Streaming SIMD Extensions and General Vector Operations

David Mackay and Steven Chin
ISV Performance Lab
Intel Corporation
EY2-05
5350 N.E. Elam Young Parkway
Hillsboro, OR 97124 USA

1 Abstract

This paper explains the memory cacheability control instructions in the Pentium® III processors and how their use benefits the performance of vector operations. The Pentium III processor introduces 70 new instructions and eight new 128 bit XMM registers to the traditional Intel® architecture processors. Eight of these new instructions are cacheability control instructions[1]. This report emphasizes their effect on the performance of matrix and vector operations as used in many workstation applications. The benefit of these instructions is not limited to codes with intensive floating point operations. The guidelines in this report may be applicable to many other algorithms. The cacheability instructions explored in this report are the data prefetch and the streaming store instructions. Ten general guidelines to follow when adding prefetch and streaming store instructions are introduced. Their influence on several vector operations is explained and illustrated. Tables show the performance improvements and source is shown in the Appendix. An explanation of the remaining Streaming SIMD Extensions and their performance benefit in different types of applications may be available in other documents.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.
Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © 1999 Intel Corp.

[1] Fifty of the new instructions operate on data in the new XMM registers. The remaining twelve new instructions operate on the MMX™ technology registers, which are also the floating point stack.

1 ABSTRACT **1**

2 SCOPE OF STUDY **3**

3 PREFETCH AND STREAMING STORE **3**

3.1 PREFETCH INSTRUCTIONS 4
3.2 STREAMING STORE INSTRUCTIONS 5
3.3 PROGRAMMING 6
3.4 COMPILER INTRINSICS 6

4 EXAMPLES AND USAGE GUIDELINES **7**

4.1 OVERVIEW OF STREAM, DAXPY, AND DCOPY BENCHMARKS 7
4.2 GUIDELINES - USAGE AND APPLICATION 10
4.2.1 Unroll loops *10*
4.2.2 Optimize prefetch distance *11*
4.2.3 Minimize switching from reading to writing data *12*
4.2.4 Prefetch variables being read and not written *12*
4.2.5 Do not prefetch those variables that can benefit from streaming store *13*
4.2.6 Use temporary arrays to store data that is write-only and can benefit from the use of the streaming store instruction *14*
4.2.7 Combine prefetch and streaming store for optimal performance *15*
4.2.8 Ensure Page Table Entries are resident in the TLB *15*
4.2.9 Use prefetch only when appropriate *16*
4.2.10 Prefetch data that is not reused *17*
4.3 VTUNE™ PERFORMANCE TOOL DATA FOR DAXPY 18
4.3.1 VTune tool counters *19*
4.3.2 VTune tool Data *21*

5 LINEAR ALGEBRA NOTES **22**

5.1 DAXPY NON UNIT STRIDE 23
5.2 LU MATRIX FACTORIZATION 23

6 CONCLUSIONS **24**

7 ACKNOWLEDGEMENTS **25**

8 APPENDIX A. SOURCE CODE **26**

8.1 STREAM KERNELS 26
8.1.1 The COPY Kernel *26*
8.1.2 The SCALE kernel *26*
8.1.3 The SUM Kernel *27*
8.1.4 The TRIAD Kernel *27*
8.2 DCOPY 28
8.3 DAXPY 30

2 Scope of Study

Most workstation application codes perform extensive floating point operations. These floating point operations run exclusively in double precision. Many workstation applications perform extensive operations on matrices, vectors and in copying large blocks of memory. The prefetch and streaming store instructions within the Streaming SIMD Extensions set benefit these operations. This report explains how these instructions benefit these kernels. There is no benefit to these operations from the use of the MMX™ technology instructions, which operate on integers and the SIMD-fp instructions in the Streaming SIMD Extensions, which operate on single precision floating point quantities. This report begins by explaining the prefetch and streaming store instructions and how they improve performance. Next we present several guidelines to follow towards the effective use of these instructions and demonstrate the effect of these guidelines on kernel codes. Finally, we discuss the impact of these instructions on a matrix solver. The last section summarizes the results.

3 Prefetch and Streaming Store

In many long vector/matrix operations the data may not always be in cache. In these cases data must be retrieved from memory. In many cases computational operations are slowed down while the processor waits for data to be retrieved from main memory. When the processor must wait for data retrieval, it is said to stall. A prefetch operation retrieves data from memory and places it in the processor's cache before it is needed in the processor's register. This is illustrated in Figure 1. For compute bound code segments prefetch can entirely eliminate the processor stall when waiting for data. The prefetch operation brings the data into cache while the computation is being performed. In this case the stall process can be avoided as shown in Figure 2. In some circumstances the stall will only be partially hidden. This happens when the computation finishes faster than the fetch operation. This case is illustrated in Figure 3.

The benefit of prefetch comes from the ability to hide the latency of retrieving data from main memory. Streaming store on the other hand improves the flow of data from the processor out to main memory. In typical store operations, the destination address is read into cache for ownership, then the data is written out. So instead of a single operation going across the bus to store the data, there are actually two operations going across the bus. With the streaming store commands, the destination address does not need to be brought into cache, and the read for ownership of the cacheline is eliminated. The data is written directly to memory. Since the read for ownership is avoided altogether, this means there are fewer bus operations. The reduction in operations being transferred over the bus effectively increases the bus bandwidth by eliminating unnecessary traffic and freeing up these cycles for effective data transfer. When the destination address already lies in cache, the streaming store writes the data to the cache instead of to memory. In this

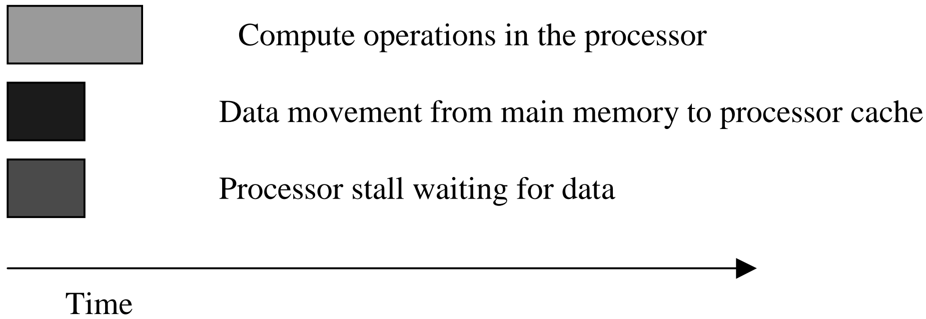

Figure 1. Processor stalls periodically while waiting to retrieve data from main memory into cache or into the processor's register

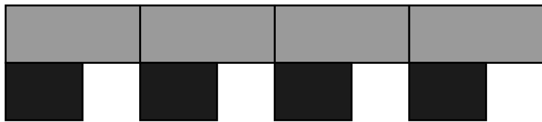

Figure 2. Prefetch operations issued in advance of a compute intensive phase bring data into cache before processor completes previous task. This eliminates a processor stall to wait for data from memory.

Figure 3. For short compute intensive operations, the prefetch operation may not complete as fast as the previous processor task. In this event the processor may still stall waiting for data from memory, but the length of the stall is shortened.

examples in Section 4.2.5.

3.1 Prefetch Instructions

There are four variants of the prefetch instructions available in the Streaming SIMD Extensions. These instructions bring in one cache line of data, which is 32 bytes. The first of these, **prefetchnta**, brings data into the L1 cache only. It does not copy the data into the L2 cache. The second instruction is **prefetcht0**. This instruction brings data into all cache levels, (L1 and L2). **Prefetcht1** brings data into the L2 cache only. On systems

available to us **prefetcht2** brings in data to the L2 cache only and behaves the same as prefetcht1. In the event future systems have additional levels of cache, prefetctht1 and prefetcht2 may behave differently. The prefetch instruction takes a byte address as its only operand. The cache line that contains that byte address is prefetched from memory into the appropriate level in the cache hierarchy. On current Intel Architecture systems a cache line is 32 bytes. Therefore, each prefetch instruction will prefetch 32 bytes of data from memory. The focus of this study is double precision floating point values and the Streaming SIMD Extensions. Each double precision floating point value is 8 bytes. When using the prefetch instruction, each instruction operates on 4 double precision floating point values. There is an additional point to understand about the prefetch instructions.

First, prefetch instructions never raise exceptions. The prefetch instructions act as hints. There is no error in attempting to prefetch an invalid address. The effect of the translation lookaside buffer, or TLB, is also important. The TLB stores the most recently used page directories and page table entries in an on chip cache. When a memory address is dereferenced, the physical location is obtained by checking the addresses page directory, the page table, and its offset in the page. If the page directory and page table is not present in the TLB, then page directory and page table entries must be retrieved from memory for memory address translation. When the page directory and page table information is resident in the TLB, a prefetch instruction prefetches the appropriate 32 byte cache line into the specified cache. If the appropriate entries are not in the TLB, the processor will not retrieve the necessary entries into the TLB and load the specified data into the appropriate cache structure. In this instance the prefetch is treated as a hint, and the data load to cache is not performed. In some of the kernels we examine in this report this is not a concern. In one case it is important and Section 4.2.8 explains how to handle this situation.

3.2 Streaming store instructions

The streaming store commands are **movntps** and **movntq**. Movntps writes 128 bits of data from the new XMM registers to memory. Movntq writes 64 bits of data from the MMX registers (floating point stack) to memory. As previously mentioned if the data already resides in cache, data is written to cache rather than to main memory. The streaming store instruction streams data from either MMX or XMM registers to main memory. The MMX registers are 8 bytes wide, while the XMM registers are 16 bytes wide. Each streaming store instruction operates either on 8 bytes or 16 bytes of data. When using the streaming store instructions for the XMM registers each instruction operates on two double precision floating point values.

It is important to note that both of the streaming store instructions move data from either the MMX registers or the new XMM registers. During context switching between different processes, the NT operating system does not automatically save the state of the floating point unit, the floating point stack, and the XMM registers. If the process which was swapped in on the context switch attempts to use any of the these resources (the floating point registers, or the XMM registers), then the NT Operating system interrupts

and saves the floating point state, the floating point stack and the XMM registers. When the previous task is swapped back in, everything is restored from memory. When processes that only operate within the general registers swap in and out, the NT operating system never needs to save or restore the floating point units. Within the context of the NT operating system, saving and restoring the state of the XMM registers is coupled with the saving and restoring the floating point stack. For applications which perform floating point operations, or integer operations in the MMX registers there is no penalty for using the streaming store operations. If an application which does not perform any floating point operations or integer operations in the MMX registers begins to use streaming store operations to improve memory bandwidth, there will be an additional cost for the saving and restoring of the floating point state, the floating point stack and the XMM registers. In many cases the benefit will more than compensate for the cost of save and restore. However, in cases where the streaming store is used very little, the cost to save and restore this information may outweigh the benefit or improvement. This will depend on the amount of data transferred using the streaming store instructions. For the types of applications examined in this report, this is not an issue. The algorithms already make extensive use of the floating point unit. If the guidelines for usage of the streaming store instructions is applied to different types of applications this should be considered.

3.3 Programming

There are several different ways to program using the Streaming SIMD Extensions. These methods are: write in assembly, use a set of compiler instrinsics, or use a vector class library. Intel developed and introduced a set of vector class libraries to ease the use of Streaming SIMD Extensions. These vector class libraries can be used in C++ programs to take advantage of the Streaming SIMD Extensions. In addition, Intel developed a set of intrinsics to be used with their C/C++ compiler. These intrinsics are like macros, which are defined using the appropriate Streaming SIMD Extensions. The third way to utilize Streaming SIMD Extensions is to program in assembler. For this paper we utilized the compiler intrinsics calls and the Intel 1.0 compiler from the 1.0 Pentium III SDK. All results reported in this paper came from runs on the 1.0 Pentium III SDK. The different codes we studied and their results are described in the Section 4.1. The compiler intrinsics we used in this study are presented below. These intrinsics are used in the coding examples of Section 3 and the appendix

3.4 Compiler Intrinsics

Prefetch intrinsics:

_mm_prefetch(address, hint type) ;

This intrinsic fetches the 32 byte cache line containing the *address* to the appropriate cache specified by the *hint type*. For example: *_mm_prefetch(a, _MM_HINT_NTA) ;* fetches the 32 byte cache line containing the address a to the L1 cache. _MM_HINT_T0 specifies both L1 and L2 cache,_MM_HINT_T1 specifies L2 cache only. Hint types are defined in the xmmintrin.h file and in the compiler user's guide.

_mm_stream_pi(destination address, source) ;
_mm_stream_ps(destination address, source) ;

These intrinsics streaming store 64 bits or 128 bits of data to main memory. The instruction *_mm_stream_pi* uses the floating point stack registers, or MMX registers. The instruction *_mm_stream_ps* uses the new XMM registers. Addresses must be properly aligned (16 byte alignment for *_mm_stream_ps,* and 8 byte alignment for *_mm_stream_pi*). For example: *_mm_stream_pi((__m64&)&a[I+2],(__m64*)&tarr[0])* ; stores 64 bits of data from tarr, to the address beginning a[I+2].

__m128 t = mm_load_ps(source address) ;
__m128 t = mm_loadu_ps(source address) ;

The *mm_load_ps* instruction loads 128 bits of data from the source address to the new XMM registers. The source address must be 16 byte aligned. The *_mm_loadu_ps* instruction loads 128 bits of data from the source address to the new XMM registers, the source address does not need to be 128 byte aligned. Its use might be combined with one of the streaming store commands as illustrated here: *_mm_stream_ps(&a, _mm_load_ps(b)) ;*

4 Examples and Usage Guidelines

In this section we describe the basic linear algebra kernels used to illustrate optimal usage of the prefetch and streaming store instructions. In Section 4.1 we give an overview of the kernels. We also report the performance improvements obtained by adding prefetching and streaming store instructions to the code. In Section 4.2 we present ten basic guidelines to follow when adding prefetch and streaming store commands to applications. The vector operations that are considered for this study are represented by the STREAM benchmark, and supplemented with a traditional DAXPY and DCOPY operations. They are described in more detail below.

4.1 Overview of STREAM, DAXPY, and DCOPY Benchmarks

The STREAM benchmark is a well known benchmark that is slowly becoming a standard for measuring memory sub-system performance on current generation computer architectures. The benchmark was created and is maintained by John McCalpin from Silicon Graphics, Inc. The source code for the benchmark is freely available, and results from many different computer systems are published and generally available from the STREAM web site, which is located at http://www.virginia.edu/stream.

STREAM is a synthetic benchmark, written in both Fortran and C, which measures the performance of four long vector operations. The four long vector operations are typical building blocks for vector operations, and are as follows:

Table 1. List of kernels that comprise the STREAM benchmark.

Name	Kernel	Bytes/Iteration	FLOPS/Iteration
COPY	A(I) = B(I)	16	0
SCALE	A(I) = q * B(I)	16	1
SUM	A(I) = B(I) + C(I)	24	1
TRIAD	A(I) = B(I) + q * C(I)	24	2

STREAM defines the vector lengths to be large enough such that they do not fit within the cache hierarchy. This does not imply that there is no data re-use in real applications, but it is intended to attempt to couple the floating point performance of a system to the memory sub-system performance, rather than to the cache efficiency. On most current generation architectures, floating point performance is now much better than memory performance. Therefore, typically all four kernels will execute at roughly the same performance.

STREAM operates on double precision or 64-bit operands. The default size of each vector is 2 million elements, or 8 Mbytes. The code is also written to avoid re-use of data in the cache. This is true within a kernel, as well as when going from kernel to kernel. Since STREAM uses double precision floating point values, STREAM cannot take advantage of the floating point SIMD instructions in the Streaming SIMD Extensions.

The third column in the table defines the number of bytes transferred to or from the memory sub-system per iteration of the loop. STREAM counts explicit read and write operations separately. A memory copy operation therefore transfers two double precision words, one for reading and one for writing, or 16 bytes per iteration. STREAM does not take implicit memory transfers into account. For example, on those architectures that use a write-allocate policy there will be an extra read to load data that will be written. This data traffic is not explicitly specified in the calculation, and therefore is not included in the measurement.

For this reason, the TRIAD operation in the STREAM benchmark is not identical to the level 1 BLAS DAXPY operation. In the DAXPY operation one of the operands is over-written with the computed result. The DAXPY operation is defined below.

Table 2. The characteristics of the DAXPY kernel.

Name	Kernel	Bytes/Iteration	FLOPS/Iteration
DAXPY	Y(I) = alpha * X(I) + Y(I)	24	2

On architectures which use a write-allocate policy, the TRIAD operation in STREAM will require the extra read of the A array before it is written. In the case of the DAXPY operation, the result is written into Y, and Y is also used as an operand. The extra read may not be required, but the cache line must be transitioned to modified after being updated.

The DCOPY routine is identical to the first COPY operation in STREAM. Like DAXPY it is one of the level 1 BLAS routines. In the STREAM benchmark all of the code is in one main program without any subroutines. For the DCOPY test case we wrote the code as a separate subroutine and were more aggressive in its optimization. The DCOPY routine shares a lot in common with a more general memcopy routine. The chief difference is that it operates on multiples of 8 byte words and we assume 8 byte data alignment of the addresses on entering the routine. The version in the appendix, Section 8.2, can be modified to handle a more general case.

Table 3. Measurements for STREAM, DAXPY, and DCOPY in Mbytes/Sec, for initial code and with improvements using prefetch and streaming store.

Name	**Kernel**	**Initial Mbytes/Sec**	**Improved Mbytes/Sec**	**% Improvement**
COPY	A(I) = B(I)	290.9	543.2	86.7%
SCALE	A(I) = q * B(I)	293.7	540.5	84.0%
SUM	A(I) = B(I) + C(I)	340.3	569.9	67.5%
TRIAD	A(I) = B(I) + q * C(I)	331.2	564.9	70.6%
DAXPY	Y(I) = alpha *X(I) + Y(I) (unrolled daxpy loop)	298.8 (413)	514.6	72.2% (24.6%)
DCOPY	Y(I) = X(I)	278.2	620	123%

The table above gives the initial results without any modifications and results using the Pentium III processor prefetch and streaming store instructions. The initial STREAM measurements are taken using the code as-is, without any modifications or optimizations. The improved version shows how the addition of prefetch and streaming store can improve the code. Since the code was modified these measurements do not qualify as STREAM benchmark results. The DAXPY case was measured using a small test program that is similar to the STREAM benchmark, but this operation is not part of STREAM and therefore is executed as a separate application. The vector length for the DAXPY was selected to match the same length as the vector operations in STREAM. The initial DAXPY and DCOPY measurements were made with this code, using standard C language constructs. In the case of DAXPY we compared performance to both the simplest case where the loop is not unrolled, and when the loop is unrolled by a factor of four. The final versions of the source code used for measurements are included in Appendix A. Source Code

The use of the Pentium III prefetch and streaming store instructions improves performance of these five operations from a minimum of 24.6% for DAXPY, to 123% faster for the DCOPY routine.

The specific optimizations and transformations that have been performed for these five operations are described in Section 4.2. Measurements from each of the optimization steps are provided, analyzed and discussed. The application of each of the basic guidelines is discussed for each example.

4.2 Guidelines - Usage and Application

This section discusses ten basic implementation guidelines and demonstrates their implementation or application to the STREAM, DCOPY and DAXPY kernels. The optimal results for each kernel were already reported in Section 4.1. The optimal results are often obtained by combining multiple guidelines for a particular kernel. In the subsection for each guideline we often present timings that show how much is gained incrementally by adopting just that one guideline.

4.2.1 Unroll loops

In order to take optimal advantage of the datatype associated with the prefetch and streaming store instructions, double precision floating point loops should be unrolled to operate on 4 values in one iteration.

Unrolled versions of the STREAM COPY benchmark and the DAXPY benchmark are shown below. Both codes have been unrolled by a factor of 4. When unrolling the loops, the loop increments by 4 instead of 1, and four values are computed for each iteration.

```
For (I=0; I<N; I++)                 For (I=0; I<N; I+=4)
{                                   {
     A[I] = B[I]                       A[I]   = B[I]
}                                      A[I+1] = B[I+1]
                                       A[I+2] = B[I+2]
                                       A[I+3] = B[I+3]
                                    }

For (I=0; I<N; I++)                 For (I=0; I<N; I+=4)
{                                   {
     Y[I] = Y[I] + X[I]*alpha          Y[I]   = Y[I]+ X[I]*alpha
}                                      Y[I+1] = Y[I+1] + X[I+1]*alpha
                                       Y[I+2] = Y[I+2] + X[I+2]*alpha
                                       Y[I+3] = Y[I+3] + X[I+3]*alpha
                                    }
```

The unrolling of the loops by a factor of 4 optimizes re-use of data within the cache line. This is known as spatial locality. The performance improvement that is obtained by unrolling by a factor of 4 compared to a factor of 2 ranges from 9.6% for STREAM COPY to 40.2% for STREAM SCALE. In our tests unrolling the loop to consume all of the data in one cache line per iteration always showed improvement.

Table 4. Effect of unrolling the main loop by a full (4 double precision words) or half cache line (2 double precision words) on the STREAM benchmarks.

	Loop Unrolling	**MB/sec**	**% Improvement**
COPY	4 double precision words	543.2	9.38%
COPY	2 double precision words	496.6	
SCALE	4 double precision words	540.5	38.37%
SCALE	2 double precision words	390.62	
ADD	4 double precision words	569.9	36.90%
ADD	2 double precision words	416.3	
TRIAD	4 double precision words	564.9	10.81%
TRIAD	2 double precision words	509.8	

Table 5 Effect of unrolling the main loop by a full (4 double precision words) or half cache line (2 double precision words) on DAXPY.

	Loop Unrolling	**MB/sec**	**% Improvement**
DAXPY	1 double precision word	326.4	
DAXPY	2 double precision words	493.9	51.3%
DAXPY	4 double precision words	514.6	57.7%

4.2.2 Optimize prefetch distance

Prefetch scheduling distance should be far enough in advance to ensure that the operation can complete before the data is actually needed. If the data is being prefetched from memory, on a Slot 1 Pentium III system the latency to memory is on the order of 44 - 72 processor cycles. Prefetch scheduling distance therefore should be far enough in advance such that as much of this latency as possible can be hidden, or overlapped. In general, for STREAM and DAXPY operations, we recommend prefetching at least one full cache line in advance. As explained below, this means the prefetch distance on each loop is 7 values (56 bytes) in advance.

Within code loops this typically means issuing a prefetch for all of the data for the next loop iteration rather than the current loop iteration. Data alignment must be considered when calculating this loop iteration distance. Cache lines on current Intel Architecture platforms are 32 bytes long. Each cache line therefore contains 4 double precision floating point values. For the following discussion assume that loops are unrolled by a factor of 4. If addresses are 32 byte aligned, and therefore start at the beginning of a cache line, the data for the next iteration of the loop lies 4-7 double precision floating point values away. If addresses are 8 Byte-aligned, it is possible that the beginning address for an array is in the middle of a cache line. In this case the last data to complete the next loop iteration may lie 7-10 double precision floating point values away from the initial address at the beginning of the current iteration of the loop. Therefore, when dealing with double precision floating point values, if loops are unrolled by a factor of 4, prefetch 7 double precision floating point values in advance to ensure the cache line needed for the next iteration of a loop is prefetched.

The general rule for unrolled loops is to prefetch an array element *(i +2*(cache line size)/sizeof(data type))-1)* from the first element in the current iteration of the loop, where i is the number of cache lines in advance to prefetch.

Notice how varying vector alignment can affect performance of the DAXPY operation as shown in Table 6. For the case when the Y vector is only 8 Byte aligned, prefetching ahead by 7 offered the best overall performance. Even though it is the X vector which is prefetched, the proper advance prefetch distance appears to help to smooth out the cache effects.

Table 6. The performance on DAXPY by prefetching ahead 4 words versus 7 words (only vector X is prefetched).

Y Data Alignment	X Data Alignment	X Prefetch Distance	Mbytes/sec	% Improvement
32 Byte	32 Byte	No prefetch (loop unrolled)	417.8	
32 Byte	32 Byte	4 double precision words	516.4	23.6%
32 Byte	32 Byte	7 double precision words	518.4	24.1%
8 Byte	Page	No prefetch (loop unrolled)	383.2	
8 Byte	Page	4 double precision words	439.3	14.6%
8 Byte	Page	7 double precision words	512.0	33.6%

4.2.3 Minimize switching from reading to writing data

The DCOPY routine is optimized by combining the previous tips already explained and by grouping all the prefetch instruction together into a block and grouping the streaming stores together in another block. This minimizes the number of transitions from reading data to writing data. For DCOPY we use block sizes that are 4K, or one page. Following this strategy we see that the bandwidth improves significantly.

Table 7. The performance of DCOPY prefetching 4K at a time versus prefetching 4 double precision words at a time.

Copy size	32 Byte blocks Mbytes/sec	4 KB blocks Mbytes/sec	% Improvement
8MB	461.4	626.7	35.8%

4.2.4 Prefetch variables being read and not written

Prefetch is intended to initiate memory requests in advance of when the data is actually needed. This helps to overlap and effectively hide the latency associated with accessing memory. By the time the data is actually needed for computation, it is as close to the processor as directed. Prefetch behaves similar to a "read" request, but consumes less processor resources compared to an actual read. For example, prefetch does not require the use of a processor register, since the value is only brought into the cache hierarchy. Therefore, prefetch works best for those values which are read, but not modified (i.e., not written).

Table 8. DAXPY performance measured when prefetching variables only being read, variables being read and written, and prefetching both simultaneously.

Data Prefetch	Mbytes/sec	% Improvement
No Prefetch (loop unrolled)	417.8	0.0%
X only	514.6	23.2%
Y only	451.3	8.0%
X and Y	506.0	21.1%

In the case of the DAXPY benchmark, the Y array is both read and written, while the X array is read only. If X is prefetched, the performance of the DAXPY improves by 23.2%. If Y is prefetched, the performance only improves by 8%, and is 15% less than the performance measured when prefetching the X array.

4.2.5 Do not prefetch those variables that can benefit from streaming store

One of the main advantages of the steaming store instruction is if the data is not in the cache, the store bypasses the cache and avoids cache pollution. Prefetch brings data into different levels of the cache hierarchy, depending upon which prefetch instruction is used. If the streaming store instruction finds the data in the L2 cache, it reads the data into the L1, updates the data in the L1 and marks that line as modified. It then transitions the line in the L2 cache to invalid. Therefore, the behavior of the streaming store when writing data that is already in the cache is similar to the use of traditional stores, and the performance benefit of the streaming store is eliminated. For this reason, it is not recommended to use both prefetch and streaming store on the same data item.

Table 9. STREAM benchmark performance when prefetching both the source and the destination.

	MB/sec (prefetch source, stream destination)	**MB/sec (prefetch source and destination, stream destination)**	**% Improvement**
COPY	543.2	262.6	106.85%
SCALE	540.5	292.2	84.98%
ADD	569.9	304.3	87.28%
TRIAD	564.9	304.3	85.64%

The performance on all the STREAM benchmarks decreases dramatically when the destination operand is prefetched prior to using streaming store to write out the data. The prefetch instruction creates unnecessary traffic on the bus to load the destination into cache. Once the destination is in cache, the subsequent streaming store instruction writes the data to the cache and does not stream into memory. In the specific case of the STREAM benchmarks cache pollution is probably less of an issue since the benchmarks are designed not to re-use data. However, in a more general case this could also be a significant performance factor.

4.2.6 Use temporary arrays to store data that is write-only and can benefit from the use of the streaming store instruction.

Based on the arguments given above, one should naturally conclude that it is beneficial to avoid having to load variables prior to using the streaming store instructions. Temporary arrays can be used to hold the resultant values prior to streaming the results to memory. If the temporary array is the same size as a cache line, the temporary array will remain in the L1 cache and be quickly updated. The streaming store instruction will move data directly to memory, without polluting the cache, and without creating unnecessary traffic on the system bus.

```
for (i = 0 ; i < n ; i++)
{
      A[i] = q * B[i] ;
}
```

```
for (i=0; i<n; i+=4)
{
    temp[0] = q * B[I] ;
    temp[1] = q * B[I+1] ;
    temp[2] = q * B[I+2] ;
    temp[3] = q * B[I+3] ;
    mm_stream_ps(A[I],
       *(__m128*)&temp[0]) ;
    mm_stream_ps(A[I+2],
       *(__m128*)&temp[2]) ;
}
```

The STREAM SCALE benchmark using the streaming store intrinsic is shown above. The loop has been unrolled by a factor of 4, and the result is stored into a temporary array TEMP. The values in TEMP are then stored to , A, the proper resultant memory locations using the streaming store. By using the temporary array TEMP the resultant array A is never loaded into the cache and optimal benefit from the streaming store results. The casting of the TEMP array to a __m128 datatype is necessary to meet the proper prototype for the streaming store.

Table 10. Measured performance improvement from the use of temporary arrays to implement streaming store on the STREAM benchmark.

	MB/sec (no prefetch or streaming store)	MB/sec (use of temporary arrays to implement streaming store)	% Improvement
COPY	290.9	486.5	67.24%
SCALE	293.7	399.2	35.92%
ADD	340.3	516.9	51.90%
TRIAD	331.2	333.2	0.60%

The use of streaming store considerably improves performance on three out of four of the STREAM benchmarks.

4.2.7 Combine prefetch and streaming store for optimal performance

The use of prefetch and streaming store can provide performance improvement beyond the cumulative improvement offered by the simple addition of the two instructions. In the case of both the STREAM SCALE and SCALE TRIAD, the performance using both prefetch and streaming store is significantly greater than the additive effect from using prefetch and streaming store individually. In the case of STREAM COPY and STREAM ADD, the performance using prefetch and streaming store combined is slightly better than the sum of the individual contributions. These results indicate that the prefetch and streaming store operations can be overlapped, resulting in a synergistic effect, or additional performance improvements beyond the individual components. The expected performance improvements based on a cumulative effect of prefetch and streaming store is shown in Column 5 in Table 11 while the measured results combining prefetch and streaming store is shown in Column 6.

Table 11. Effect of prefetch and streaming store on the STREAM benchmarks. The effects are analyzed individually and then together.

	MB/sec (no prefetch or streaming store)	**MB/sec (prefetch only)**	**MB/sec (streaming store only)**	**MB/sec (Δ prefetch + Δ streaming store)**	**MB/sec (prefetch and streaming store)**
COPY	290.9	321.3	486.5	516.9	543.2
SCALE	293.7	314.9	399.2	420.4	540.5
ADD	340.3	355.7	516.9	532.3	569.9
TRIAD	331.2	350.0	333.2	352.0	564.9

It should be noted that in the case of the STREAM benchmarks, the advantage of combining prefetch and streaming store is primarily due to the fact that the result vector is never used as an operand. Therefore, the prefetch and streaming store instructions operate on completely different memory locations and do not conflict with each other.

4.2.8 Ensure Page Table Entries are resident in the TLB

The TLB is a fast memory buffer that is used to improve performance of the translation of a virtual memory address to a physical memory address by providing fast access to page table entries. If memory pages are accessed and the page table entry is not resident in the TLB, a TLB miss results and the page table must be read from memory. The TLB miss results in a performance degradation since memory access is slower than TLB access. The TLB can be pre-loaded with the page table entry for the next desired page by accessing (or touching) an address in that page. This is similar to prefetch, but instead of a data cache line the page table entry is being loaded in advance of its use. This will help to ensure that the page table entry is resident in the TLB. This is very important issue for prefetch instructions. If the page table entry is not resident in the TLB prefetch instructions are retired without successfully bringing data into the cache.

Table 12. Performance improvement obtained on the STREAM COPY and TRIAD benchmarks by pre-loading the TLB entry for the next page.

	MB/sec (prefetch and streaming store, no TLB pre-load)	**MB/sec (prefetch and streaming store, with TLB pre-load)**	**% Improvement**
COPY	538.0	543.2	1.0%
TRIAD	489.9	564.9	15.3%

Significant performance advantage is obtained on the STREAM TRIAD benchmark by pre-loading the TLB. The TRIAD benchmark uses three different arrays. In the examples used for this study, each array is 8 Mbytes in size, for a total of 24 Mbytes in the TRIAD benchmark. This is more than 3,000 (4096 Byte) pages of data and is much larger than the capacity of the TLB.

Making sure the TLB entries are current also helps even out performance for the DCOPY routine. As we pass through memory and call DCOPY with different address alignments we get varying results. Pre-loading the TLB evens out the performance as shown in Table 13. Notice the large drop in performance when X is only 16 Byte aligned. Our test code steps through 40 MB of data prior to each call to DCOPY in order to flush out the cache and alter the TLB. We do think it is the page alignment of the source vector that affects performance as much as it is the TLB activity. The beginning address shifts for the tests, the code will touch one more or fewer page which can affect the TLB status. Pre-loading the TLB evens out this affect as the data Table 13 shows

Table 13. DCOPY rates in MB/sec for different alignments of X and varying block sizes. (Y is page aligned).

X alignment	Block size	No TLB preload	TLB pre-loading
Page	256 Bytes	105.4	121.6
Page	8 MB	475.8	619.1
24 Byte	256 Bytes	121.2	117.0
24 Byte	8 MB	626.4	627.9
16 Byte	256 Bytes	100.6	114.3
16 Byte	8 MB	376.4	614.2
8 Byte	256 Bytes	112.2	110.1
8 Byte	8 MB	603.7	614.6

4.2.9 Use prefetch only when appropriate

As previously mentioned, prefetch consumes less processor resources compared to a read request. However, this does not imply that prefetch does not consume any processor resources. The use of prefetch still consumes load buffers in the processor, as well as memory bandwidth on the bus. Prefetch is not free, and should not be used in excess. This may result in degraded performance due to increased resource related stalls in the processor.

Table 14. The effect on the performance of DAXPY by prefetching only one cache line in advance versus prefetching two adjacent cache lines simultaneously.

Data Prefetch	Mbytes/sec	% Improvement
No Prefetch	411.2	0.0%
X[j+4]only	512.8	24.7%
X[j+4] and X[j+8]	384.7	-6.4%

In the case of the DAXPY benchmark, prefetching one cache line in advance improves performance by 24.7%. However, prefetching one and two cache lines in advance actually degrades performance compared to the case without prefetch. This is due to the fact that prefetch does indeed consume resources both within the processor as well as on the system bus. Too much prefetching introduces resource contention and processor stall conditions.

4.2.10 Prefetch data that is not reused

As the last guideline we introduce a new algorithm that is not a standard vector operation. The case for this example comes from the factorization of a symmetric matrix. The basic algorithm presented here is written in C and has been modified. The essence of the data access pattern was preserved sufficiently to illustrate the principle. There is a sequence of vectors which are modified based on a short dot product. The dot product here has been unrolled. The elements of one of the vectors in the dot product are stored as scalars rather than as an actual vector. This short vector is doted with a row of the matrix B. The vectors being updated are stored sequentially in variable A. As each vector stored in A is updated, the same matrix B is reused. Therefore it is beneficial to keep B in cache. The elements of A are never reused. It is preferred to get these values into the L1 cache, update the value and write them out, without ever putting them in the L2 cache. If A were put in the L2 cache, it might have the unfortunate effect of forcing out other data, which will be reused. The code with prefetch instructions is shown below. The performance of the loop shown below improved by 15.7% with the addition of the prefetch instructions. It is more important to prefetch data that is only used once. Data that is being reused will be brought into cache on its first reference, and has a good chance of remaining in cache.

```
joffst = 1 ;
for (j = 0 ; j < m ; j++)
{
   _mm_prefetch((char *)(a+joffst), _MM_HINT_NTA) ;
   _mm_prefetch((char *)(a+joffst+7), _MM_HINT_NTA) ;
   c1 = . . . ; c2 = . . . ; . . . c6 = . . . ;
   klen =  len & ~3 ;
   for (k = 0 ; k < klen ; k += 4)
   {
      _mm_prefetch((char *)(a+joffst+7), _MM_HINT_NTA) ;
      a[joffst++] += c1 * b[1][k+j] + c2 * b[2][k+j] + c3 * b[3][k+j]
                  +  c4 * b[4][k+j] + c5 * b[5][k+j] + c6 * b[6][k+j] ;
      a[joffst++] += c1 * b[1][k+j+1] + c2 * b[2][k+j+1]
                  +  c3 * b[3][k+j+1] + c4 * b[4][k+j+1]
                  +  c5 * b[5][k+j+1] + c6 * b[6][k+j+1] ;
      a[joffst++] += c1 * b[1][k+j+2] + c2 * b[2][k+j+2]
                  +  c3 * b[3][k+j+2] + c4 * b[4][k+j+2]
                  +  c5 * b[5][k+j+2] + c6 * b[6][k+j+2] ;
      a[joffst++] += c1 * b[1][k+j+3] + c2 * b[2][k+j+3]
                  +  c3 * b[3][k+j+3] + c4 * b[4][k+j+3]
                  +  c5 * b[5][k+j+3] + c6 * b[6][k+j+3] ;
   }
   for ( ; k < len ; k++)
      a[joffst++] += c1 * b[1][k+j]  + c2 * b[2][k+j]
                  +  c3 * b[3][k+j]  + c4 * b[4][k+j]
                  +  c5 * b[5][k+j]  + c6 * b[6][k+j] ;
   len-- ;
}
```

4.3 VTune™ performance tool data for DAXPY

The VTune™ tool is the central component of the VTune Performance Enhancement Environment. The VTune tool collects, analyzes, and provides Intel Architecture-specific software performance data from the system-wide view down to a specific module, function, and instruction in your code**.**

Vtune tool's time- or event-based sampling analysis provides the capability to monitor all active software on the system, including the application, non-intrusively. Time-based sampling periodically interrupts the processor and collects samples of the instruction addresses, matches these addresses with an application or an operating system routine, and creates a database with the resulting samples data. The VTune performance tool can then graphically display the %CPU time associated with each active module, process, and processor (on a multiprocessor system). Event-based sampling can be used together with the hardware performance counters available in the Intel Architecture to provide detailed information on the behavior of specific events in the micro-processor. Some of the micro-processor events that can be sampled include L2 cache misses, branch mis-predictions, mis-aligned data access, processor stalls, and instructions executed.

VTune performance tool is a valuable tool to help identify and analyze performance bottlenecks at the micro-architecture, application and system level. At the micro-architecture level, VTune performance tool can be used to better understand the cache and bus behavior of the DAXPY benchmark.

The DAXPY results shown in this section use variants of final DAXPY presented in the Appendix. The variants were chosen to illustrate behavioral differences with and without the prefetch and streaming store instructions.

4.3.1 VTune tool counters

VTune performance enhancement environment provides access to many performance counters. At first glance, it is difficult to know which counters are relevant for understanding the performance effects of the prefetch and streaming store instructions. Obviously, the main focus areas relate to activity on the system bus, as well as the cache hierarchy. Particular counters that appear relevant include:

1. **L1 cache misses** - this event indicates the number of outstanding L1 cache misses at any particular time.
2. **L2 cache misses** - this event indicates all data memory traffic that misses the L2 cache. This includes loads, stores, locked reads, and ItoM requests.
3. **L2 cache requests** - this event indicates all L2 cache data memory traffic. This includes loads, stores, locked reads, and ItoM requests.
4. **Data memory references** - this event indicates all data memory references to the L1 data and instruction caches and to the L2 cache, including all loads from and to any memory types.
5. **External bus memory transactions** - this event indicates all memory transactions.
6. **External bus cycles processor busy receiving data** - VTune counts the number of bus clock cycles during which the processor is busy receiving data.
7. **External bus cycles DRDY asserted** - this event indicates the number of clocks during which DRDY is asserted. This, essentially, indicates the utilization of the data bus.

Other counter of interest for the DAXPY operation include:

8. **Instructions retired** - this event indicates the number of instructions that retired or executed completely. This does not include partially processed instructions executed due to branch mispredictions.
9. **Floating point operations retired** - this event indicates the number of floating point computational operations that have retired.
10. **Clockticks** - this event initiates time-based sampling by setting VTune counters to count the processor's clock ticks.
11. **Resource related stalls** - this event counts the number of clock cycles executed while a resource-related stall occurs. This includes stalls due to register renaming buffer entries, memory buffer entries, branch misprediction recovery, and delay in retiring mispredicted branches.
12. **Prefetch NTA** - this event counts the number of Pentium III prefetchnta instructions.

Table 15. Raw VTune tool data from DAXPY tests.

Function	BW (Mbytes/sec)	Clockticks	Instructions Retired	FP Instructions Retired	Resource Related Stalls	Prefetch NTA	Data Memory Reference
dax1	411	1195.13	361.50	102.19	978.38	0.00	151.50
dax1v	396	1247.80	335.13	102.02	1015.25	0.00	151.73
dax3	270	1835.06	594.88	101.69	1301.88	12.46	379.88
dax5c	389	1277.10	374.75	101.86	1006.50	12.50	151.93
dax5	519	968.62	347.88	101.72	719.13	12.55	163.80
dax5a	440	1116.72	349.88	101.59	890.25	12.54	163.98
dax5b	490	1006.24	336.38	101.76	774.13	12.50	176.25

Function	BW (Mbytes/ sec)	External Bus Cycles processor Busy receiving Data	External Bus DRDY Asserted	External Bus Memory Transactions	L1 Cache Misses	L2 Cycles Data Bus Busy Transferring Data to CPU	L2 Cache Request Misses	L2 Cache Requests
dax1	411	101.15	51.32	38.21	3413.13	361.00	25.45	25.48
dax1v	396	101.84	50.44	38.32	3429.38	370.13	25.48	25.46
dax3	270	102.30	48.98	37.87	1480.63	308.88	25.44	25.49
dax5c	389	101.16	51.04	38.43	3494.38	387.00	25.41	25.60
dax5	519	102.33	49.23	37.71	1500.63	310.88	25.33	25.41
dax5a	440	101.60	55.43	39.47	1216.88	223.63	25.37	25.42
dax5b	490	101.11	46.72	37.26	146.88	322.25	25.44	25.39

dax1 C code
dax1v Unrolled C code
dax3 Unrolled with prefetch of x(I+4) with streaming store of y(I) using temp
dax5c Unrolled with prefetch of x(I+4) and x(I+8)
dax5 Unrolled with prefetch of x(I+4)
dax5a Unrolled with prefetch of y(I+4)
dax5b Unrolled with prefetch of x(I+4) and y(I+4)

The seven different routines all do the same amount of work. They iterate 200 times on a vector of 262144 double precision words. This translates to 52.43M iterations of the main loop. Each iteration of the main loop requires two double precision floating point operations, for a total of 104M floating point operations. When the loop is unrolled by a factor of 4 to accommodate the cache line size of 32 bytes, 13.1M iterations of the loop are computed for each routine. In each iteration of the unrolled loop, two cache lines are required, for a total of 26M cache line requests per routine. All the routines request the same number of cache lines and all the routines have the same number of L2 cache misses. This is consistent with the fact that the routines are all designed to operate outside the caches.

The number of cycles that the external data bus is being utilized is very consistent across all the routines. This is probably a result of the fact that all the routines need to move the same amount of data from memory to the processors. The number of cycles required to move this data remains the same, regardless of whether prefetch and streaming store are used or not.

The number of resource related stalls varies considerably across the different routines. For the most part, the performance of the routines is inversely related to the number of resource related stalls. According to the definition of this counter, the number of resource related stalls measures the number of clock cycles executed while a stall exists, such as due to register renaming buffer entries, memory buffer entries, branch misprediction recovery, and delay in retiring mispredicted branches occurs. For the most part the DAXPY code has very little branches. The main stall of interest is related to memory buffers.

4.3.2 VTune tool Data

The raw VTune data given in the tables above can be used to compute other quantities. Some examples are given below.

Table 16. Performance characteristics for DAXPY tests computed from the raw VTune tool data.

	BW (Mbytes/ sec)	CPI	% Data Memory References / Instruction	%Cycles processor Busy receiving Data	% Cycles DRDY Asserted	% External Bus Memory Transactions / Data Memory Reference	% FP Instruction / Instruction
dax1	411	3.306	41.91%	33.85%	17.18%	25.22%	28.27%
dax1v	396	3.723	45.27%	32.65%	16.17%	25.26%	30.44%
dax3	270	3.085	63.86%	22.30%	10.68%	9.97%	17.09%
dax5c	389	3.408	40.54%	31.69%	15.98%	25.30%	27.18%
dax5	519	2.784	47.09%	42.26%	20.33%	23.02%	29.24%
dax5a	440	3.192	46.87%	36.39%	19.85%	24.07%	29.04%
dax5b	490	2.991	52.40%	40.19%	18.57%	21.14%	30.25%

	BW (Mbytes/ sec)	% L2 Cache Misses	% Cycles Stalls	% Cycles L2 Data Bus Busy	% Cycles L2 Data Bus Busy Transferring Data to CPU	% L2 Evictions per L2 Request	% L2 M-State Evictions per L2 Request
dax1	411	99.88%	81.86%	56.34%	30.21%	100.03%	49.97%
dax1v	396	100.06%	81.36%	54.35%	29.66%	100.01%	49.51%
dax3	270	99.80%	70.94%	27.98%	16.83%	99.98%	73.25%
dax5c	389	99.28%	78.81%	54.30%	30.30%	99.06%	49.94%
dax5	519	99.70%	74.24%	52.85%	32.09%	100.43%	71.90%
dax5a	440	99.80%	79.72%	29.36%	20.03%	100.22%	40.78%
dax5b	490	100.20%	76.93%	52.00%	32.03%	99.90%	55.75%

The dax5 routine also has the highest bus utilization. This is true whether the metric of %Cycles processor busy receiving data (42.3%), or %Cycles where DRDY# is asserted is

used (20.3%). Using the same metrics, (at 22.3% or 10.7%), the dax3 routine has the lowest bus utilization. This is also the routine which has the worst performance of all the routines. In the case of dax3, prefetch and streaming store are used. The source operand is prefetched, and the destination is streaming stored to memory. However, the destination is also used as a source operand, and as such is in the cache when the store occurs. Therefore, the streaming store simply adds contention on the system bus without providing any additional performance benefit. Hence the low bus utilization and overall performance.

All seven routines execute with 100% L2 cache misses. The use of prefetch does not reduce the percentage of L2 Cache Misses, as counted by the Vtune tool. This is because a miss is generated for each cache line requested. In some routines the miss is generated by the load instruction itself. In other routines the miss may be caused by the prefetch instruction.

The CPI, or Cycles Per Instruction, for the dax5 routine is 2.78, which is the lowest CPI for any of the routines. All of the routines execute the same number of floating point operations, and similar numbers of total operations. The difference in the CPI is a result of the difference in the number of cycles required to complete the instruction mix. The use of prefetch is effective in overlapping the relatively long latency memory operations with the computational instructions.

5 Linear Algebra Notes

In this section we share additional data on several common linear algebra operations. We do not introduce any additional guidelines. This section shares data on LU matrix factorization and on DAXPY operations for non-unit strides.

Many workstation codes reduce a scientific or engineering problem into a set of linear equations. These equations are then solved using various matrix solution methods. These methods may use a dense or sparse matrix solver. These solvers are often built from using a basic set of linear algebra operations commonly known as the Basic Linear Algebra Subroutines or simply BLAS. These routines are used so commonly that they are commonly included in optimized libraries. The DAXPY and DCOPY operations discussed in Section 3 belong to the BLAS I set of routines.

These BLAS routines are available in the Intel Math Kernel Libraries. It is beyond the scope of this study to analyze each BLAS routine and how prefetch benefits it. In Section 4 we considered DAXPY for the case when both vectors stepped through with stride 1. In this section we consider the case for non unit strides. We also examine the influence of prefetch in DGEMM when placed within the scope of the LAPACK linear equation solver.

5.1 DAXPY non unit stride

As previously mentioned within a DAXPY operation, the stride does not need to be a unit stride. A more general description of the DAXPY operation is shown below.

```
IY = 1
IX = 1
DO I = 1, N
   Y(IY) = Y(IY) + ALPHA * X(IX)
   IY = IY + INCY
   IX = IX + INCX
ENDDO
```

In this definition the stride for `Y` is `INCY`, and the stride for the `X` vector is `INCX`. There is still a performance improvement for prefetching values of the X vector. Since these strides may be greater than four, the value of unrolling the loop decreases. As the value of `INCX` and `INCY` increases, the benefit of prefetch decreases. Table 17 shows some of the performance values of DAXPY for various stride rates through the vectors Y and X. Performance is shown in Mflops, a more traditional measure value for DAXPY.

Table 17. Performance of DAXPY in Mflops for 1000 elements for various strides

INCY	INCX	Prefetch X	Prefetch Y	No prefetch
1	10	17.9	17.8	16.3
1	50	17.0	16.4	15.5
1	1000	11.1	11.1	10.43
50	1	14.6	14.6	14.2
50	100	9.5	9.5	8.9
50	1000	8.2	8.2	8.2
100	100	9.4	9.5	9.0

5.2 LU matrix factorization

The block LU matrix factorization is the most commonly used dense matrix linear solver. The block LU factorization method factors a submatrix and then uses the factored submatrix to update the remainder of the matrix. By operating in a block fashion, dgemm becomes the most commonly called subroutine and code profiles show that most of the time in block LU matrix factorization is spent in this subroutine. The dgemm routine, or matrix matrix multiply routine is rich in data reuse, which is why the block LU factorization executes faster than vector based DAXPY versions. For this study we used the block LU matrix factorization program LAPACK available from netlib. This program was built using the Pentium III 1.0 MKL with prefetch instructions and with the 2.1 MKL without prefetch instructions. We made changes to control matrix size and the block dimensions.

The performance improvement of DGEMM using Streaming SIMD Extensions varied depending on the dimensions of the three matrices involved in the operation. For this reason we felt the usage of LAPACK to measure the influence would be beneficial. In this algorithm, the width of the column block size used for factoring the matrix remains constant. The remaining two matrix dimensions change as the matrix factorization progresses. So the dimensions of the matrices entered in the DGEMM call varies throughout the algorithm. The results from running this program are illustrated Table 18. We discovered that the optimal block size increased when the BLAS routines with prefetch instructions were used. The optimal block size increased from 32 to 48 or 64.

Table 18. LAPACK LU matrix factorization.

Matrix Size	No Streaming SIMD extensions Block size = 16 **Mflops**	BLAS using streaming SIMD extensions Block size =48 **Mflops**	% increase	BLAS using streaming SIMD extensions Block size=64 **Mflops**	% Increase
1000	233	244	4.4	243	4.0
2000	251	274	9.2	267	6.6
3000	258	279	8.14	279	8.2
4000	262	285	8.7	285	8.8
5000	262	287	9.6	289	10.3

6 Conclusions

This study of the performance improvement provided by the memory cacheability instructions shows that vector operations can improve 24 to 85%. This is a significant improvement. The optimization requires only limited code changes. It is necessary to add several calls to the compiler intrinsics and adjust some loop controls also, but there is no need to change any of the data structures. This paper presents ten guidelines to follow when adding prefetch and streaming store instructions to other codes rich in vector/matrix operations. These ten guidelines are summarized below:

1. Unroll loops.
2. Optimize prefetch distance.
3. Minimize switching from reading to writing data.
4. Prefetch variables being read and not written.
5. Do not prefetch those variables that can benefit from streaming store.
6. Use temporary arrays to store data that is write-only and can benefit from the use of the streaming store instruction..
7. Combine prefetch and streaming store for optimal performance.
8. Ensure Page Table Entries are resident in the TLB.
9. Use prefetch only when appropriate.
10. Prefetch data that is not reused.

The programmer should be careful in testing each modification. In the course of this test we wrote the code to modify the position of the vector blocks and their alignment. Steps like this helped us uncover effects such as that shown in Section 4.2.8. These issues are important when the code will be incorporated into the full source code base. An important note is to not overdue the use of prefetch instructions. The use of prefetch instructions is not an exact science. It is important to add them incrementally and measure the effect of each change.

7 Acknowledgements

The suggestions and discussions on the advance work done by others were an assistance to our work. We acknowledge appreciate the informative discussions we had with Art Kauffman, Bruce S. Greer, Ram Ramanujam, and Kevin B. Smith. We also appreciate the efforts of those who reviewed previous drafts of this document.

8 Appendix A. Source Code

8.1 STREAM Kernels

```
INCR = 4;
INCRP4 = INCR + 4;
STRIDE = 4;
PAGESIZE = 512;
```

8.1.1 The COPY Kernel

```
      times[0][k] = second();
      for (kk=0; kk<N; kk+=PAGESIZE) {
            temp = a[kk+PAGESIZE];
      for (j=kk; j<kk+PAGESIZE; j+=STRIDE) {
//    c[j] = a[j];

        _mm_prefetch((char*)&a[j+INCR], _MM_HINT_NTA);

        _mm_stream_ps ((float*)&c[j],  _mm_load_ps((float*)&a[j]));
        _mm_stream_ps((float*)&c[j+2],_mm_load_ps((float*)&a[j+2]));
      }
      }
      _mm_sfence();
      times[0][k] = second() - times[0][k];
```

8.1.2 The SCALE kernel

```
      times[1][k] = second();
      _mm_prefetch((char*)&c[3], _MM_HINT_NTA);
      _mm_prefetch((char*)&c[7], _MM_HINT_NTA);
      for (j=0; j<N; j+=STRIDE) {
//          b[j] = scalar*c[j];

            _mm_prefetch((char*)&c[j+INCRP4], _MM_HINT_NTA);

            tarr[0] = scalar*c[j];
            tarr[1] = scalar*c[j+1];
            tarr[2] = scalar*c[j+2];
            tarr[3] = scalar*c[j+3];
            _mm_stream_ps ((float*)&b[j], *(__m128*)&tarr[0]);
            _mm_stream_ps ((float*)&b[j+2], *(__m128*)&tarr[2]);

      }
      _mm_sfence();
      times[1][k] = second() - times[1][k];
```

8.1.3 The SUM Kernel

```
        times[2][k] = second();
        _mm_prefetch((char*)&a[3], _MM_HINT_NTA);
        _mm_prefetch((char*)&b[3], _MM_HINT_NTA);
        for (j=0; j<N; j+=STRIDE) {
//              c[j] = a[j] + b[j];

                _mm_prefetch((char*)&a[j+INCR], _MM_HINT_NTA);
                _mm_prefetch((char*)&b[j+INCR], _MM_HINT_NTA);

                tarr[0]   = a[j]  +b[j];
                tarr[1]   = a[j+1]+b[j+1];
                tarr[2]   = a[j+2]+b[j+2];
                tarr[3]   = a[j+3]+b[j+3];
                _mm_stream_ps ((float*)&c[j], *(__m128*)&tarr[0]);
                _mm_stream_ps ((float*)&c[j+2], *(__m128*)&tarr[2]);

        }
        _mm_sfence();
        times[2][k] = second() - times[2][k];
```

8.1.4 The TRIAD Kernel

```
        times[3][k] = second();
        _mm_prefetch((char*)&b[0], _MM_HINT_NTA);
        _mm_prefetch((char*)&c[0], _MM_HINT_NTA);
        for (kk=0; kk<N; kk+=PAGESIZE) {
                temp = b[kk+PAGESIZE];
                temp1= c[kk+PAGESIZE];
        for (j=kk; j<kk+PAGESIZE; j+=STRIDE) {
//              a[j]   = b[j]  +scalar*c[j];

                _mm_prefetch((char*)&b[j+INCR], _MM_HINT_NTA);
                _mm_prefetch((char*)&c[j+INCR], _MM_HINT_NTA);

                tarr[0]   = b[j]  +scalar*c[j];
                tarr[1]   = b[j+1]+scalar*c[j+1];
                tarr[2]   = b[j+2]+scalar*c[j+2];
                tarr[3]   = b[j+3]+scalar*c[j+3];
                _mm_stream_ps ((float*)&a[j], *(__m128*)&tarr[0]);
                _mm_stream_ps ((float*)&a[j+2], *(__m128*)&tarr[2]);

        }
        }
        _mm_sfence();
        times[3][k] = second() - times[3][k];
        }

}
```

8.2 DCOPY

```
void dcopy(int *ptrsize, double *srcptr, int *pincx, double *dstptr, int
*pincy
{
   const double *src  ;
   double *dest ;
   int size ;
   int   i, j, isize, irem ;

   size = *ptrsize ;
   if (size > 0)
   {
      src = (double *)srcptr;
      dest = (double *)dstptr;
      sum = src[0] ;

     /* 16 byte alignment required for fastest copy - stream store ps */
     // aligned for ps copy
      if (((long)dest & 8) == 8)
      {
         *dest = *src ;
         ++src ; ++dest ;
         size-- ;
      }
      // Begin efficient memcopy
      isize = size>>9 ;
      irem  = size & 511 ;
      if (irem == 0)
      {
         isize-- ;
         irem = EL_PER_BLOCK ;
      }
      // - iterate over blocks of 4K each
      for( i = 0; i < isize; i++ )   // 256 __m128 writes each
      {
         // "tag" the next block to update the TLB for next 4K block
         //  This lets prefetch pre-loop work.
         sum = src[512] ;
         // do some NOPs to give TLB time to load before prefetch loop
         __asm {
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            nop
         }
         // NOW ON WITH THE REAL COPY
         // PREFETCH 4K PRE-LOOP - get 4K into L1 cache
         //   This avoids reading from memory during copy
         //   and avoids using fill buffers, which can
```

```
         //   cause partially-filled WC buffer flushes.
         for( j = 0; j < EL_PER_BLOCK ; j += 4 ) // prefetch 128 cache
lines
         {
            _mm_prefetch( (char *)(src + j), _MM_HINT_NTA );
         }

         // copy 128 cache lines from L1 (4096 bytes)
         for( j = 0; j < EL_PER_BLOCK ; j += 4 ) // copy 128 cache lines
         {
            _mm_stream_ps((float *)dest,  _mm_loadu_ps((float *)src ) );
            _mm_stream_ps((float *)&dest[2],_mm_loadu_ps((float
*)&src[2]));

            src += 4;
            dest += 4;

            __asm {   // NOPs seem to reduce partial WC buffer flushes?
               nop   // unclear why should be any.
               nop
               nop
               nop

               nop
               nop
               nop
               nop
               nop
               nop
            }
         }
      }
      for( j = 0; j < irem ; j += 4 )   // prefetch 128 cache lines
      {
         _mm_prefetch( (char *)(src + j), _MM_HINT_NTA );
      }
      isize = (irem & ~3) ;
      for( j = 0; j < isize ; j += 4 )
     {
         _mm_stream_ps((float *)dest,  _mm_loadu_ps((float *)src ) );
         _mm_stream_ps((float *)&dest[2],_mm_loadu_ps((float*)&src[2]));
         src += 4;
         dest += 4;
      }
      for (j = 0 ; j < irem%4 ; j++)
         dest[j] = src[j] ;
   }
   _mm_sfence();      // Make sure all streaming writes have finished

   return ;
}
```

8.3 DAXPY

```
void daxpy(int *n, double *alph, double *x, int *incrx, double *y, int
*incry)
{
   double alpha ;
   int i, in ;
   int incx, incy ;
   int irem,ix,j,iy ;

   if (*n >= 1)
   {
      incx = *incrx ;
      incy = *incry ;
      alpha = *alph ;
      if (incx == 1 && incy == 1)
      {
         _mm_prefetch((char *)x,_MM_HINT_NTA) ;
         _mm_prefetch((char *)&x[4],_MM_HINT_NTA) ;
         in = n[0] >> 9 ; // /512 ;
         irem = n[0] & 511 ; // %512 ;
         if (irem == 0)
         {
            in-- ;
            irem = 512 ;
         }
             j = 0 ;
         for (i=0 ; i < in ; i++)
         {
                iy = j + 512 ;
            for (; j < iy ; j+= 4)
            {
               _mm_prefetch((char *)&x[j+7],_MM_HINT_NTA) ;
               y[j  ] += alpha * x[j] ;
               y[j+1] += alpha * x[j+1] ;
               y[j+2] += alpha * x[j+2] ;
               y[j+3] += alpha * x[j+3] ;
            }
         }
             in = n[0] & ~ 3 ;
             for ( ; j < in ; j+= 4)
                 {
                _mm_prefetch((char *)&x[j+7],_MM_HINT_NTA) ;
                y[j  ] += alpha * x[j  ] ;
                y[j+1] += alpha * x[j+1] ;
                y[j+2] += alpha * x[j+2] ;
                y[j+3] += alpha * x[j+3] ;
                 }
         for ( ; j < n[0] ; j++)
         {
            y[j  ] += alpha * x[j] ;
         }
      } else {
          iy = 0 ; ix = 0 ;
         _mm_prefetch((char *)x,_MM_HINT_NTA) ;
         _mm_prefetch((char *)&x[incx],_MM_HINT_NTA) ;
         in = n[0] ;
         for (i=0 ; i < in ; i++)
         {
            _mm_prefetch((char *)&x[ix+2*incx],_MM_HINT_NTA) ;
            y[iy  ] += alpha * x[ix] ;
            iy += incy ;
```

```
            ix += incx ;
         }
      }
   } else {
      if (*n == 1) y[0] += alpha * x[0] ;
   }
}
```