Program Optimization Project

Rachel Hwang

December 3, 2013

1 Procedure

For this project I chose Option A: accelerating on a single core. All computation was performed on a UChicago CSIL Linux machine (Ponyta). Configuration details are attached.

All cache rate information was collected using the valgrind cachegrind tool. Performance timing was done using the C time library $< time.h >$, by taking the difference in wall clock time at the beginning and end of the relevent code sections using $clock\_gettime()$. This is by no means a perfect method for measuring execution time, since we have little control over how the kernel schedules processes. Although the machine configuration is more fundamental to execution time, the fact that the machine may be running many other processes during runtime indroduces noise into our measurements. This is particularly relevant with shared machines. In recognition of this, all time measurements in this report are averaged over 20 runs and listed with the standard deviation. Data was also collected using a script in order to minimize the ammount of time between runs and ensure that programs were run in as standardized an enviroment as possible.

Code for this project may be found at https://github.com/rahwang/CarvingStool/Optimization.

2 mmult.c

Optimizations target an input size of 1024 (matrix width) and all compilation in this section is done with "gcc -msse4.1 program.c -lrt -lm".

2.1 Transposition

The original implementation has an obvious flaw, which has to do with data locality. In this original implementation, although we are accessing the elements of the first array sequentially, in the second array we are *striding*, accessing elements which are n elements apart. This does not allow us to take advantage of data locality since the accessed elements are not contiguous.

This issue can be resolved by transposing (effectively swapping content across the top-left to bottom-right diagonal) the second input matrix before performing any multiplication. While the transposition does add some overhead time because it involves n reads and writes, it is negligible compared to the benefits of being able to access both matrices in sequential order. As seen in Table 2, transposing the second matrix improves cache hit rate and overall performance significantly as the input size increases. For small inputs, the original implementation performs well because a the matrix is small enough to benefit from data locality despite its stride issue. However, for larger inputs, the cache rate hit is abysmal, whereas the transposed version has a better and consistent cache hit rate.

```c
// Transpose a matrix (stored as a flat array)
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    dst[j+i*n] = first[i+j*n];
  }
}
```

Table 1: A comparison of the program with and without transpositon:

Input	Original miss rate	Transposed miss rate	Original time	Transposed time
128	2.3%	12.0%	0.0091 ± 0.0029	0.0095 ± 0.0029
256	2.1%	12.4%	0.0499 ± 0.0043	0.0448 ± 0.0041
512	39.2%	12.4%	0.4409 ± 0.0035	0.3412 ± 0.0024
1024	39.3%	12.5%	7.0497 ± 0.0040	2.7049 ± 0.0015

2.2 Vectorization

After transposition, the program can also easily be vectorized, since the cells we want to operate on are now aligned. Our motivation for vectorization is simple: it allows us to perform 4 multiplication operations simultaneously and thus effectively trim down the number of overall operations. To vectorize, we can simply load two vectors, with four elements from each array. We keep the structure of the original program. The code for this is as follows:

```c
__m128i a_v, b_v;
__m128i sum0 = _mm_setzero_si128();

for ( i = 0 ; i < n ; i++ )  {
  for ( j = 0 ; j < n ; j++ ) {
    for ( k = 0 ; k < n ; k+= 4 ) {

      // Get a and b, multiply vectors, add to sum.
      sum0 = _mm_add_epi32(sum0, _mm_mullo_epi32(_mm_load_si128(a + i*n + k),
          _mm_load_si128(b + j*n + k)));
    }
    // Sum the sums.
    sum0 = _mm_hadd_epi32(sum0, sum0);
    sum0 = _mm_hadd_epi32(sum0, sum0);

    // Extract sum and store in res.
    res[i*n + j] = _mm_extract_epi32(sum0, 0);

    // Zero sum vector.
    sum0 = _mm_setzero_si128();
  }
}
```

Table 2: A comparison of the program with vectorization and with just transposition:

Input	Transposed time	Vectorized time
128	0.0095 ± 0.0029	0.0064 ± 0.0002
256	0.0448 ± 0.0041	0.0217 ± 0.0042
512	0.3412 ± 0.0024	0.1591 ± 0.0029
1024	2.7049 ± 0.0015	1.2439 ± 0.0030
2048	21.6534 ± 0.0133	10.0256 ± 0.0207

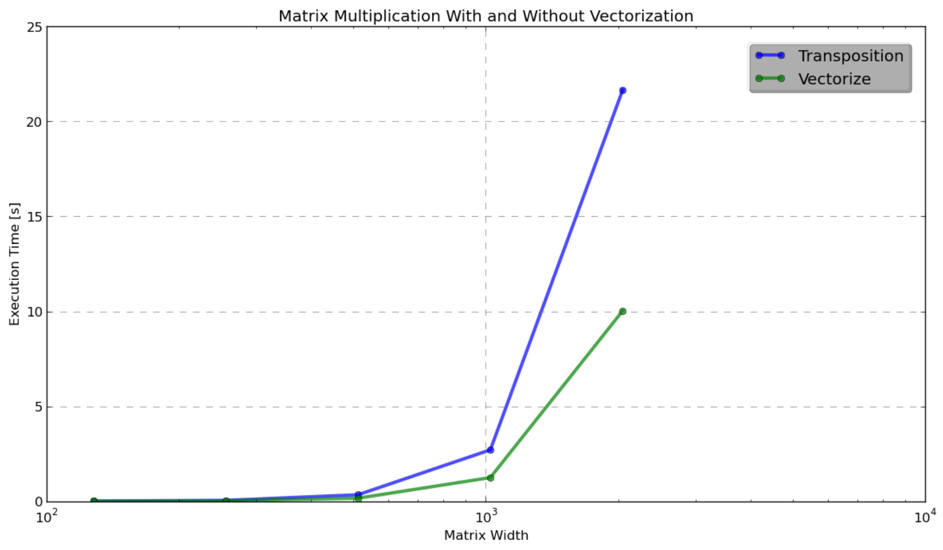

Figure 1: Vectorization offers a proportional improvement that becomes clearer on larger inputs

Vectorization does indeed show an improvement of over 2x for sufficiently large input matrices. It is not an improvement of 4x because multiplications are not the only relevant operations. Vector operations, both loading and extracting values from them, are expensive. However, as observed, the saved multipcations compensate for the added costs.

2.3 "Blocking"

Surprisingly, with the transpose method, classic blocking or tiling does *not* improve performance in our vectorized program, but instead worsens it. As block size decreases from n, performance worsens. Although blocking improves data locality and increases the cache hit rate, it adds tremendous overhead in the number of vector operations that are necessary, since we are increasing the number of loops. Expensive operations like extraction and the horizontal vector addition are actually performed *more* times in the blocked version, where they would ordinarily be done only once per j iteration.

Table 3: Vectorized code with blocking (n = 1024)

Block size	Time
1024	1.2439 ± 0.0029
512	2.9440 ± 0.0051
256	8.2555 ± 0.0146
128	26.7049 ± 0.3542
64	96.2188 ± 1.0061

Somewhat related to blocking, one optimization that did prove useful was the following, given operand matrices a and b

```
for ( i = 0 ; i < n ; i++ )  {
   for ( j = 0 ; j < n ; j+= 8 ) {
      for ( k = 0 ; k < n ; k+= 4 ) {
         // Get a and b, multiply vectors, add to sum.
        tmp = _mm_load_si128(a + i*n + k);
        sum0 = _mm_add_epi32(sum0, _mm_mullo_epi32(tmp, _mm_load_si128(b + j*n +
            k)));
        sum1 = _mm_add_epi32(sum1, _mm_mullo_epi32(tmp, _mm_load_si128(b + (j+1)*n
            + k)));
        sum2 = _mm_add_epi32(sum2, _mm_mullo_epi32(tmp, _mm_load_si128(b + (j+2)*n
            + k)));
        sum3 = _mm_add_epi32(sum3, _mm_mullo_epi32(tmp, _mm_load_si128(b + (j+3)*n
            + k)));
        sum4 = _mm_add_epi32(sum4, _mm_mullo_epi32(tmp, _mm_load_si128(b + (j+4)*n
            + k)));
        sum5 = _mm_add_epi32(sum5, _mm_mullo_epi32(tmp, _mm_load_si128(b + (j+5)*n
            + k)));
        sum6 = _mm_add_epi32(sum6, _mm_mullo_epi32(tmp, _mm_load_si128(b + (j+6)*n
            + k)));
        sum7 = _mm_add_epi32(sum7, _mm_mullo_epi32(tmp, _mm_load_si128(b + (j+7)*n
            + k)));
                                    .
                                    .
                                    .
```

In the above code, we unroll the j loop so that 8 sums are calculated in a single iteration. Crucially, one of the addition operands, what would be a[i*n + k], is saved in a temporary variable, so that the number of memory reads is reduced. Although this alteration seems trivial, it creates a significant and surprising speedup.

Admittedly, this optimization continues to be somewhat mysterious. Reducing the number of memory reads is helpful, but doesn't seem enough to account for the 25% improvement in performance. One likely explanation is pipelining. Because this alteration readies one of the values (tmp) to be used in multiple, similar operations, it is more obvious to the compiler that those consequtive operations are independent and pipelining is viable.

Table 4: Vectorized code with ”blocking” and tmp variable:

Input	Time
2048	8.1469 ± 0.1049
1024	$.8895 \pm 0.0027$
512	0.1072 ± 0.0027
256	0.0157 ± 0.0030
128	0.0042 ± 0.00001

2.4 Results

Table 5: Comparision of final optimized mmult with various compiler options:

Options	Time
-O0	.8895 $\pm$ 0.0027
-O1	0.1431 $\pm$ 0.0053
-O2	0.1322 $\pm$ 0.0019
-O3	0.1370 $\pm$ 0.0021

Clang and g++ did not perform noticibly differently. It seems that with gcc, any optimization level past -O0 offers a fairly similar improvement, although -O2 offers a very slight advantage. Since the largest improvement is from -O0 to -O1, it's likely the most basic compiler opimizations that offer the most benefit. A more in depth project would examine the effects of these compilations flags one by one. As for -O2, it's likely that the extra increase there comes from the added inlining and data aligning since looping is so central to matrix multiplication.

Overall, for $n = 1024$, compiled with -O2 set, the original mmult.c runs in 7.0482 $\pm$ 0.0012. This means that our final optimized version runs in $\frac{0.1322}{7.0482} \approx 1.875\%$ the time of the original. This is a speedup of more than 50 times!

3 bsort.c

Optimizations in this section will target an input size of $n = 65536$.

Table 6: bsort.c initial runtime:

Input	bubblesort
1024	0.0035 $\pm$ 0.0001
16384	0.3709 $\pm$ 0.0032
32768	1.4585 $\pm$ 0.0062
65536	1.4585 $\pm$ 0.0062

3.1 Vectorization

An easy optimzation here is simply to change the algorithm. Mergesort outperforms bubblesort simply because there are fewer comparisons made in the algorithm.

However, vectorization can offer some savings because, once loaded into vectors, four pairs of numbers may be compared at once. In order to vectorize bubblesort, groups of four elements in the array are loaded into a list of vectors,

```c
// Fill vector list with chunks of 4
int n = SIZE;
__m128i *vlist = (__m128i *)malloc(sizeof(__m128i)*n/4);
for (i = 0; i < n/4; i++) {
    vlist[i] = _mm_load_si128(array+i*4);
}
```

Bubblesort can now be performed using the loaded vectors and the SSE functions *_mm_max_epi*32() and *_mm_min_epi*32(). Given vectors A and B (in that order) in a list of vectors, A is removed from the list and replaced with the vector constructed using *_mm_min_epi*32() and B is replaced with the vector from *_mm_max_epi*32(). Now, at within each index 0-3 of the two vectors, the elements are ordered, meaning $A[0] < B[0]$, $A[1] < B[1]$ and so on. After bubblesort has been performed on the vector list in that fashion, the resulting vector list is sorted within each index. The elements at each of the four indices can then be extracted into four sorted lists. Those four lists can be merged into a final sorted result list. We are effectively bubblesorting each fourth of the list and merging the results.

However, using this method requires that we extract elements from the list by index (rather than just copying the contents of entire vectors at a time to a list), which is inefficient. To maximize cache usage in extracting those elements we can take advantage of the bubblesort algorithm. After a single complete pass over a list, bubblesort will never touch the new last element in the list again, since it has ensured that it is the maximum of the entire list. Similarly, after a second pass, the penultimate element will never be touched again. Therefore, we can extract the elements of the vector we have just put into its final place at the end of each complete pass. This offers significantly better data temporality than doing all the extraction after the bubblesort, because the vector we are extracting from has just been accessed, and so will already be in the cache. The code is as follows:

```c
__m128i tmp;

// Create four arrays to store the sorted elements extracted from vectors
int *l0 = (int *)malloc(sizeof(int)*n/4);
int *l1 = (int *)malloc(sizeof(int)*n/4);
int *l2 = (int *)malloc(sizeof(int)*n/4);
int *l3 = (int *)malloc(sizeof(int)*n/4);

// Bubblesort the vectors
for (c = 0 ; c < ( n/4 - 1 ); c++)  {
  for (d = 0 ; d < n/4 - c - 1; d++) {
    // Swap a pair of vectors out for their min and max, in that order
    tmp =  _mm_max_epi32(vlist[d], vlist[d+1]);
    vlist[d]   = _mm_min_epi32(vlist[d], vlist[d+1]);
    vlist[d+1] = tmp;
  }
  // Extract elements from the vector that has just been finalized
  l0[d] = _mm_extract_epi32(vlist[d], 0);
  l1[d] = _mm_extract_epi32(vlist[d], 1);
  l2[d] = _mm_extract_epi32(vlist[d], 2);
  l3[d] = _mm_extract_epi32(vlist[d], 3);
}
l0[0] = _mm_extract_epi32(vlist[0], 0);
l1[0] = _mm_extract_epi32(vlist[0], 1);
l2[0] = _mm_extract_epi32(vlist[0], 2);
l3[0] = _mm_extract_epi32(vlist[0], 3);

// Merge the resulting lists
res = m(m(l0, l1, n/2), m(l2, l3, n/2), n);
```

Although this is a definite improvement on bubblesort – the overhead of building the vector list is less than the improvements from the four simultaneous comparisons – this is still slower than mergesort, largely because bubblesort is still an inferior algorithm.

3.2 Using Mergesort

To use the power of mergesort *and* leverage off the vectorized comparisons, we bubblesort more, shorter lists and perform more merges. Let *div* be the number of vectors we bubble sort. Above, $div = 1$ so we get 4 sorted lists, and perform 3 merges to get the final res. When $div = 2$, we get 8 sorted lists and perform 7 merges. Generally, when $div = n$, we get $div \cdot 4$ lists and perform $2^n - 1$ merges.

```c
// len is the length of each vector list to be bubble sorted
int len = n/(4*divs);
// vecs is the list of what will eventually be sorted lists of length len
int **vecs = (int **)malloc(sizeof(int *)*divs*4);
for (i = 0; i < divs*4; i++) {
   vecs[i] = (int *)malloc(sizeof(int)*len);
}

__m128i tmp;
// Bubble sort
for (c = 0 ; c < ( len - 1 ); c++)  {
  for (d = 0 ; d < len - c - 1; d++) {
    // Do for each div
    for (i = 0; i < divs; i++) {
       idx = (len*i) + d;
       tmp = _mm_max_epi32(vlist[idx], vlist[idx+1]);
       vlist[idx]   = _mm_min_epi32(vlist[idx], vlist[idx+1]);
       vlist[idx+1] = tmp;
     }
   }
  // Extract vector elements into lists in vecs
  for (i = 0; i < divs; i++) {
     idx = (len*i) + d;
     vecs[i*4][d] = _mm_extract_epi32(vlist[idx], 0);
     vecs[i*4 + 1][d] = _mm_extract_epi32(vlist[idx], 1);
     vecs[i*4 + 2][d] = _mm_extract_epi32(vlist[idx], 2);
     vecs[i*4 + 3][d] = _mm_extract_epi32(vlist[idx], 3);
  }
}
for (i = 0; i < divs; i++) {
  vecs[i*4][0] = _mm_extract_epi32(vlist[i*len], 0);
  vecs[i*4 + 1][0] = _mm_extract_epi32(vlist[i*len], 1);
  vecs[i*4 + 2][0] = _mm_extract_epi32(vlist[i*len], 2);
  vecs[i*4 + 3][0] = _mm_extract_epi32(vlist[i*len], 3);
}

// Merge all the lists in vecs
for (i = log2(divs*4), c=2; i > 0; i--, c *= 2) {
   for (j = 0; j < divs*4; j += c) {
     vecs[j] = m(vecs[j], vecs[j+c/2], len*c);
   }
}

res = vecs[0];
```

The performance of vectorized mergesort looks like this for different values of div.

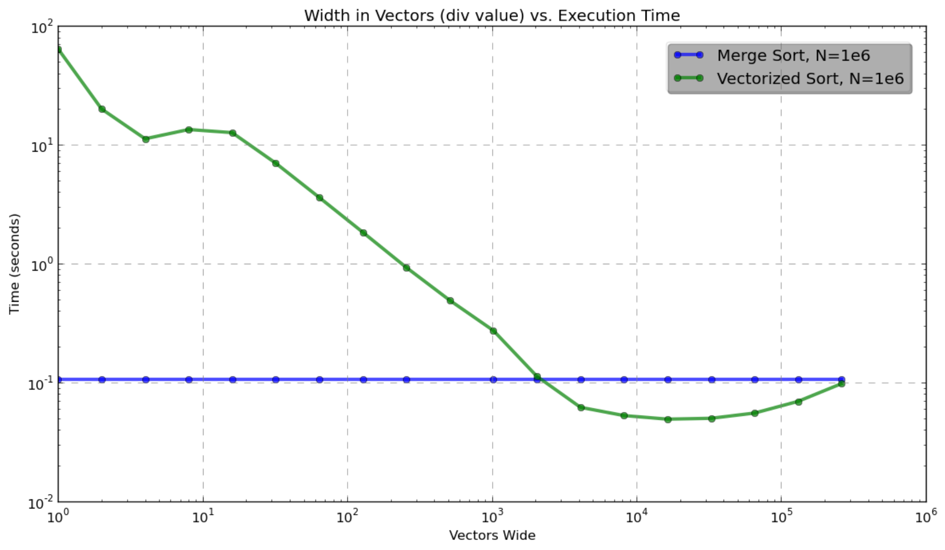

Figure 2: Overall, increasing div improves time, but this figure shows that other factors in this optimization also effect performance.

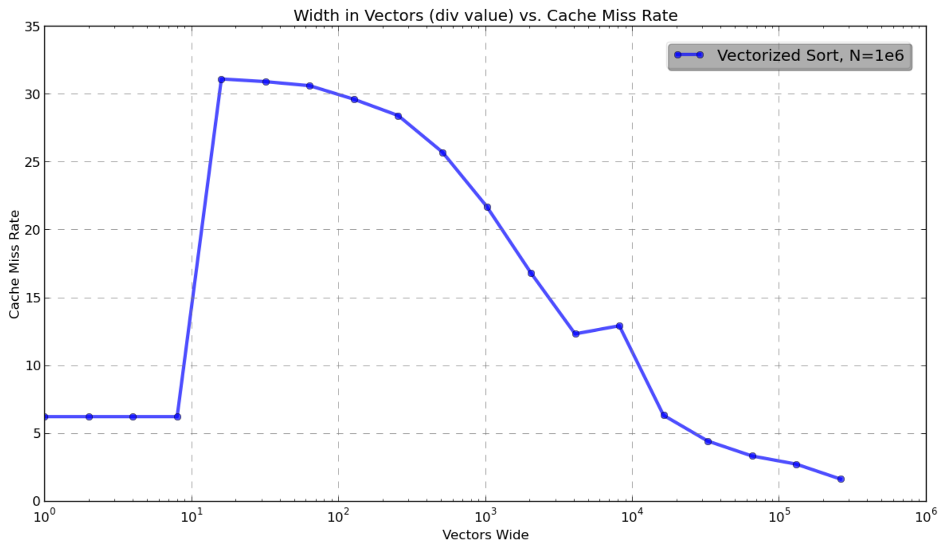

Figure 3: Miss rate behavior illustrates cache size.

Table 7: Runtimes with different amounts of merging:

div	Runtime	cache miss rate
$2^0 = 1$	64.3871 ± 0.4678	6.2%
$2^1 = 2$	20.0480 ± 0.3316	6.2%
$2^2 = 4$	11.2404 ± 0.1265	6.2%
$2^3 = 8$	13.4572 ± 0.0982	6.2%
$2^4 = 16$	12.6701 ± 0.0193	31.1%
$2^5 = 32$	7.0102 ± 0.0113	30.9%
$2^6 = 64$	3.6064 ± 0.0515	30.6%
$2^7 = 128$	1.8375 ± 0.0172	29.6%
$2^8 = 256$	0.9283 ± 0.0056	28.4%
$2^9 = 512$	0.4902 ± 0.0087	25.7%
$2^{10} = 1024$	0.2733 ± 0.0097	21.7%
$2^{11} = 2048$	0.1124 ± 0.0059	16.8%
$2^{12} = 4096$	0.0617 ± 0.0021	12.3%
$2^{13} = 8192$	0.0525 ± 0.0029	12.9%
$2^{14} = 16384$	0.0490 ± 0.0013	6.3%
$2^{15} = 32768$	0.0499 ± 0.0020	4.4%
$2^{16} = 65536$	0.0552 ± 0.0009	3.3%
$2^{17} = 131072$	0.0692 ± 0.0019	2.7%
$2^{18} = 262144$	0.0978 ± 0.0017	1.6%

By these results, using $div = 2^{14}$ yields the best performance. More generally, the optimal div value is around $\frac{n}{64}$.

As illustrated by Figure 2 above, the behavior of this vectorized sort has an interesting relationship with the div value. This is because there are several competing factors which influence performance: the comparisons saved using more vectors and less bubblesort, the overhead cost of adding and extracting from more vectors, and the improved data locality from shortening the length of the lists to be merged. A complicated but fascinating issue!

First, data locality: in Figure 3, we observe that the cache miss rate is low for values of div up to 8, about 1 in 16 cache reads miss. This is because our cache is an 8-way associative cache. When $div \leq 8$, we can comfortably perform an iteration of the innermost loop (which performs a single vector comparsion for each of the vector bubblesort lists) and find our data from each vector list still in the cache. However, as soon as div exceeds 8, on any iteration over the vector lists when the program reaches the ninth vector list, the data from the first vector list will be evicted. This is because of how the vectors are all stored and accessed from in a single array.

When looking for a[i][j], the cache index portion of the memory address is determined by j, since if div = 16, the length of a[i] is a large power of 2. a[0][0] and a[1][0] therefore likely have the same index, meaning they map to the same set of 8 blocks in the cache. Similarly, a[2][0], a[3][0], etc also map to the same 8 blocks, so a[i][0] all fit in the cache, where i = 0,...7. However, once we look for a[8][0] and try to place it in the cache, it still maps to the same 8 blocks, but now it's full (8 way assoc). Therefore, we evict a[0][0], since it is the least recently used. When we are done with one iteration of the innermost loop, we require a[0][1]. This would have been in the cache if a[0][0] were not evicted. Hence, the miss rate jumps dramatically.

When div increases, say to 32 or 64, we still have these evictions, but since we are doing less bubblesorting and more merging, we miss slightly less often. This accounts for the decrease in miss rate as the div value increases.

Returning to Figure 2, we observe the relationship of miss rate to overall execution time. There is a sharp overall improvement in performance as div increases, with small areas with *positive* slope around $div = 8$, which we have discussed, and after $div = 2^{14}$. This is because after the bubblesort vector lists are shorter than that length, the improvements granted by having a higher percentage of merges do not compensate for the additional overhead associated with using the extra vectors. For instance, the $\_mm\_extract\_epi32()$ function has a long latency. Optimization is a balance of all these factors.

3.3 Loop Reordering for Data Locality

A this point, an easy optimization is simply reordering the loops, changing the way that we iterate over the bubblesort vectors lists such that a complete pass is finished for each vector list before moving on to the next. For high div values, thus short individual vector lists, a single bubblesort vector list may be short enough to fit entirely within the cache. In addition, due to the nature of bubblesort, the same elements, in this case vectors, are likely to be accessed many times within a short period of time. Performing entire bubblesort passes for each list at a time thus improves data temporality.

```
__m128i tmp;
for (i = 0; i < divs; i++) {
  for (c = 0 ; c < ( len - 1 ); c++)  {
    for (d = 0 ; d < len - c - 1; d++) {
      idx = (len*i) + d;
      tmp = _mm_max_epi32(vlist[idx], vlist[idx+1]);
      vlist[idx]   = _mm_min_epi32(vlist[idx], vlist[idx+1]);
      vlist[idx+1] = tmp;
    }
    idx = (len*i) + d;
    vecs[i*4][d] = _mm_extract_epi32(vlist[idx], 0);
    vecs[i*4 + 1][d] = _mm_extract_epi32(vlist[idx], 1);
    vecs[i*4 + 2][d] = _mm_extract_epi32(vlist[idx], 2);
    vecs[i*4 + 3][d] = _mm_extract_epi32(vlist[idx], 3);
  }
  vecs[i*4][0] = _mm_extract_epi32(vlist[i*len], 0);
  vecs[i*4 + 1][0] = _mm_extract_epi32(vlist[i*len], 1);
  vecs[i*4 + 2][0] = _mm_extract_epi32(vlist[i*len], 2);
  vecs[i*4 + 3][0] = _mm_extract_epi32(vlist[i*len], 3);
 }
```

Table 8: Improved locality runtimes with different amounts of merging (n = 2^{20}):

div	Runtime
2	16.2234 ± 0.0164
4	8.0491 ± 0.0081
8	3.9011 ± 0.0090
16	1.8262 ± 0.0099
32	0.8973 ± 0.0016
64	0.4537 ± 0.0019
128	0.2289 ± 0.0016
256	0.1263 ± 0.0009
1024	0.0526 ± 0.0009
2048	0.0422 ± 0.0024
4096	0.0377 ± 0.0010
8192	0.0372 ± 0.0024
16384	0.0382 ± 0.0011
32768	0.0431 ± 0.0024
65536	0.0508 ± 0.0022
131072	0.0662 ± 0.0021
262144	0.0731 ± 0.0024

Note that with this alteration, the optimal div value is 2^{13}. Generally speaking, optimal div value is still at about the sam ratio as before.

3.4 Results

Now that we've optimized how the code is written, there are still a few compiler optimizations that can improve performance,

Table 9: Comparision of final optimized mmult with various compiler options (n=65536):

Options	Time
-O0	.08875 ± 0.0273
-O1	0.0052 ± 0.0003
-O2	0.0051 ± 0.0002
-O3	0.0048 ± 0.0003

Once again, each level above -O0 improves performance dramatically as compared to -O0, but not significantlly with respect to each other. With -O3, the original bsort runs in 2.463s ± 0.0124. This means that the optimized version runs in $\frac{0.0048}{2.46} \approx 0.2\%$ of the original's time. That's a speedup of about 500 times!

For comparsion, just plain mergesort unvectorized runs in 0.0115s ± 0.0031 on a list of 65536 elements. So the final version of the optimized code, vectorized mergesort is about twice as fast!

Machine specific information:
(from /proc/cpuinfo)

```
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 58
model name  : Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
stepping    : 9
microcode   : 0x16
cpu MHz     : 1600.000
cache size  : 8192 KB
physical id : 0
siblings    : 8
core id     : 0
cpu cores   : 4
apicid      : 0
cpuid level : 13
bogomips    : 6784.41
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
```