Integer matrix multiplication at various precisions using AMX and IFMA*†

Joris van der Hoeven[A], Marc Mezzarobba[B], Laboratoire d'informatique de l'École polytechnique

Preliminary version of June 1, 2026

In this paper we study hardware-accelerated integer matrix multiplication, with coefficients of sizes between 8 and 1000 bits. More particularly, we study two relatively new hardware features in Intel CPUs: the IFMA "integer FMA" instruction and the AMX matrix extensions. We study various algorithms and analyze to what extent our implementations on top of the JIL library can approach theoretical peak performance.

1 Introduction

Motivation

Arguably the most significant recent trend in computer hardware is the eruption of tensor cores in both GPUs and CPUs. Although driven by applications in artificial intelligence, is there a potential for tensor cores to be used for other purposes? After all, GPUs have found many applications beyond their initial ones in the gaming industry.

The main motivation of this paper is to study this question for applications in scientific computation and more particularly in computer algebra and computer arithmetic. We will focus on the problem of $n \times n$ matrix multiplication with p-bit integer entries. Tensor cores are designed to accelerate such computations in the case where n is large and $p = 8$. Can this be used to efficiently deal with other bit precisions, ranging from small ones like $p = 16, 24, 32, 48, 64, \ldots$ to larger ones like $p = 128, 256, 512, 1024, \ldots$?

We mainly restrict our attention to the multiplication of square $n \times n$ matrices. This is indeed most favorable for tensor cores, but also most significant from a complexity perspective, since the complexity of many problems in linear algebra and beyond can be expressed in terms of the complexity of square matrix products [5]. Of course, this restriction will also simplify comparisons, because varying both n and p will already give rise to numerous different cases, as we shall see.

To be fair, we should also compare implementations that exploit tensor cores with alternative ones that are based on more conventional vector arithmetic. For this reason, we chose to focus on recent Intel Xeon processors which support both AMX matrix products and AVX512 vector arithmetic with IFMA. Using AMX, each core can do a $16 \times 64 \times 16$ matrix FMA (MFMA) $C \mathrel{+}= A \cdot B$ in about 16 cycles, where A and B (of dimensions 16×64 and 64×16) have 8-bit entries and C (of dimension 16×16) has 32-bit entries. Roughly speaking, an integer FMA (IFMA) allow us to compute $z \mathrel{+}= x \cdot y$ in one cycle, where x and y are 8-wide vectors of 52-bit integers and z is an 8-wide vector of 116-bit integers. Note that we preferred the terminology MFMA over GeMM and (general matrix multiplication $C := \alpha A B + \beta C$) because AMX only supports positive accumulation.

In this paper, we focus on the performance of implementations that use a single core of our CPU. This simplification allows to measure the relevance of different implementation strategies that exploit different hardware features (IFMA or AMX), without being distracted by mostly orthogonal considerations on how to scale up these base implementations on multi-core architectures.

All our timings will be relative to theoretical peak performance. For instance, for a naive implementation of $n \times n$ matrix multiplication using $2\,n^3$ IFMAs, we will report on the cost of the matrix multiplication divided by the theoretical cost to perform $2\,n^3$ IFMAs. These ratios are fairly independent of the particular machine on which we execute our programs and indicate the quality of our implementation. Quotients close to one mean that we could not have done much better for the algorithm at hand. Higher ratios indi-

A. Laboratoire d'informatique de l'École polytechnique, LIX, UMR 7161 CNRS, Bâtiment Alan Turing, CS35003, 1, rue Honoré d'Estienne d'Orves, 91120 Palaiseau, France, vdhoeven@lix.polytechnique.fr

B. Laboratoire d'informatique de l'École polytechnique, LIX, UMR 7161 CNRS, Bâtiment Alan Turing, CS35003, 1, rue Honoré d'Estienne d'Orves, 91120 Palaiseau, France, marc@mezzarobba.net

∗. This work has been supported by an ERC-2023-ADG grant for the ODELIX project (number 101142171).

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

†. This article has been written using GNU TeXmacs [37].

cate potential problems such as suboptimal memory access patterns or asymptotically subdominant costs that become high for certain sizes. In an ideal world in which all ratios are one, the theoretical complexity analysis of our methods would correspond perfectly to their practical performance. For completeness, we also added an appendix with absolute timings, both for our new implementation and some existing software.

All implementations in this paper are done inside or on top of the C++ library JIL [2, 40] for computations with straight-line programs (SLPs). This work is the first step of a project to the JIL library with fast linear algebra over the integers and fixed point numbers. We would like to cover all ranges of matrix dimensions and bit-sizes of the integers, while taking advantage of recent hardware features such as IFMA and AMX.

Conversely, matrix multiplication is an excellent problem for extending JIL beyond the mere compilation of SLPs. Another important motivation behind the present work is to use matrix multiplication as a case study for how to integrate SLPs inside larger pieces of code such as multiple loops. The integration of AMX instructions into the SLP framework is also a challenge.

Previous work

Multiple precision integer multiplication has a long history and there exist many algorithms for various bit precisions, such as the traditional schoolbook method, Karatsuba's algorithm [43], Toom's method [62], and modular methods [24, Chapter 5]. For high precisions, there are also methods based on the fast Fourier transform [11, 57, 34], but we will not use them in this paper. GMP [28] and MPFR [31] are the reference libraries for multiple precision integer and floating point arithmetic. We also refer to [38, 29, 18, 8, 39] for some recent work on this topic in the HPC context.

In the context of integer matrix multiplication, fast multiple precision methods tend to become useful at already very low precisions. This is due to the fact that all known methods can be regarded as *modular* or *evaluation-interpolation* methods. Roughly speaking, multiplying two $n \times n$ matrices with $(\ell\, p)$-bit entries is reduced to $\mathsf{E}(\ell)$ multiplications of $n \times n$ matrices with p-bit entries, where the quantity $\mathsf{E}(\ell)$ depends on the method (this will be detailed in Section 3 and Table 1 below). The cost of this reduction scales with n^2, whereas the cost of the $\mathsf{E}(\ell)$ low precision matrix multiplications scales with n^3. Consequently, the cost of the reduction becomes negligible for large n and, for a given ℓ, it is recommended to pick the method for which $\mathsf{E}(\ell)$ is minimal. Good practical implementations should reflect this fact and this is indeed what our work aims at.

Theoretically speaking [67], $n \times n$ matrices can be multiplied in time $O(n^\omega)$ with $\omega \leqslant 2.371552$ for large n. However, among the asymptotically faster algorithms with $\omega < 3$, only Strassen's algorithm [60] with $\omega = \log_2 7$ has allowed for some modest practical gains so far. The recent trend to accelerate matrix multiplication directly in the hardware leads to more significant practical gains. (In fact, these hardware accelerations make the practical complexity of matrix multiplication behave as if ω were well below 3. The potential impact of this observation is another motivation behind our work.)

Several works [12, 19] and research implementations [50, 49] specifically address the challenges raised by Intel's AMX technologies. OpenBlas [61] also contains a bf16 kernel (`sbgemm_kernel_16x16_spr_tmp1.c`) and GGML [26] an int8 kernel interleaved by encoding and decoding of quantized tensors (`ggml-cpu/amx/mmq.cpp`). Other types of CPUs integrate tensor accelerations in different ways. For instance, Apple's secret matrix coprocessor in M1–M3 CPUs is based on fast "outer products" [10], for which separate techniques have been developed [20]. We refer to [52] for an overview of different types of accelerators for matrix multiplication in recent CPUs and to [42, Chapter 20] for details about Intel's AMX units. The use of tensor cores to multiply matrices of floating-point numbers of precisions beyond the word length of the accelerator is studied in [54, 53, 63, 1]. Georganas *et al.* [25] propose a batch-reduce kernel that may also be useful for this purpose.

Before the emergence of tensor cores, classical reference implementations for matrix multiplication and other routines from linear algebra include Blas [6], Lapack [47], Atlas [3], Blis [7] and OpenBlas [61]. We refer to [48, 15, 14, 30, 66, 27, 64, 51] for some of the underlying scientific background. For matrices with multiple precision integer entries, the Chinese remainder theorem (CRT) is typically used to reduce matrix multiplication to modular matrix multiplication, which is then performed using a standard BLAS [16, 13]. When working with very high precisions, FFT-based techniques are also used [35, 33]. The Flint library [32] performs well on a large spectrum of matrix dimensions and integer sizes. We refer to [41, 21] for SIMD-accelerated modular arithmetic and to [4, 63] for CRT-based matrix multiplication on modern hardware. Doubling the precision of a BLAS has been considered in [55].

Our work on JIL also draws inspiration from the compilation of codelets for critical tasks as pioneered in FFTW3 [22, 23] and SPIRAL [56]. Matrix multiplication has also been studied from a similar compilation-oriented perspective [65, 45, 46].

Main contributions

We report on an experimental HPC implementation of integer matrix multiplication for a wide range of matrix dimensions and bit sizes of integers. We have spent a comparable amount of effort on exploiting two types of hardware acceleration, namely the IFMA and AMX extensions in recent Intel processors. In Table 1, we presented an idealized complexity model for large matrix dimensions as a function of the bit size of the coefficients and the integer multiplication method being used. Although the underlying methods are not new, we expect this synthesis to be useful as a guideline for implementors. Our implementations indeed often come close to the theoretical peak performance for which our idealized model is most relevant. This holds especially for our IFMA-based implementation and a bit less for the AMX-based one. For "large" $n \times n$ matrices, say $n \geqslant 128$, we observe that AMX-based algorithms outperform the IFMA-based ones. Although the gain is modest for now, we expect that it can be improved with more work. Another potential advantage of AMX is that we have a more fine-grained control over the bit-precision. Our implementations also outperform the current reference libraries FLINT and NTL.

2 Preliminaries

2.1 Notations

Throughout this paper, we will write $\mathbb{N} := \{0, 1, 2, \ldots\}$ for the set of positive integers, including zero, and $\mathbb{N}_p = \{0, \ldots, 2^{p-1} - 1\}$ for the set of unsigned p-bit integers.

Given a ring $\mathbb{A}$ and $n \in \mathbb{N}$, we write $\mathbb{A}^n$ for the set of vectors $(a_0, \ldots, a_{n-1})$ of length n with entries in $\mathbb{A}$. We will also write $\mathbb{A}[x]_n$ for the set of polynomials of degree $< n$ in x over $\mathbb{A}$. We will assume the natural dense representation for such polynomials $a_0 + \cdots + a_{n-1} x^{n-1}$ by their vectors $(a_0, \ldots, a_{n-1})$ of coefficients.

We formally extend the polynomial notation to the case when x is replaced by an integer radix B, typically of the form $B = 2^\mu$ for some $\mu \in \mathbb{N}$. This allows us to rewrite unsigned integers in $i \in \mathbb{N}_{l\mu}$ as "polynomials" $i = i_0 + i_1 B + \cdots + i_{\ell-1} B^{\ell-1} \in \mathbb{N}_\mu[2^\mu]_\ell$ in the radix. Following terminology from GMP [28], it is customary to call the coefficients $i_0, \ldots, i_{\ell-1} \in \mathbb{N}_\mu$ *limbs*. If $\mathbb{N}_\mu$ is a machine integer type, then we may represent an integer $i \in \mathbb{N}_{l\mu}$ by its vector $(i_0, \ldots, i_{\ell-1})$ of limbs.

For dimensions $r, c \in \mathbb{N}$, we also write $\mathbb{A}^{r \times c}$ for the set of $r \times c$ matrices with entries in $\mathbb{A}$. Unless stated otherwise, we will always assume that matrices $(a_{i,j}) \in \mathbb{A}^{r \times c}$ are represented in row-major order on a computer, as vectors $(a_{0,0}, \ldots, a_{0,c-1}, \ldots, a_{r-1,0}, \ldots, a_{r-1,c-1})$. By "$r \times s \times c$ matrix multiplication", we understand the multiplication of an $r \times s$ matrix with an $s \times c$ matrix into an $r \times c$ matrix.

2.2 SIMD arithmetic

Let μ be the machine precision (or one of the available machine precisions). Standard arithmetic operations on $\mathbb{N}_\mu$ naturally extend to SIMD vectors of width w in $\mathbb{N}_\mu^w$. Whenever a function that only involves basic arithmetic operations needs to be evaluated exactly w times, this can be done replacing all operations by their SIMD versions.

For instance, given a matrix $(a_{i,j}) \in \mathbb{N}_\mu^{2\times 2}$ and a vector $(b_j) \in \mathbb{N}_\mu^2$, consider the function that computes the matrix-vector product $(a_{0,0}\, b_0 + a_{0,1}\, b_1, a_{1,0}\, b_0 + a_{1,1}\, b_1) \in \mathbb{N}_\mu^2$. If we need to evaluate this function w times, then the simplest solution is to present the input as a matrix $(a_{i,j}) \in (\mathbb{N}_\mu^w)^{2\times 2}$ and a vector $(b_j) \in (\mathbb{N}_\mu^w)^2$, in which case the exact same formula $(a_{0,0}\, b_0 + a_{0,1}\, b_1, a_{1,0}\, b_0 + a_{1,1}\, b_1) \in (\mathbb{N}_\mu^w)^2$ yields all w matrix-vector products in SIMD format.

We call this way of vectorizing the *pure SIMD mode*. This works best whenever a given function needs to be evaluated a multiple of w times. This also assumes that all data types are replaced functorially by their *standard* vectorized versions. For instance, the standard vectorization of the matrix type $\mathbb{N}_\mu^{r\times c}$ is $(\mathbb{N}_\mu^w)^{r\times \ell}$. The standard vectorization of a multiple precision integer type $\mathbb{N}_{\ell\mu} \cong \mathbb{N}_\mu[2^\mu]_\ell$ consists of working with "limbs" that are vectors in $\mathbb{N}_\mu^w$. Hence SIMD multiple precision integers $a \in \mathbb{N}_{\ell\mu}^w$ are represented as $a = a_0 + a_1 2^\mu + \cdots + a_{\ell-1} 2^{(\ell-1)\mu} \in \mathbb{N}_\mu^w[2^\mu]_\ell$ with $a_0, \ldots, a_{\ell-1} \in \mathbb{N}_\mu^w$ and $B = 2^\mu$. Note that this may not seem to be most "natural", since one might prefer to represent $a \in \mathbb{N}_{\ell\mu}^w$ as $(a_0, \ldots, a_{w-1})$ with $a_0, \ldots, a_{w-1} \in \mathbb{N}_{\ell\mu}$.

SIMD arithmetic can often be used as well for evaluating a function only once. For instance, the multiplication $C := AB$ of two matrices $A \in \mathbb{A}^{r\times s}$ and $B \in \mathbb{A}^{s\times c}$ can benefit from SIMD acceleration whenever $c = \tilde{c}w$ is a multiple of the SIMD width w. Indeed, it suffices to reinterpret B as a matrix $\tilde{B} \in (\mathbb{A}^w)^{s\times\tilde{c}}$ and C as a matrix $\tilde{C} \in (\mathbb{A}^w)^{r\times\tilde{c}}$, while casting A to a matrix $\tilde{A} \in (\mathbb{A}^w)^{r\times s}$ by repeating each of its entries w times. After that, we can compute $\tilde{C} = \tilde{A}\tilde{B}$ in the pure SIMD mode. However, these reinterpretations require some gymnastics at least, and exploiting SIMD arithmetic becomes even more delicate when none of the dimensions r, s, c are a multiple of w. We will come back to this *tinkered SIMD mode* in Section 4.3 below.

2.3 Benchmarks

In this paper, we mainly report on relative timings with respect to the theoretical peak performance. The only exception is the appendix, which contains absolute timings, as well as timings for existing software. All benchmarks were obtained on an Intel Xeon Gold 6538Y+ processor with 32 cores and running at a clock rate of 2.2 GHz. Our timings rely on the `rdtsc` instruction for measuring time in terms of clock cycles. Recall that all our algorithms use only one core of the CPU.

3 Multiple precision arithmetic

3.1 The standard representation

Let μ be our favorite (and supported) machine bit precision for integer arithmetic, like $\mu = 32$ or $\mu = 64$. The *standard* machine representation of an integer $a = a_0 + a_1 2^\mu + \cdots + a_{\ell-1} 2^{(\ell-1)\mu} \in \mathbb{N}_\mu[2^\mu]_\ell$ is the vector $(a_0, \ldots, a_{\ell-1})$ of "its" limbs.

The hardware is assumed to provide an instruction for multiplying two limbs $a, b \in \mathbb{N}_\mu$, with a result in $\mathbb{N}_{2\mu}$ (sometimes, the low and high parts need to be computed separately). For small precisions, multiple precision libraries typically implement integer multiplications $(a_0 + \cdots + a_{\ell-1}B^{\ell-1})(b_0 + \cdots + b_{m-1}B^{m-1})$ by cross multiplying all limbs $a_i b_j$ and summing the results while handling carries appropriately.

3.2 Redundant representations

Unfortunately, carry propagation is unpleasant to vectorize and SIMD units typically do not provide "vector carry registers" (except for certain GPUs that handle this via mask registers). In a SIMD world, it is therefore more convenient to work with *redundant* representations: we still represent integers as polynomials $a = a_0 + a_1 B + \cdots + a_{\ell-1} B^{\ell-1}$ in some radix $B := 2^\mu$, but the limbs $a_0, \ldots, a_{\ell-1} \in \mathbb{N}_{\bar{\mu}}$ are stored with a higher precision $\bar{\mu}$ (which may be different for multiplicands and products). If $a_0, \ldots, a_{\ell-2} \in \mathbb{N}_\mu$, then $a \in \mathbb{N}_{l\mu+\bar{\mu}-\mu}$ and the representation is said to be *normalized*. We will call μ the *bit precision* and $\bar{\mu}$ the *bit capacity*. The difference $\bar{\mu} - \mu$ is the number *nail bits* in the terminology of GMP.

On recent Intel Xeon processors there are two instructions ifmalo and ifmahi for *integer FMAs* (IFMA). Formally, given $c \in \mathbb{N}_{64}$ and $a, b := \mathbb{N}_{52}$, we have

$$\begin{aligned} \mathsf{ifmalo}(c, a, b) &:= (c + (a \cdot b \operatorname{rem} 2^{52})) \operatorname{rem} 2^{64} \\ \mathsf{ifmahi}(c, a, b) &:= (c + (a \cdot b \operatorname{quo} 2^{52})) \operatorname{rem} 2^{64}, \end{aligned}$$

where $s \operatorname{quo} m$ and $s \operatorname{rem} m$ denote the quotient and the remainder of the euclidean division of s by m. The *IFMA representation* is the redundant representation with $\mu = 52$ and $\bar{\mu} = 64$. Given a two-limb integer $c = c_0 + c_1 2^\mu \in \mathbb{N}_{\mu+\bar{\mu}}$, we can accumulate $c \mathrel{+}= a\,b$ the product of two limbs $a, b \in \mathbb{N}_\mu$ using one ifmalo and one ifmahi. As long as $c_0, c_1 < 2^{\bar{\mu}} - 2^\mu$, this computation does not overflow, but c typically does not remain normalized.

On even more recent Intel Xeon processors with AMX support, there is an instruction for matricial FMAs $C \mathrel{+}= AB$, where $A \in \mathbb{N}_8^{16\times 64}$, $B \in \mathbb{N}_8^{64\times 16}$, and $C \in \mathbb{N}_{32}^{16\times 16}$. For individual integer coefficients of the product, this corresponds to working with a redundant *AMX representation* with $\mu = 8$ and $\bar{\mu} = 32$. Note that products of limbs are two limbs long for the IFMA representation, but only one limb long for the AMX representation. Indeed, when using AMX, products and multiplicands are represented using limbs of different 32-bit and 8-bit capacities, respectively.

3.3 Naive multiple precision multiplication

Redundant representations have the advantage that the $\bar{\mu} - \mu$ nail bits can be used to accumulate carries, while postponing normalization to the very end.

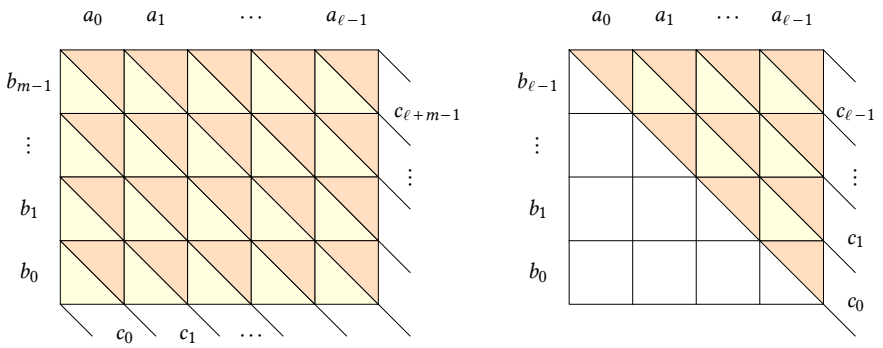


Figure 1. Illustration of a full multiple precision integer product (on the left) and a truncated high product (on the right) using integer FMAs (the lower yellow triangles and the upper orange triangles correspond to ifmalo and ifmahi, respectively).

Consider for instance the multiplication of $a_0 + \cdots + a_{\ell-1} B^{\ell-1}$ and $b_0 + \cdots + b_{m-1} B^{m-1}$ using integer FMAs, where $a_0, \ldots, a_{\ell-1}, b_0, \ldots, b_{m-1} \in \mathbb{N}_{52}$ and $B = 2^{52}$. Assume also that $\max(\ell, m) \leqslant 2^{\bar{\mu}-\mu} = 4096$. Then we first compute the non-normalized product $c = c_0 + \cdots + c_{\ell+m-1} B^{\ell+m-1}$ with $c_0, \ldots, c_{\ell+m-1} \in \mathbb{N}_{64}$, and then normalize it, as follows:

Multiply		*Normalize*	
c_0	$:=$ ifmalo$(0, a_0, b_0)$	c_1	$:= c_1 + (c_0 \operatorname{quo} 2^{52})$
c_1	$:=$ ifmalo$(0, a_0, b_1)$	c_0	$:= c_0 \operatorname{rem} 2^{52}$
c_1	$:=$ ifmahi(c_1, a_0, b_0)	c_2	$:= c_2 + (c_1 \operatorname{quo} 2^{52})$
c_1	$:=$ ifmalo(c_1, a_1, b_0)	c_1	$:= c_1 \operatorname{rem} 2^{52}$
c_2	$:=$ ifmalo$(0, a_0, b_2)$		$\vdots$
	$\vdots$	$c_{\ell+m-1}$	$:= c_{\ell+m-1} + (c_{\ell+m-2} \operatorname{quo} 2^{52})$
$c_{\ell+m-1}$	$:=$ ifmahi$(0, a_{\ell-1}, b_{m-1})$	$c_{\ell+m-2}$	$:= c_{\ell+m-2} \operatorname{rem} 2^{52}$

This algorithm has been represented graphically in Figure 1; the lower and upper triangles correspond to ifmalo and ifmahi operations, respectively.

If $\ell = m$, then the mere highest ℓ limbs of the result can be computed similarly. With the naive algorithm, the right picture in Figure 1 shows that this can be done at roughly half the cost of a full multiplication. This is very useful for fixed point arithmetic, by representing unsigned ℓ-limb fixed point numbers as ℓ-limb integers divided by $2^{52\ell}$. It is interesting to note that the $\bar{\mu} - \mu$ nail bits can be exploited to represent small integer parts in this case.

However, our current implementation is limited to unsigned fixed point numbers. Signed variants can be obtained by representing signed integers $a, b \in \pm\mathbb{N}_{52\ell-1}$ by $\tilde{a} := a + 2^{52\ell-1}$ and $\tilde{b} := b + 2^{52\ell-1}$ in $\mathbb{N}_{52\ell}$, so that $a\,b := \tilde{a}\,\tilde{b} - (\tilde{a} + \tilde{b})\,2^{52\ell-1} + 2^{104\ell-2}$. Alternatively, one may use a two-complement representation for the highest limb and adapt multiple precision multiplication accordingly. But it becomes harder to exploit the extra $\bar{\mu} - \mu$ nail bits in both cases.

3.4 Vector and matrix limbs

We briefly discussed pure SIMD vectorization of multiple precision integers in Section 2.2. The idea to represent such integers as polynomials in 2^{μ} with vectorial limbs also works for redundant representations. This allows us to apply the algorithms from the previous subsection directly to pure SIMD multiple precision integers in $\mathbb{N}_{52}^{8}[2^{52}]_{\ell}$.

In a similar way, matrices of multiple precision integers in $\mathbb{N}_{8\ell}^{r\times c}$ can be rewritten into polynomials in $\mathbb{N}_{8}^{r\times c}[2^{8}]_{\ell}$ with matrix limbs in $\mathbb{N}_{8}^{r\times c}$. Computing a non-normalized product $C = A \cdot B$ with $A \in \mathbb{N}_{8}^{16\times 64}[2^{8}]_{\ell}$, $B \in \mathbb{N}_{8}^{64\times 16}[2^{8}]_{\ell}$, and $C \in \mathbb{N}_{32}^{16\times 16}[2^{8}]_{2\ell-1}$ can then be done naively using ℓ^2 AMX-accelerated matrix multiplications $\mathbb{N}_{8}^{16\times 64} \times \mathbb{N}_{8}^{64\times 16} \to \mathbb{N}_{32}^{16\times 16}$. Note that we do not compute low and high parts separately in this case. When doing high ℓ-limb multiplication in this way, this means that we need to do $\ell\,(\ell + 1)/2$ multiplications, which is slightly more than one half of the number for full products.

3.5 Karatsuba multiplication

For large $n \times n$ matrix multiplications with ℓ-limb coefficients, using a naive algorithm for polynomial multiplication becomes suboptimal. If $\ell > 1$ is small, then it becomes better to use Karatsuba's algorithm [43].

Given a ring $\mathbb{A}$, the traditional Karatsuba trick reduces one multiplication $R = PQ$ with $P, Q \in \mathbb{A}[x]_2$ into three multiplications in $\mathbb{A}$ and several additions and subtractions using the formulas $R_0 = P_0\, Q_0$, $R_2 = P_1\, Q_1$, and $R_1 = (P_0 + P_1)\,(Q_0 + Q_1) - R_0 - R_2$. This trick generalizes to the case where $\mathbb{A}$ is replaced by $\mathbb{A}[x]_\ell$ and x by x^ℓ, in which case the multiplication of two polynomials $P, Q \in \mathbb{A}[x]_{2\ell} \cong \mathbb{A}[x]_\ell[x^\ell]_2$ essentially reduces to three multiplications of polynomials in $\mathbb{A}[x]_\ell$. These latter polynomials can be computed recursively using the same trick.

When working with multiple precision integers $a, b \in \mathbb{N}_\mu[2^\mu]_2$ one technical difficulty is that $a_0 + a_1$ and $b_0 + b_1$ are in $\mathbb{N}_{\mu+1}$ and not necessarily in $\mathbb{N}_\mu$. This makes it better to work with limbs of bit-size $\mu - 1$ instead of μ. If our input integers used μ-bit limbs, then this requires them to be rewritten into the $(\mu-1)$-bit limb representation and *vice versa* for the output. When using Karatsuba's algorithm recursively for 2^κ-limb integers, we may either directly use limbs of bit-size $\mu - \kappa$ (which yields a multiplication algorithm for $2^\kappa(\mu - \kappa)$-bit integers), or do the limb rewritings recursively as well (which yields a multiplication algorithm for $(2^\kappa(\mu-2)+2)$-bit integers).

Karatsuba's trick can also be applied to matrices $A, B \in \mathbb{N}_{\mu-\kappa}^{n\times n}[2^\mu]_{2^\kappa}$. In that case, we need to do $3^\kappa n^3$ multiplications in $\mathbb{N}_\mu$, but only $O(3^\kappa n^2)$ additions and subtractions, which becomes negligible for large n. Asymptotically, we thus gain a factor $4^\kappa / 3^\kappa$ with respect to naive integer matrix multiplication.

Toom generalized Karatsuba's trick [62] and gave a way to reduce the multiplication of two polynomials in $\mathbb{A}[x]_\ell$ to $2\,\ell - 1$ multiplications in $\mathbb{A}$ at the cost of an increased loss of bit-precision for the integer variant. Interestingly, Karatsuba's trick can also be used directly for $P, Q \in \mathbb{A}[x]_\ell$, while losing a single bit of precision.

The idea (see also [36, Section 4.4.1], [44], [9, Exercise 1.4], and [58]) is to observe that the contribution of $P_i Q_j + P_j Q_i$ to $(PQ)_{i+j}$ can be computed using $P_i Q_j + P_j Q_i = (P_i + P_j)(Q_i + Q_j) - P_i Q_i - P_j Q_j$, for all $0 \leqslant i < j < \ell$. In this way, we need to first compute the ℓ products $P_0 Q_0, \ldots, P_{\ell-1} Q_{\ell-1}$ and then $\ell(\ell-1)/2$ further products of the form $(P_i + P_j)(Q_i + Q_j)$, which yields a total number of $\ell(\ell+1)/2$ products. If we just want the high coefficients $(PQ)_{\ell-1}, \ldots, (PQ)_{2\ell-1}$, then we only need to do $\ell + \lfloor \ell/2 \rfloor \lceil \ell/2 \rceil \sim \ell^2/4$ multiplications. This is actually just as good as Toom's algorithm for $\ell = 3$ and almost as good for $\ell = 4$ and $\ell = 5$.

3.6 Chinese remaindering

A well known method for multiplying two matrices $A, B \in \mathbb{N}_\ell^{n\times n}$ is based on the Chinese remainder theorem. We first pick $2\,\ell$ pairwise coprime moduli $m_0, \ldots, m_{2\ell-1} \in \mathbb{N}_\mu$. Best is to choose them as large as possible while fitting into μ bits. Let $M := m_0 \cdots m_{2\ell-1}$ and assume that $AB \in \mathbb{A}_{2\ell\mu-\epsilon}$ for some small $\epsilon > 0$ with $M \geqslant 2^{2\ell\mu-\epsilon}$.

In order to compute $C = AB$, we first reduce A and B modulo m_i for $i = 0, \ldots, 2\,\ell - 1$. There is an elegant way to evaluate the map $\mathrm{CRT}: \mathbb{N}_{\ell\mu} \to \mathbb{N}_\mu^{2\ell}; a \mapsto (a \operatorname{rem} m_i)_{i<2\ell}$ using a vector-matrix product [13]:

$$\begin{pmatrix} \tilde{a}_0 & \cdots & \tilde{a}_{2\ell-1} \end{pmatrix} := \begin{pmatrix} a_0 & \cdots & a_{\ell-1} \end{pmatrix} \begin{pmatrix} 1 & \cdots & 1 \\ 2^\mu \operatorname{rem} m_0 & & 2^\mu \operatorname{rem} m_{2\ell-1} \\ \vdots & & \vdots \\ 2^{(\ell-1)\mu} \operatorname{rem} m_0 & \cdots & 2^{(\ell-1)\mu} \operatorname{rem} m_{2\ell-1} \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} a \operatorname{rem} m_0 & \cdots & a \operatorname{rem} m_{\ell-1} \end{pmatrix} = \begin{pmatrix} \tilde{a}_0 \operatorname{rem} m_0 & \cdots & \tilde{a}_{\ell-1} \operatorname{rem} m_{\ell-1} \end{pmatrix}.$$

Taking the limbs of the entries of A as our rows $(a_0, \ldots, a_{\ell-1})$, the computation of $A \operatorname{rem} m_i$ for $i = 0, \ldots, 2\,\ell - 1$ thus essentially reduces to an $n^2 \times \ell \times (2\,\ell)$ matrix product over $\mathbb{N}_\mu$ and similarly for B. We next compute

$$C \operatorname{rem} m_i := (A \operatorname{rem} m_i)(B \operatorname{rem} m_i) \operatorname{rem} m_i$$

for $i = 0, \ldots, 2\,\ell - 1$. We finally recover C from $C \operatorname{rem} m_0, \ldots, C \operatorname{rem} m_{2\ell-1}$ by applying the Chinese remainder theorem. Consider the map $\mathrm{CRT}^{-1}: \mathbb{N}_\mu^{2\ell} \to \mathbb{N}_{2\ell\mu}; (r_i)_{i<2\ell} \mapsto x$ with $0 \leqslant x < M$ and $x \operatorname{rem} m_i = r_i$ for $i = 0, \ldots, 2\,\ell - 1$. We have

$$x = \frac{M}{m_0}\, u_0\, r_0 + \cdots + \frac{M}{m_{2\ell-1}}\, u_{2\ell-1}\, r_{2\ell-1},$$

Method	1	2	3	4	5	6	8	10	12	16	ℓ	Bit loss
Naive	1	4	9	16	25	36	64	100	144	256		0
Karatsuba	1	3	6	10	15	21	36	55	78	136		ℓ
Karatsuba^2				9		18	30	45	63	108		$2\,\ell$ or $^3/_2\,\ell$
CRT	2	4	6	8	10	12	16	20	24	32		ϵ
Naive, high	1	3	6	10	15	21	36	55	78	138		0
Karatsuba, high	1	3	5	8	11	15	24	35	48	80		ℓ
CRT, high	2	4	6	8	10	12	16	20	24	32		ϵ
Naive, IFMA, high	1	4	9	16	25	36	64	100	144	256		0
Karatsuba, IFMA, high	1	4	8	13	19	26	43	64	89	151		ℓ
CRT, IFMA, high	4	8	12	16	20	24	32	40	48	64		ϵ

Table 1. Overview of the factor $\mathsf{E}(\ell)$ as a function of ℓ for various multiplication schemes, as well as the corresponding losses in bit precision. For CRTs, the bit loss is $\epsilon = 1$ for $\mu = 52$ or $\mu = 8$ and $\ell \leqslant 8$, but $\epsilon = 5$ for $\mu = 8$ and $\ell = 16$. The first four rows correspond to full multiplication, whereas the other rows correspond to high multiplication. The last three rows counts the number of IFMA operations that are required if low and high machine products need to be computed separately. For Karatsuba multiplication, we do not recurse, except for Karatsuba^2, which recurses once.

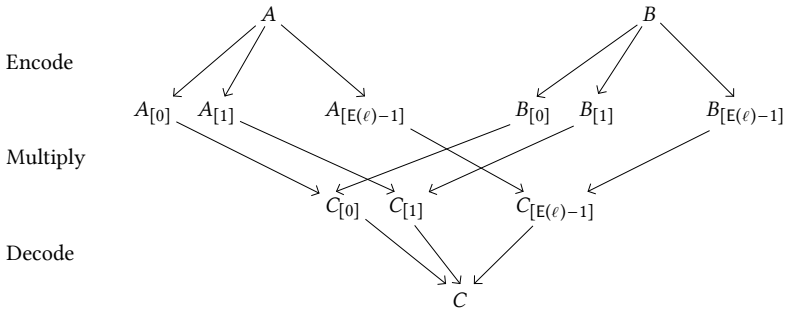
where u_i is the inverse of M/m_i modulo m_i for $i = 0, \ldots, 2\,\ell - 1$. After precomputing the 2^μ-adic expansions $\frac{M}{m_i} u_i = \alpha_{i,0} + \cdots + a_{i,2\ell-1} 2^{(2\ell-1)\mu}$, it is again possible to compute CRT^{-1} using a vector-matrix product [13]:

$$\begin{pmatrix} \tilde{x}_0 & \cdots & \tilde{x}_{2\ell-1} \end{pmatrix} := \begin{pmatrix} r_0 & \cdots & r_{2\ell-1} \end{pmatrix} \begin{pmatrix} \alpha_{0,0} & \cdots & \alpha_{0,2\ell-1} \\ \vdots & & \vdots \\ \alpha_{2\ell-1,0} & \cdots & \alpha_{2\ell-1,2\ell-1} \end{pmatrix} \tag{2}$$
$$x = \left(\tilde{x}_0 + \cdots + \tilde{x}_{2\ell-1} 2^{(2\ell-1)\mu}\right) \operatorname{rem} M.$$

In a similar way as above, the recovery of C from $C \operatorname{rem} m_0, \ldots, C \operatorname{rem} m_{2\ell-1}$ thus essentially boils down to an $n^2 \times (2\,\ell) \times (2\,\ell)$ matrix product over $\mathbb{N}_\mu$. In order to ease the computation of the remainder with respect to M, it is sometimes preferred to first compute the remainders of $a_i r_i$ modulo m_i and then to work with the 2^μ-adic expansions of M/m_i instead of $u_i M/m_i$. In this way, the quotient x/M is always bounded by $2\,\ell$.

3.7 Summary and asymptotic complexity

Each of the schemes that we described so far can be regarded as a way to reduce the multiplication $C = AB$ of two matrices in $\mathbb{N}_{\ell\mu}^{n\times n}$ to $\mathsf{E}(\ell)$ multiplications $A_{[0]} B_{[0]}, \ldots, A_{[2\ell-1]} B_{[2\ell-1]}$ of matrices in $\mathbb{N}_\mu^{n\times n}$, through the use of an appropriate encoding:



(3)

The cost of the reduction may depend on ℓ, but scales linearly with the size n^2 of the matrices. Since the cost of $\mathsf{E}(\ell)$ multiplications of matrices in $\mathbb{N}_\mu^{n\times n}$ is proportional to $\mathsf{E}(\ell)\, n^3$, the cost of the reduction is negligible for large n. For large matrices, it is therefore best to select the scheme for which $\mathsf{E}(\ell)$ is minimal. The encoding/decoding process may also induce a loss of precision, which should be kept minimal as well.

Table summarizes the factors $\mathsf{E}(\ell)$ and the loss of precision for the methods from the previous sections. For our implementations, we only considered the naive method, Karatsuba's method, and Chinese remaindering. In certain cases, Toom's algorithm or FFT-based algorithms may also be of interest.

```
template<typename C> void
matmul_222 (C* c, C* a, C* b) {
  for (int k=0; k<2; k++)
    for (int i=0; i<2; i++)
      for (int j=0; j<2; j++)
        if (j == 0)
          c[2*i+j] = a[2*i+k] * b[2*k+j];
        else
          c[2*i+j]+= a[2*i+k] * b[2*k+j];
}
```

record ⟶

$$
\begin{aligned}
&\mathrm{in}(a_{00}, a_{01}, \ldots, b_{10}, b_{11})\\
&c_{00} := a_{00} \times b_{00}\\
&c_{01} := a_{00} \times b_{01}\\
&c_{10} := a_{10} \times b_{00}\\
&c_{11} := a_{10} \times b_{01}\\
&c_{00} := \mathrm{fma}(a_{01}, b_{10}, c_{00})\\
&c_{01} := \mathrm{fma}(a_{01}, b_{11}, c_{01})\\
&c_{10} := \mathrm{fma}(a_{11}, b_{10}, c_{10})\\
&c_{11} := \mathrm{fma}(a_{11}, b_{11}, c_{11})\\
&\mathrm{out}(c_{00}, c_{01}, c_{10}, c_{11})
\end{aligned}
$$

Figure 2. Illustration of the recording process of a program as an SLP. For the purpose of readability, we have chosen nice names $a_{00}, a_{01}, \ldots, c_{11}$ for the data fields of the SLP. Inside JIL, the data fields are rather referred to through indices $0, 1, \ldots, 11$.

4 Multiplying small matrices using IFMA

In this section, we consider the problem of multiplying "small" integer matrices. Since AMX-acceleration is mainly interesting for quite large multiplications (of size at least $16 \times 64 \times 16$), we focus on SIMD-accelerated IFMA-based algorithms.

4.1 Implementation strategy

Since we assume both n and ℓ to be "small", we will use naive algorithms for both matrix and polynomial multiplication. Here the qualifier "naive" is meant to capture the family of all algorithms that perform $n^3 \ell^2$ pairs of ifmalo/ifmahi integer FMAs in total. However, there are many such "naive" algorithms and the precise order in which we do perform the integer FMAs matters, since this crucially influences how well we can keep coefficients in registers and how much pressure we put on registers.

Instead of implementing various multiplication strategies directly in a language like C++, our strategy is to rely on the JIL library [2, 40] for computations and JIT compilation of straight-line programs (SLPs): we implement a large amount of strategies and ways to combine strategies as symbolic programs. The JIL library provides highly efficient mechanisms to record the execution of these programs as SLPs and to JIT compile the resulting SLPs into highly efficient machine code. The recording plus compilation time is typically about 1000 times larger than a single execution of the generated code, which is one to three orders of magnitude faster than traditional compilers.

Let us illustrate this with a short example. We recall that an SLP is just a sequence of arithmetic operations. The operations belong to a fixed signature like $\Sigma = \{+, -, \times, \mathrm{fma}\}$ and we operate on a finite number of data fields that are all of the same type. This type could either be a scalar type like $\mathbb{N}_{64}$ or FP32, or a SIMD vector type like $\mathbb{N}_8^{64}$. A finite number of the data fields are designated to be "input" and "output" fields.

Now given a generic function operating on data fields of the same time, but which may involve loops and function calls, JIL provides a way to record the trace of its execution as an SLP. In Figure 2, we illustrated the result of recording a C++ template for $2 \times 2 \times 2$ matrix multiplication as an SLP.

When implementing multiplication algorithms in this way, the focus is not on the design of efficient C++ code templates, but rather on the design of clean templates, whose recording produces good SLPs. This makes it easy and efficient to build new strategies on top of other ones, like a multiplication algorithm for polynomials in $\mathbb{N}_{64}^{2\times 2}[x]_2$ on top of a multiplication algorithm for matrices in $\mathbb{N}_{64}^{2\times 2}$. JIL also provides standard mechanisms for composition and tensor products of maps, reindexation, etc. We refer to [40] for more information.

4.2 Pure SIMD mode

Let us first consider the problem of multiplying integer matrices in the pure SIMD mode, using IFMA-based arithmetic. This means that our multiplicands are $n \times n$ matrices in $(\mathbb{N}_{52}^w[2^{52}]_\ell)^{n\times n}$ with coefficients that are SIMD multiple precision integers in $\mathbb{N}_{52}^w[2^{52}]_\ell$, whose limbs are SIMD vectors in $\mathbb{N}_{52}^w$.

Multiplicands in $(\mathbb{N}^8_{52\ell})^{n\times n}$

n \ ℓ	1	2	3	4	6	8	12	16
1	4.16	1.54	1.34	1.20	1.04	0.94	0.89	1.18
2	1.13	1.09	1.03	1.05	1.07	0.98	0.98	1.63
3	0.97	1.29	1.22	1.14	1.02	1.22	1.39	1.74
4	1.12	1.32	1.19	1.11	1.35	1.22	1.51	1.81
6	0.93	1.35	1.71	1.62	1.47	1.30	1.43	3.57
8	1.04	1.96	1.87	1.68	1.43	1.25	3.50	6.98
12	1.17	1.90	1.79	1.62	2.69	4.46	5.13	6.57
16	1.53	2.09	1.85	4.63	5.61	4.44	5.15	8.28

Multiplicands in $(\mathbb{N}^8_{52})^{n\times n}[2^{52}]_\ell$

n \ ℓ	1	2	3	4	6	8	12	16
1	3.94	1.48	1.28	1.15	0.99	0.91	0.86	1.15
2	1.11	1.07	0.97	1.00	1.02	1.07	1.35	1.58
3	0.95	0.97	1.05	1.06	1.04	1.44	1.46	1.50
4	1.09	1.10	1.11	1.12	1.62	1.65	1.68	1.69
6	0.92	0.86	1.30	1.34	1.33	1.36	1.37	1.72
8	1.05	1.57	1.62	1.61	1.59	1.58	3.70	5.37
12	1.16	1.28	1.25	1.22	1.58	3.17	3.98	4.38
16	1.45	1.55	1.78	3.82	5.69	5.73	5.72	5.78

Table 2. Pure SIMD integer matrix multiplication using a naive algorithm and IFMA. We show the ratios with respect to the theoretical peak performance of two IFMAs per cycle.

Two things matter for the design of efficient "naive" algorithms. We first should decide whether we view our multiplicands as matrices in $(\mathbb{N}^8_{52\ell})^{n\times n}$ with SIMD integer entries or as polynomials in $(\mathbb{N}^8_{52})^{n\times n}[2^{52}]_\ell$ with SIMD matrix "limbs". The first point of view tends to be faster when ℓ is large with respect to n, whereas the second one tends to be faster when n is large with respect to ℓ.

Secondly, divide and conquer algorithms tend to reduce register spilling. For instance, $r\times s\times c$ matrix products can recursively be reduced to smaller matrix products by halving r, c, or s. However, recursing too far down may increase register pressure. We implemented an *ad hoc* strategy that recurses until a certain threshold is reached (namely 4). In the future, we plan to automatically generate many strategies of this kind and select the best one.

After implementing the desired multiplication strategy as a `map` in JIL, we record one evaluation of this map as an SLP, and JIT compile the SLP into machine code.

Remark 1. It is interesting to note that the SLP framework allows us to *physically* work with a specific memory layout (say $n\times n$ matrices with coefficients in $\mathbb{N}^8_{52\ell}$), while applying an algorithm that works *logically* with another representation (like polynomials in 2^{52} with coefficients in $(\mathbb{N}^8_{52})^{n\times n}$). Indeed, the alternative representation just corresponds to a permutation of coefficients in memory. JIL provides an abstraction (the `aliased` type in `domain.hpp`) that allows to automate this kind of re-interpretations (and which we indeed used for the implementation described in this section).

Table 2 shows the ratios of our timings for an $\ell\times\ell\to 2\,\ell$ limb long $n\times n$ matrix multiplication divided by the theoretical time to the theoretical time to perform $2\,n^3\,\ell^2$ integer FMAs over $\mathbb{N}^8_{52}$ (in theory, our processor has a throughput of two integer FMAs per cycle). The left-hand table shows the ratios for the standard representation; the right-hand one concerns the alternative representation of our multiplicands as polynomials in the radix 2^{52} with matrix coefficients.

Interestingly, the observed ratios sometimes drops *below* the theoretical peak performance. This is probably due to subtleties with cycle counters in recent CPUs, in which different cores may run at different clock speeds. It might also indicate that the processor can sometimes pipeline IFMA instructions a bit better than officially announced.

We also recall that the generated codelet unrolls the entire computation. For $n=\ell=16$, this means that the generated program performs 2^{21} IFMAs. The codelet therefore takes more than 12Mb in memory and does not even fit in the L3 cache. This explains the dramatic increase in the ratios for large n and/or ℓ.

4.3 Tinkered SIMD mode

Let us now turn to the computation of a single $n\times n\times n$ matrix product $C=AB$ over $\mathbb{N}_{52\ell}$. In Section 2.2, we already described a reduction to the computation of a $n\times n\times(n/8)$ matrix product $\tilde{C}=\tilde{A}\tilde{B}$ over $\mathbb{N}^8_{52\ell}$, whenever n is divisible by 8. We may then use an algorithm in pure SIMD mode to compute this product.

In fact the situation is even a bit better than that: instead of casting A into a matrix $\tilde{A}\in(\mathbb{N}^8_{52\ell})^{n\times n}$, it is better to regard A directly as an $n\times(n/8)$ SIMD matrix over $\mathbb{N}^8_{52\ell}$, and to replicate individual coefficients of A across lanes whenever needed. This more compact representation of A has the advantage that we

Multiplicands in $\mathbb{N}_{52\ell}^{n\times n}$

n	1	2	3	4	6	8	12	16	ℓ
2	4.77	2.55	2.18	1.81	2.14	1.59	1.30	1.28	
4	1.66	1.34	1.29	1.17	1.13	1.02	1.02	1.68	
6	1.73	1.70	1.59	1.40	1.54	1.58	1.84	2.00	
8	1.21	1.11	1.17	1.10	1.39	1.21	1.46	1.74	
12	1.02	1.56	1.90	1.69	1.49	1.34	1.54	4.40	
16	0.97	1.82	1.78	1.63	1.44	1.31	3.80	6.76	
24	1.09	2.02	1.87	1.66	2.92	3.98	5.52	6.77	
32	1.61	2.15	2.71	3.94	5.14	4.71	5.49	7.01	

Multiplicands in $\mathbb{N}_{52}^{n\times n}[2^{52}]_\ell$

n	1	2	3	4	6	8	12	16	ℓ
2	4.64	2.47	2.13	1.77	2.05	1.57	1.29	1.27	
4	1.66	1.35	1.31	1.17	1.03	0.95	0.85	1.33	
6	1.75	1.46	1.57	1.43	2.06	1.91	2.01	1.87	
8	1.23	1.12	1.09	0.98	1.59	1.53	1.51	1.47	
12	1.02	1.00	1.48	1.41	1.36	1.30	1.27	1.71	
16	0.97	1.53	1.49	1.38	1.33	1.25	2.55	3.76	
24	1.06	1.32	1.26	1.22	2.12	3.30	3.92	3.91	
32	1.60	1.77	2.38	3.72	5.57	5.34	5.20	5.18	

Table 3. Tinkered SIMD integer matrix multiplication using a naive algorithm and IFMA. We show the ratios with respect to the theoretical peak performance of two IFMAs per cycle.

may typically keep A wholly in registers, after which obtaining coefficients from A does not involve memory accesses.

If n is not divisible by 8, then more work is required to benefit from SIMD acceleration: for an $r\times s\times c$ product such that rsc is divisible by 8, we may write $r=\tilde{r}w_r$, $s=\tilde{s}w_s$, and $c=\tilde{c}w_c$ for some w_r, w_s, w_c with $w_r w_s w_c = 8$. We next redundantly encode $A\in\mathbb{N}_{52\ell}^{r\times s}$, $B\in\mathbb{N}_{52\ell}^{s\times c}$, and $C\in\mathbb{N}_{52\ell}^{r\times c}$ by matrices $\tilde{A}\in(\mathbb{N}_{52\ell}^{8})^{\tilde{r}\times\tilde{s}}$, $\tilde{B}\in(\mathbb{N}_{52\ell}^{8})^{\tilde{s}\times\tilde{c}}$, $\tilde{C}\in(\mathbb{N}_{52\ell}^{8})^{\tilde{r}\times\tilde{c}}$ by taking

$$\begin{aligned}(\tilde{A}_{i,k})_{(iw_s+k')w_c+j'} &:= A_{w_ri+i',w_sk+k'}\\ (\tilde{B}_{k,j})_{(iw_s+k')w_c+j'} &:= B_{w_sk+k',w_cj+j'}\\ \sum_{k'}(\tilde{C}_{i,j})_{(iw_s+k')w_c+j'} &=: C_{w_ri+i',w_cj+j'},\end{aligned}\tag{4}$$

where $0\leqslant i<\tilde{r}$, $0\leqslant j<\tilde{s}$, $0\leqslant k<\tilde{c}$ and $0\leqslant i'<w_r$, $0\leqslant j'<w_s$, $0\leqslant k'<w_c$. Every SIMD multiplication in $\mathbb{N}_{52\ell}^{r\times c}$ then computes a $w_r\times w_s\times w_c$ matrix product with coefficients in $\mathbb{N}_{52\ell}$. If $w_s>1$, then some of the lanes of $\tilde{C}$ need to be summed in order to retrieve C.

The standard SLP signature in JIL includes instructions for permutations of lanes for SIMD base types (mathematically speaking, really maps on lanes, because replication of lanes is also allowed). Consequently, the tinkered SIMD multiplication algorithms can still be recorded into SLPs and compiled within the JIL framework. As a consequence of Remark 1, we note in addition that any of the pure SIMD multiplication strategies can naturally be used in conjunction with any of the tinkered encoding strategies.

Table 3 shows timings that we obtained, with similar conventions as for Table ?. This time, we divided the total number of cycles by $n^3\ell^2/8$ instead of $n^3\ell^2$.

4.4 Fixed point entries

One important application of integer matrix multiplication is linear algebra over fixed point numbers. In that case, we only need to compute the highest ℓ limbs of the product, which saves about half of the computations. Table 4 reports on the obtained timings. We recall that a shortcoming of our implementation is that it is currently limited to unsigned fixed point numbers.

n	1	2	3	4	6	8	12	16	ℓ
2		2.67		1.86	1.91	1.88	1.37	1.10	
4	1.91	1.47	1.29	1.71	1.13	1.03	0.90	0.83	
6		1.35		1.04	1.14	0.94	1.33	1.14	
8	1.31	1.13	1.07	0.98	0.93	0.96	0.97	0.89	
12	1.08	0.89	1.00	1.29	1.23	1.11	1.06	0.99	
16	0.99	0.91	1.22	1.17	1.14	1.06	1.01	1.35	
24	0.88	1.08	1.26	1.22	1.16	1.08	2.95	3.41	
32	1.06	1.45	1.37	1.26	1.86	2.93	3.45	3.33	

Table 4. Tinkered SIMD $n\times n$ matrix multiplication for ℓ limb fixed point entries and using a naive algorithm and IFMA. We show the ratios with respect to the theoretical peak performance of two IFMAs per cycle.

5 Multiplying large matrices using IFMA

5.1 Implementation strategy

The simplest way to accelerate a large matrix multiplication $C := AB$ via JIL is to consider A, B, and C as block matrices. We use JIL for the compilation of efficient SLPs to multiply or multiply-accumulate blocks. The block matrices themselves are multiplied using conventional C++ code. The last possibly incomplete row and column are treated using zero-padding or recursive decomposition. The block sizes should be carefully chosen and may differ for A, B, and C. By taking sufficiently large blocks, we intend to make the overhead of the C++ part of the code as small as possible, so that the total cost essentially boils down to multiplications of blocks.

One goal of the above strategy is that JIL can focus on compiling SLPs of a fixed size and that we can use a higher level language like C++ for the rest, with minimal overhead. However, for practical block sizes, the overhead of the C++ code is small, but not necessarily negligible, which makes it desirable to further improve the interface between JIT compiled SLPs and their use within high level C++ programs. For the purposes of this paper, we therefore extended JIL with a new `promise` abstraction, which allows us to JIT compile slightly more complex programs than mere SLPs.

For instance, an extremely common scenario is to execute the same SLP on a vector of n successive input and output chunks. This happens for example when encoding all $n = r \times c$ entries of an integer matrix in $\mathbb{N}_{52\ell}^{r\times c}$ into CRT format. JIL now contains a "loop promise" to cover this scenario. Our implementation first rewrites the SLP into a new one that can handle 8 entries at a time in SIMD fashion (this requires transpositions in memory to work in the SIMD representation), or even a multiple of 8 entries at a time via unrolling. In the future, we also plan to automatically determine and factor out loop invariants.

Back to matrix multiplication, another common scenario is that the input of the SLP has first to be gathered from memory and/or that the output needs to be scattered to memory, as we explained in Section 3.7 and (3). For instance, Chinese remaindering essentially reduces the multiplication of two matrices in $\mathbb{N}_{52\ell}^{n\times n}$ to $2\,\ell$ multiplications of matrices in $\mathbb{N}_{52}^{n\times n}$. Now it is more natural to CRT encode an input matrix in $\mathbb{N}_{52\ell}^{n\times n}$ as a vector of matrices in $(\mathbb{N}_{52}^{n\times n})^{2\ell}$ rather than a matrix with vector entries in $(\mathbb{N}_{52}^{2\ell})^{n\times n}$. The latter case corresponds to the loop scenario that we described above, whereas the first one requires us to submit the output to an additional $(2\,\ell)\times n^2$ matrix transposition over $\mathbb{N}_{52}$. In order to reduce unnecessary memory accesses, we preferred to introduce a new scenario that scatters entries in $\mathbb{N}_{52}^{2\ell}$ directly into the $2\,\ell$ desired matrices $\mathbb{N}_{52}^{n\times n}$ while processing the CRTs.

5.2 Relative timings with respect to theoretical peak performance

We have tested most of the strategies of Table 1 for the multiplication of large matrices using integer FMAs. For the "Multiply" step of (3), we use the blocking technique that we described above, and also a tinkered SIMD as in Section 4.3 for the multiplication of blocks. We report on relative timings with respect to the theoretical peak performance. Tables with absolute timings are given in Appendix A.3.

In Table 5, we consider long $\ell \times \ell \to 2\,\ell$ limb multiplications of $n \times n$ matrices for the naive and the Chinese remaindering (CRT) strategies. More precisely, we show the number of cycles that it takes to compute the matrix product, divided by $\mathsf{E}(\ell)\,n^3$, the theoretical number of cycles to perform all IFMA operations in the multiplication step of (3). In Table 6, we show timings for the non-recursive variant of Karatsuba's algorithm, both in the case of long $\ell \times \ell \to 2\,\ell$ limb multiplication and high $\ell \times \ell \to \ell$ limb multiplication.

The tables show that we are not far from the theoretical peak performance. The factors from Table 1 can therefore guide the design of efficient algorithms when n gets large. In particular, it is interesting to note that Karatsuba's algorithm is most efficient for high multiplication until five limbs.

There is still quite some room for further improvements. Our implementation of the naive algorithm was done first and based on a suboptimal implementation of matrix transposition for the encoding and decoding steps; this explains the poor performance for small ℓ. In general, with more work, we expect that it should be possible to reach factors that are very close to 1 (at least for, say, $n \geqslant 256$ and $\ell \leqslant 8$).

	1	2	3	4	6	8	12	16
32	4.54	2.66	2.27	2.11	2.06	1.97	2.05	2.02
64	2.62	1.65	1.49	1.40	1.43	1.42	1.43	1.39
96	1.96	1.37	1.31	1.26	1.25	1.23	1.22	1.22
128	1.73	1.32	1.24	1.20	1.18	1.17	1.17	1.18
192	1.54	1.21	1.16	1.12	1.11	1.12	1.17	1.17
256	1.42	1.19	1.15	1.13	1.12	1.12	1.22	1.20
384	1.25	1.10	1.08	1.07	1.18	1.16	1.15	1.13
512	1.24	1.13	1.27	1.24	1.20	1.17	1.16	1.15
768	1.20	1.30	1.23	1.19	1.16	1.13	1.14	1.13
1024	1.46	1.27	1.21	1.18	1.15	1.13	1.12	1.11
1536	1.46	1.32	1.28	1.26	1.25	1.23	1.24	1.21
2048	1.39	1.30	1.26	1.24	1.22	1.18	1.17	1.16

	1	2	3	4	6	8	12	16
32	2.28	2.39	2.46	2.64	2.86	3.22	4.13	5.35
64	1.48	1.51	1.59	1.66	1.91	2.18	2.67	3.16
96	1.18	1.27	1.38	1.44	1.56	1.75	2.06	2.37
128	1.14	1.24	1.27	1.30	1.37	1.50	1.72	1.97
192	1.12	1.12	1.14	1.17	1.21	1.29	1.45	2.45
256	1.10	1.10	1.13	1.15	1.18	1.85	2.12	2.37
384	1.06	1.07	1.08	1.52	1.61	1.68	1.79	1.93
512	1.07	1.36	1.39	1.44	1.49	1.54	1.62	1.71
768	1.25	1.31	1.32	1.34	1.36	1.40	1.48	1.54
1024	1.26	1.31	1.32	1.33	1.34	1.37	1.41	1.46
1536	1.42	1.44	1.45	1.45	1.45	1.46	1.51	1.51
2048	1.35	1.36	1.36	1.37	1.40	1.42	1.40	1.42

Table 5. Relative timings for one long $\ell \times \ell \to 2\,\ell$ limb $n \times n$ matrix product using IFMA. The left-hand and right-hand sides respectively indicate the relative timings (with respect to theoretical peak performance) for the naive and the CRT strategies.

	1	2	3	4	6	8	12	16
32	2.54	2.32	2.39	2.30	2.40	2.49	2.84	3.20
64	1.52	1.49	1.56	1.54	1.72	1.76	1.80	4.23
96	1.22	1.28	1.41	1.40	1.41	1.43	2.92	3.08
128	1.11	1.28	1.27	1.27	1.25	2.41	2.51	2.89
192	1.18	1.17	1.16	1.14	1.91	2.01	2.27	2.43
256	1.12	1.13	1.13	1.70	1.83	1.95	2.06	2.18
384	1.06	1.06	1.44	1.55	1.69	1.69	1.76	1.81
512	1.05	1.38	1.48	1.54	1.56	1.56	1.59	1.62
768	1.31	1.39	1.42	1.42	1.44	1.45	1.46	1.48
1024	1.33	1.35	1.36	1.36	1.37	1.37	1.37	1.37
1536	1.42	1.42	1.42	1.42	1.42	1.41	1.40	1.42
2048	1.40	1.40	1.39	1.38	1.39	1.40	1.40	1.36

	1	2	3	4	6	8	12	16
32	2.96	2.23	2.14	2.11	2.17	2.18	2.37	2.74
64	1.64	1.38	1.28	1.34	1.51	1.71	1.70	1.77
96	1.32	1.20	1.16	1.31	1.43	1.44	1.38	2.84
128	1.16	1.27	1.17	1.21	1.26	1.21	2.31	2.36
192	1.21	1.23	1.06	1.10	1.07	1.83	1.96	2.17
256	1.18	1.16	0.98	1.00	1.66	1.75	1.92	2.00
384	1.13	1.03	1.33	1.40	1.52	1.57	1.63	1.68
512	1.06	1.45	1.27	1.39	1.44	1.46	1.51	1.49
768	1.13	1.40	1.24	1.30	1.34	1.36	1.37	1.38
1024	1.34	1.36	1.19	1.24	1.28	1.30	1.29	1.31
1536	1.43	1.39	1.21	1.26	1.27	1.29	1.31	1.33
2048	1.37	1.36	1.18	1.24	1.29	1.29	1.31	1.30

Table 6. Relative timings for one long $\ell \times \ell \to 2\,\ell$ or one high $\ell \times \ell \to \ell$ limb $n \times n$ matrix product (on the left-hand and right-hand sides, respectively) using IFMA and the non-recursive variant of Karatsuba's algorithm.

6 Accelerating matrix multiplication using AMX

6.1 Implementation strategy

Let $A \in \mathbb{N}_8^{r \times s}$, $B \in \mathbb{N}_8^{s \times c}$, and $C \in \mathbb{N}_{32}^{r \times c}$, where $r \leqslant 16$, $s \leqslant 64$, and $c \leqslant 16$ are such that s is a multiple of four. Then the AMX extension offers an instruction for computing the matrix FMA $C \mathrel{+}= A\,B$ in 16 cycles if $s = 64$ and somewhat faster if $s < 64$. In practice, the total number $r\,s\,c$ of scalar FMAs per cycle is highest when using the maximal allowed dimensions $r = 16$, $s = 64$, and $c = 16$, which is the main configuration used in our implementation.

For $(r, s, c) = (16, 64, 16)$, the matrices A, B, and C need to be loaded from memory and stored in dedicated *tile* registers. There are eight available tile registers of 1Kb each and there are dedicated strided load and store operations for tiles. In addition, the second multiplicand B is stored in a special VNNI format, which coincides neither with the row major nor column major formats.

Since the 1Kb size of tiles is very different from the 64 byte size of AVX512 registers, AMX and AVX512 instructions do not fit well together into the SLP framework of JIL. For this reason, we decided to start with a base C++ implementation of the *matrix FMA* (MFMA) $C \mathrel{+}= A\,B$ for arbitrary sizes r, s, c, using Intel instrinsics. In the future, we plan to extend JIL with special "promises" (supplementing the scenarios described in Section 5.1) for building code that gracefully mixes AMX and AVX512 instructions.

For now, the core of our work on AMX relies on an efficient implementation of MFMA for $8 \times 8 \to 32$ bit entries. On top of this, we implemented multiple precision matrix multiplication using various schemes as in (3). The encoding and decoding stages are mostly done using AVX512 instructions and our main concern will be to keep the cost of these stages reasonable. This is challenging due to the fact that the throughput of AVX512 instructions is roughly sixteen times lower than the throughout of AMX

instructions (for eight bit entries, we do $16 \times 64 \times 16$ FMAs per 16 cycles with AMX and 64 operations per cycle with AVX512). We again heavily rely on SLPs from JIL and a similar extra layer as described in Section 5.1 for loops.

6.2 Int8 matrix FMAs

Our 8-bit MFMA kernel follows a relatively standard design informed by the classic paper by Goto and van de Geij [27]; see also [64, Sec. 5.1] and the recommendations of the Intel Optimization Reference Manual [42, Chapter 20].

Like many implementations of general matrix multiplication on AMX units, it is based on a microkernel that multiplies a 2×1 tile block $A' \in \mathbb{N}_8^{32\times 64}$ from A with a 1×2 tile block $B' \in \mathbb{N}_8^{64\times 32}$ from B and accumulates the result in a 2×2 tile block $C' \in \mathbb{N}_{32}^{32\times 32}$ of C, using all eight tile registers. Assuming that this microkernel is run repeatedly with the four accumulation tiles fixed, it requires about four loads and four tile FMAs per iteration. While, in ideal circumstances, the architecture we target can sustain up to two tile loads from L1 cache (and one store) per tile FMA, the 2×2 pattern makes it easier to hide memory movements behind computations even when not all loads hit L1 cache.

The typical scenario we aim for is for A' to reside in L1 cache while B' resides in L2 cache, as it is still possible in that case to fully overlap the loads with the FMAs in the steady state. We access the B matrix using non-temporal load instructions so as not to needlessly evict data from A from L1 cache. In addition, we use the strided load instructions mentioned above to read the 16 rows making up a tile of A from their original, typically non-contiguous memory locations without first packing them together, and similarly for writing tiles back to C. Our experiments suggest that, for data aligned on a cache line and in the absence of cache conflicts, strided loads and stores are not significantly slower than contiguous ones (see however [50] for more on the limits of this strategy). We deal with odd numbers of tiles in either dimension using a variant of the same microkernel, and with incomplete A- or C-tiles using a temporary buffer and AVX512 masked loads and stores.

The microkernel is wrapped in loops over i, j, k (from outermost to innermost) that multiply a series of $2 \times t$ tile blocks of A each fitting in L1 cache by a single block of B that fits in L2 cache. On a single core, we can typically process blocks of about $128 \times 16 \times 64$ tiles in this fashion. The resulting macrokernel is itself wrapped in a cache-blocking kernel, using the same loop order, that repacks roughly L2-sized blocks of B into contiguous VNNI-encoded full tiles (padded with zeros if necessary) before processing them using the macrokernel. The encoder operates on blocks of up to 64×32 entries of B, corresponding to two adjacent tiles, using AVX512 instructions. Slightly surprisingly, we found this block format to perform better in our context than the alternative of 64×64 entries, even though the latter makes it possible to load full cache lines (rows of the 64×64 block, to be split among four tiles) at once. We speculate that this may be because the L1 cache can serve up to three 256-byte loads but only two 512-byte loads per cycle.

We also implemented some microkernels dedicated to the special cases where one of the dimensions of the product is less than or equal to one tile.

Remark 2. Endo, Ohshima and Nanri [19] recently proposed an alternative microkernel that reserves six tile registers for accumulating a 2×3 block of C and uses the remaining two to hold the current tiles of A and B. The tiles of B are loaded once per block and those of A are loaded twice. While this results in an fma/load ratio of only $6/7$, the idea is that two of the loads are virtually guaranteed to hit L1 cache, whereas, depending on the blocking strategy, all four of the tiles accessed by the 2×2 kernel may have been evicted by the time they are reused. While Endo *et al.* report reaching better performance with their 2×3 kernel than with the standard one, in our experiments, it performed worse except for $c = 48$. Endo *et al.*'s code does not appear to be publicly available, but a plausible explanation would be that the use of non-temporal tile loads makes their technique unnecessary.

6.3 Naive multiple precision multiplications

The most natural approach to build an algorithm for ℓ-limb matrix multiplication on top of the work from the previous subsection is to rewrite $n \times n$ input matrices as elements in $\mathbb{N}_8^{n\times n}[2^8]_\ell$. Then a long

n	AMX	N16	N24	N32	N48	N64	L16	L24	L32	L48	L64
64	1.73	7.18	5.67	4.56	3.95	3.36	6.64	5.14	4.11	3.51	2.89
96	2.91	5.66	5.29	4.88	4.37	4.52	5.61	5.22	4.84	4.60	4.41
128	2.45	4.09	3.70	3.52	3.32	3.17	3.88	3.71	3.52	3.43	3.38
192	1.85	2.88	2.57	2.68	2.67	2.73	2.74	2.55	2.62	2.92	2.95
256	1.40	2.35	2.40	2.52	2.57	2.47	2.44	2.76	2.67	2.54	2.43
384	1.02	2.30	2.19	2.09	1.93	1.86	2.41	2.19	2.07	1.90	1.83
512	1.27	2.19	1.96	1.88	1.75	1.67	2.21	1.97	1.87	1.74	1.64
768	1.13	1.74	1.61	1.52	1.44	1.41	1.72	1.57	1.51	1.45	2.63
1024	1.11	1.61	1.44	1.39	2.51	2.36	1.55	1.45	2.97	2.68	2.50
1536	1.30	1.59	2.82	2.72	2.55	2.40	1.64	3.13	2.91	2.53	2.39
2048	1.27	3.02	2.94	2.72	2.51	2.28	3.32	3.05	2.70	2.41	2.23
3072	1.29	2.90	2.65	2.46	2.24	2.10	2.99	2.61	2.47	2.21	2.03
4096	1.75	2.95	2.85	2.70	2.51	2.41	3.07	2.84	2.68	2.45	2.34

Table 7. Factors with respect to theoretical peak performance for multiplying two $n \times n$ matrices with p bit positive integer coefficients. The columns Np corresponds to a $p \times p \to p$ bit low product, whereas the columns Lp corresponds to a $p \times p \to 2p$ bit long product. The column AMX corresponds to the raw $8 \times 8 \to 32$ bit product from the Section 6.2.

multiplication boils down to ℓ^2 matrix FMAs and a short (low or high) multiplication to $\ell(\ell+1)/2$ matrix FMAs.

Table 7 shows the factors between the performance of our implementation and the theoretical peak performance. If one is satisfied with a factor of three or less, then our current implementation will do. (In Table 22 of the appendix, we can see that the naive AMX-based implementation slightly outperforms our IFMA-based one.) But it seems significantly more difficult to approach the theoretical peak performance as well as we did for IFMA-based methods.

The variance of the timings obtained with AMX is quite large, with frequent outliers that can significantly alter mean timings. For this reason, our tables show the median instead of the mean timings over a large number of runs.

The cost of encoding and decoding using AVX512 is far from negligible. It seems hard to entirely get rid of this cost, since the algorithms from the previous subsection benefit from the fact that most of the matrices have a natural memory layout. Nonetheless, by interleaving the AMX instructions appropriately with the encoding and decoding, we expect that the factors can be reduced significantly.

As n increases, the cost of decoding and encoding should *a priori* become smaller with respect to the cost of the inner matrix multiplications. This is indeed what we observe for small values $n^2\ell$ (say $n^2\ell \leqslant 2^{23}$). However, for larger values of $n^2\ell$, we start to suffer from cache misses. In the future, we plan to reduce these misses through a more careful interleaving of memory loads and stores with the actual computations. But again, part of the slow-down is probably unavoidable.

6.4 Chinese remaindering

The implementation of the Chinese remaindering approach from section 3.6 for AMX-accelerated kernels is work in progress. We are currently working on an implementation in which the transforms (1) and (2) are also done using AMX instructions, by replacing the left-hand rows by huge matrices with one row per object to be transformed. Unfortunately, this means that two of the three matrices are far from being square, and AMX instructions are less efficient for such matrices. In addition, for various interesting bit-sizes like $64 \times 64 \to 128$ or $128 \times 128 \to 256$, we cannot plainly profit from $16 \times 64 \times 16$ multiplications and we have to resort to smaller configurations, such as $16 \times 16 \times 16$ or $16 \times 32 \times 16$. For larger bit-sizes, we also run out of 8 bit primes. This forces us to resort to 16 bit primes, for which we pay a factor two (or to 14 bit primes with Karatsuba multiplication, which still leads to an overhead of $^{12}/_{7}$). In addition to the matrix multiplications, one also needs to do several modular reductions, overlapping sums, and permutations in memory using AVX512 instructions. Altogether, this makes the algorithms time consuming and tricky to implement. For now, we only completed the implementation of the direct transforms.

Table 8 contains some absolute timings. Comparing with Table 22, we observe that the required CRT for a 1024×1024 matrix product over $\mathbb{N}_{64}$ is about 16 times faster than the current naive product. Since

n	8×16	16×16	16×32	32×32	$\ell\times N$
32	4.41 µs	4.20 µs	7.13 µs	7.22 µs	
64	16.8 µs	16.6 µs	27.9 µs	26.7 µs	
128	67.7 µs	66.4 µs	0.11 ms	0.11 ms	
256	0.27 ms	0.27 ms	0.46 ms	0.46 ms	
512	1.12 ms	1.05 ms	2.06 ms	3.43 ms	
1024	4.55 ms	4.42 ms	8.21 ms	9.58 ms	
2048	20.2 ms	20.6 ms	54.5 ms	60.3 ms	
4096	0.13 s	0.14 ms	0.35 s	0.38 s	

Table 8. Absolute timing for n^2 direct CRT on 8 ℓ-bit numbers and N moduli, using the reduction to an $n^2\times\ell\times N$ matrix product.

we need to do two direct CRTs and one inverse CRT (typically of doubled cost) and four times less modular matrix multiplications, we expect to gain a factor two in this case. The threshold for CRTs is thus around $n\approx1024$.

BIBLIOGRAPHY

[1] A. Abdelfattah, J. Dongarra, M. Fasi, M. Mikaitis, and F. Tisseur. Analysis of floating-point matrix multiplication computed via integer arithmetic. 2026.

[2] A. Ahlbäck, J. van der Hoeven, and G. Lecerf. JIL: a high performance library for straight-line programs. https://sourcesup.renater.fr/projects/jil, 2025.

[3] Automatically Tuned Linear Algebra Software (ATLAS). https://math-atlas.sourceforge.net.

[4] J. Berthomieu, S. Graillat, D. Lesnoff, and T. Mary. Multiword matrix multiplication over large finite fields in floating-point arithmetic. https://hal.science/hal-04917201, 2026.

[5] D. Bini and V. Y. Pan. *Polynomial and matrix computations. Vol. 1.* Birkhäuser Boston Inc., Boston, MA, 1994. Fundamental algorithms.

[6] BLAS (Basic Linear Algebra Subprograms). https://netlib.org/blas.

[7] BLAS-like library instantiation software framework. https://github.com/flame/blis.

[8] J. Bradbury, N. Drucker, and M. Hillenbrand. NTT software optimization using an extended Harvey butterfly. *Cryptology ePrint Archive, Paper 2021/1396*, 2021. https://eprint.iacr.org/2021/1396.

[9] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic.* Cambridge University Press, 2010.

[10] P. Cawley. Apple AMX instruction set. https://github.com/corsix/amx, 2024.

[11] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.

[12] W. da Silva Pereira. Accelerating floating-point computations with Intel AMX. Technical Report NREL/TP-2C00-93622, National Renewable Energy Laboratory, 2025.

[13] J. Doliskani, P. Giorgi, R. Lebreton, and É. Schost. Simultaneous conversions with the Residue Number System using linear algebra. *Transactions on Mathematical Software*, 44(3), 2018. Article 27.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[15] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[16] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. Linbox: a generic library for exact linear algebra. In *First Internat. Congress Math. Software ICMS*, pages 40–50. Beijing, China, 2002.

[17] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*

[18] T. Edamatsu and D. Takahashi. Accelerating large integer multiplication using intel AVX-512IFMA. In *Algorithms and Architectures for Parallel Processing: 19th International Conference, ICA3PP 2019, Melbourne, Australia*, pages 60–74. Springer-Verlag, 2019.

[19] Y. Endo, S. Ohshima, and T. Nanri. Optimization of a GEMM implementation using Intel AMX. In *Proc. of the Supercomputing Asia and Intern. Conf. on High Performance Computing in Asia Pacific Region*, SCA/HPCAsia '26, pages 81–90. ACM, 2026.

[20] D. Filho, G. Brandão, and J. López. Fast polynomial multiplication using matrix multiplication accelerators with applications to NTRU on Apple M1/M3 SoCs. *IACR Communications in Cryptology*, page 0, 2024.

[21] P. Fortin, A. Fleury, F. Lemaire, and M. Monagan. High performance SIMD modular arithmetic for polynomial evaluation. *ArXiv:2004.11571*, 2020.

[22] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180. New York, NY, USA, 1999. ACM.

[23] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[24] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, New York, NY, USA, 3rd edition, 2013.

[25] E. Georganas, K. Banerjee, D. Kalamkar, S. Avancha, A. Venkat, M. Anderson, G. Henry, H. Pabst, and A. Heinecke. High-performance deep learning via a single building block. 2019.

[26] G. Gerganov et al. Ggml. https://github.com/ggml-org/ggml, 2026.

[27] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.

[28] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. http://www.swox.com/gmp, 1991.

[29] S. Gueron and V. Krasnov. Accelerating big integer arithmetic using intel IFMA extensions. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 32–38. 2016.

[30] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn. A family of high-performance matrix multiplication algorithms. In *Computational Science – ICCS 2001*, pages 51–60. Berlin, Heidelberg, 2001. Springer.

[31] G. Hanrot, V. Lefèvre, K. Ryde, and P. Zimmermann. MPFR, a C library for multiple-precision floating-point computations with exact rounding. http://www.mpfr.org, 2000.

[32] William Hart et al. FLINT: fast library for number theory. https://flintlib.org, 2010.

[33] D. Harvey and J. van der Hoeven. On the complexity of integer matrix multiplication. *JSC*, 89:1–8, 2018.

[34] D. Harvey and J. van der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, 193(2):563–617, 2021.

[35] D. Harvey and A. V. Sutherland. Computing Hasse–Witt matrices of hyperelliptic curves in average polynomial time. In *Algorithmic Number Theory – Eleventh International Symposium (ANTS XI)*, volume 17 of *London Mathematical Society Journal of Computation and Mathematics*, pages 257–273. Cambridge University Press, 2014.

[36] J. van der Hoeven. Relax, but don't be too lazy. *JSC*, 34:479–542, 2002.

[37] J. van der Hoeven. *The Jolly Writer. Your Guide to GNU TeXmacs*. Scypress, 2020.

[38] J. van der Hoeven and G. Lecerf. Faster FFTs in medium precision. In *22nd IEEE Symposium on Computer Arithmetic (ARITH)*, pages 75–82. June 2015.

[39] J. van der Hoeven and G. Lecerf. Implementing number theoretic transforms. Technical Report, HAL, 2024. https://hal.science/hal-04841449, accepted for publication in AAECC.

[40] J. van der Hoeven and G. Lecerf. Towards a library for straight-line programs. *AAECC*, 2026. https://doi.org/10.1007/s00200-025-00719-0.

[41] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix. *ACM Trans. Math. Softw.*, 43(1):5–1, 2016.

[42] Intel Corporation. Intel® 64 and IA-32 architectures optimization reference manual: volume 1. 2024.

[43] A. Karatsuba and J. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[44] G. H. Khachatrian, M. K. Kuregian, K. R. Ispiryan, and J. L. Massey. Fast multiplication of integers for public-key applications. In S. Vaudenay and A. M. Youssef, editors, *Selected Areas in Cryptography*, pages 245–254. Berlin, Heidelberg, 2001. Springer.

[45] D. Khaldi, Y. Luo, B. Yu, A. Sotkin, B. Morais, and M. Girkar. Extending LLVM IR for DPC++ matrix support: a case study with Intel® advanced matrix extensions (Intel® AMX). In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 20–26. 2021.

[46] B. Kuzma, I. Korostelev, J. P. L. de Carvalho, J. E. Moreira, C. Barton, G. Araujo, and J. N. Amaral. Fast matrix multiplication via compiler-only layered data reorganization and intrinsic lowering. *Software: Practice and Experience*, 53(9):1793–1814, 2023.

[47] Linear Algebra PACKage. https://www.netlib.org/lapack.

[48] Ch. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Software*, 5(3):308–323, 1979.

[49] H.-J. Lebbink. Hjlebbink/amx-matmul. 2024. https://github.com/HJLebbink/AMX-matmul.

[50] T. Li. Usstq/mm_amx. https://github.com/usstq/mm_amx, 2024.

[51] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Softw.*, 43(2):12–1, 2016.

[52] H. Martínez, A. Castelló, F. D. Igual, and E. S. Quintana-Ortí. The Cambrian explosion of mixed-precision matrix multiplication for quantized deep learning inference. *Future Generation Computer Systems*, page 108231, 2025.

[53] C. S. Mummidi, V. C. Ferreira, S. Srinivasan, and S. Kundu. Highly efficient self-checking matrix multiplication on tiled AMX accelerators. *ACM Transactions on Architecture and Code Optimization*, 21(2):1–22, 2024.

[54] Hiroyuki Ootomo, Katsuhisa Ozaki, and Rio Yokota. Dgemm on integer matrix multiplication unit. *The Intern. J. of High Performance Comp. Appl.*, 38(4):297–313, 2024.

[55] D. N. Parikh, R. A. van de Geijn, and G. M. Henry. Cascading gemm: high precision from low precision. 2023. https://arxiv.org/abs/2303.04353v2.

[56] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[57] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[58] Michael Scott. Missing a trick: Karatsuba variations. *Cryptography and Communications*, 10(1):5–15, 2018.

[59] V. Shoup. NTL: a library for doing number theory. 1996. www.shoup.net/ntl.

[60] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:352–356, 1969.

[61] The OpenBLAS developers. Openblas. http://www.openmathlib.org/OpenBLAS/, feb 2026.

[62] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics*, 4(2):714–716, 1963.

[63] Y. Uchino, K. Ozaki, and T. Imamura. High-performance and power-efficient emulation of matrix multiplication using INT8 matrix engines. In *Proc. of the SC '25 Workshops of the Intern. Conf. for High Performance Comp., Networking, Storage and Analysis*, pages 1824–1831. ACM, 2025.

[64] F. G. Van Zee and R. A. van de Geijn. BLIS: a framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.*, 41(3):14–1, 2015.

[65] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In *Proc. of the Intern. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 1–12. New York, NY, USA, 2013. ACM.

[66] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[67] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proc. of the 2024 Annual ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

A Absolute timings

As motivated in the introduction, all timings that we reported so far are relative to the theoretical peak performance. For completeness, this appendix reports on the corresponding absolute timings. We also added absolute timings for existing software.

A.1 Some reference timings for existing software

Tables 9 to 14 present some timings obtained with Flint [32], NTL [59], and Fflas [17], three well-known algebraic computation libraries. Flint and Fflas are configured to link with OpenBlas. (We thank Pascal Giorgi for his help with Fflas.)

Since modular matrix multiplication has often benefited from more optimization effort than plain integer matrix multiplication, we include both matrices with entries in $\mathbb{N}_p$ and in $\mathbb{Z}/2^p\mathbb{Z}$.

n	24	32	52	64	104	128	208	256	416	512	832	1024	p
32	64.3 µs	30.5 µs	30.4 µs	88.6 µs	87.8 µs	0.29 ms	0.56 ms	0.67 ms	1.28 ms	2.11 ms	3.11 ms	3.96 ms	
64	70.3 µs	0.21 ms	0.21 ms	0.64 ms	0.64 ms	1.50 ms	2.77 ms	3.23 ms	6.25 ms	7.56 ms	14.5 ms	18.9 ms	
96	0.34 ms	0.67 ms	0.67 ms	2.10 ms	2.06 ms	4.25 ms	7.34 ms	8.75 ms	15.9 ms	19.2 ms	36.2 ms	46.4 ms	
128	0.56 ms	1.51 ms	1.52 ms	4.84 ms	4.69 ms	8.67 ms	14.9 ms	17.6 ms	31.9 ms	40.2 ms	73.6 ms	95.1 ms	
192	2.10 ms	3.82 ms	3.74 ms	14.6 ms	15.8 ms	25.5 ms	42.5 ms	50.7 ms	92.1 ms	0.11 s	0.20 s	0.26 s	
256	4.35 ms	8.69 ms	8.44 ms	23.4 ms	29.9 ms	60.4 ms	93.0 ms	0.11 s	0.20 s	0.24 s	0.43 s	0.55 s	
384	15.9 ms	29.1 ms	27.2 ms	57.0 ms	82.9 ms	86.6 ms	0.17 s	0.18 s	0.34 s	0.38 s	0.70 s	0.85 s	
512	37.9 ms	68.6 ms	69.7 ms	0.16 s	0.20 s	0.19 s	0.33 s	0.32 s	0.59 s	0.67 s	1.26 s	1.54 s	
768	0.10 s	0.20 s	0.20 s	0.39 s	0.39 s	0.42 s	0.73 s	0.80 s	1.21 s	1.39 s	2.45 s	3.03 s	
1024	0.27 s	0.44 s	0.46 s	0.65 s	0.70 s	0.78 s	1.30 s	1.37 s	2.22 s	2.64 s	4.87 s	5.95 s	
2048	0.92 s	1.44 s	1.59 s	1.98 s	2.32 s	2.68 s	4.12 s	4.83 s	7.64 s	9.50 s	18.6 s	25.0 s	

Table 9. Absolute timings for multiplication of two matrices in $\mathbb{N}_p^{n\times n}$ using the `fmpz_mat_mul` function from Flint.

n	24	32	52	64	104	128	208	256	416	512	832	p
32	6.84 µs	6.82 µs	8.95 µs	77.2 µs	77.2 µs	0.23 ms	0.25 ms	0.35 ms	0.59 ms	0.78 ms	1.72 ms	
64	32.9 µs	32.9 µs	54.6 µs	0.48 ms	0.48 ms	1.34 ms	2.27 ms	2.71 ms	4.57 ms	6.39 ms	12.8 ms	
96	0.10 ms	0.10 ms	0.16 ms	1.71 ms	1.48 ms	3.60 ms	6.00 ms	6.96 ms	12.8 ms	16.7 ms	32.1 ms	
128	0.26 ms	0.26 ms	0.43 ms	3.75 ms	3.35 ms	7.58 ms	12.8 ms	14.7 ms	26.7 ms	34.6 ms	65.9 ms	
192	0.76 ms	0.76 ms	1.27 ms	10.7 ms	10.7 ms	21.2 ms	35.6 ms	41.0 ms	74.2 ms	94.6 ms	0.18 s	
256	1.77 ms	1.77 ms	3.41 ms	23.3 ms	31.9 ms	44.6 ms	75.3 ms	86.3 ms	0.16 s	0.20 s	0.36 s	
384	5.51 ms	5.48 ms	10.7 ms	70.5 ms	95.6 ms	0.13 s	0.22 s	0.25 s	0.46 s	0.58 s	1.03 s	
512	11.7 ms	15.3 ms	23.6 ms	95.0 ms	0.13 s	0.19 s	0.32 s	0.36 s	0.61 s	0.76 s	1.36 s	
768	45.7 ms	51.8 ms	0.10 s	0.34 s	0.43 s	0.56 s	0.86 s	0.97 s	1.70 s	2.03 s	3.58 s	
1024	70.4 ms	0.11 s	0.17 s	0.59 s	0.77 s	1.10 s	1.59 s	1.80 s	3.10 s	3.88 s	6.83 s	
2048	0.27 s	0.33 s	0.58 s	2.16 s	2.63 s	3.74 s	5.54 s	6.52 s	12.5 s	14.5 s	24.4 s	

Table 10. Absolute timings for multiplication of two matrices in $(\mathbb{Z}/2^p\mathbb{Z})^{n\times n}$ using the `nmod_mat_mul` (for $p<64$) or `mpn_mod_mat_mul` (otherwise) function from Flint.

n	24	32	52	64	104	128	208	256	416	512	832	1024	p
32	0.32 ms	0.45 ms	0.32 ms	0.45 ms	0.48 ms	0.61 ms	0.78 ms	0.91 ms	1.58 ms	1.72 ms	3.26 ms	4.80 ms	
64	2.40 ms	3.72 ms	2.49 ms	3.77 ms	4.01 ms	5.04 ms	7.41 ms	7.98 ms	13.4 ms	14.6 ms	25.5 ms	37.8 ms	
96	8.49 ms	12.4 ms	8.50 ms	13.7 ms	13.5 ms	19.1 ms	24.5 ms	34.3 ms	54.8 ms	73.8 ms	85.2 ms	0.13 s	
128	22.9 ms	33.5 ms	23.6 ms	40.5 ms	39.4 ms	49.1 ms	0.10 s	95.3 ms	0.16 s	0.17 s	0.21 s	0.30 s	
192	0.13 s	0.19 s	0.13 s	0.18 s	0.19 s	0.23 s	0.30 s	0.33 s	0.44 s	0.40 s	0.72 s	1.08 s	
256	0.32 s	0.45 s	0.33 s	0.45 s	0.49 s	0.57 s	0.63 s	0.69 s	0.83 s	1.27 s	1.93 s	2.62 s	
384	1.18 s	1.61 s	1.21 s	1.60 s	1.72 s	2.00 s	2.23 s	2.49 s	3.60 s	4.53 s	7.01 s	9.30 s	

Table 11. Absolute timings for multiplication of two matrices in $\mathbb{N}_p^{n\times n}$ represented as NTL objects of type `mat_ZZ`.

n	24	32	52	64	104	128	208	256	416	512	832	1024	p
32	0.11 ms	0.11 ms	0.14 ms	0.17 ms	0.24 ms	0.29 ms	0.43 ms	0.51 ms	0.86 ms	1.11 ms	2.08 ms	2.62 ms	
64	0.38 ms	0.38 ms	0.50 ms	0.60 ms	0.87 ms	1.05 ms	1.60 ms	1.89 ms	3.29 ms	4.19 ms	7.86 ms	10.1 ms	
96	0.83 ms	0.84 ms	1.11 ms	1.33 ms	1.95 ms	2.35 ms	3.64 ms	4.23 ms	7.42 ms	9.34 ms	17.6 ms	22.7 ms	
128	1.52 ms	1.51 ms	2.03 ms	2.43 ms	3.59 ms	4.20 ms	6.50 ms	7.60 ms	13.2 ms	16.8 ms	31.6 ms	40.4 ms	
192	3.66 ms	3.64 ms	4.81 ms	5.75 ms	8.43 ms	10.0 ms	15.5 ms	18.0 ms	31.0 ms	40.7 ms	78.7 ms	0.10 s	
256	6.86 ms	6.78 ms	9.17 ms	10.9 ms	16.1 ms	19.5 ms	29.9 ms	35.1 ms	60.2 ms	78.7 ms	0.16 s	0.21 s	
384	17.4 ms	17.2 ms	23.6 ms	28.3 ms	42.7 ms	51.1 ms	79.6 ms	94.1 ms	0.18 s	0.24 s	0.44 s	0.56 s	
512	35.3 ms	34.0 ms	47.3 ms	56.6 ms	85.8 ms	0.11 s	0.17 s	0.21 s	0.35 s	0.45 s	0.83 s	1.04 s	
768	0.11 s	0.11 s	0.15 s	0.17 s	0.26 s	0.31 s	0.46 s	0.55 s	0.92 s	1.15 s	2.07 s	2.64 s	
1024	0.23 s	0.23 s	0.31 s	0.37 s	0.55 s	0.65 s	0.99 s	1.17 s	1.93 s	2.42 s	4.30 s	5.46 s	
2048	1.39 s	1.39 s	2.00 s	2.34 s	3.59 s	4.23 s	6.43 s	7.70 s	12.5 s	15.6 s	25.9 s	32.4 s	

Table 12. Absolute timings for multiplication of two matrices in $(\mathbb{Z}/2^p\mathbb{Z})^{n\times n}$ represented as NTL objects of type `mat_ZZ_p`.

n	24	32	52	p
32	4.89 µs	13.8 µs	13.9 µs	
64	29.1 µs	0.11 ms	0.11 ms	
96	88.5 µs	0.35 ms	0.35 ms	
128	0.20 ms	0.83 ms	0.83 ms	
192	0.63 ms	2.78 ms	2.78 ms	
256	1.48 ms	6.59 ms	6.59 ms	
384	4.87 ms	22.3 ms	22.3 ms	
512	11.7 ms	47.7 ms	47.6 ms	
768	37.4 ms	0.16 s	0.16 s	
1024	88.5 ms	0.34 s	0.34 s	
2048	0.66 s	2.42 s	2.42 s	

Table 13. Absolute timings for multiplication of two matrices in $(\mathbb{Z}/2^p\mathbb{Z})^{n\times n}$ represented as NTL objects of type `mat_zz_p` (word-sized p).

n	24	32	52	64	104	128	208	256	416	512	832	1024	p
32	1.77 ms	1.80 ms	1.90 ms	1.94 ms	2.12 ms	2.21 ms	2.55 ms	2.76 ms	3.55 ms	4.08 ms	5.65 ms	6.70 ms	
64	2.59 ms	2.66 ms	3.12 ms	3.25 ms	3.85 ms	4.35 ms	5.74 ms	6.63 ms	9.40 ms	10.6 ms	16.3 ms	19.7 ms	
96	4.13 ms	4.30 ms	5.49 ms	5.69 ms	7.59 ms	8.14 ms	10.9 ms	13.4 ms	18.6 ms	22.1 ms	35.8 ms	43.6 ms	
128	5.80 ms	6.26 ms	7.49 ms	8.41 ms	11.1 ms	12.2 ms	17.8 ms	21.4 ms	31.9 ms	39.7 ms	64.1 ms	83.2 ms	
192	10.4 ms	12.2 ms	15.1 ms	17.3 ms	23.6 ms	26.4 ms	39.4 ms	51.0 ms	81.4 ms	0.10 s	0.17 s	0.22 s	
256	18.0 ms	21.2 ms	26.8 ms	30.9 ms	43.4 ms	47.9 ms	76.6 ms	93.1 ms	0.17 s	0.22 s	0.36 s	0.47 s	
384	42.2 ms	53.2 ms	67.0 ms	73.5 ms	0.12 s	0.13 s	0.21 s	0.27 s	0.42 s	0.54 s	0.87 s	1.09 s	
512	78.8 ms	94.5 ms	0.13 s	0.17 s	0.24 s	0.26 s	0.41 s	0.52 s	0.84 s	1.02 s	1.67 s	2.06 s	
768	0.20 s	0.23 s	0.33 s	0.40 s	0.59 s	0.70 s	1.05 s	1.26 s	2.03 s	2.46 s	4.04 s	5.01 s	
1024	0.48 s	0.48 s	0.67 s	0.84 s	1.23 s	1.40 s	2.12 s	2.58 s	4.16 s	5.00 s	8.28 s	10.2 s	
2048	2.46 s	2.93 s	4.01 s	4.47 s	6.58 s	8.05 s	12.2 s	15.1 s	23.7 s	29.1 s	46.5 s	59.0 s	

Table 14. Absolute timings for multiplication of two matrices $\mathbb{N}_p^{n\times n}$ using `FFLAS::fgemm` over `Givaro::ZRing<Givaro::Integer>`.

A.2 Multiplying small matrices using IFMA

Tables 15, 16, and 17 are the absolute counterparts of Tables 2, 3, and 4 from Section 4. Note that Table 15 corresponds to the left-hand of Table 2 (for multiplicands in $(\mathbb{N}^8_{52\ell})^{n\times n}$), whereas Table 16 corresponds to the right-hand of Table 3 (for multiplicands that use the alternative representation in $\mathbb{N}_{52}^{n\times n}[2^{52}]_\ell$).

n	1	2	3	4	6	8	12	16	ℓ
1	1.86 ns	2.76 ns	5.37 ns	8.48 ns	16.7 ns	26.8 ns	57.9 ns	0.14 µs	
2	4.07 ns	15.6 ns	32.7 ns	60.2 ns	0.14 µs	0.22 µs	0.51 µs	1.50 µs	
3	11.9 ns	61.5 ns	0.13 µs	0.22 µs	0.46 µs	1.03 µs	2.49 µs	5.53 µs	
4	32.8 ns	0.15 µs	0.32 µs	0.53 µs	1.43 µs	2.30 µs	6.32 µs	13.5 µs	
6	91.1 ns	0.53 µs	1.50 µs	2.51 µs	5.19 µs	8.19 µs	20.6 µs	88.9 µs	
8	0.25 µs	1.81 µs	3.94 µs	6.38 µs	11.8 µs	18.7 µs	0.10 ms	0.36 ms	
12	0.93 µs	5.99 µs	12.6 µs	20.1 µs	75.7 µs	0.18 ms	0.56 ms	1.24 ms	
16	2.77 µs	15.1 µs	40.7 µs	0.11 ms	0.31 ms	0.51 ms	1.36 ms	3.27 ms	

Table 15. Absolute timings for pure SIMD multiplication of two matrices in $(\mathbb{N}^8_{52\ell})^{n\times n}$.

n	1	2	3	4	6	8	12	16	ℓ
2	2.16 ns	4.63 ns	8.94 ns	13.1 ns	34.3 ns	46.2 ns	85.3 ns	0.15 µs	
4	6.03 ns	19.6 ns	42.7 ns	68.6 ns	0.14 µs	0.22 µs	0.46 µs	1.27 µs	
6	21.9 ns	72.8 ns	0.18 µs	0.29 µs	0.94 µs	1.53 µs	3.62 µs	5.83 µs	
8	35.9 ns	0.13 µs	0.29 µs	0.47 µs	1.69 µs	2.94 µs	6.59 µs	11.3 µs	
12	0.10 µs	0.40 µs	1.35 µs	2.34 µs	4.92 µs	8.34 µs	18.4 µs	48.1 µs	
16	0.23 µs	1.45 µs	3.16 µs	5.35 µs	11.4 µs	19.4 µs	85.0 µs	0.23 ms	
24	0.87 µs	4.38 µs	9.70 µs	18.6 µs	65.1 µs	0.17 ms	0.44 ms	0.81 ms	
32	3.16 µs	15.5 µs	41.4 µs	0.12 ms	0.38 ms	0.67 ms	1.49 ms	2.62 ms	

Table 16. Absolute timings for tinkered SIMD multiplication of two matrices in $\mathbb{N}_{52}^{n\times n}[2^{52}]_\ell$.

n	1	2	3	4	6	8	12	16	ℓ
2		2.49 ns		6.88 ns	15.7 ns	26.9 ns	45.1 ns	63.9 ns	
4	3.47 ns	10.8 ns	21.4 ns	33.0 ns	75.1 ns	0.13 µs	0.25 µs	0.41 µs	
6		34.7 ns		0.11 µs	0.26 µs	0.40 µs	1.26 µs	1.92 µs	
8	20.7 ns	71.0 ns	0.15 µs	0.24 µs	0.51 µs	0.94 µs	2.11 µs	3.46 µs	
12	55.8 ns	0.19 µs	0.47 µs	1.07 µs	2.28 µs	3.67 µs	7.88 µs	13.1 µs	
16	0.12 µs	0.44 µs	1.33 µs	2.25 µs	4.88 µs	7.98 µs	17.2 µs	43.6 µs	
24	0.35 µs	1.72 µs	4.64 µs	7.84 µs	16.6 µs	31.5 µs	0.17 ms	0.35 ms	
32	1.02 µs	5.61 µs	12.1 µs	19.7 µs	55.0 µs	0.18 ms	0.48 ms	0.82 ms	

Table 17. Absolute timings for high multiplication of matrices in $(\mathbb{N}_{52\,\ell}^{8})^{n\times n}$ in tinkered SIMD mode.

A.3 Multiplying large matrices using IFMA

Tables 18 and 19 present the absolute counterparts of the timings in Table 5 from Section 5 (the left-hand and right-hand tables respectively). Similarly, Tables 20 and 21 provide absolute counterparts for the timings in Table 6.

n	1	2	3	4	6	8	12	16	ℓ
32	8.37 µs	19.8 µs	39.3 µs	64.7 µs	0.14 ms	0.24 ms	0.54 ms	0.95 ms	
64	39.6 µs	99.4 µs	0.21 ms	0.35 ms	0.78 ms	1.40 ms	3.15 ms	5.47 ms	
96	99.8 µs	0.28 ms	0.61 ms	1.04 ms	2.29 ms	4.04 ms	9.09 ms	16.1 ms	
128	0.21 ms	0.64 ms	1.33 ms	2.31 ms	5.12 ms	8.93 ms	20.3 ms	36.1 ms	
192	0.62 ms	2.01 ms	4.31 ms	7.28 ms	16.3 ms	29.2 ms	68.5 ms	0.12 s	
256	1.35 ms	4.55 ms	9.90 ms	17.1 ms	38.9 ms	68.9 ms	0.17 s	0.30 s	
384	4.10 ms	14.1 ms	31.4 ms	55.7 ms	0.14 s	0.24 s	0.54 s	0.93 s	
512	9.33 ms	34.3 ms	87.5 ms	0.15 s	0.33 s	0.58 s	1.29 s	2.24 s	
768	30.1 ms	0.14 s	0.29 s	0.50 s	1.07 s	1.88 s	4.21 s	7.50 s	
1024	91.6 ms	0.31 s	0.66 s	1.15 s	2.55 s	4.72 s	10.4 s	18.3 s	
1536	0.32 s	1.16 s	2.51 s	4.33 s	9.46 s	16.6 s	37.4 s	64.3 s	
2048	0.71 s	2.67 s	5.81 s	10.1 s	23.1 s	40.8 s	91.7 s	161. s	

Table 18. Absolute timings for a naive long $\ell \times \ell \to 2\,\ell$ limb $n \times n$ matrix product using IFMA.

n	1	2	3	4	6	8	12	16	ℓ
32	10.5 µs	21.0 µs	31.9 µs	43.7 µs	70.7 µs	0.11 ms	0.21 ms	0.34 ms	
64	52.7 µs	0.10 ms	0.16 ms	0.22 ms	0.37 ms	0.56 ms	1.07 ms	1.72 ms	
96	0.15 ms	0.30 ms	0.48 ms	0.64 ms	1.03 ms	1.51 ms	2.68 ms	4.23 ms	
128	0.32 ms	0.67 ms	1.02 ms	1.38 ms	2.19 ms	3.16 ms	5.47 ms	8.37 ms	
192	1.02 ms	1.99 ms	3.03 ms	4.12 ms	6.40 ms	9.09 ms	15.2 ms	34.6 ms	
256	2.32 ms	4.56 ms	6.95 ms	9.31 ms	14.5 ms	30.8 ms	51.4 ms	78.3 ms	
384	7.61 ms	14.9 ms	22.4 ms	42.8 ms	68.3 ms	95.6 ms	0.15 s	0.22 s	
512	18.0 ms	44.7 ms	70.0 ms	95.8 ms	0.15 s	0.20 s	0.32 s	0.45 s	
768	70.3 ms	0.15 s	0.22 s	0.31 s	0.47 s	0.64 s	1.00 s	1.38 s	
1024	0.17 s	0.34 s	0.51 s	0.68 s	1.03 s	1.40 s	2.15 s	2.96 s	
1536	0.61 s	1.23 s	1.84 s	2.45 s	3.68 s	4.91 s	7.60 s	10.3 s	
2048	1.38 s	2.73 s	4.08 s	5.48 s	8.21 s	11.1 s	16.7 s	22.4 s	

Table 19. Absolute timings for a long $\ell \times \ell \to 2\,\ell$ limb $n \times n$ matrix product using IFMA and CRT.

n	1	2	3	4	6	8	12	16	ℓ
32	4.98µs	13.1µs	26.8µs	45.2µs	95.2µs	0.17ms	0.42ms	0.81ms	
64	24.3µs	68.7µs	0.14ms	0.25ms	0.58ms	1.00ms	2.24ms	8.88ms	
96	66.9µs	0.20ms	0.44ms	0.78ms	1.62ms	2.76ms	12.0ms	22.1ms	
128	0.15ms	0.48ms	0.98ms	1.64ms	3.32ms	10.7ms	23.8ms	48.2ms	
192	0.50ms	1.49ms	2.94ms	4.80ms	16.5ms	29.4ms	73.0ms	0.14 s	
256	1.15ms	3.38ms	6.63ms	16.4ms	37.0ms	68.7ms	0.16 s	0.29 s	
384	3.62ms	10.8ms	28.3ms	50.0ms	0.11 s	0.20 s	0.44 s	0.80 s	
512	8.90ms	32.3ms	67.1ms	0.12 s	0.25 s	0.43 s	0.96 s	1.71 s	
768	33.6ms	0.11 s	0.21 s	0.36 s	0.76 s	1.30 s	2.83 s	5.00 s	
1024	79.0ms	0.24 s	0.49 s	0.81 s	1.72 s	2.93 s	6.38 s	11.3 s	
1536	0.29 s	0.85 s	1.73 s	2.87 s	5.99 s	10.3 s	22.3 s	38.8 s	
2048	0.66 s	1.99 s	3.99 s	6.67 s	13.9 s	24.0 s	51.9 s	91.8 s	

Table 20. Absolute timings for a long $\ell \times \ell \to 2\,\ell$ limb $n \times n$ matrix product using IFMA and the non-recursive variant of Karatsuba's algorithm.

n	1	2	3	4	6	8	12	16	ℓ
32	3.81µs	10.6µs	17.0µs	28.0µs	57.2µs	96.9µs	0.22ms	0.40ms	
64	15.8µs	50.8µs	84.3µs	0.14ms	0.33ms	0.57ms	1.21ms	2.10ms	
96	36.2µs	0.14ms	0.25ms	0.45ms	0.97ms	1.63ms	3.41ms	12.0ms	
128	80.4µs	0.34ms	0.59ms	1.00ms	2.07ms	3.44ms	13.7ms	24.0ms	
192	0.27ms	1.03ms	1.79ms	3.00ms	6.11ms	17.5ms	39.2ms	72.3ms	
256	0.60ms	2.34ms	4.05ms	6.80ms	21.8ms	39.1ms	87.4ms	0.15 s	
384	1.87ms	7.27ms	18.1ms	31.7ms	70.3ms	0.12 s	0.25 s	0.44 s	
512	4.54ms	23.2ms	41.1ms	71.9ms	0.15 s	0.26 s	0.54 s	0.93 s	
768	15.6ms	77.8ms	0.14 s	0.23 s	0.48 s	0.81 s	1.69 s	2.90 s	
1024	43.0ms	0.18 s	0.30 s	0.51 s	1.06 s	1.80 s	3.73 s	6.43 s	
1536	0.16 s	0.63 s	1.10 s	1.87 s	3.86 s	6.43 s	13.4 s	23.6 s	
2048	0.33 s	1.38 s	2.46 s	4.48 s	8.97 s	15.2 s	29.9 s	50.3 s	

Table 21. Absolute timings for a high $\ell \times \ell \to \ell$ limb $n \times n$ matrix product using IFMA and the non-recursive variant of Karatsuba's algorithm.

A.4 Multiplying matrices using AMX

Table 22 shows absolute timings for naive AMX-accelerated matrix multiplication at various precisions; the corresponding relative timings were given in Table 7.

n	AMX	N16	N24	N32	N48	N64	L16	L24	L32	L48	L64
64	0.20µs	2.51µs	3.97µs	5.59µs	9.70µs	14.5µs	3.19µs	5.64µs	8.37µs	16.0µs	26.0µs
96	1.22µs	7.19µs	13.0µs	20.3µs	39.8µs	65.7µs	9.21µs	18.8µs	31.2µs	66.7µs	0.12ms
128	2.28µs	12.4µs	22.1µs	34.3µs	67.0µs	0.11ms	15.0µs	30.4µs	52.3µs	0.12ms	0.19ms
192	5.21µs	28.0µs	53.0µs	92.8µs	0.18ms	0.30ms	32.0µs	80.7µs	0.13ms	0.32ms	0.57ms
256	11.8µs	62.4µs	0.11ms	0.19ms	0.41ms	0.67ms	76.5µs	0.18ms	0.33ms	0.72ms	1.16ms
384	29.9µs	0.18ms	0.36ms	0.57ms	1.11ms	1.83ms	0.27ms	0.55ms	0.92ms	1.89ms	3.18ms
512	86.3µs	0.44ms	0.78ms	1.22ms	2.37ms	3.86ms	0.59ms	1.16ms	1.93ms	3.95ms	6.39ms
768	0.23ms	1.04ms	1.93ms	3.08ms	6.22ms	10.3ms	1.37ms	2.83ms	4.81ms	10.5ms	30.8ms
1024	0.53ms	2.22ms	4.20ms	6.91ms	24.2ms	38.7ms	2.89ms	6.20ms	20.9ms	42.7ms	71.3ms
1536	1.99ms	7.53ms	27.1ms	42.7ms	84.2ms	0.13 s	10.2ms	44.0ms	74.4ms	0.14 s	0.24 s
2048	4.77ms	33.6ms	64.0ms	0.10 s	0.20 s	0.30 s	49.1ms	0.10 s	0.16 s	0.32 s	0.54 s
3072	16.4ms	0.11 s	0.20 s	0.33 s	0.61 s	0.96 s	0.15 s	0.30 s	0.51 s	1.03 s	1.69 s
4096	51.1ms	0.26 s	0.51 s	0.80 s	1.58 s	2.57 s	0.37 s	0.77 s	1.29 s	2.69 s	4.58 s

Table 22. Time to multiply two $n \times n$ matrices with p bit positive integer coefficients using naive AMX-accelerated arithmetic. The columns Np corresponds to a $p \times p \to p$ bit low product, whereas the columns Lp corresponds to a $p \times p \to 2\,p$ bit long product. The column AMX corresponds to the raw $8 \times 8 \to 32$ bit product from Section 6.2.