Int J Parallel Prog (2009) 37:345–369
DOI 10.1007/s10766-009-0106-9

Brain Derived Vision Algorithm on High Performance Architectures

Jayram Moorkanikara Nageswaran · Andrew Felch · Ashok Chandrasekhar · Nikil Dutt · Richard Granger · Alex Nicolau · Alex Veidenbaum

Received: 22 January 2009 / Accepted: 20 April 2009 / Published online: 28 May 2009
© The Author(s) 2009. This article is published with open access at Springerlink.com

Abstract Even though computing systems have increased the number of transistors, the switching speed, and the number of processors, most programs exhibit limited speedup due to the serial dependencies of existing algorithms. Analysis of intrinsically

J. Moorkanikara Nageswaran (✉)
Department of Computer Science, University of California, Irvine 92697, USA
e-mail: jmoorkan@uci.edu

A. Felch · A. Chandrasekhar
Neukom Institute for Computational Science, Dartmouth College, HB 6255, Hanover, NH 03755, USA

A. Felch
e-mail: andrew.felch@dartmouth.edu

A. Chandrasekhar
e-mail: ashok.chandrasekhar@dartmouth.edu

N. Dutt
Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA 92697-3435, USA
e-mail: dutt@uci.edu

R. Granger
Department of Psychological and Brain Sciences, Dartmouth College, HB 6207 Moore Hall , Hanover, NH 03755, USA
e-mail: richard.granger@dartmouth.edu

A. Nicolau · A. Veidenbaum
Department of Computer Science, Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA 92697-3435, USA

A. Nicolau
e-mail: nicolau@ics.uci.edu

A. Veidenbaum
e-mail: alexv@ics.uci.edu

parallel systems such as brain circuitry have led to the identification of novel architecture designs, and also new algorithms than can exploit the features of modern multiprocessor systems. In this article we describe the details of a brain derived vision (BDV) algorithm that is derived from the anatomical structure, and physiological operating principles of thalamo-cortical brain circuits. We show that many characteristics of the BDV algorithm lend themselves to implementation on IBM CELL architecture, and yield impressive speedups that equal or exceed the performance of specialized solutions such as FPGAs. Mapping this algorithm to the IBM CELL is non-trivial, and we suggest various approaches to deal with parallelism, task granularity, communication, and memory locality. We also show that a cluster of three PS3s (or more) containing IBM CELL processors provides a promising platform for brain derived algorithms, exhibiting speedup of more than 140× over a desktop PC implementation, and thus enabling real-time object recognition for robotic systems.

Keywords IBM CELL · Biological computer vision · Parallel algorithms · Cognitive computing

Abbreviations

BDV	Brain derived vision
B-U	Bottom-up engine
FD	Feature detector
LST	Line segment triples

1 Introduction

Our brains outperform engineering methods across a wide range of tasks, from perception and learning to higher level cognitive processing. The computing field is recognizing that biology is a valuable source of inspiration for solving problems in architecture, integration, and nanoscale systems [1]. Also, understanding and implementing brain-like systems in silicon is one of the grand challenges of the 21st century [2]. Recognizing this researchers are increasingly investigating brain mechanisms as models for novel algorithmic and architectural approaches [3–11].

The human brain contains billions of low-precision processing elements (neurons) that store memories in a highly distributed fashion via their connections (synapses), without a central processor. These brain circuits are organized into specific architectures that can perform complex and quite understandable algorithms, conferring unexpectedly powerful functions to the resulting composed circuits [8,12].

A key aspect of the power of brain circuit architectures is their massive parallelism that apparently provides a method for using large numbers of low-precision processing elements to compute complex calculations in relatively few serial steps. For instance, neurons are remarkably slow, taking milliseconds to compute or transmit information, yet animals recognize visual objects in a fraction of a second; i.e., the entire computation is carried out using millions of neurons but less than 100 serial steps [13]. In contrast, except for few applications, parallelization of serial code typically elicits very limited speedup, due to Amdahl's Law [14]; and even the task of identifying the possible amount of speedup for a given task may require a significant investment of time.

Brain systems constitute massively parallel architectures that carry out intrinsically parallel algorithms, rather than parallelizations of inherently serial methods. Algorithms derived from the operating rules of the brain circuits thus have correspondingly few serial dependencies, and are well poised to take advantage of parallel hardware such as multicore processors, graphics processors, etc.

In this article we present a comprehensive overview of one type of brain-derived vision (BDV) algorithm, based on studies of the mammalian visual system, and applicable to real-world visual object recognition tasks [5]. As expected from a brain-derived method, the system is highly parallel, and computationally efficient. We demonstrate its implementation in the IBM Cell processor [15,16] and in small clusters of these processors as in Sony Playstation 3 systems.[1] In this article we make two important contributions: first, we present the detailed computational model of the bottom-up engine in the BDV algorithm by extending our previous article [7]; second, we also show the various trade-offs in the implementation and performance prediction of this model on IBM CELL.

2 Background

The brain is a non-homogeneous structure with various regions or sub-structures whose computation emerged through an evolutionary approach, rather than using the formal notion of Boolean algebra [18]. Among the various structures, the outer surface of the brain called the cerebral cortex forms a critical part in the brain for cognitive functions. The structure of different cortex circuits are surprisingly similar, even though different parts of the cortex are responsible for various cognitive functionality like audition, vision, sensation, decision making, and motor control. Various theories and models have been postulated by the computational neuroscience community about the underlying mechanisms that are involved in the behavior of cortex. In this article we demonstrate a practical realization of a brain derived vision algorithm inspired by the anatomical structure and physiological operation of the brain. We would like to stress that the algorithm presented here is a simplified model of the human vision. The actual human vision, even at the V1 stage, is extremely complicated and many things are unknown [19].

Many interesting computational principles can be derived by understanding how the visual pathway of the brain achieves object recognition and scene understanding. We briefly look at the elements of the visual pathway that will be helpful for deriving our simplified vision algorithm. Light enters the visual pathway by activating the rod and cone receptor cells in the retina. Further processing is done by the last stage of retinal cells called the ganglion cells [20]. These cells respond to a specific pattern of light, called the receptive field of the cell. Two kinds of receptive field exist in these cells, namely on-center/off-surround and off-center/on-surround. The responses from these cells are based on the difference in light between the center and the surround region. Thus a uniform light pattern will evoke a very low response compared to a well

[1] Some early results were also presented at 2007 CELL/B.E. University Challenge, and won the first prize. The B-U engine source code can be obtained from SourceForge [17].

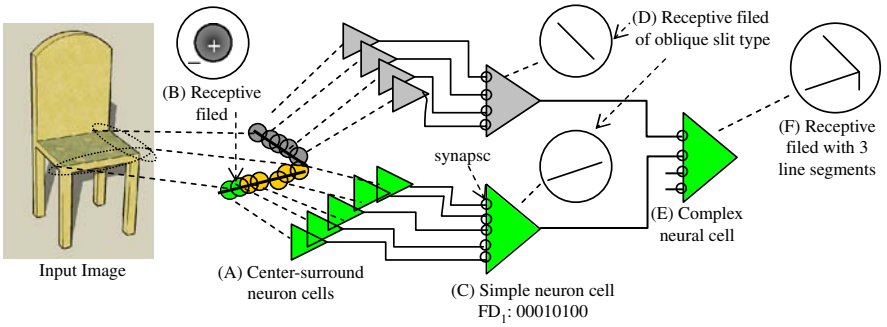

Fig. 1 Initial stages of the visual pathway in brain architecture: We do not show the photo receptors (rods and cones) in this figure. Cells indicated as (**a**) corresponds to the ganglial cells having the receptive field shown in (**b**). The simple cells (**c**) are the first layer of the cortex and have a simple receptive field (**d**) which is sensitive to line orientations. The simple cells reads many inputs from neurons in the previous layer and send the output to many complex cells. The receptive field of a complex cell is indicated in (**f**) which in our case is a combination of three line segments. The LGN cells are not indicated because its receptive field is similar to the ganglial cells

placed bright spot or line or an edge passing through the center of the cell's receptive field. We can also observe from Fig. 1 that the edge of the chair in the input image triggers a group of neurons because the line falls at the center of their receptive field. The axons of the ganglion cells bundle together as the optic nerve, and proceed to the lateral geniculate nucleus (LGN). Here the path bifurcates, and one path proceeds to superior colliculus and other path to the primary visual cortex. The neurons in the primary visual cortex can be broadly classified into Simple cells and Complex cells [20]. Both these cells are orientation specific cells, meaning they best respond to a particularly orientated stimulus. We show two simple cells that respond to lines of different orientation (Fig. 1d). If the stimulus orientation changes more than 10–20° then the response of the simple cells decreases drastically. At about 90° the simple cells give no response. While the simple cells respond to specifically oriented lines or edges, the complex cells exhibit more interesting characteristics, such as larger receptive field, directional selectivity (responds to one direction of movement of a line or edge), more complex organization of lines, crossing detection, and length tuning. All these properties of complex cells are achieved by appropriately connecting a group of simple cells. Receptive fields of cells in visual area V2 is shown to exhibit similar kind of response for complex features [21]. In Fig. 1 we show a complex cell (E) that responds to a specific pattern which is a combination of three line segments shown in (F). Thus a hierarchical organization of complex cells can be used to encode or decode complex shape, patterns and other details present in the image (see Fig. 2). The actual human visual system goes beyond complex cells in V2 to higher layers like V3, V4 (tuned to spatial frequency and color) and MT (tuned for perception of motion). These aspects are not represented in our current model.

The visual pathway that we briefly described above can be converted to a computational model by modeling and encoding the different elements (complex cells, synapses, etc.) using a suitable abstraction. In our BDV algorithm we employ a simplified visual cortex abstraction model that is based on line segments and the encoding scheme by means of sparse bit vectors [22]. We now describe some principles used

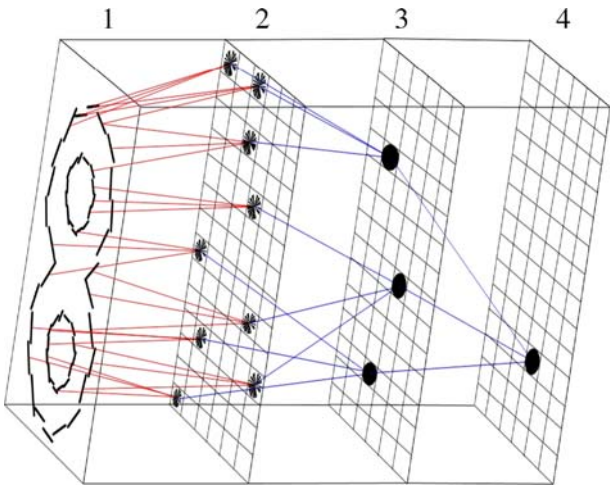

Fig. 2 Hierarchical organization of the BDV algorithm into the Bottom-Up unit (Layer 2) and hierarchical nodes (Layers 3, 4). The algorithm is operating to recognize number 8 in a hierarchical fashion. Each element in Layer 2 corresponds to a feature detector (FD). A group of FD in Layer 2 triggers an element in the higher layers. Each FD in this figure corresponds to a complex cell

in the BDV computational model. A given set of input neurons (pattern) activates a particular neuron and this set is loosely referred in our article as *feature detector vector (FD)*. Hence the property of the neuron (simple or complex) can be represented by the feature detector vector because the neuron fires only for that specific set of input pattern or feature (see Fig. 1d). Any two shapes are considered similar based on the number of activated neurons shared in their activation patterns. As a result of sparse population codes [23], most neurons are inactive; this concept is represented in a highly simplified form as sparse bit-vectors. The intrinsic random connectivity tends to select some areas of neurons to respond to some input features. These neurons train via increments to their synaptic connections, solidifying their connection based on preferences to specific input features. After a simulated developmental phase, synapses are either present or absent and each neurons level of activation can be represented as a bit vector. Also each complex cell characteristic can be represented as a feature detector vector (FD). When many complex neuronal cells (FDs) are triggered only some selected neurons are allowed to proceed to trigger the next stage in the hierarchy, and this is achieved by local inhibition of other neurons. Neurons activate local inhibitory cells that in turn deactivate their neighbors; the resulting competition among neurons is often modeled as the K best (most activated) "winners" take all or kWTA [24,25]. More details of the FD and kWTA will be presented in Sect. 3. Some of the neurological underpinnings presented in this article are not completely new, especially the concept of hierarchical models is very commonly used in various vision algorithms. But the main contribution of our approach lies in efficient coding, hierarchical representation and recognition of complex shapes using a simple sparse bit-vector scheme.

We now describe other computational models inspired from human vision system. Stringer and Rolls [26] presented a four-layer feed-forward network model called

VisNet, which is based on primate ventral visual system. The model employs trace learning rule which uses the temporal proximity of patterns for implementing view invariant object recognition. Riesenhuber and Poggio [10,27] demonstrated a feed-forward model having a hierarchy of feature detector layers. Each element of the feature detector layer is modeled based on the tuning properties of cell in the visual system. This feed-forward model was demonstrated to perform a range of object recognition tasks, and was also found to be invariant to scale and position. George and Hawkins [28] demonstrated a hierarchical Bayesian model for pattern recognition inspired from some of the characteristics of visual cortex. In contrast to other approaches our approach is based on sparse bit-vector which is most suitable for implementation on hardware or SIMD based processors. For the rest of this paper we mainly concentrate on the details of our brain derived vision algorithm using sparse bit-vector representations, and its implementation on the CELL processor. For more details on biological validity and different types of brain inspired algorithms the reader is directed to other sources [5,22].

3 Algorithm

The BDV algorithm can be divided into two components: the bottom-up (B-U) engine, and hierarchical nodes. The B-U engine models some of the functionality of the initial stages of the visual pathway that is illustrated in Fig. 1. Basically the B-U engine extract shapes from the input images, and measures how similar each shape is to a reference set of shapes. The resulting match values, along with the shape's size and position, are fed to a set of nodes that are organized into separate layers. The layers are organized hierarchically, so that the first layer receives input from the B-U engine; the second layer receives input from the first, and so on (see Fig. 2). Each node recognizes a shape by first combining multiple input shapes into a single more complex shape, resulting in a hierarchical organization. This enables later layers to recognize larger shapes, and eventually entire objects. From our initial studies we identified that the B-U layer constitutes the major portion of the computation, and hence in the remaining section we mainly focus on the details and performance of the B-U engine in the BDV algorithm.

Two operations namely line extraction and line segment triple coding are important in the understanding of the BDV algorithm. These operations characterize the behavior of the visual pathway until the first stage of the complex cells. Before we explain the steps in the BDV algorithm we examine the functionality of these two operations.

- `Line extraction`: In this pre-processing step, the input image is converted to a gray scale image, and the edges are extracted using the Canny edge detector. The line segments are then derived from the edge detected image by starting a line at an edge, and gradually moving the second end point along that edge until the average distance between the edge and the line segment gets too high. The processed image in Fig. 3 shows the lines extracted from an image in an office scene. Next, line segments are joined together into groups, called line-triplets (indicated as extracted shapes in Fig. 3), such that each group contains three different line segments. The number of unique line segment groups is n-choose $-3(= n!/(6 * (n-3)!)$, where

Algorithm 1 B-U computation in the BDV algorithm: The PC captures a viedo frame, and extract 400 to 600 line segments and groups them into 20,000 to 100,000 line triples. PC then sends encoded line triples to some nodes in the cluster. For each line triple (encoded as 160-bit LSH vector), the B-U engine generates a list of best matching shapes, threshold, and maximum of the match value

Input:
FD1 → Feature Detector Vector 1 table (8192 entries of 160 bits each)
FD2 → Feature Detector Vector 2 table (1000 entries of 8192 bits each)
LSHVector → Line segment vector of 160 bit (20,000 entries).

1 **foreach** *LSHVector in the frame* **do**
`// 1. Input Stage`
2 read new LSHVector from PC
`// 2. FD1 Stage`
3 Computes the dot product of FD1 neurons with a 160-bit LSHVector based on input frame data; Additionally computes a threshold so that about 512 FD1 neurons are active
4 $FD1Match[.] \leftarrow 0$ `// 8192x8bit array, stores FD1 match`
5 $histogramBin[.] \leftarrow 0$ `// '[i]' has number of FD1s with match=i`
6 **for** $i \leftarrow 1$ **to** 8192 **do**
7 $popCount = Compare(LSHVector, FD1[i])$
8 $FD1Match[i] = popCount$
9 $histogramBin[popCount] + +$
10 Below we determine the threshold match value so that k FD1 elements($k = 512$) has match value greater than the threshold
11 $i \leftarrow 160, totalsum \leftarrow 0$
12 **while** $totalSum \leq 512$ **or** $i \neq 0$ **do**
13 $totalSum = totalSum + histogramBin[i--]$
14 $threshold = i$
`// 3. k-WTA Stage`
15 Computes a vector indicating which FD1 neurons are receptive (has a match greater than the threshold value)
16 **for** $k \leftarrow 1$ **to** 8192 **do**
17 **if** $((FD1Match[k] > threshold)$ **then** $midVector[k] = 1$
18 **else** $midVector[k] = 0$
`// 4. FD2 Stage`
19 Computes dot products between the resultant midVector and all FD2 neurons; Outputs the FD2 index if the population count of the dot product is over threshold2
20 $current\_max \leftarrow 0$
21 **for** $i \leftarrow 1$ **to** 1024 **do**
22 $popCount = CompareBits(midVector, FD2[i])$
23 **if** $(popCount > current\_max)$ **then** $current\_max = popCount$
24 **if** $popCount >= threshold2$ **then store** $i, popcount \quad in \quad Table1$
`// 5. Output Stage`
25 **Output** $threshold2$, $current\_max$, $Table1$

n is the number of line segments in the image. This results in an explosion of line-triplets when there are many line segments in the image. To keep this manageable, a filter is used based on how close the line segments are to their center. A line-triples center is calculated as the average x and y values of the six end points that makeup the line-triple. Next, the average distance between the end points and the midpoint is calculated, which is called the line triples shape size or shape scale. An example for various data that is extracted from the shape is shown in Fig. 3.

- `Line coding`: The vision system attempts to create a scale-invariant representation of the line-triplets by encoding angle relationships between the line end

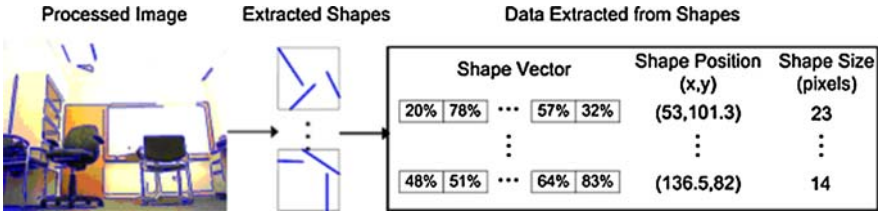

Fig. 3 Overall flow of bottom-up part in the BDV algorithm. Features or shapes are extracted by the bottom-up engine in the form of small groups of line segments. Each shapes size and position is extracted, along with its similarity to each shape in a reference set. This information is used by the hierarchical layers in the BDV algorithm

points (see Fig. 4 for an example). Each angle of the end points will be between 0 and 90°. Rather than using a floating point number, we compress this to a single integer between 0 and 56 (inclusive).[2] In the end, we have a list of angles, each from 0 to 56, sorted in a special order that encode relationships between end points. The representation is made translation invariant by suitably choosing the origin. The representation is *scale-invariant* because the angle relationships do not change if the line segments are zoomed in or out. Thus if we zoom the example in Fig. 4 two times, the resulting encoding would remain the same. This representation of line-segment triple is termed as *Line segment hash vector (LSH vector)* [5]. The encoding process is based upon the ordering of the line-segment triples and the angles taken by each line segment triples. If for example the second segment shown in Fig. 4 goes along the 45° line further, then at some point the ordering would change because of other lines present in the line segment triples and a new code will be generated. Also, if any one of the line angles changes by more than (90/57) degrees either due to change in length or actual change in angle then a new code value will be generated by the LSH line coding approach.

The above two steps (line extraction and line coding) form the primitive operations in our algorithm. We now discuss the details of the various stages of the B-U computation in the BDV algorithm. The pseudo-code listing is shown in Algorithm 1 and the functional model of the BDV algorithm is shown in Fig. 7. During each iteration of the B-U computation, the Input Stage reads a new encoded line segment triple data, and passes the data to the remaining stages. The input line segment triple can come from either a pre-processed file stored locally or from another PC connected to a camera. The details of the remaining stages are described below (the line number in the description corresponds to listing shown in Algorithm 1).

- `FD1 stage (Lines 3–14):` In the FD1 step each angle encoded line segment triple is compared against a table of reference line triplets (see Fig. 5 for an example). Each element of the table is called the feature detector vector 1 ("FD1") or reference vector set. The FD1 table is created during the simulated

[2] 57 different angles can be represented by an 8-hot of 64 bit-vector, where the 8-hot bits are contiguous. This allows dot-product of different angles, and similar angles have a higher dot-product. This was a desirable feature based on the biological plausibility of sparse encodings and bit-vector dot-product. The number of hot bits was chosen to be eight because this allows non-zero dot-products between angles that differ by up to 26 degrees, which seemed reasonable.

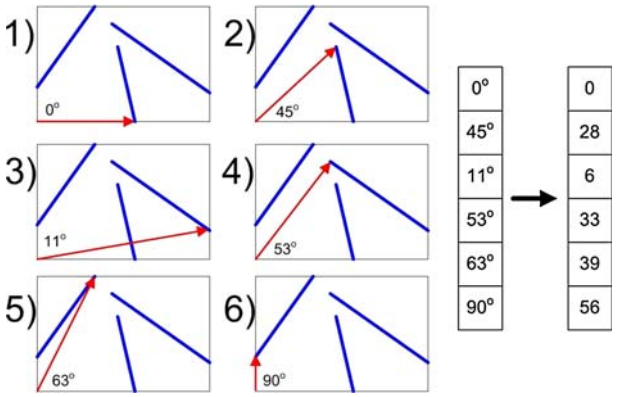

Fig. 4 Line segment triple coding: The process of converting the line segment triples into a vector of quantized angles. The resulting code is scale-invariant because the angle remains the same if we zoom in or out

developmental period, during which the line-triplets were taken randomly from many different images and gradually added to a table until the count reached 8000.[3] More specifically, a line-triplet was added if it did not have a very good match with any of the shapes already in the set. The effect is that when new shapes are matched against the 8000, they will match at least one very well. Two line segment triplets consisting of lists of angles a and b can be compared to get a single match value using the expression, match $= \sum_{i=1}^{n} \max(8 - |(a[i] - b[i]|, 0)$. This equation simulates the dot-product of angles (explained in Footnote 2). The result is that comparing similarly shaped line triplets results in higher matches. Very different looking line-triplet shapes result in lower matches. The resulting match value for each of the line segment is passed on to the next stage of computation.

- `kWTA stage (Line 15–18):` Neurons activate local inhibitory cells that in turn deactivate their neighbors; the resulting competition among neurons is often modeled as the K best (most activated) winners-take-all or kWTA (see Fig. 5 where K = 3). The kWTA effect occurs after the comparison of the 8000 FD1s with a new input line-triplet. The 8000 FD1 matches are analyzed and the highest K matches are considered active. This is represented in an 8,000 bit vector by setting 500 bits to ON (when K = 500),[4] corresponding to those shapes that matched best. We shall call this new 500-of-8,000 bit vector, the "mid-vector", because it occurs in the middle of a two stage process. kWTA computation provides high robustness, and variation tolerance compared to single winner-take-all computation.
- `FD2 Stage (Line 19–24):` This stage uses the mid-vector as input to a set of 1,000 bit-vectors (called "FD2s"), each 8,000 bits in length (see Fig. 6).

[3] We assumed an inhibitory neuron connection density of 10% of excitory neuron. And each inhibitory neuron is connected roughly to 1000 other neurons yielding a total of 10000 complex neuron cell. Each of this neuron represents a feature detector in the FD1 stage. The object detection performance did not change with 8000 elements.

[4] Assuming only 6% of the neuron is firing after FD1 stage, hence approximately 480 will be active out of 8000 neurons giving an activity density of 6% [25].

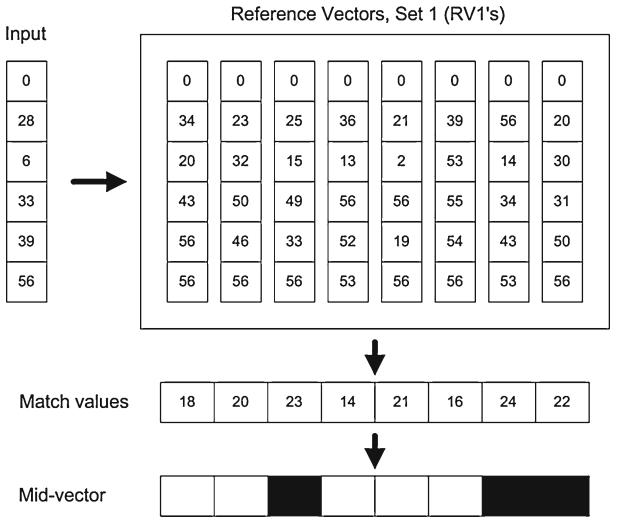

Fig. 5 FD1 Stage and kWTA stage: The quantized vector of angles is compared to each vector from the reference set of line-triplets (FD1s) and their match values are computed. Last, the top K (in this case, $K = 3$) match values are set to 1, and all others to 0, resulting in the mid-vector representation

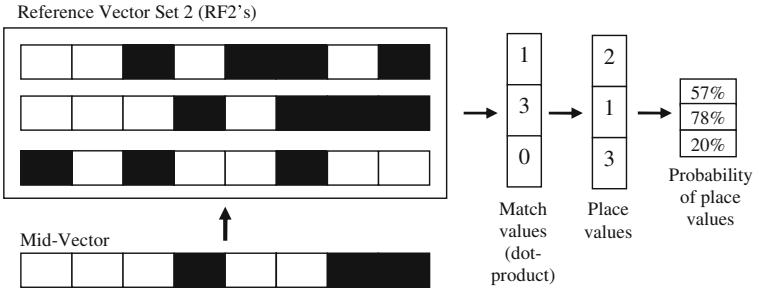

Fig. 6 FD2 Stage: The mid-vector from previous stage is compared against FD2's and sorted based on the match value. Also the match value can be converted to a vector of probabilities

The mid-vector is dot-product with each FD2 to determine how well each FD2 matches the input. As a second step, the match values are sorted, so that each FD2 is assigned its index in the sorted list of matches starting from highest match to the lowest. This index is called the FD2s "rank" value. If an FD2 has a rank value of x, it means that it was the xth highest matching FD2 for that input. These place values are converted to a probability based on previously collected statistics. The results including the list of matching FD2 are transferred to the PC or stored locally by the Output Stage for further processing. All objects that need to be recognized are a combination of elements of FD2, and located at specific relative distances with respect to each other. To enable recognition, an early prototype of the hierarchical node was implemented, built to serve the very simple purpose of creating expected locations of particular shapes inside a recognizable object relative to other shapes within that object. The approach presented above is

useful for recognition of object with a limited change in 3D perspective. If the 3D perspective changes drastically then new line-segment representation is generated and hence the recognition performance would drop drastically [22].

4 Mapping BDV Onto the IBM CELL

In this section we analyze the computational and parallelism aspects of the BDV algorithm explained in the previous sections. A salient feature of the BDV Algorithm (described in Listing 1) is the high degree of parallelism at various levels. The simplest inherent parallelism is the bottom-up computation for different line segment triples (see Line 1 in Algorithm 1). In our experiments, most pictures contain about 20,000 to 100,000 useful line segment triples. Hence the best matching FD2 for all of these line segment triples can be potentially concurrently evaluated. The next level of parallelism is achieved by the shape comparison operations (Line 6, and Line 21 corresponds to the shape comparison using FD1 and FD2 tables). For each line segment triples we need to find a best matching shape from the given table of shapes (FD2 and FD1 tables). This search can also be parallelized to a higher degree based on available computing resources and the communication overhead. The algorithm also exhibits large amounts of bit-level parallelism and SIMD parallelism. For example in the FD2 computation we need to evaluate an 8,000-bit dot-product and population count[5] on the result to estimate the degree of match between two FD2 vectors (Line 21). This can be concurrently executed either at the bit, byte or at higher word levels.

Among the various means of acceleration of the B-U engine (for example FPGA, GPU, computing clusters), the IBM CELL provided an excellent platform for exploiting the various types of parallelism present in the algorithm. The IBM CELL Broadband Engine (CELL BE) is a high-performance, low-cost multi-processor with eight specialized Synergistic processing engine (SPE) and one dual-threaded PowerPC Processor operating at around 3.2 GHz [30]. Even though the CELL BE has been designed targeting graphics and multimedia applications, many features of the CELL BE facilitate acceleration of brain derived algorithms. Some of the features that make it promising for BDV algorithm are: the 128-bit datapath allowing large bit length operations; good amount of on-chip parallelism and bandwidth; special instructions for bit based computations; and dedicated hardware for synchronization using signals and message queues that allow fast synchronization between threads. Also the Sony Play Station3 (PS3), powered by the IBM CELL processor, is an affordable platform at a cost of about $500. The PS3's built-in Gigabit Ethernet port makes it suitable for building small clusters. The main challenge is to efficiently parallelize and optimize an algorithm to exploit the capabilities of the CELL processor.

We now present the programming methodology and trade-offs involved in mapping the BDV algorithm on the CELL platform. We have used different levels of programmable parallelizations available on the CELL: (1) parallel execution of many CELLs (network or cluster of CELLs), (2) parallel execution of SPEs and PowerPC

[5] Also called *sideways sum*, is the process of counting the number of '1' in a bit-vector [29].

Processing Element (PPE) within the CELL, (3) concurrent computation and communication (DMA operation) within the SPEs, (4) parallel execution of two instructions in the SPE, and (5) parallel computation at the sub-word boundary by using SIMD instructions (up to 16 single byte operations in one cycle). In the remainder of this section we examine the approaches for mapping BDV algorithm across the CELL clusters and within the CELL. Compiler assisted parallelization using IBM XL compiler [31] will be part of our future studies.

4.1 Application Computation Analysis

A simple functional model of the bottom-up computation is shown in Fig. 7. The input stage reads the line triplets encoded as a sparse vector. The FD1 and FD2 stage finds the best matching line triplets from the reference set. The B-U engine was initially executed on a 2.13 GHz Intel Core2 (E6400) CPU and the execution time of critical functions were profiled to determine the various computational bottlenecks. The model is single threaded and was compiled with highest level of optimization supported by gcc. No SIMD optimizations were performed on the desktop PC implementation. A fractional breakdown of execution time for different functional units is shown in Fig. 8, and the absolute values are shown in Column 2 in Table 3. Approximately 1.89 ms was required to execute the B-U computation for a single line segment triple on the Intel architecture. This corresponds to a B-U computation throughput of about 526 line segment triples per second (526 LST/s). In Fig. 8 we can observe that FD1 and FD2 stage take more than 95% of the overall execution time. These are the critical functions that need to be optimized, and parallelized to increase the throughput of the B-U computation. From these results, the runtime of processing an entire video frame with 30,000 line-triples on a desktop PC can be estimated at approximately one minute. To execute the BDV algorithm for interactive robots, the recognition time of humans must be achieved (approximately 150 ms per frame [32]), thus requiring a speedup of about 400×, i.e., from 526 LST/s to more than 200,000 LST/s.

4.2 Application Data Analysis

The code size and the data size (both static and dynamic) need to be evaluated to effectively determine the memory footprint, and the bandwidth requirements of the given application. Each SPE has a local store (LS) of 256 KB that can be used for both code and data. This small memory size influences the way code and data for the applications are partitioned across the CELL. For many programs with a small code size, function overlaying and resident partition management [31] might not be necessary. The total code size for the B-U computation is about 25 KB and hence no function overlaying mechanism was used in our implementation. Furthermore, we need to determine how to map the data sets with size larger than the available SPE LS.

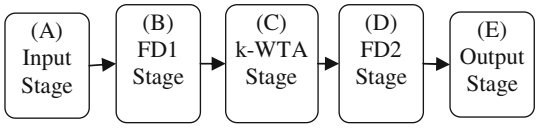

Fig. 7 A simple functional model of the Bottom-Up computation. The functionality of each stage in this model is shown in Algorithm 1

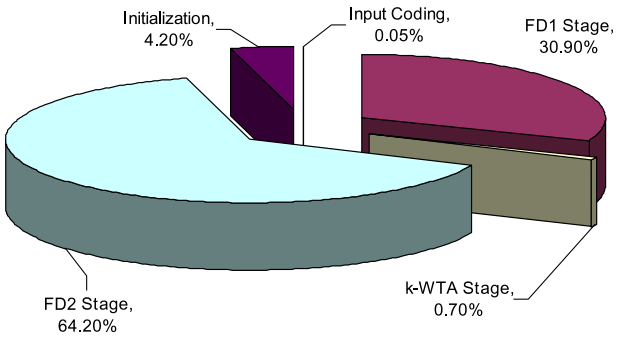

Fig. 8 Breakdown of the execution time of the functional model on desktop PC with Intel Core2 CPU. FD2 stage and FD1 stage takes more than 95% of overall execution time

Table 1 Data structure analysis of B-U engine in BDV algorithm

Main data structures	Data size (bytes)	Data usage	Data access pattern	Accessing tasks
FD2 Vector table	1.1M	Partitionable	Linear	(D)
FD1 Vector table	169K	Partitionable	Linear	(B)
Sparse code table	125K	Static/fixed	Random	(A)
SC angle table	48K	Static/fixed	Random	(A)
popCount FD1	8K	Static/fixed	Random	(B)(C)
Histogram kWTA	2K	Static/fixed	Linear	(B)(C)

We list major data structures, its size and some additional properties. If the data structures divides during parallelization the data is termed "partitionable", else the data structure is termed "fixed or static". The labels indicated in the last column corresponds to different stages in Fig. 7

To reduce this constraint, various techniques such as software cache, double buffering, pre-fetching [31], etc., can be used depending upon the data access pattern.

Some important data structures used by the program and its properties are listed in Table 1. The main parameters that were evaluated are : memory size, data usage, and data access pattern. The data structures are classified based on their usage as static (or fixed) and partitionable. A data structure is termed as static (or fixed) if the data set need to be duplicated in each SPE while parallelizing the application. A data structure is termed as partitionable if the data set gets divided across SPE while parallelizing the application. This information influences the parallelism and data access mechanism used by the application. This information is also useful to determine the SPE bandwidth and the SPE LS memory requirements for a particular parallel model.

For our application we have four large data structures (Table 1) namely FD1 Table, FD2 Table, Sparse Code (SC) Ordering Table and Angle Table. The SC Ordering and Angle tables are used by the PPE for LSH bit-vector generation at the Input Stage. The remaining data sets can be accessed either through software cache or directly on SPE LS. Also, we observe from the algorithm that FD1 and FD2 table elements are accessed linearly (one after another in a specific sequence) during the comparison operation with the given input data. Hence if FD1 and FD2 do not fit into the SPE LS, then they can utilize either double buffering or software cache with data access optimization to allow efficient access to large data arrays.

4.3 Parallelism Analysis

Various kinds of generic parallel models can be developed from the functional model of the B-U engine. The CELL Programming handbook [30] explains some generic strategies for parallel programming on IBM CELL. Analysis of generic parallel programming patterns or models for different architectures is a challenging task [33]. In this article we address the different parallelism strategies and performance prediction that is specifically applicable for brain derived algorithms on the CELL. These techniques can be also used for parallelism analysis of other applications as well. The possible parallel models are: overlapped functional parallel model (OFP), overlapped data parallel model (ODP), series-parallel model (SP), and overlapped series-parallel model (OSP). Section 4.4 presents performance prediction approach of these models. A model is termed overlapped if the communication and computation can happen concurrently, and hence the waiting time associated with communication can be mitigated. We present below the characteristics of the parallel models relevant to brain derived algorithms.

- Overlapped functional parallel (OFP) model: In this model each functional block is mapped onto an SPE, and the model works in a pipelined fashion (see Fig. 9a). Hence the actual execution time of the model is dependant on the execution time of the slowest functional block. An OFP model can be extended to a process networks or data flow networks [34] using appropriate communication APIs. Since the SPE LS is of a very small size, usage of these communication APIs will reduce the available memory resources even further. Load balancing limits the performance of the OFP model because the overall execution is determined by the slowest task in the pipeline.
- A Series-Parallel (SP) model: A simplified version of this model (see Fig. 9b) overcomes the load balancing limitation of a fully overlapped functional parallel model by splitting the sequential model into a series-parallel graph at loop boundaries either manually or by using the compiler (OpenMP primitives). If the data and computation is evenly partitioned across these loop boundaries, this kind of parallel model exhibits good load balancing, speedup, and reduced SPE LS requirements. A version of the SP model for the B-U engine is shown in Fig. 10. The serial portion can either be executed in the PPE or SPE, depending upon the complexity of the serial task. The main qualitative advantage of this model is lower data size requirement in each SPE, as well as reduced application latency and reduced SPE bandwidth. Two disadvantages of this model are, (1) the serial portion can affect the overall execution time, (2) potentially high synchronization and communication time between series and parallel task can increase the waiting time of communicating tasks.
- Overlapped Series-Parallel (OSP) model: This model (see Fig. 9c) extends the series-parallel model by overlapping computation and communication using either a double buffer or a FIFO. Thus instead of waiting for the serial portion to finish its operation and communicate the result to the SPE, the SPE performs the computation for the next input data. During this time the serial portion completes the execution and communicates the data by means of DMA so that the data is

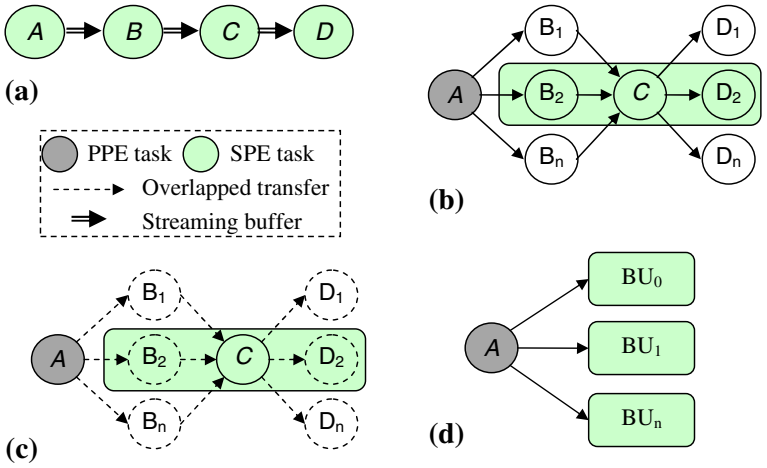

Fig. 9 Summary of various parallel models on CELL. In model (C) overlapped data transfer is achieved by using DMA MFC within each SPE. The functional parallel model (**a**) is obtained from the simple functional model in Fig. 7 by using functional parallelism. Also streaming buffer is added to transfer the data between different task running on different SPE. In (**b**) and (**c**) the same functional task is executed by different SPEs using a partial reference set of vectors

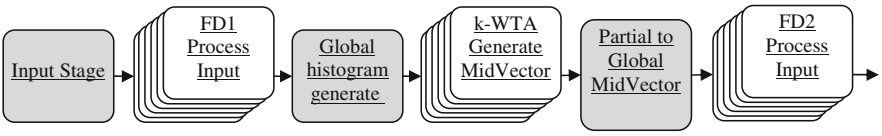

Fig. 10 Series-Parallel Model of the B-U Computation. The computationally intensive functions FD1, kWTA and FD2 is parallelized across SPEs by dividing the reference tables equally between the SPEs

ready for the next cycle of SPE computation. OSP model solves many disadvantages of SP model, but does not solve the problem caused by potentially high synchronization requirements between the serial and parallel portions.

- Overlapped Data-Parallel (ODP) model: For much higher performance the overlapped data parallel (ODP) model can be employed (see Fig. 9d). In this type there is no data sharing or communication between the SPEs. Earlier models parallelize the bottom-up so that only part of the code or data is mapped on to the SPE. But in ODP model an SPE can be treated as a full processor and the complete bottom-up computation for a line segment triple is mapped to a single SPE. This model requires large SPE bandwidth and large SPE LS because all the data and code for the execution of the application must be present or accessible by the SPE. The performance of this model is dependent on the technique used to overcome the code and data size restrictions within the SPE LS. Table 2 gives a quick qualitative comparison of different kinds of parallel models.

4.4 Performance Prediction

For each application it is important to understand the trade-offs associated with the various parallel models, and select one that matches the characteristics of the

Table 2 Brief summary of important characteristics of different parallel models on IBM CELL

Model name	Compute latency	On-chip bandwidth	Memory usage	Modeling effort	Performance
Functional parallel model	Medium	Medium	Low	High	Low-medium
Series-parallel	Medium	Medium	Medium	Medium	Medium
Overlapped series-parallel	Low	Medium	Medium	Medium-high	Medium-high
Overlapped data-parallel	High	High	High	High	High

Compute latency is the time delay between applying a specific input and obtaining the required output

application. Obtaining the performance of all the parallel models by implementing them on the CELL BE is a difficult task. In this section we suggest a mechanism for obtaining the approximate performance of various parallel models from the performance of a single model (namely series-parallel model). Column 3 of Table 3 shows the actual execution time of a series-parallel model running on a PS3 with IBM CELL. The remaining columns contain the estimated values of the execution time for other parallel models. The tasks shown in Column 1 of Table 3 corresponds to the various tasks in Figs. 7 and 10. It should also be noted that the two functions, namely Global histogram (C) and Global MidVector (E) in Table 3, take into account the serial computation part and communication between the serial and the parallel parts. The estimated execution time for an overlapped data-parallel model was obtained by evaluating the computation time for each function when it is completely mapped onto the SPE, and assuming that we can overlap the computation and communication. For example, if the execution time for the FD1 stage in an SP model is 13.78us on each of 5 SPEs and 13.76us on the 6th SPE, then the execution time for the FD1 stage in the ODP model become 82.66us (sum of all execution times). Thus different SPEs will execute the complete B-U functionality on different line segment triples concurrently, and in this model we do not have any SPE–SPE communication. The summarized result for predicting the performance of ODP is shown in Column 4 of Table 3.

Next the approximate execution time for an overlapped series-parallel model can be obtained from a series-parallel model using the approach shown with an example in Fig. 11. $P_{1(2)}$ represents the execution time of the second iteration of Task 1 on PPE. $S_{1(2)}$ represents the execution time of the second iteration of Task 1 on an SPE. $C_{1(2)}$ represents the time required for communication by Task 1 in Iteration 2. In Fig. 11a we show an example series-parallel graph with execution time for different tasks. Although in Fig. 11a and b we show only one $S_{1(1)}$ task it represents many concurrently executing SPE threads having the same code. The following factors are taken into account to modify a SP model into a OSP model: (1) two iterations are represented so that execution of the first iteration and second can be overlapped, (2) at any stage shown in the figure the independent tasks (that do not share any resources) can execute simultaneously (for example in Stage 2, fetching data for the second iteration $C_{1(2)}$ happens concurrently with execution of the first iteration in SPE $S_{1(1)}$), and (3) the execution time at any stage is equal to the execution time of the slowest task (e.g., the Stage 3 execution time is dominated by $S_{1(2)}$). Based on the above conditions the execution time for two iterations can be calculated by adding the execution time from each stage as shown in Fig. 11b.

Table 3 Actual and estimated performance of B-U engine on Desktop PC and PS3 (with a single CELL processor)

Function name	Serial model (μs)	Series-parallel (SP) model (μs)	Overlapped data-parallel (ODP) model (μs)	Overlapped functional parallel (OFP) model (μs)	Stage	Overlapped series-parallel (OSP) model (μs)
(A) Input stage	0.89	0.80	0.80	0.80	1	0.80
(B) FD1 stage	603.53	13.78	82.66	82.66	2	13.78
(C) Partial to global histogram	0.00	2.71	0.00	0.00	3	13.78
(D) Partial midVector(kWTA)	13.34	2.48	3.87	3.87	4	2.71
(E) Partial to global midVector	0.00	1.91	0.00	0.00	5	2.48
(F) FD2 stage	1276.68	12.91	77.23	77.23	6	12.91
(G) Dump output	0.60	0.75	0.75	0.75	7 8	12.91 0.75
Performance evaluation	Actual	Actual	Estimated	Estimated		Estimated
Effective time per LST	1895.04	35.34	27.55	41.33		30.06
Speedup w.r.t serial	1.00	53.63	68.78	45.85		63.04
Approx. data memory req. (per SPE)	-NA-	220KB	1.270M	1.1M		230KB

The serial model corresponds to the performance of the B-U engine on a Desktop PC running 2.13 GHz Intel Core2 (E6400) CPU. Column 3 shows the actual performance of series-parallel model on one PS3. A speedup of 54x was possible using the series-parallel model on PS3. The performance of the overlapped series-parallel model is indicated in the last column, and the stage column is obtained by using the approach shown in Fig. 11b. As listed in the above table, the overlapped data-parallel model can give the best performance, but at the cost of a large data memory. All units of time in the table is in μs

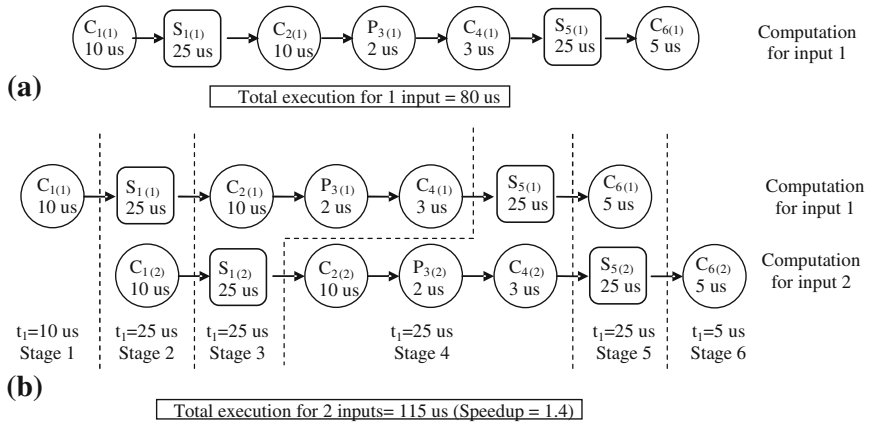

Fig. 11 An illustrated example of the performance of (**a**) simple series-parallel model (**b**) overlapped series-parallel model. In the OSP model the SPE starts the computation for the second input data before proceeding to the next stage. $P_1(2)$ ($S_1(2)$) indicates the execution time for the 2nd iteration of task 1 on PPE (SPE), $C_1(2)$ indicate the communication time for data produced by task 1 in its 2nd iteration

The OFP model or a function pipeline model is obtained by mapping each task to either PPE or SPE. The Input Stage is mapped to PPE as the task is not compute intensive; the remaining functions are mapped to SPEs (FD1 Stage on SPE0, kWTA/Generate Partial MidVector on SPE1, and FD2 Stage on SPE2). Six SPEs can execute two overlapped functional parallel models with each model implementing the B-U computation on different line segment triples. Hence the total execution time for the B-U computation on two line segment triples in an OFP model is equal to the execution time of the slowest task in the pipeline (namely FD1 Stage = 82.66us). Table 3 also contains the information regarding the approximate amount of data memory required in each SPE LS for implementing different parallel models. In the table we also show that the estimated performance of different models. The overlapped data-parallel model gives the maximum speedup (about 68×) but requires more memory than other approaches. Our future work is to experimentally validate the performance of other kinds of parallel models on IBM CELL processor.

4.5 Code Optimization

We examine specific code optimizations applied on the B-U computation that exploit various architectural features of CELL. Additional types of programmer optimizations are discussed in IBM CELL Handbook [30]. Table 4 shows the speedup achieved by each of the optimization discussed below.

4.5.1 DMA Alignment Optimization

DMA operations in CELL can have a size of 1, 2, 4, 8, 16 bytes or multiples of 16 bytes. If a particular transactions address crosses the 128 byte boundary, additional DMA transactions are necessary to fetch the required data. Hence by means of careful alignment of data structures that are communicated regularly, the overall

Table 4 Different optimization experiments on IBM CELL using the B-U engine and its corresponding performance

Optimizations	1	2	3	4
Aligned DMA	×	×	×	×
Signals mechanism	×		×	×
Mailbox mechanism		×		
SIMD optimization			×	×
Loop-unrolling				×
Results				
Speed (μs per LST)	826	1049	63.6	34.9
Speed-up	1.0	0.79	13.0	23.6
SPE code size (KB)	18.0	17.4	13.4	25
SPE data size (KB)	211	211	229	229

Each X mark in the table indicates the optimization included in specific case. Case 1 and 2 shows that the signal mechanism is better than mailbox mechanism for simple synchronizations. SIMD optimization achieves maximum speedup for our application

communication bandwidth required by the application can be reduced significantly. If DMA alignment optimizations are carried out on too many data sets, the required SPE LS memory increases significantly. For our application, DMA optimization did not show significant improvement in performance as the application was not limited by on-chip or off-chip bandwidth.

4.5.2 Mailbox Versus Signaling Mechanism Optimization

CELL processor allows synchronization by means of regular DMA operations, mailboxes and signaling mechanisms [30]. The mailbox mechanism allows 32-bit communication, but takes about 0.5us for each 32-bit transaction (including the function call overhead). The signal communication mechanism allows 1-bit synchronization between the SPE and PPE at a much higher speed thus allowing faster synchronization. We experimented with both these approaches and the results are shown in Table 4. Case 1 and 2 corresponds to the performance of signaling and mailbox mechanisms, respectively. For our application the signaling mechanism gave better performance then mailbox approach.

4.5.3 SIMD Optimization

The performance of compute intensive kernels in the BDV code (like dot-product and population counting) can be significantly improved by using CELL SIMD instructions. After optimizing the kernels using 128-bit SIMD instructions such as *absdiff, abs_sumb, spu_add, spu_cntb* [35], the inner loop computation in the FD1 task takes 31.2 cycles on an average (the desktop PC version takes 164 cycles), and the inner loop computation in FD2 task takes about 246 cycles (the desktop PC version takes about 2879 cycles). The speedup of more than 13× (see Case 3 in Table 4) is mainly due to the 128-bit SIMD instructions that computes eight 16-bit operations concurrently, and

spu_cntb instruction (useful for counting number of ones within a byte). The original and the optimized code for dot product computation of very large length bit vector are shown in the listing below. The optimized version shows the usage of CELL SIMD C/C++ intrinsics.

```
// Original Code
int popCountFunc(uint64_t num)  {
        int count = popTable[num&0xFF] + popTable[(num>>8)&0xFF]  +  .....  +
popTable[(num>>54)&0xFF];
        return count;
}
#define SPU_FD2_SIZE        1000
#define SPU_FD2_ELE_LEN     8448/sizeof(uint64_t)
inline void ComputeFD2DotProduct()
{
    for (i = 0; i  < SPU_FD2_SIZE; i++)   {
        popCount = 0;
        for (j = 0; j < SPU_FD2_ELE_LEN; j++) {
           popCount += popCountFunc((FD2LSH[i][j] & FD1BitMatchVec[j]));
        }
        FD2MatchCount[i]  = popCount;
    }
}
```

```
// Optimized code with CELL SIMD C/C++ Intrinsics
const vector unsigned short ZeroVec = {0,0,0,0,0,0,0,0};
inline void ComputeFD2DotProduct(vector unsigned char FD1BitMatchVec[])
{
  vector unsigned short ushortTempSum1, ushortTempSum2, ushortTempSum3;
  // Three different counts are used to avoid overflows in the individual bytes
  // (although data is in bytes, sum is done in terms of shorts for efficiency
  for ( i = 0; i < SPU_FD2_SIZE; i++) {
   ushortTempSum1 = ZeroVec;
   ushortTempSum2 = ZeroVec;
   ushortTempSum3 = ZeroVec;
   #define SPU_FD2_SIZE        1000
   #define SPU_FD2_ELE_LEN     8448/sizeof(vector char)
   for ( j = 0; j < SPU_FD2_ELE_LEN/3; j++) {
       ushortTempSum1 = spu_add(ushortTempSum1, (vector unsigned short)
                spu_cntb(spu_and(FD1BitMatchVec[j], FD2LSH[i][j])));
       ushortTempSum2 = spu_add(ushortTempSum2, (vector unsigned short)
                spu_cntb(spu_and(FD1BitMatchVec[j+1], FD2LSH[i][j+1])));
       ushortTempSum3 = spu_add(ushortTempSum3, (vector unsigned short)
                spu_cntb(spu_and(FD1BitMatchVec[j+2], FD2LSH[i][j+2])));
   }
   ushortTempSum1 = spu_sumb((vector unsigned char)ushortTempSum1,
                             (vector unsigned char)ushortTempSum2);
   ushortTempSum2 = spu_sumb((vector unsigned char)ushortTempSum3,
                             (vector unsigned char)ZeroVec);
   FD2MatchCount[i] = SumOfShortInts(spu_add(ushortTempSum1, ushortTempSum2));
  }
}
```

4.5.4 Loop Optimization

The SPE does not have a branch prediction unit, and assumes a sequential program flow. Thus pipeline stalls due to mispredicted branches can be very costly (on the order of 18 to 19 cycles). Various techniques such as function inlining, loop-unrolling and branch hinting mechanisms were used to reduce the branch misprediction penalty. All these techniques increases ILP and in turn increases the usage of dual-issues pipeline within the SPE. A speedup of 2× was achieved over other techniques by using loop optimizations (see Case 4 in Table 4).

5 Results

In our first set of experiments we evaluated the object recognition performance of the BDV algorithm on a difficult problem in computer vision called shape based recognition. In this problem class the contiguous textures pieces of an object are not sufficient to perform recognition using traditional algorithms based on rectangular templates or key-point features like SIFT [36]. Especially for wiry man-made objects such as chairs, tables, and lamps, the conventional approach fails because of lack of appreciable texture area to calculate the key features and identify the objects. To address this difficult area of computer vision, we used videos from the Wiry Object Recognition Database and compared our engine with the precision of the only other system reporting results on this dataset; namely the aggregation sensitive version of the cascade edge operator probes (EOP-AS) [37]. In this dataset, the task is to find a sitting-stool in various cluttered office scenes. The BDV model (8,448 FD1 vectors, 1020 FD2 vectors) was trained with first frames of Videos 0 and 2 (Room A401 Clips 5 and 6) and tested on 30 frames of video from different rooms (Room A408 Clip 4 and Room sh201 Clips 1 and 3). Figure 12a and b show a sample screenshot of the stool recognition process during different iterations of the BDV algorithm. For comparison, results using the EOP-AS were estimated from the reported accuracy on "other room" test sets of 22% (8.8 false positives per image) used as the probability of an arbitrary true positive guess. In Fig. 13 the y-axis is the probability of finding the stool within the number of guesses defined by the x-axis. We can see that the neocortical based BDV model achieves similar performance to EOP-AS. As the system increases the number of guesses they make per image from 1 to 5, the probability of finding the sitting-stool in the image increases from 20 to 70%.

We evaluated the performance of the B-U computation on different architectures (FPGA, Intel PC, clusters, etc.), and the summary of the result is shown in Table 5. A detailed description of mapping B-U computation on FPGA is presented in [6]. The Xilinx Virtex4 family of FPGA having about 9.6 Mbit of BlockRAM memory, was used for mapping the algorithm. Almost all of the memory was utilized for storing the FD2 and FD1 tables, and only 61% of logic was used. It was shown in [6] that a single Xilinx Virtex4 FPGA provided a 62× performance improvement.

For obtaining the performance on IBM CELL we mapped the application on a single PS3, and a small cluster of PS3s (having 1–3 nodes). As shown in Table 5,

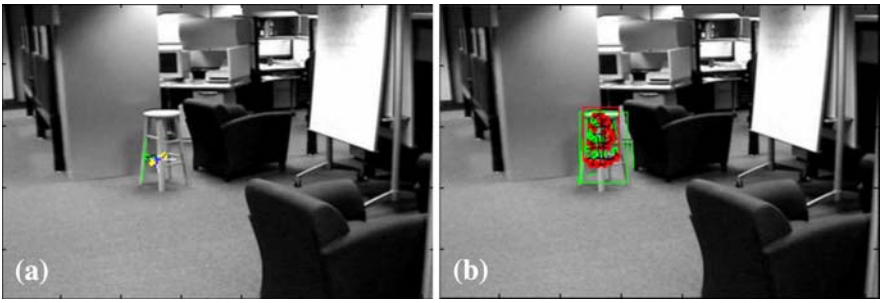

Fig. 12 Screenshots of different iterations of the BDV algorithm running to recognize a stool in a scene from the WORD video database [37]. The iterations proceeds based on the feedback from the B-U engine

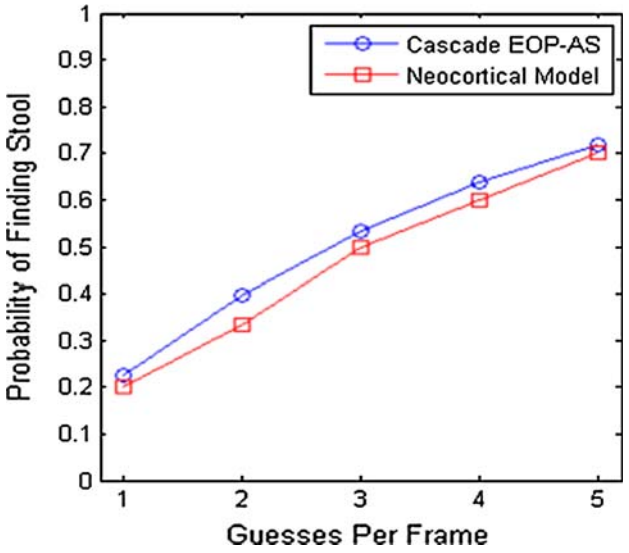

Fig. 13 Comparison of object recognition performance of BDV model and EOP-AS [37] on WORD database

Table 5 Summary of performance of different platforms for simulation the BDV algorithm

Device	Cycles	Freq. (MHz)	Time (μs)	Speedup	Cost ($)	Speedup/ $1K	Approx power(W)
Intel Core2 Duo	5423430	2930	1895	1.00x	2000	0.500	65.0
Xilinx FPGA	2242	76.00	29.50	64.2x	10000	6.27	1.63
IBM CELL (PS3)	113088	3200	35.34	53.62	500	107.24	100.0

the PS3 version of IBM CELL was able to achieve a speedup of 53.62× compared to the desktop PC implementation. After eliminating some of the bottlenecks associated with Gigabit Ethernet communication (which requires all PPE cycles to achieve 500 mbps) we scaled the implementation to a cluster of three PS3 and achieved a full round trip time of 1.24 seconds on a video frame with 93,267 line triples

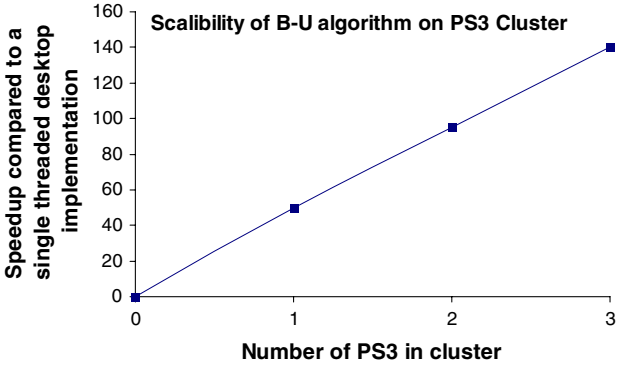

Fig. 14 Scalability of PS3 cluster. With three PS3s connected by mean of a Gigabit Ethernet Router a speed up 140× was achieved by running the series-parallel model

achieving a speedup over the desktop of 140×. The performance on a cluster of PS3 is shown in Fig. 14. Built from base model PS3s, and using two built-in Gigabit network interfaces and one PCI Gigabit network card, the cost of the cluster hardware was approximately $1500. Thus the speedup of approximately 140 times was achieved at a cost similar, or cheaper than a top-of-the-line desktop machine, enabled by the intrinsically parallel nature of the derived brain algorithms. When compared to an FPGA implementation, IBM CELL on PS3 offers similar performance, easier scalability, and lower cost for simulation of B-U computation. Easy scalability with PS3 enabled the brain derived vision algorithm to support much larger reference tables, and the ability to work on different parts of the input frame for object recognition.

6 Conclusion

Standard computer and engineering approaches remain uncompetitive with humans at most natural tasks (such as visual and auditory recognition); moreover, standard approaches are typically customized for each task and do not transfer readily to new tasks nor scale well to large size. In contrast, brains are intrinsically parallel, operating billions of neurons and trillions of low-precision synapses concurrently so as to flexibly learn and perform tasks that are so complex that we might not otherwise know them to be computable. Given these abilities of human brains, it is pragmatic to study and attempt to imitate algorithms used by brain circuits. As shown here and elsewhere, analysis of brain circuit properties have already demonstrated initial promising results in processing tasks of known difficulty.

In this article we have shown that implementing a brain-derived method on parallel hardware can give rise to powerful speedup; the current results suggest the ability to achieve visual recognition in less than one second, i.e., in real time. Such findings may enable a range of applications in sensors, control systems, and robotics that are currently beyond the scope of standard approaches. The results reported herein and previously [6,7] demonstrate that alternate architectures can be constructed to provide

large-scale parallelism, and may serve as platforms for research in currently recalcitrant computational tasks requiring real-time processing of real-world information, such as sensor processing, and perception.

The findings indicate that these new approaches lend themselves to direct hardware implementation, and scale well on complex tasks. As engineering fields increasingly recognize the importance of parallel system design [1,2], the study of inherently parallel computational systems such as the brain may become of increasing theoretical and practical utility.

Acknowledgments The research described herein was supported in part by the Office of Naval Research and the Defense Advanced Research Projects Agency. The authors would like to thank Jeff Furlong for his FPGA experimental results. We also thank the reviewers Jeff Krichmar, Sudeep Pasricha for their detailed comments and insights.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Carruthers, J., Hammerstrom, D., Colwell, B., Bourianoff, G., Zhirnov, V.: ITRS emerging research architecture working group white paper. http://www.itrs.net/Links/2005ITRS/Home2005.htm. Accessed April 2009 (2005)
2. NAE: Grand challenges for engineering. http://www.engineeringchallenges.org/. Accessed April 2009 (2008)
3. Granger, R.: Brain circuit implementation: high-precision computations from low-precision components. In: Toward Replacement Parts for the Brain, pp. 277–294. MIT Press, Cambridge (2004)
4. Itti, L., Koch, C.: Computational modelling of visual attention. Nat. Rev. Neurosci. **2**, 194–203 (2001)
5. Granger, R.: Engines of the brain: the computational instruction set of human cognition. AI Mag. **27**, 15–32 (2006)
6. Furlong, J., Felch, A., Jayram, M.N., Dutt, N., Nicolau, A., Veidenbaum, A., Chandrashekar, A., Granger, R.: Novel brain-derived algorithms scale linearly with number of processing elements. In: Proceedings of ParaFPGA Conference: Parallel Computing with FPGAs (2007)
7. Felch, A., Jayram, M.N., Chandrashekar, A., Furlong, J., Dutt, N., Granger, R., Nicolau, A., Veidenbaum, A.: Accelerating brain circuit simulations of object recognition with cell processors. In: International Workshop on Innovative Architecture for Future Generation Processors and Systems, pp. 33–42 (2007)
8. Granger, R., Hearn, R.: Models of thalamocortical system. Scholarpedia **2**(11), 1796 (2007)
9. Izhikevich, E.M., Edelman, G.M.: Large-scale model of mammalian thalamocortical systems. Proc. Natl. Acad. Sci. **105**, 3593–3598 (2008)
10. Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M., Poggio, T.: Robust object recognition with cortex-like mechanisms. IEEE Trans. Pattern Anal. Mach. Intell. **29**, 411–426 (2007)
11. Krichmar, J., Seth, A., Nitz, D., Fleischer, J., Edelman, G.: Spatial navigation and causal analysis in a brain-based device modeling cortical-hippocampal interactions. NeuroInformatics **3**, 197–221 (2005)
12. Rodriguez, A., Whitson, J., Granger, R.: Derivation and analysis of basic computational operations of thalamocortical circuits. J. Cogn. Neurosci. **16**, 856–877 (2004)
13. Lynch, G., Granger, R.: Big Brain: The Origins and Future of Human Intelligence, 1st edn. Palgrave Macmillan, New York (2008)
14. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS Conference Proceedings (1967)
15. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the CELL multiprocessor. IBM J. Res. Develop. **49**, 589–604 (2005)
16. Gschwind, M., Hofstee, H., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic processing in cell's multicore architecture. IEEE Micro **26**, 10–24 (2006)

17. Brain derived vision on IBM CELL: http://sourceforge.net/projects/bdv-cell. Accessed April 2009 (2007)
18. Mead, C.: Analog VLSI and Neural Systems, 1st edn. Addison Wesley Publishing Company, USA (1989)
19. Olshausen, B.A., Field, D.J.: How close are we to understanding v1? Neural Comput. **17**(8), 1665–1699 (2005)
20. Hubel, D.H.: Eye, Brain, and Vision. W.H. Freeman, USA (1995)
21. Lee, H., Chaitanya, E., Ng, A.Y.: Sparse deep belief net model for visual area v2. In: Advances in Neural Information Processing Systems, vol. 20. MIT Press, Cambridge (2007)
22. Felch, A.C.: From synapse to psychology: emergence of a language, speech, and vision engine from bottom-up brain modeling. Ph.D. dissertation, University of California, Irvine (2006)
23. Foldiak, P., Endres, D.: Sparse coding. Scholarpedia **3**(1), 2984 (2008)
24. Marr, D., Nishihara, H.K.: Representation and recognition of the spatial organization of three-dimensional shapes. Proc. R. Soc. Lond. B **200**, 269–294 (1978)
25. Coultrip, R., Granger, R., Lynch, G.: A cortical model of winner-take-all competition via lateral inhibition. Neural Netw. **5**, 47–54 (1992)
26. Stringer, S.M., Rolls, E.T.: Invariant object recognition in the visual system with novel views of 3d objects. Neural Comput. **14**(11), 2585–2596 (2002)
27. Riesenhuber, M., Poggio, T.: Hierarchical models of object recognition in cortex. Nature Neurosci. **2**(11), 1019–1025 (1999)
28. George, D., Hawkins, J.: A hierarchical bayesian model of invariant pattern recognition in the visual cortex. In: IJCNN '05: Proceedings of IEEE International Joint Conference on Neural Networks, vol. 3, pp. 1812–1817 (2005)
29. Oram, A., Wilson, G.: Beautiful Code: Leading Programmers Explain How They Think. O'Reilly Media, Inc., USA (2007)
30. IBM: CELL broadband engine programming handbook—version 1.1 (2005)
31. Eichenberger, A.E., O'Brien, K., O'Brien, K., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.: Optimizing compiler for the cell processor. In: PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, pp. 161–172. IEEE Computer Society, Washington, DC (2005)
32. Thorpe, S., Fize, D., Marlot, C.: Speed of processing in the human visual system. Nature **381**, 520–522 (1996)
33. Mattson, T., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming, 1st edn. Addison-Wesley, USA (2004)
34. Lee, E.A., Parks, T.M.: Dataflow Process Networks, pp. 59–85. Kluwer Academic Publishers, Boston/Dordrecht/London (2002)
35. IBM: Language extension in C/C++ for cell—version 2.5 (2008)
36. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vision **60**(2), 91–110 (2004)
37. Carmichael, O.: Discriminative techniques for the recognition of complex-shaped objects. Ph.D. dissertation, The Robotics Institute, Carnegie Mellon University (2003)