UDC 681.3 https://doi.org/10.15407/pp2021.02.054

Dmytro V. Rahozin, Anatoliy Yu. Doroshenko

EXTENDED PERFORMANCE ACCOUNTING USING VALGRIND TOOL

Modern workloads, parallel or sequential, usually suffer from insufficient memory and computing performance. Common trends to improve workload performance include the utilizations of complex functional units or coprocessors, which are able not only to provide accelerated computations but also independently fetch data from memory generating complex address patterns, with or without support of control flow operations. Such coprocessors usually are not adopted by optimizing compilers and should be utilized by special application interfaces by hand. On the other hand, memory bottlenecks may be avoided with proper use of processor prefetch capabilities which load necessary data ahead of actual utilization time, and the prefetch is also adopted only for simple cases making programmers to do it usually by hand. As workloads are fast migrating to embedded applications a problem raises how to utilize all hardware capabilities for speeding up workload at moderate efforts. This requires precise analysis of memory access patterns at program run time and marking hot spots where the vast amount of memory accesses is issued. Precise memory access model can be analyzed via simulators, for example Valgrind, which is capable to run really big workload, for example neural network inference in reasonable time. But simulators and hardware performance analyzers fail to separate the full amount of memory references and cache misses per particular modules as it requires the analysis of program call graph. We are extending Valgrind tool cache simulator, which allows to account memory accesses per software modules and render realistic distribution of hot spot in a program. Additionally the analysis of address sequences in the simulator allows to recover array access patterns and propose effective prefetching schemes. Motivating samples are provided to illustrate the use of Valgrind tool.
Keywords: workload, performance analysis, coprocessors, prefetch, computer system simulator.

Introduction

The ultimate goal of computer hardware development activities is the improvement of application performance some way. During early days of microprocessor development, the basic hardware features were adopted – such as hardware pipeline, on-board cache memory and SIMD instructions, so one microprocessor instruction transforms operands during minimal number of clock cycles. After that the more advanced hardware methods were adopted – for example, the extraction of instruction-level parallelism and introduction of complex instructions, for example RSA crypto support. Excessive amount of research efforts was spent to gather the most often used computation patterns, and so extra hundreds of complex instructions were added. These instructions apply complex computation pipelines over small amount of data, and usually are used only with help of sophisticated optimizing compiler or direct assembly language instructions. The next level of performance improvement is the use of specialized coprocessors, which not only apply the complex computation pipeline, but also provide sophisticated address generation so that the coprocessor is able to access big memory areas. The cornerstone problem of this so called "next level" is the coprocessor complexity and the absence of good optimizing compilers which can transform original program code and map it onto the hardware coprocessor. So, mapping of the original algorithm to specialized hardware coprocessor is usually done by hand and rises software development costs. There are many examples of different kinds of mappings: starting at simpler RSA crypto-algorithm accelerators in various platforms – x86/ARM/PowerPC, where the coder just takes a code example from an application note; up to programming graphics card using shader concept, using a complex compiler to generate parallel code for GPU.

This so called "next level" of specialized hardware applications requires not only analysis of computational patterns over some scalar data, but also requires the analysis of data flow and transformations in nested

loops. The goal of the analysis is not only code optimization, but gathering requirements for useful coprocessor employment to accelerate complex computing patterns. Let's define the *complex computing pattern* as a part of computing algorithm which includes at least one single or nested loop and needs a complex memory access pattern (much more complex than just SIMD data path).

The goal of this paper is to make a step forward in the topic of software performance analysis and optimization for solving the following cases: 1) semi-automatic extraction of the information for possible optimization of the complex computing patterns with the help of co-processors with programmable memory access; 2) analysis of mapping of complex computing patterns for co-processors with programmable memory access. We consider off-the-shelf software and possible optimization cases for it and we propose techniques for this software optimization using performance analysis tools. We consider the extensions for Valgrind software (especially its subtool Cachegrind) which enable additional analysis of complex computing patterns.

1. Application performance analysis

So, complex computational patterns we are looking for usually reflect commonly used computation procedures, for example convolutions, matrix multiplication loops, loops similar to high-level BLAS kernels, neural network computational kernels, various DSP kernels. For many cases we already have appropriate tools, there iterative optimization techniques are employed for the existing code [1] and this greatly improves developer experience and reduces efforts necessary to optimize software. Another way which allows to simplify development and decrease efforts is the use of formal methods, were the development system already operates with parallel algorithms [2]. Although many complex systems can be modeled using high-level formal models, usually the model formalization is the second or third step of technology adaptation. Let us consider the modern topic of convolutional neural networks with popular implementations from Nvidia [3] and Berkeley [4]. Both implementations, despite a neural network can be well described as the formal models, are brilliant high-level frameworks for neural networks implementation, but still are hardly portable to any architecture except initial targets - Nvidia video cards and x86-compatible multicore processors. If a different hardware is required to run the neural network back-end and this hardware includes special coprocessor, we need to analyze the initial programs for "hot spots" – kernels or loop nest where the program spent the most of time. As software became more and more complex we need to have appropriate tools to analyze the code and there is no a trend to use highly formal techniques to define neural networks due to high complexity of infrastructure for formal models for parallel applications. Instead, we see the use of simpler tools such as Darknet [5] for implementing complex pipelines such as Yolo-v4 [6]. The analysis of such complex applications [7] made us to start looking for effective tools for analyzing the programs which can be optimized on modern multiprocessors minimizing human effort need to be spent on this analysis.

Today in practice neural network algorithms can be efficiently optimized with the use of a specialized coprocessor (such as Qualcomm's Neural Processing Engine [8]) and this is the common trend in system-on-chip design. The number of workloads, which performance can be improved with various coprocessors, quickly increases, so the interest of employing a coprocessor in application constantly increases. The impressive application – AI Benchmark (ai-benchmark.com), which uses QNPE SDK [8] for neural network processing – enables optimized neural network processing for system-on-chips widely used in off-the-shelf smartphones.

But this is the only side of hardware development. The fact is that the performance of some applications is bounded by execution speed (e.g. cryptographic hashes computations), another by memory performance (matrix multiplications, neural network convolutions), another by both memory performance and execution speed. Modern software is extremely complex and can be analyzed and optimized mostly in parts, but the common rule is that 90% of execu-

tion time is spent in 10% of code, for some workloads this ratio is 99%/1%. Even basic analysis shows that the most of modern applications are memory bounded, but the more detailed analysis is able to detect the most time consuming "hot spots" in application, as is successfully done using e.g. Intel Vtune performance analyser [9], or another similar software. Anyway, Intel Vtune and another analysis software are based on statistical approach and shows only "hot spot" with quite accurate number of memory traffic per executed instructions. But the high-level information about address sequences for memory-related instructions is not gathered during this analysis, but this missed information is the key for understanding the algorithm behavior and possible algorithm mapping on the coprocessor.

As current optimizing compilers are not able to map nested loops and complex addressing patterns on complex hardware, the co-processor should be utilized by hand using the SDK such as described in [8]. In order to simplify the handmade optimizations, performance analysis software should analyze addressing patterns, addressing patterns spatial locality and issued operations. This type of reporting is used in two ways: 1) reporting potential computation patterns, which can be optimized by hands; 2) reporting "hot spots" which are potentially optimizable if a corresponding coprocessor is included into system-on-chip. Another valuable point is the definition of prefetch scheme, which may be used for cache utilization optimization. Data access analysis is able to recover at least simple addressing patterns, which may be used for hand-made prefetch optimization.

An important trend is that commonly used hardware employs more and more microprocessor kernels (usually ARM in latest system-on-chips as ARM hardware kernel are small and energy efficient), and these kernels share or sit on common cache memory. Each kernel has enough computation power – with up to 3.5GHz clock frequency, with limited instruction level parallelism extraction (as in Cortex-A57/A75/A77) – and is able to process complex modern algorithms even without coprocessors. Note, that coprocessors also use the common cache memory for operations, so executing a computational thread on coprocessor hardware just heavily increases load on cache memory bus. This can be illustrated by the old fact related to employment of hyperthreading technology, when a processor core is able to execute instructions of two threads simultaneously. Running parallel threads which does not operate on same data effectively halves the cache size, which reduces performance despite employing two threads. The same works e.g. for typical conjugate gradient solver – it scales well only for several threads. Cache behavior analysis helps to determine cache bottlenecks and check if the bottleneck can be avoided, and additionally cache prefetch scheme can be defined for a pattern.

2. Efficient cache memory use and prefetch techniques

Memory bounded applications (such as mentioned above conjugate gradient solver) performance may be improved by proper prefetching scheme. The cache memory never works foreheads, so the cost of L1 cache misses remains extremely high. For high processor frequencies (~3 GHz) one L2 hit (i.e. L1 miss) costs up to dozen of clock cycles, one L3 hit (i.e. L2 miss) costs up to 70 cycles for cache interconnection type typical for Intel multicore processors. Read from DRAM costs up to 300 cycles. If compared to usual floating-point operations time, which equals to 1-2 clock cycles, memory access in case of L1 cache miss looks extremely high.

Farther in the paper while considering processor operations speed, we omit time spent for computational operations. It was mid-1980s when the off-the-shelf processor executed numerical operations taking multiple clock cycles. Starting at 32-bit processors in 1990s we never expect that basic floating-point operations (addition, multiplication) takes more than 1 clock cycle. On the other hand, if in early-1980s DRAM memory access appeared without additional wait cycles even at ancient ISA bus, each time after processor clock speed was significantly increased, the memory access time raised and the complexity of memory hierarchy in-

creased up to be enormously complex. Sure, that the main focus was moved from "how to compute fast" to "how to feed a processor with data fast".

A thoughtful reader may note that some memory-hungry operations may be introduced into memory controller, especially so called "reduction" operations, when some scalar operation is executed over large data array, for example convolution operation. Talking more precisely, a coprocessor which is able to speed up various BLAS kernels may improve significantly the performance of operations in many current workloads, but BLAS kernels work fast only if cache memory is used efficiently, that's why modern BLAS libraries always make a tuning for BLAS library before compiling it for particular machine for processor type and its cache memory configuration. This also applies for graphics processors, as BLAS package is compiled separately for each GPU architecture and modification type to use GPU registers in efficient way. So prefetch here is able to set the cache memory into desired configuration to avoid L1 cache misses. Modern prefetch instructions are able to move data between cache layers in order to hide even L2/L3 latencies. Anyway, this does not look a silver bullet – multithreaded applications perform side-effects on cache contents and Cachegrind [10] is a good tool to check efficiency of cache utilization in application regardless if application works on CPU or on GPU, as the memory traffic requirements are practically the same.

Several dozens of prefetch application schemes are considered in [11], but the fully automatic prefetch is extremely limited by adopting only simpler addressing schemes and compiler ability to determine loop constructions. Hardware prefetch schemes are also considered in [11], still they work for simpler addressing schemes.

Current prefetch hardware includes not only prefetch instructions, but also cache lines locking instructions, data invalidating instructions, changing priority of data invalidation/eviction in set and other service commands. The most complex case includes one or more separate prefetch coprocessors, which are able to prefetch arrays of large data blocks with stride synchronously with main thread(s) or prefetch linked lists of large data blocks. Here the prefetching coprocessor needs to be programmed by a full-blown control code and looks to be quite complex hardware.

Although current optimizing compilers use powerful algorithms, complex prefetch schemes are too hard for them – if a simple loop can be analyzed usually successfully, the loop nest is harder to analyze. The analysis of memory accesses sequence (address patterns) is the valuable way to check possible "hot spots" potentially optimizable by prefetch patterns.

3. Valgrind performance analysis tool

Valgrind tool [10] basically emulates a microprocessor instruction set, memory state of Intel x86, ARM and several other processors. Valgrind is able to emulate basic operation system libraries and run a program on the top of emulated system. The program is not modified any way (except special cases extending the program to control Valgrind behavior in run-time). Several useful tools are based on the simulator: a usual debugger *gdb*, a memory error detector (it checks for incorrect heap memory use), a cache memory and branch predictor simulator (a.k.a. Cachegrind), a call-graph generator, a concurrent thread error checker and even more profiling tools. Our target sub-tool is Cachegrind, the cache memory simulator and profiler. The sub-tool simulates two-level cache memory with a bunch of programmable parameters, and the most important parameter is the size of cache memory levels storage. Basically, the cache simulator provides tracking of: 1) traffic from register file to cache memory, 1st level, this is the application data throughput; 2) data traffic to 2nd level cache as the result of cache misses for 1st level cache memory for both reads and writes; 3) cache misses for 2nd cache level which is the final traffic to memory. The ratio (1) to (3) is the cache memory utilization efficiency, which depends on application type, compiler optimizations and applied threading model. With help of changeable cache

parameters Cachegrind analyzes cache performance deeply even for multithreaded programs. The simplest analysis case is graph of memory traffic (3) vs cache size, which allows to define the most comfortable cache size for application.

Before considering why we are limited in Cachegrind functionality we need to introduce types of workloads we are going to analyse with Cachegrind tool.

Microprocessor performance is still a keystone for enabling computing technologies for mass market, and more and more computing performance nowadays is reached by introducing more and more parallelism to newer hardware. Well-known Moore law says that the number of gates (transistors) on (silicon) die doubles every 18 months, but these extra transistors do not help to compute faster, so they need to be used in computational units which are able to perform in parallel and the computing task should have enough level of coarse-grained parallelism. The ultimate case is a video card equipped with a massive parallel processor, which initially was used to compute pixel color inside a rendered triangle, which is a highly parallel task performed on independent data. But the performance bottleneck for any video card is the memory channel, so the huge number of architectural solutions in video cards solves the problem of hiding memory latency. For less specialized computing software the matrix processing operations still are championing the race for computing resources. Modern neural network processing is based on large matrix multiplications, and despite the matrix multiplication optimizations are well studied, they are combined with another memory operations. This combination is not studied well due to its novelty, so it is our main case for performance analysis as 1) neural networks are emerging topic for mass market and challenging topic for computer hardware; 2) huge industrial demand for object and environment recognition, which is potentially solved by neural networks technology.

Recently we conducted a research [7] where we simulated neural network runs in order to project this workload performance for an embedded platform. We proved that Valgrind/Cachegrind is able to simulate huge modern neural networks runs with getting full performance results for cache memory from Cachegrind. Let us consider a result got from Cachegrind run simulating Yolo-v4 run on Intel x86 platform, details can be found in [7]. Note, that original program – Darknet infrastructure [5] was not changed anyway.

Fig 1. shows the "hot spot" in Darknet Yolo-v4 run, which is *gemm_nn* function, providing general matrix multiplication. The function generates 83.5% misses in 1st level cache memory and 98.5% misses in 2nd level cache memory and due to analysis, it is the main source of memory traffic in the application. This information looks very short in terms of the performance analysis of parallel or coprocessor-supported execution of the sample. Looking at fig. 1 we consider the matrix multiplication information, which can benefit from blocked matrix multiplication, but no one can easily determine if some particular threading affinity configuration makes the program faster due to sharing cache data among threads or makes the program slower due to extra data spills from cache. Cachegrind simulation potentially can provide such type of analysis, but current output at fig. 1 limits a Cachegrind user in ability to analyze the software run – practically all the memory resource is spent in one *gemm_nn* function.

```
                .                 .                 .                 .          void gemm_nn(int M, int N, int K, float ALPHA,
                .                 .                 .                 .              float *A, int lda,
                .                 .                 .                 .              float *B, int ldb,
                .                 .                 .                 .              float *C, int ldc)
                0                 0                 0           237,419 ( 0.00%)  {
                .                 .                 .                 .              int i, j, k;
          203,502 ( 0.00%)        0                 0            33,917 ( 0.00%)   for (i = 0; i < M; ++i) {
      128,693,815 ( 0.33%)  6,173,888 ( 0.11%)          18 ( 0.00%)          0          for (k = 0; k < K; ++k) {
       64,296,032 ( 0.17%)  6,912,576 ( 0.12%)  4,018,604 ( 0.28%)          0              PUT_IN_REGISTER float A_PART = ALPHA * A[i * lda + k];
      128,659,898 ( 0.33%)  3,081,313 ( 0.05%)          0           271,336 ( 0.00%)      for (j = 0; j < N; ++j) {
32,167,924,337 (83.51%) 5,680,796,154 (99.02%) 1,433,439,177 (98.49%) 16,083,979,127 (86.44%)          C[i*ldc + j] += A_PART*B[k*ldb + j];
                .                 .                 .                 .                  }
                .                 .                 .                 .              }
                .                 .                 .                 .          }
          237,419 ( 0.00%)        33,917 ( 0.00%)        2,752 ( 0.00%)          0  }
```

Fig. 1. Excerpt from Yolo-v4 run performance result: D1mr, D1mw, D2mr, D2mw values

4. Performance analysis shortcomings

Let us consider fig. 2. It includes three functional elements (FEs), #1, #2, #3, which represent three separate neural network layers, each calls several kernels from library.

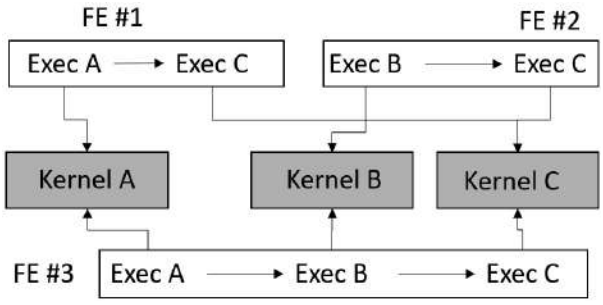

Fig. 2. A sample call graph for an excerpt from neural network.

So, FE #1 call kernels *A* and *C*, FE #2 calls *B* and *C*, and FE #3 calls all *A*, *B* and *C*. So, kernel *A* collects cache events from FE #1, #3; *B* from #2, #3 and *C* from all FEs. Therefore the performance gathering via Cachegrind does not reflect real distribution of spent resources per original FEs. Fig. 3 shows the sequence of basic neural network layers, forming Yolo-v4 pipeline. In practice, Valgrind mixes performance information from all the layers at fig. 3 into one function *gemm_nn*, generating the report at fig. 1, where *gemm_nn* gathers practically all memory operations into one function. This prevents proper analysis, as there is no information for each separate network layer, which calls *gemm_nn* function. Similar considerations work for sharing cache contents between several execution threads, here we have the case that the cache memory is simulated correctly, but it is unknown how each execution thread influences the cache content and how to compare one-threaded and multithreaded program execution in terms of cache behavior and calculate the [non]efficiency of cache use if threading model and number of threads are changed in run-time.

The only way to overcome this limitation is to instrument (add the functionality for controlling performance analysis) the code, so that the performance accounting for all FEs is separated. "From-the-box" Cachegrind is not controllable someway, but Valgrind has extensible API to control the behavior of other Valgrind tools.

5. Extending Valgrind tool

Valgrind includes a mechanism which allows user to control Valgrind-based execution of a program. The user is able to place "specific client requests" into program to control several Valgrind components, for example for Callgrind and add new client requests.

To control the tool user should use *callgrind.h* file from Valgrind distribution and use predefined macros, for example *CALLGRIND_START_INSTRUMENTATION*. This "C-style" macro definition and any other client request macros are directly translated into a specific processor instruction for target platform (x86, PowerPC), which is "void" i.e. does not change microprocessor state, but allows to pass arguments to Valgrind kernel. Client request parameters are passed into Cachegrind in similar way to standard function argument list. In run-time Valgrind core intercepts the compiled binary instruction and passes control to appropriate Cachegrind handler, which provides necessary functionality to analyze or change Cachegrind internal state.

Cachegrind handles cache memory state in separate data structures, including cache memory contents, memory tags state, eviction candidates information and statistics for cache misses/loads/traffic to

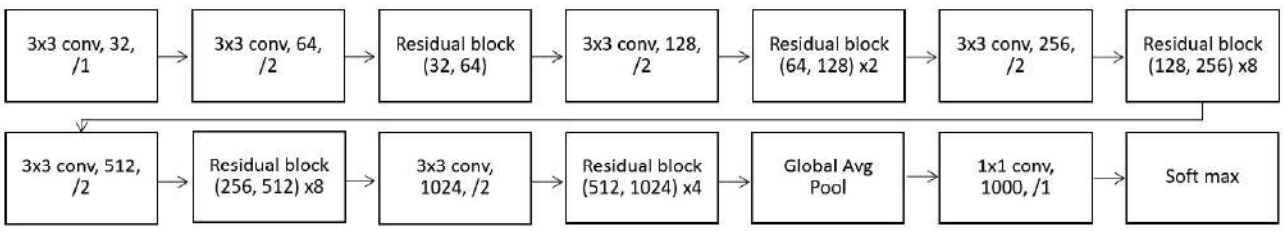

Fig 3. Computational structure of Yolo-v4 blocks.

memory. In order to improve cache simulation, we provide several copies of cache statistics – we call it "context" - and each context copy is filled with statistics separately by Cachegrind simulator. The number of contexts is set by user during Valgrind compilation. Basically, we add two new client requests – *CG_PUSH_CONTEXT* and *CG_POP_CONTEXT*. "CG" stands for "Cachegrind". A new client request *CG_PUSH_CONTEXT* switches current context to another enumerated one so that the further statistics about cache misses and data traffic is added to another context. Default context number is 0, it is used from the start of cache simulation. After the context switch the previous context number is saved in the stack of context numbers, so that the following request *CG_POP_CONTEXT* is able to restore the previous context. Appropriate client requests for changing cache context are inserted into the source code so that the selected hot spots and kernels are separated in different cache contexts. Another improvement is adding an "old context" bit into cache content descriptors per each cache memory line. This bit is set in the case of cache context switch for all data stored in cache and is cleared in the case if a cache "hit" into the cache data was detected or in the case of data eviction from cache. The accounting for the number of bit clears of the "old context" bit in the case of a cache hit gives us the amount of data which was reused across different hot spots/kernels in the program code.

Some useful inter-thread data usage statistics may use the similar principle – track the thread identifiers (ids) and variable operations (read/write) to analyze variable sharing efficiency. Valgrind already have the tool analyzing the multithreaded program for dangerous data races, but we leave this research for near future.

We decided to research in steering for generated addresses – handling a pool of last used addresses, keeping a record of possible increments for each address and establishing a sequence of accessed memory cell for addressing schemes in a loop allows a user to analyze an address patterns and form a report notifying about structural accesses i.e. data accesses with some specific address change pattern over one variable, structure, array, array of structures. The similar functionality is embedded into x86 processors: hardware-based automatic data prefetch. For example, the loop:

For(int i=0; i<N; i++)
*{ c[i] = A*a[i] + B*b[i] + P; }*

has the following structural accesses (we use the pattern *variable_name*[*start_index*:*step*:*end_index*]: *read a*[0:1:N], *read b*[0:1:N], *set c*[0:1:N]. This analysis helps in cases of mapping cache accesses and real arrays and allows to determine how much time the array was accessed and reused in cache. The address steering information is dumped at the end of simulation and does not require additional user control.

6. Use of extended Valgrind functionality

Let us return back to figure 2 and consider block FE #1, #2, #3, where the kernels *A*, *B* and *C* are called in some sequence. Default cache simulation shows, that the number of cache misses is distributed as follows:

Kernels	A	B	C
	22%	45%	25%

The same distribution for FEs is:

FE	#1	#2	#3
	0.7%	1%	0.5%

These tables show just that all memory traffic is utilized in library kernels *A*, *B* and *C* but says nothing about actual memory traffic distribution across FEs, which is necessary to get real distribution of memory accesses per FE. Note that the use of hardware counter in e.g. Intel Vtune shows the same picture for this case. To improve the analysis let us use cache context 1, 2, 3 for FEs #1, #2, #3 and add corresponding Cachegrind controls CG_PUSH/POP CONTEXT into the source code of FEs. Passing the updated program into the simulator shows the next picture:

Kernel	A	B	C
FE #1	3%	6%	3%
FE #2	11%	21%	14%
FE #3	8%	18%	8%

Now we clearly see how much memory traffic is utilized in each FE. The difference

between the default performance data and improved data does not require any comments.

Conclusions

The article considers the use and extension of a microprocessor and system simulator Valgrind for performance accounting and performance analysis of big modern workloads. As a conclusion, we accent several points which are helpful for studying of efficiency of modern workloads:

1. Even big modern workloads such as object detection neural networks are able to be analyzed by system simulator using off-the-shelf computers in short time. Also, memory behavior simulation is the main hardware subsystem we need to analyze in order to understand the workload performance bottlenecks.

2. Valgrind tool may be easily extended for research purpose to control or change the simulation process behavior via client requests.

3. Our extensions for Cachegrind control allows to analyze big pipelines in parts and determine bottlenecks in memory subsystem (cache memory and memory bus).

4. We have checked methods for analyzing data address streams for recovering prefetch pattern for nested loops and found that basic memory addressing schemes are recovered good enough to provide data for necessary cache traffic. This works extremely good jointly with (3).

5. Data stream recovery allows to separate automatically cache misses while loading several data arrays (streams) and compare cache performance for each array (and its prefetch method) for various data layouts and prefetch methods.

6. Cachegrind allows to simulate various cache behavior, so we can change e.g. cache data eviction policies in order to check if this can improve performance while executing some parts of workloads.

This kind of performance statistics gathering and grouping allows the qualified software engineer to find potentially optimizable part of code much faster and enable various preprocessors for a workload or apply profitable prefetch schemes. Simulator analysis here works more efficiently than direct workload runs as the simulations allows to gather and keep information which is lost while fast direct software runs.

These results give us some prospects for future work. Valgrind supports multithreaded execution and memory races analysis for several threads. One of interesting ways to use Valgrind is to run different combinations and affinity of software threads in a multithreaded workload, checking performance effects on shared cache data, and this topic is the research target for near future.

References

1. A. Doroshenko, O. Beketov. Large-Scale Loops Parallelization for GPU Accelerators. //In Proc. of the 15th Int. Conf. on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer. Vol I. Kherson, Ukraine, June 12-15, 2019. CEUR-WS, vol. 2387 (2019).-P.82-89. http://ceur-ws.org/Vol-2387/
2. A. Doroshenko, O. Yatsenko. Formal and Adaptive Methods for Automation of Parallel Programs Construction: Emerging Research and Opportunities. IGI Global, Hershey, Pennsylvania, USA. 2021, 279 p. DOI: 10.4018/978-1-5225-9384-3
3. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. & others (2016). TensorFlow: A System for Large-Scale Machine Learning.. *OSDI* (p./pp. 265--283)
4. Y. Jia and Evan Shelhamer and J. Donahue and S. Karayev and J. Long and Ross B. Girshick et al. Caffe: Convolutional Architecture for Fast Feature Embedding. // In Proc. of the 22nd ACM international conference on Multimedia, 2014
5. J.Redmon. (2013) [Online]. Darknet: Open Source Neural Networks in C. – Available from https://pjreddie.com/darknet/
6. Bochkovskiy, A.; Wang, C.Y.; Liao, H.Y.M. YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv 2020, arXiv: 2004.10934.
7. D. Ragozin, A. Doroshenko. Memory Subsystems Performance Analysis for CNN Workloads. // In Proc. of AUTOMATION 2020: 26-th Scientific conf. in memory of

L. Pontryagin, N. Krasovsky and B. Pshenichny, 2020, Kyiv, Ukraine. P. 12-122.
8. Ignatov A. et al. (2019) AI Benchmark: Running Deep Neural Networks on Android Smartphones. In: Leal-Taixé L., Roth S. (eds) Computer Vision – ECCV 2018 Workshops. ECCV 2018. Lecture Notes in Computer Science, vol 11133. Springer, Cham. https://doi.org/10.1007/978-3-030-11021-5_19
9. J. Rainders and J. Jeffers. High Performance Parallelism Pearls. Morgan-Kauffmann, 2015. 502 p., https://doi.org/10.1016/C2014-0-01797-2
10. J. Weidendorfer. Sequential Performance Analysis with Callgrind and KCachegrind. // In Proc. of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart, pp. 93-113
11. Mittal, Sparsh. (2016). A Survey of Recent Prefetching Techniques for Processor Caches. ACM Computing Surveys. 49. 10.1145/2907071.
12. Kim, Yoongu; Yang, Weikun; Mutlu, Onur (2016): Ramulator: A Fast and Extensible DRAM Simulator. Carnegie Mellon University. Journal contribution. https://doi.org/10.1184/R1/6469208.v1

Received: 17.05.2021

***About authors*:**

Dmytro V. Rahozin, candidate of tech. sciences (PhD)
More than 10 publication in Ukrainian and foreign journals.
https://orcid.org/0000-0002-8445-9921

Anatoliy Doroshenko, Doctor of Sciences in Physics and Mathematics, Professor,
Head of the Department of Computing Theory, Institute of Software System of the National Academy of Sciences of Ukraine,
Professor of Department of Automation and Control in Technical Systems
Igor Sikorsky Kyiv Polytechnic Institute.
Number of scientific publications in Ukrainian publications - more than 180.
Number of scientific publications in foreign publications - more than 70.
Hirsh index - 6.
http://orcid.org/0000-0002-8435-1451

Affiliations:

Institute of Software Systems, NAS of Ukraine
03187, Kyiv-187, Acad. Hlushkov avenue, 40
Tel. +38 068 575 91 25
E-mail: dmytro.rahozin@gmail.com