A Vectorized Hash-Join

Rich Martin
University of California at Berkeley
rmartin@CS.Berkeley.EDU
May 11, 1996

Abstract

A vector instruction set is a well known method for exposing bandwidth to applications. Although extensively studied in the scientific programming community, less work exists on vectorizing other kinds of applications. This work examines vectorizing a traditional database operation, a Grace hash-join. We how to vectorize both the hash and join phases of the algorithm, and present performance results on a Cray C90 as well as traditional microprocessors. We concluded that vector scatter-gather and compress are essential to both this algorithm as well as to other non-scientific codes.

1. Introduction

A well known method for exposing bandwidth at the architectural level is through a vector instruction set architecture. Although extensively studied in the scientific programming community, little work exists on vectorizing other types of applications. This work examines vectorizing a traditional database operation, a Grace hash-join.

The join operation is one of the most time-consuming and data-intensive operations performed in relational databases. The join operation is also a frequently executed relational operator [5]. Due to its high cost and frequency, hundreds of papers exist on a multitude of facets of the join operation. Most of the cost models presented attempt to minimize the number of disk accesses because disk accesses are the most expensive operation. This work explores vectorizing the computational aspects of the hash and join phases.

2. Background

For the purposes of this paper, a **relation c**an be thought of as a table, and a **tuple** as row in the table. An **attribute** is a field type in the table row.

This section presents a short review of the equijoin. In an equijoin, = is the operator used to compare attributes. Different kinds of joins use other comparison operators besides =. For a complete description of the join operator, see [5].

Relation R

Product	Customer
Ultra	Dave
Indy	John
Alpha	Hank

Relation S

Customer	ZIP
John	94305
Hank	98195
Bill	02139

Relation Q

Product	Customer	ZIP
Indy	John	94305
Alpha	Hank	98195

FIGURE 1. Example Join Operation

Figure 1 illustrates the inputs and resulting output tables after an equijoin operation. For the relations R and S, the output relation Q is the table formed by concatenating attributes in R and S which have matching elements in the key attribute. For example, in Figure 1, the attribute *cus-*

tomer is used to form entries in Q from entries in R and S which have the same customer. We call the attribute type which is matched the *key*.

2.1 Nested-Loops Algorithm

In the naive algorithm, called the *nested-loops join*, each tuple in the first relation is compared with every tuple in the second relation. The pseudo-code below illustrates this operation:

```
For each row in S do
      For each row in R do
            if (Ra.key == Sb.key) then
                         concatenate Ra, Sb and place in result Q
```

The pseudo-code shows how this algorithm is $O(n^2)$. The nested-loops join might be fine for small relations [3], but for large relations, even ones that fit in memory, the cost is much too high.

2.2 Hash-Joins

We define the join *load* as the number of keys which must be compared between the two relations. The central idea in a hash-join algorithm is that the load can be reduced by hashing the tuples into buckets. Tuples which hash to different buckets will not need to be compared. Figure 2 shows how hashing reduces the join load. Each point on the abscissa represents a key from R. Likewise, each point on the ordinate represents a key from S. The lines represent the groupings of keys into buckets. In the nested-loops join, the number of comparisons is proportional to the entire area. In a hash-join algorithm, only those keys which hash to the same bucket (the shaded areas) need to be compared.

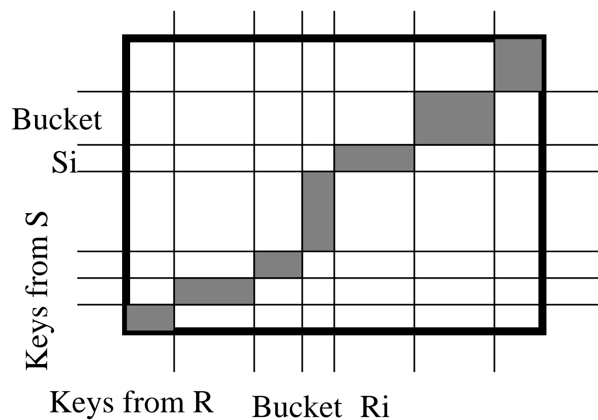

FIGURE 2. Reduction of join load by hashing

2.3 Grace Hash-Join

The Grace hash-join [4] uses hashing to reduce join load on two levels. On one level, hashing is used to break up large relations that reside on disk into buckets small enough such that each

bucket fits into memory. Once each bucket is small enough to fit into memory, hashing is used again to reduce the join load.

The Grace hash-join hash has two passes. In the first pass, the relations are hashed into separate buckets which resided on disk. Each bucket is small enough to fit into main memory. In the second pass, a bucket from one relation is brought into main memory and hash table is constructed from it. Then, for each record in the second relation, it's key is hashed and compared to every key which hashed to same bucket in the first relation.

The next section presents the sequential version of the hashing algorithm used in this study. The following sections present the vectorized version of the hash algorithm. Both the hashing to disk buckets phase and hashing to compare keys phase of the Grace hash-join use algorithm presented below.

2.4 Sequential Hash Algorithm

The hash algorithm used in this work is unlike most hashing algorithms in that it is not constructed for easy insert and delete operations. Rather, this hash algorithm and associated data structures are designed only to **group keys into buckets**. Recall that the main purpose of the building the hash table is to reduce join load, not to build a persistent hash table.

The hash algorithm used in this paper is closely related to the radix sort first described in [6]. In the first pass of the radix sort, the keys to be sorted are moved into buckets based on a digit within the key. They idea carried over from radix sorting to hashing is that both move keys into buckets. Unlike a generic hash, the radix sort uses a fixed hash function. The a hash algorithm has four phases: *Extract-Buckets*, *Histogram, Scan* and *Rank-and-Permute*. The next sections describe these phases in detail. Throughout the next sections, the term key and records will be used somewhat interchangeably.

2.4.1 Extract-Buckets and Histogram Phases

The extract-buckets phase maps each record into a bucket. It is very simple and will not be discussed further. In the histogram phase, a pass is made over every key. For each key, a counter of

its corresponding bucket number is incremented. Figure 3 illustrates this phase of the hash, for 9 keys and 4 buckets. The hash function used in this example is *key mod (number of buckets)*.

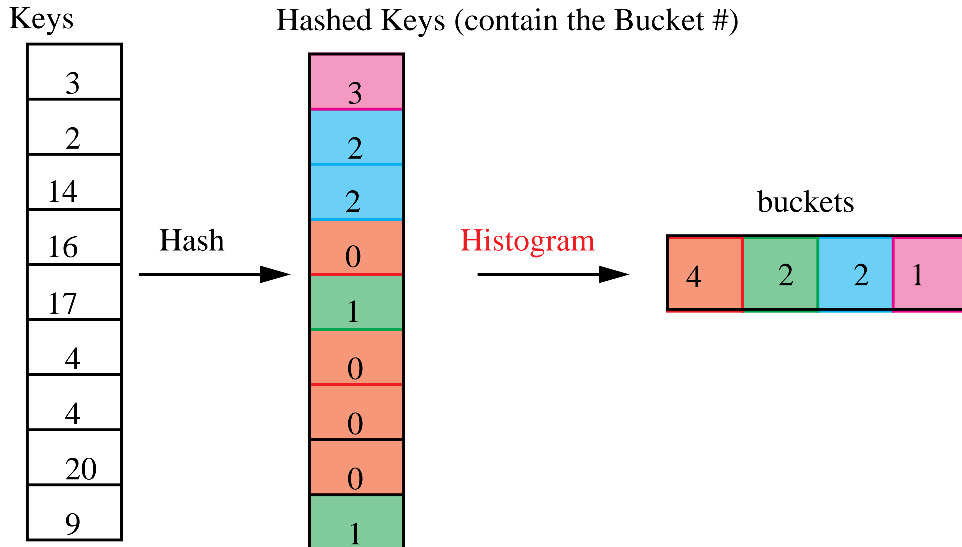

FIGURE 3. Building a histogram of keys

At the end of the histogram phase, the array *buckets* contains the number of keys which hash into each of the buckets. The pseudo code for this phase is very simple:

```
for (i := 0 to number of keys) {
      buckets[hashed_keys[i]]++ ;
}
```

2.4.2 Scan Phase

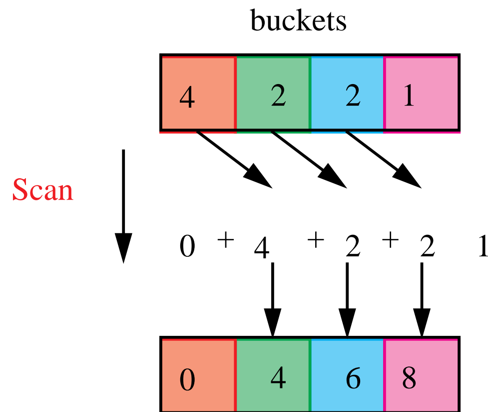

FIGURE 4. The Scan operation

The scan phase takes the array *buckets* and preforms a *+scan* operation on it. The *+scan* operation can be thought of as summing the array and shifting is over by one. Figure 4 shows the results of the buckets array after the scan. The pseudo-code is very simple:

```
sum :=0;
for (i :=0 to number of buckets ) {
        val := buckets[i];
        buckets[i] := sum;
        sum += val;
}
```

The important observation to make at this point is that after the scan phase, the ith entry in the *buckets* array contains the number of keys in the buckets *0... i-1*. The algorithm then uses this information to move records which hashed to the same bucket together.

2.4.3 Rank-and-Permute Phase

In the rank-and-permute phase, another pass is made over the keys. For each key, it's bucket number is determined and then the key is moved into a the result array. The key's position in the result array is indexed by the current bucket. The bucket position is incremented as each key is moved.

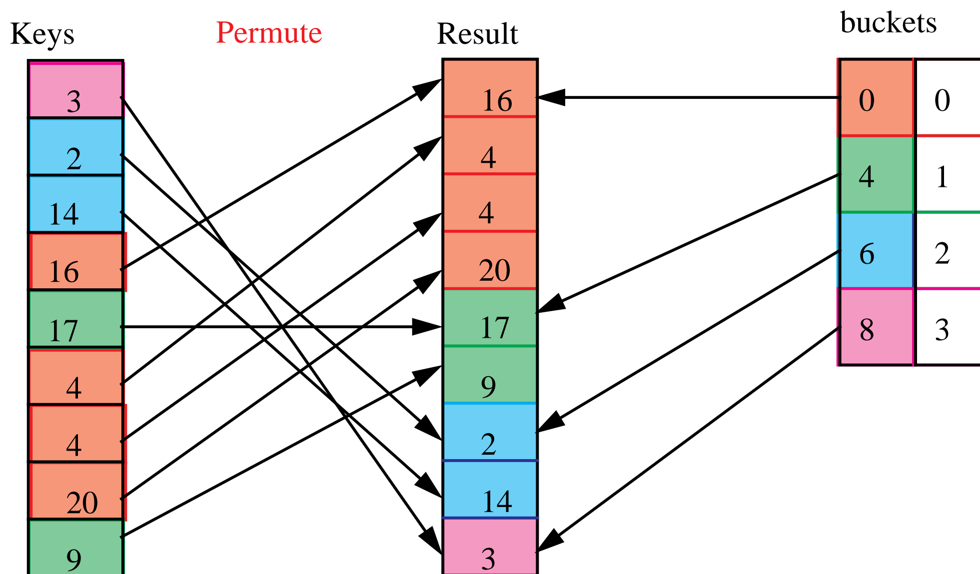

FIGURE 5. Rank and Permute Phase

Figure 5 shows how the keys are moved in the rank-and-permute step. The array *buckets* is used as an index vector to move keys into the result array. After the rank-and-permute step, the *result*

and *buckets* arrays form a hash table. We can find the first key of $bucket_i$ is found by indexing into the array *result[bucket[i]]*. The size of bucket *i* is given by *buckets[i+1]-buckets[i],* if *i<number of keys,* else *0.*

```
for (i := 0 to number of keys ) {
      rank := buckets[hashed_keys[i]];
      buckets[hashed_keys[i]]++;
      result[rank] := keys[i];
}
```

2.5 Sequential Join Algorithm

The sequential join algorithm is very simple. First, the disk bucket R_i is brought into memory. Next, a hash table is constructed with as many buckets as possible on R_i. Increasing the number of buckets reduces the keys per bucket which in turn reduces the join load. Next, the algorithm passes once through each record in the corresponding disk bucket S_i. Each record of S_i is hashed and then compared to every record in R_i which hashed to the same bucket. If any matches are found, the records are concatenated and written to the final output file.

3. Vectorized Algorithm

The vectorized algorithm follows the same steps as the sequential algorithm. The basic difference is the data structure used to build a hash table. Instead of a one-dimensional array *buckets* we expand the array to two dimensions. We use the two techniques first described in [6]: v*irtual processors* and *loop raking*.

3.1 Vectorized Hash

The vectorized hash relies heavily on the idea of v*irtual processors*. Imagine trying to vectorize

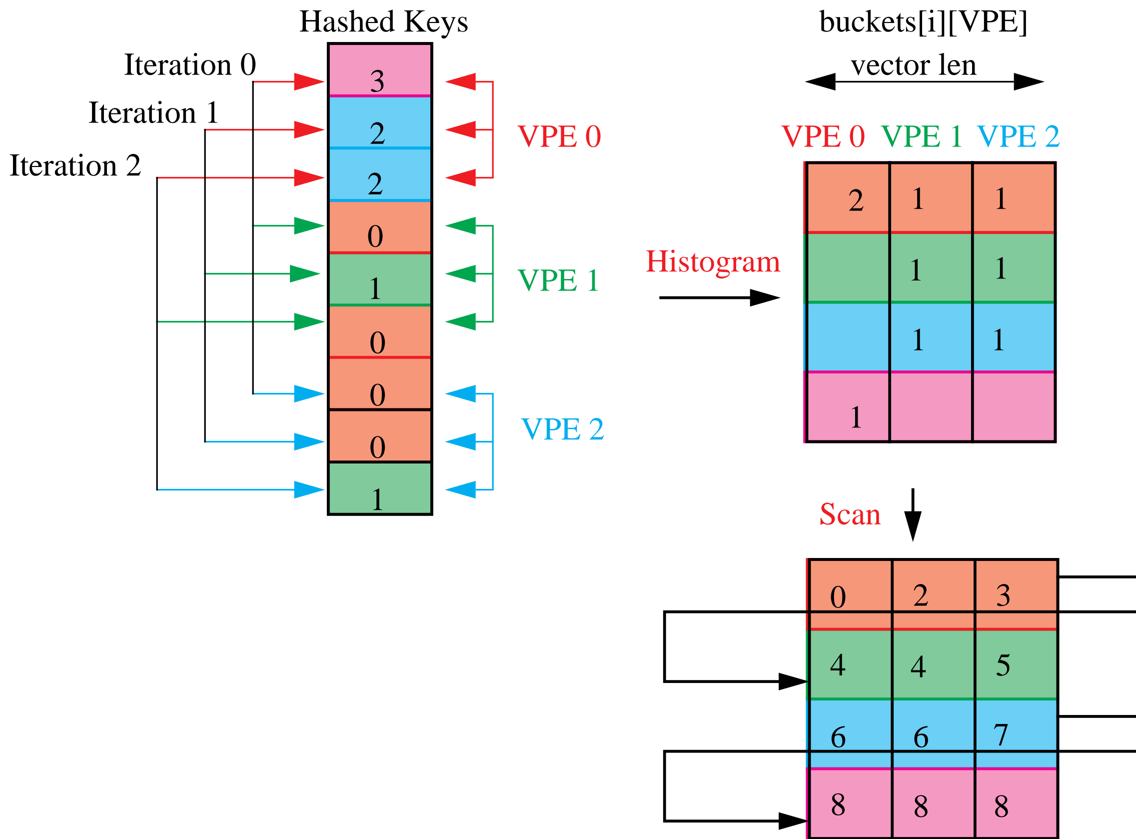

FIGURE 6. Vectorized Hash

the sequential version of the histogram phase shown in Section 2.4.1 on page 3. Multiple keys may map to the same bucket, which would imply more than one vector register would have to update the same bucket on the same loop iteration.

To solve this problem we look at every vector register as it's own SIMD processor. We call each of these "processors" a Vector Processing Element (VPE). The problem then becomes how to divide the keys among the VPEs. Because each VPE must have an independent "memory" (recall the SIMD nature of VPEs), we give each VPE its own copy of the array *buckets.* In effect, we have made the bucket array a two-dimensional array the number of VPEs wide. Figure 6 shows a bucket mapping with 3 VPEs (and thus a vector register length of 3). Notice how the how each column of the array *buckets* can now be updated independently. In the example shown in Figure 6, There would be 3 passes through the keys because 9 keys divide up into three passes with a vector length of 3 . During each pass, three buckets would be updated. The pseudo-code for this vectorized histogram is shown below. If the number of keys is not a multiple of the vector register length, than an second loop is needed to 'clean up' the extra elements.

```
for (i := 0 to Number of Elements Per VPE ) {
        for (vpe :=0 to number of VPEs ) {
                offset += Number of Elements Per VPE
                buckets[hashed_keys[i+offset]][vpe]++;
        }
}
```

The second technique used is loop raking. Loop raking was first developed to maintain the stability of radix sort. While not strictly necessary to for the correctness of hashing, loop raking is useful to prevent bank conflicts in the code. It also keeps the records in order. Figure 6 shows how the VPEs map to update the bucket array. VPE 0 (and thus vector register 0) is shown in red, VPE 1 in green and VPE 2 in blue. So on the first pass, the algorithm would generate bucket indexes for keys 0, 4 and 7.

The scan phase must now create a total ordering of all the keys. We can create a total ordering on the buckets by scanning each row of the buckets array and carrying the result to the next row. Figure 6 shows the resulting linear ordering "snake" after the scan is completed. Notice how the last column of the 2-dimensional array *buckets* is the same single dimensional case.

Vectorizing the scan operation is non-trivial and beyond the scope of this report. The reader is referred to [1] for the vectorized version of the scan.

3.2 Join

A equijoin builds a set of keys which match on the = operator. We look to the vectorized quicksort [6] to find a method to quickly build this set. In the vectorized quicksort, the keys are partitioned into two sets. One is the set of keys greater than the comparison key and the other set is less than the comparison key. In the vectorized join, we are only looking for the set of keys which are equal. However, both algorithms share the same basic idea. Both do a scalar-vector compare followed by a compress to "filter out" the correct set of keys. Figure 7 shows how an mask vector is generated by a scalar-vector compare followed by a compress to filter out the correct data.

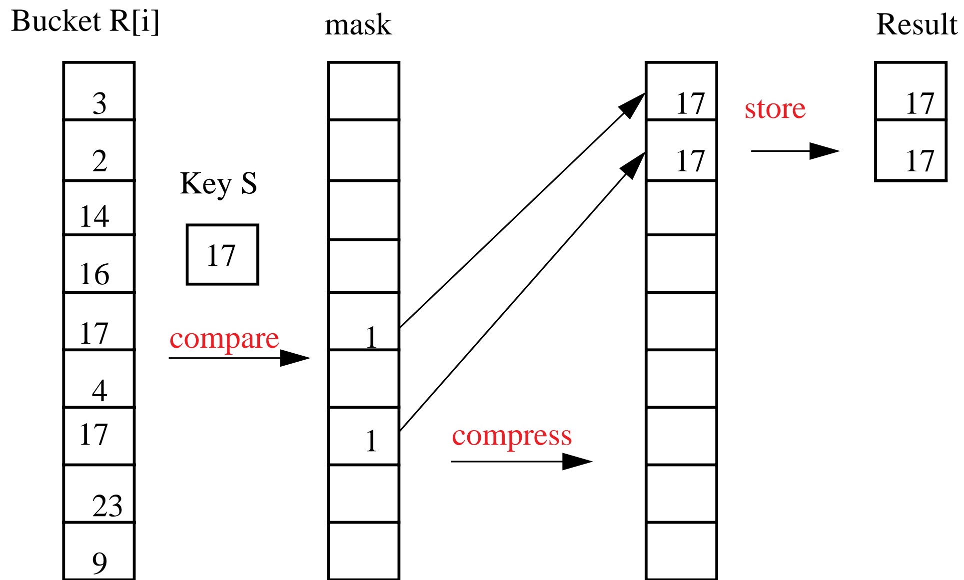

FIGURE 7. Vectorized Join

4. Implementation Details and Experience.

The hash-join program, HMJ, implements a the Grace hash-join algorithm. HMJ is written in standard C and runs on Unix workstations and the Cray C-90. HMJ assumes all keys are 32-bit integers, but can handle variable-length records, as long as each record is a multiple of the target machine's integer size.

An important point is that during the *Rank-and-Permute* phase, HMJ moves entire records, not just the keys. Moving entire records is necessary in the first pass of the Grace Hash-Join when the records are being moved into disk. During the join phase, it's not strictly necessary to move the entire record because if a key the first relation key doesn't match any key in the hashed bucket, then the record will be discarded.

The sequential version was written quickly. However, coaxing the compiler to vectorize certain loops in the vectorized version to an acceptable degree took about a week. For example, one would think the compiler would vectorize the two lines of the following loop in the same way. The most straightforward way to vectorize both lines in the loop would be a strided load, shift, mod and strided store.

```
      rec_p = (generic_rec *) input_buf;
      for (i=0,key_p = (unsigned int*) input_buf; i< num_rec ;
            i++, key_p += num_ints_rec) {
(1)         hash_keys[i]=  (rec_p[i].key >>32) % num_buckets;
(2)         hash_keys[i]=  ( (*key_p) >>32) % num_buckets;
```

```
}
```

The compiler did vectorize both loops. However, on the C-90, line 1 obtained an asymptotic rate of 2.5 µsec per element, which is the same as the scalar version. Changing the code in the inner loop to line 2 achieves an asymptotic rate of 0.02 µsec per record, a 124 times speedup! A quick review of the assembly code generated for line 1 shows many scalar instructions in the inner loop.

The astute reader will notice that the first line uses a structure definition to replace a general pointer operation. Cray's vectorizing compiler could not vectorize much code without defining the records as structures.

For the join phase, the compiler would not vectorize the compress. A similar result is reported in

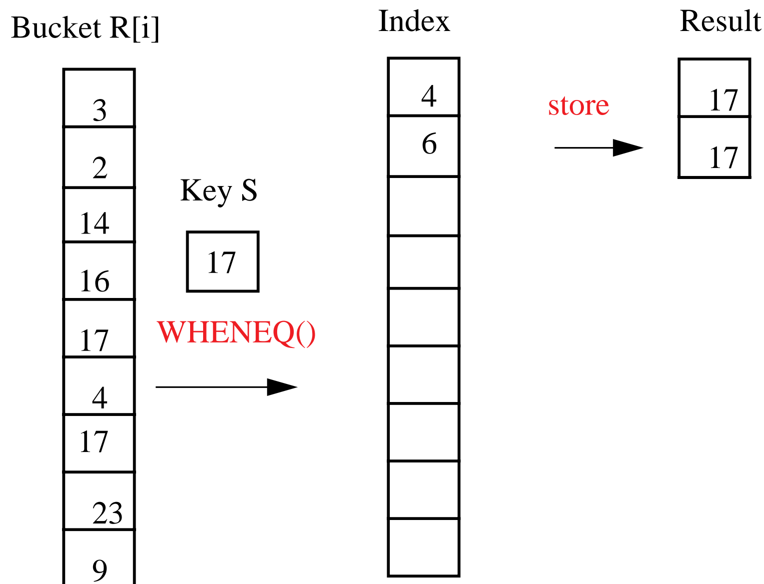

FIGURE 8. HMJ implementation of Join

the description of the vectorized the quicksort. HMJ uses the same work-around as the quicksort. Instead of using compress, the join phase builds an index vector using the WHENEQ() Cray library routine. WHENEQ() takes an input vector key and returns an index vector of all the elements in the input vector which match the key. The loop which permutes the keys into the final output array did vectorize with a pragma definition to the compiler. Calling a library routine for each input key is certainly not optimal, but it is much faster than the scalar code to implement the same function.

5. Results

Because this study concentrates on vectorizing the computation, the following section does not examine I/O time.

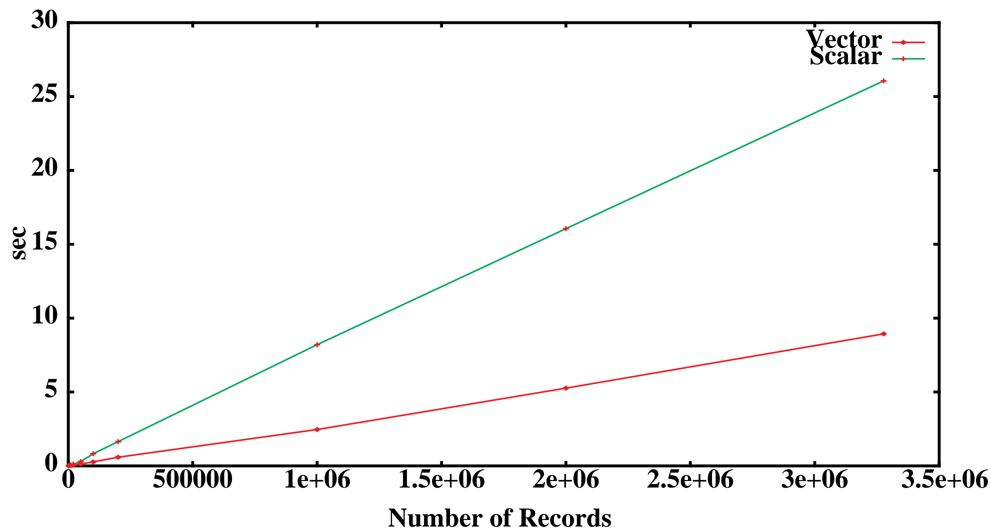

FIGURE 9. Vectorized vs. Scalar Compute Times

The plot in Figure 9 shows the total compute time on the C90 for the vectorized version of HMJ vs. the non-vectorized version. The vectorized version used the vectorized data structures and the scalar version uses the simpler scalar data structures. The plot shows the total compute time, in seconds, plotted against the combined number of records in both tables (so 1,000,000 records on the abscissa means each table was 500,000 records). Both tables had 64 byte records, and the keys were random 32 bit integers. The compute time is defined as the sum of the following compute phases: *Extract-Buckets, Histogram, Scan, Rank-and-Permute* and *Join*. The figure clearly shows that the vectorized version is faster than the scalar version.

!

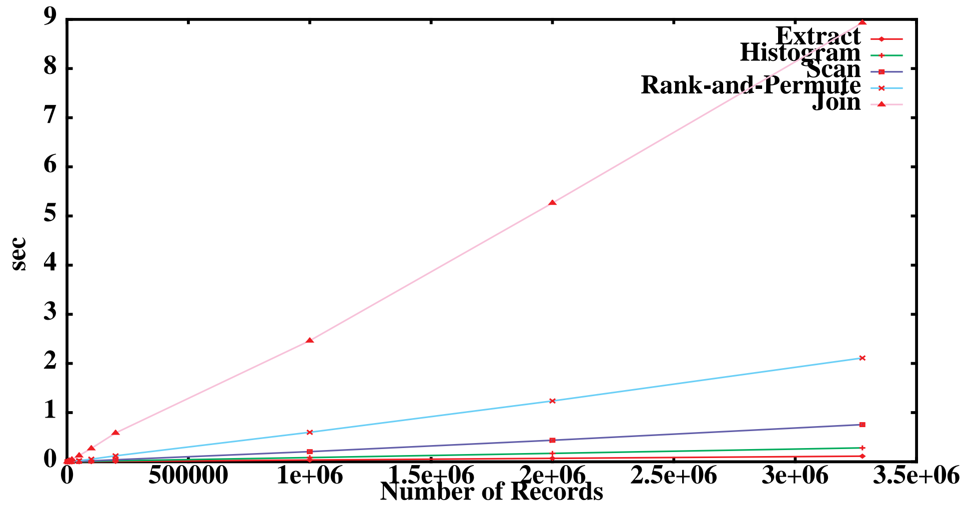

FIGURE 10. Vectorized Compute Times

Figure 10 shows the breakdown for the compute phases of the vectorized version of HMJ. The graph is a "stack-graph", so the time shown for a phase includes all the phases "below" it. The most interesting feature of the vectorized version is that the join time dominates all the other phases. Recall that for each key in the join, the function WHENEQ() is called. The vector code tries to size each bucket at 128 elements. This is because the fastest time to call WHENEQ() for the join phase was empirically determined to be 128 elements, which is exactly the vector length of the C90. This also has the property of reducing the number of buckets, which speeds the other phases. Unfortunately, time constraints did not permit empirically measuring or modelling the relationship between the number of buckets in the join phase and execution time.

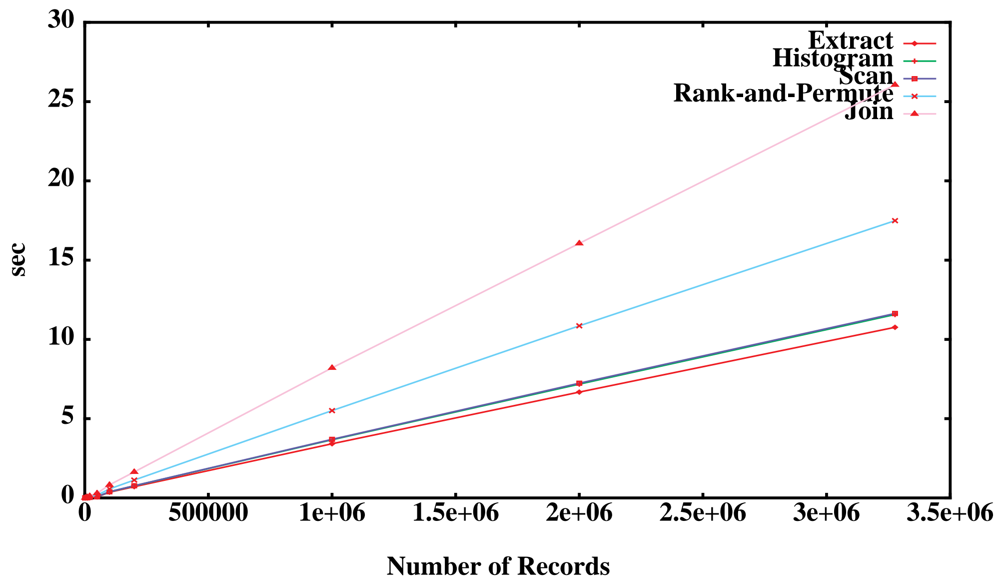

FIGURE 11. Scalar Compute Times

Figure 11 shows the same breakdown of phases as the vector version. Notice how slow the *Extract-Keys* phase is on the scalar code compared to the vector code. The join phase is also comparable to the vectorized version. The slowness of the scalar *Extract-Keys* exposes just how slow the scalar unit on the C90 is. The C90 designers seems to be flirting with a common pitfall: ignoring the speed of the scalar unit.

.

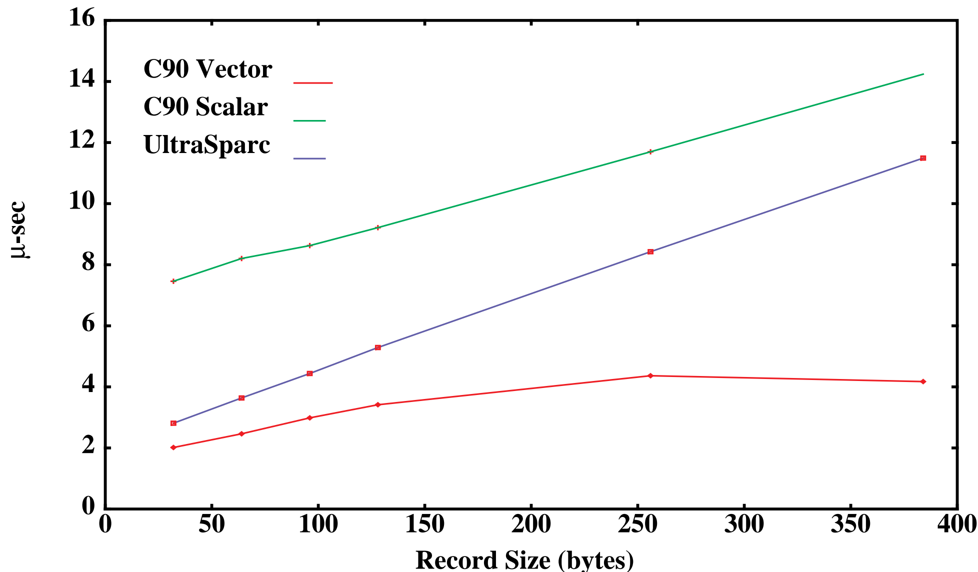

FIGURE 12. Asymptotic Time to Move a Record

Figure 12 shows where the vectorization can make a large difference in execution time. The three plots are the asymptotic time to move a record. That is, when running at the peak rate, how fast can joined data be moved through the processor? The time per record is obtained by taking the total computation time and dividing it by the number of records. An UltraSparc running the scalar code (the middle line) is included for comparison. For records of up to 128 bytes, the scalar HMJ on the UltraSparc and vectorized HMJ on the C90 are roughly comparable. When moving large records however, the vectorized HMJ is significantly faster. The C90's asymptotic rate for perform the Hash-Join on 384 byte records is one record every 4.1 µsec, which translates into a bandwidth for the entire compute phase of 94 MB/s! By contrast, the UltraSparc's asymptotic rate for the same sized record is one every 11.5 µsec, which translates into a Hash-Join bandwidth of 33.4 MB/s.

The other point to note is that the scalar unit on the C90 is slower than the UltraSparc. The Cray designers seem to be ignoring the importance of a fast scalar unit in the design of the C90.

6. Future work

This work did not present a cost model for the vector and scalar phases of the algorithm. One is sorely needed to evaluate trade-offs between a vector and scalar machine on this algorithm. Also, a cost model is needed in the join phase to determine the optimal bucket size. While the sequential version should make the buckets as small as possible, the same is not true in a vector machine. Trade-offs between comparing keys using vector instructions and reducing the number of buckets are possible.

7. Conclusions

The initial results in this paper show a hash-join can be vectorized. The results also show that vectorization can be accomplished even for applications written in C. The usefulness of the compress instruction for both the hash-join and quicksort cannot be overstated. Compress is essential for “filtering” data sets. While many traditional scientific codes do not “filter-out” data, the author (with no empirical evidence other than sorting and joining) would claim that is exactly what many large non-scientific codes do.

The importance of vector scatter-gather has been widely recognized: the author is unaware of any vector instruction set after the Cray-1 which did not have it. Scatter-gather is also a critical instruction for the vectorized hash-join.

8. Acknowledgments

The author would like to thank Andrea Dusseau for writing the original radix sort in C on the Cray Y-MP. The code was an invaluable guide for the hash algorithm. The author would also like Krste Asanovic for his contributions to the join algorithm.

REFERENCES

1. Chatterjee, S., Blelloch, G., Zagha, M. Scan Primitives for Vector Processors. In *Proceedings of Supercomputing '90*, pages 666-675, November 1990.

2. DeWitt, D., Gerber, R. Multiprocessor Hash-Based Join Algorithms. In *Proceedings of VLDB* 1985.

3. Dusseau, A., Ghormley, D., Keeton, K., Radix Sort: Squeezing Performance out of the Cray Y-MP. unpublished UC Berkeley CS-267 class project, April 1992.

4. Goodman, J., An Investigation of Mutiprocessor Structures and Algorithms for Database Management, Technical Report UCB/ERL M81/33, University of California, Berkeley, May, 1981.

5. Harris, E., Ramamohanarao, K. Join Algorithm costs revisited. *The VLDB Journal*, 5(1), pages 64-84, 1996.

6. Levin, S. A Fully Vectorized Quicksort. *Parallel Computing*, December 1990.

7. Mishra, P., Eich, M. Join Processing in Relational Databases. *ACM Computing Surveys*, March 1992

8. Zagha M., Blelloch G. Radix Sort for Vector Multiprocessors. In P*roceedings of Supercomputing '91*.