Proc. 2002 Australasian Conference on Robotics and Automation
Auckland, 27-29 November 2002

Vectorized Machine Vision Algorithms using AltiVec - Part 1

Bing-Chang Lai, Phillip John McKerrow and Jo Abrantes
School of Information Technology and Computer Science
University of Wollongong
Wollongong NSW 2522
Email: bl12@uow.edu.au, phillip@uow.edu.au, jo_abrantes@uow.edu.au

Abstract

The beginnings of a generic vectorized machine vision library implemented in AltiVec is discussed. The general vector loop for machine vision algorithms requiring only a single input pixel per input image to produce a single output pixel is discussed and analysed. Factors discussed that affect the speedup of the vector program are data alignment (when loading and storing), data streaming instructions (used to prefetch data), and function complexity.

In addition, two simple operations are presented — an image doubling function and a threshold functor.

1 Introduction

Machine vision processes typically involve image acquisition, image processing, image analysis and output display. Machine vision algorithms can thus be categorised by their position in this process. Alternatively they can be divided by their operation producing categories such as arithmetic operations, logical operations and lookup operations. The authors instead proposes a division based instead on the input to output correlation of machine vision algorithms. This method of division should enable generic programming techniques to be readily applied to machine vision algorithms.

The vector processor refers to a computational unit that is capable of computing a single operation for more than one element set as a single instruction. Vector processors are widespread, with many processors having some sort of vector capabilities. In desktop computers, many microprocessors have a fake vector unit which is simply the floating point unit in disguise. Examples of such processors include offerings from Intel and AMD. AltiVec technology, which found in Motorola's PowerPC architectures, however has its own specialised vector unit. IBM's Power4 CPUs also support AltiVec. However the Power4 CPUs are available only in servers and are not built for the desktop.

This paper describes part of an effort to merge the division based on input to output correlation with the vector processor[1] to produce a generic vectorized machine vision library. Only machine vision algorithms requiring only a single input pixel per input image to produce a single output pixel in a single output image are discussed.

2 Writing High Performance Vector Programs

High performance programs are required for machine vision applications. Faster programs means more analysis and/or decision making can be done. While the desktop computer's processing power increases by leaps and bounds every year, programs written incorrectly would not experience the same speedup. This is because today's processors use many different and complex techniques to achieve such high speeds. Programs written with little regard to these techniques typically waste much of their processing time. These inefficiencies affect both scalar and vector programs. However, vector programs are more susceptible due of their higher memory bandwidth requirements. This section discusses how to write high performance vector processor programs with AltiVec.

2.1 The Vector Processor

Normal sequential processors compute values one at a time. A vector processor is simply a processor that computes a number of values at a time (see Figure 1). This is different from multi-threading where the processor still does computation on one value at a time, but runs two or more separate instruction streams, switching between them at will.

Vector processors normally augment sequential processors, because not all algorithms are suitable for vectorisation. Vector processors are suitable for applications where there is a huge amount of data that the same series of instructions are to be applied to — Single Instruction Multiple Data (SIMD) problems. Examples of applications which should make good use of the vector processor include matrix multiplication, video, image and sound processing. In

[1] Currently, only AltiVec is being considered.

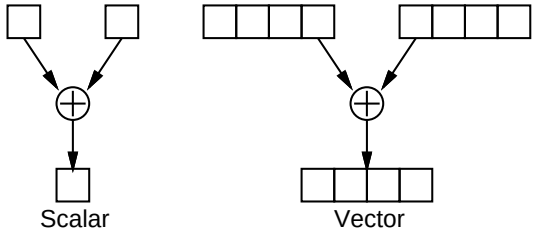

Figure 1: A sequential processor adds a pair of numbers together in the same time a vector processor adds *n* pairs of numbers. In the above picture, the vector processor adds four pairs.

fact, on the desktop, vector processors were built primarily for multimedia applications. One of the main reasons why consoles such as the PlayStation 2 can render millions upon millions of polygons per second is because of vector processor units (Ref [Stokes, 2000]). Examples of vector processor technologies include MMX, SSE, and AltiVec.

Vector processors have a number of common restrictions that affect how vectorized programs function. In general, all vector processors are able to process *n* elements in a vector exactly — no more, no less. This "magic" number of elements together form a vector. Usually, the number of elements in a vector differs across types while the overall size of the vector remains constant. In addition, they usually have restrictions on where they can fetch and store data quickly. Such locations are referred to as being aligned; any other locations are unaligned. In fact, one of the main bottlenecks with vector processors is memory. A vector processor requires a much larger memory bandwidth than an equivalent scalar processor due to its larger input sizes. The final characteristic that could impact programs is that vectors of different types are all the same size differing in the different numbers of elements.

This discussion uses AltiVec because it has a separate vector processing unit. Having a distinct vector processing unit gives AltiVec better vector processing performance, and more vector operations. For more information about AltiVec programming, please consult [Mot, 1999; AltiVec.org, 2002; App, 2002a; Ollmann, 2001; Lai and McKerrow, 2001b].

2.2 High Throughput Computing

A common method of optimisation is low-latency optimisation. Low-latency optimisation seeks to reduce the running times of individual functions. Basically, the slowest function is identified and given a speed boost. High-throughput optimisation however emphasises crunching as much data as possible in the shortest feasible time. This is usually done by unrolling loops, and reducing data dependencies. Such activities usually results in a fuller pipeline which leads to faster speeds for larger amounts of data. According to [App, 2002e], high-throughput optimisation usually produces faster programs than low-latency optimisation for larger data sets. High-throughput optimisation has a tendency of producing larger functions that are more difficult to read. For more information on high-throughput computing, please consult [App, 2002e].

2.3 Memory and Caches

Today's processors are much faster than main memory. This has lead to the creation of L1, L2 and even L3 caches. Since memory is so much slower than the processor, it is a good idea to simply do more with the data once it has been loaded into the processor. In addition, it is good to reduce the dependency on memory. This speed difference is felt more acutely for vector programs because they require higher memory bandwidth, since they crunch through more data than equivalent scalar processors in the same time frame. [App, 2002d] discusses such issues and provides some solutions to reduce the dependency of algorithms on memory. In fact, [Ollmann, 2001; App, 2002d] recommends against using lookup tables in vector programs unless the operation is really complex, or the lookup table is really small. Avoiding large lookup tables can reduce the dependency of algorithms on main memory.

2.4 Data Streams

In order to reduce the impact of slower main memory on the operation of the processor, AltiVec provides a set of instructions to prefetch data from memory to the caches, known as data streaming instructions. Such instructions schedule loading of data from main memory to the cache independently of the processor. The idea is that if the data is prefetched early enough, by the time the processor requires it, it will be in the cache. Since the processor only fetches data from the cache, the processor does not need to wait for the data to be transfered into the cache from main memory. Data streams are discussed in more detail in [App, 2002b].

3 The Scalar Loop

A scalar implementation of machine vision algorithms that requires only a single pixel from each input image to produce a single output pixel is discussed in this section. Such a program is easy to write, with suitable generic versions already existing in VIGRA (Ref [Köethe, 2001; Lai and McKerrow, 2001a]) and the Standard Template Library (STL). VIGRA provides a `vigra::transformImage` function which uses a 2-D iterator to walk through the image. The `std::transform` function provided by STL can also be used if the image is presented as a 1-D iterator. Since 2-D images are typically stored as 1-D arrays, this should not be a problem. Algorithm 1 is an example of such a loop. Like `std::transform`, it is also a 1-D loop. Because the vector loop discussed in the next section uses only a 1-D loop (for simplicity), Algorithm 1 is also a 1-D loop. This is the loop that the vector version will be profiled against.

It should be noted that `std::transform`, `vigra::transformImage`, and Algorithm 1 are all capable of using any memory location for read and write; they are capable of handling unaligned loads and stores. In

addition, all three algorithms will fail when the input region overlaps with the output region in certain configurations. This is because the algorithms writes the output to the correct position immediately. Therefore, if this position is also in the input area and has not yet been processed, the output will be erroneous since the input would have changed.

Algorithm 1 Scalar loop

```
 1 /** A 1-D scalar transform loop.
 2  *  T is the input and output image type
 3  *  F is the functor type
 4  */
 5 template<class T, class F>
 6 void transform(T* start,T* end,T* out,F f)
 7 {
 8     T *pi, *po;
 9     for(pi=start,po=out; pi!=end; ++pi,++po)
10     {
11         *po = f(*pi);
12     }
13 }
```

4 The Vector Loop

While invoking Algorithm 1 using vector types would compile and run, the resultant program would behave in a slightly different fashion. The program would only work correctly for input and output images that start on an aligned memory location and whose sizes are a multiple of the vector size. While such constraints might be perfectly acceptable in some situations, for a machine vision library, being so draconian about input and output parameters seems less than ideal. Ideally, the vector version should be able to run at full speed, and be able to work with images having any number of pixels and starting at any address.

A vector algorithm that performs in the same manner as Algorithm 1 is presented in Algorithm 2. Several `#define` macros control its capabilities — `DST` turns on data streaming instructions, `UNALIGNED_LOAD` allows it to load from unaligned location and `UNALIGNED_STORE` allows it to write to unaligned locations. If `UNALIGNED_LOAD` or `UNALIGNED_STORE` is not specified, then the aligned version is used.

Unaligned loading is accomplished using techniques suggested by [App, 2002c], while unaligned store uses suggestions from [Ollmann, 2001; Lai and McKerrow, 2001b]. When processing arbitrary image sizes, leftovers are computed using the scalar processor. Instead of the scalar processor, the vector processor can be used by computing an entire vector of values, and grafting the result back onto the original. Using the vector processor is probably better for a machine vision library since the user needs only to supply one version of the functor. However, [App, 2002c] suggests that using the scalar processor would be faster, because since the scalar processor is able to work in parallel with the vector processor, it is possible to get the leftover

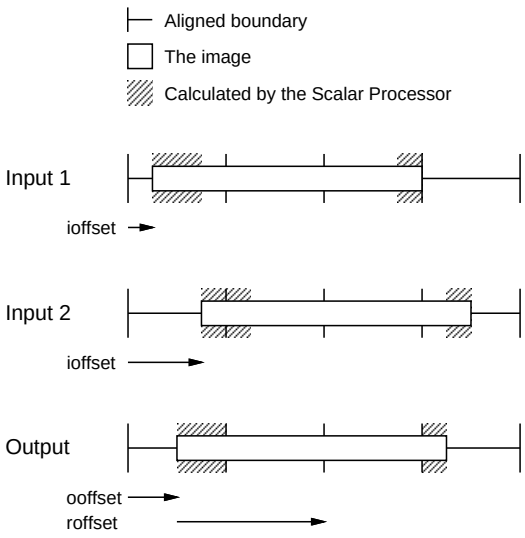

Figure 2: Aligning the input to the output for Algorithm 3

parts computed for free. In addition, using the scalar processor is safer because it does not involve writing to a location beyond the end of the array. For the vector version to be safe, the array allocated needs to be longer by one vector.

For the entire 1-D array, there can only be a single leftover piece at the end of the array. This is because the unaligned loading technique loads the unaligned beginning to an aligned location. Only the end bit is not loaded into the vector processor and computed normally.

The results from this initial version is unfortunately less than spectacular. The program loses half its potential speedup just trying to do unaligned store. As is clearly evident from Figure 3, unaligned store is the major cause of loss of speed. The cost of unaligned loading on the other hand is not too bad.

Clearly the next step is to remove the unaligned store. A simple approach is simply to constrain all outputs to aligned locations. A more elegant approach is to load the input, aligned to the output. This means that the data is loaded unaligned, but can be stored aligned, even though the output location is unaligned. However, because the output can start and end on unaligned positions, there will be two leftover pieces – one at the beginning and one at the end. There is another place where such an algorithm could lose speed. Before reading, the algorithm will need to calculate the offset from the start of unaligned input data to the location that corresponds with the first aligned location in the output data. This offset can result in loading the first aligned vector from either the first or second input vector, as illustrated by Input 1 and Input 2 respectively from Figure 2. Figure 2 illustrates how input images would align to the output image. `ioffset`, `ooffset` and `roffset` from Algorithm 3 are also depicted in Figure 2.

The resultant algorithm is shown in Algorithm 3. It supports loading and storing to unaligned memory locations. Algorithm 3 uses the `DST` macro to include data streaming instructions.

Algorithm 2 Vectorized version of Algorithm 1

```
 1 template<class T,class V,class F,class SF>
 2 void transform(T* start, T* end, T* out,
 3                F f, SF sf)
 4 {
 5     typedef __vector unsigned char vec_uc;
 6     const int step = sizeof(V)/sizeof(T);
 7     V ov, iv, iv_xtr;
 8     vec_uc ifix;
 9
10     int count = end - start;
11     T* pi = start;
12     T* po = out;
13
14 #ifdef DST
15     vec_dst(pi, 0x10010100, 0);
16 #endif
17 #ifdef UNALIGNED_LOAD
18     // Do Initial load
19     iv_xtr = vec_ld(0, pi);
20     pi += step;
21     ifix = vec_lvsl(0, pi);
22 #endif
23     for(int i = 0; i < count; i += step)
24     {
25 #ifdef DST
26         vec_dst(pi, 0x10010100, 0);
27 #endif
28 #ifdef UNALIGNED_LOAD
29         // Load unaligned
30         iv = iv_xtr;
31         iv_xtr = vec_ld(0, pi);
32         pi += step;
33         iv = vec_perm(iv, iv_xtr, ifix);
34 #else
35         iv = vec_ld(0, pi);
36         pi += step;
37 #endif
38
39         ov = f(iv);
40
41 #ifdef UNALIGNED_STORE
42         // Write unaligned
43         store(ov, (vec_uc*)po);
44         po += step;
45 #else
46         vec_st(ov, 0, po);
47         po += step;
48 #endif
49     }
50 #ifdef UNALIGNED_STORE
51     // Do end with scalar processor
52     int starti = (count / step) *  step;
53     for(int i = starti; i < count; ++i)
54         out[i] = sf(pi[i]);
55 #endif
56 }
```

Algorithm 3 Optimised version of Algorithm 2

```
 1 /** This vector loop will load unaligned data
 2  *  to the alignment of the output, so that
 3  *  only unaligned load is needed
 4  */
 5 template<class T,class V,class F,class SF>
 6 void transform(T* start, T* end, T* out,
 7                F f, SF sf)
 8 {
 9     typedef __vector unsigned char vec_uc;
10     const int step = sizeof(V)/sizeof(T);
11     V ov, iv, iv_xtr;
12     vec_uc ifix;
13     int count = end - start;
14     T* pi = start;
15     T* po = out;
16 #ifdef DST
17     vec_dst(pi, 0x10010100, 0);
18 #endif
19     // Load location = input offset +
20     //   (element count - output offset)
21     // % step required because want values
22     //   0 to element count - 1
23     int ioffset = ((unsigned long)pi & 0xf);
24     int ooffset = ((unsigned long)po & 0xf);
25     int roffset = ((step - ooffset) % step);
26     if(roffset + ioffset > step)
27         pi += step;
28     // Do Initial load
29     iv_xtr = vec_ld(0, pi);
30     pi += step;
31     ifix = vec_lvsl(roffset, pi);
32     // Do front with scalar processor
33     for(int i = 0; i < roffset; ++i)
34         out[i] = sf(start[i]);
35     po += roffset;
36     // Do middle with vector processor
37     for(int i = roffset; i < count; i += step)
38     {
39 #ifdef DST
40         vec_dst(pi, 0x10010100, 0);
41 #endif
42         // Load unaligned
43         iv = iv_xtr;
44         iv_xtr = vec_ld(0, pi);
45         pi += step;
46         iv = vec_perm(iv, iv_xtr, ifix);
47         // Do operation
48         ov = f(iv);
49         // Write aligned
50         vec_st(ov, 0, po);
51         po += step;
52     }
53     // Do end with scalar processor
54     int starti = (count / step) *  step;
55     for(int i = starti; i < count; ++i)
56         out[i] = sf(start[i]);
57 }
```

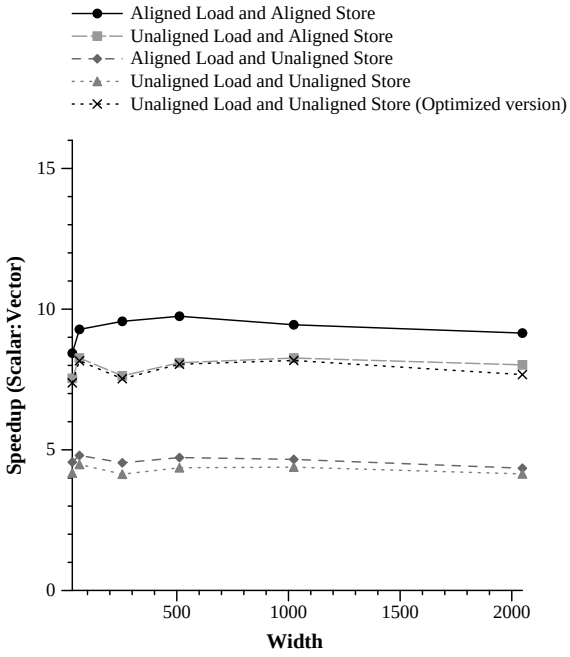

Figure 3: Effect of Alignment on Speedup

Figures 3, 4 and 5 show the effect of data alignment, data streaming instructions and the complexity of the functors on speedup. As expected, unaligned load and unaligned store both reduce the speed of the program. In addition, as mentioned previously, unaligned store has a much larger negative impact on performance than unaligned load. The optimised version from Algorithm 3 performs as expected – it is just a bit slower than unaligned load and aligned store. Figure 3 clearly shows that speeds increase to a maximum and then trail off. The speedup is lower for very small sizes because of the larger overhead involved in using the vector processor. The speedup tails off for large data sizes, probably because the program becomes more dependent on memory. There is no operation executed for the data that is loaded from memory.

From Figure 4, it is easy to see that adding data streaming instructions, as suggested by [App, 2002b], allows the program to run faster, reaching a higher maximum speedup and sustaining the speed for larger image sizes.

Increasing function complexity should reduce the memory-dependency of the algorithm, thereby allowing the speedup ratio to be increased. Figure 5 shows mixed results. While increasing function complexity does help it perform better, the speedup ratio does not seem to be proportional. Increasing function complexity should have positive effects because it does more with the data once it is in memory - a technique suggested by [App, 2002d] for improving program performance.

Figures 3, 4 and 5 show that it is possible to get speedups even with unaligned loading and unaligned storing. However, they also show that the speedup is not at its theoretical maximum. Nevertheless, Algorithm 3, in the authors' opinion, shows a good speedup ratio.

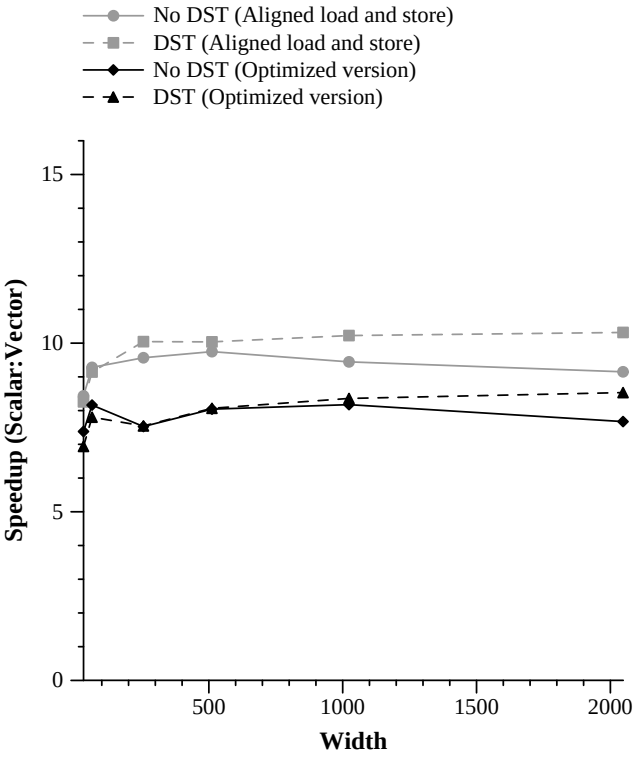

Figure 4: Effect of DST on Speedup

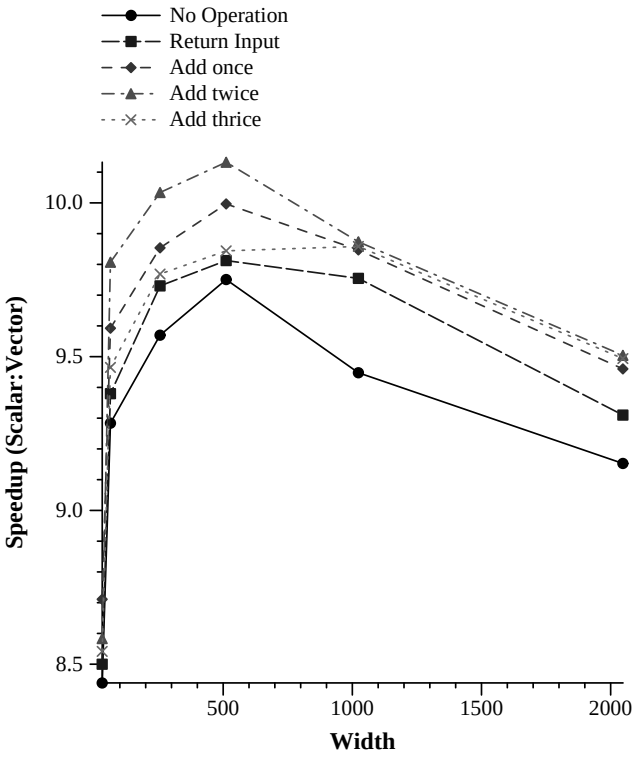

Figure 5: Effect of function complexity on Speedup

5 Some sample functors

Algorithms 1, 2 and 3 will accept functions and functors. A simple function is shown in Algorithm 4. This function doubles the image values.

Algorithm 4 A function that doubles the original pixel value

```
template<class V>
V simple(const V& in)
{ return vec_add(in, in); }
```

A more useful example is the threshold functor described in Algorithm 5, which computes Equation 1. Algorithm 5 sets the output to true and false, 0xFF and 0x00 respectively. Returning 0x01 instead of 0xFF, as in toolkits such as IMAQ (Ref [Nat, 1999]), will be a bit slower because an additional vector operation will be required to convert all 0xFF to 0x01.

$$f(X_0) = \begin{cases} true & \text{if } X_0 \geq \text{threshold} \\ false & \text{if } X_0 < \text{threshold} \end{cases} \quad (1)$$

Algorithm 5 Vector Threshold Functor

```
template<class V>
class Threshold
{
public:
    Threshold(const V& threshold)
    : mThreshold(threshold)
    {}
    V operator()(const V& a)
    { return (V)vec_cmpgt(a, mCmp); }
protected:
    V mThreshold;
};
```

6 Conclusion

The beginnings of a generic vectorized machine vision library was presented. Only algorithms that require a single input pixel from a single input image source to compute a single output pixel are considered. Examples of such algorithms are threshold, and lookup operations. The technique discussed can be easily extended to apply to two input images, so that arithmetic and logical operations can considered. Convolutions however, do not fit this model at all.

Using the vector processor can produce significant gains in speed. Unfortunately, the speedup is not close to the attainable theoretical speedup. Several factors affect the maximum speedup attainable by the vectorized program. Alignment appears to have the most impact. Unaligned data leads to lower speedups. Function complexity should help speedup, but the results obtained appear inconclusive. Data streaming instructions can increase the maximum speedup and helps the function sustain higher speedups for larger image sizes.

There is still room for improvement in the algorithms discussed. The main area of optimisation concerns unaligned storage. Unaligned storage can be avoided by imposing constraints on the use of the function. For some applications, such a method is perfectly valid. For a machine vision library however, such constraints limits its usefulness and ease-of-use.

Additional Resources

Source code and other additional information about this paper will be published on the web, and be available through http://www.bclai.net.

Acknowledgements

This work is funded by an Apple University Consortium grant.

References

[AltiVec.org, 2002] AltiVec.org. The altivec information source, 2002. see http://www.altivec.org.

[App, 2002a] Apple Computer Inc. *Apple's AltiVec Home Page*, 2002. see http://developer.apple.com/hardware/ve/.

[App, 2002b] Apple Computer Inc. *The Caches*, 2002.

[App, 2002c] Apple Computer Inc. *Code Optimization*, 2002.

[App, 2002d] Apple Computer Inc. *Performance Issues: Memory Usage*, 2002.

[App, 2002e] Apple Computer Inc. *Throughput and Latency*, 2002.

[Köethe, 2001] Ullrich Köethe. *VIGRA Reference Manual for 1.1.2*, April 2001.

[Lai and McKerrow, 2001a] Bing-Chang Lai and Phillip John McKerrow. Image processing libraries. Ausralian Conference on Robotics & Automation, November 2001.

[Lai and McKerrow, 2001b] Bing-Chang Lai and Phillip John McKerrow. Programming the velocity engine. In Neville Smythe, editor, *e-Xplore 2001: a face to face odyssey*. Apple University Consortium, September 2001.

[Mot, 1999] Motorola, Inc. *AltiVec Technology Programming Interface Manual*, 1999.

[Nat, 1999] National Instruments. *IMAQ Vision User Manual*, May 1999.

[Ollmann, 2001] Ian Ollmann. Altivec. 2001.

[Stokes, 2000] Jon "Hannibal" Stokes. Sound and vision: A technical overview of the emotion engine. *arstechnica.com*, March 2000.