Data Mining Algorithms on the Cell Broadband Engine

Rubing Duan and Alfred Strey

Institute of Computer Science, University of Innsbruck,
Technikerstrasse 21a, A-6020 Innsbruck, Austria
{rubing.duan,alfred.strey}@uibk.ac.at

Abstract. The Cell Broadband Engine (CBE) is a new heterogeneous multi-core processor from IBM, Sony and Toshiba, and provides the potential to achieve an impressive level of performance for data mining algorithms. In this paper, we describe our implementation of three important classes of data mining algorithms: clustering (k-Means), classification (RBF network), and association rule mining (Apriori) on the CBE. We explain our parallelization methodology and describe the exploitation of thread- and data-level parallelism in each of the three algorithms. Finally we present experimental results on the Cell hardware, where we could achieve a high performance of up to 10 GFLOP/s and a speedup of up to 40.

Keywords: Cell Broadband Engine, multi-core, k-Means, RBF, Apriori.

1 Introduction

The rapid evolution in sub-micron process technologies enables the manufacturing of multi-processor system with a large number of processing cores per chip. Such multi-core architectures are mostly built by replicating a standard processor design with its local caches several times and adding an on-chip interconnection network for coupling the cores and the external bus interface. The management of several threads on the available cores is done by the operating system. An alternative solution represents the Cell Processor developed by IBM, Sony and Toshiba [1,2]. Here a simplified PowerPC core (PPU) controls 8 Synergistic Processing Elements (SPEs) that only operate on data read from local stores. The operating system only starts one thread on the PPU; the creation of the SPE threads and all data transfers between PPU and SPEs must be explicitly controlled by the application program. The Cell processor was mainly designed to accelerate multi-media and graphics algorithms, but it is also well suited for various algorithms from other application areas (e.g. digital signal processing, data compression/decompression, data encryption/decryption) [2].

Data mining represents a rather new application area of High Performance Computing (HPC). It mainly deals with the finding of useful patterns in very large data sets. Its algorithms are not only based on standard floating-point calculations, but often operate on simple integer numbers. Three important classes

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 665–675, 2008.
© Springer-Verlag Berlin Heidelberg 2008

of data mining algorithms exist: classification, clustering and association rule mining. To analyze the suitability of the Cell architecture for such algorithms, a representative algorithm of each of the three classes has been selected and implemented exemplarily on the Cell processor: the k-means algorithm for clustering, a neural network for classification and the Apriori algorithm for association rule mining. The performance of these algorithms on the Cell chip will be analyzed and compared with the performance on Xeon and Opteron.

The rest of this paper is organized as follows: Section 2 describes the architecture of the CBE. In Section 3 we present the data mining algorithms. The optimized algorithms for CBE are presented in Section 4. Experimental results are described in Section 5. We overview the related work in Section 6 and conclusions are drawn in Section 7.

2 The Cell Broadband Engine Architecture

The Cell Broadband Engine architecture was jointly developed by Sony Computer Entertainment, Toshiba, and IBM. Its first commercial application was in Sony's PlayStation 3 game console. The Cell architecture emphasizes efficiency/watt, prioritizes bandwidth over latency, and favors peak computational throughput over simplicity of program code. For these reasons, Cell is widely regarded as a challenging environment for software development.

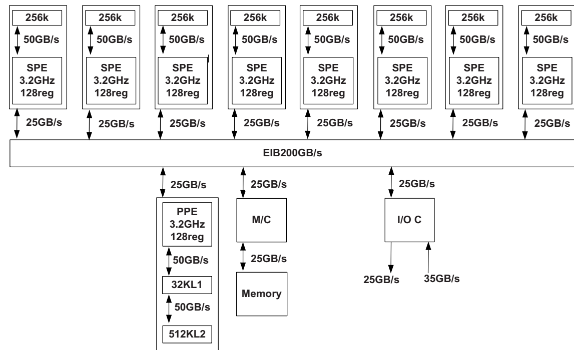


Fig. 1. The architecture of the Cell processor

The Cell processor can be split into four components: external input/output structures, the main processor called the Power Processing Element (PPE) (a two-way simultaneous multithreaded Power ISA v.2.03 compliant core), eight fully-functional co-processors called the Synergistic Processing Elements (SPEs), and a specialized high-bandwidth circular data bus connecting the PPE, input/output elements and the SPEs, called the Element Interconnect Bus (EIB), as depicted in Figure 1. Each SPE can operate in a SIMD-like (Single Instruction, Multiple Data) way on several elements in parallel that are packed into a 128 bit vector. The PPE memory is not shared with the SPEs, all data transfers between SPE and PPE memory are realized by DMA and must be programmed explicitly. The PPE, which is capable of running a conventional operating system, has control over the SPEs and starts, stops, interrupts, and schedules threads running on the SPEs. For the synchronization of threads and for the transfer of short control words a special mailbox system can be used.

Algorithm 1. k-Means

Input: D = Dataset
k = the number of centers
Output: Set of k centroids $c \in C$ representing a good partitioning of D into k clusters

1: Select the initial cluster centroids c
2: **repeat**
3: changed=0
// Find the closest centroid to every data point d ...
4: **for all** data point $d_i \in D$ **do**
5: $assignedCenter = d_i.center$
6: **for all** center $c_j \in C$ **do**
7: Compute the squared Euclidean distance $dist = dist(d_i, c_j)$
8: **if** $dist < d_i.centerDistance$ **then**
9: $d_i.centerDistance = dist$
10: $d_i.center = j$
11: **end if**
12: **end for**
13: **if** $d_i.center <> assignedCenter$ **then**
14: changed++
15: Recompute $c_j.new$ for next iteration
16: **end if**
17: **end for**
18: **until** changed==0

3 Data Mining Algorithms

In this section, we briefly sketch the algorithms under study.

3.1 K-Means Algorithm for Clustering

The k-means algorithm is one of the simplest unsupervised learning algorithms to cluster n objects based on attributes into k partitions, $k < n$. The main idea is to define k centroids, one for each cluster. The next step is to assign each object to the group characterized by the closest centroid. When all objects have been assigned, recalculate the positions of the k centroids. The last steps are repeated until the centroids no longer move. The complete k-means algorithm is presented in Alg. 1.

3.2 RBF Neural Network for Classification

A radial basis function network (RBF) is an artificial neural network model with two layers that is suitable for approximation and classification. It uses

neurons with radial basis functions as activation functions in the first (hidden) neuron layer and linear neurons in the output layer. Each RBF neuron j in the hidden layer (see Fig. 2) computes the squared Euclidean distance x_j between an input vector $\mathbf{u}$ and the weight vector c_j (represented by the jth column of a weight matrix C) and applies a radial symmetric output function f (typically a Gaussian function) to x_j. The resulting output $y_j = f(x_j)$ is communicated via weighted links w_{jk} to the linear neurons of the output layer where the sum z_k is calculated.

To achieve good results, the RBF network requires a proper initialization of all weights c_{ij} (e.g. by a clustering algorithm) and of the width σ_j of the Gaussian bells in all RBF neurons (according to the distances between the cluster centers). After the initialization, the network is trained by a gradient descent training algorithm that adapts all wights c_{ij}, w_{ij} and σ_j (the center coordinates, heights and widths of Gaussian bells) according to the error at the network outputs. The complete RBF training algorithm is presented in Alg. 2. Data mining applications can either perform the RBF training itself or a classification of new inputs by an already trained RBF network. In the later case only the forward phase of the algorithm is required.

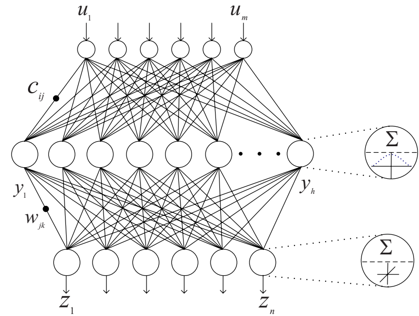


Fig. 2. Architecture of a $m - h - n$ RBF network (with m input nodes, h RBF nodes and n linear outputs nodes)

Algorithm 2. RBF

Input: Training set of patterns (u, t)
Output: Trained RBF network with weights c_{ij}, w_{jk} and s_j

1: Initialize all c_{ij}, w_{jk} and s_j
2: **for** $loop = 0; loop < max_iterations;$ **do**
3: Select one pattern u and corresponding output class t
//Forward phase for classification
4: $x_j = \sum_{i=1}^{m} (u_i - c_{ij})^2$
5: $y_j = f(x_j) = e^{-x_j/2\delta^2} = e^{-x_j s_j}$
6: $z_k = \sum_{j=1}^{h} y_j w_{jk}$
//Backward phase for training
7: $\delta_k^z = t_k - z_k$
8: $\delta_j^y = \sum_{k=1}^{n} \delta_k^z w_{jk}$
9: $s_j = s_j - \eta_s x_j y_j \delta_j^y$
10: $w_{jk} = w_{jk} + \eta_w y_j \delta_k^z$
11: $c_{ij} = c_{ij} + \eta_c (u_i - c_{ij}) \delta_j^y y_j s_j$
12: **end for**

Algorithm 3. Apriori

Input: D: database over the set of items $\jmath$,
Output: F: the set of frequent itemsets

1: $k = 1$; $C_k = \jmath$
2: **while** $C_k \neq 0$ **do**
3: support_count(D, C_k)
4: **for all** candidates $c \in C_k$ **do**
5: **if** $c.support \geq minsup$ **then**
6: $F_k = c$
7: **end if**
8: **end for**
9: $C_{k+1} = candidate_generation(F_k)$
10: $k = k + 1$
11: **end while**
12: $F = \cup_{j=1}^{k} F_j$

3.3 Apriori for Association Mining

Apriori is a classic data mining algorithm for learning association rules. Given a set of itemsets, the algorithm attempts to find subsets of k elements which are common to at least a minimum part *minsup* of the itemsets. The algorithm uses a bottom-up approach: in each iteration frequent subsets of k elements are extended to subsets of $k+1$ elements (candidate generation, see Alg. 3), and groups of candidates are tested against the data. The algorithm terminates when no further successful extensions are found. The complete Apriori algorithm is presented in Alg. 3.

4 Optimization on the Cell

We optimized the parallel implementation of the three algorithms using the following methods: (1) SPE thread parallelism. The key is to minimize the communication and the number of synchronization points. (2) SPE data parallelism. Together with SPE thread parallelism, vectorization can be used to reduce the execution time of k-means and RBF (see code snippets in Listings 1.1 and 1.2). This approach was chosen because some data mining algorithms have the following four characteristics: first, inherently parallel; second, wide dynamic range, hence floating-point based; third, regular memory access patterns; last, data independent control flow. Thus the algorithms could be restructured to leverage the SIMD intrinsics available on the Cell. (3) Data overlay. A double buffer scheme is introduced to overlap calculation and transfer. While part i of the data is transferred into one buffer, we concurrently execute the computation on data part $i-1$ from the other buffer. The performance might be improved if calculation time and transfer time of each part are approximately identical. (4) Elimination of memory consuming parts of the code.

4.1 K-Means on the Cell

It is straightforward to parallelize k-Means. We partition the input datasets such that the number of records on each processor is approximately equal. Note that there are two main constraints to the address and the number of records being passed to a SPE. First, the all addresses must be aligned to 16-byte boundary. Second, the Cell only supports aligned transfer sizes of 1, 2, 4, or 8 byte, and multiples of 16 byte, with a maximum transfer size of 16 kbyte.

Vectorization has the biggest impact in terms of relative gains that can be achieved by calculating distance at once. To take advantage of vectorization, the user must explicitly program the parallel execution in the code by applying special SIMD instructions.

Listing 1.1. Scalar distance calculation

```
1  for(i=0;i<dim;i++)
2      distance = distance+(float)(p[i]-c[i])*(p[i]-c[i]);
```

Listing 1.2. Vectorized distance calculation

```
3  float results[4] __attribute__ ((aligned(16)));
4  vector float *vA = (vector float *) p;
5  vector float *vB = (vector float *) c;
6  vector float *vC = (vector float *) results;
7  vector float vD;
8  for(i=0; i<iter; i++)
9  {
10     vD = spu_sub(vA[i],vB[i]);
11     vC[0] = spu_madd(vD,vD,vC[0]);
12 }
13 for (i = 0; i < 4; ++i)
14     distance = distance + results[i];
```

We include an example showing a snippet of code before and after vectorization in Lsts. 1.1 and 1.2, respectively. Here we make use of two intrinsics, namely *spu_sub* and *spu_madd*. *spu_sub* subtracts corresponding elements of vector vA and vB and stores the result in vD. *spu_madd* multiplies the elements of vD and vD and adds the results to vC. The vectorized code can effective reduce the number of operations, especially when the number of dimensions is much greater than 4. The complete pseudo-code for a SPE is shown in Alg. 4.

4.2 RBF on the Cell

The optimization of RBF is similar to that of k-Means. We need to partition the matrices c_{ij}, w_{jk} and the vectors δ_{zk}, δ_{yj} carefully so that only a part is stored and used in each SPE. Synchronization only occurs at the end of the forward phase for classification (after step 6 in the Alg. 2). Here the local part

of vector z needs to be transferred to PPE, where the components z_k from all SPEs are summed up. Each SPE fetches the sum vector z from PPE and resume its calculation. Pseudo-code for RBF has been omitted due to space constraints.

Algorithm 4. k-Means SPE (with data overlay optimization)

Input: D = Dataset, k = the number of centers, D_p = part of Dataset for one SPU
Output: Each $d \in D$ is assigned to its closest center $c \in C$

1: GetContext(D_p, k); $totalData = |D_p|$
2: Calculate the available memory m for D_p
3: **while** $message \neq STOP$ **do**
4: wait for $START$ message from PPE
5: Fetch centers C into local store via DMA call(s)
6: Fetch $m/2$ data D_p^1 into local store via DMA call(s)
7: **while** $totalData > 0$ **do**
8: Verify all transfers are finished
9: (a) Fetch $m/2$ data D_p^2 into local store via DMA call(s);
(b) Concurrently calculate distances and assign center for D_p^1
10: Put the results of D_p^1 back into system memory
11: Verify all transfers are finished
12: (a) Fetch $m/2$ data D_p^1 into local store via DMA call(s);
(b) Concurrently calculate distances and assign center for D_p^2
13: Put the results of D_p^2 back into system memory
14: $totalData = totalData - m$
15: **end while**
16: send $COMPLETE$ message to PPE
17: **end while**

4.3 Apriori on the Cell

The optimization of Apriori is a big challenge due to the high complexity of its implementation and the limited local memory on each SPE. Our code is based on the code of Bodon which is known as one of the fastest realizations of Apriori [3]. First, this implementation uses a red-black-tree, and it is a difficult task to parallelize a tree-based algorithm. Second, the algorithm needs C++ STL which invokes some memory consuming functions, e.g. *new* and *delete*. We replaced all *new*/*delete* with $malloc()/free()$ in both apriori programs and STL implementations, which could help us to obtain additional 60 KByte memory in the local store.

We used the count distribution idea to parallelize Apriori [4]. The count distribution algorithm is a parallel version of Apriori that distributes the data set over all processors. All processors generate the entire candidate k-itemset from the set of frequent $(k-1)$-itemsets. Each SPE can thus independently calculate partial supports of candidates from its local data set partition. Next, the PPE performs the sum reduction to obtain the global support counts that are distributed to all SPEs. Compared to other parallel Apriori implementations, count distribution algorithm has minimum communication, because only count values are exchanged among SPEs. However, in order to exchange count values,

it requires that the candidate k-itemsets must be identical on all SPEs, and the entire red-black-tree must be replicated on each SPE.

5 Experimental Results

In this section we present a detailed evaluation of our parallel codes on the Cell chip and present a comparison to implementations on Xeon and Opteron CPUs.

We execute the programs on one Cell chip of a Cell cluster built from IBM BladeCenter QS20 dual-Cell blades; each blade houses two 3.2 GHz Cell BE processors and 1 GB XDRAM (512 MB per processor). With a clock speed of 3.2GHz, the Cell processor has a theoretical peak performance of 204.8 GFLOP/s (single precision). For comparison, we provide execution times for single-threaded reference implementations on the following two processors:

Xeon E5310: 1.6GHz, 4MB L2 cache, Intel C compiler 9.1

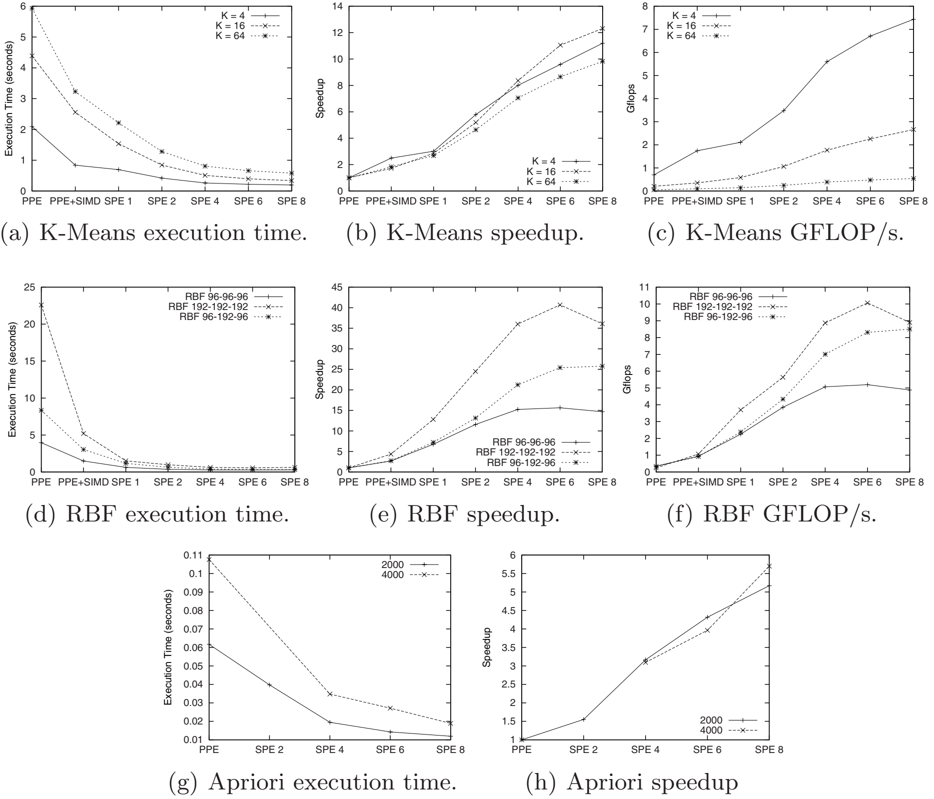
Opteron 880: 2.6GHz, 1MB L2 cache, Intel C compiler 9.0

Tab. 1 illustrates the performance advantage of the Cell executing k-Means, RBF and Apriori as compared to two commodity processors. The parameters for k-Means were DataPoints=6K, Dimensions=128, and centers=4. The parameters for RBF were InputNodes=128, RBFNodes=128, and OutputNodes=128. The parameters for Apriori were TransactionNo=2K with transactions from the well-known database T10I4D100K. The Cell outperforms the others a little when only PPE is used, but it scales very well with SPEs. The second best performance was afforded by the Opteron 880, and the Xeon E5310 gave the third.

Table 1. Performance comparison of selected algorithms for various processors

Processor	*k-Means (sec.)*	*RBF (sec.)*	*Apriori (sec.)*
Xeon	2.547792	7.571673	0.059752
Opteron	2.363729	6.456178	0.059523
Cell PPE	2.091699	5.201312	0.055320
Cell 8 SPE with vectorization	0.196709	0.302930	0.023957

In Fig. 3 we present the results of our implementation when using PPE and PPE with vectorization, and when using one or more SPEs with vectorization. All performance data are given for small data sets, but because of the double buffer scheme the performance for large data sets is similar. In our implementation only the centroid data (k-Means), the neural weights (RBF) and the candidate tree (Apriori) must fit into the local stores of the SPE. As shown in the Figs. 3(a) and 3(d), k-Means and RBF hugely benefit from eight SPE. However, Fig. 3(g) shows that Apriori does not benefit a lot from SPE parallelization, since vectorization cannot be used and the 256KB local store limits the problem size. Figs. 3(b) and 3(e) present the speedup of k-Means and RBF which is given against the PPE code. RBF is the most computing intensive application, thus RBF achieves a high speedup of 40.67 when using 6 SPEs and computing



(a) K-Means execution time. (b) K-Means speedup. (c) K-Means GFLOP/s.

(d) RBF execution time. (e) RBF speedup. (f) RBF GFLOP/s.

(g) Apriori execution time. (h) Apriori speedup

Fig. 3. Experimental results

the largest RBF network 192-192-192. Finally we calculated the GFLOP/s of k-Means and RBF using the total number of arithmetic operations divided by the measured execution time. K-Means attains a single precision performance of 6.8 GFLOP/s, and RBF 10 GFLOP/s.

6 Related Work

The large potential of the Cell architecture for scientific computations is discussed in [5,6]. There are some additional papers in which the performance of data mining algorithms [7] and other computation intensive algorithms [8,9] on the Cell Broadband Engine is investigated. In [7], only the performance of three rather similar algorithms (k-Means, k-Nearest Neighbors, ORCA) is analyzed that are based on the calculation of Euclidean distances. This makes their optimization quite similar and shows less generality. By parallelizing three important classes of data mining algorithms, we proved that data mining algorithms can achieve good performance on the Cell. Additionally we used data overlay to

overcome physical limitations, thus enabling larger problem sizes on the Cell. Moreover, no other studies have investigated how effectively the Cell BE can be employed to perform association rules finding and neural network training, and how it compares against the data mining implementations on other architectures in terms of performance. The problem of finding association rules poses difficult challenges. During the parallelization of Apriori, we encountered some problems that were not met by others before, and proposed effective solutions.

7 Conclusion

We have demonstrated that efficient implementations of data mining algorithms are possible on the Cell architecture. Applications with small local memory demand can be implemented without many changes to the code, for example, k-Means and RBF. Nevertheless they can operate on large data sets by applying the data overlay technique. Operations on vectors and matrices can efficiently be vectorized on the SPEs. The implementation of tree-based algorithms like Apriori remains a big challenge because partitioning the tree is difficult. Here we analyzed a parallel version of Apriori in which only the data set is distributed over the SPEs whereas the tree is replicated in each node.

The experimental evaluations have shown that the Cell BE can achieve impressive performance for data mining algorithms. Compared to other processors, a speedup of one order of magnitude is possible. The keys to achieve high performance are a clear understanding of characteristics of the algorithms and, more importantly, a clear understanding of the system and environment limitations. Other data mining algorithms can be implemented on the Cell in a similar way. This will be a topic of our future work.

Acknowledgments

The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

References

1. Kahle, J., et al.: Introduction to the Cell multiprocessor. IBM Journal of Research and Development 49(4), 589–604 (2005)
2. IBM Corporation. Cell Broad Band Engine technology, `http://www.alphaworks.ibm.com/topics/cell`
3. Bodon, F.: A fast apriori implementation. In: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI 2003) (2003)
4. Zaki, M.: Parallel and Distributed Association Mining: A Survey. IEEE Concurrency 7(4), 14–25 (1999)
5. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the cell processor for scientific computing. In: Proceedings of the 3rd conference on Computing Frontiers (CF 2006), pp. 9–20 (2006)

6. Bader, D., Agarwal, V., Madduriet, K.: On the Design and Analysis of Irregular Algorithms on the Cell Processor: A case study on list ranking. In: 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2007)
7. Buehrer, G., Parthasarathy, S.: The Potential of the Cell Broadband Engine for Data Mining. In: Proceedings of the 33rd Int. Conference on Very Large Data Bases (VLDB) (2007)
8. Bader, D., Agarwal, V.: FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 172–184. Springer, Heidelberg (2007)
9. Petrini, F., et al.: Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In: 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2007)