Sparse Matrix-Vector Multiplication on the Single-Chip Cloud Computer Many-core Processor

Juan C. Pichel and Francisco F. Rivera[a]

[a] *Centro de Investigación en Tecnoloxías da Información (CITIUS)*
Universidade de Santiago de Compostela, Spain

Abstract

The microprocessor industry has responded by turning to many-core processors, increasing parallelism as the primary method to improve processor performance. These processors are expected to consist of tens or even hundreds of cores. One of these future processors is the 48-core experimental processor Single-Chip Cloud Computer (SCC). The SCC was created by Intel Labs as a platform for many-core software research.

In this work we study the behavior of an important irregular application such as the Sparse Matrix-Vector multiplication (SpMV) on the SCC processor in terms of performance and power efficiency. In addition, some of the most successful optimization techniques for this kernel are evaluated. In particular, reordering, blocking and data compression techniques have been considered. Our experiments give some key insights that can serve as guidelines for the understanding and optimization of the SpMV kernel on this architecture.

Furthermore, an architectural comparison of the SCC processor with several leading multicore processors and GPUs is performed.

Keywords: many-core, sparse matrix, performance, power efficiency, optimization

1. Introduction

The general trend in the industry is to include a higher number of cores on the same chip, the so-called *many-core* processors. These future processors are expected to include tens or even hundreds of cores. In this paper we consider one of those experimental many-core processors, the Single-Chip Cloud Computer (SCC). The SCC is an experimental processor created by Intel Labs as a platform for many-core software research. It consists of 48 cores connected through a mesh network, four memory controllers given access to the main memory and several power management functions. These characteristics turns it into a big

Email address: juancarlos.pichel, ff.rivera@usc.es (Juan C. Pichel and Francisco F. Rivera)

challenge for researchers in order to extract performance from such complex architecture.

The objective of this work is to study the behavior of the Sparse Matrix-Vector product (SpMV) on the SCC processor, which is one of the most important computational kernels in scientific and engineering applications. It is notorious for sustaining low fractions of peak performance (typically, about 10%) on modern processors. Among the factors that have an impact on its performance, two of them are of the utmost importance: the memory bandwidth limitation and the low locality caused by the irregular and indirect memory accesses. An additional drawback is that the sparse matrix patterns that characterize such irregular accesses are only known at run-time.

In this work we present an in-depth analysis of the effects on the SpMV performance of several issues. One of them is the influence of mapping the units of execution to different cores. We also analyze the relation between performance and working set size, and the effect of the irregular accesses. Furthermore, since the SCC processor allows to change the cores, mesh network and memory clock frequency, an evaluation in terms of performance and power efficiency of different hardware configurations is provided.

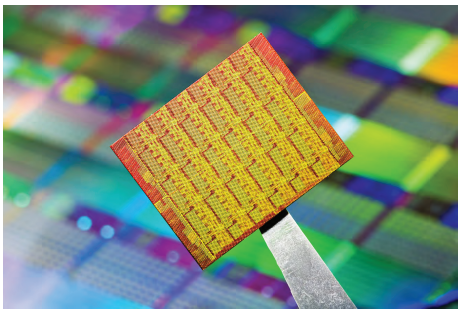
In addition, we evaluate several optimization techniques for the SpMV kernel on the SCC. First, some of the most successful reordering techniques are considered. These techniques evaluate the sparsity pattern of the matrix to find an appropriate permutation of rows and columns of the original matrix that improves the SpMV performance. Next, the effects of applying blocking, as code restructuring technique, is studied. Finally, two data compression techniques are also implemented and evaluated. The aim of these techniques is to reduce the working set and improve the SpMV performance by reducing the pressure on the memory hierarchy.

The paper finishes with an architectural comparison of the SCC processor with several leading multicore processors and GPUs in terms of performance and power efficiency.

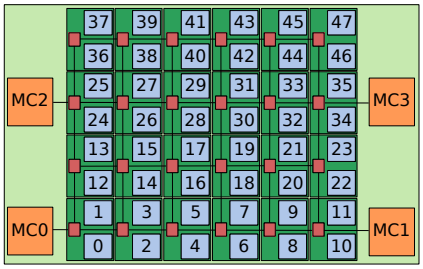
This work is structured as follows: Section 2 introduces a description of the SCC processor. Section 3 presents the main characteristics of the SpMV kernel together with the sparse matrices testbed. The results of the experimental evaluation on the SCC processor are analyzed and discussed in Section 4. Section 5 introduces several optimization techniques for the SpMV, which have been evaluated on the SCC processor. In Section 6 an architectural comparison is performed. Section 7 discusses relevant previous work. The paper finishes with the main conclusions extracted from this work.

2. The Single-Chip Cloud Computer (SCC)

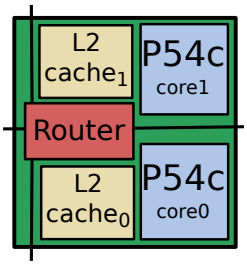
The Single-Chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs as a platform for many-core software research. The SCC consists of 48 independent x86 cores arranged in 24 tiles. Figure 1 shows a SCC wafer together with an overview of the processor and a scheme of a single tile.



(a)



(b)



(c)

Figure 1: Single-Chip Cloud Computer (SCC): wafer (a), overview of the processor (b) and a tile scheme (c).

A tile consists of two P54c Pentium cores with an increased L1 instruction and data cache sizes of 16 KB each [1]. It also holds separate 256 KB L2 caches for each core. Caches are 4-way set associative with a pseudo-LRU replacement policy. L2 cache is write-back only. The SCC offers no coherency among cores caches to the programmer. This coherency must be implemented through software methods, by flushing caches for instance. Furthermore each tile also contains a 16KB (8 KB per core) Message Passing Buffer (MPB). It is intended to support a message-passing programming model on the chip. The MPB is implemented as shared memory that can be accessed efficiently by all 48 cores on the SCC. Furthermore, each tile has its own frequency domain in such a way that the clock frequency can be set separately for every of the 24 tiles. Clock frequencies from 100 to 800 MHz can be used.

The mesh network is a 2D grid that connects all tiles [2]. Each tile contains a router with four ports, which connects its tile with other units on the SCC in four different directions. The routing scheme is a simple (x,y) scheme, that is, data is routed first horizontally and then vertically through the network. The network has its own clock and power source, and can be initialized to operate on either 800 MHz or 1.6 GHz. However dynamic changes during run-time are currently not supported by the chip.

The overall system admits a maximum of 64GB of main memory accessible

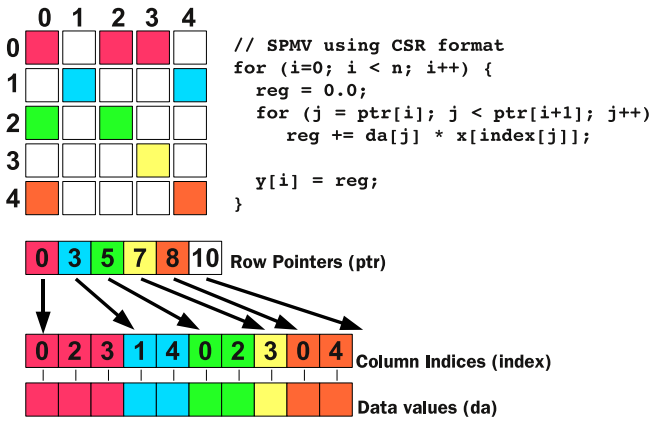


Figure 2: Compressed-Sparse-Row (CSR) format example, and a basic CSR-based sparse matrix-vector product (SpMV) implementation.

through 4 DDR3 memory controllers evenly distributed around the mesh. Our setup uses 32 GB. Each core is attributed a private domain in this main memory whose size depends on the total memory available (640 MB in the system used here). The memory controllers are attached to the routers of four tiles at the edges of the chip at the coordinates (0,0), (2,0), (0,5) and (2,5) (see MC boxes in Figure 1(b)). Six tiles (12 cores) share one of the four memory controllers to access their private memory. In addition, a part of the main memory is shared between all cores. The memory controllers operate on their own clock and power source, and can be clocked at 800 or 1066 MHz. But, like the mesh, the clock frequency of the memory controllers must be chosen during the initialization of the chip.

Intel provides RCCE (pronounced "rocky") [3], a simple message passing library which supports basic message passing functions that are specifically designed to use the special architecture characteristics of the SCC (e.g. the message passing buffer). RCCE has two basic communication primitives: point-to-point communications and collective operations. It also provides access to other entities like the voltage controller, or it allows the allocation of shared memory. When using RCCE, the cores are not directly accessed by their identification (as shown in Figure 1(b)). Instead, for each execution of a RCCE application, the cores to be used and their order can be configured differently [4]. Each one of the participant cores is mapped to an unit of execution (UE). Within the applications, these UEs are addressed by their ranks, in a similar way to addressing MPI processes.

3. Sparse Matrix-Vector Multiplication (SpMV)

In this work the sparse matrix-vector product (SpMV) operation is considered. This kernel is notorious for sustaining low fractions of peak processor performance due to its indirect and irregular memory access patterns. Let us

	Matrix	# rows (n)	# nonzeros (nnz)	nnz/n	ws (MBytes)
1	av41092	41092	1683902	70.9	20.1
2	bcsstm36	23052	320606	13.9	4.1
3	crystk03	24696	1751178	70.9	20.5
4	e40r0100	17281	552562	32.0	6.7
5	exdata_1	6001	2269501	378.2	26.1
6	F2	71505	5294285	74.0	62.0
7	fp	7548	848553	112.4	9.9
8	garon2	13535	390607	28.9	4.7
9	gupta2	62064	4248286	68.5	49.8
10	gyro_k	17361	1021159	58.8	12.0
11	lhr10	10672	232633	21.8	2.9
12	li	22695	1350309	59.5	15.9
13	mixtank_new	29957	1995041	66.6	23.4
14	msc10848	10848	1229778	113.4	14.3
15	Na5	5832	305630	52.4	3.6
16	nc5	19652	1499816	76.3	17.5
17	ncvxbqp1	50000	349968	7.0	5.0
18	nd3k	9000	3279690	364.4	37.7
19	net25	9520	401200	42.1	4.8
20	nmos3	18588	386594	20.8	4.8
21	pct20stif	52329	2698463	51.6	31.9
22	psmigr_1	3140	543162	173.0	6.3
23	rajat15	37261	443573	11.9	5.8
24	ship001	34920	4644230	133.0	53.8
25	sme3Da	12504	874887	70.0	10.3
26	sme3Dc	42930	3148656	73.3	36.9
27	sparsine	50000	1548988	31.0	18.7
28	syn12000a	12000	1436806	119.7	16.7
29	tandem_vtx	18454	253350	13.7	3.3
30	thread	29736	4470048	150.3	51.7
31	TSOPF_FS_b300_c2	56814	8767466	154.3	101.4
32	tsyl201	20685	2454957	118.7	28.5

Table 1: Matrix benchmark suite.

consider the operation `y=A×x`, where `x` and `y` are dense vectors, and `A` is a $n \times m$ sparse matrix. The most common data structure used to store a sparse matrix for SpMV computations is the Compressed-Sparse-Row format (CSR). According to this format the nonzero elements (nnz) of a sparse matrix with n rows are stored contiguously in memory in row-major order. The `index` array stores the column indices of each element in the original matrix, and the `ptr` array of size $n+1$ stores a pointer to the beginning of each row. da stores the nnz data values of the sparse matrix.

Figure 2 shows, in addition to an example of the CSR format for a 5×5 sparse matrix, a basic implementation of the SpMV for this format. This implementation enumerates the stored elements of `A` by streaming both `index` and `da` with unit-stride, and loads and stores each element of `y` only once. However, `x` is accessed indirectly, and unless we can inspect `index` at run-time, it is difficult or impossible to reuse the elements of `x` explicitly. Note that the locality of the accesses to `x` depends on the sparsity pattern of the considered matrix.

As matrix test set 32 square sparse matrices from different real problems

that represent a variety of nonzero patterns were selected. These matrices are from the University of Florida Sparse Matrix Collection (UFL) [5]. Table 1 summarizes the features of the matrices: number of rows/columns (n), nonzeros (nnz), nonzeros per row/column (nnz/n) and size of the working set (ws). Considering 32-bit integer indexing and double precision arithmetic, the working set in bytes can be calculated as: $4 \cdot (n + 1 + nnz) + 8 \cdot (nnz + 2 \cdot n)$.

4. Experimental Evaluation

The SCC platform is based on Ubuntu Linux with a 2.6.32-24 kernel. Codes in this work were written in C and compiled with the Intel's 8.1 Linux C compiler (`icc`). RCCE library was used to parallelize the sparse matrix-vector multiplication code of Figure 2. In particular, the results of this section were obtained by using RCCE 1.0.13. The partitioning scheme splits the matrix row-wise in such a way that the same amount of nonzeros would be assigned to each unit of execution.

For all the experiments (with the exception of those in Section 4.4), cores, routers and memory are clocked at the default speeds of 533 MHz, 800 MHz and 800 MHz respectively. Note that since the SCC cores do not provide a constant clock that is independent of their operational frequency, `RCCE_wtime()` was used as time measurement function, which returns the RCCE wall time and ensures correct time measurements even for different clock frequencies. In order to measure the performance, we have used the floating operations per second (FLOPS/s), which was calculated by dividing the total number of floating point operations ($2 \times nnz$) by the execution time.

Among the factors that affect the SpMV performance, two are of the utmost importance: the memory bandwidth limitation and the low locality caused by the irregular and indirect memory accesses. Therefore, it is important to evaluate the influence of the SCC memory hierarchy on the SpMV performance. Different topics related to this will be analyzed and discussed next in Sections 4.1, 4.2 and 4.3. On the other hand, the SCC processor allows to change the cores, mesh and memory clock frequency (see Section 2). The impact on the SpMV performance of different hardware configurations will be studied in Section 4.4. A power consumption and efficiency analysis is also provided.

4.1. Mapping Units of Execution to Cores

Next we will evaluate the influence of mapping the units of execution to different cores of the SCC processor. Several topics related to the SCC architecture must be taken into consideration. First, as indicated in Section 2, six tiles (12 cores) share one of the four memory controllers to access their private memory. In particular, the cores are, by default, partitioned into quadrants. For example, the lower left quadrant contains cores 0-5 and 12-17 (see Figure 1(b)), and the main memory are accessed through the memory controller `MC0`. And secondly, the memory latency (and bandwidth) of a core depends on the distance to the memory controller [2, 6]. The formula to determine the time

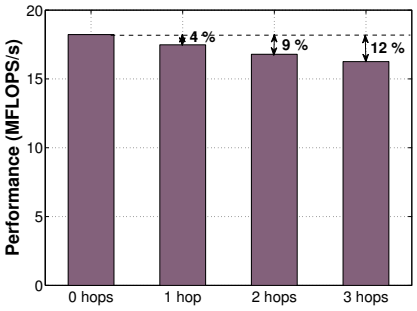


Figure 3: Comparison of the performance obtained when mapping one unit of execution to cores with different distance to their memory module.

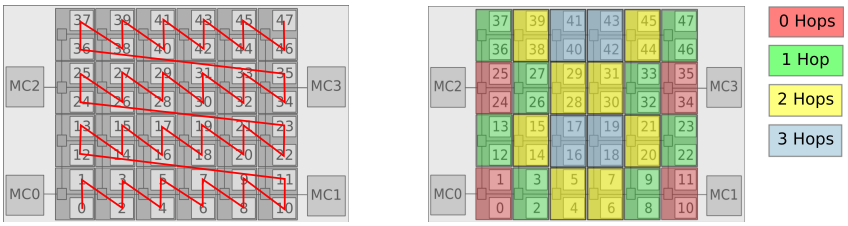
required by a core since sending a memory request and receiving the respective data is:

$$40C_{core} + 4 \cdot h \cdot 2C_{mesh} + 46C_{mem} \quad (1)$$

where C_{core}, C_{mesh} and C_{mem} are the clock cycles of the core, the mesh network and the memory respectively, and h denotes the number of mesh network *hops* required to reach the memory controller. According to this, the only property that need to be taken into account to measure the memory latency is the number of mesh network hops. For this reason, we will consider the memory performance of cores with 0, 1, 2 and 3 hops to the memory controller, which covers all the possible distances in the default configuration.

Next, we will evaluate the SpMV performance when using a single core accessing the memory. Cores with a different distance (number of hops) to their memory module were analyzed. Results in Figure 3 show the average performance obtained for all the matrices of our testbed. As expected, a better performance is observed when the distance to the memory is the lowest, that is, when the number of hops is 0. According to the Equation 1, this case corresponds with the smallest memory latency. We must highlight that the differences in the performance are noticeable, especially when a 3 hops core is considered. In this case, a degradation of about 12% was observed. This impact can be even more important for parallel executions if the units of execution are not carefully mapped to the cores.

A study to determine the impact on the parallel SpMV performance of different mapping configurations has been carried out. We have compared two mapping configurations: *standard* and *distance reduction*. The standard mapping is the default mapping of units of execution to cores (see Figure 4(a)). In this case, each RCCE rank is equal to the core identification. For instance, the cores involved in a parallel execution with 4 units of execution would be 0, 1, 2 and 3. This configuration does not take into consideration the distance to the memory. We propose the distance reduction configuration (Figure 4(b)), which maps the units of execution to the available cores with the lowest number of



(a) (b)

Figure 4: Different mappings of the units of execution to cores: (a) standard and (b) considering the distance (hops) to the memory module from the core.

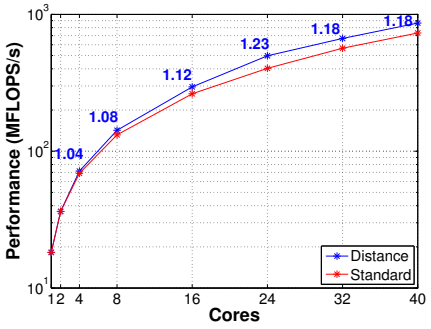


Figure 5: Performance when considering the two mapping configurations for different number of cores.

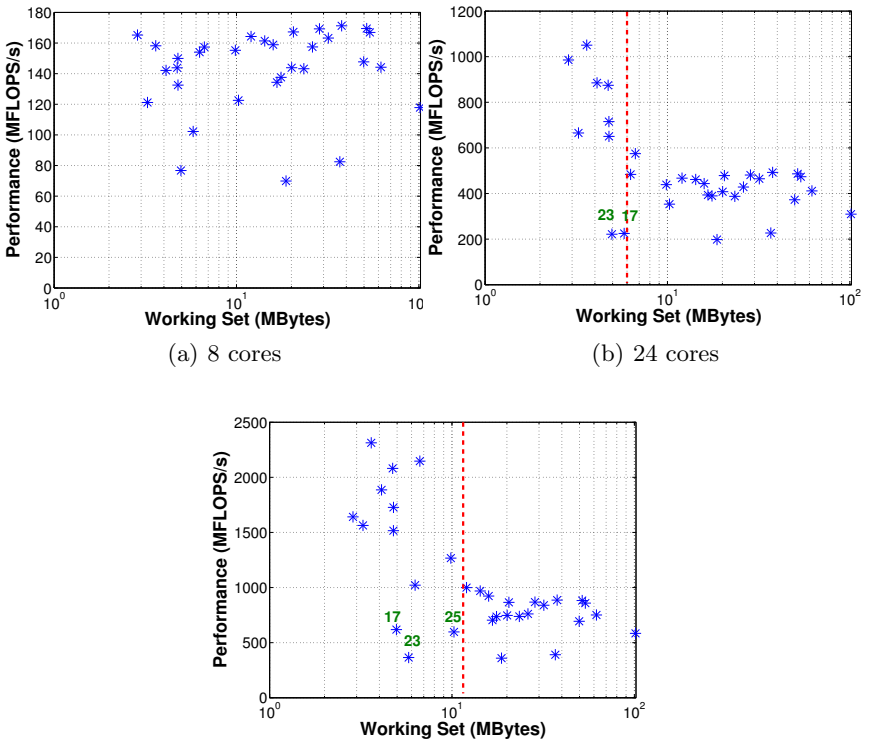
hops to the memory. Considering the previous example, the cores 0, 1, 10 and 11 would perform the SpMV operation when using this configuration.

A comparison of the two mapping configurations is displayed in Figure 5. The graphic shows the average SpMV performance of both mappings for different number of cores. In addition, the speedup values obtained by the distance reduction mapping with respect to the standard configuration are shown. As expected, the distance reduction configuration clearly outperforms the standard one, reaching speedups up to 1.23×. Note that there is no difference in the selected cores by both mappings when using 1 and 2 cores, and therefore, the observed performance is the same.

We conclude that the mapping of units of execution to cores on the SCC has a big impact on the SpMV performance, especially with a high number of cores. The best solution is to select those cores placed near their memory module. For this reason, we have selected the distance reduction mapping configuration in the experiments of the remainder of the paper.

4.2. Influence of the Working Set Size

A study about the influence of the working set of the matrices in the SpMV performance is presented in this section. Figure 6 shows the performance for



(a) 8 cores

(b) 24 cores

(c) 48 cores

Figure 6: Performance of the matrices in relation to their working set using different number of cores.

all the sparse matrices from the testbed relative to their working set size (see Section 3 for details about how it is defined). Results with 8, 24 and 48 cores are displayed. Those matrices with a working set per core that fits the L2 cache are on the left side of the red dotted line in the graphs.

It is clear from the figure that the performance across the matrix set presents a great diversity. However, we can group the matrices into two categories: matrices whose working set per core fits into the L2 cache of the cores, and those whose working set per core is larger than the L2 cache. In the first case, those matrices will only experience compulsory misses, while in the second case, matrices may also produce capacity ones. When using 8 cores (Figure 6(a)), there are no matrices with a working set per core smaller than the L2 cache. This fact explains the reason why there is not a relation between the working set size and the performance. Nevertheless, when using a higher number of cores, there are many matrices whose working set per core fits into the L2 cache (256 KBytes). Figures 6(b) and 6(c) show the results for executions with 24 and 48 cores respectively. An important boost in the performance of the smallest

Kernel	# Cores			
	1	8	24	48
CSR SpMV	18.2	142.3	496.8	1040.6
CSR SpMV with no `x` misses	20.9 [1.15]	166.6 [1.17]	584.5 [1.18]	1238.4 [1.19]

Table 2: Impact on the performance (in MFLOPS/s) of the irregular accesses on vector `x`. Speedups are shown between square brackets.

matrices with respect to the larger ones is observed. For example, considering 24 cores, small matrices reach performance values up to 1 GFLOPS/s, while the bigger ones are always in the range of 450 MFLOPS/s. This behavior is not observed for the small matrices 17 and 23. The cause is the presence of very short row lengths, that is, small nnz/n values. This fact leads to a small trip count in the inner loop of the SpMV kernel, a fact that may degrade the performance due to the increased overhead of the loops.

We have performed another experiment to confirm the observations detailed above. The SCC processor allows to boot the cores disabling the L2 caches. A comparison of the SpMV performance when the L2 caches of the cores are disabled with respect to the default case was carried out. An important degradation in the performance was observed, especially as the number of cores grows (for example, 30% when using 48 cores). If the results with L2 caches disabled were shown by matrix, no relation between the performance and the working set size even for a high number of cores would be found. That is, the behavior would be similar to the one observed in Figure 6(a). These observations confirm that the differences in the SpMV performance between the matrices of the testbed are, in most of the cases, caused by the capacity cache misses.

4.3. Influence of the Irregular Accesses

One of the most important factors that affects the performance of the SpMV is the low locality caused by the irregular and indirect memory accesses. In a similar way to [7], we have performed a test to evaluate the impact of irregular accesses on vector `x` (see the SpMV code in Figure 2). With this purpose a modification of the SpMV code to eliminate cache misses on vector `x` has been carried out. In particular, in this version of the code each reference to `x` will now access only its first element `x[0]`, which results in a perfect access pattern on `x`. In this way, we can assume that any performance variation observed between the original and the modified version of the code is caused by the effect of the irregular accesses on the vector `x`. Table 2 shows the variations in the average performance of both versions of the SpMV code using different number of cores. Speedups are also provided.

Note that irregular accesses have a big impact on the SpMV performance. Speedups of over 10% has been found in more than 50% of the matrices using different number of cores. This behavior is not observed on other multicore systems [7], where the authors conclude that the irregular accesses is not a prevailing performance problem. We find the cause in the cache hierarchy of the SCC cores with small L2 caches of 256 KB. Modern multicore processors

Configuration (cores,mesh,memory)	# Cores			
	1	8	24	48
0 (533,800,800)	18.2	142.3	496.8	1040.6
1 (800,800,800)	21.7 [1.19]	168.9 [1.19]	596.9 [1.20]	1284.6 [1.23]
2 (800,1600,800)	21.9 [1.20]	171.1 [1.20]	608.7 [1.22]	1333.5 [1.27]
3 (800,800,1066)	24.7 [1.35]	191.3 [1.35]	662.2 [1.33]	1394.3 [1.34]
4 (800,1600,1066)	25.3 [1.39]	196.3 [1.38]	690.5 [1.39]	1513.3 [1.45]

Table 3: Comparison of the performance (in MFLOPS/s) using different hardware configurations of the SCC. Cores, mesh and memory clock frequencies (in MHz) for each configuration are shown between brackets. Speedups with respect to the default configuration are shown between square brackets.

have typically three levels of cache, with L3 of bigger size (see Section 6). It is worth to mention that the most important speedups using the modified SpMV version were obtained for those matrices that perform poorly with the original SpMV code. For example, this is the case of matrices 17, 23, 25, 26 and 27, whose speedups are higher than 1.8× (see Figure 6 for the results about the performance of the original SpMV code). Therefore, the influence of the locality in the SpMV performance on the SCC processor is high, especially for those matrices with an irregular nonzero pattern.

4.4. Evaluation of Different SCC Configurations

The SCC processor allows to change the cores, mesh network and memory clock frequency (see Section 2). In particular, each tile (2 cores) has its own frequency domain in such a way that the clock frequency can be set separately for every of the 24 tiles from 100 to 800 MHz. The network can be initialized to operate on either 800 MHz or 1.6 GHz. And the memory controllers can be clocked at 800 or 1066 MHz. The default frequency for the cores, mesh and memory are 533, 800 and 800 MHz respectively. Until now we have only considered this configuration in our experiments.

Now, we will evaluate the impact on the SpMV performance of different hardware configurations of the SCC processor. We have considered five configurations, which are detailed in Table 3. This table shows the average performance attained using a different number of cores for each configuration. Speedups with respect to the default configuration are also provided.

According to the results several conclusions can be made. First, the SpMV performance improvement caused by an increase in the mesh frequency becomes significant only as the number of cores increases. We can observe this behavior comparing the configurations 1 and 2 (DDR at 800 MHz) and also 3 and 4 (DDR at 1066 MHz), with differences up to 1.09× using 48 cores. In contrast, the impact on the performance of the memory controllers frequency is noticeable even in the sequential case. For example, the speedup between configurations 3 and 1 is always higher than 1.09×. This behavior confirms the memory bound nature of the SpMV kernel. To analyze the influence of the cores frequency, we focus on configurations 0 and 1. As expected, the increase in the cores frequency has the most important impact on the performance, reaching speedups up to

Configuration	Power Consumption (W)	Efficiency
0	83.3	12.5
1	101.6	12.6
2	105.7	12.6
3	103.2	13.5
4	107.4	14.1

Table 4: Comparison of the power consumption and efficiency of the SCC using different hardware configurations. Results considering the full system.

1.23×. Finally, it is worth to mention that the best results overall are achieved by the configuration 4, which corresponds to the highest available frequency values of the SCC processor. In this case, speedups ranging from 1.39× to 1.45× using 1 and 48 cores respectively were achieved.

However, we have to take into consideration another important issue as consequence of using higher clock frequencies: the power consumption. We have measured the average power consumption of the SCC processor for all the considered configurations when the SpMV kernel is running on. These values are displayed in Table 4 considering the full system, that is, 48 cores. An important increase has been observed for all the configurations with respect to the default one. This is mainly caused by the higher frequency of the cores. For instance, the SCC processor increases its power consumption from 83.3 to 101.6 W using all the cores and configuration 1, which means an increase of about 22%. Mesh and memory frequencies only increase the consumption about 4% and 1.5% respectively.

Concerning the power consumption we can study the power efficiency, which is one of today's most important considerations in HPC acquisition. Table 4 shows the efficiency calculated as the MFLOPS/s-to-Watt ratio based on the average SpMV performance and the measured power consumption for the five configurations. Note that the results illustrate the efficiency considering the full system. The best result is obtained by configuration 4, demonstrating that its power consumption is compensated by a noticeable improvement in the performance. Nevertheless, the efficiency of configurations 0, 1 and 2 is practically the same. This fact points out that the performance improvement due to the higher frequencies of cores and mesh is not enough to overcome the increase in the power consumption in comparison with the default configuration. According to the results, what makes the difference in terms of efficiency is clocking the memory controllers at 1066 MHz, which is the case of configurations 3 and 4.

5. SpMV Optimizations

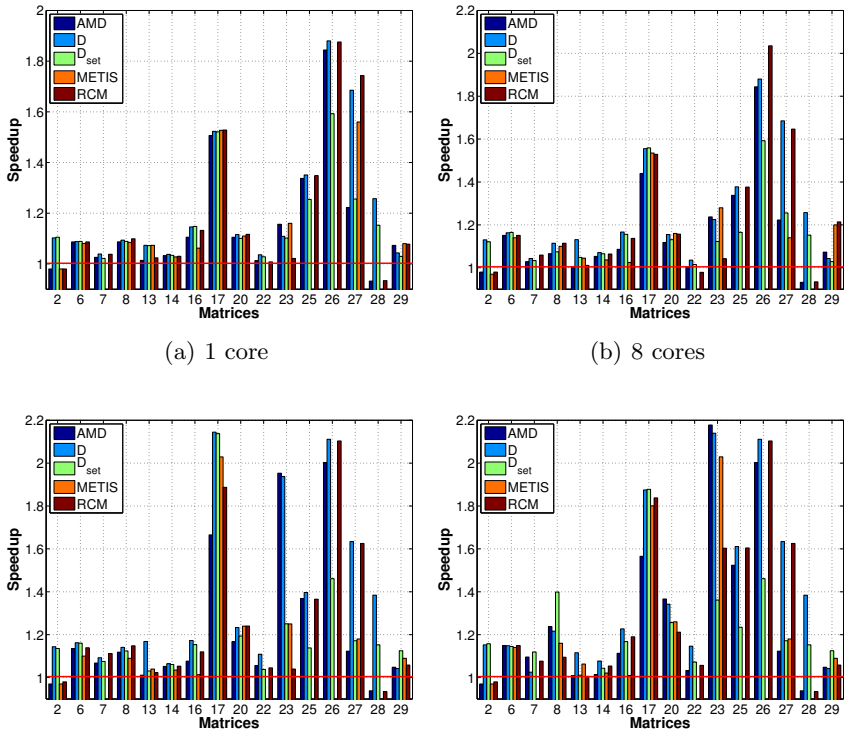
Until now we have evaluated and studied the behavior of the SpMV kernel on the SCC processor. No optimizations have been applied in order to increase its performance. As we have mentioned previously, this kernel is notorious for sustaining low fractions of peak performance on modern processors. Some of the

factors affecting the SpMV performance are the memory bandwidth limitation and the low data reuse caused by the irregular and indirect memory access patterns. Note that the sparse matrix pattern that characterizes the accesses is only known at run-time. To this end, different strategies focusing on some of the SpMV performance problems have been proposed. Next we will evaluate some of the most successful optimization techniques. In particular, reordering, blocking and data compression techniques are considered. Note that the default configuration was used in the results shown in this section.

5.1. Reordering

Reordering techniques have been a successful approach to improve the performance of the sparse matrix-vector multiplication. These techniques evaluate the sparsity pattern of the matrix to find an appropriate permutation of rows and columns of the original matrix. In some cases the main goal of these techniques was different than the optimization of the sparse algebra operation. For example, the main goal of the Cuthill–McKee ordering is to modify the sparsity pattern of the considered matrix with the aim of reducing its bandwidth [8]. Later works demonstrate that applying this reordering technique also increases the performance of the SpMV [9, 10]. This kind of techniques has obtained good results on different architectures, from multicores to GPUs [11]. In this work some of the most successful reordering techniques for improving the performance of the SpMV have been considered. A brief description of them is provided next:

- Approximate Minimum Degree (AMD) [12]: The objective of this algorithm is to find a permutation of the original matrix that reduces the fill-in when a Cholesky factorization is performed.
- Distance function [13]: The goal of this technique is to increase the grouping of nonzero elements in the sparse matrix pattern that characterizes the irregular accesses and, as a consequence, increasing the locality in the execution of the SpMV code. This algorithm allows to permute individual rows/columns of the original matrix or sets of consecutive rows/columns. Depending on the choice we denote the technique as D or D_{set} respectively.
- Reverse Cuthill–McKee (RCM) [8]: The aim of this technique is to reduce the bandwidth of the original matrix. Reverse Cuthill–McKee algorithm is the same algorithm as the original one but with the resulting index numbers reversed.
- METIS library [14]: It is considered a standard in terms of graph partitioning and reordering. The reordering technique included in the library computes fill-reducing orderings using a particular implementation of the nested dissection algorithm. In particular, we have used the `onmetis` program. This technique can only be applied to matrices with symmetric pattern.

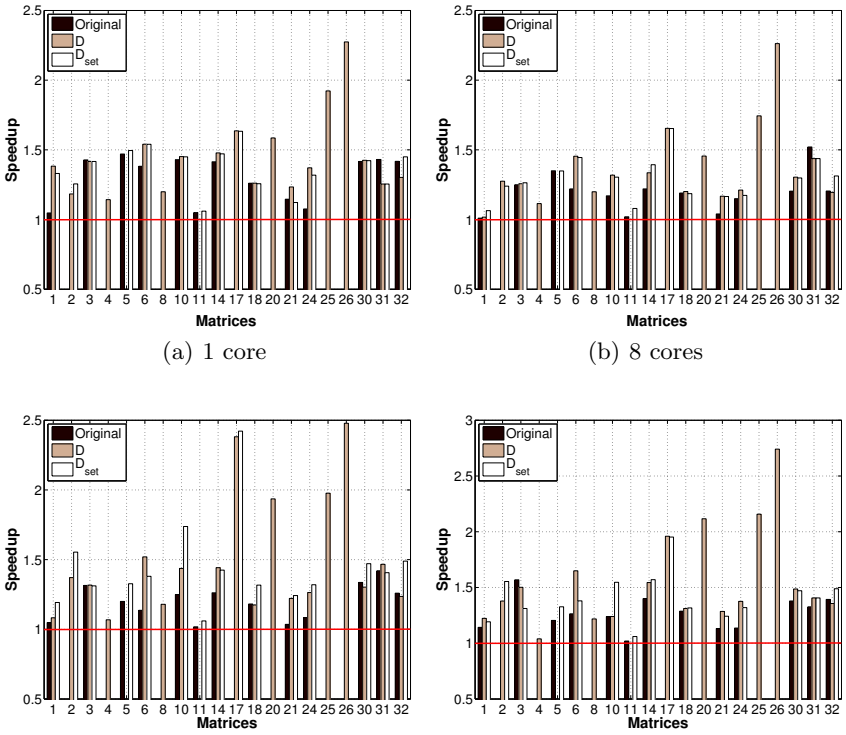


(a) 1 core (b) 8 cores

(c) 24 cores (d) 48 cores

Figure 7: Speedups obtained using different reordering techniques with respect to the original (non-reordered) matrices.

Performance results considering different number of cores are shown in Figure 7. We have compared the performance obtained by the original matrices (without reordering) with respect to the reordered ones. Results for a particular matrix are displayed in the figures only if the original performance differs from the obtained by any of its reorderings in more than 2%. The fact that 50% of the matrix test set was included in the figures demonstrates that the reordering techniques have a great influence on the SpMV performance for the SCC processor. Best results overall are obtained by the D reorderings, showing average speedups of 1.11×, 1.13×, 1.19× and 1.20× when using 1, 8, 24 and 48 cores respectively. These values are calculated considering all the matrices from the testbed. The order in terms of performance for all the considered techniques is D, RCM, D_{set}, AMD and METIS. We have detected that the behavior of the reordering techniques is slightly better with a higher number of cores. It is worth to mention that speedups up to 2.2× were reached. Reordering techniques were especially effective for matrices 17, 23, 25, 26 and 27, which correspond to the matrices with the most irregular nonzero pattern. Reorderings boost the



(a) 1 core (b) 8 cores

(c) 24 cores (d) 48 cores

Figure 8: Speedups obtained applying a blocking technique to the original matrices (black bars). Results of applying the combination of reordering and blocking techniques are also shown.

performance of these matrices due to the increase in the data locality.

5.2. Blocking

Blocking is one of the most well-known code restructuring techniques and it can be applied to different levels of memory hierarchy such as physical memory, caches and registers. Blocking reshapes an iteration space over a data domain by partitioning it into pieces that fit into the selected level of hierarchy. In this way, all the computations on each piece can be completed before moving to the next one. Blocking rearranges the order of the computations, so that multiple references to a data element occur in inner loops while such element is still resident in the corresponding level of hierarchy.

In this work we have used the SPARSITY register blocking technique [15], using the RCCE library to port the code to the SCC platform. Register blocking reorganizes the sparse matrix into a set of fixed-size dense blocks by finding a small block-size that fits into the target machines's register set. A reasonably small number of extra zeros elements are stored to compose dense blocks. These

extra zero values increase the number of floating-point operations because they are involved as a part of the blocks in the sparse matrix computation. So, the matrix is stored as a sequence of small dense blocks, and the computations are reorganized to compute each block before moving on to the next. One of the main drawbacks of this technique is the strong dependence with the sparsity pattern of the matrix. Register blocking only achieves good performance for matrices with small dense-blocks in the pattern such as Finite Element Method (FEM) matrices.

Figure 8 shows the speedups obtained after applying the register blocking technique to the original matrices (black bars). Results for a particular matrix are displayed only if there is a block-size that improves the performance with respect to the original SpMV kernel. Speedup values always correspond to the best block-size found. First, note that blocking can be successfully applied to 13 matrices, that is, about 40% of the testbed. The average speedups including all the matrices are 1.12×, 1.08×, 1.08× and 1.11× with 1, 8, 24 and 48 cores respectively. Speedups up to 1.5× have been reached for those matrices with higher nnz/n values. So, it is very likely that these matrices have dense blocks of nonzeros in their pattern.

On the other hand, we have introduced D and D_{set} reorderings in the previous section. Recall that this technique reorganizes the data guided by a locality model by increasing the grouping of nonzero elements in the sparse matrix pattern. And, in this way, the creation of blocks in the pattern of the reordered matrix is favored [13]. Therefore, we expect that register blocking improves its performance when combining with this reordering technique.

Results of the combination of both techniques are also shown in Figure 8 labeled as D and D_{set}. Several conclusions can be made. First, reorderings make possible the application of register blocking to more matrices. In particular, 18 and 15 matrices when considering D and D_{set} respectively. In addition, in most of the cases, the combination of techniques outperforms applying just blocking. For instance, using the D reorderings and blocking obtained an average speedup of 1.25×, 1.21×, 1.28× and 1.31× with 1, 8, 24 and 48 cores. And finally, some noticeable speedups were measured with values up to 2.7× (matrix 26 using 48 cores).

5.3. Compression

The memory subsystem is identified as the main performance bottleneck of the SpMV kernel, and in particular, the memory bandwidth. Data compression techniques try to reduce the working set and improve the SpMV performance by reducing the pressure on the memory hierarchy. We have evaluated on the SCC processor two of these compression techniques. The first one was introduced in [16] and implements a simple index reduction approach. It tries to use 16 bits indices for the columns when it is possible, which means that it can only be applied to matrices with a number of columns fewer than 64K. In our case, we cannot use the index reduction with the matrix 6 (see Table 1). Performance results using this technique are displayed in Figure 9(a). We show the average

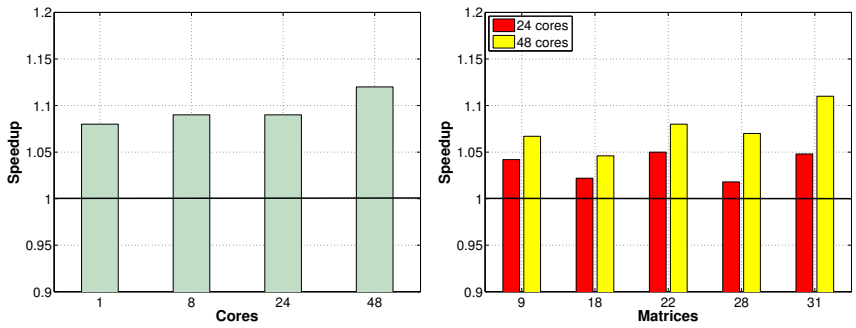


Figure 9: Performance results using different data compression techniques: (a) index compression and (b) CSR-DU storage format.

speedup using different number of cores and considering all the matrices. Measurements point out that the reduction in the memory traffic for the matrices has an impact on the SpMV performance, with speedups ranging from 1.08× to 1.12× with 1 and 48 cores respectively. Despite of being a simple approach, all the considered matrices (except matrix 6) increase their performance with no dependence on the characteristics of the sparsity pattern.

The second compression technique ported and evaluated on the SCC processor was first introduced in [17]. This method compresses the indexing structure of the sparse matrix taking into consideration the distribution of the nonzeros in the columns of the matrix and exploiting the distances between elements instead of using absolute indexes. The technique is based on the idea that there exists parts within the matrices exhibiting some level of density without necessarily containing contiguous nonzeros. These parts can contribute to reduce the storage volume of the matrix indexing data. In particular, delta encoding is used, where each delta value is defined as the difference of the current column index with the previous one. These values are positive and less or equal than its corresponding column index. Therefore, parts of the matrices that exhibit density allow for their delta values to be stored using integers smaller than 4 bytes, leading to a reduction of the index data size. We denote this technique as CSR-DU storage format. Note that in this format the `ptr` and `index` arrays of CSR are replaced by a single byte array containing all the necessary indexing information.

First, we have analyzed the matrix size reduction when using CSR-DU. This format on average decreases the size of the working set of the matrices about 21%. However, as shown next, there is not a correlation between the size reduction and the speedup obtained. This is caused by other factors affecting performance such as the branch mispredictions in the CSR-DU kernel.

Figure 9(b) shows the speedup results when using the CSR-DU format with respect to the CSR case. Only in five cases CSR-DU outperforms CSR, and

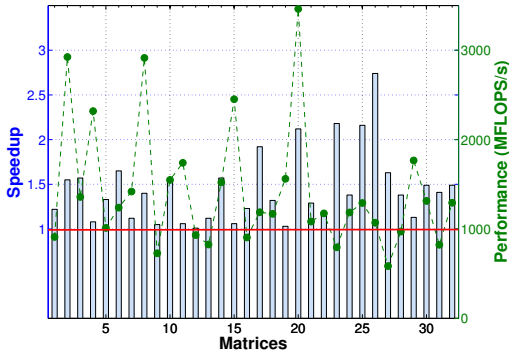


Figure 10: Summary of the optimization results for all the matrices of the testbed considering the full SCC system in terms of the speedup (blue bars) and performance (green dots).

using 24 or more cores. These matrices present a high number of nonzeros per row (nnz/n), which confirms the behavior observed in [17] where CSR-DU format degrades the performance of small row matrices due to the loop overhead they induce. In addition, CSR-DU will perform better on multicore architectures where a number of processing elements shares the cache, which is not the case of the SCC processor. In any case, speedups up to 1.11× have been reached.

5.4. Summary of the optimization results

Next we will summarize the SpMV performance results when the optimization techniques detailed above are applied. We have demonstrated that the SpMV on the SCC is very sensitive to reordering, blocking and compression techniques. This is mainly caused by the small L2 caches of the cores, which produce a high number of capacity and conflict misses. Thefeore, increasing the data locality of the SpMV on the SCC has a bigger impact on performance than in other multicore CPUs.

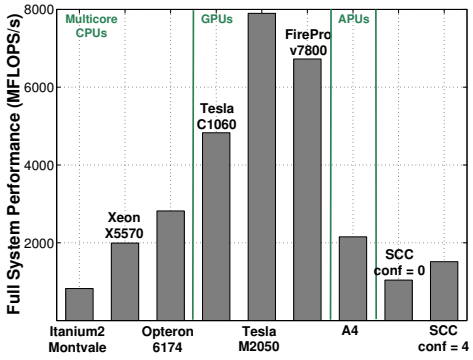
Figure 10 shows the results of the optimization techniques in terms of the speedup (blue bars) and performance (green dots) for all the considered matrices using the full SCC system. Only the best cases are displayed. For example, matrix 26 obtains its best performance after applying the combination of reordering and register blocking. Improvements have been found for all the matrices. The average performance with optimizations is 1,426 MFLOPS/s. In other words, an average speedup of 1.45× with respect to the standard case. Best results in terms of performance were obtained by the smallest matrices, as we point out in Section 4.2, boosted by the optimization techniques. This is the case of matrices 2, 8 and 20.

6. Architectural Comparison

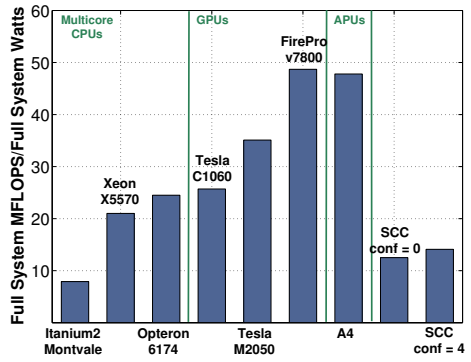
Finally, we will compare the behavior of the SpMV in terms of performance and power efficiency on different systems including the SCC many-core processor. The systems evaluated are the following:

- Multicore CPUs:
  - Intel Itanium2 Montvale: This processor comprises two cores running at 1.6 GHz, and three cache levels (including an unified 9 MB L3 cache per core). The peak performance per core is 6.4 GFLOPS/s. Power consumption: 104 W (TDP).
  - Intel Xeon X5570: It is a quad-core processor, which includes three cache levels with a shared 8 MB L3 cache. Each core operates at 2.93 GHz, with 11.72 GFLOPS/s as peak performance per core. Power consumption: 95 W (TDP).
  - AMD Opteron 6174: This processor consists of 12 cores running at 2.2 GHz. It has three cache levels with a shared 12 MB L3 cache. Power consumption: 80 W (ADP), 115 W (TDP) [18].
- Graphic Processing Units (GPUs):
  - NVIDIA Tesla C1060: It has 240 cores with a double precision arithmetic peak performance of 78 GFLOPS/s. Power consumption: 187.8 W (TDP).
  - NVIDIA Tesla M2050: This GPU was designed using the new Fermi architecture from NVIDIA. It has 448 cores, with a double precision peak performance of 515.2 GFLOPS/s. Power consumption: 225 W (TDP).
  - ATI FirePro V7800: It consists of 1440 stream processors cores with a double precision arithmetic peak performance of 403 GFLOPS/s. Power consumption: 138 W (TDP).
- Accelerated Processing Units (APUs):
  - AMD A4-3310MX: This APU belongs to the Fusion family, and integrates a dual-core CPU and a GPU on the same die with low power consumption. The GPU, which has been used in the tests, consists of 240 cores. The single precision peak performance is 231.1 GFLOPS/s. Note that double precision arithmetic is not supported. Power consumption: 45 W (TDP).

Some issues related to these systems must be clarified. The parallelization of the SpMV code of Figure 2 was performed using OpenMP on the Intel and AMD multicore processors. In addition, the Intel's 10.0 Linux C compiler was used with the Xeon and Itanium2 systems, while for the Opteron the compiler was Open64. For the NVIDIA GPUs, we have used the SpMV kernel from the



(a)



(b)

Figure 11: Comparison of the performance (a) and power efficiency (b) of different architectures with respect to the SCC processor.

NVIDIA library detailed in [19] and CUDA 3.2. Finally, an implementation in OpenCL 1.1 of the CSR SpMV kernel of the NVIDIA library was evaluated on the ATI GPU and APU.

Figure 11(a) shows the average SpMV performance obtained on the above systems together with the results of the SCC processor using the default configuration and the configuration 4 (see Table 3 for details) with no optimizations. These results were obtained using all the available cores on the different systems. Results indicate that SCC only outperforms the dual-core Itanium2 processor. Best results are achieved by the GPUs, especially the one with the Fermi architecture and the ATI FirePro. We must highlight that the Tesla M2050 obtains a noticeable average performance of 7.9 GFLOPS/s, which means a speedup of 7.6× with respect to the SCC default configuration. This good behavior is mainly caused by the important effort made to improve the double precision arithmetic capabilities of the last generation of NVIDIA GPUs. This GPU is followed by the ATI FirePro, reaching also a noticeable performance of 6.7

GFLOPS/s. However, these values only mean about 1.5% of the peak performance for both GPUs. Multicore CPUs are far from the performance obtained by the GPUs, but their behavior is better with respect to the peak performance value.

Next, we will consider the power efficiency of these systems in the same way as in Section 4.4. Figure 11(b) shows the MFLOPS/s-to-Watt ratio based on the average SpMV performance and the power consumption for all the considered systems. The power consumption of the processors has been obtained from the manufacturer's documentation. For comparison purposes, the ACP (Average CPU Power) value provided by AMD for the Opteron processor was transformed into its corresponding TDP (Thermal Design Power) according to [18].

The measurements provide some important results. Regarding the SCC processor, it outperforms the Itanium2 again, but this difference increases with respect to the one observed in Figure 11(a). Overall, the SCC processor behaves better in terms of power efficiency in comparison with the performance results. If the efficiency was calculated using the average performance with optimizations (see Section 5.4), its value raises to 17.1, close to the Xeon processor. On the other hand, we must highlight that the ATI FirePro and APU show the best behavior overall in terms of efficiency. In fact, despite of the difference in performance of both platforms, their efficiency is quite similar. In particular, 48.7 and 47.8 MFLOPS/s per Watt respectively. Note that double precision arithmetic is not supported by the APU. Finally, it is worth to mention that the efficiencies of the Xeon and Opteron processors are quite similar to the observed for Tesla C1060, even when the GPU shows speedups of 2.4× and 1.7× with respect to the performance on both processors.

7. Related Work

The Single Chip-Cloud Computer (SCC) is an experimental processor but there are some interesting recent works that deal with it. Melot *et al.* [20] evaluated the performance of the SCC processor when accessing the off-chip memory. One of the main conclusions of this work is that, unlike the per-core read memory bandwidth, the memory access performance decreases when several cores write simultaneously to the memory. In addition, they found that the available bandwidth depends on the memory access pattern, and especially, in the case of irregular accesses. Gschwandtner *et al.* [2] performed an evaluation of the SCC processor using different benchmarks, but they did not consider irregular applications as the SpMV. The authors focus on benchmarking those characteristics of the SCC that previous multi- and many-core processors did not possess. In other work [4], the authors explored the performance of a memory-intensive application on the SCC. They evaluate different mappings of processes to cores as well as different communication scenarios showing significant improvement in terms of performance. The present paper is an extended version of the preliminary results shown in [21]. The main additions presented in this article are: an in-depth hardware configuration analysis, and evaluation of some of the most

successful SpMV optimization techniques, and an architectural comparison including some of the most modern GPUs and APUs.

There exist in the literature numerous works that undertake the study and optimization of the SpMV. For example, in [7], the authors studied the performance issues of the SpMV on several multicore platforms. Based on different experiments, they gave some useful optimization guidelines about the most important performance bottlenecks and how these should be addressed effectively. In a more recent work [22], the authors studied the behavior of the SpMV on a SMP-NUMA system. They focused on several issues that affect the SpMV performance: the memory and the processor affinity, as well as the influence of the memory hierarchy.

The works that deal with the SpMV optimization have traditionally been split up into those that perform a data reordering to increase locality and those focused on code restructuring techniques. Nevertheless, this is not a strict categorization and some methods incorporate both approaches.

Throughout the classic data reordering techniques, those that focus on bandwidth reduction algorithms, which stem from the Cuthill-McKee algorithm [8], are the most widely used. Oliker *et al.* [23] investigate and discuss the effects of various ordering strategies for sparse iterative solvers in a number of shared and distributed-memory machines. Unlike standard reordering algorithms, their main objective is to increase the data reuse which leads, as a side effect, to an improvement in the data locality. Coutinho *et al.* [24] compare different data reordering algorithms for the SpMV in edge-based unstructured grid computations. In a previous work, the authors of this paper also evaluate in [13] some *minimum degree*-based heuristics such as the *approximate minimum degree* algorithm [12] on multicore processors.

Regarding code reorganization, several techniques like *blocking* or *tiling* have been successfully applied to different irregular codes. Im *et al.* [15] propose register and cache blocking as SpMV optimization techniques. Also for the blocked SpMV, a performance model is developed in Buttari *et al.* [25]. The model proposed in that work allows to pick in nearly all cases the optimal block-size. In a similar line, Vuduc *et al.* [26] extend the notion of blocking in order to exploit variable block shapes by splitting the original matrix in a proper sum of submatrices, then storing each submatrix in a variation of the blocked CSR format. In a more recent work [27], a comparative study of different blocking storage techniques for sparse matrices on several multicore platforms is presented. One of the main drawbacks of these techniques is the strong dependence with the sparsity pattern of the matrix. Some authors have demonstrated that both groups of techniques are complementary [9, 10, 28].

Another novel approach to optimize the SpMV consists in the usage of data compression techniques. In [17], two compression methods targeting index and numerical values are proposed. In the first case, the authors assume that there exist parts within sparse matrices exhibiting some level of density without necessarily containing contiguous nonzeros. They demonstrate that, in some cases, these parts can contribute significantly to the storage volume reduction of the sparse matrix indexing data. Williams *et al.* [16] apply an index reduction tech-

nique, in which 16-bits indices are used when it is possible. In the same work, the authors propose additional optimization techniques for the SpMV that are evaluated on different multicore platforms. Some of the proposed techniques are: software pipelining, prefetching approaches, and register and cache blocking. Finally, Belgin *et al.* [29] introduce a representation for sparse matrices based on the observation that many matrices can be divided into blocks that share a small number of different patterns. The goal is to reduce the SpMV memory bandwidth requirements by reducing the index overhead.

8. Conclusions

The Single-Chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs as a platform for many-core software research. In this work we have studied the behavior of the Sparse Matrix-Vector product (SpMV) on the SCC processor, which is one of the most important computational kernels in scientific and engineering applications. In addition, several optimization techniques for the SpMV have been ported and evaluated on the SCC. In particular, reordering, blocking and data compression algorithms have been considered.

The effects on the SpMV performance of several issues were evaluated. First, the influence of mapping the processes to different cores. Our experiments show important degradations with sequential and parallel executions depending on the mapping configuration. According to this, we propose a mapping configuration that maps the units of execution to the available cores with the lowest distance to the memory.

The influence of the working set size in the SpMV performance was also studied. The experiments demonstrate that matrices whose working size per core fits into the L2 cache boost their performance with a high number of cores. In order to analyze the effect of the irregular accesses a modification of the SpMV code to eliminate cache misses on accesses to vector `x` has been carried out. Our observations point out that the locality has a big importance on the SpMV performance, in contrast with the behavior observed in other multicore systems [7].

Since the SCC processor allows to change the cores, mesh network and memory clock frequencies, an evaluation in terms of performance and power efficiency of different hardware configurations has been provided. We have detected that an increase in the mesh frequency is only significant when using a high number of cores, while increasing the cores and memory frequencies has an immediate effect on the SpMV performance even in the sequential case. But, we must take into account that using higher frequencies increase the power consumption with respect to the default configuration. In particular, when using the full SCC system, higher frequencies for the cores, mesh and memory increase the consumption about 22%, 4% and 1.5% respectively. However, the best power efficiency is obtained using the configurations with the memory clocked at the highest frequency.

We detected that SpMV on the SCC is very sensitive to reordering, blocking and compression optimization techniques. In particular, reordering techniques

were especially effective with those matrices with irregular nonzero patterns. Best results of the register blocking technique are obtained for those matrices with dense blocks of nonzeros in their patterns. For this reason, a combination of reorderings that favor the nonzeros grouping and blocking has been also studied. Results are very encouraging, reaching speedups up to 2.7×. Two compression methods have been also evaluated. First, a simple index reduction technique, which improves the performance of all the matrices of the testbed regardless of the characteristics of the sparsity pattern. Secondly, a technique that compress the indexing structure of the matrices exploiting the distances between elements. In this case, despite of decreasing the working set of the matrices about the 21%, this method does not obtain good results on the SCC.

Finally, an architectural comparison with several leading multicore processors and GPUs was performed. Results indicate that SCC only outperforms a dual-core Itanium2 system when considering performance and power efficiency. Best results overall in terms of performance were obtained by Tesla M2050 and ATI FirePro GPUs. However, when the power efficiency is considered, the ATI FirePro and APU are the best choices.

Acknowledgements

The authors would like to thank Intel for providing the opportunity to experiment with the many-core processor Single-Chip Cloud Computer (SCC).

This research was partly funded by Ministry of Education and Science (Spain) and Xunta de Galicia through the projects TIN2010-17541 and 09TIC002CT respectively.

References

[1] Intel Labs, SCC External Architecture Specification (EAS) (november 2010).

[2] P. Gschwandtner, T. Fahringer, R. Prodan, Performance analysis and benchmarking of the Intel SCC, in: IEEE Int. Conf. on Cluster Computing (CLUSTER), 2011, pp. 139–149.

[3] R. F. van der Wijngaart, T. Mattson, W. Haas, Light-weight communications on Intel's single-chip cloud computer processor, ACM SIGOPS Operating Systems Review 45 (2011) 73–83.

[4] M. Korch, T. Rauber, C. Scholtes, Memory intensive applications on a many-core processor, in: IEEE Int. Conf. on High Performance Computing and Communications, 2011, pp. 126–134.

[5] T. Davis, University of Florida Sparse Matrix Collection, NA Digest 97 (23), `http://www.cise.ufl.edu/research/sparse/matrices`.

[6] Intel Labs, SCC Programmer's Guide (november 2011).

[7] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, N. Koziris, Performance evaluation of the sparse matrix-vector multiplication on modern architectures, The Journal of Supercomputing 50 (1) (2009) 36–77.

[8] E. Cuthill, J. McKee, Several strategies for reducing the bandwidth of matrices, Rose and Willoughby, 1972.

[9] A. Pinar, M. Heath, Improving performance of sparse matrix-vector multiplication, in: Proc. of Supercomputing, 1999.

[10] S. Toledo, Improving memory–system performance of sparse matrix–vector multiplication, in: Proc. of the 8th SIAM Conf. on parallel processing for scientific computing, 1997.

[11] J. C. Pichel, F. F. Rivera, M. Fernández, A. Rodríguez, Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs, Microprocessors and Microsystems 36 (2) (2012) 65–77.

[12] P. R. Amestoy, T. A. Davis, I. S. Duff, An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications 17 (4) (1996) 886–905.

[13] J. C. Pichel, D. E. Singh, J. Carretero, Reordering algorithms for increasing locality on multicore processors, in: Proc. of the IEEE Int. Conf. on High Performance Computing and Communications, 2008, pp. 123–130.

[14] G. Karypis, V. Kumar, METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (1997).

[15] E. J. Im, K. A. Yelick, R. Vuduc, SPARSITY: Framework for optimizing sparse matrix-vector multiply, Int. Journal of High Performance Computing Applications 18 (1) (2004) 135–158.

[16] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix-vector multiply on emerging multicore platforms, in: Proc. of the ACM/IEEE Conf. on Supercomputing, 2007.

[17] K. Kourtis, G. Goumas, N. Koziris, Optimizing sparse matrix-vector multiplication using index and value compression, in: Proc. of the Conference on Computing Frontiers, 2008, pp. 87–96.

[18] Intel, Measuring Processor Power: TDP vs. ACP, white paper (april 2011).

[19] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on CUDA, Tech. rep., NVIDIA (2008).

[20] N. Melot, K. Avdic, C. Kessler, J. Keller, Investigation of main memory bandwidth on Intel Single-Chip Cloud Computer, in: 3rd Many-core Application Research Community Symposium, 2011, pp. 107–110.

[21] J. C. Pichel, F. F. Rivera, Experiences with the sparse matrix-vector multiplication on a many-core processor, in: Proc. of the IEEE 26th Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2012, pp. 7–15.

[22] J. C. Pichel, J. A. Lorenzo, D. B. Heras, J. C. Cabaleiro, T. F. Pena, Analyzing the execution of sparse matrix-vector product on the Finisterrae SMP-NUMA system, The Journal of Supercomputing 58 (2) (2011) 195–205.

[23] L. Oliker, X. Li, P. Husbands, R. Biswas, Effects of ordering strategies and programming paradigms on sparse matrix computations, SIAM Review 44 (3) (2002) 373–393.

[24] A. L. G. A. Coutinho, M. A. D. Martins, R. M. Sydenstricker, R. N. Elias, Performance comparison of data-reordering algorithms for sparse matrix-vector multiplication in edge-based unstructured grid computations, Int. Journal for Numerical Methods in Engineering 66 (3) (2006) 431–460.

[25] A. Buttari, V. Eijkhout, J. Langou, S. Filippone, Performance optimization and modeling of blocked sparse kernels, Int. Journal of High Performance Computing Applications 21 (4) (2007) 467–484.

[26] R. Vuduc, H. Moon, Fast sparse matrix-vector multiplication by exploiting variable block structure, in: High Performance Computing and Communications, volume 3726 of Lecture Notes in Computer Science, 2005, pp. 807–816.

[27] V. Karakasis, G. Goumas, N. Koziris, A comparative study of blocking storage methods for sparse matrices on multicore architectures, in: Proc. of IEEE Int. Conf. on Computational Science and Engineering, 2009, pp. 247–256.

[28] E. J. Im, K. Yelick, Optimizing sparse matrix vector multiplication on SMPs, in: Proc. of the 10th SIAM Conf. on parallel processing for scientific computing, 1999.

[29] M. Belgin, G. Back, C. J. Ribbens, Pattern-based sparse matrix representation for memory-efficient SMVM kernels, in: Proc. of the Int. Conf. on Supercomputing, 2009, pp. 100–109.