2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing

LDPC Decoding on the Intel SCC

Andreas Diavastos, Panayiotis Petrides, Gabriel Falcão*, Pedro Trancoso

Department of Computer Science
University of Cyprus,
Nicosia, Cyprus
Emails: {cs06da1, csp7pp5, pedro}@cs.ucy.ac.cy

*Instituto de Telcomunicações,
Dept. of Electrical and
Computer Engineering
University of Coimbra, Portugal
Email: gff@co.it.pt

Abstract

Low-Density Parity-Check (LDPC) codes are powerful error correcting codes used today in communication standards such as DVB-S2 and WiMAX to transmit data inside noisy channels with high error probability. LDPC decoding is computationally demanding and requires irregular accesses to memory which makes it suitable for parallelization. The recent introduction of the many-core Single-chip Cloud Computer (SCC) from Intel research Labs has created new opportunities and also new challenges for programmers that wish to exploit conveniently the high level of parallelism available in the architecture. In this paper we propose three different implementations: a distributed, a shared and a multi-codeword implementation, for LDPC decoding algorithms that explore the Intel SCC scaling opportunities. From the experimental results we observed that the distributed memory model couldn't scale due to the large number of messages exchanged by the parallel kernels, while the shared memory model had a limited scaling due to the overhead added by the uncacheable shared memory. On the other hand, the multi-codeword implementation scales almost linearly achieving a relative throughput of 28 for 32 cores.

1. Introduction

The inclusion of multiple cores in the same chip has become the new de-facto standard for the processor architecture. This multi-core approach has resulted as a solution to the power- and complexity-walls of previous monolithic single core processors. The continuous advances in technology result in the integration of more and more devices on the same chip. Therefore, more cores are being added to the processors. Soon we will be able to have hundreds of cores in a single chip. These new architectures, also known as many-core architectures, offer a significant performance and power benefit over previous ones, but on the other hand lead to some serious challenges. In order to scale to large numbers of cores the architectures need to be simple, so large-scale many-core processors will not have as many shared resources and most probably will not support hardware cache-coherency. In addition, large-scale many-core processors will have limited access to external main-memory and the cores themselves will be simple compute engines. Nevertheless, in order to keep programmability within acceptable ranges, it is desirable to support a standard ISA such that it may be possible to execute legacy codes without major changes. Intel has recently proposed the Many Integrated Core (MIC) Architecture [8] and the Single-chip Cloud Computer (SCC), which contains 48 simple Pentium cores. Given that the industry has proven that technically it is possible to build such large-scale many-core processors, the challenge is on exploiting the available parallelism in an efficient and effective way. In this study we use the Intel SCC processor to address these challenges in future many-core architectures to harvest the available parallelism on a widely used computation and communication intensive algorithm.

Low-Density Parity-Check (LDPC) codes have been proposed in the 1960s [5] and recaptured the attention of academia and industry in the late 1990s [10]. They represent powerful error correcting codes (ECC) used for a more error-free transmission of data through noisy and high error probability channels. They can achieve a performance close to the Shannon limit and for that reason they have been adopted today in many modern communication and storage standards [2, 3, 11]. Binary LDPC codes are linear (N, K) block codes defined by sparse parity-check matrices of dimension $(N - K) \times N$. They can be represented by bipartite Tanner graphs [13] where Bit Nodes (BN) communicate with Check Nodes (CN) they are connected to. The decod-

978-0-7695-4633-9/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2012.79

ing part of the system uses belief propagation algorithms that propagate messages between nodes linked by common edges to infer the probability of a given input bit being 0 or 1. One such algorithm is the well-know Sum-Product Algorithm (SPA), which is adopted under the context of this paper [12, 15].

Since LDPC decoders demand very intensive computation and irregular memory accesses, until recently they have been exclusively developed using VLSI hardware [1, 2, 11] in order to achieve high throughput real-time decoding. But the phase preceding the ASIC design is also a fundamental part of the process. It consists of developing good LDPC codes and simulate their coding performance, which basically consists in calculating the corresponding Bit Error Rate (BER) curves. Depending on the code length and rate, these simulations can take weeks to months to compute on a conventional CPU for evaluating the coding performance of LDPC codes able of guaranteeing extremely low BER values. Recent advances in parallel computing architectures such as the Compute Unified Device Architecture (CUDA) [7] have made LDPC decoders a reality on GPUs [4, 9] and parallel kernels have also been proposed for the Cell/B.E. [3].

This paper proposes three different strategies for using the SPA to perform LDPC decoding on the 48 core Intel SCC: the Distributed Parallel Decoder, the Shared Parallel Decoder and the Parallel Multi-codeword Decoder. From the experimental results we show that the target architecture achieves scalable parallelism if the use of memory and communication models are accommodated efficiently. For the LDPC decoding we achieve this efficiency with the multi-codeword implementation which scales almost linearly, achieving a relative throughput of 28 for 32 cores. Even though the results have been obtained for the SCC, this is without loss of generality as the SCC is a good representative of future large-scale many-core processors.

The paper is organized as follows. Section 2 addresses the properties of the SPA LDPC decoding algorithm and the Tanner graph representation that illustrates message-passing requirements. Section 3 describes the Intel SCC architecture, namely the memory hierarchy of the system and the Programming API. Section 4 addresses the parallel kernels developed for LDPC decoding on the SCC and the different parallel approaches that we have followed under the context of this work in order to obtain the desired scalability performance analysis. Section 5 presents some experimental results and Section 6 concludes the paper.

2. Belief Propagation

Belief propagation, which is also known as the SPA, consists of an iterative algorithm [15] used in information theory that calculates joint probabilities on graphs. It is typically used in a vast set of domains and applications such as channel coding and stereo vision. One of these applications includes binary LDPC codes where the SPA is used for inference calculation over bipartite graphs that apply probabilistic techniques to put a bound to the deviation of a random variable from its expected value. Given a probability graph, message-passing procedures between neighboring nodes, denoted by two nodes connected by an edge, allow all vertices to make a contribution to update the beliefs in the graph in order to infer the correct codeword. The structure of the graph is defined by matrix $\mathbf{H}$ where all edges are identified.

2.1. LDPC Decoding

The SPA calculates the maximum *a posteriori* probabilities of vertices in a graph [15]. Given a (N, K) binary LDPC code, we assume Binary Phase Shift Keying (BPSK) modulation, which maps a codeword $\mathbf{c} = (c_0, c_1, c_2, \cdots, c_{n-1})$ into a sequence $\mathbf{x} = (x_0, x_1, x_2, \cdots, x_{n-1})$, according to $x_i = (-1)^{c_i}$. Then, $\mathbf{x}$ is transmitted through an additive white Gaussian noise (AWGN) channel, producing a received sequence $\mathbf{y} = (y_0, y_1, y_2, \cdots, y_{n-1})$ with $y_i = x_i + n_i$, where n_i represents AWGN with zero mean and variance σ^2. Algorithm 1 illustrates the SPA applied to LDPC decoding and it mainly consists of two distinct horizontal and vertical computationally intensive kernels defined by (1), (2) and (3), (4), respectively. Kernel 1, also defined as Horizontal Processing, calculates the message updates from CN_m to BN_n, by performing accesses to $\mathbf{H}$ in a row-major order. These obtained messages indicate the probability of BN_n being 0 or 1. Kernel 2, which is also known by Vertical Processing, updates messages sent from BN_n to CN_m and they consider accesses to $\mathbf{H}$ in a column-major order. After Kernel 2 concludes, in (5) and (6) the *a posteriori* pseudo-probabilities are calculated and finally in (7) is performed hard decoding to obtain the decoded word $\hat{\mathbf{c}}$ at the end of an iteration. This iterative procedure automatically ends if the decoded word $\hat{\mathbf{c}}$ verifies all parity-check equations of the code ($\hat{\mathbf{c}}\,\mathbf{H}^T = \mathbf{0}$), or if the maximum number of iterations ($\mathbf{I}$) is reached.

2.2. Tanner Graph Illustrating Communications

The underlying data structure that supports the decoding of the LDPC code is the Tanner graph, as mentioned before. A bipartite Tanner graph [13] represents adjacent connections between N BNs and $N - K$ CNs. It allows finding which nodes communicate with each other, a fundamental operation for the processing of Kernels 1 and 2 from Algorithm 1 and can be defined by a parity-check sparse

Algorithm 1 SPA

1: {Initialization}
$p_n = p(y_i = 1)$;
$q^{(0)}_{nm}(0) = 1 - p_n$; $q^{(0)}_{nm}(1) = p_n$;
2: **while** ($\hat{\mathbf{c}}\,\mathbf{H}^T \neq \mathbf{0} \wedge \mathrm{i} < \mathrm{I}$) {c-decoded word; I-Max. no. of iterations.} **do**
3: {For all node pairs (BN_n, CN_m), corresponding to $\mathbf{H_{mn}} = \mathbf{1}$ in the parity check matrix $\mathbf{H}$ of the code **do**:}
4: {Compute the message sent from CN_m to BN_n, that indicates the probability of BN_n being 0 or 1:}

(Kernel 1 - Horizontal Processing)

$$r^{(i)}_{mn}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in N(m)\backslash n} \left(1 - 2q^{(i-1)}_{n'm}(1)\right) \quad (1)$$

$$r^{(i)}_{mn}(1) = 1 - r^{(i)}_{mn}(0) \quad (2)$$

{where $N(m)\backslash n$ represents BNs connected to CN_m excluding BN_n.}
5: {Compute message from BN_n to CN_m:}

(Kernel 2 - Vertical Processing)

$$q^{(i)}_{nm}(0) = k_{nm}(1 - p_n) \prod_{m' \in M(n)\backslash m} r^{(i)}_{m'n}(0) \quad (3)$$

$$q^{(i)}_{nm}(1) = k_{nm} p_n \prod_{m' \in M(n)\backslash m} r^{(i)}_{m'n}(1) \quad (4)$$

{where k_{nm} are chosen to ensure $q^{(i)}_{nm}(0) + q^{(i)}_{nm}(1) = 1$, and $M(n)\backslash m$ is the set of CNs connected to BN_n excluding CN_m.}
6: {Compute the *a posteriori* pseudo-probabilities:}

$$Q^{(i)}_n(0) = k_n(1 - p_n) \prod_{m \in M(n)} r^{(i)}_{mn}(0) \quad (5)$$

$$Q^{(i)}_n(1) = k_n p_n \prod_{m \in M(n)} r^{(i)}_{mn}(1) \quad (6)$$

{where k_n are chosen to guarantee $Q^{(i)}_n(0) + Q^{(i)}_n(1) = 1$.}
7: {Perform hard decoding} $\forall n$,

$$\hat{c}^{(i)}_n = \begin{cases} 1 & \Leftarrow \quad Q^{(i)}_n(1) > 0.5 \\ 0 & \Leftarrow \quad Q^{(i)}_n(1) < 0.5 \end{cases} \quad (7)$$

8: **end while**

binary $\mathbf{H}$ matrix as the one depicted in Figure 1. During the processing of Kernel 1 (Horizontal Processing), each CN updates the corresponding BNs that are connected to it. The first row of Figure 1 exemplifies CN_0 updating BN_1, BN_2, BN_5 and BN_7. An identical approach is followed for Kernel 2 (Vertical Processing), where each BN updates the CNs linked to it.

For LDPC codes with large dimensions (e.g. $N = 64800$-bit in the DVB-S2 standard for satellite communications), it can be clearly seen from Figure 1 that the number of messages exchanged between nodes of the graph during one single iteration can be extremely high. If we consider that usually several iterations occur (e.g. dozens), then it becomes clear how communications have an impact in the performance of such a parallel algorithm.

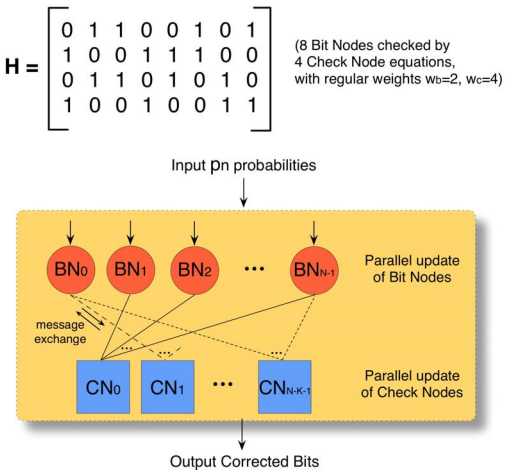

Figure 1. Tanner graph representing a 4×8 $\mathbf{H}$ matrix

3. Intel SCC Many-core Architecture

The Intel Single-chip Cloud Computer (SCC) experimental processor is a 48-core 'concept vehicle' created by Intel research Labs as a platform for many-core software research [6]. This processor consists of 24 dual-core tiles interconnected by a 2D-grid network as Figure 2 illustrates. The tiles are organized in a 6x4 mesh with each tile containing:

- Two P54C cores with 16KB L1 and 256KB L2 cache dedicated to each core;
- A 16KB Message Passing Buffer (MPB) for storing messages to be sent to other cores (8KB per core);
- A Traffic Generator for testing the performance capabilities of the mesh network;
- A Mesh Interface Unit (MIU) to connect to a router of the network;
- Two test-and-set registers.

As mentioned before the MIU connects each tile to a router. All routers are connected together using a mesh interconnection network. The MIU is responsible for packetizing data that will go onto the network and de-packetizing data that come from the network. This unit is shared by the two cores in a round-robin scheme [14]. The maximum main memory the system can support is 64GB and the 32-bit memory addresses of the core are translated into system addresses by the MIU through a lookup table (LUT). This system main memory is located outside the chip and in order for the cores to access the data, the chip contains four DDR3 Memory Controllers (MC).

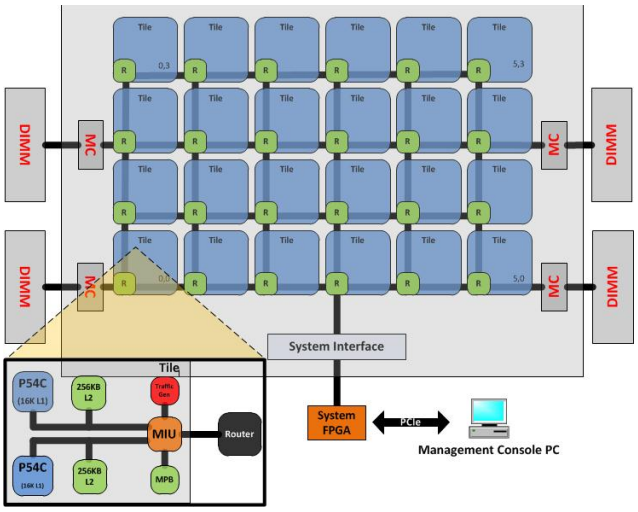

Figure 2. Intel SCC top-level architecture and tile top-level architecture

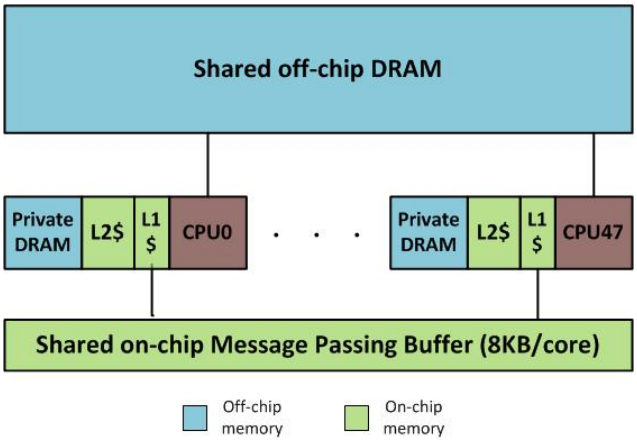

Figure 3. Intel SCC Memory Architecture as used by the programmer through the RCCE message passing API

In Figure 3 we present an overview of the SCC Memory Architecture and how the programmer can view the system and the core's memory through the RCCE message passing API. The SCC supports both distributed and shared memory programming models as it can be seen from Figure 3 and the system memory is configured and separated in four regions:

- Private off-chip memory: Each core's LUT is configured such that a specific region of the off-chip DRAM (equally divided between all cores) is only accessible by that single core. This is the cores' main memory. The addresses corresponding to this memory are served by a single MC;
- Shared off-chip memory: This region of the off-chip DRAM is mapped by all LUTs and all cores have direct access to it through any MC. As the system does not provide hardware cache-coherency, the data that is in the shared address space is not cached, thus avoiding errors in the programs. This, though, results in large overheads in the latency to access shared data;
- Shared on-chip memory: Also called Message Passing Buffer (MPB), this on-chip SRAM is cacheable to the L1 caches of the cores and is used for message exchanging between the cores. Each tile is equipped with 16KB of this type of memory, equally divided to each core. This results to 8KB of shared on-chip memory for each core and totalling to 384KB for the entire system;
- L2 cache: L2 cache is only used by the private off-chip memory since all other memory types are uncacheable to L2, for coherency reasons.

3.1. Intel SCC Programming API

In this section we briefly explain the RCCE Programming API provided by Intel [14]. The RCCE API supports the Single Program Multiple Data (SPMD) model of execution. More specifically, all Units of Execution (UEs) are created at the same time but with no ordering to their execution among them. Intel SCC supports C/C++ language with RCCE API extensions to implement the message passing communication of the system. RCCE is the SCC communication environment and implements *send* and *receive* functions for sending and receiving messages to and from the cores. The RCCE API provides control to the MPB and also controls allocation in the shared off-chip DRAM memory. Some more advanced features that a programmer can find in this API is the power and voltage control functions. These functions allow the programmer to change the voltage and the working frequency of the voltage islands (groups of tiles), therefore controlling the total power consumption of the chip. More information on the RCCE communication environment and the functions used can be found in [14].

4. Parallel LDPC Decoding on the Intel SCC

In this section we describe how we implemented three different parallel models on the Intel SCC. As explained in Section 3 the SCC processor can support both distributed and shared off-chip memory models, which from now on will be referenced as distributed and shared memory. Taking advantage of the shared memory model we implemented the Shared Parallel LDPC decoder. In order to explore the private off-chip memory of each core we implemented the

Distributed Parallel and the Parallel Multi-codeword LDPC decoders.

4.1. Data Structures

The H matrix of an LDPC code defines the Tanner graph, whose edges represent the bidirectional flow of messages exchanged between BNs and CNs as detailed in Section 2-B. The information about $\mathbf{H}$ is separately coded in two independent data streams, $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$, suitable for processing Kernel 1 (Horizontal Processing) and Kernel 2 (Vertical Processing), respectively, as extensively addressed in [4]. In each Kernel, data elements may be read sequentially but have to be stored in non-contiguous positions (or vice-versa) due to the random nature of LDPC codes, which defines expensive random memory accesses. These costly write operations demand special effort from the programmer [4] in order to efficiently accommodate them in parallel architectures with distinct levels of memory hierarchy.

4.2. Distributed Parallel Decoder

In the Distributed Parallel Decoder we store a decoding graph in the private off-chip memory of each core. Each core then takes one portion of a codeword to decode. Except from the dependencies between Kernel 1 and Kernel 2 we also need to update all cores about the changes each core makes in each iteration inside the Kernels. Since we are not using a shared address space, we must use messages to exchange the computed information in each kernel execution. When a core starts computation associated to a CN, it must pass this information to all cores that decode the same codeword (Figure 4). At the end of each iteration, each core will update all other cores of the changes it conducted.

This communication is represented with the horizontal ar-

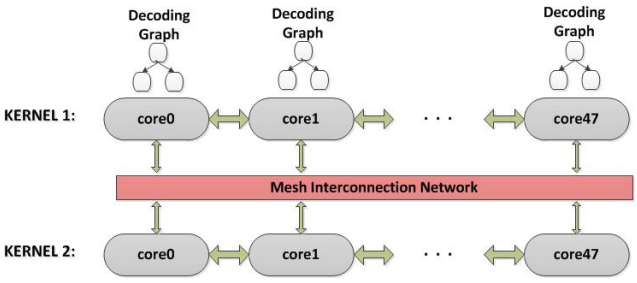

Figure 4. Distributed memory model.

rows in Figure 4. The vertical arrows represent the last messages computed and broadcasted to all cores before moving the execution to Kernel 2. As shown in the figure, all these messages are traveling through the Interconnection Network of the chip. This message exchange described above is implemented using broadcast from one core to all the others. To implement the broadcast operation on the SCC we used three different approaches. First we used the RCCE API function call RCCE_bcast(), which implements the broadcast on the SCC by passing as arguments a pointer to the data buffer, the size of the buffer and the communicator pointer that shows which cores will participate in this message transaction. We call this the broadcast (bcast) implementation. The second implementation follows the traditional send/receive model. By using the RCCE_send() and RCCE_ recv() functions from the RCCE API and defining which core sends and which core receives at each iteration we manually implemented a broadcasting system within our application. At the end of each iteration in the Kernel execution we synchronize the cores and start sending update messages from one core at a time to all the others. As soon as all cores send their update messages to all other cores, the execution of the Kernels proceeds in parallel. This helps us to create a faster broadcasting implementation by avoiding the extra overheads of the previous implementation, which are mostly related with runtime error checking overheads. We call this the send/receive (send/recv) implementation. The third implementation consists of accessing the MPB directly. All messages in the SCC system are passed through the MPB of each tile. We call this the fast broadcast (fastbcast) implementation. For both send/receive and broadcast implementations the messages will eventually go through the MPB to reach the destination. Instead of using the runtime system to guide these messages through the MPB we use direct access to it by copying data to and from the MPBs of all cores.

4.3. Shared Parallel Decoder

Taking advantage of the 64MB off-chip shared memory the system offers, it is possible to store the decoding graph in the shared address space. This way, instead of using a message passing technique like in the Distributed Parallel Decoder we take advantage of the shared off-chip memory where all cores can see the changes made by any other core. All messages that previously needed to be passed as messages through the Interconnection Network, are now stored in the shared memory that each core can directly access. In Figure 5 we show the representation of this model. At the bottom of the figure we have Kernel 1 executions in which we update all the CNs and at the top of the figure we have Kernel 2 where we update BNs. The execution flow starts from the left and finishes at the right of the figure. Figure 5 also shows that there is only one instance of the decoding graph for all the cores and it is stored in the shared memory address space. Again, each core takes a portion of the execution as it happens to any Single Program Multiple Data (SPMD) application and computes on the input of each Kernel and stores the result back in the shared address space

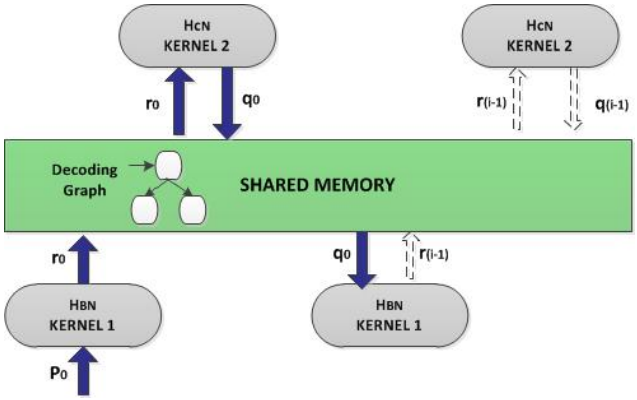

Figure 5. Shared memory model.

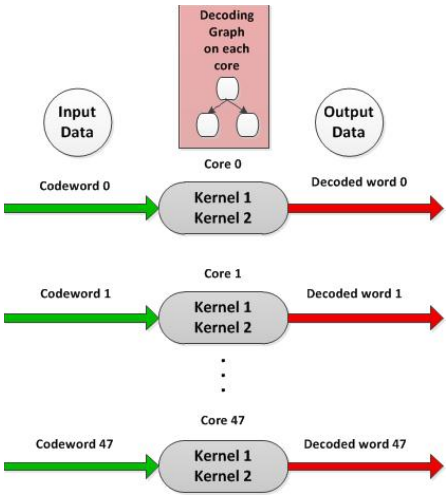

Figure 6. Multi-codeword model.

for other cores to read. This store-back to shared memory is mandatory since all other cores will need this information for future calculations.

4.4. Parallel Multi-codeword Decoder

In this Section we describe a Multi-codeword Decoder where the system decodes multiple codewords in parallel. In this model we exploited the independence of the cores from each other. Each core on the SCC chip runs a separate Linux image that allows it to be independent and able to execute different instructions or even programs from the rest of the cores. What we implemented here is a Multi-codeword Decoder where each core has an instance of the Decoding graph stored in its private memory. We keep these instances private to each core and we give each core a different codeword to decode. This means that, when using 48 cores, each core decoding a different codeword, the system will be decoding 48 codewords simultaneously. Figure 6 represents this multi-codeword decoding strategy. The execution flow starts from the left and finishes on the right. Starting, each core has its own decoding graph located in its private memory as mentioned before. Then, each core gets a different word as an input, executes iteratively both Kernel 1 and Kernel 2 on the input word and finally produces one decoded word as output, which leads to N decoded words when using N cores.

5. Experimental Results

5.1. Experimental Setup

For this work we used the Intel SCC experimental processor, RockyLake version. The system has a total of 32GB main memory and we adopted the default voltage and frequency settings for normal execution, which are 533MHz for the tile, 800MHz for the mesh interconnection network and 800MHz for the DDR3 memory and Memory Controllers. The operating system used for the Intel SCC cores is the default Linux kernel provided by the RCCE SCC Kit 1.3.0 and the compiler we used to cross compile our programs for the SCC was gcc v3.4.5. The host PC responsible for controlling the applications execution on the Intel SCC, is configured with an Intel Core i7 processor running at 3.7GHz and with 4GB of memory. The connection between the host and the SCC is managed through a PCIe Expansion Card and a PCIe x4 cable. For porting and executing the applications on the SCC we used the RCCE v1.3.0 tool-chain.

For the purpose of this work we used the LDPC decoding implementation for multi-core architectures in OpenMP from [4] and implemented it for the SCC using the RCCE API for the three parallel models described in Section 4. For our evaluation we used two different decoding graph and codeword input sizes, the 4000×8000 and the 10000×20000 (number of CNs $\times$ number of BNs of the decoding graph and with the number of BNs also representing the length N in bits of the codeword to be decoded).

It is relevant to notice that most results presented are normalized and not absolute values. This is due to the fact that the goal in this study is to show the scalability of a research architecture for future many-core systems. Thus we intend to show the potential of such architecture and not results to compete with other alternative existing architectures. Regarding the algorithm studied, its goal is to achieve a high throughput. As such, most results show the *Normalized Throughput* metric, which is defined as the Throughput relative to the sequential execution on a single SCC core using its private off-chip memory. So, if a setup achieves a *Normalized Throughput* of 2 this means that the setup is able to achieve a Throughput which is twice as large as

the Throughput achieved by the serial execution on a single core. Finally, even though we have executed all experiments for the 4000×8000 and 10000×20000 problem sizes, we show the results for the 4000×8000 case as there are no relevant differences in the normalized results.

Notice that we parallelize the implementations to use all 48 cores except for the Distributed Parallel Decoder where we used only 32 cores. The reason for that is the fact that the communication overhead was too high for a larger number of cores.

5.2. Distributed Parallel Decoder

The first results that we present are the ones corresponding to the Distributed Parallel Decoder implementations. In Figure 7 we present the *Normalized Throughput* values for the send/receive, broadcast and fast broadcast implementations, for different number of cores. These implementations are represented as *msg(send/recv)*, *msg(bcast)*, and *msg(fastbcast)*, respectively.

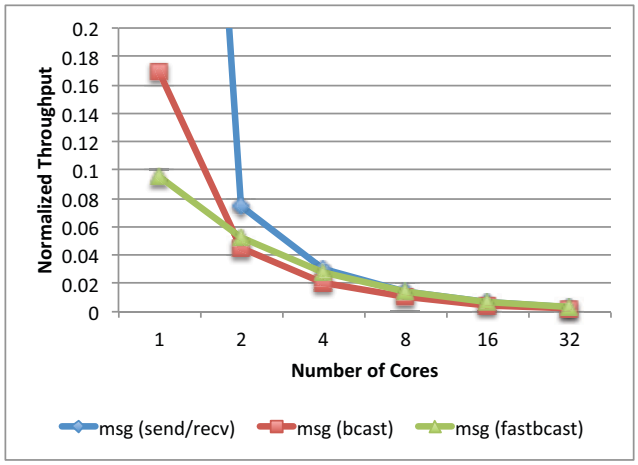

Figure 7. Normalized throughput for different number of cores for the distributed parallel decoder implementations.

From Figure 7 it is possible to observe that the highest throughput is achieved for the send/receive implementation, for most configurations. For the larger number of cores the best results are achieved for the fast broadcast implementation, where we use direct memory access on the MPB by copying the data to send directly on the receiver's MPB. The reason why the broadcast implementation is the slowest one is related to the fact that it performs a large number of message copies as to send the data to all needed destinations. The fast broadcast implementation is an optimized version of the same operation that avoids unnecessary data copies. The send/receive seems to be more efficient for a small number of cores as it sends fewer messages while the fast broadcast seems to be more efficient for a large number of cores as it reduces the number of data copies.

The most relevant result from this chart though is the fact that the relative throughput is always smaller than one and also that the relative throughput decreases as the number of cores increases. A more detailed analysis of the execution time has shown that the communication time accounts for more than 95% of the total execution time. As such, the communication operations will dominate the execution. Also, a detailed breakdown of the execution time can be seen in Figure 8, normalized to the total execution time on the setup with two cores.

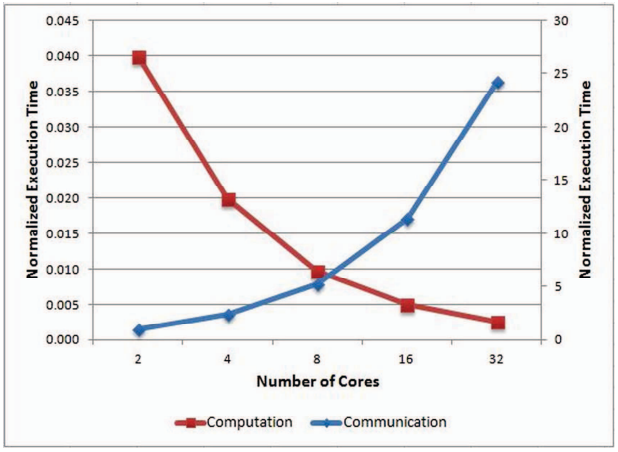

Figure 8. Normalized execution time for different number of cores for the distributed send/receive parallel decoder implementations.

The results in Figure 8 show that while the parallelism is resulting in a significant reduction of the computation time, this time was always a small portion of the total execution time. The communication time, on the other hand, is increasing significantly for the increasing number of cores. This is due to the fact that more messages are sent to more destinations. Overall, because the communication time dominates, we have a poor performance for the Distributed Parallel Decoder implementation.

5.3. Shared Parallel Decoder

In this Section we present the results for the Shared Parallel Decoder implementation. The normalized throughput values for this implementation are shown in Figure 9.

The results in Figure 9 show that the throughput for this implementation increases with the number of cores up to nearly 1, which is reached for the 4 core configuration.

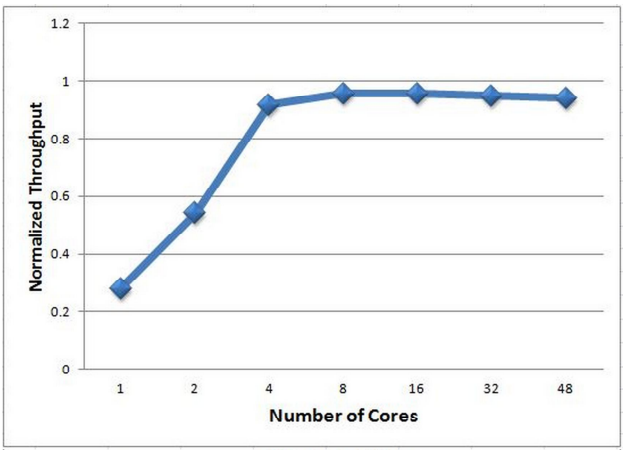

Figure 9. Normalized throughput for different number of cores for the shared parallel decoder implementation.

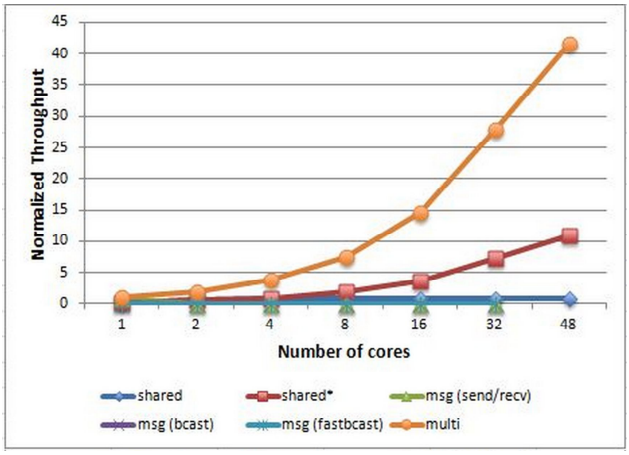

Figure 10. Normalized throughput for different number of cores for the different implementations.

This means that the Shared Parallel Decoder implementation achieves a speedup up to the 4 core configuration. Nevertheless, the results are never better than the results achieved for the sequential execution on a single core when using the private off-chip memory. This is due to two factors. First the large number of memory transfers to the off-chip shared address space that take place while executing the Kernels. Second, when using a shared address space, the data items in the shared space are not cached. This is done as to avoid problems in the execution due to the lack of hardware cache-coherency. But obviously this strategy results in large overheads as a consequence of not being able to benefit from data reuse that could be stored in the cache.

5.4. Parallel Multi-codeword Decoder

Finally we present the results for the Multi-codeword Decoder implementation. We show in Figure 10 the normalized throughput for all implementations, the ones described in the previous sections and two new ones: *multi* which is the simple multi-codeword implementation that consists of replicating the serial execution on multiple cores and *shared*$*$ which represents the replication of the best Shared Parallel Decoder setup, which was achieved using 4 cores. Thus for 8 cores we launch two instances of the shared 4 core implementations.

From Figure 10 it is possible to observe that clearly the best implementation is the multi-codeword as it achieves a near linear throughput increase with the increasing number of cores. For a setup with 32 cores we achieve a relative throughput of 27.8. This high throughput is achieved due to the fact that the architecture is made in a way that it is able to support a large amount of parallel tasks running on the different cores. For this particular application the interconnection network and the memory controllers do not seem to be a limitation to the performance.

Another interesting fact that can be observed is that even though the *shared*$*$ implementation achieves much better throughput than the original *shared* implementation, its results are still far from the results obtained with the *multi* approach. As already mentioned before, this is mostly due to the fact that for the *shared* implementations the caches are bypassed resulting in higher overall memory latency. Overall, the multi-codeword decoder results show that the SCC is an excellent architecture to exploit throughput parallelism as long as bandwidth is not a limitation, which is the case for the LDPC application.

6. Conclusions and Future Work

In this paper we address the development of parallel LDPC decoders on the SCC many-core processor from Intel. We propose three different execution models that are able to overcome limitations imposed both by the algorithm and parallel architecture, namely the Shared Parallel, the Distributed Parallel and the Parallel Multi-codeword Decoder. Although we observed that the large number of messages exchanged didn't allow the distributed approach to scale and the large number of uncacheable memory accesses limited the scaling of the shared approach, we managed to scale the throughput almost linearly on the SCC architecture by using the Parallel Multi-codeword Decoder. Our experimental results show that using the multi-codeword approach the relative throughput achieved is almost 28 for 32 cores.

As future work we plan focusing on the mapping of data that is usually irregularly accessed, into equivalent data blocks that occupy contiguous memory positions in order to control memory accesses more efficiently and improve the global performance of the parallel algorithm. Also, we plan to explore techniques to better utilize the on-chip memory as to cache the data in order to improve the shared memory implementation.

7. Acknowledgment

The authors would like to thank Intel Labs for lending the Intel SCC research processor.

References

[1] L. Chih-Hao, Y. Shau-Wei, C. Chih-Lung, C. Hsie-Chia, L. Chen-Yi, H. Yar-Sun, and J. Shyh-Jye. An ldpc decoder chip based on self-routing network for ieee 802.16e applications. *IEEE Journal of Solid-State Circuits*, 43(3):684–694, March 2008.

[2] J. Dielissen, A. Hekstra, and V. Berg. Low cost ldpc decoder for dvb-s2. *Proceedings of Design, Automation and Test in Europe (DATE '06), Munich, Germany*, March 2006.

[3] G. Falcao, V. Silva, and L. Sousa. High coded data rate and multicodeword wimax ldpc decoding on cell/be. *IET Electronics Letters*, 44(24):1415–1417, November 2008.

[4] G. Falcao, L. Sousa, and V. Silva. Massively ldpc decoding on multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(2):309–322, February 2011.

[5] R. G. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, January 1962.

[6] J. Howard and et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. *In Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, February 2010.

[7] http://developer.nvidia.com/object/cuda.ht ml. Cuda homepage.

[8] http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core architecture.html. Intel many integrated core architecture.

[9] J. Hyunwoo, C. Junho, and S. Wonyong. Massively parallel implementation of cyclic ldpc codes on a general purpose graphics processing unit. *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS'09)*, pages 285–290, October 2009.

[10] D. J. C. Mackay and R. M. Neal. Near shannon limit performance of low density parity check codes. *IEE Electronics Letters*, 32(18):1645–1646, August 1996.

[11] S. Muller, M. Schreger, M. Kabutz, M. Alles, F. Kienle, and N. Wehn. A novel ldpc decoder for dvb-s2 ip. *Proceedings of Design, Automation and Test in Europe (DATE '09), Nice, France*, March 2009.

[12] L. Shu and J. C. Daniel. *Error Control Coding.* Prentice Hall, 2004.

[13] R. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, September 1981.

[14] G. Timothy, Mattson, and et al. The 48-core scc processor: the programmer's view. *In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking and Storage Analysis*, April 2010.

[15] S. B. Wicker and S. Kim. *Fundamentals of Codes, Graphs, and Iterative Decoding.* Kluwer Academic Publishers, 2003.