# Parallel Arbitrary-precision Integer Arithmetic

on GPUs and Multi-core CPUs

Davood Mohajerani

ORCCA, University of Western Ontario, Canada

Ph.D. Thesis Presentation

March 19, 2021

## Outline

## Outline

## Problem definition

### Arbitrary-precision integer arithmetic:

Compute-intensive problems in computational algebra driven by applications in solving systems of polynomial equations and cryptography.

### Use cases:

- When high precision is required (with large input values that fit into multiple machine words), or
- in order to avoid coefficient overflow due to intermediate expression swell.

### What are the main difficulties with arbitrary-precision arithmetic?

- Harder to implement and parallelize.
- Far more challenging to sharply control the hardware resources.

# The case for parallel arbitrary-precision integer arithmetic

## The main drivers behind the development of parallel computing technology

- the growing demand for faster computation, and
- the recent advances in the hardware technology.

## Examples of currently available parallel computing infrastructure

- Hardware: many-core and multi-core processors.
- Software: programming models, and language extensions (e.g. CUDA, `OpenCL`, and `OpenACC` for GPUs, and `OpenMP` and `Cilk` for multi-core CPUs).

## Utilizing the parallel processors for arbitrary-precision arithmetic

- Viable targets for carrying out arbitrary-precision integer arithmetic.
- GPUs are specially interesting due to fine-grained control of resources.

## Two main challenges:

- developing parallel algorithms, and
- implementing and optimizing them as efficient parallel programs.

## Thesis statement

- We explain the current state of research on parallel arbitrary-precision integer arithmetic.
- We propose a number of solutions for some of the challenging problems related to this subject.
- The solutions include parallel algorithms, complexity analysis, and notes on implementation.

### Primary objective

- An end-to-end optimization effort for better use of the hardware resources, equivalently,
- maximizing the performance by minimizing the running time throughout the entire system.

## Important notes:

### Note 1. This thesis is inspired by the following works:

- research in code generation and optimization such as SPIRAL [1] and FFTW [2, 3],
- research in auto-tuning such as ATLAS [4], and finally,
- the mathematical libraries such as GMP [5], FLINT [6], and NTL [7].

### Note 2. The emphasis on *arbitrary-precision* arithmetic:

- Arbitrary in the sense that precision is known at the beginning but can take any value.
- Not the same as *fixed multi-precision* arithmetic: specifically tuned implementation for the numbers that are larger than one machine word and the size cannot change.
- Downside of fixed-precision arithmetic: the solutions do not scale well, or not at all.

### Note 3. This entire work is restricted to the following areas:

- large integers from $\mathbb{Z}$,
- modular integers from $\mathbb{Z}/p\mathbb{Z}$ where characteristic $p$ fits into multiple machine words, and
- dense univariate polynomials with large coefficients, in $\mathbb{Z}/p\mathbb{Z}[X]$ or $\mathbb{Z}[x]$.

## Outline

## Co-authorship Statement

- This is a joint work with Liangyu Chen, Svyatoslav Covanov, and Marc Moreno Maza.
- The article has been published in ISSAC 2017.

## Prime field arithmetic and its applications

- Prime field arithmetic plays a central role in computer algebra.
- Essential to algorithms in solving systems of polynomial equations and cryptography.

### Definitions

- Field: A set on which addition, subtraction, multiplication, and division are defined.
- Finite field: A field with a finite number of elements.
- Examples: $\mathbb{C}$ is a field, however, $\mathbb{Z}$ is not a field as inverse is not defined for every element!

### Prime field $\mathbb{Z}/p\mathbb{Z} = \{0, 1, \ldots, p-1\}$

- The sum, the difference, and the product are computed, then, reduced modulo $p$.
- The modular inverse is computed using “extended Euclidean algorithm”.
- Example: $\mathbb{Z}_5/\mathbb{Z} = \{0, 1, 2, 3, 4\}$
  - $3 + 4 \equiv 2 \pmod 5$
  - $3 - 4 \equiv -1 \pmod 5 \equiv 4 \pmod 5$
  - $3 * 2 \equiv 1 \pmod 5$
  - $3^{-1} \equiv 2 \pmod 5$

# Using small prime field arithmetic for higher precision

- The prime fields used in computer algebra systems are often small: $p$ fits in one machine word.
- Increasing precision beyond the machine word size can be done via algebraic techniques:
  Example: the Chinese Remainder Theorem (CRT).

$$\begin{array}{ccc} N & \xrightarrow{\bmod p_1} & m_1 \\ N & \xrightarrow{\bmod p_2} & m_2 \\ \vdots & & \vdots \\ N & \xrightarrow{\bmod p_e} & m_e \end{array} \Leftrightarrow N = CRT(m_1, m_2, \ldots, m_e;\ p_1,\ p_2,\ \ldots, p_e);$$

- Example of CRT for $N = 23$ and primes $p_1 = 2$, $p_2 = 3$, and $p_3 = 5$:

$$\begin{array}{ccc} 23 & \xrightarrow{\bmod 2} & 1 \\ 23 & \xrightarrow{\bmod 3} & 2 \\ 23 & \xrightarrow{\bmod 5} & 3 \end{array} \Leftrightarrow 23 = CRT(1, 2, 3; 2, 3, 5);$$

- Using small primes leads to major issues in certain modular methods.
- Example: For solving systems of non-linear equations, the *unlucky primes* are to be avoided [8, 9].
- This makes using larger primes desirable.

## Large prime field arithmetic

- Consider $\mathbb{Z}/p\mathbb{Z}$ with $p$ fitting on $k$ machine words where $k$ is a power of $2$.
- The prime $p$ is a generalized Fermat number (GFN) if $p = r^k + 1$ for machine word size radix $r$.
- Arithmetic ops. over GFN fields offer considerable performance: algebraic complexity + parallelism.
- Such operations can be vectorized and efficiently implemented on GPUs.
- This leads to efficient implementation of fast Fourier transforms (FFT) on GPUs.

### The reasons for a CUDA implementation of large prime field arithmetic:

- The programming model provides opportunities for vectorization of the arithmetic ops.
- Availability of highly optimized CUDA implementation of FFTs over small prime fields which are required for our experimental comparison (developed by W. Pan [10, 11] as part of the CUMODP library[1]).

[1]see http://cumodp.org

## Contributions and main results

- Algorithms and CUDA implementation for arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$, where $p$ is sparse radix generalized Fermat prime of size 8 or 16 (64-bit) machine words.
- Algorithms and CUDA implementation for arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$, where $p$ is generalized Fermat prime of size 8 to 1024 (32-bit) machine words.
- CUDA implementation of FFT over such big prime field $\mathbb{Z}/p\mathbb{Z}$
- Theoretical and practical comparison of this big prime field FFT vs. an approach based on small prime fields and CRT.

## How does Fourier transform help solving a problem?

$$\begin{tikzcd}
\boxed{\text{Problem}} \arrow[r, "\text{Fourier transform}"] \arrow[d] & \boxed{\text{Transformed problem}} \arrow[d] \\
\boxed{\text{Solution}} & \boxed{\text{Transformed solution}} \arrow[l, "\text{Inverse Fourier transform}"']
\end{tikzcd}$$

## DFT over a prime field

- For prime field $\mathbb{Z}/p\mathbb{Z}$ with $\omega \in \mathbb{Z}/p\mathbb{Z}$ as $N$-th root of unity, the following linear map is the DFT$_N$ at $\omega$:
$$\vec{a} = (a_0, \ldots, a_{N-1})^T \xrightarrow{\Omega} \vec{b} = (b_0, \ldots, b_{N-1})^T$$
with matrix $\Omega$ defined as follows:
$$\Omega = (\omega^{jk})_{0 \le j,k \le N-1} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

- Cooley-Tukey factorization formula: For $J, K > 1$, we have:
$$\begin{aligned} \mathrm{DFT}_{JK} &= (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L_J^{JK} \\ D_{J,K} &= \bigoplus_{j=0}^{J-1} \mathrm{diag}(1, \omega^j, \ldots, \omega^{j(K-1)}) \\ L_J^{JK} &= \mathbf{x}[iJ + j] \mapsto \mathbf{x}[jK + i] (0 \le j < J,\ 0 \le i < K) \end{aligned}$$

Various fast Fourier transform (FFT) algorithms can be used for computing DFT.

## Related work (1/2)

### Previous work on Fermat number transform (FNT)

- A DFT over a GFN $\mathbb{Z}/p\mathbb{Z}$ can be seen as a generalization of the FNT (Fermat number transform).
- The paper [12] discusses the idea of using FNT for computing convolutions modulo $F = 2^b + 1$, where $b$ is a power of $2$.
- This is an effective way to avoid round-off error caused by twiddle factor multiplication in computing DFT over $\mathbb{C}$.
- The paper [13] considers *generalized Fermat Mersenne* (GFM) primes;
- GFM numbers are of the form $(q^{pn} - 1)/(q - 1)$ where, typically, $q$ is $2$ and both $p$ and $n$ are small.
- Neither Fermat primes, nor GFM primes are the same as GFN primes.

## Related work (2/2)

### Previous work on Number theoretic transform (NTT)

- NTT refers to modular DFT, i.e., with integer coefficients.
- FNT itself is a specific case of the NTT (number theoretic transform).
- The computation of a NTT can be done via various methods used for a DFT (e.g., radix-2 Cooley-Tukey).
- Computing a DFT over a GFN $\mathbb{Z}/p\mathbb{Z}$ implies additional considerations which are not taken into account in the literature on NTT or FNT computations [12, 13].

### **FNT** DFT $\subseteq$ **GFN** DFT $\subseteq$ **NTT** DFT

- In the context of GFN $\mathbb{Z}/p\mathbb{Z}$, there is a better choice than the radix-2 Cooley-Tukey.
- The method used in the present work is related to the article [14], which is derived from Fürer's algorithm [15] for the multiplication of large integers.
- The practicality of Fürer's algorithm is an open question, however, this work is inspired by it.

## The advantage of using Fürer's trick for computing DFT

### Fürer's trick:

For GFN $p = r^k + 1$, let $K = 2k$ and $N = K^e$. Assuming $\mathbb{Z}/p\mathbb{Z}$ admits $N$'th primitive root of unity $\omega$, then we can prove that multiplying any element of $\mathbb{Z}/p\mathbb{Z}$ by a power of $\eta = \omega^{N/K}$ is as cheap as an addition.

### Applying Furer's trick to computing FFT

- Using $p = r^k + 1$, with $K = 2k$, $\mathrm{DFT}_N(\omega)$ amounts to:
  $$O(N\ \log_2(N)\,k\ +\ N\ \log_k(N)\,\mathsf{M}(k)) \text{ word ops.}$$
- Without our assumption, the same DFT would run in $O(N\ \log_2(N)\,\mathsf{M}(k))$ word ops.
- Using $p = r^k + 1$ results in a speedup factor of $\log(K)$.

### How competitive is the large prime field approach via Fürer's trick?

Answer: depends on the alternative small prime field approach that uses CRT!

# A comparable computation using small primes and the CRT

- Let $p_1, \ldots, p_k$ be pairwise different machine word size prime numbers and let $m$ be their product.
- Assume $N | p_i - 1$ such that field $\mathbb{Z}/p_i\mathbb{Z}$ admits an $N$-th primitive roots of unity $\omega_i$ (for $1 \leq i \leq p_k$).
- Then, $\omega = (\omega_1, \ldots, \omega_k)$ is an $N$-th primitive root of $\mathbb{Z}/m\mathbb{Z}$
  (Indeed, the ring $\mathbb{Z}/p_1\mathbb{Z} \otimes \cdots \otimes \mathbb{Z}/p_k\mathbb{Z}$ is a direct product of fields isomorphic to $\mathbb{Z}/m\mathbb{Z}$.)

## Compute the DFT of $f \in \mathbb{Z}/m\mathbb{Z}[x]$ a polynomial of degree $N-1$ at $\omega$:

1. Compute the images $f_1 \in \mathbb{Z}/p_1\mathbb{Z}[x], \ldots, f_k \in \mathbb{Z}/p_k\mathbb{Z}[x]$ of $f$ ($O(N \times \mathsf{M}(k) \log_2(k))$ word ops.)
2. Compute the DFT of $f_i$ at $\omega_i$ in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$ ($O(k \times N \log(N))$ word ops.)
3. Combine the results using CRT so as to obtain a DFT of $f$ at $\omega$ ($O(N \times \mathsf{M}(k) \log_2(k))$ word ops.)

- In total: $O(N \log_2(N)\, k + N\, \mathsf{M}(k) \log_2(k)) \simeq O(N\, \mathsf{M}(k) \log_2(k))$ word ops.
- Recall that using GFP $p = r^k + 1$ computing $\mathrm{DFT}_N(\omega)$ amounts to $O(N\, \mathsf{M}(k) \log_k(N))$ word ops.
- These estimates yield a running-time ratio of $\log(N)/\log_2^2(k)$ between the two approaches.
- For $k$ large enough the big prime field approach may outperform the CRT-based approach.
- This analysis and experimental results indicate that the two approaches are competitive in practice.

## Computing modulo generalized Fermat numbers

### Representing elements of $\mathbb{Z}/p\mathbb{Z}$

- Let $p = r^k + 1$ where $k$ is a power of 2.
- Represent $x \in \mathbb{Z}/p\mathbb{Z}$ as $\vec{x} = (x_{k-1}, x_{k-2}, \ldots, x_0)$ such that:

$$x \equiv x_{k-1}\, r^{k-1} + x_{k-2}\, r^{k-2} + \cdots + x_0 \quad \bmod p.$$

1. either $x_{k-1} = r$ and $x_{k-2} = \cdots = x_1 = 0$, or
2. $0 \le x_i < r$ for all $i = 0 \cdots (k-1)$

# Arithmetic and FFT over $\mathbb{Z}/p\mathbb{Z}$

## Addition and subtraction in $\mathbb{Z}/p\mathbb{Z}$

- Schoolbook addition/subtraction with carry for $x, y \in \mathbb{Z}/p\mathbb{Z}$.

## Multiplication in $\mathbb{Z}/p\mathbb{Z}$

- Associate $x, y \in \mathbb{Z}/p\mathbb{Z}$ with polynomials $f_x, f_y \in \mathbb{Z}[T]$.
- For large $k$, $f_x f_y \mod T^k + 1$ can be computed in $\mathbb{Z}[T]$ by asymptotically fast algorithms.
- For small $k$ (e.g., $k \leq 16$) using plain multiplication is reasonable.

## Multiplication by $r^i$ for some $0 < i < 2k$ in $\mathbb{Z}/p\mathbb{Z}$ reduces to a negacyclic shift:

- Recall that $r^{2k} \equiv 1$, $r^k \equiv -1$, and $r^{k+i} \equiv -r^i$, for $0 < i < k$
- Assume that $0 < i < k$ holds and $0 \leq x < r^k$ holds in $\mathbb{Z}$, then:

$$\begin{aligned} x\, r^i &\equiv x_{k-1}\, r^{k-1+i} + \cdots + x_0\, r^i \mod p \\ &\equiv \sum_{j=0}^{j=k-1} x_j r^{j+i} \mod p \equiv \sum_{h=i}^{h=k-1+i} x_{h-i} r^h \mod p \\ &\equiv \sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k} \mod p \end{aligned}$$

## Implementation notes (1/4)

### Choice of platform

- We use NVIDIA CUDA C, a platform for writing scalable parallel programs on GPUs
- Our 64-bit code is tuned for SRGFN's $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$.
- Our 32-bit code covers GFN primes of size between 8 and 1024 machine words.

### Choice of algorithm:

- Block parallelism can be realized by $I_J \otimes \mathrm{DFT}_K$.
- The CT algorithm cannot exploit block parallelism on GPUs!
- Instead, we use the six-step recursive FFT algorithm
$$\mathrm{DFT}_N = L_K^N (I_J \otimes \mathrm{DFT}_K) L_J^N D_{K,J} (I_K \otimes \mathrm{DFT}_J) L_K^N.$$
- We expand $I_K \otimes \mathrm{DFT}_J$ to turn all DFT computations to base-case $\mathrm{DFT}_K$.

### Granularity (i.e., the strategy for distribution of work among threads)

Each arithmetic operation is computed by one thread.

## Implementation notes (2/4)

### Memory-bound kernels + Very low data reuse (read, compute once, write)

- Performance is limited by frequent accesses to memory.
- Important to maximize occupancy (no. of active warps on each multiprocessor) to hide latency.
- Using shared-memory negatively impacts the occupancy, so we avoid it.
- Better to keep all the data on global memory!

### Decomposing computation into multiple kernels

- Using too many registers per thread can lower the occupancy, or can even lead to register spilling.
- Solution: register-intensive kernels are broken into multiple smaller ones.

### Size of thread blocks

- Kernels are not restricted to a fixed thread block size.
- We choose size of thread blocks for maximizing:
  - the occupancy,
  - the IPC (instruction per clock cycle), and
  - bandwidth-related performance metrics (read and write throughput).

## Implementation notes (3/4)

### The inherent suboptimality of the 64-bit implementation

- The 64-bit integer instruction on NVIDIA GPUs are not native, instead, they are emulated [16].
- This conversion heavily impacts the the performance, specially, 64-bit multiplication!
- Serious consequences: lower memory efficiency, ILP, IPC, occupancy, and higher register pressure.
- We have reimplemented the entire project based on 32-bit integer arithmetic in CUDA.

### Notes on 32-bit implementation

**The radix $r$ in $p = r^k + 1$ was supposed to be sparse with only 2 bits set.**

- This limited the number of suitable GFN primes for the implementation.
- This restriction was due to some initial assumptions that now have been relaxed.

**The implementation was limited to 8 and 16 machine word primes.**

- The new implementation covers GFN primes of size between 8 and 1024 machine words.

## Implementation notes (4/4)

### Maximizing global memory efficiency

- Assume that for a vector of $\vec{X}$ of $N$ elements of $\mathbb{Z}/p\mathbb{Z}$, consecutive digits of each element are stored in adjacent memory addresses.
- Such a vector can be considered as the row-major layout of a $N \times k$ matrix.
- This data structure will hurt performance due to increased overhead caused by non-coalesced accesses.
- An effective solution is to perform a stride permutation $L_k^{kN}$ on all input vectors
- This results in fully-coalesced accesses to global memory, increasing memory load/store efficiency, and lowering the memory overhead.

$$\begin{bmatrix} \vec{X}_0 \\ \vec{X}_1 \\ \vdots \\ \vec{X}_{N-1} \end{bmatrix} = \begin{bmatrix} \vec{X}_{(0,0)} & \cdots & \vec{X}_{(0,k-1)} \\ \vec{X}_{(1,0)} & \cdots & \vec{X}_{(1,k-1)} \\ \vdots & & \vdots \\ \vec{X}_{(N-1,0)} & \cdots & \vec{X}_{(N-1,k-1)} \end{bmatrix}_{(N\times k)} \xrightarrow{L_k^{kN}} \begin{bmatrix} \vec{X}_{(0,0)} & \cdots & \vec{X}_{(N-1,0)} \\ \vec{X}_{(0,1)} & \cdots & \vec{X}_{(N-1,1)} \\ \vdots & & \vdots \\ \vec{X}_{(0,k-1)} & \cdots & \vec{X}_{(N-1,k-1)} \end{bmatrix}_{(k\times N)}$$

## Example: Profiling $\mathrm{DFT}_{K^4}$ with $P_3$ $(K = 16)$ and $P_4$ $(K = 32)$

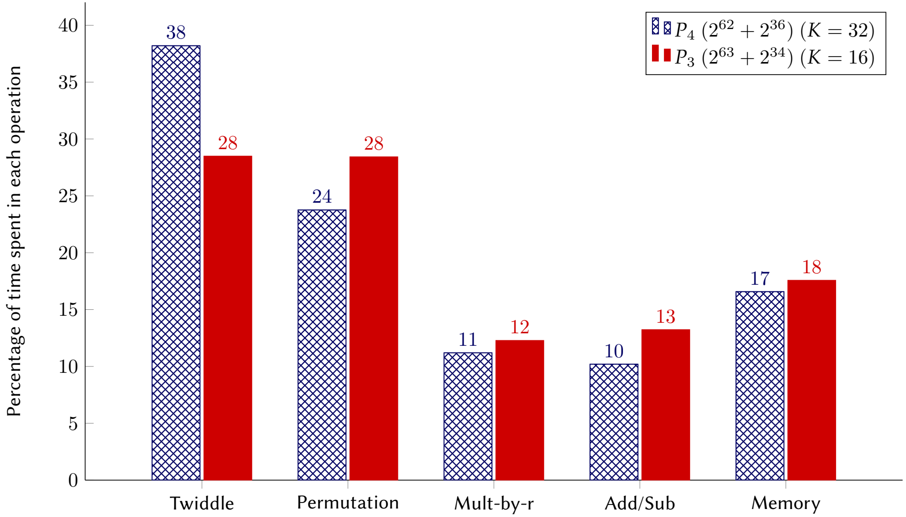


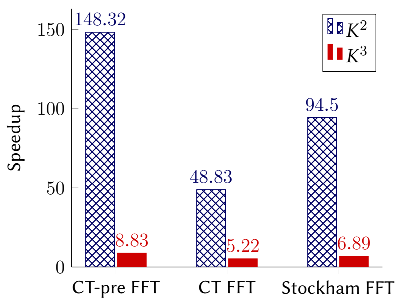
Figure: Collected on NVIDA GeforeGTX760M.

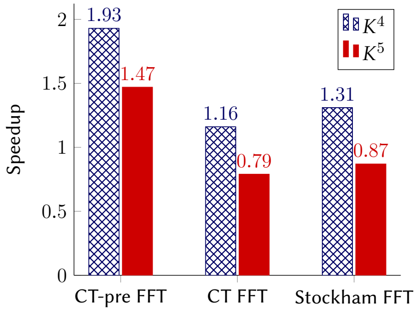
# Experimental results

## Computing FFT: big prime vs. small prime

- Small prime approach: pairwise different primes $p_1, \ldots, p_k$
  1. compute image $f_i$ of $f$ in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots, \mathbb{Z}/p_k\mathbb{Z}[x]$ (*projection*)
  2. compute $\mathrm{DFT}_N(f_i)$ at $\omega_i$ in $\mathbb{Z}/p_i\mathbb{Z}[x]$
  3. combine the results using the CRT (*recombination*)
- The small primes of CUMODP are 32-bit machine words, hence, it is fair to use $2k$ of them!
- Pick 16 and 32 small primes for comparing against $P_3 = (2^{63} + 2^{34})^8 + 1$ and $P_4 = (2^{62} + 2^{36})^{16} + 1$.
- Small prime FFTs from the CUMODP library compute $\mathrm{DFT}_{2^n}$ for $8 \le n \le 26$:
  - the Cooley-Tukey FFT,
  - the Cooley-Tukey FFT with pre-computed powers of $\omega$, and
  - the Stockham FFT.
- Tests completed on a NVIDA GeforeGTX760M card.

## Benchmarking for $P_3 = (2^{63} + 2^{34})^8$ $(K = 16)$

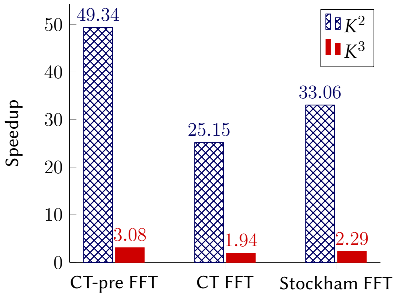


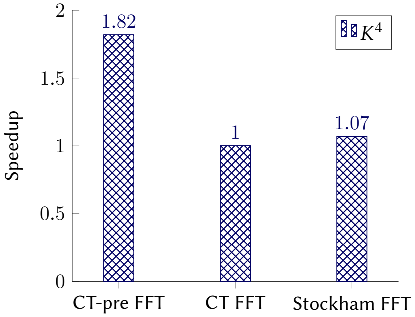


### Performance analysis:

- The CUMODP FFTs are fast for $N \geq 2^{16}$ (high occupancy)
- Larger $p$ in $\mathbb{Z}/p\mathbb{Z} \Rightarrow$ a higher number of cheap multiplications
- For $p = r^8 + 1$ and $K = 2k = 16$, the best results for $\mathrm{DFT}_{N=K^4}$
- Beyond that point, we have more expensive multiplications!

## Benchmarking for $P_4 = (2^{62} + 2^{36})^{16}$ $(K = 32)$





## Concluding remarks

- Big prime field arithmetic is required by advanced algorithms in computer algebra.
- Arithmetic modulo a big prime can be efficiently computed on GPUs.
- Computing FFT in $\mathbb{Z}/p\mathbb{Z}$ is competitive with a CRT-based approach for a range of sizes.
- Multiplication in $\mathbb{Z}/p\mathbb{Z}$ (except for the case of a multiplication by a power of $r$) remains a bottleneck.

### Challenge: Efficient multiplication of two elements of the big prime field

This was the motivation for efforts on two fronts:

- Porting the project to multi-core CPUs, plus, experimenting with FFT-based integer multiplication.
- Finding a parallel solution for multiplying large integers not suitable for FFT-based multiplication.

### Future work

We need to reevaluate the performance of the 32-bit implementation.

## Outline

## Co-authorship Statement

- This is a joint work with Svyatoslav Covanov, Marc Moreno Maza, and Linxiao Wang.
- The article has been published in ISSAC 2019.

## Differences between multi-core CPUs and GPUs

- Once again, consider prime fields $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1$ fitting on $k$ machine words ($k$ a power of $2$).
- Can we utilize the commonly used multi-core processors of today's laptops and desktops for parallel big prime field arithmetic?

### Can we port the GPU implementation to a multi-core platform?

- Answer: NO! The implementation techniques developed in big prime field FFT on GPUs [17] can not be easily ported and applied to the context of multi-core processors.
- In contrast with GPUs, the architectures are fundamentally not suitable for fine-grained parallelism.
- There are significant differences in memory hierarchies and inter-thread communication mechanisms.
- GPU architectures offer programmers a finer control of hardware resources than multi-core processors and thus more opportunities to reach high performance.

## Efficient sequential FFT on CPUs

### Can we develop efficient serial implementations of FFT over $\mathbb{Z}/p\mathbb{Z}$ on CPUs?

Answer: YES! provided that we go through the following route:

- Our implementation is written in C and inline assembly.
- We prioritize cache friendliness over performing many batches of arithmetic operations.
- We focus on optimizing the multiplication between two arbitrary elements $x, y$ of $\mathbb{Z}/p\mathbb{Z}$ via FFT.

### The importance of using FFT-based multiplication

- Note that we encode elements of $\mathbb{Z}/p\mathbb{Z}$ in radix $r$ expansion.
- For $p = r^k + 1$, each element of $\mathbb{Z}/p\mathbb{Z}$ can be seen as a polynomial of degree less than $k$.
- Thus, multiplying two arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$ requires computing the product of two univariate polynomials (of degree less than $k$) in $\mathbb{Z}[X]$ modulo $X^k + 1$.
- In our GPU implementation this is done by using plain multiplication, thus $\Theta(k^2)$ machine word ops.

## DFT-based polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$

### Cyclic convolution:

$$\begin{tikzcd}
f(x) = \sum_{i=0}^{n-1} a_i x_i \arrow[d] \arrow[r] & \vec{a} \arrow[r, "\mathrm{DFT}_n(\omega)"] & \mathrm{DFT}(\vec{a}) \arrow[d] \\
g(x) = \sum_{i=0}^{n-1} b_i x_i \arrow[r] & \vec{b} \arrow[r, "\mathrm{DFT}_n(\omega)"] & \mathrm{DFT}(\vec{b}) \\
f(x)g(x) \equiv \sum_{i=0}^{n-1} c_i x_i \pmod{x^n - 1} \arrow[r] & \vec{c} & \mathrm{DFT}(\vec{a}) * \mathrm{DFT}(\vec{b}) \arrow[l, "\mathrm{DFT}_n^{-1}(\omega^{-1})"']
\end{tikzcd}$$

## Example: DFT-based polynomial multiplication over $\mathbb{Z}_{17}$

Need to pad the upper half of the vector with zeros.

$$\begin{aligned}
f(x) &= \underline{1x^0 + 2x^1 + 3x^2 + 4x^3} + 0x^4 + 0x^5 + 0x^6 + 0x^7 \\
g(x) &= \underline{5x^0 + 6x^1 + 7x^2 + 8x^3} + 0x^4 + 0x^5 + 0x^6 + 0x^7 \\
f(x)g(x) &\equiv \underline{5x^0 + 16x^1 + 0x^2 + 9x^3 + 10x^4 + 1x^5 + 15x^6} + 0x^7 \pmod p
\end{aligned}$$

Vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$ are the vector of cofficients for $f(x)$, $g(x)$, and $f(x)g(x)$, respectively.

$$\begin{array}{lllcll}
\vec{a} = & [\underline{1, 2, 3, 4}, 0, 0, 0, 0] & \xrightarrow{\mathrm{DFT}_8(\omega)} & \mathrm{DFT}_8(\vec{a}) & = & [10, 15, 7, 13, 15, 11, 6, 16] \\
\vec{b} = & [\underline{4, 5, 6, 7}, 0, 0, 0, 0] & \xrightarrow{\mathrm{DFT}_8(\omega)} & \mathrm{DFT}_8(\vec{b}) & = & [9, 7, 7, 7, 15, 8, 6, 15] \\
\vec{c} = & [\underline{5, 16, 0, 9, 10, 1, 15}, 0] & \xleftarrow{\mathrm{DFT}_8^{-1}(\omega^{-1})} & (\mathrm{DFT}_8(\vec{a}) * \mathrm{DFT}_8(\vec{b})) & = & [5, 3, 15, 6, 4, 3, 2, 2]
\end{array}$$

## Related work (1/2)

### Previous work on parallel implementation of finite field FFT on CPUs

- Efficient multi-threaded implementation of FFTs with coefficients in single or double precision is a standard research topic [18, 19, 20, 21].
- The case of higher precision has received little attention so far!

### Previous work on FFT-based multiplication

- With coefficients in the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$, our FFT is in the spirit of the algorithms of Schönhage and Strassen [22] and Fürer [23].
- In the algorithms of Schönhage and Strassen [22] and Fürer [23] fast multiplication is achieved by “composing” FFTs operating on different vector sizes.
- Once again, this work is inspired by the Fürer’s algorithm and is not an implementation of it.

## Related work (2/2)

### Several algorithms similar to Fürer's have been proposed:

- In [24, 25] De et al. gave a similar algorithm which relies on *finite field arithmetic* and achieves the same "theoretical" running time as Fürer's algorithm.
- In [26], Harvey, Van der Hoeven and Lecerf proposed a theoretical improvement to Fürer's algorithm based on Bluestein's chirp transform for the integer multiplication.
- In [27], Harvey, Van der Hoeven and Lecerf propose a similar algorithm for the multiplication over finite fields, achieving a Fürer-like complexity. This work led to an efficient implementation in [28], using multiplication of polynomials over the special field $\mathbb{F}_{2^{60}}$.
- In [29], Covanov and Thomé proposed an algorithm based on generalized Fermat primes and the same scheme as Fürer's algorithm, to multiply integers with a Fürer-like complexity.

## Contributions

### Code and experimental results

- We have reported the experimental results in [30], ISSAC 2019.
- Our code is part of the *Basic Polynomial Algebra Subprograms* (BPAS) library [31] [2].

### Performance of our implementation

- Optimized implementation of the generalized Fermat prime field $\mathbb{Z}/p\mathbb{Z}$ based on specialized arithmetic.
- Comparison: a naive implementation of $\mathbb{Z}/p\mathbb{Z}$ arithmetic based on modular sum ($a + b \mod p$) and modular product ($a * b \mod p$) functions from GNU Multiple Precision Arithmetic Library (GMP) [5].
- The performance of the naive implementation degrades substantially.

### Key performance metric: sharp management of computing resources

- Reduced number of arithmetic instructions due to using specialized arithmetic.
- Minimal memory usage.
- Unrolling base-case DFT's (i.e., hard coding the constants).

[2] publicly available at http://www.bpaslib.org/

## Conclusions and future work

### Conclusions

- FFT can be used effectively to improve multiplication time in big prime field.
- Using generalized Fermat prime fields can lower the average time spent in multiplications in FFT.
- The big prime field FFT can be implemented on CPU efficiently.

### Issue #1: Limitations on input sizes

- Current implementation only supports vectors of size $N = K^e$ for $K = 2k$ and $e \in \mathbb{Z}$.
- Future work: can extend the implementation to support arbitrary vector sizes ($N \neq K^e$).

### Issue #2. Inability to handle very large input sizes

- Future work: can extend the implementation to support huge vectors that cannot fit into main memory.

## Outline

## Co-authorship Statement

- This is a joint work with Marc Moreno Maza.
- It is still in progress and will be submitted to ISSAC 2022.

## Multiplication as the core arithmetic operation

- Multiplication of large integers is at the core of many other algorithms.
- We can apply the algorithms for integers to polynomials of very large degrees and vice versa.
- Researchers have been mostly focused on improving the theoretical time complexity of multiplication.
- Table 2 presents the complexity estimates of the multiplication algorithms, see [32, 33, 22, 34, 15, 23, 28, 35, 36, 37].

Table: Comparison of multiplication algorithms for multiplying polynomials $f(x), g(x) \in R[x]$ of degree less than $k$.

| Variant | Algorithm | $\mathrm{M}(k)$ |
|---|---|---|
| Non FFT-based | Classical (Toom-1) | $O(k^2)$ |
| | Karatsuba-Ofman (Toom-2), 1962 | $O(k^{\log_2^3}) = O(k^{1.58})$ |
| | Toom-Cook (Toom-3), 1963 | $O(k^{\log_3^5}) = O(k^{1.46})$ |
| FFT-based | NTT/FFT multiplication | $O(\frac{9}{2}k\log(k))$ |

Table: Comparison of multiplication algorithms for multiplying $k$ machine word integers.

| Variant | Algorithm | $\mathrm{M}(k)$ |
|---|---|---|
| FFT-based | Fürer, '09 | $O(k\log(k)\cdot 2^{O(\log^* k)})$ |
| | Harvey-van der Hoeven-Lecerf, '15 | $O(k\log(k)\cdot 2^{3\log^* k})$ |
| | Harvey-van der Hoeven, '18 (proved as an unconditional bound) | $O(k\log(k)\cdot 2^{2\log^* k})$ |
| | Covanov-Thomé, '19 | $O(k\log(k)\cdot 4^{\log^* k})$ |
| | Harvey-van der Hoeven, '19 | $O(k\log(k))$ |

## When should we consider Non FFT-based algorithms?

- FFT-based algorithms need very large input sizes to be practically efficient.
- Even when computing with FFT-based solutions with large input sizes we may still need to multiply large integers that are not as large as input values but still require multiple machine words.
- Example: The twiddle factor multiplications during computation of big prime field FFTs.

### It is hard for FFT-based algorithms to beat plain algorithms on small sizes!

- This can be explained by the fact that FFT-based are more difficult to schedule and have more complicated memory access patterns.
- This is more crucial in case of parallel implementations.

### Plain algorithms have a high degree of parallelism due to their simple structure

This makes such algorithms favorable candidates for a parallel implementation.

## Problem definition

Let $X, Y$ be two vectors of $N$ elements of ring $R$:

$$\vec{X} = (X_0, X_1, \dots, X_{N-1})$$

$$\vec{Y} = (Y_0, Y_1, \dots, Y_{N-1})$$

then, we are interested in a fast solution for pointwise multiplication of $X$ by $Y$:

$$\vec{Z} = (X_0Y_0, X_1Y_1, \dots, X_{N-1}Y_{N-1})$$

### Choices for $R$

- Either $R = \mathbb{Z}/m\mathbb{Z}$ for $m = \beta^k$ where $\beta$ is the size of a machine word, or
- $R = \mathbb{Z}/p\mathbb{Z}$ for $p = r^k + 1$ where the radix $r$ fits into a machine word.

### Assumptions

- The value of $\beta$ depends on the architecture and can be either $\beta = 2^{32}$ or $\beta = 2^{64}$.
- For this work, we assume $\beta = 2^{32}$, $k$ is a multiple of 32.

## Contributions

- We have developed a new fine-grained parallel algorithm for multiplying arbitrary-precision integers.
- Our new algorithm which we will refer to as `M5Mult` is based on classical $\mathcal{O}(k^2)$ algorithm but slower.
- It supports batch multiplication of integers on NVIDIA GPUs.
- Certainly, not suitable for a sequential implementation!
- However, it is well-structured to be parallelized due to a high degree of parallelism.

### Benchmarks: fair comparison of GPUs vs. GPUs

- The work in [38], up to our knowledge, is the only known parallelization of schoolbook multiplication.

### Benchmarks: unfair but pragmatic comparison of GPUs vs. CPUs

- We are interested to know how long will it take to multiply a batch of large integers using the sequential CPU implementation of GMP[5] compared to our GPU implementation.

## Schoolbook multiplication

**Note**: The $LHC$ normalization (carry-handling) step has been factored out of the following algorithm.

**Algorithm 1** Naive algorithm for computing the product of large $k$-digit integers $a$ and $b$.

```
1: input: Large $k$-digit integers $a$ and $b$ stored as vectors $\vec{a}, \vec{b}$.
2: output: Large $2k$-digit integer $m$ as the product of $a$ by $b$.
3: procedure NaiveMult($\vec{a}, \vec{b}, k$)
4:     $\vec{M}(M_0, M_1, \cdots, M_{2k-2}) \leftarrow (0, 0, \cdots, 0)$
5:     for $0 \leq i < k$ do
6:         for $0 \leq j < k$ do
7:             $M_{(i+j)} \leftarrow M_{(i+j)} + a_i b_j$
8:         end for
9:     end for
10: end procedure
```

## Karatsuba intermediate product

### Definition

The Karatsuba intermediate product for machine word size digits $a_i, a_j, b_i, b_j$ is defined as follows:

$$a_i b_j + a_j b_i = (a_i - a_j)(b_j - b_i) + z_i + z_j$$

where $z_i = a_i b_i$ and $z_j = a_j b_j$.

## Schoolbook multiplication + Karatsuba intermediate product (1/2)

**Note**: The $LHC$ normalization step (carry-handling) has been factored out of the following algorithm.

**Algorithm 2** Sequential algorithm for computing the product of large $k$-digit integers $a$ and $b$ while taking advantage of Karatsuba intermediate products.

```
 1: input: Large $k$-digit integers $a$ and $b$ stored as vectors $\vec{a}, \vec{b}$.
 2: output: Large $2k$-digit integer $m$ as the product of $a$ by $b$.
 3: procedure SequentialMult($\vec{a}, \vec{b}, k$)
 4:     for $0 \leq i < k$ do                         ▷ Loop I (can be executed in parallel.)
 5:         $z_i \leftarrow a_i b_i$
 6:     end for
 7:     $\vec{M}(M_0, M_1, \cdots, M_{2k-2}) \leftarrow (0, 0, \cdots, 0)$
 8:     for $0 \leq i < k$ do                         ▷ Loop II
 9:         for $i + 1 \leq j < k$ do
10:             $M_{(i+j)} \leftarrow M_{(i+j)} + (a_i - a_j)(b_j - b_i) + z_i + z_j$
11:         end for
12:     end for
13:     for $0 \leq i < k$ do                         ▷ Loop III (can be executed in parallel.)
14:         $M_{2i} \leftarrow M_{2i} + z_i$
15:     end for
16: end procedure
```

## Schoolbook multiplication + Karatsuba intermediate product (2/2)

There are three main ideas at the core Algorithm 2:

1. we precompute the values of $z_i = a_i b_i$ for $0 \leq i < k$,
2. we lower the number of multiplications to (almost half) compared to Algorithm 1, and finally,
3. for-loop nests I, II, and III are well-structured for parallel execution.

### GPU implementation

- Loops I and III are well-suited for a GPU implementation.
- Loop II needs a chunking strategy for efficient usage of the global memory!
- We divide the vector of coefficients for each of the operands $a$ and $b$ into chunks of size $s$
- For the sake of simplicity, we assume $k$ is a multiple of $s$, also, $s$ is a multiple of 32.

## M5Mult

**Algorithm 3** Chunk-based algorithm for computing the product of large $k$-digit integers $a$ and $b$ while taking advantage of Karatsuba intermediate products.

```
1: input: Large $k$-digit integers $a$ and $b$ stored as vectors $\vec{a}, \vec{b}$.
2: output: Large $2k$-digit integer $m$ as the product of $a$ by $b$.
3: procedure M5Mult($\vec{a}, \vec{b}, k$)
4:     $\vec{M}(M_0, M_1, \cdots, M_{2k-1}) \leftarrow (0, 0, \cdots, 0)$
5:     for $0 \leq i < k$ do                          ▷ Loop I can be executed in parallel.
6:         $z_i \leftarrow a_i b_i$
7:     end for
8:     $\lambda \leftarrow \frac{k}{s}$
9:     for $0 \leq b_i < \lambda$ do                  ▷ Can be executed in parallel.
10:        SingleCM($\vec{M}, \vec{A}, \vec{B}, \vec{Z}, b_i, s$);
11:    end for
12:    for $0 \leq b_i < \lambda$ do                  ▷ This loop CANNOT be executed in parallel.
13:        for $b_i + 1 \leq b_j < \lambda$ do        ▷ Can be executed in parallel.
14:            DoubleCM($\vec{M}, \vec{A}, \vec{B}, \vec{Z}, b_i, b_j, s$);
15:        end for
16:    end for
17:    for $0 \leq i < k$ do                          ▷ Loop III can be executed in parallel.
18:        $M_{2i} \leftarrow M_{2i} + z_i$
19:    end for
20: end procedure
```

## `M5Mult` is *NOT* suitable for a sequential implementation!

- Let $\eta = \frac{T_{\mathbf{M1}}}{T_{\mathbf{A1}}}$ as the ratio of cost of a single-digit multiplication to cost of a single-digit addition.
- For small values of $\eta$ the `NaiveMult` is supposed to be more efficient than `SequentialMult`.
- Neither `SequentialMult`, nor `M5Mult` are suitable for a sequential implementation!

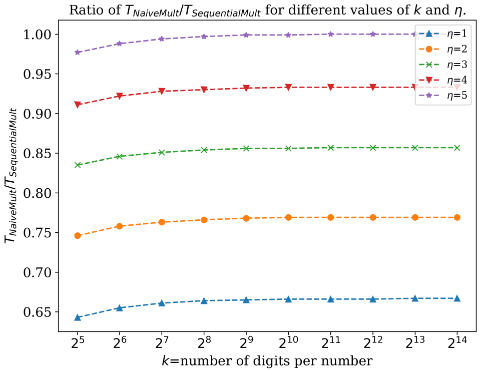


Figure: Estimated running-time ratio for various values of $k$ and $\eta$.

## `M5Mult` is suitable for a parallel implementation!

- Let $\eta = \frac{T_{\mathbf{M1}}}{T_{\mathbf{A1}}}$ as the ratio of cost of a single-digit multiplication to cost of a single-digit addition.
- `M5Mult` has a high degree of parallelism!

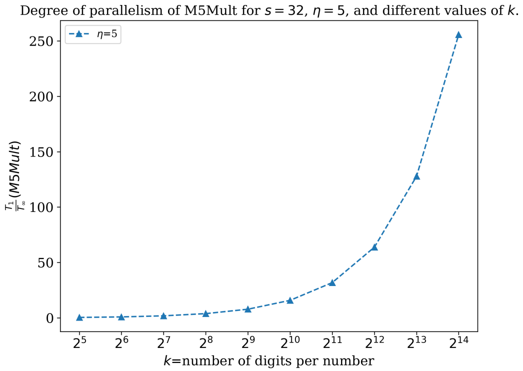
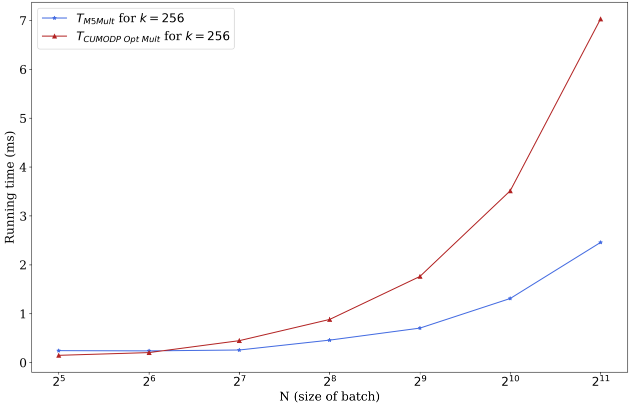


Figure: Estimated degree of parallelism for various values of $k$ with $s = 32$ and $\eta = 5$.

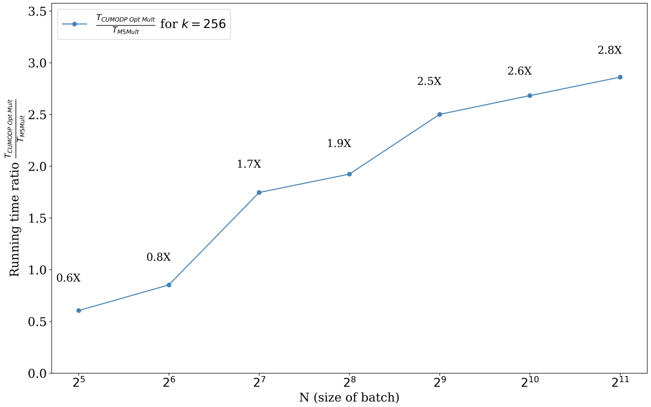
## GPU vs. GPU comparison (1/2)

- Data collected on `NVIDIA GTX 1080Ti`.



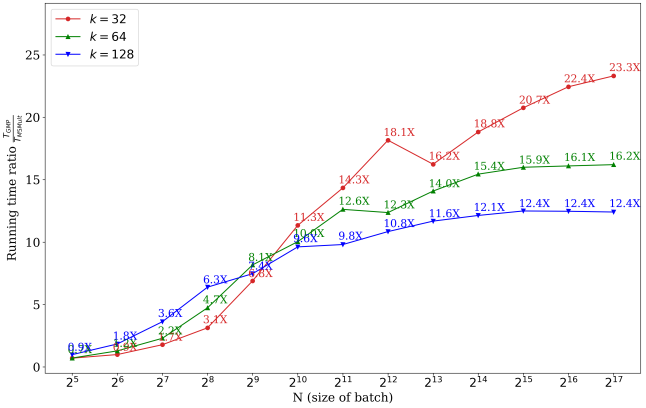
## GPU vs. GPU comparison (2/2)

- Data collected on `NVIDIA GTX 1080Ti`.



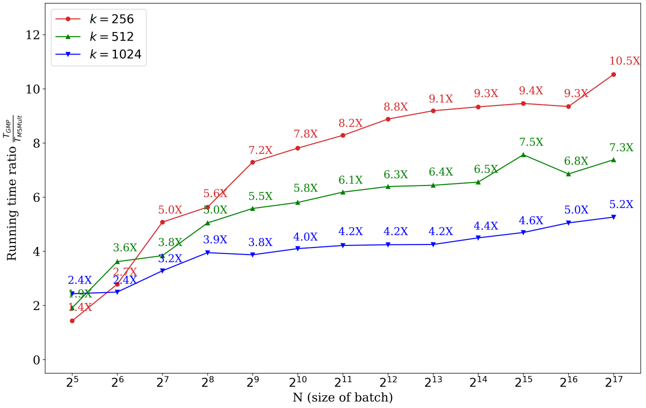
## GPU vs. CPU comparison (1/2)

- Data collected on `NVIDIA GTX 1080Ti` and `Intel-i7-7700K`.



# GPU vs. CPU comparison (2/2)

- Data collected on `NVIDIA GTX 1080Ti` and `Intel-i7-7700K`.



## Outline

## Co-authorship Statement

- This is a joint work with Alexander Brandt, Marc Moreno Maza, Jeeva Paudel, and Linxiao Wang.
- A preprint version of this work is available at CoRR (vol. abs/1911.02373, 2019).

## Motivation

### Why programming for HPC is a difficult task?

- Programmers must be conscious of many factors impacting performance.
- For example: scheduling, synchronization, and data locality.
- Other *parameters* are independent from the code and greatly affect performance.

### Three types of parameters that influence performance of parallel programs:

- *Data parameters*, such as input data and its size;
- *Hardware parameters*, such as cache capacity and number of available registers; and
- *Program parameters*, such as granularity of tasks and the quantities that characterize how tasks are mapped to processors (e.g., dimension sizes of a thread block for a CUDA kernel).

## Performance portability for CUDA programs

### How are different parameters determined?

- Data and hardware parameters: determined by the needs of the user + available hardware resources.
- Program parameters: largely influence the performance (orders of magnitude difference in timings.)
- Crucial to choose values of program parameters that yield the best performance.

### Choice of program parameters in the CUDA programming model

- The kernel launch parameters control the size and shape of thread blocks.
- Will kernel launch parameters greatly impact performance?
  Yes, if the memory accesses pattern depends on the block's dimension sizes.
- An optimal thread block format for one GPU architecture may not be optimal for another [39].
- This emphasizes the need for performance portability:
  enabling users to efficiently execute the same code on different architectures of the same platform.

## Related work (1/2)

### Analyzing the performance of parallel programs at an abstract level:

- The *Parallel Random Access Machine* (PRAM) model [40, 41].
- PRAM variants tailored to GPU code analysis such as TMM [42] and MCM [43].

### GPU performance models that estimate the execution time of GPU kernels:

Models such as MWP-CWP [44, 45] rely on the profiling information.

### Other research on improving performance of CUDA programs:

- Auto-tuning techniques:
  multiple versions generated *off-line* + applied/refined *on-line* once the runtime parameters are known.
- Auto-tuning have achieved great results in projects such as ATLAS [4], FFTW [2], and SPIRAL [46].
- Loop transformation [47], auto-tuning [48, 49, 50, 51], dynamic instrumentation [52].
- Combination of auto-tuning and dynamic instrumentation [53].

## Related work (2/2)

### Literature on impact of kernel launch parameters on performance

- [54] and [39] suggest that kernel launch parameters must be considered as a target for optimization.
- The authors of [50] use a linear regression model to predict optimal thread block configurations.
- The authors in [55] have also developed a method determining the best thread block configuration.
- The authors of latter two works assume that kernel execution time scales linearly with data size.

### Approaches based on statistical learning

- In [56], the authors present an input-adaptive GPU code optimization framework G-ADAPT, which uses statistical learning to find a relation between the input sizes and the thread block sizes. At linking time, the framework predicts the best block size for a given input size using the linear model obtained from compile time. This approach only considers the total size of the thread blocks and not their configuration.
- In [57], machine learning techniques are used in combination with auto-tuning to search for optimal configurations of OpenCL kernels, but their examples are limited to stencil computations.

## Contributions

- A technique for devising a mathematical expression in the form of a *rational program* to evaluate a performance metric from a set of program and data parameters.
- KLARAPTOR: a tool implementing the rational program technique to dynamically optimize CUDA kernels by choosing optimal launch parameters.
- An empirical and comprehensive evaluation of our tool on kernels from the `Polybench/GPU` benchmark suite.

### KLARAPTOR (Kernel LAunch parameters RAtional Program estimaTOR):

- A tool for automatically and dynamically determining the optimal CUDA kernel launch parameters
- Performance optimization based on the actual data and target device of a kernel invocation.
- Allowing for dynamic data-dependent performance and portable performance.
- Core technique: encoding a performance prediction model as a so-called *rational program.*
- The statically built rational program is then used dynamically at runtime to determine optimal program parameters for a given multithreaded program on specific data and hardware parameters.

## Rational Program: optimization via parametric estimation

### Optimizing performance metric $\mathcal{E}$ for a program $\mathcal{P}$

Given data parameters values $D_1, \ldots, D_d$, the goal is to find program parameter values $P_1, \ldots, P_p$ such that the execution of $\mathcal{P}$ optimizes $\mathcal{E}$.

### Statically building RP (at compile-time)

We compute a mathematical expression, parameterized by data and program parameters, in the format of a rational program $\mathcal{R}$.

### Dynamically using RP (at runtime)

- Given specific values of $D_1, \ldots, D_d$, we can efficiently evaluate/estimate value of $\mathcal{E}$ using $\mathcal{R}$.
- We can then determine values of $P_1, \ldots, P_p$ optimizing $\mathcal{E}$ and feed them to $\mathcal{P}$ for execution.

## What is a “Rational Program”?

- Let $X_1, \ldots, X_n, Y$ be pairwise different variables.
- Let $\mathcal{S}$ be a sequence of three-address code instructions such that $\{X_1, \ldots, X_n\}$ is the set of the variables that occur in $\mathcal{S}$ and are never assigned a value by an instruction of $\mathcal{S}$.

### Rational sequence vs. Rational program

- **Definition**: The sequence $\mathcal{S}$ is rational if every arithmetic operation used in $\mathcal{S}$ is either an addition, a subtraction, a multiplication, or a comparison ($=$, $<$), for integer numbers either in fixed precision or in arbitrary precision.
- **Definition**: The rational sequence $\mathcal{S}$ is a rational program in $X_1, \ldots, X_n$ evaluating $Y$ if after specializing each of $X_1, \ldots, X_n$ to an arbitrary integer value in $\mathcal{S}$, the execution of the specialized sequence $\mathcal{S}$ always terminates and the last executed instruction assigns an integer value to $Y$.

### Notes

- One can easily extend the above definition by allowing the use of the Euclidean division for integers.
- Also the definition can be extended for rational numbers.

## Example: Building an RP for occupancy (1/2)

### Hardware occupancy as defined in CUDA

A measure of how effectively a program is utilizing the Streaming Multiprocessors (SM's).

### Calculating occupancy from parameters:

- the maximum number $R_{\max}$ of registers per thread block (Hardware),
- the maximum number $Z_{\max}$ of shared memory words per thread block (Hardware),
- the maximum number $T_{\max}$ of threads per thread block (Hardware),
- the maximum number $B_{\max}$ of thread blocks per Streaming Multiprocessor (SM) (Hardware),
- the maximum number $W_{\max}$ of warps per SM (Hardware),
- the number $R$ of registers used per thread (Low-level metric),
- the number $Z$ of shared memory used per thread block (Low-level metric),
- the number $T$ of threads per thread block (Program).

### The occupancy of a CUDA kernel: $W_{\mathrm{active}}/W_{\max}$

The ratio between the number of active warps per SM and the maximum number of warps per SM where

$$W_{\mathrm{active}} = \min\left(\lfloor B_{\mathrm{active}} T/32 \rfloor, W_{\max}\right)$$

and $B_{\mathrm{active}}$ is given as a flow chart by Figure 4.

## Example: Building an RP for occupancy (2/2)

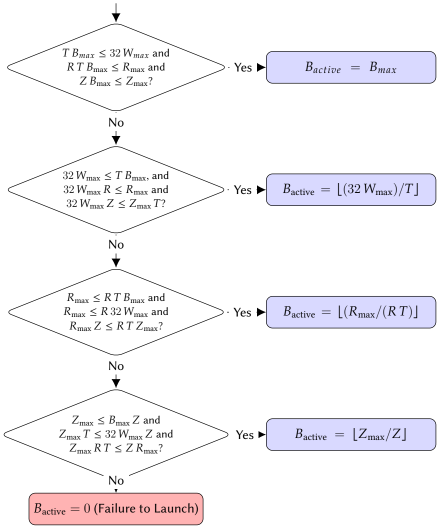

$T\,B_{max} \leq 32\,W_{max}$ and $R\,T\,B_{\max} \leq R_{\max}$ and $Z\,B_{\max} \leq Z_{\max}$?
Yes
$B_{active} = B_{max}$
No
$32\,W_{\max} \leq T\,B_{\max}$, and $32\,W_{\max}\,R \leq R_{\max}$ and $32\,W_{\max}\,Z \leq Z_{\max}\,T$?
Yes
$B_{\text{active}} = \lfloor (32\,W_{\max})/T \rfloor$
No
$R_{\max} \leq R\,T\,B_{\max}$ and $R_{\max} \leq R\,32\,W_{\max}$ and $R_{\max}\,Z \leq R\,T\,Z_{\max}$?
Yes
$B_{\text{active}} = \lfloor (R_{\max}/(R\,T) \rfloor$
No
$Z_{\max} \leq B_{\max}\,Z$ and $Z_{\max}\,T \leq 32\,W_{\max}\,Z$ and $Z_{\max}\,R\,T \leq Z\,R_{\max}$?
Yes
$B_{\text{active}} = \lfloor Z_{\max}/Z \rfloor$
No
$B_{\text{active}} = 0$ (Failure to Launch)

Figure: Rational program (presented as a flow chart) for the calculation of hardware occupancy in CUDA.

## Six steps for building a rational program $\mathcal{R}$ (1/3)

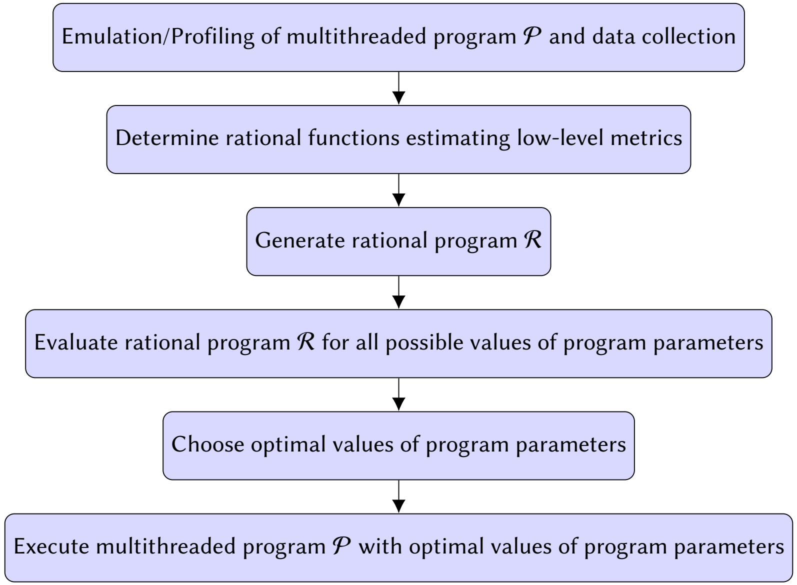
### Assumptions

1. $\mathcal{E}$ is a high-level performance metric for the multithreaded program $\mathcal{P}$ (e.g. execution time, memory consumption, and hardware occupancy).
2. $\mathcal{E}$ is given (by a program execution model, e.g. MWP-CWP) as a rational program depending on hardware parameters $H_1, \ldots, H_h$, low-level metrics $L_1, \ldots, L_\ell$, and program parameters $P_1, \ldots, P_p$.
3. The values of the hardware parameters $H_1, \ldots, H_h$ are known at compile-time for $\mathcal{P}$.
4. The values of the data parameters $D_1, \ldots, D_d$ are known at runtime for $\mathcal{P}$.
5. The data and program parameters $D_1, \ldots, D_d, P_1, \ldots, P_p$ take integer values.

### Decomposing the process into six steps:

- the first three take place at compile-time,
- the other three are performed at execution-time.

## Six steps for building a rational program $\mathcal{R}$ (2/3)


Emulation/Profiling of multithreaded program $\mathcal{P}$ and data collection
Determine rational functions estimating low-level metrics
Generate rational program $\mathcal{R}$
Evaluate rational program $\mathcal{R}$ for all possible values of program parameters
Choose optimal values of program parameters
Execute multithreaded program $\mathcal{P}$ with optimal values of program parameters

## Six steps for building a rational program $\mathcal{R}$ (3/3)

1. **Data collection**: We select a set of points $K_1, \ldots, K_k$ in the space of the possible values of $(D_1, \ldots, D_d, P_1, \ldots, P_p)$. We call this space $F$. Then we run (or emulate) the program $\mathcal{P}$ on these points and measure the low-level performance metrics $L_1, \ldots, L_\ell$; and for each $1 \leq i \leq \ell$, we record the values $(v_{i,1}, \ldots, v_{i,k})$ measured for $L_i$ at the respective points $K_1, \ldots, K_k$.
2. **Rational function determination**: For each $1 \leq i \leq \ell$, we use the values $(v_{i,1}, \ldots, v_{i,k})$ measured for $L_i$ at the respective points $K_1, \ldots, K_k$ to estimate the rational function $g_i(D_1, \ldots, D_d, P_1, \ldots, P_p)$ using curve fitting (e.g. by linear least squares).
3. **Code generation**: Each rational function is converted into sub-routines for evaluating one of $L_1, \ldots, L_\ell$. Those sub-routines are included into code for computing $\mathcal{E}$ based on some model, yielding the desired rational program $\mathcal{R}$.
4. **Rational program evaluation**: At execution time the data parameters $D_1, \ldots, D_d$ now have known, specific values, say $\delta_1, \ldots, \delta_d$. With those data parameter values and possible program parameter values the rational program $\mathcal{R}$ can be run to evaluate $\mathcal{E}$.
5. **Selection of optimal values of program parameters**: Using either exhaustive search or a numerical optimization technique we can determine values of the program parameters which optimize $\mathcal{E}$.
6. **Program execution**: Executing $\mathcal{P}$ using the selected configuration of program parameters $P_1, \ldots, P_p$ along with with the values $\delta_1, \ldots, \delta_d$ of $D_1, \ldots, D_d$.

## Experimentation (1/4)

- performance of KLARAPTOR by applying it to the CUDA programs of the `Polybench`/GPU benchmark suite [48].
- Data was collected using a GTX 1080Ti.
- The one-time compile-time cost of optimization can often be amortized by only a few executions of the kernel.

## Experimentation (2/4)

Table: KLARAPTOR vs. exhaustive search for thread block configuration choice for kernels in `Polybench/GPU`.

| Kernel | N | KLARAPTOR Time (ms) | Chosen Config. | Optimal Time (ms) | Optimal Config. |
|---|---|---|---|---|---|
| atax K1 | 4096 | 2.35 | 32, 4 | 0.85 | 32, 1 |
| | 8192 | 27.83 | 1, 64 | 4.33 | 16, 2 |
| atax K2 | 4096 | 1.09 | 16, 2 | 1.04 | 32, 1 |
| | 8192 | 2.20 | 32, 1 | 2.19 | 64, 1 |
| bicg K1 | 4096 | 1.05 | 256, 1 | 1.05 | 32, 1 |
| | 8192 | 2.23 | 256, 1 | 2.21 | 64, 1 |
| bicg K2 | 4096 | 1.15 | 8, 4 | 0.85 | 32, 1 |
| | 8192 | 12.58 | 256, 4 | 4.35 | 512, 1 |
| convolution2d | 4096 | 0.79 | 256, 1 | 0.77 | 32, 4 |
| | 8192 | 2.54 | 256, 4 | 2.35 | 32, 4 |
| corr | 4096 | 5700.65 | 256, 1 | 5075.77 | 32, 1 |
| | 8192 | 27846.91 | 256, 1 | 26024.94 | 32, 1 |
| covar | 4096 | 5682.96 | 256, 1 | 5076.77 | 32, 1 |
| | 8192 | 27865.89 | 256, 1 | 26182.65 | 32, 1 |
| fdtd_step1 | 4096 | 0.56 | 256, 1 | 0.56 | 32, 2 |
| | 8192 | 2.22 | 256, 4 | 2.22 | 32, 4 |
| fdtd_step2 | 4096 | 0.58 | 256, 1 | 0.58 | 512, 1 |
| | 8192 | 2.33 | 32, 16 | 2.30 | 512, 1 |
| fdtd_step3 | 4096 | 0.77 | 256, 1 | 0.77 | 512, 2 |
| | 8192 | 3.06 | 256, 4 | 3.05 | 1024, 1 |

## Experimentation (3/4)

Table: KLARAPTOR vs. exhaustive search for thread block configuration choice for kernels in `Polybench/GPU`.

| Kernel | N | KLARAPTOR Time (ms) | Chosen Config. | Optimal Time (ms) | Optimal Config. |
|---|---|---|---|---|---|
| gemm | 4096 | 723.29 | 256, 1 | 386.76 | 32, 32 |
| | 8192 | 7481.13 | 256, 1 | 3069.66 | 32, 16 |
| gesummv | 4096 | 8.19 | 2, 16 | 1.62 | 32, 1 |
| | 8192 | 82.21 | 32, 16 | 11.58 | 64, 1 |
| gramschmidt K1 | 4096 | 0.09 | 4, 32 | 0.09 | 256, 1 |
| | 8192 | 0.20 | 8, 32 | 0.17 | 64, 1 |
| gramschmidt K2 | 4096 | 0.01 | 32, 2 | 0.01 | 256, 1 |
| | 8192 | 0.01 | 512, 2 | 0.01 | 256, 1 |
| gramschmidt K3 | 4096 | 2.15 | 256, 1 | 2.11 | 32, 1 |
| | 8192 | 4.68 | 256, 1 | 4.61 | 32, 1 |
| mm2 K1 | 4096 | 695.23 | 256, 1 | 384.93 | 32, 32 |
| | 8192 | 7531.13 | 256, 1 | 3062.26 | 32, 16 |
| mm2 K2 | 4096 | 761.49 | 256, 1 | 386.61 | 32, 32 |
| | 8192 | 7533.08 | 256, 1 | 3077.75 | 32, 16 |
| mm3 K1 | 4096 | 749.27 | 256, 1 | 388.40 | 32, 32 |
| | 8192 | 7531.56 | 256, 1 | 3065.34 | 32, 16 |
| mm3 K2 | 4096 | 816.08 | 256, 1 | 389.13 | 32, 16 |
| | 8192 | 7532.66 | 256, 1 | 3067.87 | 32, 16 |
| mm3 K3 | 4096 | 737.21 | 256, 1 | 392.81 | 32, 16 |
| | 8192 | 7530.24 | 256, 1 | 3085.43 | 32, 16 |
| mvt K1 | 4096 | 1.15 | 8, 4 | 0.86 | 32, 1 |
| | 8192 | 12.90 | 256, 4 | 4.35 | 16, 2 |
| mvt K2 | 4096 | 1.05 | 256, 1 | 1.05 | 32, 1 |
| | 8192 | 2.23 | 256, 1 | 2.21 | 128, 1 |
| syr2k | 4096 | 7050.62 | 1, 64 | 2097.15 | 4, 32 |
| | 8192 | 18013.51 | 16, 64 | 17398.88 | 4, 8 |
| syrk | 4096 | 2973.88 | 2, 16 | 1165.24 | 16, 16 |
| | 8192 | 15936.21 | 32, 16 | 9368.56 | 16, 16 |

# Experimentation (4/4)

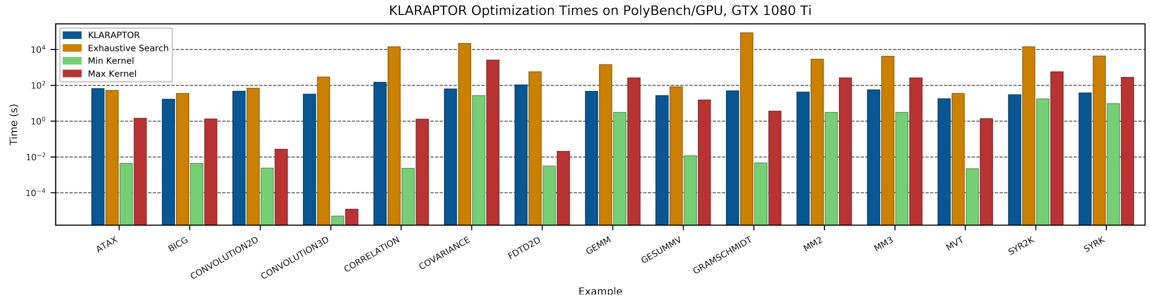


Figure: Comparing times (log-scaled) for (1) compile-time optimization steps of KLARAPTOR, (2) exhaustive search over all thread block configurations, the execution time for a kernel given (3) the best thread block configuration, and (4) the worst thread block configuration. Exhaustive search is given as a sum for values up to $N = 8192$ (except convolution3d with $N = 1024$).

## Future work

- Recently, the author of [58] and [59] has suggested a GPU performance model relying on Little's law; it measures concurrency as a product of latency and throughput.
- This model considers both warp and instruction concurrency while previous models [16, 44, 45, 60] consider only warp concurrency.
- The author's analysis of those models suggests their limitation is the significant underestimation of occupancy when arithmetic intensity (the number of arithmetic instructions per memory access) is intermediate.
- Future work: we look to apply an improved performance prediction model.
- Future work: the heuristics used for picking the best block configuration need further improvement.

THE END

## References

[1] 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"*, vol. 93, no. 2, pp. 232– 275, 2005.

[2] M. Frigo and S. G. Johnson, "FFTW: an adaptive software architecture for the FFT," in *IEEE,ICASSP '98, Seattle, Washington, USA, May 12-15, 1998*, pp. 1381–1384, 1998.

[3] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," in *Proceedings of the IEEE*, vol. 93, pp. 216–231, 2005.

[4] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 IEEE/ACM Conference on Supercomputing*, 1998.

[5] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 ed., 2012.
http://gmplib.org/.

[6] W. B. Hart, "Fast Library for Number Theory: An Introduction, booktitle = Proceedings of the Third International Congress on Mathematical Software, series = ICMS'10, year = 2010, location = Kobe, Japan, pages = 88–91, numpages = 4, publisher = Springer-Verlag, address = Berlin, Heidelberg, note = http://flintlib.org,"

[7] V. Shoup, "Number theory library (NTL) for C++," *Available at Shoup's homepage http://www.shoup.net/ntl/*, 2016.

[8] E. A. Arnold, "Modular algorithms for computing Gröbner bases," *J. Symb. Comput.*, vol. 35, no. 4, pp. 403–419, 2003.

[9] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie, "Lifting techniques for triangular decompositions," in *ISSAC 2005, Proceedings* (M. Kauers, ed.), pp. 108–115, ACM, 2005.

[10] M. Moreno Maza and W. Pan, "Fast polynomial arithmetic on a GPU," *J. of Physics: Conference Series*, vol. 256, 2010.

[11] M. Moreno Maza and W. Pan, "Solving bivariate polynomial systems on a GPU," *J. of Physics: Conference Series*, vol. 341, 2011.

[12] R. Agarwal and C. Burrus, "Fast convolution using fermat number transforms with applications to digital filtering," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 22, no. 2, pp. 87–97, 1974.

[13] V. S. Dimitrov, T. V. Cooklev, and B. D. Donevsky, "Generalized fermat-mersenne number theoretic transform," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 41, pp. 133–139, Feb 1994.

[14] S. Covanov and E. Thomé, “Fast arithmetic for faster integer multiplication,” *CoRR*, vol. abs/1502.02800, 2015.

[15] M. Fürer, “Faster integer multiplication,” in *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007* (D. S. Johnson and U. Feige, eds.), pp. 57–66, ACM, 2007.

[16] NVIDIA Corporation, “CUDA C Programming Guide.” https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[17] L. Chen, S. Covanov, D. Mohajerani, and M. Moreno Maza, “Big prime field FFT on the GPU,” in *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*, pp. 85–92, 2017.

[18] F. Franchetti, Y. Voronenko, and M. Püschel, “Tools and techniques for performance - FFT program generation for shared memory: SMP and multicore,” in *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, p. 115, 2006.

[19] A. Ali, L. Johnsson, and J. Subhlok, “Scheduling fft computation on smp and multicore systems,” in *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS ’07, (New York, NY, USA), pp. 293–301, ACM, 2007.

[20] L. Meng, Y. Voronenko, J. R. Johnson, M. Moreno Maza, F. Franchetti, and Y. Xie, "Spiral-generated modular FFT algorithms," in *PASCO* (M. Moreno Maza and J. Roch, eds.), pp. 169–170, ACM, 2010.

[21] M. Moreno Maza and Y. Xie, "FFT-based dense polynomial arithmetic on multi-cores," in *HPCS*, vol. 5976 of *Lecture Notes in Computer Science*, pp. 378–399, Springer, 2009.

[22] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, 1971.

[23] M. Fürer, "Faster integer multiplication," *SIAM J. Comput.*, vol. 39, no. 3, pp. 979–1005, 2009.

[24] A. De, P. P. Kurur, C. Saha, and R. Saptharishi, "Fast integer multiplication using modular arithmetic," in *STOC*, pp. 499–506, 2008.

[25] A. De, P. P. Kurur, C. Saha, and R. Saptharishi, "Fast integer multiplication using modular arithmetic," *SIAM J. Comput.*, vol. 42, no. 2, pp. 685–699, 2013.

[26] D. Harvey, J. van der Hoeven, and G. Lecerf, "Even faster integer multiplication," *J. Complexity*, vol. 36, pp. 1–30, 2016.

[27] D. Harvey, J. v. d. Hoeven, and G. Lecerf, "Faster polynomial multiplication over finite fields," *J. ACM*, vol. 63, pp. 52:1–52:23, Jan. 2017.

[28] D. Harvey, J. van der Hoeven, and G. Lecerf, “Fast polynomial multiplication over f260,” in *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC ’16, (New York, NY, USA), pp. 255–262, ACM, 2016.

[29] S. Covanov and E. Thomé, “Fast integer multiplication using generalized Fermat primes,” *Mathematics of Computation*, 2018.

[30] S. Covanov, D. Mohajerani, M. Moreno Maza, and L. Wang, “Big prime field FFT on multi-core processors,” in *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation, ISSAC 2019, Beijing, China, July 15-18, 2019* (J. H. Davenport, D. Wang, M. Kauers, and R. J. Bradford, eds.), pp. 106–113, ACM, 2019.

[31] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. Moir, M. Moreno Maza, L. Wang, N. Xie, and Y. Xie, “Basic Polynomial Algebra Subprograms (BPAS),” 2019. http://www.bpaslib.org.

[32] D. E. Knuth, *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[33] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra (3. ed.).* Cambridge University Press, 2013.

[34] D. G. Cantor and E. Kaltofen, “On fast multiplication of polynomials over arbitrary algebras,” *Acta Informatica*, vol. 28, no. 7, pp. 693–701, 1991.

[35] D. Harvey and J. van der Hoeven, “Faster integer multiplication using short lattice vectors,” *CoRR*, vol. abs/1802.07932, 2018.

[36] S. Covanov and E. Thomé, “Fast integer multiplication using \goodbreak generalized fermat primes,” *Math. Comput.*, vol. 88, no. 317, pp. 1449–1477, 2019.

[37] D. Harvey and J. Van Der Hoeven, “Integer multiplication in time o (n log n),” 2019.

[38] S. A. Haque and M. Moreno Maza, “Plain polynomial arithmetic on gpu,” in *Journal of Physics: Conference Series*, vol. 385, p. 012014, IOP Publishing, 2012.

[39] Y. Torres, A. González-Escribano, and D. R. Llanos, “ubench: exposing the impact of CUDA block geometry in terms of performance,” *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1150–1163, 2013.

[40] L. J. Stockmeyer and U. Vishkin, “Simulation of parallel random access machines by circuits,” *SIAM J. Comput.*, vol. 13, no. 2, pp. 409–422, 1984.

[41] P. B. Gibbons, “A more practical PRAM model,” in *Proc. of SPAA*, pp. 158–168, 1989.

[42] L. Ma, K. Agrawal, and R. D. Chamberlain, “A memory access model for highly-threaded many-core architectures,” *Future Generation Comp. Syst.*, vol. 30, pp. 202–215, 2014.

[43] S. A. Haque, M. Moreno Maza, and N. Xie, “A many-core machine model for designing algorithms with minimum parallelism overheads,” in *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015*, vol. 27 of *Advances in Parallel Computing*, pp. 35–44, IOS Press, 2015.

[44] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *36th ISCA 2009*, pp. 152–163, 2009.

[45] J. Sim, A. Dasgupta, H. Kim, and R. W. Vuduc, “A performance analysis framework for identifying potential benefits in GPGPU applications,” in *PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*.

[46] M. Püschel, J. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. Johnson, “Spiral: A generator for platform-adapted libraries of signal processing alogorithms,” *IJHPCA*, vol. 18, no. 1, pp. 21–45, 2004.

[47] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, “A compiler framework for optimization of affine loop nests for GPGPUs,” in *ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pp. 225–234, 2008.

[48] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to GPU codes,” in *Proc. InPar*, pp. 1–10, IEEE, 2012.

[49] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, “A script-based autotuning compiler system to generate high-performance CUDA code,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, 2013.

[50] K. Sato, H. Takizawa, K. Komatsu, and H. Kobayashi, “Automatic tuning of CUDA execution parameters for stencil processing,” in *Software Automatic Tuning, From Concepts to State-of-the-Art Results*, 2010.

[51] J. Kurzak, Y. Tsai, M. Gates, A. Abdelfattah, and J. J. Dongarra, “Massively parallel automated software tuning,” in *ICPP 2019, Kyoto, Japan, 2019*, pp. 92:1–92:10, 2019.

[52] T. Kistler and M. Franz, “Continuous program optimization: A case study,” *(TOPLAS)*, vol. 25, no. 4, pp. 500–548, 2003.

[53] C. Song, L.-P. Wang, and T. J. Martínez, “Automated code engine for graphical processing units: Application to the effective core potential integrals and gradients,” *Journal of chemical theory and computation*, vol. 12, no. 1, pp. 92–106, 2015.

[54] C. Chen, X. Chen, A. Keita, M. Moreno Maza, and N. Xie, “MetaFork: A compilation framework for concurrency models targeting hardware accelerators and its application to the generation of parametric CUDA kernels,” in *Proceedings of CASCON 2015*, pp. 70–79, 2015.

[55] R. V. L., B. N., and A. D. M., "Autotuning GPU kernels via static and predictive analysis," in *ICPP 2017*, pp. 523–532, 2017.

[56] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10, IEEE, 2009.

[57] J. D. Garvey and T. S. Abdelrahman, "Automatic performance tuning of stencil computations on gpus," in *ICPP 2015*, pp. 300–309, 2015.

[58] V. Volkov, "A microbenchmark to study GPU performance models," in *Proc. PPoPP*, pp. 421–422, 2018.

[59] V. Volkov, *Understanding Latency Hiding on GPUs.*
PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[60] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," PPoPP '10, pp. 105–114, ACM, 2010.

## Appendix

## Example: controlling intermediate expression swell

### For $f = 3x^4 - 4x^2$ and $p = 5$, compute $f^5 \pmod p$

- First compute $f^5$, then, reduce mod $p$:
  $f^2 = 9x^8 - 24x^6 + 16x^4$
  $f^3 = 27x^{12} - 108x^{10} + 144x^8 - 64x^6$
  $f^4 = 81x^{16} - 432x^{14} + 864x^{12} - 768x^{10} + 256x^8$
  $f^5 = 243x^{20} - 1620x^{18} + 4320x^{16} - 5760x^{14} + 3840x^{12} - 1024x^{10}$
  $f^5 \equiv 3x^{20} + x^{10} \pmod p$
- Reduce mod $p$ in each step:
  $f^2 \equiv 4x^8 + x^6 + x^4 \pmod p$
  $f^3 \equiv 2x^{12} + 2x^{10} + 4x^8 + x^6 \pmod p$
  $f^4 \equiv x^{16} + 3x^{14} + 4x^{12} + 2x^{10} + x^8 \pmod p$
  $f^5 \equiv 3x^{20} + x^{10} \pmod p$

## Example: modular methods in polynomial system solving

Triangular decomposition over the rationals with finitely many solutions can be computed as follows:

1. Solve modulo a “small” prime $p_1$ and use Hensel lifting to recover a triangular decomposition over the rationals.
2. Use another “small” prime $p_2$ to check whether the lifted decomposition reduced modulo $p_2$ matches the decomposition computed modulo $p_2$
3. If success then returns the lifted decomposition otherwise go back to Step 1.

For large input systems, it is desirable to use larger primes, possibly of size several machine words, so as to increase success in one loop iteration.

## Complexity analysis

### Fürer's trick

- Let $N = K^e$ for some "small" $K$ (say $K = 32$) and an integer $e \geq 2$.
- Define $\eta = \omega^{N/K}$, let $J = K^{e-1}$, and assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by $\eta^i$ $(0 \leq i \leq K)$ can be done within $O(k)$ word ops.
- Every arithmetic operation involved in $\mathrm{DFT}_K$ at $\eta$ costs $O(k)$ word ops.
- Therefore, such $\mathrm{DFT}_K$ can be performed within $O(K \log(K)\, k)$ word ops.
- This latter result holds whenever $p$ is a *generalized Fermat number*.

### Recall the CT factorization:

$$\mathrm{DFT}_{JK} = (\mathrm{DFT}_J \otimes I_K) D_{J,K} (I_J \otimes \mathrm{DFT}_K) L_J^{JK}, \quad (1)$$

the DFT of $f$ at $\omega$ is essentially performed by:

- $J = K^{e-1}$ DFT's of size $K$,
- $N$ multiplication by a power of $\omega$ (coming from the diagonal matrix $D_{J,K}$)
- $K$ DFT's of size $J = K^{e-1}$.

# Applying Furer's trick to CT factorization

## Unrolling Formula (1) (replace $\mathrm{DFT}_J$ by $\mathrm{DFT}_K$) yields:

- $e\,K^{e-1}$ $\mathrm{DFT}_K$'s (each at cost $O(K\,\log_2(K)\,k)$ word ops.), in total $O(e\,N\,\log_2(K)k)$ word ops. ,
- $(e-1)\,N$ multiplication by a power of $\omega$ (each at cost $O(\mathsf{M}(k))$ word ops.), in total $O(e\,N\,\mathsf{M}(k)) = O(N\,\log_K(N)\,\mathsf{M}(k))$ word ops .
- So, $\mathrm{DFT}_N(\omega)$ amounts to $O(N\,\log_2(N)\,k \;+\; N\,\log_K(N)\,\mathsf{M}(k))$ word ops.
- Using generalized Fermat primes, we have $K = 2k$, therefore:

$$O(N\,\log_2(N)\,k \;+\; \underbrace{N\,\log_k(N)\,\mathsf{M}(k)}_{\text{dominant term}}) \tag{2}$$

- Without our assumption, the same DFT would run in $O(N\,\log_2(N)\,\mathsf{M}(k))$ word ops.
- Therefore, using generalized Fermat primes brings a speedup factor of $\log(K)$ w.r.t. the direct approach using arbitrary prime numbers.

## CPU vs. GPU implementations

Using NTL library, we have developed sequential C code for both approaches.

Table: The smallest/largest running time ratio ($\frac{T_{\text{CPU}}}{T_{\text{GPU}}}$) for computing the benchmark for $N = K^e$ for $P_3$ and $P_4$ (timings in milliseconds), collected on Intel Xeon X5650 @ 2.67GHz CPU, Intel Core i7-4700HQ @ 2.40GHz CPU, AMD FX(tm)-8350 @ 2.40GHz CPU, NVIDIA Tesla GTX760m, NVIDIA Tesla C2075, NVIDIA Tesla M2050.

| Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ $(K = 16)$ (timings in milliseconds) | | | |
|---|---|---|---|
| e | NTL Small FFT | Small FFT GPU | Speed-up |
| 2 | 2.51 - 4.06 | 2.73 - 12.92 | 0.19**X** - 1.48**X** |
| 3 | 14.19 - 23.19 | 6.27 - 15.35 | 0.92**X** - 3.69**X** |
| 4 | 232.76 - 372.19 | 15.57 - 50.49 | 4.61**X** - 23.90**X** |
| e | C Big FFT | BigFFT GPU | Speed-up |
| 2 | 0.73-4.13 | 0.03-0.05 | 14.6**X** - 137.6**X** |
| 3 | 20.01-35.08 | 0.88-1.24 | 16.13**X** - 39.86**X** |
| 4 | 505.96 - 750.40 | 17.41-26.06 | 19.41**X** - 43.10**X** |

| Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ $(K = 32)$ (timings in milliseconds) | | | |
|---|---|---|---|
| e | NTL Small FFT | Small FFT GPU | Speed-up |
| 2 | 12.00 - 14.94 | 9.33 - 27.22 | 0.44**X** - 1.60**X** |
| 3 | 233.26 - 384.10 | 23.39 - 62.98 | 3.70**X** - 16.42**X** |
| 4 | 7573.65 - 11303.76 | 437.29 - 1772.92 | 4.27**X** - 25.84**X** |
| e | C Big FFT | BigFFT GPU | Speed-up |
| 2 | 8.00 - 12.91 | 0.27 - 0.37 | 21.62**X** - 47.81**X** |
| 3 | 396.00 - 692.16 | 14.80 - 20.80 | 19.03**X** - 46.76**X** |
| 4 | 22992.00 - 33351.29 | 695.02 - 971.28 | 23.67**X** - 479.62**X** |

## Motivation: an idea of Martin Fürer

### Assumptions

- Let $p$ be a $k$-machine-word generalized Fermat prime and $N > 0$ an integer diving $p - 1$.
- Consider the FFT of a vector of size $N$ over the prime field $\mathbb{Z}/p\mathbb{Z}$.
- Assume $N = K^e$ for some "small" $K$ (say $K = 32$) and an integer $e \geq 2$.
- Let $\omega$ be a $N$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$ and let $\eta = \omega^{N/K}$.
- Assumption: multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by $\eta^i$ $(0 \leq i \leq K)$ can be done within $O(k)$ word ops.

### Consequences

- Every arithmetic operation involved in $\mathrm{DFT}_K$ (on $K$ points of $\mathbb{Z}/p\mathbb{Z}$), can be done within $O(k)$ word ops.
- Therefore, $\mathrm{DFT}_K$ can be performed within $O(K\,\log(K)\,k)$ word ops instead of the "a priori' $O(K\,\log(K)\,M(k))$ word ops.
- Under our hypotheses, unrolling Cooley-Tukey formula, it follows that $\mathrm{DFT}_N$ can be performed within $O(N\,\log_2(N)\,k\;+\;N\,\log_k(N)\,k\log_2(k))$ word ops instead of $O(N\,\log_2(N)\,k\;+\;N\,k\log_2^2(k))$ word ops for small-primes+CRT.

## Generalized Fermat prime (GFP)

- Prime in the form of $p = r^k + 1$, where $k$ is a power of 2 and the radix $r$ is a machine-word size integer.
- Note that $r$ is a $2k$-th primitive root of unity modulo $p$.
- From now on, $p$ is a generalized Fermat prime (GFP), we refer to $\mathbb{Z}/p\mathbb{Z}$ as a GFPF.

### Representing elements of $\mathbb{Z}/p\mathbb{Z}$

An element $x \in \mathbb{Z}/p\mathbb{Z}$ is represented in one of the following equivalent ways:

- by a vector $\vec{x} = (x_{k-1}, \ldots, x_0)$ of length $k$
$$x \equiv x_{k-1}\, r^{k-1} + x_{k-2}\, r^{k-2} + \cdots + x_1\, r + x_0 \mod p$$
- by a polynomial $f_x \in \mathbb{Z}[R]$,
$$f_x = \sum_{i=0}^{k-1} x_i\, R^i$$
such that $x \equiv f_x(r) \mod p$.
- Either way, we have two cases for the value of $x$:
  1. When $x \equiv p - 1 \mod p$ holds, we have $x_{k-1} = r$ and $x_{k-2} = \cdots = x_0 = 0$.
  2. When $0 \leq x < p - 1$ holds, we have $0 \leq x_i < r$ for $i = 0, \ldots, k-1$.

# Arithmetic in $\mathbb{Z}/p\mathbb{Z}$

- Addition and multiplication can be computed like grade school arithmetic.
- In practice, $p$ is too small for considering multi-threaded addition or multiplication.

## Multiplication of two arbitrary elements

- For $x, y \in \mathbb{Z}/p\mathbb{Z}$, multiplying $x \cdot y$ in $\mathbb{Z}/p\mathbb{Z}$, is done computing $f_x(R) \cdot f_y(R)$ in $\mathbb{Z}[R]/\left\langle R^k + 1 \right\rangle$ followed by a conversion into the radix $r$ representation.
- Challenge: How to multiply the two polynomials $f_x(R) \cdot f_y(R)$ efficiently, when the size of the intermediate products $x_i\, y_j$ can be larger than one machine word?

# Low-cost multiplication: multiplying by a power of radix $r$

- For an arbitrary element $x \in \mathbb{Z}/p\mathbb{Z}$, we want to compute $x \cdot r^i \mod p$, for $0 \leq i < k$.
- Computing modulo $p = r^k + 1$, we can replace every $r^k$ by $-1$.
- For $0 < i < k$

$$\begin{aligned} xr^i &\equiv (x_{k-1}\, r^{k-1+i} + \cdots + x_0\, r^i) \mod p \\ &\equiv (\sum_{h=i}^{h=k-1} x_{h-i} r^h - \sum_{h=k}^{h=k-1+i} x_{h-i} r^{h-k}) \mod p \end{aligned}$$

- We reduce a multiplication to a negacyclic shift and subtraction.

## FFT-based multiplication between arbitrary elements of $\mathbb{Z}/p\mathbb{Z}$

### Computing $f_x(R) \cdot f_y(R)$

- Treat elements in $\mathbb{Z}/p\mathbb{Z}$ as polynomials over $\mathbb{Z}$.
- Using FFT over small prime fields to multiply two polynomials.
- Normalizing the result polynomial into an element in $\mathbb{Z}/p\mathbb{Z}$.
- Convolution computes $f_x \cdot f_y \mod (R^k - 1)$.
- However, we need to compute $f_x \cdot f_y \mod (R^k + 1)$.

### Negacyclic convolution computes $f_u = f_x \cdot f_y \mod (R^k + 1)$ in a field:

- Assume that $\theta$ is a $2k$-th primitive root of unity.
- With $\vec{A} = (1, \theta, \ldots, \theta^{k-1})$ and $\vec{A}' = (1, \theta^{-1}, \ldots, \theta^{1-k})$, negacyclic convolution computes:
$$\vec{u} = \vec{A}' \cdot \mathsf{InverseDFT}(\mathsf{DFT}(\vec{A} \cdot \vec{x}) \cdot \mathsf{DFT}(\vec{A} \cdot \vec{y}))$$

## Implementation of the FFT-based multiplication in BPAS (1/4)

**Each $|u_i|$ is at most $kr^2$; it can exceed the size of a machine word!**

$$\begin{array}{rcl}
f_u(R) & = & f_x(R)\cdot f_y(R) \mod (R^k+1) \\
 & = & \sum_{m=0}^{2k-2} \sum_{0\le i,j<k}^{i+j=m} x_i\, y_j\, R^m \mod (R^k+1) \\
 & = & (x_{k-1}\, y_0 + x_{k-2}\, y_1 + x_{k-3}\, y_2 + \cdots + x_1\, y_{k-2} + x_0\, y_{k-1})\, R^{k-1} \\
 & + & (x_{k-2}\, y_0 + x_{k-3}\, y_1 + \cdots + x_1\, y_{k-3} + x_0\, y_{k-2} - x_{k-1}\, y_{k-2})\, R^{k-2} \\
 & & \ldots \\
 & + & (x_0\, y_0 - x_{k-1}\, y_1 - \cdots - x_1\, y_{k-1}) \\
 & = & \sum_{m=0}^{k-1} (\sum_{0\le i,j<k}^{i+j=m} x_i\, y_j - \sum_{0\le i,j<k}^{i+j=k+m} x_i\, y_j)\, R^m
\end{array}$$

## Implementation of the FFT-based multiplication in BPAS (2/4)

### CRT step

- Choose $\mathbb{Z}/q_1\mathbb{Z}$ and $\mathbb{Z}/q_2\mathbb{Z}$, where $q_1$ and $q_2$ are machine word size primes.
- Compute FFT over $\mathbb{Z}/q_i\mathbb{Z}$ and deduce FFT over $\mathbb{Z}/(q_1\,q_2)\mathbb{Z}$ via CRT.

### LHC step

- After CRT, coefficients of $f_u = f_x \cdot f_y \mod (R^k+1) \in \mathbb{Z}$ are 128-bit numbers.
- Each coefficient $u_i$ of $f_u$ is re-written as $u_i = c_i\,r^2 + h_i\,r + \ell_i$ where $0 \leq \ell_i, h_i < r$ and $c_i \in [-k, k]$.

$$\begin{aligned} f_u(R) &= f_x(R) \cdot f_y(R) \mod (R^k+1) \\ &= (c_0R^2 + h_0R + \ell_0) + (c_1R^2 + h_1R + \ell_1)R + \cdots + (c_{k-1}R^2 + h_{k-1}R + \ell_{k-1})R^{k-1} \\ &= \sum_{i=0}^{k-1}(c_i\,R^{2+i} + h_i\,R^{1+i} + \ell_i\,R^i) \\ &= R^2\sum_{i=0}^{k-1}(c_i\,R^i) + R\sum_{i=0}^{k-1}(h_i\,R^i) + \sum_{i=0}^{k-1}(\ell_i\,R^i) \end{aligned}$$

## Implementation of the FFT-based multiplication in BPAS (3/4)

### FFT-based multiplication algorithm

```
1: procedure FFT-basedMultiplication($\vec{x}, \vec{y}, r, k$)
2:     $\vec{z_1} := \text{NegacyclicConvolution}(\vec{x}, \vec{y}, p_1, k)$
3:     $\vec{z_2} := \text{NegacyclicConvolution}(\vec{x}, \vec{y}, p_2, k)$
4:     for $0 \le i < k$ do
5:         $[s_{0i}, s_{1i}] := \text{CRT}(p_1, p_2, m_1, m_2, z_{1i}, z_{2i})$
6:     end for
7:     for $0 \le i < k$ do
8:         $[\ell_i, h_i, c_i] := \text{LHC}(s_{0i}, s_{1i}, r)$
9:     end for
10:    $\vec{c} := \text{MulPowR}(\vec{c}, 2, k, r)$
11:    $\vec{h} := \text{MulPowR}(\vec{h}, 1, k, r)$
12:    $\vec{u} := \text{BigPrimeFieldAddition}(\vec{\ell}, \vec{h}, k, r)$
13:    $\vec{u} := \text{BigPrimeFieldAddition}(\vec{u}, \vec{c}, k, r)$
14:    return $\vec{u}$
15: end procedure
```

## Implementation of the FFT-based multiplication in BPAS (4/4)

**Problem: Lots of modular multiplications in the negacyclic convolutions**

Solution: We use Montgomery multiplication inside convolutions!

**Problem: CRT and LHC parts need multi-precision arithmetic!**

- gcc provides 128-bit arithmetic, however, it is not the most efficient way!
- Solution: Using assembly code for CRT and LHC computation.

## The big prime field FFT in the BPAS library

- Our implementation is based on the **six-step FFT algorithm**:
$$\mathrm{DFT}_N = L_K^N \,(I_J \otimes \mathrm{DFT}_K) L_J^N \, D_{K,J}\,(I_K \otimes \mathrm{DFT}_J)\, L_K^N \;\; with \;\; N = J\,K.$$
- Here, $\otimes$ denotes the tensor product of two matrices $A$ and $B$:
$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$
- The stride permutation $L_m^{mn}$ permutes an input vector $\vec{x}$ of length $mn$:
$$\vec{x}[in+j] \mapsto \vec{x}[jm+i].$$
- The twiddle factor $D_{K,J}$ is a diagonal matrix of the powers of $\omega$.
$$D_{K,J} = \bigoplus_{j=0}^{K-1} \mathrm{diag}\,(1, \omega_i^j, \ldots, \omega_i^{j(J-1)}).$$

## Computing $\mathrm{DFT}_{K^e}$ through base-case $\mathrm{DFT}_K$

- Since $p = r^k + 1$, we know that $r^k = -1 \mod p$, $r$ is a $2k$-th primitive root of unity in $\mathbb{Z}/p\mathbb{Z}$.
- We know that multiplying by powers of the radix $r$ is cheap.

### Recall Fürer's trick

- Define $\eta = \omega^{N/K}$, let $J = K^{e-1}$, and assume that multiplying an arbitrary element of $\mathbb{Z}/p\mathbb{Z}$ by $\eta^i$ $(0 \leq i \leq K)$ can be done within $O(k)$ word ops.
- Every arithmetic operation involved in $\mathrm{DFT}_K$ at $\eta$ costs $O(k)$ word ops.
- Therefore, such $\mathrm{DFT}_K$ can be performed within $O(K \log(K)\, k)$ word ops.

### Reducing to base-case

- Let $N = K^e$, then we can compute $\mathrm{DFT}_{K^e}$ by $\mathrm{DFT}_K$.
- By choosing $K = 2k$, the multiplication inside $\mathrm{DFT}_{2k}$ is cheap.

# Computing base-case $\mathrm{DFT}_K$ (1/3)

## Reducing $\mathrm{DFT}_K$ to $\mathrm{DFT}_2$ for $K = 2^n$

- Definition of $\mathrm{DFT}_2$:
$$\mathrm{DFT}_2(x_0, x_1) = (x_0 + x_1, x_0 - x_1)$$
- Six-step factorization of $\mathrm{DFT}_{2^n}$:
$$\mathrm{DFT}_{2^n} = L_2^{2^n}\,(I_{2^{n-1}} \otimes \mathrm{DFT}_2)\,L_{2^{n-1}}^{2^n}\,D_{2,2^{n-1}}\,(I_2 \otimes \mathrm{DFT}_{2^{n-1}})\,L_2^{2^n}$$
- Example of $\mathrm{DFT}_8$ through $\mathrm{DFT}_2$:
$$\mathrm{DFT}_8 = L_2^8\,(I_4 \otimes \mathrm{DFT}_2)\,L_4^8\,D_{2,4}\,(I_2 \otimes \mathrm{DFT}_4)\,L_2^8 \tag{3}$$
$$\mathrm{DFT}_4 = L_2^4\,(I_2 \otimes \mathrm{DFT}_2)\,L_2^4\,D_{2,2}\,(I_2 \otimes \mathrm{DFT}_2)\,L_2^4 \tag{4}$$
$$\begin{aligned}\mathrm{DFT}_8 = {} & L_2^8\,(I_4 \otimes \mathrm{DFT}_2)\,L_4^8\,D_{2,4}\,(I_2 \otimes L_2^4)(I_4 \otimes \mathrm{DFT}_2) \\ & (I_2 \otimes L_2^4)(I_2 \otimes D_{2,2})(I_4 \otimes \mathrm{DFT}_2)(I_2 \otimes L_2^4)(L_2^8).\end{aligned} \tag{5}$$

## Computing base-case $\mathrm{DFT}_K$ (2/3)

### Avoiding permutation

- Avoiding the permutation and actually data movement.
- Pre-compute the position of elements after each permutation and hard-code those values in the algorithm for computing the base-case.
- Index of input data: $\vec{M} = (0, 1, 2, 3, 4, 5, 6, 7)$

$$\vec{M}_1 = L_2^8 \vec{M} = (0, 2, 4, 6, 1, 3, 5, 7) \tag{6}$$

$$\vec{M}_2 = (I_2 \otimes L_2^4)\vec{M}_1 = (0, 4, 2, 6)(1, 5, 3, 7) \tag{7}$$

$$\mathrm{DFT}_2(0, 4) \longrightarrow \mathrm{DFT}_2(2, 6) \longrightarrow \mathrm{DFT}_2(1, 5) \longrightarrow \mathrm{DFT}_2(3, 7) \tag{8}$$

### Twiddle multiplications

- Then, we have the following twiddle matrices as part of $\mathrm{DFT}_8$:

$$D_{2,2} = \ \mathrm{diag}\,(1, 1, \omega_1^0, \omega_1^1), \quad D_{2,4} = \ \mathrm{diag}\,(1, 1, 1, 1, \omega_0^0, \omega_0^1, \omega_0^2, \omega_0^3) \tag{9}$$

- We have $\omega_0 = r$ and $\omega_1 = r^2$ ($r^8 \equiv 1 \mod p$, for $p = r^4 + 1$).
- Then, the twiddle matrices are updated as follows:

$$D_{2,4} = \mathrm{diag}\,(1, 1, 1, 1, 1, r, r^2, r^3) \tag{10}$$

$$D_{2,2} = \mathrm{diag}\,(1, 1, 1, r^2) \tag{11}$$

## Computing base-case $\mathrm{DFT}_K$ (3/3)

### Example: Computing $\mathrm{DFT}_8$ for vector $\vec{\mathrm{a}}$

```
1: DFT2($a_0, a_4$); DFT2($a_1, a_5$);
2: DFT2($a_2, a_6$); DFT2($a_3, a_7$);
3: $a_6 := a_6\, \omega^2$;
4: $a_7 := a_7\, \omega^2$;
5: DFT2($a_0, a_2$); DFT2($a_1, a_3$);
6: DFT2($a_4, a_6$); DFT2($a_5, a_7$);
7: $a_5 := a_5\, \omega^1$;
8: $a_3 := a_3\, \omega^2$;
9: $a_7 := a_7\, \omega^2$;
10: DFT2($a_0, a_1$); DFT2($a_2, a_3$);
11: DFT2($a_4, a_5$); DFT2($a_6, a_7$);
12: swap($a_1, a_4$);
13: swap($a_3, a_6$);
14: return $\vec{a}$;
```

# Parallelization of the FFT

## Choice of the FFT algorithm

- The six-step FFT can be implemented in an iterative fashion.
- Unroll $I_K \otimes \mathrm{DFT}_J$ until there's only DFT on $K$ points.
- There is no data dependency between the iterations of each inner-loop.

## Programming considerations

- `Cilk`: work-stealing scheme, light-weight threads (re-use of the threads), etc.
- The FFT over the crafted BPAS implementation of GFPF incurs less memory accesses than the FFT based on GMP arithmetics; see experimental results.

## Implementation of the six-step FFT

```
 1: procedure DFT_GENERAL($\vec{x}, K, e, \omega,$)
 2:     for $0 \le i < e - 1$ do
 3:         for $0 \le j < K^i$ do                                  ▷ Can be replaced with Parallel-For.
 4:             stride_permutation($x_{jK^{e-i}}, K, K^{e-i-1}$)                                 ▷ Step 1
 5:         end for
 6:     end for
 7:     $\omega_a := \omega^{K^{e-1}}$
 8:     for $0 \le j < K^{e-1}$ do                                  ▷ Can be replaced with Parallel-For.
 9:         $\mathtt{idx} := jK$
10:         DFT_K($x_{\mathtt{idx}}, \omega_a$)                                                ▷ Step 2
11:     end for
12:     for $e - 2 \ge i \ge 0$ do
13:         $\omega_i := \omega^{K^i}$
14:         for $0 \le j < K^i$ do                                  ▷ Can be replaced with Parallel-For.
15:             $\mathtt{idx} := j\,K^{e-i}$
16:             twiddle($x_{\mathtt{idx}}, K^{e-i-1}, K, \omega_i$)                               ▷ Step 3
17:             stride_permutation($x_{\mathtt{idx}}, K^{e-i-1}, K$)                            ▷ Step 4
18:         end for
19:         for $0 \le j < K^{e-1}$ do                              ▷ Can be replaced with Parallel-For.
20:             $\mathtt{idx} := jK$
21:             DFT_K($x_{\mathtt{idx}}, \omega_a$)                                            ▷ Step 5
22:         end for
23:         for $0 \le j < K^i$ do                                  ▷ Can be replaced with Parallel-For.
24:             $\mathtt{idx} := jK^{e-i}$
25:             stride_permutation($x_{\mathtt{idx}}, K, K^{e-i-1}$)                            ▷ Step 6
26:         end for
27:     end for
```

## Benchmarks

### Two approaches: GFPF vs. GMP

- Generalized Fermat prime field arithmetic relying on FFT-based multiplication
- GMP arithmetic.

### Benchmarks: Comparing performance of the following

- Multiplication of two arbitrary elements of the big prime field.
- Serial vs. parallel implementation of each approach for computing FFT of large vectors over different big prime fields.
- Each step of an FFT computation.

### Experimentation Setup

Table: The set of big primes of different sizes which are used for experimentations.

| prime | $K(=2k)$ | $k$ | $r$ |
|---|---|---|---|
| $P_8$ | 16 | 8 | $2^{59}+2^{57}+2^{39}$ |
| $P_{16}$ | 32 | 16 | $2^{58}+2^{55}+2^{45}$ |
| $P_{32}$ | 64 | 32 | $2^{58}+2^{55}+2^{17}$ |
| $P_{64}$ | 128 | 64 | $2^{57}+2^{56}+2^{11}$ |

- `Intel-i7-7700K`: 4-cores @4.50 GHz (8 threads when hyper-threading is enabled), 16 GB of memory (@2133 MHz).
- `Xeon-X5650`: 6-cores @2.66 GHz (12 threads when hyper-threading is enabled), 48 GB of memory (@1133 MHz).

## Multiplication between arbitrary elements in big prime fields

Table: The running-time of computing $10^6$ modular multiplications in $\mathbb{Z}/p\mathbb{Z}$ for $P_8$, $P_{16}$, $P_{32}$, and $P_{64}$ (measured on `Intel-i7-7700K`).

| prime | $k$ | GFPF | GMP | Ratio ( $\frac{t_{GFPF}}{t_{GMP}}$ ) |
|---|---|---|---|---|
| $P_8$ | 8 | 645 (ms) | 171(ms) | 3.77**X** |
| $P_{16}$ | 16 | 1318 (ms) | 417 (ms) | 3.16**X** |
| $P_{32}$ | 32 | 2852 (ms) | 1179 (ms) | 2.41**X** |
| $P_{64}$ | 64 | 6101 (ms) | 3452 (ms) | 1.76**X** |

Table: Time (in milliseconds) and percentage (%) of the total time spent in different steps of computing $10^6$ GFPF multiplications of arbitrary elements in $\mathbb{Z}/p\mathbb{Z}$ for primes $P_8$, $P_{16}$, $P_{32}$, and $P_{64}$ (measured on `Intel-i7-7700K`).

| prime | $k$ | Convolution | | CRT | | LHC | | Normalization | |
|---|---|---|---|---|---|---|---|---|---|
| | | Time | % | Time | % | Time | % | Time | % |
| $P_8$ | 8 | 323 | 45 | 150 | 21 | 208 | 29 | 35 | 5 |
| $P_{16}$ | 16 | 851 | 52 | 288 | 18 | 425 | 26 | 64 | 4 |
| $P_{32}$ | 32 | 2083 | 57 | 563 | 15 | 847 | 23 | 177 | 5 |
| $P_{64}$ | 64 | 4751 | 61 | 1115 | 14 | 1497 | 19 | 434 | 6 |

# FFT over big prime fields: measured on `Intel-i7-7700K`

Table: The running-time (in milliseconds) and ratio ($t_{\text{GFPF}}/t_{\text{GMP}}$) of serial and parallel computation of FFT on vectors of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ for $P_4$, $P_8$, $P_{16}$, $P_{32}$, $P_{64}$, and $P_{128}$ (measured on `Intel-i7-7700K`).

| prime | $k$ | $K$ | $e$ | Serial | | | Parallel | | | Parallel Speedups | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | GFPF | GMP | $\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$ | GFPF | GMP | $\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$ | GFPF | GMP |
| $P_4$ | 4 | 8 | 2 | 0.019 | 0.030 | 0.63**X** | 0.057 | 0.118 | 0.48**X** | 0.33 | 0.25 |
| $P_4$ | 4 | 8 | 3 | 0.314 | 0.363 | 0.86**X** | 0.215 | 0.276 | 0.77**X** | 1.46 | 1.32 |
| $P_8$ | 8 | 16 | 2 | 0.181 | 0.202 | 0.89**X** | 0.117 | 0.143 | 0.81**X** | 1.55 | 1.41 |
| $P_8$ | 8 | 16 | 3 | 5.771 | 5.486 | 1.05**X** | 1.603 | 2.247 | 0.71**X** | 3.60 | 2.44 |
| $P_{16}$ | 16 | 32 | 2 | 1.644 | 1.730 | 0.95**X** | 0.513 | 0.693 | 0.74**X** | 3.20 | 2.50 |
| $P_{16}$ | 16 | 32 | 3 | 103.423 | 104.620 | 0.98**X** | 24.052 | 35.017 | 0.68**X** | 4.30 | 2.99 |
| $P_{32}$ | 32 | 64 | 2 | 14.815 | 20.341 | 0.72**X** | 3.507 | 5.411 | 0.64**X** | 4.22 | 3.76 |
| $P_{32}$ | 32 | 64 | 3 | 1922.373 | 2431.867 | 0.79**X** | 462.746 | 702.163 | 0.65**X** | 4.15 | 3.46 |
| $P_{64}$ | 64 | 128 | 2 | 140.995 | 278.188 | 0.50**X** | 33.507 | 69.879 | 0.47**X** | 4.21 | 3.98 |
| $P_{128}$ | 128 | 256 | 2 | 580.961 | 3745.353 | 0.15**X** | 154.064 | 905.799 | 0.17**X** | 3.77 | 4.13 |

## FFT over big prime fields: measured on Xeon-X5650

Table: The running-time (in milliseconds) and ratio ($t_{\text{GFPF}}/t_{\text{GMP}}$) of serial and parallel computation of FFT on vectors of size $N = K^e$ over $\mathbb{Z}/p\mathbb{Z}$ for $P_4$, $P_8$, $P_{16}$, $P_{32}$, $P_{64}$, and $P_{128}$ (measured on Xeon-X5650).

| prime | $k$ | $K$ | $e$ | Serial | | | Parallel | | | Parallel Speedups | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | GFPF | GMP | $\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$ | GFPF | GMP | $\frac{t_{\text{GFPF}}}{t_{\text{GMP}}}$ | GFPF | GMP |
| $P_4$ | 4 | 8 | 2 | 0.051 | 0.071 | 0.71**X** | 0.155 | 0.114 | 1.35**X** | 0.33 | 0.62 |
| $P_4$ | 4 | 8 | 3 | 0.843 | 0.917 | 0.91**X** | 0.452 | 0.577 | 0.78**X** | 1.87 | 1.59 |
| $P_8$ | 8 | 16 | 2 | 0.472 | 0.546 | 0.86**X** | 0.217 | 0.320 | 0.67**X** | 2.18 | 1.71 |
| $P_8$ | 8 | 16 | 3 | 16.661 | 15.231 | 1.09**X** | 2.837 | 4.806 | 0.59**X** | 5.87 | 3.17 |
| $P_{16}$ | 16 | 32 | 2 | 4.444 | 5.085 | 0.87**X** | 0.877 | 1.371 | 0.63**X** | 5.07 | 3.71 |
| $P_{16}$ | 16 | 32 | 3 | 284.080 | 297.904 | 0.95**X** | 41.012 | 66.635 | 0.61**X** | 6.93 | 4.47 |
| $P_{32}$ | 32 | 64 | 2 | 39.809 | 64.307 | 0.61**X** | 5.701 | 11.640 | 0.48**X** | 6.98 | 5.52 |
| $P_{32}$ | 32 | 64 | 3 | 4674.079 | 6501.669 | 0.71**X** | 696.311 | 1289.061 | 0.54**X** | 6.71 | 5.04 |
| $P_{64}$ | 64 | 128 | 2 | 376.450 | 909.041 | 0.41**X** | 53.578 | 140.610 | 0.38**X** | 7.03 | 6.46 |
| $P_{128}$ | 128 | 256 | 2 | 1395.310 | 13371.369 | 0.10**X** | 240.362 | 1811.282 | 0.13**X** | 5.81 | 7.38 |

## Time spent in each step of FFT

### Time spent in each step of FFT

Table: Time spent (milliseconds) in different steps of serial and parallel computation of DFT of size $N = K^3$ over $\mathbb{Z}/p\mathbb{Z}$, for prime $P_{32}$ $(K = 2k = 64)$ measured on `Intel-i7-7700K`.

| Mode | Variant | Precomputation | Permutation | $\mathrm{DFT}_K$ | Twiddle |
|---|---|---|---|---|---|
| Serial | GFPF | 14 (ms) | 72 (ms) | 444 (ms) | 1406 (ms) |
| | GMP | 6 (ms) | 177 (ms) | 1229 (ms) | 1026 (ms) |
| Parallel | GFPF | 14 (ms) | 51 (ms) | 82 (ms) | 330 (ms) |
| | GMP | 6 (ms) | 181 (ms) | 284 (ms) | 237 (ms) |

### Average multiplication time in FFT

| $K$ | 16 | | | | 32 | | | | 64 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $e$ | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 |
| FFT-based | 0.32 | 2.96 | 3.53 | 3.77 | 0.65 | 5.72 | 6.84 | 7.31 | 1.44 | 10.87 | 12.98 |
| GMP | 4.17 | 4.17 | 4.17 | 4.17 | 11.79 | 11.79 | 11.79 | 11.79 | 34.53 | 34.53 | 34.53 |

Table: Average time (in milliseconds) spent in one modular multiplication during computation of FFT over big prime fields, presented for the three implementations measured on `Intel-i7-7700K`.

## Comparing memory accesses GFPF vs. GMP

The number of memory references measured for each variant (and the ratio $\frac{\#\text{GFPF D refs}}{\#\text{GMP D refs}}$) of serial computation of FFT on vectors of size $N = K^2$ over $\mathbb{Z}/p\mathbb{Z}$ for $P_{16}$, $P_{32}$, $P_{64}$, and $P_{128}$.

Table: Measured on `Intel-i7-7700K` using `Valgrind`.

| input size | D refs | | | D1 miss rate (%) | |
|---|---|---|---|---|---|
| $N = K^2$ | GFPF | GMP | $\frac{\#\text{ GMP refs}}{\#\text{ GFPF refs}}$ | GFPF | GMP |
| K=16 | 689,220 | 1,042,440 | 1.51**X** | 0.2 | 0.9 |
| K=32 | 5,704,483 | 7,810,065 | 1.36**X** | 0.5 | 0.7 |
| K=64 | 50,718,515 | 82,608,297 | 1.62**X** | 0.4 | 0.5 |
| K=128 | 535,935,616 | 1,063,157,320 | 1.98**X** | 0.8 | 0.5 |

Table: Measured on `Xeon-X5650` using `Valgrind`.

| input size | D refs | | | D1 miss rate (%) | |
|---|---|---|---|---|---|
| $N = K^2$ | GFPF | GMP | $\frac{\#\text{ GMP refs}}{\#\text{ GFPF refs}}$ | GFPF | GMP |
| K=16 | 645,018 | 1,043,169 | 1.61**X** | 0.2 | 0.9 |
| K=32 | 5,340,965 | 7,824,678 | 1.46**X** | 0.5 | 0.7 |
| K=64 | 49,143,357 | 82,748,934 | 1.68**X** | 0.4 | 0.5 |
| K=128 | 556,770,530 | 1,070,452,476 | 1.92**X** | 0.7 | 0.5 |

## System cache specifications

| Metric | Intel-i7-7700K | xeonnode02 |
|---|---|---|
| Line size | 64 | 64 |
| L1d cache | 32K | 32K |
| L1i cache | 32K | 32K |
| L2 cache | 256K | 256K |
| L3 cache | 8192K | 12288K |

## Considerations for GMP implementation

- Function `mpz_mul` is not in-place, better to have a separate destination than input arguments.
- Immediate mpz functions are cheaper to use (such `mpz_add_ui`).
- Due to memory management overhead, `mpz_mod` is more expensive than `mpz_tdiv_r`.

## Precomputation of twiddle factors

### Twiddle matrices are in the form of $D_{K,K^{e-s}}$ where $\omega_i = \omega^{K^{(s-1)}} (1 \le s < e)$

- We know that $\omega^N \equiv r^{2k} \equiv r^K \equiv 1 \mod p$.
- For $y = x \cdot \omega^{i(N/K)+j}$, we only need to compute:
  1. $y' = x \cdot \omega^{i(N/K)} = x \cdot r^i$ (a cheap multiplication), and
  2. $y = y' \cdot \omega^j$ (arbitrary multiplication).
- We can pre-compute $\omega^j$ with $0 < j < N/K$, leading to a lower pre-computation expense and less memory usage.

## Example: inline assembly for multiplying two 64-bit integers

```
void mult_u64_u64
(const usfixn64 *a, const usfixn64 *b,
 usfixn64 *s0_out,usfixn64 *s1_out)
{
  usfixn64 s0=0, s1=0;
  __asm__ __volatile__(
  "movq  %2, %%rax;\n\t"// rax = a
  "movq  %3, %%rdx;\n\t"// rdx = b
  "mulq  %%rdx;\n\t"    // rdx:rax = a * b
  "movq  %%rax, %0;\n\t"// s0 = rax (low part)
  "movq  %%rdx, %1;\n\t"// s1 = rdx (high part)
  :"=&q" (s0),"=&q"(s1)
  :"q"(*a), "q"(*b)
  :"%rax", "%rdx", "memory");
  *s0_out = s0;
  *s1_out = s1;
}
```