Practical Altivec Strategies

The Why, How and When of Optimization Success and Failure Using Altivec

Ian Ollmann, Ph.D.
iano@cco.caltech.edu

Abstract

Despite the widespread availability of Altivec enabled processors, proven performance advantages and high end user demand for Altivec accelerated applications, most PowerPC software contains little or no Altivec code. The largest single barrier to widespread adoption seems to be developer awareness of programming tools, techniques and proven software design paradigms required to become successful with SIMD. This paper discusses key hardware features that affect Altivec performance, presents software architecture principles based on these that work well with Altivec, and concludes with a discussion of a practical strategy for incorporating Altivec into your existing application.

Introduction

The PowerPC's Altivec unit (a.k.a. Velocity Engine or vector unit) is a Single Instruction Multiple Data (SIMD) unit separate from the existing integer and floating point units (scalar units). The most significant difference between the Altivec unit and the scalar units is that the Altivec register is 128 bits wide and can hold many data at once: 128 bits, 16 chars, 8 shorts, 8 sixteen bit pixels, 4 longs, 4 32-bit pixels or 4 floats.

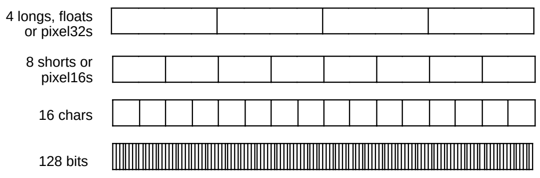


Figure 1. An Altivec register may be divided into four 32-bit types, eight 16-bit types, sixteen 8-bit types or 128 bits.

Each Altivec instruction will simultaneously operate in parallel on all of the data at once in about the same amount of time it takes to do the same operation on a single piece of data in the scalar units. For example, a simple addition operation might look like this:

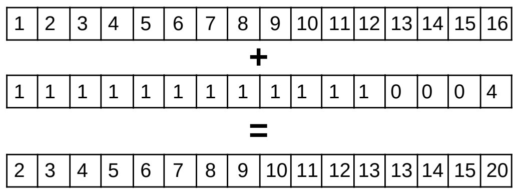


Figure 2. Altivec does single operations on multiple data in parallel.

...and the execution phase takes one cycle, just like ordinary integer addition.

The Altivec instruction set is fully equipped for any mathematical task with instructions for subtraction, multiplication, division, square roots, exponentials. In addition, it has Boolean operations, comparators, and type conversion functions, which have been the subject of a number of introductory articles on Altivec: [Bettag98], [Rosenberg99], [Motorola99a], [Motorola99b], [Clarke00], [Ollmann01], [Apple00].

Because of its inherent parallelism, Altivec code can in principle be 4, 8 or 16 times greater throughput than the traditional integer unit for integer math and four times greater throughput than the scalar floating point unit for floating point math. In addition, the Altivec instruction set adds a number of instructions for advanced data cache manipulation. This means that in some cases, performance increases can be even greater!

Speed-Critical Hardware Features

Sadly, new Altivec programmers are frequently disappointed to discover that their Altivec code is not much faster than their pre-existing scalar code. In some cases it may even be slower! Unfortunately, a number of time tested classical programming techniques have been found to be spectacularly unsuccessful when applied to Altivec. Many programmers find that they need to learn some new programming paradigms to make the most of the vector unit. First, however, a good understanding of processor hardware features is required to understand why Altivec code designs succeed and fail.

The Instruction Cache

When you enter a function, the instructions that make up that function have to be loaded in from memory before they can be executed. It is impossible for the whole program to fit in the instruction cache unless it is a trivially small program. Thus, it is not unusual that a function may have to be loaded in from main memory, which can be very, very slow. This means that often the first loop in any function may be executed considerably more slowly than the successive iterations. This will show up in Sim_G4, a processor simulator and code profiler highly useful for code optimization, as large gaps when no instructions are operating. Typically these stalls can be from 35-40 cycles per set of eight instructions. (Eight PPC instructions fill up one cache block.)

Knowing this, it may be worthwhile to position blocks of code that are frequently called near one another close to one another in memory. In addition, knowing that you are likely to lose some time the first time through a function, it may be a good idea to start any streaming data pre-fetch operations as soon as possible in the function. This will allow more data to be pre-loaded by the time the function starts operating at peak efficiency. Sometimes prefetching larger blocks at the start may help as well.

In addition, it doesn’t much matter how you write rarely called code as long as the instruction count is low. You have an average load time of four or five cycles per instruction. Most instructions only take one cycle to execute. Few take more than five. Thus, the largest speed determinant for uncached code is very likely to be the size of the code itself. This is one more reason to only optimize the small part of the program that consumes 80% of the CPU’s attention. Optimized code tends to be larger.

The Data Cache

The processor also has several caches for storing frequently used data. When a piece of data is needed, first the processor checks the L1 cache to see if it is there. If it is not, the aligned 32 byte block (a.k.a. cacheline) that contains that piece of data is loaded into the L1 cache.

The L1 cache is extremely fast. Most reads from it take only 2 cycles (3 on PPC 7450). By comparison, getting a cacheline from main RAM to the L1 cache so it can be used can take up to

250 cycles, though 30-80 is a more common number. Thus, proper use of the L1 is extremely important. Sim_G4 will quickly reveal any memory-related stalls that may be occurring due to poor use of the caches. You will see long stalls in `lvx` and `lvxl`.

Cachelines stored in the L1 cache are organized into 128 sets, each containing eight cachelines. Cachelines belonging to the same set all share the same address when masked with 0x0FE0. In other words, cachelines 4096 bytes apart map to the same set. When a new cacheline is loaded, the oldest cacheline in its set is flushed out of the L1 to make room, following a pseudo Least Recently Used algorithm.

When data is displaced out of the L1, it ends up in the L2 cache. This is the only way for data to end up in the L2 cache. The L2 is called a 'victim cache' for this reason. The L2 cache is much larger than the L1, but is only 2 way set associative (8 way on the PPC 7450) — two cachelines per set. If you have a particularly large set of data that you would like to stay in the L2, then you need consider using the Altivec transient cache instructions or the Altivec LRU load and store functions. These prevent data displaced from the L1 cache from being written to the L2, preserving data that is already there. Data access times to the L2 are still quite quick. They are 9 cycles on the 7450 and 10-15 cycles on the 7400 and 7410.

Set associative caches have a weakness. If you skip through memory with a stride of 4096 bytes you will end up only using a single cacheline set in the L1 cache. That is less than 1% of the cache. Linear memory access is best.

Another facet of memory management that can cause occasional problems is the TLB (translation lookaside buffer). Where memory is actually living in hardware is fairly complicated. Thus even though you have an address for it (e.g. 0xAC7E3500), actually locating it in hardware is a bit of work. Most memory is grouped together into pages, which are typically about 4 kB in size. These can be stored in any of a number of places (RAM, disk, device, etc.) The translation lookaside buffer caches 128 of these translated addresses (2 way set associative on 7400). If you need a piece of data from a page that is not in the TLB, then a rather expensive process of looking it up in a page table ensues. This can be as expensive or more expensive than a cache miss. Based on 4096 bytes per page and 128 entries in the TLB, only about 512 kB of data can be referenced by the TLB.

Thus to make a long story short, for a number of reasons it is a very good idea to place data that is used together near one another in memory. That way they chance that they will share the same page or even the same cacheline are very high, and your code will not encounter many long memory associated stalls.

Alignment and Data Layout

AltiVec does not include hardware support for loading and storing unaligned vectors. A review of what is required to align vectors in software should quickly convince you that dealing with misaligned vectors is very slow. (Code showing how to handle misaligned memory loads and stores appears in [Motorola99a].) Unaligned vector store operations (`vec_st` and `vec_stl`) are especially slow. If you must decide between unaligned loads and unaligned stores, pick unaligned loads. This is because unaligned stores may overwrite data adjacent to the target. Extra overhead is required to avoid this problem.

The best possible solution is to simply align your data properly. MacOS heap blocks returned to you using `NewPtr()`, `OTAllocMem()` or `malloc()` are already 16 byte aligned. Blocks allocated with `MPAllocateAligned()` can be aligned to suit your taste. Likewise, global and static storage starts 16 byte aligned at the start of every compilation block. Thus, all you have to do is make sure that your data and structs are properly arranged to preserve the 16 byte alignment that you are given. Vector types placed on the stack are automatically 16 byte aligned. A detailed description of alignment associated slowdowns can be found here: [Rentzsch01]

In the special case where you wish to align non-vector types to 16 bytes on the stack, you may do so by using a union:

```
//A union that allows memberwise access
to a vector float
typedef union
{
      float                f[4];
      vector float         v;
}Float4;
```

Individual stack frames may only be 8 byte aligned, so don't depend on the alignment of the start of the stack frame to correctly align non-vector types to 16 byte bounds. Stack frame conventions are detailed in [Motorla99b] Chapter 3.3.

Pipelining

The number of cycles each instruction takes is listed in [Motorola99c (Chapter 6, page 46), also Motorola01a–b]. For the most part, they take from 1-5 cycles each, depending on the vector subunit that they execute in. Most things take one cycle, except for operations in the Vector Complex Integer Unit (VCIU) and the Vector Floating Point Unit (VFPU). The vector permute unit (VPU) takes one cycle on PPC 7400 and 7410 but two cycles on the PPC 7450. The VCIU has three stages (four on PPC 7450) and the VFPU has four or five stages depending on whether or not Java mode is turned on. (Though the word Java is used here, this mode has almost nothing to do with the Java development platform. It is so named because it shares some numerical standards with Java.)

Both the VCIU and the VFPU are pipelined. This means that you can have multiple instructions executing at once in each. Each cycle, one new instruction can be added to the pipeline and another one can exit out the other end and be retired. This allows for a throughput of one instruction per cycle, even though it may take several cycles to process each instruction.

In order to make full use of the pipeline, you have to make sure that you have enough independent data available. Otherwise you will find that instructions will be prevented from starting down the pipeline in a timely fashion because they are waiting on the result of another calculation. As an example, suppose you are doing a vector dot product on a pair of very long vectors. A simple version might look like this:

```
//Simple dot product function for long
vectors
float SlowDotProduct(
        vector float *v1,
        vector float *v2,
        int length )
{
  vector float temp, temp2;
  float result;
  temp = (vector float) vec_splat_s8(0);

  //Loop over the length of the vectors
  multiplying like terms and summing
  for( int i = 0; i < length; i++)
    temp = vec_madd( v1[i], v2[i], temp);
  //Add across the vector
  temp2 = vec_sld( temp, temp, 4 );
  temp = vec_add( temp, temp2);
  temp2 = vec_sld( temp, temp, 8 );
  temp = vec_add( temp, temp2);

  //Store the result on the stack so it
  can be loaded into the FPU and then
  return it
  vec_ste( temp, 0, &result );
  return result;
}
```

The problem with this function is that each call to `vec_madd` depends on the result of the last one, so we don't actually get any pipelining here. We only do one `vec_madd` every four or five cycles. A faster method would be to load 64 bytes from each vector each loop iteration. This would allow you to stuff the pipeline:

```
//Do v1 dot v2 faster. In this one we
make sure the pipeline is full
float FasterDotProduct( vector float *v1,
                       vector float *v2,
                       int length )
{
  vector float t, t2, t3, t4;
  float result;

  t = (vector float) vec_splat_s8(0);
  t2 = t3 = t4 = t;

  //Loop over the length of the
  vectors, this time doing 4 vectors
  in parallel to stuff the pipeline.
  This is the only part that is
  substantially different. We have
  unrolled the loop to allow multiple
  vec_madds to pipeline with one
  another. This is now possible
  because the result from one
  vec_madd is not used in the next
  one.
  for( int i = 0; i < length; i += 4)
  {
    t  = vec_madd( v1[i],   v2[i],   t );
    t2 = vec_madd( v1[i+1], v2[i+1], t2);
    t3 = vec_madd( v1[i+2], y2[i+2], t3);
    t4 = vec_madd( v1[i+3], v2[i+3], t4);
  }

  //Sum our temp vectors
  t = vec_add( t, t2 );
  t3 = vec_add( t3, t4 );
  t = vec_add( t, t3 );

  //Add across the vector
  t2 = vec_sld( t, t, 4 );
  t = vec_add( t, t2);
  t2 = vec_sld( t, t, 8 );
  t = vec_add( t, t2);

  //Copy the result to the stack so we
  can return it via the FPU
  vec_ste( t, 0, &result );
  return result;
}
```

Clearly more can be done with this function, such as correctly handling the case when the vectors are not an even multiple of 64 bytes long. Also, some streaming cache instructions would really speed it up, since it is likely that a bigger bottleneck is memory overhead. However, it should benchmark a bit faster.

Processor Resource Scarcity

Rename Registers and the Completion Queue quite often surprise new AltiVec programmers trying their hand at aggressive scheduling of instructions. The problem is that the PPC 7400 and 7410 are starved for both vector rename registers and entries in the completion queue. Lack of available rename registers or slots in the completion queue can keep instructions that otherwise are ready to go from entering the execution stage. They will typically stall waiting in the instruction buffer for as long as is required until enough resources become available.

Rename Registers

Vector rename registers are temporary buffers used to store results from instructions that have finished execution but have not completed and been retired. There are six vector rename registers, six integer rename registers and six FPU rename registers. For an instruction to be successfully dispatched and to start executing, a rename register must be available for each destination operand specified by the instruction. Once the instruction is done executing, the result is written to the rename register. During the writeback stage of the instruction, the data is copied from the rename register to the destination register. If a subsequent instruction needs the result as a source operand, it is made available simultaneously to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the register file.

In some cases, it is possible that there are more than six instructions in a given unit scheduled to be in flight at a time. In such cases the seventh and later instuctions will stall and wait for one of the other six to complete before it will start. In principle, you can dispatch one complex vector operation and one permute operation to the vector unit per cycle. As VFPU operations can take five cycles to complete, they can consume most of the vector rename registers. If VPU operations dispatched simultaneously with them, you will run out of rename registers in three cycles. To understand why, one must consider the instruction completion queue.

Instruction Completion Queue

On the PowerPC 7400 and 7410 up to eight instructions can be "in flight" at any given time. The 7450 can have 16. When an instruction enters the execution phase it is placed on the completion queue. The instructions occupying the completion queue are not limited to just vector operations, they include all other types of PPC instructions, most significantly load/store operations, due to their long execution times. The completion queue is a queue in the true sense of the word. The completion unit only retires an instruction when all instructions ahead of it have been completed, the instruction has finished execution, and no exceptions are pending. This helps guarantee that instructions finish in the order that they were started. Only two instructions many be retired per cycle on PPC 7400 and 7410. The 7450 can issue and retire three instructions per cycle. For this reason, there is generally very little acceleration to be gained from simultaneously doing calculations in the VFPU and the FPU at the same time. Between load and store operations and VFPU ops and FPU ops, the completion queue can fill up rapidly because more instructions are dispatched than can be completed per cycle.

Even though an instruction may only take one cycle to execute, it can sit in the queue for tens or hundreds of cycles waiting to retire if the instructions ahead of it take a long time to finish executing. This will cause the queue to rapidly fill up, preventing new instructions from entering execution and preventing the release of rename registers for use by other instructions. This is in part why memory stalls are so bad. When a load stalls, only 7 (15 on 7450) more instructions can enter execution. They will not complete until the load completes and it will take up to four additional cycles to empty out the queue. Thus, even though the data to be loaded is not needed for many many cycles, stalled loads will cause other instructions to stall long before then. The remaining intervening instructions then add to the execution time of the function with unusual repercussions as I will show later in "Software Architecture Principles: Memory Speed!"

Vector Unit Overlap

One limitation of the PowerPC 7400, the first generation of G4 processors that entered the market, is that only one of the vector complex integer unit (VCIU), vector simple integer unit (VSIU) and vector floating point unit (VFPU) can accept a new instruction in any given processor cycle. The vector permute unit (VPU) is independent however. This means that if you mix instructions intended for these subunits in the same instruction stream, you won't get quite the amount of parallelism you were hoping for. Most functions do not contain both VFPU and VCIU/VSIU code in the same function. However, functions that do integer operations will frequently mix VCIU and VSIU operations. This limitation will prevent instructions from being issued to the VCIU and VSIU in the same cycle. This limitation was removed on the latest generation of G4, the PowerPC 7450.

Branching

Branching can be a bit of a problem in AltiVec and elsewhere. An example of a branch might be an if statement:

```
if( test )
        value++;
```

Often, when the processor encounters a branch, it may not have enough time to finish evaluating the test before it is time to decide whether to branch or not. As a result, all the processor can do is guess. If it guesses incorrectly, it has to dismantle all operations currently in progress, and restart in the correct place. This is costly.

There are predictable rules about which way the processor will guess. If the branch jumps forward (the else part of an if-else statement) then it is assumed not to be taken. If the branch jumps backward (a loop) then it is assumed to be taken — loops tend to loop more than once. So if you have to add an if…else statement to your code, it is best to place the rarely used case after the else, and the most common case after the if.

If … else ... statements usually concern only a single data stream. (Exception: the `vec_all_*` and `vec_any_*` instructions.) This makes it impossible to write code that can be pipelined or that operates in parallel. As a result, code with a lot of branching in it will operate many times more slowly in the vector unit than branchless code that does the same thing.

The best thing to do about this problem is to find a way to get rid of the branches and write algorithms that work for all possible inputs without special cases. Even if this means that the amount of code triples, it may still be faster.

This is the scalar version of some code that converts a audio mixing buffer (an array of 32 bit signed ints) into an array of 16 bit signed ints with clipping. The simple version of the function might look like this:

```
//Clip an array of 32 bit ints down to an
array of 16 bit ints.
void Convert( SInt32 *src, Sint16 *dest,
UInt32 sampleCount )
{
  SInt32 value;
  while( sampleCount-- )
  {
    value = src[0];

    if( value > SHRT_MAX )
      value = SHRT_MAX;
    else
      if( value < SHRT_MIN )
        value = SHRT_MIN;

    dest[0] = value;
    src++;
    dest++;
  }
}
```

There is a fine tradeoff between branching and branchless algorithms. The branchless variety are often longer, which can make for slower code. On the other hand, branches mispredict, causing the CPU to backtrack.

Real world solutions require testing. In this case, several ways of doing the clip were considered for the integer unit. The simple version looks like this:

```
#define Clip16( value )                 \
  if( value > SHRT_MAX )                \
    value = SHRT_MAX;                   \
  else                                  \
    if( value < SHRT_MIN )              \
      value = SHRT_MIN
```

A branchless version looks like this:

```
#define Clip16_2( value )               \
  sign = value >> 31;                   \
  value ^= sign;                        \
  value = (value | ((32767-value)       \
                >> 31)) & 32767;        \
  value ^= sign
```

A version with limited branching and a very short execution path looks like this:

```
#define Clip16_3( value )               \
  if( value != SInt16(value ) )         \
  {                                     \
    value >>= 31;                       \
    value ^= 0x7FFF;                    \
  }
```

(Some of these fail to give correct results in a very limited set of circumstances for values around MAX_LONG and MIN_LONG, but this is not a problem for a audio mixing buffer.)

In the end, the version without branching (Clip16_2) proved to be 4% faster in worst case scenarios in which most of the data needed to be clipped. However in best case scenarios in which less than half needed to be clipped, the short limited branching version (Clip16_3) was up to 50% faster. Which version to choose is a bit difficult to decide. While it is often said that it is best to optimize for the worst case, 4% is not a very large difference. In addition, Clip16_3 has many fewer instructions and so should execute much faster when the instructions themselves are not in the cache. Since this particular function is only called once every 11 milliseconds at the most, uncached performance must be considered. When the first pass through the benchmark loop was examined (when the instructions were not in the cache), the shorter version was found to be 20-30% faster.

For test code with source, please see [Ollmann01].

Software Architecture Principles

A number of general principles can be derived from these basic factors impacting code performance:

Economies of Scale

The net effect of Alignment and Pipelining is that AltiVec is rarely running at top efficiency unless it can work on 64 bytes or more of data at a time. The reason for this as follows:

With unaligned data, handling the edge cases is slow. You may have to load in some surrounding data from the destination buffer, merge it with your results and then save it back again. So these parts of your AltiVec code may be slower than the scalar version of the same thing. You make it back in the middle regions of your data set where alignment costs drop off to nearly zero. In order to be at least 50% not edge case, you need to have four vectors (64 bytes) worth of data. The exception to this rule is when you can pass data into your function by value, rather than memory addresses that have to be loaded.

However, passing vectors in via register or correctly aligning your data does not in itself keep the vector unit happy and well fed. In order to get proper pipelining, it is often necessary to have four independent data streams moving through your function simultaneously. The Vector Floating Point Unit (VFPU) has at least a four stage execution pipeline and the Vector Complex Integer Unit (VCIU) has a three stage execution pipeline (four on the 7450). Thus for optimal speed you need to be able to be ready to dispatch four VFPU instructions or three/four VCIU instructions at any given point that do not have any dependencies on one another. Better yet, for the 7450 use both the VFPU and VCIU at the same time. Having four vector instructions in flight at once means you need to process four (or more) independent vectors at a time through your function. Here again, we need 64 bytes worth of data to get good use of the vector unit. If you write functions that use less data at a time, you should consider declaring them inline in the hopes that they may be able to schedule themselves among the caller's other instructions.

High Throughput vs. Low–Latency

Most programmers think in terms of low latency algorithms — "How can I apply this function to that piece of data in the shortest period of time?" A somewhat different question you could be asking is "How can I apply this function to my entire data set in the shortest period of time?" The latter approach takes into account the effect of pipelining, parallelization, storage and other factors into the overall design process. This is often described as the difference between low latency and high throughput algorithms. Such algorithms may only be efficient with large amounts of data, but if you have lare amounts of data.

Because Altivec has comparatively long pipelines, operates on data in parallel, and is often limited by memory bandwidth, high throughput designs are usually much more successful in the vector unit than low latency designs. For example, using the vector unit to multiply a single floating point quantity by another takes four or five cycles. Using the vector unit to multiply four floating point quantities by four others also takes four or five cycles. Using the vector unit to multiply sixteen floating point quantities by sixteen others takes seven or eight cycles. Clearly there is a lot of advantage to handling a lot of data at once!

Suppose you design your functions so that this fact is obvious from the function interface. For example:

```
typedef vector float VF;
//rN = vNa * vNb
inline void Multiply(
  VF v1a, VF v2a, VF v3a, VF v4a,
  VF v1b, VF v2b, VF v3b, VF v4b,
  VF *r1, VF *r2, VF *r3, VF *r4 )
{
  vector float neg_zero;
  neg_zero = vec_neg_zero();
  *r1 = vec_madd( v1a, v1b, neg_zero );
  *r2 = vec_madd( v2a, v2b, neg_zero );
  *r3 = vec_madd( v3a, v3b, neg_zero );
  *r4 = vec_madd( v4a, v4b, neg_zero );
}

//Generate a vector full of -0.0.
inline vector float vec_neg_zero( void )
{
  vector unsigned  result;
  result = vec_splat_u32(-1);
  return (vector float )
          vec_sl( result, result );
}
```

Anyone calling this function would immediately realize that it is a waste of time to just multiply two floating point vectors together with this function, and that he would be much better off doing it four at a time. This sort of design motif helps reinforce high throughput programming practices. If this seems foreign or counterproductive to you, it may be because you are valuing too heavily the advantages of being able to work quickly on a single piece of data. Fundamentally, Altivec is all about working on lots of data. High throughput function designs such as these make the cost of low latency algorithms clear.

Note that it is difficult to return more than one vector from a function in C. Some care must be taken when crafting functions like this so that the return values are passed in register rather than by the stack, which would kill performance. Using pointers or references for return values in an inline function may allow the compiler to optimize away the load store overhead. However, you may need to disassemble the output to make sure the compiler does the right thing until you are satisfied with its behavior.

Another option is to use a C preprocessor macro. However the macros are notorious for causing obscure bugs. They also lack rigorous typechecking.

Memory Speed!

The biggest speed impediment to PowerPC performance today is memory overhead. This is increasingly true as processors move to higher and higher bus speed ratios between processor clock speed and motherboard clock speed. Whereas a 68k Macintosh might have had its processor running at the same clock speed as its motherboard, modern PowerPC machines might be running four, five, six or even seven times as fast as their memory subsystems. What this means is when there is a cache miss, you may have to wait for tens or hundreds of CPU cycles for the memory systems to catch up to you. With classical code writing styles, roughly every third instruction in a program is a memory operation. [Diefendorff01]

The thing to know about the G4 is that the L2 (and L3) caches serve as a victim caches on the G4 — data only comes to be in the L2 or L3 caches after being cast out of the L1 (or L2) cache. Data has to be moved to the L1 cache before it can be loaded into register. This means that every piece of data that you use has to be loaded in the slow way from RAM to the L1 cache at some point, and if you only touch a piece of data once or once in a while, it will almost always be loaded in the slow way. Furthermore, those data that are either too large to fit in the caches (e.g pixel buffers) or data not in the caches because it is accessed infrequently (e.g. sound data) tend to also be exactly the data for which Altivec has the most impact, so the problem of uncached data must be taken seriously.

Loading a cacheline from RAM to L1 takes about 35-40 cycles on my G4/400, provided that the page is in the TLB. Loading a 32 byte cacheline from L2 takes from 10-15 cycles. If all you do with the data that you paid so dearly to get is add two vectors together (as little as 1 cycle) and store the result, then during the other 39 cycles, your code will do nothing. If the prospect of having your code running at 1/40th of its expected speed bothers you, then this section is for you.

Because the speed bottlenecks in the processor have changed so much over the last few years, it stands to reason that some aspects of code optimization have to change too. I am trying to introduce this idea gradually because I know that a lot of very experienced programmers are going to be very resistant to the idea that some of their favorite code optimization techniques rely too heavily on memory, perhaps slowing down the code rather than speeding it up. If you are one of these programmers, I urge you to keep reading. Some optimization paradigms presented here may be new to you. I think you will find them useful:

Do More with the Data

If the speed of your function is limited by the speed of memory, you have approximately 40 cycles of time to work on each 32 byte chunk of uncached data. (This is the time it takes to load in each cacheline.) If you don't use all of that time, then the remaining amount is wasted. You will likely spend the rest of those 35-40 cycles stalled waiting for more data to appear. This is easily verified by profiling a memory intensive function using Sim_G4. Chances are you will see some very long stalls in `lvx`.

It is a common programmer instinct to break down complex problems into simple ones. Resist that urge in AltiVec code that touches lots of memory. 35-40 cycles is a very long time! You will have to work quite hard to use all of it processing only 32 bytes of data. Remember that these extra cycles have already been spent for you, so if you don't use them, they are gone. If you can replace any work anywhere else in your program with code here, you will gain that much more speed because additional work done here is "free". You save a load/store pair and perhaps an additional cache miss by doing the work now rather than later.

Surprisingly, doing more work on a set of data, <u>even if it is totally gratuitous work</u>, can speed up the function. *What?!* This is a rather peculiar side effect of the extra penalty you incur when you stall. It is common wisdom that the performance of a function looks something like shown in Figure 3a:

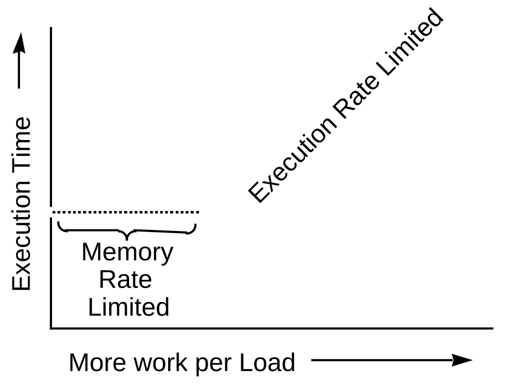


Figure 3a. The execution time of a function is dependent on how much work it has to do, unless some other factor (e.g. memory speed) becomes the bottleneck.

Actually, the line shape is more like Figure 3b. Being memory bound causes stalls, and stalls cause the eight or sixteen item instruction completion queue to quickly become full, stopping the whole instruction pipeline. As a result, loads and data processing start to happen serially relative to each other, rather than in parallel, causing a slowdown with more instructions in what should be a flat area.

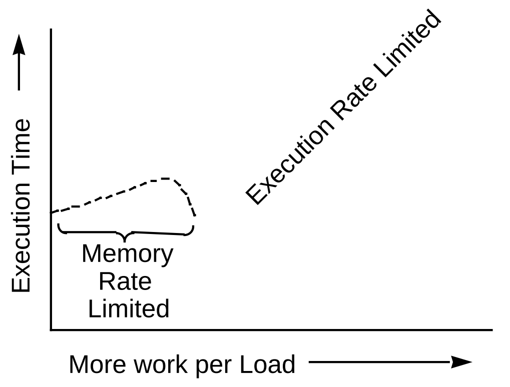


Figure 3b. Instruction completion queue stalls make the problem worse.

However, when you add more work per datum, such that the memory speed is no longer the dominant rate limiting factor, memory stalls disappear. Provided that cache hints are used, this means that the memory unit and the rest of the processor can operate in parallel rather than in series. The function runs faster in this case, because the time required to load the data is removed from the execution time of the function. This leads to the performance curve as shown in Figure 3b, which has maximum performance at the point where data load latencies and instruction completion latencies are roughly the same. *Notice the curious result that over some regions of the function you actually run faster with increasing amounts of work!*

Just in case there are a few doubters out there who refuse to believe that a function that does extra work can run faster than one that does not, I have written a program to prove it. This program loads data, does set amount of work on it, stores it and then loads the next data. Figure 4 shows the execution time of the function plotted against the amount of work done. Multiple data sets were acquired for various sizes of cache prefetch streams to make sure that data prefetching didn't bias the results.

The actual speed difference between the case at 35 add ops (1.5 MTicks) and the slower case at 25 add-ops (1.8 MTicks) that does less work represents about a 20% speed difference.

The red line represents what happens if you do not use cache instructions. Because there is no prefetch, loads and data processing have to happen serially. Because we can't do much in parallel, memory overhead and data processing times are additive and there is no flattening off effect.

The blue and black lines represent the best and worst case when using `vec_dst()` to prefetch data in blocks sized between 32 and 512 bytes. (I ran 16 such cases. They all fall between these two lines.) In all such cases, the function is moving at its fastest when 35 cycles of processing time is devoted to the data, roughly the same amount of time it takes to load in the data. It is particularly reassuring that the part of the curve that corresponds to CPU rate limited code can be extrapolated to run through nearly through zero. That shows that memory overhead for high workload functions using `vec_dst()` is near zero. We also note that the version that does not use `vec_dst()(red)` and the versions that do (black and blue) have very nearly the same slope. Thus, it appears, the complexity of the function is the prime rate determining factor with highly complex functions. This is exactly what you want!

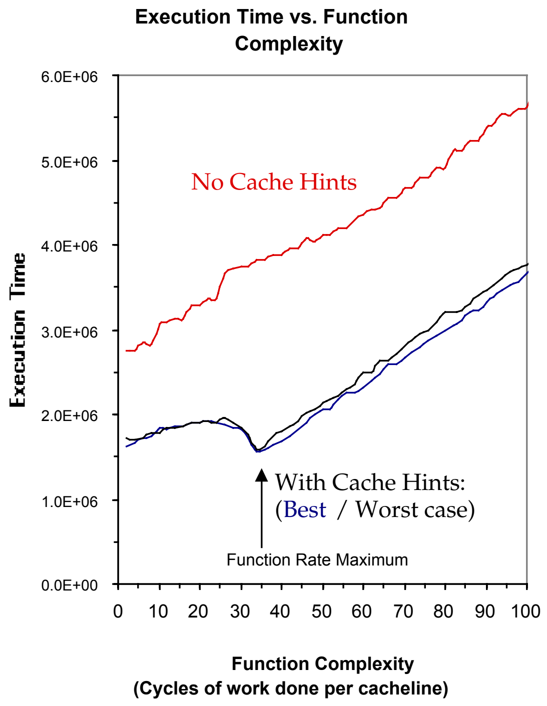


Figure 4. Execution time cycles for a function that does a set amount of work on each piece of data to process a large uncached buffer of data. With streaming cache instructions, optimal rates

are found when the time to complete the work matches the time to load the next data.

How does this finding shed light on your code? Unless they are very complicated functions, chances are most or all of your AltiVec functions do much less than 35 cycles worth of work per cacheline data. 35 cycles is a very long time. 35 cycles is enough time to calculate the dot product of two vectors with 112 elements in each! Since we can process nearly a kilobyte of memory in the time it takes to load 32 bytes, it should be apparent that it really requires a *very* complex function to be slower than RAM. Thus, the more real work you can do in your function per load, the better off you will be!

Unfortunately, the exact position of the "sweet spot" where the function has the greatest throughput likely varies from machine to machine. This is because memory load times are dependent on how the memory systems are set up, particularly the bus speed ratio between motherboard and CPU, the nature of the RAM used, and other factors. Therefore, it is probably not a good idea to just add a bunch of noops to your function to speed it up.

You can still take advantage of this effect however, provided you can find some real work to do to fill the extra time. As long as you are to the right of the sweet spot on the graph, your function is running at near 100% efficiency with near zero memory overhead. You really can't ask for anything more! It should also be noted that code written in this fashion will work equally well no matter where the data is (except for page misses), making it a great example of a place to use the transient cache instructions and LRU loads and stores. Even if you can't manage to do that much work on your data, you can still take advantage of the fact that any extra work that you can do in between one load and the next can be done for free.

I'm still skeptical. Why dont I just do what I always did and rely on the caches to speed memory access?

Sure the caches are fast, but they are only so big and the data will only be in the cache the second time you use it within a short period of time. You *will* be paying the price of a slow trip to RAM at some point to get that data no matter what you do. Why not do all the work on the data the first time you load it into register, and then flush it back out of the cache immediately making use of the transient/LRU instructions so that it doesn't displace frequently needed stuff. Save the caches for things that really need it. You dont need the L2/L3 caches if you do enough work on your data while you have it. You are actually better off if that data stays in RAM in such cases.

No really. I can fit all of my data in the L2 cache!

Great! However, loading data from the L2 cache into the L1 still takes 10-15 cycles — only 2.5 to 4 times faster than RAM. The same strategies still apply. The more work you do on your data at once, the more likely you are to be able to spend your time doing work instead of stalling waiting for memory. In addition, keep in mind that you have to share the L2 and L3 caches with cached instructions in addition to data. Also you always run the risk of an interrupt firing or a thread context switch occurring, tossing your data out of the cache. No data in the L2/L3 caches are safe.

The Fastest Algorithms Are Often the Ones That Use Less Data

If based on the above evidence you accept that memory overhead is the rate limiting factor for your AltiVec code most of the time, then it almost goes without saying that the fastest algorithms are the ones that use less data. However, did you consider the implications of this statement? What your mother told you about writing fast code is quite possibly no longer true! For example…

Lookup Tables Are Not Fast

Lookup tables, especially large ones, are not fast for a number of reasons. Most obviously, if you incur one cache miss accessing your lookup table, you can lose 40-250 cycles waiting for the data to load. That is a HUGE amount of time! Think of what you could have done with it.

Consider also that if you are using a lookup table, you are hopefully using it to do lookups on a lot of data. (A rarely used lookup table is almost guaranteed to not be in the cache, meaning abysmal performance because of lots of cache misses.) Functions that use a lot of data have high memory throughput needs. This means that you are probably already memory rate limited just loading in all the data that you want to use to index the looup table. In such cases, your lookup table only manages to further tax the memory systems. Recall that the execution time data shown above showed that memory bound code has about 35 cycles of dead time to fill with calculations. You could use that time to do the brute force calculation instead of the lookup and avoid further taxing the memory systems reading data from your table.

It is also hard to look up data in parallel in the vector unit. Often you have to do it one item at a time. Why not do a brute force calculation for 4, 8 or 16 items at a time?

Finally, every time you load part of the lookup table in, you displace something else from the cache. Whatever that is, chances are it will have to be loaded back in later. Doing a brute force calculation will preserve that data in place meaning that code elsewhere in your application will run more quickly. Brute force calculations can be fast, free, and more accurate. Your data caches will thank you.

The only lookup tables likely to do you any good relative to brute force methods are the ones that save a LOT of calculation (e.g. CRC-32), and those lookup tables that are so small you can preload them into register or a very small part of the L1 cache and then process a lot of data. (These approaches are not useful for low latency function designs because they add a significant degree of setup overhead.)

You can do a nice small fast register based lookup table with `vec_perm()`, but this approach seems to limit you to tables of perhaps 32-64 entries and extra work is required if the table cells are not 8 bits in size.

If you are still doubtful lookup tables are slow, I suggest you run some experiments. It would be helpful to do it on a bottleneck function in place in the app so that the full effect of displacing other needed data from the caches impacts the performance of your app and is measurable.

Larger Functions Are Slower

Data is not the only thing that needs to be loaded into the cache before it can be used. Instructions need to be loaded too. The instructions are loaded eight at a time, and the speed penalty for each such load is once again usually around 35-40 cycles. Thus, each uncached instruction that your function has to load has an average memory overhead of around four to five cycles. Since most instructions only take one cycle to execute and few take more than five, more often than not, the fastest uncached code is the shortest code. **Thus, for rarely used code paths, there is a very good reason NOT to attempt to optimize them, since optimized code is often longer**. When executing rarely used code, there is a lot of extra time spent standing around, which could be used for other things. If you habitually code funny math "shortcuts" using *many* short instructions to avoid *single* multi-cycle instructions like integer multiplication or division, you may be better off not doing so. Likewise, using large switch statements just to avoid a few cycles worth of work are likely to be counterproductive. Unoptimized, your code will be easier to read and shorter. In addition, keeping your code small means that it displaces less other code from the instruction caches.

One clear exception to the rule is any code in a loop. Loops get very good code reuse and have great temporal locality. The first time you read through a loop, it will execute as uncached code (if it is uncached) but after that it will be running at full speed. So if you are going to make gratuitous optimizations to rarely executed code, save it for loops.

Be Careful of Constants and Globals

Most programmers new to AltiVec make copious use of variables that have to be loaded in from memory each time the function is called. These may be globals or static constants defined like this:

```
vector float c = (vector float) (23.0);
```

These can be quite slow.

Perhaps you have a global used in a tight loop. Normally one might think that the compiler will do the smart thing and load the global into register and then use the copy in the loop, but it can't. The reason is that some other thread or interrupt level task might change the global, and so it has to be loaded in every time. **Always explicitly load in globals, constants and other items that have to be loaded from memory into a variable local to the function, and use the local variable in your function.** This will enable the compiler to avoid any excess memory overhead associated with redundantly loading in data over and over again.

Almost All AltiVec Code Is A Blitter

When it comes right down to it, most functions that can be accelerated for AltiVec move large quantities of data from A to B, possibly changing it along the way. Since the time it takes to load and store the data is usually the rate-limiting factor for these operations, such functions are to most standards simply blitters — functions for rapidly copying data from place to place. For that reason a lot of them look like blitters. They typically have this general high-throughput form:

```
/*src  and  dest  are  16  byte  aligned.
  Sizeinbytes is a multiple of 16.*/
void DoStuff(  vector float *src,
               vector float *dest,
               int sizeinbytes )
{
```

```
int count, loopcount;
vector float v1, v2, v3, v4;
/*Initiate a prefetch right away so
  that while we wait for the
  instructions to load on uncached
  code, the data can be loading too. */
vec_dst( src, 0x10010100, 0);

/*Now do any one time setup that is
  required before the loop */
count = sizeinbytes /  sizeof( v1 );
loopcount = count / 4;

/*Enter the loop */
while( loopcount-- )
{
  /*Prefetch the data in multiple
    overlapping segments – 256 bytes
    here. The optimal size will vary */
  vec_dst( src, 0x10010100, 0);

  /*The loop is unrolled to bite off 64
    bytes at a time for proper
    pipelining in our work segment*/
  v1 = src[0];
  v2 = src[1];
  v3 = src[2];
  v4 = src[3];
  src += 4;

  /* Insert work on the data here */
  ...

  /*write the result back out */
  dest[0] = v1;
  dest[1] = v2;
  dest[2] = v3;
  dest[3] = v4;
  dest += 4;
}

/*Deal with any stragglers */
if( count & 2 )
{
  v1 = src[0];
  v2 = src[1];
  src += 2;

  /* Insert work on v1 and v2 here */
  ...

  /*write the result back out */
  dest[0] = v1;
  dest[1] = v2;
  dest += 2;
}

/*Deal with any remaining stragglers */
if( count & 1 )
{
  v1 = src[0];

  /* Insert work on just v1 here */
  ...

  /*write the result back out */
  dest[0] = v1;
}

/*Dont forget to stop the prefetch! */
vec_dss( 0 );
}
```

Small Altivec functions may also appear in a low latency form that passes data in and out by register. If possible these should be declared inline, since ultimately it is likely they will be called from functions like the high-throughput example above. Inlining such Altivec functions saves a lot of stack overhead for setting up the VRSAVE special purpose register, and allows the compiler to pipeline your code against other tasks ongoing in the caller.

```
//Copy the alpha, red, green and blue
channels from four 32 bit pixels into
four vector floats.
inline void Pixel32ToFloat(
  vector unsigned char pixels,
  vector float &alpha,
  vector float &red,
  vector float &green
  vector float &blue )
{

}
```

Data Organization

It should be clear that in order to take advantage of this sort of high throughput code architecture, your data should be all in one place and accessible as an array. If you are jumping around in memory, especially if you can't even load data as whole vectors, performance will be poor. To make matters worse, the translation lookaside buffer (TLB, part of the unit that maps memory addresses to hardware locations) is up to 256 times less likely to cause a 150+ cycle stall with linear memory reads than for random memory reads.

It should be noted that the scalar unit can benefit from keeping data together as well, so even if you are not sure you will use AltiVec for a function, it doesn’t hurt to plan ahead. In some cases, large arrays present a problem for object oriented code. In these cases you have to evaluate your opportunities for parallelism within OO code. Often there is none so there is not much loss really. Large segments of OO code only need to operate on a single data thread, so wouldn't benefit from SIMD much anyway. Only when you can operate on multiple data in parallel is Altivec worth the effort.

Alignment

Because data is aligned in software not hardware, there is a pronounced disadvantage to using unaligned data. If possible, attempt to ensure that every vector is 16 byte aligned at least. Even if you are not sure you are going to use AltiVec, it is often a good idea to align your data anyway. It is harder to retrofit the changes into your application later, and good alignment almost never hurts scalar code.

Uniform vs. Non-uniform Vectors

Uniform vectors are those vectors whose elements all represent the same kind of quantity. An example would be a vector full of x coordinates for a data set on a graph. A non-uniform vector is one that holds different types of data in the same vector. An example might be a vector that holds {x, y, z, w} for a 3D graph. A vector full of 32 bit pixels is simultaneously uniform and non-uniform. If the pixels are treated as 32 bit pixels, you simply have four pixels there — a uniform vector. If you treat it as 4 sets of 4 different color channels, then you have 4 alpha channels, 4 red channels, 4 green channels and 4 blue channels all mixed up in the same vector — non-uniform.

Avoid non-uniform vectors!

If the elements in your vector represent different types of quantities, then typically you will find your function growing very complicated with a lot of permute operations, data shuffling on the stack, redundant calculations and lost opportunity for parallelism. Permute operations are inefficient because to a certain degree they can be said to do no real work. All they do is swap data around. You may only get a factor of two or three speed gain with your data organized this way, whereas a uniform vector based approach is likely to get the full factor of 4, 8 or 16.

Functions that use uniform vectors are typically easier to read and write because they look just like the scalar code. They usually take better advantage of pipelining within the processor. They rarely require the use of the permute unit at all. The constants that they use tend to be simpler and more easily generated without resorting to loading them from global storage. You almost never do redundant work.

Using uniform vectors in your code often means taking a hammer to your scalar code base. Chances are that for reasons unclear to man or compiler, you previously organized your data into nice neat structs with all sorts of different kinds of data interleaved with each other. C++ objects are almost always this way. Even some classical procedural constructs such as the lowly pixel have this architecture. It's a problem. Look for new ways to flatten out the data so that it is interleaved on the level of a whole vector or whole cacheline.

Unfortunately, it is quite common to have to significantly reorganize preexisting code and data to see large speed gains from Altivec. This is a fact that you will have to accept. The good news is that rarely are the changes so pervasive that they make the process impossible or impractical. Do not expect however that Altivec can always simply be clipped in in place of scalar code without advance planning.

Optimization - A practical Example

When adding AltiVec to a pre-existing program, you find that you need to vectorize preexisting functions. As an example, this is a function that calculates a third order polynomial of x:

```
// result = c_0 + c_1 * x + c_2 * x^2 + c_3 * x^3
float PolyNomial3( float c0, float c1,
       float c2, float c3, float x )
{
    return c0 + c1 * x + c2 * x * x
               + c3 * x * x * x;
}
```

Before we continue, I should mention that I have made no attempt to optimize the scalar version above beyond what the compiler already does. At the very least, this could be optimized using “Horner's scheme”:

```
// result = c_0 + c_1 * x + c_2 * x^2 + c_3 * x^3
  but faster
float PolyNomial3( float c0, float c1,
       float c2, float c3, float x )
{
   return c0 + x * (c1 + x *
               (c2 + c3 * x ));
}
```

…which is faster in principle because it compiles to three `fmadds` rather than three `fmuls` and three `fmadds`.

I have failed to make this optimization in both the scalar code and the vector code that you will see in a moment. It would benefit both roughly equally. One might guess based on instruction count alone that we would nearly double performance this way. (I received a letter claiming that it would.) I will return to the subject of the Horner scheme optimization at the end of the optimization process to see how much a simple code optimization like that really helps. In the mean time, take this as an early lesson to make sure that you are using the right algorithm before investing heavily into AltiVec. It's no fun to go back and rewrite everything.

The Simple Approach

Usually the first approach taken by most programmers when rewriting scalar code for AltiVec is to attempt to make the new AltiVec function fit into the mold of the old scalar version, using the same name and argument types and the same return value. This makes sense. The calling code won’t have to change. The data can stay organized the same way. So, let’s do that for this function to see how well that works out:

```
//We will use this union type to move
data from the FPU to vector unit
typedef union
{
        vector float    vec;
        float           elements[4];
}Float4;

//Our first attempt to vectorize
PolyNomial3.
float PolyNomial3( float c0, float c1,
float c2, float c3, float x )
{
  Float4  constants;
  Float4  the_Xs;
  float   returnVal;
  vector float result;

  //Load some values into the vectors
  constants.elements[0] = c0;
  constants.elements[1] = c1;
  constants.elements[2] = c2;
  constants.elements[3] = c3;
  the_Xs.elements[0] = 1.0;
  the_Xs.elements[1] = x;
  the_Xs.elements[2] = x * x;
  the_Xs.elements[3] = x * x * x;

  //Now do constants • the_Xs  (Dot
  product)
  result  =  vec_madd(  constants.vec,
             the_Xs.vec, ZERO );
  result  =  vec_add(  result,  vec_sld(
             result, result, 8 ) );
  result  =  vec_add(  result,  vec_sld(
             result, result, 4 ) );

/*All the elements of result now contain
  the same value, our result. Write it to
  returnVal so we can return it as a
  floating point quantity */
  vec_ste( &returnVal, 0, result );

  return returnVal;
}
```

Ok, lets benchmark this function and see how we did. Calling the floating point version 10,000 times takes 16733 time units. Calling our new vectorized version 10,000 times takes 46215 time units. Our AltiVec version is three times slower! Obviously we have done something wrong. But what could it be?

<u>The problem with our approach is that the interface of the function itself is inherently scalar.</u> This forces us to do so much data organization to set up the data for use by the vector unit that not only is the AltiVec speed advantage lost, we are actually three times slower than the simple FPU code.

With a quick inspection, it should be apparent that almost all of it is stack overhead — getting variables arranged where they need to be. Also, the `vec_add()` lines are doing a lot of redundant work, so even when we finally reach the stage that we are supposed to be operating efficiently in the vector unit, we aren't. There is no opportunity for pipelining here like there is in the scalar code. Notice the presence of non-uniform vectors.

The Vectorized Approach

The solution is usually to redesign the function interface to be a vector interface and go back and tweak the caller a little. It isn't too hard, but it makes a huge speed difference! Here is a fully vectorized polynomial function:

```
// constants = { c0, c1, c2, c3 };
//  x  =  four  different  x's  that  we
evaluate at the same time
vector float Polynomial3(
        vector float constants,
        vector float x )
{
  vector float c0, c1, c2, c3, x2, x3;
  vector float result;

  //Expand out our constants
  c0 = vec_splat( constants, 0 );
  c1 = vec_splat( constants, 1 );
  c2 = vec_splat( constants, 2 );
  c3 = vec_splat( constants, 3 );

  //calculate x^2 and x^3
  x2 = vec_madd( x, x, ZERO );
  x3 = vec_madd( x, x2, ZERO ); and

  //result = c0 + c1*x[4]
  result = vec_madd( c1, x, c0 );

  //result += c2 * x^2[4]
  result = vec_madd( c2, x2, result );

  //return result + c3 * x^3[4];
  return  vec_madd( c3, x3, result );
}
```

How did this new version do? On my machine, it evaluates 10,000 floats in 6410 time units. That is over twice as fast as the scalar code and nearly seven times faster than our first attempt at vectorizing this function!

So, what is the difference? First of all, we have completely gotten rid of all of the load/store instructions ...in this function, anyway. That was a huge overhead. Also, in roughly the same number of instructions or fewer as our previous example, we are evaluating the polynomial for four different X's at the same time! Finally, the code itself is straightforward, matching to a high degree the standard FPU code, making it *much* easier to read and debug.

Also, notice the difference between how we handled the data in this version compared to the last one. In the last version, each element in the X vector stood for something different: {1.0, x, x^2, x^3} — a non-uniform vector. In this version, each element in every vector stands for the same thing as the other elements in that vector — uniform vectors. (We ignore the non-uniform constants vector for the moment. Notice it is the only vector with lots of permutes associated with it.) Working with uniform vectors means that all the elements of the vector can be handled in the same way, which is exactly what we want for a SIMD architecture. In our earlier approach, because our vectors did not contain similar elements, we ended up spending a lot of time shifting elements around maneuver them into the right place. Calculating in parallel works fastest when the 4, 8 or 16 streams are always in the right place and independent of each other.

However, don't mistake these results to indicate there is a hard and fast rule about how to handle data. There are a number of times when you don't have to use uniform vectors. For some tasks (e.g. inverting a matrix), where there is quite a bit of symmetry built into the operation, you can get reasonably good performance without having to resort to something like inverting four matrices at a time in parallel.

Adding Pipelining

OK, how do we improve this further? Well, we still need to work on scheduling. The `vec_madd()` function takes either 4 or 5 cycles to execute. We have three of them in a row, each of which depends on the result of the last one. For this reason, the three take 12-15 cycles to finish, instead of 6-7. The pipeline is hardly full. Thus, we are only completing one `vec_madd` every 4 to 5 cycles, when we could be finishing one `vec_madd` per cycle.

We can fill the VFPU pipeline by evaluating four vectors in parallel. This is easily done by "unrolling the loop", a common trick for writing blitters, an example of which appears in the section above entitled "*Almost All AltiVec Code Is A Blitter*". Once again, we have had to go back and edit the caller, this time to make it pass us a pointer to all of the data, instead of small bits of it at a time.

Another approach I could have taken instead is to declare the function inline and hope that the compiler was able to schedule the instructions in with whatever else the caller is doing. This can be particularly beneficial because AltiVec stack overhead tends to be large. When an inline function is called within the confines of a loop, the compiler may automatically unroll the loop allowing the function to be interleaved with itself many times over, achieving the same effect that I worked hard to produce by hand.

Unfortunately, the compiler can be picky about what to inline, so it doesn't always work. The function generally must be small. In addition, for the purposes of a paper, I wanted to show the explicit unrolling of the loop so you get to see what it looks like. Loop unrolling makes for very large code, so please forgive this next code segment.

```
// A fully vectorized version optimized
   for better scheduling and cache usage

// constants = { c0, c1, c2, c3 };
// x = four different x's that we
   evaluate at the same time

void Polynomial3( vector float constants,
                vector float *input,
                vector float *output,
                UInt32 vectorCount )
{
  //Set up all the constants
  vector float k0, k1, k2, k3, zero;
  vector float a, b, c, d;
  vector float a2,b2, c2,d2;
  vector float a3,b3, c3,d3;
  vector float atemp,btemp,ctemp,dtemp;
  UInt32 loopCount;

  k0 = vec_splat( constants, 0 );
  k1 = vec_splat( constants, 1 );
  k2 = vec_splat( constants, 2 );
  k3 = vec_splat( constants, 3 );
  zero = (vector float) vec_splat_u8(0);

  //Manually unroll the loop four times.
  This allows for better scheduling.
  loopCount = vectorCount / 4;
  while( loopCount-- )
  {
    //Load 64 bytes of data
    a = vec_ld( 0, input );
    b = vec_ld( 1 * sizeof( b), input );
    c = vec_ld( 2 * sizeof( c), input );
    d = vec_ld( 3 * sizeof( d), input );
    input += 4;

    //Calculate x² for each data set
    a2 = vec_madd( a, a, zero );
    b2 = vec_madd( b, b, zero );
    c2 = vec_madd( c, c, zero );
    d2 = vec_madd( d, d, zero );
    //Calculate x^3 for each data set
    a3 = vec_madd( a, a2, zero );
    b3 = vec_madd( b, b2, zero );
    c3 = vec_madd( c, c2, zero );
    d3 = vec_madd( d, d2, zero );

    //Calculate c_0 + c_1 * x for each
    atemp = vec_madd( k1, a, k0 );
    btemp = vec_madd( k1, b, k0 );
    ctemp = vec_madd( k1, c, k0 );
    dtemp = vec_madd( k1, d, k0 );

    //add c_2 * x^2 for each
    atemp = vec_madd( k2, a2, atemp );
    btemp = vec_madd( k2, b2, btemp );
    ctemp = vec_madd( k2, c2, ctemp );
    dtemp = vec_madd( k2, d2, dtemp );

    //add c_3 * x^3 for each
    atemp = vec_madd( k3, a3, atemp );
    btemp = vec_madd( k3, b3, btemp );
    ctemp = vec_madd( k3, c3, ctemp );
    dtemp = vec_madd( k3, d3, dtemp );

    //store the result
    vec_st( atemp, 0, output );
    vec_st( btemp, 16, output );
    vec_st( ctemp, 32, output );
    vec_st( dtemp, 48, output );
    output += 4;
  }

  //At this point we may have 0, 1, 2 or
  3 vectors left to process

  //If 2 or 3, process two of them now
  if( vectorCount & 2 )
  {
    a = vec_ld( 0, input );
    b = vec_ld( 1 * sizeof( b), input );
    input += 2;

    a2 = vec_madd( a, a, zero );
    b2 = vec_madd( b, b, zero );
    a3 = vec_madd( a, a2, zero );
    b3 = vec_madd( b, b2, zero );

    atemp = vec_madd( k1, a, k0 );
    btemp = vec_madd( k1, b, k0 );
    atemp = vec_madd( k2, a2, atemp );
    btemp = vec_madd( k2, b2, btemp );
    atemp = vec_madd( k3, a3, atemp );
    btemp = vec_madd( k3, b3, btemp );

    vec_st( atemp, 0, output );
    vec_st( btemp, 16, output );
    output += 2;
  }

  //If we have one vector left, process
  it now
  if( vectorCount & 2 )
  {
    a = vec_ld( 0, input );
    a2 = vec_madd( a, a, zero );
    a3 = vec_madd( a, a2, zero );
    atemp = vec_madd( k1, a, k0 );
    atemp = vec_madd( k2, a2, atemp );
    atemp = vec_madd( k3, a3, atemp );
    vec_st( atemp, 0, output );
  }
}
```

How well does evaluating four vectors in parallel enhance performance? We can now evaluate 10,000 polynomials in 2400 time units! That is over seven times as fast as the scalar code.

```
2136:addi    | 7116 | .....IIIIDFFFFFFFR........................................................................ | 7128
2137:addi    | 7116 | .....IIIIIDFFFFFFFR....................................................................... | 7129
2138:lvx     | 7118 | .......IIIIDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEER................................. | 7167
2139:addi    | 7119 | ........IIIIDFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFR................................. | 7167
2140:lvx     | 7119 | ........IIIIIIIIDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEER................... | 7181
2141:addi    | 7120 | .........IIIIIIIIDFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFR................... | 7181
2142:vmaddfp | 7121 | ..........IIIIIIIDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEFFFFFFFFFFR.................. | 7182
2143:lvx     | 7122 | ...........IIIIIIDDDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEER......... | 7191
2144:vmaddfp | 7123 | ............IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIDEEEEFFFFFFFFFFFFFFFFFFFR......... | 7191
2145:lvx     | 7124 | EEEER........IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE | 7205
2146:vmaddfp | 7128 | FFFFR............IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIDDDDDDDDDDDDDEEEEFFFFFFFFFFFFFFFF | 7205
2147:addi    | 7129 | FFFFFR............IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIDFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF | 7206
2148:vmaddfp | 7129 | FFFFFR............IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIDEEEEFFFFFFFFFFFFFF | 7206
2149:vmaddfp | 7130 | FFFFFFR............IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIDEEEEFFFFFFFFFFFFF | 7207
2150:vmaddfp | 7169 | FFFFFFR...................................................IIIIIIIIIIIIIIIIIIIIIIIDEEEEFFFF | 7207
2151:vmaddfp | 7169 | FFFFFFFR..................................................IIIIIIIIIIIIIIIIIIIIIIIIDEEEEFFF | 7208
2152:vmaddfp | 7170 | IIIIIDEEEER................................................IIIIIIIIIIIIIIIIIIIIIIIIIIIIIII | 7211
2153:vmaddfp | 7170 | IIIIIIDEEEER...............................................IIIIIIIIIIIIIIIIIIIIIIIIIIIIIII | 7212
2154:vmaddfp | 7183 | IIIIIIIDEEEER...........................................................IIIIIIIIIIIIIIIIII | 7213
2155:vmaddfp | 7184 | IIIIIIIIDEEEER...........................................................IIIIIIIIIIIIIIIII | 7214
2156:vmaddfp | 7193 | IIIIIIIIIDDEEEER..................................................................IIIIIIII | 7216
2157:vmaddfp | 7194 | IIIIIIIIIIIDEEEER..................................................................IIIIIII | 7217
2158:vmaddfp | 7207 | ......IIIIIIDEEEER........................................................................ | 7218
2159:vmaddfp | 7208 | .......IIIIIIDEEEER....................................................................... | 7219
2160:stvx    | 7209 | ........IIIIIDEEEEER...................................................................... | 7220
2161:vmaddfp | 7210 | .........IIIIIDEEEER...................................................................... | 7220
2162:vmaddfp | 7211 | ..........IIIIIDDEEEER.................................................................... | 7222
2163:vmaddfp | 7213 | ............IIIIIDEEEER................................................................... | 7223
2164:stvx    | 7214 | .............IIIIDEEEEER.................................................................. | 7224
2165:vmaddfp | 7215 | ..............IIIIDDEEEER................................................................. | 7225
2166:stvx    | 7215 | ..............IIIIDEEEEEER................................................................ | 7226
2167:vmaddfp | 7216 | ...............IIIIIDDDDDEEEER............................................................ | 7230
2168:stvx    | 7217 | ................IIIIDDDEEEEEEER........................................................... | 7231
2169:addi    | 7219 | ..................IIIDFFFFFFFFR........................................................... | 7231
2170:bc+     | 7219 | ..................IIIIFFFFFFFFFR.......................................................... | 7232
```

Figure 5. A Sim_G4 trace of the execution of one particular loop iteration from our vectorized pipelined function. Large stalls are evident on lvx, which cause the rest of the code to stall.

Optimizing Cache Usage

Are we done yet? Well no. To see why, we will examime our routine with Sim_G4. Let's take a look at how well we are executing so far. The above trace (Figure 5) shows the actual amount of time each instruction takes inside our function's main loop. (This is just a snapshot of one particular pass through the loop.) Each instruction is listed on the left, then the clock at which the instruction started, a graphical display of what it was doing each tick, and finally a number showing during which clock the instruction finished:

Each instruction goes through four or five stages. It is fetched in the instruction buffer (**I**), Dispatched (**D**) to the appropriate execution unit, executes (**E**), and retires (**R**). If there is an instruction ahead of it in the completion buffer, then it will display (**F**) for a number of ticks until the item ahead of it in the completion buffer is retired. Two instructions can be retired per cycle on 7400/7410, three on 7450. The meaning of each of these stages is discussed in detail on Apple's site [Apple00].

The good news is that our pipelining attempts largely worked. The `vec_madd()` instructions are being executed with a throughput of about one per cycle. We seem to be able to dispatch, execute and complete one or two instructions per cycle, which is pretty good. Well, for the most part...

Unfortunately, there is a big stall that happens each time we call `lvx` (the asm translation of `vec_ld`) near the beginning of the loop. These seem to be taking 40–80 cycles to complete! The entire rest of our function only takes about 35 cycles to complete, so we are losing over half or two-thirds of our speed due to this one problem. Really big stalls on `lvx` usually happen as a result of a cache miss — the memory unit was asked to provide data and the data was neither in the L1 or L2 cache, so it had to take a long, slow trip to main RAM for it.

The solution is to add in cache instructions to help the CPU anticipate what data it is going to need. In the fourth example function, I've added a call to `vec_dstt()` to make sure our source buffer is loaded in time. This was done in the same manner as described above in the section entitled, "Almost All Altivec Code is a Blitter." (Source code: [Ollmann01])

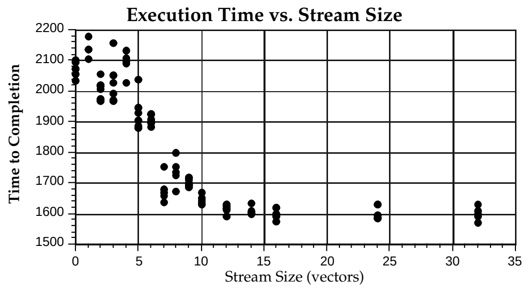
I've also called `dcbz` (data cache block zero) to zero the blocks that we are writing to before we write to them. Why do that? If you zero a block, the memory controller simply puts a bunch of zeros in the cache. No data is loaded from RAM. Since we are just going to overwrite this data anyway, this is a way of putting a block in the cache without having to load it. This can save us a lot of memory overhead.

It is pretty clear when to use `dcbz`. Zero any blocks just before you overwrite them. Just be careful not to zero memory in front of your target buffer. The `dcbz` instruction rounds all addresses down to 32 byte bounds, so if you aren't careful about alignment you can zero some data in advance of your block. If it is a heap block, the area just before your data is typically heap information. You will corrupt the heap when you free the block. Another thing to be cautious about with `dcbz` is that some day Motorola may decide to change the size of the cacheline. If that happens, you may end up zeroing too much data causing a bug. Apple provides `MPBlockZero()`, which may be used instead. If you prefer, `DriverServices.h` provides a `GetDataCacheLineSize()` function.

Using `vec_dst` or `vec_dstt` also requires some care. While you could attempt to stream in the entire input data set at once with a single call to `vec_dst`, in practice this generally doesn't work very well because interrupt level code or other preemptive threads may interrupt and call `vec_dst` on the same stream, halting your stream and replacing it with its own. Also the stream may outpace your code, displacing needed data with data we don't need yet.

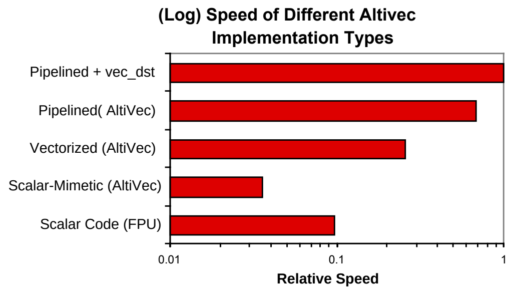
Typically what you want to do is set up many small overlapping streams. In each loop iteration, ask for a small stream that reads 64, 128 or 256 bytes forward from your current location in memory. It is ok to repeatedly use the same stream id. Try to stay away from id 3. `BlockMoveData()` uses it frequently at interrupt level.

How many bytes to read ahead usually must be determined experimentally. Generally there is a number beyond which no performance advantage is seen. If your data set sizes vary, you may also need to check different data sizes. In this particular case, the optimum stream size was in the 10–16 vector range (5-8 cache blocks). I found this out by doing a lot of testing:



Notice that I gathered a lot of data. There is some fluctuation in the numbers that you get, so usually you have to sample each data point a few times. I repeated each five times.

The combination of cache streaming and zeroing cache blocks improves the efficiency of our memory use a bit. We can now do our task in 1600 time units — ten times faster than the FPU and nearly thirty times faster than our first vector attempt! Here is a graphical representation of our different implementations. Longer bars are better. Values are given in Log format so you can see which steps gave the most improvements more easily:



Quick inspection of these results reveals that although we did see a 25% rate acceleration due to cache hints, we did not apparently get back all of those 50 cycles wasted per loop. If we had, the speed might have more than doubled. Unfortunately, at this point we have probably run into a fundamental weakness of the hardware. The memory subsystems are woefully inadequate to keeping the vector unit properly fed when running at a full gallop.

Sim_G4 reports that adding cache hints does drastically accelerate the function for about the first four or five loop iterations. However, after that point, we start to stall again in `lvx` (though in dispatch, rather than execution).

What appears to be going on is that the first time through the loop, the code is running very slowly because the instructions themselves are being loaded in from RAM. This gives the memory unit plenty of time to pre-fetch some data. However, in the second or later iterations through the loop, the instructions are already loaded, so we are able to proceed at maximum speed. We quickly catch up to the data stream and then start to stall again. Fortunately, because we are still prefetching data, the stalling isn't quite as bad as it could be, but it is quite significant. Often one stall will delay long enough that the next cacheline loaded from won't miss, so we only stall some of the time. This explains our overall speedup.

How do we get more speed?

The only thing that we can do now is to do more with each piece of data before we store it. It looks as though our calculation could easily be 3 times as complex and still run at memory fill rates. If we had something else we wanted to do with this polynomial, such as calculate where the points go when we plot it out on screen, or calculate a tenth order polynomial instead, we could probably do that now and get the extra math essentially for free. Sadly that is beyond the scope of this paper. It is something you will have to experiment with in your own program.

What about Horner's optimization discussed at the beginning of this chapter? Thanks for reminding me! It accelerated the scalar code by 8% and the AltiVec code by 3%. That is not quite the factor of two that was claimed based on just counting instructions! Actually, once you consider execution times and the fact that some of the `fmuls` can pipeline in the original version, one estimates 9 cycles for Horner and 11 for the original scalar version on 7400. Thus, we really should only predict a 22% acceleration based on the instructions themselves. We dont even see that much.

We don't see much speed improvement for the AltiVec code either, even though our work with pipelining means that that full factor of two is theoretically possible. We already know we are limited by the speed of memory, so this isn't too surprising. Any time saved is lost immediately to `lvx` stalls.

Clearly, it isn't just what you code, it is how you call the code, when you call it, and where the data is! In this case, Horner didn't improve any of those other things for us. Just improving the implementation of the function itself didn't do us much good.

Thus, microoptimization on code alone is only one part of the optimization process. Paying attention to all facets of how your program is constructed, including how data is passed into a function, how data is stored in memory, pipelining, temporal locality, your use of constants, etc. can yield far greater rewards.

The Optimization process

Hopefully by now, you have seen that the AltiVec optimization process is somewhat like for other code. However due to its speed, there are many more bottle necks from other parts of the system that must be taken into consideration. The overall optimization procedure can be summarized as follows:

(1) Only optimize those functions that are frequently called and are the performance bottleneck in your application. A good profiler is a must.

(2) Find the best algorithm. While AltiVec might buy you a factor of ten in performance, it surely isn't going to get you a factor of one hundred or one thousand. Often you can get that by doing something a different way. Picking the best algorithm also benefits your scalar version. You can still accelerate that with AltiVec.

(3) Once you have found the best method, arrange it for maximum parallelism. If you find you are doing a lot of permute operations to shift vector elements around relative to one another, it is a bad sign. The best implementations tend to use uniform vectors — vectors in which all elements in the vector stand for the same thing and can be processed in parallel. You may have to go back to rewrite the caller a little bit to make sure that the data is handed to you in a useful format. Likewise, you might have to change your data storage format to make the process of loading uniform vectors from memory a lot easier.

(4) Look to find ways to reduce memory overhead, either by passing constants and globals in as arguments or by generating them on the fly. Don't waste too much time creating constants. At worst you can load in a cacheline full of constants, and splat them out if you need to.

(5) Optimize your function for best instruction scheduling. If you use the VFPU or VCIU, typically this means that you will be processing data in a loop 64 bytes at a time so that you can have 4 independent vectors to stuff the pipelines with. If your function takes its data passed by value, either declare the function inline or take multiple vectors full of data at once. If your function reads data from memory, unroll your loop a little to read four or more vectors at a time. Do not unroll the loop completely because this will mean more instructions will have to be loaded into the cache, which may hurt performance.

(6) Only once you have done all other optimizations should you start looking at cache instructions. This way your memory access patterns are set in stone. If your function does any memory access, quite often it looks a bit like a blitter.

Calculate your prefetch constant and place a call to `vec_dst()` at the very beginning of your function. This ensures that while you are going through the relatively slow process of loading in the instructions for the function you can also be prefetching the data that you need. Also place a call to `vec_dst()` at the start of the loop and call `vec_dss()` for the stream at the end of the function.

There is no one correct stream block size that fits all functions. Typically, you have to test experimentally to find out what the best size is going to be. Typically block sizes in the range 64-256 bytes work best. This can be done in the context of a test app. Make sure that your data set resembles a real data set if it is likely to impact performance. Take multiple data points for each block size — the times can be somewhat variable. If a wide variety of sizes work, pick one that is not too close to the poor performance area. Hopefully this will mean that the function is more flexible with different bus ratios and RAM speed.

If your destination buffer does not overlap with your source buffer and you are just going to overwrite the destination buffer, call `dcbz` to zero the destination buffer before writing to it. This zeroes those blocks and places them in the cache without actually doing any loading of data. This can double the speed of your function if it is completely memory bound.

A good rule of thumb is that unless you are eating up at least 20 cycles of CPU time per vector load (after pipelining) and your data has to be loaded in, you are probably memory rate limited. This means that you will be stalling on loads and backing up the completion queue. If you can find more work to do per vector this can greatly accelerate your application. You will not only get more done per load/store pair, you will also be able to do memory access in parallel with data processing. Code running at this level of complexity will run at the speed of the CPU rather than the memory bus, a very desireable thing. If you can achieve this level of complexity in your function, it no longer matters whether your data starts in RAM or the caches. For this reason, this is a very good situation to investigate the transient cache instructions and LRU loads and stores with your primary data stream. This will help leave data that depend on the caches for fast processing in the caches.

(7) Move the function into your app and see if vec_ldl(), vec_stl() or vec_dstt() work better or worse in place of vec_ld(), vec_st() and vec_dst(). Since the transient / LRU versions tend to speed up code around your function at the expense of the function itself, its performance impact is difficult to measure correctly in a test app where there are no surrounding functions to benefit.

If you have lots of time to waste, go back and repeat steps 5 and 6 to see if a different block size works better with the new cache instruction variants you added in step 6. Also check performance on different machines.

Conclusion

Programming for Altivec is a mixture of old and new. Many old optimizations ideas still apply. However, Altivec is so fast that memory bandwidth rather than CPU speed is usually the performance bottleneck. In addition, the SIMD architecture requires that one design for parallelism, which impacts how data must be organized. For this reason many new software design principles must be employed for use with Altivec.

Data that is frequently used together should be stored together, preferably in large aligned arrays. Most Altivec functions that directly access data should be written in a high-throughput blitter format, designed to reduce memory overhead as much as possible and process a lot of data concurrently thereby maximizing pipelining opportunities.

Even with such functions, memory throughput will typically remain the performance bottleneck, not CPU cycles. For this reason it is best to process your data using a few large complicated functions that do a lot of work on each piece of data rather than a lot of little ones that repeatedly load and store data. Where streaming data prefetch instructions are used, additional expensive calculations may in many cases be added for free, because the additional cost is hidden by the larger cost of memory access. If memory stalls can be completely eliminated, functions may actually run faster as the result of adding more work. This may in some cases be used to enhance the quality of the calculation. In others, work may be combined from several functions into one.

Memory intensive programming techniques such as large lookup tables, and constants that must be read from memory are more likely to slow down Altivec code than speed it up. Except where it is very, very, VERY expensive, it us usually faster to generate constants on the fly and use brute force calculations instead.

The slow speed of the memory bus can reduce the speed of uncached code by 80%. The first iteration of a loop may run five times slower than successive loop iterations. For that reason, begin prefetching data as soon as possible in functions that handle a lot of data. This gives you more time for the data to appear, before the loop achieves full speed.

In addition, gratuitous optimization of rarely executed code can make your application slower, in cases where the optimization makes the function longer. Avoid large switch statements, aggressive loop unrolling, reduction in strength optimizations that replace single expensive instructions with numerous "cheap" ones, and other code bloating optimizations with rarely executed code. It is okay to optimize rarely executed loops.

Avoid branching in Altivec code. Use the Altivec comparator operations instead. Branching and lookups are single threaded by nature and do not work well with parallelized code.

Use uniform vectors. Reorganize data and algorithms so that individual vectors contain only one kind of data (e.g a vector full of x's instead of a vector full of x,y,z.) This usually enhances code readability, reduces reliance on permute instructions, eliminates redundant work, shortens code, increases execution speed, and enhances parallelism. Such modifications can also enhance the performance of optimized scalar code as well, by enhancing temporal locality of memory access and providing more opportunities for pipelining or superscalar execution.

Write code for maximum throughput in preference to low latency. Any small utility functions should be written with vector not scalar interfaces so data can be passed by register not on the stack. Where reasonable, construct these functions so that the the cost of using them in a low latency fashion is readily apparent from the interface.

Do not be afraid of undertaking substantial modifications to legacy scalar code. Most data layout or code architecture modifications that you make to benefit Altivec will also benefit scalar code, because they are generally designed to enhance memory throughput.

Acknowledgement

I would like to thank all of the members of the Altivec_forum@forum.altivec.org [Altivec00] mailing list for their collective wisdom, insight, patience and knowledge about all things Altivec. Special mention goes to Holger Bettag, Jean-Manuel Prudon, Sandra Nielsen, David Duncan, Craig Mattox, Carl Manaster, Chuck Fleming, Anton Rang, Steve Poole and Sam Vaughan, for their excellent suggestions for improving the parent document for this work. Finally, sincere thanks to Alex Rosenberg for his

contributions to the list. Alex has an uncanny ability to turn rampant speculation into rampant progress with just a sentence or two.

Bibliography

[Altivec00] The Altivec.org mailing list. Please direct your browser to www.altivec.org to subscribe to the list. A mirror may be found at altivec@groups.yahoo.com.

[Apple00] Apple Computer. Altivec Website. http://developer.apple.com/hardware/ve/performance.html

[Bettag98] Bettag, Holger. "Introduction to Altivec" Published on the web, 1998. http://www.informatik.uni-bremen.de/~hobold/AltiVec.html

[Clarke00] Clarke, Douglas. "Introduction To AltiVec" Paper. MacHack, 2000. http://home.san.rr.com/altivec/Pages/AltiVecEd.html

[Diefendorff01] Diefendorff, Keith. "PC Processor Microarchitecture" Article. Microprocessor Report, volume 13, number 9. http://www.chipanalyst.com/x86/microarchitecture/

[Motorola99a] Motorola, Inc. "Altivec Technology: Programming Environments Manual." http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/ALTIVEC/ALTIVECPEM.pdf

[Motorola99b] Motorola, Inc. "Altivec Technology: Programming Interface Manual." http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/ALTIVEC/ALTIVECPIM.pdf

[Motorla00] Motorola, Inc. " MPC7400 RISC Microprocessor User's Manual" http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/MPC7XX/MPC7400UM.pdf

[Motorla01a] Motorola, Inc. " MPC7410 RISC Microprocessor User's Manual" http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/MPC7XX/MPC7410UM.pdf

[Motorla01b] Motorola, Inc. " MPC7450 RISC Microprocessor User's Manual" http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/POWERPC/MPC7XX/MPC7450UM.pdf

[Ollmann01] Ollmann, Ian. "Altivec Tutorial" Published on the web, March 2001. http://www.alienorb.com/AltiVec

[Rentzsch01] Rentzsch, Jonathan Wolf "Straighten Up and Fly Right". Paper. Machack, 2001. http://redshed.net/macHack/2001/straightenUpAndFlyRight.html

[Rosenberg99] Rosenberg, Alex. Presentation on Altivec, MacHack, 1999.