

# Vectorization of Insertion Sort using Altivec, and an Extended Merge Sort algorithm

©2005 Konstantinos Margaritis, markos@debian.gr  
Licensed under the LGPL

11/07/2005

## Abstract

Parallelization of a sort algorithm is not always easy nor possible, or if it is, this does not necessarily mean significant gains in performance. After reading in Slashdot about a GPU version of Quicksort (lgpu), I was curious if vectorization of common sorting algorithms, such as qsort, insertion sort and merge sort could be vectorizable and adapted to be used with Altivec, the PowerPC SIMD unit. The results were more than interesting, in some cases offering speed gains of 54%!! in this paper, we will present vectorization techniques for all 3 of these algorithms. Note: This paper does not yet include the quicksort vectorization.

## 1 Vectorizing Insertion sort

The original (scalar) Insertion sort algorithm is at worst an  $O(N^2)$  algorithm. While simple in concept, it's not considered one of the fast sorting routines so it's not very popular in code where performance is necessary. A very thorough description of the algorithm can be found in [wika]. We provide here a simple implementation against which comparisons have been made.

```
void inssort_c(uint8_t a[], size_t N) {
    int i, j;
    for(i = 1; i < N; i++) {
        int value = a[i];
        for (j = i-1; j >= 0 && a[j] > value; j--) {
            a[j+1] = a[j];
        }
        a[j+1] = value;
    }
}
```

Here, the algorithm will sort a set of  $N$  unsigned chars. The sorting will be stable, meaning that the position of equal elements will not be changed. The algorithm implementation remains the same in case we're dealing with ints or floats.

The basic concept of vectorization implicitly denotes the steps we must take for vectorization of this algorithm. We shall divide the set of  $N$  elements in 16

Figure 1: Sorting 16 sets in parallel using Altivec

22	4	18	55	44	81	3	22	2	34	8	49	42	30	44	75
8	6	13	35	19	9	90	89	109	55	6	22	39	3	52	145
11	12	5	68	15	51	41	0	25	3	5	34	99	39	2	5
5	7	2	32	53	9	1	10	12	5	69	19	91	36	84	45



5	4	2	32	15	9	1	0	2	3	5	19	39	3	2	5
8	6	5	35	19	9	3	10	12	5	6	22	42	30	44	45
11	7	13	55	44	51	41	22	25	34	8	34	91	36	52	75
22	12	18	68	53	81	90	89	109	55	69	49	99	39	84	145

sets (as an Altivec register can hold 16 chars, which can be processed in parallel). Again, the situation is analogous if we're having ints or floats deviding into 4 sets (or 8 sets if dealing with short ints).

The idea is to sort 4/8/16 sets in parallel and then merge the results. This is actually a variant of the Shell Sort algorithm ([wikb]). How we will do the merging stage, still remains to be explained later. First we will have to deal with the sorting of 16 parallel columns of data. In theory this is easy, but we must make sure we handle a vector at a time, and in as few instructions as possible, preferably in one go. To do that, we first have to load the data into Altivec registers. We will use two registers as the comparison will be used between two sets, one used as the key. Obviously, this means that the Altivec version will work only for sizes > 32 chars (which is the size of 2 vectors).

```
vector uint8_t va_key, va_cur,
vector unsigned char *cur;
if (length >= 32) {
    // How many sets of N/16 do we have to deal with?
    loops = length/16;
    for(i = 1; i < loops; i++) {
        // Get the key set of elements
        // and load it into an Altivec vector register
        cur = a + i*16;
        va_key = vec_ld(0, cur);
        // Compare all the previous sets to the key set.
        for (j = i-1; j >= 0; j--) {
            cur = a + j*16;
            va_cur = vec_ld(0, cur);
            // Compare the sets
        }
    }
}
```

```
}  
}
```

How do we compare 2 sets of 16 characters at once? Even more tricky, how do we swap the elements between themselves in the description and implementation of the Insertion sort algorithm? Altivec comes to the rescue with its powerful comparison and selection instructions. Using `vec_cmpgt`, Altivec provides us a comparison mask of the 2 vectors, which we can use then to select individual elements of each vector, to combine new vectors.

Let's explain this a little more:

- The algorithm compares keys at positions  $i$  and  $j$  (initial  $j$  value is set to  $i - 1$ )
- While  $a[j] > a[i]$  the  $a[j]$  is moved one position down to  $a[j + 1]$ . Decrease  $j$ , while it's greater than zero.
- When the first element is found that is smaller or equal to  $a[i]$ , it is replaced by  $a[i]$ .

To do this in parallel for 16 sets, we would have to do 16 comparisons at once, and how would we handle 16 swaps, that might happen in totally different positions in each set? Instead of swapping elements, we create a new vector that holds the appropriate value according to the comparison mask. So the swapping code is equivalent to:

- Generate comparison mask for elements  $a[i]$  and  $a[j]$ .
- Generate first result using the comparison mask and the two elements as operators (eg. select  $a[i]$  if mask is 1, or  $a[j]$  if mask is 0).
- Generate second result using the same comparison mask but reversed the order element (ie. select  $a[j]$  if mask is 1,  $a[i]$  if mask is 0).
- Store the first result in  $a[j]$

Now this repeated 16 times over a vector is what Altivec does. What is the benefit? The benefit is that this code does not use any branches and it can be pre-calculated in the pipeline by the processor, giving very good performance results. Plus it is constant, meaning it will take exactly the same number of instructions regardless the data given. So the initial scalar loop can be replaced by the following Altivec instructions (the following code sorts an array of chars):

```
int vecinssort_c(uint8_t a[], int length) {  
    vector uint8_t va_key, va_cur, va_next, va_first, va_second;  
    vector bool char va_cmpmask;  
    vector uint8_t *cur;  
    int i, j, k;  
    if (length >= 32) {  
        // How many sets of N/16 do we have to deal with?  
        loops = length/16;  
        for(i = 1; i < loops; i++) {
```

```

// Get the key set of elements
// and load it into an Altivec vector register
cur = a + i*16;
va_key = vec_ld(0, cur);
// Compare all the previous sets to the key set.
for (j = i-1; j >= 0; j--) {
    cur = a + j*16;
    /* Load current and next 16-byte sets
    into Altivec registers
    */
    va_cur = vec_ld(0, cur);
    va_next = vec_ld(16, cur);

    /* Generate the comparison mask between
    the current vector and the key vector
    */
    va_cmpmask = vec_cmpgt(va_cur, va_key);

    /* And construct the first and second
    result vectors to replace the 2 16-byte sets
    */
    va_first = vec_sel(va_cur, va_key, va_cmpmask);
    va_second = vec_sel(va_next, va_cur, va_cmpmask);

    // Store the vectors to their appropriate
    // positions.
    vec_st(va_first, 0, cur);
    vec_st(va_second, 16, cur);
}
}
}
}

```

The result will be 16 sorted sets of  $N/16$  elements. As the algorithm is still basically an Insertion Sort the time required is still  $O(N'^2)$  but this time  $N' = N/16$ . So for example for  $N = 64$  elements, the original algorithm would do  $N^2 = 4096$  steps to execute, while the Altivec version would do just  $N' = 4$ ,  $N'^2 = 16$  steps!!

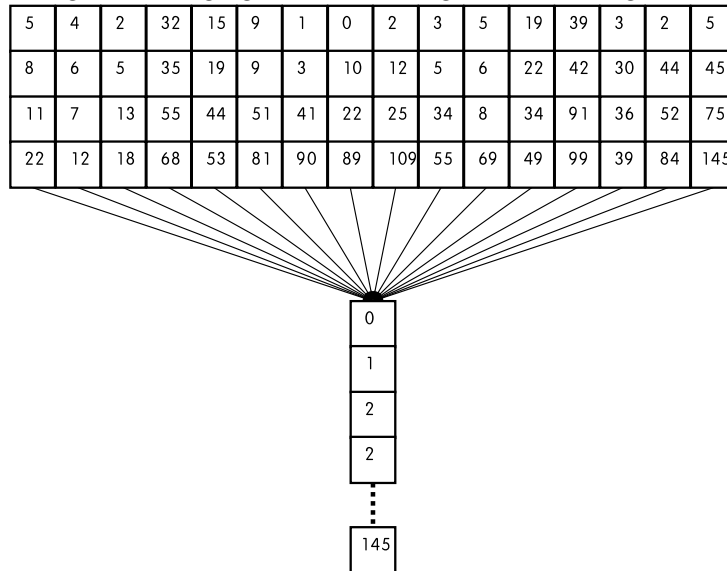
So is this over? No, because the result is still not what we want, we have to merge the remaining sets to a final sorted set. For that reason we have to use an extended version of Merge Sort.

## 2 Extending/Vectorizing Merge sort

Why extended? Well, the original Merge Sort algorithm copes for 2 sorted sets, while we need to merge 4/8/16 sets of already sorted elements.

The process is quite quite generic (i.e. not Altivec specific), with a similar concept to the 2-set case, but obviously extended to cover more (at first we will

Figure 2: Merging the 16 sets using extended Merge sort



initially cover  $N$  cases, optimizing afterwards for 4/8/16 sets). Let's begin by describing the algorithm, assuming we have  $N$  sets of sorted data:

1. Initialize a `indices[]` array of integers, with size  $N$ . Set all elements to 0.
2. Initialize an `flag[]` array of booleans, size  $N$ . Set all elements to 0.
3. Initialize the `result[]` array to store the final sorted data (which has size  $N * \text{columnsize}$ ).
4. Get the next  $N$ -tuple of data which we want to sort. The  $N$ -tuple will be comprised of the elements in the set, each in the position indicated by the `indices[]` array.
5. Find the minimum value(s) in the  $N$ -tuple and note its position(s), `pos`.
6. Insert the value(s) to the next available slot in the `result[]` array.
7. Increase `indices[pos]` by 1 (for all elements if more than one is found).
8. Set `flag[pos]` to 1.
9. Go back to 5 until all elements in `flag[]` are set to 1.
10. Go back to 4 while there are remaining sets.

This looks even easier if you see it in code:

```
#define INDEX(n, x, y)    ((x)*(n) + (y))

int mergesort_c(uint8_t a[], size_t sets, size_t columnsize) {
    // Set up variables
```

```

uint32_t flag = 0, remsets = sets, length = sets*columnsize;
size_t i, j, set = 0, index = 0;
uint8_t min, cur;

// Initialize and zero the counters[] array.
size_t counters[sets];
memset(counters, 0, sets*sizeof(size_t));
// Allocate the memory for the final sorted array.
uint8_t *target = malloc(length*sizeof(uint8_t));
if (target == NULL) {
    printf("Error: mergesort_c(): malloc() \\
        failed to allocate %d bytes\n", length);
    exit(10);
}

// While we have remaining sets to sort, loop
while (remsets) {
    // set up the initial min value to the max value,
    // so that it gets reset immediately.
    min = UCHAR_MAX;

    // Loop over all the sets
    for (j=0; j < sets; j++) {
        // Check if this particular set has been finished
        // (its bit is set) otherwise consider it when
        // searching for the min value.
        if (!(1 << j) & flag) {
            cur = a[INDEX(sets, counters[j], j)];

            if (min >= cur) {
                min = cur;
                set = j;
            }
        }
    }
    // Insert the min value to the next available position
    // in the final array
    target[index++] = min;
    // Increase the counter for this particular set
    counters[set]++;
    // If this counter exceeds the columnsize, then
    // we have finished the set. Set its bit in
    // the flag variable to 1 and decrease remsets.
    if (counters[set] >= columnsize) {
        flag |= (1 << set);
        remsets--;
    }
}
// Copy the target array to our given array and free target
memcpy(a, target, length);

```

Size	It. (Altivec)	It. (Scalar)	Ratio
8	41	14	0.33
16	86	59	0.75
32	188	286	1.10
64	440	917	1.33
128	1136	4335	2.25
256	3296	16019	2.67
512	10688	66644	3.24
1024	37760	262649	3.50
2048	141056	1064615	3.73
4096	544256	4237939	3.81
8192	2137088	16521839	3.76

Figure 3: Insertion Sort: Altivec versus Scalar (ints)

```

if (target)
    free(target);

return;
}

```

You may have noticed that the code is scalar and with nothing Altivec specific. It is possible to do some Altivec optimizations in this one, especially with regards to finding the minimum value, as Altivec provides such a function (`vec_min`) that will find the minimum value in a vector in just 1 CPU cycle! This will follow in a next revision of this paper.

Also worth mentioning, we have used a 32-bit integer and setting bits to denote that a set has been completed. This was done for performance reasons, but it also presents a limitation. We can't use more than 32 sets of sorted data to merge. But that's quite acceptable for our purposes, as we only need 16 sets maximum.

This Extended Merge Sort algorithm, like the original algorithm is not an in-place sorting algorithm, that is, the data get sorted in another array and then copied back to the original array. We have used `memcpy()` for the copying process, as we believe it to be faster than other methods (eg. per-element copying), but there is of course no other particular reason or this choice. In particular, since this function has already been vectorized for Altivec, we might benefit from that as well.

## 2.1 Performance Results

Here we will present the results of using our Insertion sort for unsigned integers and characters (ie 32-bit and 8-bit integers) for 100 loops. Figure 3 presents the case with 32-bit integers, while figure 4 presents the same results for 8-bit unsigned characters. As you can see, the Altivec version gives really impressive results! Of course this has to do with the advanced Merge Sort and the fact that for large sizes, the chance of finding the same keys increases. This is obvious for chars, as there are only 255 available values for a char.

Size	It. (Altivec)	It. (Scalar)	Ratio
128	2048	4199	1.00
256	4096	16429	2.00
512	8192	67361	4.25
1024	16384	259680	7.33
2048	32768	1036143	13.79
4096	65536	4116630	20.27
8192	131072	16996544	29.85
16384	262144	67815998	37.50
32768	524288	265531367	42.12

Figure 4: Insertion Sort: Altivec versus Scalar (chars)

### 3 Conclusions

Altivec is really a very powerful tool but so far its use is really mainly centered around Multimedia and or Linear Algebra scientific applications. We strongly believe that Altivec can be of real benefit to generic system-wide OS usage as well. We will show eventually with a series of papers like this one that it can be used for other generic applications, like data manipulation, other sort algorithms, etc.

### References

[gpu] Gpusort.

[wika] Shell sort.

[wikb] Shell sort.