Data Structure and Cache Behaviour of Sparse Polynomials

Yozo Hida
yozo@cs.berkeley.edu

May 28, 2002

Abstract

We investigate the effect of various data structures on cache behaviour of sparse polynomial multiplication and find vector-hashtable combination to be the best performer. We then investigate the effect of cache optimizations such as blocking, which improves performance by $40 - 50\%$ for both floating point and arbitrary precision coefficient polynomials.

1 Introduction

Many algorithms used in computer algebra systems involve manipulating sparse polynomials of several variables. Such algorithms include include Gröbner basis computations, resultant computations, and determinants. Since sparse polynomial manipulations are the building blocks of these algorithms, it makes sense to take a closer look at the performance characteristics of sparse polynomial multiplication.

There are many possible data structures that can be used to represent a sparse polynomial on a computer. We take a look at the performance of few of these: dense, vector, hashtable, and hierarchical semi-dense. In all cases, the standard $O(mn)$ multiplication algorithm is used, since sparsity generally makes it uneconomical to apply more sophisticated algorithms such as FFT. Once a particular data structure and algorithm has been selected, there are still numerous factors affecting the actual performance, including:

- Instruction count. Obviously more instructions means longer execution time.
- Instruction mix. Since modern pipelined processors execute more than one instruction at a time, limited resources means that some instructions cannot be executed at maximum speed.
- Branches. Conditional branches can cause processor pipeline to stall.
- Data alignment. Some processors execute faster if certain data is aligned on word or double-word boundary.
- Memory access and cache misses. Memory access to uncached data is slow compared to processor speeds.

Of these, the first three (instruction count, instruction mix, and branches) are difficult to control since programs using sparse data structures do not have the rigid control regimes using in, for example, oblivious linear algebra code. There is inevitably a considerable percentage of bookkeeping

code, and indeed unless the coefficient arithmetic is over some elaborate domain (like arbitrary precision integers), the bookkeeping can dominate the actual arithmetic. The fourth item, aligning data, can be achieved by padding the data structure with dummy bytes, if necessary. This increases the memory requirement, but often results in faster code (note, however, increased memory requirement can imply more cache misses). In this report, we concentrate on the last item: controlling cache misses. Modern processors have cycle times less than 1ns, but memory speed has not changed to keep pace and is still at 100ns or so. This means that 100 instructions can be executed during one cache miss, and reducing cache misses can therefore affect the execution speed significantly.

Of the five data structures we looked at, the hashtable implementation (with a vector actually storing the data) came out to be the best performer. We then took this base algorithm and attempted to reduce the cache misses further by applying a technique called blocking. This enabled speedup of $40 - 50\%$ for both floating-point coefficients and for arbitrary precision coefficients.

2 Cache Optimization Techniques

Sparse polynomial multiplications can be done in $O(mn)$ time, where m and n are the sizes of the input polynomials (number of nonzero terms), and where we disregard, for the moment, any variation in the cost of coefficient operations. The multiplication output will consist of at most mn terms, but usually many fewer since terms with matching exponents will be added together. If we assume that the polynomials are stored in slow memory, but they can only be operated on in fast memory, traditional algorithms will read the terms into fast memory mn times, and writes out mn times. However, the input only consists of $m + n$ terms, so there is some room for improvement. If we can get away with substantially fewer reads than mn, then the running time can be reduced by almost a half (although we still may need to write mn terms out), giving some hope to cache optimization.

In addition to blocking, we also experimented with other techniques to reduce cache misses including reducing indirection and sequential accessing. Reducing indirection leads to fewer memory access instructions, and hence can lead to fewer cache misses. This is achieved, for example, by using an index computation to locate data instead of following a pointer. (If the pointer is to a location in cache, this is not a problem.) Sequential memory access is generally desirable because cache hardware brings in memory in chunks, so that if one requests memory at location 0, then data at locations 0 through 32 are brought in as well.

3 Data Structures

In this section, we describe several possible data structures that can be used to represent sparse polynomials.

3.1 Dense Data Structures

A sparse polynomial can be represented in the same way as a dense polynomial, where a slot for every possible coefficient is preallocated. This results in a very regular structure which is ideal in two ways: cache-friendliness and fast algorithms. Regular, contiguous data means that blocking and sequential access becomes natural. The regularity also enables fast algorithms such as FFT

(for fast cache optimized implementation, see [2]). Finally, coefficient lookup can be done on $O(1)$ time.

The major problem is that a dense data structure is very wasteful of space if it is used to store sparse polynomials For a polynomial with d variables and degree n in each variable it takes $O((n+1)^d)$ space, which is unacceptable except for small degree polynomials. One of the polynomial we computed, $p_{100} = (x + y + z)^{100}$, would require $101^3 \approx 1.3 \times 10^6$ slots for coefficients, which leads to 8 megabytes if double precision is used. In reality, there are only 5151 terms in the above polynomial, so a sparse data structure would only require 60 kilobytes or so (assuming all of the exponents of x, y, and z of a term can together be stored in a 4-byte word). Another disadvantage of dense representation is that if each zero entry is treated as any other entry, it requires considerable pointless computation, multipliying 0 by 0 and adding 0+0.

3.2 Semi-Dense Hierarchical Data Structure

Hierarchical data structures represent a polynomial recursively. For example, a polynomial in three variables x, y, and z is first considered as a polynomial in x, whose coefficients are polynomial in y and z. The recursive structure of these data structures lends to recursive algorithms, which are often cache oblivious [1].

In this data structure, a dense vector is used to represent coefficients at each level. A zero coefficient is explicitly represented. This leads to some waste of space, but much better than dense data structure. For example, the polynomial $p_{100} = (x + y + z)^{100}$, with 5151 terms, requires 1.4 megabytes. This is still quite large compared to 60 kilobytes for sparse representation, but still manageable.

3.3 Flat Data Structures

Flat data structures represent polynomials as a collection of monomials arranged in some way. For each monomial, the exponents and the coefficients are stored. For sparse polynomials, this is the most space efficient data structure since the therms with zero coefficients are omitted.

3.3.1 Vectors

Flat vector data structure stores monomials in a vector. After multiplication, sorting is required to collapse monomials with same exponents.

3.3.2 Hashtables

Flat hashtable data structures stores monomials in a hashtable keyed by the exponents. This results in average $O(1)$ coefficient lookup time, but requires more space since hashtables usually are not full. There are two possibilities when implementing hashtables regarding collisions and storage of data:

- Chaining. In this method, the hashtable is a vector of pointers, and an entry points to the data (exponent / coefficient pair). When there are collisions, the entry points to a linked list or an array of data.

- Open Addressing. In this method, the data is stored directly into the table. When a collision is detected, another slot (one that is empty) is chosen, and the data is placed there.

In our implementation, the latter method (with double hashing for collisions) is chosen primarily because there is less indirection (so that less memory access is needed). However, this means that more memory may be needed, since space for all possible data is preallocated. We also tried to keep the load factor below 0.5 so that collisions were rare. A more flexible hashtable structure with automatic provision for growth as it fills up (done in Common Lisp hashtables) reduces the complexity of the programmer's task at the cost of ruining some of the cache-friendly behavior of small programs.

3.3.3 Hashtable and Vector

If the data in the hashtable is non-trivial, we can consider an alternative which is to combines the vector and hashtable ideas to give $O(1)$ lookup time but with fast sequential access to all the monomials. A vector stores the exponent and coefficient pairs of each monomial, while the hashtable stores location into the vector keyed by the exponents.

One alteration in this proposal is to supply the input polynomials as flat vectors of coefficient / exponent pairs. Perhaps, but not necessarily sorted by exponent. The result can be computed in a hashtable and in a final step copied into a vector[1].

4 Cache Behaviour

Cache behaviour was monitored through three parameters: execution time, L1 data cache misses, and L2 total cache misses. All runs were done on an 800 MHz Intel Pentium III (Coppermine) processor running Linux 2.4.18. This processor has 16 KB L1 data cache, 16 KB L1 instruction cache, 256 KB unified L2 cache. The cache line is 32 bytes for both caches. L1 cache is 4-way associative and L2 is 8-way associative. Cache events were monitored with the PAPI library [6].

Figure 1 shows the execution time for sparse polynomial multiplication for polynomials of varying sizes. Each polynomial was of the form $p_k = (x + y + z)^k$, and was multiplied by itself: $p_{2k} = p_k \times p_k$. (The fact that the input was stored only once probably favored cache activities slightly).

The number of L1 cache misses is shown in figure 2. It is interesting to note that number of L1 and L2 cache misses is not proportional to the execution time. For vector data structure, the sorting step swaps seemingly unrelated elements, and hence results in a high L2 miss rate. The hierarchical data structure follows the cache-oblivious pattern with low L2 misses. However, it has high L1 misses. We suspect this is because the L1 cache is too small to handle even the last recursion level.

¿From these graphs, the preferred data structure seems to be a vector of monomials with hashtable indexed by the exponent. The next section details the cache optimization applied to this data structure and algorithm.

[1]Brief experiments suggested this copying would add only about 5% to the cost, while reducing the "permanent" storage of the hash structure.

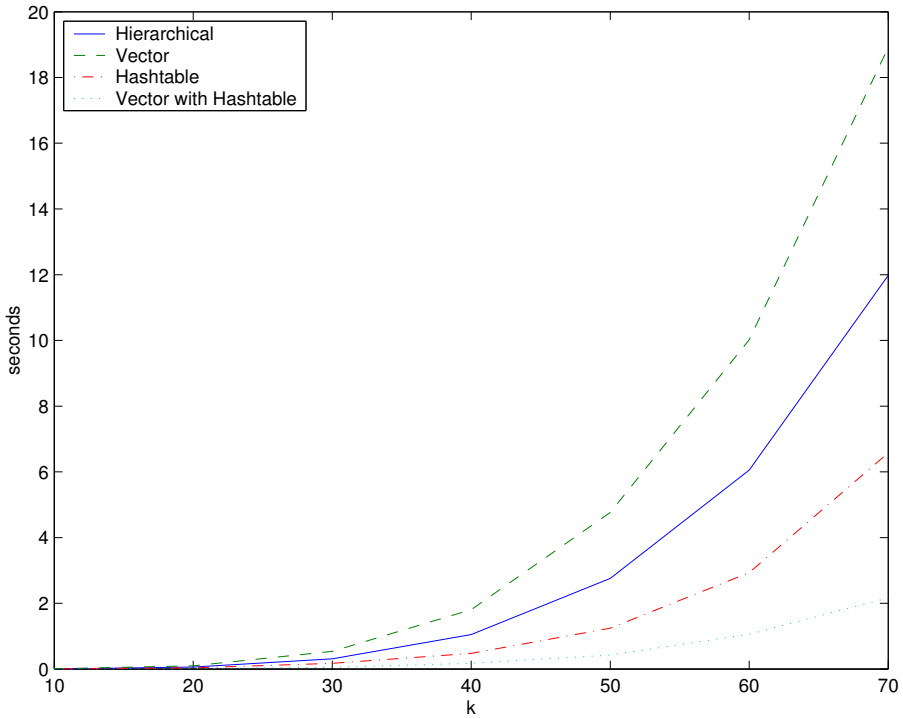

Figure 1: Execution time for sparse polynomial multiplication for various data structures.

5 Cache Optimizations

We now consider the cache optimization that can be done to improve the cache behavior of vector-hashtable data structure. Since the data is stored outside the hashtable (the hashtable only contains pointers to the data), the data is stored contiguously in memory. This means that in order to get to the list of monomials, there is no need to go through the hashtable (with its empty slots).

We applied the technique of blocking to further reduce cache misses. Suppose we are multiplying two polynomials p and q. For each term of p, the original algorithm traverses entire list of terms in q. This will almost guarantee a cache miss for every term computed. Instead, we modify the algorithm to select r terms from p and s terms from q, and compute all possible rs terms first. Then we keep the r terms from p, but move on to next s terms of q and continue. In this way, for every $r+s$ term loaded into memory, we compute rs terms of output. So theoretically, we reduced the read portion of the algorithm by $(r+s)/rs$. We want to choose r and s as large as possible so that $r+s$ terms can fit in cache, along with any other necessary data.

If the output actually touches $O(rs)$ terms, say krs, then our goal is to maximize the number of terms computed, rs, under the constraint $r+s+krs \leq L$ (where L is the number of terms the cache can hold) This gives the solution $r = s = (\sqrt{1+kL}-1)/k$ which is approximately $\sqrt{L/k}$ if kL is substantially larger than 1.

In more detail, blocking will take a doubly nested inner loop and change it into a quadruply nested loop. It must also handle cases where the size of the input is not divisible by the block size. These issues will complicate the code, and adds slightly to instruction cache misses.

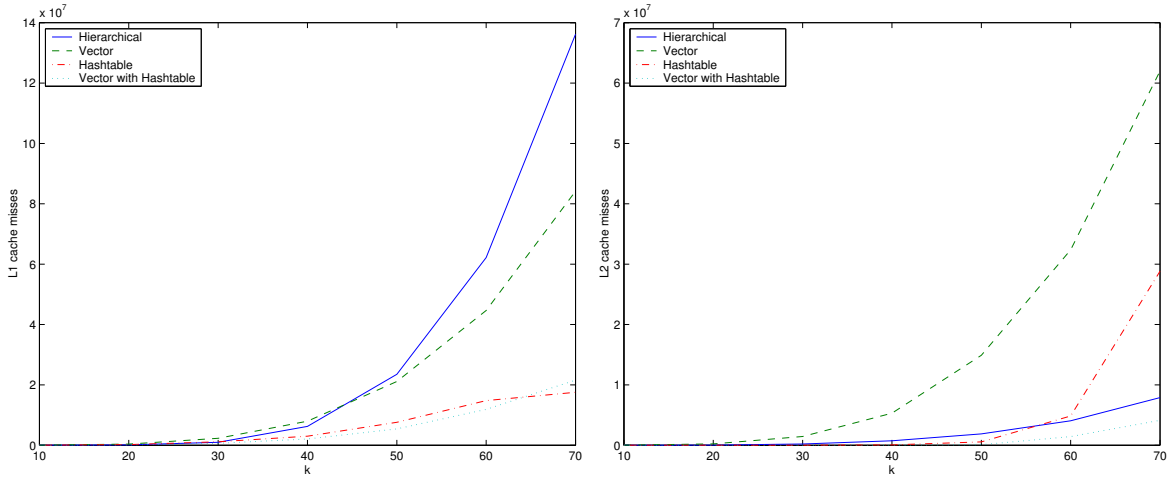

Figure 2: L1 and L2 cache misses for sparse polynomial multiplication for various data structures.

In order to simplify the test program, we assumed maximum exponent for each of the three variables of 1024, so that an exponent can be represented in 10 bits. This way, exponents of three variable polynomial can be represented in 32 bits. The following pseudocode describes the modified inner loop;

```
m = nr_terms(P) / r;
n = nr_terms(Q) / s;

for(ii = 0; ii < m; ii++) {
  for (jj = 0; jj < n; jj++) {

    for (i = 0; i < r; i++) {
      for (j = 0; j < s; j++) {
        t1 = P[ii*r+i];
        t2 = Q[jj*s+j];
        output(t1 * t2);
      }}}}
```

Although the formula suggest an optimimum for $r = s$, in practice, different $r > s$ seems to provide better performance. This may have to do with various pointer accesses. Cache behaviour is quite hard to explain in some instances and justifies the use of automatic tuning packages to do exhaustive searches. Figure 3 shows the color plot of the execution time for various blocks sizes $r \times s$, for the multiplication $p_{140} = p_{70} \times p_{70}$. Note the values on x and y axis are not the actual block sizes. The best performer is when $r = 352$ and $s = 28$. The multiplication was done in 0.6855 seconds, an improvement of about 40% from the unblocked code. This means that the cache (L2 in this case) can hold $rs = 9856$ terms (about 160 KB) plus the output, which is consistent with the fact that the L2 cache is 256 KB. The exact block size for optimal performance varies somewhat with input polynomials, but for the most part lies in the same region. Figure 4 shows the color plot of the megaflops rate. This computation $p_{140} = p_{70} \times p_{70}$ involves approximately 1.3×10^7 floating

point operations. The maximum speed achieved is 19 megaflops, which is only a small fraction of maximum speed of 800 megaflops, but this is expected of sparse data structures involving quite a few memory operations.

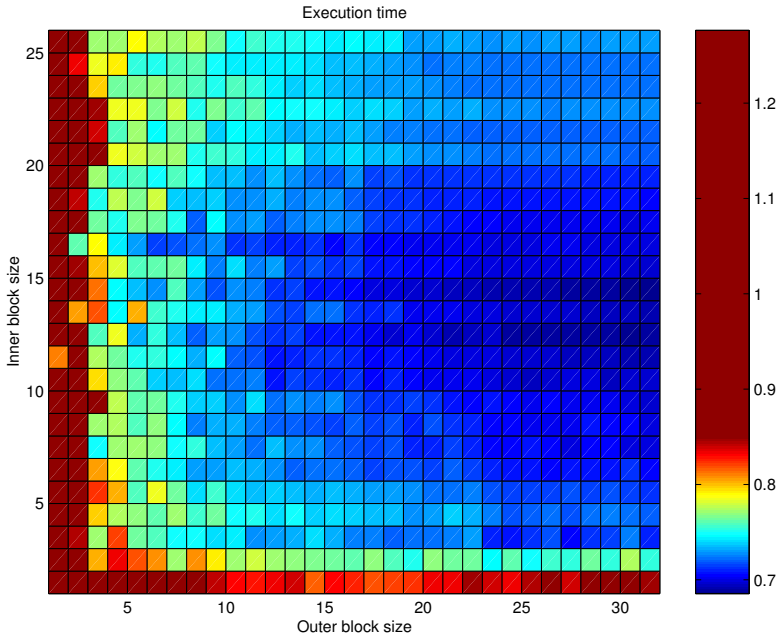

Figure 3: Execution time for various block sizes. Note the values on the x and y axis are not the actual block sizes. The x and y axis varies as follows: 1, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 36, 40, 48, 64, 72, 96, 128, 144, 160, 192, 224, 240, 256, 272, 288, 304, 320, 352, and 384.

Color plot for L1 and L2 misses are shown in figure 5. L1 cache misses is minimized at block size 36×10, while L2 cache misses is minimized at block size 384×72. It seems L2 cache miss contribute more to the execution time, since the overall pattern of L2 cache miss plot matches that of the execution time plot.

Finally, we consider the same code but using arbitrary precision arithmetic. We used the GMP (GNU multiprecision) package [3]. Figure 6 shows the color plot of the execution time. The best execution time occurs with block size 320×36, similar to the case with double precision coefficients. The multiplication $p_{140} = p_{70} \times p_{70}$ was done in 3.72 seconds, an improvement of about 47% over unblocked code. The greater speedup compared to the floating-point case results from the fact that arbitrary precision numbers requires more memory, and hence results in more opportunities for cache optimizations.

6 Conclusion and Future Work

We have seen that data structure and algorithm plays a fundamental role in the performance of sparse polynomial multiplication. Cache awareness is essential to the choice and design of the data structure. Once a data structure is chosen, cache optimization techniques such as blocking can be used to improve performance by as much as 47%.

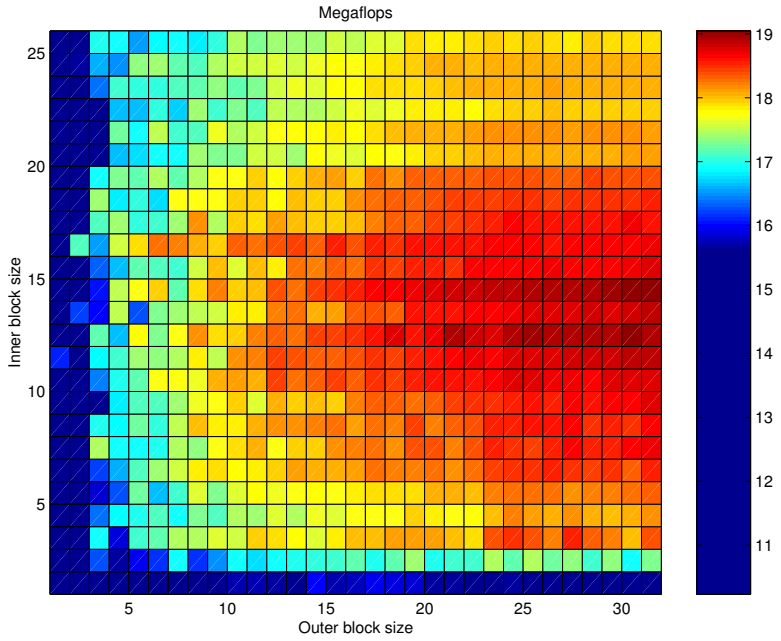

Figure 4: Megaflops rate for various block sizes. Note the values of x and y axis are not the actual block sizes. The x and y axis varies as follows: 1, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 36, 40, 48, 64, 72, 96, 128, 144, 160, 192, 224, 240, 256, 272, 288, 304, 320, 352, and 384.

Automatic optimization techniques used in PHiPAC [4] and ATLAS [5] can be applied to sparse polynomial multiplication. Techniques from sparse matrix-vector multiplication such as statistical sampling to determine optimal block size may prove useful as well.

7 Acknowledgments

This paper is based on a class project for CS282 Algebraic Algorithms, Spring, 2002, taught at UC Berkeley, Prof. Richard Fateman. This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley.

References

[1] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of IEEE Foundations of Computer Science*, 1999.

[2] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pp. 1381–1384, Seattle, WA, May 1998.

[3] GMP. `http://swox.com/gmp/`.

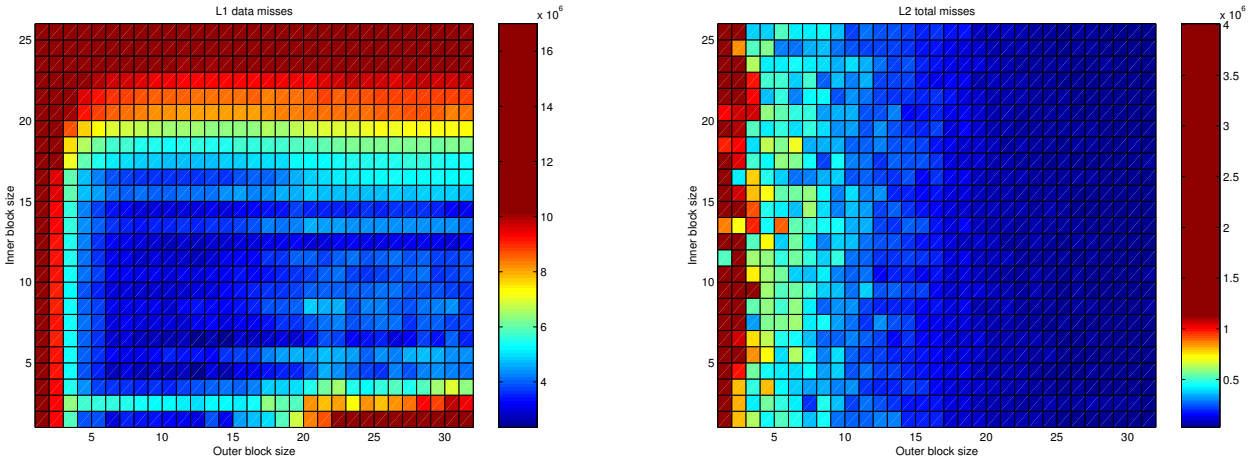

Figure 5: L1 and L2 cache misses for $p_{70} \times p_{70}$ for various block sizes. Note the values of x and y axis are not the actual block sizes. The x and y axis varies as follows: 1, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 36, 40, 48, 64, 72, 96, 128, 144, 160, 192, 224, 240, 256, 272, 288, 304, 320, 352, and 384.

[4] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology, In *Proceedings of International Conference on Supercomputing*, pp. 340-347, 1997.

[5] ATLAS. `http://math-atlas.sourceforge.net/`.

[6] PAPI. `http://icl.cs.utk.edu/projects/papi/`.

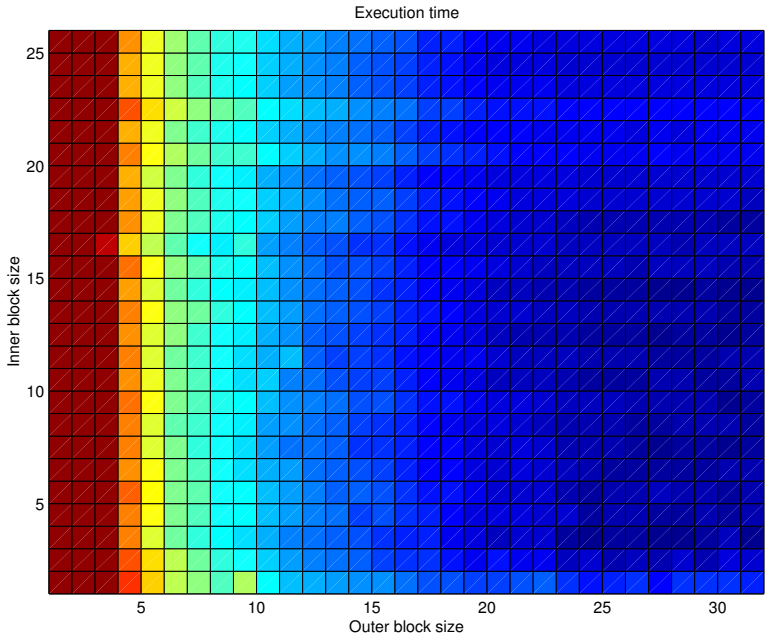

Figure 6: Execution time for $p_{70} \times p_{70}$ using various block sizes, with bignum coefficients. Note the values of x and y axis are not the actual block sizes. The x and y axis varies as follows: 1, 2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 36, 40, 48, 64, 72, 96, 128, 144, 160, 192, 224, 240, 256, 272, 288, 304, 320, 352, and 384.