Performance Optimization for Pattern Recognition Using Associative Neural Memory

Qing Wu[1], Prakash Mukre[1], Richard Linderman[2], Tom Renz[2], Daniel Burns[2], Michael Moore[3], Qinru Qiu[1]

[1]Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902
[2]Air Force Research Laboratory, Rome Site, 26 Electronic Parkway, Rome, NY 13441
[3]ITT Advanced Engineering & Sciences, 775 Daedalian Drive, Rome, NY 13441

Abstract – In this paper, we present our work in the implementation and performance optimization of the recall operation of the Brain-State-in-a-Box (BSB) model on the Cell Broadband Engine processor. We have applied optimization techniques on different parts of the algorithm to improve the overall computing and communication performance of the BSB recall algorithm. Runtime measurements show that, we have been able to achieve about 70% of the theoretical peak performance of the processor.

1 Introduction

Modeling and simulation of human cognizance functions involve large scale mathematical models, which demand high performance computing platform. We need a novel computing architecture which meets the computational capacity and communication bandwidth of a large scale associative neural memory model.

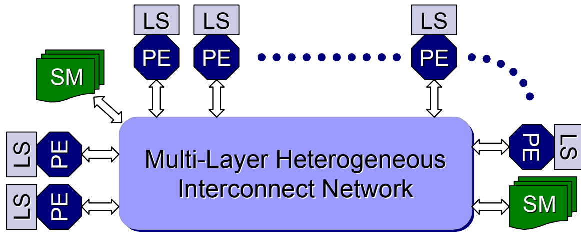

Figure 1. High performance cognitive computing system.

Figure 1 shows the targeting architecture of a high performance cognitive computing system. This system consists of multiple Processing Elements (PEs) interconnected with multi-layer heterogeneous interconnect network. All PEs have access to a local memory called Local Store (LS) and Shared Memory (SM) connected to the interconnect network.

In this paper we describe the performance optimization in software and hardware solutions for a cognitive computing model called *Brain State in a Box (BSB)*. This BSB model is implemented using the Cell Broadband Engine.

1.1 Brain State in a Box (BSB) Model

BSB model is a simple, auto-associative, nonlinear, energy-minimizing neural network [1][2]. A common application of the BSB model is to recognize a pattern from a given noisy version. BSB model can also be used as a pattern recognizer that employs a smooth nearness measure and generates smooth decision boundaries [3].

There are two main operations in a BSB model, *Training* and *Recall*. In this paper, we will focus on the BSB recall operation. The mathematical model of a BSB ***recall operation*** can be represented in the following form:

$$\mathbf{x}(t+1) = S(\alpha * \mathbf{A} * \mathbf{x}(t) + \lambda * \mathbf{x}(t) + \gamma * \mathbf{x}(0))$$

where:

- **x** is an N dimensional real vector
- **A** is an NxN connection matrix
- **A*x** (t) is a matrix-vector multiplication operation
- α is a scalar constant feedback factor
- λ is an inhibition decay constant
- γ is a nonzero constant if there is a need to maintain the input stimulation
- $S\,()$ is the "squash" function defined as follows:

$$S(y) = \begin{cases} 1 & if \quad y \geq 1 \\ y & if \quad -1 < y < 1 \\ -1 & if \quad y \leq -1 \end{cases}$$

Note that in our implementation, we choose λ to be 1.0 and γ to be 0.0. But they can be easily changed to other values.

1.2 Cell Broadband Engine Architecture

Cell Broadband Engine Architecture [4][5][6] is a novel multi-core architecture designed for high performance computing. The architecture features nine microprocessors on single chip. A Power architecture compliant core called *Power Processing Element* (PPE) and 8 other attached processing cores called *Synergetic Processing Elements* (SPEs) are interconnected by high bandwidth *Element Interconnect Bus* (EIB). This heterogeneous architecture with high performance competing cores is designed for distributed computing.

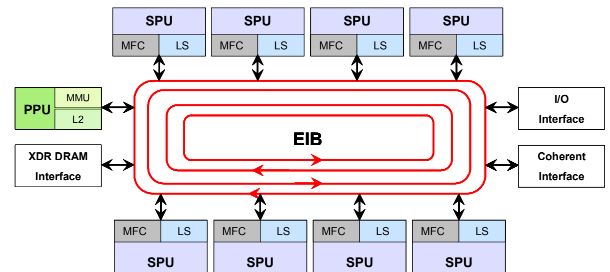

Figure 2. Cell Broadband Engine block diagram.

Rest of the paper is organized as follows: Section 2 discusses the implementation details of BSB recall function on Cell processor; The experimental results and conclusions are presented in Sections 3 and 4.

2 Implementing BSB Recall Operation on Cell Processor

This section describes the implementation and performance optimization of the recall operation on a single cell processor. One of the major challenges in implementing the recall operation in software is the high computation demand. For one 128-dimensional BSB model recall we need a matrix to vector multiplication that involves 16384 floating-point multiplication and 16256 floating-point additions. In addition, we also need 128 floating point multiplications for the feedback factor and 256 comparisons. A large scale associative neural model would involve large number of BSB models (of the order of 100,000). We need a parallel distributed computational model which can perform many BSB recall operations in parallel.

Cell processor with high performance computing cores is an ideal platform for distributed computing. We can run one BSB recall on each SPE. Next, we will describe the implementation details of a 128-dimensional BSB recall operation on one SPE.

2.1 *Matrix-Vector Multiplication*

Multiplication of a 128x128 matrix with 128x1 vector can be represented as follows:

$$\begin{bmatrix} ax_0 \\ ax_1 \\ ax_2 \\ ax_3 \\ .. \\ .. \\ .. \\ ax_{127} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & .. & .. & .. & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & .. & .. & .. & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & .. & .. & .. & a_{2,127} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & .. & .. & . & a_{3,127} \\ .. & .. & .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & . \\ a_{127,0} & a_{127,1} & a_{127,2} & a_{127,3} & .. & .. & .. & a_{127,127} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ .. \\ .. \\ .. \\ x_{127} \end{bmatrix}$$

where,

$$ax_0 = a_{0,0}*x_0 + a_{0,1}*x_1 + a_{0,2}*x_2 + ... + a_{0,127}*x_{127}$$
$$ax_1 = a_{1,0}*x_0 + a_{1,1}*x_1 + a_{1,2}*x_2 + ... + a_{1,127}*x_{127}$$
$$ax_2 = a_{2,0}*x_0 + a_{2,1}*x_1 + a_{2,2}*x_2 + ... + a_{2,127}*x_{127}$$

..

..

$$ax_{127} = a_{127,0}*x_0 + a_{127,1}*x_1 + a_{127,2}*x_2 + ... + a_{127,127}*x_{127}$$

The scalar implementation of the above equations can be written in C as:

```
float ax[128], x[128], a[128*128];
int row,col;

for(row=0;row<128;row++){
   ax[row]=0;
   for(col=0;col<128;col++) {
        ax[row]+=x[col]*a[row*128+col];
   }
}
```

We can improve the performance by using the Single Instruction Multiple Data (SIMD) model of SPE. Most of the instructions in SPEs operate on 16 bytes of data and also the data fetching from local store is 16-byte aligned. To implement the scalar computation using SIMD instructions, complier has to keep track of the relative offsets between the scalar operands to get the correct results. This implementation ends up having additional rotation instructions added, which is an overhead. So, to get better performance and correct result it is wise to handle the data as vectors of 16 bytes (or 4 single-precision floating-point numbers) each.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & .. & .. & .. & a_{0,127} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & .. & .. & .. & a_{1,127} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & .. & .. & .. & a_{2,127} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & .. & .. & . & a_{3,127} \\ .. & .. & .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. & .. & .. & . \\ a_{127,0} & a_{127,1} & a_{127,2} & a_{127,3} & .. & .. & .. & a_{127,127} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ .. \\ .. \\ .. \\ x_{127} \end{bmatrix}$$

The matrix multiplication using SIMD instructions can be implemented as below

```
float ax[128] __attribute__((aligned (128)));
float x[128] __attribute__((aligned (128)));
float a[128*128] __attribute__((aligned (128)));
int row,col;
vector float  *x_v, *a_v;
vector float temp;
x_v = (vector float *)x;
a_v = (vector float *)a;


for(row=0;row<128/4;row++){
   temp = (vector float){0.0,0.0,0.0,0.0};
   for(col=0;col<128/4;col++) {
        temp=spu_madd(x_v[col],a_v[row*32+col],temp);
   }
   ax[row] = (spu_extract(temp,0)+spu_extract(temp,1)+
            spu_extract(temp,2)+spu_extract(temp,3));
}
```

spu_madd and *spu_extract* are intrinsics which make the underlying Instruction Set Architecture (ISA) and SPE hardware accessible from C programming language. *spu_madd* is the C representation for multiply and add instruction. *spu_extract* returns the vector member value specified by the offset. Compared to the scalar implementation, SIMD code reduces 16384 multiplication operations to 1024 vector multiplications. The above implementation still performs scalar addition to get the final result. We can further improve the performance by rearranging the matrix so that we can apply SIMD instructions on all the matrix-vector multiplication operations.

We divide the entire matrix into smaller 4x4 matrices. Elements of each of these 4x4 matrices are shuffled according to a specific pattern as shown below.

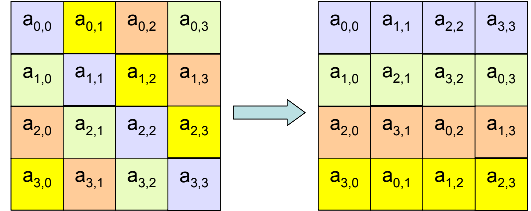

Figure 3. Matrix shuffling from original order to SIMD order.

The shuffled matrix is multiplied with the X vector as shown below. Note that each row of the shuffled matrix is a vector of 4 single precision floating point numbers. And (x_0, x_1, x_2, x_3) is the other vector in the SIMD operation.

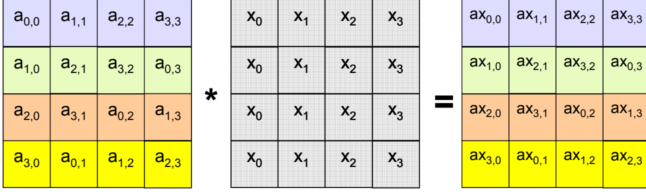

Figure 4. Shuffled matrix multiplication.

To obtain the final result we need to rotate the some of the elements to align them back to their original offset and add all the rows.

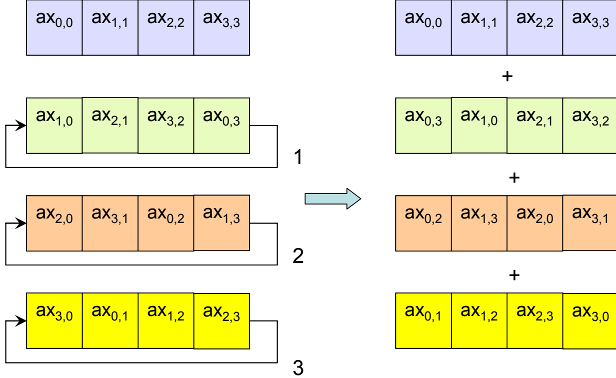

Figure 5. Product alignment and accumulation.

By shuffling the input matrix we have effectively replaced the 3*128 scalar additions from previous implementation with 3*32 rotations and also we reduced the overhead added by the compiler to perform scalar addition.

For every 4 rows of the 128x128 matrix, we will have 32 4x4 matrices. The above shuffle-multiply-rotation-accumulate operation is repeated for each one of them, and at the end we will be able to obtain the 4 elements of the resulting 128x1 vector. For 128 rows, we need to repeat the above procedure 32 times and obtain the complete result. In our current program implemented on PS3, we assume that the 128x128 matrix has already been shuffled before being stored into the main memory. Therefore, the shuffle operations are not needed in the algorithm.

2.2 Other operations in BSB recall

From Equation (1) we know that **A*****x**(t) has to be multiplied with feed back constant α and the result is added with **x**(t). This multiplication and addition can be performed by using *spu_madd* intrinsic, which multiplies given two vectors and adds the result with the third vector. This requires 32 *spu_madd* intrinsics.

The final operation in recall is squash function. As given in Equation (1) this operation needs two comparisons to check whether the result of the previous computation is >1 or <-1. To perform this kind of compare and assign operations on vector data SPE provides special instructions.

spu_cmpgt is an intrinsic which performs element-wise comparison on given two vectors. If an element in first vector is greater than corresponding element in second vector then all the bits of corresponding element in result vector are set to '1' else '0'. This result can be used as a multiplexer select, which when '1' assigns input-1 to the output and when '0' assigns input0 to the output. An intrinsic called *spu_sel* is used to perform the selection. *spu_sel* takes two input vectors and select pattern. For each bit in the 128-bit vector pattern, the corresponding bits from either input vector-0 or input vector-1 is selected.

2.3 Performance Tuning

SPE has no dynamic branch prediction it always predicts the branch to be taken. Branches are detected late in the pipeline at a time where there are already multiple fall-through instructions in flight. Due to its architecture SPE has high branch misprediction penalty of 18 cycles. If we implement the multiplications as in Example 1 or 2, then at every loop entry there will be a branch misprediction. To reduce this branch miss penalty, we unrolled the entire inner loop including squash function. Apart from reducing branch misprediction, loop unrolling reduces dependencies and increases the dual-issue rates.

One of the important factors affecting the performance of SPE is the data transfer time. Due to limited local store size it is not possible to get all the data required for the computation at once. So SPE has to initiate DMA transfers whenever it requires additional data from the main memory, which takes additional time. However we can leverage the communication-computation concurrency provided by the Cell's asynchronous DMA model by performing computation while data for future computation is begin fetched. This is called *double buffering*. Figure 5 shows the computational flow with and without double buffering. In the regular communication model, the weight matrix required for the recall is fetched and 10 recall iterations are performed. Then the DMA request for the weight matrix of next recall is initiated. This induces holes in computational flow reducing the effective throughput. In the double buffered implementation, when the computation of the current recall starts, a DMA request for the weight matrix of next recall is initiated in parallel. The MFC can work in background to fetch the data from the memory. By the time when 10 iterations of current recall are completed, the data for the next recall will be available, so SPU can continue with its computational flow.

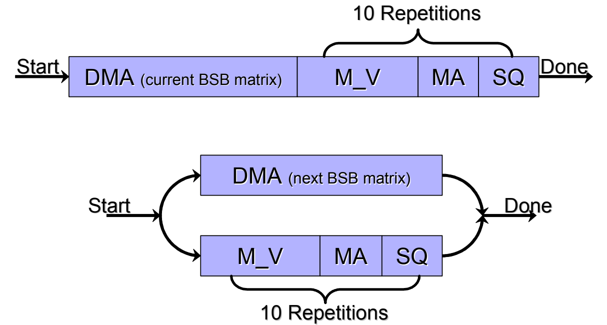

Figure 6. BSB recall computational flow without and with double buffering.

3 Experimental Results

A recall function for 128-neuron BSB model is implemented and tested on PlayStation®-3 (PS3). PS3 consists of one Cell Broadband Engine processor, 256MB XDR DRAM. Only six SPEs are available for the programmer, while the seventh is used by the Sony virtualization software and eighth is disabled. Fedora Core 6 (FC6) version of linux is installed on PS3, as well as the Cell *Software Development Kit* (SDK) 2.1.

We follow *function-offload model* of programming the cell processor. In this model SPEs are used as accelerators of computational intensive parts of the application. Main application

runs on the PPE and it calls selected procedures to run on SPE. Multiple threads can be initiated to call different procedures on different SPEs. This way PPE can take advantage of asynchronous parallelism of SPEs. Both SPE and PPE source are compiled separately by different compilers. PPE passes the input parameters to the procedure running on SPE. Usually these parameters are either address of the data values for computation or synchronization signals.

In our implementation of BSB recall function, PPE sends the starting address of the weight matrices in main memory as parameter to the recall procedure running on SPE. After receiving the matrix address, SPE initiates the DMA transfer of weight matrix to its local store. We use double buffering as described in previous section, so that while the current recall is computed the next weight matrix will be fetched from the main memory through DMA. For testing purposes a random X vector is generated locally by the SPE.

PPE spawns 6 SPE threads once which implements 100,000 iterations of computation. For each iteration, the SPE performs the operation of loading a 128x128 weight matrix from memory and computing 10 BSB recall operations. To find out the computation/communication relationship of the algorithm, we have done different runs from 1 recall per iteration to 15 recalls per iteration. However, we keep the total number of recalls to be 1,000,000. For example, we run 100,000 iterations for 10 recalls per iteration, 200,000 iterations for 5 recalls per iteration. Figure 7 shows the performance in GFLOPS (Giga Floating-point Operations Per Second) per SPE obtained for the complete recall algorithm and Figure 8 shows the GFLOPS per SPE for only Matrix-vector multiplication part of the algorithm.

In both the graphs we can see that performance increases almost linearly to certain point and then saturates. In the linear region computation time is less than the communication time (even with double buffering), so SPE is idle till the weight matrix for the next recall is fetched. Since we use double buffering, the communication time is hidden by the computation time. This is the reason we get saturated performance when we computation time takes over communication time.

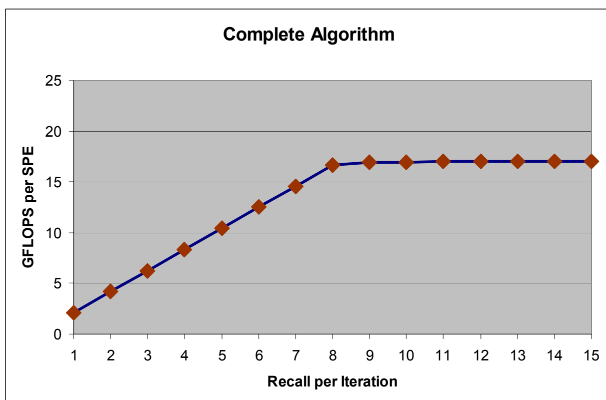

Figure 7. Performance measurement for complete algorithm.

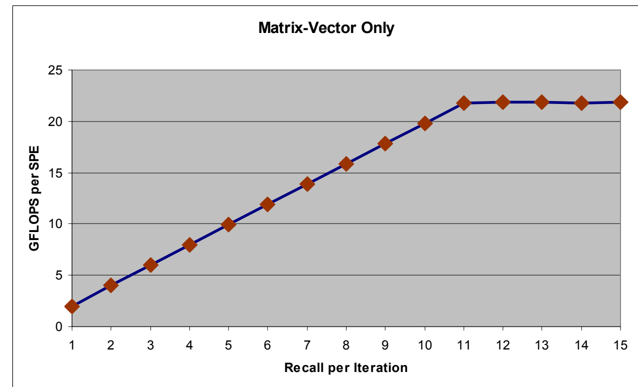

Figure 8. Performance measurement for matrix-vector multiplication operation.

4 Conclusions

Through our work, we have shown that performance optimization of an algorithm does not only limit to improving the computing efficiency, but should also consider the data communication workload of the algorithm. Balancing the computing and communication workload of the algorithm implementation is important to achieving overall optimal system performance.

5 References

[1]. J. A. Anderson, J. W. Silverstein, S. A. Ritz, and R. S. Jones, “Distinctive features, categorical perception, probability learning: Some applications of a neural model,” in *Neurocomputing; Foundations of Research*, J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA: The MIT Press, 1989, ch. 22, pp. 283–325, reprint from *Psychological Review* 1977, vol. 84, pp. 413–451.

[2]. “*Associative Neural Memories: Theory and Implementation*,” Mohamad H. Hassoun, Editor, Oxford University Press, 1993.

[3]. A. Schultz, “Collective recall via the Brain-State-in-a-Box network,” *IEEE Transactions on Neural Networks*, vol. 4, no. 4, pp. 580–587, July 1993.

[4]. Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its first implementation. Technical report, IBM, 2005. http://www.ibm.com/developerworks/power/library/pa-cellperf/

[5]. IBM. Cell Broadband Engine Architecture, http://www-128.ibm.com/developerworks/power/cell/docs documentation.html.

[6]. IBM. Cell Broadband Engine resource center. http://www-128.ibm.com/developerworks/power/cell/.

[7]. Annapolis Microsystems Inc, http://www.annapmicro.com/

[8]. Annapolis Wildstar-II pro data sheet, http://www.annapmicro.com/datasheets/ws2propci_markdata_13364_1_3.pdf

[9]. Xilinx Vertex-II pro devices, http://www.xilinx.com