Programming Applications for the AltiVec Architecture

by: Richard Jaenicke
Mercury Computer Systems

For years, DSPs held the signal-processing performance edge. Fast RISCs could not compete against the DSP's CISC architectures. Today, that's no longer true. Developers can get DSP performance with RISC programmability on a standard PowerPC architecture. Motorola's AltiVec technology has turned the PowerPC RISC into a signal-processing powerhouse that can take advantage of existing PowerPC software and tools.

This is not just a slam-dunk in the DSP vs. RISC wars. For while AltiVec tilts the scales decidedly in favor of RISC-based computing for embedded real-time signal and image processing, the traditional DSP community is not standing still. Their continuing adoption of architectural elements from general-purpose processors may ultimately lead them to products that are as easy to use as RISC processors such as the next-generation PowerPC with AltiVec. The open question is, within the same time frame, how much further will the PowerPC and other RISC architectures advance their own ease of use?

Today's RISC processors, most notably the PowerPC family, have achieved clock speeds that allow their raw performance to surpass DSP processors for typical vector operations. In the future, the fourth-generation (G4) PowerPC microprocessor may offer up to a fourfold increase over the current generation of PowerPC 750 microprocessors.

Much of this performance boost comes from Motorola's new AltiVec technology, which adds DSP-like capability to the PowerPC architecture. AltiVec technology expands the chip's processing capabilities through the addition of a 128-bit vector execution unit that operates concurrently with existing integer and floating-point units. This new engine permits highly parallel operations, allowing for the simultaneous execution of up to 16 operations in a single clock cycle.

AltiVec works with a set of SIMD instructions that dramatically increases the computational efficiency of a PowerPC processor. Programmers continue to enjoy all the software advantages of the friendly PowerPC core; extra transistors dedicated to AltiVec are available to handle DSP tasks, at the cost of a little extra programming effort.

```
void vadd (A, B, C, N)      /* A, B, C assumed aligned                    */
const float  *A;            /* input vector                       */
const float  *B;            /* input vector                       */
float      *C;              /* output vector C = A + B                    */
unsigned int  N;            /* number of elements                 */
{
    /* AltiVec in G4 consumes floats in blocks of four.                   */
    /* "extra" holds the number of valid floats in final block.           */
    const unsigned int extra = N % vec_step(vector float);

    /* The parameters (above) could have been vector floats.              */
    /* Because they were not, we need to tell the compiler that           */
    /* we want to use AltiVec (through typecasts).                        */
#define vA ((const vector float *)A)
#define vB ((const vector float *)B)
#define vC ((vector float *)C)

    /* This is the main loop        */
    for (N = N / vec_step(vector float); N > 0; N—) {
        *vC = vec_add (*vA, *vB);
        vA++;
        vB++;
        vC++;
    }

    /* Here we load beyond the end of the array.                          */
    /* There is no chance of access violation.          */
    /* Extra, random values harmless in !Java mode,                       */
    /* may cause extra cycles in Java mode.           */
    if (extra) {
        const vector float tempC=vec_add(*vA, *vB);
        unsigned int i;
        for (i=0; i<extra; i++)
          vec_ste(tempC, i*sizeof(float), C);
    }
}
```

Figure 1 AltiVec SIMD Programming.

As a RISC processor, a PowerPC processor with AltiVec technology also contains a PowerPC memory management unit (MMU). The MMU lets the processor work with virtual addresses, providing memory protection for embedded real-time applications. Without such protection, a simple memory access violation can cause a complex set of system-level symptoms that can be hard to diagnose.

AltiVec Programming

As with a DSP chip, programming for AltiVec involves C language extensions that take advantage of the chip's extended functionality. Any non-standard extension comes with an impact on productivity today and portability tomorrow. However, AltiVec compiler extensions are superior to those found in the DSP world because they represent additions to scalar compilers, which are more advanced and mature than those found in the DSP world. AltiVec programmers should never need to drop into assembly code to handle critical loops, thus avoiding a necessity that often bogs down DSPprogrammers.

Figure 1 shows a simple AltiVec program illustrating a few of AltiVec's SIMD instructions and the corresponding C

```
/* C = A + B + X for N elements using a vadd like Figure One's.                */
/* This code fragment uses L1 cache stripmining on buffers the                 */
/* size of a PowerPC MMU page (4096 bytes).                                    */
/* We assume A, B, C, X, and Temp are page aligned.                            */

/* This macro creates a command word for any one of the four                   */
/* data prefetch engines inside G4.
*/
/* Figure Three shows these command words in illustration format.              */
#define dstdata(size, count, stride) \
(((unsigned char)size & 0b00011111)<<24 \
      | (unsigned char)count<<16 \
      | (signed short)stride)

/* page contains the number of floats in an MMU page                           */
const unsigned int page = 4096/sizeof(float);

/* extra contains the number of floats overflowing the last page               */
const unsigned int extra = N%page;

/* distrB is the prefetch command word used in the main loop.                  */
/* It tells the prefetch unit to DMA one page worth of floats.                 */
const unsigned int dstrB = dstdata(1, page/vec_step(vector float),
sizeof(vector float));

/* We need pointers into arrays A, B, C, and X                                 */
const float *pA=A, *pB=B, *pX=X;
float *pC=C;

unsigned int strip;
for ( strip = N/page; strip>0; strip--) {
      vec_dstt(pX, dstrB, 0);           /* Prefetch X on DMA channel           0*/
      vadd(pA, pB, Temp, page, MMC);                  /* Temp = A + B          */
      vec_dstt(pA+=page, dstrB, 0);/* Prefetch A on DMA channel                0*/
      vec_dstt(pB+=page, dstrB, 1);/* Prefetch B on DMA channel                1*/
      vadd(Temp, pX, pC, page, CMM);                  /* C = Temp + X          */
      vec_dssall();                 /* Stop all prefetch              */
      pX+=page; pC+=page;           /* Next page of elements          */
}
if (extra) {
      /* Prefetch X on DMA channel 0                                           */
      vec_dstt(pX, dstdata(1,extra/vec_step(vector float),
sizeof(vector float)), 0);
      vadd(pA, pB, Temp, extra, MMC);
      vadd(Temp, pX, pC, extra, MMM);
      vec_dssall();
}
```

Figure 2 AltiVec Cache Control.

language extensions. This code implements a trivial vector addition operation using Motorola's AltiVec extensions to the C language. The most obvious enhancement is a new data type, "vector float." Specifying this type informs the compiler that you wish to use AltiVec technology to process arrays of floating-point numbers in SIMD blocks. The constant vec_step (vector float) contains the number of floating-point values in a single SIMD block. In this example, the number is four. It can go as high as 16 (if you're working with bytes instead of floats).

The compiler automatically generates AltiVec instructions required to load and store floating-point values in these SIMD blocks of four. However, the number of elements specified through parameter N may not always be a perfect multiple of four. Thus, only a subset of the final SIMD block of elements may be valid. In this final case we use an AltiVec "vector store element" (vec_ste) instruction to write only the valid elements of the last SIMD block.

One common question concerns why the addition operation is expressed as vec_add (*vA, *vB) instead of simply (*vA +*vB). The answer is that AltiVec, like a DSP chip, supports more than one type of integer arithmetic operation for each data type (AltiVec offers signed and unsigned with both modulo and saturating clamping modes). The C language contains no way to specify different kinds of addition. Thus, the AltiVec programming model, like DSP C language extensions that came before it, uses a subroutine-like convention for specifying operations.

A traditional version of this program, one that does not use AltiVec, requires four G4 processor cycles for each element (there are N elements). The program shown in Figure 1 is almost four times faster at 1.125 cycles per element. Unrolling the loop, then rearranging the C statements to hide load latency, reduces execution time even further, to 0.75 cycles per element (more than five times faster than the traditional program). AltiVec offers even higher acceleration factors when programs manipulate 16-bit or 8-bit quantities.

Leveraging AltiVec's Cache

Caching is another area of divergence between DSPs and RISCs. DSPchips avoid large cache; instead, they offer internal RAM loaded under program control using a DMA engine. This combination allows a DSP programmer to "double buffer" (use DMA to load future data while computing against previously loaded data). AltiVec's data stream touch (DST) instructions bring this DSP technique to the PowerPC. DST puts four independent prefetch engines under program control.

Experienced DSP programmers know that DMAinto a cache is not sufficient. Also required is a method to keep coefficients inside the chip (real-time data streams quickly displace coefficients from traditional caches). AltiVec addresses this requirement through new "LRU" and "transient" options on load and store instructions. These options allow a programmer to protect data already in the L1 and L2 caches, respectively, from replacement by newer data.

Figure 2 shows an example of AltiVec's cache control facilities. The code in Figure 2 uses a subroutine like vadd from Figure 1 to calculate C = A + B + X. Doing so requires two calls to vadd, and generates an intermediate result called Temp. Our goal here is to keep Temp in the L1 cache and not disturb the contents of the L2 cache.

The program implements double buffering, using DMAto load buffers it will soon need while operating out of buffers previously loaded. The vec_dstt instruction initiates DMAoperations. The final "t" in the name vec_dstt marks the resulting DMA data as transient (not to go into L2).

There is an extra parameter on the vadd used in this example that was not in the Figure 1 example. The new flag parameter specifies how vadd should load the data pointed to by parameters A, B, and C. If the flag is "M," vadd loads or stores its data using the LRU option. Data loaded or stored using this option will be the first thing knocked out of L1 (and never knocked into L2). The LRU option protects any coefficients that may be in L1 or L2 from replacement by real-time data streams. The "C" flag specifies normal load or store behavior. This flag parameter is not an extension to C, but instead represents something Mercury implemented in a longer version of vadd.