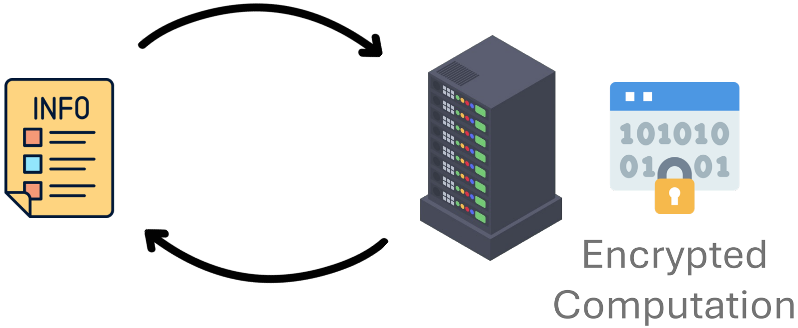
Code Generation for Cryptographic Kernels using Multi-word Modular Arithmetic on GPU

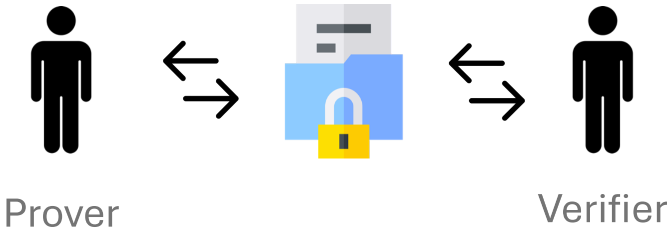
Naifeng Zhang, Franz Franchetti

Carnegie Mellon University

CGO 2025

Great Data Security Comes at a High Cost





Fully homomorphic encryption (FHE)

Zero-knowledge proofs (ZKPs)

Cost: Prohibitive computational overhead

Polynomial Operations with LARGE Integer Arithmetic

- **Polynomial addition** over a finite field $\mathbb{Z}_q$: $c_i = a_i + b_i \ mod \ q$

$$
\begin{array}{rl}
 & a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\
\odot \ - \ + & b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n \\
\hline
 & c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n
\end{array}
$$

If q has 768 bits

9400404716571063508556852750529110312590163184420194305731309276787470628524068602693276977567248081577601725741713586280758645193178925688817930839047860937980852238409160852231667754423147488134061040342175941846528472731375862 3
+
655259184398296582466247295398763281354874911315584037974648631746070155473174338128454088121843365430983733012799018315411809397370431870750882804530437926738049254081789423214828789402508715705785545949365131995115367952377606 09
mod
502127587881804164602367968438793419423193996407906432742324490413550496813367821396697643972790891987357506879353926720786742250232418444888384822364918563149787330341477236841223229936947388726464713811935837712185398267636318627

Cryptographic Kernel I: BLAS(-Like) Operations

Polynomial addition (over $\mathbb{Z}_q$)

$c_i = a_i + b_i \; mod \; q$

$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$
$\hookrightarrow [a_0, a_1, a_2, \cdots, a_n]$

Polynomial subtraction

Point-wise **polynomial multiplication**

Vector addition

$c_i = a_i + b_i \; mod \; q$

$[c_0, c_1, c_2, \cdots, c_n] = [a_0, a_1, a_2, \cdots, a_n] + [b_0, b_1, b_2, \cdots, b_n]$

Vector subtraction

Point-wise **vector multiplication**

Basic Linear Algebra Subprograms (BLAS)

Cryptographic Kernel II: Number Theoretic Transform

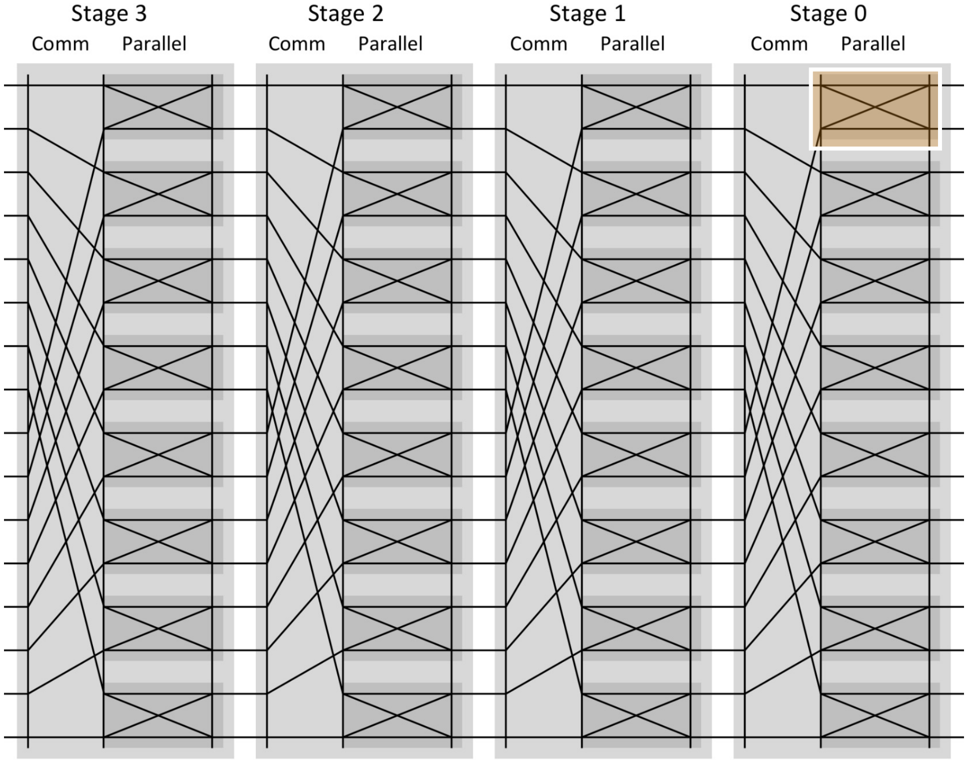
- **Polynomial multiplication**
  - Schoolbook multiplication takes $O(n^2)$

$$
\begin{array}{rl}
 & a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\
\times & b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n \\
\hline
 & c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n
\end{array}
$$

Not obvious!

- **Number Theoretic Transform (NTT):** $O(n \log n)$

NTT, the Butterfly, and MORE Large Integer Arithmetic



Pease NTT algorithm

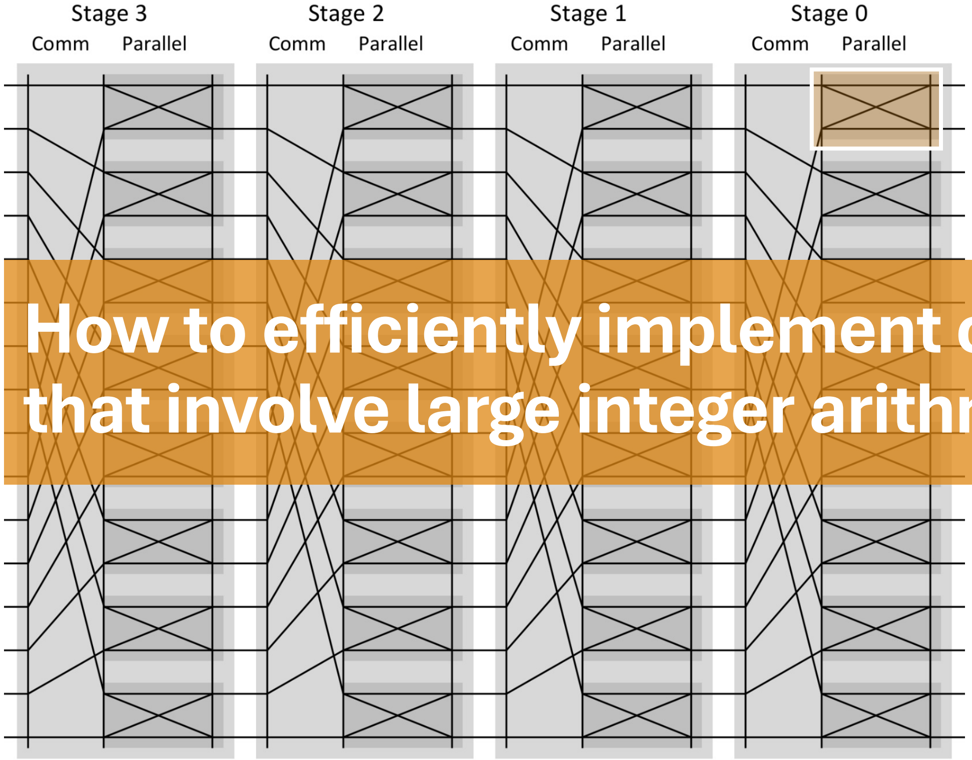
- **Butterfly**
  - 1 modular addition
  - 1 modular subtraction
  - 1 modular multiplication

  …on large integers

  94004047165710635085568527505291103125901631844201943057313092767874706285240686026932769775672480815776017257417135862807586451931789256888179308390478609379808522384091608522316677544231474881340610403421759418465284727313758623
  +
  65525918439829658246624729539876328135487491131558403797464863174607015547317433812845408812184336543098373301279901831541180939737043187075088280453043792673804925408178942321482878940250871570578554594936513199511536795237760609
  mod
  50212758788180416460236796843879341942319399640790643274232449041355049681336782139669764397279089198735750687935392672078674225023241844888384822364918563149787330341477236841223229936947388726464713811935837712185398267636318627

- **>90%** runtime for FHE-based and **~30%** for ZKP-based workloads

NTT, the Butterfly, and MORE Large Integer Arithmetic



Pease NTT algorithm

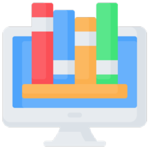
- **Butterfly**
  - 1 modular addition
  - 1 modular subtraction
  - 1 modular multiplication

How to efficiently implement cryptographic kernels that involve large integer arithmetic?

9400404716571063508556852750529110312590163184420194305731309276787470628524O
60269327697756724808157760172574171358628075864519317892568881793083904786O
79808522384091608522316677544231474881340610403421759418465284727313758623
+
655259184398296582466247295398763281354874911315584037974648631746070155473I7
43381284540881218433654309837330127990183154118093973704318707508828045304379
2673804925408178942321482878940250871570578554594936513199511536795237760609
mod
50212758788180416460236796843879341942319399640790643274232449041355049681336
78213966976439727908919873575068793539267207867422502324184488838482236491856
3149787330341477236841223229936947388726464713811935837712185398267636318627

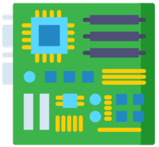
- **>90%** runtime for FHE-based and **~30%** for ZKP-based workloads

State-of-the-Art Approaches



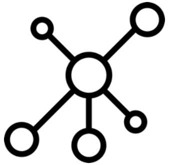
Arbitrary precision **libraries or programming languages**

- GNU multiple precision (GMP) library, Python, Rust



Specialized hardware support on **application-specific integrated circuits (ASICs)**

Performance



Generalizability

Cost

→ Multi-word Modular Arithmetic (MoMA)

Part I: Modular Arithmetic

Math (over $\mathbb{Z}_q$)

$$c = a + b \mod q$$

$$c = a - b \mod q$$

$$c = ab \mod q$$

Algorithm

$$c = \begin{cases} a + b - q, & \text{if } (a + b) > q, \\ a + b, & \text{otherwise.} \end{cases}$$

$$c = \begin{cases} a - b + q, & \text{if } a < b, \\ a - b, & \text{otherwise.} \end{cases}$$

$$c = ab - \lfloor ab \lfloor 2^k / q \rfloor / 2^k \rfloor q, \quad \mu = \lfloor 2^k / q \rfloor$$

Barrett reduction

Part II: Multi-digit Arithmetic

Multi-digit representation $[x_0, x_1, \ldots, x_{n-1}]_z = x_0 z^{n-1} + x_1 z^{n-2} + \ldots + x_{n-1} = x$

$[8{,}9]_{10} = 8 \cdot 10 + 9 = 89$

$[1152921504606846975{,}18446744073709550897]_{2^{64}}$
$= 21267647932558653966460912964485512497$

1

Modular addition algorithm

$$c = \begin{cases} a + b - q, & \text{if } (a+b) > q, \\ a + b, & \text{otherwise.} \end{cases}$$

$a = [a_0, a_1]_z = a_0 z + a_1$

$b = [b_0, b_1]_z = b_0 z + b_1$

2 3

Double-Word modular addition

2^{64} (On x86-64 architectures)

$[\delta, c_2]_z = a_1 + b_1,$

$[c_0, c_1]_z = a_0 + b_0 + \delta,$

where $c = [c_0, c_1, c_2]_z$ and $\delta \in \{0, 1\}$.

Multi-word Modular Arithmetic via Recursion

- Let the input bit-width be λ
- For each operation, apply **double-word modular arithmetic** to break it down to computations with bit-width $\lambda/2$
- Repeat until every resulting data type has bit-width $\lambda/2^k \leq \omega_0$
  - ω_0 is the machine word width

How to implement this?

Code Generation for MoMA: Rewriting on Data Types

$$
\begin{aligned}
a^{2\omega} &\rightarrow [a_0^\omega, a_1^\omega] && (19)\\
c_0^\omega = \lfloor [a_0^\omega, a_1^\omega]/2^\omega \rfloor &\rightarrow c_0^\omega = a_0^\omega && (20)\\
c_0^\omega = [a_0^\omega, a_1^\omega] \bmod 2^\omega &\rightarrow c_0^\omega = a_1^\omega && (21)\\
[c_0^1, c_1^\omega, c_2^\omega] = [a_0^\omega, a_1^\omega] + [b_0^\omega, b_1^\omega] &\rightarrow [\delta_0^1, c_2^\omega] = a_1^\omega + b_1^\omega,\ [c_0^1, c_1^\omega] = \delta_0^1 + a_0^\omega + b_0^\omega && (22)\\
[c_0^1, c_1^\omega] = a_1^\omega + b_1^\omega &\rightarrow c_0^1 = \lfloor (a_1^\omega + b_1^\omega)/2^\omega \rfloor,\ c_1^\omega = (a_1^\omega + b_1^\omega) \bmod 2^\omega && (23)\\
[c_0^\omega, c_1^\omega] = [a_0^1, a_1^\omega, a_2^\omega] \bmod [q_0^\omega, q_1^\omega] &\rightarrow \delta_0^1 = [q_0^\omega, q_1^\omega] < [a_1^\omega, a_2^\omega],\\
&\qquad \delta_1^1 = (0 < a_0^1) \vee ((a_0^1 =_? 0) \wedge \delta_0^1),\\
&\qquad [b_0^\omega, b_1^\omega] = [a_1^\omega, a_2^\omega] - [q_0^\omega, q_1^\omega], && (24)\\
&\qquad [c_0^\omega, c_1^\omega] = \begin{cases} [b_0^\omega, b_1^\omega], & \text{if } \delta_1^1 =_? 1,\\ [a_1^\omega, a_2^\omega], & \text{otherwise} \end{cases}\\
[c_0^\omega, c_1^\omega] = [a_0^\omega, a_1^\omega] - [b_0^\omega, b_1^\omega] &\rightarrow c_1^\omega = a_1^\omega - b_1^\omega,\ \delta_0^1 = a_1^\omega < b_1^\omega,\ c_0^\omega = a_0^\omega - b_0^\omega - \delta_0^1 && (25)\\
\delta_0^1 = [a_0^\omega, a_1^\omega] < [b_0^\omega, b_1^\omega] &\rightarrow \delta_0^1 = (a_0^\omega < b_0^\omega) \vee ((a_0^\omega =_? b_0^\omega) \wedge (a_1^\omega < b_1^\omega)) && (26)\\
\delta_0^1 = [a_0^\omega, a_1^\omega] =_? [b_0^\omega, b_1^\omega] &\rightarrow (a_0^\omega =_? b_0^\omega) \wedge (a_1^\omega =_? b_1^\omega) && (27)\\
[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [a_0^\omega, a_1^\omega] \cdot [b_0^\omega, b_1^\omega] &\rightarrow [d_0^\omega, d_1^\omega] = a_1^\omega \cdot b_1^\omega,\ [e_0^\omega, e_1^\omega] = a_0^\omega \cdot b_0^\omega,\\
&\qquad [f_0^\omega, f_1^\omega] = a_0^\omega \cdot b_1^\omega,\ [g_0^\omega, g_1^\omega] = a_1^\omega \cdot b_0^\omega, && (28)\\
&\qquad [h_0^1, h_1^\omega, h_2^\omega] = [f_0^\omega, f_1^\omega] + [g_0^\omega, g_1^\omega],\\
&\qquad [c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [e_0^\omega, e_1^\omega, d_0^\omega, d_1^\omega] + [h_0^1, h_1^\omega, h_2^\omega, 0]\\
[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [a_{0\text{-}3}^\omega] + [b_{0\text{-}3}^\omega] &\rightarrow [\delta_0^1, c_3^\omega] = a_3^\omega + b_3^\omega,\ [\delta_1^1, c_2^\omega] = a_2^\omega + b_2^\omega + \delta_0^1, && (29)\\
&\qquad [\delta_2^1, c_1^\omega] = a_1^\omega + b_1^\omega + \delta_1^1,\ [0, c_0^\omega] = a_0^\omega + b_0^\omega + \delta_2^1
\end{aligned}
$$

MoMA core rewrite rules ($[x_0, x_1, \dots, x_{k-1}]_{2^{\omega_0}} = [x_0^{\omega_0}, x_1^{\omega_0}, \dots, x_{k-1}^{\omega_0}]$)

Code Generation for MoMA: Rewriting on Data Types

$$
\begin{aligned}
a^{2\omega} &\rightarrow [a_0^\omega, a_1^\omega] && (19)\\
c_0^\omega = \lfloor [a_0^\omega, a_1^\omega]/2^\omega \rfloor &\rightarrow c_0^\omega = a_0^\omega && (20)\\
c_0^\omega = [a_0^\omega, a_1^\omega] \bmod 2^\omega &\rightarrow c_0^\omega = a_1^\omega && (21)\\
[c_0^1, c_1^\omega, c_2^\omega] = [a_0^\omega, a_1^\omega] + [b_0^\omega, b_1^\omega] &\rightarrow [\delta_0^1, c_2^\omega] = a_1^\omega + b_1^\omega,\ [c_0^1, c_1^\omega] = \delta_0^1 + a_0^\omega + b_0^\omega && (22)\\
[c_0^1, c_1^\omega] = a_1^\omega + b_1^\omega &\rightarrow c_0^1 = \lfloor (a_1^\omega + b_1^\omega)/2^\omega \rfloor,\ c_1^\omega = (a_1^\omega + b_1^\omega) \bmod 2^\omega && (23)\\
[c_0^\omega, c_1^\omega] = [a_0^1, a_1^\omega, a_2^\omega] \bmod [q_0^\omega, q_1^\omega] &\rightarrow \delta_0^1 = [q_0^\omega, q_1^\omega] < [a_1^\omega, a_2^\omega],\\
&\qquad \delta_1^1 = (0 < a_0^1) \lor ((a_0^1 =_? 0) \land \delta_0^1),\\
&\qquad [b_0^\omega, b_1^\omega] = [a_1^\omega, a_2^\omega] - [q_0^\omega, q_1^\omega], && (24)\\
&\qquad [c_0^\omega, c_1^\omega] = \begin{cases} [b_0^\omega, b_1^\omega], & \text{if } \delta_1^1 =_? 1,\\ [a_1^\omega, a_2^\omega], & \text{otherwise}\end{cases}\\
[c_0^\omega, c_1^\omega] = [a_0^\omega, a_1^\omega] - [b_0^\omega, b_1^\omega] &\rightarrow c_1^\omega = a_1^\omega - b_1^\omega,\ \delta_0^1 = a_1^\omega < b_1^\omega,\ c_0^\omega = a_0^\omega - b_0^\omega - \delta_0^1 && (25)\\
\delta_0^1 = [a_0^\omega, a_1^\omega] < [b_0^\omega, b_1^\omega] &\rightarrow \delta_0^1 = (a_0^\omega < b_0^\omega) \lor ((a_0^\omega =_? b_0^\omega) \land (a_1^\omega < b_1^\omega)) && (26)\\
\delta_0^1 = [a_0^\omega, a_1^\omega] =_? [b_0^\omega, b_1^\omega] &\rightarrow (a_0^\omega =_? b_0^\omega) \land (a_1^\omega =_? b_1^\omega) && (27)\\
[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [a_0^\omega, a_1^\omega] \cdot [b_0^\omega, b_1^\omega] &\rightarrow [d_0^\omega, d_1^\omega] = a_1^\omega \cdot b_1^\omega,\ [e_0^\omega, e_1^\omega] = a_0^\omega \cdot b_0^\omega,\\
&\qquad [f_0^\omega, f_1^\omega] = a_0^\omega \cdot b_1^\omega,\ [g_0^\omega, g_1^\omega] = a_1^\omega \cdot b_0^\omega, && (28)\\
&\qquad [h_0^1, h_1^\omega, h_2^\omega] = [f_0^\omega, f_1^\omega] + [g_0^\omega, g_1^\omega],\\
&\qquad [c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [e_0^\omega, e_1^\omega, d_0^\omega, d_1^\omega] + [h_0^1, h_1^\omega, h_2^\omega, 0]\\
[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [a_{0\text{-}3}^\omega] + [b_{0\text{-}3}^\omega] &\rightarrow [\delta_0^1, c_3^\omega] = a_3^\omega + b_3^\omega,\ [\delta_1^1, c_2^\omega] = a_2^\omega + b_2^\omega + \delta_0^1, && (29)\\
&\qquad [\delta_2^1, c_1^\omega] = a_1^\omega + b_1^\omega + \delta_1^1,\ [0, c_0^\omega] = a_0^\omega + b_0^\omega + \delta_2^1
\end{aligned}
$$

Double-Abstract-Word → Single-Abstract-Word

MoMA core rewrite rules ($[x_0, x_1, \ldots, x_{k-1}]_{2^{\omega_0}} = [x_0^{\omega_0}, x_1^{\omega_0}, \ldots, x_{k-1}^{\omega_0}]$)

Example: Rewriting Modular Addition

Double-Abstract-Word Modular Addition

$$c^{2\omega} = (a^{2\omega} + b^{2\omega}) \bmod q^{2\omega}$$

$\downarrow$ Applying rule (19) – (26)

$$[\delta_0^1, d_2^\omega] = a_1^\omega + b_1^\omega,$$

$$[d_0^1, d_1^\omega] = \delta_0^1 + a_0^\omega + b_0^\omega,$$

$$\delta_0^1 = (q_0^\omega < d_1^\omega) \vee ((q_0^\omega =_? d_1^\omega) \wedge (q_1^\omega < d_2^\omega)),$$

$$\delta_1^1 = (0 < d_0^1) \vee ((d_0^1 =_? 0) \wedge \delta_0^1),$$

$$f_1^\omega = d_2^\omega - q_2^\omega, \delta_0^1 = d_2^\omega < q_2^\omega, f_0^\omega = d_1^\omega - q_1^\omega - \delta_0^1,$$

$$[c_0^\omega, c_1^\omega] = \begin{cases} [f_0^\omega, f_1^\omega], & \text{if } \delta_1^1 =_? 1, \\ [d_1^\omega, d_2^\omega], & \text{otherwise.} \end{cases}$$

Double-Machine-Word Modular Addition

```c
// addition: quad = double + double
void _dadd(i64 *c0, i64 *c1, i64 *c2, i64 *c3,
           i64 a0, i64 a1, i64 b0, i64 b1) {
    i128 s; int cr; s = (i128) a1 + (i128) b1;
    *c3 = (i64) s; cr = s >> 64;
    s = (i128) a0 + (i128) b0 + (i128) cr;
    *c2 = (i64) s; *c1 = s >> 64; *c0 = 0; }

// subtraction
void _dsub(i64 *c0, i64 *c1, i64 a0, i64 a1,
           i64 b0, i64 b1) {
    int br; *c1 = a1 - b1; br = a1 < b1;
    *c0 = a0 - b0 - br; }

// less than
void _dlt(int *c, i64 a0, i64 a1, i64 b0, i64 b1) {
    int i0, i1, i2, i3; i0 = (a0 < b0);
    i1 = (a0 == b0); i2 = (a1 < b1);
    i3 = i1 && i2; *c = i0 || i3; }

// modular addition
void _daddmod(i64 *c0, i64 *c1, i64 a0, i64 a1,
              i64 b0, i64 b1, i64 q0, i64 q1) {
    i64 t0, t1, t2, t3, t4, t5; int i;
    _dadd(&t0, &t1, &t2, &t3, a0, a1, b0, b1);
    _dlt(&i, q0, q1, t2, t3);
    _dsub(&t4, &t5, t2, t3, q0, q1);
    *c0 = i ? t4 : t2; *c1 = i ? t5 : t3; }
```

Optimization for Non-power-Of-Two Input Bit-Widths

$$x = \left[0, \dots, 0, x_0^{\omega_0}, x_1^{\omega_0}, \dots, x_{k-1}^{\omega_0}\right]$$

- For each operation, apply **double-word modular arithmetic** to break it down to computations with bit-width $\lambda/2$
  - Apply **copy propagation**, **dead code elimination**, **strength reduction**, etc.

$$[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [\,0,\, a_1^\omega] \cdot [b_0^\omega, b_1^\omega] \quad \rightarrow \quad \begin{aligned} &[d_0^\omega, d_1^\omega] = a_1^\omega \cdot b_1^\omega,\ [e_0^\omega, e_1^\omega] = a_0^\omega \cdot b_0^\omega, \\ &[f_0^\omega, f_1^\omega] = a_0^\omega \cdot b_1^\omega,\ [g_0^\omega, g_1^\omega] = a_1^\omega \cdot b_0^\omega, \\ &[h_0^1, h_1^\omega, h_2^\omega] = [f_0^\omega, f_1^\omega] + [g_0^\omega, g_1^\omega], \\ &[c_0^\omega, c_1^\omega, c_2^\omega, c_3^\omega] = [e_0^\omega, e_1^\omega, d_0^\omega, d_1^\omega] + [h_0^1, h_1^\omega, h_2^\omega, 0] \end{aligned} \tag{28}$$

Implementing MoMA in Spiral

Software/Hardware Generation for Performance

Spiral
Software/Hardware Generation for Performance

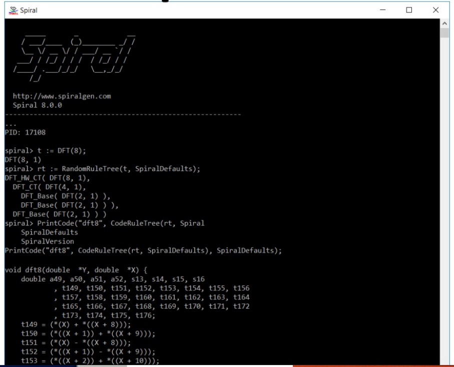
Carnegie Mellon

SpiralGen

SPIRAL 8.5.0: Available Under Open Source

- **Open Source SPIRAL available**
  - non-viral license (BSD)
  - Initial version, effort ongoing to open source whole system
  - Commercial support via SpiralGen, Inc.
- **Developed over 20 years**
  - Funding: DARPA (OPAL, DESA, HACMS, PERFECT, BRASS, PAPPA), NSF, ONR, DoD HPC, JPL, DOE (ECP, XStack, SciDAC),
  - SRC, CMU SEI, Intel, VMWare, Nvidia, Mercury
  - Open sourced under DARPA PERFECT

www.spiral.net



F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, J. M. F. Moura:
SPIRAL: Extreme Performance Portability, **Proceedings of the IEEE, Vol. 106, No. 11, 2018.**
Special Issue on *From High Level Specification to High Performance Code*

Slide borrowed from Franz Franchetti

SPIRAL-Generated MoMA-Based NTT

```
/*
 * This code was generated by Spiral 8.5.1, www.spiral.net
 */

#include <stdint.h>
__device__ uint64_t P2[1048576];
__device__ uint64_t P1[1048576];

__device__ void MPMulQDD_L3(uint64_t &t6747, uint64_t &t6748, uint64_t &t6745, uint64_t
    int a25235, a25237, a25238, a25248, a25250, a25251, a25256, a25279,
            a25281, a25282, a25292, a25294, a25295, a25300, a25323, a25325,
            a25326, a25336, a25338, a25339, a25344, a25367, a25369, a25370,
            a25380, a25382, a25383, a25388, a25395, a25397, a25398, a25403,
            a25405, a25406, a25411, a25412, a25413, a25414, a25415, a25416,
            a25417, a25421, a25423, a25424, a25429, a25430, a25431, a25436,
            a25438, a25439, a25444, a25445, a25446, a25447, a25448, a25449,
            a25450, a25451, a25452, a25453, c578, c579, c580, c581,
            c582, c583, c584, c585, c586, c587, c588, c589,
            c590, c591, c592, c593, c594, c595, c596, c597,
            c598, c599, c600, c601, c602, c604, c605, c606,
            c607, c608, c609, c610, c611, c612, c613, c614,
            c615, c616, c617, c618, c619, c620, c621, c622,
            c623, c624, c625, c626, c627, c628, c630, c631,
            c632, c633, c634, c635, c636, c637, c638, c639,
            c640, c641, c642, c643, c644, c645, c646, c647,
            c648, c649, c650, c651, c652, c653, c654, c656,
            c657, c658, c659, c660, c661, c662, c663, c664,
            c665, c666, c667, c668, c669, c670, c671, c672,
            c673, c674, c675, c676, c677, c678, c679, c680,
            c682, c683, c684, c685, c686, c687, c688, c689,
            c690, c691, c692, c693, c694, c695, c696, c697,
            c698, c699, c700, c701, c702, c703, c704, c705,
            c706, c707, c709, c710;
    uint64_t a25236, a25239, a25249, a25252, a25257, a25280, a25283, a25293,
            a25296, a25301, a25324, a25327, a25337, a25340, a25345, a25368,
            a25371, a25381, a25384, a25389, a25396, a25399, a25404, a25407,
            a25422, a25425, a25437, a25440, t8241, t8242, t8243, t8244,
            t8245, t8246, t8247, t8248, t8249, t8250, t8251, t8252,
            t8253, t8254, t8255, t8256, t8257, t8258, t8259, t8260,
            t8261, t8262, t8263, t8264, t8265, t8266, t8267, t8268,
            t8269, t8270, t8271, t8272, t8273, t8274, t8275, t8276,
```

```
...

uint128_t s1955, s1956, s1957, s1958, s1959, s1960, s1961, s1962,
        s1963, s1964, s1965, s1966, s1967, s1968, s1969, s1970,
        s1971, s1972, s1973, s1974, s1975, s1976, s1977, s1978,
        s1979;
for(int i15 = 0; i15 <= 63; i15++) {
    a27652 = (128*i15);
    a27653 = (a27652 + threadIdx.x);
    b1376 = (threadIdx.x + a27652);
    a27654 = (b1376 + 8192);
    a27655 = (a27654 % 128);
    a27656 = (128 + a27655);
    /* Begin of MPModMul 256 */
    a27657 = (2*a27656);
    a27658 = (a27657 + 1);
    a27659 = (2*a27654);
    a27660 = (a27659 + 1);
    /* MPAssignDD 128 */
    /* MPTypeCastDI 64 */
    a27661 = (2*a27657);
    a27662 = (a27661 + 1);
    /* MPAssignDD 64 */
    a27663 = (2*a27662);
    t10011 = twiddles[a27663];
    a27664 = (a27663 + 1);
    t10012 = twiddles[a27664];
    /* MPAssignDD 128 */
    a27665 = (2*a27658);
    /* MPAssignDD 64 */
    a27666 = (2*a27665);
    t10013 = twiddles[a27666];
    a27667 = (a27666 + 1);
    t10014 = twiddles[a27667];
    a27668 = (a27665 + 1);
    /* MPAssignDD 64 */
    a27669 = (2*a27668);
    t10015 = twiddles[a27669];
    a27670 = (a27669 + 1);
    t10016 = twiddles[a27670];
```

```
...

        /* MPCondD 128 */
        a29432 = (2*a29431);
        /* MPCondD 64 */
        a29433 = (2*a29432);
        Y[a29433] = ((i497) ? (t10795) : (t10788));
        a29434 = (a29433 + 1);
        Y[a29434] = ((i497) ? (t10794) : (d2111));
        a29435 = (a29432 + 1);
        /* MPCondD 64 */
        a29436 = (2*a29435);
        Y[a29436] = ((i497) ? (t10791) : (d2107));
        a29437 = (a29436 + 1);
        Y[a29437] = ((i497) ? (t10790) : (d2105));
        /* End of MPModSub 256 */
    }
}

void nttmpcuda(uint64_t  *Y, uint64_t  *X, uint64_t modulus
    dim3 b68(128, 1, 1), b69(128, 1, 1), b70(128, 1, 1), b7
    b76(128, 1, 1), b77(128, 1, 1), b78(128, 1, 1), b79(128
    g11(2, 1, 1), g12(2, 1, 1), g13(2, 1, 1), g14(2, 1, 1),
    g6(2, 1, 1), g7(2, 1, 1), g8(2, 1, 1), g9(2, 1, 1);
    ker_code0<<<g1, b68>>>(X, Y, modulus, twiddles, mu);
    ker_code1<<<g2, b69>>>(X, Y, modulus, twiddles, mu);
    ker_code2<<<g3, b70>>>(X, Y, modulus, twiddles, mu);
    ker_code3<<<g4, b71>>>(X, Y, modulus, twiddles, mu);
    ker_code4<<<g5, b72>>>(X, Y, modulus, twiddles, mu);
    ker_code5<<<g6, b73>>>(X, Y, modulus, twiddles, mu);
    ker_code6<<<g7, b74>>>(X, Y, modulus, twiddles, mu);
    ker_code7<<<g8, b75>>>(X, Y, modulus, twiddles, mu);
    ker_code8<<<g9, b76>>>(X, Y, modulus, twiddles, mu);
    ker_code9<<<g10, b77>>>(X, Y, modulus, twiddles, mu);
    ker_code10<<<g11, b78>>>(X, Y, modulus, twiddles, mu);
    ker_code11<<<g12, b79>>>(X, Y, modulus, twiddles, mu);
    ker_code12<<<g13, b80>>>(X, Y, modulus, twiddles, mu);
    ker_code13<<<g14, b81>>>(X, Y, modulus, twiddles, mu);
}

void destroy_nttmpcuda() {
    /* skip */
}
```

2^{14}-point 384-bit CUDA NTT, >**15,000** lines of code omitted

Why GPU?

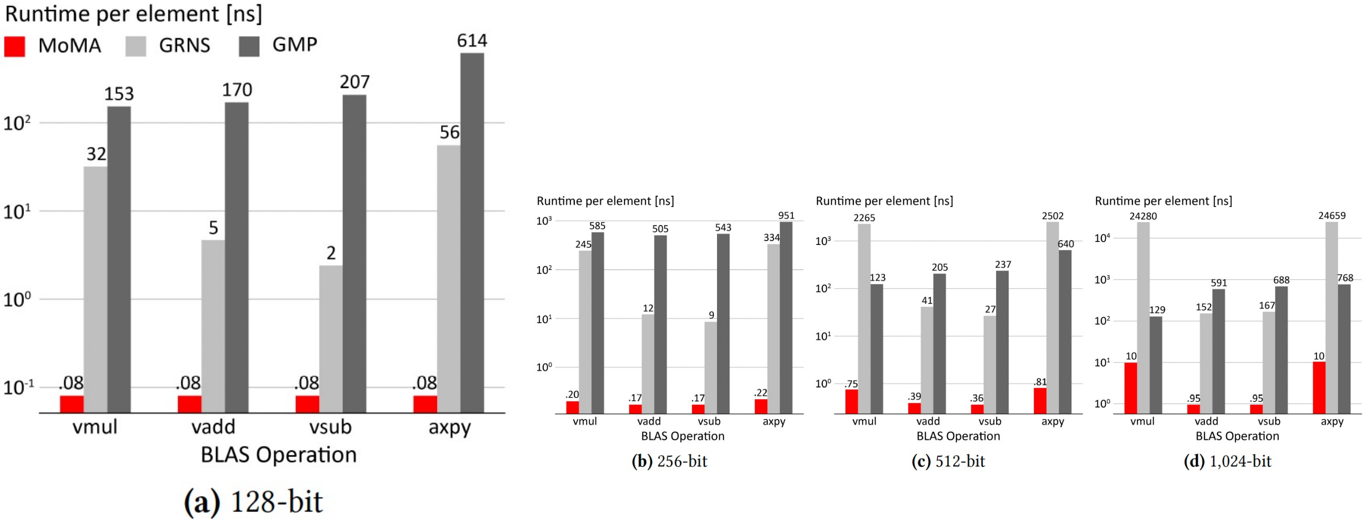
- Operations on large input bit-width become highly computationally intensive
  - Massive parallelism
  - High on-chip performance



Model	H100	RTX 4090	V100
#Cores	16896	16384	5120
Max Freq.	1980 MHz	2595 MHz	1530 MHz
RAM Size	80 GB	24 GB	32 GB
Bus Type	HBM3	GDDR6X	HBM2
Toolkit	12.2	12.0	11.7

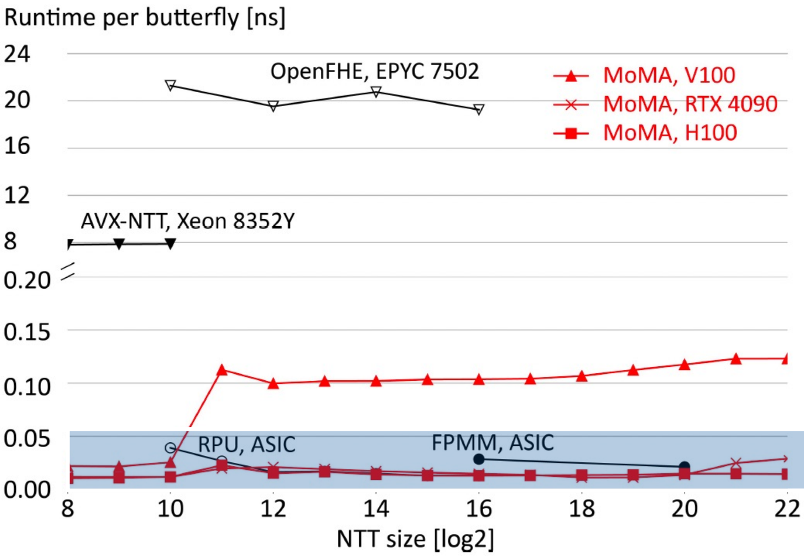
NVIDIA GPUs from different generations and price points

BLAS Operations Results

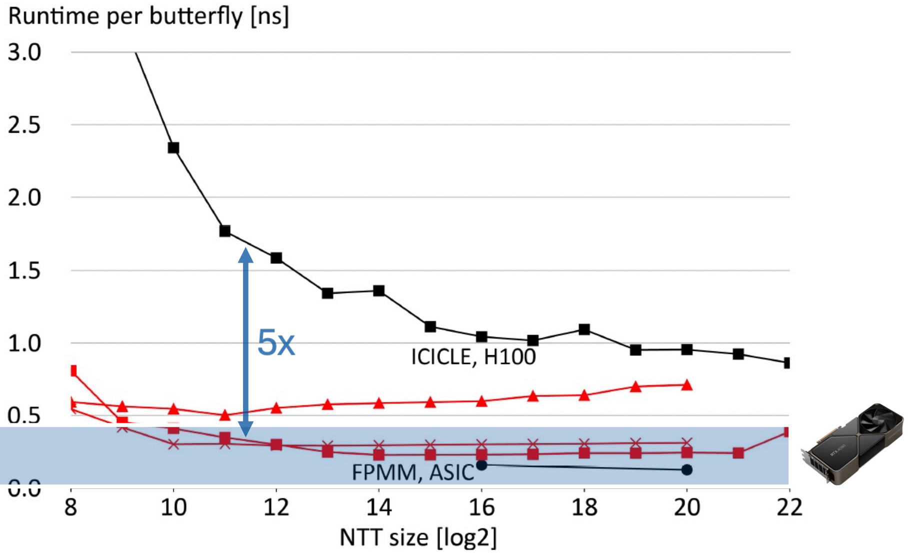


Performance of BLAS operations with various input bit-widths
on CPU (GMP) and GPU (MoMA & GRNS)

NTT Results



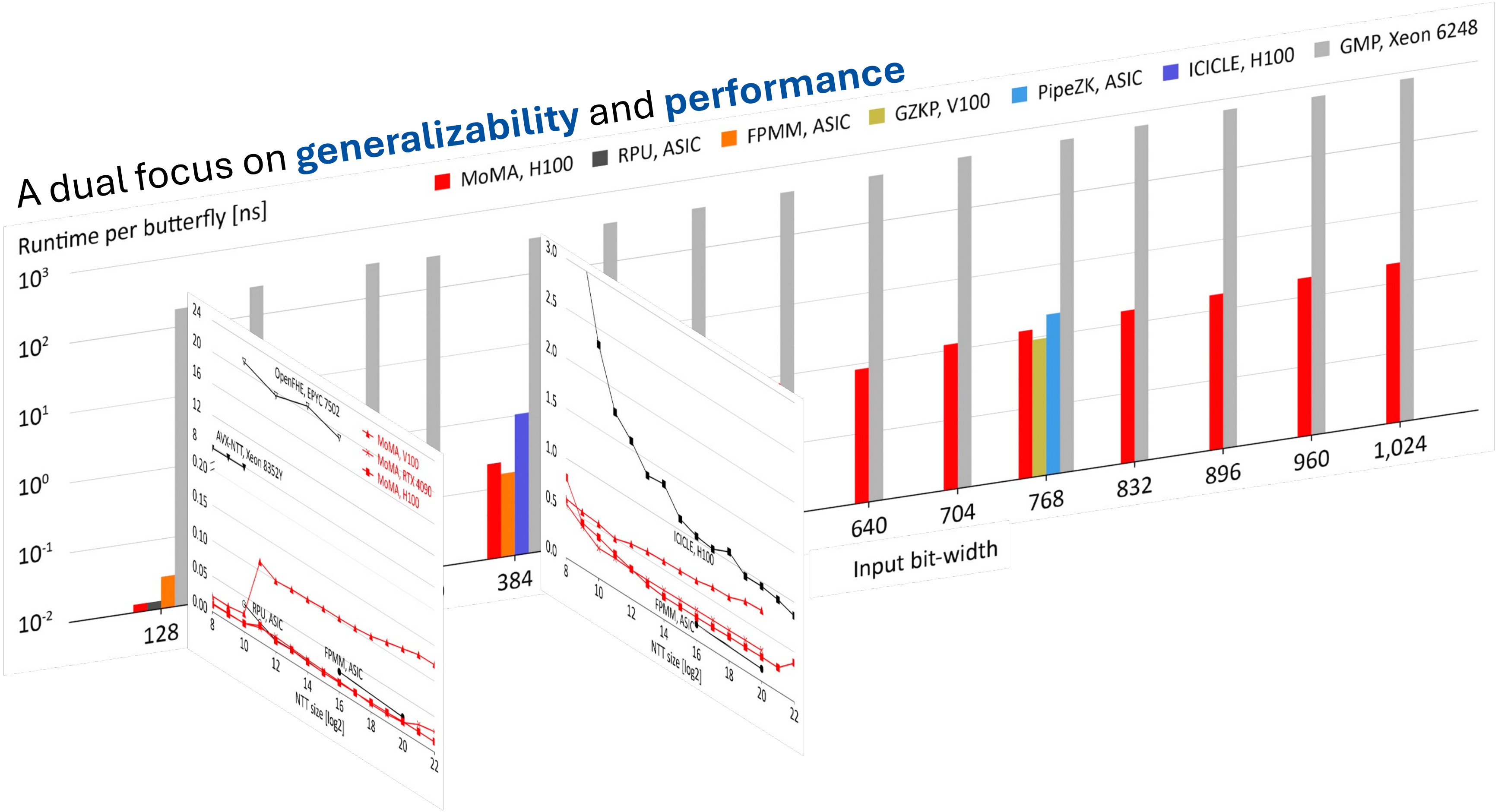
(a) 128-bit



(c) 384-bit

Performance of NTT with various input bit-widths on CPUs, GPUs and ASICs

A dual focus on generalizability and performance



Good Luck!

- Publicly available at github.com/naifeng/moma
- Reach me at naifengz@cmu.edu

```
   _____       _            __
  / ___/____  (_)________ _/ /
  \__ \/ __ \/ / ___/ __ `/ /
 ___/ / /_/ / / /  / /_/ / /
/____/ .___/_/_/   \__,_/_/
    /_/

  http://www.spiral.net
  Spiral 8.5.0
----------------------------------------------------------

PID: 36679

spiral>
```