Neural, Parallel & Scientific Computations 15 (2007) 253-268

Multi-Threaded SIMD Implementation of the Back-Propagation Algorithm on Multi-Core Intel Xeon Processors

Mostafa I. Soliman
Computer & Control Section, Electrical Engineering Department, Faculty of Engineering, South Valley University, Aswan, Arab Republic of Egypt

Abstract
A combinations of using efficient algorithms and well designed implementations leads to great high performance applications. This paper show how to make the back-propagation algorithm run faster on multi-core processors and scale to the future hardware that may have more cores and faster memory. On two dual core Intel Xeon processors, each supports Hyper-Threading technology, the performance of the multi-threaded SIMD implementation of the matrix back-propagation (MBP) algorithm gives around 20 times higher than the best conventional implementation on the same hardware. On reasonably large networks, experimental results show that the use of Intel streaming SIMD extensions, matrix blocking, loop unrolling, and multi-threading on eight logical processors speed up the MBP by factors of 1.4, 1.75, 1.8, 4.6, respectively. Moreover, five single-precision floating-point operations can be performed in a single clock cycle by exploiting the memory hierarchy, by executing multiple instructions from multiple threads on multiple data (MIMD), and by selecting an efficient algorithm, which is based on matrix operations (MBP algorithm) instead of matrix-vector operations (BP algorithm).

Keywords – multi-core computation, multi-threaded implementation, reusing cached data, Streaming SIMD Extensions, back-propagation algorithm, neural computation.

1. INTRODUCTION

The dual-core Intel's Xeon processor contains approximately 0.3 billion transistors running at 3GHz. This processor is over 600 times faster than 8088, which had 29 kilo transistors, (the Intel microprocessor used in the first personal computer soled by IBM) [1]. The dual-core Intel Xeon processor features multi-core, Hyper-Threading (HT) technology and supports multi-processor platforms. This means that a dual-core Intel's Xeon processor provides four logical processors in a physical package (two logical processors for each core due to HT technology). This increasable number of logical processors per physical package would be used to improve the performance of many applications. This paper shows how to exploit increasingly parallel hardware to accelerate the learning of artificial neural networks.

It is known that learning a neural network is computationally expensive. To speedup the learning phase, two nonexclusive approaches have been proposed. The first approach tries to find more efficient optimization algorithm [2, 3]. The other tries to use the parallel processing

Received December 19, 2006; final form May 23, 2007 1061-5369 $15.00 © Dynamic Publishers, Inc.

techniques to reduce the learning time of already existing algorithms [4-7]. In this paper, the matrix back-propagation (MBP) algorithm, which is based on matrix operations (to satisfy the first approach), is selected to be implemented on multi-core Xeon processors (to satisfy the second approach). The target system is Dell Precision 690 running Microsoft Windows Vista operating system. It has two duel-core Xeon processors running at 3GHz, each supports HT technology (i.e., the system has eight logical processors), 2GB memory, and 4MB second level cache (L2). Our results show that on Xeon processors, a good performance of MBP algorithm can be obtained with reasonably large networks by exploiting the multi-core hardware [8], memory hierarchy [9], and Intel Streaming SIMD Extensions (SSE) [10]. Besides, using matrix blocking and loop unrolling [11] techniques are used to further improve the performance.

Obviously, the well designed implementations may not lead to great high performance applications without the selection of efficient algorithms [10]. The MBP algorithm is selected to implement on multi-core Xeon processors because it is highly parallel algorithm based on matrix operations (the operations of choice for parallel processing). By arranging the back-propagation (BP) algorithm, which is based on matrix-vector operations, all the computations of MBP are mainly done through matrix computations. See [12] for more details about BP algorithm and [13, 14] for a complete description of MBP algorithm. This is an important step to switch from matrix-vector operations, which needs $O(n^2)$ floating-point operations on $O(n^2)$ memory operations, to matrix-matrix operations, which needs $O(n^3)$ floating-point operations on $O(n^2)$ memory operations, where the matrix size is $n \times n$. The use of matrix operations would give high performance because of reusing the loaded data into cache memory and keeping the execution pipeline busy with useful work for a long time [15]. These result in decreasing the total execution time of matrix operations and improve many applications based on them.

This paper is organized as follows. The next section describes the matrix back-propagation algorithm. Section 3 presents the best conventional (sequential) implementation of the matrix back-propagation algorithm. It, also, discusses how to exploit the memory hierarchy to improve the performance of the MBP. In Section 3, Intel streaming SIMD extensions are introduced and used as a first step to improve the performance of the MBP. Reusing the cached data, which represents our second step to further improve the performance of MBP, is discussed in Section 4. The result of using matrix blocking and loop unrolling techniques is presented in Section 5. Section 6 introduces multi-threading and multi-core technologies and shows their impacts on the performance of MBP. Finally, Section 7concludes our paper.

2. MATRIX BACK-PROPAGATION

Matrix back-propagation (MBP) is an efficient implementation of the back-propagation (BP) algorithm because it works mainly on matrices to exploit parallel architectures [4]. Consider a network with two fully connected layers, m neurons in the input layer, n hidden neurons and k output neurons, as shown in Figure 1. The connections between the input and output layers are not considered for simplicity. Let $W1_{m \times n}$ is a matrix containing the weights of the first layer and $W2_{n \times k}$ is the weights matrix of the second layer. The corresponding biases are stores in vectors $b1_{1 \times n}$ and $b2_{1 \times k}$, respectively. The learning set consists of p patterns. The input patterns and the target patterns are stored in matrices $S0_{p \times m}$ and $T_{p \times k}$, respectively. Matrices $S1_{p \times n}$ and $S2_{p \times k}$ contain the output of the hidden and output layers when $S0_{p \times m}$ is applied to the input of the network.

All these matrices are aligned and stored in memory in row order, which is proper for programming with C/C++. The order of storing is particularly important for the efficiency of the implementation. Additionally, by aligning frequently used data to the cache line size of a specific processor, the cache performance is improved. Misaligned data access can incur significant performance penalties [16]. For example, in the Xeon processors, the size of a cache line is 64 bytes (16 single-precision floating-point elements). An access to data unaligned on 64-byte boundary leads to two memory accesses and requires several μops to be executed (instead of one).

In the first step of the algorithm, p patterns (stored in $S0_{p\times m}$) are applied from the input layer to the hidden layer. As shown in Figure 1, the net sum ($Net1_{p\times n}$) of the input layer is computed then the status of the hidden layer ($S1_{p\times n}$) is calculated by applying the activation function $f(\ )$. (In Figure 1 and equations, × means matrix-matrix multiplication, and *, +, and – mean element-wise matrix multiplication, matrix addition, and matrix subtraction operations, respectively)

$Net1_{p\times n} = S0_{p\times m} \times W1_{m\times n}$ (2pmn FLOPs for matrix×matrix multiplication)

$Net1_{p\times n} = Net1_{p\times n} + 1_{p\times 1} \times b1_{1\times n}$ (2pn FLOPs for outer-product)

$S1_{p\times n} = f(Net1_{p\times n})$ (hpn FLOPs to apply the approximated $tanh(\ )$)

An h floating-point operations are required for implementing the activation function, $tanh(\ )$, however, its derivative needs only two floating-point operations ($f'(\ ) = 1 - tanh^2(\ )$). To reduce the execution time, the approximated version of the hyperbolic function $tanh(\ )$ is used, which requires only four floating-point operations ($h = 4$) [4].

The same calculations are done for computing the status of the output layer

$Net2_{p\times k} = S1_{p\times n} \times W2_{n\times k}$ (2pnk FLOPs)

$Net2_{p\times k} = Net2_{p\times k} + 1_{p\times 1} \times b2_{1\times k}$ (2pk FLOPs)

$S2_{p\times k} = f(Net2_{p\times k})$ (4pk FLOPs)

Then the error between the network output ($S2_{p\times k}$) and the desired target ($T_{p\times k}$) is back-propagated through the network.

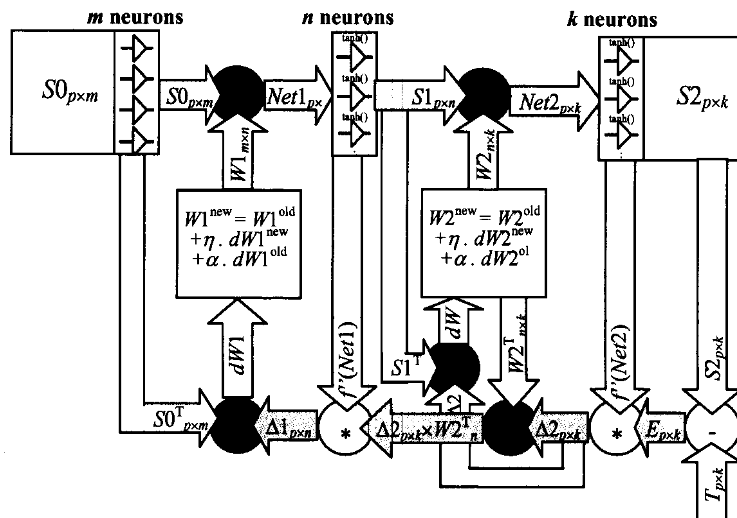

Figure 1: Block diagram of the MBP algorithm

$\Delta 2_{p\times k} = f'(Net2_{p\times k}) * (T_{p\times k} - S2_{p\times k})$ (2pk FLOPs to apply *tanh*'( ) and 2pk for element-wise matrix operations)

$\Delta 1_{p\times n} = f'(Net1_{p\times n}) * (\Delta 2_{p\times k} \times W2^T{}_{n\times k})$ (2pnk FLOPs for matrix×matrix^T multiplication)

The weight variation can be calculated and stored in matrices $dW1^{new}_{m\times n}$ and $dW2^{new}_{n\times k}$ as follows (note that the old values of $dW1_{m\times n}$ and $dW2_{n\times k}$ as well as the new values are needed for updating the weights of the two layers). The same is done for biases.

$dW1^{new}_{m\times n} = S0^T{}_{p\times m} \times \Delta 1_{p\times n}$ (2pmn FLOPs for matrix^T×matrix multiplication)

$db1^{new}_{1\times n} = 1^T{}_{p\times 1} \times \Delta 1_{p\times n}$ (pn FLOPs for accumulating n vectors)

$dW2^{new}_{n\times k} = S1^T{}_{p\times n} \times \Delta 2_{p\times k}$ (2pnk FLOPs for matrix^T×matrix multiplication)

$db2^{new}_{1\times k} = 1^T{}_{p\times 1} \times \Delta 2_{p\times k}$ (pk FLOPs for accumulating k vectors)

Finally, the weights ($W1_{m\times n}$ and $W2_{n\times k}$) and the biases ($b1_{1\times n}$ and $b2_{1\times k}$) are updated. The required operations are based on element-wise multiplication and addition of matrices and vectors.

$W1^{new}_{m\times n} = W1^{old}_{m\times n} + \eta \,.\, dW1^{new}_{m\times n} + \alpha \,.\, dW1^{old}_{m\times n}$ (4mn FLOPs)

$b1^{new}_{1\times n} = b1^{old}_{1\times n} + \eta \,.\, db1^{new}_{1\times n} + \alpha \,.\, db1^{old}_{1\times n}$ (4n FLOPs)

$W2^{new}_{n\times k} = W2^{old}_{n\times k} + \eta \,.\, dW2^{new}_{n\times k} + \alpha \,.\, dW2^{old}_{n\times k}$ (4nk FLOPs)

$b2^{new}_{1\times k} = b2^{old}_{1\times k} + \eta \,.\, db2^{new}_{1\times k} + \alpha \,.\, db2^{old}_{1\times k}$ (4k FLOPs)

From the above equations, the total number of floating-point operations needed to implement the MBP algorithm ($FLOP_{MBP}$) can be calculated as follows.

$$FLOP_{MBP} = 4pmn + 6pnk + 9pn + 9pk + 4mn + 4nk + 4n + 4k$$

It is clear that the MBP algorithm is based mainly on three forms of matrix-matrix multiplications; $C_{n\times n} = A_{n\times n} \times B_{n\times n}$, $C_{n\times n} = A_{n\times n} \times B^T{}_{n\times n}$, and $C_{n\times n} = A^T{}_{n\times n} \times B_{n\times n}$. Assuming $p = m = n = k$, Figure 2 shows the percentage of matrix products FLOPs ($FLOP_{Matrix\text{-}Products}$) over the total number of FLOPs needed for the MBP algorithm ($FLOP_{MBP}$). By applying Amdahl Law [17], improving the performance of matrix products leads to improving the performance of the MBP algorithm because the unparallel code is negligible especially when the matrices are large.

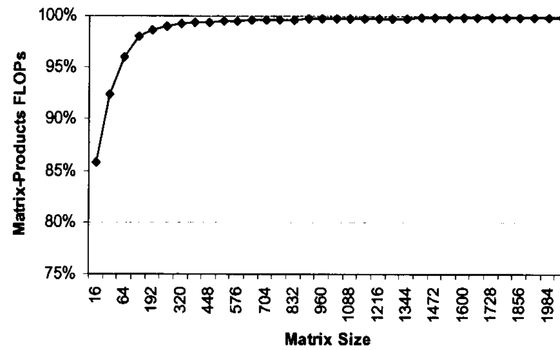

Figure 2: The percentage of $FLOP_{Matrix\text{-}Products}$ over $FLOP_{MBP}$

3. CONVENTIONAL IMPLEMENTATION

The computational load of the matrix back-propagation algorithm belongs to the following equations, as shown in Figure 2.

- $Net1_{p\times n} = S0_{p\times m} \times W1_{m\times n}$ (conventional matrix product)
- $Net2_{p\times k} = S1_{p\times n} \times W2_{n\times k}$ (conventional matrix product)
- $\Delta1_{p\times n} = \Delta2_{p\times k} \times W2^{T}_{n\times k}$ (matrix product with the second matrix transposed)
- $dW1^{new}_{m\times n} = S0^{T}_{p\times m} \times \Delta1_{p\times n}$ (matrix product with the first matrix transposed)
- $dW2^{new}_{n\times k} = S1^{T}_{p\times n} \times \Delta2_{p\times k}$ (matrix product with the first matrix transposed)

To improve the performance of the MBP algorithm, all these three forms of matrix products have to be implemented efficiently.

It is known that there are six variants (*ijk*, *ikj*, *jik*, *jki*, *kij*, and *kji*) to multiply two $n\times n$ matrices due to the triply nested loops (*i*, *j*, and *k*) [11]. Each of these six variants has the same number of FLOPs ($2n^3$), however, their access patterns of memory are different. The following variant called *ijk*, which is the native technique for a matrix product.

```
for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
          for (k = 0; k < n; k++)
               C[i][j] += A[i][k] * B[k][j];
```

By interchanging the order of *i*, *j*, and *k* loops, the remaining five variants can be calculated. Assume that the matrices $A_{n\times n}$, $B_{n\times n}$, and $C_{n\times n}$ are stores in the main memory by rows (as in C/C++ programming language). The naive technique (*ijk* variant) gives a poor performance and we will explain the reason of that in details.

In *ijk* variant, fortunately, each element of $C_{n\times n}$ is accessed *n* times consecutively and its elements are accessed row by row, with a stride of one within each row. The stride of an array refers to the way in which its elements are referenced; it is equal to the difference of the addresses of successive elements over the element size. The access patterns for the elements of $A_{n\times n}$ and $B_{n\times n}$ are different. Accesses to the inner loop elements of matrix $A_{n\times n}$ (`A[i][k]`) are made with a stride of one within the ith row (the rows of $A_{n\times n}$ are visited in order). However, accesses the elements of $B_{n\times n}$ (`B[k][j]`) are made in the kth column a stride of *n* in the row–order layout of $B_{n\times n}$ (the columns of $B_{n\times n}$ are visited in succession).

Consider now the execution of this code on Xeon processor with eight set–associative cache with a line size of 64-byte. The number of consecutive elements of the arrays that will fit into a cache line is thus 16 (= 64/4), where the elements are 4-byte single-precision floating-point numbers. When row order accesses are made with unit stride, in the worst case, the first access to a row element within a cache line will result in a miss, while the remaining accesses within the line (15 accesses) will result in cache hits. In contrast, assuming $n > 16$, accesses to elements with a stride *n* will all result in misses in the worst case.

Assuming that the cache access time on a hit or to detect a miss is t_c and miss service time to be t_m, the access times for $A_{n\times n}$, $B_{n\times n}$, and $C_{n\times n}$ matrices during the execution of the *ijk* variant are $n^2(n{*}t_c + t_m/16)$, $n^3(t_c + t_m)$, and $2n^2(t_c + t_m/16)$ clock cycles, respectively. It results in a total access time of $n^3(2t_c + t_m) + n^2(2t_c + 3t_m/16)$ clock cycles. In contrast, $2n^3$ floating-point operations are needed to implement the MBP algorithm. Assuming two clock cycles are needed for executing a floating-point operation and two floating-point units can operate in parallel, the

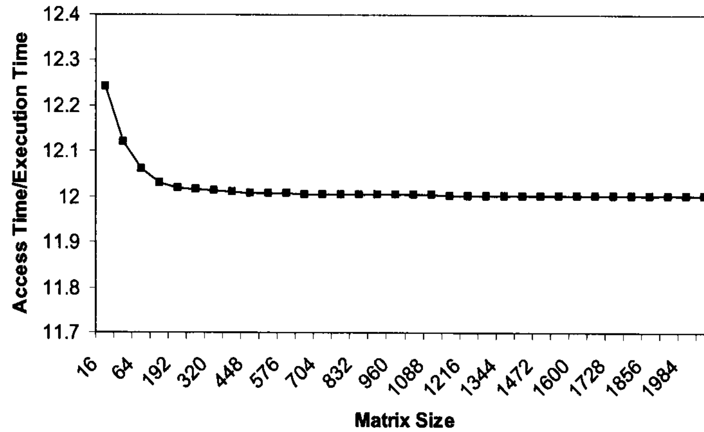

Figure 3: Access time over execution time

total execution time of $2n^3$ FLOPs is $2n^3$ clock cycles. Figure 3 shows the percentage of memory access time over the execution time of floating-point operations, assuming $t_c = 2$ clock cycle, and $t_m = 20$ clock cycles [18]. Note that the data access time is dominated by the miss handling times. As the CPU cycle time decreasing at a faster rate than the cache miss handling time, the execution time will become increasingly dominated by the cache miss handling time. To reduce the number of cache misses, stride memory accesses should be avoided as much as possible.

As shown in Figure 4, the best variant for the conventional matrix product ($C_{n\times n}$ += $A_{n\times n} \times B_{n\times n}$) is *ikj*, where the inner loop is based on SAXPY (Single-precision scalar *A* times vector *X* Plus vector *Y*). It can be gotten by interchange the two inner loop (iterating on *j* and *k*, respectively) of the native technique.

```
for (i = 0; i < n; i++)
      for (k = 0; k < n; k++)
            for (j = 0; j < n; j++)
                  C[i][j] += A[i][k] * B[k][j];
```

As the inner loop (iterating on *j*) executes, the access to the elements of matrix $C_{n\times n}$ (`C[i][j]`) are made with a stride of one within the ith row. Besides, each element of $A_{n\times n}$ (`A[i][k]`) is accessed *n* times consecutively and the elements are accessed row by row, with a unit stride within each row. Moreover, elements of matrix $B_{n\times n}$ (`B[k][j]`) are made with a stride of one within the kth row. This means all elements of the three matrices are accessed with a unit stride, which results in increasing the chance of cache hits (decreasing the number of cache misses).

The following code (*ijk* variant) gives the best performance of the matrix product with the second matrix transposed ($C_{n\times n}$ += $A_{n\times n} \times B^T{}_{n\times n}$), as shown in Figure 5.

```
for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                  C[i][j] += A[i][k] * B[j][k];
```

The inner loop is based on dot-product. Because of the transposition of *B* matrix, all accesses are done with a unit stride in this *ijk* variant.

In case of $C_{n\times n}$ += $A^{T}_{n\times n} \times B_{n\times n}$, the best variant is *ikj* (see Figure 6), where the inner loop is based on SAXPY. Also, all matrices are accessed with a unit stride, as shown in the following code.

```
for (i = 0; i < n; i++)
     for (k = 0; k < n; k++)
          for (j = 0; j < n; j++)
               C[i][j] += A[k][i] * B[k][j];
```

Figure 7 shows the best performance of the conventional MBP algorithm by selecting the best implementation of each matrix product (*ikj* variant for both of the conventional matrix product and matrix product with the first matrix transposed, and the *ikj* variant for the matrix product with the second matrix transposed). The number of clock cycles per floating-point (*CPF*) is selected as a metric for performance evaluation because it is more indicative than the absolute number of clock cycles. Since the number of FLOPs is constant, the total number of clock cycles can be calculated easily.

In the following sections the performance of the conventional MBP algorithm will be improved step by step using Intel Streaming SIMD Extensions to process multiple data using a single instruction, by blocking the matrices to reuse the loaded data in cache, by unrolling loops to reduce the number of load/store operations and reuse the loaded data in registers, and finally by using multi-core to execute multiple threads in parallel.

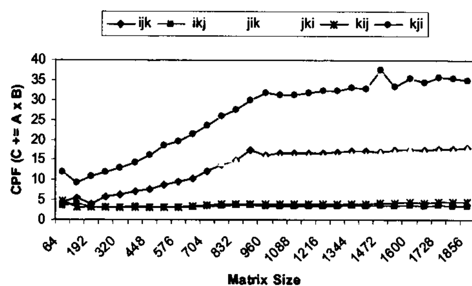

Figure 4: CPF for matrix×matrix multiplication

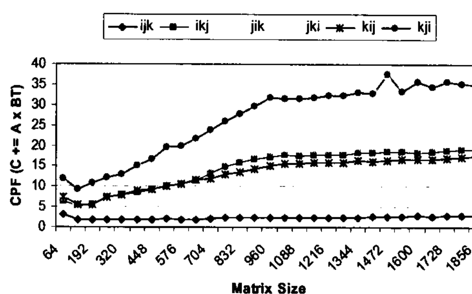

Figure 5: CPF for matrix×matrix^{T} multiplication

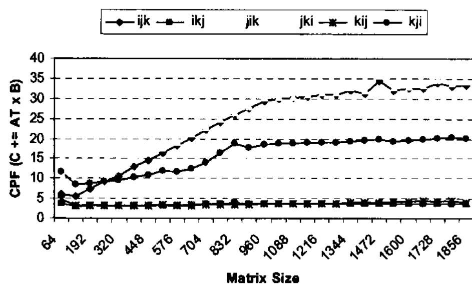

Figure 6: CPF for matrix^{T}×matrix multiplication

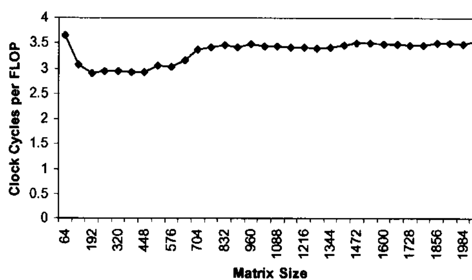

Figure 7: CPF for conventional implementation of MBP

4. INTEL STREAMING SIMD EXTENSIONS

Single instruction, multiple data (SIMD) technology forms an important performance extension to Intel Architecture processors, starting with 64-bit MMX technology of Intel Pentium processor [10]. MMX technology supports 8/16/32/64-bit integer data types. Since then, all 32-bit Intel Architecture and Intel EM64T processors (like Xeon) have extended SIMD technology continuously [1].

A typical SIMD instruction achieves higher performance by operating on multiple data elements at the same time, as shown in Figure 8. Streaming SIMD Extensions (SSE), is a SIMD instruction set designed by Intel and introduced in 1999 in their Pentium III series processors. SSE contains 70 new instructions for processing four packed single-precision floating-point numbers stored in eight new 128-bit registers known as XMM0 through XMM7. Thus, Intel solved the two main problems of MMX: MMX reused existing floating-point registers making the CPU unable to work on both floating-point and SIMD data at the same time, and MMX only worked on integers. Because SSE adds floating-point support, it sees much more use than MMX.

The addition of SSE2's integer support makes SSE even more flexible. While MMX is redundant, operations can be operated in parallel with SSE operations offering further performance increases in some situations. SSE2, introduced with the Pentium 4, is a major enhancement to SSE. SSE2 adds new math instructions for double-precision (64-bit) floating-point and 8/16/32-bit integer data types, all operating on the same 128-bit XMM vector register-file previously introduced with SSE. SSE3 is an incremental upgrade to SSE2, adding a handful of DSP-oriented mathematics instructions and some process (thread) management instructions. SSE3 is introduced with the Pentium 4 with Hyper-Threading technology.

Finally, processors with Intel EM64T like dual-core Xeon extends the SIMD register set to 16 registers (XMM0-XMM15) [1]. Xeon processors support 8/16/32/64/128-bit integer data types and 32/64-bit single/double-precision data types. This section shows that the use of streaming SIMD extensions improve the performance of the MBP algorithm on the same machine with the same hardware.

Table 1 shows the implementation code of matrix products using intrinsics of streaming SIMD instruction set, which are supported by Microsoft Visual Studio 2005. The semantic of each intrinsic used in matrix products are explained in Table 2, where SP and FP stand for single-precision and floating-point, respectively.

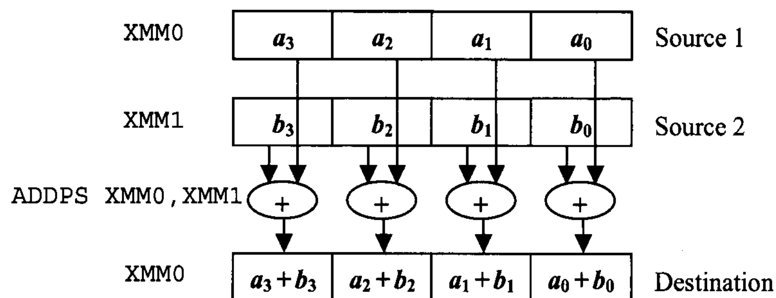

Figure 8: Adding four elements by SIMD

Table 1: The implementation of matrix products using steaming SIMD extensions

$C_{n\times n} \mathrel{+}= A_{n\times n} \times B_{n\times n}$ *ikj* variant

```
int i , j , k ;
__m128 x , y , z , w;
for(i=0;i<n;i++){
for(k=0;k<n;k++){
x=_mm_load1_ps(&A[i][k]);
for(j=0;j<n;j+=4){
y=_mm_load_ps(&B[k][j]);
z=_mm_mul_ps(x,y);
w=_mm_load_ps(&C[i][j]);
w=_mm_add_ps(w,z);
_mm_store_ps(&C[i][j],w);
}
}
}
```

$C_{n\times n} \mathrel{+}= A_{n\times n} \times B^{T}_{n\times n}$ *ikj* variant

```
int i , j , k ;
__m128 x , y , z , w;
for(i=0;i<n;i++){
for(j=0;j<n;j++){
w=_mm_xor_ps(w,w);
for(k=0;k<n;k+=4){
x=_mm_load_ps(&A[i][k]);
y=_mm_load_ps(&B[j][k]);
z=_mm_mul_ps(x,y);
w=_mm_add_ps(w,z);
}
z=_mm_movehl_ps(w,w);
z=_mm_add_ps(z,w);
w=_mm_shuffle_ps(z,z,1);
w=_mm_add_ss(z,w);
_mm_store_ss(&C[i][j],w);}
}
```

$C_{n\times n} \mathrel{+}= A^{T}_{n\times n} \times B_{n\times n}$ *ikj* variant

```
int i , j , k ;
__m128 x , y , z , w;
for(i=0;i<n;i++){
for (k=0;k<n;k++){
x=_mm_load1_ps(&A[k][i]);
for(j=0;j<n;j+=4){
y=_mm_load_ps(&B[k][j]);
z=_mm_mul_ps (x,y);
w=_mm_load_ps(&C[i][j]);
w=_mm_add_ps(w,z);
_mm_store_ps(&C[i][j],w);
}
}
}
```

Table 2: The semantic of intrinsics used in matrix products

Intrinsic	Semantic
`x=_mm_load1_ps(&A[i][k])`	Loads the single SP-FP value `A[i][k]` and copying it into all four elements of `x`.
`y=_mm_load_ps(&B[k][j])`	Loads the four SP-FP values `B[k][j:j+3]` into `y`.
`_mm_store_ps(&C[i][j],w)`	Stores the four SP-FP values from `w` into `C[i][j:j+3]`
`_mm_store_ss(&C[i][j],w)`	Stores the lower SP-FP value from `w` into `C[i][j]`.
`z=_mm_mul_ps(x,y)`	Multiplies the four SP-FP values of `x` and `y` and pass the result to `z`.
`w=_mm_add_ps(w,z)`	Adds the four SP-FP values of `w` and `z` and pass the result to `w`.
`z=_mm_movehl_ps(x,y)`	Moves the upper two SP-FP values of `x` to the lower two SP-FP values of `z` and the upper two SP-FP values of `y` are passed through to the result `z`
`w=_mm_shuffle_ps(x,y,i)`	Selects four specific SP-FP values from `x` and `y`, based on the mask `(i)` and pass them to `w`.

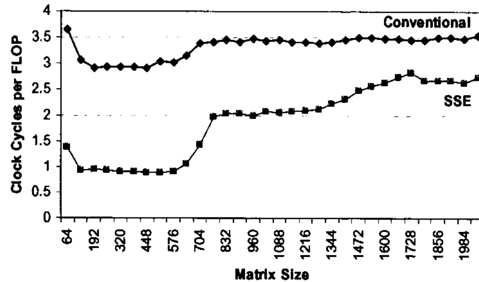

Figure 9: SSE performance of MBP

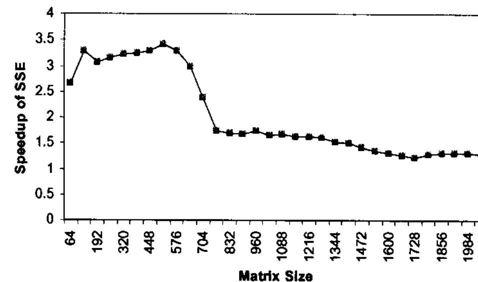

Figure 10: Speedup of using SSE on MBP

Figure 9 shows the enhancement on the performance of the conventional MBP by processing four floating-point numbers by a single SIMD instruction. The number of clock cycles per FLOP (CPF) decreased more than three times when the size of matrices is small (see the left part of Figure 10). However, as the size of matrices is getting larger, the speedup of the SIMD version over the conversional version is decrease from three to 1.3 (see the right part of Figure 10). The main reason of decreasing the performance is that as the matrices getting larger they cannot fit in the cache memory, which results in increasing the cache misses. Moreover, matrix products are implemented as a multiple of vector operations (dot-product or SAXPY operations), which can not efficiently reuse the loaded data into cache memory.

To further improve the performance of MBP, the loaded data in cache memory should be reused many times before replacing them. The matrix blocking is the technique for reusing the loaded data by processing blocks of matrices instead of processing strips of vectors using the strip mining technique.

5. REUSING THE CACHED LOADED DATA

In general, the use of unit stride and memory alignment reduce the cache memory line miss and is possibly to fully exploit the bandwidth between cache memory and processor, which results in improving the performance of an application [16]. However, not all the data is in the cache memory. In fact, initially the data is in the main memory and it takes a long time to be loaded into cache memory because of cache miss handling time.

Implementing a matrix product based on vector operations like dot-product and SAXPY is an inefficient technique because $O(n)$ memory operations are needed for processing $O(n)$ floating-point elements. Two strips of vector are loaded from the main memory to the cache and then discarded from the cache because other strips of vectors replace them. The technique to avoid such behavior is the matrix blocking. Matrices are partitioned into block. The block size depends on the size of the cache memory. Thus, all calculations are done based on matrix operations instead of vector operations. On other words, the loaded $b \times b$ blocks are reused b times before leaving the cache memory in the worst case. The parameter b should be large enough to avoid many load/store operations but small enough to fit the required blocks in the cache memory.

Table 3 shows the SIMD version of the conventional matrix product (*ikj* variant) based on 4×4 matrix blocking. The effect of using 4×4 matrix blocking on the performance of MBP is shown in Figure 11. The performance of the MBP algorithm is improved by a factor of 2.5 because of using SIMD instruction set and matrix blocking technique (see Figure 12).

To further improve the performance, the loaded data from cache memory to registers should be reused since the speed of registers is faster than cache memory as well as the register file is closer to the execution units. The reuse of loaded data in register file results in decreasing the number of load/store operations, which are equal or more expensive than arithmetic operations. The loop unrolling is the technique of choice to make a balance between the load/store operations and arithmetic operations in the inner loop.

From Table 1, the inner loop of the conventional matrix product (*ikj* variant) requires three load/store and two floating-point SIMD operations, which results in 1.5 load/store operations per FLOP. This means that the pipeline stalls many times waiting for loading/storing data, which decreases the throughput of the execution pipeline (see [15] for more detail). The situation becomes worser for a processor with multiple executions units. However, the use of loop unrolling technique with 4×4 matrix blocking improves the fraction of load/store operations per FLOP from 1.5 to 1, see Table 4. Moreover, the use of loop unrolling technique reduces the loop overheads. As shown in Table 4, it decreases the number of branch instructions by a factor of 16, when the inner loops (*ii* and *kk*) in Table 3 are unrolled. Reducing the number of branch instructions reduces the probability of flushing the pipeline (see [15] for more detail).

Table 3: The SIMD implementation of *ikj* variant using matrix blocking

```
Int i , j , k ;
__m128 x , y , z , w;
for(i=0;i<n;i+4){
  for(k=0;k<n;k+4){
    for(j=0;j<n;j+=4){
      for(ii=i;ii<i+4;ii++){
        w=_mm_load_ps(&C[ii][jj]);
          for(kk=k;kk<k+4;kk++){
           x=_mm_load1_ps(&A[ii][kk]);
           y=_mm_load_ps(&B[kk][j]);
           z=_mm_mul_ps(x,y);
           w=_mm_add_ps(w,z);
          }
        _mm_store_ps(&C[ii][j],w);}}}}
```

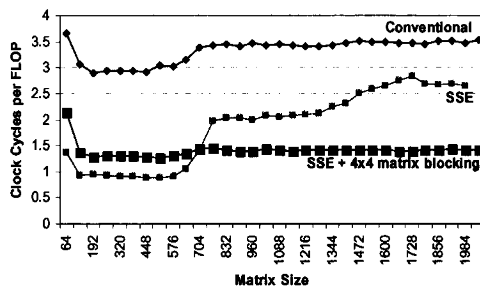

Figure 11: The effect of using matrix blocking

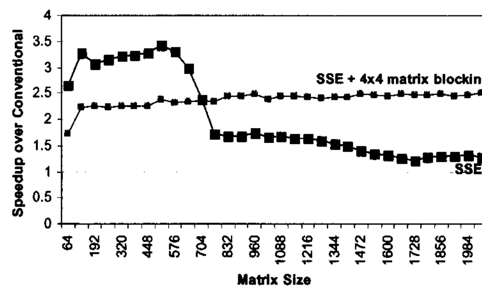

Figure 12: Speedup of using matrix blocking

The performance of the MBP after using the 4×4 matrix blocking with loop unrolling is shown in Figure 13. Its performance is more than four times higher than the conventional implementation (see Figure 14). To further improve the performance 8×8 matrix blocking with loop unrolling is used, which shifts the balance of the inner loop from memory-bound to CPU-bound (0.5 load/store operation per FLOP). The speedup of using SIMD instruction set, 8×8 matrix blocking, and loop unrolling is more than 5.5 over the conventional implementation, as shown in Figure 14.

Table 4: Loop unrolling of the SIMD implementation of ikj Variant using 4×4 blocks

```
for( i = 0 ; i < n ; i += 4){
  for( k = 0 ; k < n ; k += 4){
    for( j = 0 ; j < n ; j += 4 ){
      x=_mm_load_ps(&A[i][k]);              xxxx=_mm_shuffle_ps(x,x,0x00);
      y[0]=_mm_load_ps(&B[k][j]);           z = _mm_mul_ps (xxxx, y[0]);
      w = _mm_load_ps (&C[i][j]);           w = _mm_add_ps (w, z);
      xxxx=_mm_shuffle_ps(x,x,0x55);        y[1]=_mm_load_ps (&B[k+1][j]);
      z = _mm_mul_ps (xxxx, y[1]);          w = _mm_add_ps (w, z);
      xxxx=_mm_shuffle_ps(x,x,0xAA);        y[2]=_mm_load_ps (&B[k+2][j]);
      z = _mm_mul_ps (xxxx, y[2]);          w = _mm_add_ps (w, z);
      xxxx=_mm_shuffle_ps(x,x,0xFF);        y[3]=_mm_load_ps (&B[k+3][j]);
      z = _mm_mul_ps (xxxx, y[3]);          w = _mm_add_ps (w, z);
      _mm_store_ps (&C[i][j] , w);          x = _mm_load_ps (&A[i+1][k]);
      xxxx=_mm_shuffle_ps(x,x,0x00);        z = _mm_mul_ps (xxxx, y[0]);
      w = _mm_load_ps (&C[i+1][j]);         w = _mm_add_ps (w, z);
      :
      :
      _mm_store_ps (&C[i+3][j] , w);}}}
```

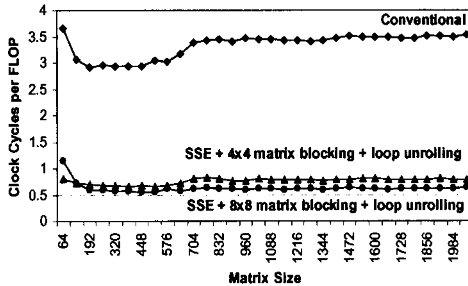

Figure 13: The effect of loop unrolling

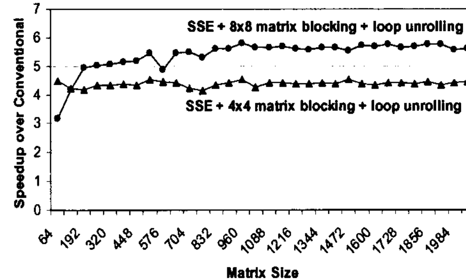

Figure 14: Speedup of using loop unrolling

6. MULTI-THREADING

Superscalar processors rely on instruction-level parallelism for executing multiple instructions simultaneously on multiple execution datapaths [19]. Recently, the situation is changed after announcing the Hyper-Threading (HT) technology in Intel Pentium 4 and Xeon processors [20]. HT technology enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package. Each logical processor has its own architectural state and its advanced programmable interrupt controller (APIC) and shares the core resources of the physical processor (see Figure 15). This includes the execution engine and the system bus interface [1]. Two or more separate code streams (threads) can be executed concurrently using these shared execution resources, which improves the performance of multi-threaded applications or single-threaded applications under multi-tasking environments [16].

The next logical step from simultaneous multi-threading is the multi-core processor [1, 8]. Multi-core technology enhances hardware multi-threading capability by providing two or more execution cores in a physical package. The dual-core Intel Xeon processor features multi-core, Hyper-Threading technology and supports multi-processor platforms. It provides four logical processors in a physical package (two logical processors for each processor core). The two cores share a smart second level cache, which enables efficient data sharing between two cores to reduce memory traffic bus, as shown in Figure 15.

Matrix products (*ikj* variant for conventional matrix product, *ijk* variant for matrix product with the second matrix transposed, and *ikj* variant for matrix product with the first matrix transposed) are easy to implement on multi-core processors using multi-threading technique. The data access pattern of these variants makes matrix products based on these variants extremely easy to parallelize. As shown if Figure 16, all elements of matrix $B_{n\times n}$ should be read by every thread, however, matrices $A_{n\times n}$ and $C_{n\times n}$ are distributed among threads. Since our system has two physical Xeon processors, dual-core each, supports HT technology, eight threads can be executed in parallel on eight logical processors.

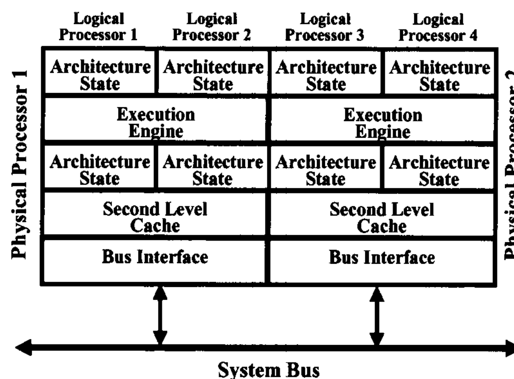

Figure 15: Dual-core Intel Xeon Processor supports HT technology

No synchronization is needed for accessing the elements of $A_{n\times n}$, $B_{n\times n}$ and $C_{n\times n}$ matrices. Each thread reads a block of matrix $A_{n\times n}$ and read/write another block from/to matrix $C_{n\times n}$. (The block sizes are not necessarily the same.) Assuming that n is multiple of eight, thread i works on block form row number (i*n/8) to row number ((i+1)*n/8 -1). Since each thread accesses all elements of matrix $B_{n\times n}$, caching the matrix $B_{n\times n}$ in the second level cache memory supports four logical processors, where two cores share the second level cache. Most importantly, *ikj* variant, which is used for conventional matrix product and for matrix product with the first matrix transposed, is based on SAXPY. SAXPY produces a row of the result matrix $C_{n\times n}$ for each row of $A_{n\times n}$ used. Thus, if a block (set of rows) of $C_{n\times n}$ is assigned to a thread, no synchronization is necessary for the writes to $C_{n\times n}$ from the various threads. The *ijk* variant for matrix product with the second matrix transposed is based on dot-product, which produces an element of the result matrix $C_{n\times n}$ for each row of $A_{n\times n}$ used. Thus, no synchronization also is necessary for the writes to $C_{n\times n}$ from the various threads.

Figure 17 shows the result of implementing the MBP algorithm on eight logical processors using multi-threading technique. One the large networks, the performance of MBP algorithm is around 20 times higher than the best conventional implementation on multi-core processors, as shown in Figure 18. However, on the small networks, the performance of multi-threaded MBP is no so good because of the overhead of creating threads (see [20 , 8] for more detail).

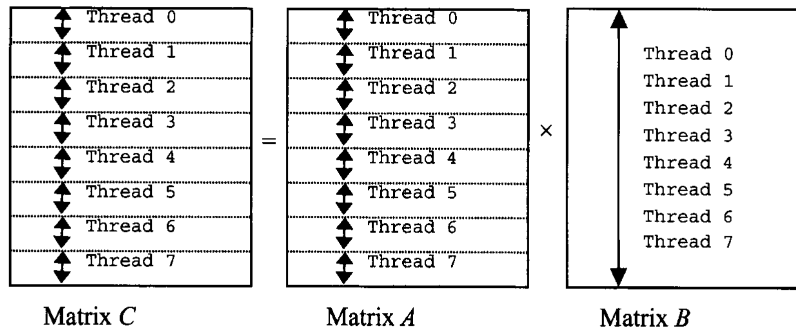

Figure 16: Partitioning matrices among threads

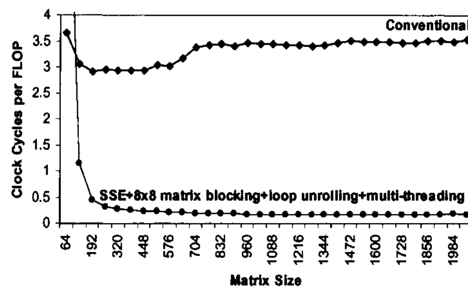

Figure 17: Multi-threaded performance of MBP

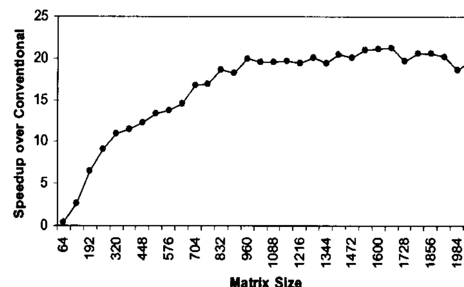

Figure 18: Speedup of multi-threaded MBP

7. CONCLUSION

Selecting the efficient algorithm and exploiting the available features of an architecture would speed up the execution of applications tens of time. This paper exploits memory hierarchy and increasingly logical processors per physical package to accelerate the learning of neural networks, which is computationally intensive. For comparison, the performance of the best conventional (sequential) implementation of the matrix back-propagation algorithm is computed. The use of unit-stride and align memory accesses improves the performance of the sequential/parallel implementation of the matrix back-propagation because of reducing cache misses. The performance of the best conventional implementation of matrix back-propagation algorithm is improved step by step using Intel Streaming SIMD Extensions to process multiple data using a single instruction, by blocking the matrices to reuse the loaded data in cache, by using loop unrolling technique to reduce the number of load/store operations and reuse the loaded data in registers, and finally by using multi-core to execute multiple threads in parallel. On reasonably large networks, the performance of multi-threaded SIMD implementation of the matrix back-propagation algorithm is around 20 times higher than the performance of the best conventional implementation on two dual-core Intel Xeon processors.

REFERENCES

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 1: Basic Architecture, http://www.intel.com/products/processor/manuals/index.htm.

[2] E. Barnard, "Optimization for Training Neural Nets," *IEEE Transaction of Neural Networks*, Vol.3, No. 2, Mar 1992, pp. 232-240.

[3] P. Phua and D. Ming, "Parallel Nonlinear Optimization Techniques for Training Neural Networks," *IEEE Transactions on Neural Networks*, Vol. 14, No. 6, Nov 2003, pp. 1460- 1468.

[4] D. Anguita, G. Parodi, and R. Zunino, "Efficient Implementation of BP on RISC-Based Workstations," *Neurocomputing*, Vol. 6, No. 1, 1994, pp. 57-65.

[5] S. Taraglio and F. Massaioli, "An Efficient Implementation of a Backpropagation Learning Algorithm on A Quadrics Parallel Supercomputer," *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, Lecture Notes In Computer Science; Vol. 919, 1996, pp. 664-671.

[6] E. Kerckhoffs, F. Wedman, and E. Frietman, "Speeding up Backpropagation Training on a Hypercube Computer," *Neurocomputing*, Vol. 4, No. 1, 1992, pp. 43-63.

[7] S. Suresh, S. Omkar, and V. Mani, "Parallel Implementation of Back-Propagation Algorithm in Networks of Workstations," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 16, No. 1, Jan 2005, pp. 24-34.

[8] S. Akhter and J. RobertsIntel, *Multi-Core Programming: Increasing Performance through Software Multithreading*, Intel PRESS, 2006, ISBN 0976483246.

[9] J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, L. Torczon, and W. Gropp, *The Sourcebook of Parallel Computing*, Morgan Kaufmann, November 2002, ISBN 1558608710.

[10] R. Gerber, A. Bik, K. Smith and X. Tian, *The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms*, Second Edition, Intel PRESS, 2006, ISBN 0976483211

[11] G. Golub and C. Van Loan, *Matrix Computations*, 3rd Edition, The Johns Hopkins University Press, Baltimore and London, 1996, ISBN 0801854148.

[12] J. Zurada, *Introduction to Artificial Neural Systems*, New York, PWS Publishing Company, 1992, ISBN 053495460X.

[13] D. Anguita, "MBP-Matrix Back Propagation v1.1: An Efficient Implementation of the BP Algorithm," *University of Genova Report*, November 1993.

[14] D. Anguita and B. Gomes, "Mixing Floating- and Fixed-Point Formats for Neural Network Learning on Neuroprocessors," *Microprocessing and Microprogramming*, Vol. 41, 1996, pp.757-769.

[15] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 3rd Edition, 2003, ISBN 1558605967.

[16] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, http://www.intel.com/products/processor/manuals/index.htm.

[17] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conference Proceedings*, 1967.

[18] C. Guiang, K. Milfeld, A. Purkayastha and J. Boisseau, "Memory Performance of Dual-Processor Nodes: Comparison of Intel Xeon and AMD Opteron Memory Subsystem Architectures," *Proceedings for Cluster World Conference & Expo*, San Jose, CA. June, 2003.

[19] J. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE*, Vol. 83, No. 12, December 1995, pp. 1609-24.

[20] A. Binstock and R. Gerber, *Programming with Hyper-Threading† Technology How to Write Multithreaded Software For Intel(r) IA-32 Processors*, Intel PRESS 2 003, ISBN 0970284691.