Influence of architectural features of the SNC-4 mode of the Intel Xeon Phi KNL on matrix multiplication

Ruben Laso[1] [0000−0003−2574−4025], Francisco F. Rivera[1] [0000−0002−6728−9350], and J. C. Cabaleiro[1] [0000−0002−5674−5162]

Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, Spain
{r.laso,ff.rivera,jc.cabaleiro}@usc.es

Abstract. The Sub-NUMA Clustering 4 (SNC-4) affinity mode of the Intel Xeon Phi Knights Landing introduces a new environment for parallel applications, that provides a NUMA system in a single chip.
The main target of this work is to characterize the behaviour of this system, focusing in nested parallelization for a well known algorithm, with regular and predictable memory access patterns as the matrix multiplication. It has been studied the effects of thread distribution in the processor on the performance when using SNC-4 affinity mode, the differences between cache and flat modes of the MCDRAM and the improvements due to vectorization in different scenarios in terms of data locality.
Results show that the best thread location is the scatter distribution, using 64 or 128 threads. Differences between cache and flat modes of the MCDRAM are, generally, not significant. The use of optimization techniques as padding to improve locality has a great impact on execution times. Vectorization resulted to be efficient only when the data locality is good, specially when the MCDRAM is used as a cache.

Keywords: Intel Xeon Phi KNL · SNC-4 · MCDRAM · vectorization

1 Introduction

Manycore architectures as the presented by the Intel Xeon Phi Knights Landing (KNL) provide highly parallel environments with a large number of cores in a single chip, allowing developers to exploit parallelism in their algorithms. In the case of the Intel Xeon Phi KNL, the second generation of the Intel's manycore processors, the most interesting features are the clustering modes and the integrated on-package memory known as MCDRAM (Multi-Channel DRAM).

This work is focused on the SNC-4 (Sub-NUMA Clustering 4) mode, in which the chip is partitioned into four nodes, being considered as a NUMA system. However, this configuration differs from typical NUMA systems in latency and bandwidth. Given that, a characterization of the KNL behaviour in its SNC-4 configuration can be useful for its users. In this work it is also included a brief

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_41

comparison between the cache and flat modes of the MCDRAM memory and a study of the performance improvement achieved using vectorization.

To study these elements, different implementations of the classic matrix multiplication have been used, as this algorithm has some interesting properties that fit the goals of this work. It is a well known code and its memory access patterns are predictable, regular and easy to modify by changing the order of the loops, providing different scenarios in terms of data locality. Note that it is not intended to optimize this code, like in [3] or [7], but to use it just as a case study.

Other works have shown studies with very specific benchmarks, testing the MCDRAM in its different modes [12] or obtaining models of its behaviour [10]. Other authors have used benchmarks with dense matrix multiplication of small dimensions [4] or have looked for the roofline model using benchmarks based on sparse matrices algebra [2]. Finally, works like [6] show performance comparison on commonly used software. In contrast, this paper is intended to present a general study with different conditions, as the given when optimizing and parallelizing a code with regular memory patterns as the matrix multiplication.

2 Intel Xeon Phi KNL architecture and benchmarks

The Intel Xeon Phi is a manycore processor which bases its architecture in tiles [5]. Each tile has two cores and a 1 MB shared L2 cache memory. Each core has two VPUs (Vector Processing Unit), compatible with the AVX-512 instructions [11], and it is capable of execute up to four threads simultaneously. With a maximum of 36 tiles, the KNL can have up to 72 cores, 144 VPUs, and can execute up to 288 threads concurrently.

This processor has another singular features, like the clustering modes and the MCDRAM memory [13]. Concerning the clustering modes, three main configurations are available: All-to-all, Quadrant and SNC-4. The most interesting one is the SNC-4, where the chip behaves like a singular NUMA system. Additionally, the MCDRAM memory is an integrated high-bandwidth memory (up to 450 GB/s) with 16 GB of capacity that has three different configurations: cache, flat and hybrid modes. Using the cache mode, the MCDRAM behaves as a L3 direct-mapped cache memory. In the flat mode, the MCDRAM is configured as another main memory. In the hybrid mode, the memory is divided 50%-50% or 75%-25% in cache and flat modes, respectively. Note that using SNC-4 and flat modes, there are four NUMA nodes that correspond to the cores and the main memory, and four (of 4 GB each) corresponding to the MCDRAM.

In this study, the parallel (using OpenMP [1]) dense matrix multiplication has been used, $C = A \times B$, $A, B, C \in \mathcal{M}_{n \times n}$. The matrices are located in the MCDRAM when using flat mode, and in DDR4 memory with cache mode. The following nests in the loops has been studied: ijk, ikj, jik and jki. E.g. the ijk is referred as the one which has the i index in the outer-most loop, the index j in the intermediate one, and the k in the inner-most. The matrices elements accessed are always a_{ik}, b_{kj}, c_{ij}. The k loop cannot be parallelized due to race conditions, so nested parallelization has been tested only with ijk and jik nests.

In terms of data locality, the ikj loop should have the best performance because it accesses all the matrices by rows (the best way in C language). The ijk nest accesses the elements of A and C by rows and B by columns, being moderately efficient. The jik nest gets the elements of the matrices B and C by columns, and the elements of A by rows. Finally, the jki order should show the worst performance as all the matrices are accessed by columns.

Given the structure of the code and the organization of the cores in the SNC-4 mode, it is interesting to use a two-level parallelization. In the outer-most loop, iterations are distributed to different thread groups. In the second loop, iterations are shared out to the threads of each group. In the inner-most loop, vectorization can be applied using AVX-512 instructions. The natural distribution, given the architecture of the KNL, would be 4 groups of 64 threads, denoted as 4×64 from now on. This way of sharing out the iterations allows us to consider different locations for the threads of each group, scattering or compacting them across the cores, while the groups will be always scattered.

The basic algorithm where all the data are in contiguous positions, might have performance issues because of the replacements in the caches when using matrices with size $n = a2^b$, $a,b \in \mathbb{N}$, for certain values of b. To solve this problem, another version that uses padding has been implemented, so 64 bytes (the length of a cache line) are added to the end of each row of the matrices.

In the execution of the benchmarks, it has been used the Intel Xeon Phi KNL 7250 using 64 cores and 48 GB of DDR4 memory. The compiler used in the tests was the Intel ICC 18.0.0 with the options `-qopenmp`, `-O2`, `-xMIC-AVX512` and `-vec-threshold0` to ensure that the AVX-512 instructions were used. Other optimization options might transform the code too much, so the memory access patterns might not be the ones written in the source file. Metrics about cache misses and memory performance have been obtained with Intel VTune 2018.

3 Results

This section shows the results of the benchmarks with matrices of dimension $n = 4096$. Other sizes were tested too, achieving similar results.

3.1 Nested parallelism

A brief summary of the results for the codes using nested parallelization and 4 groups of 16 threads each (4×16) are shown in Tables 1 and 2, where each entry contains the execution times (in seconds) of the sequential and parallel executions and the corresponding speedup.

Padding: The programs that do not use padding show higher execution times than those which implement this technique as it reduces the L2 cache misses up to a 98%. The ijk code shows an improvement of 97% in the sequential execution time, while jik improves up to 98%, showing the relevance of this kind of techniques in the KNL, improving drastically the execution times and changing the behaviour in terms of thread location and vectorization.

Table 1: Summary of the results obtained with the benchmarks without padding.

		Cache, scatter	Cache, compact	Flat, scatter	Flat, compact
ijk	vect.	7586.96 – 238.26 (×31)	7586.93 – 106.29 (×71)	3720.43 – 155.28 (×23)	3720.37 – 144.89 (×25)
	no vect.	3720.42 – 72.90 (×51)	3683.96 – 95.26 (×38)	3720.62 – 154.57 (×24)	3721.08 – 159.03 (×23)
jik	vect.	7601.64 – 1971.87 (×3.8)	7736.51 – 485.35 (×15)	3721.72 – 581.76 (×6.3)	3731.53 – 2158.89 (×1.7)
	no vect.	3729.80 – 531.64 (×7.0)	3965.88 – 302.08 (×13)	3731.37 – 781.83 (×4.7)	3731.61 – 2563.39 (×1.4)

Table 2: Summary of the results obtained with the benchmarks with padding.

		Cache, scatter	Cache, compact	Flat, scatter	Flat, compact
ijk	vect.	247.91 – 6.96 (×35)	242.34 – 91.65 (×2.6)	248.94 – 28.16 (×8.8)	243.80 – 88.76 (×2.7)
	no vect.	404.73 – 7.96 (×50)	400.84 – 83.86 (×4.7)	399.47 – 45.18 (×8.8)	401.81 – 80.67 (×4.9)
jik	vect.	176.25 – 4.93 (×35)	178.17 – 12.58 (×14)	181.18 – 7.08 (×25)	175.72 – 12.17 (×14)
	no vect.	330.00 – 5.28 (×62)	331.51 – 19.24 (×17)	328.87 – 11.42 (×28)	330.05 – 16.91 (×19)

Thread location: The results of the programs which do not use padding show that the scatter distribution is not always the best. A compact distribution can cause the threads of the same core/tile share cache lines, reducing the traffic in the mesh. This situation also implies a faster communication in comparison with sharing data with external tiles. In addition, a compact distribution divides the resources of each core, probably causing a lower performance. Also, it can saturate the buses of the caches or the interconnection mesh, degrading the execution times, as in the *jik* nest with the MCDRAM in flat mode.

Vectorization: The codes that do not use padding present poor performance results when using vectorization. However, when padding is applied, improvements up to 54% are obtained, showing that vectorization is only beneficial when the programs have efficient data access patterns.

MCDRAM: Using the MCDRAM in cache mode and having inefficient access patterns causes important performance problems in the vectorized programs, due to the difficulties in feeding the registers. Note that these performance issues do not appear with the flat mode. In contrast, cache mode shows a slightly better performance in parallel executions.

Best execution: The best execution time obtained using nested parallelism has been 2.66 seconds (51.66 GFlop/s), achieved using the *jik* nest, 64 groups of 2 threads with a compact distribution, MCDRAM in cache mode, and using padding and vector instructions.

3.2 One-level parallelism

A summary of the results obtained by the codes where only the outer loop is parallelized is shown in Tables 3 and 4, including the *ikj* and *jki* nests.

Padding: All loop nests take advantage of this technique because of the significant reduction in the number of cache misses, up to 98%.

Thread location: When only the outer most loop is parallelized, the scatter distribution gives the best execution times. The differences are specially noticeable in the *ikj* and *jki* cases using padding, improving up to 60% and 85%.

Table 3: Summary of the results obtained with the benchmarks without padding.

		Cache, scatter	Cache, compact	Flat, scatter	Flat, compact
ijk	vect.	7586.96 – 292.25 (×25)	7586.93 – 99.18 (×76)	3720.43 – 97.22 (×38)	3720.37 – 110.56 (×33)
	no vect.	3720.42 – 60.37 (×61)	3683.96 – 79.42 (×46)	3720.62 – 97.07 (×38)	3721.08 – 109.36 (×34)
jik	vect.	7601.64 – 177.20 (×42)	7736.51 – 108.61 (×71)	3731.72 – 190.99 (×19)	3731.53 – 189.86 (×19)
	no vect.	3729.80 – 97.26 (×38)	3965.88 – 101.02 (×39)	3731.37 – 188.71 (×19)	3731.61 – 191.46 (×19)
ikj	vect.	79.54 – 9.69 (×8.2)	80.68 – 5.53 (×14)	76.94 – 6.61 (×12)	76.89 – 6.41 (×13)
	no vect.	189.32 – 10.23 (×18)	189.35 – 9.66 (×19)	174.39 – 8.43 (×20)	173.38 – 8.69 (×20)
jki	vect.	8670.13 – 505.25 (×17)	8831.75 – 277.52 (×31)	8624.93 – 516.27 (×16)	8584.76 – 526.59 (×16)
	no vect.	8568.79 – 511.14 (×16)	8497.72 – 245.05 (×34)	8624.20 – 517.28 (×16)	8584.68 – 516.50 (×16)

Table 4: Summary of the results obtained with the benchmarks with padding.

		Cache, scatter	Cache, compact	Flat, scatter	Flat, compact
ijk	vect.	247.91 – 4.59 (×54)	242.34 – 4.39 (×55)	248.94 – 4.56 (×54)	243.80 – 4.75 (×51)
	no vect.	404.73 – 7.33 (×55)	400.84 – 7.65 (×52)	399.47 – 7.20 (×55)	401.81 – 7.21 (×55)
jik	vect.	176.25 – 3.32 (×53)	178.17 – 3.27 (×54)	181.18 – 3.79 (×47)	175.72 – 3.67 (×47)
	no vect.	330.00 – 5.02 (×65)	331.51 – 5.36 (×61)	328.87 – 6.54 (×50)	330.05 – 7.14 (×46)
ikj	vect.	69.75 – 2.61 (×26)	69.54 – 4.58 (×15)	69.61 – 3.67 (×18)	69.42 – 4.63 (×14)
	no vect.	180.51 – 3.92 (×46)	185.23 – 10.80 (×17)	179.36 – 4.08 (×43)	180.27 – 10.77 (×16)
jki	vect.	538.61 – 22.70 (×23)	551.16 – 148.99 (×33)	535.75 – 100.78 (×5.3)	534.93 – 158.36 (×3.3)
	no vect.	550.08 – 22.73 (×24)	545.46 – 147.59 (×26)	535.11 – 98.26 (×5.4)	536.07 – 159.72 (×3.3)

Vectorization: The *ikj* nest, the best access pattern, takes the maximum benefit of vector instructions, reducing up to 63% the execution time.

MCDRAM: Using one-level parallelism, the cache mode usually achieves a slightly better performance in parallel executions. Even though, differences between flat and cache modes are generally not significant.

Best execution: The best time obtained using one-level parallelism has been of 1.17 seconds (117.46 GFlop/s), using the *ikj* order, with padding, 256 threads, MCDRAM in cache mode, and using AVX-512 instructions.

3.3 Comparative with a real NUMA

A summary of the comparison of the Intel Xeon Phi KNL with a NUMA server that consists of four Intel Xeon E5 4620 is shown in Table 5.

Memory latencies: To study the effect of non-local accesses in the NUMA server and in the KNL, the `numademo` command [8] with the sequential STREAM benchmark [9] has been executed in both machines. Results in Table 6 show that the penalties in the NUMA server go from 28% up to 40% while in the KNL these penalties are much lower. With the MCDRAM in cache mode, the differences are between 3.8% and 6.9%. Using the flat mode, both kind of penalties, those related to the DDR memory and the MCDRAM, are around 2% and 3%.

Padding This optimization technique has a low impact in the Intel Xeon E5 in comparison with the KNL. In the tests, adding padding to the data has improved the sequential execution times up to 30%.

Vectorization: The Intel Xeon E5 is not compatible with the AVX-512 vector instructions, being only compatible with the AVX2 instruction set. This kind

Table 5: Best execution times and speedup of the KNL and NUMA server.

	Sequential		As many threads as cores		All available threads	
	KNL	NUMA	KNL (64 threads)	NUMA (40 threads)	KNL (256 threads)	NUMA (80 threads)
ijk	242.34	545.73	4.39 (×55)	28.76 (×19)	11.55 (×21)	15.04 (×36)
jik	175.72	521.63	3.27 (×53)	33.54 (×15)	2.53 (×69)	16.96 (×30)
ikj	69.42	107.43	2.61 (×26)	4.11 (×26)	1.17 (×59)	2.20 (×48)
jki	535.11	521.71	22.70 (×23)	44.11 (×11)	137.90 (×3.8)	47.92 (×10)

Table 6: Average bandwidth (in MB/s) and penalty percentage given by `numademo` with STREAM copy benchmark.

	NUMA	KNL (cache)	KNL (flat)	
			DDR4	MCDRAM
Local	10,287	9,064	8,266	9,082
Remote	7,409 (28%)	8,712 (3.8%)	8,084 (2.2%)	8,909 (1.9%)
Worst	6,142 (40%)	8,439 (6.9%)	8,066 (2.4%)	8,803 (3.0%)

of instructions works with 4 operands with FMA operations, so its performance is lower, improving just up to 9% the execution times.

Thread location: The performance shows that using more NUMA nodes is not always the best option to improve the performance, like in the Intel Xeon Phi, due to the higher penalties of remote accesses on a real NUMA server.

Best execution: In the NUMA server the best execution time has been obtained by the *ikj* nest, computing the matrix multiplication in 2.20 seconds (62.47 GFlop/s), using 80 threads with a scatter distribution, vectorization and no padding. This performace is noticeably lower than the provided by the KNL.

4 Conclusions

In this work, it has been studied the behaviour of the Intel Xeon Phi KNL in different situations, characterizing its performance in terms of thread location, MCDRAM mode, vectorization, data locality and, also, comparing the SNC-4 mode of the KNL with a real NUMA system.

Generally, the most efficient thread distribution is the scatter location, using all the NUMA nodes of the KNL, and one or two threads per core. In a real NUMA system, the behaviour is different, depending heavily on the algorithm because of the higher penalties in the communications between nodes.

The way the data is located in memory has a deep impact in the performance in the KNL compared to other processors. In this case, adding padding to the data has produced a reduction of the execution time up to 98%.

Using vector instructions has shown an irregular behaviour. With a low locality, the use of vector instructions had a negative impact. In opposition, with good data locality, their use has improved the execution times up to 60%.

Differences between cache and flat modes of the MCDRAM are, generally, not significant. Flat mode seems to perform better under inefficient data access patterns, but cache mode has usually given better results on parallel codes.

Acknowledgment

This work has received financial support from the Ministerio de Economía, Industria y Competitividad within the project TIN2016-76373-P and network CAPAP-H. It was also funded by the Consellería de Cultura, Educación e Ordenación Universitaria of Xunta de Galicia (accr. 2016-2019, ED431G/08 and reference competitive group 2019-2021, ED431C 2018/19) and network R2016/045.

References

1. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering **5**(1) (1998)
2. Doerfler, D., et al.: Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor. In: International Conference on High Performance Computing. pp. 339–353. Springer (2016)
3. Guney, M.E., Goto, K., Costa, T.B., Knepper, S., Huot, L., Mitrano, A., Story, S.: Optimizing Matrix Multiplication on Intel® Xeon Phi TH x200 Architecture. In: 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH) (July 2017). https://doi.org/10.1109/ARITH.2017.19
4. Heinecke, A., Breuer, A., Bader, M., Dubey, P.: High order seismic simulations on the Intel Xeon Phi processor (Knights Landing). In: International Conference on High Performance Computing. pp. 343–362. Springer (2016)
5. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition. Morgan Kaufmann (2016)
6. Kang, J.H., Kwon, O.K., Ryu, H., Jeong, J., Lim, K.: Performance Evaluation of Scientific Applications on Intel Xeon Phi Knights Landing Clusters. In: 2018 Int. Conf. on High Performance Computing & Simulation (HPCS). IEEE (2018)
7. Kim, R.: Implementing general matrix-matrix multiplication algorithm on the Intel Xeon Phi Knights Landing Processor. Ph.D. thesis, Department of Mathematical Sciences, Seoul National University (2018)
8. Kleen, A.: A NUMA API for Linux. Novel Inc (2005)
9. McCalpin, J.D., et al.: Memory bandwidth and machine balance in current high performance computers. IEEE computer society technical committee on computer architecture (TCCA) newsletter **1995**, 19–25 (1995)
10. Ramos, S., Hoefler, T.: Capability models for manycore memory systems: a case-study with Xeon Phi KNL. In: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International. pp. 297–306. IEEE (2017)
11. Reinders, J.: AVX-512 instructions. Intel Corporation (2013)
12. Rosales, C., Cazes, J., Milfeld, K., Gómez-Iglesias, A., Koesterke, L., Huang, L., Vienne, J.: A comparative study of application performance and scalability on the Intel Knights Landing processor. In: International Conference on High Performance Computing. pp. 307–318. Springer (2016)
13. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights Landing: Second-generation Intel Xeon Phi product. IEEE micro **36**(2), 34–46 (2016)