

Multi-limb integer matrix multiplication using AMX and AVX512-IFMA instructions^{*†}

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

Preliminary version of September 25, 2026

We study the hardware-accelerated multiplication of matrices with integer entries of 8 to about 800 bits. Our focus is on two relatively new hardware features in Intel CPUs: the IFMA “integer FMA” instruction and the AMX matrix extensions. We survey a number of approaches to the multiplication of matrices of multiple-precision integers using these instruction sets and analyze to what extent our implementations on top of the JIL library 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. While this trend is driven by applications in artificial intelligence, is there a potential for tensor cores to be used for other purposes? After all, GPUs have found applications well 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. A basic question is that of matrix multiplication with p -bit integer entries. Tensor cores are typically designed to accelerate such computations when the matrix dimensions are large and $p = 8$. Can this be used to efficiently deal with other bit precisions, ranging from small ones like $p = 16, 24, 32, 48, 64$ to larger ones like $p = 128, 256, 512, 1024, \dots$? We focus here on the case 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]. 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 compare implementations that exploit tensor cores with alternatives based on more conventional vector arithmetic. For this reason, we 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) $C += A \cdot B$ 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, INTEL's AVX512 integer FMA (IFMA) can compute $z += x \cdot y$ in one or half a 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 (general matrix multiplication, $C := \alpha AB + \beta C$) because AMX only supports positive accumulation.

Recent AMD processors also support AVX-512 and the IFMA instruction, and part of our work also applies to these processors. However, AMD processors do not yet support the AMX instruction set. Current ARM processors only support 128-bit-wide NEON vector arithmetic (although approximately twice as efficient as on X86 architectures) and there is no equivalent of the IFMA instruction. The matrix coprocessors integrated in APPLE's M1–M5 CPUs are based on fast pipelined outer products [10], not dot products like AMX-style matrix multiplication, so that adapting our work to such processors would require significant changes.

^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 T_EX_{MACS} [38].

In most of the article, we focus on the performance on a single CPU core. This simplification allows us to compare implementation strategies that exploit different hardware features without being distracted by largely orthogonal considerations on how to scale up these base implementations on multi-core architectures. On request of one of the referees, though, we briefly present very preliminary work on a multi-threaded implementation at the end of the paper.

The timings presented in the main body of the article are relative to theoretical peak performance. For instance, for a naive implementation of $n \times n$ matrix multiplication using $2n^3$ IFMAs, we give the measured cost in cycles divided by the theoretical cost of $2n^3$ IFMAs based on published instruction throughputs. These ratios are fairly independent of the particular machine 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 indicate potential problems such as suboptimal memory access patterns, asymptotically subdominant costs that become high for certain sizes, or insurmountable hardware obstacles. 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 provide 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, 41] for computations with straight-line programs (SLPs). This work is the first step of a project to equip 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. Conversely, matrix multiplication is an excellent problem for extending JIL beyond the mere compilation of SLPs: another important motivation is to use it as a case study for how to integrate SLPs inside larger pieces of code such as multiple loops. Integrating 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 [44], Toom's method [65], and modular methods [25, Chapter 5]. For high precisions, there are also methods based on the fast Fourier transform [11, 59, 35], but we will not use them in this paper. GMP [29] and MPFR [32] are the reference libraries for multiple precision integer and floating point arithmetic. We also refer to [39, 30, 19, 8, 40] 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 already at 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 (ℓp) -bit entries is reduced to $E(\ell)$ multiplications of $n \times n$ matrices with p -bit entries, where the quantity $E(\ell)$ 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 $E(\ell)$ 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 best to pick the method for which $E(\ell)$ is minimal. Good practical implementations should reflect this fact and this is indeed what our work aims at.

Theoretically speaking, $n \times n$ matrices can be multiplied in time $O(n^\omega)$ with $\omega \leq 2.38$ [12] for large n . Among the algorithms with $\omega < 3$, only Strassen's algorithm [63] with $\omega = \log_2 7$ has allowed for some modest practical gains so far. However, due to the high arithmetic intensity of matrix multiplication and now to the availability of dedicated hardware accelerators, the practical complexity of matrix multiplication *behaves* as if ω were well below 3, making matrix multiplication a crucial building block for other fast algorithms, both in theory and in practice.

Classical reference implementations for matrix multiplication and other routines from linear algebra include BLAS [6], LAPACK [48], ATLAS [3], BLIS [7] and OPENBLAS [64]. We refer to [49, 16, 15, 31, 69, 28, 67, 52] 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 [17, 14]. When working with very high precisions, FFT-based techniques are also used [36, 34]. The FLINT library [33] performs well on a large spectrum of matrix dimensions and integer sizes. We refer to [42, 22] for a discussion of SIMD-accelerated modular arithmetic and to [4, 66] for CRT-based matrix multiplication on modern hardware. Doubling the precision of a BLAS has been considered in [56].

Several works [13, 20] and research implementations [51, 50] specifically address the challenges raised by Intel's AMX units. OPENBLAS [64] also contains a bf16 kernel (`sbgemv_kernel_16x16_spr_tmp1.c`) using AMX instructions, and GGML [27] an int8 kernel interleaved with encoding and decoding of quantized tensors (`ggml-cpu/amx/mmq.cpp`). Separate techniques have been developed for other tensor accelerator architectures, including the matrix coprocessors of Apple M1–M3 CPUs [21]. We refer to [53] for an overview of the types of matrix multiplication accelerators available in current CPUs and to [43, 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 [55, 54, 66, 1]. Georganas *et al.* [26] propose a batch-reduce kernel that may also be useful for this purpose.

Finally, our work on JIL draws inspiration from the compilation of codelets for critical tasks pioneered in FFTW3 [23, 24] and SPIRAL [57]. Matrix multiplication has also been studied from a similar compilation-oriented perspective [68, 46, 47].

Contributions

We present an experimental high performance 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. Our implementation is available at <https://sourcesup.renater.fr/projects/jil/> on the `amx` branch and distributed under the GNU GPLv3. The version described in the present paper corresponds to revision `a04ea451`, based on JIL revision `999ee20d`.

In Table 1, we present 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 IFMA-based kernels, and to a lesser extent the AMX-based ones, often come close to the theoretical peak performance for which the idealized model is most relevant. In absolute terms, we observe that AMX-based algorithms outperform the IFMA-based ones for matrices of dimensions about 128 and higher. For many matrix dimensions and bit precisions, our sequential implementations outperform the current reference libraries FLINT and NTL.

Our preliminary multi-threaded implementation scales well up to approximately 32 threads for the IFMA-based algorithms. For 64 threads, it might be an issue that our computer is equipped with two physical CPUs. The multi-threaded versions of our AMX-based algorithms often lose a factor two with respect to the theoretically maximal parallel speed-up. It seems that the performance is penalized by memory operations, and further work is needed to better approach the theoretical peak performance.

Note on AI usage

For the present revised version of the paper, we improved and completed the AMX-accelerated CRT-based matrix multiplication kernel. Claude Opus 5 was used as an assistant for this work, as detailed in Section 6.4.

2 Preliminaries

2.1 Notations

Throughout this paper, we write $\mathbb{N} := \{0, 1, 2, \dots\}$ for the set of nonnegative integers, and $\mathbb{N}_p = \{0, \dots, 2^p - 1\}$ 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 $(a_0, \dots, a_{n-1})$ 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 $a_0 + \dots + a_{n-1}x^{n-1}$ by their vectors $(a_0, \dots, a_{n-1})$ 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” $i = i_0 + i_1 B + \dots + i_{\ell-1} B^{\ell-1} \in \mathbb{N}_\mu[2^\mu]_\ell$ in the radix. Following terminology from GMP [29], it is customary to call the coefficients $i_0, \dots, i_{\ell-1} \in \mathbb{N}_\mu$ *limbs*. If $\mathbb{N}_\mu$ is a machine integer type, then we may represent an integer $i \in \mathbb{N}_{\ell\mu}$ by its vector $(i_0, \dots, i_{\ell-1})$ 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, as vectors $(a_{0,0}, \dots, a_{0,c-1}, \dots, a_{r-1,0}, \dots, a_{r-1,c-1})$. 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 $(\mathbb{N}_\mu^w)^{r \times c}$. 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 $a = a_0 + a_1 2^\mu + \dots + a_{\ell-1} 2^{(\ell-1)\mu} \in \mathbb{N}_\mu^w[2^\mu]_\ell$ with $a_0, \dots, a_{\ell-1} \in \mathbb{N}_\mu^w$ and $B = 2^\mu$. Note that this may not seem to be the most natural choice since one might prefer to represent $a \in \mathbb{N}_{\ell\mu}^w$ as $(a_0, \dots, a_{w-1})$ with $a_0, \dots, a_{w-1} \in \mathbb{N}_{\ell\mu}$.

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

All experiments were conducted on a system equipped with two INTEL XEON GOLD 6538Y+ (Emerald Rapids) processors (each with 32 cores running at a base frequency of 2.2 GHz and maximum frequency of 4GHz, 2 MB of L2 cache per core, and a 60 MB shared L3 cache). The system was running Ubuntu 24.04.4 with Linux 6.8.0 and we compiled our code with gcc 13.3.0.

We use the `rdtsc` instruction for measuring time in terms of TSC cycles. For the relative timings with respect to theoretical single-core peak performance, we multiply the number of elapsed TSC cycles by the ratio $20/11$ between the maximum core frequency and the TSC frequency to obtain an upper bound on the number of core cycles. (Compared to disabling dynamic frequency scaling, this avoids artificially slowing down the processor relative to memory accesses.) In the case of absolute timings, we directly convert the number of TSC cycles to a duration in seconds.

The reported values are means or median figures over a number of runs depending on the problem scale. For full details, we refer to the source code of the programs used for producing our timings tables, which are distributed with our implementations. We stress however that the timings provided in this article are meant to provide a first validation of each implementation strategy as well as some elements of comparison between them but not intended as definitive performance figures.

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 $a = a_0 + a_1 2^\mu + \dots + a_{\ell-1} 2^{(\ell-1)\mu} \in \mathbb{N}_\mu[2^\mu]_\ell$ is the vector $(a_0, \dots, a_{\ell-1})$ 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 $(a_0 + \dots + a_{\ell-1} B^{\ell-1})(b_0 + \dots + b_{m-1} B^{m-1})$ 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 $a = a_0 + a_1 B + \dots + a_{\ell-1} B^{\ell-1}$ in some radix $B := 2^\mu$, but the limbs $a_0, \dots, a_{\ell-1} \in \mathbb{N}_{\bar{\mu}}$ are stored with a higher precision $\bar{\mu}$ (which may be different for multiplicands and products). If $a_0, \dots, a_{\ell-2} \in \mathbb{N}_\mu$, then $a \in \mathbb{N}_{\mu+\bar{\mu}-\mu}$ 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 of *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 $a, b \in \mathbb{N}_{52}$, we have

$$\text{ifmalo}(c, a, b) := (c + (a \cdot b \bmod 2^{52})) \bmod 2^{64}, \quad \text{ifmahi}(c, a, b) := (c + (a \cdot b \text{ quo } 2^{52})) \bmod 2^{64},$$

where $s \text{ quo } m$ and $s \bmod m$ 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 $c += ab$ 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 $C += AB$, 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, with our convention, 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.

Consider for instance the multiplication of $a_0 + \dots + a_{\ell-1} B^{\ell-1}$ and $b_0 + \dots + b_{m-1} B^{m-1}$ using integer FMAs, where $a_0, \dots, a_{\ell-1}, b_0, \dots, b_{m-1} \in \mathbb{N}_{52}$ and $B = 2^{52}$. Assume also that $\max(\ell, m) \leq 2^{\bar{\mu}-\mu} = 2048$. Then we first compute the non-normalized product $c = c_0 + \dots + c_{\ell+m-1} B^{\ell+m-1}$ with $c_0, \dots, c_{\ell+m-1} \in \mathbb{N}_{64}$, and then normalize it, as follows:

<i>Multiply</i>	<i>Normalize</i>
$c_0 := \text{ifmalo}(0, a_0, b_0)$	$c_1 := c_1 + (c_0 \text{ quo } 2^{52})$
$c_1 := \text{ifmalo}(0, a_0, b_1)$	$c_0 := c_0 \bmod 2^{52}$
$c_1 := \text{ifmahi}(c_1, a_0, b_0)$	$c_2 := c_2 + (c_1 \text{ quo } 2^{52})$
$c_1 := \text{ifmalo}(c_1, a_1, b_0)$	$c_1 := c_1 \bmod 2^{52}$
$c_2 := \text{ifmalo}(0, a_0, b_2)$	$\vdots$
$\vdots$	$c_{\ell+m-1} := c_{\ell+m-1} + (c_{\ell+m-2} \text{ quo } 2^{52})$
$c_{\ell+m-1} := \text{ifmahi}(0, a_{\ell-1}, b_{m-1})$	$c_{\ell+m-2} := c_{\ell+m-2} \bmod 2^{52}$

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.

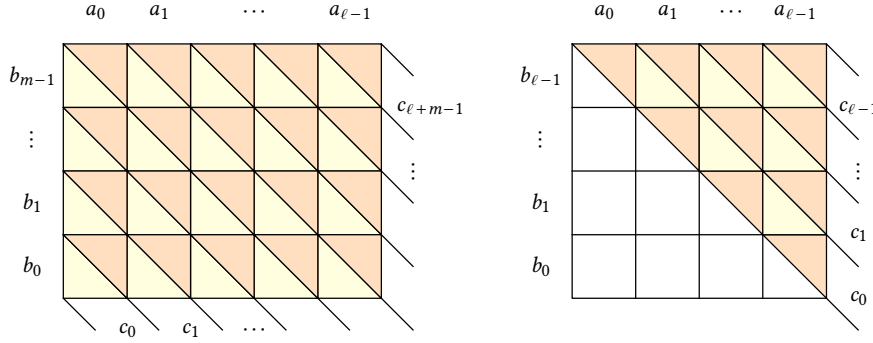


Figure 1. Illustration of a full multiple precision integer product (on the left) and a truncated high product (on the right) using integer FMAs (the lower yellow triangles and the upper orange triangles correspond to ifmalo and ifmahi, respectively).

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 $ab := \tilde{a}\tilde{b} - (\tilde{a} + \tilde{b})2^{52\ell-1} + 2^{104\ell-2}$. Alternatively, one may use a two's-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 $C \in \mathbb{N}_8^{16 \times 16}[2^8]_{2\ell-1}$ can then be done naively using ℓ^2 AMX-accelerated matrix multiplications $\mathbb{N}_8^{16 \times 64} \times \mathbb{N}_8^{64 \times 16} \rightarrow \mathbb{N}_8^{16 \times 16}$. 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 [44].

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^K -limb integers, we may either directly use limbs of bit-size $\mu - \kappa$ (which yields a multiplication algorithm for $2^K(\mu - \kappa)$ -bit integers), or do the limb rewritings recursively as well (which yields a multiplication algorithm for $(2^K(\mu - 2) + 2)$ -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 [65] 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 [37, Section 4.4.1], [45], [9, Exercise 1.4], and [60]) 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 $0 \leq i < j < \ell$. In this way, we need to first compute the ℓ products $P_0 Q_0, \dots, P_{\ell-1} Q_{\ell-1}$ 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 $(PQ)_{\ell-1}, \dots, (PQ)_{2\ell-2}$, then we only need to do $\ell + \lfloor \ell/2 \rfloor \lfloor \ell/2 \rfloor \sim \ell^2/4$ multiplications. This is 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 $A, B \in \mathbb{N}_{\ell\mu}^{n \times n}$ is based on the Chinese remainder theorem. We first pick 2ℓ pairwise coprime moduli $m_0, \dots, m_{2\ell-1} \in \mathbb{N}_\mu$. 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 $M \geq 2^{2\ell\mu-\epsilon}$.

In order to compute $C = AB$, we first reduce A and B modulo m_i for $i = 0, \dots, 2\ell - 1$. There is an elegant way to evaluate the map CRT: $\mathbb{N}_{\ell\mu} \rightarrow \mathbb{N}_\mu^{2\ell}; a \mapsto (a \bmod m_i)_{i < 2\ell}$ using a vector-matrix product [14]:

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

Taking the limbs of the entries of A as our rows $(a_0, \dots, a_{\ell-1})$, the computation of $A \bmod m_i$ for $i = 0, \dots, 2\ell - 1$ 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

$$C \bmod m_i := (A \bmod m_i)(B \bmod m_i) \bmod m_i$$

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

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

where u_i is the inverse of M/m_i modulo m_i for $i = 0, \dots, 2\ell - 1$. After precomputing the 2^μ -adic expansions $\frac{M}{m_i} u_i = \alpha_{i,0} + \cdots + \alpha_{i,2\ell-1} 2^{(2\ell-1)\mu}$, it is again possible to compute CRT⁻¹ using a vector-matrix product [14]:

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

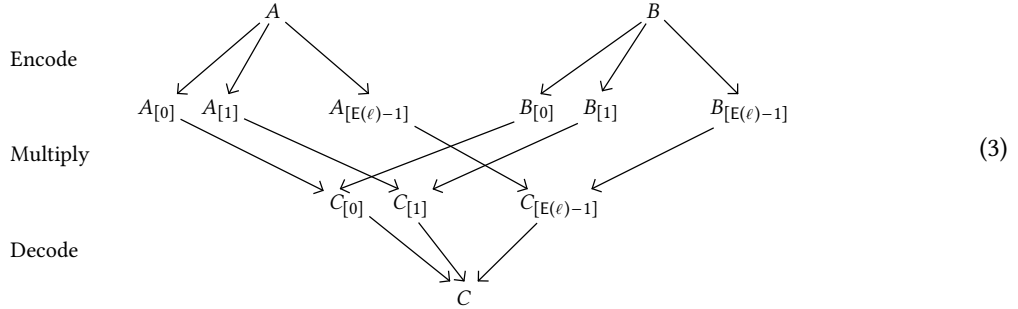
In a similar way as above, the recovery of C from $C \bmod m_0, \dots, C \bmod m_{2\ell-1}$ thus essentially boils down to an $n^2 \times (2\ell) \times (2\ell)$ matrix product over $\mathbb{N}_\mu$. In order to ease the computation of the remainder with respect to M , it is sometimes preferred to first compute the remainders of $u_i r_i$ 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 2ℓ .

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	2ℓ 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	136	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 $E(\ell)$ 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 $\ell \leq 8$, 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 count 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 Karatsuba², which recurses once.

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 $\mathbb{N}_{\mu}^{n \times n}$ to $E(\ell)$ multiplications $A_{[0]} B_{[0]}, \dots, A_{[E(\ell)-1]} B_{[E(\ell)-1]}$ of matrices in $\mathbb{N}_{\mu}^{n \times n}$, through the use of an appropriate encoding:



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

Table 1 summarizes the factors $E(\ell)$ 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.

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

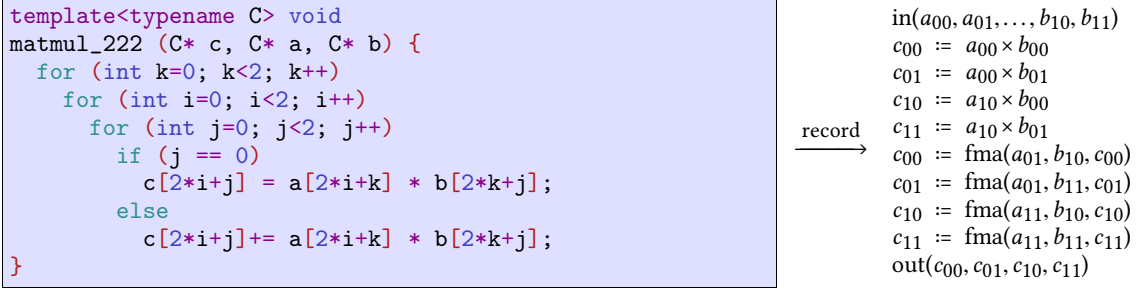


Figure 2. Illustration of the recording process of a program as an SLP. For the purpose of readability, we have chosen nice names $a_{00}, a_{01}, \dots, c_{11}$ for the data fields of the SLP. Inside JIL, the data fields are referred to by indices $0, 1, \dots, 11$.

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, 41] for computations and JIT compilation of straight-line programs (SLPs): we implement a large number 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 $\Sigma = \{+, -, \times, \text{fma}\}$ 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 type, 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 [41] 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 $\mathbb{N}_{52}^w$.

Two things matter for the design of efficient “naive” algorithms. We first should decide whether we view our multiplicands as matrices in $(\mathbb{N}_{52}^8)^{n \times n}$ with SIMD integer entries or as polynomials in $(\mathbb{N}_{52}^8)^{n \times n}[2^{52}]_\ell$ 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. 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.

Multiplicands in $(\mathbb{N}_{52\ell}^8)^{n \times n}$									Multiplicands in $(\mathbb{N}_{52}^8)^{n \times n} [2^{52}]_\ell$								
$n \setminus \ell$	1	2	3	4	6	8	12	16	$n \setminus \ell$	1	2	3	4	6	8	12	16
1	6.55	2.41	2.09	1.86	1.61	1.47	1.41	1.88	1	6.54	2.39	2.07	1.86	1.61	1.47	1.41	1.88
2	1.78	1.70	1.60	1.64	1.67	1.53	1.53	2.56	2	1.78	1.70	1.56	1.60	1.65	1.69	2.25	2.60
3	1.54	2.02	1.91	1.81	1.65	1.95	2.23	2.78	3	1.54	1.76	2.05	2.11	2.03	2.66	2.73	2.79
4	1.79	2.09	1.94	1.83	2.18	1.97	2.43	2.93	4	2.15	2.16	2.28	2.22	3.09	3.08	3.30	3.22
6	1.51	2.18	2.74	2.58	2.37	2.08	2.29	6.13	6	1.83	1.70	2.42	2.64	2.68	2.72	2.74	3.35
8	1.71	3.21	3.09	2.78	2.30	2.03	5.69	9.99	8	2.12	3.18	3.27	3.30	3.28	3.85	6.58	9.80
12	1.92	3.16	2.93	2.62	5.17	5.85	8.08	10.1	12	2.33	2.57	2.51	2.48	4.53	6.03	7.38	7.42
16	2.48	3.41	3.92	7.07	7.60	6.96	8.10	10.6	16	2.99	3.16	3.87	6.45	9.35	9.18	9.20	9.41

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.

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

The timings in Tables 2, 3, and 4 were obtained by running the program `mat_mul_bench` in the JIL library (GIT hash tag 999ee20) with the options `--delay=5 --peak --scale=0.55`.

Table 2 shows the ratios of our timings for an $\ell \times \ell \rightarrow 2\ell$ limb long $n \times n$ matrix multiplication divided by the theoretical time to perform $2n^3\ell^2$ integer FMAs over $\mathbb{N}_{52}^8$ (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.

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 12 MB in memory and does not even fit in the L2 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 $\mathbb{N}_{52\ell}^8$, 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 $\tilde{A} \in (\mathbb{N}_{52\ell}^8)^{n \times n}$, it is better to regard A directly as an $n \times (n/8)$ SIMD matrix over $\mathbb{N}_{52\ell}^8$, and to replicate individual coefficients of A across lanes whenever needed. This more compact representation of A has the advantage that we 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

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

where $0 \leq i < \tilde{r}$, $0 \leq k < \tilde{s}$, $0 \leq j < \tilde{c}$ and $0 \leq i' < w_r$, $0 \leq k' < w_s$, $0 \leq j' < w_c$. 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 .

Multiplicands in $\mathbb{N}_{52\ell}^{n \times n}$									Multiplicands in $\mathbb{N}_{52}^{n \times n}[2^{52}]_{\ell}$								
$n \setminus \ell$	1	2	3	4	6	8	12	16	$n \setminus \ell$	1	2	3	4	6	8	12	16
2	9.31	4.95	4.24	3.54	3.89	3.17	2.56	2.51	2	7.84	4.27	3.65	3.03	3.39	2.63	2.17	2.12
4	3.27	2.63	2.55	2.33	2.25	2.03	2.03	3.20	4	2.76	2.22	2.17	1.95	1.71	1.59	1.43	2.23
6	3.44	3.41	3.17	2.77	2.99	2.92	3.62	3.69	6	2.92	2.43	2.65	2.40	3.50	3.22	3.41	3.14
8	2.42	2.21	2.32	2.19	2.74	2.39	2.87	3.42	8	2.05	1.89	1.86	1.67	2.70	2.59	2.60	2.50
12	1.97	2.53	3.13	2.75	2.42	2.16	2.48	6.91	12	1.74	1.70	2.65	2.52	2.24	2.14	2.11	2.68
16	1.77	3.36	3.21	2.93	2.48	2.14	6.06	10.3	16	1.58	2.52	2.43	2.22	2.17	2.06	4.36	7.78
24	1.84	3.56	3.15	2.78	4.57	6.25	8.49	10.4	24	1.85	2.31	2.28	2.18	3.89	7.24	6.62	6.71
32	2.71	3.63	4.81	6.65	7.99	7.36	8.77	11.6	32	2.79	3.07	3.34	8.22	9.87	9.57	9.88	10.4

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.

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 2. 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.

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 mechanism (called “promises”, but unrelated to

$n \setminus \ell$	1	2	3	4	6	8	12	16
2		4.73		3.30	3.37	3.28	2.42	1.98
4	3.47	2.76	2.45	2.13	2.15	1.98	1.74	1.61
6		2.57		1.99	2.19	1.85	2.60	2.23
8	2.60	2.23	2.10	1.91	1.82	1.89	1.87	1.72
12	2.10	1.76	1.96	2.38	2.41	2.18	2.07	1.94
16	1.93	1.77	2.37	2.26	2.20	2.06	1.98	2.31
24	1.72	2.12	2.50	2.40	2.26	2.52	6.02	5.96
32	2.10	2.89	2.58	2.51	3.06	6.01	6.26	5.98

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.

$n \setminus \ell$	1	2	3	4	6	8	12	16	$n \setminus \ell$	1	2	3	4	6	8	12	16
32	3.12	2.42	2.41	2.34	2.34	2.54	2.83	3.29	32	4.58	2.71	2.27	2.35	2.32	2.37	2.66	2.87
64	1.91	1.63	1.67	1.77	1.92	2.00	2.09	5.89	64	2.47	1.76	1.52	1.69	1.86	1.98	2.06	2.09
96	1.58	1.62	1.70	1.79	1.80	1.81	4.24	4.72	96	1.91	1.70	1.50	1.69	1.81	1.82	1.80	4.31
128	1.45	1.60	1.65	1.68	1.63	3.44	3.88	4.66	128	1.68	1.70	1.52	1.62	1.64	1.63	3.51	3.95
192	1.52	1.52	1.49	1.47	2.66	3.02	3.57	3.81	192	1.66	1.61	1.38	1.43	1.42	2.78	3.27	3.60
256	1.46	1.43	1.42	2.32	2.62	2.93	3.06	3.19	256	1.64	1.50	1.28	1.32	2.38	2.70	3.03	3.09
384	1.50	1.47	1.47	1.86	2.90	3.16	3.53	3.76	384	1.80	1.61	1.38	1.44	2.52	3.03	3.37	3.59
512	1.46	1.39	1.37	2.41	2.56	2.86	3.04	3.14	512	1.70	1.44	1.25	1.31	2.49	2.67	3.01	3.07
768	1.44	1.38	1.38	2.40	2.55	2.83	3.02	3.13	768	1.67	1.43	1.25	1.30	2.48	2.63	2.99	3.04
1024	1.44	1.37	1.34	2.40	2.54	2.83	3.00	3.15	1024	1.67	1.43	1.24	1.29	2.47	2.65	2.97	3.05
1536	1.43	1.37	1.34	2.40	2.53	2.81	2.99	3.13	1536	1.66	1.46	1.24	1.29	2.46	2.62	2.96	3.03
2048	1.42	1.37	1.34	2.39	2.51	2.80	2.98	3.12	2048	1.65	1.44	1.25	1.29	2.46	2.60	2.95	3.02

Table 5. Relative timings for one long $\ell \times \ell \rightarrow 2\ell$ or one high $\ell \times \ell \rightarrow \ell$ 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.

the classical concurrency primitive of the same name) that allows it to compile slightly more complex programs than mere SLPs.

For instance, a 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. Our implementation of this step uses a “loop promise” whose compilation transparently 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 2ℓ 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 2ℓ desired matrices $\mathbb{N}_{52}^{n \times n}$ while processing the CRTs.

For the present revised version of our paper, we further optimized the critical loop of the main IFMA-based kernel using AVX512 intrinsics. This deviation from our original design strategy allowed us to test some ideas more quickly than with a pure JIL-based implementation. Through these experiments, we also identified a few features that are currently missing in JIL. We plan to extend JIL accordingly in the future and hope that this will further improve our timings.

5.2 Experiments

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. The timings were obtained using the command `matmul_ifma_bench --threads=1 --peak --scale=0.55`.

In Table 6, we consider long $\ell \times \ell \rightarrow 2\ell$ 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 $E(\ell)n^3/8$, the theoretical number of cycles to perform all IFMA operations in the multiplication step of (3). In Table 5, we show timings for the non-recursive variant of Karatsuba's algorithm, both in the case of long $\ell \times \ell \rightarrow 2\ell$ limb multiplication and high $\ell \times \ell \rightarrow \ell$ 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.

$n \setminus \ell$	1	2	3	4	6	8	12	16	$n \setminus \ell$	1	2	3	4	6	8	12	16
32	3.78	2.95	2.14	1.79	1.65	1.51	1.54	1.83	32	2.64	2.69	2.90	3.15	3.69	4.24	5.94	8.21
64	2.37	1.89	1.57	1.44	1.42	1.36	1.34	1.84	64	1.74	1.80	1.95	2.18	2.58	2.95	3.92	4.77
96	1.88	1.71	1.51	1.40	1.32	1.26	1.26	1.63	96	1.49	1.69	1.84	1.98	2.26	2.54	3.07	3.62
128	1.69	1.60	1.42	1.33	1.28	1.25	1.60	1.62	128	1.47	1.62	1.74	1.84	2.01	2.21	2.57	2.98
192	1.62	1.47	1.35	1.28	1.25	1.30	1.56	1.51	192	1.41	1.49	1.54	1.59	1.68	1.80	2.13	4.05
256	1.54	1.41	1.31	1.26	1.29	1.27	1.45	1.44	256	1.35	1.39	1.43	1.45	1.56	2.87	3.37	3.84
384	1.63	1.45	1.34	1.28	1.32	1.29	1.41	1.43	384	1.41	1.45	1.52	1.56	1.67	2.24	2.94	4.16
512	1.54	1.38	1.29	1.32	1.28	1.27	1.45	1.43	512	1.35	1.37	1.40	1.43	1.64	2.79	3.21	3.63
768	1.54	1.39	1.28	1.32	1.37	1.29	1.45	1.43	768	1.36	1.36	1.40	1.42	1.78	2.77	3.21	3.60
1024	1.53	1.37	1.28	1.31	1.37	1.28	1.45	1.42	1024	1.34	1.35	1.39	1.42	1.85	2.77	3.17	3.59
1536	1.52	1.37	1.27	1.31	1.28	1.26	1.44	1.42	1536	1.34	1.35	1.41	1.44	1.61	2.76	3.17	3.56
2048	1.52	1.36	1.26	1.31	1.28	1.26	1.44	1.42	2048	1.34	1.35	1.40	1.44	1.61	2.75	3.14	3.52

Table 6. Relative timings for one long $\ell \times \ell \rightarrow 2\ell$ 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.

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, $n \geq 256$ and $\ell \leq 8$).

6 Accelerating matrix multiplication using AMX

6.1 Implementation strategy

The AMX unit can be configured for computing the matrix FMA $C += AB$ where $A \in \mathbb{N}_8^{r \times s}$, $B \in \mathbb{N}_8^{s \times c}$, and $C \in \mathbb{N}_{32}^{r \times c}$ for any $r \leq 16$, $s \leq 64$, and $c \leq 16$ such that s is a multiple of four. The processor can issue one such MFMA every 16 cycles when $s = 64$, and somewhat faster if $s < 64$. The total number rsc 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 1 KB 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 1 KB 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) $C += AB$ for arbitrary sizes r, s, c , using INTEL intrinsics. In the future, we plan to extend JIL with special “promises” (supplementing those described in Section 5.1) for building code that gracefully mixes AMX and AVX512 instructions.

The core of our work on AMX is an efficient implementation of MFMA for $8 \times 8 \rightarrow 32$ bit entries. On top of it, we implemented multiple precision matrix multiplication using various schemes as in (3). The encoding and decoding stages are mostly done using AVX512 instructions and one of our main concerns will be to keep the cost of these stages reasonable. This is challenging due to the fact that the throughput of AVX512 instructions is 8 to 16 times lower than that of AMX instructions (for eight bit entries, we can do $16 \times 64 \times 16$ FMAs per 16 cycles with AMX versus 64 operations per cycle and per execution unit 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 Geijn [28]; see also [67, Sec. 5.1] and the recommendations of the Intel Optimization Reference Manual [43, 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 [51] 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 due to slightly different pressures on execution ports or differences between the speed of 256 and 512-bit loads from L1 cache.

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

The Np and Lp columns in Table 7 show the factors between the performance of our implementation and the theoretical peak performance. These timings can be reproduced with the command `matmul_alt_bench --peak --scale=0.55 --threads=no` using the provided software distribution. As the variance of the timings obtained with AMX is quite large, with frequent outliers that can significantly alter mean timings, this table shows median timings over a large number of runs. If one is satisfied with a factor of 2 to 3, then our current implementation will do, but approaching the theoretical peak performance as well as we did for IFMA-based methods seems to require significantly more effort. In Table 21 in the appendix, we can see that the naive AMX-based implementation is competitive with the IFMA-based one for large matrices.

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

n	AMX	N16	N24	N32	N48	N64	L16	L24	L32	L48	L64
64	2.71	12.9	9.78	7.97	6.39	5.56	12.1	8.75	7.22	6.10	5.31
96	4.43	9.69	8.72	7.85	7.06	6.73	8.87	7.94	7.40	6.89	6.66
128	3.79	6.99	6.25	5.82	5.37	5.15	6.53	5.80	5.47	5.31	5.20
192	2.74	4.61	4.18	4.08	4.11	4.35	4.28	4.17	4.16	4.56	4.41
256	2.34	3.84	3.77	4.03	4.01	3.89	3.78	4.29	4.19	4.00	3.81
384	1.75	4.02	3.71	3.55	3.30	3.15	4.09	3.74	3.51	3.24	3.08
512	1.80	4.26	3.40	3.19	2.93	2.82	3.81	3.35	3.12	2.90	2.77
768	1.82	2.87	2.62	2.49	2.36	2.31	2.83	2.57	2.44	2.36	3.98
1024	1.75	2.52	2.35	2.32	3.91	3.53	2.48	2.40	4.46	3.94	3.58
1536	2.05	2.60	2.59	4.28	3.86	3.67	2.72	4.84	4.44	3.86	3.51
2048	2.00	4.76	4.40	4.11	3.62	3.50	5.11	4.49	4.12	3.77	3.38
3072	1.96	4.24	3.87	3.70	3.32	3.06	4.84	4.09	3.72	3.23	2.97
4096	2.75	4.38	4.10	3.97	3.65	3.46	4.71	4.23	3.92	3.57	3.39

Table 7. Median factors with respect to theoretical peak performance for multiplying two $n \times n$ matrices of unsigned integers, measured using the `matmul_alt_bench` executable provided with our software. The AMX column corresponds to the raw $8 \times 8 \rightarrow 32$ bit product from Section 6.2. The Np columns correspond to a $p \times p \rightarrow p$ bit low product, and the Lp columns to a $p \times p \rightarrow 2p$ bit product, in both cases computed with the long multiplication strategy of Section 6.3.

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 $n^2 \ell \leq 2^{23}$). 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

We also implemented an AMX-accelerated version of the Chinese remaindering approach from section 3.6 in which both the modular products and the CRT transforms are based on AMX tile multiplication. As mentioned in the introduction, we used large language models, including in particular Claude Opus 5, to develop this kernel. More precisely, after deciding on a general implementation strategy (modular products using the AMX kernel presented above, CRT transforms by dedicated AMX-accelerated kernels in the style of [14], cache blocking), we used Claude Opus to prototype several variants of the algorithm, interactively refine certain details, and develop a first draft of the implementation we settled on. We then reviewed and significantly modified this first draft to obtain the version presented here.

Our implementation currently supports up to 32 moduli of 8 bits. The transforms (1) and (2) are done in large batches by replacing the left-hand rows in (1), (2) by matrices with one row per integer being transformed. This amounts to the multiplication of a tall and skinny matrix by a fixed encoding or decoding matrix of more balanced dimensions. Thus, while the multiplication itself benefits from AMX acceleration, memory traffic and the remaining reduction steps dominate the cost. In addition, typical integer multiplication formats like $64 \times 64 \rightarrow 128$ bits lead to matrix products with an inner dimension significantly smaller than 64, which cannot use the full power of the AMX unit and are best performed using adapted tile configurations such as $16 \times 16 \times 16$ or $16 \times 32 \times 16$.

In some more detail, our CRT-AMX kernel works as follows. The output matrix C is divided into square blocks to be computed sequentially as the product of a full-width horizontal panel of A and a full-height vertical panel of B . We determined experimentally that a block side of 256 to 512 works well in practice but have no complete theoretical explanation for this choice. The modular reductions of each horizontal panel of A are computed just before starting the work on the blocks of C that depend on it. The modular reductions of B , in contrast, are computed upfront for the whole matrix and immediately repacked into VNNI-encoded panels. In both cases, the multiplication by the encoding matrix yields a matrix of integers of at most 21 bits that need to be reduced modulo the 8-bit moduli and scattered in memory so that each reduced matrix is stored contiguously. This is done as the tiles of the product by the encoding matrix become available, using AVX512 single precision floating-point arithmetic for the 21-bit to 8-bit modular reduction. The VNNI-encoding step is a separate pass over blocks of B .

n	AMX										
		6	8	10	16	19	28	32		16	32
		C16	C24	C32	C55	C64	C96	C112		M124	M237
64	2.71	121.	94.8	59.4	57.2	51.4	39.9	45.3		40.2	45.4
96	4.43	79.3	62.3	38.8	37.4	33.8	26.7	31.1		26.5	32.4
128	3.79	59.3	46.7	28.8	28.2	25.5	21.4	24.6		19.9	25.8
192	2.74	39.8	31.7	20.1	20.4	18.5	14.8	16.9		14.3	17.3
256	2.34	31.1	25.6	16.3	16.1	14.6	11.9	13.3		11.7	13.6
384	1.75	21.5	17.9	11.6	11.3	10.4	8.56	9.60		8.48	9.82
512	1.80	16.7	14.0	9.07	8.84	8.15	6.85	7.52		6.76	7.77
768	1.82	11.9	9.72	6.53	6.53	6.06	5.21	5.84		5.21	8.95
1024	1.75	9.21	7.63	5.38	5.32	5.04	5.97	6.63		4.46	7.39
1536	2.05	7.33	6.13	4.63	5.70	6.49	6.07	6.58		4.94	7.50
2048	2.00	6.56	6.89	5.43	5.65	5.48	5.13	5.54		5.14	6.14
3072	1.96	5.35	5.18	4.36	4.44	4.33	4.17	4.45		4.09	4.69
4096	2.75	5.45	5.00	4.40	4.47	4.37	4.25	4.43		4.13	4.63

Table 8. This table complements Table 7 with timings for the CRT-based algorithms. We first repeated the column with the raw AMX times. The Cp columns correspond to a $p \times p \rightarrow 2p + 12$ bit multiplication (that is, multimodular multiplication of matrices with p -bit entries using enough moduli for the integer result to be exact for all considered matrix dimensions). The Mp columns correspond to multiplication modulo a fixed p -bit integer. Above these columns, we indicate the number of 8-bit moduli that we used.

For each block, we loop over the moduli and compute each modular product by a variant of the AMX Int8 matrix multiplication kernel of section 6.2. Compared to that kernel, the variant accepts a right operand pre-packed as a matrix of VNNI-encoded tiles instead of reorganizing it on the fly. In addition, it reduces each of the 32-bit entries of its output matrix modulo the current 8-bit modulus m and narrows it to 8 bits before storing it. For the reduction of a 32-bit integer $x_0 + x_1 2^8 + x_2 2^{16} + x_3 2^{24}$, we precompute $c_1 := 2^8 \bmod m$, $c_2 := 2^{16} \bmod m$ and $c_3 := 2^{24} \bmod m$, after which $x_0 + c_1 x_1 + c_2 x_2 + c_3 x_3$ can be computed in a single AVX512-VNNI instruction.

As soon as all the modular reductions of a given block of C have been computed, they are lifted to recover the corresponding integer entries. The inverse CRT map is implemented as a *left* multiplication by a fixed matrix, namely the transpose of the encoding matrix from (2), again followed by some postprocessing. An advantage of multiplying to the left is that the result comes out in a layout that is convenient for further processing using AVX512 instructions. The entries of the modular reduction of C can also be read by full cache lines in their natural layout, but must be VNNI-encoded before they are multiplied by the decoding matrix. As in the direct CRT phase, the tiles holding the result X of that multiplication are stored to a temporary buffer and postprocessed on the fly.

Writing M for the product of all the moduli in use as in section 3.6, the postprocessing needed to recover one entry of the final output matrix C consists in reducing modulo M the sum $x = \sum_j x_j 2^{8j}$ where the x_j are the entries of a given column of the matrix X . This is done for 16 columns of X in parallel, using vector instructions. Each x_j is the dot product of two vectors of length at most 32 of 8-bit integers, so $x_j < 2^{21}$ and we need to subtract from x a multiple $\lambda \cdot M$ of M with $\lambda < 2^{13}$. We estimate $\lambda = \lfloor x/M \rfloor$ from the most significant terms of the sum $\sum_j x_j (2^{8j}/M)$ evaluated in floating-point arithmetic, then compute $x + \lambda \cdot (2^{8\ell} - M) \bmod 2^{8\ell}$ in ℓ -limb integer arithmetic with full carry propagation. If our estimated λ was correct, this yields the normalized representation of $x \bmod M$. In rare cases, the estimate will be off by one unit, but provided that 8ℓ is strictly larger than the bit length of M (which can always be arranged), this can be detected from the result of the reduction and fixed. Finally, the $\ell \times 16$ matrix (represented as ℓ AVX512 vectors) of limbs obtained this way is transposed and each entry of C can be written to the destination buffer.

Table 8 shows the performance of our implementation relative to theoretical peak performance. For $n \times n \times n$ matrices and μ moduli, we compute this peak performance as the cost in cycles of μ AMX matrix multiplications in dimensions $n \times n \times n$; in other words, the normalization factor does not take into account the cost of Chinese remaindering. We see that the measured overhead is large for small matrices but comes within a factor of two of that of the naive method from the previous section when $n \geq 1024$ and $p \geq 32$. In view of the relative costs of the underlying algorithms, this means that the CRT-AMX kernel outperforms the schoolbook multiplication kernel in these cases. This is confirmed by the absolute timings in Table 22 of the appendix, which also shows that the CRT-AMX kernel is faster than our IFMA-based implementation for comparable bit sizes when $n \geq 512$.

Again, we have not investigated in detail how to best schedule the various stages of the algorithm to limit memory transfers as much as possible and overlap work using different CPU resources, but we expect that this could lead to some speedups for small matrices. Finally, as mentioned, our implementation is limited to 32 moduli, allowing the computation of results with entries of up to 237 bits. The same design could be extended to at most 54 coprime moduli and 362 bits, after which we run out of 8-bit primes. For larger bit-sizes, we plan 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 $12/7$.

7 Multi-threaded matrix multiplication

7.1 Implementation strategy and limitations

In order to measure the interest of IFMA and AMX instructions for matrix multiplication in the cleanest possible way, it is natural to first focus on their behavior on a single core, as we did in most of this paper. However, for real-world applications, we will also need multi-threaded versions, and we should examine how well the proposed strategies scale in that context.

In this section, we describe our preliminary multi-threaded implementation for the multiplication of very large integer matrices. The strategy is simple: for an $r \times s \times c$ matrix multiplication $A := BC$ or matrix FMA $A += BC$, we subdivide the A matrix in $b \times b$ blocks and (roughly) use $\lfloor r/b \rfloor \lfloor c/b \rfloor$ threads to execute $b \times s \times b$ matrix multiplications in parallel. Each $b \times s \times b$ multiplication is itself subdivided into $b \times b' \times b$ multiplications, where b and b' are chosen such that the $b \times b$ destination matrix and the $b \times b'$ and $b' \times b$ multiplicands fit into the local L2 cache. In fact, our implementation copies these basic blocks into hot thread-local memory, as this turned out to be more efficient than working directly on blocks of the original matrices.

For one large $r \times s \times c$ matrix multiplication, reading the $b \times b'$ and $b' \times b$ blocks from global memory requires about $\sim 2rsc/b$ coefficients to be transferred. These blocks typically do not fit into the L3 cache, so they really come from global memory. On the other hand, the design requires the rc output coefficients to be written only once to the global memory. In our implementation, all blocks are transferred using massive block matrix copy operations, and we usually pay the full price for cache misses. If the bandwidth of the global memory is small, then our implementation will become memory bound and the $\sim 2rsc/b$ read operations will dominate the total cost.

Our strategy is a compromise between performance and ease of implementation. It has the advantage that the $b \times b' \times b$ base multiplications can be done directly using any abstract single-threaded implementation. However, if the number of available threads becomes large, then it would be better to interleave the block transfers with the actual computations. But this improvement requires the single-threaded implementation to be adapted accordingly to allow for this kind of interleaving. For our first implementation, we have decided to ignore this intricate optimization, which doubles the efficiency at best.

It should be noted that the multiplication of small or medium size integer matrices may also be of interest in certain HPC applications in computer algebra, like sparse interpolation [58] or numerical homotopy continuation [62]. In that case, parallelization is trivial (although memory issues may still play a role), and it is relevant to optimize single-threaded algorithms, as we did in this work.

7.2 Experiments

In Table 9, we give the speed-ups for naive $n \times n$ matrix multiplication using IFMA instructions for various numbers of threads, for $\ell \times \ell \rightarrow 2\ell$ limb long multiplication with $\ell = 1, 2, 4$. Table 10 is similar, but for AMX instructions. For $t \leq 16$ threads, our CPU runs at its maximal boosted frequency of 4 GHz. For $t = 16, 32, 64$ threads, the frequency drops to 3.92, 3.42, and 2.65 GHz, allowing for theoretically maximal parallel speed-ups of 15.7, 27.4, and 42.4, respectively.

At present, we only implemented multi-threaded versions of the naive multi-limb algorithms. For variants based on Karatsuba multiplication and CRTs, it is natural to distribute the transformed products among the available cores, but to obtain a significant speedup this way it is necessary to parallelize the encoding and decoding phases as well.

$n \setminus t$	2	4	8	16	32	64	$n \setminus t$	2	4	8	16	32	64	$n \setminus t$	2	4	8	16	32	64
256	1.9	3.0	2.9	3.3	4.4	4.4	256	1.9	3.2	5.4	8.9	8.4	8.4	256	1.3	2.0	3.5	5.1	4.9	4.6
512	1.9	3.6	6.2	10.8	9.1	15.2	512	2.0	3.1	6.7	12.2	17.1	11.4	512	1.9	3.2	7.4	14.6	22.1	27.8
1024	2.0	3.7	7.0	13.2	21.0	23.5	1024	1.9	3.1	6.9	11.1	19.7	12.8	1024	2.0	2.6	6.9	12.1	20.1	26.0
2048	2.1	3.7	7.2	13.9	23.6	27.7	2048	1.9	3.2	7.2	12.0	18.9	23.9	2048	2.1	2.6	7.4	12.6	21.3	19.1
4096	2.0	3.7	7.3	14.0	23.0	29.3	4096	2.0	3.2	7.3	13.1	19.8	25.6	4096	2.1	2.6	7.5	13.6	21.1	20.4
8192	2.0	3.7	7.6	14.5	25.2	25.5	8192	2.0	3.3	7.8	14.1	21.3	27.1	8192	2.1	2.6	7.3	13.7	20.6	19.7
16384	2.0	3.8	7.4	14.9	25.0	26.8	16384	2.0	3.9	7.7	14.2	21.1	26.9	16384	1.3	2.4	6.2	8.6	20.1	19.4
32768	2.0	3.8	7.5	14.5	25.2	30.5														
$52 \times 52 \rightarrow 104$ bits							$104 \times 104 \rightarrow 208$ bits							$208 \times 208 \rightarrow 416$ bits						

Table 9. Speed-ups for naive multi-threaded $n \times n$ matrix multiplication using IFMA instructions and t threads. The three tables show the parallel speed-ups for $\ell \times \ell \rightarrow 2\ell$ limb long multiplication with $\ell = 1, 2, 4$ and $p = 104, 208, 416$, respectively.

$n \setminus t$	2	4	8	16	32	64	$n \setminus t$	2	4	8	16	32	64	$n \setminus t$	2	4	8	16	32	64
1024	0.7	0.8	1.3	2.1	3.1	3.5	1024	1.8	2.7	4.7	8.0	11.4	15.2	1024	3.0	5.2	9.9	17.3	23.7	29.6
2048	1.0	1.4	2.5	4.6	6.8	10.0	2048	1.7	5.0	7.7	10.7	15.6	15.8	2048	1.8	3.0	7.0	10.9	17.4	18.8
4096	1.3	1.8	3.6	7.1	12.4	19.7	4096	1.7	2.5	4.6	12.7	15.1	18.4	4096	1.7	3.0	5.5	7.9	7.3	21.2
8192	1.4	1.9	3.7	7.4	13.3	21.4	8192	1.7	2.9	4.6	5.9	6.7	6.7	8192	1.8	3.4	5.5	7.6	8.8	7.0
16384	1.4	1.8	3.6	7.0	13.6	20.9	16384	1.9	3.3	5.5	7.1	7.6	6.1	16384	2.0	3.9	7.2	10.8	13.6	11.3
32768	1.5	2.1	3.7	7.3	13.7	20.8	32768	1.9	3.7	6.7	10.5	12.5	11.5	32768	1.9	3.2	7.4	13.5	21.0	23.4
65536	1.6	2.5	4.5	8.5	15.2	23.6														
$8 \times 8 \rightarrow 16$ bits							$16 \times 16 \rightarrow 32$ bits							$32 \times 32 \rightarrow 64$ bits						

Table 10. Speed-ups for naive multi-threaded $n \times n$ matrix multiplication using AMX instructions and t threads. The three tables show the parallel speed-ups for $\ell \times \ell \rightarrow 2\ell$ limb long multiplication with $\ell = 1, 2, 4$ and $p = 8, 16, 32$, respectively.

8 Conclusion and perspectives

We have described a first implementation of accelerated integer matrix multiplication using INTEL's IFMA and AMX instructions. Most of the theory behind our work is well known, but we carefully describe how to adapt known algorithms in order to exploit the IFMA and AMX instructions. For $r \times s \times c$ matrix multiplication with p -bit coefficients, the best strategy and its efficiency greatly depend on r , s , c , and p . For many (r, s, c, p) regimes, our single-threaded implementation is within a factor 1.5 (IFMA) or 3 (AMX) of the theoretical peak performance of the best known multiplication strategy.

The IFMA-based variant is faster than state-of-the-art implementations based on GMP or double precision BLAS in almost all regimes, often by a substantial margin. In particular, it provides a good base strategy whenever IFMA instructions are available. Our implementation is very efficient for small matrices as well, thanks to the use of dedicated codelets for various sizes that are generated via JIL. The performance of the preliminary multi-threaded version scales reasonably well up to about 16 or 32 threads, but more work is needed to reach the theoretical peak performance for high numbers of threads. Nonetheless, we are confident that it should eventually be possible to make it as efficient as traditional double precision BLAS libraries, since IFMA is really a variant of the usual FMA instruction.

On a single core, our AMX-based implementation further improves upon the IFMA-based one for large matrices (e.g. $\min(r, s, c) \geq 128$ with best performance if r , s , and c are multiples of 64), even when the behavior of the former is close to optimal. It also provides much finer control over the bit-precision p . However, obtaining these results required a good deal of trial and error. The extremely high efficiency of the AMX instruction also requires the asymptotically subdominant AVX512 parts of the algorithms to be implemented with care. AMX-based implementations finally tend to become memory-bound for a lower number of threads, which makes multi-threading challenging.

Our code is still work in progress and there are multiple directions for future improvements. From a software engineering point of view, our ultimate aim is to provide a mixed-precision BLAS library that can easily be integrated into other software. For this, a better high-level interface is needed which automatically selects the best strategy for every (r, s, c, p) , after an auto-tuning phase for the specific machine. We believe that there is room for further lowering the overhead of our AMX-based multiplication kernels

by experimenting with other strategies and local optimizations. Last, but not least, we plan to extend the CRT-AMX kernel to 14 and 16 bit moduli, which will allow us to better take advantage of AMX accelerators for bit precisions $p > 128$.

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. Ahlback, 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*, 2021.
- [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] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. of the 19th Annual Symposium on Theory of Computing*, pages 1–6. New York City, may 25–27 1987.
- [13] W. da Silva Pereira. Accelerating floating-point computations with Intel AMX. Technical Report NREL/TP-2C00-93622, National Renewable Energy Laboratory, 2025.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.*
- [19] 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.
- [20] 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.
- [21] 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.
- [22] P. Fortin, A. Fleury, F. Lemaire, and M. Monagan. High performance SIMD modular arithmetic for polynomial evaluation. *ArXiv:2004.11571*, 2020.
- [23] M. Frigo. A fast Fourier transform compiler. In *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 34, pages 169–180. ACM, May 1999.
- [24] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on “Program Generation, Optimization, and Platform Adaptation”.
- [25] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 3rd edition, 2013.
- [26] 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.
- [27] G. Gerganov et al. Ggml. <https://github.com/ggml-org/ggml>, 2026.
- [28] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.
- [29] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. <http://www.swox.com/gmp>, 1991.
- [30] S. Gueron and V. Krasnov. Accelerating big integer arithmetic using intel IFMA extensions. In *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*, pages 32–38. 2016.
- [31] 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.
- [32] 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.

- [33] William Hart et al. FLINT: fast library for number theory. <https://flintlib.org>, 2010.
- [34] D. Harvey and J. van der Hoeven. On the complexity of integer matrix multiplication. *JSC*, 89:1–8, 2018.
- [35] D. Harvey and J. van der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, 193(2):563–617, 2021.
- [36] 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.
- [37] J. van der Hoeven. Relax, but don't be too lazy. *JSC*, 34:479–542, 2002.
- [38] J. van der Hoeven. *The Jolly Writer. Your Guide to GNU TeXmacs*. Scypress, 2020.
- [39] 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.
- [40] 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.
- [41] 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>.
- [42] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix. *ACM Trans. Math. Softw.*, 43(1):5–1, 2016.
- [43] Intel Corporation. Intel® 64 and IA-32 architectures optimization reference manual: volume 1. 2024.
- [44] A. Karatsuba and J. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.
- [45] 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.
- [46] 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.
- [47] 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.
- [48] Linear Algebra PACKage. <https://www.netlib.org/lapack>.
- [49] 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.
- [50] H.-J. Lebbink. HJlebbink/amx-matmul. 2024. <https://github.com/HJlebbink/AMX-matmul>.
- [51] T. Li. Usstq/mm_amx. https://github.com/usstq/mm_amx, 2024.
- [52] 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.
- [53] H. Martínez, A. Castelló, F. D. Igual, and E. S. Quintana-Orti. The Cambrian explosion of mixed-precision matrix multiplication for quantized deep learning inference. *Future Generation Computer Systems*, page 108231, 2025.
- [54] 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.
- [55] 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.
- [56] 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>.
- [57] 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. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.
- [58] D. S. Roche. What can (and can't) we do with sparse polynomials? In *Proc. ISSAC '18*, pages 25–30. New York, NY, USA, 2018. ACM.
- [59] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.
- [60] Michael Scott. Missing a trick: Karatsuba variations. *Cryptography and Communications*, 10(1):5–15, 2018.
- [61] V. Shoup. NTL: a library for doing number theory. 1996. www.shoup.net/ntl.
- [62] A. J. Sommese and C. W. Wampler. *The Numerical Solution of Systems of Polynomials Arising in Engineering and Science*. World Scientific, 2005.
- [63] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:352–356, 1969.
- [64] The OpenBLAS developers. Openblas. <http://www.openmathlib.org/OpenBLAS/>, feb 2026.
- [65] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics*, 4(2):714–716, 1963.
- [66] 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.
- [67] 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.
- [68] 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.
- [69] R. C. Whaley, A. Petit, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

A Absolute timings

As motivated in the introduction, the timings that we reported so far are relative to the theoretical peak performance. This appendix provides some of the corresponding absolute timings. We also added absolute reference timings for existing software.

Of course, the running times are typically best for bit lengths that are multiples of a limb's bit-length, and our present implementation is limited to such bit lengths. The limb size is typically 64 bits for traditional implementations, but 52 or 51 bits for our implementation based on the IFMA instruction. The tables for existing implementations therefore show the timings for multiples of both 52 and 64 bits and they need to be interpreted with the necessary care.

A.1 Pre-existing software

Tables 11 to 16 present some timings obtained with FLINT [33], NTL [61], and FFLAS [18], three well-known algebraic computation libraries. We use the last development versions available at the time we ran our experiment, namely revision 1e74b110 of the FLINT source tree, NTL release 11.6.0, and FFLAS revision 06aec3f8 installed as part of LINBOX revision bb590e22. FLINT and FFLAS are configured to link with OPENBLAS (revision 9e857d00). The code we use to build the external libraries and measure their running times is available at¹ <http://marc.mezzarobba.net/supplementary-material/amx/bench-others.tar>. 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 implementations working over $\mathbb{Z}/2^p\mathbb{Z}$ or $\mathbb{Z}/2^{p-1}\mathbb{Z}$ in addition to implementations working over $\mathbb{N}_p$.

$n \setminus p$	8	16	24	32	52	64	104	128	208	256	416	512	832
8	0.41 μ s	0.41 μ s	0.41 μ s	0.61 μ s	0.60 μ s	2.05 μ s	2.07 μ s	6.21 μ s	8.51 μ s	8.34 μ s	18.4 μ s	28.6 μ s	53.7 μ s
16	1.66 μ s	1.65 μ s	1.63 μ s	3.52 μ s	3.91 μ s	12.6 μ s	13.0 μ s	65.3 μ s	86.7 μ s	96.6 μ s	0.18 ms	0.23 ms	0.36 ms
32	9.21 μ s	9.36 μ s	9.24 μ s	28.9 μ s	29.7 μ s	87.6 μ s	87.0 μ s	0.29 ms	0.46 ms	0.53 ms	0.92 ms	2.11 ms	3.00 ms
64	21.3 μ s	19.7 μ s	61.4 μ s	0.20 ms	0.20 ms	0.64 ms	0.63 ms	1.43 ms	2.46 ms	2.61 ms	4.50 ms	5.52 ms	10.3 ms
96	92.1 μ s	90.6 μ s	0.33 ms	0.68 ms	0.66 ms	2.06 ms	2.01 ms	4.03 ms	6.39 ms	7.21 ms	12.1 ms	14.9 ms	27.1 ms
128	0.20 ms	0.19 ms	0.79 ms	1.52 ms	1.50 ms	4.77 ms	4.75 ms	8.70 ms	13.4 ms	15.1 ms	26.2 ms	31.8 ms	56.9 ms
192	0.51 ms	0.51 ms	2.15 ms	3.52 ms	3.47 ms	14.2 ms	15.5 ms	25.3 ms	39.9 ms	44.1 ms	76.7 ms	91.8 ms	0.16 s
256	1.07 ms	1.06 ms	4.79 ms	7.81 ms	7.97 ms	12.2 ms	19.2 ms	53.6 ms	84.6 ms	96.4 ms	0.17 s	0.20 s	0.36 s
384	2.84 ms	2.69 ms	15.3 ms	25.9 ms	25.6 ms	33.5 ms	61.5 ms	73.3 ms	0.13 s	0.13 s	0.23 s	0.28 s	0.50 s
512	5.95 ms	5.53 ms	36.5 ms	61.0 ms	62.7 ms	83.0 ms	0.15 s	0.15 s	0.25 s	0.25 s	0.42 s	0.53 s	0.96 s
768	16.3 ms	15.5 ms	0.12 s	0.21 s	0.23 s	0.25 s	0.38 s	0.38 s	0.64 s	0.69 s	1.11 s	1.40 s	2.37 s
1024	39.4 ms	33.6 ms	0.17 s	0.48 s	0.49 s	0.49 s	0.75 s	0.78 s	1.29 s	1.37 s	2.25 s	2.72 s	5.00 s
2048	0.28 s	0.28 s	1.00 s	1.49 s	2.00 s	2.36 s	3.49 s	4.36 s	6.18 s	7.52 s	11.8 s	15.2 s	27.6 s

Table 11. Matrix multiplication in $\mathbb{N}_p^{n \times n}$ using the `fmpz_mat_mul` function from FLINT.

$n \setminus p$	8	16	24	32	52	64	104	128	208	256	416	512	832
8	0.25 μ s	0.38 μ s	0.38 μ s	0.38 μ s	0.38 μ s	0.38 μ s	2.13 μ s	2.17 μ s	6.39 μ s	6.01 μ s	13.7 μ s	19.2 μ s	36.8 μ s
16	0.55 μ s	2.13 μ s	2.13 μ s	2.11 μ s	2.12 μ s	2.14 μ s	11.5 μ s	11.5 μ s	39.7 μ s	41.9 μ s	90.1 μ s	0.12 ms	0.25 ms
32	1.65 μ s	7.69 μ s	7.77 μ s	7.70 μ s	10.6 μ s	10.2 μ s	71.0 μ s	70.2 μ s	0.27 ms	0.37 ms	0.65 ms	0.91 ms	1.90 ms
64	6.31 μ s	38.9 μ s	39.0 μ s	39.0 μ s	62.3 μ s	60.4 μ s	0.50 ms	0.47 ms	2.10 ms	2.51 ms	5.02 ms	7.06 ms	8.49 ms
96	15.6 μ s	0.12 ms	0.12 ms	0.12 ms	0.25 ms	0.25 ms	1.56 ms	1.50 ms	5.75 ms	6.85 ms	11.8 ms	13.1 ms	22.8 ms
128	38.1 μ s	97.4 μ s	97.4 μ s	0.32 ms	0.61 ms	0.61 ms	3.61 ms	3.45 ms	13.2 ms	14.7 ms	24.9 ms	28.6 ms	49.4 ms
192	0.10 ms	0.28 ms	0.27 ms	0.96 ms	1.97 ms	1.96 ms	11.9 ms	20.6 ms	35.8 ms	41.2 ms	71.1 ms	82.5 ms	0.14 s
256	0.23 ms	0.66 ms	0.63 ms	2.37 ms	4.38 ms	4.37 ms	36.2 ms	46.4 ms	79.9 ms	92.0 ms	0.15 s	0.18 s	0.32 s
384	0.74 ms	1.90 ms	1.78 ms	7.42 ms	14.0 ms	14.0 ms	0.11 s	0.14 s	0.24 s	0.27 s	0.46 s	0.55 s	0.93 s
512	1.72 ms	3.93 ms	17.6 ms	23.5 ms	33.6 ms	41.5 ms	0.17 s	0.21 s	0.37 s	0.42 s	0.70 s	0.82 s	1.36 s
768	5.45 ms	12.3 ms	48.9 ms	65.2 ms	90.7 ms	0.11 s	0.48 s	0.60 s	0.98 s	1.11 s	1.84 s	2.15 s	3.56 s
1024	12.8 ms	29.6 ms	0.10 s	0.16 s	0.20 s	0.30 s	1.03 s	1.30 s	2.10 s	2.40 s	4.02 s	4.76 s	7.79 s
2048	91.4 ms	0.24 s	0.75 s	0.99 s	1.58 s	1.88 s	6.41 s	8.03 s	12.9 s	14.8 s	24.9 s	25.4 s	43.4 s

Table 12. Matrix multiplication in $(\mathbb{Z}/2^{p-1}\mathbb{Z})^{n \times n}$ using the `nmod_mat_mul_ui` (for $p \leq 8$), `nmod_mat_mul` (for $8 < p \leq 64$), or `mpn_mod_mat_mul` (otherwise) function from FLINT.

¹. Temporary location to be updated to a more stable one on publication.

$n \setminus p$	8	16	24	32	52	64	104	128	208	256	416	512	832
8	5.50 μ s	5.45 μ s	5.49 μ s	6.48 μ s	5.71 μ s	7.01 μ s	8.01 μ s	9.23 μ s	13.4 μ s	14.6 μ s	23.7 μ s	26.9 μ s	54.9 μ s
16	43.8 μ s	43.4 μ s	43.1 μ s	59.6 μ s	45.3 μ s	60.7 μ s	65.3 μ s	80.8 μ s	0.10 ms	0.12 ms	0.20 ms	0.23 ms	0.45 ms
32	0.33 ms	0.33 ms	0.33 ms	0.49 ms	0.35 ms	0.50 ms	0.51 ms	0.64 ms	0.84 ms	0.98 ms	1.63 ms	1.83 ms	3.55 ms
64	2.58 ms	2.57 ms	2.56 ms	3.91 ms	2.68 ms	4.02 ms	4.21 ms	5.44 ms	8.19 ms	9.83 ms	15.6 ms	17.2 ms	30.4 ms
96	9.21 ms	9.10 ms	9.10 ms	14.1 ms	9.48 ms	13.5 ms	15.4 ms	18.0 ms	36.5 ms	42.2 ms	68.3 ms	81.2 ms	0.13 s
128	32.1 ms	25.2 ms	31.2 ms	46.3 ms	35.1 ms	44.9 ms	53.0 ms	59.5 ms	0.11 s	0.12 s	0.20 s	0.20 s	0.34 s
192	0.13 s	0.13 s	0.13 s	0.19 s	0.13 s	0.19 s	0.20 s	0.23 s	0.32 s	0.33 s	0.55 s	0.66 s	1.10 s
256	0.32 s	0.32 s	0.33 s	0.46 s	0.34 s	0.46 s	0.48 s	0.59 s	0.74 s	0.80 s	1.31 s	1.41 s	2.65 s
384	1.16 s	1.15 s	1.18 s	1.60 s	1.23 s	1.62 s	1.77 s	2.02 s	2.41 s	2.58 s	4.48 s	5.31 s	8.50 s

Table 13. Multiplication of matrices in $\mathbb{N}_p^{n \times n}$ represented as NTL objects of type `mat_ZZ`.

$n \setminus p$	8	16	24	32	52	64	104	128	208	256	416	512	832
8	7.47 μ s	7.04 μ s	7.44 μ s	8.86 μ s	7.25 μ s	9.78 μ s	9.95 μ s	12.7 μ s	17.5 μ s	19.6 μ s	30.4 μ s	37.2 μ s	72.8 μ s
16	56.5 μ s	49.0 μ s	48.8 μ s	60.4 μ s	51.6 μ s	65.7 μ s	74.0 μ s	92.6 μ s	0.12 ms	0.14 ms	0.21 ms	0.26 ms	0.51 ms
32	93.6 μ s	0.12 ms	0.14 ms	0.14 ms	0.18 ms	0.21 ms	0.29 ms	0.34 ms	0.51 ms	0.60 ms	1.02 ms	1.34 ms	2.40 ms
64	0.36 ms	0.45 ms	0.49 ms	0.48 ms	0.64 ms	0.75 ms	1.06 ms	1.28 ms	1.94 ms	2.28 ms	3.96 ms	5.11 ms	9.31 ms
96	0.80 ms	0.98 ms	1.08 ms	1.09 ms	1.42 ms	1.66 ms	2.40 ms	2.87 ms	4.43 ms	5.23 ms	8.92 ms	11.2 ms	20.6 ms
128	1.39 ms	1.75 ms	1.92 ms	1.93 ms	2.57 ms	2.97 ms	4.30 ms	5.11 ms	7.91 ms	9.13 ms	15.5 ms	19.7 ms	36.6 ms
192	3.20 ms	4.12 ms	4.68 ms	4.64 ms	6.06 ms	7.01 ms	10.2 ms	12.1 ms	18.2 ms	21.3 ms	37.5 ms	47.5 ms	87.6 ms
256	5.91 ms	7.65 ms	8.59 ms	8.58 ms	11.4 ms	13.3 ms	19.3 ms	22.9 ms	34.8 ms	40.8 ms	71.4 ms	93.1 ms	0.18 s
384	14.1 ms	18.5 ms	21.4 ms	21.4 ms	28.8 ms	33.6 ms	50.0 ms	59.8 ms	92.6 ms	0.11 s	0.20 s	0.28 s	0.50 s
512	27.7 ms	36.3 ms	41.7 ms	42.0 ms	57.3 ms	67.8 ms	0.11 s	0.13 s	0.21 s	0.25 s	0.41 s	0.53 s	0.96 s
768	75.6 ms	0.10 s	0.12 s	0.13 s	0.18 s	0.21 s	0.32 s	0.37 s	0.55 s	0.65 s	1.09 s	1.37 s	2.45 s
1024	0.17 s	0.23 s	0.28 s	0.28 s	0.38 s	0.45 s	0.67 s	0.79 s	1.20 s	1.42 s	2.34 s	2.93 s	5.12 s
2048	0.95 s	1.33 s	1.67 s	1.66 s	2.40 s	2.79 s	4.27 s	5.05 s	7.58 s	9.12 s	14.9 s	18.6 s	30.7 s

Table 14. Multiplication of matrices in $(\mathbb{Z}/2^p\mathbb{Z})^{n \times n}$ represented as NTL objects of type `mat_ZZ_p`.

$n \setminus p$	8	16	24	32	52
8	0.58 μ s	0.57 μ s	0.58 μ s	0.74 μ s	0.73 μ s
16	0.84 μ s	0.85 μ s	0.85 μ s	3.97 μ s	4.03 μ s
32	3.09 μ s	3.04 μ s	2.84 μ s	12.4 μ s	16.3 μ s
64	14.5 μ s	14.4 μ s	14.3 μ s	92.0 μ s	0.12 ms
96	38.8 μ s	38.4 μ s	38.1 μ s	0.30 ms	0.41 ms
128	78.1 μ s	78.2 μ s	77.3 μ s	0.71 ms	0.95 ms
192	0.22 ms	0.22 ms	0.26 ms	2.38 ms	3.19 ms
256	0.49 ms	0.49 ms	0.55 ms	5.62 ms	7.61 ms
384	1.53 ms	1.53 ms	1.84 ms	19.1 ms	25.6 ms
512	3.54 ms	3.54 ms	4.35 ms	41.1 ms	54.8 ms
768	11.6 ms	11.4 ms	14.3 ms	0.14 s	0.18 s
1024	32.0 ms	30.2 ms	35.6 ms	0.29 s	0.39 s
2048	0.27 s	0.26 s	0.30 s	2.10 s	2.81 s

Table 15. Multiplication of matrices in $(\mathbb{Z}/2^{p-1}\mathbb{Z})^{n \times n}$ represented as NTL objects of type `mat_zz_p`.

$n \setminus p$	8	16	24	32	52	64	104	128	208	256	416	512	832
8	1.59 ms	1.60 ms	1.61 ms	1.62 ms	1.63 ms	1.65 ms	1.68 ms	1.69 ms	1.75 ms	1.80 ms	1.98 ms	2.04 ms	2.44 ms
16	1.63 ms	1.65 ms	1.67 ms	1.68 ms	1.70 ms	1.72 ms	1.78 ms	1.81 ms	1.96 ms	2.04 ms	2.37 ms	2.51 ms	3.14 ms
32	1.82 ms	1.82 ms	1.86 ms	1.92 ms	2.03 ms	2.07 ms	2.25 ms	2.38 ms	2.77 ms	3.00 ms	3.82 ms	4.24 ms	6.16 ms
64	2.56 ms	2.56 ms	2.80 ms	2.93 ms	3.35 ms	3.51 ms	4.13 ms	4.71 ms	6.20 ms	7.06 ms	10.3 ms	12.1 ms	18.1 ms
96	3.78 ms	3.72 ms	4.31 ms	4.59 ms	5.62 ms	5.95 ms	7.77 ms	8.64 ms	11.8 ms	14.7 ms	21.6 ms	25.9 ms	39.2 ms
128	5.52 ms	5.44 ms	6.03 ms	6.91 ms	8.37 ms	9.27 ms	12.9 ms	14.5 ms	21.0 ms	25.5 ms	39.9 ms	44.6 ms	72.4 ms
192	10.4 ms	10.4 ms	11.9 ms	13.7 ms	17.1 ms	19.0 ms	26.0 ms	28.9 ms	42.7 ms	51.5 ms	82.8 ms	0.11 s	0.18 s
256	16.5 ms	16.5 ms	19.3 ms	22.9 ms	28.8 ms	33.0 ms	47.0 ms	54.1 ms	85.2 ms	0.10 s	0.18 s	0.23 s	0.38 s
384	36.9 ms	44.1 ms	45.0 ms	55.4 ms	71.0 ms	78.6 ms	0.13 s	0.14 s	0.22 s	0.29 s	0.44 s	0.55 s	0.89 s
512	66.5 ms	79.3 ms	79.9 ms	0.10 s	0.14 s	0.17 s	0.25 s	0.27 s	0.42 s	0.53 s	0.86 s	1.04 s	1.69 s
768	0.16 s	0.20 s	0.24 s	0.24 s	0.34 s	0.41 s	0.60 s	0.71 s	1.06 s	1.28 s	2.15 s	2.60 s	4.21 s
1024	0.32 s	0.40 s	0.49 s	0.50 s	0.70 s	0.87 s	1.25 s	1.43 s	2.17 s	2.64 s	4.22 s	5.08 s	8.36 s
2048	1.59 s	2.04 s	2.56 s	3.06 s	4.15 s	4.64 s	6.71 s	8.22 s	12.8 s	15.3 s	24.2 s	29.6 s	48.4 s

Table 16. Multiplication of matrices in $\mathbb{N}_p^{n \times n}$ with entries of type `Givaro::ZRing<Givaro::Integer>` using the `f_gemm` function from FFLAS.

A.2 Multiplying small matrices using IFMA

Tables 17 and 18 show absolute counterparts of Tables 2, 3, and 4 from Section 4. Note that the left-hand of Table 17 corresponds to the left-hand of Table 2 (for multiplicands in $(\mathbb{N}_{52\ell}^8)^{n \times n}$), whereas the right-hand corresponds to the right-hand of Table 3 (for multiplicands that use the alternative representation in $\mathbb{N}_{52}^{n \times n}[2^{52}]_\ell$). The timings were obtained using `mat_mul_bench --delay=5`.

p	52	104	156	208	312	416	624	832		52	104	156	208	312	416	624	832
$n \setminus \ell$	1	2	3	4	6	8	12	16		1	2	3	4	6	8	12	16
1	1.86ns	2.76ns	5.37ns	8.48ns	16.7ns	26.8ns	57.9ns	0.14µs		2.15ns	4.61ns	8.91ns	13.2ns	34.3ns	46.4ns	84.2ns	0.15µs
2	4.07ns	15.6ns	32.7ns	60.2ns	0.14µs	0.22µs	0.51µs	1.50µs		20.7ns	68.7ns	0.17µs	0.27µs	0.90µs	1.49µs	3.60µs	5.86µs
3	11.9ns	61.5ns	0.13µs	0.22µs	0.46µs	1.03µs	2.49µs	5.53µs		5.95ns	19.2ns	42.0ns	67.4ns	0.13µs	0.22µs	0.43µs	1.20µs
4	32.8ns	0.15µs	0.32µs	0.53µs	1.43µs	2.30µs	6.32µs	13.5µs		20.7ns	68.7ns	0.17µs	0.27µs	0.90µs	1.49µs	3.60µs	5.86µs
6	91.1ns	0.53µs	1.50µs	2.51µs	5.19µs	8.19µs	20.6µs	88.9µs		35.6ns	0.13µs	0.29µs	0.47µs	1.71µs	2.90µs	6.51µs	11.1µs
8	0.25µs	1.81µs	3.94µs	6.38µs	11.8µs	18.7µs	0.10ms	0.36ms		99.9ns	0.39µs	1.31µs	2.22µs	4.81µs	8.20µs	17.7µs	35.2µs
12	0.93µs	5.99µs	12.6µs	20.1µs	75.7µs	0.18ms	0.56ms	1.24ms		0.22µs	1.39µs	3.07µs	5.09µs	10.8µs	18.2µs	84.6µs	0.23ms
16	2.77µs	15.1µs	40.7µs	0.11ms	0.31ms	0.51ms	1.36ms	3.27ms		0.86µs	4.17µs	9.09µs	16.7µs	53.3µs	0.17ms	0.45ms	0.78ms
24										2.99µs	13.1µs	37.7µs	0.12ms	0.37ms	0.65ms	1.45ms	2.54ms
32																	
multiplicands in $(\mathbb{N}_{52\ell}^8)^{n \times n}$									multiplicands in $\mathbb{N}_{52}^{n \times n}[2^{52}]_\ell$								

Table 17. Absolute timings for pure and tinkered SIMD multiplication of two matrices in $(\mathbb{N}_{52\ell}^8)^{n \times n}$ or $\mathbb{N}_{52}^{n \times n}[2^{52}]_\ell$.

p	52	104	156	208	312	416	624	832
$n \setminus \ell$	1	2	3	4	6	8	12	16
2		2.34ns		6.57ns	15.1ns	25.6ns	42.8ns	63.1ns
4	3.47ns	10.8ns	21.5ns	33.3ns	75.4ns	0.12µs	0.24µs	0.40µs
6		33.4ns		0.10µs	0.25µs	0.39µs	1.22µs	1.87µs
8	20.4ns	69.0ns	0.15µs	0.24µs	0.50µs	0.93µs	2.08µs	3.40µs
12	54.3ns	0.18µs	0.45µs	1.03µs	2.19µs	3.50µs	7.52µs	12.4µs
16	0.11µs	0.42µs	1.27µs	2.18µs	4.73µs	7.89µs	17.1µs	39.0µs
24	0.35µs	1.71µs	4.53µs	7.71µs	16.4µs	54.0µs	0.17ms	0.35ms
32	0.99µs	5.39µs	13.8µs	18.6µs	72.7µs	0.17ms	0.47ms	0.80ms

Table 18. Absolute timings for high multiplication of matrices in $\mathbb{N}_{52\ell}^{n \times n}$ in tinkered SIMD mode.

A.3 Multiplying large matrices using IFMA

Tables 19 presents the absolute counterparts of the timings in Table 6 from Section 5. Similarly, Table 20 provides absolute counterparts for the timings in Table 5.

p	52	104	156	208	312	416	624	832		51	103	155	207	311	415	623	831
$n \setminus \ell$	1	2	3	4	6	8	12	16		1	2	3	4	6	8	12	16
32	3.87µs	11.8µs	20.1µs	29.4µs	61.2µs	98.7µs	0.22ms	0.48ms		5.41µs	11.2µs	18.8µs	25.5µs	44.7µs	69.6µs	0.15ms	0.27ms
64	19.4µs	61.5µs	0.11ms	0.19ms	0.42ms	0.73ms	1.58ms	3.89ms		28.5µs	59.0µs	97.0µs	0.15ms	0.26ms	0.39ms	0.77ms	1.26ms
96	51.9µs	0.19ms	0.38ms	0.63ms	1.32ms	2.25ms	5.03ms	11.8ms		82.1µs	0.19ms	0.31ms	0.44ms	0.77ms	1.14ms	2.05ms	3.22ms
128	0.11ms	0.42ms	0.84ms	1.41ms	3.04ms	5.25ms	15.3ms	27.6ms		0.19ms	0.43ms	0.70ms	0.97ms	1.60ms	2.32ms	4.08ms	6.27ms
192	0.36ms	1.30ms	2.70ms	4.57ms	10.1ms	18.7ms	51.0ms	86.5ms		0.63ms	1.33ms	2.10ms	2.83ms	4.51ms	6.42ms	11.5ms	28.7ms
256	0.82ms	2.98ms	6.21ms	10.7ms	24.8ms	43.4ms	0.11s	0.19s		1.44ms	2.94ms	4.53ms	6.14ms	10.1ms	24.3ms	42.5ms	63.8ms
384	2.89ms	10.3ms	21.5ms	36.4ms	85.7ms	0.15s	0.37s	0.66s		5.01ms	10.4ms	16.4ms	22.3ms	35.8ms	64.2ms	0.13s	0.24s
512	6.49ms	23.3ms	49.3ms	91.2ms	0.20s	0.35s	0.89s	1.55s		11.4ms	23.2ms	35.7ms	48.3ms	84.5ms	0.19s	0.32s	0.49s
768	21.8ms	78.1ms	0.17s	0.31s	0.68s	1.17s	3.01s	5.22s		39.4ms	77.7ms	0.12s	0.16s	0.29s	0.64s	1.09s	1.64s
1024	51.8ms	0.19s	0.39s	0.72s	1.94s	3.07s	7.12s	12.3s		90.3ms	0.18s	0.28s	0.38s	0.70s	1.51s	2.58s	3.91s
1536	0.17s	0.62s	1.30s	2.42s	5.28s	9.25s	23.9s	41.5s		0.30s	0.61s	0.93s	1.29s	2.14s	4.93s	8.48s	12.7s
2048	0.41s	1.47s	3.08s	5.74s	12.5s	21.9s	56.6s	98.2s		0.71s	1.43s	2.22s	3.04s	5.15s	11.6s	19.9s	29.9s
Naive									CRT								

Table 19. Absolute timings for a long $\ell \times \ell \rightarrow 2\ell$ limb $n \times n$ matrix product using IFMA, both for the naive algorithm and the CRT method. The top row indicates the corresponding bit-precisions p .

$n \setminus p$	51	102	153	204	306	408	612	816	51	102	153	204	306	408	612	816
ℓ	1	2	3	4	6	8	12	16	1	2	3	4	6	8	12	16
32	3.19 μ s	7.58 μ s	15.3 μ s	23.6 μ s	50.8 μ s	91.5 μ s	0.22 ms	0.45 ms	2.32 μ s	5.60 μ s	9.48 μ s	15.6 μ s	31.0 μ s	52.0 μ s	0.12 ms	0.22 ms
64	15.7 μ s	40.1 μ s	85.1 μ s	0.15 ms	0.33 ms	0.58 ms	1.30 ms	6.49 ms	10.1 μ s	28.8 μ s	49.9 μ s	92.0 μ s	0.19 ms	0.34 ms	0.74 ms	1.25 ms
96	43.9 μ s	0.13 ms	0.28 ms	0.49 ms	1.03 ms	1.75 ms	9.01 ms	17.7 ms	26.4 μ s	87.8 μ s	0.17 ms	0.31 ms	0.64 ms	1.06 ms	2.18 ms	8.89 ms
128	96.2 μ s	0.31 ms	0.64 ms	1.08 ms	2.20 ms	8.01 ms	19.5 ms	40.7 ms	55.9 μ s	0.22 ms	0.39 ms	0.68 ms	1.37 ms	2.27 ms	10.2 ms	19.2 ms
192	0.33 ms	1.00 ms	1.98 ms	3.21 ms	12.3 ms	23.7 ms	60.7 ms	0.11 s	0.18 ms	0.70 ms	1.22 ms	2.05 ms	4.02 ms	13.0 ms	31.7 ms	59.3 ms
256	0.76 ms	2.24 ms	4.38 ms	12.0 ms	28.3 ms	54.2 ms	0.12 s	0.22 s	0.43 ms	1.54 ms	2.65 ms	4.44 ms	16.1 ms	29.8 ms	69.7 ms	0.12 s
384	2.64 ms	7.73 ms	15.5 ms	32.8 ms	0.11 s	0.20 s	0.48 s	0.89 s	1.58 ms	5.63 ms	9.61 ms	16.3 ms	59.9 ms	0.11 s	0.26 s	0.47 s
512	6.01 ms	17.3 ms	33.9 ms	99.7 ms	0.22 s	0.42 s	0.98 s	1.77 s	3.49 ms	12.0 ms	20.8 ms	35.3 ms	0.13 s	0.24 s	0.55 s	0.96 s
768	20.3 ms	57.9 ms	0.12 s	0.34 s	0.75 s	1.42 s	3.29 s	5.96 s	11.8 ms	40.2 ms	70.0 ms	0.12 s	0.45 s	0.79 s	1.86 s	3.22 s
1024	47.6 ms	0.14 s	0.27 s	0.79 s	1.76 s	3.36 s	7.74 s	14.1 s	27.7 ms	94.9 ms	0.17 s	0.28 s	1.06 s	1.87 s	4.37 s	7.64 s
1536	0.16 s	0.46 s	0.90 s	2.68 s	5.93 s	11.2 s	26.1 s	47.6 s	93.2 ms	0.33 s	0.55 s	0.94 s	3.58 s	6.28 s	14.7 s	25.6 s
2048	0.38 s	1.09 s	2.13 s	6.33 s	14.0 s	26.6 s	61.6 s	112. s	0.22 s	0.78 s	1.33 s	2.22 s	8.43 s	14.8 s	34.7 s	60.3 s
	long								short							

Table 20. Absolute timings for long $\ell \times \ell \rightarrow 2\ell$ and short $\ell \times \ell \rightarrow \ell$ limb $n \times n$ matrix products using IFMA and the non-recursive variant of Karatsuba’s algorithm. The top row also indicates the corresponding bit-precisions p .

A.4 Multiplying matrices using AMX

Table 21 shows absolute timings for naive AMX-accelerated matrix multiplication at various precisions; absolute timings for the CRT strategy are presented in Table 22. The corresponding relative timings with respect to the theoretical peak performance were given in Tables 7 and 8, respectively.

n	AMX	N16	N24	N32	N48	N64	L16	L24	L32	L48	L64	I52
64	0.17 μ s	2.48 μ s	3.76 μ s	4.78 μ s	8.32 μ s	12.3 μ s	3.20 μ s	4.78 μ s	7.14 μ s	13.5 μ s	21.5 μ s	19.4 μ s
96	0.93 μ s	5.90 μ s	11.2 μ s	16.9 μ s	31.5 μ s	52.3 μ s	7.61 μ s	15.3 μ s	25.4 μ s	54.1 μ s	91.9 μ s	51.9 μ s
128	1.96 μ s	10.4 μ s	19.0 μ s	29.7 μ s	58.0 μ s	95.5 μ s	13.1 μ s	26.9 μ s	44.8 μ s	99.1 μ s	0.18 ms	0.11 ms
192	4.40 μ s	23.6 μ s	43.5 μ s	70.5 μ s	0.15 ms	0.27 ms	29.6 μ s	64.5 μ s	0.12 ms	0.29 ms	0.49 ms	0.36 ms
256	9.57 μ s	47.6 μ s	96.3 μ s	0.16 ms	0.35 ms	0.58 ms	65.6 μ s	0.15 ms	0.28 ms	0.61 ms	1.01 ms	0.82 ms
384	24.1 μ s	0.16 ms	0.31 ms	0.49 ms	0.95 ms	1.57 ms	0.23 ms	0.47 ms	0.78 ms	1.63 ms	2.75 ms	2.89 ms
512	62.2 μ s	0.37 ms	0.67 ms	1.05 ms	2.02 ms	3.32 ms	0.50 ms	0.99 ms	1.65 ms	3.41 ms	5.79 ms	6.49 ms
768	0.21 ms	0.96 ms	1.75 ms	2.79 ms	5.56 ms	9.37 ms	1.26 ms	2.59 ms	4.36 ms	9.67 ms	28.8 ms	21.8 ms
1024	0.45 ms	1.96 ms	3.61 ms	5.94 ms	21.9 ms	34.6 ms	2.73 ms	5.45 ms	19.8 ms	39.3 ms	62.4 ms	51.8 ms
1536	1.81 ms	7.00 ms	25.0 ms	38.8 ms	73.2 ms	0.12 s	9.99 ms	39.5 ms	63.0 ms	0.13 s	0.21 s	0.17 s
2048	4.24 ms	30.5 ms	55.8 ms	84.9 ms	0.16 s	0.28 s	43.9 ms	85.7 ms	0.14 s	0.28 s	0.46 s	0.41 s
3072	14.4 ms	91.9 ms	0.17 s	0.26 s	0.50 s	0.79 s	0.13 s	0.26 s	0.42 s	0.84 s	1.37 s	1.41 s
4096	48.0 ms	0.22 s	0.41 s	0.67 s	1.29 s	2.11 s	0.31 s	0.64 s	1.06 s	2.18 s	3.67 s	3.36 s

Table 21. 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 $p \times p \rightarrow p$ bit low product, whereas the columns Lp corresponds to a $p \times p \rightarrow 2p$ bit long product. The column AMX corresponds to the raw $8 \times 8 \rightarrow 32$ bit product from Section 6.2. For reference, the column I52 indicates the timings of $1 \times 1 \rightarrow 2$ limb products using IFMA.

n	AMX	10 L32	16 L55	19 L64	28 L96	30 L104	I52	I104
64	0.17 μ s	37.6 μ s	58.3 μ s	62.5 μ s	71.1 μ s	93.1 μ s	19.4 μ s	61.5 μ s
96	0.93 μ s	83.4 μ s	0.13 ms	0.14 ms	0.16 ms	0.22 ms	51.9 μ s	0.19 ms
128	1.96 μ s	0.15 ms	0.23 ms	0.25 ms	0.30 ms	0.41 ms	0.11 ms	0.42 ms
192	4.40 μ s	0.35 ms	0.56 ms	0.60 ms	0.72 ms	0.94 ms	0.36 ms	1.30 ms
256	9.57 μ s	0.65 ms	1.03 ms	1.13 ms	1.37 ms	1.78 ms	0.82 ms	2.98 ms
384	24.1 μ s	1.57 ms	2.51 ms	2.74 ms	3.37 ms	4.30 ms	2.89 ms	10.3 ms
512	62.2 μ s	2.97 ms	4.72 ms	5.12 ms	6.39 ms	8.18 ms	6.49 ms	23.3 ms
768	0.21 ms	7.34 ms	11.9 ms	13.1 ms	17.2 ms	31.8 ms	21.8 ms	78.1 ms
1024	0.45 ms	14.5 ms	23.5 ms	27.2 ms	47.5 ms	61.5 ms	51.8 ms	0.19 s
1536	1.81 ms	50.3 ms	0.10 s	0.11 s	0.15 s	0.19 s	0.17 s	0.62 s
2048	4.24 ms	0.12 s	0.20 s	0.22 s	0.31 s	0.38 s	0.41 s	1.47 s
3072	14.4 ms	0.31 s	0.51 s	0.58 s	0.84 s	1.02 s	1.41 s	4.93 s
4096	48.0 ms	0.74 s	1.22 s	1.41 s	2.04 s	2.44 s	3.36 s	11.7 s

Table 22. Time to multiply two $n \times n$ matrices with p bit positive integer coefficients using AMX-accelerated arithmetic and CRTs. The columns Lp corresponds to a $p \times p \rightarrow 2p$ bit long product; above the entry Lp, we also indicate the number of 8-bit prime moduli that we use (e.g., for a $104 \times 104 \rightarrow 208$ bit multiplication, we use 30 moduli). The column AMX corresponds to the raw $8 \times 8 \rightarrow 32$ bit product from the Section 6.2. For reference, the columns I52 and I104 indicate timings for $1 \times 1 \rightarrow 2$ and $2 \times 2 \rightarrow 4$ limb products using IFMA.

A.5 Multi-threaded variants

In Table 23, we show absolute timings for the multi-threaded algorithms from section 7.

p	52	104	156	208	8	16	24	32
$n \setminus \ell$	1	2	3	4	1	2	3	4
256	0.18 ms	0.26 ms	0.35 ms	2.29 ms	93.5 μ s	0.10 ms	0.10 ms	0.11 ms
512	0.42 ms	1.18 ms	2.01 ms	3.15 ms	0.13 ms	0.13 ms	0.14 ms	0.19 ms
1024	2.17 ms	6.92 ms	14.1 ms	26.5 ms	0.14 ms	0.21 ms	0.41 ms	0.72 ms
2048	14.8 ms	60.2 ms	0.12 s	0.29 s	0.50 ms	3.04 ms	5.43 ms	7.99 ms
4096	0.11 s	0.45 s	0.95 s	2.17 s	2.58 ms	17.9 ms	32.4 ms	52.0 ms
8192	1.02 s	3.51 s	7.02 s	17.9 s	19.2 ms	0.32 s	0.59 s	1.12 s
16384	7.76 s	28.3 s	57.7 s	141. s	0.15 s	2.47 s	4.33 s	6.65 s
32768	54.6 s				1.27 s	11.7 s	19.1 s	36.3 s
65536					10.4 s			
	IFMA				AMX			

Table 23. Absolute timings for a long $\ell \times \ell \rightarrow 2\ell$ limb $n \times n$ matrix product using 64 threads. We both show the timings for the IFMA- and AMX-accelerated algorithms and indicate the corresponding bit precisions in the top row.