Automatic Cache Aware Roofline Model Building and Validation Using Topology Detection

Nicolas Denoyelle, Aleksandar Ilic, Brice Goglin, Leonel Sousa, Emmanuel Jeannot

Automatic Cache Aware Roofline Model Building and Validation Using Topology Detection

Nicolas Denoyelle & Aleksandar Ilic & Brice Goglin & Leonel Sousa & Emmanuel Jeannot

Inria - France – INESC-ID – Portugal
nicolas.denoyelle@inria.fr ilic@sips.inesc-id.pt brice.goglin@inria.fr las@sips.inesc-id.pt emmanuel.jeannot@inria.fr

Abstract

The ever growing complexity of high performance computing systems imposes significant challenges to exploit as much as possible their computational and memory resources. Recently, the Cache-aware Roofline Model has gained popularity due to its simplicity when modeling multi-cores with complex memory hierarchy, characterizing applications bottlenecks, and quantifying achieved or remaining improvements. In this short paper we involve hardware locality topology detection to build the Cache Aware Roofline Model for modern processors in an open-source locality-aware tool. The proposed tool also includes a set of specific micro-benchmarks to assess the micro-architecture performance upper-bounds. The experimental results show that by relying on the proposed tool, it was possible to reach near-theoretical bounds of an Intel 3770K processor, thus proving the effectiveness of the modeling methodology.

Keywords Roofline Model, DRAM, Cache, Tool, Cache Aware Roofline Model, hwloc

I. Introduction

Since the advent of multi-core era, computer systems tend to incorporate an increasing number of cores, while the relative memory bandwidth and memory space per core is decreasing [11]. In order to address application requirement and improve the overall performance, current computing platforms rely on memory hierarchies of increasing complexity. Reshaping applications data layout to take full advantage of those architectures can significantly improve the overall performance at the cost of tremendous development efforts. The Cache Aware Roofline Model (CARM) [5] is able to aggregate this complexity in a single insightful model, and guide application optimization to fit the micro-architecture performance upper-bounds. Its effectiveness motivated us to bring it to non expert developer a robust tool equipped with deep benchmarking of multi-core platforms with complex memory hierarchy, which automatically builds the model and provides the application optimization insights.

To conduct a thorough evaluation of memory and compute capabilities of a given platform, the proposed tool also includes the necessary software support to identify both micro-architecture instruction set and cache topology. The former can be found with compiler support [1], whereas the latter has only been mastered in a portable way by hwloc (hardware locality) library [3]. By relying on this run-time detection of compute and memory resources, the proposed tool automatically instantiates a set of custom platform-specific micro-benchmarks for deep evaluation of platform capabilities, upon which the Cache-aware Roofline Model is generated. Furthermore, the proposed tool also includes a lightweight library to provide access to the hardware counters and extract, at runtime, the application features to be mapped in the model. To the best of our knowledge, there are no existing cross-platform and open-source tools that allow automating this process (i.e building the CARM and mapping applications in it).

The remainder of this paper is organized as follow: Section II describes the original Roofline Model and the Cache Aware Roofline Model. Section III details our tool features, design choices to model the cache hierarchy, and take full advantage of the architecture, and provides preliminary results. Section IV concludes the paper.

II. The Roofline Model Then and Now

The Roofline modeling, in general, is an insightful approach to represent the performance upper-bounds of a processor micro-architecture. Since computations and memory transfers can be simultaneously performed, the Roofline modeling is based on the assumption that the overall execution time can be limited either by the time to perform

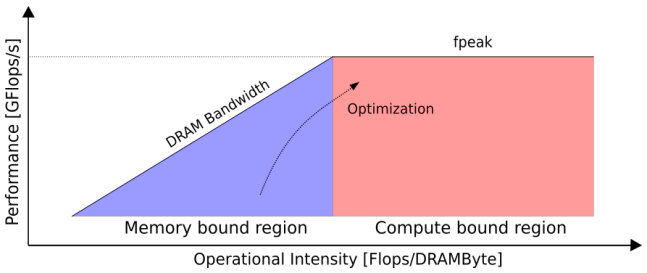

Figure 1: ORM chart

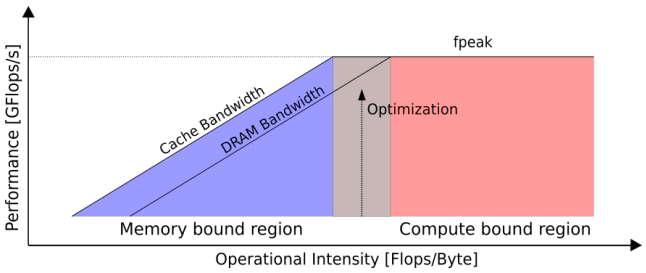

Figure 2: CARM chart

computations or by the time to transfer data. Hence, from the micro-architecture perspective, the overall performance (typically expressed in flops/s) can be limited by the peak performance of computational units or by the capabilities of memory system (i.e., memory bandwidth). To this date, there are two main approaches for Roofline modeling, namely: the Original Roofline Model (ORM) [13] and the Cache-aware Roofline Model (CARM) [5]. These two approaches provide different perspectives when describing the micro-architecture upper-bounds, and they are also differently constructed, validated, and used for application characterization and optimization.

The ORM targets the systems with a processing element (PE) connected to a single (slow) memory (usually, the DRAM). The ORM's PE encapsulates computational units and a set of fast memories (i.e., caches). As such, the ORM mainly considers the memory transfers between the last level cache and the DRAM (commonly referred as DRAM-Bytes). Hence, it denotes the theoretical DRAM bandwidth as one of the potential execution bottlenecks. Depending on the "operational intensity", i.e., the ratio of compute operations (flops) over the quantity of DRAM data (DRAMBytes), the applications can be characterized as compute-bound or memory-bound. The model was used in several works for application optimization [6] [10] [12], as well as to model other.

Figure 1 represents the ORM for a hypothetical computing platform. The axes of the chart are presented in log-log scale, where the "operational intensity" (in flops/DRAMByte) stands on abscissa and the performance (in flops/s) stands in ordinate.

In contrast, the CARM perceives the memory transfers from a consistent micro-architecture point of view, i.e., a core, where the memory transactions are issued. As such, the CARM targets contemporary systems where the PE encloses only compute units and registers, while all other memory levels are separately and explicitly considered. For this purpose, the CARM includes several memory lines in the same plot, each corresponding to the realistically achievable bandwidth of a specific memory level to the core, i.e., cache levels and DRAM. When characterizing the applications, the CARM relies on the true "arithmetic intensity", i.e., the ratio of performed compute operations (flops) over the total volume of requested data (in bytes) by taking into account the complete memory hierarchy (i.e., caches and DRAM).

Fig. 2 shows the CARM general layout for a hypothetical micro-architecture with a single cache level and DRAM. The CARM axes are presented in the log-log scale, where the x-axis refers to the arithmetic intensity (in flops/byte) and the y-axis to the performance (in flops/s). As presented in Fig. 2 (see dashed line), the CARM allows visualizing whether an application with a given arithmetic intensity is memory-bound or compute-bound by observing if a straight vertical line hits a peak (FP) roof or a bandwidth roof.

For these reasons, we base our methodology on the Cache Aware Roofline Model. As explained above, the CARM differs from the original model, it is usually capable of providing deeper insights when analyzing the applications execution bottlenecks, and it also has potential to be adapted to future memory designs. Moreover, the ORM has already a dedicated tool [7] for a similar purpose as ours, but the approach adopted in the herein proposed tool significantly differs and it targets a more consistent and concrete analysis.

III. Locality-Aware Roofline Tool

Our main contribution consists in the development of the open-source tool named Locality Aware Roofline Tool (LART)[1], which exploits hwloc topology detection to automatically build the Cache Aware Roofline Model (CARM).

[1]available at: https://github.com/NicolasDenoyelle/LARM-Locality-Aware-Roofline-Model-

Main tool features.

The proposed LART is composed of 3 main components, namely:

- A set of micro-benchmarks for automatic CARM construction on a given micro-architecture;
- The library for counter-based extraction of CARM metrics from a user application (i.e., the number of performed flops and transferred bytes, as well as the overall execution time);
- A visualization tool to present the model with architecture bounds and applications metrics extraction.

The first component consists in a program that automatically builds the CARM for the specific processor micro-architecture where the tool is run. By relying on a set of hwloc features, the proposed tool automatically detects the memory hierarchy and processor compute capabilities, based on which specific micro-benchmarks are instantiated to deeply evaluate the bandwidth of each memory level, as well as the peak floating point (FP) performance according to the CARM methodology. In addition, the proposed tools also permits to perform the CARM validation tests, by running a set of micro-benchmarks with variable arithmetic intensity. The second component of the tool represents a library with a set of API calls. These API calls are aimed at performing the automatic CARM characterization of a given user application, by instrumenting the application source code. To provide a wider cross-platform portability, this component relies on PAPI [9] features to collect all necessary CARM metrics via hardware performance counters, i.e., to determine the application arithmetic intensity and performance. The third component of the proposed tool is a command-line generating a visual plot of the CARM using platform analysis results. It enables a user to plot application metrics extracted with the above-referred library in the CARM chart. The model validation and bandwidth deviation can also be seen and provide a straightforward evaluation of the confidence one can grant to the model.

Building the model from a hierarchical topology

Discovering all the computing and memory resources in a computing platform can be performed with tools such as hwloc [4]. Prior to hwloc, the similar approaches were often less portable or they were not capable of exposing as many details about cache sharing etc. The hwloc framework models the machine topology as a tree and suits particularly well the caches structure. As presented in Figure 3, the view returned by the hwloc represents, express this structure with

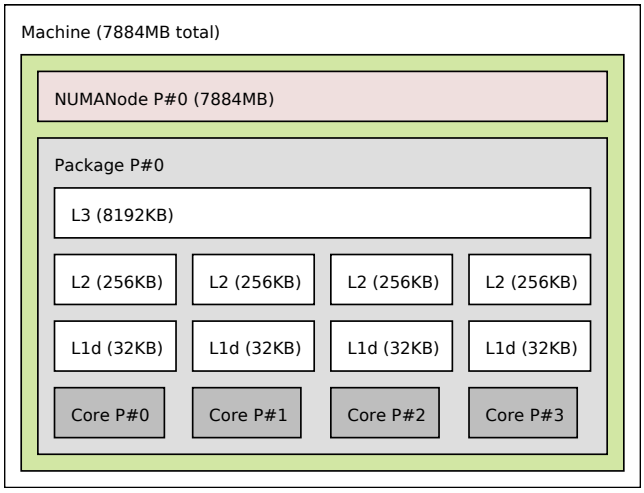

Figure 3: Topology of Intel Ivy Bridge processor model i7 3770k as seen by hwloc

nested boxes. Each core has a stack of 2 private caches, while all cores share the last level cache and the main memory (DRAM). This model where each Core sees the cache hierarchy as a cache stack of increasing size[2], perfectly suits the way how the CARM perceives the memory hierarchy. In addition, the hwloc library also allows a straightforward identification of the cache and memory sizes via the attributes of the Core parent nodes. These parameters are further used in the proposed tool in order to instantiate the appropriate micro-benchmarks for different memory subsystem levels by using the state of the art technique (i.e. buffer streaming of increasing sizes). The floating point peak performance is determined by executing a set of flop instructions in parallel on each core detected by hwloc. However, determining the bandwidth for different levels of memory hierarchy is more challenging, since it is required detecting the cache hierarchy structure with hwloc. This "topology aware" benchmarking technique is detailed in algorithm 1. For each cache level and memory, the proposed tool automatically determines an upper bound and lower bound size, which are subsequently used to build buffers of varying size fitting only the target cache. Afterwards, a specifically developed bandwidth benchmark is performed several times, the median value of all benchmarked sizes is reported as the experimentally determined bandwidth for the target memory level.

[2] The processors use a cache replacement policy where old data from closer caches is evicted in favor of more frequently used ones. The replacement policy defines the method how data is moved from bottom caches to top ones(see in Figure 3). Since the size of the caches closer to cores is smaller than the one for the farther memory levels, the cache stack as seen by each core has an increasing size from bottom to top.

```
Data: topology, repeat
n_threads = hwloc_get_nbobjs_by_type(topology,
 HWLOC_OBJ_CORE);
Core0 = hwloc_get_obj_by_type(topology,
 HWLOC_OBJ_CORE, 0);
/* See subsection III.3 for benchmark details
   */
fpeak = median(parallel_flop_uops(repeat));
/* Cache here is a memory subsystem.          */
foreach cache in ancestors(topology, Core0) do
    min_size = cache.size *
     hwloc_get_nbobj_inside_cpuset_by_type(topology,
     cache.cpuset, HWLOC_OBJ_CORE);
    max_size = ancestor_cache(topology, cache).size;
    for size in min_size:max_size do
        buffer = array_of_size(size/n_threads);
        /* See subsection III.3 for benchmark
           details                                */
        time = parallel_mem_uops(copy(buffer));
        bandwidths[size] = buffer.size*n_threads/time;
    end
    cache.bandwidth = median(bandwidths);
end
```

Algorithm 1: Memory subsystem benchmark algorithm

```
loop:
 vmulpd %%ymm0, %%ymm0, %%ymm0
 vmulpd %%ymm1, %%ymm1, %%ymm1
 ...
 vmulpd %%ymm15, %%ymm15, %%ymm15
sub $1, (%[n_times])
jnz loop
```

Figure 4: assembly sample for MUL fpeak benchmark

```
loop:
vmovapd (%[buf]), %%ymm0
vmovapd 32(%[buf]), %%ymm1
vmovapd %%ymm2, 64(%[buf])
vmovapd 96(%[buf]), %%ymm3
vmovapd 128(%[buf]), %%ymm4
vmovapd %%ymm2, 150(%[buf])
add $182, %[buf]
sub $182, %[buf_size]
jnz loop
```

Figure 5: assembly sample for 2LD 1ST bandwidth benchmark

Reaching the architecture upper-bounds

Nowadays, general purpose processors usually implement a variety of vector operations, also named as Single Instruction Multiple Data (SIMD) operations. Depending on the target micro-architecture, the tool proposed herein is able to automatically detect the operation type that allows to fully exploit the micro-architecture capabilities (typically, the widest vector instructions). These instructions refer to both compute operations and memory transactions, where the performance upper-bound of each involved unit is expressed as a function of the register size (i.e the number of floating point elements it contains) and the achievable throughput. By compiling the benchmarks on target architecture, we ensure that the largest vector size is used for the benchmarks by interpreting the compiler macros. For instance, figure 4 presents a set of instruction for MUL roof measure on architecture supporting AVX SIMD instructions. Each MUL instruction (vmulpd) is performed using a single register (ymm) for both MUL operands, i.e., it is equivalent to squared value. By ensuring the use of a single register per FP operation, the register dependencies among different instructions are avoided, which allows exercising the full potential of FP units in terms of the achievable throughput.

It is worth to emphasize that typically there are several types of memory/compute instructions on modern processors, and separate hardware units capable of performing different operations simultaneously. For instance, a core may perform a multiplication (MUL) and an addition (ADD) on separate FPUs, which can also be performed in parallel when there are no dependencies between them. Hence, a core can provide significantly higher performance for the codes that fully interleave ADD and MUL operations. This principle also applies to the memory subsystem, where several ports can be dedicated in modern processors to simultaneously serve different number of load (LD) and store (ST) operations, e.g., two LD and one ST 128-bit ports in the Intel Ivy Bridge micro-architecture. Hence, in order to exercise the full compute and memory capabilities of the target architecture, the proposed tool relies on several types of operations to benchmark the platform and it selects by default the one used by the CARM, e.g.for the Intel Ivy Bridge, it interleaves 2 LD and 1 ST instruction when assessing the peak memory bandwidth, while one ADD and one MUL are interleaved for peak FP performance. Figure 5 shows a 2 LD and 1 ST instruction set as used in our bandwidth benchmarks for architecture supporting AVX SIMD instructions.

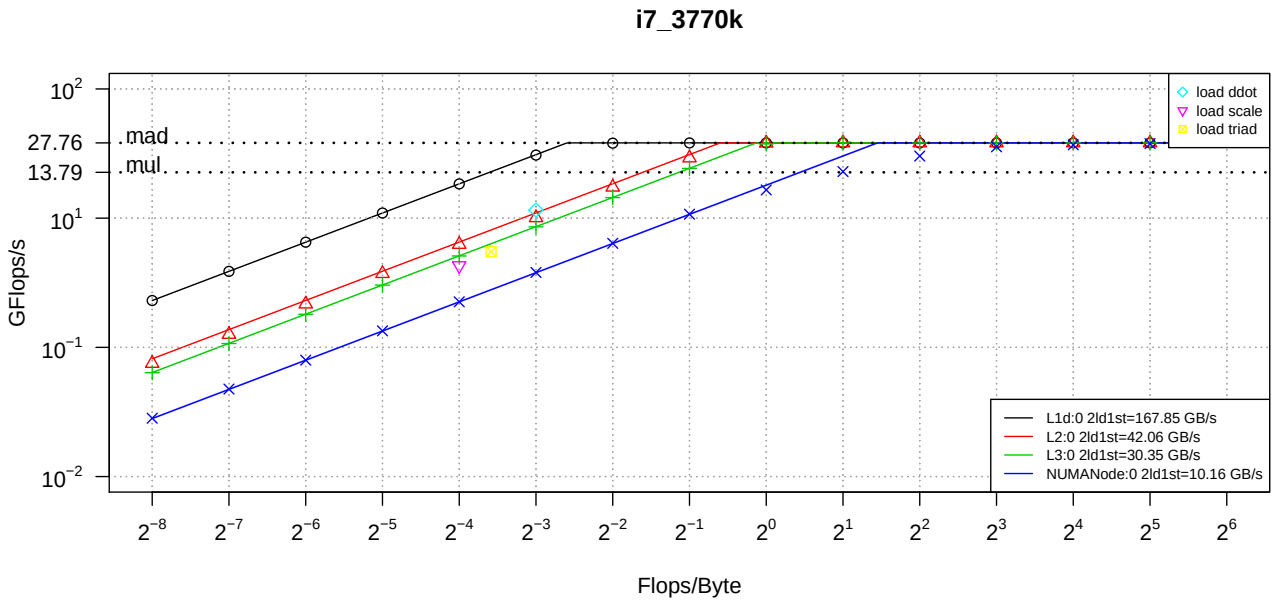

Figure 6: CARM on i7 3770k with LART tool.

LART Reproducing CARM Experimental Results on Intel Ivy Bridge

Figure 6, shows an output of the CARM plot generated by the herein proposed tool for an Intel i7 3770k (Ivy Bridge) processor, which topology is previously displayed in Figure 3. The black, red, green and blue oblique lines distinguish several regions of the attainable performance upper-bounds for AVX instructions, which are limited by the bandwidth of different memory levels , i.e., L1, L2, L3 and DRAM, respectively. The two horizontal lines represent the peak FP performance for MUL/ADD and multiplication with addition (MAD).

It is worth to note that the proposed tool was capable of reaching the near-theoretical upper-bounds of the tested micro-architecture both for the the L1 bandwidth and peak FP performance. In particular, by relying on the CARM testing methodology, the throughput of 1.49 instructions per cycle (IPC) was achieved for the L1 AVX-256 accesses. In addition, the IPC of 1.98 was achieved for FP performance, which closely match the theoretical throughput of AVX FP instructions when overlapping ADD and MUL operations.

The colored points matching the CARM lines represent the results of the validation benchmarks provided within the proposed tool, i.e., a set of synthetic benchmarks tailored to hit the performance upper-bounds of the micro-architecture for different arithmetic intensities.

As presented in Figure 6, legend in the bottom right corner, includes first the memory subsystem, then the micro-operation type(i.e. 2ld1st - interleaving of 2 LD and 1 ST) and the experimentally obtained bandwidth. On the top right corner in Figure 6, the legend refers to the tested applications for which the CARM metrics were extracted with our library. Those applications express different arithmetic intensity and are well suited to be analyzed with this model. In particular they represent application potential hot spot and come from well known benchmarks named as HPCCG(from Mantevo [2] mini-applications) and STREAM [8]. Although deep performance evaluation of those applications is out of the scope of the paper, it is worth to note that the proposed LART tool is capable of providing the facilities visually analyze the behaviour even for real-world applications.

IV. Conclusion and future work

On the path of extreme scale computing, computer systems complexity is increasing to address hardware and software constraints. The CARM is able to aggregate this complexity and by relying on hwloc topology detection capability we developped a robust tool to build this model and characterize applications. The LART tool is capable of performing

deep platform analysis, as well as model validation with automatic detection of micro architecture capabilities and topology. In order to further ease the burden of platform-specific benchmarking for non expert developers the proposed tool also provides a library to project and visualize applications in the model. The efficiency of the proposed tool was verified on a computing platform with Intel Ivy Bridge micro-architecture, where the obtained experimental results show that the proposed tool was capable of reaching near-theoretical performance.

In a close future, we plan to extend the tool and the model to cover heterogeneous memory systems and show their usefulness to improve data spatial locality in Non-uniform memory access (NUMA) systems, while the current model is manly used to improve data temporal locality with cache usage optimization.

V. Acknowledgements

We would like to acknowledge Action IC1305 (NESUS) for funding this work.

References

[1] GCC documentation on platform specific macros. `https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc/x86-Built-in-Functions.html#x86-Built-in-Functions`.

[2] Richard F Barrett, Paul S Crozier, DW Doerfler, Michael A Heroux, Paul T Lin, HK Thornquist, TG Trucano, and Courtenay T Vaughan. Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications. *Journal of Parallel and Distributed Computing*, 75:107–122, 2015.

[3] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010.

[4] Brice Goglin. Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications. In *1st ACM International Symposium on Memory Systems (MEMSYS16)*, Washington, DC, United States, October 2016. ACM.

[5] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014.

[6] Ki-Hwan Kim, KyoungHo Kim, and Q-Han Park. Performance analysis and optimization of three-dimensional {FDTD} on {GPU} using roofline model. *Computer Physics Communications*, 182(6):1201 – 1207, 2011.

[7] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. *Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis*, pages 129–148. Springer International Publishing, Cham, 2015.

[8] John D McCalpin. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what*, 22, 1995.

[9] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, pages 7–10, 1999.

[10] Diego Rossinelli, Christian Conti, and Petros Koumoutsakos. Mesh–particle interpolations on graphics processing units and multicore central processing units. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 369(1944):2164–2175, 2011.

[11] Avinash Sodan. Multi Core Trends in High Performance Computing. `https://www.sics.se/sites/default/files/pub/sics.se/avinash_final_sweden_many_core_day_keynote_-_avinash_final_-_clean.pdf`.

[12] Rob V. van Nieuwpoort and John W. Romein. Using many-core hardware to correlate radio astronomy signals. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 440–449, New York, NY, USA, 2009. ACM.

[13] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.