An overview of the recent progress on matrix multiplication

Virginia Vassilevska Williams

Abstract

The exponent ω of matrix multiplication is the infimum over all real numbers c such that for all $\varepsilon > 0$ there is an algorithm that multiplies $n \times n$ matrices using at most $O(n^{c+\varepsilon})$ arithmetic operations over an arbitrary field. A trivial lower bound on ω is 2, and the best known upper bound until recently was $\omega < 2.376$ achieved by Coppersmith and Winograd in 1987. There were two improvements on ω, one by Stothers in 2010 who showed that $\omega < 2.374$, and one by myself that ultimately resulted in $\omega < 2.373$. Here I discuss the road to these improvements and conclude with some open questions.

1 Introduction

Matrix multiplication is one of the most basic mathematical operations outside of everyday arithmetic. Many other essential matrix operations can be efficiently reduced to it, such as Gaussian elimination, LUP decomposition, the determinant or the inverse of a matrix [1]. Matrix multiplication is also used as a subroutine in many computational problems that, on the face of it, have nothing to do with matrices. As a small sample illustrating the variety of applications, there are faster algorithms relying on matrix multiplication for all pairs shortest paths in a graph [10, 16], context free grammar parsing [14], and even learning juntas [8].

For the practical applications of matrix multiplication, there is a common goal: *a simple, fast algorithm which exploits the sparsity of the matrices, can be distributed and has great performance on modern hardware.*

To obtain this goal, one typically starts from a simple efficient algorithm and then attempts to add the rest of the restrictions: make it distributed with good cache performance, and any other side goal that is important for the application. The more insight one has into the workings of the algorithm, the easier this latter task gets, and hence simple algorithms are good. The question then is, what is the most efficient, hopefully simple, algorithm for multiplying typically sparse matrices?

It turns out that unless the input matrices have some very special structure, in general even the sparsest matrices have dense components, and in the end the question really becomes:

What is the best algorithm for multiplying arbitrary (possibly dense) matrices?

This leads us to one of the central questions in the theory of algorithms: determine the constant ω, called the exponent of matrix multiplication. This constant is defined as the infimum of all real numbers such that for all $\varepsilon > 0$ there is an algorithm for multiplying (arbitrary) $n \times n$ matrices running in time $O(n^{\omega+\varepsilon})$. Until the late 1960s it was believed that $\omega = 3$, i.e. that no improvement can be found for the problem. In 1969, Strassen surprised everyone by showing that two $n \times n$ matrices can be multiplied in $O(n^{2.81})$ time. This discovery spawned a twenty-year-long extremely productive time in which the upper bound on ω was gradually lowered to 2.376. After a twenty-year stall, some very recent research has brought the upper bound down to 2.373.

Even if the original $n \times n$ matrices are sparse, in the worst case their product can contain $\Omega(n^2)$ nonzero entries, and so $\Omega(n^2)$ is a natural lower bound for the product of $n \times n$ matrices.

Could this also be an upper bound? Is it theoretically possible that computing the product of two matrices does not require much more overhead than just writing the output?

The best known lower bounds for the problem are of the form $c \cdot n^2$ for explicit constants $c \leq 3$ [4, 7, 11]. Perhaps because of this, a majority of experts believe that there should exist an essentially quadratic time algorithm for matrix multiplication. Some prominent experts, however, such as Strassen, have also expressed the opinion that matrix multiplication requires strictly more than quadratic time. The answer has evaded computer science researchers for more than 40 years!

Determining the answer to the above question would not only be a very important milestone for theoretical computer science, but it would also be important for practical applications. Any such breakthrough in our knowledge of matrix multiplication algorithms would require huge insight into the problem, and such insight is what we need to start designing practical algorithms.

2 Some notions and their meaning

In the study of matrix multiplication algorithms, we often restrict ourselves to the arithmetic circuit complexity model, which is standard for computing polynomials over multiple variables. In this model, starting with the inputs (the matrix entries), the algorithm is allowed to multiply any already computed expression by a scalar, or add or multiply two already computed expressions, and each such operation is assumed to have unit cost. (One could also talk about adding divisions to the model, however Strassen [13] showed that divisions do not help asymptotically.) Then, ω is defined as the infimum of all real numbers h such that $O(n^h)$ arithmetic operations (over an arbitrary underlying field) suffice to compute the product of two $n \times n$ matrices.

Bilinear algorithms. It turns out that since matrix multiplication is a problem of computing a polynomial of degree two, we can further restrict our attention to so-called *bilinear* algorithms that have the following form.

Given two matrices A and B, compute r products

$$P_l = \left(\sum_{i,j} u_{ijl} A[i,j]\right) \left(\sum_{i,j} v_{ijl} B[i,j]\right),$$

i.e. take r linear combinations of entries of A and multiply each one with a linear combination of entries of B. Then, compute each entry of the product AB as a linear combination of the P_l:

$$AB[i,j] = \textstyle\sum_l w_{ijl} P_l \;\text{ for constants } w_{ijl}.$$

The minimum number of products r that a bilinear algorithm can use to compute the product of two $n \times n$ matrices is called the *rank* of $n \times n$ matrix multiplication, abbreviated as $R(\langle n,n,n \rangle)$. One can then also define $\omega = \inf_n\{h \mid R(\langle n,n,n\rangle) = O(n^h)\}$.

Any bilinear algorithm for $k \times k$ matrix multiplication for constant k can be used to multiply larger matrices as well by exploiting the inherent recursive nature of matrix multiplication: the product of two $kn \times kn$ matrices can be viewed as the product of two $k \times k$ matrices, the entries of which are $n \times n$ matrices.

Suppose that we have a bilinear algorithm ALG for multiplying two $k \times k$ matrices (for constant k) that computes r products P_l. Then one can create a recursive algorithm for multiplying $k^i \times k^i$ matrices (for any integer $i > 1$) as well: view the $k^i \times k^i$ matrices as $k \times k$ matrices the entries of which are $k^{i-1} \times k^{i-1}$ matrices; then multiply the $k \times k$ matrices using ALG and when ALG requires us to multiply two matrix entries, recurse. The point is, of course, that any operation in the arithmetic circuit model can also be viewed as an operation on matrices, and that by definition, bilinear algorithms do not exploit commutativity.

The recursive approach using an upper bound of r on $Rank(\langle k, k, k\rangle)$ gives a bound $\omega \leq \log_k r$. To see this, notice that the number of additions that one has to do in each recursive step is no more than $3rk^2$: at most $2k^2$ to compute the linear combinations for each P_l and at most r for each of the k^2 outputs $AB[i, j]$. Since matrix addition takes linear time in the matrix size, we have a recurrence of the form $T(k^i) = rT(k^{i-1}) + O(rk^{2i})$.

Note that as long as $r < k^3$ we get a nontrivial bound on ω. Strassen's famous algorithm used $k = 2$ and $r = 7$ thus showing that $\omega \leq \log_2 7 < 2.81$. A lot of work went into getting more and more "base algorithms" for varying constants k. This approach culminated in Pan's algorithm for multiplying 70×70 matrices that used $143,640$ products and hence showed that $\omega \leq \log_{70} 143,640 < 2.796$.

Approximate algorithms and Schönhage's theorem. A further step was to look at more general algorithms, so called *approximate* bilinear algorithms. In the definition of a bilinear algorithm the coefficients $u_{ijl}, v_{ijl}, w_{ijl}$ were constants. In an approximate algorithm, these coefficients can be formal linear combinations of the integer powers of an indeterminate, λ. The entries of the product AB are then only "approximately" computed, in the sense that $AB[i, j] = \sum_l w_{ijl}P_l + O(\lambda)$, where the $O(\lambda)$ term is a linear combination of *positive* powers of λ. The term "approximate" comes from the intuition that if you set λ to be close to 0, then the algorithm would get the product almost exactly. The minimum number of products r that an approximate bilinear algorithm can use to compute the product of two $n \times n$ matrices is called the *border rank* $\underline{R}(\langle n, n, n\rangle)$ of matrix multiplication.

Interestingly enough, Bini [2] showed that when dealing with the asymptotic complexity of matrix multiplication, approximate algorithms suffice for obtaining bounds on ω, i.e. bounds on the border rank of constant size matrices suffice. This is not obvious! What Bini showed, in a sense, is that as the size of the matrices grows, the "approximation" part can be replaced by a sort of bookkeeping which does not present an overhead asymptotically. The upshot is that if $\underline{R}(\langle k, k, k\rangle) \leq r$, then $\omega \leq \log_k r$.

Bini et al. [3] gave the first approximate bilinear algorithm for a matrix product. Their algorithm used 10 entry products to multiply a 2×3 matrix with a 3×3 matrix. Although this algorithm is for rectangular matrices, it can easily be converted into one for square matrices: a 12×12 matrix is a 2×3 matrix with entries that are 3×2 matrices with entries that are 2×2 matrices, and so multiplying 12×12 matrices can be done using Bini's algorithm three times, taking $1,000$ entry products. Hence $\omega \leq \log_{12} 1,000 < 2.78$.

Schönhage [9] developed a sophisticated theory involving the bilinear complexity of rectangular matrix multiplication that showed that approximate bilinear algorithms are even more powerful. His paper culminated in the so-called Schönhage τ-theorem, or the *asymptotic sum inequality*. This theorem is one of the most useful tools in designing and analyzing matrix multiplication algorithms.

Schönhage's τ-theorem says roughly the following. Suppose we have several instances of matrix multiplication, each involving matrices of possibly different dimensions, and we are somehow able to design an approximate bilinear algorithm that solves all instances and uses fewer products than would be needed when computing each instance separately. Then this bilinear algorithm can be used to multiply (larger) square matrices and would imply a nontrivial bound on ω. The exact statement is as follows: Suppose we have an upper bound of r on the border rank of computing p independent instances of matrix multiplication with dimensions $k_i \times m_i$ by $m_i \times n_i$ for $i = 1, \ldots, p$. Then $\omega \leq 3\tau$, where $\sum_i (k_i m_i n_i)^\tau = r$.

One very interesting point about Schönhage's theorem is that, when it comes to *exact* bilinear algorithms, it is believed that one cannot use fewer products to compute several instances than one would use by just computing each instance separately. This is known as Strassen's additivity conjecture. Schönhage showed that the additivity conjecture is *false* for approximate bilinear algorithms. In particular, he showed that one can approximately compute the product of a 3×1 by a 1×3 vector and the product of a 1×4 by a 4×1 vector together using only 10 entry products, whereas any exact bilinear algorithm provably needs at least $3 \cdot 3 + 4 = 13$ products. His theorem then implied $\omega < 2.55$, and this was a huge improvement over the previous bound of Bini et al.

Using fast solutions for problems that are not matrix multiplications. The next realization was that there is no immediate reason why the "base algorithm" that we use for our recursion has to compute a matrix product at all. Let us focus on the following family of computational problems. We are given two vectors x and y and we want to compute a third vector z. The dependence of z on x and y is given by a three-dimensional tensor t as follows: $z_k = \sum_{ij} t_{ijk} x_i y_j$. The vector z is a bilinear form. Notice that the tensor t completely determines the computational problem. Some examples of such bilinear problems are polynomial multiplication and of course matrix multiplication. For polynomial multiplication, $t_{ijk} = 1$ if and only if $j = k - i$, and for matrix multiplication, $t_{(i,i'),(j,j'),(k,k')} = 1$ if and only if $i' = j, j' = k$ and $k' = i$.

The theory of approximate bilinear algorithms naturally extends to the family of bilinear problems. A bilinear algorithm computing a problem instance for tensor t computes r products P_l of the form $P_l = (\sum_i u_{il} x_i)(\sum_j v_{jl} y_j)$ and then sets $z_k = \sum_k w_{kl} P_l$.

Here, an algorithm is nontrivial if the number of products r that it computes is less than the number of positions t_{ijk} where the tensor t is nonzero (also called the support of t). The notions of rank and border rank also immediately extend to arbitrary bilinear problems: $R(t)$ (respectively $\underline{R}(t)$) is the minimum number of products that a bilinear (respectively, approximate bilinear) algorithm can use to compute the bilinear problem defined by t.

In order to be able to talk about recursion for general bilinear problems, it is useful to define the *tensor product* $t \otimes t'$ of two tensors t and t':

$$(t \otimes t')_{(i,i'),(j,j'),(k,k')} = t_{ijk} \cdot t_{i'j'k'}.$$

Thus, the bilinear problem defined by $t \otimes t'$ can be viewed as a bilinear problem $z_k = \sum_{ij} t_{ijk} x_i y_j$ defined by t, where each product $x_i y_j$ is actually itself a bilinear problem $z_{ijk'} = \sum_{i'j'} t'_{i'j'k'} x_{ii'} y_{jj'}$ defined by t'.

This allows one to compute an instance of the problem defined by $t \otimes t'$ using an algorithm for t and an algorithm for t'. One can similarly define the kth tensor power of a tensor t as tensor-multiplying t by itself k times. Then any bilinear algorithm computing an instance defined by

t using r entry products can be used recursively to compute the kth tensor power of t using r^k products, just as in the case of matrix multiplication.

A crucial development in the study of matrix multiplication algorithms was the discovery that sometimes algorithms for bilinear problems that do not look at all like matrix products can be converted into matrix multiplication algorithms. This was first shown by Strassen in the development of his "laser method" and was later exploited in the work of Coppersmith and Winograd. The basic idea of the approach is as follows.

Consider a bilinear problem P for which you have a good approximate algorithm ALG that uses r entry products. Take the nth tensor power P^n of P (for large n), and use ALG recursively to compute P^n using r^n entry products. P^n is a bilinear problem that computes a long vector z from two long vectors x and y. Suppose that we can embed the product C of two $N \times N$ matrices A and B into P^n as follows: we put each entry of A into some position of x and set all other positions of x to 0, we similarly put each entry of B into some position of y and set all other positions of y to 0, and finally we argue that each entry of the product C is in some position of the computed vector z (all other z entries are 0). Then we would have a bilinear algorithm for computing the product of two $N \times N$ matrices using r^n entry products, and hence

$$\omega \leq \log_N r^n.$$

The goal is to make r as small as possible and N as large of a function of n as possible, thus minimizing the upper bound on ω.

Strassen's laser method and Coppersmith and Winograd's paper, and even Schönhage's τ-theorem, present ways of embedding a matrix product into a large tensor power of a different bilinear problem. In fact, all known improvements on the bound on ω since 1981 have this form!

Schönhage's theorem is often used as a shortcut, as instead of embedding a single matrix product, one can embed several independent instances, obtain a bound on their border rank and then conclude with the bound on ω that follows from the τ-theorem. We'll give a brief overview of the Coppersmith-Winograd algorithm [6].

The Coppersmith-Winograd algorithm. Coppersmith and Winograd start by studying the following interesting bilinear problem. Let q be a positive integer (typically, $q = 5$ or $q = 6$). We are given two vectors x and y of length $q + 2$ and we want to compute a vector z of length $q + 2$ defined as follows:

$$z_0 = \sum_{i=1}^{q}(x_i y_i) + x_0 y_{q+1} + x_{q+1} y_0,\ z_i = x_i y_0 + x_0 y_i \text{ for } i \in \{1, \ldots, q\} \text{ and } z_{q+1} = x_0 y_0.$$

Notice that z is far from being a matrix product. However, it is related to 6 matrix products:

1. $z'_0 = \sum_{i=1}^{q} x_i y_i$ which is the inner product of two q-length vectors,
2. $z''_0 = x_0 y_{q+1}$, $z'''_0 = x_{q+1} y_0$, and $z_{q+1} = x_0 y_0$, which are three scalar products, and
3. the two matrix products computing $z'_i = x_i y_0$ and $z''_i = x_0 y_i$ for $i \in \{1, \ldots, q\}$ which are both products of a vector with a scalar.

If we could somehow convert Coppersmith and Winograd's bilinear algorithm into one computing these 6 products as *independent* instances, then we would be able to use Schönhage's τ-theorem. Unfortunately, however, the 6 matrix products are merged in a strange way: $z_0 = z'_0 + z''_0 + z'''_0$ and $z_i = z'_i + z''_i$ for $i \in \{1, \ldots, q\}$. It is unclear how to get anything meaningful out of an algorithm that solves this merged bilinear problem.

Grouping the variables. Consider now grouping the indices of the x, y, and z variables so that all indices in $\{1, \dots, q\}$ are in group 1, the zero indices are in group 0 and the $q+1$ indices are in group 2. This grouping has the property that each of the 6 matrix products above only uses x variables that map to the same group, y variables that map to the same group, and z output variables that map to the same group. In fact, there is a *bijection* between the 6 matrix products and the triples of numbers from $\{0, 1, 2\}$ that sum to 2: For instance, $\sum_{i=1}^{q} x_i y_i$ is associated with $(1, 1, 0)$ since its x variables are all in group 1, its y variables are all in group 1 and it only contributes to z_0, whereas the vector-scalar product $z'_i = x_i y_0$ for $i \in \{1, \dots, q\}$ is associated with $(1, 0, 1)$.

Consider now the nth tensor power of the Coppersmith-Winograd bilinear form. Its variables have indices that are n-length sequences from $\{0, \dots, q+1\}$. The mapping from indices of the original construction to groups can be extended to a mapping of the indices of the nth tensor power to n-length *group sequences* just by mapping each n-length sequence index i to an n-length sequence of groups a componentwise: for every $t \in \{1, \dots, n\}$ we set $a[t]$ to be the group that $i[t]$ is mapped to in the original construction. For instance, the index $(q, q+1, 0)$ from the third tensor power is in group $(1, 2, 0)$.

Just as in the original construction the 6 matrix products were in a one-to-one correspondence with the triples (a, b, c) of integers in $\{0, 1, 2\}$ that sum to 2, for the nth tensor power, there is a bijection between 6^n matrix products and the triples (a, b, c) of n-length integer sequences in $\{0, 1, 2\}^n$ that sum to 2 in every component: If we restrict our attention to the subset of products $x_i y_j$ in the nth tensor power for which i is in group a and j is in group b, then these products only contribute to the computation of z_k where k is in group c with $a[t] + b[t] + c[t] = 2$ in every component t. Each of these products forms a matrix multiplication instance since it is just the tensor product over all $t \in \{1, \dots, n\}$ of those matrix products (from the original 6) associated with the triples $(a[t], b[t], c[t])$.

The approach that Coppersmith and Winograd take is to construct a large set S of triples of sequences (a, b, c) with $a, b, c \in \{0, 1, 2\}^n$ and $a[t] + b[t] + c[t] = 2$ for all t, such that for any two triples $(a, b, c), (a', b', c') \in S$ we must have $a \neq a', b \neq b'$ and $c \neq c'$.

This distinctness property means that the subproblems corresponding to (a, b, c) and (a', b', c') are on completely disjoint sets of variables. That is, the instances corresponding to the triples in S are completely independent instances of matrix multiplication, exactly what you need to apply the Schönhage τ-theorem.

Moreover, S is constructed so that whenever a triple (a, b, c) is not picked to be in S, then for all triples $(a', b', c') \in S$, either $a' \neq a$, or $b' \neq b$, or $c' \neq c$. This means that one can either zero out all x variables with indices in group a, or all y variables with indices in group b, or all z variables with indices in group c. After one does this for all triples $(a, b, c) \notin S$, what remains are exactly the independent bilinear forms corresponding to the triples in S. This zeroing out defines an embedding of many independent instances of matrix multiplication in the nth tensor power of the original bilinear problem, as we discussed earlier.

The special set S is constructed to also have the property that for every choice for $i, j, k \in \{0, 1, 2\}$, the number of components t for which $a[t] = i, b[t] = j, c[t] = k$ is exactly the same for all triples $(a, b, c) \in S$. That is, there are numbers a_{ijk} representing the *fraction of components t* in which every triple in S is exactly the group triple (i, j, k). This balanced choice for triples in S guarantees that, after the zeroing out, the matrix product instances remaining have exactly the same dimensions. This makes the analysis simpler, as one only needs to analyze the size of S in order to use the τ-theorem. (However, it is possible that the best embedding of matrix product instances is not balanced, so that this is one potential place to improve the analysis.)

The choice of S is actually parametrized by the variables a_{ijk}. For any choice for a_{ijk}, one defines X_i, Y_j, Z_k for $i, j, k \in \{0, 1, 2\}$ with $X_i = \sum_j a_{ij(2-i-j)}$, $Y_j = \sum_i a_{ij(2-i-j)}$, $Z_k = \sum_i a_{i(2-i-k)k}$. To construct S, one initially zeroes out all x variables that do not have exactly an X_i fraction of is in their index, all y variables that do not have exactly a Y_j fraction of js in their index and all z variables that do not have exactly a Z_k fraction of ks in their index. After this variable restriction, more variables are zeroed out to pick the independent group triples, as discussed earlier. The size of S is then represented as a function of n and the variables a_{ijk}. This bound on $|S|$ can be used in tandem with Schönhage's theorem to give a bound on ω, as n is taken to ∞, and this bound on ω now is a function of the variables a_{ijk}.

The variables a_{ijk} define a **search space for algorithms.**

The variables are constrained: They sum to 1 since they are fractions of the index length. They are also related by the linear system $\{X_i = \sum_j a_{ij(2-i-j)},\ Y_j = \sum_i a_{ij(2-i-j)},\ Z_k = \sum_i a_{i(2-i-k)k}\}$ described above. Also, there are constraints that serve to maximize the number of independent matrix product instances, i.e. the size of S as n grows. To find the best bound on ω then one formulates a constraint program describing the algorithm search space in terms of the variables a_{ijk} and the constraints that come from the analysis. Then one can use software to solve the constraint program, essentially finding the best matrix multiplication algorithm within the search space.

The analysis is suboptimal. Consider the second tensor power of the Coppersmith-Winograd bilinear form. Here the indices of the x, y, z variables are pairs of integers in $\{0, \ldots, q+1\}$. The products needed to compute an entry $z_{i,j}$ of this new bilinear form are obtained by taking a product $x_u y_v$ that contributes to entry z_i of the original form and a product $x_{u'} y_{v'}$ that contributes to z_j and concatenating their indices to obtain $x_{u,u'} y_{v,v'}$. For instance,

$$z_{0,0} = x_{0,0}y_{q+1,q+1} + x_{0,q+1}y_{q+1,0} + x_{q+1,0}y_{0,q+1} + x_{q+1,q+1}y_{0,0} +$$
$$+ \sum_{i=1}^{q}(x_{i,q+1}y_{i,0} + x_{q+1,i}y_{0,i} + x_{i,0}y_{i,q+1} + x_{0,i}y_{q+1,i}) + \sum_{i=1}^{q}\sum_{j=1}^{q} x_{i,j}y_{i,j}.$$

The original 6 matrix products that were merged to obtain the original bilinear form now give rise to $6^2 = 36$ matrix products, and just as the original matrix products were in one-to-one correspondence with the triples (a, b, c) with $a + b + c = 2$ and $a, b, c \in \{0, 1, 2\}$, the new matrix products are in one-to-one correspondence with the triples (a, b, c) of length-2 group sequences. These can just be represented as the 6-tuples $(a[1], a[2], b[1], b[2], c[1], c[2])$ with $a[i], b[i], c[i] \in \{0, 1, 2\}$ and $a[i] + b[i] + c[i] = 2$ for $i = 1, 2$.

One can carry out the same analysis to decouple tensor products of these matrix products in a large tensor power, and one would obtain exactly the same upper bound on the exponent ω. The brilliant insight that Coppersmith and Winograd had, however, is that there is a different way to group the entry products of the second tensor power that gives rise to matrix product instances of different dimensions. For instance, consider the following sum contributing to $z_{0,0}$

$$x_{0,q+1}y_{q+1,0} + x_{q+1,0}y_{0,q+1} + \sum_{i=1}^{q}\sum_{j=1}^{q} x_{i,j}y_{i,j}. \qquad (1)$$

This is an inner product of a $1 \times (q+2)$ vector by a $(q+2) \times 1$ vector. This rectangular matrix product is not one of the 36 matrix products obtained by tensoring.

The new grouping that Coppersmith and Winograd use for the second tensor power is obtained from the original grouping as follows.

For all nonnegative integers I, J, K with $I + J + K = 4$, merge the groups with group sequences (a, b, c) for which $a[1] + a[2] = I, b[1] + b[2] = J, c[1] + c[2] = K$. We obtain groups of entry products that are in one-to-one correspondence with the triples $(0, 0, 4)$, $(0, 1, 3)$, $(0, 2, 2)$, $(1, 1, 2)$ and their permutations.

It turns out that whenever $I = 0$ or $J = 0$ or $K = 0$, the bilinear form corresponding to (I, J, K) is a matrix product instance (e.g. the expression (1) above that corresponds to group $(2, 2, 0)$).

If all of the subproblems defined this way were matrix products, then exactly the same analysis as for the original construction can be performed. However, it turns out that the bilinear forms corresponding to $(1, 1, 2), (1, 2, 1)$ and $(2, 1, 1)$ are not matrix products.

Values. Even though not all groups correspond to matrix products, Coppersmith and Winograd find a way to use their analysis. They define the concept of *value* of a bilinear form which corresponds to how close a bilinear form is to containing a large matrix product. The exact definition of value is somewhat complicated. It is tied to the statement of the τ-theorem, and is a function of τ. Intuitively[1], the value $V(\tau)$ of a bilinear form T means that for very large n, the nth tensor power of T contains an embedding of a square matrix multiplication instance, the dimensions of which grow roughly as $V(\omega/3)^{n/\omega}$. For instance, the value of the bilinear form for $k \times k$ matrix multiplication is just $k^{3\tau}$.

If the border rank of T is r, then the border rank of the nth tensor power of T is at most r^n. If one can embed a square matrix product instance of dimensions $V(\omega/3)^{n/\omega}$ in the nth power of T, then one can use T to claim $(V(\tau)^{n/\omega})^{\omega} \leq r^n$ for each $\tau \leq \omega/3$. Hence, one can obtain the bound $\omega \leq 3\tau$ for $V(\tau) = r$.

Hence, the larger $V(\tau)$ is as a function of τ, the better the bilinear form is in terms of giving matrix multiplication algorithms.

The argument for the second tensor power proceeds similarly to the argument for the first power: find an embedding of a large number of independent instances of bilinear forms in the nth tensor power and then use a generalization of Schönhage's theorem to obtain a bound on ω. The bilinear forms that one uses in the nth tensor power are now no longer matrix products, but are all isomorphic tensor products of the 36 small bilinear forms that the second tensor power is partitioned in. They are hence tensor products of a matrix multiplication instance with some powers of the bilinear forms corresponding to $(1, 1, 2)$, $(1, 2, 1)$ and $(2, 1, 1)$. The generalization of Schönhage's theorem just says that if one can embed f independent instances isomorphic to some bilinear form T of value $V(\tau)$ into a bilinear form of border rank r, then

$$\omega \leq 3\tau \text{ where } V(\tau) = r/f.$$

Just as before, the embedding of the independent instances depends on variables a_{IJK} that define the fraction of each instance that comes from each of the 36 starting bilinear forms associated with the groups (I, J, K). The variables a_{IJK} together with some constraints define a new search

[1] This is a slight lie since the notion of value requires some symmetrization, so that the value of multiplying a $1 \times q$ vector by a $q \times 1$ vector is the third root of the value of $q \times q$ matrix multiplication.

space of algorithms, and one solves a constraint program to find the best algorithm in the search space, thus determining f and hence τ in the above expression.

The question now is, how does one analyze the values of the small bilinear forms? Coppersmith and Winograd analyzed the values of the $(1,1,2)$, $(1,2,1)$ and $(2,1,1)$ bilinear forms using Strassen's "laser method". Then they obtained their longstanding bound of $\omega < 2.376$.

They left as an open problem: if one starts from the third, or fourth tensor power of their construction, does the bound on ω improve?

To answer this question, one may need to consider new ways to group the entry products of the starting bilinear form so that the values of the obtained smaller bilinear forms are easy to analyze.

A straightforward generalization of the Coppersmith-Winograd second tensor power grouping for the Pth tensor power is for each integer I between 0 and $2P$ to group all variables the indices of which have group sequences that sum to I componentwise, i.e. if a variable has group sequence a, then place it in group I if $\sum_t a[t] = I$. Then the products $x_i y_j$ of the bilinear form are partitioned into groups corresponding to the triples (I, J, K) of nonnegative integers with $I + J + K = 2P$.

One can again show that if one of I, J, K is 0, then the bilinear form corresponding to (I, J, K) is a matrix product. It is not immediately clear, however, how one can determine the values of the bilinear forms for arbitrary groups (I, J, K). Moreover, since the number of these forms scales as P^2 (I and J can range almost independently up to P), analyzing large tensor powers P would be extremely time-consuming, if one is to analyze each separate form by hand.

For the third tensor power, the number of values to be analyzed is still manageable- there are only 7 values up to permutation, and 4 of these are matrix products so that only 3 need to be analyzed separately. Several groups of people attempted the Coppersmith-Winograd approach starting from the third tensor power. Surprisingly, however, the third power offered no improvement on the bound on ω. The fourth tensor power remained untouched until in 2010, Andrew Stothers [12] analyzed it. For the fourth tensor power, there are 10 values (up to permutation) to be analyzed. Stothers analyzed them by hand, formulated the nonlinear program defining the search space, solved it and determined that $\omega < 2.374$.

3 A Generalization and the Automated Approach

The main goal of my recent paper [15] is to present a generalization of the Coppersmith-Winograd analysis to arbitrary starting tensor powers. This entails first giving a systematic way of analyzing the values of the small tensors for arbitrary starting tensor powers, and then also giving a systematic way of constructing the nonlinear program defining the search space of matrix multiplication algorithms. To do this, I prove two main theorems. The first theorem shows that once you have upper bounds on the values, defining the nonlinear program for the search space can be done by just solving a linear system of equations. The second theorem shows that upper bounds on the values of the small bilinear forms can be computed by just solving linear systems and linear programs. As linear systems can be solved using matrix multiplication algorithms, we get the following interesting phenomenon.

Good matrix multiplication algorithms can be used to prove theorems about the existence of better matrix multiplication algorithms.

Generalizing the nonlinear program construction. The main difficulty in generalizing the construction of the nonlinear program defining the search space of algorithms is in the construction of the constraints. The original nonlinear programs for the first and second tensor powers of the Coppersmith-Winograd constructions had two main constraints: the linear constraint stating that the fractions a_{IJK} must sum to 1, and the constraint $V(\tau) = r/f$ that comes from the generalization of Schönhage's theorem. When considering higher tensor powers, additional constraints need to be added in order to optimize the bound on ω. These constraints stem from the fact that now the number of variables a_{IJK} is large.

The a_{IJK} variables are related in the linear system $\{X_I = \sum_J a_{IJK},\ Y_J = \sum_I a_{IJK},\ Z_K = \sum_I a_{IJK}\}$ mentioned earlier, where X_I, Y_J, Z_K are the fractions of the components of the $(x, y$ and $z)$ variable index sequences of the final tensor power n that are in groups I, J and K of the starting tensor power P. If the linear system has full rank (as in the case of the first and second tensor power), then for any fixed setting of X_I, Y_J, Z_K, the variables a_{IJK} are determined, so that once one picks X_I, Y_J, Z_K, there is no more freedom and τ is determined. For larger starting tensor powers however, the linear system no longer has full rank. Because of this, now the variables a_{IJK} are no longer determined from X_I, Y_J, Z_K and can vary so that the search space for matrix multiplication algorithms is larger. The extra constraints that I add in my final nonlinear program then restrict the a_{IJK} space to those values that minimize the bound on τ, and hence on ω. It turns out that to formulate these constraints, it suffices to be able to solve the linear system $\{X_I = \sum_J a_{IJK},\ Y_J = \sum_I a_{IJK},\ Z_K = \sum_I a_{IJK}\}$ for a subset of the a_{IJK} variables in terms of X_I, Y_J, Z_K and the rest of the a_{IJK}.

Generalizing the value computation. The basic idea for the value computation is that the values of the small bilinear forms can be analyzed in a very similar way to the original bilinear form, in terms of the original 6 matrix products. Since for every I, J, K, the bilinear form T_{IJK} corresponding to the group triple (I, J, K) is just formed by merging the groups corresponding to group sequences that sum to I, J and K componentwise, T_{IJK} is just the sum of all forms T_{ijk} corresponding to the length P group sequences (i, j, k) for which $\sum_t i[t] = I, \sum_t j[t] = J, \sum_t k[t] = K$. Each form T_{ijk} is just a tensor product of the original 6 matrix products and is hence a matrix product instance itself. Therefore, we have that each T_{IJK} is just a sum of matrix product instances. Of course, these matrix products again share variables and are not independent. We show that after taking a large tensor power of T_{IJK} we can essentially remove the dependencies and use Schönhage's theorem to obtain a lower bound on the value of T_{IJK}.

We consider the nth tensor power of T_{IJK} and we want to show that it contains a large number f of bilinear forms isomorphic to some bilinear form T. T is actually a tensor product of n of the bilinear forms T_{ijk} for different i, j, k. For all length P group sequences i, j, k with $i[t]+j[t]+k[t] = 2$ in each component t, we define a variable a_{ijk} so that a_{ijk} is the fraction of the n bilinear forms forming the tensor product T that are of the form T_{ijk}. Then just as in the case of the nonlinear program formulation, we have that the variables a_{ijk} are related via a linear system. This linear system constrains some of the variables, while leaving some of them free, depending on the linear system rank. We then show that for any guess for τ, we can maximize our lower bound on the value of T_{IJK} by just adding linear constraints. That is, after solving the linear system, the value computation for any guess for τ only boils down to solving a linear program. (We then do binary search for the best τ, solving a new nonlinear program each time.)

In the end, my paper shows that the Coppersmith-Winograd approach can be automatically applied to any starting tensor power of the Coppersmith-Winograd construction. After establishing the framework, I actually apply the approach to the eighth tensor power and obtain $\omega < 2.3727$.

One could ask, why did I concentrate on the 8th tensor power instead of the 5th or 6th? Intuitively, it makes sense to start from as large as a tensor power as possible, as you would get better and better bounds on ω. In his thesis, Stothers found a shortcut in computing the lower bounds for the values for the 4th tensor power. He showed that you can use the values for the second tensor power to compute those for the 4th. In my paper I generalize his observation and used it in tandem with my value theorem to show that lower bounds for the values for any even tensor power $2K$ can be obtained from the values for the Kth power by solving a linear program. This recursive approach reduces the number of variables in the linear program instances for the value computation for the Kth tensor power from $\Omega(K^3)$ to $O(K^2)$. In this sense, the jump from the 4th power to the 8th seemed very doable.

Should we worry about precision? When solving a problem numerically we typically need to worry about the precision to which our constraints are satisfied. The nonlinear program that defines the search space for matrix multiplication algorithms contains several equality constraints. Moreover, its variables a_{IJK} need to be set to rational numbers as for some large n, each value $a_{IJK} \cdot n$ needs to be an integer. Hence, it may look like we would need really good precision to make sure that all constraints are satisfied.

It turns out, however, that we do not need to solve our nonlinear program to high precision. The reason for this is that the equality constraints have some nice structure, and given any solution to the nonlinear program that has some imprecision, we can convert it to an *exact* feasible solution that may change the upper bound on ω only slightly.

The equality constraint coming from the extended τ-theorem, $V(\tau) = r/f$, can be relaxed to an inequality constraint $V(\tau) \geq r/f$, and this would only affect the quality of the upper bound on ω. The equality constraint stating that the sum of the variables a_{IJK} is 1 can also be replaced by a ≤ 1 inequality, and after a feasible solution is found, the equality can always be satisfied since one of the variables does not appear in any of the other equality constraints.

The rest of the equality constraints have some very nice structure as follows. There is a subset of the variables S so that each constraint sets some variable of S to a ratio of powers of variables outside of S. Because of this, if we get rational values for the variables outside of S, regardless of the original precision, we can obtain rational values for the variables in S that satisfy all equality constraints exactly. The only thing that gets affected by this change of the setting for the variables in S is the quality of the bound on ω.

4 Some open questions

The most lingering question about this approach is, can one express the bound on ω as a function of the starting tensor power P? If we could do that, then we could just take P to ∞ and get the best bound obtainable with this approach.

Another possible direction to consider is to find a new way to partition the products of the starting bilinear form into pieces that have large and easy to analyze values. In the current arguments, we obtained the bilinear form pieces of each tensor power of the construction by merging together bilinear forms that were tensor powers of the original 6 matrix product instances. Finding entirely different ways to split up and analyze the bilinear form may require genuinely new insight.

More generally, consider some tensor power T of the Coppersmith-Winograd construction. Our current arguments only allow us to find an *approximation* to the best matrix multiplication instance that one can embed in T. Can we figure out what the *best* embedding of a matrix product instance is, even for a fixed tensor power? What matrix product instances does the 100th tensor power contain? Figuring this out would not only entail bounds on ω, but it may give us some intuition on what the best embeddings look like in general, as the tensor power grows.

Furthermore, it is unclear that the Coppersmith-Winograd bilinear form is the best one to start from. How can we find good bilinear forms to use as our "base case", and how can we find good bounds on their border rank? Can we automate the search for either of these?

The notion of border rank was a useful relaxation of the notion of rank that allowed for large improvements in matrix multiplication algorithms. Cohn and Umans [5] have recently introduced a new relaxed notion of tensor rank called *s-rank* that could lead to new progress on our bounds on ω. The s-rank of a bilinear form T is the smallest rank of a bilinear form T' that has the same tensor support as T, i.e. if T has tensor t_{ijk} then one considers all T' with tensors t'_{ijk} such that $t_{ijk} \neq 0$ iff $t'_{ijk} \neq 0$ and takes the minimum of their ranks. The s-rank of a bilinear form is always at most its rank, but it can be smaller or larger than its border rank. One can define a new exponent $\omega_s = \inf\{h \mid s\text{-}Rank(\langle n, n, n\rangle)\}$ which is clearly at most ω. However, Cohn and Umans show that in fact if $\omega_s = 2$, then also $\omega = 2$. Hence if one is trying to show that $\omega = 2$, it suffices to show that some bilinear form with the same support as matrix multiplication has small asymptotic rank. The question then is, which bilinear form?

Finally, the latest improvements on ω have stemmed from an automated approach. A natural question is, what other algorithm design problems can such an automated approach solve? For instance, can one find better algorithms for integer multiplication in this way?

References

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. The design and analysis of computer algorithms. *Addison-Wesley Longman Publishing Co., Boston, MA*, 1974.

[2] D. Bini. Relations between exact and approximate bilinear algorithms. applications. *Calcolo*, 17(1):87–97, 1980.

[3] D. Bini, M. Capovani, F. Romani, and G. Lotti. $O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication. *Inf. Process. Lett.*, 8(5):234–235, 1979.

[4] M. Bläser. A lower bound for the rank of matrix multiplication over arbitrary fields. In *Proc. FOCS*, pages 45–, 1999.

[5] H. Cohn and C. Umans. Fast matrix multiplication using coherent configurations. In *Proc. SODA*, 2013.

[6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9(3):251–280, 1990.

[7] J. M. Landsberg. New lower bounds for the rank of matrix multiplication. *CoRR*, abs/1206.1530, 2012.

[8] E. Mossel, R. O'Donnell, and R. A. Servedio. Learning juntas. In *Proc. STOC*, pages 206–212, 2003.

[9] A. Schönhage. Partial and total matrix multiplication. *SIAM J. Comput.*, 10(3):434–455, 1981.

[10] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. FOCS*, pages 605–614, 1999.

[11] A. Shpilka. Lower bounds for matrix product. *SIAM J. on Computing*, 32(5):1185–1200, 2003.

[12] A. Stothers. *Ph.D. Thesis, U. Edinburgh*, 2010.

[13] V. Strassen. Vermeidung von Divisionen. *Crelle J. Reine Angew. Math.*, 1973(264):184–202, 1973.

[14] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10:308–315, 1975.

[15] V. Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proc. STOC*, pages 887–898, 2012.

[16] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *JACM*, 49(3):289–317, 2002.