Applications Tuning for Streaming SIMD Extensions

James Abel, Kumar Balasubramanian,
Mike Bargeron, Tom Craver, Mike Phlipot, Microprocessor Products Group, Intel Corp.

Index words: SIMD, streaming, MMX™ instructions, 3D, video, imaging

ABSTRACT

In early 1997, Intel formed an engineering lab whose charter was to apply a new set of instructions to the optimization of software applications. This lab worked with commercial software companies to increase the performance of their applications by using these new instructions. Two years later, this new instruction set has been made public as a principal new feature of the Pentium® III processor, the Streaming SIMD Extensions. Many of the commercial software companies' applications on which the lab consulted have been brought to market, demonstrating significant performance improvements by using the Streaming SIMD Extensions. This paper describes many of the principles and concepts developed as a result of that activity.

The Streaming SIMD Extensions expand the Single Instruction/Multiple Data (SIMD) capabilities of the Intel® Architecture. Previously, Intel® MMX™ instructions allowed SIMD integer operations. These new instructions implement floating-point operations on a set of eight new SIMD registers. Additionally, the Streaming SIMD Extensions provide new integer instructions operating on the MMX registers as well as cache control instructions to optimize memory access. Applications using 3D graphics, digital imaging, and motion video are generally well suited for optimization with these new instructions.

Data organization plays a pivotal role in the performance of applications in the above areas. This paper explores three data organizations (Array of Structure, Structure of Array, and Hybrid data orders) and their impact on SIMD processing performance. The impact of cache control instructions, such as the prefetch instructions, is also examined.

Examples of applying the Streaming SIMD Extensions to 3D transform and lighting, bilinear interpolation, video block matching, and motion compensation are considered. The principles applied in these examples can be extended to many other algorithms and applications.

INTRODUCTION

It is desirable to have many products available at the initial launch of a processor to help establish consumer interest. The development of these products begins with understanding the full potential of the new processor. This process requires optimizing select algorithms to achieve maximum performance. For the Pentium® III processor, that activity started in 1997 with a focus on the Streaming SIMD Extensions.

Although applicable to a wide variety of programs, the extended instruction set is designed to be especially effective in applications involving 3D graphics, digital imaging, and digital motion video. The purpose of this paper is to describe how those particular applications are best optimized with the new SIMD instructions.

Rather than optimize an entire application, specific algorithms or components were selected that would offer the best speedup. Analysis tools such as the VTune™ Performance Enhancement Environment [1] identified the most processor-intensive components of an application. The identified components were further examined for algorithms that execute similar operations on large data sets with a minimal amount of branching.

Data flow in and out of the processor is an important element in optimization so various data organization strategies were tested, including the impact of prefetch.

All algorithms were coded with and without the Streaming SIMD Extensions. The two versions of the algorithms were run on the same Pentium III processor platform to determine the relative performance difference.

DATA AND THE STREAMING SIMD EXTENSIONS

This section examines issues that must be taken into account to achieve the best possible performance with the Streaming SIMD Extensions. The order of data in memory and the methods by which such data are moved to and from the processor can have a significant impact.

SIMD and Memory Interactions

The floating-point instructions in Streaming SIMD Extensions generally operate "vertically"; that is, they operate between corresponding positions of the SIMD registers, or the equivalent positions of data being loaded directly from memory. Since the same operation must be done to all four floating-point values in a register, typically the best approach is to use each of the four positions of an SIMD register to store the same variable of an algorithm, but from different iterations. For example, if the algorithm is A[j] = B[j] + C[j], one would want to put four B's in one register, four C's in another register, and use a single SIMD add operation to create four resulting A's.

Each SIMD register can be thought of as a small array of data. A set of these registers can be thought of as a structure of arrays (SoA for short):

```
struct { float A[4], B[4], C[4]; } SoA;
```

This SoA approach is not always applicable. In the equation B[j] = B[j-1] + C[j], the dependency between iterations would require a different approach.

The Pentium® III processor typically loads data from memory 32 bytes at a time, from 32-byte aligned addresses. Each 32 bytes is stored to one "cache line" of the L1 and/or L2 caches. Frequent use of instructions that load or store data that is split over two cache lines will cause a significant performance penalty.

Most of the Streaming SIMD Extensions floating-point instructions that access memory require a 16-byte aligned address, thus avoiding the penalty. The `movups`, `movlps`, and `movhps` instructions were included to support unaligned accesses, at the risk of incurring the penalty. The `movlps` and `movhps` instructions can access 8-byte aligned addresses without penalty, since they move only 8 bytes at a time (compared to `movups` which move 16 bytes).

Using the Prefetch Instructions

Prefetch instructions can be useful for algorithms limited by CPU processing speed, ensuring that data is always ready as soon as they can be used. The prefetch instructions *load data ahead of use*, thereby hiding load latency so that the CPU can take full advantage of memory bandwidth. The Pentium III processor loads data to cache when a cache line is written to, so prefetches can also reduce latency for storing data.

Prefetch instructions can be useful for memory bandwidth-limited algorithms as well. For example, the `prefetchnta` instruction fetches data only to the L1 cache, avoiding some overhead incurred when data is also loaded to the L2 cache (as occurs with normal load and store).

Loading data only to the L1 cache, if it is not going to be needed again soon, also avoids unnecessarily evicting data from the L2 cache. When data is unnecessarily evicted, it can impose a double penalty. Modified cache lines that are evicted must be written back to memory and reloaded later when they are again needed.

The `prefetcht2` instruction might be used to load the L2 cache with a data set larger than can fit in the L1 cache. Meanwhile a CPU speed-limited algorithm could be executing and randomly accessing data. As it proceeds, it would find more and more of its data in the L2 cache.

To get the most efficient use of prefetch, loops should be unrolled (i.e., multiple passes of the algorithm should be in each loop iteration) so that each iteration prefetches and uses one cache line worth of each variable of the algorithm.

Data Order and SIMD Algorithm Performance

The SoA order is the most natural order for SIMD operations, so it would seem equally natural to use it as an order for data in memory:

```
struct
{
      float A[1000], B[1000], C[1000];
} SoA_data;
```

In some cases, this approach can work fine. But for a larger number of structure members, SoA can have memory access performance penalties. PC memory systems can only keep a limited number of pages (typically 4KB blocks) of memory "open" for fast access. If the number of members exceeds that number, so that each set of four values used in a SIMD computation must come from a different area of memory, the memory subsystem may spend inordinate amounts of time "re-opening" pages.

An Array of Structures (AoS) data order is more conventional in non-SIMD programming:

```
struct
{
      float A, B, C;
```

```
} AoS_data[1000];
```

Sequential processing through an AoS data set will find needed data close together in memory, thus avoiding the "open pages" limitation of SoA. However, the data are clearly not well ordered for SIMD computations.

The solution to this is generally to load the AoS data into SIMD registers and convert them to the SoA format via data reordering ("shuffling") instructions. This process can be thought of as "transposing" the data order. See Figure 1.

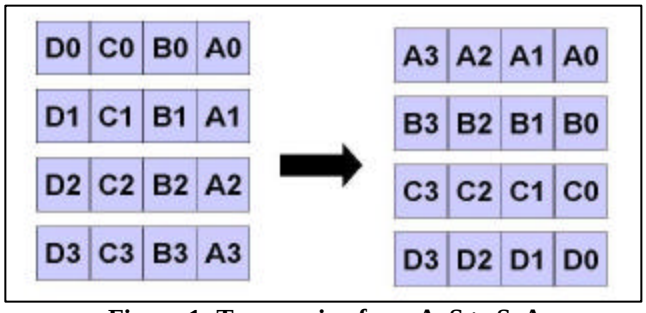

Figure 1: Transposing from AoS to SoA

While there is some performance cost due to this transposition, this approach generally works reasonably well, and may be the only viable solution if other factors mandate an AoS data order in memory. Existing data may be in AoS format; existing programs may have interface specifications that require AoS data; or data may be randomly accessed rather than sequentially accessed.

The Pentium III processor loads 32 bytes at a time from memory to cache. If there are members in an AoS structure that are not needed in the current computation, they will nonetheless be pulled across the memory bus, incurring unnecessary bus overhead, and limiting performance.

Data caching can sometimes offset this overhead by keeping data in cache until they are needed in a subsequent processing step. But for large data sets, the cache may not be large enough, and data may be evicted before they can be used. In general, it is a good idea to limit AoS structures to just members that will often be used at the same time. (This applies to non-SIMD code as well.)

It would be preferable, when AoS is not forced on us by external factors, to find a data order that preserves the AoS data adjacency, while supporting the SoA load order. An example of this "hybrid" data order is

```
struct
{
        float A[8], B[8], C[8];
} Hybrid_data[125];
```

As with SoA, this order allows the processor to load four values at a time (e.g., with `movaps`) from any member array. While structure members with the same index are not immediately adjacent, they are still close enough that they will usually be in the same memory page.

If a hybrid structure starts 32-byte aligned, the data will remain 32-byte aligned (since there are eight entries in each of the 4-byte float sub-arrays). This is convenient for Streaming SIMD Extensions instructions that require 16-byte alignment, as well as for prefetching a full cache line that contains just one particular member. For sequential processing of large data sets, it reliably provides good results.

Figure 2 illustrates the impact of SIMD and data order on the performance of a dot product algorithm.

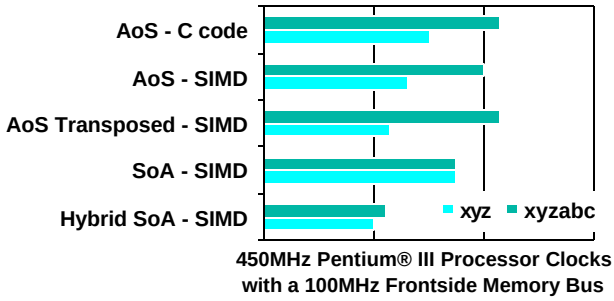

Figure 2: Data order and performance

A dot product was done between vectors of two large sets of 3-component (xyz) vectors. All were coded in C: The Streaming SIMD Extensions versions were implemented using "intrinsic functions" built into and optimized by an Intel® compiler. All use prefetch instructions to optimize the use of memory bandwidth. The "xyz" bars represent tests with data structures having only three members in the data structure, while the "xyzabc" bars represent tests where three extra structure members, not involved in the dot product, were included in the data set.

The Hybrid SoA approach gave the best overall performance. AoS algorithms did poorly (and became memory bound) when extra members were included in the same structure. The SoA and Hybrid SoA algorithms were nearly immune to extra structure members. The Streaming SIMD Extensions provided some small benefit to the AoS 'xyz' algorithm if the dot products were done one at a time (AoS – SIMD), and somewhat more if the data were

transposed to SoA form before processing. The SoA algorithm was fully memory limited; it was unable to approach the best performance despite the natural SIMD ordering of data.

Optimizing Memory Use for Block Processing

Block processing algorithms typically read sequentially through a large array of data, modify the data, and write out to another large array. For example, converting an image from RGB format to YUV format entails sequentially reading the RGB components, computing the equivalent YUV components, and writing the latter out to a new array. An even simpler example is a block copy.

Many such algorithms will be memory bound, so anything that optimizes the flow of data is highly desirable. One attribute of such algorithms is that typically they process data once, and need not touch them again. In such a case, there is little point in saving results in cache, where they might displace other useful data.

The streaming store instructions (`movntps`, `movntq`) can be used to write results to the destination memory buffer without going through the caches. However, these instructions work best if they can take full advantage of write combining. Any access to memory or the L2 cache can cause premature flushing of the write-combining buffers, resulting in inefficient use of the memory bus. While writing out results with the streaming store instructions, data should only be read from the L1 cache, to avoid this performance penalty.

To ensure this is the case for a block-processing algorithm, a loop can be added that uses `prefetchnta` to read a sub-block of data (typically about 4KB) into the L1 cache. A normal processing loop would follow this, reading the L1 cached sub-block of data and writing results out with streaming store instructions. An outer loop, around both of the sub-block loops, would go through all the sub-blocks that make up the data set. One key issue arises when using this approach. The prefetch instructions only work when the virtual memory page addressed by the prefetch is mapped to a physical memory page by the Translation Lookaside Buffer (TLB) in the Pentium III processor. Typically the TLB is updated for a page the first time that page is accessed. Since the TLB has a finite number of entries (e.g., 64), page mappings that have not been used recently may no longer be cached in the TLB, which means the prefetch will not work.

To make sure the prefetches work, one merely needs to do one read from each 4KB memory page of the source data, shortly before starting the prefetch loop. Since initializing the TLB will take a while, if the read is done right before the prefetch loop, many of the prefetches might be quickly executed with no effect. This can be adjusted for in a variety of ways. For example, one can read once from an address 4KB ahead of the address where the prefetch loop begins, making sure not to read past the end of valid data.

The approach of breaking data into cache-fitting sub-blocks can also be useful if one wishes to do multiple passes over data that cannot all fit into cache. For example, one might wish to do a sequence of processing steps, each taking the previous step's output as its input.

If one were to do each processing pass separately, intermediate results would have to be written out to memory and later reloaded from memory for the next step. Instead, one can often do all passes over each of many smaller, cache-fitting blocks, thereby minimizing memory data bus traffic.

TUNING 3D APPLICATIONS

In a typical 3D geometry engine, one would expect to find various functional components such as transformation, lighting and shading, clipping, culling, and perspective correction modules [2]. Deciding which component should be optimized can be difficult. Using the criteria discussed in the introduction, the transform and lighting functions were determined to be good candidates for optimization using the Streaming SIMD Extensions.

Transform and lighting functions are compute-intensive, SIMD-friendly inner loops that perform the same operation on large amounts of contiguous data. Use of the prefetch instruction allows data in either loop to begin loading several iterations prior to their use. The new approximation instructions are beneficial in eliminating long-latency square-root and division operations in the lighting loop, or in the transform loop when perspective correction is performed. Finally, clamping to a range of values within the lighting loop can be replaced by the new packed min/max instructions, eliminating two unpredictable branches per iteration.

The details of each optimization method are discussed below. In each case, data order and alignment are as discussed in the previous section.

3D Transform

The 3D transform is performed by multiplying a 4x4 transformation matrix by a 4-element vector. The vector is comprised of vertex elements X, Y, Z, and the constant 1, while the transformation matrix itself is calculated individually for each object in the scene. This operation produces intermediate values X', Y', Z', and W'. In some cases, the W' value is immediately used to normalize the intermediate vector (perspective divide), generating final values X", Y", and Z". Since the final result of the fourth

element is always 1.0, the division of W' by itself can be ignored.

3D Transform Optimizations

Because the same transformation matrix applies to all vertices of a given mesh, and since there is a large amount of data to be processed, the transform was found to be a good candidate for SIMD programming. To experience the largest benefit from the Streaming SIMD Extensions, the full capacity of the SIMD registers was exploited. One register was loaded with four X values, X_0, X_1, X_2, and X_3, another with four Y values, and yet another with four Z values.

The first set of matrix elements was then loaded into a fourth register. To do this, two methods were possible. Each matrix element could either be (1) stored as a single floating-point value in memory, read into the lowest position of the 4-wide register using `movss`, then replicated four times using the `shufps` instuction; or (2) the element could be stored as an array of four identical floating-point values, aligned on a 32-byte boundary and read in four at a time using the `movaps` instruction. The latter proved to be optimal. Storing the entire matrix in this manner did increase the immediate size of the structure from 64 bytes to 256 bytes, but this was a small price to pay for the performance gained.

Matrix elements m_{01}, m_{02}, and m_{03} were loaded from memory in a similar fashion. With all of the data in registers in true SIMD format, the transform became a simple series of three multiply instructions followed by three addition intructions for each set of results (see Figure 3). Knowing that the same vertex data used to calculate the X' results would be needed to compute the Y' results, instructions were used in such a way as to overwrite the registers containing matrix data. Once this set of computations was completed, intermediate values X_0', X_1', X_2', and X_3' were written to a 32-byte aligned output buffer.

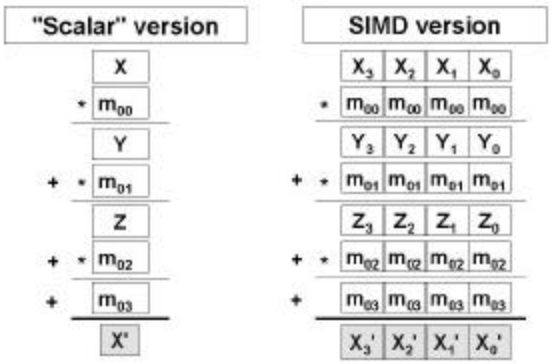

Figure 3: The SIMD transform produces four results while the conventional transform produces only one

The process was repeated for the first four Y values, this time loading matrix elements m_{10}, m_{11}, m_{12}, m_{13}, and again for Z' and W', with each of their respective matrix elements. The final result of the first transform iteration was 16 intermediate values. If at this point of the pipeline, a perspective divide is done, the `rcpps` instruction is of tremendous benefit. (See the section entitled *3D Lighting Optimizations* for more details.) Figure 4 compares the results of the Pentium III processor-specific code with the optimizations discussed above and a standard 'C' implementation of the same algorithm.

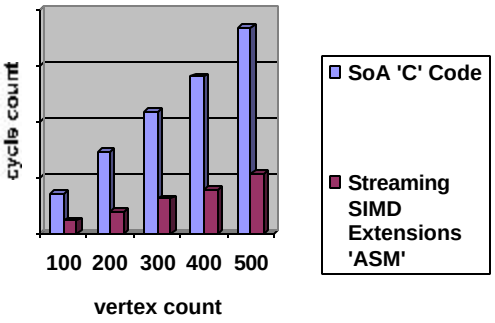

Figure 4: Transform cycle time for optimized Pentium® III processor assembly versus conventional 'C' code

3D Lighting

The point light is probably the most widely used light source in 3D graphics applications. To apply a point light to a given vertex, the vector from the vertex to the light source is first calculated. The length of this vector is computed and used to normalize the vertex-to-light vector. From the normalized light vector, the *diffuse* component of the current vertex is computed. Finally, the overall vertex color is calculated and checked to ensure that it falls within the range of [0.0, 1.0]. Values exceeding the range

in either direction must be "clamped" to either the upper or lower bound. Once vertex color values are computed and clamped, they are stored to memory for use at render time.

3D Lighting Optimizations

The lighting distance calculation involves a standard square-root of a sum-of-squares. Again the full capacity of SIMD registers was used by loading four vertex values into each register and calculating the distance of four vertices from the same light source simultaneously. The normalization of the light vector was then performed by dividing the vector itself by the previously calculated distance. Though seemingly simple, these steps require two long-latency floating-point operations, the square-root and the divide, each of which requires about 36 clock cycles to complete. Beyond that, they are "unpipelined" instructions, meaning that no other instruction may be submitted to the execution port on which they are executing until the given instruction retires.

To overcome this, the Streaming SIMD Extensions include `rsqrtps` and `rcpps`, reciprocal approximation instructions that allow developers to accomplish the same workload as the long-latency instructions in a shorter time. By performing hardware table look-ups, these instructions have a reduced latency of two clock cycles and are fully pipelined, so that other operations may be issued during their execution. As a tradeoff, the instructions guarantee at least 11 bits of mantissa precision, as opposed to the full 23 bits offered by true single-precision instructions. Though 11 bits is typically enough for most 3D applications, some applications may require (or prefer) more.

The Newton-Raphson method is a mathematical algorithm designed to regain precision lost by this type of approximation. A single-pass Newton-Raphson iteration doubles the resultant accuracy to 22 bits; the 23rd bit can be recuperated after a second pass. An approximation followed by a single iteration is notably faster than its long-latency equivalent and can be determined by

$$rcp'(a) = 2*rcp(a) - a*rcp(a)^2$$

$$rsqt'(a) = (0.5)*rsqt(a)*(3-a*rsqt(a)^2)$$

Four normalized direction vectors were generated by multiplying the reciprocal square root of each distance squared by the previously calculated light vector. These values were then used to calculate the diffuse component of the vertex color.

The diffuse calculation involves a dot product of two vectors and could potentially produce a negative result. Graphics applications typically overcome this by performing a less-than-zero check and setting the value to zero if 'true.' This type of unpredictable, conditional branch performed once per iteration can be a performance bottleneck. To eliminate the branch, a `maxps` instruction was performed on the register of four diffuse results and a 4-wide register of 0.0 values. The max instruction zeroes out the negative values, while permitting non-negatives to pass through unchanged.

A similar optimization can be performed when calculating the final vertex color. This time, the tendency is for the value to exceed 1.0. The `minps` instruction clamps values, above the threshold, to 1.0 without the use of conditional branching. Large advantages of lighting optimizations versus a standard 'C' language lighting function are shown in Figure 5.

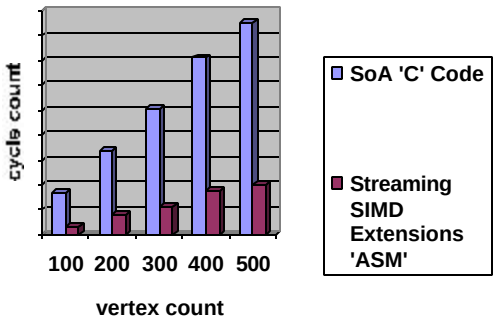

Figure 5: Lighting cycle time for optimized Pentium® III processor assembly versus conventional 'C' code

DIGITAL IMAGING

Digital imaging applications are typically comprised of algorithms wherein a small set of mathematical operations needs to be performed on large volumes of pixel data. Furthermore, each pixel consists of four components: Red, Green, Blue, and Alpha values. Hence, MMX™ technology significantly enhanced the performance of digital imaging applications. As most state-of-the-art imaging applications continue to embed richer video and graphics capabilities, the applications continue to demand much higher performance. The Pentium® III processor, with its associated Streaming SIMD Extensions, helps meet these new performance goals. The remainder of this section discusses how some of these new features help digital imaging algorithms enhance performance beyond those already achieved through MMX technology.

INTEGER SIMD EXTENSIONS

When implementing an SIMD imaging algorithm, one often encounters the need to rearrange data within an MMX™ register. The integer SIMD extensions include a shuffle instruction (`pshufw`) to enhance the performance of such frequently used operations. For example, an

efficient SIMD implementation of alpha saturation would compare all of the R, G, and B components with the corresponding alpha value in parallel. To be able to do so, the alpha value itself needs to be replicated in a different MMX register as shown in Figure 6.

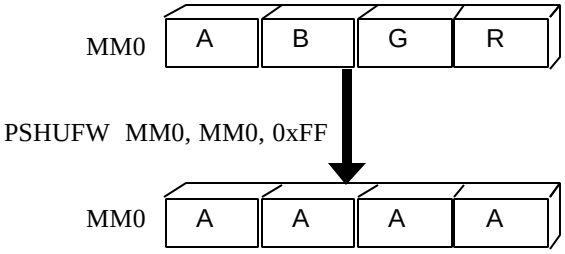

Figure 6: Broadcast alpha value

While this requires three instructions in MMX technology, the new instruction set would need just one.

Quite often, data-dependent branching has been an impediment in the process of mapping certain imaging algorithms to SIMD. For example, after computing an intermediate set of RGBA values, another set of computations might need to be executed if any of the R, G, B, and A values were below a certain threshold value. In a typical MMX implementation, the result of the condition check would be multiple mask patterns within an MMX register. However, extracting the required bits from these mask patterns into a register that can be used for addressing is usually very cumbersome. In certain cases, this might even negate the performance gains from an SIMD implementation of the algorithm. The new instruction `pmovmskb` addresses precisely this need. It extracts the required bits from the mask patterns in the MMX register and places them in a register that can be used for addressing.

Table look-up operations, such as the ones found in histogram-related algorithms, have always been critical to the performance of digital imaging. In such cases, each of the computed R, G, and B values is used as an index into its respective color look-up table. Such operations have been difficult to implement in MMX technology due to the fact that the computed RGBA values would be residing in an MMX register, which could not be used directly for addressing. Extracting each of them into the appropriate registers for addressing, fetching the contents from the table, and inserting them back into MMX registers was cumbersome and detrimental to performance. The integer SIMD extensions include a pair of instructions (`pinsrw/pextrw`) that helps enhance the performance of such algorithms.

In addition to the instructions mentioned above, the new integer SIMD extensions include several others that help enhance the performance of frequently used imaging algorithms. For example, the SIMD *unsigned multiply* instruction helps in the implementation of certain filter operations that were cumbersome using MMX technology. Likewise, the minimum/maximum instructions are useful during alpha saturation for bound checks, and the complete set of comparison operators facilitate all condition checking.

SIMD FLOATING-POINT

Current imaging implementations primarily involve fixed-point integer arithmetic. However, most state-of-the-art imaging applications are increasingly richer in their graphics capabilities and in their image quality. The algorithms therein should benefit significantly from the SIMD floating-point capability of the Streaming SIMD Extensions. Even if the underlying algorithms are implemented in floating-point, the enhanced floating-point performance helps yield near real-time response to typical user requests. Also, for intermediate results, the extra bits of available precision in a floating-point representation (relative to 16-bit fixed point) helps yield superior image quality. Moreover, implementing the algorithms in floating-point form reduces the need to deal with fixed-point arithmetic. This greatly boosts productivity by easing the task of code development, debugging, and maintenance. In imaging algorithms, the fundamental data object (RGBA pixel value) is of type integer. However, for the above-mentioned reasons, such as the need for extra precision and programming ease, several data transformations are implemented in floating-point. SIMD floating-point capability significantly enhances the performance of these implementations. The following bilinear interpolation example helps illustrate the usage of some of these SIMD floating-point instructions and also highlights some of the performance tradeoffs involved.

Bilinear Interpolation Example

The RGBA value of each pixel in the display image is calculated by a bilinear interpolation using RGBA values of four neighboring pixels in the source image (see Figure 7).

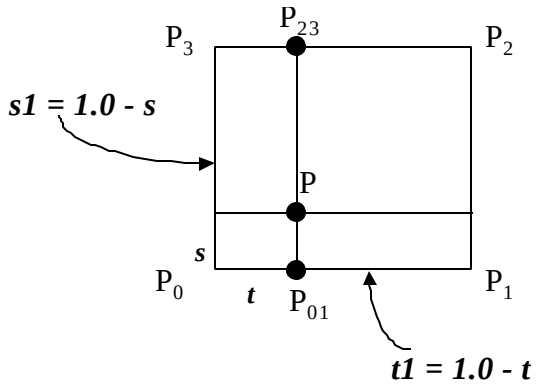

Figure 7: Bilinear interpolation

The R component of pixel P is calculated as follows:

$$R_{01} = t1 * R_0 + t * R_1$$

$$R_{23} = t * R_2 + t1 * R_3$$

$$R = s1 * R_{01} + s * R_{23}$$

From the above equations, it is evident that the bilinear interpolation steps involve a series of three linear interpolations. Each linear interpolation itself involves two multiplications and one addition for each value of R, G, B and A. Of course, when implemented in SIMD, all the four RGBA components can be computed in parallel. Initially, let us assume that we would like to perform these computations in floating-point since the SIMD floating-point capability might help us meet our performance goal. If so, as a first step, we will need to convert the RGBA pixel values from their typical byte representation to their float format. The steps involved in this are given in Figure 8. This conversion needs to be done for each of the four pixels in the source image. Note that both MMX™ technology and SIMD floating-point instructions are used in these steps, as are the MMX registers and the new Pentium® III processor registers. Overlapping the conversion steps for the four pixels better exploits available hardware as both the floating-point SIMD and the integer SIMD units will be operating in parallel.

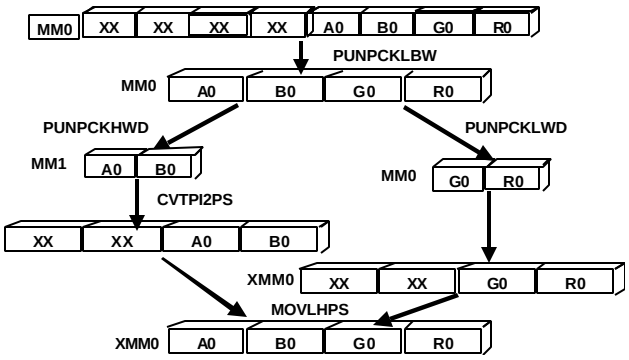

Figure 8: Packed byte to float conversion

Subsequent to this type conversion, the actual multiply-add step for each linear interpolation becomes relatively trivial (see Figure 9). Now, since the RGBA value of the result pixel is in float format, it needs to be converted back to integer type. The steps involved here are similar to those shown in Figure 8.

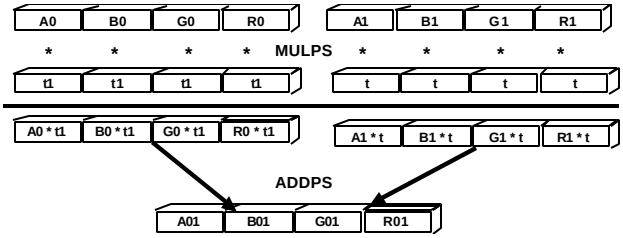

Figure 9: Linear interpolation

Analyzing the implementation indicates that the algorithm inherently required about nine basic instructions: two MULS and one ADD for each of the three linear interpolations. The decision to implement it using SIMD floating-point added about 29 additional instructions (six for each of the four source pixels from byte->float and five for the result display pixel from float->byte). However, the application would often perform several other floating-point operations such as lighting or other effects on the bilinearly interpolated pixel. In such cases, the byte<->float conversion time overhead can be amortized across all these additional floating-point operations. This helps yield enhanced performance using SIMD floating-point.

CACHE CONTROL INSTRUCTIONS

Given the typically large data sets in imaging, efficient cache utilization has a significant impact on performance. The Streaming SIMD Extensions have a few cache control instructions that help better utilize available hardware resources and minimize cache pollution. The different prefetch instructions help fetch data from memory to the

different relevant levels in cache sufficiently in advance of their actual usage. For example, in a tile-based imaging architecture, while the execution units of the processor could be busy processing a certain tile's data, the memory subsystem could be busy *prefetching* the next tile's data. Likewise, when the final display pixel values have been computed, the streaming store instructions could be used to store them directly in memory without first fetching them into cache. This also helps minimize the potential for valuable data already in cache and needed for other computations from being evicted out of the cache.

To maximize the benefits from these cache control instructions, careful attention should be paid to issues such as identifying the data sets worth prefetching, the cache levels to prefetch to, and when to issue the prefetch.

VIDEO CODECS

Video codecs, such as MPEG and Digital Video (DV) codes, can obtain a performance increase by using streaming SIMD extensions. Table 1 gives examples of these increases.

	PSADBW	**PAVG**	**Prefetch, Streaming Stores**
Encode	Motion Estimation	Motion Estimation, Motion Compensation	Color Conversion, Motion Compensation
Decode		Motion Compensation	Color Conversion, Motion Compensation

Table 1: Uses of Streaming SIMD Extensions for video codecs

MOTION ESTIMATION

Block matching is essential in motion estimation. Equation 1 is used to calculate the Sum of Absolute Differences (also referred to as the Sum of Absolute Distortions), which is the output of the block-matching function.

$$SAD = \sum_{i=0}^{15}\sum_{j=0}^{15} \left| Block_{ref}[i][j] - Block_{pred}[i][j] \right|$$

Figure 10 illustrates how motion estimation is accomplished.

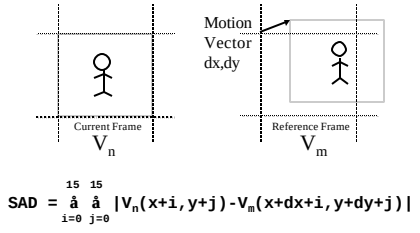

Figure 10: Block matching

`dx` and `dy` are candidate motion vectors. Motion estimation is accomplished by performing multiple block matching operations, each with a different dx,dy. The motion vector with the minimum SAD value is the best motion vector.

Streaming SIMD Extensions provide a new instruction, `psadbw`, that speeds up block matching. The operation of this instruction is given in Figure 11.

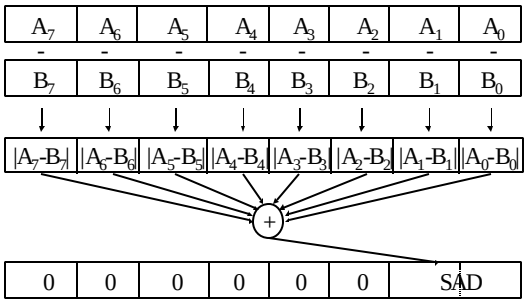

Figure 11: PSADBW

Block matching is implemented by using the PSAD instruction as illustrated in Figure 12. The code to perform this operation is given in Table 2. This code has been observed to provide a performance increase of up to twice that obtained when using MMX™ technology.

Note that the nature of memory access of block matching will cause data cache line splits when the loads straddle 32-byte boundaries. This is due to the dx, dy changes of 1 (i.e., address variances are one byte at a time). The data loads are eight bytes at a time.

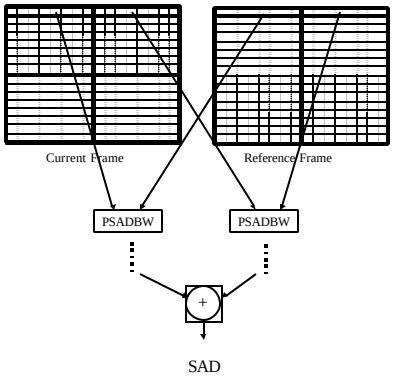

Figure 12: Block matching with PSADBW

```
psad_top:                  // 16 x 16 block
matching
     //  Do  PSAD  for  a  row,  accumulate
results
    movq mm1, [esi]
    movq mm2, [esi+8]
    psadbw mm1, [edi]
    psadbw mm2, [edi+8]

    // Increment pointers to next row
    add esi, eax
    add edi, eax

    // Accumulate diff in 2 accumulators
    paddw mm0, mm1
    paddw mm7, mm2
```

```
    dec  ecx         //  Do  all  16  rows  of
macroblock
    jg psad_top

    //  Add  partial  results  for  final  SAD
value
    paddw mm0, mm7
```

Table 2: Block matching

Hierarchical motion estimation is a popular technique used to reduce computational complexity and to provide potentially better motion vectors. Subsampling is illustrated in Figure 13.

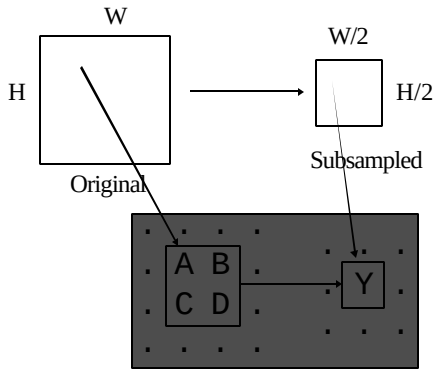

Figure 13: Subsampling for hierarchical motion estimation

Subsampling the original picture is sped up using the pavg instruction. Figure 14 shows the operation of pavgb.

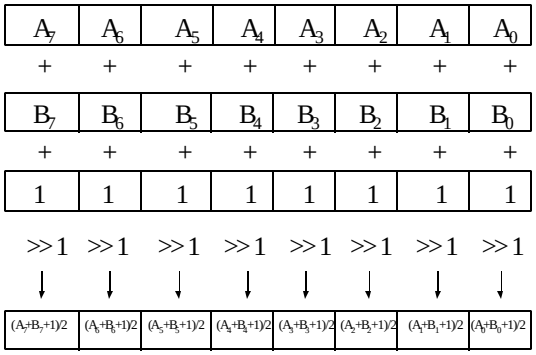

Figure 14: PAVGB

The `pavgw` instruction is also provided in streaming SIMD extensions. It works like the pavgb instruction, but performs the averaging on four 16-bit values.

It is important to note that the additions are performed with an additional bit for accuracy (9 bits for `pavgb`, and 17 bits for `pavgw`). This avoids overflow errors. Once the average is performed (after the divide-by-2), the width of the result is the same as the input (8 or 16 bits).

While the `pavg` instructions operate on two values at a time, it is possible to use three `pavg` instructions to approximate 4-value averaging. The line below illustrates this in pseudo-code:

```
Y = pavg(pavg(A,B),pavg(C,D)-1)
```

This value is close to (A + B + C + D + 2)/4 which is the typical calculation used to perform subsampling. However, for the approximation, 87.5% of values match exactly, and 12.5% of the values are off by one least significant bit (LSbit). The maximum error is one LSbit. This error is often acceptable for performing motion estimation.

MOTION COMPENSATION

Motion compensation (MC) is used in both video decoders and encoders. Decoders perform inverse motion compensation (iMC), and encoders perform both MC and iMC. The accuracy of these calculations is important, especially for encoders, since their local decoder should track the operation of a high-quality decoder. In MC, bi-directional B-frames can require interpolation of two values. The MPEG standard specifies this as

$$Y = (A + B + 1)/2$$

The `pavg` instructions provide exactly this calculation.

Streaming SIMD extensions also provide prefetch and streaming store instructions. Since MC is often memory bound, prefetch operations can speed up MC. `prefetchnta` and `prefetcht0` have both been observed to provide a speedup. Which one offers the best improvement is dependent on how the decoder is implemented. For decoders that are writing the decoded picture to a graphics card memory, `movntq` (move non-temporal quad-word) can offer a benefit by not polluting the caches with data that will never again be needed by the decoder.

DISCRETE COSINE TRANSFORM

The Discrete Cosine Transform (DCT) and inverse Discrete Cosine Transform (iDCT) are used in video codecs. Decoders use the iDCT, and encoders use the DCT and usually the iDCT (if they have a local decoder). It is possible to gain a speedup from streaming SIMD extensions; however, the speedup is application-dependent. The SIMD floating-point instructions can be used to calculate a very accurate DCT/iDCT. However, it is possible to be IEEE 1180-1990 [3] compliant using SIMD integer instructions, such as those found in MMX™ technology. In general, for consumer electronics versions, SIMD integer implementations are sufficiently accurate and are the fastest. For professional or reference codecs, SIMD floating-point may be the preferred choice. To ease the burden on codec developers, both of these implementations are available in Intel's Image Processing Library.

VARIABLE LENGTH ENCODE

Encoders must create a bit stream based on the values after the Discrete Cosine Transform and quantization. This is called the Variable Length Encode (VLE). Often, especially in the case of B-frames, there are many zero values that must be detected and "skipped over." To aid in the processing of these values, the `pmovmskb` instruction can be used to evaluate eight values. Table 3 illustrates how `pmovmskb` can be used for this.

```
pxor      mm7,mm7    // zero mm7
movq      mm0,[esi]  // get eight Q values
pcmpeqb   mm0,mm7    // find zeros
pmovmskb  eax,mm0    // 8 flags into eax
```

Table 3: Variable length encode

If `eax` holds 0xff, then all eight values are zero.

COLOR CONVERSION

Color conversion is used by both encoders and decoders. Often encoders receive data in a format other than what they can directly encode (wrong chromenance space, interleaved vs. planar data, etc.). Decoders sometimes have to write the decoded picture to a graphics card's memory in a color space other than the color space that naturally is produced from the decode; this also requires a color conversion.

For encoders, color conversion is typically a memory-bound operation. It loads picture data from main memory (i.e., DMA'ed in from a video capture card), performs some (typically simple) calculation, and writes the data back out to memory. `prefetchnta` can speed up color conversion by bypassing the L2 cache on the load. The non-temporal prefetch is often the best prefetch for color conversion since the input will not be needed again by the codec. The store can then be performed using a normal store (e.g., `movq`) so the picture resides in L2 cache after the color conversion.

CONCLUSION

The order in which data is stored in memory, and how it is moved to and from the processor and its caches, can have a significant impact on the performance of an application. While the hybrid data order is technically the best overall match for SIMD, if an application must use the conventional array of structures order, it is generally best to transpose the data into the structure of arrays order in the SIMD registers for processing. The prefetch instructions can often reduce memory latency or optimize memory bandwidth. When processing large blocks of data, splitting the data into subsets that fit the Pentium® III processor caches can avoid unnecessary memory overhead.

Managed use of memory and the 4-wide SIMD registers provide big benefits in the 3D transform. The results of the transform code tested showed an improvement of 3.0x to 3.7x for Pentium III processor-optimized assembly code over standard 'C' code. 3D Lighting also showed

significant gains (~4x) through the use of approximation and branch-elimination instructions.

The integer extensions ease implementation of typical imaging algorithms in SIMD while also extending their performance beyond those achieved through MMX™ technology. Likewise, the floating point SIMD, when used appropriately, enhances the accuracy and performance of algorithms with floating-point implementations. Moreover, it eases code development and validation by reducing the need to deal with fixed point arithmetic. Several of these techniques have been successfully applied in the high-performance Image Processing Library which is part of the Intel® Performance Library Suite [4].

Streaming SIMD Extensions can be used to greatly speed up functions commonly found in video codecs. These functions include motion estimation, motion compensation, variable length encode, and color conversion. The new `psadbw`, `pavgb`, and `pavgw` instructions, as well as prefetch and streaming stores, are paticularly useful for video codecs. Speedups of 2x have been observed for motion estimation, and speedups of 1.3x have been observed for entire encoder applications.

ACKNOWLEDGMENTS

The tuning concepts contained in this paper include refinements based on the optimization work of Intel engineers and organizations from groups such as the Microprocessor Labs, the Folsom Design Center, and Developer Relations and Engineering.

REFERENCES

[1] J. Wolf "Programming Methods for the Pentium® III Processor's Streaming SIMD Extensions using the VTune™ Performance Enhancement Environment," Intel Technology Journal, Q2, 1999.

[2] A. Watt, *3D Computer Graphics 2nd Edition*, Addison-Wesley Publishers Ltd., Essex, England.

[3] IEEE Circuits and Systems Society, IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform, *IEEE Std. 1180-1990*.

[4] http://developer.intel.com/vtune/

AUTHORS' BIOGRAPHIES

James Abel focuses on software applications for future Intel® processors. In his ten years at Intel, he has held several software and hardware positions, including the development of Intel's software Dolby* Digital decoder, embedded microcontroller design, and Design Automation. James obtained a B.S. degree in engineering from Bradley University in Peoria, Illinois, in 1983 and an M.S. degree in computer science from Arizona State University in 1991. His e-mail is james.c.abel@intel.com.

Kumar Balasubramanian works with software developers to help their applications take advantage of Intel's new processor capabilities. He managed the integration of the Streaming SIMD Extensions into several business applications. Kumar has been with Intel for seven years and has held leadership roles in Intel's CAD engineering organization and with Intel Architecture Labs to develop some of the first applications using MMX™ technology. He has an M.S. degree in computer engineering from Dartmouth College. His e-mail is kumar.balasubramanian@intel.com.

Mike Bargeron obtained a B.S. degree in electrical engineering from Brigham Young University. He started with Intel's Software Performance Lab in 1997. Since coming to Intel, Mike has been involved in performance tuning 2D and 3D graphics applications for the PC. Specifically, he has worked with MPEG motion video as well as several 3D game titles. His e-mail is michael.l.bargeron@intel.com.

Tom Craver works with 3D graphics IHVs to help them optimize their driver software on Intel's latest processors. Previously he developed and validated driver and user interface software for cable modems and for Intel's DVI multimedia technology. Prior to joining Intel, Tom was a member of the technical staff at the David Sarnoff Research Center in Princeton, NJ, and before that, he was with AT&T's Bell Laboratories. Tom holds B.S. degrees in physics and computer science from the University of Illinois. He also has a M.S. degree from Purdue University. His e-mail is tom.r.craver@intel.com.

Mike Phlipot works with desktop software developers to integrate Intel's newest processor capabilities into their applications. Most recently he has been helping 3D game developers take advantage of the Streaming SIMD Extensions. In his ten years with Intel, he has held various engineering and management positions in technologies that include digital video compression and cable modems. Mike has a B.S. degree in mechanical engineering from General Motors Institute and a M.S. degree in computer engineering from the University of Michigan. His e-mail is mike.p.phlipot@intel.com.

---

*All other brand names are the property of their respective owners.