T.J. Smeding

Fast Large-Integer Matrix Multiplication

Bachelor thesis

11 July 2018

Thesis supervisors: dr. P.J. Bruin
dr. K.F.D. Rietveld

Leiden University
Mathematical Institute
Leiden Institute of Advanced Computer Science

Abstract

This thesis provides two perspectives on fast large-integer matrix multiplication. First, we will cover the complexity theory underlying recent developments in fast matrix multiplication algorithms by developing the theory behind, and proving, Schönhage's τ-theorem. The theorems will be proved for matrix multiplication over commutative rings. Afterwards, we will discuss two newly developed programs for large-integer matrix multiplication using Strassen's algorithm, one on the CPU and one on the GPU. Code from the GMP library is used on both of these platforms. We will discuss these implementations and evaluate their performance, and find that multi-core CPU platforms remain the most suited for large-integer matrix multiplication.

Contents

1 Introduction

Matrix multiplication is the core operation in linear algebra, and this branch of mathematics has earned its place as one of the most practically applicable in computer science: while the field of computer graphics thrives on fast small matrix multiplication, numerical simulations, machine learning and some graph-theoretic algorithms often deal with larger matrices. Fast dense matrix multiplication over floating-point numbers has enjoyed much attention in academic research, partly due to its applicability, partly because it is a simple but non-trivial problem to test optimisation and parallelisation techniques on. However, the same cannot be said about large-integer matrix multiplication.

When looking for a fast program performing a specific task, we are aided by complexity theory on the theoretical side and optimisation techniques on the practical side. Often these are complementary, the road to success being selection of an optimal algorithm followed by optimisation of an implementation of that algorithm. In the case of matrix multiplication, complexity theory yields a variety of algorithms, the most naive using $O(n^3)$ and the most sophisticated (at the time of writing) using $O(n^{2.3728639})$ [13] element multiplications. (It is conjectured that algorithms exist with $O(n^{2+\varepsilon})$ element multiplications for all $\varepsilon > 0$, but the required techniques remain elusive [1].) However, the proofs of existence of these low-complexity algorithms are in general non-constructive, and no evidence has been found that such an implementation exists or would be feasible to make.

To multiply large-integer matrices in practice, we first need to know how to multiply large integers efficiently. However, this is largely a solved problem: even though no known-optimal algorithm has been found yet, it is clear that multiplication of two n-bit numbers takes at least $\Omega(n)$ steps, and there are algorithms that miss this bound by only logarithmic factors (e.g. the Schönhage–Strassen algorithm [22], which takes $O(n \log n \log\log n)$ steps for n-bit numbers). Practical implementations like the GMP library have a variety of finely tuned algorithms [8]. Therefore, at least when the subject concerns complexity theory, we will only focus our attention on the question of matrix multiplication itself.

With exception of new group-theoretic developments (e.g. [4]), most modern results, starting with Coppersmith and Winograd in 1987 [5], build upon the so-called "τ-theorem" (or: asymptotic sum inequality) proved by Schönhage in 1981 [21]. The proof given there is long and assumes much background knowledge; other sources either provide no proofs of this and its theoretical background and refer to previous papers (e.g. [13], [28] (corrects conference version [27])) or give comprehensive, though imperfect or informal accounts (e.g. [3], [7]). Finally, the found sources state the theorems over a field, while they can also be proved using just commutative rings. *To understand the theoretical perspective on the topic of matrix multiplication, the first part of this thesis aims to give an accessible and rigorous proof of the τ-theorem and its supporting theory over commutative rings.*

Having understood what complexity theory yields on the question of fast matrix multiplication, the focus shifts to more practical issues. For large integer multiplication, we will use, and in some cases adapt, the GMP library as mentioned; for matrix multiplication, we will use Strassen's algorithm (with $O(n^{2.81})$ element multiplications). We will compare various implementations of this combination on both the CPU and the GPU. *The second part of this thesis aims to describe and optimise implementations for large-integer matrix multiplication and to evaluate their performance.*

The theoretical part will be covered in Section 2, and the practical part in Section 3. Having explored both sides of the topic, we will compare their methods and results in Section 4.

2 Complexity Theory

2.1 Matrix Multiplication

Consider a $k \times m$ matrix $X = (x_{\kappa\mu})_{1\leq\mu\leq m}^{1\leq\kappa\leq k}$ and an $m \times n$ matrix $Y = (y_{\mu\nu})_{1\leq\nu\leq n}^{1\leq\mu\leq m}$ over some ring R. Multiplied together, we get a $k \times n$ matrix $Z = XY = (z_{\kappa\nu})_{1\leq\nu\leq n}^{1\leq\kappa\leq k}$, where the entries are given by:

$$z_{\kappa\nu} = \sum_{\mu=1}^{m} x_{\kappa\mu} y_{\mu\nu}$$

We would like to know how many arithmetic operations are sufficient to compute this matrix product. By computing each entry naively, we get $O(kmn)$ multiplications and $O(kmn)$ additions. If we take all dimensions to be equal to n, this means we can multiply matrices in $O(n^3)$ arithmetic operations. But can we do better?

Note: The order of presentation of the material and a number of proofs in Section 2 are inspired by [3]. Whenever a proof is not original, this is indicated with a citation after the word "*Proof.*"

2.2 Computation

To be able to reason correctly about arithmetic operations, we need to define exactly what we mean by those.

Definition 2.2.1 (Computational Structure). A computational structure is a tuple $(M, \Phi, ¢)$ where M is a set, Φ is a set of functions $\varphi : M^s \to M$ of a certain arity s (dependent on φ), and $¢ : \Phi \to \{0, 1, 2, \ldots\}$ assigns a *cost* to each of the functions in Φ.

In a computational structure, we want to compute things.

Definition 2.2.2 (Computation). A computation in a computational structure $(M, \Phi, ¢)$ is a pair (β, X), where β is a sequence of length m and $X \subseteq M$ is a set of *inputs*. (If the set X is understood or irrelevant, we may omit it and write just β for the computation.) For each $1 \leq i \leq m$, the i'th item of β is either a singleton (w_i), where $w_i \in X$, or an $(s+2)$-tuple of the form $(w_i, \varphi_i, j_1^i, \ldots, j_s^i)$, where $w_i \in M$, $\varphi_i \in \Phi$, s is the arity of φ_i, and $1 \leq j_k^i < i$ for $1 \leq k \leq s$. We require that $w_i = \varphi_i(w_{j_1^i}, \ldots, w_{j_s^i})$. Furthermore:

- We say that β *computes* a set $Y \subseteq M$ if for all $y \in Y$, we have $y = w_i$ for some $1 \leq i \leq m$.
- The *cost* $C(\beta)$ of β is $\Gamma(\beta) \coloneqq \sum_{i \in I} ¢(\varphi_i)$, where $I \subseteq \{1, \ldots, m\}$ is the subset of indices i where β_i is not a singleton.

In other words, in a computation, each item must either be an input, or be computable in one step from previously calculated items. The kind of computation defined here is closest to the usual concept of a "straight-line program", since we do not allow conditional branching. This allows us to represent these computations in a more structured way later on.

Definition 2.2.3 (Complexity). The complexity of the set Y given X is $C(Y, X) \coloneqq \min\{\Gamma(\beta) : \beta \text{ computes } Y \text{ and has inputs } X\}$.

If there is any computation (β, X) that computes Y (in other words, if Y only "depends on" X with respect to the operations in Φ), then the set in Definition 2.2.3 is non-empty, so since the naturals are well-ordered, $C(Y, X)$ exists. This existence then exhibits at least one *optimal computation*: a computation of Y from X with cost equal to the complexity of Y given X.

2.3 Tensors

Recall that if $X = (x_{\kappa\mu})_{1\leq\mu\leq m}^{1\leq\kappa\leq k}$ and $Y = (y_{\mu\nu})_{1\leq\nu\leq n}^{1\leq\mu\leq m}$, that for $Z = XY = (z_{\kappa\nu})_{1\leq\nu\leq n}^{1\leq\kappa\leq k}$ we then have:

$$z_{\kappa\nu} = \sum_{\mu=1}^{m} x_{\kappa\mu} y_{\mu\nu}$$

From the $km \cdot mn$ possible products of an entry from X and an entry from Y, this sum contains exactly m terms, each with weight 1. Now consider an $n_1 \times n_2$ matrix A as just a collection of $n_1 \cdot n_2$ entries, numbered $a_1, \ldots, a_{n_1 \cdot n_2}$. Then we can give a three-dimensional array of $km \times mn \times kn$ numbers, say $t_{i_1 i_2 i_3}$, such that for all $1 \leq i_3 \leq kn$:

$$z_{i_3} = \sum_{i_1=1}^{km} \sum_{i_2=1}^{mn} t_{i_1 i_2 i_3} x_{i_1} y_{i_2} \tag{1}$$

Such a three-dimensional array of numbers we will call a *tensor*. In this case, we have $t \in R^{km\times mn\times kn}$. (In the more usual definition of a tensor, $R^{km\times mn\times kn}$ corresponds with the R-module $M_{k,m}^* \otimes M_{m,n}^* \otimes M_{k,n}$, where $M_{a,b}$ is the R-module of $a \times b$ matrices and M^* denotes the R-dual of M.) Note that a tensor encodes an array of bilinear forms, and in the case of the matrix product, each of these forms computes one entry of the product matrix.

Definition 2.3.1. For $k, m, n \in \mathbb{N}$, the *matrix multiplication tensor* is the tensor defined above, i.e. the $t \in R^{km\times mn\times kn}$ such that, if x_i and y_i are the coefficients of, respectively, a $k \times m$ matrix X and an $m \times n$ matrix Y, the values given by Eq. 1 are the coefficients of the $k \times n$ matrix XY. This tensor is written $\langle k, m, n\rangle$.

Remark 2.3.2. Let $t = \langle k, m, n\rangle$. If we index t using double indices, ranging from $t_{11,11,11}$ to $t_{km,mn,kn}$, we find that $t_{ii',jj',\ell\ell'} = \delta_{i'j}\delta_{i\ell}\delta_{j'\ell'}$ (where δ is the Kronecker delta).

Later, in Section 2.5, this notation $\langle k, m, n\rangle$ will return and be of importance.

2.3.1 Reduction of Abstraction

In practice, we will specialise our definition of a computation as follows. The work will be based on the computational structure $(R, \Phi, \dot{\varphi})$, where R is a *commutative ring* such that $R \neq \{0\}$, and Φ is the following set: $\Phi_0 \coloneqq \{+, -, \cdot\} \cup \{(x \mapsto r \cdot x) : r \in R\} \cup R$. (The first set contains binary functions, the second unary functions, and the third nullary functions (i.e. functions without arguments).) Operations of the type $(x \mapsto r \cdot x)$ for some constant $r \in R$ will be called *constant-multiplications* to distinguish them from the usual multiplication $(\cdot)$. Note that all operations are basic ring operations and are thus respected by ring homomorphisms.

Note that our requirement that $R \neq \{0\}$ is not particularly limiting: if $R = \{0\}$, all matrices are zero matrices and we have nothing to compute.

The reason we focus on commutative rings and not on rings in general is that dividing out a commutative ring over a maximal ideal produces a field. This means that an algorithm working on R will also work on that smaller field, and being able to reason about fields will give us some leverage to prove things later on.

As a cost function, we will always set $\dot{\varphi}(r) = 0$ for $r \in R$. The cost of the three binary operations and of constant-multiplication will vary, but usually, we will use a *modified Ostrowski model*: here, we assign cost 1 to multiplication, and cost zero to addition, subtraction and constant-multiplication. This cost function will play an important role once we are ready to turn to low-complexity algorithms for matrix multiplication.

The actual *Ostrowski model* assumes that we are working over a field K and division is one of the operations, which then gets cost 1 like multiplication. In Section 2.4 we will see that if K is infinite, we can eliminate divisions without impacting cost, so we can do without them anyway; if K is finite, we might not be able to eliminate divisions, but we can replace them with at most a constant number of multiplications (dependent on $\#K$), so because we will be using big-O notation later, we can still eliminate divisions without problems.

2.3.2 Cost

The operations in Φ_0 above are, besides constant introduction, all natural ring operations, and thus restrictions of the same operations on a polynomial ring of R. Let $\tilde{\Phi}_0$ be the set of operations $\{+, -, \cdot\} \cup \{(x \mapsto r \cdot x) : r \in R\} \cup R$, which are now to be taken as functions on some polynomial ring over R. Which polynomial ring exactly will depend on context, but the use of $\tilde{\Phi}_0$ will never be ambiguous. (We denote the polynomial ring over R in the variables $x_1, \ldots, x_n$ by $R[x_1, \ldots, x_n]$.)

Using $\tilde{\Phi}_0$, the number of operations needed to calculate a set of quadratic forms may be captured as follows.

Definition 2.3.3. For a set of quadratic forms $F = \{f_1, \ldots, f_k\}$ ($f_\kappa = \sum_{\mu,\nu=1}^{N} t_{\mu\nu\kappa} x_\mu x_\nu$), we define its *cost* $C(F)$ with respect to the computational structure $(R, \Phi_0, ¢)$ (with R a commutative ring) as its complexity $C(F, \{x_1, \ldots, x_N\})$ in the computational structure $(R[x_1, \ldots, x_N], \tilde{\Phi}_0, ¢)$.

In the definition, we do not provide the *values* of the elements of the vector x as inputs, but only the abstract variables corresponding to those elements. Considering the operations in $\tilde{\Phi}_0$, all computation steps can only yield polynomials in the input variables, which is why a polynomial ring is used as M in the computational structure. In this manner, we guarantee that the required values are computed for any given input values, since the calculation cannot "depend" on properties that only hold for certain input values. However, this definition may not be intuitive at first; an alternative that does capture the requirement that the computation must work for all inputs, is having no inputs and letting computation steps be functions with N arguments. The operations would then construct functions from functions. This captures the intuition of actually computing with values, but makes it harder to reason about computation steps since functions are opaque. Since these functions would in practice be represented by polynomial expressions anyway, we choose to use polynomials directly.

The result of this choice is that we do not actually "compute" in our computations, we merely find a way to *build* a specific set of polynomials, usually a set of bilinear forms, in as few steps as possible. Since the things we can do without cost are introduction of constants and inputs (in our case, $x_1, \ldots, x_N$) and forming linear combinations, we will build them up from degree ≤ 1 polynomials.

Definition 2.3.4. We call the tensor t in Definition 2.3.3 the tensor *corresponding to* F. It is clear that we can identify sets of bilinear forms and their corresponding tensors, since one uniquely identifies the other.

For brevity, we will denote by $C^{\cdots}(F)$ the cost of the set of quadratic forms F where the operations listed in the superscript have cost 1 and the other operations have cost 0. For example, the Ostrowski cost of F is $C^{*/}(F)$ (which only works over a field and if we add division to Φ), and the modified Ostrowski cost of F is $C^{*}(F)$ (which works over commutative rings).

We will see later that our choice of the (modified) Ostrowski model (i.e. ignoring linear operations) is not completely arbitrary; in the light of the algorithms we will study later, it will be most

natural to use Ostrowski, but more importantly, we will see that penalising additions and constant-multiplications will make no difference in the overall algorithmic complexity in our case.

2.3.3 Rank

Introduce the notation $\operatorname{lin}_R\{A_1, \ldots, A_n\} \coloneqq \{\lambda_1 A_1 + \cdots + \lambda_n A_n : \lambda_1, \ldots, \lambda_n \in R\}$, the linear combinations of the elements $A_1, \ldots, A_n$ of an R-module. If the ring is understood, "lin_R" may be written "lin". Note that $\operatorname{lin} \varnothing = \{0\}$.

We make the following definition in relation to quadratic forms.

Definition 2.3.5. For a set of quadratic forms $F = \{f_1, \ldots, f_k\}$ ($f_\kappa = \sum_{i,j=1}^n t_{ij\kappa} x_i x_j$), define its *rank* $R_{\text{quad}}(F)$ as the minimal nonnegative integer ℓ such that there exist products $P_1, \ldots, P_\ell$ of the form $P_i = \left(\sum_{j=1}^n u_{ij} x_j\right)\left(\sum_{j=1}^n v_{ij} x_j\right)$ for certain $u_{ij}, v_{ij} \in K$ such that $F \subseteq \operatorname{lin}\{P_1, \ldots, P_\ell\}$.

In light of what we will do later, it makes sense make an analogous definition for bilinear forms with a more specific representation.

Definition 2.3.6. For a set of *bilinear* forms $F = \{f_1, \ldots, f_k\}$ ($f_\kappa = \sum_{i=1}^m \sum_{j=1}^n t_{\kappa ij} x_i y_j$), define its *rank* $R(F)$ as the minimal nonnegative integer ℓ such that there exist products $P_1, \ldots, P_\ell$ of the form $P_i = \left(\sum_{j=1}^m u_{ij} x_j\right)\left(\sum_{j=1}^n v_{ij} y_j\right)$ for certain $u_{ij}, v_{ij} \in K$ such that $F \subseteq \operatorname{lin}\{P_1, \ldots, P_\ell\}$.

Note that for the bilinear rank we also want the individual products to be bilinear, while such a requirement would not make sense in the case of quadratic forms.

Since a bilinear form is also a quadratic form, a set of bilinear forms F not only has a defined $R(F)$ but also a defined $R_{\text{quad}}(F)$. As the next lemma shows, they differ at most by a factor 2.

Lemma 2.3.7. For a set of bilinear forms F as in Definition 2.3.6, we have:

$$R_{\text{quad}}(F) \leq R(F) \leq 2R_{\text{quad}}(F)$$

Proof. (After [3, Th. 4.7]) Since bilinear products are also quadratic products, we have the first inequality: $R_{\text{quad}}(F) \leq R(F)$.

For the second inequality, we have by Definition 2.3.5: (for certain $w_i, u_{ij}, u'_{ij}, v_{ij}, v'_{ij} \in R$)

$$\begin{aligned} f_\kappa &= \sum_{i=1}^{R_{\text{quad}}(F)} w_i \left(\sum_{j=1}^m u_{ij} x_j + \sum_{j=1}^n u'_{ij} y_j\right)\left(\sum_{j=1}^m v_{ij} x_j + \sum_{j=1}^n v'_{ij} y_j\right) \\ &= \sum_{i=1}^{R_{\text{quad}}(F)} w_i \left(\sum_{j=1}^m u_{ij} x_j\right)\left(\sum_{j=1}^n v'_{ij} y_j\right) + \sum_{i=1}^{R_{\text{quad}}(F)} w_i \left(\sum_{j=1}^n v_{ij} x_j\right)\left(\sum_{j=1}^m u'_{ij} y_j\right) + \underbrace{(\cdots)}_{=0} \end{aligned}$$

We can ensure that the terms $(\cdots)$ all have degree at least 2 in either x or y; then they are zero because f_κ is a *bilinear* form in the vectors x and y. But here we have written down $2R_{\text{quad}}(F)$ products of the form required by Definition 2.3.6 that suffice to produce F, so we find $R(F) \leq 2R_{\text{quad}}(F)$. □

Since it seems unproductive to multiply terms of degree > 1 when computing quadratic forms, we might guess that $C^*(F) = R_{\text{quad}}(F)$. In fact, this can be proved, but like a number of other proofs here it may require more gymnastics than expected.

Theorem 2.3.8. For a set of quadratic forms F as in Definition 2.3.5 we have $C^*(F) = R_{\text{quad}}(F)$.

Proof. We have $C^*(F) \leq R_{\text{quad}}(F)$ because by definition we have an algorithm with $R_{\text{quad}}(F)$ multiplications, so below we prove that $R_{\text{quad}}(F) \leq C^*(F)$.

Let $c = C^*(F)$, and consider a computation β that computes F using c multiplications. Note that β cannot use divisions, since we are working over a ring. Each of the c multiplications in β produces a polynomial in $x_1, \ldots, x_m$; call them $P_1, \ldots, P_c$. Then by the nature of our cost model, each f_κ is a polynomial of degree ≤ 1 plus a linear combination of these P_i.

For a polynomial P, let $H_j(P)$ be the *homogeneous part* of degree j of P, i.e. the sum of all monomials of degree j in P.

Remark ($\star$). If at some point in a computation we have computed a polynomial P, then we can calculate without cost the polynomials $H_0(P)$, $H_1(P)$ and $H_{\geq 2}(P)$, where $H_{\geq n}(P)$ indicates the sum of all monomials in P with degree $\geq n$. Indeed, the first two can be computed without cost using addition, constant-multiplication, introduction of ring constants and reference of inputs, and the third can be computed by subtracting the first two from P.

The left and right inputs of each multiplication are linear combinations of the previous multiplication results, plus possibly any terms that can be computed without cost. In other words, for each i we have:

$$P_i = \underbrace{\left(\sum_{j=1}^{i-1} u_{i,j} P_j + \sum_{j=1}^{n} v_{i,j} x_j + w_i\right)}_{L_i} \underbrace{\left(\sum_{j=1}^{i-1} u'_{i,j} P_j + \sum_{j=1}^{n} v'_{i,j} x_j + w'_i\right)}_{R_i}$$

for certain $u_{i,j}, u'_{i,j}, v_{i,j}, v'_{i,j}, w_i, w'_i$ in R.

If for a particular i, $\sum_{j=1}^{i-1} u_{i,j} P_j$ or $\sum_{j=1}^{i-1} u'_{i,j} P_j$ is a nonzero constant, the computation can be changed such that this constant is part of w_i or w'_i instead and the relevant linear combination of previous P_j in P_i is zero. So without loss of generality, we can assume that the linear combinations of previous P_j are either zero or have degree ≥ 1. Therefore, we can extract w_i and w'_i from L_i and R_i by ($\star$), so without loss of generality we can take $w_i = w'_i = 0$ for all i. (Otherwise, using some constant-multiplications and extra additions, we could construct an alternative computation with an equal number of products that does have $w_i = w'_i = 0$ for all i by multiplying the w_i's and w'_i's out.) Using induction on i, one can then show that $H_0(P_i) = H_1(P_i) = 0$ for all i (because $H_0(L_i) = H_0(R_i) = 0$). This means (by ($\star$)) that for all i we can separate L_i into $\sum_{j=1}^{i-1} u_{i,j} P_j$ and $\sum_{j=1}^{n} v_{i,j} x_j$, and R_i into analogous parts. But that means we can separate each P_i into $H_2(P_i)$ and $H_{\geq 3}(P_i)$, where $H_2(P_i)$ only depends on the linear combination of inputs in L_i and R_i and not on the previous products.

Since the polynomials in F are homogeneous of degree 2, we have $F \subseteq \text{lin}\{H_2(P_1), \ldots, H_2(P_c)\}$. By working inductively backwards, we find that we only need $H_2(P_i)$ for each i (and not $H_{\geq 3}(P_i)$), so we can let $u_{i,j} = u'_{i,j} = 0$ in each P_i. This way, each of the P_i becomes a product of two linear combinations of input variables, which means that β had at least $R_{\text{quad}}(F)$ multiplications in the beginning. Therefore, $R_{\text{quad}}(F) \leq c = C^*(F)$. □

Corollary 2.3.9. For F a set of bilinear forms as in Definition 2.3.6, $C^*(F) \leq R(F) \leq 2C^*(F)$.

Proof. Follows from Lemma 2.3.7 and Theorem 2.3.8. □

2.4 Divisions & Rank

If our commutative ring R is actually a field K, we should also be able to divide in our computations. Intuitively, even if we have divisions, it seems unnecessary to use them to calculate bilinear forms. If K is finite, say $\#K = q$, then $x/y = x \cdot y^{q-2}$ for $x \in K$ and $y \in K^*$. So we can actually eliminate divisions from the computation while keeping cost within a constant factor of the original cost. This means that later, when in Definition 2.6.3 we look at the big-O growth order of the cost, this constant factor gets discarded, and adding divisions did not in fact help at all.

If K is *infinite*, this trick does not work, but there is a more involved argument that we can use to prove that in this case, adding divisions does not even impact cost at all. This is what we will do in the rest of this section. We will measure complexity in a computational structure with $M = K(x_1, \ldots, x_N)$ (as opposed to $R[x_1, \ldots, x_N]$).

One potential issue about allowing divisions is worth noting: since division is only a *partial* operation, it being impossible to divide by zero, care must be taken that a computation using divisions is actually defined on all input matrices. Since in a computation with divisions there is at least one division (namely, the first one) that has a polynomial in the inputs as its denominator, this problem of definition is particularly striking when K is an algebraically closed field, where every polynomial of degree ≥ 1 has a root. Any computation with divisions will, however, have a subset of input values $S \subseteq R^N$ for which no division by zero is attempted. For such a computation, Theorem 2.4.3 below constructs a new computation without divisions that, naturally, provides the same answers when input values are chosen in S. However, the above is not a problem in the theorem as stated, because in our computational structure, no actual values are ever plugged into the computation.

We will first study more general quadratic forms instead of bilinear forms, because in that case we can derive a more natural special form of the calculation as well.

First we need two lemmas that do not depend on the fact that K is infinite. For a polynomial φ in multiple variables, let $\deg \varphi$ denote its *total degree*.

Lemma 2.4.1. Let K be any field and let $S \subseteq K$ be a subset. If $\varphi_1, \ldots, \varphi_n \in K[x_1, \ldots, x_N]$ are nonzero polynomials and $\#S > \sum_{i=1}^n \deg \varphi_i$, then there are $\alpha_1, \ldots, \alpha_N \in S$ such that $\varphi_i(\alpha_1, \ldots, \alpha_N) \neq 0$ for all i.

Proof. Let $\varphi = \prod_{i=1}^n \varphi_i$. Then φ is still nonzero and $d = \deg \varphi = \sum_{i=1}^n \deg \varphi_i < \#S$. The lemma for $n = 1$ then immediately proves the result, so we just consider the case $n = 1$. We use induction on N.

If $N = 1$, then φ has at most $d < \#S$ roots, so there is an $\alpha_1 \in S$ such that $\varphi(\alpha_1) \neq 0$.

If $N > 1$, assuming that the lemma holds for any polynomial in $K[x_1, \ldots, x_{N-1}]$, we write $\varphi = \sum_{j=1}^d x_1^j \psi_j(x_2, \ldots, x_N)$. Let $k > 0$ be the largest index such that ψ_k is nonzero. (This k exists since φ is nonzero.) Since ψ_k has degree at most $d - k < d$, we have $\#S > \deg \varphi > \deg \psi_k$, so by the induction hypothesis there are $\alpha_2, \ldots, \alpha_N \in K$ such that $\psi_k(\alpha_2, \ldots, \alpha_N) \neq 0$.

But now $\varphi(x_1, \alpha_2, \ldots, \alpha_N)$ is an element of $K[x_1]$ of degree at most $d < \#S$, and it is nonzero since the term x_1^k has a nonzero coefficient (namely, $\psi_k(\alpha_2, \ldots, \alpha_N)$). Using the induction base case, we obtain an $\alpha_1 \in S$ such that $\varphi(\alpha_1, \ldots, \alpha_N) \neq 0$. □

Lemma 2.4.2. For any field K, the formal power series $\sum_{i=0}^{\infty} a_i z^i \in K[[z]]$ is invertible if and only if $a_0 \neq 0$. Then its inverse is equal to $\frac{1}{a_0}(1 + q + q^2 + \cdots)$ where $q = -\sum_{i=1}^{\infty} \frac{a_i}{a_0} z^i$.

Proof. In $K[[z]]$ we have the equality: $\frac{1}{1-q} = 1 + q + q^2 + \cdots$. If $a_0 \neq 0$, we have:

$$\left(\sum_{i=0}^{\infty} a_i z^i\right) \cdot \frac{1}{a_0}(1 + q + q^2 + \cdots) = \left(\sum_{i=0}^{\infty} a_i z^i\right) \cdot \frac{1}{a_0} \frac{1}{1 + \sum_{i=1}^{\infty} \frac{a_i}{a_0} z^i} = \frac{\sum_{i=0}^{\infty} \frac{a_i}{a_0}}{1 + \sum_{i=1}^{\infty} \frac{a_i}{a_0} z^i} = 1$$

so the series is indeed invertible. On the other hand, if $a_0 = 0$, any power series $\sum_{i=0}^{\infty} b_i z^i$ multiplied by the given series (which is then equal to $\sum_{i=1}^{\infty} a_i z^i$) will give as product the series $z\left(\sum_{i=0}^{\infty} a_{i+1} z^i\right)\left(\sum_{i=0}^{\infty} b_i z^i\right)$, which can never equal 1. □

Now we can prove the theorem about quadratic forms, which is originally due to Strassen [25].

Theorem 2.4.3 (Strassen). Let F be a set of quadratic forms as in Definition 2.3.5. If K is an infinite field, then we have $C^*(F) = C^{*/}(F)$.

Proof. (After [3, Th. 4.1]) We will prove that there is a computation that computes F with no divisions and $C^{*/}(F)$ multiplications.

Since zero is a linear combination of any set of numbers, we assume without loss of generality that $0 \notin F$. Let β be an optimal computation for F (in the Ostrowski model), and let L be the length of β.

Recall that $H_j(g)$ is the *homogeneous part* of degree j of g. Note that by the allowed operations in our computational structure, we can write $w_i = g_i/h_i$ for all $1 \leq i \leq L$, where $g_i, h_i \in K[x_1, \ldots, x_N]$ and $h_i \neq 0$. We would first like to achieve in some way that $H_0(h_i) \neq 0$ for all i.

Let $\alpha_1, \ldots, \alpha_N \in K$ be constants whose values are yet to be determined. Define the polynomials $\bar{g}_i, \bar{h}_i \in K[x_1, \ldots, x_N]$ (for each i) given by $\bar{g}_i(x_1, \ldots, x_N) = g_i(x_1 + \alpha_1, \ldots, x_N + \alpha_N)$ and $\bar{h}_i(x_1, \ldots, x_N) = h_i(x_1 + \alpha_1, \ldots, x_N + \alpha_N)$. Let $\bar{w}_i = \bar{g}_i/\bar{h}_i$. Replace all values in the computation β (the "w_i") by the values $\bar{w}_i$; the result, call it $\bar{\beta}$, is still a computation since the inputs have been substituted in the entire computation. However, this new computation $\bar{\beta}$ has inputs $\bar{X} \coloneqq \{x_1 + \alpha_1, \ldots, x_N + \alpha_N\}$. But since we can calculate $\bar{X}$ from X without cost (with a few additions) it suffices to achieve our goal (eliminating divisions) using $\bar{X}$ as input set instead.

β computed $F = \{f_1, \ldots, f_k\}$ from X with $f_\kappa = \sum_{\mu,\nu=1}^{N} t_{\mu\nu\kappa} x_\mu x_\nu$, so $\bar{\beta}$ computes the set $\bar{F} \coloneqq \{\bar{f}_1, \ldots, \bar{f}_k\}$ from $\bar{X}$ with

$$\bar{f}_\kappa \coloneqq \sum_{\mu,\nu=1}^{N} t_{\mu\nu\kappa}(x_\mu + \alpha_\mu)(x_\nu + \alpha_\nu) = f_\kappa + (\text{terms of degree} \leq 1 \text{ in } x_1, \ldots, x_N)$$

Since terms of degree ≤ 1 can be computed without cost in the Ostrowski model by introduction of inputs, introduction of constants, constant-multiplication and addition, f_κ can be computed from $\bar{f}_\kappa$ without cost for each κ. This means that it suffices to achieve our goal (eliminating divisions) using $\bar{F}$ as goal instead, since to the alternative computation that uses no divisions we can just append some steps that calculate F without cost from the computed values $\bar{F}$. So we will now continue to eliminate divisions from $\bar{\beta}$.

So why did we introduce those α_i everywhere in our computation? By Lemma 2.4.1, since K is infinite, we can choose $\alpha_1, \ldots, \alpha_N$ in such a way that $H_0(\bar{h}_i) = H_0(h_i(x_1 + \alpha_1, \ldots, x_N + \alpha_N)) = h_i(\alpha_1, \ldots, \alpha_N) \neq 0$ for all $1 \leq i \leq L$. (Note that L is also the length of $\bar{\beta}$.) Now define $\tilde{g}_i, \tilde{h}_i \in K[x_1, \ldots, x_N][[z]]$ (for each i) given by $\tilde{g}_i(x_1, \ldots, x_N, z) = \bar{g}_i(x_1 z, \ldots, x_N z)$ and

$\tilde{h}_i(x_1, \ldots, x_N, z) = \bar{h}_i(x_1 z, \ldots, x_N z)$. These are of the form:

$$\tilde{g}_i = H_0(\bar{g}_i) + H_1(\bar{g}_i)z + H_2(\bar{g}_i)z^2 + \cdots$$
$$\tilde{h}_i = H_0(\bar{h}_i) + H_1(\bar{h}_i)z + H_2(\bar{h}_i)z^2 + \cdots$$

since the total degree of a single term of either $\tilde{g}_i$ or $\tilde{h}_i$ is exactly the number of x_i's present in that term, which is equal to the exponent of the z in that term. (In fact, these power series are still finite, but that does not matter.) Since we obtained that $H_0(\bar{h}_i) \neq 0$, by Lemma 2.4.2, $\tilde{h}_i$ has an inverse in $K(x_1, \ldots, x_N)[[z]]$ (which contains $K[x_1, \ldots, x_N][[z]]$). This means that we can define $\tilde{w}_i \coloneqq \tilde{g}_i / \tilde{h}_i \in K(x_1, \ldots, x_N)[[z]]$. Note that we do not put these values in a computation.

So we can write $\tilde{w}_i = c_i + c_i' z + c_i'' z^2 + \cdots$ for certain $c_i, c_i', c_i'' \in K(x_1, \ldots, x_N)$. Since we are computing a set of quadratic forms, for the final answers, terms with powers of z greater than 2 do not matter. Note that each $\tilde{w}_i$ either comes from an input $x_j + \alpha_j$ for some j (and is therefore equal to $(x_j + \alpha_j)z$), or is the sum, difference, product, quotient, or constant-multiple of earlier $\tilde{w}_j$. We will now transform our computation $\bar{\beta}$ (still the one over $K(x_1, \ldots, x_N)$) to a new computation, β_3. For each item $(x_j + \alpha_j)$ in $\bar{\beta}$, i.e. a reference to an input, copy that item to β_3 unchanged; for each item $(\bar{w}_i, \varphi_i, j_1^i, \ldots, j_s^i)$ in $\bar{\beta}$, we will insert a number of items calculating, with cost equal to $\dot{\varphi}(\varphi_i)$, the values c_i, c_i' and c_i'' assuming that of the referenced items in $\bar{\beta}$ these three polynomials have already been calculated. Exact management of indices is left to the reader. The result of this is a longer computation β_3 that correctly calculates the first three coefficients of all power series $\tilde{w}_i$, and among those, the first three coefficients of the power series corresponding to $\bar{f}_\kappa$ for $1 \leq \kappa \leq k$. Since evaluating the power series $\tilde{w}_i$ at $z = 1$ gives the original $\bar{w}_i$, and we have have seen that the terms with degree three or higher will be or have become zero by the time the computation reaches $\bar{f}_\kappa$, we can compute $\bar{f}_\kappa$ at the end of β_3 with two additions, which have cost zero. This means that we will have proved that there is a computation that does not use divisions and has the same complexity as the original (optimal) computation that computes the polynomials $\bar{F}$ from $\bar{X}$; we have previously shown that this is sufficient.

To construct these items corresponding to each $(\bar{w}_i, \varphi_i, j_1^i, \ldots, j_s^i)$, we first note the following. For each i, the c_i, c_i' and c_i'' are homogeneous polynomials in the input variables $x_1, \ldots, x_N$ of total degree respectively 0, 1 and 2. (Note that the α_i are just constants, which we have already given a value above.) Regardless of φ_i, c_i and c_i' can always be calculated without cost in the Ostrowski model since they have degree ≤ 1, like earlier with the $\bar{f}_\kappa$. It remains to calculate c_i'' with cost equal to the original φ_i.

So perform the following case analysis on $(\varphi_i, j_1^i, \ldots, j_s^i)$:

- Case $(+, j, k)$. c_i'' can be computed using just addition of the earlier values c_j'' and c_k'', which is free of cost.
- Case $(-, j, k)$. Analogously.
- Case $(\cdot, j, k)$. We have:

$$\begin{aligned}\tilde{w}_i &= (c_j + c_j' z + c_j'' z^2 + \cdots)(c_k + c_k' z + c_k'' z^2) \\ &= c_j c_k + (c_j c_k' + c_j' c_k) z + (c_j'' c_k + c_j' c_k' + c_j c_k'') z^2 + \cdots\end{aligned}$$

  So $c_i'' = c_j'' c_k + c_j' c_k' + c_j c_k''$. Since c_j and c_k are constants in K, the products $c_j'' c_k$ and $c_j c_k''$ are just constant-multiplications, which are free of cost, just like additions; this means that the only costing operation is the multiplication $c_j' c_k'$, which has cost 1, like the original operation.

- Case $(/, j, k)$. We have, using Lemma 2.4.2:

$$\begin{aligned}
\tilde{w}_i &= \frac{c_j + c'_j z + c''_j z^2 + \cdots}{c_k + c'_k z + c''_k z^2 + \cdots} \\
&= \left(c_j + c'_j z + c''_j z^2 + o(z^3)\right) \frac{1}{c_k} \left(1 - \left(\frac{c'_k}{c_k} z + \frac{c''_k}{c_k} z^2 + o(z^3)\right) + \left(\frac{c'_k}{c_k} z + o(z^2)\right)^2 + o(z^3)\right) \\
&= \left(c_j + c'_j z + c''_j z^2 + o(z^3)\right) \left(\frac{1}{c_k} - \frac{c'_k}{c_k^2} z + \left(-\frac{c''_k}{c_k^2} + \frac{(c'_k)^2}{c_k^3}\right) z^2 + o(z^3)\right) \\
&= \frac{c_j}{c_k} + \left(\frac{c'_j}{c_k} - \frac{c_j c'_k}{c_k^2}\right) z + \left(\frac{c''_j}{c_k} - \frac{c'_j c'_k}{c_k^2} + c_j \left(-\frac{c''_k}{c_k^2} + \frac{(c'_k)^2}{c_k^3}\right)\right) z^2 + o(z^3)
\end{aligned}$$

  So $c''_i = \frac{1}{c_k} c''_j - \frac{1}{c_k^2} c'_j c'_k - \frac{c_j}{c_k^2} c''_k + \frac{c_j}{c_k^3} c'_k c'_k = \frac{1}{c_k} c''_j - \frac{c_j}{c_k^2} c''_k + \left(-\frac{1}{c_k^2} c'_j + \frac{c_j}{c_k^3} c'_k\right) \cdot c'_k$, which can be calculated with a number of constant-multiplications (since expressions consisting of c_j and c_k are known elements of K), additions, and subtractions, and one actual multiplication (indicated with "$\cdot$"). This means that the resulting computation has cost 1, which is equal to the original cost 1 of the division.

- Case $(\lambda\cdot, j)$. Three constant-multiplications cost the same as one constant-multiplication, i.e. zero.

By the argument before the case analysis, this proves the theorem. □

Theorem 2.4.3 gives us that $C^*(F) = C^{*/}(F)$. Clearly we have $C^*(F) \geq C^{*/}(F)$, so the interesting statement is that $C^*(F) \leq C^{*/}(F)$, i.e. divisions do not help to give a lower cost. Since a field is also a commutative ring, by Theorem 2.3.8 it follows that $C^{*/}(F) = C^*(F) = R_{\text{quad}}(F)$. And if F was actually a set of *bilinear* forms, we can use Lemma 2.3.7 to give the following.

Corollary 2.4.4. For F a set of bilinear forms as in Definition 2.3.6, $C^{*/}(F) \leq R(F) \leq 2C^{*/}(F)$.

Proof. Clear from the above. □

2.5 Tensors & Rank

We have now found that the rank of a set of bilinear forms is related to the complexity of that set: $C^*(F) \leq R(F) \leq 2C^*(F)$ if F is a set of bilinear forms as in Definition 2.3.6. In the previous section we have seen that in addition, if R is actually a field and we also allow divisions (with cost 1), the complexity is still bounded within a factor 2 about $R(F)$. So proving bounds on the rank of a set of bilinear forms is a good way of proving bounds on the complexity of that set (since constant factors disappear in big-O notation).

But we would like to work towards tensors only and forget about the bilinear forms that they correspond to. To start in that direction we will define the rank of a tensor and connect that to the rank of a set of bilinear forms.

Earlier, we defined a *tensor* to be a three-dimensional array of numbers. Most occurrences of tensors will be three-dimensional here, but it is worth knowing that there is nothing special about the number three.

Definition 2.5.1 (Tensor). An *n-tensor*, or simply *tensor*, over a ring R is an n-dimensional array of elements of R. Its size is given by a vector in $\mathbb{N}^n$, say $(s_1, \ldots, s_n)$. Then the tensor is an element of $R^{s_1 \times s_2 \times \cdots \times s_n}$.

For example, any $n \times m$ matrix is also a 2-tensor with size vector (n, m). If t and t' are tensors with the same size vector, they can be added $(t + t')$ and multiplied $(t \cdot t')$ elementwise, but tensors also have natural non-elementwise addition and multiplication operators.

Definition 2.5.2. The *tensor sum* of two n-tensors t (sizes $(s_1, \ldots, s_n)$) and t' (sizes $(s'_1, \ldots, s'_n)$) is the n-tensor $t \oplus t'$ (sizes $(s_1 + s'_1, \ldots, s_n + s'_n)$), given by:

$$(t \oplus t')_{i_1 \ldots i_n} = \begin{cases} t_{i_1 \ldots i_n}, & i_1 \leq s_1 \wedge \cdots \wedge i_n \leq s_n, \\ t'_{(i_1 - s_1) \ldots (i_n - s_n)}, & i_1 > s_1 \wedge \cdots \wedge i_n > s_n, \\ 0, & \text{otherwise.} \end{cases}$$

The *tensor product* of an n-tensor t (sizes $(s_1, \ldots, s_n)$) and an m-tensor t' (sizes $(s'_1, \ldots, s'_m)$) is the $(n+m)$-tensor $t \otimes t'$ (sizes $(s_1, \ldots, s_n, s'_1, \ldots, s'_n)$), given by $(t \otimes t')_{i_1 \ldots i_n j_1 \ldots j_m} = t_{i_1 \ldots i_n} t'_{j_1 \ldots j_m}$.

For $m \geq 0$, the m'th *tensor power* of a tensor t is $t^{\otimes m} = \bigotimes_{i=1}^{m} t$.

The tensor sum is most easily understood by visualising t and t' as blocks which are placed corner-to-corner in an n-dimensional hypercube.

According to the above definition, the tensor product of two 3-tensors is a 6-tensor; in practice, however, we will also regard it as a 3-tensor by "merging" the corresponding indices. Effectively, we identify them via the bijection that sends the 6-tensor t, sizes $(s_1, \ldots, s_6)$, to the 3-tensor $\bar{t}$, sizes $(s_1 s_4, s_2 s_5, s_3 s_6)$, given by $\bar{t}_{((i_4-1)s_1+i_1)((i_5-1)s_2+i_2)((i_6-1)s_3+i_3)} = t_{i_1 i_2 i_3 i_4 i_5 i_6}$ (with the natural ranges for the indices). This new 3-tensor $\bar{t}$ we can index using *double indices*, i.e. use $\bar{t}_{\kappa\kappa', \mu\mu', \nu\nu'} = t_{\kappa\mu\nu\kappa'\mu'\nu'}$.

Definition 2.5.3. The *rank* of an n-tensor t, written $R(t)$, is the least number $r \in \mathbb{N}$ such that there exist 1-tensors (i.e. vectors) $u_{i,j}$ $(1 \leq i \leq r,\ 1 \leq j \leq n)$ such that $t = \sum_{i=1}^{r} \bigotimes_{j=1}^{n} u_{i,j}$.

In the special case of 3-tensors, this expression becomes $t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$ for 1-tensors u_i, v_i, w_i. Note that the tensor $u \otimes v \otimes w$ is given by $(u \otimes v \otimes w)_{ijk} = u_i v_j w_k$.

For matrices, we already knew a concept of rank, and fortunately, the definition given in Definition 2.5.3 (for $n = 2$) agrees: the rank of a matrix is indeed exactly the minimal number of rank-one matrices needed to produce (when added elementwise) the original matrix. In the world of 3-tensors, we now have a rank of a set of bilinear forms as well as of the corresponding tensor. Luckily, these are also equal, as the following theorem shows.

Theorem 2.5.4. For a set of bilinear forms F as in Definition 2.3.6 and the corresponding tensor t, we have $R(F) = R(t)$.

Proof. [3, below Def. 4.4] We will prove that $R(F) \leq r \ \Leftrightarrow\ R(t) \leq r$ for all r. Indeed, $R(F) \leq r$ is equivalent to the existence of linear forms $u_1, \ldots, u_r$ in the variables $x_1, \ldots, x_m$ and linear forms $v_1, \ldots, v_r$ in the variables $y_1, \ldots, y_n$ such that $F \subseteq \operatorname{lin}\{u_1 v_1, \ldots, u_r v_r\}$. But this is equivalent to the existence of coefficients $w_{\kappa,i}$, u_{i,j_1} and v_{i,j_2} $(1 \leq \kappa \leq k,\ 1 \leq i \leq r,\ 1 \leq j_1 \leq m,\ 1 \leq j_2 \leq n)$ such that $f_\kappa = \sum_{i=1}^{r} w_{\kappa,i} \big(\sum_{j_1}^{m} u_{i,j_1} x_{j_1}\big) \big(\sum_{j_2}^{n} v_{i,j_2} y_{j_2}\big)$.

Since $f_\kappa = \sum_{j_1=1}^{m} \sum_{j_2=1}^{n} t_{j_1 j_2 \kappa} x_{j_1} y_{j_2}$, the previous is again equivalent to the statement $t_{j_1 j_2 \kappa} = \sum_{i=1}^{r} w_{\kappa,i} u_{i,j_1} v_{i,j_2}$. This is equivalent to saying that there exist vectors w_i (length k), u_i (length m) and v_i (length n) such that $t_{j_1 j_2 \kappa} = \sum_{i=1}^{r} (w_i)_\kappa (u_i)_{j_1} (v_i)_{j_2}$, which is to say that $t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$. But the existence of such vectors (1-tensors) is equivalent to $R(t) \leq r$, as required. $\square$

Note that using the above proof, one can manually do the actual translation between sets of bilinear forms and their corresponding tensors. Given e.g. the standard formulation of Strassen's matrix multiplication algorithm [24], which can be written as a set of bilinear forms, it is fairly easy to construct the corresponding tensor and its decomposition.

2.5.1 Tensor Properties

We will analyse the behaviour of tensor rank under some transformations of the tensor. Let S_3 be the symmetric group over 3 elements.

Definition 2.5.5. For a 3-tensor $t \in R^{n_1 \times n_2 \times n_3}$ and for any $\pi \in S_3$, define the 3-tensor $\pi t \in R^{n_{\pi^{-1}(1)} \times n_{\pi^{-1}(2)} \times n_{\pi^{-1}(3)}}$ by $(\pi t)_{i_1 i_2 i_3} \coloneqq t_{i_{\pi(1)} i_{\pi(2)} i_{\pi(3)}}$. In other words, permute the indices using π.

Example 2.5.6. Let $t \in R^{k \times m \times n}$ and $\pi = (1\,2\,3)$. Then $\pi t \in R^{n \times k \times m}$ given by $(\pi t)_{i_1 i_2 i_3} = t_{i_2 i_3 i_1}$. Indeed, the ranges $1 \leq i_2 \leq k$, $1 \leq i_3 \leq m$ and $1 \leq i_1 \leq n$ are valid for both t and πt in last equation. Then if $t = u^1 \otimes u^2 \otimes u^3$, we get $(\pi t)_{i_1 i_2 i_3} = t_{i_2 i_3 i_1} = u^1_{i_2} u^2_{i_3} u^3_{i_1} = u^3_{i_1} u^1_{i_2} u^2_{i_3}$, so $\pi t = u^3 \otimes u^1 \otimes u^2$. In general for $\tau \in S_3$, we have $\tau t = u^{\tau^{-1}(1)} \otimes u^{\tau^{-1}(2)} \otimes u^{\tau^{-1}(3)}$.

Note that for any $k, m, n \geq 0$, this gives a left group action of S_3 on the space $R^{k \times m \times n}$ of 3-tensors, since the group elements are really permutations on the set of 3 "index slices" of the tensor. In the next lemma we see that the rank is invariant under this group action.

Lemma 2.5.7. For any $\pi \in S_3$ and any 3-tensor t, we have $R(t) = R(\pi t)$.

Proof. We prove only $R(\pi t) \leq R(t)$, since equality then follows using π^{-1}.

By definition of rank, we have $t = \sum_{j=1}^{R(t)} u_{1,j} \otimes u_{2,j} \otimes u_{3,j}$ for certain vectors $u_{\ell,j}$. But then $\pi t = \sum_{j=1}^{R(t)} u_{\pi^{-1}(1),j} \otimes u_{\pi^{-1}(2),j} \otimes u_{\pi^{-1}(3),j}$, so indeed $R(\pi t) \leq R(t)$. □

The tensor product $u \otimes v \otimes w$ of three vectors u, v, w is called a *triad*.

Definition 2.5.8. For a triad $u \otimes v \otimes w \in R^{k \times m \times n}$ and homomorphisms $A : R^k \to R^{k'}$, $B : R^m \to R^{m'}$ and $C : R^n \to R^{n'}$, define $(A \otimes B \otimes C)(u \otimes v \otimes w) = A(u) \otimes B(v) \otimes C(w)$. For a 3-tensor $t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$, define $(A \otimes B \otimes C)t \coloneqq \sum_{i=1}^{r} A(u_i) \otimes B(v_i) \otimes C(w_i)$.

Lemma 2.5.9. "$(A \otimes B \otimes C)t \coloneqq \sum_{i=1}^{r} A(u_i) \otimes B(v_i) \otimes C(w_i)$" in Definition 2.5.8 is independent of decomposition, so Definition 2.5.8 is valid.

Proof. Observe that any triad $u \otimes v \otimes w \in R^{k \times m \times n}$ can be written with respect to the standard basis as:

$$\begin{pmatrix} u_1 \\ \vdots \\ u_k \end{pmatrix} \otimes \begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix} \otimes \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} = (u_1 e_1 + \cdots + u_k e_k) \otimes (v_1 e_1 + \cdots + v_m e_m) \otimes (w_1 e_1 + \cdots + w_n e_n)$$

$$= \sum_{a=1}^{k} \sum_{b=1}^{m} \sum_{c=1}^{n} u_a v_b w_c \; e_a \otimes e_b \otimes e_c$$

Since A, B, C are homomorphisms, we must have $\sum_{a=1}^{k} \sum_{b=1}^{m} \sum_{c=1}^{n} u_a v_b w_c \, A(e_a) \otimes B(e_b) \otimes C(e_c) = A(u) \otimes B(v) \otimes C(w)$.

Suppose t is written $t = \sum_{a,b,c} \lambda_{a,b,c} \; e_a \otimes e_b \otimes e_c$ against the standard basis, and suppose $t = \sum_{i=1}^{r} u_i \otimes v_i \otimes w_i$ is an expansion in triads. Then each triad $u_i \otimes v_i \otimes w_i$ can be written against the standard basis; the sum of these basis expansions should be the basis expansion of t by unicity in a vector space. Thus since A, B, C are homomorphisms, taking $(A \otimes B \otimes C)$ of the basis expansion of t yields the same result as taking that of each of the triads and summing the results; since this holds for any expansion of t in triads, we have proved that the definition is independent of the decomposition. □

Lemma 2.5.10. For all homomorphisms $A : R^k \to R^{k'}$, $B : R^m \to R^{m'}$ and $C : R^n \to R^{n'}$ and $t \in R^{k\times m\times n}$, we have $R\big((A \otimes B \otimes C)t\big) \leq R(t)$. If A, B, C are isomorphisms, we have equality.

Proof. By definition, if $R(t) \leq r$ then $R\big((A \otimes B \otimes C)t\big) \leq r$, so $R\big((A \otimes B \otimes C)t\big) \leq R(t)$.

If A, B and C are invertible, we have $(A^{-1} \otimes B^{-1} \otimes C^{-1})(A \otimes B \otimes C)t = t$, and thus $R(t) = R\big((A^{-1} \otimes B^{-1} \otimes C^{-1})(A \otimes B \otimes C)t\big) \leq R\big((A \otimes B \otimes C)t\big) \leq R(t)$, so we have equality. □

Using these two operations, we can now show that matrix multiplication does not get any harder when we swap sizes around. Recall the notation $\langle k, m, n\rangle$ from Section 2.3.

Lemma 2.5.11. For any $k, m, n \geq 1$, we have $R(\langle k, m, n\rangle) = R(\langle n, k, m\rangle)$.

Proof. We use double indices to index $t \coloneqq \langle k, m, n\rangle$, so t ranges from $t_{11,11,11}$ to $t_{km,mn,kn}$. From Remark 2.3.2 we know that $t_{ii',jj',\ell\ell'} = \delta_{i'j}\delta_{i\ell}\delta_{j'\ell'}$. By permuting the basis vectors of R^{kn}, we can swap the indices in the "third slice" of t to produce t', so that $t'_{ii',jj',\ell'\ell} = t_{ii',jj',\ell\ell'}$. This operation is an automorphism, so $R(t) = R(t')$ by Lemma 2.5.10.

Then use $\pi = (1\,2\,3) \in S_3$ to permute t' to $t'' \coloneqq \pi t'$. Then we have $t''_{ii',jj',\ell'\ell} = t'_{jj',\ell'\ell,ii'}$ (compare Example 2.5.6) and $R(t'') = R(t')$ by Lemma 2.5.7. Again swapping the indices of the third slice of t'' to produce t''' (again $R(t''') = R(t'')$), we get:

$$t'''_{ii',jj',\ell\ell'} = t''_{ii',jj',\ell'\ell} = t'_{jj',\ell'\ell,ii'} = t_{jj',\ell'\ell,i'i} = \delta_{j'\ell'}\delta_{ji'}\delta_{\ell i} = \delta_{i'j}\delta_{i\ell}\delta_{j'\ell'}$$

Since $t \in K^{(k\times m)\times(m\times n)\times(k\times n)}$, we have $t''' \in K^{(n\times k)\times(k\times m)\times(n\times m)}$, which is the same space $\langle n, k, m\rangle$ is an element of. Since the coordinates of t''' and $\langle n, k, m\rangle$ are given by the same expression with Kronecker deltas, we have $t''' = \langle n, k, m\rangle$ and therefore $R(\langle n, k, m\rangle) = R(\langle k, m, n\rangle)$. □

Lemma 2.5.12. For any $k, m, n \geq 1$, we have $R(\langle k, m, n\rangle) = R(\langle n, m, k\rangle)$.

Proof. Again use double indices and $t \coloneqq \langle k, m, n\rangle$, and $t_{ii',jj',\ell\ell'} = \delta_{i'j}\delta_{i\ell}\delta_{j'\ell'}$. Apply the permutation $(1\,2)$ to obtain t' given by $t'_{ii',jj',\ell\ell'} = t_{jj',ii',\ell\ell'}$. Then swap indices in each slice to obtain t'' given by $t''_{ii',jj',\ell\ell'} = t'_{i'i,j'j,\ell'\ell} = t_{j'j,i'i,\ell'\ell} = \delta_{ji'}\delta_{j'\ell'}\delta_{i\ell} = \delta_{i'j}\delta_{i\ell}\delta_{j'\ell'}$. Since $t \in K^{(k\times m)\times(m\times n)\times(k\times n)}$, we have $t'' \in K^{(n\times m)\times(m\times k)\times(n\times k)}$, so $t'' = \langle n, m, k\rangle$. Therefore, $R(\langle n, m, k\rangle) = R(\langle k, m, n\rangle)$. □

Corollary 2.5.13. For any $k, m, n \geq 1$, permuting matrix sizes does not influence the rank:

$$R(\langle k, m, n\rangle) = R(\langle k, n, m\rangle) = R(\langle m, k, n\rangle) = R(\langle m, n, k\rangle) = R(\langle n, k, m\rangle) = R(\langle n, m, k\rangle)$$

Proof. [3, Lem. 5.5] Follows from Lemma 2.5.11 and Lemma 2.5.12. □

2.5.2 Rank Properties

The easiest form of matrix tensor rank to look at is $R(\langle n, n, n\rangle)$ for some n. This is what we will base most of our work on, but besides it being nice to also be able to multiply non-square matrices, later algorithms will also not necessarily be for square matrices. But note that we can combine matrix tensors into a larger matrix tensor as follows:

Lemma 2.5.14. We have $\langle k, m, n\rangle \otimes \langle k', m', n'\rangle = \langle kk', mm', nn'\rangle$, where we see the tensor product of two 3-tensors as a 3-tensor again, as explained below Definition 2.5.2.

Proof. Use indices $\langle k, m, n\rangle_{\kappa\bar{\kappa},\mu\bar{\mu},\nu\bar{\nu}}$ and $\langle k', m', n'\rangle_{\kappa'\bar{\kappa}',\mu'\bar{\mu}',\nu'\bar{\nu}'}$. We have:

$$\begin{aligned}(\langle k, m, n\rangle \otimes \langle k', m', n'\rangle)_{(\kappa\bar{\kappa},\kappa'\bar{\kappa}'),(\mu\bar{\mu},\mu'\bar{\mu}'),(\nu\bar{\nu},\nu'\bar{\nu}')} &= \delta_{\bar{\kappa}\mu}\delta_{\kappa\nu}\delta_{\bar{\mu}\bar{\nu}}\delta_{\bar{\kappa}'\mu'}\delta_{\kappa'\nu'}\delta_{\bar{\mu}'\bar{\nu}'} \\ &= \delta_{(\bar{\kappa},\bar{\kappa}')(\mu,\mu')}\delta_{(\kappa,\kappa')(\nu,\nu')}\delta_{(\bar{\mu},\bar{\mu}')(\bar{\nu},\bar{\nu}')} = \langle kk', mm', nn'\rangle_{(\kappa\kappa',\bar{\kappa}\bar{\kappa}'),(\mu\mu',\bar{\mu}\bar{\mu}'),(\nu\nu',\bar{\nu}\bar{\nu}')}\end{aligned}$$

Because the sizes also coincide, the lemma follows. □

So if we could say something about the rank of a tensor product in terms of the ranks of the arguments, we would be able to deduce statements about the complexity of square matrix multiplication from algorithms for specific, non-square matrix sizes.

Lemma 2.5.15. For 3-tensors t and t', we have $R(t \otimes t') \leq R(t)R(t')$.

Proof. [3, Lem. 5.8] For $r = R(t)$ and $r' = R(t')$, we have the expansions $t = \sum_{p=1}^{r} u_p \otimes v_p \otimes w_p$ and $t' = \sum_{p'=1}^{r'} u'_{p'} \otimes v'_{p'} \otimes w'_{p'}$ for certain $u_{p'} \in R^k$ etc. Now for each $1 \leq p \leq r$ and $1 \leq p' \leq r'$, we define $\bar{u}_{pp'} \in R^{kk'}$, $\bar{v}_{pp'} \in R^{mm'}$ and $\bar{w}_{pp'} \in R^{nn'}$ given by $\bar{u}_{pp',ii'} = u_{pi}u'_{p'i'}$, $\bar{v}_{pp',jj'} = v_{pj}v'_{p'j'}$ and $\bar{w}_{pp',\ell\ell'} = w_{p\ell}w'_{p'\ell'}$. Enumerate $t \otimes t'$ with double indices as explained under Definition 2.5.2. Then:

$$\begin{aligned}(t \otimes t')_{ii',jj',\ell\ell'} &= \left(\sum_{p=1}^{r} u_{pi}v_{pj}w_{p\ell}\right)\left(\sum_{p'=1}^{r'} u'_{p'i'}v'_{p'j'}w'_{p'\ell'}\right) \\ &= \sum_{p=1}^{r}\sum_{p'=1}^{r'} \bar{u}_{pp',ii'}\bar{v}_{pp',jj'}\bar{w}_{pp',\ell\ell'} = \left(\sum_{p=1}^{r}\sum_{p'=1}^{r'} \bar{u}_{pp'} \otimes \bar{v}_{pp'} \otimes \bar{w}_{pp'}\right)_{ii',jj',\ell\ell'}\end{aligned}$$

So we have $R(t \otimes t') \leq rr' = R(t)R(t')$. □

A similar statement holds for the tensor sum.

Lemma 2.5.16. For 3-tensors t and t', we have $R(t \oplus t') \leq R(t) + R(t')$.

Proof. [3, Lem. 5.6] For $r = R(t)$ and $r' = R(t')$, we have the expansions $t = \sum_{p=1}^{r} u_p \otimes v_p \otimes w_p$ and $t' = \sum_{p=1}^{r'} u'_p \otimes v'_p \otimes w'_p$ for certain $u_p \in R^k$ etc. Define $\bar{u}_p = (u_{p1}, \ldots, u_{pk}, 0, \ldots, 0) \in R^{k+k'}$ and $\bar{u}'_p = (0, \ldots, 0, u'_{p1}, \ldots, u'_{pk'}) \in R^{k+k'}$ for each p, and $\bar{v}, \bar{v}', \bar{w}, \bar{w}'$ analogously. Then we have $(t \oplus t')_{ij\ell} = \sum_{p=1}^{r} \bar{u}_{pi}\bar{v}_{pj}\bar{w}_{p\ell} + \sum_{p=1}^{r'} \bar{u}'_{pi}\bar{v}'_{pk}\bar{w}'_{p\ell}$, which proves the lemma as before. □

Armed with this knowledge, we can now define the matrix multiplication exponent and begin proving results about it.

2.6 Matrix Multiplication Exponent

Recall from Definition 2.3.1 that the matrix multiplication tensor $\langle k, m, n\rangle$ is the tensor $t \in R^{kn \times km \times mn}$ such that, if we enumerate its indices with pairs of numbers, the tensor t has 0's everywhere except for 1's at the coordinates with indices of the form $(\kappa, \mu), (\mu, \nu), (\kappa, \nu)$ for $1 \leq \kappa \leq k$, $1 \leq \mu \leq m$, $1 \leq \nu \leq n$. It is the tensor corresponding to the set of bilinear forms making up matrix multiplication of a $k \times m$ matrix with an $m \times n$ matrix.

In Corollary 2.3.9 (or Corollary 2.4.4 if you like fields) and Theorem 2.5.4, we made a connection between the rank of a tensor t and the modified Ostrowski complexity of its corresponding set F of bilinear forms: $C^*(F) \leq R(F) = R(t) \leq 2C^*(F)$. This means that if we want to bound the complexity of matrix multiplication, it suffices to bound the rank of the corresponding tensor. In the future, we will be concerned with proving upper bounds on $R(\langle k, m, n\rangle)$ to get upper bounds on complexity of matrix multiplication. To do that, however, we sometimes need the following *lower* bound, which on first sight may be trivial, but has a reasonably technical proof.

We first prove the result if R is actually a field K, and then use a theorem about maximal ideals to lift the result to commutative rings.

Lemma 2.6.1. If the commutative ring R is a field K and $n \geq 2$, then $R(\langle n,n,n\rangle) > n^2$.

Proof. We first prove $(\star)$: if $S,T \subseteq \mathrm{Mat}(n,K)$, $\{MN : M \in S, N \in T\} = \mathrm{Mat}(n,K)$, and F is a set of bilinear forms that, when evaluated on two matrices $M \in S$ and $N \in T$, gives the entries of MN, then $R(F) \geq n^2$.

Suppose that $R(F) \leq n^2 - 1$. By Definition 2.3.6, we then have products $P_1, \ldots, P_{n^2-1}$ ($P_i = \left(\sum_{j=1}^{n^2} u_{i,j}x_j\right)\left(\sum_{j=1}^{n^2} v_{i,j}y_j\right)$) and coefficients $w_{i,j} \in K$ such that if $(M,N) \in S \times T$ and x and y are respectively the entries of M and N, then $z_i = \sum_{j=1}^{n^2-1} w_{i,j}P_j$ are the entries of MN. In the vector space K^{n^2-1}, the n^2 vectors $(w_{1,1}, \ldots, w_{1,n^2-1}), \ldots, (w_{n^2,1}, \ldots, w_{n^2,n^2-1})$ must be linearly dependent, so there are coefficients $k_1, \ldots, k_{n^2} \in K$ (not all zero) such that for all j, we have $\sum_{i=1}^{n^2} k_i w_{i,j} = 0$. Since $\{MN : M \in S, N \in T\} = \mathrm{Mat}(n,K)$, the entries $z_1, \ldots, z_{n^2}$ of any matrix $Z \in \mathrm{Mat}(n,K)$ will then obey $\sum_{i=1}^{n^2} k_i z_i = \sum_{i=1}^{n^2} k_i \sum_{j=1}^{n^2-1} w_{i,j}P_j = \sum_{j=1}^{n^2-1} P_j \sum_{i=1}^{n^2} k_i w_{i,j} = 0$. Since not all k_i are zero, there is a matrix in $\mathrm{Mat}(n,K)$ such that this linear dependency is *not* fulfilled, which is a contradiction. The claim follows.

If we apply $(\star)$ to $S = T = \mathrm{Mat}(n,K)$ and take for F the set corresponding to $\langle n,n,n\rangle$, by Theorem 2.5.4, we get $R(\langle n,n,n\rangle) \geq n^2$.

Then assume that $R(\langle n,n,n\rangle) = n^2$. Then the elements of the result matrix are linear combinations of products $P_1, \ldots, P_{n^2}$ of the form $P_i = \left(\sum_{j=1}^{m_{i,1}} u_{i,j}a_{k_{i,j}}\right)\left(\sum_{j=1}^{m_{i,2}} v_{i,j}b_{\ell_{i,j}}\right)$ (where $a_{1\ldots n^2}$ and $b_{1\ldots n^2}$ are the coefficients of the two input matrices, respectively) for certain coefficients $u_{i,j}, v_{i,j} \in K$ that are *all nonzero* (for zero coefficients, just remove them from the list and decrease $m_{i,1}$ or $m_{i,2}$ appropriately) and for indices $k_{i,j}$ and $\ell_{i,j}$ such that $k_{i,j_1} = k_{i,j_2} \Leftrightarrow j_1 = j_2$ and $\ell_{i,j_1} = \ell_{i,j_2} \Leftrightarrow j_2 = j_2$. (This last requirement just ensures that a single linear combination only adds distinct inputs.)

- If any value $m_{i,1}$ is zero, we may discard that product and use $(\star)$ to arrive at a contradiction.
- If all values $m_{i,1}$ are one, then $P_i = u_{i,1}a_{k_{i,1}}\left(\sum_{j=1}^{m_{i,2}} v_{i,j}b_{\ell_{i,j}}\right)$ for all i. So for some i, $k_{i,1}$ must indicate an element off the main diagonal, since each entry in the left input matrix should appear in some product and since $n \geq 2$. Taking the identity matrix as left argument to the matrix multiplication then makes $a_{k_{i,1}} = 0$, discarding product i; but even with the identity matrix as one argument, matrix multiplication is still surjective, so $(\star)$ may again be used to arrive at a contradiction.
- Otherwise, there is an i such that $m_{i,1} \geq 2$. We claim there is an invertible matrix A such that $a_{k_{i,1}} = -\sum_{j=2}^{m_{i,1}} u_{i,j}a_{k_{i,j}} =: \alpha$. If there is, matrix multiplication with the restriction that the left argument to the i'th product is zero is still surjective (since for any matrix C, we have $A \cdot A^{-1}C = C$); but then again $(\star)$ applies, so we arrive at a contradiction.

  Suppose $k_{i,1}$ is off the main diagonal. Then the identity matrix with $a_{k_{i,1}}$ replaced by α is row-equivalent to the identity matrix, and is therefore invertible. Otherwise, $k_{i,1}$ is on the main diagonal. If $(e_1, \ldots, e_n)$ is the standard basis of K^n, consider the matrix B with rows $e_2, \ldots, e_n, e_1$. This matrix is invertible, and since $n \geq 2$, the entry $a_{k_{i,1}}$ in B is off the "shifted diagonal", so replacing it with α yields a matrix that is row-equivalent to B and thus invertible.

In each case we arrive at a contradiction, so together with $(\star)$, we have $R(\langle n,n,n\rangle) > n^2$. $\square$

Theorem 2.6.2. If $n \geq 2$, then $R(\langle n, n, n \rangle) > n^2$.

Proof. Since R is a ring not equal to $\{0\}$, it has a *maximal ideal*, say $I \subseteq R$. Since R is in fact a commutative ring, the quotient $K \coloneqq R/I$ is a field. Now suppose that $R(\langle n, n, n \rangle) \leq n^2$. Then the at most n^2 bilinear products that should then produce the product of two matrices should also produce the product of two matrices over $K = R/I$, since the canonical map $R \to K$ is a ring homomorphism (and all our operations are respected by ring homomorphisms). Therefore, we would conclude that we also have $R(\langle n, n, n \rangle) \leq n^2$ over the field K, which is a contradiction with Lemma 2.6.1. The theorem follows. □

The rank of a matrix multiplication tensor is what we will be looking at; in particular, we will focus on proving upper bounds for the following value.

Definition 2.6.3 (Matrix Multiplication Exponent). For a commutative ring R, the exponent of matrix multiplication is defined as $\omega_R \coloneqq \inf\{\beta : (n \mapsto R(\langle n, n, n \rangle)) \in O(n^\beta)\}$, where $\langle n, n, n \rangle \in R^{n^2 \times n^2 \times n^2}$ is a matrix multiplication tensor over R.

A priori, this value ω_R is dependent on the chosen commutative ring R. However, the theorems providing upper bounds (e.g. Theorem 2.6.6) and lower bounds (Theorem 2.6.2) hold for all commutative rings R. Since R is a fixed (but arbitrary) commutative ring in this thesis, we will leave out the subscript and simply write ω.

As definition of the big-O notation here, we use that $f(n) \in O(g(n))$ iff $\limsup_{n \to \infty} \frac{|f(n)|}{g(n)} < \infty$. In the case of ω, $(n \mapsto R(\langle n, n, n \rangle))$ is positive for $n \geq 1$, so we can drop the absolute value function. Note that the function $(n \mapsto R(\langle n, n, n \rangle))$ is also increasing, i.e. $R(\langle n, n, n \rangle) \leq R(\langle n+1, n+1, n+1 \rangle)$, since we can pad smaller matrices with zeros to produce matrices suitable for larger multiplication algorithms. We will use this later.

Note that the complexity of a specific instance of matrix multiplication and the rank of the corresponding tensor are within a factor 2 apart, so since constants disappear in big-O notation (i.e. in the outcome of the limit for which we only require finiteness), usage of either would give the same ω. Since we want to work with tensors, we choose the rank.

By Theorem 2.6.2 we know that $\omega \geq 2$, and since the naive algorithm has complexity $O(n^3)$, we know that $\omega \in [2, 3]$. Strassen's algorithm already gave a nontrivial upper bound of $\log_2(7) \approx 2.81$ on ω, and better upper bounds have been proved; at the time of writing, the best result is $\omega < 2.3728639$ [13].

The fact that ω is defined as an infimum means that if we prove that $\omega \leq \beta$ for some $\beta \in \mathbb{R}$, we do not necessarily have an algorithm for matrix multiplication with $O(n^\beta)$ multiplications; we do, however, know that for each $\varepsilon > 0$, there is a sequence of algorithms that perform $n \times n$ matrix multiplication for each $n \geq 1$ and for which the number of multiplications required is in $O(n^{\beta+\varepsilon})$. This is an illustration of the fact that this construction is a theoretical exercise, rather than a way to practically speed up matrix multiplication.

But to at least instill a bit more confidence in this ω, which only considers multiplications, we will now show that also giving addition, subtraction and constant-multiplication cost 1 does not change the exponent.

Recall cost from Definition 2.3.3. Let $C^{\text{all}}(F)$ denote the cost of the set of bilinear forms F with the usual M and Φ, but with $¢(\pm) = ¢(\cdot) = ¢(\lambda\cdot) = 1$.

Theorem 2.6.4. Let $\tilde{\omega} \coloneqq \inf\{\beta : (n \mapsto C^{\text{all}}(F_n)) \in O(n^\beta)\}$, where F_n is the set of bilinear forms corresponding to $\langle n, n, n\rangle$. Then $\omega = \tilde{\omega}$.

Proof. [3, Th. 5.2] Since also punishing additions and constant-multiplications can only increase the cost, we clearly have $\omega \leq \tilde{\omega}$. For the reverse inequality, we use the existence of an algorithm with $O(n^{\omega+\varepsilon})$ multiplications to give a sequence of algorithms with C^{all}-cost $O(n^{\omega+\varepsilon+\delta})$ with δ approaching zero. Since we do this for all ε, the theorem will then follow.

So let an $\varepsilon > 0$ be arbitrary. By definition of ω, we have $\forall \tilde{\varepsilon} > 0 : R(\langle n, n, n\rangle) \in O(n^{\omega+\tilde{\varepsilon}})$, so:

$$\forall \tilde{\varepsilon} > 0 : \exists \alpha > 0, m_0 \geq 1 : \forall m \geq m_0 : R(\langle m, m, m\rangle) \leq \alpha m^{\omega+\tilde{\varepsilon}}$$

Let α and m_0 be such that $\forall m \geq m_0 : r \coloneqq R(\langle m, m, m\rangle) \leq \alpha m^{\omega+\varepsilon}$. Now take some arbitrary $m \geq m_0$. Our algorithm fills the input matrices up with zeros until the problem is reduced to $\langle m^i, m^i, m^i\rangle$ for some $i \in \mathbb{N}$, then recursively solves the problem by applying the algorithm with r multiplications. Suppose this input algorithm uses, besides its r multiplications, c additions, subtractions and constant-multiplications. Then for the number of arithmetic operations $A(i)$ used for multiplying two $m^i \times m^i$ matrices, we have the following:

$$A(i) \leq rA(i-1) + c(m^{i-1})^2$$

because the r multiplications involve a recursive call to the algorithm and the c elementwise operations have to be performed with $m^{i-1} \times m^{i-1}$ matrices as elements. Solving this inequality recursively, we get:

$$\begin{aligned}
A(i) &\leq r^i A(0) + r^{i-1}c + r^{i-2}cm^2 + \cdots + cm^{2(i-1)} = r^i A(0) + c\sum_{j=1}^{i} r^{i-j}m^{2(j-1)} \\
&= r^i A(0) + \frac{cr^i}{m^2}\sum_{j=1}^{i}\left(\frac{m^2}{r}\right)^j \overset{(\star)}{=} r^i A(0) + \frac{cr^i}{m^2}\,\frac{m^2}{r}\,\frac{1-\left(\frac{m^2}{r}\right)^i}{1-\frac{m^2}{r}} = r^i A(0) + cr^i\,\frac{1-\frac{m^{2i}}{r^i}}{r-m^2} \\
&= r^i\left(A(0) + \frac{c}{r-m^2}\right) - \frac{c\,m^{2i}}{r-m^2} \overset{(\star\star)}{\leq} r^i \underbrace{\left(A(0) + \frac{c}{r-m^2}\right)}_{\text{constant in } i} \in O(r^i)
\end{aligned}$$

Note that Theorem 2.6.2 gives us that $r > m^2$. This ensures that in the step marked $(\star)$ we do not divide by zero and that the inequality at $(\star\star)$ is valid. (The $O(r^i)$ concerns functions of i as $i \to \infty$.)

Since we can add zeros to matrices to expand their size, we have the inequality $C^{\text{all}}(\langle n, n, n\rangle) \leq C^{\text{all}}(\langle n', n', n'\rangle)$ if $n \leq n'$. So for any $n \geq 1$:

$$\begin{aligned}
C^{\text{all}}(\langle n, n, n\rangle) &\leq C^{\text{all}}\left(\left\langle m^{\lceil \log_m n\rceil}, m^{\lceil \log_m n\rceil}, m^{\lceil \log_m n\rceil}\right\rangle\right) = A(\lceil \log_m n\rceil) \\
&\in O(r^{\lceil \log_m n\rceil}) \subseteq O(r^{1+\log_m n}) = O(r^{\log_m n}) = O(n^{\log_m r})
\end{aligned}$$

(The O-notations concern functions of n as $n \to \infty$.) Since we knew that $r \leq \alpha m^{\omega+\varepsilon}$, we have $C^{\text{all}}(\langle n, n, n\rangle) \in O(n^{\omega+\varepsilon+\log_m \alpha})$, so $\tilde{\omega} \leq \omega + \varepsilon + \log_m \alpha$. Since this holds for all $m \geq m_0$, and $\lim_{m\to\infty} \log_m \alpha = 0$, we have $\tilde{\omega} \leq \omega + \varepsilon$.

The above holds for any $\varepsilon > 0$, so we have $\tilde{\omega} \leq \omega$, which remained to prove. □

Using the machinery in Section 2.5, we can now prove certain upper bounds on ω given algorithms for particular matrix sizes. To complete these proofs, we will use the following lemma.

Lemma 2.6.5. If $N \in \mathbb{Z}_{\geq 2}$, $r \in \mathbb{R}_{>0}$ and $R(\langle N^i, N^i, N^i\rangle) \leq N^{ir}$ for all $i \in \mathbb{Z}_{\geq 1}$, then $\omega \leq r$.

Proof. Assume the contrary, that $\omega > r$. Then $(n \mapsto R(\langle n, n, n\rangle)) \notin O(n^r)$, so we have:

$$\neg\left(\exists \alpha > 0, n_0 \geq 1 : \forall n \geq n_0 : R(\langle n, n, n\rangle) \leq \alpha n^r\right)$$

or in other words:

$$\forall \alpha > 0, n_0 \geq 1 : \exists n \geq n_0 : R(\langle n, n, n\rangle) > \alpha n^r$$

Take $\alpha = N^r$ and $n_0 = N + 1$. Then we have an $n \geq N + 1$ such that $R(\langle n, n, n\rangle) > N^r n^r$. Now choose an $i \geq 1$ such that $N^i < n \leq N^{i+1}$. Then $R(\langle n, n, n\rangle) > N^r n^r > N^r (N^i)^r = N^{(i+1)r}$, but since $n \leq N^{i+1}$ we also have $R(\langle n, n, n\rangle) \leq R(\langle N^{i+1}, N^{i+1}, N^{i+1}\rangle) \leq N^{(i+1)r}$. This is a contradiction, so we conclude that $\omega \leq r$. □

Theorem 2.6.6. We have $\omega \leq 3 \cdot \log_{kmn} R(\langle k, m, n\rangle)$ for all $k, m, n \geq 1$, $kmn \geq 2$. In particular, we also have $\omega \leq \log_n R(\langle n, n, n\rangle)$ for all $n \geq 2$.

Proof. [3, Th. 5.9] We have:

$$\begin{aligned} R(\langle (kmn)^i, (kmn)^i, (kmn)^i\rangle) &= R\big((\langle k, m, n\rangle \otimes \langle m, n, k\rangle \otimes \langle n, k, m\rangle)^{\otimes i}\big) \\ &\leq R(\langle k, m, n\rangle)^i R(\langle m, n, k\rangle)^i R(\langle n, k, m\rangle)^i = R(\langle k, m, n\rangle)^{3i} \end{aligned}$$

where the first equality is Lemma 2.5.14, the inequality is Lemma 2.5.15 and the last equality is Corollary 2.5.13.

So taking $N = kmn$ and $r = R(\langle k, m, n\rangle)$, we have $R(\langle N^i, N^i, N^i\rangle) \leq r^{3i} = N^{3i \log_N r}$; which by Lemma 2.6.5 (since $N \geq 2$) implies that $\omega \leq 3 \log_N r$. The second statement follows since $3 \log_{n^3} x = \log_n x$. □

Example 2.6.7. From the bilinear algorithm given by Strassen [24] we know that $R(\langle 2, 2, 2\rangle) \leq 7$, and Winograd [29] confirmed that $R(\langle 2, 2, 2\rangle) = 7$. Using Theorem 2.6.6, we obtain the bound $\omega \leq \log_2 7 \approx 2.81$, which is the complexity obtained when using Strassen's algorithm recursively on 2×2 block matrices.

2.7 Border Rank

The concept of "rank" of a tensor has let us set up some amount of theory, but it has been found that rank alone will not get us good improvements on the matrix multiplication exponent. We use a peculiar feature of the rank of a 3-tensor that does not appear when working with 2-tensors (more commonly known as "matrices"), which is that the rank of the limit of a sequence of 3-tensors need not equal the limit of the ranks of the tensors.

This means that sometimes we can get arbitrarily close to a certain tensor using r multiplications, while we would need $r + 1$ for the tensor itself. This means that this tensor has rank $r + 1$, but *border rank* r, as we will soon define.

Intuitively, we want to not compute the tensor t, but instead $\varepsilon^h t + o(\varepsilon^{h+1})$ for arbitrarily small ε and a certain constant $h \geq 0$. Of course, this only works in normed fields, but we can define a more general analogue.

Definition 2.7.1. For a 3-tensor $t \in R^{k \times m \times n}$ and an integer $h \geq 0$, define $R_h(t)$ to be the minimal integer $r \geq 0$ such that there exist $u_1, \ldots, u_r \in R[\varepsilon]^k$, $v_1, \ldots, v_r \in R[\varepsilon]^m$, $w_1, \ldots, w_r \in R[\varepsilon]^n$ and a tensor $t' \in R[\varepsilon]^{k \times m \times n}$ for which we have $\sum_{i=1}^r u_i \otimes v_i \otimes w_i = \varepsilon^h t + \varepsilon^{h+1} t'$.

Definition 2.7.2 (Border Rank). For a 3-tensor t, define the *border rank* of t: $\underline{R}(t) \coloneqq \min_h R_h(t)$.

Remark 2.7.3. Some properties of $R_h(t)$ and $\underline{R}(t)$ are evident from their definition.

1. We have $R(t) = R_0(t) \geq R_1(t) \geq \cdots \geq R_h(t) = \underline{R}(t)$ for some h.
2. For $R_h(t)$, it is sufficient to consider u_i, v_i, w_i with degree in ε no higher than h: any higher terms can be safely omitted anyway.
3. We have $\forall \pi \in S_3 : R_h(t) = R_h(\pi t)$ (completely analogous to Lemma 2.5.7)
4. Corollary 2.5.13 extends to R_h, i.e. $R_h(\langle k, m, n\rangle) = \cdots = R_h(\langle n, m, k\rangle)$.

Another property is to be expected, but less obvious.

Lemma 2.7.4. For 3-tensors t and t', $R_{h+h'}(t \otimes t') \leq R_h(t)R_{h'}(t')$.

Proof. [3, Th. 6.3(3)] Suppose there are approximate expansions for t and t' as follows:

$$\sum_{i=1}^{r} u_i \otimes v_i \otimes w_i = \varepsilon^h t + \varepsilon^{h+1} s \qquad \text{and} \qquad \sum_{i=1}^{r'} u'_i \otimes v'_i \otimes w'_i = \varepsilon^{h'} t' + \varepsilon^{h'+1} s'$$

where $t \in R^{k\times m\times n} \subseteq R[\varepsilon]^{k\times m\times n} \ni s$ and $t' \in R^{k'\times m'\times n'} \subseteq R[\varepsilon]^{k'\times m'\times n'} \ni s'$. Using the bilinearity of the tensor product, we can take the tensor product of the two sums above and get as a result an expansion (with $R_h(t)R_{h'}(t')$ items) of $t'' \coloneqq (\varepsilon^h t + \varepsilon^{h+1} s) \otimes (\varepsilon^{h'} t' + \varepsilon^{h'+1} s')$, which has entries:

$$\begin{aligned} t''_{ii',jj',\ell\ell'} &= (\varepsilon^h t_{ij\ell} + \varepsilon^{h+1} s_{ij\ell})(\varepsilon^{h'} t'_{i'j'\ell'} + \varepsilon^{h'+1} s'_{i'j'\ell'}) = \varepsilon^{h+h'} t_{ij\ell} t'_{i'j'\ell'} + \varepsilon^{h+h'+1} s'' \\ &= \varepsilon^{h+h'} (t \otimes t')_{ii',jj',\ell\ell'} + \varepsilon^{h+h'+1} s'' \end{aligned}$$

for some tensor $s'' \in R[\varepsilon]^{kk'\times mm'\times nn}$. So the constructed expansion was actually an approximate expansion for $t \otimes t'$, so we have $R_{h+h'}(t \otimes t') \leq R_h(t)R_{h'}(t')$ as required. □

Fortunately, we can turn approximate computations into actual computations; however, we do need to take into account a possible polynomial growth of the rank.

Lemma 2.7.5. There are constants $c_h \leq \binom{h+2}{2}$ such that for all 3-tensors t, $R(t) \leq c_h R_h(t)$.

Proof. [3, Lem. 6.4] Assume that $\underline{R}(t) = R_h(t) = r$. Then there is an expansion of $t \in R^{k\times m\times n}$ as follows:

$$\sum_{i=1}^{r} \left(\sum_{a=0}^{h} \varepsilon^a u_{ia} \right) \otimes \left(\sum_{b=0}^{h} \varepsilon^b v_{ib} \right) \otimes \left(\sum_{c=0}^{h} \varepsilon^c w_{ic} \right) = \varepsilon^h t + \varepsilon^{h+1} t'$$

(for some tensor $t' \in R[\varepsilon]^{k\times m\times n}$) with $u_{ia} \in R^k$ etc. (It is sufficient to look at powers of ε up to h by Remark 2.7.3(2).)

The left-hand side is equal to the following:

$$\sum_{i=1}^{r} \sum_{a=0}^{h} \sum_{b=0}^{h} \sum_{c=0}^{h} \varepsilon^{a+b+c} u_{ia} \otimes v_{ib} \otimes w_{ic}$$

Since $\#\{a, b, c \in \{0, \ldots, h\} : a + b + c = h\} = \binom{h+2}{2}$, for each i, at most $\binom{h+2}{2}$ terms have the right degree in ε to contribute to t. So to compute t exactly, $r \cdot \binom{h+2}{2}$ products suffice. □

An obtained upper bound on $R_h(t)$ can be turned into a statement about $R(t)$ immediately using Lemma 2.7.5, but because c_h is only polynomial in h, it is often better to first tensor up and only then, in the limit case, convert back to an exact computation. This works because the polynomial growth of c_h is negligible against the exponential growth of the ranks. Using Theorem 2.6.6, we can prove its border rank analogue.

Theorem 2.7.6. We have $\omega \leq 3 \cdot \log_{kmn} \underline{R}(\langle k,m,n\rangle)$ for all $k,m,n \geq 1$, $kmn \geq 2$.

Proof. [3, Th. 6.6] Define $N = kmn \geq 2$, and let h be such that $r_h \coloneqq R_h(\langle k,m,n\rangle) = \underline{R}(\langle k,m,n\rangle)$. (Such an h exists by Remark 2.7.3(1).) By Lemma 2.5.14, Remark 2.7.3(4) and Lemma 2.7.4 we have $R_{3hi}(\langle N^i, N^i, N^i\rangle) \leq R_{3h}(\langle N,N,N\rangle)^i \leq r_h^{3i}$ for all $i \geq 1$.

We also have by Lemma 2.7.5: (note that c_{3hi} is polynomial in h and i)

$$\log_{N^i} R(\langle N^i, N^i, N^i\rangle) \leq \log_{N^i}\big(R_{3hi}(\langle N^i, N^i, N^i\rangle)\, c_{3hi}\big) = 3\log_N r_h + \frac{1}{i}\log_N \operatorname{poly}(h,i)$$

Take $k' = m' = n' = N^i \geq 2$. Then $3\log_{k'm'n'} R(\langle k',m',n'\rangle) = \log_{N^i} R(\langle N^i, N^i, N^i\rangle)$, so by Theorem 2.6.6 we get $\omega \leq 3\log_N r_h + \frac{1}{i}\log_N \operatorname{poly}(h,i)$. If we let $i \to \infty$, then $\frac{1}{i}\log_N \operatorname{poly}(h,i) \to 0$, so we get $\omega \leq 3\log_N r_h$ as required. □

2.7.1 Schönhage's τ-Theorem

The τ-theorem is a very productive theorem by Arnold Schönhage that lets us give bounds on ω using bounds on the border rank of basically arbitrary matrix-tensor-like objects. We will use it to obtain a significant improvement over Strassen's algorithm in terms of complexity. First we need some extra notes on tensors.

Definition 2.7.7. Denote by $\langle r\rangle$ the tensor in $R^{r\times r\times r}$ given by $\langle r\rangle_{ij\ell} = 1$ if $i = j = \ell$, else 0.

Definition 2.7.8. For 3-tensors t and t', t is a *restriction* of t', or $t \leq t'$, if there exist R-module homomorphisms $A : R^k \to R^{k'}$, $B : R^m \to R^{m'}$, $C : R^n \to R^{n'}$ such that $t = (A \otimes B \otimes C)t'$. t and t' are said to be *isomorphic*, or $t \cong t'$, if A, B, C are isomorphisms.

Intuitively, $t \leq t'$ means that t cannot be harder to compute than t': after all, an expansion of t' can be turned into one for t by just applying $A \otimes B \otimes C$ to it.

Lemma 2.7.9. $R(t) \leq r \;\Leftrightarrow\; t \leq \langle r\rangle$.

Proof. Note that $\langle r\rangle = \sum_{i=1}^r e_i \otimes e_i \otimes e_i$, where $(e_1, \ldots, e_r)$ is the standard basis of R^r.

[⇐] Since $t \leq \langle r\rangle$, there are homomorphisms A, B, C such that $t = (A\otimes B\otimes C)\langle r\rangle = \sum_{i=1}^r A(e_i)\otimes B(e_i)\otimes C(e_i)$, so indeed $R(t) \leq r$.

[⇒] Since $R(t) \leq r$, we have $t = \sum_{i=1}^r u_i \otimes v_i \otimes w_i$ for certain vectors $u_i \in R^k$, $v_i \in R^m$, $w_i \in R^n$. Define homomorphisms $A(e_i) = u_i$, $B(e_i) = v_i$ and $C(e_i) = w_i$, with which we get $(A \otimes B \otimes C)\langle r\rangle = \sum_{i=1}^r A(e_i) \otimes B(e_i) \otimes C(e_i) = \sum_{i=1}^r u_i \otimes v_i \otimes w_i = t$. □

Lemma 2.7.10. For three N-tensors $t \in R^n$, $t' \in R^{n'}$ and $s \in R^m$, the following holds:

1. $t \oplus t' \cong t' \oplus t$
2. $t \otimes t' \cong t' \otimes t$
3. $s \oplus (t \oplus t') = (s \oplus t) \oplus t'$
4. $s \otimes (t \otimes t') = (s \otimes t) \otimes t'$

5. $s \otimes (t \oplus t') = (s \otimes t) \oplus (s \otimes t')$
6. $\langle 0 \rangle \oplus t = t$; $\langle 1 \rangle \otimes t = t$; $\langle 0 \rangle \otimes t = \langle 0 \rangle$.

Any required isomorphisms are basis permutations.

Proof. We will prove the six statements for 1-tensors, after which the given isomorphisms (the identity map in cases (3)–(6)) can be applied in all N dimensions to obtain the lemma. Note that this works because when computing $t \otimes t'$, their i'th indices merge for each i, meaning that each of their N indices can be viewed independently. Write a basis permutation of R^k as a permutation of $\{1, \ldots, k\}$. A permutation given by $a, b, c \mapsto d, e, f$ should be read as the function sending $a \mapsto d$, $b \mapsto e$ and $c \mapsto f$.

1. The permutation $1, \ldots, n,\ n+1, \ldots, n+n' \ \mapsto\ n'+1, \ldots, n'+n,\ 1, \ldots, n'$ suffices.
2. We have $t_i t'_j = (t \otimes t')_{n(j-1)+i} = (t' \otimes t)_{n'(i-1)+j}$. We can build a permutation by mapping $n(j-1)+i \mapsto n'(i-1)+j$ for each $(i,j) \in \{1, \ldots, n\} \times \{1, \ldots, n'\}$, and this permutation gives the required isomorphism.
3. Clear from the definition.
4. We have $t_j t'_k = (t \otimes t')_{n(k-1)+j}$ and $s_i t_j = (s \otimes t)_{m(j-1)+i}$, so:

$$s_i t_j t'_k = (s \otimes (t \otimes t'))_{m(n(k-1)+j-1)+i} = (s \otimes (t \otimes t'))_{mn(k-1)+m(j-1)+i}$$
$$s_i t_j t'_k = ((s \otimes t) \otimes t')_{mn(k-1)+(m(j-1)+i)}$$

   So indeed $s \otimes (t \otimes t') = (s \otimes t) \otimes t'$.
5. Note that for $1 \le i \le m$ and $1 \le j \le n+n'$:

$$(s \otimes (t \oplus t'))_{m(j-1)+i} = s_i (t \oplus t')_j = \begin{cases} s_i t_j, & 1 \le j \le n, \\ s_i t'_{j-n}, & n+1 \le j \le n+n', \\ 0, & \text{otherwise} \end{cases}$$
$$((s \otimes t) \oplus (s \otimes t'))_{m(j-1)+i} = \begin{cases} (s \otimes t)_{m(j-1)+i} = s_i t_j, & 1 \le j \le n, \\ (s \otimes t')_{m(j-n-1)+i} = s_i t'_{j-n}, & n+1 \le j \le n+n', \\ 0, & \text{otherwise} \end{cases}$$

   So indeed $s \otimes (t \oplus t') = (s \otimes t) \oplus (s \otimes t')$.
6. Clear from the definition. □

Remark 2.7.11. This means that we can use 3-tensors under $\oplus$ and $\otimes$ as if they form a commutative semiring, provided that we always only assume equivalence under isomorphism instead of equality. In particular, by Lemma 2.5.10, if $t \cong t'$ then $R(t) = R(t')$.

We first need two technical lemmas, then we can prove Schönhage's theorem. Write $N \odot t \coloneqq \bigoplus_{i=1}^{N} t$ for $N \ge 0$ and any 3-tensor t.

Lemma 2.7.12. If $N, s \ge 1$ and $R(N \odot \langle k, m, n \rangle) \le b$, then $R(N \odot \langle k^s, m^s, n^s \rangle) \le \lceil \frac{b}{N} \rceil^s \cdot N$.

Proof. Use induction on s. If $s = 1$ then the result is $b \le \lceil \frac{b}{N} \rceil \cdot N$. Otherwise we assume that $R(N \odot \langle k^s, m^s, n^s \rangle) \le \lceil \frac{b}{N} \rceil^s \cdot N$ as induction hypothesis and prove the statement for $s+1$. We have, using the distributivity and $\otimes$-commutativity properties ((5) and (2)) of Lemma 2.7.10:

$$N \odot \langle k^{s+1}, m^{s+1}, n^{s+1} \rangle \overset{(2.5.14)}{\cong} (N \odot \langle k, m, n \rangle) \otimes \langle k^s, m^s, n^s \rangle \overset{(2.7.9)}{\le} \langle b \rangle \otimes \langle k^s, m^s, n^s \rangle \overset{(\star)}{\cong} b \odot \langle k^s, m^s, n^s \rangle$$

where $(\star)$ holds since $\langle b\rangle = b \odot \langle 1\rangle$. Therefore, we have:

$$R(N \odot \langle k^{s+1}, m^{s+1}, n^{s+1}\rangle) \overset{(2.5.10)}{\leq} R(b \odot \langle k^s, m^s, n^s\rangle) \leq R\left(\left\lceil \frac{b}{N}\right\rceil \cdot N \odot \langle k^s, m^s, n^s\rangle\right)$$
$$\leq \left\lceil \frac{b}{N}\right\rceil \cdot R(N \odot \langle k^s, m^s, n^s\rangle) \overset{\text{(IH)}}{\leq} \left\lceil \frac{b}{N}\right\rceil \cdot \left\lceil \frac{b}{N}\right\rceil^s \cdot N = \left\lceil \frac{b}{N}\right\rceil^{s+1} \cdot N$$

This completes the induction. □

Lemma 2.7.13. If $N \geq 1$ and $R(N \odot \langle k, m, n\rangle) \leq b$, then $\omega \leq \frac{3\cdot\log\lceil \frac{b}{N}\rceil}{\log(kmn)}$. $(k, m, n \geq 1,\ kmn \geq 2.)$

Proof. (After [3, Lem. 7.7]) For all $s \geq 1$, we have by Theorem 2.6.6 (since $kmn \geq 2$):

$$\omega \leq \log_{(kmn)^s} R(\langle (kmn)^s, (kmn)^s, (kmn)^s\rangle)$$
$$\overset{(2.5.14)}{=} \log_{(kmn)^s} R(\langle k^s, m^s, n^s\rangle \otimes \langle m^s, n^s, k^s\rangle \otimes \langle n^s, k^s, m^s\rangle) \overset{(2.5.13,\ 2.5.15)}{\leq} \log_{(kmn)^s} R(\langle k^s, m^s, n^s\rangle)^3$$
$$\overset{(N\geq 1)}{\leq} \log_{(kmn)^s} R(N \odot \langle k^s, m^s, n^s\rangle)^3 \overset{(2.7.12)}{\leq} \log_{(kmn)^s}\left(\left\lceil \frac{b}{N}\right\rceil^s \cdot N\right)^3$$
$$= \frac{3}{s}\left(\log_{kmn}\left\lceil \frac{b}{N}\right\rceil^s + \log_{kmn} N\right) = \frac{3\log\lceil \frac{b}{N}\rceil + \frac{3}{s}\log N}{\log(kmn)}$$

Letting $s \to \infty$, we get $\omega \leq \frac{3\log\lceil \frac{b}{N}\rceil}{\log(kmn)}$, which we needed to prove. □

Theorem 2.7.14 (Schönhage's τ-theorem). If $\underline{R}\big(\bigoplus_{i=1}^p \langle k_i, m_i, n_i\rangle\big) \leq r$ with $r > p \geq 1$ and if for all i we have $k_i, m_i, n_i \geq 1$ and $k_i m_i n_i \geq 2$, then $\omega \leq 3\tau$ where τ is defined by $\sum_{i=1}^p (k_i m_i n_i)^\tau = r$.

Proof. (After [3, Th. 7.5]) If $p = 1$, this is Theorem 2.7.6; otherwise, we have $p > 1$.

There is an h such that $R_h\big(\bigoplus_{i=1}^p \langle k_i, m_i, n_i\rangle\big) \leq r$. By Lemma 2.7.4, for $s \geq 1$ we have that $R_{hs}\left(\big(\bigoplus_{i=1}^p \langle k_i, m_i, n_i\rangle\big)^{\otimes s}\right) \leq r^s$, or written out using multinomial coefficients:

$$R_{hs}\left(\bigoplus_{\substack{\sigma_1+\cdots+\sigma_p=s\\ \sigma_i\geq 0}} \frac{s!}{\sigma_1!\cdots\sigma_p!} \odot \left\langle \underbrace{\prod_{i=1}^p k_i^{\sigma_i}}_{k'(\vec\sigma)}, \underbrace{\prod_{i=1}^p m_i^{\sigma_i}}_{m'(\vec\sigma)}, \underbrace{\prod_{i=1}^p n_i^{\sigma_i}}_{n'(\vec\sigma)}\right\rangle\right) \leq r^s$$

(These ranks are equal by Remark 2.7.11.) Use Lemma 2.7.5 to obtain:

$$R\left(\bigoplus_{\substack{\sigma_1+\cdots+\sigma_p=s\\ \sigma_i\geq 0}} \frac{s!}{\sigma_1!\cdots\sigma_p!} \odot \langle k'(\vec\sigma), m'(\vec\sigma), n'(\vec\sigma)\rangle\right) \leq r^s c_{hs} \tag{2}$$

Raising $\sum_{i=1}^p (k_i m_i n_i)^\tau = r$ to the s'th power $(s \geq 1)$, we get:

$$\sum_{\substack{\sigma_1+\cdots+\sigma_p=s\\ \sigma_i\geq 0}} \underbrace{\frac{s!}{\sigma_1!\cdots\sigma_p!}\big(k'(\vec\sigma)m'(\vec\sigma)n'(\vec\sigma)\big)^\tau}_{(\star)} = r^s \tag{3}$$

This sum has $\binom{s+p-1}{p-1} = \frac{s+p-1}{p-1}\cdots\frac{s+1}{1} \leq (s+1)^{p-1}$ terms. Pick values of $\sigma_1, \ldots, \sigma_p$ such that $(\star)$ is maximal. Then $\vec\sigma$ is a constant vector. Define:

$$N = \frac{s!}{\sigma_1!\cdots\sigma_p!}, \qquad b = r^s c_{hs}, \qquad k = k'(\vec\sigma), \qquad m = m'(\vec\sigma), \qquad n = n'(\vec\sigma).$$

Now clearly:

$$R\left(\frac{s!}{\sigma_1!\cdots\sigma_p!}\odot\langle k'(\vec{\sigma}),m'(\vec{\sigma}),n'(\vec{\sigma})\rangle\right)\leq R\left(\bigoplus_{\substack{\sigma_1+\cdots+\sigma_p=s\\ \sigma_i\geq 0}}\frac{s!}{\sigma_1!\cdots\sigma_p!}\odot\langle k'(\vec{\sigma}),m'(\vec{\sigma}),n'(\vec{\sigma})\rangle\right)$$

which is $\leq r^s c_{hs} = b$ by (2), so $R(N\odot\langle k,m,n\rangle)\leq b$. Since the sum in (3) contained at most $(s+1)^{p-1}$ terms, we have $N\cdot(kmn)^\tau\cdot(s+1)^{p-1}\geq r^s$, so $\frac{r^s}{N}\leq(kmn)^\tau(s+1)^{p-1}$ and $(kmn)^\tau\geq\frac{r^s}{N\cdot(s+1)^{p-1}}$, and therefore:

$$\left\lceil\frac{b}{N}\right\rceil\leq\frac{r^s c_{hs}}{N}+1\leq c_{hs}(kmn)^\tau(s+1)^{p-1}+1,\qquad\text{and}\tag{4}$$

$$(kmn)^\tau\geq\frac{r^s}{p^s\cdot(s+1)^{p-1}}\qquad(\text{since } N\leq p^s \text{ (multinomial theorem)})\tag{5}$$

For $s\geq 1$, we have $N\geq 1$ and $R(N\odot\langle k,m,n\rangle)\leq b$. Now since $s\geq 1$, there is a j such that $\sigma_j\geq 1$. Since $k_jm_jn_j\geq 2$ by assumption, we get $kmn=k'(\vec{\sigma})m'(\vec{\sigma})n'(\vec{\sigma})=\prod_{i=1}^p(k_im_in_i)^{\sigma_i}\geq 2$. Therefore we get by Lemma 2.7.13 that $\omega\leq\frac{3\cdot\log\lceil\frac{b}{N}\rceil}{\log(kmn)}$. Since this works for all $s\geq 1$, we can take the limit $s\to\infty$ and obtain: (note that k,m,n depend on s)

$$\begin{aligned}\omega&\overset{(4)}{\leq}\lim_{s\to\infty}3\cdot\frac{\log(c_{hs}(kmn)^\tau(s+1)^{p-1}+1)}{\log(kmn)}\overset{(\star1)}{=}\lim_{s\to\infty}3\cdot\frac{\log(c_{hs}(kmn)^\tau(s+1)^{p-1})}{\log(kmn)}\\&=\lim_{s\to\infty}3\cdot\frac{\log c_{hs}+\tau\log(kmn)+(p-1)\log(s+1)}{\log(kmn)}=3\tau+3\lim_{s\to\infty}\frac{\log c_{hs}+(p-1)\log(s+1)}{\log(kmn)}\end{aligned}$$

and:

$$\lim_{s\to\infty}\frac{\log c_{hs}+(p-1)\log(s+1)}{\log(kmn)}\overset{(\star2)}{\leq}\lim_{s\to\infty}\frac{\log\text{poly}(s)+(p-1)\log(s+1)}{s\cdot\frac{\log r-\log p}{\tau}}=\lim_{s\to\infty}\frac{O(\log(s))}{O(s)}=0$$

where:

($\star1$) holds because c_{hs} and kmn are positive integers (i.e. ≥ 1) and τ and p are constants with $p>1$, so $c_{hs}(kmn)^\tau(s+1)^{p-1}\to\infty$ as $s\to\infty$;

($\star2$) holds because:

$$\log(kmn)=\frac{1}{\tau}\log((kmn)^\tau)\overset{(5)}{\geq}\frac{s}{\tau}\log r-\frac{s}{\tau}\log p-\frac{p-1}{\tau}\log(s+1)\geq s\cdot\frac{\log r-\log p}{\tau}$$

and $\log r-\log p>0$ (since $r>p$ by assumption).

Therefore, $\omega\leq 3\tau$. □

2.7.2 A Simple Application

From [21, Lem. 6.1], we know that $\underline{R}(\langle k,1,n\rangle\oplus\langle 1,(k-1)(n-1),1\rangle)\leq kn+1$ for $k,n\geq 2$. Using the τ-theorem, Theorem 2.7.14, yields $\omega\leq 3\tau$ where $(kn)^\tau+((k-1)(n-1))^\tau=kn+1$; the resulting value for τ is smallest for $k=n=4$, which gives $\tau<0.849332$, or $\omega<2.548$. This means that matrix multiplication of $n\times n$ matrices is in $O(n^{2.548})$. While this is not yet close to the state of the art ($O(n^{2.3728639})$ [13]), it is a significant improvement over the Strassen algorithm, which has complexity $O(n^{2.81})$. However, the downside is that this improved bound on ω does not come with an explicitly constructed algorithm, so in practice, Strassen remains the preferred choice.

3 Performance

3.1 Introduction

In this section, we will look at the practical execution of large-integer matrix multiplication. We will describe and optimise a CPU and a GPU implementation and evaluate their performance to determine which platform is most suitable. Large-integer matrix multiplication separates into two components: multiplying matrices efficiently in general, and multiplying large integers efficiently. The second component is largely solved by existing large-integer libraries like GMP [9], and on the CPU platform we will use this library unchanged. On the GPU, we make a selection of functionality and implement an extension to support an interleaved representation of numbers. In Section 3.2, we will describe the GMP implementation on a high level, and explain the GPU adaptation and extensions.

The first component is less completely solved; dense matrix multiplication over floating-point numbers is well-studied (e.g. [2]), since that is the common case and it has hardware support on modern architectures, which provide wide vector units for performing multiple floating-point operations in parallel. On the other hand, matrix multiplication over different elements, in this case large integers, has not enjoyed as much attention. On the CPU platform, there are implementations in number theoretic libraries like FLINT [11] and NTL [23] and in software packages like PARI/GP [10]; here, a custom one will be developed, both for CPU and GPU, for full parametrisability.

An alternative method of multiplying large-integer matrices is to reduce the matrix elements using repeated applications of the Chinese Remainder Theorem to transform the matrix multiplication into many small-integer matrix multiplications, which can be executed using native machine integers. This approach is also called using "multi-modular reduction", and is taken for large enough matrices by libraries like FLINT, but is considered out of scope here due to time constraints. Therefore, we will look only at directly multiplying matrices over large integers.

A comparison of our CPU implementation to the existing CPU libraries mentioned above is in Section 3.5.3.

3.1.1 Input Data Set

The degrees of freedom in an input to a large-integer matrix multiplication program are the matrix sizes and the distribution of sizes of the elements of these matrices. We choose to not measure the results of irregular distributions of the element sizes, so in each test case, we draw elements uniformly from the set $\{-2^b+1, \ldots, 2^b-1\}$ for some $b \in \mathbb{Z}_{\geq 1}$, the *bitsize* of the elements. Considering expected use cases, and because of the time required to calculate products with very large input parameters, the following matrix sizes and bitsizes are used:

- Matrix sizes: 32, 64, 128, 512, 1024, 2048 (number of elements along a side)
- Bitsizes: 16, 512, 1024, 2048

Small parameters were also included to be able to judge tradeoffs introduced by constant overheads like the set-up and teardown of a GPU context, or the creation and destruction of CPU threads. The choice of powers of 2 for the bitsizes is to cover a large range of bitsizes with few data points, but the choice of powers of two for the matrix sizes is due to the fact that the implementations work most naturally on these inputs. However, since it is unreasonable to only be able to multiply square matrices with size a power of 2, we consider irregular input sizes in Section 3.7.

The set of input matrices consists of one pair of randomly generated matrices for each combination of a matrix size and a bitsize listed above. The same pair of input matrices is used for each test that combines the same matrix size and bitsize.

3.2 Large-Integer Arithmetic

In GMP, and in many other large-integer arithmetic libraries, a large integer is represented as an array of *limbs*, each of which is typically a normal unsigned hardware-native integer. For example, on a modern 64-bit CPU, each of these limbs would be a 64-bit integer, capable of representing the nonnegative integers $< 2^{64}$. We assume that the hardware in question operates with two's-complement base-2 arithmetic. If b is the number of limb bits (in the example, $b = 64$) and the array of limbs that makes up the large integer is $(a_0, \ldots, a_{n-1})$, then if the library uses little-endian representation (which is usual), the represented value is $\sum_{i=0}^{n-1} a_i 2^{bi}$. In GMP, the structure that stores a large integer contains a pointer to a dynamically allocated array of limbs, a field for number of limbs allocated ("alloc"), and a field for the number of limbs currently in use in the array ("size"). Negative numbers are represented using a negative size field. In the text below, we will use the terms *alloc* and *size* for respectively the capacity and usage of the limb array of a large integer.

To perform arithmetic with these structures, GMP contains functions that implement the usual arithmetic operations (addition, subtraction, multiplication, division), as well as more specialised operations (taking of square roots, computing the greatest common divisor, etc.). If at some point, the array storing the limbs of the destination number is too small to contain the entire result, this array is enlarged with a reallocation and the structure is updated.

The core of GMP's logic is in the internal *mpn* (Multi-Precision Naturals) module, which implements basic unsigned large-integer operations on arrays. This module is then used by the public *mpz* (Multi-Precision $\mathbb{Z}$) module, which uses the structure mentioned above and implements allocation and negative (signed) integer handling. (GMP contains other modules for multi-precision rationals (*mpq*) and floating-point numbers (*mpf*), but these are not relevant to the subject at hand.)

3.2.1 Multiplication Algorithms

The arithmetic operations we use are addition, subtraction, multiplication, and their combinations "addmul" ($c = c + a \cdot b$) and "submul" ($c = c - a \cdot b$). Addition and subtraction use the normal linear algorithms, but multiplication has a more complicated implementation, using a variety of algorithms for increasing sizes of the input numbers. Generally, for large-integer multiplication algorithms, the better its asymptotic complexity, the higher the constant factor in the actual number of native arithmetic operations. Therefore, the lower-complexity algorithms are chosen for larger input sizes, and simpler algorithms for smaller input sizes.

The algorithms implemented by GMP are the following: [8]

- Basecase multiplication. This is the naive $O(km)$ algorithm (if k and m are the sizes of the input numbers) that is simple and very efficient on modern hardware, using vectorised instructions.
- The Toom family of algorithms. This includes Karatsuba's algorithm as toom22, and generalised variants of Karatsuba as toom33, toom44, toom6h and toom8h.
- FFT multiplication (the Schönhage–Strassen algorithm [22]), which is only used for very large operands.

The thresholds for switching between these algorithms depend on the machine GMP is tuned for, but as a general indication, one such set of thresholds is given. After tuning the thresholds in GMP 6.1.2 on an the processor used for the CPU tests (on which GMP uses 64-bit limbs), toom22, toom33, toom44, toom6h and toom8h are respectively used starting at 20 limbs (1280 bits), 65 limbs (4160 bits), 172 limbs (11008 bits), 274 limbs (17536 bits) and 357 limbs (22848 bits). FFT multiplication starts at 4736 limbs (303104 bits). The determination of these thresholds is approximate, since since the crossover points are usually not sharp; and in the case of unbalanced multiplication (where one of the multiplication operands is significantly larger than the other), GMP includes a number of Toom-inspired algorithms that are specially constructed for varying degrees of imbalance, and these have their own list of thresholds.

The *basecase* algorithm performs naive multiplication of the two operands as follows. Assume that the product $c = a \cdot b$ is to be calculated, where c (alloc n), a (size k) and b (size m) are large integers with limb arrays indexed respectively as c_i, a_i and b_i. Assume that $n \geq k + m + 1$, otherwise a reallocation is needed to ensure this is true. Then the multiplication proceeds as follows:

```
1 C ← 0
2 for j = 0 to k − 1 do
3       c_j ← a_j b_0 + C, carry in C
4 c_k ← C
5 for i = 1 to m − 1 do
6       C ← 0
7       for j = 0 to k − 1 do
8             c_{i+j} ← c_{i+j} + a_j b_i + C, carry in C
9       c_{i+k} ← C
```

Where a multiplication of limbs is performed, the result is two limbs in size. The lower result limb is returned as usual, but the higher result limb is taken as *carry* and stored in C in the above pseudocode. Note that a limb, plus a product of two limbs, plus the carry of a product of two limbs, always fits within two limbs; this means that the above algorithm is well-defined. As mentioned, this algorithm has complexity $O(km)$; however, the constant factor is very low (i.e. few extra arithmetic operations are hidden by the big-O notation), so the algorithm is optimal in practice for small enough numbers.

All other algorithms are recursive algorithms: they reduce the multiplication of two large integers to a number of smaller multiplications, each of which is executed using the same or a different algorithm. This recursion always terminates at basecase multiplication.

The first of these recursive algorithms is *toom22*, also known as Karatsuba's algorithm. In essence, this is an algorithm to multiply two degree-one polynomials with three coefficient multiplications instead of four: given $a_0 + a_1 X$ and $b_0 + b_1 X$, the product polynomial is $a_0 b_0 + (a_0 b_1 + a_1 b_0) X + a_1 b_1 X^2$. The middle coefficient can be calculated as $(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$, yielding the coefficients of the product in just three multiplications. This is applied to multiplication of the (nonnegative) large integers a and b by choosing some value k (typically half the number of bits in one of the operands) and writing $a = a_0 + 2^k a_1$ with $0 \leq a_0 < 2^k$, and $b = b_0 + 2^k b_1$ analogously. Then using four additions and three multiplications on operands that are half as long as the original values a and b, the above algorithm gives c_0, c_1 and c_2 such that $a \cdot b = c_0 + 2^k c_1 + 2^{2k} c_2$. Since multiplications by 2^k are especially cheap in large-integer implementations with a limb size that is a power of 2, the product can be then calculated efficiently using another two additions.

Toom22 splits the operands in two pieces and makes linear combinations of the parts to save sub-multiplications. The higher Toom variants split the operands in more pieces to save a larger fraction of the multiplications, at the cost of even more additions. All Toom algorithms can be interpreted as evaluation and interpolation of polynomials; for a description, see e.g. [8]. The

final algorithm, Schönhage–Strassen, uses fast Fourier transforms to perform the multiplication of two n-bit numbers in time $O(n \log n \log\log n)$; for a description, see technical notes for GMP [8] or the original text [22].

3.3 Implementations

Two main implementations have been created:[1] one that computes the product using a CPU, and one that computes the product using a GPU for the main computations. Each of these implementations has parameters that can be varied, and the goal is to identify the platform that is most suited to multiplying large-integer matrices quickly, and with what parameter settings that optimum is reached. Traditionally, the CPU is the platform of choice for this kind of operation, but the GPU is tried as well because of the inherent parallelism in matrix multiplication: each of the target matrix elements can in principle be calculated independently from the others.

In both cases, the core small-matrix multiplication code uses the naive algorithm with three nested loops and can handle matrices of any size (barring memory constraints), but the recursive divide-and-conquer algorithms used to speed up multiplication of larger matrices assume that all matrices are square with side length a power of two. This simplifies and speeds up the subdivision code in these algorithms. The routines that implement these algorithms are wrapped in a function that subdivides the input matrices into submatrices that are either square with side length a value with enough factors of 2, or small enough to multiply directly with the core routines that can handle rectangular matrices. This ensures that matrices of any (compatible) sizes can be multiplied, while the recursive algorithms do not need to cope with the difficulties introduced by non-square matrices.

We will first detail how the used recursive algorithms work in theory, and then how the specific implementations for the CPU and GPU use these algorithms to multiply the matrices.

3.3.1 Recursive Algorithms

If the product of two matrices A and B is a matrix C, then A, B and C should have dimensions $k \times m$, $m \times n$ and $k \times n$ for certain $k, m, n \in \mathbb{Z}_{\geq 1}$, as also set out in Section 2. In this subsection, we index matrix A as follows:

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{k1} & \cdots & A_{km} \end{pmatrix}$$

and matrix B and C analogously. The **core multiplication** procedure is the naive algorithm, represented in pseudocode as follows:

```
1 for i = 1 to k do
2     for j = 1 to n do
3         C_ij ← A_i1 B_1j
4         for ℓ = 2 to m do
5             C_ij ← C_ij + A_im B_mj
```

When applying this algorithm to block matrices of 2×2 blocks each, this gives a recursive algorithm that turns one large matrix multiplication into 8 smaller matrix multiplications and 4 additions. This will be called the **naive** recursive algorithm. It gives no improvement in complexity whatsoever, but permits distribution of work with the characteristic that the created

[1] The code for the implementations can be found at `https://github.com/tomsmeding/bachelor-thesis`.

jobs have a more localised memory access pattern. This is one of the possible algorithms used in the CPU implementation to distribute work over processors.

However, the multiplication of 2×2 matrices can be accomplished in 7 element multiplications (which may be smaller matrix multiplications), although 7 is the minimum for 2×2 matrices [29]. The traditional algorithm with 7 multiplications is **Strassen**'s algorithm, and it performs the following multiplication:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

using the calculations given below: [24]

$$\begin{aligned} P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) & P_5 &= (A_{11} + A_{12})B_{22} \\ P_2 &= (A_{21} + A_{22})B_{11} & P_6 &= (-A_{11} + A_{21})(B_{11} + B_{12}) \\ P_3 &= A_{11}(B_{12} - B_{22}) & P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ P_4 &= A_{22}(-B_{11} + B_{21}) \end{aligned}$$

$$C_{11} = P_1 + P_4 - P_5 + P_7 \qquad C_{21} = P_2 + P_4 \qquad C_{12} = P_3 + P_5 \qquad C_{22} = P_1 + P_3 - P_2 + P_6$$

On each recursion level it uses 18 additions.

There is a variant (called **Winograd** here) of Strassen's algorithm due to Shmuel Winograd that accomplishes the multiplication of 2×2 matrices in 7 multiplications and only 15 additions [20, Lemma 17]. For 7 multiplications, this number of additions is optimal [20]. The algorithm performs the same matrix multiplication as above using the following calculations:

$$\begin{aligned} P_1 &= A_{21} + A_{22} & Q_1 &= P_2 P_6 & R_1 &= Q_3 + Q_2 & C_{11} &= R_1 \\ P_2 &= P_1 - A_{11} & Q_2 &= A_{11} B_{11} & R_2 &= Q_1 + Q_2 & C_{21} &= R_6 \\ P_3 &= A_{11} - A_{21} & Q_3 &= A_{12} B_{21} & R_3 &= R_2 + Q_4 & C_{12} &= R_5 \\ P_4 &= A_{12} - P_2 & Q_4 &= P_3 P_7 & R_4 &= R_2 + Q_5 & C_{22} &= R_7 \\ P_5 &= B_{12} - B_{11} & Q_5 &= P_1 P_5 & R_5 &= R_4 + Q_6 \\ P_6 &= B_{22} - P_5 & Q_6 &= P_4 B_{22} & R_6 &= R_3 - Q_7 \\ P_7 &= B_{22} - B_{12} & Q_7 &= A_{22} P_8 & R_7 &= R_3 + Q_5 \\ P_8 &= P_6 - B_{21} \end{aligned}$$

3.3.2 CPU

The CPU implementation is in the C language and has three parameters that we vary in the experiments. The values assigned to the parameters are given below:

- `algorithm`: Strassen, Winograd, naive/Winograd
- `baseblocksize`: 16, 32, 64
- `jobblocksize`: $n/2$, $n/4$, $n/8$ (where n is the matrix size); but only those for which `baseblocksize` $\leq$ `jobblocksize`

The program functions as follows. We will first describe how square and power-of-2 sized matrices are multiplied, and afterwards how arbitrarily sized input matrices are subdivided to be able to multiply them in parts using the procedure specified here.

Assume two $n \times n$ ($n = 2^k$ for some $k \in \mathbb{Z}_{\geq 0}$) matrices A and B are given, and their product C needs to be calculated. One of the three recursive algorithms from Section 3.3.1 is used to divide A and B in smaller and smaller matrices until the size of the small matrices is no more than `jobblocksize`. All matrix operations that need to be performed for this subdivision to work, i.e. all additions, subtractions and (smaller) multiplications, are registered in a job system including

all of their dependencies on previous jobs. (See below for the workings of the job system.) The addition and subtraction jobs execute the required operations without further subdivision, while the multiplication jobs perform another few recursion steps to subdivide the matrices, potentially using a different recursive algorithm (from Section 3.3.1) than before. This recursion process happens entirely in one CPU thread and does not suffer from job distribution overhead, so the size of the smallest multiplications produced can be much smaller than would be reasonable when using the job system for this process. The second recursion process continues until the small multiplications are no larger than `baseblocksize`, at which point the core multiplication routines (see Section 3.3.1) perform the final small multiplication. It is clear that we have a requirement that `baseblocksize` $\leq$ `jobblocksize`.

For the above process, two algorithm choices must be made: one for job distribution and one for thread-local recursion. The `algorithm` parameter determines which are chosen: the Strassen and Winograd options use those respective algorithms for both job distribution and thread-local recursion, while the naive/Winograd option uses naive recursion for job distribution and Winograd for thread-local recursion. The third option is included since it allows complete parallelism without needing a lot of extra memory, while using the most efficient implemented algorithm for the thread-local calculations; it is therefore attractive in many-core scenarios.

The description above mentions the use of a *job system*: this is a simple module that accepts jobs in the form of a function pointer and a list of dependency jobs, and registers them for execution later. When all jobs are registered, the job system starts a number of threads (specified by the number of CPU threads allocated to the multiplication using a parameter) and starts distributing jobs to these threads. Jobs of which all dependencies have been run to completion are egligible for execution, and the system attempts to keep all threads busy with executable jobs as much as possible. The benefit of using a job system as described here, as opposed to explicit parallelism in the subdivision algorithms, is that any number of threads may be used effectively. Implementations that attempt "perfect" parallelism (e.g. [2]) can typically only make use of a thread count that is a power of 7, stemming from the usage of a Strassen-like algorithm for the recursion steps. The author knows of no production CPU that has a power-of-7 core count greater than 1.

Note that while we assumed that n is a power of 2 in the above, the process actually works for any $n = b2^p$ for $b, p \in \mathbb{Z}_{\geq 0}$ with $1 \leq b \leq$ `baseblocksize`. If this is the case, the recursion processes are always able to divide n by two when they need to, so no unexpected conditions occur. This is used in the *squarify* function that generates jobs for multiplication of arbitrarily (but compatibly) sized matrices: this function first checks whether the input matrices are square with side length $b2^p$ as before, and if so, directly calls into the recursive procedure outlined above. If the matrix sizes are less than `baseblocksize`, only one job is created using the core matrix multiplication routine. Otherwise, the input matrices are $k \times m$ and $m \times n$. The largest $p \in \mathbb{Z}_{\geq 0}$ is found such that $2^p \leq \min\{k, m, n\}$. Both matrices are then subdivided into four submatrices as follows:

$$A = \begin{array}{|c|c|} \hline 2^p \times 2^p & 2^p \times (m-2^p) \\ \hline (k-2^p) \times 2^p & (k-2^p) \times (m-2^p) \\ \hline \end{array} \qquad B = \begin{array}{|c|c|} \hline 2^p \times 2^p & 2^p \times (n-2^p) \\ \hline (m-2^p) \times 2^p & (m-2^p) \times (n-2^p) \\ \hline \end{array}$$

These submatrices are then multiplied using the naive recursive algorithm. (The more complicated Strassen algorithms in general only work for square and even-sized matrices, but can be made to work with irregularly sized matrices using a lot of bookkeeping and extra loop conditions; the decision was made to not implement this and stay with naive recursion.) Each of the smaller

multiplications is again handled by the squarify function, which might then choose to directly perform the multiplication or to subdivide further.

Arithmetic on the large integers in the matrices to be multiplied is performed using the GMP library [9]. This library was tuned for the target machine using the `tuneup` program bundled with GMP; furthermore, it it statically linked into the executable to eliminate any function call overhead arising from dynamic linking.

3.3.3 GPU

For the GPU implementation, we have chosen to use only one GPU and consider multi-GPU parallelism out of scope. On the topic of multi-GPU processing, it has been found that when exploiting the massive parallelism of multi-GPU setups, the advantages of using Strassen-like algorithms diminish and it may be faster to use naive multiplication directly [30]. Because the individual element multiplications for large-integer matrices are relatively costly for their memory footprint increase (since the large-integer multiplication algorithms used are not linear in complexity), in this case we expect the conclusion from [30] to remain valid, albeit only for even larger amounts of parallel hardware.

The GPU implementation uses the CUDA framework, and is therefore written in CUDA C++ and runs on compatible NVIDIA GPU's. All core multiplications are executed on the GPU, but the recursion steps are executed on the CPU: the leaves of the recursion tree are kernel launches. Unlike the CPU version, no parallelism in the recursion tree is exploited, since the multiplication kernels themselves already use the parallelism that the GPU offers. (In particular, we do not use multiple CUDA streams to execute kernels concurrently, since in practice this was found to decrease, instead of increase, performance.) The implementation has three parameters that we vary in the experiments. The values assigned to the parameters are given below:

- `algorithm`: Strassen, Winograd
- `interleaved`: yes, no
- `gputhreadblock`: width 8, 16, 32, 64, 128, 256; height 2, 4, 8, 16, 32, 64

Since no recursion-tree parallelism is used, unlike in the CPU version, it is unnecessary to consider naive recursion in addition to Strassen-like recursion. The `interleaved` parameter indicates whether digit interleaving is performed on the matrix elements; this will be detailed later. The `gputhreadblock` parameter determines the GPU thread block size used. (For information about the details of the GPU programming model including thread blocks, see e.g. the NVIDIA CUDA Programming Guide [18, §2 Programming Model].) The specifics of the GPU kernels and their register usage result in a maximum thread block size of 512 (when restricting the dimensions to powers of 2), after which the maximal thread block dimensions follow directly from the minimum dimensions. Due to this maximum thread block size, there are 36 valid values for `gputhreadblock`.

The core GPU kernels can multiply arbitrarily (but compatibly) sized matrices, being limited only by GPU memory capacity. They are used for the small multiplications needed by the recursive algorithms. Like in the CPU implementation, the recursive subdivision code in the GPU implementation assumes square and power-of-2 sized matrices, where an allowance is made for matrix sizes that have enough factors 2 such that they can be evenly subdivided into small enough matrices. A *squarify*-like function is included that allows these procedures to be used with arbitrarily sized matrices (see the previous section). Note that in the current implementation, `interleaved` is incompatible with squarify, which means that irregularly sized matrices cannot be multiplied in interleaved mode. In the following, the code will be described that works under the assumption of square and power-of-2 sized matrices, like in the previous section.

A number of recursion steps with the specified `algorithm` are performed until the matrix size is at most the global constant `TRANSFER_SIZE`. At this point, the matrices are copied to GPU memory. The reason this is not done with the initial matrices is that these may not fit in GPU memory, and subdivision may be necessary for them to fit. This constant is not varied in the experiments because a higher value, if admissible with the amount of GPU memory available, will always induce less overhead and will therefore be faster. In the tested implementation, we have chosen the value `TRANSFER_SIZE` = 512. Using Winograd on 2048×2048 matrices with bitsize 2048, this results in a peak GPU memory usage of 867MiB. A higher limit was possible, but only a negligible speedup is expected, with memory transfers already taking less than 2% of the run time according to `nvprof`.

After the transfer, another few recursion steps with the `algorithm` are performed until the matrices are no larger than `BASE_SIZE` (in the tested implementation, `BASE_SIZE` = 256), at which point the multiplication is executed directly with the naive GPU kernel. We have found that selecting a smaller `BASE_SIZE` is not beneficial for the total run time, so this constant is taken.

To multiply matrices over large integers on a GPU, we need to be able to multiply large integers on a GPU. The next section details how this was accomplished.

3.3.4 GMP on the GPU

While no research could be found concerning large-integer matrix multiplication on the GPU, performing large-integer multiplication itself on the GPU has been studied before. In [6], hand-optimised PTX assembly routines were used to multiply separate large integers on the GPU. The only multiplication algorithm implemented is Schönhage–Strassen FFT multiplication, and measurement results are reported for 384K bits and higher. While the speedups compared to the CPU are promising (2.2× at 384K bits to 5.8× at 16384K bits), our focus on smaller bitsizes means the code for this work is likely not directly useful here.

In [19], a different approach was taken, while the focus was also on separate large-integer multiplications of sizes larger than we consider here (they perform multiplication tests with bitsizes between 1024K and 131072K bits). The CUDA library cuFFT was used to evaluate the fast Fourier transforms necessary for Schönhage–Strassen, which means that presumably, the FFT code is well-optimised. However, cuFFT is a floating-point FFT library, so barring extensive error analysis of the computations and measures to prevent such errors, the results will be inaccurate; and indeed, it is reported [19, Table 5.5] that for e.g. 1024K-bit multiplication (which takes operands on the order of $2^{1048576}$), the relative error between the GPU results and accurate GMP results on the CPU is $1.21 \cdot 10^{-6} \pm 5.52 \cdot 10^{-6}$, which can mean that most of the bits are incorrect. While they also report that for the majority of the calculations, the relative error was zero [19, Figure 5.5], the possibility of errors of this magnitude is not acceptable for our purposes.

In [15], a floating-point FFT is also used, resulting in inaccurate multiplication results. Improvements over CPU processing were found for bitsizes larger than 65536 bits. An alternative to FFT multiplication is given in [12], which uses a novel data structure they call a "product digit table" to parallelise the multiplication; results are reported for bitsizes of 8192 bits and higher.

Summarising the above, FFT multiplication techniques will not be suitable for our purposes, and results on other techniques for large-integer multiplication on the GPU are scarce. Furthermore, the found sources all parallelise a single large-integer multiplication, while we can get parallelism from just executing many multiplications in parallel. Since in addition, our bitsize range is outside the ranges for which results are reported in the found sources, we choose to create a new

implementation for performing large-integer arithmetic on the GPU.

On the CPU, we used the GMP library to perform large-integer arithmetic, but this library is large and not directly usable on the GPU, even with CUDA, because of the use of dynamically-sized limb buffers and heap allocations to manage these buffers. Since GMP contains optimised code written in platform-independent C code, we would still like to use the algorithms contained therein for our GPU implementation. Therefore, a number of source files from the GMP 6.1.2 source distribution (the newest at the time of writing) are extracted and modified to create a small library that can be directly compiled using the NVIDIA CUDA C++ compiler (`nvcc`). The extraction is performed by selecting the files implementing the basic arithmetic functions needed for our algorithms from the GMP mpz module (addition, subtraction, multiplication, addmul ($c = c + a \cdot b$) and submul), including their transitive dependencies in the GMP source tree. A custom, reduced header file `gmp.h` was also created, merging the relevant parts of `gmp.h` and `gmp-impl.h` from the GMP source distribution. All defined types and functions are renamed by prefixing them with `gpu_` to enable linking a normal CPU GMP library into an application using our GPU large-integer library.

The large-integer data structure in the original GMP library contains three fields: a pointer to an array of limbs, the allocated size of that array, and the currently used size of that array. In our library, we replace the pointer and the allocated size by a single limb array of a statically-known size. This means that the library enforces a static limit on the bitsize of the integers represented, but this static limit can be set to any desired value with a recompilation. The advantage to this static representation is that a fixed amount of storage is sufficient for each integer and no heap allocations are performed. Therefore, a matrix of large integers is one contiguous buffer, which is amenable to efficient transfers between main memory and GPU memory.

For multiplication, the basecase algorithm and the toom22 algorithm are ported, and all integers above the toom22 threshold are multiplied using the toom22 algorithm. (See Section 3.2.1 for details on the multiplication algorithms.) The reason for not porting the other lower-complexity algorithms is that they are more involved and need much more code; the amount of time necessary for porting these algorithms was considered too large for the expected negligible return. However, this should be kept in mind when reading the experiment results, because this may put the GPU at a slight disadvantage to the CPU for the largest bitsizes, even though the CPU should also not use algorithms above toom22. Note that the ported library uses 32-bit limbs, instead of 64-bit limbs like the CPU version, since there is no efficient way to multiply two 64-bit integers and access the full 128-bit result on a GPU.

On the GPU used for testing, data is fetched from GPU memory in blocks of 256 bits, regardless of the size of the memory fetch instructions in the code. Since the limbs used are 32 bits long, and in a direct port to the GPU the matrix elements are put in memory as a whole, this means that memory bandwidth efficiency is about 12.5%. During development, this was also observed using the `nvprof` tool. To improve on this, the necessary ported GMP functions were duplicated and altered to assume the limbs of their arguments are not adjacent in memory, but rather are further apart; the offset of the next limb to the previous limb, counted in limb sizes, is called the *stride* of the large integer. Having these functions allows interleaving the limbs of adjacent matrix elements so that the first limbs of the numbers are adjacent and followed by all the second limbs, etc. This interleaving is performed in blocks of `INTERLEAVE_BLOCK_SIZE` elements, which in the tested implementation is chosen equal to `BASE_SIZE`. This optimisation requires pre- and post-processing on the CPU to interleave and de-interleave the matrix elements, but improves memory efficiency on the GPU by allowing the hardware to perform memory access coalescing [17, §9.2.1 Coalesced Access to Global Memory]. The `interleaved` parameter determines whether this interleaved implementation is used. Note again that this implementation requires the matrix size (and all subdivided matrices) to be a multiple of `INTERLEAVE_BLOCK_SIZE`, and will only be

used with square power-of-2 sized matrices in the experiments.

Note that we have not compared the performance of our implementation with the existing implementations referenced above, because drawing conclusions from a comparison between an implementation that derives parallelism only from performing multiple multiplations at once (ours) and an implementation that performs one multiplication in parallel (those mentioned above) would be very hard. To still illustrate the relative performance of our implementation as described above, it was run on the CPU and compared to the vanilla tuned CPU GMP library. For bitsizes in the input data range, the code is about 8 times as slow as the tuned GMP code. This factor is of course large, but the speedup of the original GMP code can be attributed to the super-optimised assembly code in the basecase multiplication routine and the use of 64-bit limbs instead of 32-bit. Since doing the former with PTX assembly is out of scope due to time constraints and the latter is necessary, we assume that our GPU code is performant enough to justify CPU and GPU comparison.

3.4 Experimental Setup

For the CPU tests, the following system properties and versions apply:

- GMP version: 6.1.2; tuned using `tuneup` on the testing machine and statically linked into the executable
- Source compiled using `gcc -O3 -flto -march=native -mtune=native`; GCC version 5.4.0
- Platform: Ubuntu 16.04.4 LTS on Linux 4.4.0
- Processor: 2 sockets, each an Intel Xeon E5-2667 v2 at 3.30Ghz. 8 cores per socket, two threads per core, for a total of 32 virtual cores. 48GiB memory.

For the GPU tests, the setup is as follows:

- GMP version from which code was extracted: 6.1.2
- Source compiled using `nvcc -O3` with GCC version 5.4.0; CUDA release 8.0, V8.0.61
- Platform: Ubuntu 16.04.4 LTS on Linux 4.4.0
- Processor: Intel Core i7-6950X at 3.00Ghz. 10 cores, two threads per core, for a total of 20 virtual cores. 48GiB memory.
- GPU: NVIDIA Titan X Pascal

3.5 CPU Results

Because of the hardware parallelism of the machine on which the CPU tests were run, the tested thread counts are 1, 2, 4, 8, 16 and 32.

Three variables are realistically out of control of the implementor: matrix size, bitsize and thread count, the latter because the maximum thread count is dictated by the hardware parallelism of the used machine and starting more threads than necessary incurs only a near-constant overhead. These three variables are called *input conditions* in this section. For example, when we say "for each input condition", we mean for each combination of matrix size, bitsize and thread count, where each varies in its designated set: $\{1, 2, 4, 8, 16, 32\}$ for the thread count, and the sets given in Section 3.1.1 for the other variables.

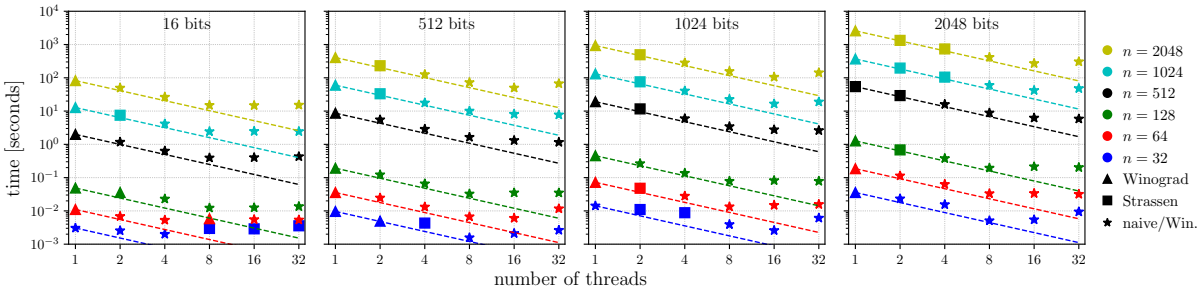

Figure 3.1: (log–log) Time taken and optimal algorithm chosen for given bitsize and matrix size (colour). The dashed lines indicate ideal timings if work division scaled perfectly with the number of threads.

The variables `algorithm`, `baseblocksize` and `jobblocksize` can be tuned by the implementor, and these are called *parameters* in this section.

For each input condition, each possible parameter set was enumerated, and a matrix multiplication was executed with this input condition and parameter set. The measured time is the entire multiplication, without I/O. The data set of the resulting timings is analysed below. For quantification of the measurement error, each *test case* (one for each pair of an input condition and a parameter set) of which the original timing was less than 2 seconds was re-run twice, and of the other test cases, 20% was randomly selected to be re-run twice as well. For each input condition, the optimal parameter set was determined by selecting the parameter set with the minimal run time. These optimal sets can be found in Table A.1. There, and in the rest of this section, the notation "St 16 /2" indicates the parameter set `algorithm` = Strassen, `baseblocksize` = 16 and `jobblocksize` = $\frac{n}{2}$, where n is the matrix size. The algorithms Winograd and naive/Winograd are respectively indicated with "Wi" and "nW".

The timings corresponding to the optimal parameter sets are in Table A.2, and the maximum relative errors of the re-runs compared to the original timings for each input condition are in Table A.3. The *relative error* of a re-run timing t_1 compared to an original timing t_0 is computed as $\frac{|t_1-t_0|}{t_0}$; such a computation will always be meant when "relative error" is used. (Table A.3 should be read with the absolute times in Table A.2 in mind; small timings fluctuate more, but are also less important in the analysis below.)

3.5.1 Algorithm Choice

The naive/Winograd algorithm provides the most parallelism of the three algorithms, while Winograd trades less memory usage and fewer additions for less parallelism. Strassen is somewhere between these two algorithms in all dimensions. This means that we expect Winograd to perform best with 1 thread, and that Strassen, and later naive/Winograd, will overtake Winograd as the number of available threads increases.

For each input condition, there is a data point in Figure 3.1, which contains a frame for each tested bitsize and within each frame a colour for each tested matrix size. The different markers indicate which algorithm was optimal in that case. (The chosen block sizes are not shown here.) The dashed, coloured lines indicate for each matrix size and bitsize what the theoretical ideal runtimes would be for higher thread counts: they are calculated by dividing the runtime for 1 thread by the number of threads.

What we see is that, apart from the smallest multiplications which take only a few milliseconds, the Winograd algorithm is indeed fastest for 1 thread, naive/Winograd is fastest for large numbers of threads, and Strassen is best for some number of thread counts in between, possibly zero. The

early flattening of the data series at the bottom of the 16 bits graph is due to the overhead of setting up all the threads and distributing work between them, which at that size dominates the runtime.

It is clear that at least with the input sizes tested, there is a limit on the amount of parallelism that can be effectively extracted from the computation. For larger matrices, effective speedups are in general only obtained for up to 8 threads, after which the graph flattens. If the individual element multiplications are expensive enough, i.e. for large bitsizes, using 16 threads can still provide a respectable speedup. (For example: for matrix size and bitsize equal to 2048, the best 8-thread timing is about 415 seconds, while the best 16-thread timing is about 270 seconds; see Table A.2.) We observe that using 32 threads never provides a speedup, and in most cases is even somewhat slower than 16 threads. (In the 2048/2048 case, the best 32-thread timing is about 305 seconds.) This should not be surprising, since the machine used for testing has 32 virtual CPU cores but only 16 physical cores, and the GMP library uses hand-optimised assembly code to utilise the CPU core's functional units as well as possible. Using hyperthreading, which shares a core's functional units between multiple threads, is therefore suboptimal as observed.

3.5.2 A Model

The job distribution over the allocated threads is non-deterministic, since small timing differences may influence the exact order in which the jobs are issued: whether all a job's dependencies have been completed at the time of a new job issue might change between runs. Although the block size choices can in general make a large difference in performance, the optimal block sizes are often only a few percent ahead of the next-best choices in the collected measurement data set:

1. In about 49% of the cases, the timings for the best and worst block size choices, everything else being constant, are at least 50% (factor 1.5) apart;
2. In about 42% of the cases, the timings for the best and second-best block size choices, everything else being constant, are at most 5% (factor 1.05) apart.

Together, this means that barring many reruns of the test suite (which is impractical because of the long runtime of the large test cases), it is hard to draw exact conclusions about optimal parameter settings. Indeed, since the job distribution introduces some volatility in the timings, there might not be a single best setting.

However, after acknowledging that the measurements are not exact, a few conclusions can still be drawn. First, reading point (2.) above, it should be noted that the optimal parameters in Table A.1 are not necessarily the only best answers. To be able to judge how significant various differences in timings are, the relative errors in Table A.3 can be used. Since the relative errors are often larger than 5%, it is clear that the specific optimal settings given will not give us a good understanding of what good parameter settings are. What we can do, is try to construct a simple model that, at least in the input condition ranges tested, predicts what parameter setting will produce good performance. This model can then be evaluated by computing the relative error to the measured optimal parameter settings and verifying that this error does not grow large.

Consider the following model, where b is the bitsize, n the matrix size and C the thread count:

- For $C = 1$, use Wi 16 /2.
- For $2 \leq C \leq 8$, or $16 \leq C$ and $b = 16$, use nW 16 /2.
- For $16 \leq C$ and $512 \leq b$, use nW 16 /4.

The relative errors of always using Wi 16 /2 compared to the optimal parameter settings are shown in Table A.4, those of nW 16 /2 in Table A.5 and those of nW 16 /4 in Table A.6. In

those tables, the input conditions for which the optimal time is less than 2 seconds are coloured to indicate which results may be unreliable. From these three tables, it can be read that applying the above model for the chosen input condition ranges yields performance losses that are no more than the largest observed measurement inaccuracies for the input conditions for which the optimal time is at least 2 seconds. Therefore, at least for the tested input condition ranges and parameter ranges, we can conclude that this model gives good results. (The cases where the data indicates the model might not be a good fit are cases where our data is too unreliable to prove or disprove the statement.)

There are, however, trends in the data that are not captured by the above model. Correctly captured is the algorithm choice trend already analysed in Section 3.5.1, but looking at the data in Table A.1, we see that as the matrix size increases, the optimal ratio of matrix size to `jobblocksize` increases as well; this can be explained by noticing that the larger the matrices are, the less overhead results from creating many jobs, so the the better use can be made from the concurrency offered by the allocated threads. With more data, we expect to see this trend continue, and expect that a better model would incorporate this trend as well.

Additionally, it can be expected that as the bitsize increases, the optimal `baseblocksize` should decrease: after all, the reason why `baseblocksize` $= 1$ is far from optimal at these bitsizes is that element multiplications are not sufficiently more expensive than element additions. Looking at the data in Table A.1, this trend is visible by looking at the `baseblocksize` values for larger matrices for increasing bitsizes. However, with the chosen parameter range for `baseblocksize` (i.e. ≥ 16), this trend is not significant enough to consider a conclusion.

Finally, we note that the results above should be correct in the general sense, but the specific boundary values may vary because of the inaccuracies in the measurement results.

3.5.3 Comparison to Existing CPU Implementations

As mentioned in Section 3.1, there are existing libraries that can perform, among other operations, large-integer matrix multiplication on the CPU. Examples are the C library FLINT [11], the C++ library NTL [23] and the C library and programmable calculator PARI/GP [10]. In this section, we will compare the performance of our implementation and these three implementations (in this section: the *alternatives*) on a small set of test cases: for each combination of (square) matrix size in the set $\{128, 256, 512, 1024\}$ and bitsize in the set $\{512, 1024\}$, each of the following programs is run and the multiplication timing recorded.

- A C program using FLINT that reads in two large-integer matrices into variables of type `fmpz_mat_t` and multiplies those with the function `fmpz_mat_mul`. The call to `fmpz_mat_mul` is timed.
- A C++ program using NTL that reads the matrices into variables of type `Mat<ZZ>` and multiplies those with the function `mul(mat_ZZ&, const mat_ZZ&, const mat_ZZ&)` from the `NTL/mat_ZZ.h` header file. The call to `mul` is timed.
- A GP script that reads in the matrices from two files and then multiplies them using the statement `C = A * B;`. (The matrices are read in using `matconcat(readvec("file.txt")~);`.) PARI/GP is usable both as a library (`libpari`) and as a scripting language (GP); we used the scripting language here. Since in GP, the multiplication of the two matrices can be performed with a single operator, we do not believe that using `libpari` would give a significant speedup for the matrix multiplication compared to GP, apart from a small constant-time overhead. Possible overheads for reading in the input matrices are not problematic, since only the multiplication is timed, like in the other tests. To be able

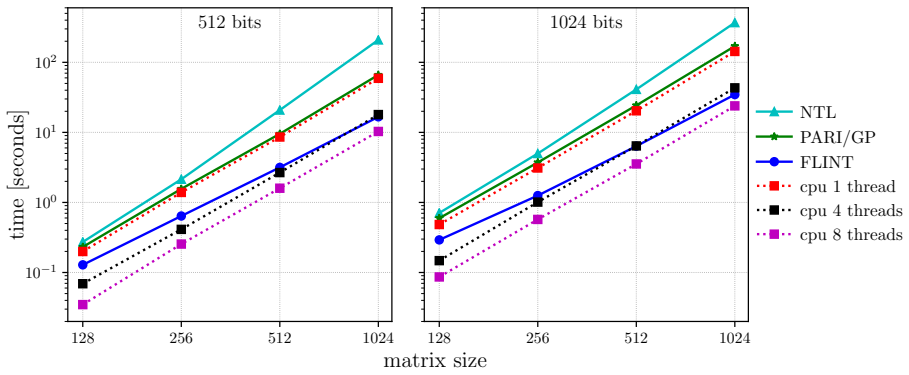

Figure 3.2: (log–log) Average timings over three runs for different implementations with the test cases described in Section 3.5.3. "cpu" refers to our CPU implementation.

to multiply the largest matrices in this test, the PARI memory usage limit (`parisize`) is raised to 8'000'000'000.

- Our CPU implementation, using 1 thread, parameter set `Wi 16 /1` (i.e. the Winograd algorithm, `baseblocksize` = 16 and `jobblocksize` = $\frac{n}{1} = n$, where n is the matrix size). As in all other tests in this thesis, only the multiplication is timed.
- Our CPU implementation, using 4 threads, parameter set `nW 16 /2`.
- Our CPU implementation, using 8 threads, parameter set `nW 16 /2`.

The measurements are performed on the same machine as the main CPU tests, which is the first machine described in Section 3.4. The versions used are FLINT 2.5.2, NTL 11.2.0 and PARI/GP 2.9.5. Each measurement was performed 6 times, and the maximum relative and absolute errors can be found in Table A.7. (The measurement errors do not influence the rankings between implementations.) The average values are plotted in Figure 3.2, where our program has dotted lines and the alternatives have unbroken lines. Note that the tests for the alternatives are all single-threaded.

The relative performance of our implementation for different thread counts is in line with the results earlier in Section 3.5. Below we will compare the alternative implementations to our implementation in terms of performance and algorithm.

FLINT is faster than our implementation on a single thread, but when using multiple threads, we overtake FLINT on the bitcounts tested. However, the graph appears to show that FLINT uses an asymptotically better algorithm than our implementation. On inspection of the FLINT 2.5.2 source code, the function `fmpz_mat_mul` switches to multi-modular matrix multiplication for matrix sizes larger than a certain bound depending on the bitsize; in the cases tested here, multi-modular multiplication is always used. This is performed in the function `_fmpz_mat_mul_multi_mod`, which generates a list of sequential primes (say $p_1, \ldots, p_N$) and then builds N new pairs of matrices, the i'th pair being the original input matrices with elements reduced modulo p_i. Using N matrix multiplications, these pairs are multiplied, and the element at position (i, j) in the result matrix is then reconstructed from the elements at position (i, j) in the N calculated matrices. This reconstruction is possible using the Chinese Remainder Theorem, because the moduli are pairwise coprime. The N matrix multiplications over smaller integers (in FLINT, N is chosen such that these are typically about two limbs large) are performed using Winograd's algorithm. In addition, when a modular matrix multiplication has matrix size ≥ 20, the second matrix is first transposed before the actual multiplication is performed.

Since FLINT uses the same matrix multiplication algorithm as our implementation, i.e. Winograd,

the only algorithmic difference (thus discarding constant-factor optimisations) is using the multi-modular technique to gain small and constant-size matrix elements at the cost of pre- and post-processing work quadratic in the matrix size and having to perform $O(\text{bitsize})$ matrix multiplications instead of 1. Despite being faster for the tested sizes in practice, this should only improve the algorithmic complexity with respect to the bitsize, not the matrix size.

NTL is slower than our implementation on a single thread, and the distance increases as the matrix size grows in the cases tested here. On inspection of the NTL 11.2.0 source code, only naive matrix multiplication is performed without Strassen or multi-modular optimisations. This means that performance should scale as $O(n^3)$, where n is the matrix size, instead of $O(n^{2.81})$ like our implementation does using a Strassen-like algorithm. This explains the difference in performance shown in the graph.

PARI/GP scales similarly to our single-threaded code, being on average about 17% slower. (A constant difference on a log-plot is a constant ratio between the y-values.) Inspecting the PARI/GP 2.9.5 source code, the path taken for matrices over large integers uses Winograd's algorithm without further optimisations, which the same as our implementation. The measured speed difference can probably be attributed to invoking the PARI garbage collector between sub-multiplications to clean up temporaries.

Of the three alternatives listed, only PARI/GP has an implementation comparable to ours, and compared to PARI/GP our implementation is about 17% faster for the matrix sizes and bitsizes tested in this section. Fortunately, our implementation is also faster than NTL, which does not use a Strassen-like algorithm. On the other hand, the multi-modular multiplication used in FLINT is superior to our code. On the "Development" page on the FLINT website [11], the authors list that implementing multithreaded matrix multiplication is a possible enhancement; based on the results above, we expect that this will be a very effective speedup.

3.6 GPU Results

In this section, an *input condition* is a pair of a matrix size and a bitsize, and a *parameter set* is a combination of a chosen algorithm, whether it is in interleaved mode, the GPU thread block width (`gputhreadblock_width`) and the GPU thread block height (`gputhreadblock_height`). In the GPU tables and in the implementation, being in interleaved mode is a property of the algorithm, so in this case there will generally be four algorithms (Strassen, Winograd, Strassen strided and Winograd strided — strided meaning "interleaved mode"); with this convention, there are three components to a parameter set (algorithm, width and height).

As before, for each input condition, each possible parameter set was enumerated and a matrix multiplication was executed with this input condition and parameter set. A *test case* is again a pair of an input condition and a parameter set. The measured time is the entire multiplication including possible pre- or post-processing for interleaving, but without I/O. Each test case of which the original timing was less than 2 seconds was re-run twice, and of the other test cases 20% was randomly selected to be re-run twice as well. The optimal parameter sets are in Table B.1, the corresponding timings in Table B.2 and the maximum relative errors of the re-runs compared to the original run in Table B.3. Short notation for parameter sets will be e.g. "Wi 64 2", meaning the Winograd algorithm with GPU thread block size 64×2. (The algorithms Strassen, Winograd, Strassen strided and Winograd strided are respectively indicated using St, Wi, SS and WS.) Looking at Tables B.2 and B.3, it should be noted that for input conditions with optimal timings of at least 2 seconds, the relative re-run errors are all less than 1%, giving confidence that these values are very reliable. However, the data for the other input conditions, and especially those with timings less than 10ms, is too unreliable to use.

Bitsize	16	64	256	512	1024	2048
gld_efficiency	46.22%	44.50%	34.57%	27.83%	22.12%	18.19%
gst_efficiency	43.01%	46.60%	49.34%	50.31%	51.38%	52.96%

Table 3.1: Memory efficiency of Winograd strided, as reported by ***nvprof****.*

Because we focus on one GPU only, there are no parallelism or memory usage motivations to prefer Strassen over Winograd; since Winograd uses fewer additions than Strassen, it therefore seems unlikely that it is beneficial to use Strassen at all. In Table B.1, there are a few input conditions where a parameter set with the Strassen algorithm is optimal, but this is deceptive: when restricting the parameter space to only Winograd-based algorithms (i.e. Winograd and Winograd strided), the relative error of the new optimal parameter set compared to the original optimal set does not exceed 3% on all input conditions, and does not exceed 0.7% on the reliable subset. These relative errors are shown in Table B.4. We conclude that we can indeed disregard Strassen-based algorithms and focus only on Winograd-based algorithms.

GPU thread block size optimisation is in general a non-trivial problem [26], but in this case we find that, in many cases but within certain bounds, the thread block size has little impact on performance. For each combination of matrix size, bitsize and algorithm choice, we compare the timing for the optimal block size with each of the tested block sizes for that combination. The fraction of block sizes that is then within 3% of the optimal block size is shown in Table B.5. Since each combination has 21 possible block sizes, we see that in each of the combinations, at least 3 block sizes yield timings within 3% of the optimal timing for that combination. On the other hand, in most other cases, half or more of the possible block sizes show no significant (here: $> 3\%$) slowdown compared to the optimal block size.

To find a good block size to use, we consider the parameter set (i.e. algorithm and block size) with the least maximum relative error over all the input conditions. This parameter set is Winograd with block size 8×4, with a maximum relative error of 17%, which occurs at bitsize 16 and matrix size 2048. When excluding this input condition from the maximum relative error calculation, this parameter set is still optimal with a maximum relative error of 8%. The relative errors of this parameter set (Wi 8 4) are shown in Table B.6.

Especially for small bitsizes, Wi 8 4 works very well, but the data indicates that for larger bitsizes this block size is suboptimal: in Table B.6, relative errors increase as the bitsize increases. Indeed, choosing Wi 32 2 or Wi 32 16 improves performance for larger bitsizes, as suggested by the optimal sets in Table B.1, but the collected data is not sufficient to give confidence about these alternate block sizes.

3.6.1 Analysis

For the non-interleaved algorithms, the GPU profiler `nvprof` reports very low memory bandwidth efficiency (12.5% for global loads and stores) because memory accesses cannot be effectively coalesced. To remedy this, the interleaved variants were made, and indeed the memory efficiency is higher with these variants, but not as much as expected, and the improvement varies with bitsize.

In Table 3.1, the global memory load and store efficiency (respectively `gld_efficiency` and `gst_efficiency`) as reported by `nvprof` for the Winograd strided algorithm are shown for various bitsizes. While it is clear that these figures are better than the 12.5% for plain Winograd, the cause for this suboptimality has not been found; the NVIDIA Visual Profiler (`nvvp`) does not give insight as to where the non-coalesced memory accesses occur.

Added to this suboptimality is the need to pre-process the matrix memory on the CPU to

Metric name	Description	Min	Max	Avg
issue_slot_utilization	Issue Slot Ut.	9.79%	9.79%	9.79%
tex_utilization	Unified Cache Ut.	Low (1)	Low (1)	Low (1)
l2_utilization	L2 Cache Ut.	Low (3)	Low (3)	Low (3)
shared_utilization	Shared Memory Ut.	Idle (0)	Idle (0)	Idle (0)
ldst_fu_utilization	Load/Store Function Unit Ut.	Low (1)	Low (1)	Low (1)
cf_fu_utilization	Control-Flow Function Unit Ut.	Low (1)	Low (1)	Low (1)
special_fu_utilization	Special Function Unit Ut.	Low (1)	Low (1)	Low (1)
tex_fu_utilization	Texture Function Unit Ut.	Low (2)	Low (2)	Low (2)
single_precision_fu_utilization	Single-Precision Function Unit Ut.	Low (1)	Low (1)	Low (1)
double_precision_fu_utilization	Double-Precision Function Unit Ut.	Idle (0)	Idle (0)	Idle (0)

*Table 3.2: **nvprof** metrics for 256×256 matrices, bitsize 512, and its optimal parameter set Wi 16 32. The metrics using WS 16 32 only differ in issue slot utilisation, which becomes 9.10%.*

interleave the numbers before running the GPU multiplication algorithm, and to post-process the result to deinterleave the product matrix elements. This incurs an $O(n^2)$ cost (as opposed to the multiplication algorithm with cost between $O(n^{2.81})$ and $O(n^3)$), so theoretically, as the matrix size increases, this CPU work should become less relevant, and favourability of the interleaved algorithms should increase. Together with the memory efficiencies reported in Table 3.1, it is not surprising that the one tested input condition for which an interleaved algorithm is optimal, is the largest matrix size ($n = 2048$) with the smallest bitsize ($b = 16$) (see Table B.1). If the input matrices have integers of at most 16 bits, a better approach would clearly be to use native data types instead of turning to a big integer library; but we note that the memory efficiency does not decline much yet at 64 bits, where using native data types does not suffice anymore. (Current GPU's do not have a native 256-bit integer type, and product matrix elements can get larger than 128 bits with 64-bit inputs).

In Table B.7, we see the effect described above: interleaved algorithms are more favourable for smaller bitsizes, but also for larger matrix sizes. We predict that for even larger matrix sizes than tested here (and small bitsizes), interleaved algorithms will eventually overtake non-interleaved versions.

3.6.2 GPU Performance Bottleneck

Results from `nvvp` indicate that the matrix multiplication kernel is limited by arithmetic and memory latency. This is evidenced by the metrics collected using `nvprof`, shown in Table 3.2, and the fact that `nvvp` reports that in "PC sampling" (i.e. frequently sampling the current instruction execution state), 88.7% of the time, the current instruction is stalled on a memory dependency, and in a further 5.6% of the time the current instruction is stalled on an execution dependency. The low issue slot utilisation and function unit utilisation in Table 3.2 also point to stalls as a performance inhibitor, reinforcing the results from `nvvp`. This means that memory latency is the largest bottleneck of the GPU kernel, with arithmetic latency a much smaller but still present problem.

Methods for reducing memory latency issues may be further improving memory access coalescing, or making use of shared memory on the GPU, which is faster but smaller than global memory. Arithmetic latency is most likely caused by an insufficient number of integer multipliers on the GPU that was used; because most current GPU's are aimed at consumer gaming or machine learning, the most abundant functional units are floating-point arithmetic units. These could potentially be used by further reducing the large integer limb size such that the limbs, and potentially their products, fit in the mantissa part of a single-precision floating-point number. However, reducing the limb size also carries a performance penalty, if only a constant factor.

Matrix sizes	Bitsize	CPU 1 thread	CPU 2 threads	CPU 4 threads	CPU 8 threads
$\langle 768, 790, 1023\rangle$	1024	nW 16 512 (102.0)	nW 16 512 (53.34)	Wi 16 512 (28.91)	Wi 16 128 (16.32)
$\langle 445, 271, 1034\rangle$	500	Wi 16 128 (8.427)	nW 16 512 (4.611)	Wi 32 512 (2.627)	nW 32 512 (1.525)
$\langle 221, 598, 1468\rangle$	1000	nW 16 512 (37.53)	Wi 16 128 (20.35)	Wi 16 256 (12.17)	nW 16 128 (7.905)
$\langle 1466, 800, 841\rangle$	1500	nW 16 512 (259.5)	nW 16 512 (132.9)	Wi 16 256 (69.41)	Wi 16 128 (38.43)
$\langle 679, 1128, 734\rangle$	2000	Wi 16 128 (244.1)	Wi 16 256 (125.3)	Wi 16 256 (65.27)	Wi 16 256 (34.7)

Matrix sizes	Bitsize	GPU
$\langle 768, 790, 1023\rangle$	1024	Wi 16 32 (34.77)
$\langle 445, 271, 1034\rangle$	500	Wi 16 2 (9.502)
$\langle 221, 598, 1468\rangle$	1000	St 8 4 (11.9)
$\langle 1466, 800, 841\rangle$	1500	St 8 4 (117.2)
$\langle 679, 1128, 734\rangle$	2000	Wi 8 8 (93.74)

Table 3.3: The matrix sizes and bitsizes used for irregular matrix testing, and optimal parameter sets and timings (in seconds). CPU: algorithm, `baseblocksize`, `jobblocksize`, (timing); GPU: algorithm, width & height of thread block, (timing).

3.7 Results for Irregular Matrices

As mentioned in Section 3.1.1, we should separately look at matrices that are not square with a power of 2 side length. Such matrices will be called *irregular*. The hypothesis is that even with the simplistic implementation of the *squarify* function described in Section 3.3.2, extrapolation of optimal parameter sets and timings from measurement results with input conditions close to the irregular multiplication in question will give a good indication of performance.

First, we note again that for non-square matrix multiplication, the matrix sizes can be described fully with the triple of integers $\langle k, m, n\rangle$, denoting matrix multiplication of a $k \times m$ and an $m \times n$ matrix to produce a $k \times n$ matrix.

To make the hypothesis plausible, five matrix size triples were selected, the first to combine some favourable and unfavourable numbers, and the other four randomly. Five bitsizes were chosen, and the resulting pairs were tested on the CPU for 1, 2, 4 and 8 threads and all parameter sets, and on the GPU for all parameter sets. The inputs chosen with their optimal parameter sets and corresponding timings are shown in Table 3.3.

In the CPU data, it is immediately clear that the Winograd algorithm has better results for larger numbers of threads than the previous sections. This is because the squarify operation already introduces sufficient parallelism into the computation, and no more is needed to use the threads adequately.

When comparing the optimal times in Table 3.3 with Tables A.2 and B.2, it can be verified that these are in the expected ranges: e.g. for $\langle 1466, 800, 841\rangle$ with bitsize 1500, we expect the runtime to be between those for $b = 1024$ and $b = 2048$, on the row of $n = 1024$. Indeed, this holds for all CPU thread counts and for the GPU.

The `baseblocksize` parameter for the CPU conforms to the behaviour extrapolated from Table A.1: for smaller bitsizes, larger values of `baseblocksize` are optimal, while its optimal value decreases quickly for larger bitsizes. For bitsizes at least 1000, the optimal value is always 16, like in Table A.1. The `jobblocksize` parameter is more volatile than maybe expected, but when restricting the data to the records for which `jobblocksize` $= 256$, the maximum relative error is less than 5%. This indicates that for matrix sizes in the range sampled here, choosing the value 256 does not hurt performance significantly.

For the GPU, the optimal runtimes are as expected from Table B.2 as mentioned, but the algorithms and block sizes do not all correspond to those in the correct neighbourhoods of Table B.1. However, as before, the optimal parameter sets here can be deceptive: for the last four samples, the parameter sets expected from Table B.1 are Wi 16 32, Wi 32 16, Wi 32 16 and Wi 32 2, respectively, and these all have relative error with the optimal choice less than 5%.

Therefore, we can conclude that the data for square matrices with size a power of 2 can provide a good indication for the parameter sets for irregular matrix sizes.

3.8 Comparison of Results

Comparing Table A.2 and Table B.2, for small bitsizes GPU performance is comparable to 4-thread CPU performance, while this drops to below 2-thread CPU performance for larger bitsizes (≥ 1024). This indicates that when the hardware offers enough parallelism, the CPU implementation is more performant than the GPU implementation.

As noted in Section 3.6.2, the bottleneck of the GPU implementation is principally memory latency. Since we do not use shared memory in the GPU code, all data is stored in global memory, which is only cached in the GPU L2 cache, not the faster L1 cache [17, §9.2 Device Memory Spaces]. By contrast, the CPU implementation freely uses all layers of CPU cache available, as a result of the design of modern CPU's. In addition, memory accesses are not optimally coalesced in the GPU code, even in the interleaved versions; in the CPU version, the memory infrastructure does not assume such coalescing, and may therefore cope better with many separate memory accesses. The fact that modern CPU's have intelligent memory prefetch units, while GPU's typically do not employ such techniques [16], also speaks in the favour of a CPU platform for this approach to large-integer matrix multiplication.

3.9 Conclusion

We have optimised parameter sets for each combination of matrix size, bitsize and thread count on the CPU, and each combination of matrix size and bitsize on the GPU. For the CPU, it has become clear that for larger thread counts, using 8-fold parallelism using naive recursion scales well, while using an asymptotically faster recursion algorithm can be better for small thread counts. The total number of 7-fold recursion steps should increase when element multiplications become more expensive (i.e. when the bitsize increases), and the number of recursion steps distributed over threads should increase as the number of threads increases.

For the GPU, using Winograd's algorithm has proved to be the best choice in most circumstances. It remains unclear exactly which GPU thread block sizes are optimal, but good indications have been given, with a thread block size of 8×2 working well for a large part of the input data set. It is expected, but not proven, that larger block sizes like 32×2 and 32×16 become optimal as the input grows, both in matrix size and in bitsize. The GPU performance bottleneck has been identified as memory latency, with arithmetic latency being second but much less important.

For irregular matrix sizes, the results generally coincide with those for square matrices with power of 2 sizes, and the same recommendations perform well. The exact performance characteristics of the implementations in this case have not been studied.

The aim of Section 3 was to describe and evaluate implementations for large-integer matrix multiplication and determine the most suitable platform. In Section 3.8 we have seen that in this case, a multicore CPU platform is the most suitable for large-integer matrix multiplication.

3.10 Future Research

As in all theses, many possibilities and research directions lay yet unexplored. For Section 3 specifically, the following optimisations, extensions and points of further study may still prove fruitful but have not been applied here due to time constraints.

- Using the Chinese Remainder Theorem, or "multi-modular reduction", to reduce the single matrix multiplication over large integers to many matrix multiplications over native integers, is sometimes more performant than the approach taken here. When using this method, fast floating-point GPU matrix multiplications kernels (e.g. GPU8 in [14]) may be adapted to use integers, since native integers are equally large as single-precision floating-point numbers on a current GPU.
- The GPU implementation does not make use of shared memory, which is a small amount of memory local to each GPU thread block that is generally faster than main GPU memory, called "global memory". However, it is not large enough to store a significant piece of the matrices, so a selection will have to be made.
- The Strassen and Winograd algorithm implementations for the CPU could be parallelised more by using more temporary matrices and thus more memory. The current implementation was chosen to support running on a more resource-constrained system than the final tests were run on. Even though it is expected that their 7-fold parallelism will not be as efficient as the 8-fold parallelism of the naive/Winograd algorithm in many-core scenarios, this is worth testing.
- In Section 3.8, conjectures have been made as to the reason for performance difference between the GPU and CPU implementations; a more thorough study is needed to identify the reasons that are most relevant in practice.
- It is still unclear how to select parameter sets when given very asymmetric input matrix sizes. This would require a more extensive study of different kinds of matrix sizes, and an in-depth study of the wrapper function to convert to square multiplications. Alternatively, Strassen-like algorithms can be adapted for non-square inputs; the performance implications of this can be studied.
- Loop blocking increases memory locality and might be applied in the GPU kernels to optimise cache usage. This could be combined with the above shared memory optimisation, if shared memory is large enough to contain one such block of a matrix. Loop blocking might not yield much improvement on the CPU because `baseblocksize` is already small. However, for small caches, an improvement may be observed.
- Using a job scheduling system like that used for the CPU implementation might enable the usage of multiple GPU's. It should be kept in mind that the use of many GPU's can make Strassen-like algorithms less attractive [30].

4 Reflection

We have looked at large-integer matrix multiplication both from the theoretical side, where we have set out results from complexity theory to prove a bound on the exponent ω (see Definition 2.6.3) in the theoretical complexity of matrix multiplication, and from the practical side, where we have discussed two implementations and evaluated their performance. Both times, we have tried to make progress on the question of how to efficiently multiply matrices over large integers, but the results are of a very different character.

In Section 2, the theoretical bound on the complexity of the problem does not yield an actual algorithm for performing the computation with that asymptotic performance. In fact, Arnold Schönhage writes the following in his article introducing the τ-theorem: [21, p. 448]

> It is a remarkable fact that the preceding argument has led to an upper bound for the complexity of matrix multiplication though no algorithm has been specified, not even implicitly.

On the other hand, when creating practical implementations in Section 3, we have limited ourselves to the more standard algorithm of Strassen, which provides a useful decrease in complexity but does not increase the hidden constant factor by too much. In fact, we have seen that it is useful to consider naive recursion in addition to Strassen recursion to provide a branching factor (namely, 8) that is in line with the number of CPU cores in modern processors.

This is in contrast with the developments in Section 2, where we have ignored all constant factors for the sake of an improvement in complexity. An illustration of this is Theorem 2.6.4, where we show that the potentially considerable amount of work in all other operations, an amount that nevertheless scales only quadratically with n, is dominated by the number of multiplications necessary. To prove this, we have made use of the fact that no algorithm for matrix multiplication ($n > 1$) can perform its task in $O(n^2)$ multiplications (Theorem 2.6.2), allowing the quadratic work to be ignored.

Even if the results of the two parts of this thesis are quite different, we believe to have illuminated both perspectives on the question of large-integer matrix multiplication.

References

[1] A. Ambainis, Y. Filmus, and F. Le Gall. "Fast Matrix Multiplication: Limitations of the Laser Method". In: *ArXiv e-prints* (Nov. 2014). arXiv: 1411.5414 [cs.CC]. URL: https://arxiv.org/abs/1411.5414 (visited on 06/09/2018).

[2] G. Ballard et al. "Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication". In: *ArXiv e-prints* (Feb. 2012). arXiv: 1202.3173 [cs.DS]. URL: https://arxiv.org/abs/1202.3173 (visited on 06/10/2018).

[3] Markus Bläser. *Fast Matrix Multiplication*. Graduate Surveys 5. Theory of Computing Library, 2013, pp. 1–60. DOI: 10.4086/toc.gs.2013.005. URL: http://www.theoryofcomputing.org/library.html.

[4] Henry Cohn and Christopher Umans. "Fast Matrix Multiplication Using Coherent Configurations". In: *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '13. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2013, pp. 1074–1086. ISBN: 978-1-611972-51-1. URL: http://dl.acm.org/citation.cfm?id=2627817.2627894.

[5] D. Coppersmith and S. Winograd. "Matrix Multiplication via Arithmetic Progressions". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, pp. 1–6. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28396. URL: http://doi.acm.org/10.1145/28395.28396.

[6] Niall Emmart. "A Study of High Performance Multiple Precision Arithmetic on Graphics Processing Units". Feb. 2018. URL: https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=2252&context=dissertations_2 (visited on 07/09/2018).

[7] Yuval Filmus. *Matrix Multiplication I (Lecture Notes)*. Feb. 2012. URL: http://www.cs.toronto.edu/~yuvalf/MatMult.pdf (visited on 06/09/2018).

[8] Torbjörn Granlund and contributors. *GMP 6.1.2 Manual: 15.1 Multiplication*. URL: https://gmplib.org/manual/Multiplication-Algorithms.html (visited on 06/09/2018).

[9] Torbjörn Granlund and contributors. *The GNU Multiple Precision Arithmetic Library*. URL: https://gmplib.org/ (visited on 06/10/2018).

[10] The PARI Group. *PARI/GP*. URL: https://pari.math.u-bordeaux.fr (visited on 06/10/2018).

[11] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. URL: http://flintlib.org (visited on 06/10/2018).

[12] Koji Kitano and Noriyuki Fujimoto. "Multiple Precision Integer Multiplication on GPUs". In: *2014 International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, Nevada, US: CSREA Press, 2014, pp. 236–242. ISBN: 1-60132-284-4.

[13] François Le Gall. "Powers of Tensors and Fast Matrix Multiplication". In: *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. ISSAC '14. Kobe, Japan: ACM, 2014, pp. 296–303. ISBN: 978-1-4503-2501-1. DOI: 10.1145/2608628.2608664. URL: http://doi.acm.org/10.1145/2608628.2608664.

[14] J. Li, S. Ranka, and S. Sahni. "Strassen's Matrix Multiplication on GPU's". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. Dec. 2011, pp. 157–164. DOI: 10.1109/ICPADS.2011.130.

[15] Hao Jun Liu and Chu Tong. *GMP implementation on CUDA - A Backward Compatible Design With Performance Tuning*. Tech. rep. URL: http://individual.utoronto.ca/haojunliu/courses/ECE1724_Report.pdf (visited on 07/09/2018).

[16] Nuno Neves, Pedro Tomás, and Nuno Roma. "Stream data prefetcher for the GPU memory interface". In: *The Journal of Supercomputing* 74.6 (June 2018), pp. 2314–2328. ISSN: 1573-0484. DOI: 10.1007/s11227-018-2260-6. URL: https://doi.org/10.1007/s11227-018-2260-6.

[17] NVIDIA. *CUDA v9.2.88 Best Practices Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (visited on 06/13/2018).

[18] NVIDIA. *CUDA v9.2.88 Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 06/13/2018).

[19] Petr Petrouš. "CUDA implementation of the GMP library." Feb. 2016. URL: https://dspace.cvut.cz/bitstream/handle/10467/65185/F8-DP-2016-Petrous-Petr-thesis.pdf (visited on 07/08/2018).

[20] R. Probert. "On the Additive Complexity of Matrix Multiplication". In: *SIAM Journal on Computing* 5.2 (1976), pp. 187–203. DOI: 10.1137/0205016. URL: https://doi.org/10.1137/0205016.

[21] A. Schönhage. "Partial and Total Matrix Multiplication". In: *SIAM Journal on Computing* 10.3 (1981), pp. 434–455. DOI: 10.1137/0210032. URL: https://doi.org/10.1137/0210032.

[22] A. Schönhage and V. Strassen. "Schnelle Multiplikation großer Zahlen". In: *Computing* 7.3 (Sept. 1971), pp. 281–292. ISSN: 1436-5057. DOI: 10.1007/BF02242355. URL: https://doi.org/10.1007/BF02242355.

[23] Victor Shoup. *NTL: A Library for doing Number Theory*. URL: http://www.shoup.net/ntl (visited on 06/10/2018).

[24] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. ISSN: 0945-3245. DOI: 10.1007/BF02165411. URL: https://doi.org/10.1007/BF02165411.

[25] Volker Strassen. "Vermeidung von Divisionen". ger. In: *Journal für die reine und angewandte Mathematik* 264 (1973), pp. 184–202. URL: http://eudml.org/doc/151394.

[26] Vasily Volkov. "Better Performance at Lower Occupancy". In: vol. 10. Sept. 2010. URL: http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf (visited on 06/20/2018).

[27] Virginia Vassilevska Williams. "Multiplying Matrices Faster Than Coppersmith-Winograd". In: *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*. STOC '12. New York, New York, USA: ACM, 2012, pp. 887–898. ISBN: 978-1-4503-1245-5. DOI: 10.1145/2213977.2214056. URL: http://doi.acm.org/10.1145/2213977.2214056.

[28] Virginia Vassilevska Williams. *Multiplying Matrices in $O(n^{2.373})$ time*. URL: http://theory.stanford.edu/~virgi/matrixmult-f.pdf (visited on 06/09/2018).

[29] Shmuel Winograd. "On Multiplication of 2×2 Matrices". In: *Linear Algebra and its Applications* 4.4 (Oct. 1971), pp. 381–388. ISSN: 0024-3795. DOI: 10.1016/0024-3795(71)90009-7. URL: https://doi.org/10.1016/0024-3795(71)90009-7.

[30] Peng Zhang and Yuxiang Gao. "Matrix Multiplication on High-Density Multi-GPU Architectures: Theoretical and Experimental Investigations". In: *High Performance Computing*. Ed. by Julian M. Kunkel and Thomas Ludwig. Cham: Springer International Publishing, 2015, pp. 17–30. ISBN: 978-3-319-20119-1.

A Appendix: CPU Tables

This section contains tables that were considered too large to fit in the main text.

$b = 16$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	nW 16 /2	nW 16 /2	nW 16 /2	St 16 /2	St 16 /2	St 16 /2
$n = 64$	Wi 16 /2	nW 16 /2	nW 16 /2	Wi 32 /2	nW 16 /2	nW 32 /2
$n = 128$	Wi 32 /4	Wi 32 /4	nW 16 /2	nW 32 /2	nW 32 /2	nW 32 /2
$n = 512$	Wi 32 /2	nW 32 /2	nW 32 /2	nW 32 /2	nW 32 /2	nW 64 /2
$n = 1024$	Wi 32 /8	St 32 /4	nW 32 /2	nW 32 /2	nW 32 /2	nW 32 /2
$n = 2048$	Wi 32 /4	nW 32 /2	nW 32 /2	nW 32 /2	nW 64 /4	nW 32 /2

$b = 512$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	Wi 16 /2	Wi 16 /2	St 16 /2	nW 16 /2	nW 16 /2	nW 16 /2
$n = 64$	Wi 16 /2	nW 16 /2	nW 16 /4	nW 16 /2	nW 32 /2	nW 32 /2
$n = 128$	Wi 16 /2	nW 16 /2	nW 16 /2	nW 32 /2	nW 32 /2	nW 32 /2
$n = 512$	Wi 32 /2	nW 16 /2	nW 16 /2	nW 16 /2	nW 32 /4	nW 64 /8
$n = 1024$	Wi 16 /4	St 16 /4	nW 16 /2	nW 16 /2	nW 64 /4	nW 64 /8
$n = 2048$	Wi 16 /2	St 16 /4	nW 16 /2	nW 16 /2	nW 16 /4	nW 32 /4

$b = 1024$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	nW 16 /2	St 16 /2	St 16 /2	nW 16 /2	nW 16 /2	nW 16 /2
$n = 64$	Wi 16 /2	St 16 /2	nW 16 /2	nW 16 /2	nW 16 /2	nW 16 /2
$n = 128$	Wi 16 /2	nW 16 /2	nW 16 /2	nW 32 /2	nW 32 /2	nW 16 /2
$n = 512$	Wi 16 /4	St 16 /8	nW 16 /2	nW 16 /2	nW 16 /4	nW 16 /4
$n = 1024$	Wi 16 /8	St 16 /8	nW 16 /2	nW 16 /2	nW 16 /4	nW 32 /8
$n = 2048$	Wi 16 /4	St 16 /4	nW 16 /2	nW 16 /2	nW 16 /4	nW 16 /8

$b = 2048$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	Wi 16 /2	nW 16 /2	nW 16 /2	nW 16 /2	nW 16 /2	nW 16 /2
$n = 64$	Wi 16 /2	nW 16 /2	nW 16 /2	nW 32 /2	nW 16 /2	nW 16 /2
$n = 128$	Wi 16 /2	St 16 /4	nW 16 /2	nW 16 /2	nW 16 /2	nW 16 /2
$n = 512$	St 16 /8	St 16 /4	nW 16 /2	nW 16 /2	nW 16 /4	nW 16 /4
$n = 1024$	Wi 16 /8	St 16 /4	St 16 /8	nW 16 /2	nW 16 /4	nW 16 /4
$n = 2048$	Wi 16 /2	St 16 /4	St 16 /8	nW 16 /2	nW 16 /4	nW 16 /4

Table A.1: Optimal parameter settings for CPU. Each table concerns a specific bitsize, each row concerns a matrix size, and each column concerns a thread count. The parameter values are formatted as "`algorithm baseblocksize jobblocksize`*", where* ***`algorithm`*** *is* ***St****rassen,* ***Wi****nograd or* ***n****aive/****W****inograd. A* ***`jobblocksize`*** *of "/k" should be read as* $\frac{n}{k}$*, where n is the matrix size for that row.*

$b = 16$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.003056	0.00255	0.002002	0.002969	0.002888	0.003565
$n = 64$	0.011026	0.006833	0.005297	0.005837	0.005498	0.005367
$n = 128$	0.048697	0.035726	0.022926	0.012277	0.012482	0.013563
$n = 512$	1.99084	1.17213	0.632578	0.393434	0.400736	0.428842
$n = 1024$	12.7539	7.49704	4.15086	2.42802	2.46425	2.43809
$n = 2048$	82.3275	49.5242	26.3804	14.9801	14.7321	15.2245

$b = 512$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.009629	0.004915	0.004305	0.001568	0.0021	0.002634
$n = 64$	0.035638	0.024637	0.013188	0.006706	0.006044	0.011652
$n = 128$	0.192117	0.122225	0.065441	0.032512	0.035059	0.034878
$n = 512$	8.64574	5.49531	2.87114	1.64548	1.31009	1.15675
$n = 1024$	59.9396	33.1425	17.6512	10.1048	8.08015	7.73611
$n = 2048$	405.977	230.506	125.554	72.3295	49.7583	66.8393

$b = 1024$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.014214	0.011	0.008814	0.003908	0.002584	0.006109
$n = 64$	0.072059	0.047888	0.027993	0.013369	0.014793	0.015712
$n = 128$	0.455301	0.264321	0.135949	0.078247	0.081889	0.078164
$n = 512$	19.1484	11.6325	5.94216	3.42747	2.75695	2.61307
$n = 1024$	131.2	75.7815	40.5061	22.5626	16.4799	18.9358
$n = 2048$	930.033	493.231	282.409	155.974	105.194	142.379

$b = 2048$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.03565	0.023163	0.01553	0.00509	0.005439	0.009402
$n = 64$	0.185363	0.113432	0.062952	0.033163	0.033515	0.031824
$n = 128$	1.26826	0.681322	0.37713	0.19516	0.21351	0.202236
$n = 512$	54.3766	29.2129	15.8977	8.84537	6.22498	5.83046
$n = 1024$	368.219	196.092	104.767	59.6969	41.7866	47.8295
$n = 2048$	2579.94	1333.49	736.926	415.238	270.013	305.14

Table A.2: Timings in seconds for each input condition and the parameter settings in Table A.1.

$b = 16$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.718 (6)	0.474 (6)	1.42 (6)	0.579 (6)	0.482 (6)	1.18 (6)
$n = 64$	0.821 (18)	0.877 (18)	0.879 (18)	0.788 (18)	2.66 (18)	0.508 (18)
$n = 128$	0.2 (36)	0.185 (36)	0.226 (36)	0.466 (36)	0.4 (36)	0.374 (36)
$n = 512$	0.0422 (12)	0.114 (52)	0.0825 (54)	0.137 (54)	0.399 (54)	0.148 (54)
$n = 1024$	0.156 (18)	0.0575 (6)	0.135 (12)	0.0702 (8)	0.102 (10)	0.151 (14)
$n = 2048$	0.122 (12)	0.0517 (10)	0.0708 (20)	0.0962 (12)	0.0373 (10)	0.0719 (8)

$b = 512$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.372 (6)	1.13 (6)	0.99 (6)	0.623 (6)	0.464 (6)	0.948 (6)
$n = 64$	0.138 (18)	0.252 (18)	0.387 (18)	0.636 (18)	0.337 (18)	0.64 (18)
$n = 128$	0.135 (36)	0.203 (36)	0.26 (36)	0.25 (36)	0.242 (36)	0.603 (36)
$n = 512$	0.0905 (14)	0.141 (14)	0.0764 (12)	0.0747 (26)	0.0841 (28)	0.196 (20)
$n = 1024$	0.0094 (6)	0.0997 (8)	0.171 (14)	0.106 (14)	0.0654 (8)	0.0106 (2)
$n = 2048$	0.09 (10)	0.0422 (10)	0.0579 (16)	0.0718 (8)	0.021 (4)	0.0514 (14)

$b = 1024$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.453 (6)	0.394 (6)	0.381 (6)	0.448 (6)	1.59 (6)	0.408 (6)
$n = 64$	0.114 (18)	0.243 (18)	0.352 (18)	0.301 (18)	0.279 (18)	0.389 (18)
$n = 128$	0.0665 (36)	0.0672 (36)	0.107 (36)	0.377 (36)	0.276 (36)	0.278 (36)
$n = 512$	0.169 (8)	0.121 (12)	0.136 (10)	0.1 (14)	0.0961 (6)	0.0616 (8)
$n = 1024$	0.139 (10)	0.0869 (12)	0.0632 (8)	0.0885 (18)	0.0756 (16)	0.171 (14)
$n = 2048$	0.0338 (8)	0.035 (8)	0.046 (8)	0.0347 (10)	0.00725 (2)	0.0478 (12)

$b = 2048$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.179 (6)	0.174 (6)	0.126 (6)	0.491 (6)	0.445 (6)	0.524 (6)
$n = 64$	0.0888 (18)	0.0904 (18)	0.167 (18)	0.157 (18)	0.503 (18)	0.291 (18)
$n = 128$	0.0278 (36)	0.147 (36)	0.0705 (36)	0.33 (36)	0.358 (36)	0.474 (36)
$n = 512$	0.146 (16)	0.0891 (12)	0.0401 (6)	0.103 (20)	0.0952 (16)	0.0627 (6)
$n = 1024$	0.0803 (10)	0.0288 (4)	0.0657 (8)	0.0489 (4)	0.0709 (14)	0.101 (16)
$n = 2048$	0.0419 (10)	0.0345 (22)	0.0202 (10)	0.0213 (4)	0.0995 (16)	0.0405 (6)

*Table A.3: For each input condition the format is "`maxrelerr` (`count`)", where **`maxrelerr`** is the the maximum relative error of all re-run testcases compared to the original timings for that input condition, where the relative error between a re-run time t_1 and an original time t_0 is $\frac{|t_1 - t_0|}{t_0}$; and where **`count`** is the number of test cases that this maximum was taken over.*

$b = 16$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.8135	0.8353	0.3142	0.2529	0.4834	1.046
$n = 64$	0.0	0.88	0.5458	1.091	0.3037	1.742
$n = 128$	0.04134	0.115	0.718	1.597	2.234	1.511
$n = 512$	0.05323	0.3045	0.9805	2.133	2.133	1.955
$n = 1024$	0.1196	0.3617	0.984	2.563	2.53	2.552
$n = 2048$	0.1644	0.33	0.9624	2.69	2.539	2.634

$b = 512$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.0	0.0	1.339	2.186	3.009	3.383
$n = 64$	0.0	0.09092	1.251	2.83	2.755	1.084
$n = 128$	0.0	0.2731	0.8672	2.603	2.431	2.416
$n = 512$	0.08925	0.2324	0.9178	2.17	3.227	3.537
$n = 1024$	0.07472	0.2614	0.9454	2.652	3.214	3.604
$n = 2048$	0.0	0.2862	0.951	2.344	4.323	2.664

$b = 1024$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.09118	0.3155	1.091	1.424	3.363	1.438
$n = 64$	0.0	0.1955	1.059	2.266	2.266	2.362
$n = 128$	0.0	0.2387	0.9677	2.381	2.213	2.547
$n = 512$	0.1137	0.2187	0.9443	2.518	3.338	3.385
$n = 1024$	0.0988	0.2855	1.012	2.477	4.259	3.454
$n = 2048$	0.01319	0.3729	0.9683	2.535	4.256	2.846

$b = 2048$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.0	0.3169	0.5563	3.886	3.297	1.609
$n = 64$	0.0	0.2036	0.8044	2.252	2.393	2.471
$n = 128$	0.0	0.3088	1.013	2.746	2.44	2.613
$n = 512$	0.04324	0.4445	1.215	2.767	4.308	4.746
$n = 1024$	0.01073	0.3822	1.072	2.799	4.155	3.549
$n = 2048$	0.0	0.3839	1.053	2.642	4.62	3.928

Table A.4: Relative error of always choosing Wi 16 /2 instead of the optimal parameter set. Coloured are input conditions for which the optimal time is less than 2 seconds.

$b = 16$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.0	0.0	0.0	0.1533	0.5301	0.2429
$n = 64$	0.3483	0.0	0.0	0.2542	0.0	0.4602
$n = 128$	0.1629	0.05856	0.0	0.1934	0.1452	0.2235
$n = 512$	0.1977	0.07044	0.07167	0.00607	0.005028	0.0001399
$n = 1024$	0.3022	0.04306	0.07522	0.06982	0.07782	0.0614
$n = 2048$	0.3889	0.05769	0.05693	0.06186	0.0872	0.04673

$b = 512$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.651	0.1784	0.6318	0.0	0.0	0.0
$n = 64$	0.09369	0.0	0.2683	0.0	0.05692	0.09149
$n = 128$	0.1453	0.0	0.0	0.1046	0.1245	0.0972
$n = 512$	0.1502	0.0	0.0	0.0	0.2034	0.4284
$n = 1024$	0.1868	0.02087	0.0	0.0	0.2335	0.3092
$n = 2048$	0.1993	0.03896	0.0	0.0	0.434	0.08453

$b = 1024$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.0	0.7163	0.05695	0.0	0.0	0.0
$n = 64$	0.1346	0.05189	0.0	0.0	0.0	0.0
$n = 128$	0.1503	0.0	0.0	0.000077	0.2761	0.0
$n = 512$	0.3371	0.07116	0.0	0.0	0.2444	0.2906
$n = 1024$	0.3215	0.07684	0.0	0.0	0.3567	0.1976
$n = 2048$	0.1794	0.1082	0.0	0.0	0.4873	0.1048

$b = 2048$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	0.06188	0.0	0.0	0.0	0.0	0.0
$n = 64$	0.1916	0.0	0.0	0.1493	0.0	0.0
$n = 128$	0.1435	0.06429	0.0	0.0	0.0	0.0
$n = 512$	0.2311	0.05976	0.0	0.0	0.3894	0.4934
$n = 1024$	0.1595	0.06851	0.05529	0.0	0.4362	0.2435
$n = 2048$	0.1744	0.1099	0.05941	0.0	0.5572	0.3712

Table A.5: Relative error of always choosing nW 16 /2 instead of the optimal parameter set. Coloured are input conditions for which the optimal time is less than 2 seconds.

$b = 16$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	—	—	—	—	—	—
$n = 64$	0.3173	1.014	1.135	0.6291	0.03147	2.754
$n = 128$	0.3962	0.3246	0.3086	0.9128	1.833	2.865
$n = 512$	0.353	0.2205	0.2459	0.2016	0.09476	0.8565
$n = 1024$	0.417	0.2149	0.2026	0.3717	0.1946	0.9179
$n = 2048$	0.4989	0.1855	0.207	0.2818	0.02922	0.3716

$b = 512$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	—	—	—	—	—	—
$n = 64$	0.2309	0.02334	0.0	0.06308	0.9247	0.7274
$n = 128$	0.3407	0.1532	0.1737	0.4478	0.4667	1.711
$n = 512$	0.3738	0.06536	0.07925	0.1008	0.0284	0.1816
$n = 1024$	0.3244	0.2157	0.2009	0.1055	0.00262	0.118
$n = 2048$	0.3834	0.2285	0.1509	0.08697	0.0	0.07472

$b = 1024$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	—	—	—	—	—	—
$n = 64$	0.3421	0.02401	0.2414	0.2382	0.5887	1.317
$n = 128$	0.3038	0.1404	0.187	0.2097	0.2695	1.386
$n = 512$	0.309	0.1404	0.1914	0.09351	0.0	0.0
$n = 1024$	0.3439	0.1942	0.1257	0.1082	0.0	0.06605
$n = 2048$	0.3498	0.2613	0.1497	0.1242	0.0	0.005528

$b = 2048$	$C = 1$	$C = 2$	$C = 4$	$C = 8$	$C = 16$	$C = 32$
$n = 32$	—	—	—	—	—	—
$n = 64$	0.2892	0.1061	0.1276	0.1778	0.1048	1.945
$n = 128$	0.2814	0.2136	0.134	0.1674	0.1328	0.9066
$n = 512$	0.4889	0.2652	0.1526	0.1058	0.0	0.0
$n = 1024$	0.3245	0.2455	0.1978	0.1237	0.0	0.0
$n = 2048$	0.3264	0.2731	0.1998	0.1357	0.0	0.0

Table A.6: Relative error of always choosing nW 16 /4 instead of the optimal parameter set. Coloured are input conditions for which the optimal time is less than 2 seconds. Note that with the tested parameter ranges (i.e. `baseblocksize` ≥ 16*), the value* `jobblocksize` $= \frac{32}{4} = 8 < 16$ *is not allowed, so* $n = 32$ *is not included.*

	Relative errors						**Absolute errors [seconds]**					
$b = 512$	cpu 8	cpu 4	cpu 1	flint	ntl	pari/gp	cpu 8	cpu 4	cpu 1	flint	ntl	pari/gp
$n = 128$	0.02	0.29	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00
$n = 256$	0.29	0.04	0.00	0.01	0.08	0.05	0.07	0.01	0.00	0.00	0.18	0.08
$n = 512$	0.02	0.03	0.07	0.09	0.00	0.00	0.04	0.08	0.66	0.30	0.12	0.08
$n = 1024$	0.02	0.01	0.01	0.01	0.10	0.01	0.20	0.34	0.86	0.17	22.82	0.70

$b = 1024$	cpu 8	cpu 4	cpu 1	flint	ntl	pari/gp	cpu 8	cpu 4	cpu 1	flint	ntl	pari/gp
$n = 128$	0.27	0.02	0.00	0.01	0.00	0.01	0.02	0.00	0.00	0.00	0.00	0.00
$n = 256$	0.07	0.04	0.08	0.12	0.02	0.03	0.04	0.04	0.25	0.16	0.13	0.13
$n = 512$	0.01	0.09	0.01	0.01	0.00	0.01	0.03	0.58	0.26	0.07	0.16	0.25
$n = 1024$	0.00	0.01	0.00	0.02	0.09	0.00	0.12	0.80	0.94	0.71	36.32	1.13

Table A.7: Maximum relative and absolute errors (compared to the mean) for the 6 measurement runs of which the means are plotted in Figure 3.2. "cpu n" is our implementation using n threads. Coloured are the cases where the relative error is $\geq 0.1 = 10\%$*.*

B Appendix: GPU Tables

	$b = 16$	$b = 512$	$b = 1024$	$b = 2048$
$n = 32$	Wi 64 2	St 8 2	Wi 8 2	Wi 8 2
$n = 64$	Wi 16 2	St 16 2	St 16 2	St 8 32
$n = 128$	St 32 8	St 8 4	St 8 8	St 32 2
$n = 512$	St 8 64	Wi 32 16	Wi 32 16	Wi 32 2
$n = 1024$	Wi 8 4	Wi 16 32	Wi 32 16	Wi 32 2
$n = 2048$	SS 128 2	Wi 16 32	Wi 32 16	Wi 32 2

Table B.1: Optimal parameter settings for GPU. Each row concerns a specific matrix size and each column concerns a bitsize. The parameter values are formatted as "`algorithm gputhreadblock_width gputhreadblock_height`*", where* ***`algorithm`*** *is* ***S****trassen,* ***Wi****nograd,* ***S****trassen* ***S****trided or* ***W****inograd* ***S****trided. Strided means* ***`interleaved`*** *is true.*

	$b = 16$	$b = 512$	$b = 1024$	$b = 2048$
$n = 32$	0.004856	0.007245	0.013031	0.034703
$n = 64$	0.006273	0.011453	0.023663	0.056074
$n = 128$	0.013789	0.030598	0.090572	0.349075
$n = 512$	0.498623	2.81716	8.10032	28.302
$n = 1024$	3.90545	23.9769	67.2851	223.586
$n = 2048$	25.0781	176.909	501.448	1675.32

Table B.2: Timings in seconds for each input condition and the parameter settings in Table B.1.

	$b = 16$	$b = 512$	$b = 1024$	$b = 2048$
$n = 32$	8.38 (84)	0.0504 (84)	0.167 (84)	0.716 (84)
$n = 64$	0.133 (84)	3.22 (84)	0.648 (84)	0.408 (84)
$n = 128$	0.259 (84)	0.397 (84)	0.274 (84)	0.109 (84)
$n = 512$	0.0722 (168)	0.00557 (30)	0.00929 (28)	0.00492 (24)
$n = 1024$	0.0072 (26)	0.00174 (28)	0.0009 (30)	0.00136 (36)
$n = 2048$	0.00487 (14)	0.00124 (20)	0.00112 (18)	0.00129 (14)

Table B.3: For each input condition the format is "`maxrelerr (count)`*", where* ***`maxrelerr`*** *is the the maximum relative error of all re-run testcases compared to the original timings for that input condition, and where* ***`count`*** *is the number of test cases that this maximum was taken over.*

	$b = 16$	$b = 512$	$b = 1024$	$b = 2048$
$n = 32$	0.0	0.004141	0.0	0.0
$n = 64$	0.0	0.001834	0.0008452	0.02698
$n = 128$	0.001378	0.02458	0.007684	0.002501
$n = 512$	0.001937	0.0	0.0	0.0
$n = 1024$	0.0	0.0	0.0	0.0
$n = 2048$	0.006049	0.0	0.0	0.0

Table B.4: Relative error of disregarding the Strassen and Strassen strided algorithms compared to the optimal parameter sets in Table B.1. Coloured are input conditions for which the optimal time is less than 2 seconds.

	$b = 16$	$b = 512$	$b = 1024$	$b = 2048$
$n = 32$	100% 95%	67% 76%	67% 67%	62% 67%
$n = 64$	90% 76%	62% 57%	57% 57%	14% 10%
$n = 128$	86% 90%	24% 48%	24% 24%	67% 71%
$n = 512$	14% 19% 81% 67%	19% 38% 48% 48%	48% 48% 48% 48%	29% 29% 48% 48%
$n = 1024$	14% 19% 71% 71%	33% 33% 48% 48%	62% 52% 48% 48%	43% 43% 48% 48%
$n = 2048$	19% 19% 71% 71%	24% 24% 48% 48%	71% 67% 48% 48%	57% 52% 48% 48%

Table B.5: For each combination of matrix size, bitsize and algorithm choice, the percentage of block sizes with timings within 3% of optimal timings for that combination. Each cell contains values for the algorithms Strassen, Winograd, Strassen strided and Winograd strided in that order; the strided variants are omitted where invalid. Each combination has 21 possible block sizes, so the rounding does not introduce ambiguities. Coloured are input conditions for which the optimal time is less than 2 seconds.

	$b = 16$	$b = 512$	$b = 1024$	$b = 2048$
$n = 32$	0.006384	0.01767	0.01581	0.02642
$n = 64$	0.003667	0.004715	0.006254	0.06673
$n = 128$	0.005439	0.0403	0.06692	0.01113
$n = 512$	0.00722	0.01379	0.05651	0.07925
$n = 1024$	0.0	0.02437	0.0411	0.0811
$n = 2048$	0.1663	0.03866	0.02005	0.05078

Table B.6: Relative error of always choosing Winograd with block size 8×4 instead of the optimal parameter set. Coloured are input conditions for which the optimal time is less than 2 seconds.

	$b = 16$	$b = 512$	$b = 1024$	$b = 2048$
$n = 32$	—	—	—	—
$n = 64$	—	—	—	—
$n = 128$	—	—	—	—
$n = 512$	0.493	0.179	0.1456	0.1414
$n = 1024$	0.06859	0.03832	0.06611	0.1484
$n = 2048$	0.0	0.02654	0.02552	0.1169

Table B.7: Relative error of only considering interleaved algorithms instead of the optimal parameter set. Note that our interleaved implementation does not accept matrices smaller than 256×256.