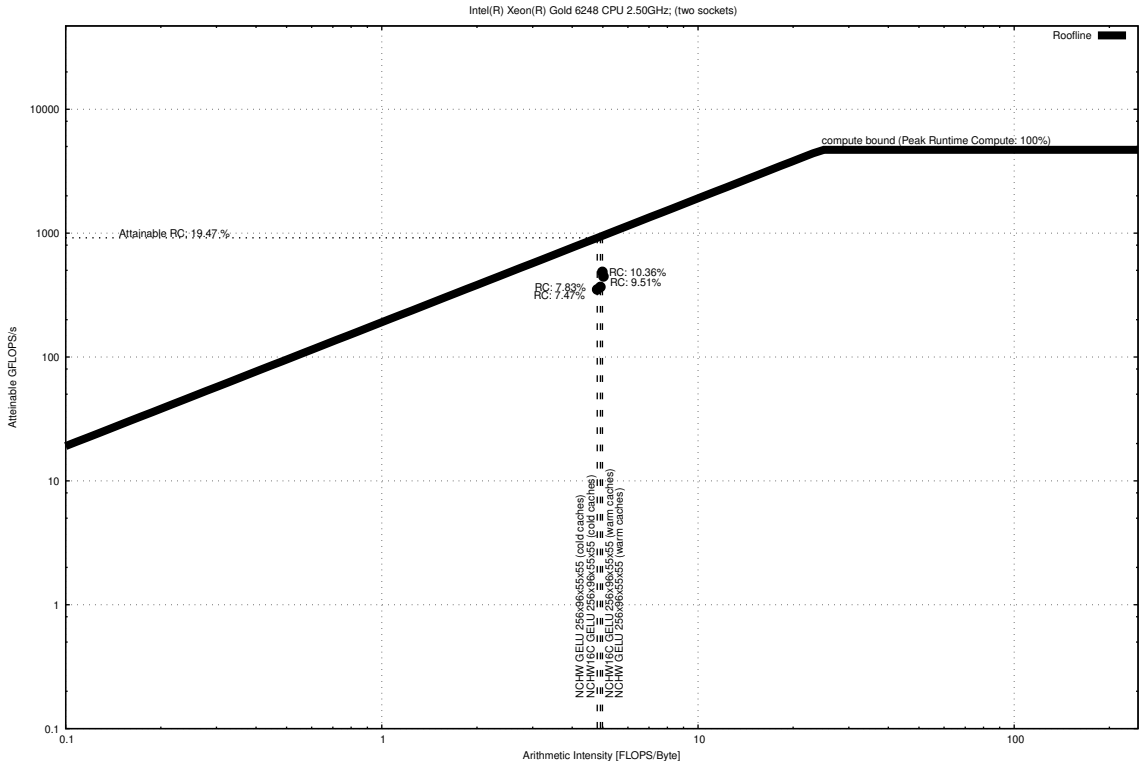




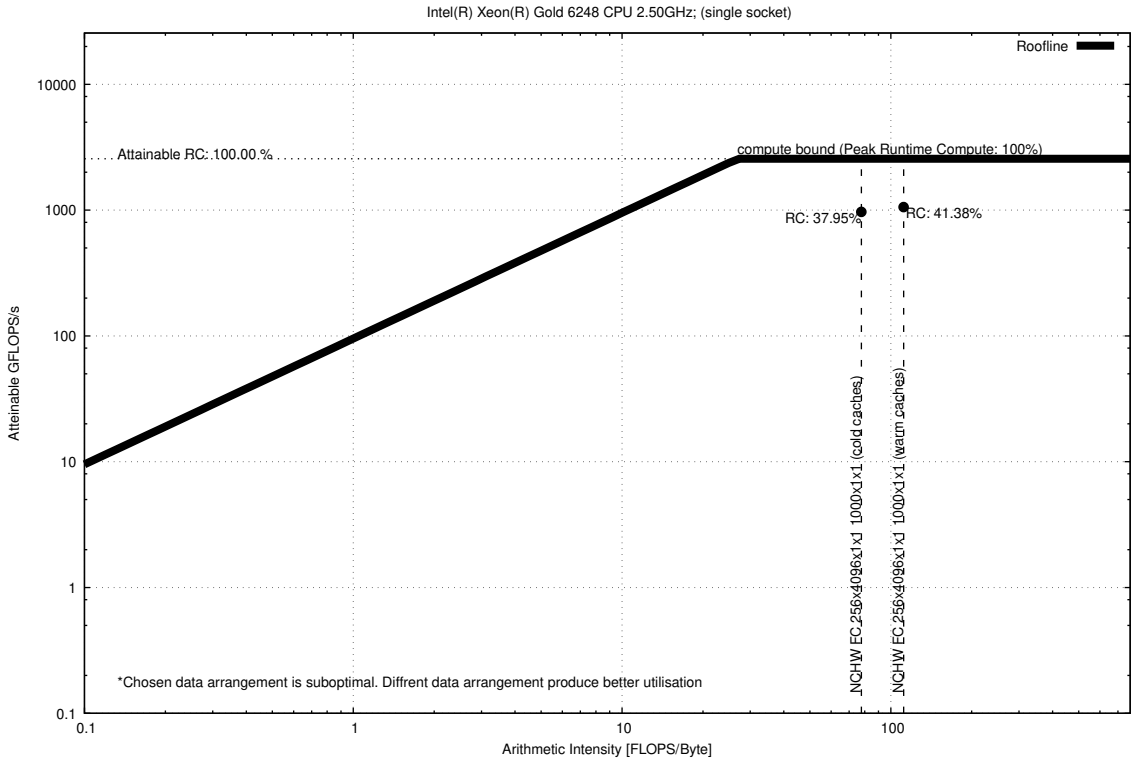
GELU activation

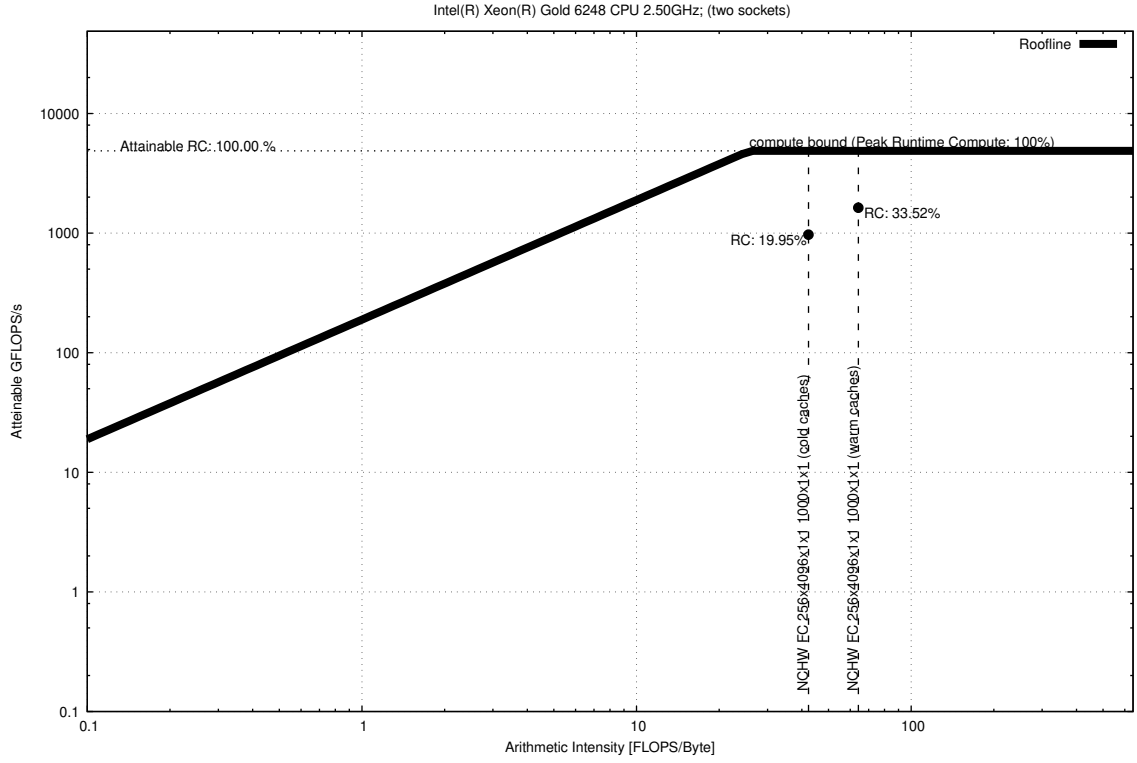



Intel(R) Xeon(R) Gold 6248 CPU 2.50GHz; (single socket)
Roofline
compute bound (Peak Runtime Compute: 100%)
Attainable RC: 17.72%
RC: 10.59%
RC: 10.47%
RC: 10.14%
RC: 10.13%
NCHW GELU 256x96x55x55 (cold caches)
NCHW16C GELU 256x96x55x55 (cold caches)
NCHW GELU 256x96x55x55 (warm caches)
NCHW16C GELU 256x96x55x55 (warm caches)
Attainable GFLOPS/s
Arithmetic Intensity [FLOPS/Byte]
0.1
1
10
100
1000
10000


Intel(R) Xeon(R) Gold 6248 CPU 2.50GHz; (two sockets)
Roofline
compute bound (Peak Runtime Compute: 100%)
Attainable RC: 19.47%
RC: 7.83%
RC: 7.47%
RC: 10.36%
RC: 9.51%
NCHW GELU 256x96x55x55 (cold caches)
NCHW16C GELU 256x96x55x55 (cold caches)
NCHW GELU 256x96x55x55 (warm caches)
NCHW16C GELU 256x96x55x55 (warm caches)
Attainable GFLOPS/s
Arithmetic Intensity [FLOPS/Byte]
0.1
1
10
100
1000
10000

Inner Product (Fully Connected)





Pooling (average)