Assessing Memory Bandwidth on ARCHER2 and LUMI Using CAMP

Wenqing Peng
EPCC
The University of Edinburgh, UK
W.Peng-12@sms.ed.ac.uk

Adrian Jackson
EPCC
The University of Edinburgh, UK
A.Jackson@epcc.ed.ac.uk

Evgenij Belikov
EPCC
The University of Edinburgh, UK
E.Belikov@epcc.ed.ac.uk

Abstract*—In this paper we present intra-node bandwidth measurements on ARCHER2 (AMD Rome) and LUMI (AMD Milan) using the open-source CAMP (Configurable App for Memory Probing) tool, which is a configurable micro-benchmark that allows varying operational intensity, thread counts and placement, and memory access patterns including contiguous, strided, various types of stencils, and random. We also gather information on power consumption from the Slurm batch scheduler to correlate it with the access patterns used. For comparison, we run another set of the measurements on a node on NEXTGenIO (Intel Ice Lake). Additionally, we report on work to extend CAMP to increase its resolution so that we can assess the range of operational intensities between zero and two in more detail compared to previous results. Moreover, we illustrate the mechanism for using custom kernels in CAMP using a dot product calculation as an example. Our results confirm and extend previous results showing that maximum bandwidth is reached using a fraction of threads compared to the maximum number of available cores on a node. In particular, for memory access with a stride of four and for a contiguous access case, we observe up to 11% higher bandwidth using 16 threads compared to the full node using 128 cores on an ARCHER2 node and up to 15% on LUMI, especially for operational intensities below 0.5. This suggests that underpopulation may be a viable option to achieve higher performance compared to full node utilisation and thus the results suggest that benchmarking should include tests using only a fraction of available cores per node. Additionally, sub-NUMA-node awareness may be required to reach the highest performance.

***Index Terms*—Memory bandwidth, Benchmarking, Performance, ARCHER2, LUMI**

I. Introduction

In his recent Turing Award lecture, Jack Dongarra has traced the co-evolution of mathematical software and High-Performance-Computing (HPC) hardware showing that *machine balance* has shifted significantly in favour of compute power with some architectures now exhibiting the balance of over 100 double-precision floating point operations per memory read operation [1]. This continuing trend leads to a situation where CPUs can be significantly under used computationally and many scientific applications that were considered compute-bound in the past have now become memory-bound with operational intensity below one. Moreover, hardware has become increasingly more hierarchical with non-uniform memory access (NUMA) [2] since the end of Dennard Scaling due to physical limitations. Thus memory access constitutes a key intra-node bottleneck and merits further study.

In this paper we present intra-node bandwidth measurements on ARCHER2 (AMD Rome), LUMI (AMD Milan) and NEXTGenIO (Intel Ice Lake) using the open-source CAMP (Configurable App for Memory Probing) tool [3]. CAMP is a configurable micro-benchmark that allows varying *operational intensity* (i.e. number of double-precision floating point operations performed per byte transferred between CPU and memory), thread counts and placement, and memory access patterns including contiguous, strided, various types of stencils, and random. We also gather information on power consumption from the Slurm batch scheduler to correlate it with the access patterns used. We have extended CAMP to increase the resolution of the benchmarks so that we can assess the range of operational intensities between zero and two in more detail compared to previous results. Moreover, we illustrate a new mechanism for using custom kernels in CAMP using dot product as an example. Our results confirm and extend previous results [3]–[5] showing that maximum bandwidth is reached using a fraction of threads compared to the maximum number of available cores on a node. In particular, for memory access with a stride of four and for a contiguous access case, we observe up to 11% higher bandwidth using 16 threads compared to the full node using 128 cores on an ARCHER2 node for operational intensities below 0.5 and up to 15% on LUMI for operational intensities below 1. These results support the view that underpopulating a node may be of benefit for performance without harming energy efficiency. Our study is aimed at CPU-based systems, although we are considering extending CAMP to include GPUs in the future. The main focus in this paper is on the relative increase in bandwidth observed across different access patterns and levels of operational intensity when using fewer threads than the available cores.

The remainder of the paper is structured as follows. Section II summarises related work which provided the inspiration for the features combined in CAMP and mentions several recent memory bandwidth measurement studies. Section III briefly describes CAMP, its configuration and providing custom kernels, with focus on changes compared to the original version [3]. Next, Section IV provides details of the experimental setup including hardware and software configurations and the description of our methodology. We present our results and analysis in Section V and conclude in Section VI.

II. Related Work

CAMP [3] combines several features and borrows some implementation techniques from other benchmarks and tools, in particular STREAM, adept, and the Empirical Roofline Toolkit, complementing them and providing memory bandwidth measurements for various operational intensities, thread counts and memory access patterns along with visualisation scripts in a convenient way. STREAM [6], [7] and its variants are the industry standard for measuring sustained memory bandwidth for four kernels: scale, add, copy and triad using the contiguous access pattern. The results are more useful than the theoretical peak values from hardware specifications and can act as a memory roof in the Roofline model [8]. Additionally, STREAM allows the user to define their own kernel, which offers a way to change access pattern, but this requires some implementation effort. The adept benchmarks [9] allow changing memory access pattern to strided, stencil or random, among other features, which allows the user to more closely investigate the behaviour of a given kernel. Unfortunately, neither of these benchmarks allows the specification of different levels of operational intensity for testing without additional programming effort. A tool that allows the tuning of operational intensity is Empirical Roofline Toolkit [10] via an approach based on preprocessor macros and requires recompilation for each operational intensity level. Other tools exist such as `lmbench` [11], the `likwid` tool suite [12], `BenchIT` [13], and Parallel Research Kernels [14], which can potentially be used to assess memory bandwidth and latency. To the best of our knowledge, their use in the literature is primarily focused on architectural comparisons and programming models [15], rather than finding the best configuration on a given architecture for a given kernel. Additionally, hardware performance counters such (e.g. via PAPI [16] or `likwid-perfctr`) can be used, however relevant counters (e.g. memory traffic from last level caches) are often not available on production systems.

In a recent study, Velten et al. [4] measure memory characteristics on AMD EPYC and Intel Cascade Lake SP servers. With respect to memory bandwidth on AMD Rome, the authors report results from using STREAM via `BenchIT` and find that using a subset of cores by spreading a small number of threads across sub-NUMA regions (CCX or Core Complex, see IV-A for a discussion of the Rome architecture; a more detailed description can be found in the aforementioned paper [4]) can deliver the best performance for memory-bound kernels. Saini at al. [5] also study AMD Rome and Intel Cascade Lake performance. STREAM results on AMD Rome show that highest bandwidth is reached using 16 cores, showing saw-tooth bandwidth curves with peaks at multiples of eight, which we have also observed in previous work [3]. It is thus of interest to assess more recent AMD Milan and Intel Ice Lake with respect to memory behaviour.

III. CAMP

CAMP is a micro-benchmark that allows convenient bandwidth measurement whilst varying operational intensity and thread placement. Below we briefly summarise its extensions and illustrate how to add and use a custom kernel using dot product as an example. The new way to configure runs is discussed in Section IV-B. CAMP is open-source and is available on GitHub (https://github.com/CAMP-benchmark/CAMP).

A. Extensions

In this work, we extend CAMP in several ways. First, we enable more fine-grained stepping when varying operational intensity moving from approximately 0.08 to approximately 0.02. This is achieved by using a kernel that is similar to STREAM add, but with additional stores for the result. Next, to enable the use of vectorisation within benchmarks we switched CAMP's innermost loop to a macro-based implementation very similar to the Empirical Roofline Tool. This resulted in the addition of an outer-level driver written in Python, which is responsible for applying the configuration and recompiling the CAMP executable for different macro settings. However, the original way of invoking CAMP directly is also still supported. Please refer to the CAMP paper for more implementation details and description on the legacy way of running it [3].

B. Using a Custom Kernel

To evaluate the underpopulation effects of different kernels, we provide a series of pre-implemented kernels which have different operational intensity and access patterns. However, we are aware that synthetic kernels are insufficient to mimic the behaviour of all kernels of interest from real application. We have thus enabled users to be able to include their own custom kernels within CAMP. This functionality allows users to implement kernels which act exactly like their target program, for example, mimicking how data is structured, and how operands are fetched and computed, and then plug this kernel into CAMP to investigate how underpopulation, thread counts and thread placements affect the performance of that kernel. This is illustrated in Figure 1 for the dot product kernel that we will use as an example below. The results indicate that highest bandwidth is achieved when using 64 threads.

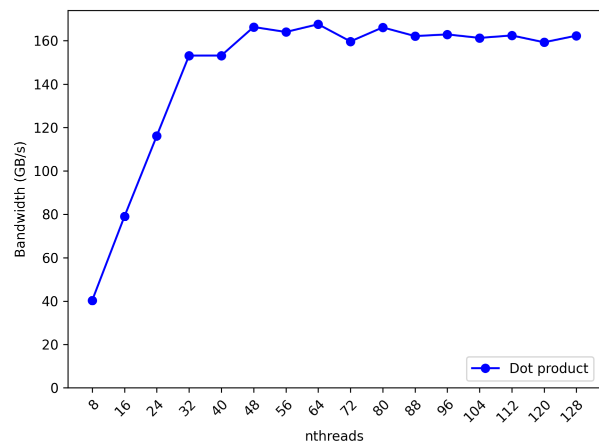


Fig. 1: Bandwidth of the *dot product* kernel on ARCHER2

The custom kernel is included in CAMP during the linking process of building the benchmark framework. The source code of the custom kernel is compiled into an object file, which will then be linked with all other object files from CAMP. We provide three function interfaces, which enables the linker to find functions with the same signature in the kernel source file and then link them to the function calls in CAMP. The three interfaces are responsible for pre-processing, running the kernel and post-processing. When run, the CAMP process creates threads and sets up thread affinity and then calls these three functions, which are initially executed on a master thread until encountering parallel regions. For example, in the dot product kernel implemented in CAMP, we allocate two vectors and initialise them in the pre-processing phase. Then in the running phase, the kernel calculates the dot product of the two vectors. In the post-processing phase the vectors are deallocated to avoid memory leaks (see Algorithm 1).

Algorithm 1 CAMP custom kernel (pseudocode)

```
procedure CAMP_PREPROCESS
    #pragma omp parallel default(none) shared(a, b) {
    int tid = omp_get_thread_num();
    a[tid] = malloc(size);
    b[tid] = malloc(size);
    for i in range(1, size) do
        a[tid][i] = 1.0;
        b[tid][i] = 2.0;
    end for
    }
end procedure

procedure CAMP_KERNEL
    #pragma omp parallel default(none) shared(a, b) reduction(+:sum) {
    int tid = omp_get_thread_num();
    for i in range(1, size) do
        sum += a[tid][i] * b[tid][i];
    end for
    }
end procedure

procedure CAMP_POSTPROCESS
    #pragma omp parallel default(none) shared(a, b) {
    int tid = omp_get_thread_num();
    free(a[tid]);
    free(b[tid]);
    }
end procedure
```

The source file name that contains the custom kernel implementation is specified in the configuration file for the compiler and linker to use (see Section IV-B). As the dot product kernel will have fixed operational intensity, the user should specify a single operational intensity value to run CAMP with that specific kernel.

IV. Experimental Setup

A. Hardware and Software

We evaluated the latest version of CAMP on the ARCHER2, LUMI and NEXTGenIO systems. We summarise the hardware and software setup in Table I. You can refer to Velten et al. [4] for a more detailed description of the AMD Rome architecture, including a discussion of memory latency measurements. We omit details on the interconnect between the nodes for each benchmark system as we are focused on intra-node memory bandwidth measurements.

A node of the UK national supercomputing service ARCHER2 consists of two 2nd generation AMD EPYC 7742 (Rome) processors [17]. Each processor has 64 cores with three supported frequency settings (e.g. as provided to the `srun` command via `--cpu-freq` option): 1.5, 2.0 and 2.25 GHz with potential for Turboboost up to 3.4GHz. There are 8 NUMA regions forming a deep memory hierarchy as illustrated in Figure 2, each of which is split into two Core Complex Dies (CCD) which in turn contain two Core Complexes (CCX). Each CCX consists of 4 cores with private L1 and L2 caches of 32KB and 512KB, respectively. The 16MB of L3 cache are shared by the cores in a CCX. Standard nodes on ARCHER2 have 256 GB DDR4 3200 RAM (ca. 2GB per core) supported by 8 memory channels per socket organised around a single I/O die. We use the HPE Cray programming environment (CCE 11.0.4, PrgEnv-cray) to compile CAMP with OpenMP and optimisations and vectorisation turned on (`-O2 -march=znver2 -Rpass=loop-vectorize -fopenmp -mllvm -force-vector-interleave=8`). The operating system (OS) is the HPE Cray Linux Environment (based on SUSE Linux Enterprise Server 15). We use Slurm to submit jobs and avoid hyperthreading by providing the `--hint=nomultithread` option to `srun`. The jobs are given exclusive use of the node on ARCHER2.

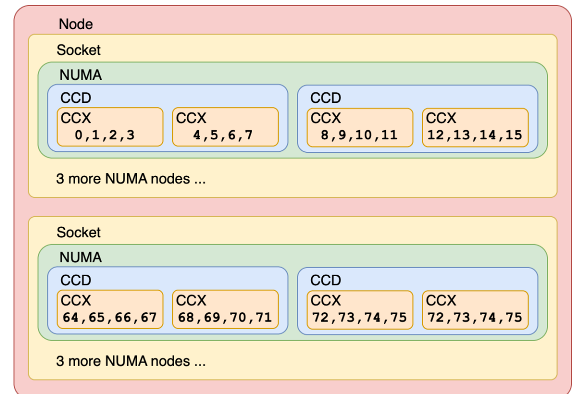


Fig. 2: Hierarchy of an ARCHER2 node [3]

For LUMI we use a node of the LUMI-C partition, the CPU-only part of the system. This is equipped with two AMD EPYC 7763 processors, each with 64 cores, and running at a base frequency of 2.45 GHz. There is 128 GB of memory per

node, and we utilise the HPE Cray programming environment. On LUMI, this is the 15.0.0 version of the PrgEnv-cray environment, leveraging Clang for the C and C++ compilers. The same compilation flags (omitting `-march`) and Slurm configuration as the ARCHER2 experiments are employed when building and running CAMP on LUMI.

On the NEXTGenIO system, we use a node equipped with dual Intel Ice Lake (Xeon Gold 6330) processors running at 2 GHz with 256 GB of RAM and 42MB level 3 cache shared among 28 cores. It offers an interesting point of comparison as the memory hierarchy is less deep than in the AMD systems, and the last level cache is shared among a larger number of cores. The system uses CentOS 7.9 and provides Intel's oneAPI compiler suite (version 2023.0.0).

TABLE I: Hardware and Software configurations

	ARCHER2	LUMI	NEXTGenIO-icx
Hardware			
Architecture	AMD Rome	AMD Milan	Intel Ice Lake
Model	EPYC 7742	EPYC 7763	Xeon Gold 6330
Cores per node	128 (2x64)	128 (2x64)	56 (2x28)
CPU freq. (GHz)	2.25	2.45	2.00
L1$ (KB)	32	32	48
L2$ (KB)	512	512	1280
L3$ (MB)	16	256	42
RAM (GB)	256	128	256
Software			
OS	SLES 15	SLES 15	CentOS 7.9
Compiler	PrgEnv-cray	PrgEnv-cray	Intel oneAPI 2023
Python	3.8.5	3.9.13	3.8.5
numpy	1.24.2	1.20.0	1.24.2
pandas	1.5.3	1.3.4	1.5.3
matplotlib	3.7.1	3.7.1	3.7.1

Furthermore, Python 3 is required for running the outer-level driver script along with the commonly used libraries numpy, pandas and matplotlib for analysis and visualisation.

B. CAMP Configuration

We extend the original CAMP to use a configuration file to set compiler flags and specify operational intensity steps, thread placement and count, data size, number of runs and memory hierarchy, among other configuration parameters in a convenient manner as key-value pairs separated by a space. The most prominent settings are summarised in Table II.

The path to the configuration file is passed as an argument when running the outer-level CAMP driver script. For a more detailed example please refer to the code repository. Note that the `FLOPS` setting controls the number of times a macro is applied to increase the innermost operational intensity for a kernel, and each value in the comma-separated list corresponds to a run of a set of CAMP measurements based on the number of threads specified. Also note that the `HIERARCHY` setting affects thread placement: for instance omitting sub-NUMA level on ARCHER2 and only specifying `128,64,16,8` would potentially allow threads inside a CCD to migrate between cores.

TABLE II: CAMP configuration options

Option	*Type*	*Function*
`RESULTS`	string	Results directory
`CC`	string	C compiler
`CFLAGS`	string	Compiler flags
`KERNEL`	string	Kernel file to use (see Section III-B)
`FLOPS`	list of ints	Settings for operational intensity
`OPENMP_THREADS`	list of ints	Thread counts of interest
`MEM`	long int	Array size (number of doubles)
`REPEAT`	int	Number of runs for a data point
`PATTERN`	string	e.g. contig, stride4, stencil5, random
`PLACEMENT`	string	Thread placement e.g. cyclic
`HIERARCHY`	list of ints	e.g. `128,64,16,8,4` on ARCHER2
`SCALING`	string	strong or weak

C. Methodology

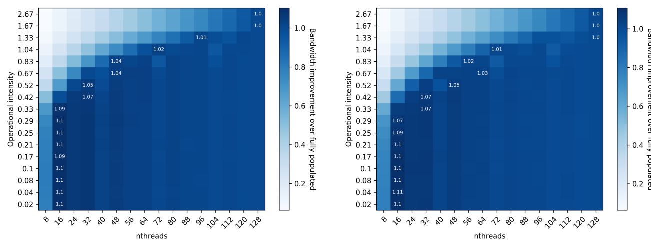
CAMP uses OpenMP's `omp_get_wtime()` function to measure runtime and calculates bandwidth by dividing the configured data size in bytes by the runtime. Although CAMP supports the use of hardware performance counters via PAPI, the relevant data traffic related counters were not available on some of the systems we were benchmarking, so that this feature has not been used. If not otherwise stated, we have performed 10 runs for each data point and report the best value. For our experiment we are using a strong scaling setup where the data size is fixed as the number of threads is increased. We ensure that the data size used is large enough that it significantly exceeds the cumulative last level cache size on each system. Threads are pinned to cores in a cyclic fashion, taking into account the hierarchy configuration to ensure a sparse distribution. Raw data is retained in a CSV file, allowing users to perform custom analyses separately. The variation in our measurements for each experiment was evaluated and is low enough to not impact any conclusions.

V. Results and Analysis

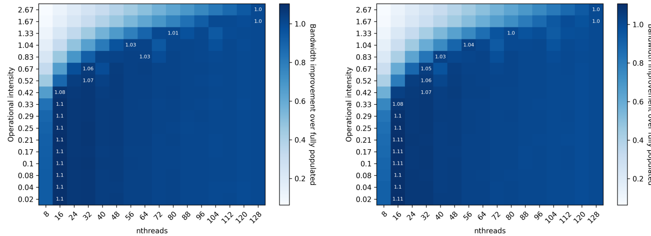
In this section we compare the results obtained on our three experiment systems for the contiguous, strided (with a stride of four) and 5-cell stencil patterns. We focus on relative changes in achieved bandwidth as operational intensity and thread counts are varied, with darker squares indicating higher bandwidth, whilst light squares indicating reduced bandwidth. Note that the y-axis is not scaled uniformly for the heatmaps as we wish to focus on operational intensities below one.

Figure 3 depicts relative bandwidth for contiguous (left column) and strided (right column) access patterns on ARCHER2 also varying the CPU frequency setting. We observe that highest bandwidth is achieved when using substantially fewer threads than cores available, especially for operational intensities below 0.5. This pattern is consistent and confirms previous results [3]–[5].

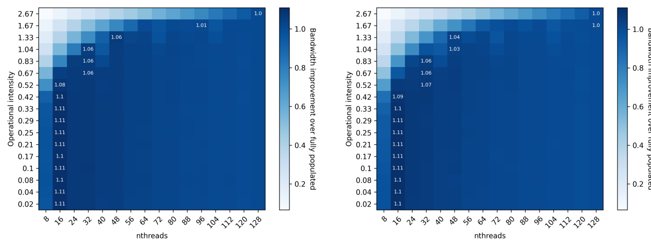
In most cases highest bandwidth is achieved using only 16 out of 128 threads and the bandwidth is higher by 9-11% on average. Additionally, we notice that increasing CPU frequency appears to strengthen this effect so that it persists for somewhat higher operational intensities. This is most strongly visible in the case of the stencil pattern, where spatial and



(a) Contiguous (1.50 GHz) (b) Stride4 (1.50 GHz)

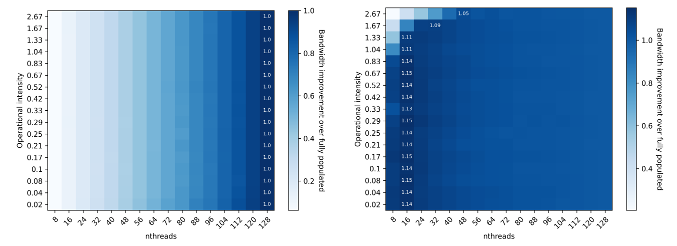


(c) Contiguous (2.00 GHz) (d) Stride4 (2.00 GHz)

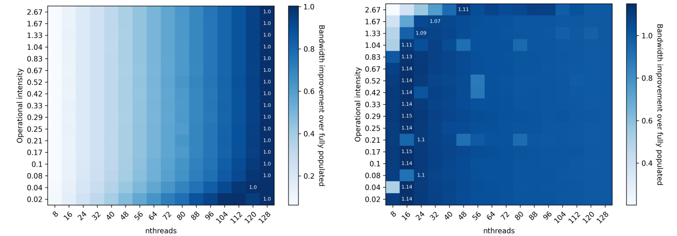


(e) Contiguous (2.25 GHz) (f) Stride4 (2.25 GHz)

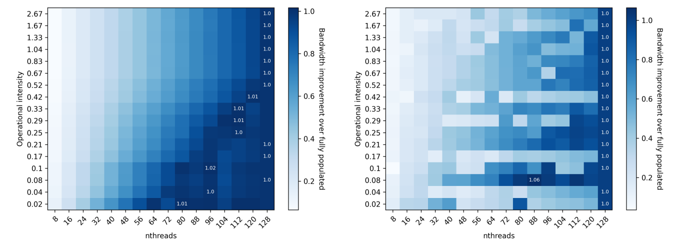
Fig. 3: Bandwidth relative to full node (darker is higher) on ARCHER2 (CPU frequency setting in parentheses) for Contiguous and Stride4 access patterns



(a) Stencil5 (1.50 GHz) (b) Contiguous



(c) Stencil5 (2.00 GHz) (d) Stride4



(e) Stencil5 (2.25 GHz) (f) Stencil5

Fig. 4: Bandwidth relative to full node (darker is higher) for Stencil5 on ARCHER2 (left column, CPU frequency setting in parentheses) and various patterns on LUMI (right column)

temporal locality is increased through reuse of cells already in the cache. In the left column of Figure 4 we can see how the effect is invisible for the lowest CPU frequency setting, barely visible for the default setting of 2.00 GHz, but appears more clearly for the fastest setting, although the increase in bandwidth is modest with 1-2% when using between 80 and 120 threads for operational intensities below 0.4.

The effect may seem less pronounced if the hierarchy setting is sub-optimal, e.g. by omitting sub-NUMA level from placement consideration, as shown in Figure 7d for strided access. The best bandwidth is now mostly achieved using 24-32 threads, rather than 16, with lower bandwidth increase of around 6-7% on average as opposed to Figure 3f. Additionally, we note that as expected the experiments using lower CPU frequency setting took longer to run and also consumed more power (data not depicted).

Figure 5 presents absolute bandwidth measured for the contiguous case with the default CPU frequency setting on ARCHER2, comparing full node against the best underpopulated value for the given operational intensity. We observe that the difference is largest for operational intensities below 0.5, whilst for the intensity of 2.67 the kernel seems no longer-memory-bound.

The right column of Figure 4 presents the bandwidth achieved on LUMI for the three chosen access patterns. The

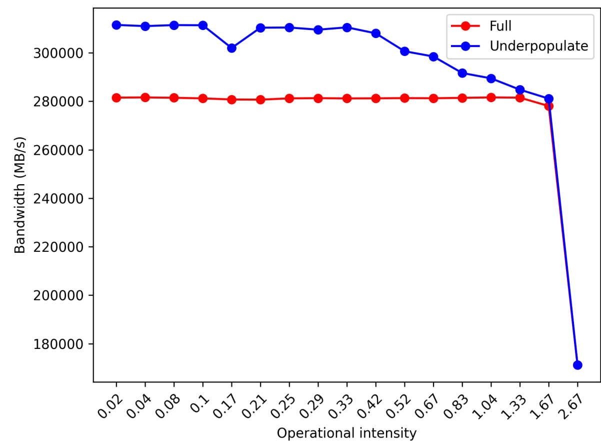


Fig. 5: ARCHER2: Best bandwidth value compared with fully populated (contig; note the y-axis does not include zero for readability)

underpopulation effect appears stronger than on ARCHER2 with ca. 14% average bandwidth increase when using 16 threads for operational intensities below one for contiguous and strided access (with a stride of 4). For the 5-cell stencil the measurements seem more erratic, although best performance is in line with our expectations. However, the reason for the patchy mid-range measurements is, as yet, unclear.

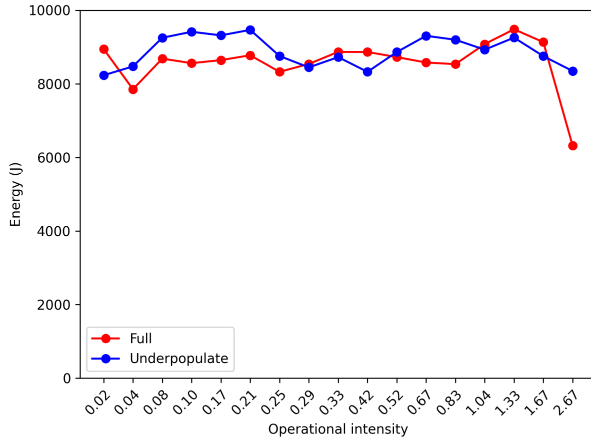
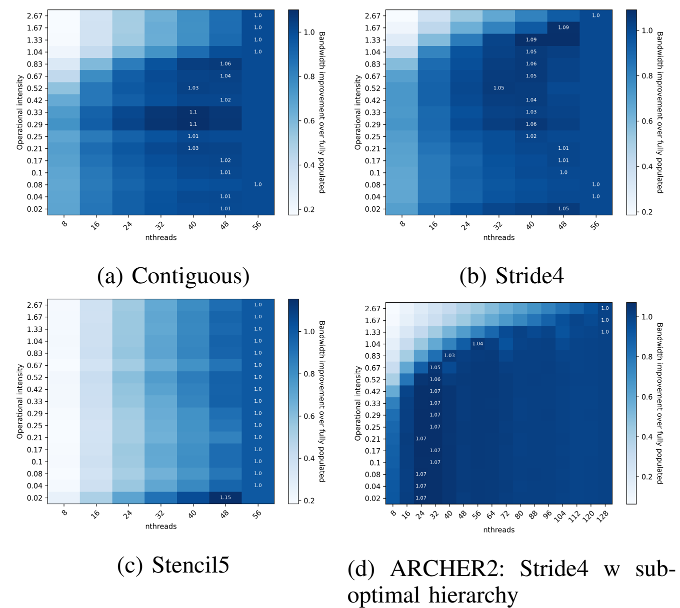


Fig. 6: ARCHER2: Energy consumption (contig 2.0GHz)



(a) Contiguous)

(b) Stride4

(c) Stencil5

(d) ARCHER2: Stride4 w sub-optimal hierarchy

Fig. 7: Bandwidth relative to full node (darker is higher) for various patterns on NEXTGenIO Ice Lake node and a Stride4 with sub-optimal hierarchy setting on ARCHER2

Figure 6 presents some data on energy consumption for the contiguous access pattern using the default CPU frequency setting on ARCHER2. Overall, the energy consumption for the full node and for the underpopulated case appear relatively similar. We hypothesise that when only small numbers of threads are used, Turboboost reaches highest levels for the best underpopulated runs (i.e. less than or equal to 3 cores used per CCD) which offsets potential energy savings. It is also possible that the granularity of results collected here are insufficient for strong conclusions to be drawn.

Finally, we assess bandwidth on NEXTGenIO as illustrated in Figure 7. As we expected, the effect is much less pronounced as the Intel Ice Lake architecture is less hierarchical compared to AMD Rome and Milan. Although we observe up to 10% improved bandwidth, the improvement happens in fewer cases and using higher fraction of total of 56 cores: e.g. with 40 threads for the contiguous access pattern and operational intensity of around 0.3 and with 32 threads for the access pattern using stride of 4 for operational intensity around 0.5. Additionally, it is not clear why the effect appears weaker for the lowest levels of operational intensity.

VI. Conclusion

Using CAMP we provide an assessment of intra-node memory bandwidth on ARCHER2 (AMD Rome), LUMI (AMD Milan) and NEXTGenIO (Intel Ice Lake) with focus on relative bandwidth increases due to underpopulation for operational intensities in range between zero and two. Our results confirm and extend previous results [3]–[5] showing that maximum bandwidth is reached using a fraction of threads compared to the maximum number of available cores on a node. In particular, for memory access with a stride of four and for a contiguous access case, we observe up to 11% higher bandwidth using 16 threads compared to the full node using 128 cores on an ARCHER2 node for operational intensities below 0.5 and up to 15% on LUMI for operational intensities below 1. On NEXTGenIO the effect is less pronounced with up to 10% improved bandwidth but in fewer cases and using a larger fraction of available cores, as the architecture is less hierarchical. These results support the view that underpopulating a node may be of benefit for performance without harming energy efficiency. By gathering some data on energy consumption we notice that underpopulation does not necessarily result in reduced energy consumption and additionally using lower CPU frequencies did not show any benefit for the access patterns chosen, whilst the underpopulation effect appeared stronger for higher CPU frequencies. We also emphasise the subtlety of pinning at sub-NUMA-region level on systems with deep memory hierarchies, which may be easy to overlook.

In the future, we envision adding GPU support to CAMP to study heterogeneous systems. Additionally, we would like to develop a library of kernels, covering various common linear algebra subroutines and relevant data structures and to apply CAMP for performance modelling.

Acknowledgments

This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk), the NEXTGenIO system (http://www.nextgenio.eu) and LUMI (https://www.lumi-supercomputer.eu/). The NEXTGenIO system was funded by the European Union's Horizon 2020 Research and Innovation program under Grant Agreement no. 671951,

and supported by EPCC, The University of Edinburgh. We acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium through a EuroHPC Regular Access call.

REFERENCES

[1] J. J. Dongarra, "The evolution of mathematical software," *Communications of the ACM*, vol. 65, no. 12, pp. 66–72, 2022.

[2] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors." *ACM Queue*, vol. 11, no. 7, pp. 40–51, 2013.

[3] W. Peng and E. Belikov, "CAMP: a synthetic micro-benchmark for assessing deep memory hierarchies," in *2022 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2022, pp. 28–36.

[4] M. Velten, R. Schöne, T. Ilsche, and D. Hackenberg, "Memory performance of AMD EPYC Rome and Intel Cascade Lake SP server processors," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 165–175.

[5] S. Saini, J. Baron, J. Chang, R. Hood, and H. Jin, "Performance evaluation of a supercomputer based on AMD Rome and Intel Cascade Lake processors," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 848–859.

[6] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Comp. Soc. Technical Committee on Computer Architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[7] L. Bergstrom, "Measuring NUMA effects with the STREAM benchmark," *arXiv preprint arXiv:1103.3225*, 2011.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] N. Johnson, "Adept deliverable D2.3 - updated report on Adept Benchmarks," 2015, https://github.com/EPCCed/adept-micro-omp, Last accessed: 2022-03-25.

[10] Y. J. Lo, S. Williams, B. V. Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 129–148.

[11] C. Staelin, "lmbench: an extensible micro-benchmark suite," *Software: Practice and Experience*, vol. 35, no. 11, pp. 1079–1105, 2005.

[12] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing workshops*. IEEE, 2010, pp. 207–216.

[13] G. Juckeland, S. Börner, M. Kluge, S. Kölling, W. E. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and R. Wloch, "BenchIT – performance measurement and comparison for scientific applications," in *Advances in Parallel Computing*. Elsevier, 2004, vol. 13, pp. 501–508.

[14] R. F. Van der Wijngaart and T. G. Mattson, "The Parallel Research Kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[15] J. R. Hammond and T. G. Mattson, "Evaluating data parallelism in C++ using the Parallel Research Kernels," in *Proceedings of the International Workshop on OpenCL*, 2019, pp. 1–6.

[16] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. of the department of defense HPCMP users group conference*, vol. 710. Citeseer, 1999.

[17] D. Suggs, M. Subramony, and D. Bouvier, "The AMD "Zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.