MOTOROLA POWERPC™ MICROPROCESSOR WITH ALTIVEC™ TECHNOLOGY – THE SYSTEM ENGINEER'S ANSWER TO REVOLUTIONIZING LIFE CYCLE COTS DESIGN PARADIGMS WITH SOFTWARE-BASED HIGH PERFORMANCE COMPUTING

Sam Fuller, Motorola SPS, Austin, TX

Introduction

System engineers are under increasing pressure to leverage standard components in their product designs. There are many reasons for this including time-to-market, complexity, supportability, and cost. However, there are many applications where commercial off the shelf technology (COTS), is just incapable of producing the solution that the system engineer has been asked to deliver. Traditionally image processing has been an area where off-the-shelf solutions have fallen woefully short in delivering acceptable system solutions, forcing the system engineer to drop back to using custom hardware, typically ASIC-based, to deliver the desired solution.

This paper will examine a new technology trend in general purpose microprocessors - the inclusion of dedicated single instruction, multiple data (SIMD) functional units specifically designed to accelerate tasks such as image processing. It will review the evolution of microprocessors, then examine the AltiVec SIMD architecture of Motorola's G4 generation of PowerPC microprocessors, the MPC7400. The paper will then present how a typical image processing task, the 2-D convolution, can be implemented using AltiVec and the performance and flexibility provided by this approach to image processing.

Microprocessor Evolution

Over the last 25 years, microprocessors have enjoyed a continuous increase in performance and attendant reduction in price/performance. Current best of breed microprocessors operate at frequencies in excess of 300 MHz and offer superscalar instruction dispatch, sophisticated branch prediction techniques, and support for high performance memory systems including external second level cache controllers.

As general-purpose microprocessors have become more powerful, new applications have become possible. In these new applications the microprocessor is able to shoulder a much larger portion of the system processing workload. In many cases the microprocessor is capable of providing all of the computational MIPS required by a system.

AltiVec technology represents the next natural step in the evolution of the microprocessor, where special circuitry is provided for the express purpose of accelerating the processing of natural data types. The term 'natural data types' is used to refer to digital data types designed to represent audio, video, image and speech. This is contrasted to the numerical and text data types that microprocessors have traditionally been designed to process.

Motorola microprocessors offering AltiVec technology represent a new class of product. In addition to providing 100% compatibility with the industry-standard PowerPC architecture, AltiVec technology will also provide system engineers with a new "one part / one code base" approach to product design. This approach will simplify design and support and revolutionize the life cycle COTS design paradigm by providing a simple software-based high-performance computing model to the system engineer. This model will scale in performance as the capabilities of the

0-7803-5749-3/99/$10.00 © 1999 IEEE

microprocessor increase with advancing process technology. Additionally, software-based multiprocessing can be used to further increase the capabilities of a system, not to mention providing the ability to revise and refine algorithms in software with no changes required to the underlying hardware platform.

AltiVec Architecture

Motorola's AltiVec technology expands the current PowerPC architecture through the addition of a 128-bit vector execution unit, which operates concurrently with the existing integer and floating point units. This new computational engine provides for highly parallel operations, allowing for the simultaneous execution of up to 16 arithmetic operations in a single clock cycle.

AltiVec technology employs a short vector parallel architecture. Depending on data size, vectors are 4, 8, or 16 elements long. In contrast, supercomputers, which were popular in the 1980's, utilized long vectors ranging up to hundreds of elements in size. This long vector approach, while useful for scientific calculations, is not optimal for the multimedia and other performance-driven applications targeted by Motorola with AltiVec technology.

AltiVec technology performs operations on multiple data elements using a single instruction. This is often referred to as SIMD (single instruction, multiple data) parallel processing. AltiVec technology offers support for:

- 16-way parallelism for 8-bit signed and unsigned integers and characters
- 8-way parallelism for 16-bit signed and unsigned integers
- 4-way parallelism for 32-bit signed and unsigned integers and IEEE floating-point numbers

AltiVec technology also includes a separate register file containing 32 entries, each 128 bits wide. These 128-bit wide registers hold the source data for the AltiVec execution units. The registers are loaded and unloaded through vector load and vector store instructions that transfer the contents of a single 128-bit register to and from memory. Support is also provided for bit and pixel data types.

AltiVec technology can be most accurately thought of as a set of registers and execution units added to the PowerPC architecture in an analogous manner to the addition of floating point units. Most mainstream microprocessors added floating-point units several years ago to provide better support for high-precision scientific calculations. Likewise, adding AltiVec technology to the PowerPC architecture will dramatically accelerate the next level of performance driven, high-bandwidth communications and computing applications.

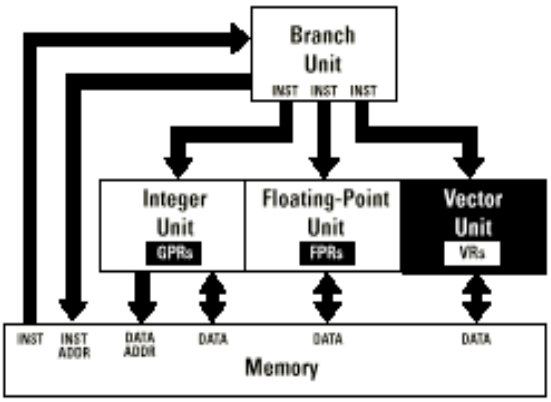

Figure 1. High-level structural overview for PowerPC microprocessor with AltiVec Technology

Each AltiVec instruction specifies up to three source operands and a single destination operand. All operands are vector registers, with the exception of the load and store instructions and a few instruction types that provide operands from immediate fields within the instruction. The AltiVec extension defines 162 new instructions. Although the AltiVec ISA

does not define these categories, the instructions can be grouped as follows:

- Vector Integer Arithmetic
- Vector Floating-point Arithmetic
- Vector Load and Store
- Vector Permute and Formatting
- Processor Control
- Memory Control

Integer and Floating-point Arithmetic Operations

Most arithmetic operations perform independent parallel computations on the elements contained in the source vector registers and place the results in the corresponding fields of the destination vector register. Both signed and unsigned integers and floating-point data types are supported. Both saturation and modulo arithmetic are supported. AltiVec provides a variety of powerful arithmetic operations: add, subtract, multiply, and multiply-add. Additional instructions perform min, max, and average, as well as conversion between floating-point and 32-bit integer numerical formats.

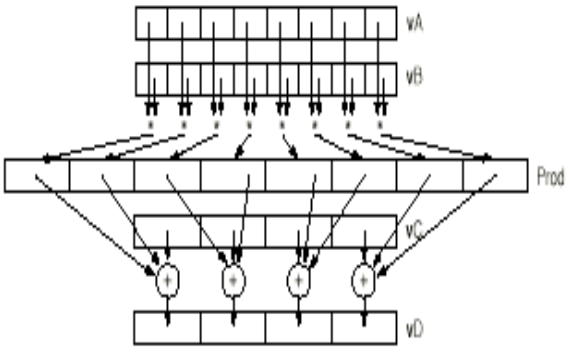

Figure 2 - A four-operand 16-element arithmetic operation

A few special arithmetic operations are provided which work across the elements in a vector register; these operations are *sum of products* and *sum across*. These operations allow for elements within a single vector register to be summed in combination with a separate accumulation register. These operations are valuable for generating vector dot products, which are the most common vector operation.

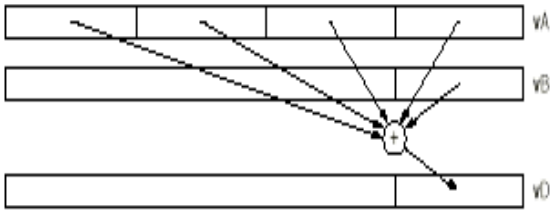

Figure 3 - Sum Across -- An Inter-element Arithmetic Operation.

Vector Logical, Formatting, and Permutation Operations

Non-arithmetic operations include various forms of compare, shift, and rotate. The following logical operations are also supported for bit and byte-wise data: AND, OR, NOT, XOR, AND-NOT. A select instruction is also provided. The select instruction is designed to select (or choose) source data from one of two source registers and transfer that data to the results register. The combination of compare and select provide a powerful way to mask and replace data elements across the entire 16-byte field of the vector registers with just a few instructions.

Other formatting instructions include wide field shift operations, as well as pack and unpack operations, including a special operation to handle the 1/5/5/5 pixel format common for 16-bit color pixels. Merge operations are also provided that can interleave data at the byte, half-word, and word levels.

Perhaps the most powerful operation offered by AltiVec is the permute operation. The permute operation is capable of arbitrarily moving data with the granularity of a byte from two 16-byte source registers into a single 16-byte destination register.

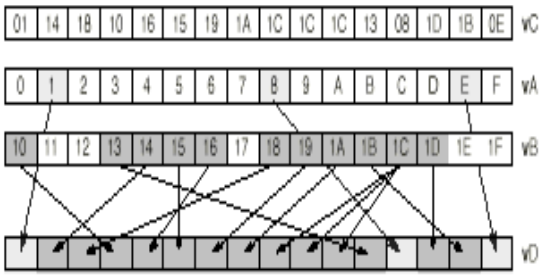

Figure 4 – AltiVec permute operation

For operations where 8 and 16-bit data items must be reorganized in memory before or after computations, permute can save significant time. In many instances a single permute operation can operate on 16 bytes of data and replace 4 or 5 operations per byte using a traditional RISC or DSP instruction set.

Applications of AltiVec Technology

Among the initial target applications for AltiVec technology are multimedia applications such as two-dimensional image and video processing, audio and speech coding, and 3D graphics geometry processing.

The combination of SIMD/Vector parallel execution units, high clock frequency and high performance cached memory system provide an opportunity for creating a high performance real-time software-based system design. This is in contrast to fixed function solutions that are most often implemented with a combination of general-purpose control processors coupled to one or more custom ASICs. While not replacing the need for ASIC circuitry in all applications, the performance offered by Motorola AltiVec processors will, in many cases, enable the system engineer to create a total software solution based on COTS technology. Most system engineers agree that a more standardized software based platform provides a much better migration path to higher performance and functionality.

Software Programming Model

The preferred programming languages for the PowerPC architecture with AltiVec technology are the C and C++ languages favored by embedded systems developers. To more easily express the parallelism presented by AltiVec technology, Motorola has developed a standardized set of C/C++ language intrinsics. These language extensions allow a software developer to use their preferred C/C++ development environment and language syntax while explicitly taking advantage of the parallel functional units and other facilities offered by the AltiVec technology.

The AltiVec C/C++ programming model provides a way for the programmer to match the parallelism in an algorithm to the functional units provided by AltiVec technology. Existing AltiVec-aware compilers typically match or exceed the capabilities of experienced assembly language programmers in terms of the performance of the generated program. Because of this, the programmer is able to work in a high-level language and avoiding the complexities inherent in assembly language programming, without sacrificing program performance. The compiler performs all of the register allocation, code scheduling, loop unrolling, code elimination and other optimization techniques to create a high performance program.

Two Dimensional Convolution

To demonstrate the capabilities of AltiVec for high performance COTS applications, we will examine an important image processing algorithm implemented on a Motorola AltiVec processor.

The convolution algorithm is widely used in image processing applications as the basis for a wide range of filtering operations including edge detection and enhancement, as well as blurring and sharpening. The same convolutional algorithm is employed with

different kernels, which are represented as small, usually 3x3 or 5x5, matrices.

Mathematically the discrete 2-D convolution is represented as:

$$Out(i,j) = \sum\sum In(i-m,j-n)Mask(m,n)$$

For the 3x3 Mask case we can write the summations explicitly. This helps us understand the calculations involved. This example shows the calculations required to generate the output pixel at location 12,10.

$$\begin{aligned} Out(12,10) = {} & In(12,10)*Mask(0,0) + \\ & In(12,9)*Mask(0,1) + \\ & In(12,8)*Mask(0,2) + \\ & In(11,10)*Mask(1,0) + \\ & In(11,9)*Mask(1,1) + \\ & In(11,8)*Mask(1,2) + \\ & In(10,10)*Mask(2,0) + \\ & In(10,9)*Mask(2,1) + \\ & In(10,8)*Mask(2,2) \end{aligned}$$

Based on the size of the convolution mask the two dimensional convolution of each output pixel requires MxN multiply sum operations. For a 3x3 mask, 9 multiply sum operations are required. For a 5x5 mask, 25 multiply sum operations would be required. Clearly the computational requirements for 2-D convolution will go up quickly as the size of the mask increases.

Scalar C Version of Convolution Algorithm

The following code is from the The Pocket Handbook of Image Processing Algorithms in C by Myler and Weeks.

```
/* 2-D Discrete Convolution */
void Convolve(struct Image *In,
struct Image *Mask,
struct Image *Out)
{
int i,j,m,n,idx,jdx;
int ms,im,val;
short *tmp;

/*the outer summation loop */
for(i=0;i<In->Rows;++i)
  for(j=0;j<In->Cols;++j)
    val = 0;
    for(m=0;m<Mask->Rows;++m)
      for(n=0;n<Mask->Cols;++n){
      ms = (Mask->Data + m*Mask->Rows + n);
     idx = i-m;
     jdx = j-n;
     if (idx >=0 && jdx>=0)
      im = (In->Data + idx*In->Rows + jdx);
     val += ms*im;
     }
   if (val > MaxValue) val = MaxValue;
   if (val < 0) val = 0;
  tmp = Out->Data+ i*Out->Rows + j;
 *tmp = val;
}
```

This code, when using the MetroWerks C Compiler with Level 4 optimization enabled, compiles down to 55 instructions in the inner mask loop across the mask rows and columns. There is also 10 instructions of loop overhead for moving across the input image. So, this scalar algorithm, compiled using an optimizing compiler, requires approximately 65 instructions per output pixel to calculate the 2-D convolution.

AltiVec Version of the Convolution Algorithm

2-D convolution provides ample opportunities for parallel execution and performance increase. Each of the multiplications between the input image pixels and the mask filter elements are independent. The final summation is however dependent on the multiplications. Each output pixel calculation is independent of every other output pixel calculation. The AltiVec algorithm for 2-D convolution takes advantage of this data parallelism.

The key to understanding this algorithm is to view the algorithm in terms of its output data. For 16-bit pixels an AltiVec functional unit is capable of computing eight multiply-sum results every cycle. Since each output pixel calculation requires nine multiply sum operations (for the 3x3 case), computationally AltiVec has the capacity to generate nearly one 2-D convolution result per cycle. However, this is just computational capability; we still

need to add instructions to load the input data and then to store the results. But with AltiVec we can load eight 16-bit pixels per vector load instruction and store eight 16-bit pixels per vector store instruction.

Our algorithm will create multiple copies of the input data in separate registers to facilitate the parallel computation of the results.

Because each output pixel is based on the input pixels around it, we will need to bring in the pixels that surround the output pixels that we are calculating. We do this by performing six load operations. We bring in 48 pixels of data initially in to six input registers. We then use the 'vector shift left double by octet immediate '(*vsldoi*) instruction to create shifted versions of the input data to align with the mask registers. Our approach is to prepare the input pixels and mask filter elements in the AltiVec registers to enable us to calculate eight output pixels in parallel with nine multiply-add operations.

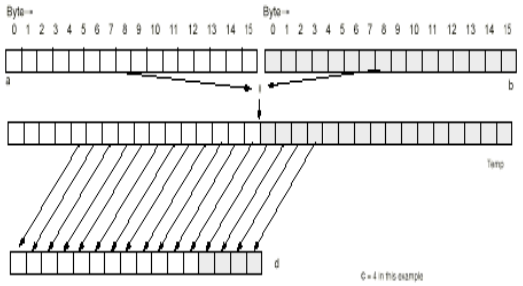

Figure 5. Vector Shift Left Double by Octet Immediate

The 'vector multiply high and add signed half word saturate' (*vmhaddshs*) instruction will be used to calculate the results. In this operation, each 16-bit input pixel in register A is multiplied by a 16-bit mask filter value in register B. The result is summed with the corresponding value in C, saturated, then stored in the target register D.

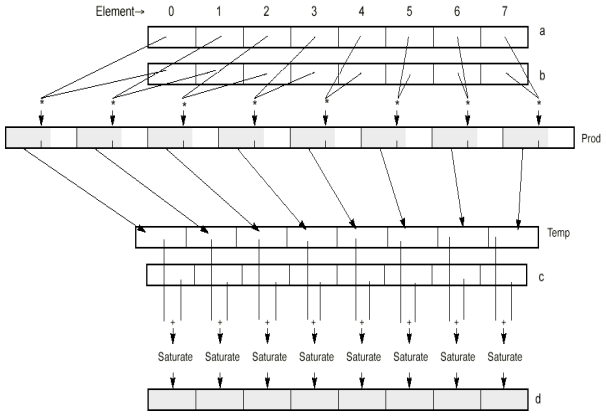

Figure 6. Vector Multiply Round and Add Saturated

For this example we will use a mask designed to crispen edges. This 3x3 mask is represented by the following matrix.

$$\begin{pmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

Because we are going to calculate eight output pixels in parallel, we will load the mask filter registers to facilitate this operation. This is done by replicating the mask filter values across the register and then creating shifted versions of the mask registers to match the shifted versions of the input pixel data. The shifting is required to provide the proper alignment for the *vmhaddshs* instruction to calculate the eight output pixel values.

When we are finished, we should have nine registers with input pixel data and nine registers with mask filter elements. The data in these eighteen input and mask registers are aligned so that *vmhaddshs*, when executed nine times while accumulating the results, will generate eight output pixel values.

The mask array because it is applied over the entire image must only be generated one time so its generation should not be part of

the main algorithm loop. Example code for generating the mask array is presented below.

```
vector signed short Mask[9] = {
/* First Row */
(vector signed short)( 1,-2, 1, 1,-2, 1, 1,-2),
/* shifted one element */
(vector signed short)(-2, 1, 1,-2, 1, 1,-2, 1),
/* Shifted two elements */
(vector signed short)( 1, 1,-2, 1, 1,-2, 1, 1),
/* second row */
(vector signed short)(-2, 5,-2,-2, 5,-2,-2, 5),
(vector signed short)( 5,-2,-2, 5,-2,-2, 5,-2),
(vector signed short)(-2,-2, 5,-2,-2, 5,-2,-2),
/*third row */
(vector signed short)( 1,-2, 1, 1,-2, 1, 1,-2),
(vector signed short)(-2, 1, 1,-2, 1, 1,-2, 1),
(vector signed short)( 1, 1,-2, 1, 1,-2, 1, 1),
};
```

The input pixels will be brought into six staging registers in the inner program loop. These registers will contain 16 pixels of data from each of three sequential rows in the image. For simplicity the algorithm will ignore edge conditions and focus on the work needed to process the first 8 pixels representing the inner row of the first 24 pixels brought in to the register file.

The inner loop of the proposed AltiVec version of the 2-D convolution is presented here.

```
/* Load the input data */
for(i = 0; i < RowCount - 2; i++) {

/* loop through each row */
/* load the pixels from the rows */
/* assumes image is 128b aligned */

// First generate the row pointers
Row1Ptr = Input + i*RowCount*RowLength;
Row2Ptr = Input + 2*i*RowCount*RowLength;
Row3Ptr = Input + 3*i*RowCount*RowLength;

// Now work across the row (inner loop)
for(j = 0; j < RowLength/VectorSize; j++){

Row1Ptr = Row1Ptr + j*Vectorsize;
Row2Ptr = Row2Ptr + j*Vectorsize;
Row3Ptr = Row3Ptr + j*Vectorsize;
Results = zero;

//  Main calculation
// Loads, shifts, mul-adds are interleaved
Row11s0 = vec_ld(0,Row1Ptr);
Row12s0 = vec_ld(16,Row1Ptr);
// Shift data
Row11s2 = vec_sld(Row11s0,Row12s0,2);
Row11s4 = vec_sld(Row11s0,Row12s0,4);
// Calculate Result
Results = vec_mradds(Row11s0,Mask[0],Results);
Results = vec_mradds(Row11s2,Mask[1],Results);
Results = vec_mradds(Row11s4,Mask[2],Results);
// Process next row
Row21s0 = vec_ld(0,Row2Ptr);
Row22s0 = vec_ld(16,Row2Ptr);
Row21s2 = vec_sld(Row21s0,Row22s0,2);
Row21s4 = vec_sld(Row21s0,Row22s0,4);
Results = vec_mradds(Row21s0,Mask[3],Results);
Results = vec_mradds(Row21s2,Mask[4],Results);
Results = vec_mradds(Row21s4,Mask[5],Results);
//Process last row
Row31s0 = vec_ld(0,Row3Ptr);
Row32s0 = vec_ld(16,Row3Ptr);
Row31s2 = vec_sld(Row31s0,Row32s0,2);
Row31s4 = vec_sld(Row31s0,Row32s0,4);
Results = vec_mradds(Row31s0,Mask[6],Results);
Results = vec_mradds(Row31s2,Mask[7],Results);
Results = vec_mradds(Row31s4,Mask[8],Results);
/* store this to the output */
/* the end result contains eight shorts which
are the convolution of the input image with the
mask */
vec_st(Results,0,(vector signed short *)
OutputMatrix[i*RowLength + j*VectorSize]);
}
}
```

Results

This AltiVec version of the code provides for the parallel computation of eight pixels for the 3x3 2-D convolution algorithm. The Mask filter values should be prepared one time outside of the main loop.

This code was compiled with the MetroWerks C compiler with AltiVec support with Level 4 optimizing enabled (the same as for the scalar algorithm.) The compiler generated 32 total instructions in the inner loop. Nine instructions were required for the pointer calculations and for the zeroing of the result register. Twenty-three instructions supported the loads, shifts and ‘multiply round and adds’ (mradds) and storing of the result. This inner loop works across and entire input image row.

Additional instructions are required to support the movement down the rows of the input image. The compiler generated 26 instructions to do this work. If we assume that a row has 512 pixels then the cost of these instructions is ~.05 instructions per pixel, the overhead of these instructions is negligible. The 32 inner loop instructions will produce 8 result pixels. In other words, this algorithm requires 4 instructions per output pixel. This is in comparison to the scalar version of the

algorithm presented earlier where the compiler generated 65 instructions per output pixel. In this example the AltiVec version should be approximately 16 times faster than the scalar version.

This simple analysis does not examine additional speed improvements that may be available due to parallel execution of the instructions. For example, the shift instructions for subsequent rows can be scheduled to occur in parallel with the 'mradds' instructions for previous rows. The Motorola G4 processor, the first production microprocessor to incorporate AltiVec technology, allows for parallel execution of load, permute or shift and arithmetic instructions with up to two new instructions started each clock cycle in conjunction with parallel execution of traditional scalar integer and floating-point operations.

Summary

With the introduction of AltiVec technology, Motorola is again demonstrating its commitment to the PowerPC architecture and to meeting the requirements of next generation high performance computing applications. AltiVec technology expands the PowerPC's capabilities by adding high-bandwidth SIMD functional units to attack complex signal processing challenges. An example 2-D convolution was examined on both Scalar and AltiVec processors. An AltiVec enabled processor provides a significant performance improvement over a scalar processor. General purpose processors with SIMD functional units will provide an aggressive performance growth path for embedded and computing systems designers, while lowering development barriers inherent in multiple architecture designs, thereby reducing the time to market and total system development expense.

Acknowledgements

I would like to thank Roger Smith, Kate Stewart, Becky Gill and Kalpesh Gala for their help in preparing this paper.

References

AltiVec Technology Programming Environments Manual, Motorola, 1998

AltiVec Technology Programming Interface Manual, Motorola, 1999

Sam Fuller, "Motorola's AltiVec Technology", www.mot.com/SPS/PowerPC/teksupport/teklibrary/papers/altivec_wp.pdf

Harley R. Myler, Arthur R. Weeks ,*The Pocket Handbook of Image Processing Algorithms in C* . Prentice-Hall, Inc. 1993

Gilbert Strang, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, 1986

William Pratt, *Digital Image Processing*, Wiley-Interscience, 1978