The Potential of the Cell Broadband Engine for Data Mining

Gregory Buehrer
The Ohio State University
Columbus, OH, USA
buehrer@cse.ohio-state.edu

Srinivasan Parthasarathy*
The Ohio State University
Columbus, OH, USA
srini@cse.ohio-state.edu

ABSTRACT

In this article we examine the performance of key data mining kernels on the STI Cell Broadband Engine architecture. This architecture represents an interesting design point along the spectrum of chipsets with multiple processing elements. The STI Cell has one main processor and eight support processing elements (SPEs), and while it represents a more general purpose architecture when compared to graphics processor units, memory management is explicit. Thus while it is easier to program than GPUs it is not as easy to program as current day dual and quad-core processors designed by AMD and Intel.

We investigate the performance of three key kernels, namely clustering, classification, and outlier detection on the STI Cell along the axes of performance, programming complexity and algorithm designs. Specifically, we formulate SIMD algorithms for these workloads and evaluate them in detail to determine both the benefits of Cell processor, as well as its inherent bottlenecks. As part of our comparative analysis we juxtapose these algorithms with similar ones implemented on modern architectures including the Itanium, AMD Opteron and Pentium architectures. For the workloads we consider, the Cell processor is up to 34 times more efficient than competing technologies. An important outcome of the study, beyond the results on these particular algorithms, is that we answer several higher level questions designed explicitly to provide a fast and reliable estimate for how well other data mining workloads will scale on the Cell processor.

1. INTRODUCTION

Every so often, humankind makes a leap in its ability to collect and store knowledge. From the carvings on pottery in 3500 BC, to Chinese paper in 100 AD, we have found that maintaining knowledge aids in the improvement and advancement of civilization. When this knowledge is lost, society is set back considerably, at times for thousands of years. Take for example, the Roman Empire. Their engineering efforts produced such improvements as hydraulic cement and the grain reaper. During the Dark Ages, both inventions were lost. Farmers used a simple blade for nearly 2000 years, until the reaper was reinvented by an Irish-American inventor named Cyrus McCormick in 1834.

In efforts to avoid losing knowledge, and also to gain a competitve edge, organizations collect and store large volumes of data. In fact, even a simple home PC may contain 500GB or more of data. In an effort to harness this information, organizations have turned to data mining. Data mining is the process of converting vast amounts of this information into insight or knowledge in a semi-automated fashion. A fundamental challenge is that the cost of extracting this information often grows exponentially with the size of the data. As data mining is an interactive process, short response times for querying and processing data sets is crucial.

Researchers address long execution times along two avenues. First, the amount of computation can be pruned via cleverly short circuiting the search space, or via intelligent indices and other data structures. Second, algorithm designers restructure and tune computations to improve the utilization of the underlying hardware. As hardware designers adapt to the ever-challenging workloads present in modern computing, maintaining a high utilization of the hardware is quite difficult. For example, recent projects by the database community have leveraged the general purpose nature of newer graphics cards for the TeraSort[10] project.

One such recent advancement in microprocessor design is chip multiprocessing (CMP). CMP designs exhibit multiple processing cores on one chip. CMPs arise in part because of the inherent challenges with increasing clock frequencies. The increase in processor frequencies over the past several years has required a significant increase in voltage, which has increased power consumption and heat dissipation. In addition, increased frequencies require considerable extensions to instruction pipeline depths. Finally, since memory latency has not decreased proportionally to the increase in clock frequency, higher frequencies are often not beneficial due to poor memory performance. By incorporating thread level parallelism, chip vendors can continue to improve IPC by exploiting parallelism without raising frequencies. As these low cost parallel chips become mainstream, designing data mining algorithms to leverage them becomes an important task. Current dual core chips include Intel's Pentium D, AMD's Opteron, and IBM's Power4. A joint venture

*This work is supported in part by NSF grants #CAREER-IIS-0347662, #RI-CNS-0403342, and #NGS-CNS-0406386.

VLDB '07, September 23-28, 2007, Vienna, Austria.

by Sony, Toshiba and IBM (STI) has produced a nine core architecture called the Cell BDEA.

As a result of this advancement, parallel algorithm designs will become increasingly important, even for mainstream commodity applications, in order to realize performance that is commensurate with such emerging processors. The spectrum of emerging chipsets in this arena span different points on the design spectrum, ranging from graphics processor units on one end to commercial general purpose POSIX-style multicore CPUs from Intel, SUN and AMD. The Cell chip is of particular interest because of its high number of cores, its 200+ GFLOPs of compute power, and its 25GB/s off chip bandwidth. All three values represent breakthroughs in commodity processing. Cell is expected to be used in high end super computing systems[1] as kernel accelerators.

The layout of the Cell chip lies somewhere between other modern CMP chips and a high end GPU, since in some views the eight SPUs mimic pixel shader units. Unlike GPUs, however, the Cell can chain its processors in any order, or have them operate independently. Target applications include medical imaging, physical model simulation, and consumer HDTV equipment. While the Cell chip is quite new, several workloads seem quite amenable to its architecture. For example, high floating point workloads with streaming access patterns are of particular interest. These workloads could leverage the large floating point throughput. In addition, because their access pattern is known *a priori*, they can use software-managed caches for good bandwidth utilization.

This work seeks to map several important data mining tasks onto the Cell, namely clustering, classification and outlier detection. These are three fundamental data mining tasks, often used independently or as precursors to solve multi-step data mining challenges. In addition, all three tasks have efficient solutions which leverage distance computations. Specifically, we seek to make the following contributions to the community. Our first goal is to pinpoint the bottlenecks in scalability and performance when mapping these workloads to this platform. We believe future streaming architectures could benefit from this study as well. We port these three tasks to the Cell, and present the reader with a detailed study regarding their performance. More importantly, our second goal is to answer the following higher level questions for data mining applications.

- *Can these applications leverage the Cell to efficiently process large data sets? Specifically, does the small local store prohibit mining large data sets?*
- *Will channel transfer time (bandwidth) limit scalability? If not, what is the greatest bottleneck?*
- *Which data mining workloads can leverage SIMD instructions for significant performance gains?*
- *What metrics can a programmer use to quickly gage whether an algorithm is amenable to the Cell?*
- *At what cost to programming development are these gains afforded?*

The outline of this paper is as follows. Related work is presented in Section 2. A description of the Cell architecture is given in Section 3. A background on the workloads in question is presented in Section 4. In Section 5, we present our Cell formulations of these workloads. We empirically evaluate these approaches, and discuss our findings in Sections 6 and 7. Finally, concluding remarks are presented in Section 8.

[1]The IBM/DOE *RoadRunner*

2. RELATED WORK

Several researchers have investigated improving the efficiency of *kMeans* clustering [18]. Alsabti, Ranka and Singh[1] use geometry trees to reduce runtimes by lowering the number of distance calculations required. Pelleg and Moore[19] also employed a kd-tree in a similar fashion to improve the *kMeans* clustering algorithm. These techniques define regions in n-dimensional space to cluster points. Then, all the points in a space can be assigned to a particular center. Subsequent assignment calculations essentially need only to verify the point lies within the bounding box. However, Weber and Zezula found that bounding trees do not scale well with increasing dimensions [22], failing completely with as little as 16 dimensions. They show that simple scans of the data set greatly outperform geometric meta structures such as bounding trees. Elkan [6] leverages a knowledge cache from previous iterations to lower execution times.

Jin and Agrawal have several works targeted at solving parallel data mining workloads [13, 14] on both shared memory and distributed systems. They implement a framework called *FREERIDE* for fast prototyping of data mining workloads on shared memory systems. A focus of the work is on locking cost reduction, which does not appear to be the bottleneck for the platforms targeted in this work.

k Nearest Neighbors [11] is used in many domains, such as biology [12], chemistry, finance, and is the basis for many machine learning techniques. Many researchers have investigated efficiency improvements for kNN, we mention several of the most relevent [3, 20]. Wang and Wang [21] develop a multi-level approximation scheme to query for nearest neighbors in high dimensional data at remote sites. Liao, Lopez and Leutenegger [17] redistribute data points in a B-tree to improve execution times for nearest neighbor queries for high-dimensional data, but their results are approximate. Kulkarni and Orlandic [15] use clustering to aggregate points into regions, thus allowing the algorithm to prune unnecessary distance calculations. The method maintains exact neighbors. Zaki, Ho and Agrawal [25] parallelized decision tree construction for classification. It is not clear that their approach is readily portable to the Cell, since the construction process uses significant main memory for the meta structures, and the SPUs have limited memory.

Mining for outliers in high dimensional data has been of recent interest. *ORCA* [2] uses a threshold to prune distance calculations, and will be presented in Section 4. Chaudhary, Szalay, and Moore [4] use kd-trees to improve execution times when searching for outliers. Their approach is similar to those employed by Alsabti, Ranka and Singh when clustering with kMeans. Angiulli and Pizzuti define a metric called *weight* to aid on finding outliers. They term *weight* as the sum of the distances to the top K nearest neighbors. This metric could be used in our algorithms without a significant modification.

Kunzman, *et al* [16] adapted their parallel runtime framework *CHARM++*, a parallel object-oriented C++ library, to provide portability between the Cell and other platforms. In particular, they proposed *the Offload API*, a general-purpose API to prefetch data, encapsulate data, and peek into the work queue.

Algorithm 1 kMeans

Input: Dataset D
Input: k, the number of centers
Output: Each $d \in D \leftarrow$ closest center $c \in C$

```
 1: while true do
 2:    changed=0
 3:    for each data point d_i ∈ D do
 4:       assignedCenter = d_i.center
 5:       for each center c_j ∈ C do
 6:          d = dist(d_i,c_j)
 7:          if d < d_i.Center then
 8:             d_i.centerDistance = d
 9:             d_i.center = j
10:          end if
11:       end for
12:       if d_i.center <> assignedCenter then
13:          changed++
14:       end if
15:    end for
16:    for each center c_j ∈ C do
17:       c_j = Mean of points i where c_i=j
18:    end for
19:    if changed==0 then
20:       break
21:    end if
22: end while
```

Several sorting algorithms leverage SIMD programming, and are relevant; we mention several here. Govindaraju *et al* developed an efficient SIMD sorting network for GPUs called TeraSort. It leverages the rasterization engine to execute bitonic sort. Zagha and Blelloch [24] developed a data-parallel SIMD version of Radix sort for the Y-MP. Furtak, Amaral and Niewiadomski [8] describe several methods to improve the performance of sorting networks using SIMD instructions. They describe a two-pass approach, which first approximately sorts using SIMD registers, and then uses traditional sorting such as merge sort to complete the process. Among their results, they show that the branch reductions afforded by a competing one pass vector sort improves overall execution times significantly.

We are not aware of existing work which attempts to marry the Cell BDEA with data mining. Williams [23] *et al* investigate the performance of the Cell for scientific workloads. They find that it provides a many-fold reduction in execution times when compared to other processors. In particular, they show speedups for GEMM, Fast Fourier Transform, and Stencil computations. They also suggest a data path modification to improve the computational throughput on double-precision workloads.

3. THE CELL BROADBAND ENGINE

The *Cell Broadband Engine Architecture* [5] (Cell) was designed over a four year period primarily for the PlayStation 3 gaming console. The chip is also available in commercial blade servers through Mercury Computer Systems. Highly tauted, it was the winner of the Microprocessor Best Technology Award in 2004[2]. It is surmised by the high performance community that the Cell's commercial uses will allow it to be produced in sufficient quantities so as to support a low price tag. This thought, in conjunction with the Cell's significant floating point computational throughput and high off chip bandwidth, have led to discussion regarding utilizing the chip in large scale clusters.

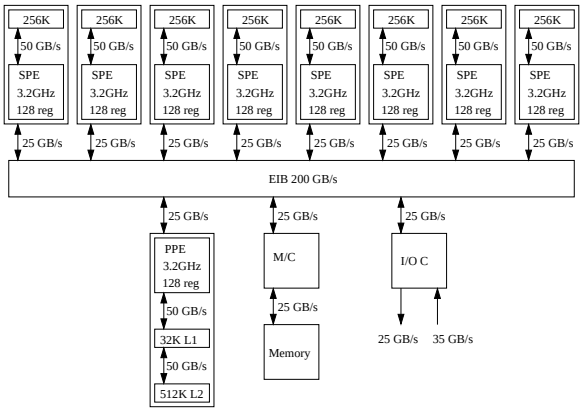

Figure 1: The layout and bandwidth of the Cell processor.

The architecture features one general-purpose processing element called the Power Processing Element (PPE) and eight support processors called Synergistic Processing Elements (SPEs). It is depicted in Figure 1. The PPE is a two-way SMT multithreaded Power 970 architecture compliant core, while all SPEs are single threaded. All processors are interconnected with a high-bandwidth ring network called the EIB. The Cell's design is one that favors bandwidth over latency, as the memory model does not include a hierarchical cache. In addition, it favors performance over programming simplicity. The memory model is software controlled. For example, all memory accesses must be performed by the programmer through DMA transfers calls, and the local cache at each SPE is managed explicitly by the programmer. Although this imparts a complexity on the programmer, it also affords the potential for very efficient bandwidth use, since each byte transferred is specifically requested by user software. There is no separate instruction cache; the program shares the local store with the data. Also, software memory management lowers required on-chip hardware requirements, thus lowering power consumption. At 3.2 GHz, an SPE uses only 4 watts per core; as a comparison, a 1.4GHz Itanium consumes about 130 watts.

Each SPE [7] contains an SPU and an SPF. The SPF consists of a DMA (direct memory access) controller, and an MMU (memory management unit) to interact with the common interconnect bus (EIB). Bandwidth from an SPE to the EIB is about 25GB/s, both upstream and downstream (see Figure 1). SPEs are SIMD, capable of operating on eight 16 bit operands in one instruction (or four 32 bit operands). The floating point unit supports multiplication and subsequent accumulation in one instruction, (*spu_madd()*), which

[2]http://www.power.org/news/besttechaward.pdf

can be issued every cycle (with a latency of 6 cycles). This equates to about 25 GFLOPs per SPE, or 200 GFLOPs for eight SPEs, a number far greater than competing commodity processors. Each SPU has a low-latency mailbox channel to the PPU which can send one word (32 bits) at a time and receive four words. SPEs have no branch predictors. All branching is predicted statically at compile time, and a misprediction requires about 20 cycles to load the new instruction stream. Finally, the SPE supports in-order instruction issue only. Thus, the programmer must provide sufficient instruction level parallelism so that compile-time techniques can order instructions in a manner which minimizes pipeline stalls.

4. DATA MINING WORKLOADS

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

4.1 Clustering

Clustering is a process by which data points are grouped together based on similarity. Objects assigned to the same group should have high similarity, and objects between groups should have low similarity. Clustering has many practical applications, such as species grouping in biology, grouping documents on the web, grouping commodities in finance and grouping molecules in chemistry. It is considered an unsupervised learning algorithm, since user input is minimal and classes or group labels need not be determined *a priori*. There are many different mechanisms by which objects of a data set can be clustered, such as distance-based clustering, divisive clustering, agglomerative clustering, and probabilistic clustering. *kMeans* [18] is a widely popular distance-based clustering algorithm, and is the chosen algorithm for our study.

As its name implies, the *kMeans* algorithm uses the average of all the points assigned to a center to represent that center. The algorithm proceeds as follows. First, each data object is assigned to one of the k centers at random. In the second step, the centers are calculated by averaging the points assigned to them. Third, each point is checked against each center, to verify the point is assigned to the center closest to it. If any point required reassignment, the algorithm loops back to step two. The algorithm terminates when a scan of the data set yields no reassignments. A sketch is presented as Algorithm 1.

4.2 Classification

Classification is a common data mining task, whereby the label or class of a data object is predicted. For example, a classification algorithm may be used to predict whether a loan applicant should be *approved* or *rejected*. Classification is said to be *supervised* since classes are known and a training data set is provided, where each object in the set has been labeled with the appropriate class. There are many methods to predict labels. Examples include Bayesian networks, neural networks, nearest neighbors algorithms, and decision tree algorithms. Algorithm designers are faced with several challenges when designing these solutions, including noise reduction, the curse of dimensionality, and scalability with increasing data set size.

One of the most widely used classifiers is the *k Nearest Neighbors* algorithm (kNN). kNN is said to work by analogy. A data object is classified by the most represented label of its k closest neighbors. In the worst case, the algorithm requires a distance calculation between each two data points. The method is considered *lazy* because a model is not built *a priori*; instead the training data is inspected only when a point is classified. In addition to avoiding model construction, kNN requires essentially no parameters and scales with data dimensionality. The data set is typically a collection of n-dimensional points, and the similarity measure is Euclidean distance. A sketch of the algorithm is presented as Algorithm 2.

4.3 Outlier Detection

Automatically detecting outliers in large data sets is a data mining task which has received significant attention in recent years [2, 4, 9]. Outlier detection can be used to find network intrusions, system alarm conditions, physical simulation defects, noise in data, and many other anomalies. The premise is that most data fit well to a model, save a few points. These few points are then classified as outliers. As with classification, there are two common techniques, namely model-based approaches and distance-based methods. Model approaches build a model of the data, and then output data which does not fit the model. Distance-based approaches define a distance calculation, and label points without nearby neighbors as outliers. Also like classification, distance-based detections schemes are well received because model construction is avoided, which is often a bottleneck with high-dimensional data.

Algorithm 2 kNearestNeighbors

Input: Dataset D
Input: k, number of neighbors
Output: $\forall d_i \in D, d_i.neighbors =$ closest k points to d_i

```
1: for each data point d_i ∈ D do
2:    d_i.neighbors = ∅
3:    for each data point d_j ∈ D where d_i <> d_j do
4:       dis = dist(d_i,d_j)
5:       if |d_i.neighbors| < k then
6:          d_i.neighbors = d_i.neighbors = ∪d_j
7:       else
8:          if max(d_i.neighbors) > dis then
9:             Remove farthest point in d_i.neighbors
10:            d_i.neighbors = d_i.neighbors = ∪d_j
11:         end if
12:      end if
13:      d_i.class = class most seen in d_i.neighbors
14:   end for
15: end for
```

ORCA[2] is an efficient distance-based outlier detection algorithm developed by Bay and Schwabacher. It uses the nearest k neighbors as a basis for determining an outlier. The insight provided by ORCA is that once a point has k neighbors which are closer to it than the kth nearest neighbor of the weakest outlier, the point cannot be an outlier. Therefore, processing of the data point is terminated. To illustrate, consider the outliers in Table 1. This data represents the top 5 outliers, with the number of neighbors $k = 4$. Thus, an outlier is determined by the distance to his 4th neighbor. The weakest outlier is the outlier with the smallest 4th neighbor, in this case outlier 5. The threshold is then 255.1, since if any data point has four neighbors closer than 255.1, that point cannot be an outlier. Often times, k near

Neighbors →	1st	2nd	3rd	4th
Outlier 1	147.6	151.2	179.1	655.1
Outlier 2	342.2	387.5	409.9	458.2
Outlier 3	100.0	131.4	219.1	325.1
Outlier 4	87.2	89.8	107.3	210.0
Outlier 5	31.0	151.2	179.1	255.1

Table 1: An example set of outliers, where outlier 5 is the weakest.

neighbors can be found by scanning just a small percentage of the data set. A sketch of the algorithm is provided as Algorithm 3.

For all three workloads we implement the Euclidean distance as our similarity metric, which is a special case (p=2) of the Minkowski metric (given below).

$$D_2(x_i, x_j) = (\Sigma_{k=1}^{d}(x_{i,k} - x_{j,k})^2)^{1/2} \quad (1)$$

In practice, since the square root function maintains the total order on positive reals, most implementations do not take the square root of the distance. Based on our findings, we believe any distance calculation which touches every byte loaded will have similar results as those presented in Section 6.

Algorithm 3 ORCA

Input: Dataset D
Input: n, number of outliers
Input: k, number of neighbors
Output: O = top n outliers

```
1: O = ∅
2: Threshold = 0
3: for each data point d_i ∈ D do
4:    d_i.Neighbors = ∅
5:    for each data point d_j ∈ D where d_i <> d_j do
6:       d = dist(d_i,d_j);
7:       if d < max(O) or |d_i.Neighbors| < k then
8:          d_i.neighbors = d_i.neighbors ∪ d_j
9:       end if
10:      if |d_i.Neighbors| = k and max(d_i.Neighbors) >
         Threshold then
11:         break;
12:      end if
13:      if |O| < n then
14:         O = O ∪ d_i
15:      else
16:         if minDistance(O) then
17:            Remove weakest outlier from O
18:            O = O ∪ d_i
19:         end if
20:      end if
21:   end for
22:   Threshold = kth value from weakest(O).neighbor
23: end for
```

5. ALGORITHMS

Developing algorithms for the proposed workloads on the Cell requires three components. First we must parallelize the workload. This is direct, as at least two of the three workloads are members of the *embarrassingly parallel* class of data mining algorithms. Second, we require an efficient data transfer mechanism to move data from from main memory to the local store. Third, we must restructure the algorithm to leverage the Single Instruction Multiple Data (SIMD) intrinsics available. In this section, we detail these components.

5.1 KMeans on the Cell

Parallelization of *kMeans* is straightforward. We partition the data set (which resides in main memory) such that two conditions hold. First, the number of records assigned to each processor is as balanced as possible. Second, the start boundary of the each processor's segment is aligned on a 16 byte boundary [3]. This can be achieved by placing the first record on a 16 byte boundary, and then verifying that the number of records assigned to a processor satisfies the constraint below.

$$records * dim * sizeof(float) \ \% \ 16 == 0 \quad (2)$$

We simply assign the processor an even share of records, and add a record until it is properly aligned. If after adding a user-defined threshold of additional records, it is still not 16 byte aligned, then we pad it with the necessary bytes.

Efficient data transfer for *kMeans* is achieved by calculating a chunk size which results in landing on a record boundary, is a multiple of 16, and is about 6KB. At values below 6KB, the startup cost to retrieve the first byte is not sufficiently amortized. The maximum DMA call permitted by the Cell is 16KB, but we found smaller values afforded better load balancing opportunities. This chunk size can be calculated by simple doubling as shown in Algorithm 4.

Algorithm 4 SPU GetChunksize

Input: M, the number of dimensions
Output: chunkSize is properly aligned

```
1: int chunkSize=sizeof(float)*M
2: int recordsToGet=1
3: while chunkSize < 4096 do
4:    chunkSize*=2;
5:    recordsToGet*=2;
6: end while
```

Restructuring *kMeans* to allow for SIMD distance calculations can be achieved by a) calculating the distance to multiple centers at once, b) calculating the distance between a center and multiple data points at once, and c) calculating multiple dimensions at once. We make use of two intrinsics, namely $v3 = spu\_sub(v1, v2)$ and $v4 = spu\_madd(v1, v2, v3)$. The former subtracts each element of vector $v1$ from $v2$ and stores the result in $v3$. The latter multiples the elements of $v1$ to $v2$, adds the result to $v3$ and stores it in $v4$. This second instruction effectively executes 8 floating point operations in a single instruction, with a 6 cycle latency. The latency can be avoided by calculating multiple points.

The strategy for calculating distances is shown in Algorithm 5. It is clear that the number of distance calculations in the inner loop can be expanded to improve throughput by further unrolling until each center in the chunk size is accommodated. In the case that the number of centers or dimensions is not modulo the largest desired block, a simple iterative halving flow of control is used to finish the calculation.

[3]The Cell's DMA controller requires 16-byte boundaries.

Note that these will have branching, which incurs a 20 cycle penalty. Fortunately, the intrinsic *_builtin_expect*$(cond, 0)$ can be employed to avoid penalty in the common case. Note that the function *spu_extract*(v, pos) extracts a scaler from vector v at position pos. The $if/then$ constructs after the looping are replaced by *spu_sel*() by the compiler, which removes simple branching. The SPUs send the number of reassigned data points back to the PPU through mailboxes. If any SPU reassigned a data point, the centers are recalculated and the SPUs are sent a message to perform another iteration; otherwise the SPUs are sent a message to terminate. The pseudo code for the $kMeans$ is shown in Algorithms 6 and 7.

An important issue when using the Cell is that any meta data which grows with the size of the data set cannot be stored locally. In the case of $kMeans$, the center assignment are an example of this type of data. The solution is to preallocate storage with each record when the data is read from disk. When each record is loaded from main memory to the SPU, the meta data is loaded as well, and when the record is purged from the SPU, the meta data is written to main memory with the record. This allows the algorithm to scale to large data sets.

Algorithm 5 AssignCenter

Input: Data *record* with M dimensions
Input: Centers C
Output: *record.center* $\leftarrow$ closest center $c \in C$

```
 1: vector v1 = (vector float*)record
 2: for i=0 to |C| step 2 do
 3:    vector v2=(vector float*)Center[i]
 4:    vector v3=(vector float*)Center[i+1]
 5:    vector float total,total2=0,0,0,0
 6:    for j = 0 to M/4 do
 7:       vector float res= spu_sub(v1[j],v2[j])
 8:       vector float res2= spu_sub(v1[j],v2[j])
 9:       total = spu_madd(res,res,total)
10:       total2 = spu_madd(res2,res2,total2)
11:    end for
12:    if _builtin_expect(M % 4<>0,0) then
13:       int k=j
14:       for j = 0 to M % 4 do
15:          float val1=(record[k*4+j]-center[i][k*4+j])
16:          total +=val1*val1
17:          float val2=(record[k*4+j]-center[i+1][k*4+j])
18:          total2 +=val2*val2
19:       end for
20:    end if
21:    float distance = spu_extract(total,0) + ... (total,4)
22:    float distance2 = spu_extract(total2,0) + ... (total2,4)
23:    if distance < record.centerDistance then
24:       record.center=i
25:       record.centerDistance=distance
26:    end if
27:    if distance2 < record.centerDistance then
28:       record.center=i+1
29:       record.centerDistance=distance2
30:    end if
31: end for
```

Algorithm 6 kMeans PPU

Input: Dataset D
Input: P, the number of processors
Input: k, the number of centers
Output: Each $d \in D \leftarrow$ closest center $c \in C$

```
 1: Assign each d ∈ D a random center
 2: Partition D among P SPUs
 3: Spawn P SPU Threads
 4: while true do
 5:    int changed=0
 6:    for each processor p do
 7:       changed += p.mailbox
 8:    end for
 9:    for each center c_j ∈ C do
10:       c_j = Mean of points i where c_i = j
11:    end for
12:    if changed==0 then
13:       ∀p ∈ P, p.mailbox ← 0
14:       break;
15:    else
16:       ∀p ∈ P, p.mailbox ← 1
17:    end if
18: end while
```

Algorithm 7 KMeans SPU

Input: Dataset D, Address A
Input: M,the number of dimensions
Input: k, the number of centers
Output: Each $d \in D \leftarrow$ closest center $c \in C$

```
 1: GetChunksize(D_p, I, K)
 2: message=1
 3: totalData = |D|
 4: while message==1 do
 5:    Load centers C into local store via DMA call(s)
 6:    while totalData > 0 do
 7:       Load data D_a into local store via DMA call
 8:       totalData = totalData - recordsToGet
 9:       for each data point d_j ∈ D_a do
10:          assignedCenter = d_i.Center
11:          AssignCenter(d_j,C)
12:          if d_i.Center <> assignedCenter then
13:             Changed++;
14:          end if
15:       end for
16:    end while
17:    p.mailBox ← changed
18:    message ← p.mailbox
19: end while
```

5.2 kNN on the Cell

The main difference in construction between $kMeans$ and kNN is that with $kMeans$ two streams are required[4]. The first stream is the test data set (the data to be labeled) and the second stream is the training data set (the prelabeled data). The same chunk size is used for both streams, and is calculated with Algorithm 4. However, the record size is the dimensionality of the data plus k, where k is the number of neighbors to store. This allocation allows the SPU to store

[4]If the centers in *kMeans* do not fit in the local store, then both algorithms use two streams.

the IDs of the neighbors with the record, and limit local meta data. This can be doubled if the user requires the actual distances as well; otherwise only one array of size k is kept on the local store to maintain this information and is cleared after each data point completes. This is a fundamental point when data mining on the Cell, which is to say that meta data must be stored with the record, to allow the Cell's SPUs to process large data. Synchronization only occurs at the completion of the algorithm. Pseudo code for kNN has been omitted due to space constraints.

5.3 ORCA on the Cell

The $ORCA$ construction is also similar to that of $kMeans$. $ORCA$ presents an additional challenge, however, because the effectiveness of computation pruning is a function of the threshold value. Without effective pruning, the algorithm grows in average case complexity from O($nlgn$) to O(n^2).)As the threshold increases, more pruning occurs. Partitioning the data set evenly may result in an uneven outlier distribution among the SPUs, thus the computation time per SPU becomes unbalanced. We can correct this by sharing local outliers between SPUs periodically. The strategy is to synchronize often early in the computation, and less frequently later in the computation. In the early stages, each data point has a higher probability to increase the threshold, since the set of outliers is incomplete. Recall that the threshold is the neighbor in the weakest outlier with the greatest distance. With all the SPUs maintaining separate outlier tables, their thresholds will vary. In most cases the thresholds will all be different, with the largest threshold being the best pruner. However, if all the SPUs share their data, the new threshold is most likely larger than any single SPU's current threshold. This is because the top five outliers from all the sets of outliers (one from each SPU) are the true outliers. Therefore, frequent synchronization early in the computation will support sifting these outliers to the top.

Partitioning the data set proceeds as it did for the previous two workloads. However, the chunk size initially is set at the first record size satisfying Equation 2 greater than 512 bytes. Each successive data movement is increased, until a chunk size of about 4K is reached, which is optimal. As we will see in Section 6, chunk sizes larger than 4K result in a greater number of distance calculations.

At each synchronization point, each SPU writes its outliers to main memory. The synchronization is initiated by each SPU writing 1 to its mailbox to the PPU. When all SPUs have written to their mailboxes, the PPU then takes the top n outliers from these eight sets and copies them back to main memory. When the SPUs start the next chunk, they also load the new outlier list, and with it the maximum threshold. When an SPU is finished with its portion of the data set, it writes a 0 to its mailbox. The algorithm terminates when all SPUs have written 0 to their mailboxes.

6. EVALUATION

In this section we present a detailed evaluation of the proposed workloads and their optimizations on the Cell processor.

6.1 Experimental Setup

We execute the programs on a Playstation3 gaming console with Fedora Core 5 (PPC) installed. The PS3 provides the programmer with only six SPUs, as one is unavailable (rumored to be for improved yield) and another is dedicated to the game console's security features. It houses 256MB of main memory, of which about 175MB is available. Performance-level simulation data, such as cycle counts, was provided by the IBM Full System Simulator (Mambo), available in the IBM Cell SDK [5]. Data was synthetically generated (32 bit floats). This allowed us to vary the number of data points, number of training points, dimensionality, and the number of outliers. Results on representative real data sets for the target applications are very similar to the corresponding synthetic data sets in our study.

As a comparison, we provide execution times for other processors as well, as shown in Table 2. Therefore, a few notes on these implementations. First, the only other multithreaded implementation is that for Intel's PentiumD processor, which has two processing cores. All other implementations were on single chip processors and use only one thread. Compiler flags had a large impact on performance, which is a topic in its own right. These implementations were compiled with a variety of different flags, and the best performing binaries are reported. For example, the Itanium performed best with *icc -$fast$*, and for the Xeon processor, the best performance was found with *icc -xW which vectorized the code*. In interesting cases, we provide two runtimes, one for Intel's compiler (icc) and another for the public gcc compiler (at least -O3 flag).

We also experimented with providing the PPU with work, which in principle is comparable to adding another SPU. Speedups were the same as adding an additional SPU.

The columns of Tables 4, 7 and 8 are as follows. The first column lists the trial number. The next four columns (five for kNN and $ORCA$) are input parameters, as shown in the headings. Each data point is an array of 32 bit floats. Columns 6-11 are the cycle statistics of the SPUs as a results of executing the program on the IBM simulator. Column 6 displays the Cycles required Per Instruction (CPI). Column 7 shows the percentage of the time that a single instruction is issued. Recall that the Cell SPU has two pipelines. Column 8 shows the percentage of the cycles that an instruction is issued on both pipelines. If this column were 100%, then all other columns would be 0% and the effective CPI would be 0.5, which is optimal. Columns 9-11 display the reason that there is not 100% double issue. Thus, columns 7-11 should sum to 100%. Branch stalls are due to branch mispredictions. Dependency stalls are due to a variety of reasons, for these workloads the common case is to stall on FP6, the floating point unit. This typically suggests that an instruction is waiting on the result of the previous instruction. Another common case is to stall waiting on a load instruction, which requires six cycles and moves the data from the local store to a register. Channel stalls are cycles lost waiting on DMA calls to load data chunks to the local store. Finally, the last column represents real execution time on the PS3.

6.2 Instruction Mix

The instruction mixes for each workload are presented in Table 3. From our description of these workloads, it is clear that the distance calculation dominates execution times, which is expected. Recall that our data sets are 32-bit floats, and the Cell executes on 128-bit registers. For many

[5]http://www-128.ibm.com/developerworks/power/cell/

Processor	Watts	MHz	Threads	Compiler
Itanium 2 (g)	130	1400	1	gcc
Itanium 2 (i)	130	1400	1	icc
Xeon (g)	110	2400	1	gcc
Xeon (i)	110	2400	1	icc
Opteron 250	89	2400	1	gcc
Pentium D	95	2800	1	icc
Pentium D 2	95	2800	2	icc
Cell SPU	4	3200	1	IBM SDK 2
Cell 6 SPU	24	3200	6	IBM SDK 2
Cell 8 SPU (sim)	32	3200	8	IBM SDK 2

Table 2: Processors used for the evaluation.

	kMeans	kNN	ORCA
FP	35%	34%	31%
ALU	17%	13%	23%
SHIFT	24%	26%	17%
LD/ST	10%	11%	13%
LOGICAL	11%	9%	6%
BRANCH	3%	5%	9%

Table 3: Instruction mixes for the Cell processor implementations.

floating point operations, this equates to 4 flops per instruction. However, about 30% of our operations are *spu_madd*() instructions, which multiply and add four 32-bit values in a single instruction. Therefore, although only 35% of the instructions are floating point, in actuality this is closer to 65% of the effective operations in a non-vectorized implementation.

ORCA has the largest number of branch instructions at 9%. This is primarily because the threshold may eliminate the need for a distance calculation for a given point, and force the loop to terminate prematurely. Both *ORCA* and *kNN* have more branching than *kMeans* because the nearest neighbors are stored in a sorted array, which inherently adds branching. All three workloads have a significant amount of loads and stores, which are required to bring the data from the local store to a register. Load and store instructions have a six cycle latency (not accounting for the channel costs to bring data chunks into the local store).

6.3 kMeans

The cycle statistics for *kMeans* is presented in Table 4 for various parameters. We fixed all trials to execute 30 iterations, to ease in comparisons. Interestingly, each iteration has the exact same statistics, since the computation is fixed and the SPU's mechanics are deterministic (no dynamic branch prediction, no cache effects, in-order issue, etc.).

From Table 4, we can see that only when the number of centers is very low is there any appreciable channel delay. Thus for *kMeans* it can be concluded that moving data to and from the local store is not the bottleneck. In fact, most of the slowdown with the first trials is not due to the channel, but because the number of dimensions is sufficiently low to stall the pipeline on loop boundaries. This can be addressed with vector pipelining, albeit painstakingly so. Also, it would likely require padding, depending on the dimensionality of the data. Rather than use the memory space

Processor	Time (sec)	Slowdown
Itanium 2 g	66	51
Itanium 2 i	29	22
Xeon g	31	24
Xeon i	12	10
Opteron 250	19	15
Pentium D	16	13
Pentium D 2	9	7
Cell SPU	7.4	5.9
Cell 6 SPU	1.25	–
Cell 8 SPU (sim)	0.95	–

Table 5: kMeans execution time comparison for various processors.

(the PS3 only has 256MB) we chose to use looping. As seen, when the number of dimensions increases, the SIMD instructions can be issued in succession, improving CPI (and FLOPs). For example, trial 1 uses 2 dimensions and has a CPI of 2.0. Trial 5 increases the dimensions to 40, and the resulting CPI drops to 1.21. Double issue rates rise from 9% to 20%. Our initial implementation did not use SIMD instructions, and the CPI was quite low. Since each floating point instruction performed only one operation, each loop in the distance calculation used many instructions, and the issue rate was high. After SIMD instructions were used, the CPI increased, but execution execution times lowered.

The scalability is healthy from 1 to 6 SPUs. For example, in trials 13 and 14, one SPU required 13.97 seconds and 6 SPUs required 2.38 seconds, for a speedup of 5.86. This near 6-fold speedup when moving from 1 to 6 SPUs is consistent in the other trials as well. Varying data set size behaved as expected, namely that twice as many points required about twice as much time (given the number of centers was far smaller than the number of data points). A final point to mention is that CPI and other statistics was generally fixed for a set of input parameters, regardless of the number of SPUs used. This is because, as long as there are sufficient data points to fill one DMA load, and the channel contention is low, the SPUs will be performing independently.

Table 5 illustrates the performance advantage of the Cell executing $kMeans$ as compared to other commodity processors. The parameters were DataPoints=200K, Dimensions=60, and Centers=24. The second best performance was afforded by the PentiumD, which is also a CMP. Because we do not have access to a real 8 core Cell, the slowdown column uses only our 6 core PS3 execution times.

6.4 kNN

The cycle statistics for *kNN* are provided in Table 7 for varying parameters. As with *kMeans*, kNN does not exhibit channel latency issues. Also, it can be seen that scalability is near linear. For example at 10 neighbors and 80 dimensions (trials 5 and 6), the execution time is reduced from 12.29 to 2.06 seconds, a 5.95-fold reduction when moving from 1 SPU to 6 SPUs. Also, in trials 2 and 6, the CPI is reduced from 1.53 to 1.02 when the workload rises from 10 neighbors and 12 dimensions to 10 neighbors and 80 dimensions. A larger number of dimensions results in longer record vectors, thus allowing more SIMD instructions per loop.

Increasing the number of neighbors degrades performance. This can be seen between the first trial and the third trial,

Trial	Input				Output						
	Centers	Dimensions	Data Points	SPUs	CPI	% Single Issue	% Dble Issue	% Branch Stalls	% Dep. Stalls	%Channel Stalls	Exec. Time(sec)
1	10	2	200000	1	2.00	32	9	16	40	0	1.17
2	10	2	200000	6	2.00	33	9	17	38	3	0.20
3	10	10	200000	1	1.32	40	18	18	20	0	2.18
4	10	10	200000	6	1.32	40	18	18	19	1	0.37
5	10	40	200000	1	1.21	42	20	14	24	0	3.49
6	10	40	200000	6	1.21	41	20	14	25	1	0.77
7	20	40	200000	1	1.17	43	21	13	23	0	4.87
8	20	40	200000	6	1.18	43	20	13	23	1	0.84
9	20	100	200000	1	1.16	44	21	7	28	0	8.91
10	20	100	200000	6	1.16	44	21	7	27	1	1.54
11	40	100	100000	1	1.05	49	23	5	23	0	6.98
12	40	100	100000	6	1.05	49	23	5	23	0	1.19
13	40	100	200000	1	1.05	49	23	5	23	0	13.9
14	40	100	200000	6	1.05	49	23	5	23	0	2.38
15	40	100	400000	1	1.05	48	23	5	24	0	28.1
16	40	100	400000	6	1.05	49	22	5	23	1	4.97

Table 4: Statistics for Kmeans on the Cell processor.

Processor	Time (sec)	Slowdown
Itanium 2 g	24	86
Itanium 2 i	9.46	34
Xeon g	9.44	34
Xeon i	8.02	28
Opteron 250	6.79	24
Pentium D	8.7	31
Pentium D (2)	4.64	16
Cell SPU	1.65	5.9
Cell 6 SPU	0.28	–
Cell 8 SPU (sim)	0.21	–

Table 6: Execution time comparison for various processors running K nearest neighbors.

Processor	Time (sec)	Slowdown
Itanium 2 g	138	19
Itanium 2 i	148	21
Xeon g	147	21
Xeon i	128	18
Opteron 250	131	18
Pentium D	126	18
Pentium D	71	10
Cell SPU	46	6.5
Cell 6 SPU	7.1	–
Cell 8 SPU (sim)	5.3	–

Table 9: Execution time comparison for various processors running $ORCA$.

where every parameter is held constant except for the number of neighbors, which is increased from 10 to 100. The subsequent CPI drops from 1.53 to 1.75, and branch stalls increase from 14% to 18%. Whenever a point d_j is found to be closer to the point being processed d_i, d_j must be added to d_i's neighbor list. This requires removing the weakest neighbor from the list and inserting d_j in sorted order. A larger neighbor list requires more search time, because a point is more likely to be a neighbor, and because adding that neighbor will be more costly. Recall that each (statically) mispredicted branch is a 20 cycle penalty.

An execution time comparison for kNN is provided in Table 6. The parameters were TrainingPoints=20K, TestPoints=2K, Dimensions=24, and Neighbors=10. As was the case with $kMeans$, the PentiumD's two execution cores afford it the second lowest execution times.

6.5 ORCA

The cycle statistics for $ORCA$, collected from the simulator, are presented in Table 8. As with the previous two workloads, scalability from 1 to 6 SPUs is excellent. The CPI clearly drops when the number of neighbors increases, because we have more branch misprediction due to inserting and sorting into a longer neighbor list. Branch stalls increase from 12% to 24% when increasing the neighbor list from 10 to 100 (trials 1 and 3). This also occurred with kNN.

The algorithm handles increasing dimensions well, as seen in trials 2 and 8. The dimensions is increased from 12 to 60, but the execution time only increases 21%. The increased dimensions improve the CPI. While the CPI only drops from 1.47 to 1.28, we point out that each additional FP instruction executes approximately 6 operations, and these operations are only 31% of the workload. Doubling the data set size from 100K to 200K requires 2.5-fold longer running times. This not surprising, since the worst case performance of the underlying algorithm is O(n^2).

The execution times for running $ORCA$ on the Cell are compared with the other processors in Table 9. The parameters are DataPoints=200K, Dimensions=32, Outliers=10, and Neighbor=40. The PentiumD is competitive, as the increased branching degrades the Cell's performance. Still, the Cell is several times quicker than the others, at least 6.5 times faster than the PentiumD.

6.6 Channel Stalls

If a significant amount of an algorithm's time is spent waiting for data transfers, the potential speedup of multiple execution threads may not be realized. In this experiment, we vary the data transfer size from 64 bytes to 8192 bytes for $ORCA$, in an effort to gain insight on channel stalls on a real machine, since our earlier channel stall data was given by the simulator. All values for this experiment are taken from trials on the PS3, and all trials use six SPUs to maximize DMA contention.

Recall that the exact chunk size must be a) a multiple

Trial	Input					Output						
	Neighbors (k)	Dim	Train Points	Test Points	SPUs	CPI	% Single Issue	% Dble Issue	% Branch Stalls	% Dep. Stalls	%Channel Stalls	Exec. Time(sec)
1	10	12	100000	1000	1	1.53	41	12	14	32	0	3.05
2	10	12	100000	1000	6	1.53	41	12	14	33	0	0.51
3	100	12	100000	1000	1	1.75	39	9	18	33	0	3.78
4	100	12	100000	1000	6	1.69	39	10	17	33	0	0.59
5	10	80	100000	1000	1	0.99	49	26	4	21	0	12.29
6	10	80	100000	1000	6	1.02	48	25	5	22	0	2.06
7	10	80	100000	2000	1	1.00	48	26	6	20	0	24.58
8	10	80	100000	2000	6	1.00	49	25	4	22	0	4.11
9	10	80	100000	5000	1	0.99	48	27	5	20	0	61.5
10	10	80	100000	5000	6	0.99	48	27	4	21	0	10.56

Table 7: Statistics for k Nearest Neighbors on the Cell processor.

Trial	Input					Output						
	Neighbors (k)	Dim.	Outliers	Data Points	SPUs	CPI	% Single Issue	% Dble Issue	% Branch Stalls	% Dep. Stalls	%Channel Stalls	Exec. Time(sec)
1	10	12	10	100000	1	1.40	49	11	12	28	0	15.8
2	10	12	10	100000	6	1.47	48	10	12	27	0	2.69
3	100	12	10	100000	1	1.96	35	8	24	22	0	65.31
4	100	12	10	100000	6	1.94	36	8	23	22	0	11.39
5	10	24	10	100000	1	1.36	53	10	10	27	0	15.36
6	10	24	10	100000	6	1.38	52	10	11	27	0	2.56
7	10	60	10	100000	1	1.28	59	9	6	26	0	19.36
8	10	60	10	100000	6	1.28	59	9	6	25	1	3.28
9	10	60	10	200000	1	1.27	59	9	6	25	0	48.26
10	10	60	10	200000	6	1.27	60	9	5	26	1	8.21
11	10	24	100	100000	1	1.36	53	10	10	27	0	28.2
12	10	24	100	100000	6	1.38	52	10	9	28	1	4.77

Table 8: Statistics for ORCA on the Cell processor.

of 16 bytes, and b) on a record boundary. Thus for this experiment we let a record be four single precision floats. The size of the data set is 80,000 records, the number of centers, neighbors and outliers are 10. As can be seen in Table 10, very small chunk sizes degrade performance significantly. However, at 64 bytes, only approximately 50% of the slowdown is attributed to cycles waiting on the channel load to complete. The balance is due to a) the decrease in SIMD parallelization (only a few calculations can be vectorized at once) and the increased number of instructions to set up channel transfers. Although in our previous experiments transfer sizes were about 6K, this experiment shows that if they are sufficiently small, channel stalls can be rather costly.

The break in the curve occurs with transfers of at least 256 bytes. At this value, the number of DMA loads required to process the data set dropped from 175 million to only 9.6 million, and the cost of each transfer only increased from 1216 cycles to 1504 cycles. The end execution time dropped from 28 seconds to 4.31 seconds. The lowest execution time occurs at a transfer size of 4096 bytes. The reason is that transfers larger than 4096 have only a marginal improvement in transfer time per byte, but incur a significant increase in the number of computations made. Recall that each synchronization allows SPUs to share their largest outliers and threshold values. These synchronizations generally increase the average threshold at the SPUs and afford improved pruning. Also we note that the cycles stalls per byte continually decreases as the size of the transfer is increased.

7. DISCUSSION

In this section, we revisit the questions posed in the Section 1.

Can data mining applications leverage the Cell to efficiently process large data sets? The answer for the applications considered is yes. The small local store of the SPU did not pose a practical limitation. In most cases, only two buffers were needed, each of which was the size of an efficient data transfer (near 6KB). Only with *kMeans* did we use more than 15KB, since the full set of centers was kept on the SPU. Note that even here we could have avoided this extra buffer space – see for example our approach with *kNearest Neighbors*. Basically, the $O(n^2)$ comparisons among the centers and the loaded points amortize the data loads to a sufficiently inexpensive cost. The overall insight here is that any meta data associated with a record can be inlined with that record, and moved to main memory when the record is ejected from the local store. For example, calculating k neighbors for D records requires $k * D * 4$ bytes. With low dimensional data, the number of records a DMA call can transfer becomes significant, and the local storage to maintain the k neighbors locally becomes the storage bottleneck. By simply inlining the k neighbors with the record, this storage requirement is mitigated (at the cost of lower cross-computational throughput per transfer). However, the local store size may be of greater concern with very large programs, since the instruction store and data store are shared.

Will channel transfer time (bandwidth) limit scalability? If not, what is the greatest bottleneck? From our experience (not just limited to this study), workloads which touch every loaded byte require less execution time on the Cell than on competing processors, regardless of the floating point computation requirements. For many data mining applications, this will be the case. Several studies, including one by Williams *et al* [23] recommend double buffering to reduce channel delays. For our workloads the channel stall times were relatively nominal when compared to other issues, such as branch and dependency stalls. For

Bytes	64	128	256	512	1024	2048	4096	8192
Execution time (sec)	28.2	13.7	4.31	2.78	1.82	1.45	1.20	1.40
DMA Loads (Millions)	175	78	29	9.6	3.3	1.0	0.36	0.10
Calculations (Millions)	216	218	222	226	228	243	264	303
Cycles per DMA Load	1216	1332	1483	1504	1950	2130	3960	7550
Channel wait time (sec)	11.1	5.01	1.89	0.64	0.26	0.11	0.07	0.04

Table 10: Channel costs as a function of data chunk size for 6 SPEs.

example, from Table 4 we can see that in $kMeans$ only 10 centers and 2 dimensions was sufficient to reduce channel stalls to 3% of the cycle time.

As a test, we implemented a simple $GetMax()$ program, which sifts through an array for the largest value. It was three times faster on the Cell than on the Xeon, with no special buffering or SIMD instructions. The lost time waiting on channel stalls was overcome by the fact that each SPU only searched 1/8th the data. As shown in Tables 4, 7 and 8, branching is a significant bottleneck. From our experience, it is a penalty which can be difficult to avoid. Using *select*() instructions will only remove the simplest cases. Dependency stalls appear high as well, but these costs can be lowered with additional loop unrolling and SIMD vectorization, and in fact were more than twice as high before we unrolled the outer loops. Branching is a natural programming concept and is used frequently. Eliminating branches, particularly when each flow of control requires complex operations, is not trivial and often cannot be amortized. For example, in our simple merge sort algorithm, up to 20% of the stall time was due to branching. In another test application, we implemented merge sort. We found it to be almost twice as fast as any other processor in our study, *without using SIMD instructions*. The Cell's benefit was the multiple concurrent cores. However, branching stalls were quite high. Also, the final merge was done such that each successive stage used only have the processors, with the last merge performed by the PPU. We are in the process of improving its performance, a topic for future work.

Which data mining workloads can leverage SIMD instructions for significant performance gains? Any distance-based algorithm has the potential for significant gains. This work targets data mining workloads which are, in some senses, the best case for the Cell. An algorithm designer can leverage the Cell's high GFLOP throughput to churn the extensive floating point calculations of these workloads. Also, the predictable nature of the access patterns (namely streaming) allow for large data transfers, where each byte in the transfer will be used in an operation. In these situations, the Cell can be extremely efficient. In many trials, our results from the Cell's real execution times via the PS3 exhibit many-fold GFLOP improvements over the theoretical maximums for all other processors in this study. This is due to the 25+ GFLOPs afforded by *each* SPU. We are currently designing pattern mining algorithms for the Cell, which do not have distance calculations. Our initial findings suggest that these algorithms also stand to benefit, primarily due to the additional threads of program control.

What metrics can a programmer use to quickly gauge whether an algorithm is amenable to the Cell? There are two questions to pose when evaluating the applicability of the Cell to workload. First, is the access pattern predictable? If so, then it is likely that chunks of data can be transferred to an SPU and most of those chunks will be involved in a computation. Second, is the workload parallelizable? In our experience, this question is often easily answered. Data mining applications in particular exhibit significant data-level parallelism, since it is common that each data object must be inspected.

At what cost to programming development are these gains afforded? Parallel programming is challenging, regardless of the target platform. For someone with parallel programming experience, the programming model afforded by the Cell is somewhat more difficult than conventional CMPs, such as Intel's PentiumD processors. The programmer must explicitly move data to and from main memory as necessary. However, after about a month of programming the Cell, we did not find this cumbersome. Several sources compare the Cell processor to GPGPUs. Both own multiple small processing elements which can be pipelined, both support SIMD instructions, and both require explicit data movement by the programmer. However, the Cell supports a very typical programming environment. The programmer uses everyday C functions, such as *malloc*(), and spawns threads in the same manner as traditional *pthread* models.

An added benefit of choosing the Cell as a development platform is that the Mambo Simulator is quite useful when tuning implementations. It provides cycle-level accuracy for the SPUs, and allows one to step through assembly level executions a cycle at a time. The programmer clearly sees which instruction stall the pipelines. In this regard, while prototype-level programs are unnaturally difficult to implement, highly efficient implementations may in fact be easier. A natural future direction then is to develop a framework which allows the programmer to specify data mining computations in a higher level language for fast prototyping. We are currently investigating such a platform.

8. CONCLUSION

In this work, we design and develop data mining algorithms and strategies for the Cell BDEA. Specifically, we illustrate that clustering, classification, and outlier detection can leverage the available bandwidth and floating point throughput to experience many-fold execution time reductions, when compared with similar codes on other commodity processors. In addition, we provide insight into the nature of a larger class of algorithms which can be executed efficiently on such a CMP platform. We believe that the findings in this effort are applicable to other domains which are considering the Cell processor as well. The structure of the general purpose CPU is in state of marked reconstruction, and future algorithm designers must consider these new platforms to see maximum utilization.

As part of ongoing and future work, we are investigating data mining algorithms which make use of complex pointer-based meta structures. Our initial experience suggests that

such algorithm would be rather cumbersome, and not overly efficient if implemented on the Cell. The local store does not have sufficient space to store the tree, which would require excessive transfers. Our observations in this study imply it is unlikely these transfers would be efficient.

9. REFERENCES

[1] K. Alsabti, S. Ranka, and V. Singh. An efficient kmeans clustering algorithm. In *In Proceedings of the IPPS/SPDP Workshop on High Performance Data Mining (HPDM)*, 1998.

[2] S. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the 9th International Conference on Knowledge Discovery and Data mining (KDD)*, pages 478–487, 2003.

[3] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995.

[4] Amitabh Chaudhary, Alexander S. Szalay, and Andrew W. Moore. Very fast outlier detection in large multidimensional data sets. In *ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 2002.

[5] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation: A performance view. In *IBM DeveloperWorks*, http://www-128.ibm.com/developerworks/power/library/pa-cellperf/, 2005.

[6] C. Elkan. Using the triangle inequality to accelerate kmeans. In *In Proceedings of the International Conference on Machine Learning (ICML)*, 2003.

[7] B. Flachs, S. Asano, S.H. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le1, P. Liu1, J. Leenstra, J. Liberty, B. Michael, H. Oh1, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano3. A streaming processing unit for a cell processor. In *Proceedings of the International Solid-State Circuits Conference*, 2005.

[8] Timothy Furtak, Jose Nelson Amaral, and Robert Niewiadomski. Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. In *University of Alberta Technical Report TR07-02*, 2007.

[9] A. Ghoting, S. Parthasarathy, and M. Otey. Fast mining of distance-based outliers in high dimensional datasets. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2006.

[10] N. K. Govindaraju, J. Gray, R. Kumar, , and D. Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *Technical Report MSR-TR-2005-183*, 2005.

[11] J. Han and M. Kamber. In *Data Mining: Concepts and Techniques*, 2000, 1967. Morgan Kaufmann Publishers.

[12] P. Horton and K. Nakai. Better prediction of protein cellular localization sites with the k nearest neighbors classifier. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 147–152, San Diego, California, USA, 2000.

[13] R. Jin and G. Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of SIAM International Conference on Data Mining (SDM)*, 2001.

[14] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.

[15] Sachin Kulkarni and Ratko Orlandic. High-dimensional similarity search using data-sensitive space partitioning. In *Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA)*, 2006.

[16] D. Kunzman, G. Zheng, E. Bohm, and L. Kale. Charm++, offload api, and the cell processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism at PACT*, 2006.

[17] S. Liao and M. Lopez S. Leutenegger. High dimensional similarity search with space filling curves. In *Proceedings of the 17th International Conference on Data Engineering*, 2001.

[18] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[19] D. Pelleg and A. Moore. Accelerating exact kmeans algorithms with geometric reasoning. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1999.

[20] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 154 – 165, Seattle, Washington, United States, 1998.

[21] Changzhou Wang and Xiaoyang Sean Wang. High-dimensional nearest neighbor search with remote data centers. *Knowl. Inf. Syst.*, 4(4):440–465, 2002.

[22] R. Weber and P. Zezula. The theory and practice of searches in high dimensional data spaces. In *Proceedings of the 4th DELOS Workshop on Image Indexing and Retrieval*, 1997.

[23] S. Williams, J. Shalf, L. Oliker, S. Kamil, P Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of Computing Frontiers*, 2006.

[24] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the International Conference on Supercomputing*, pages 712–721, 1991.

[25] M. Zaki, C. Ho, and R. Agrawal. Parallel classification for data mining on shared memory multiprocessors. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.