K3 surfaces of any Artin-Mazur height over $\mathbb{F}_5$ and $\mathbb{F}_7$ via Quasi-F-split singularities and gpu acceleration

Ryan Batubara, Jack J. Garzella, Alex Pan

February 19, 2025

1 Introduction

Given a variety X over a field of characteristic p, Artin and Mazur in [7] define the height $h(X) \in \mathbb{N} \cup \{\infty\}$ (sometimes written h if X is obvious from the context) to be the height of a certain formal group Φ associated to X. The height is in some sense a measure of the "arithmetic complexity" of a variety, with higher heights indicating "more complexity". For example, consider the case when X is a K3 surface (i.e. a surface for which $\omega_X \cong \mathscr{O}_X$ and $H^1(X, \mathscr{O}_X) = 0$). In this case, h determines the *Newton polygon* of X, which in turn gives partial information about the point counts of X over finite fields. In this case, the Newton polygon equals the Hodge polygon (i.e. the variety is ordinary) if and only if $h = 1$, and the Newton polygon is supersingular if and only if $h = \infty$.

Given a field k of characteristic p, one may ask

Question 1.1. *For which values h does there exist a variety X such that $h(X) = h$?*

In the case that X is a K3 surface, it is known from the original work of Artin and Mazur that either $1 \leq h(X) \leq 10$ or $h = \infty$. One expects that all such h are taken on by K3 surfaces in characteristic p. For example, in [6], Artin shows[1] that this holds over an algebraically closed field k of characteristic p in the case that $p = 3 \mod 4$. In [28], Taelman shows that all possible h are realized over some sufficiently big finite field $\mathbb{F}_q$. By base change, this implies the result for extensions thereof (e.g. algebraically closed fields). However, Taelman's argument is not constructive, and there is no known bound on how big q must be to guarantee existence of surfaces of any height.

More concretely, it follows from the work of Kedlaya and Sutherland in [19] that there exist quartic K3 surfaces (i.e. quartic surfaces in $\mathbb{P}^3$) of all possible heights over $\mathbb{F}_2$. More recently, Kawakami, Takamatsu, and Yoshikawa in [17] have given examples of quartic K3 surfaces of all possible heights over $\mathbb{F}_3$.

Kawakami, Takamatsu, and Yoshikawa consider, instead of the Artin-Mazur height, a different quantity called the *quasi-F-split height* (see Section 2 for a definition), which is known to be the same as the Artin-Mazur height for Calabi-Yau varieties, and thus K3 surfaces. The theory of the quasi-F-split height has its roots in the theory of F-singularites and commutative algebra/birational geometry in characteristic p. In particular, one theorem fundamental to the development of the theory of F-singularities is Fedder's criterion (Theorem 2.12), which gives a very concrete and computable way to check whether or not a variety has height 1. The main theorem of Kawakami, Takamatsu, and Yoshikawa ([17, Theorem A]) is a generalization of Fedder's criterion to higher heights which gives a computable way to check whether a variety has height h. Moreover, they provide a simpler version ([17, Theorem C]) in the case when X is a Calabi-Yau hypersurface (and thus also for quartic K3 surfaces). The forthcoming work [13] provides an implementation of Theorem A. However, this algorithm (and the ones used to compute the examples in [17]) However, the algorithm used to compute the examples in [17] is not especially practical as it requires multiplying many large polynomials ([29], [13]), limiting the computation to very low characteristic.

In this work, we push the method of Kawakami, Takamatsu, and Yoshikawa to the limits of readily available hardware, and produce examples of quartic K3 surfaces X with all possible quasi-F-split heights h over $\mathbb{F}_5$ and $\mathbb{F}_7$. Our main insight is that one of the main bottlenecks of the problem can be broken down into many repeated matrix-vector multiplications for a square matrix of size $\binom{4p-1}{3}$. Because of this, we can use Nvidia's CUBLAS library [25] to perform the matrix multiplications on the GPU, making such computations nearly instantaneous. To create this matrix, one must calculate the matrix of a certain linear operator. We provide a few novel algorithms which accomplish this task, including some that can be implemented on the GPU. The remaining algorithm is bottlenecked by the operation of raising a polynomial

[1] Artin showed this conditionally, dependent on flat duality for surfaces, which was later proved by Milne in [23].

to a large power, so we implement this on the GPU as well using a Fast Fourier Transform (FFT) approach. Our algorithms and implementations lead to a high throughput of heights of K3 surfaces: about 1400 surfaces per second over $\mathbb{F}_5$ and about 180 surfaces per second over $\mathbb{F}_7$. Our implementations can also handle $\mathbb{F}_{11}$ and $\mathbb{F}_{13}$, though they are slower than known methods of calculating the Newton polygon (e.g. [11]).

All of our algorithms are implemented in Julia [9], and make use of the OSCAR computer algebra system [12]. Code from this work is open source and available online in various Julia packages: MMPSingularities.jl [4], GPUFiniteFieldMatrices.jl [2], GPUPolynomials.jl [3], and CudaNTTs.jl [1].

In Section 2, we give background on the quasi-F-split height and Fedder's criterion. In Section 3, we describe the naive implementation of [17, Theorem C] and describe our modification. In Section 4, we describe our algorithms for calculating the matrix of the key linear operator which we term "multiplying then splitting". In Section 5, we describe our GPU implementation of the Number Theoretic Transform, which is a finite field variant of the FFT. In Section 6, we describe the considerations we need to keep in mind to use CUBLAS. In Section 7, we describe the computation of surfaces of all possible heights over $\mathbb{F}_5$ and $\mathbb{F}_7$.

Throughout the paper, we do various time tests to compare various approaches for each computational step. All timing tests were performed with an Intel i5-8400 CPU and a Nvidia GeForce RTX 3070 GPU.

1.1 Acknowledgements

The authors thank the maintainers and support staff of the UCSD research cluster for use of their devices, and Wilson Cheung for much helpful tech support. Furthermore, they thank the authors of [17] and [13] for providing preliminary implementations of Theorem C and Theorem A of [17]. The second author thanks Jakub Witaszek, Kiran S. Kedlaya, Steve Huang, and Karl Schwede for helpful discussions. The second author was partially supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 2038238, and a fellowship from the Sloan Foundation.

2 Preliminaries: the Quasi-F-Split Height and Fedder's Criterion

2.1 The Witt Vectors

We begin by defining the Witt vectors. Since the theory of Witt vectors is a vast and active topic of research, we recall the bare minimum required for the quasi-F-split Fedder's criterion algorithm. For a more complete introduction to the Witt vectors with proofs, see [26]. For an intuitive introduction or derivation of the Witt vectors, see [20]. There are many other perspectives on Witt vectors. For example: [15, Chapter 17] covers the Witt vectors and its relationship with formal groups; [18] gives a categorical perspective on Witt vectors that relates to lifts of the Frobenius; and [27, Chapter 1] defines a generalization known as the *ramified Witt vectors* in detail.

Definition 2.1. The *n-th Witt Polynomial* ω_n is defined as

$$\omega_n(X_0,\dots,X_n) = X_0^{p^n} + pX_1^{p^{n-1}} + \dots + p^n X_n$$

Now let R be a ring of characteristic p. We define the map Φ to be the map

$$\prod_{n\in\mathbb{N}} R \xrightarrow{(\omega_n)_n} \prod_{n\in\mathbb{N}} R$$

defined as ω_n for the n-th component.

Lemma 2.2. *There exist integer polynomials $S_n(X_0,\dots,X_n,Y_0,\dots,Y_n)$ with the property that*

$$\Phi((S_n)_{n\in\mathbb{N}}) = \Phi((X_n)_{n\in\mathbb{N}}) + \Phi((Y_n)_{n\in\mathbb{N}}).$$

Likewise, there exist integer polynomials $P_n(X_0,\dots,X_n,Y_0,\dots,Y_n)$ such that

$$\Phi((P_n)_{n\in\mathbb{N}}) = \Phi((X_n)_{n\in\mathbb{N}}) \cdot \Phi((Y_n)_{n\in\mathbb{N}}).$$

Proof. For example, see [26, Theorem 2.6] and the surrounding discussion. □

We now define the ring of Witt vectors $W(R)$ to be $\prod_{n\in\mathbb{N}} R$ as a set, with the ring structure defined by

$$(a_n)_{n\in\mathbb{N}} + (b_n)_{n\in\mathbb{N}} = (S_n(a_0,\ldots,a_n,b_0,\ldots,b_n))_{n\in\mathbb{N}}$$

and likewise, multiplication is defined using the P_n. The lemma then shows that Φ is a homomorphism

$$W(R) \to \prod_{n\in\mathbb{N}} R.$$

Moreover, the fact that the polynomials do not depend on the base ring R means that the construction is functorial; that is, for a map of rings $R \to R'$, we get a map $W(R) \to W(R')$.

The main example, which also provides the fundamental motivation for Witt vectors, is the case when $R = \mathbb{F}_p$. It is well known that $W(\mathbb{F}_p) = \mathbb{Z}_p$, giving an alternative construction of the p-adic numbers.

Warning 2.3. *If one takes a naive p-adic expansion $\sum_{n=0}^{\infty} c_n p^n \in \mathbb{Z}_p$ with $c_n \in \{0,\ldots,p-1\}$, this does not correspond to the Witt vector $(c_0,c_1,c_2,\ldots)$. In fact, the aforementioned sum corresponds to $(c_0,c_1^p,c_2^{p^2},\ldots)$. See [20, Section 2]. This motivates the following:*

Definition 2.4. There exists a homomorphism $F\colon W(R) \to W(R)$, called the *Frobenius*, defined by

$$(c_0,c_1,\ldots) \mapsto (c_0^p,c_1^p,\ldots)$$

(this is induced by the Frobenius on R by functoriality).

Definition 2.5. There exists a homomorphism $V\colon W(R) \to W(R)$, called the *Verschiebung*, defined by

$$(c_0,c_1,c_2,\ldots) \mapsto (0,c_0,c_1,\ldots).$$

Lemma 2.6. *The composition $F \circ V = V \circ F$ is the multiplication by p map on $W(R)$.*

Proof. [20, Proposition 5] □

Thus, we see that $W(R)/pW(R) \cong R$. This is sometimes called the first *truncated* Witt vectors. We also have higher truncated variants.

Definition 2.7. The n-truncated Witt vectors $W_n(R)$ are defined as $W(R)/p^nW(R)$.

By the above lemma, this translates to an actual truncation in the coordinates of the Witt vectors.

Remark 2.8. Our computations will end up primarily involving $W_2(R)$[2], where addition is governed by the polynomials $S_0(X_0,Y_0) = X_0 + Y_0$ and

$$S_1(X_0,X_1,Y_0,Y_1) = X_1 + Y_1 + \frac{(X_0+Y_0)^p - X_0^p - Y_0^p}{p}.$$

Thus we see that addition in $W_2(R)$ involves raising things in R (i.e. the first component) to the p-th power in a lift of R to characteristic 0.

2.2 Splittings of Frobenius

Let R be a ring of characteristic p. We have the Frobenius morphism $F\colon R \to R$, defined as $F(x) = x^p$. We describe a few alternative perspectives on the Frobenius which will be useful later.

Remark 2.9.

(1) Let R be reduced. Then we may view the Frobenius as the inclusion $R \subset R^{1/p}$, where $R^{1/p}$ is the ring of formal p-th roots of elements of R.

(2) Similarly, again assuming that R is reduced we may view the Frobenius as the inclusion $R^p \subset R$,

(3) More generally, we define $F_\star R$ to be the R-algebra with ring structure the same as R, with module structure $r \cdot x = F(r)x = r^p x$. Then we view the Frobenius as a map $R \to F_\star R$. In this description, F is a module homomorphism as well. The module $F_\star R$ corresponds to the pushforward construction from geometry (i.e. pushforward of quasi-coherent sheaves on $\operatorname{Spec} R$). Even more generally, for any R-module M we denote $F_\star M$ to be the analogously defined pushforward by Frobenius.

[2] This comes from the delta formula, [17, Theorem D]

Definition 2.10. We say that R is *F-split* if the map F is split as a map of R-modules $R \to F_\star R$.

We will be chiefly concerned with *hypersurfaces*, so we specialize to this case now. For what follows, we assume that k is a field of characteristic $p > 0$ which is *F-finite*; that is, the Frobenius map is module-finite.

Definition 2.11. Let $S = k[x_1, \dots, x_n]$. Then we say that $f \in S$ is F-split if $S/(f)$ is.

A fundamental fact about F-splitness is that there exists a very concrete criterion for whether or not a polynomial (hypersurface) f is F-split. First, we introduce some notation. If $I = (x_1, \dots, x_n)$ is a finitely generated ideal of some ring R, then $I^{[m]}$ is defined to be $(x_1^m, \dots, x_n^m)$

Theorem 2.12. *[Fedder's Criterion] Let $f \in S = k[x_1, \dots, x_n]$. Let $\mathfrak{m} = (x_1, \dots, x_n)$ be the ideal generated by the variables. Then f is F-split if and only if $f^{p-1} \notin \mathfrak{m}^{[p]}$*

Proof. See [21, Theorem 2.5] □

Of particular interest is the case when f is homogeneous of degree n, which geometrically corresponds to a *Calabi-Yau* hypersurface. In this case, we have

Theorem 2.13. *If $f \in S$ homogeneous of degree n is F-split, then the Artin-Mazur height of $Z(f) = \operatorname{Proj}(S/(f))$ is 1, i.e. $Z(f)$ is weakly ordinary.*

Recently, in [31] Yobuko introduced the notion of quasi-F-splitness, which generalizes F-splitness.

Definition 2.14. The ring R is *n-quasi-F-split* if there exists a map $\phi \colon W_n(R) \to R$ such that

$$\begin{tikzcd} W(R) \arrow[r, "F"] \arrow[d] & F_\star W(R) \arrow[dl, "\phi"] \\ R & \end{tikzcd},$$

where the vertical map is the 1-st Witt vector truncation.

We further define the *quasi-F-split height* of R as the smallest n for which R is n-quasi-F-split. As above, the quasi-F-split height of $f \in S = k[x_1, \dots, x_n]$ is that of $R = S/(f)$.

Remark 2.15. Both F-splitness and quasi-F-splitness have various geometric variants which are more general then the ring-theoretic/affine versions given here. These are covered extensively in the literature, for example see [16].

The quasi-F-split height also gives a generalization of Theorem 2.13:

Theorem 2.16. *If $f \in S$ is homogeneous of degree n, then the Artin-Mazur height of $Z(f)$ is equal to the quasi-F-split height of f.*

Proof. This is a special case of [31, Theorem 4.5]. □

3 Fedder's criterion for quasi-F-splitness: an algorithmic perspective

We now describe how the Witt Vectors can be used to calculate the quasi-F-split height / Artin-Mazur height of a Calabi-Yau hypersurface, using the version of Fedder's criterion in [17]. We will describe the algorithm in more detail, and proofs can be found in [17].

3.1 The computation of Δ_1

For the following discussion, let k be a perfect field of characteristic p and let $S := k[x_1, \dots, x_n]$. Let $f = \sum_I a_I \mathbf{x}^I$ be a polynomial in S.

Definition 3.1. Let $\Delta_1(f)$ be defined by the following equation in $W_2(S)$:

$$(0, \Delta_1(f)) = (f, 0) - \sum_I (a_I \mathbf{x}^I, 0).$$

The following proposition demonstrates how we can calculate $\Delta_1(g)$ algorithmically.

Proposition 3.2. *Let $\tilde{f}$ be a lift of f to the integers, i.e. $f = \sum_I [a_I]\mathbf{x}^I$. If $k = \mathbb{F}_p$, we can compute $\Delta_1(f)$ by taking the reduction of*

$$\frac{\tilde{f}^p - \sum_I ([a_I]x^I)^p}{p}$$

mod p.

Proof. Iteratively apply the formula of the first Witt polynomial S_1 to the monomials of $\tilde{f}$. □

Remark 3.3. If f is a homogeneous polynomial of degree d, then $\Delta_1(f)$ is a polynomial of degree pd.

Proposition 3.2 gives an obvious algorithm for calculating the term $\Delta_1(f)$:

Algorithm 1 Calculation of $\Delta_1(f)$

```
Input: $f \in \mathbb{F}_p[x_1, \dots, x_n]$
Output: $\Delta_1(f)$
$\tilde{f} \leftarrow \mathrm{lift}(f)$
$D \leftarrow \tilde{f}^p$
for $t \in \mathrm{terms}(\tilde{f})$ do
    $D \leftarrow D - t^p$
end for
$D \leftarrow D/p$
return $D\%p$
```

3.2 Splittings of Frobenius from a computational perspective

Let $S = k[x_1, \dots, x_n]$ as before. We will use perspective (3) from Remark 2.9; recall that we identify S with the target of Frobenius and S^p with the source. We see by counting degrees that we have a generating set for S as an S^p-module given by all monomials $x_1^{i_1} \dots x_n^{i_n}$ where $0 \leq i_j \leq p-1$ for all j. Moreover, since S is a polynomial ring, there are no (module-theoretic) relations and S is the free S^p-module generated by these monomials, i.e.

$$S = \bigoplus_{1 \leq j \leq n,\ 0 \leq i_j \leq p-1} x_1^{i_1} \dots x_n^{i_n} S^p.$$

Then the projection of S to any of the direct sum components is an element of $\mathrm{Hom}(S, S^p)$, which is a splitting of Frobenius. Let u be the projection onto the component of $x_1^{p-1}, \dots, x_n^{p-1}$.

The splitting u plays an important role in F-singularity theory, see for example [21, Claim 2.6]. For our purposes, we are only concerned with computing u for a polynomial in S. Given $f \in S$, we will first compute $u(f) \in S^p$, and then use the identification $S^p \cong S$ by taking p-th roots of exponents.

Algorithm 2 Splitting of Frobenius

```
Input: $f \in S$
Output: $u(f)$ considered as an element of $S$.
Discard all terms of $f$ whose exponents are not congruent to $(p-1, \dots, p-1)$ mod p.
result $\leftarrow 0$
for $t \in \mathrm{terms}(f)$ do
    subtract $p-1$ from all exponents of $t$
    Divide all exponents of $t$ by $p$.          ▷ This division is exact because of the previous step
    result $\leftarrow$ result $+ t$
end for
return result
```

3.3 The naive algorithm

Let f be a homogeneous polynomial of degree n in S, so that $Z(f)$ is a Calabi-Yau hypersurface. Following [17], we have the following algorithm to calculate the quasi-F-split height.

Algorithm 3 Quasi-F-Split Height: naive algorithm

```
1:  Input: $b \in \mathbb{N}$ chosen bound, $f$ a homogeneous polynomial of degree $n$ in $S$.
2:  Output: $h(f)$
3:  $g \leftarrow f^{p-1}$
4:  if $g \notin \mathfrak{m}^p$ then
5:      return 1
6:  end if
7:  $\Delta \leftarrow \Delta_1(f^{p-1})$                    ▷ Use Algorithm 1
8:  $h \leftarrow 2$
9:  while true do
10:     if $b < h$ then
11:         return $\infty$
12:     end if
13:     $g \leftarrow u(\Delta * g)$                     ▷ Use Algorithm 2
14:     if $g \notin \mathfrak{m}^p$ then
15:         return $h$
16:     end if
17:     $h \leftarrow h + 1$
18: end while
```

Theorem 3.4 ([17], Theorem C)**.** *Assume that $Z(f)$ has quasi-F-split height $h < b$. Then Algorithm 3 terminates and returns h.*

Proof. This is just rephrasing [17, Theorem C]. □

In the case of Calabi-Yau hypersurfaces, we have bounds on the height by [30], so we can deterministically recover the height. See also [6, Theorem 0.1], for the case of K3 surfaces. For a K3 surface, the height (if finite) is bounded by 10.

3.4 The key idea: finding the matrix of the linear operator "multiply then split"

An implementation of Algorithm 3 is provided in MMPSingularities.jl. The bottleneck ends up being polynomial multiplication, in two places:

(1) raising $g = f^{p-1}$ to the p-th power in the integers (in line 4 of Algorithm 1)

(2) multiplying g by $\Delta_1(f^{p-1})$ (in line 13 of Algorithm 3)

For a quartic K3 surface of characteristic 5, for example, each step takes about 1 second using FLINT.

We now explain how to overcome the second bottleneck. Since $Z(f)$ is Calabi-Yau (i.e. $\deg f = n$), we have that f^{p-1} has degree $n(p-1)$. Furthermore, by Proposition 3.2 the degree of $\Delta_1(f^{p-1})$ is $np(p-1)$. Thus, $\Delta_1(f^{p-1})g$ has degree $n(p^2-1)$; however, the effect of u on any polynomial is subtracting $p-1$ from the exponents of the terms and dividing by p (see Algorithm 2). Thus, the "multiply then split" map $g \mapsto u(\Delta_1(f^{p-1})g)$ is a linear map from the space of homogeneous polynomials of degree $n(p-1)$ to itself. As a consequence of this observation, if we can efficiently compute the matrix of $g \mapsto u(\Delta_1(f^{p-1})g)$, we can repeatedly apply matrix-vector multiplication.

Furthermore, when g is written as a vector in the basis of homogeneous monomials of degree $n(p-1)$, we can test if $g \notin \mathfrak{m}^{[p]}$ in an especially simple way: the only monomial that is not in $\mathfrak{m}^{[p]}$ is $x_1^{p-1} \cdots x_n^{p-1}$, see for example [17]. Thus, we can check if a single element of the vector representing g is nonzero.

The algorithm for Fedder's criterion then becomes:

Algorithm 4 Quasi-F-Split Height: matrix-based algorithm

```
Input: $b \in \mathbb{N}$ chosen bound, $f$ a homogeneous polynomial of degree $n$ in $S$.
Output: $h(f)$
$g \leftarrow f^{p-1}$
if $g \notin \mathfrak{m}^p$ then
    return 1
end if
$\Delta \leftarrow \Delta_1(f^{p-1})$                                        ▷ Use Algorithm 1
$M \leftarrow$ the matrix of $g' \mapsto u(\Delta * g')$
$h \leftarrow 2$
$g_v \leftarrow$ the representation of $g$ as a vector
$i \leftarrow$ the index of the monomial $x_1^{p-1} \cdots x_n^{p-1}$
while true do
    if $b < h$ then
        return $\infty$
    end if
    $g_v \leftarrow M * g_v$
    if $g[i] \neq 0$ then
        return $h$
    end if
    $h \leftarrow h + 1$
end while
```

Thus, we have reduced our problem (algorithmically, at least) to finding the matrix of a "mulitply then split" operation.

4 Algorithms for finding the matrix of "multiply then split"

To find the matrix of "multiply then split" in the Algorithm for the quasi-F-split height of a Calabi-Yau hypersurface, we must first multiply all possible monomials by $\Delta_1(f)$ and then apply the map u. Here, we consider a slightly more general problem. As usual, let $S = k[x_1, \dots, x_n]$. Let S_ℓ denote the vector space of homogeneous degree ℓ polynomials in S. We let M_ℓ denote a lexographically-ordered list of the monomial basis of S_ℓ. Fix some $D \in \mathbb{N}$, and let $\Delta \in S_D$. In algorithms, we will sometimes conflate Δ with a list containing all its terms. Furthermore, fix $d \in \mathbb{N}$. We consider the problem of finding the matrix of $g \mapsto u(\Delta * g)$ on the space S_d. The target of this map is $S_{d'}$, where $d' := \frac{d+D-n-1}{p}$. Note that if d' is not an integer, there are no terms which survive u and the map is zero.

Remark 4.1. If $\Delta = \Delta_1(f^{p-1})$ and $d = n(p-1)$, as in the case of calculating the quasi-F-split height of a Calabi-Yau hypersurface, then $d' = d$ and the matrix is square.

Our first observation is that naively multiplying and then applying u as in Algorithm 2 does plenty of unnecessary work. In particular, any term in the product Δg with exponents not congruent to $(p-1, \dots, p-1) \mod p$ is not needed. Even if g is a monomial, giving a linear algorithm for multiplication, using naive multiplication stores a large amount of unnecessary terms in memory. Both of the our improved algorithms ignore all of these unnecessary terms.

In what follows, when we apply arithmetic operations to arrays, particularly $\star$, $+$, and $\%$, we mean componentwise application of these operations. Use of arbitrary componentwise operations is called *broadcasting* in Julia, and we will sometimes use this term.

Definition 4.2. The *exponent tuple* of a monomial $m = ax_1^{d_1}, \dots, x_n^{d_n}$ is $(d_1, \dots, d_n)$. We will denote it $exps(m)$, and we also define $coeff(m) := a$

Definition 4.3. The weak integer compositions of k into n parts, denoted $wics(k,n)$, is the set of ordered tuples $(d_1, \dots, d_n)$ such that $d_1 + \dots + d_n = k$.

Example 4.4. $wics(2,3) = \{(2,0,0),(0,2,0),(0,0,2),(1,1,0),(1,0,1),(0,1,1)\}$.

Thus, $wics(d,n)$ generates the exponent tuples of the monomial basis of the vector space of d-homogeneous n-variate polynomials over a field F. It follows that the dimension of the vector space is $|wics(k,n)|$.

Lemma 4.5. $|wics(k,n)| = \binom{k+n-1}{n-1}$

Proof. This is a classical argument which goes by the name “sticks and stones”, “stars and bars”, or “dots and dividers”. □

Remark 4.6. The number of terms of Δ is bounded above by $\binom{D+n-1}{n-1}$ by Lemma 4.5. Likewise, the number of elements of M_d is $\binom{d+n-1}{n-1}$. For all of our time complexity calculations, we let $L_d = length(M_d) = \binom{n+d-1}{n-1}$, and $L_\Delta = length(\Delta) \leq \binom{n+D-1}{n-1}$. In the case of the quasi-F-split height of a K3 surface, $L_d = \binom{n(p-1)+n-1}{n-1} = \binom{np-1}{n-1}$, and $L_\Delta = \binom{np(p-1)+n-1}{n-1} = \binom{n(p^2-p+1)-1}{n-1}$.

Definition 4.7. Let m be a monomial in S. Then we say that m *matches* with another monomial m' if the monomial mm' is not killed by u.

Concretely, this means that $exps(m) + exps(m') \equiv (p-1, \dots, p-1) \mod p$.

4.1 The Trivial Algorithm

Algorithm 5 Matrix of Multiply then Split: trivial algorithm

```
Input: $\Delta, M_d, M_{d'}, p$
Output: $length(M_{d'}) \times length(M_d)$ matrix representing “multiply then split”
mat $\leftarrow zeros(length(M_{d'}), length(M_d))$
for $m \in M_d$ do
    $c \leftarrow indexof(m, M_d)$
    for $\delta \in \Delta$ do
        if $(exps(\delta) + exps(m))\%p = (p-1, \dots, p-1)$ then        ▷ if $\delta$ matches with $m$
            res $\leftarrow (exps(\delta) + exps(m) - (p-1, \dots, p-1))/p$        ▷ Apply $u$ to $\delta m$
            $r \leftarrow indexof($res$, M_{d'})$
            mat$[r, c] = coeff(\delta)$
        end if
    end for
end for
return mat
```

In this algorithm, which we call `TRIV`, we iterate through the monomials $m \in M_d$, each corresponding to a column in the resulting matrix. For each monomial, we search for terms that match $\delta \in \Delta$, apply u to their product δm, and get the lexographical index of the result to find which row to add to.

In this algorithm, the matrix can be generated in $O(L_d L_\Delta)$ operations. This can be seen from the nested loops, assuming that integer arithmetic operations are constant time.

In practice, the majority of the combinations of terms of Δ and M_d don’t match. This means in Algorithm 5, much of our runtime is wasted on checking for whether terms match. However, we emphasize that this algorithm, if implemented in parallel on the GPU, is indeed fast enough to not be a bottleneck in practice.

4.2 Modified Merge-based Algorithm

We now introduce an algorithm that utilizes properties of monomial ordering to reduce the number of comparisons performed in checking whether two terms match.

Lemma 4.8. *Let $A = (a_1, \dots, a_n), B = (b_1, \dots, b_n)$ be in $\{0, \dots, p-1\}^n \subset \mathbb{Z}^n$. Then $A + B = (p-1, \dots, p-1) \mod p$ if and only if $A + B = (p-1, \dots, p-1)$.*

Proof. Each coordinate has $a_i + b_i \leq 2(p-1) = 2p-2$. □

Corollary 4.9. *Let A be as above. Then there exists a unique match $m(A) \in \{0, \dots, p-1\}^n$*

Proof. $m(A) = (p-1, \dots, p-1) - A$. The claim follows from 4.8. □

Essentially, this corollary says that when we consider the exponent tuple mod p of an arbitrary monomial of M, it has a unique match mod p.

Corollary 4.10. *Let $\leq_{lex}$ denote the lexographical ordering. If $A \leq_{lex} B$, then $m(B) \leq_{lex} m(A)$*

Proof. Follows from the definition of lexographical ordering. □

Using these two corollaries, we have the following algorithm, which we call `MERGE`:

Algorithm 6 Matrix of Multiply then Split: merge-based algorithm

```
Inputs: $\Delta, M_d, M_{d'}, p$
Output: $length(M_{d'}) \times length(M_d)$ matrix representing "multiply then split"
mat $\leftarrow zeros(length(M_d), length(M_{d'}))$
$L \leftarrow [exps(\delta)\%p \mid \delta \in \Delta]$
$R \leftarrow [exps(m)\%p \mid m \in M_d]$
sort both $L$ and $R$
$\Delta', M' \leftarrow$permute $\Delta$ and $M$ by the sort permutations as $L$ and $R$.
$l \leftarrow 1, r \leftarrow length(R)$
while $1 \leq r$ and $l \leq length(L)$ do
    cmp $\leftarrow L[l] + R[r] - (p-1, \dots, p-1)$
    if cmp < 0 then
        $l \leftarrow l+1$
    else if 0 < cmp then
        $r \leftarrow r-1$
    else if 0 == cmp then
        matchr $\leftarrow R[r]$
        numLmatches $\leftarrow$ the number of adjacent entries in $L$ which are equal to $L[l]$
        while $R[r] ==$ matchr &&$1 \leq r$ do
            for $ll \in (l, l+1, \dots, l+$numLmatches$)$ do
                res $\leftarrow (exps(\Delta'[ll]) + exps(M'[r]) - (p-1, \dots, p-1))/p$
                coeff $\leftarrow coeff(\Delta'[ll])$
                $col \leftarrow indexof(M'[r], M_d)$
                $row \leftarrow indexof($res$, M_{d'})$
                mat$[row, col] = coeff$
            end for
            $r \leftarrow r-1$
        end while
        $l \leftarrow l+$numLmatches$-1$
    end if
end while
return mat
```

Note that the inner while loops account for the fact that after modding by p, one does not expect either array to have unique elements. The algorithm is justified by the previous corollaries.

In practice, instead of sorting the arrays in place, we use a method like julia's `sortperm` to get the sort permutation.

Because we perform two sorts, then a linear merge, we perform $L_d \log L_d + L_\Delta \log L_\Delta + L_d + L_\Delta$ operations, giving the algorithm complexity $O(\max(L_d \log L_d, L_\Delta \log L_\Delta))$. In practice (for Calabi-Yau hypersurfaces), the $L_\Delta \log L_\Delta$ usually dominates.

4.3 Weak Integer Compositions-based Algorithm

Lemma 4.11. *The set of exponent tuples of M_d that match with δ is given by*

$$\left\{ m(exps(\delta)\%p) + w * p \;\middle|\; w \in wics\left(\frac{d - sum(m(exps(\delta)\%p))}{p}, n\right) \right\}$$

where $sum(X)$ is the sum of the elements of tuple X, and m is the matching function from Corollary 4.9

Proof. Direct computation from the defintions of weak integer compositions and matching terms. □

Because this expression is quite convoluted, we provide an example of a computation that may come up in calculating the quasi-F-split height of a K3 quartic surface over $\mathbb{F}_5$.

Example 4.12. Let $p = 5$, $n = 4$, and δ a monomial with exponent tuple $(21, 19, 22, 18)$. Then M_{16} is the monomial basis of S_{16} (over $\mathbb{F}_5$). To find the monomials of M_{16} that match with δ, we begin by reducing $(21, 19, 22, 18)\%5 = (1, 4, 2, 3)$. This makes $m((1, 4, 2, 3)) = (3, 0, 2, 1)$ the exponent tuple of a matching monomial mod p. However, $(3,0,2,1)$ isn't in M_{16}. For it to be in M_{16}, we need to add $16-(3+0+2+1)=10$ to the degree, and to maintain congruence to $(4,4,4,4) \mod 5$, we need to add two 5's to the elements of $(3,0,2,1)$. This corresponds to $\{w*5 \mid w \in wics(2,4)\}$, which is:

$$\{(10,0,0,0),(5,5,0,0),(5,0,5,0),(5,0,0,5),(0,10,0,0),$$
$$(0,5,5,0),(0,5,0,5),(0,0,10,0),(0,0,5,5),(0,0,0,10)\}$$

Adding each of these to $(3,0,2,1)$, we get the exponent tuples of monomials that match with $(21,19,22,18)$

$$\{(13,0,2,1),(8,7,0,1),(8,0,7,1),(8,0,2,6),(3,10,2,1),$$
$$(3,5,7,1),(3,5,2,6),(3,0,12,1),(3,0,7,6),(3,0,2,11)\}$$

Because we need it later for our complexity computation, we have the corollary:

Corollary 4.13. *The number of matching monomials of term δ is bounded above by $\binom{\lfloor \frac{d}{p} \rfloor + n-1}{n-1}$.*

Proof. By Lemma 4.11, there are $\left| wics\left(\frac{d-sum(m(exps(\delta)\%p))}{p}, n\right) \right|$ monomials that match with δ. Because the division is exact, $\lfloor \frac{d}{p} \rfloor \geq \frac{d-sum(m(exps(\delta)\%p))}{p}$, giving the upper bound $|wics(\lfloor \frac{d}{p} \rfloor, n)| = \binom{\lfloor \frac{d}{p} \rfloor + n-1}{n-1}$. □

Denoting the expression from Lemma 4.11 by $generateMatchingMonomials(\delta, p, M_d)$, we have the following algorithm, which we call `WICS`:

Algorithm 7 Matrix of Multiply then Split: WICS algorithm

```
**Input**: $\Delta, M_d, M_{d'}, p$
**Output**: $length(M_{d'}) \times length(M_d)$ matrix representing "multiply then split"
mat $\leftarrow zeros(length(M_{d'}), length(M_d))$
**for** $\delta \in \Delta$ **do**
    mons $\leftarrow generateMatchingMonomials(\delta, p, M_d)$
    **for** $m \in mons$ **do**
        $c \leftarrow indexof(m, M_d)$
        res $\leftarrow (exps(\delta) + exps(m) - (p-1, \ldots, p-1))/p$
        $r \leftarrow indexof(\text{res}, M_{d'})$
        mat$[r,c] = coeff(\delta)$
    **end for**
**end for**
**return** mat
```

Letting $L_d = length(M_d)$ and $L_\Delta = length(\Delta)$, this algorithm generates the matrix in $O\left(L_\Delta \cdot \binom{\lfloor \frac{d}{p} \rfloor + n-1}{n}\right)$ operations. The number of matching monomials each term δ has is bounded above by $\binom{\lfloor \frac{d}{p} \rfloor + n-1}{n}$ by Corollary 4.13, which is effectively constant for all reasonably computable n and p

4.4 Implementation

In the implementation of `TRIV`, `MERGE`, and `WICS`, there are a few optimizations and considerations common to all the implementations. First, all exponent tuples are bitpacked into unsigned integers, improving memory efficiency and allowing for faster comparisons. Note that a comparison between two unsigned integers is the same as a lexicographical comparison between the tuples. Broadcasted addition and subtraction on bitpacked n-tuples reduces to addition and subtraction of integers, though broadcasted division and modulo still require n separate operations.

All algorithms require the $indexof()$ function, which we implement using Julia's `Dict{K,V}`, which is implemented as a hashtable.

`TRIV` and `WICS` are easily GPU-parallelizable by creating a thread for each term of Δ, and performing the insides of the loop in each thread. This creates the problem of needing an $indexof()$ function, since Julia's `Dict{K,V}` is not compatible with the GPU. The solution we implement is a static hashtable, with size and hashing function decided at compiletime. Due to Julia's JIT compilation nature, creating this hashmap is quite slow. Another solution can be to perform a binary search for the $indexof()$ function, since M_d is lexicographically sorted, but for one-off computations, users are better off using non-parallelized

MERGE or WICS to avoid first-time GPU kernel compilation times. However, if many matrices need to be created, caching the hashtable allows the GPU implementations to far outpace the non-parallelized CPU implementations.

MERGE should be parallelizable by implementing a modified parallel merge and using a parallel sort. However, we expect the sorting bottleneck to mean that such a parallel algorithm wouldn't beat GPU-parallel WICS, so we don't bother developing the modified parallel merge.

Below in Figure 4.4, we compare running times for all the p which we consider. WICS ends up being the fastest practically, both on the CPU and the GPU. We did not try running the tests for the trivial algorithm on the CPU for $p = 11, 13$ because we do not expect it to finish in a reasonable amount of time.

p	TRIV-CPU	TRIV-GPU	MERGE	WICS-CPU	WICS-GPU
3	0.01	0.0007	0.001	0.001	0.0002
5	1.6	0.005	0.046	0.028	0.0024
7	45.33	0.104	0.670	0.277	0.025
11	-	6.796	26.35	5.45	0.56
13	-	31.86	88.97	17.06	2.05

Figure 1: Comparison of timings for various algorithms that calculate the matrix of “multiply then split”. Times are in seconds and are an average of 10 different trials.

5 Polynomial powering

In this section, we describe how Algorithm 1 is computed. Specifically, we focus on Step 3, $D \leftarrow \tilde{f}^p$, which is the main bottleneck. We first explain our implementation of polynomial powering using the Number Theoretic Transform (NTT), then explore other known polynomial powering algorithms and their ability to be sped up through mass parallelization.

5.1 Multi-modular NTT

5.1.1 Kronecker Substitution

We reduce the problem of multivariate polynomial powering to univariate polynomial powering by the Kronecker substitution. Let M be a non-inclusive upper bound on the degree of any variable of the polynomial $f \in R[x_1, \ldots, x_n]$. Then the Kronecker Substitution $g(z) = f(z, z^M, z^{M^2}, \ldots, z^{M^{n-1}})$ produces a univariate polynomial $g \in R[z]$, see [5]. Algebraically, this is the same as applying the homomorphism $R[x_1, \ldots, x_n] \to R[z]$ which takes x_i to $z^{M^{i-1}}$. This map cannot be injective in general, but it is injective on the subset of elements of $f \in R[x_1, \ldots, x_n]$ whose terms have all degrees strictly less than M. Given two polynomials f_1, f_2 that we wish to multiply, we may choose M big enough so that the product lies in the subset, so we may recover the product in $R[x_1, \ldots, x_n]$ by performing the product in the ring $R[z]$.

Remark 5.1. In FLINT, exponent tuples are bitpacked into unsigned integers. The process of bitpacking an array is a Kronecker Substitution with M being a power of 2.

In the case where our polynomial is homogeneous, we can simply ignore one variable in all of our operations between monomials of the same degree. As above, algebraically this is applying the evaluation homomorphism $x_n \mapsto 1$. This map is of course not injective, but it is injective on homogeneous elements. This effectively decreases the number of variables of our polynomial by 1, which greatly lowers the degree of the univariate result of the Kronecker Substitution. This improvement is essential for dense algorithms like the NTT.

5.1.2 Polynomial Powering using NTTs

The k-th power of a univariate polynomial f may be computed mod p by taking the NTT of a tuple containing the coefficients of f, raising the components to the k, and then computing the Inverse Number Theoretic Transform. The multimodular algorithm computes f^k in the integers by choosing primes p_i, and combining the results using the Chinese remainder theorem. Thus, our algorithm is known as the multi-modular NTT approach to raising polynomials to powers.

In many cases, using the Kronecker Substitution to map a multivariate polynomial to a univariate polynomial results in a sparse univariate polynomial. As such, dense algorithms like the NTT become inefficient in both time and memory complexity when compared to sparse algorithms. However, the massive throughput of GPUs allows the NTT to be competitive for many cases, quasi-F-split heights of quartic K3 surfaces being one of them.

5.1.3 NTT Implementation

We use the Merge-NTT algorithm with Barrett modular reduction from [32] ported to Julia for our NTT implementation. So, our NTT length L is always a power of 2.

For each NTT we perform, we search for primes p that satisfy $p \equiv 1 \mod L$, compute primitive Lth roots of unity for each p, and $L^{-1} \mod p$. These are cached, so that they can be used again for problems with the same shape (i.e. the degree of f, the exponent k, and the characteristic are the same).

Our algorithm also has fallbacks when the memory required becomes too large for the GPU's on-device memory. When the problem becomes too big to fit the twiddle factors and inputs for all of the NTTs in device memory at once, we move the inputs to GPU memory and back to RAM for each NTT, and don't cache the twiddle factors. Because we are forced to move a lot of memory around for each NTT, and run $\log_2(L)$ modular exponentiations in each thread, we see a large drop-off in performance when this NTT size threshold is reached.

5.1.4 Prime Selection

In order for the NTT to simulate polynomial powering over $\mathbb{Z}$, we need to obtain an upper bound M on the resulting coefficients, then choose primes $p_1 \dots p_k$ such that $p_1 \cdot p_2 \cdot \dots \cdot p_k > M$.

Experimentally, modular multiplication with 32-bit integers is slightly over double the speed of modular multiplication with 64-bit integers. However, the kernels and parameters from [32] are more optimized for 64-bit integers, making them just over half the speed of naively using the same implementation with 32-bit integers. Thus, for prime selection, we choose primes satisfying $p \equiv 1 \mod L$ that fit within 62 bits, because of the precision of Barrett Reduction.

5.1.5 Bound Finding

The multimodular NTT requires an upper bound M in order to select primes to compute NTTs in. In this section, we present a quick way to compute a relatively tight upper bound on the resulting coefficients of raising a homogeneous polynomial to a power, which will allow for easier application of the NTT in polynomial powering problems.

Definition 5.2. Given a basis $(\mathbf{x}^{I_1}, \dots, \mathbf{x}^{I_n})$, where I_i are distinct degree sequences of equal length, the *maximal polynomial* is $(m-1)(\mathbf{x}^{I_1} + \dots + \mathbf{x}^{I_n})$, where m is a non-inclusive upper bound on the coefficients.

This is saying we should consider the "worst-case" polynomial for our bound-finding computations.

For small problems, we can simply plug the maximum polynomial g into FLINT, compute g^p, and iterate through the resulting terms to obtain an upper bound. We know this computation will be correct because FLINT uses GMP, and it also must be the optimal bound on the coefficients. However, for larger problems, like bound finding for Δ_1 (Algorithm 1) of quartic K3 surfaces over $\mathbb{F}_{11}$ or $\mathbb{F}_{13}$, FLINT does not finish the computation in a reasonable amount of time. If one only cares about a single shape of the problem (i.e. the same degree, k, and characteristic), this might be acceptable. However, for a single computation, it completely removes the advantage of using the GPU, since we must perform an expensive CPU mutliplication to set up the algorithm.

Alternatively, we can obtain a relatively tight upper bound for the case of raising a homogeneous multivariate polynomial over $\mathbb{F}_p[x_1, \dots, x_k]$ to a power by bounding the number of terms in the power.

Theorem 5.3. *Let f be a h-homogeneous polynomial of $\mathbb{Z}[x_1, \dots, x_n]$, where m is a non-inclusive upper bound on the coefficients. Then, the coefficients of f^k are bounded above by* $\left((m-1) \cdot \binom{h+n-1}{n-1}\right)^k$

Proof. We induct on k. The maximum coefficient of f^0 is 1, which is bounded above by $\left((m-1) \cdot \binom{h+n-1}{n-1}\right)^0 = 1$. Let $(d_1, \dots, d_n)$ denote the exponent tuple of a term of f^k, let $(d'_1, \dots, d'_n)$ denote the exponent tuple of a term of f^{k+1}, and let $(a_1, \dots, a_n)$ denote the exponent tuple of a term of f.

Consider an arbitrary term of f^{k+1}. To find all terms in the unreduced expansion of $f^k \cdot g$ that contribute to that term, we look at terms of f^k with degree sequences of the form $(d'_1 - a_1, \dots, d'_n - a_n)$. The number of these terms of f^k is bounded above by $|wics(h,n)|$, or $\binom{h+n-1}{n-1}$ by Lemma 4.5. Using our inductive assumption, the maximum coefficient of f^k is bounded above by $\left((m-1)\cdot\binom{h+n-1}{n-1}\right)^k$, so multiplying each of these by coefficients of f, which have a maximum value of $m-1$, and adding up $\binom{h+n-1}{n-1}$ copies of these gives $\left((m-1)\cdot\binom{h+n-1}{n-1}\right)^{k+1}$. □

To apply this formula to computing $\Delta_1(f^{p-1})$ for K3 quartic surfaces over $\mathbb{F}_p$, we plug in $h = 4p$, $m = p$, and $k = p$ to obtain $M = \left((p-1)\cdot\binom{4p+3}{3}\right)^k$ for an upper bound. To obtain the optimal upper bound, we can plug in the maximal polynomial into a multimodular NTT with primes multiplying to over M, and retrieve the maximum coefficient of that result.

5.2 Other algorithms

Many other algorithms for polynomial powering are described in [24]. They compare performance in sparse and dense cases. Here, we breifly comment on how these algorithms perform on the GPU. We refer the reader to [24] for a more precise description of all these algorithms. In the following, assume we have some polynomial f which we wish to raise to the k-th power.

RMUL is analogous to the classic FOIL algorithm which is taught in schools. This is bottlenecked (at least on the GPU) by the "collect like terms" step, which requires a parallel sort. While there are GPU-optimized sorting algorithms, such as the parallel merge sort provided by CUDA.jl, it is a relatively expensive operation on the GPU. There is an algorithm proposed in [14] which claims to beat the state-of-the-art, but we were unable to reproduce the result.

RSQR uses the binary expansion of k to find the power in less total multiplications than **RMUL**. **BINA** and **BINB** use binomial expansion to more efficiently expand f^k, using **RMUL** to merge at the end. They both perform better in the case that the problem is sparse. On the GPU, all of these are bottlenecked by sorting, just like **RMUL**.

MNE uses multinomial coefficients to expand f^k and then combines like terms with a sort. Unlike the previous four algorithms, it is not bottlenecked by the sort; instead, it is bottlenecked by the memory required to store the table of multinomial coefficients. Do note that **MNE** is competitive for small problems.

SUMS and **FPS** have dense and sparse versions which are described in [24]. In particular, **FPS** is implemented in FLINT, and is called by our code when we need powering on the CPU. They can be parallelized, as discussed in [24], but this requires the use of locks and a heap data structure, which are more challenging to implement on the GPU. It would be interesting to have a GPU-accelerated version of **FPS** and compare its performance with multimodular FFT. The authors expect one could get a big improvement in the sparse case.

5.3 Evaluation

To demonstrate the power of using the GPU for mathematical computations, we compare our implementation with two existing computational mathematics libraries, FLINT and MAGMA. We take a homogeneous polynomial of degree 16 in the integers, whose coefficients are randomly chosen in the set $\{0,\dots,4\}$, and raise it to successive powers n, starting at $n = 5$. Raising such a polynomial to the 5th power is a similar computation to the bottleneck in the calculation of the quasi-F-split height of a quartic surface over $\mathbb{F}_5$.

Note that the above numbers are really the GPU beating the CPU. The NTT is a dense algorithm, while FLINT uses a sparse algorithm. We do not know which algorithm MAGMA uses for polynomial powering in the integers. In theory, such a sparse algorithm should be much more efficient. However, the throughput of the GPU is so much better that the GPU crushes the CPU, even with a worse algorithm. The authors hope that these numbers can inspire others who rely on fundamental algorithms such as those in FLINT to consider re-implementing them on the GPU.

Additionally note that this does not mean the GPU will give improvements for every polynomial powering problem. Generally, the NTT is powerful when the resulting coefficients are small, and the polynomial is reasonably dense. In more sparse problems, as discussed before in section 5.2, the NTT is less effective, and other algorithms benefit less from parallelization, making them more suited for the CPU instead.

n	MAGMA	FLINT	GPUPolynomials.jl
5	2.10	0.80	0.001
6	5.82	1.64	0.003
7	12.03	2.93	0.005
8	21.64	5.99	0.010
9	38.78	12.5	0.011
10	63.22	20.7	0.012
11	95.48	29.8	0.038
12	137.61	40.3	0.043
13	194.73	52.2	0.074
14	267.57	70.0	0.079
15	359.22	90.1	0.087

Figure 2: Polynomial powering times (in seconds) for various powers

6 Matrix multiplication mod p

In the last few years, a lot of work has gone into optimizing matrix multiplication, especially with floating-point data types (for example, [25], [10]). Today, floating point types are faster than integer data types. For example, on Nvidia devices, Float32 multiplication is twice as fast as Int32 multiplication, and Float64 multiplication is supported in hardware while Int64 multiplication is not. Moreover, the IEEE standards guarantee that integer multiplication (i.e. those that only use the mantissa) is guaranteed to be exact even in floating point types. Thus, we can freely treat floating points as an integer type of smaller size. This trick has been utilized many times in the literature (e.g. see [8]). Here, we implement a simple version of using this trick in Julia, which suffices for our purposes.

Let ℓ be the largest possible value of an integer for our data type. Each entry ranges from 0 to $N-1$. Thus, the maximum number of operations o before our datatype overflows is one less than the value o' such that $o' \cdot (N^2 - 2N + 1) = o' \cdot (N-1)^2$ is larger than the integer limit ℓ. Thus

$$o = \left\lfloor \frac{\ell}{(N^2 - 2N + 1)} \right\rfloor - 1$$

For Nvidia GPUs, we wish to use the Float32 type, for the primes $p \in \{3, 5, 7, 11, 13\}$. Respectively, for each of these values we have $o = 4194302, 599185, 246722, 135299, 85597, 59073$. Furthermore, the sizes of the matrices in question are respectively $165, 969, 2925, 12341, 20825$. Thus, we see that for $p \leq 13$, we can use a fully floating-point library like CUBLAS naively. [3] To support larger primes, we implement a simple GPU-based fallback implementation of matrix multiplication using CUDA.jl based on [22] which reduces mod p every 32 entries. Our implementation is provided in GPUFiniteFieldMatrices.jl. While it doesn't come close to CUBLAS, and is in fact slower than a OpenBLAS when using CPU multithreading, it gives exact computations for arbitrarily large matrices (as long as $32 < o$) and is good enough that matrix multiplication won't be a bottleneck in Fedder type criterion calculations.

To illustrate the performance difference between CPU and GPU, we timed multiplying matrices of various sizes on the CPU and GPU in characteristic 11:

Size	CPU single-threaded	CPU multi-threaded	CUBLAS	Fallback implementation
5,000	2.47	1.18	0.02	1.78
10,000	18.72	8.03	0.13	15.00
15,000	62.90	25.30	0.43	54.95
20,000	150.76	58.83	3.66	114.72

Figure 3: Matrix multiplication times (in seconds) for various sizes of matrices

[3] Note that we use OpenBLAS and CUBLAS; we are really using Julia wrappers provided by libraries such as Oscar, CUDA.jl, or the Julia standard library. Note that MAGMA also provides a mod p matrix multiplication implementation, which (according to its documentation) wraps CUBLAS for for p=11 and p=13.

7 Heights of K3 Surfaces

7.1 Recollections on the moduli of K3 surfaces

We have the following:

Theorem 7.1 (Lang-Weil, Theorem 1)**.** *Let $X \subset \mathbb{P}^n$ be a projective variety of dimension r. Then*

$$\#X(\mathbb{F}_p) = p^r + O(p^{r-\frac{1}{2}})$$

Say we have an ambient projective variety Y with chosen hypersurface D. The Lang-Weil estimate roughly says that if we pick a random point x, we can expect x to lie in D with probably about $1/p$. By [6, Section 7], the locus M_i of height i such that $h \leq i$ is cut out by a single section in M_{i-1}. If we apply both of these facts, with Y being the moduli space[4] of quartic K3 surfaces, we can conclude the probability of a random point in the moduli space being in the locus M_2 is about $1/p$. Inductively, we see

Heuristic 7.2. *The probability of a randomly chosen surface having height h is about $1/p^h$.*

Thus, we may find a surface of height h by randomly choosing quartic K3 surfaces, and if we compute a few times p^h samples, we can be confident that we'll find one with high probability.

In practice, our methodology is to choose random quartic polynomials, by sampling a point in the vector space of homogeneous polynomials, which is isomorphic to $\mathbb{F}_p^{35}$. To obtain the moduli space M of quartic K3 surfaces, we must projectivize and take the quotient by the action of $\mathrm{PGL}_4(\mathbb{F}_p)$ which acts by changes of variables. Thus, there may be two sources of deviation from the expected probability–the group quotient and the actual error term in the estimate.

7.2 Computations

Over $\mathbb{F}_5$, we found that the probability of finding a K3 surface of height h was about $1/5^h$, to three digits of precision, for all heights $h \leq 6$. For higher heights we had less samples and more variance, although the probabilities seem to be less than expected for higher heights. Similarly, over $\mathbb{F}_7$ we found that the probabilities very closely matched $1/7^h$ for low heights, with a more variance at higher heights.

For $p = 5$, the algorithm throughput is about 1400 surfaces per second on Nvidia 2080Ti GPUs provided by the UCSD research cluster. Since much less time is taken for height 1 examples (since the classical Fedder's criterion suffices), for the purposes of estimating the time to compute a high height example we may ignore them. Thus, the expected compute time necessary to find a height = 10 example is about $5^9/1400 \approx 1395$ seconds, or about 23 compute minutes. For $p = 7$, the throughput is about 185 surfaces per second. Thus, the expected compute time to compute a height = 10 example is $7^9/185 \approx 218{,}127$ seconds, or about 60 compute hours. The actual times to find the examples were much longer than this, because the authors were using a less-optimized NTT and had not yet discovered `WICS` and were using `MERGE` instead.

For $p = 11$ and $p = 13$, the threshold for the NTT where GPU memory becomes a bottleneck is reached, and the algorithm faces a sharp drop in speed. The examples took about 12 compute hours on the aforementioned 2080Tis.

[4] say, the course moduli space of polarized K3 surfaces, though one should be able to make a similar statement for the moduli stack

Height	Equation
1	$x_1^4+x_2^4+x_3^4+x_4^4$
2	$4x_1^4+2x_1^3x_2+x_1^3x_4+4x_1^2x_2^2+2x_1^2x_2x_3+2x_1^2x_3^2+x_1^2x_3x_4+3x_1x_2^3+4x_1x_2^2x_3+4x_1x_2^2x_4+2x_1x_2x_3x_4+3x_1x_2x_4^2+3x_1x_3^3+x_1x_3^2x_4+x_1x_3x_4^2+x_1x_4^3+4x_2^4+2x_2^3x_3+4x_2^3x_4+4x_2^2x_3^2+x_2^2x_3x_4+2x_2^2x_4^2+3x_2x_3^3+4x_2x_3^2x_4+4x_2x_3x_4^2+2x_2x_4^3+2x_3^4+2x_3^3x_4+2x_3^2x_4^2+x_3x_4^3+4x_4^4$
3	$2x_1^4+x_1^3x_2+3x_1^3x_3+x_1^3x_4+x_1^2x_2x_3+4x_1^2x_2x_4+x_1^2x_3^2+4x_1^2x_3x_4+3x_1^2x_4^2+4x_1x_2^3+3x_1x_2^2x_3+x_1x_2^2x_4+2x_1x_2x_3^2+3x_1x_2x_3x_4+x_1x_3^3+4x_1x_3x_4^2+2x_1x_4^3+x_2^3x_3+3x_2^3x_4+4x_2^2x_3^2+4x_2^2x_3x_4+x_2^2x_4^2+2x_2x_3^3+3x_2x_3^2x_4+4x_2x_3x_4^2+3x_2x_4^3+4x_3^4+3x_3^3x_4+2x_3x_4^3+3x_4^4$
4	$4x_1^4+2x_1^3x_3+4x_1^3x_4+3x_1^2x_2^2+3x_1^2x_2x_3+4x_1^2x_2x_4+2x_1^2x_3x_4+x_1^2x_4^2+3x_1x_2^3+x_1x_2^2x_3+x_1x_2^2x_4+x_1x_2x_3^2+x_1x_2x_3x_4+x_1x_2x_4^2+2x_1x_3^3+2x_1x_3^2x_4+x_1x_3x_4^2+2x_1x_4^3+4x_2^4+3x_2^3x_3+x_2^3x_4+3x_2^2x_3^2+3x_2^2x_3x_4+x_2^2x_4^2+2x_2x_3^3+3x_2x_3^2x_4+x_2x_3x_4^2+3x_2x_4^3+3x_3^4+2x_3^3x_4+4x_3^2x_4^2+x_3x_4^3$
5	$2x_1^4+2x_1^3x_2+4x_1^3x_3+3x_1^3x_4+2x_1^2x_2^2+4x_1^2x_2x_3+x_1^2x_2x_4+2x_1^2x_3^2+3x_1^2x_3x_4+4x_1x_2^2x_3+3x_1x_2^2x_4+x_1x_2x_3^2+x_1x_2x_3x_4+2x_1x_2x_4^2+3x_1x_3^3+3x_1x_3^2x_4+x_1x_3x_4^2+x_2^4+x_2^3x_3+2x_2^2x_3^2+2x_2^2x_3x_4+3x_2^2x_4^2+2x_2x_3^3+2x_2x_3^2x_4+2x_2x_3x_4^2+4x_3^4+x_3^3x_4+2x_3^2x_4^2+3x_4^4$
6	$x_1^3x_2+x_1^3x_3+3x_1^3x_4+3x_1^2x_2^2+2x_1^2x_2x_4+4x_1^2x_3x_4+4x_1^2x_4^2+x_1x_2^3+3x_1x_2^2x_3+4x_1x_2^2x_4+2x_1x_2x_3^2+2x_1x_2x_3x_4+2x_1x_2x_4^2+2x_1x_3^3+3x_1x_3^2x_4+3x_1x_3x_4^2+x_1x_4^3+4x_2^4+x_2^3x_3+x_2^3x_4+x_2^2x_3^2+4x_2^2x_3x_4+x_2^2x_4^2+3x_2x_3^3+2x_2x_3^2x_4+3x_2x_4^3+4x_3^4+x_3^3x_4+3x_3x_4^3+x_4^4$
7	$4x_1^4+x_1^3x_3+3x_1^3x_4+4x_1^2x_2^2+2x_1^2x_2x_3+2x_1^2x_2x_4+2x_1^2x_3^2+4x_1^2x_3x_4+4x_1x_2^2x_3+2x_1x_2x_3^2+x_1x_2x_4^2+2x_1x_3^3+4x_1x_3^2x_4+2x_1x_3x_4^2+x_1x_4^3+4x_2^4+3x_2^3x_4+3x_2^2x_3^2+x_2^2x_3x_4+2x_2^2x_4^2+3x_2x_3^2x_4+4x_2x_3x_4^2+3x_2x_4^3+3x_3^3x_4+x_3^2x_4^2+4x_4^4$
8	$x_1^4+2x_1^3x_2+4x_1^3x_3+x_1^2x_2^2+4x_1^2x_2x_3+x_1^2x_2x_4+x_1^2x_3x_4+2x_1x_2^2x_3+2x_1x_2^2x_4+2x_1x_2x_3^2+4x_1x_2x_3x_4+3x_1x_2x_4^2+3x_1x_3^3+4x_1x_3^2x_4+3x_1x_3x_4^2+x_1x_4^3+4x_2^4+4x_2^3x_3+x_2^3x_4+4x_2^2x_3^2+2x_2^2x_3x_4+x_2^2x_4^2+4x_2x_3^2x_4+x_2x_4^3+x_3^4+2x_3^3x_4+x_3^2x_4^2+4x_4^4$
9	$3x_1^4+3x_1^3x_2+3x_1^3x_3+x_1^2x_2^2+3x_1^2x_2x_3+3x_1^2x_2x_4+3x_1^2x_3^2+2x_1^2x_3x_4+2x_1^2x_4^2+4x_1x_2^3+2x_1x_2^2x_3+4x_1x_2x_3^2+2x_1x_2x_3x_4+4x_1x_2x_4^2+x_1x_3^3+3x_1x_3^2x_4+3x_1x_3x_4^2+x_1x_4^3+3x_2^3x_3+4x_2^3x_4+3x_2^2x_3x_4+x_2^2x_4^2+4x_2x_3^2x_4+4x_2x_3x_4^2+4x_2x_4^3+3x_3x_4^3+4x_4^4$
10	$2x_1^4+4x_1^3x_2+3x_1^3x_3+x_1^3x_4+x_1^2x_2^2+2x_1^2x_2x_3+2x_1^2x_2x_4+4x_1^2x_3^2+4x_1^2x_3x_4+2x_1^2x_4^2+x_1x_2^3+4x_1x_2^2x_4+3x_1x_2x_3^2+3x_1x_2x_4^2+2x_1x_3^3+3x_1x_3^2x_4+2x_1x_3x_4^2+x_1x_4^3+3x_2^4+2x_2^3x_3+2x_2^3x_4+4x_2^2x_3^2+3x_2^2x_3x_4+3x_2^2x_4^2+x_2x_3^3+2x_2x_3x_4^2+2x_2x_4^3+4x_3^4+x_3^3x_4+3x_3^2x_4^2+4x_3x_4^3+3x_4^4$
∞	$x_1^4+x_2^4+x_3^4+x_4^4+xyzw$

Figure 4: Quartic K3 surfaces with specified Artin-Mazur height over $\mathbb{F}_5$

Height	Equation
1	$5x_1^4 + 5x_1^3x_3 + 2x_1^3x_4 + 3x_1^2x_2x_3 + x_1^2x_2x_4 + 6x_1^2x_3^2 + 3x_1^2x_3x_4 + 3x_1^2x_4^2 + 4x_1x_2^3 + 6x_1x_2^2x_3 + 2x_1x_2^2x_4 + 4x_1x_2x_3^2 + 5x_1x_2x_3x_4 + 4x_1x_2x_4^2 + 5x_1x_3^3 + 4x_1x_3^2x_4 + 5x_1x_4^3 + 5x_2^4 + x_2^3x_3 + 4x_2^3x_4 + 5x_2^2x_3^2 + x_2^2x_3x_4 + x_2x_3^3 + 2x_2x_3^2x_4 + 2x_2x_3x_4^2 + x_2x_4^3 + 3x_3^4 + 5x_3^3x_4 + 3x_3^2x_4^2 + x_4^4$
2	$3x_1^4 + 4x_1^3x_2 + x_1^3x_3 + x_1^3x_4 + x_1^2x_2^2 + 5x_1^2x_2x_3 + 5x_1^2x_2x_4 + 3x_1^2x_3^2 + 5x_1^2x_3x_4 + 6x_1^2x_4^2 + 2x_1x_2^3 + x_1x_2^2x_3 + 5x_1x_2^2x_4 + 2x_1x_2x_3x_4 + x_1x_2x_4^2 + 2x_1x_3^3 + 3x_1x_3^2x_4 + x_1x_3x_4^2 + x_1x_4^3 + 4x_2^4 + 4x_2^3x_3 + 4x_2^3x_4 + 6x_2^2x_3^2 + 3x_2^2x_3x_4 + 3x_2x_3^3 + 4x_2x_3x_4^2 + 2x_3^4 + 4x_3^3x_4 + 4x_3^2x_4^2 + 2x_3x_4^3 + 6x_4^4$
3	$4x_1^4 + x_1^3x_2 + 2x_1^3x_3 + 6x_1^3x_4 + 6x_1^2x_2^2 + 3x_1^2x_2x_3 + 3x_1^2x_2x_4 + 2x_1^2x_3x_4 + 4x_1^2x_4^2 + 2x_1x_2^3 + 5x_1x_2^2x_4 + 5x_1x_2x_3^2 + 4x_1x_2x_3x_4 + 4x_1x_2x_4^2 + 6x_1x_3^3 + x_1x_3^2x_4 + 5x_1x_3x_4^2 + 2x_1x_4^3 + 3x_2^4 + 2x_2^3x_3 + 5x_2^2x_3^2 + 5x_2^2x_3x_4 + 3x_2^2x_4^2 + 4x_2x_3^3 + 6x_2x_3^2x_4 + 5x_2x_3x_4^2 + 3x_2x_4^3 + 4x_3^3x_4 + 4x_3^2x_4^2 + x_3x_4^3 + 5x_4^4$
4	$2x_1^4 + 6x_1^3x_2 + 3x_1^3x_3 + x_1^3x_4 + 4x_1^2x_2^2 + 3x_1^2x_2x_3 + 3x_1^2x_2x_4 + 2x_1^2x_3^2 + x_1^2x_3x_4 + 2x_1^2x_4^2 + 3x_1x_2^3 + 6x_1x_2^2x_4 + x_1x_2x_3^2 + 6x_1x_2x_3x_4 + x_1x_2x_4^2 + 4x_1x_3^3 + 2x_1x_3^2x_4 + 5x_1x_3x_4^2 + 2x_1x_4^3 + 6x_2^4 + 3x_2^3x_3 + 5x_2^2x_3^2 + x_2^2x_3x_4 + 5x_2^2x_4^2 + 4x_2x_3^3 + 3x_2x_3^2x_4 + x_2x_4^3 + 6x_3^4 + 2x_3^3x_4 + x_3^2x_4^2 + 3x_3x_4^3 + 2x_4^4$
5	$5x_1^4 + 6x_1^3x_2 + 2x_1^3x_3 + 3x_1^3x_4 + 4x_1^2x_2^2 + 3x_1^2x_2x_4 + 2x_1^2x_3^2 + 3x_1^2x_3x_4 + 6x_1^2x_4^2 + 4x_1x_2^2x_3 + 6x_1x_2^2x_4 + 2x_1x_2x_3^2 + 3x_1x_2x_3x_4 + 5x_1x_2x_4^2 + 3x_1x_3^3 + x_1x_3^2x_4 + 5x_1x_3x_4^2 + 6x_2^4 + 5x_2^3x_3 + 3x_2^2x_3^2 + 6x_2^2x_3x_4 + 3x_2x_3^3 + 3x_2x_3^2x_4 + 4x_2x_3x_4^2 + 3x_2x_4^3 + 5x_3^4 + 6x_3^2x_4^2 + 6x_3x_4^3 + 3x_4^4$
6	$x_1^4 + x_1^3x_2 + 4x_1^3x_3 + 6x_1^3x_4 + 6x_1^2x_2^2 + 2x_1^2x_2x_4 + 6x_1^2x_3x_4 + 6x_1^2x_4^2 + 4x_1x_2^3 + 3x_1x_2^2x_3 + 2x_1x_2^2x_4 + 2x_1x_2x_3^2 + 5x_1x_2x_3x_4 + 6x_1x_2x_4^2 + 6x_1x_3^2x_4 + 3x_1x_3x_4^2 + 6x_2^4 + 2x_2^3x_3 + 3x_2^3x_4 + 5x_2^2x_3^2 + 4x_2^2x_3x_4 + 6x_2^2x_4^2 + 5x_2x_3^2x_4 + x_2x_3x_4^2 + 3x_2x_4^3 + 2x_3^4 + 2x_3^3x_4 + 5x_3^2x_4^2 + 2x_3x_4^3 + 4x_4^4$
7	$2x_1^3x_2 + 2x_1^3x_3 + 2x_1^3x_4 + x_1^2x_2^2 + 2x_1^2x_2x_3 + 3x_1^2x_2x_4 + 5x_1^2x_3^2 + 6x_1^2x_3x_4 + x_1^2x_4^2 + 2x_1x_2^3 + 5x_1x_2^2x_3 + x_1x_2x_3^2 + 2x_1x_2x_3x_4 + 6x_1x_2x_4^2 + 4x_1x_3^3 + 6x_1x_3^2x_4 + 5x_1x_3x_4^2 + 2x_1x_4^3 + 2x_2^3x_3 + 3x_2^3x_4 + 4x_2^2x_3^2 + 3x_2^2x_4^2 + 3x_2x_3^3 + x_2x_3^2x_4 + 5x_2x_3x_4^2 + 5x_2x_4^3 + 5x_3^3x_4 + x_3^2x_4^2 + 6x_3x_4^3 + 6x_4^4$
8	$2x_1^3x_2 + 2x_1^3x_4 + 4x_1^2x_2^2 + 6x_1^2x_2x_3 + 5x_1^2x_2x_4 + 4x_1^2x_3^2 + 3x_1^2x_3x_4 + 3x_1^2x_4^2 + 4x_1x_2^3 + x_1x_2^2x_3 + x_1x_2^2x_4 + 4x_1x_2x_3^2 + 5x_1x_2x_3x_4 + x_1x_2x_4^2 + 3x_1x_3^3 + x_1x_3^2x_4 + 3x_1x_3x_4^2 + x_1x_4^3 + 5x_2^3x_3 + 5x_2^3x_4 + 6x_2^2x_3x_4 + 6x_2^2x_4^2 + 4x_2x_3^2x_4 + 3x_2x_3x_4^2 + 2x_2x_4^3 + 6x_3^3x_4 + 6x_3^2x_4^2 + 4x_3x_4^3$
9	$2x_1^3x_2 + x_1^3x_3 + 6x_1^3x_4 + 6x_1^2x_2^2 + 4x_1^2x_2x_3 + 2x_1^2x_2x_4 + 3x_1^2x_3x_4 + x_1^2x_4^2 + x_1x_2^3 + x_1x_2^2x_3 + 6x_1x_2^2x_4 + 6x_1x_2x_3^2 + 6x_1x_2x_3x_4 + 6x_1x_2x_4^2 + 2x_1x_3^3 + 4x_1x_3x_4^2 + 6x_1x_4^3 + 6x_2^3x_3 + 4x_2^3x_4 + 3x_2^2x_3^2 + 4x_2x_3^3 + 5x_2x_3^2x_4 + 4x_2x_3x_4^2 + 5x_2x_4^3 + 3x_3^3x_4 + 4x_3^2x_4^2 + 2x_3x_4^3 + 3x_4^4$
10	$3x_1^4 + 2x_1^3x_2 + x_1^3x_3 + x_1^3x_4 + 4x_1^2x_2x_3 + 2x_1^2x_2x_4 + 5x_1^2x_3x_4 + 6x_1^2x_4^2 + x_1x_2^3 + 2x_1x_2^2x_4 + 5x_1x_2x_3^2 + 3x_1x_2x_3x_4 + 4x_1x_2x_4^2 + 5x_1x_3^3 + x_1x_3^2x_4 + x_1x_3x_4^2 + x_1x_4^3 + 6x_2^4 + x_2^3x_4 + 6x_2^2x_3^2 + x_2^2x_3x_4 + 4x_2^2x_4^2 + x_2x_3^3 + 5x_2x_4^3 + 2x_3^4 + 5x_3^3x_4 + 5x_3^2x_4^2 + x_3x_4^3 + 6x_4^4$
∞	$3x_1^4 + 3x_1^3x_2 + 3x_1^3x_3 + 6x_1^2x_2^2 + 3x_1^2x_2x_4 + 2x_1^2x_3^2 + 2x_1^2x_3x_4 + 3x_1^2x_4^2 + 6x_1x_2^3 + 5x_1x_2^2x_3 + x_1x_2x_3x_4 + 5x_1x_2x_4^2 + 5x_1x_3^3 + 4x_1x_3^2x_4 + 3x_1x_3x_4^2 + 6x_1x_4^3 + x_2^4 + 4x_2^3x_4 + 3x_2^2x_3^2 + 5x_2^2x_3x_4 + 5x_2x_3^3 + x_2x_3^2x_4 + 6x_2x_3x_4^2 + x_3^3x_4 + x_3^2x_4^2 + 3x_3x_4^3 + 4x_4^4$

Figure 5: Quartic K3 surfaces with specified Artin-Mazur height over $\mathbb{F}_7$

Height	Equation
1	$4x_1^4+6x_1^3x_2+x_1^3x_3+2x_1^3x_4+3x_1^2x_2^2+x_1^2x_2x_3+3x_1^2x_2x_4+6x_1^2x_3^2+6x_1^2x_3x_4+8x_1^2x_4^2+7x_1x_2^3+2x_1x_2^2x_3+8x_1x_2^2x_4+8x_1x_2x_3x_4+10x_1x_2x_4^2+10x_1x_3^3+9x_1x_3^2x_4+6x_1x_3x_4^2+3x_1x_4^3+6x_2^4+7x_2^3x_3+4x_2^3x_4+10x_2^2x_3^2+3x_2^2x_3x_4+5x_2^2x_4^2+4x_2x_3^2x_4+6x_2x_4^3+3x_3^4+4x_3^3x_4+7x_3^2x_4^2+9x_3x_4^3+5x_4^4$
2	$4x_1^4+5x_1^3x_2+9x_1^3x_3+2x_1^3x_4+8x_1^2x_2^2+x_1^2x_2x_3+9x_1^2x_2x_4+x_1^2x_3^2+8x_1^2x_3x_4+6x_1x_2^3+10x_1x_2^2x_3+2x_1x_2^2x_4+10x_1x_2x_3^2+9x_1x_2x_3x_4+6x_1x_2x_4^2+8x_1x_3^3+4x_1x_3^2x_4+7x_1x_3x_4^2+9x_1x_4^3+3x_2^4+7x_2^3x_3+6x_2^3x_4+10x_2^2x_3^2+8x_2^2x_3x_4+x_2^2x_4^2+9x_2x_3^3+6x_2x_3^2x_4+x_2x_3x_4^2+9x_3^4+10x_3^3x_4+x_3^2x_4^2+x_3x_4^3+4x_4^4$
3	$10x_1^4+9x_1^3x_2+5x_1^3x_3+4x_1^3x_4+3x_1^2x_2^2+9x_1^2x_2x_3+4x_1^2x_2x_4+10x_1^2x_3^2+4x_1^2x_3x_4+8x_1^2x_4^2+8x_1x_2^3+9x_1x_2^2x_3+3x_1x_2^2x_4+7x_1x_2x_3^2+3x_1x_2x_4^2+8x_1x_3^3+2x_1x_3^2x_4+x_1x_3x_4^2+7x_1x_4^3+2x_2^4+3x_2^3x_4+x_2^2x_3^2+x_2^2x_3x_4+x_2^2x_4^2+5x_2x_3^3+9x_2x_3^2x_4+9x_2x_3x_4^2+4x_2x_4^3+5x_3^4+10x_3^3x_4+10x_3x_4^3+10x_4^4$
4	$2x_1^4+4x_1^3x_2+9x_1^3x_3+10x_1^3x_4+2x_1^2x_2^2+4x_1^2x_2x_3+4x_1^2x_2x_4+4x_1^2x_3^2+10x_1^2x_3x_4+9x_1^2x_4^2+5x_1x_2^3+5x_1x_2^2x_3+x_1x_2^2x_4+8x_1x_2x_3^2+2x_1x_2x_3x_4+10x_1x_2x_4^2+8x_1x_3^3+7x_1x_3^2x_4+5x_1x_3x_4^2+4x_1x_4^3+3x_2^4+6x_2^3x_3+4x_2^3x_4+10x_2^2x_3^2+5x_2^2x_3x_4+5x_2^2x_4^2+x_2x_3^3+5x_2x_4^3+5x_3^4+7x_3^2x_4^2+5x_3x_4^3+9x_4^4$
5	$10x_1^4+x_1^3x_2+6x_1^3x_3+3x_1^3x_4+x_1^2x_2^2+9x_1^2x_2x_3+6x_1^2x_2x_4+6x_1^2x_3^2+8x_1^2x_3x_4+4x_1^2x_4^2+3x_1x_2^3+7x_1x_2^2x_3+3x_1x_2^2x_4+7x_1x_2x_3^2+9x_1x_2x_3x_4+8x_1x_2x_4^2+7x_1x_3^3+x_1x_3x_4^2+7x_1x_4^3+x_2^4+3x_2^3x_3+7x_2^3x_4+5x_2^2x_3^2+7x_2^2x_3x_4+8x_2^2x_4^2+8x_2x_3^3+5x_2x_3^2x_4+x_2x_3x_4^2+9x_2x_4^3+7x_3^4+4x_3^3x_4+4x_3^2x_4^2+3x_3x_4^3$

Figure 6: Quartic K3 surfaces with specified Artin-Mazur height over $\mathbb{F}_{11}$

Height	Equation
1	$6x_1^4+7x_1^3x_3+4x_1^3x_4+6x_1^2x_2^2+7x_1^2x_2x_3+9x_1^2x_2x_4+2x_1^2x_3^2+3x_1^2x_3x_4+12x_1^2x_4^2+8x_1x_2^3+4x_1x_2^2x_3+x_1x_2^2x_4+9x_1x_2x_3^2+8x_1x_2x_3x_4+10x_1x_2x_4^2+8x_1x_3^3+2x_1x_3^2x_4+9x_1x_3x_4^2+4x_1x_4^3+5x_2^4+4x_2^3x_3+2x_2^2x_3^2+x_2^2x_3x_4+2x_2^2x_4^2+10x_2x_3^3+2x_2x_3^2x_4+2x_2x_3x_4^2+5x_2x_4^3+4x_3^4+3x_3^2x_4^2+2x_4^4$
2	$12x_1^4+8x_1^3x_2+8x_1^3x_3+10x_1^3x_4+8x_1^2x_2^2+11x_1^2x_2x_4+8x_1^2x_3^2+12x_1^2x_3x_4+x_1^2x_4^2+7x_1x_2^3+9x_1x_2^2x_3+11x_1x_2^2x_4+10x_1x_2x_3^2+7x_1x_2x_4^2+8x_1x_3^3+3x_1x_3^2x_4+11x_1x_3x_4^2+x_1x_4^3+4x_2^4+7x_2^3x_3+4x_2^3x_4+8x_2^2x_3^2+12x_2^2x_3x_4+6x_2^2x_4^2+7x_2x_3^3+12x_2x_3^2x_4+4x_2x_3x_4^2+10x_2x_4^3+4x_3^4+8x_3^3x_4+5x_3^2x_4^2+4x_3x_4^3$
3	$8x_1^4+2x_1^3x_2+3x_1^3x_3+x_1^3x_4+6x_1^2x_2^2+7x_1^2x_2x_3+5x_1^2x_2x_4+2x_1^2x_3^2+x_1^2x_4^2+11x_1x_2^3+10x_1x_2^2x_3+3x_1x_2^2x_4+5x_1x_2x_3^2+10x_1x_2x_3x_4+7x_1x_2x_4^2+12x_1x_3^3+12x_1x_3^2x_4+5x_1x_3x_4^2+7x_1x_4^3+7x_2^4+6x_2^3x_3+3x_2^3x_4+10x_2^2x_3^2+5x_2^2x_3x_4+12x_2^2x_4^2+x_2x_3^3+3x_2x_3^2x_4+12x_2x_3x_4^2+8x_2x_4^3+10x_3^4+7x_3^3x_4+4x_3^2x_4^2+8x_3x_4^3+2x_4^4$
4	$4x_1^4+4x_1^3x_2+2x_1^3x_3+3x_1^3x_4+9x_1^2x_2^2+6x_1^2x_2x_3+7x_1^2x_2x_4+10x_1^2x_3^2+x_1^2x_3x_4+4x_1x_2^3+4x_1x_2^2x_3+6x_1x_2^2x_4+12x_1x_2x_3^2+7x_1x_2x_3x_4+3x_1x_2x_4^2+11x_1x_3^3+9x_1x_3^2x_4+10x_1x_3x_4^2+11x_1x_4^3+3x_2^4+5x_2^3x_3+8x_2^3x_4+5x_2^2x_3x_4+5x_2^2x_4^2+5x_2x_3^3+10x_2x_3^2x_4+2x_2x_3x_4^2+10x_2x_4^3+4x_3^4+5x_3^2x_4^2+4x_3x_4^3+6x_4^4$
5	$11x_1^4+4x_1^3x_2+12x_1^3x_3+4x_1^3x_4+6x_1^2x_2^2+10x_1^2x_2x_3+4x_1^2x_2x_4+x_1^2x_3^2+7x_1^2x_3x_4+4x_1^2x_4^2+6x_1x_2^3+11x_1x_2^2x_3+7x_1x_2^2x_4+8x_1x_2x_3^2+10x_1x_2x_4^2+x_1x_3^3+9x_1x_3^2x_4+8x_1x_3x_4^2+11x_1x_4^3+4x_2^4+8x_2^3x_3+5x_2^2x_3x_4+7x_2^2x_4^2+8x_2x_3^3+6x_2x_3^2x_4+5x_2x_4^3+2x_3^4+10x_3^3x_4+8x_3^2x_4^2+10x_3x_4^3+6x_4^4$

Figure 7: Quartic K3 surfaces with specified Artin-Mazur height over $\mathbb{F}_{13}$

References

[1] Cudantts.jl. https://github.com/alexp616/CudaNTTs.jl.

[2] Gpufinitefieldmatrices.jl. https://github.com/UCSD-computational-number-theory/GPUFiniteFieldMatrices

[3] Gpupolynomials.jl. https://github.com/alexp616/GPUPolynomials.jl.

[4] Mmpsingularities.jl. https://github.com/jjgarzella/MMPSingularities.jl.

[5] Andrew Arnold and Daniel S. Roche. Multivariate sparse interpolation using randomized kronecker substitutions. *CoRR*, abs/1401.6694, 2014.

[6] M. Artin. Supersingular $k3$ surfaces. *Annales scientifiques de l'École Normale Supérieure*, Ser. 4, 7(4):543–567, 1974.

[7] M. Artin and B. Mazur. Formal groups arising from algebraic varieties. *Annales scientifiques de l'École Normale Supérieure*, Ser. 4, 10(1):87–131, 1977.

[8] Jérémy Berthomieu, Stef Graillat, Dimitri Lesnoff, and Theo Mary. Modular matrix multiplication on GPU for polynomial system solving. *ACM Commun. Comput. Algebra*, 57(2):35–38, 2023.

[9] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[10] OpenBlas community. Openblas, 2024. http://www.openmathlib.org/OpenBLAS/docs/.

[11] Edgar Costa, David Harvey, and Kiran S. Kedlaya. Zeta functions of nondegenerate hypersurfaces in toric varieties via controlled reduction in p-adic cohomology. *Proceedings of the Algorithmic Number Theory Symposium XIII*, 2(1):221–238, 2019.

[12] Wolfram Decker, Christian Eder, Claus Fieker, Max Horn, and Michael Joswig, editors. *The Computer Algebra System OSCAR: Algorithms and Examples*, volume 32 of *Algorithms and Computation in Mathematics*. Springer, 1 edition, 2025.

[13] Anne Fayolle, Abhay Goel, Devlin Mallory, Eamon Quinlan-Gallego, and Teppei Takamatsu. The witt vectors package for macaualay2. In preparation.

[14] Sumit Kumar Gupta, Dr. Dhirendra Pratap Singh, and Dr. Jaytrilok Choudhary. New gpu sorting algorithm using sorted matrix. *Procedia Computer Science*, 218:1682–1691, 2023. International Conference on Machine Learning and Data Engineering.

[15] M. Hazewinkel. *Formal Groups and Applications*. ISSN. Elsevier Science, 1978.

[16] Tatsuro Kawakami, Teppei Takamatsu, Hiromu Tanaka, Jakub Witaszek, Fuetaro Yobuko, and Shou Yoshikawa. Quasi-f-splittings in birational geometry, 2022.

[17] Tatsuro Kawakami, Teppei Takamatsu, and Shou Yoshikawa. Fedder type criteria for quasi-f-splitting, 2022.

[18] Kiran S. Kedlaya. Notes on prismatic cohomology, 2021. Accessed: Sept 2024.

[19] Kiran S. Kedlaya and Andrew V. Sutherland. A census of zeta functions of quartic k3 surfaces over $\mathbb{F}_2$. *LMS Journal of Computation and Mathematics*, 19(A):1–11, 2016.

[20] Dongryul Kim. Witt vectors, 2017. Accessed: Sept 2024.

[21] Linquan Ma and Thomas Polstra. F-singularities, a commutative algebra approach. Accessed: Sept 2024.

[22] Lei Mao. Cuda matrix multiplication optimization, 2024. Accessed: Nov 2024.

[23] J. S. Milne. Duality in the flat cohomology of a surface. *Annales scientifiques de l'École Normale Supérieure*, Ser. 4, 9(2):171–201, 1976.

[24] Michael Monagan and Roman Pearce. Sparse polynomial powering using heaps. In Vladimir P. Gerdt, Wolfram Koepf, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 236–247, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[25] Nvidia. cublas, 2024. Accessed: Nov 2024.

[26] Joseph Rabinoff. The theory of witt vectors, 2014.

[27] Peter Schneider. *Galois Representations and (Phi, Gamma)-Modules*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2017.

[28] Lenny Taelman. K3 surfaces over finite fields with given l-function. *Algebra & Number Theory*, 10(5):1133–1146, 2016.

[29] Teppei Takamatsu. private correspondence.

[30] G. Van Der Geer and T. Katsura. On the height of calabi-yau varieties in positive characteristic. *Documenta Mathematica*, 8(1):97–113, 2003.

[31] Fuetaro Yobuko. Quasi-frobenius splitting and lifting of calabi–yau varieties in characteristic p. *Mathematische Zeitschrift*, 292(1):307–316, 2019.

[32] Ali Şah Özcan and Erkay Savaş. Two algorithms for fast GPU implementation of NTT. Cryptology ePrint Archive, Paper 2023/1410, 2023.