Big Prime Field FFTs on the GPU

Liyangyu Chen[1] Svyatoslav Covanov[2]
Davood Mohajerani[3] Marc Moreno Maza[3,4]

[1]East China Normal University, China
[2]University of Lorraine, France
[3]ORCCA, University of Western Ontario, Canada
[4]IBM Center for Advanced Studies, Markham, Canada

ISSAC 2017, University of Kaiserslautern, Kaiserslautern, Germany
July 26, 2017

Outline

Outline

Modular methods in polynomial system solving

Triangular decompositions 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.

Outline

Outline

Contributions and main results

- Algorithms and CUDA implementation for arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$, where p is Generalized Fermat prime of size 8 or 16 machhine 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.

Outline

Outline

Discrete Fourier transform

Definition

Let $\mathcal{R}$ be a commutative ring with unity and $\omega \in \mathcal{R}$ be a principal N-th root of unity. The linear map $\vec{a} = (a_0, \ldots, a_{N-1})^T \xrightarrow{\Omega} \vec{b} = (b_0, \ldots, b_{N-1})^T$ with matrix $\Omega = (\omega^{jk})_{0 \le j,k \le N-1}$ is the DFT_N at ω.

Cooley-Tukey factorization formula

- For $J, K > 1$, we have:

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

where

$$D_{J,K} = \bigoplus_{j=0}^{J-1} \mathrm{diag}(1, \omega^j, \ldots, \omega^{j(K-1)}),$$

$$L^{JK}_J = \mathbf{x}[iJ + j] \mapsto \mathbf{x}[jJ + i], (0 \le j < J,\ 0 \le i < K.)$$

- Various fast Fourier transform algorithms can be derived from this formula, including Cooley-Tukey (CT) and Stockham.

FFT over a prime field

- Let p be a prime and N be a power of 2 s.t. $N \mid p-1$.
- Then, $\mathbb{Z}/p\mathbb{Z}$ admits a N-th primitive root of unity, say ω.
- Let $f \in \mathbb{Z}/p\mathbb{Z}[x]$ be a polynomial of degree at most $N-1$.
- Computing $\mathrm{DFT}_N(f)$ at ω by a FFT amounts to
  - $N\log(N)$ additions in $\mathbb{Z}/p\mathbb{Z}$,
  - $N/2\log(N)$ multiplications by a power of ω in $\mathbb{Z}/p\mathbb{Z}$,
- If p spans k machine words, then each addition costs $\mathcal{O}(k)$ word ops., and each multiplication by a power of ω costs $O(\mathrm{M}(k))$ word ops.
- Here, $n \longmapsto \mathrm{M}(n)$ is a multiplication time.
- Hence, multiplication by a power of ω becomes a bottleneck as k grows.

Outline

Outline

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 η^i $(0 \leq i \leq K)$ can be done within $O(k)$ word ops.
- Every arithmetic operation involved in DFT_K at η costs $O(k)$ word ops.
- Therefore, such 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:

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

the DFT of f at ω is essentially performed by:

- $J = K^{e-1}$ DFT's of size K,
- N multiplication by a power of ω (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 DFT_J by DFT_K) yields:

- $e\,K^{e-1}$ 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 ω (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.

Comparison I

The cost of a comparable computation using small primes and the CRT

- Let $p_1, \ldots, p_k$ be pairwise different prime numbers of machine-word size and let m be their product.
- Assume that N divides each of $p_1 - 1, \ldots, p_k - 1$ such that each of fields $\mathbb{Z}/p_1\mathbb{Z}, \ldots, \mathbb{Z}/p_k\mathbb{Z}$ admits an N-th primitive roots of unity, $\omega_1, \ldots, \omega_k$.
- 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 $\mathbb{Z}/m\mathbb{Z}$.
- Let $f \in \mathbb{Z}/m\mathbb{Z}[x]$ be a polynomial of degree $N - 1$.

Comparison II

- One can compute the DFT of f at ω in three steps:
  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.
  2. Compute the DFT of f_i at ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$, for $i = 1, \ldots, k$,
  3. Combine the results using CRT so as to obtain a DFT of f at ω.
- The 1st and the 3rd steps will run within $O(N \times \mathsf{M}(k) \log_2(k))$ word ops.
- The 2nd step amounts to $O(k \times N \log(N))$ word ops.
- In total:

$$O(N \log_2(N)\, k \;+\; N\, \mathsf{M}(k) \log_2(k)) \tag{3}$$

- These estimates yield a running-time ratio between the two approaches of $\log(N)/\log_2^2(k)$, which suggests that for k large enough the big prime field approach may outperform the CRT-based approach.
- This analysis is part of the explanation for the observation that the two approaches are, competitive in practice, as we shall see in experimental results.

Outline

Outline

Sparse-radix generalized Fermat numbers (SRGFN)

- Generalized Fermat numbers $a^{2^n} + b^{2^n}$ where $a > 1$, $b \geq 0$ and $n \geq 0$.
- For $F_n(a) = a^{2^n} + 1$, a is a 2^{n+1}-th primitive root of unity $\mathbb{Z}/F_n(a)\mathbb{Z}$,
- A SRGFN is any $F_n(r)$ where r is $2^w + 2^u$ or $2^w - 2^u$, for $w > u \geq 0$

Representing elements of $\mathbb{Z}/F_n(r)\mathbb{Z}$

- Let $p = F_n(r)$, and $k = 2^n$ s.t. $p = r^k + 1$ holds
- Represent $x \in \mathbb{Z}/p\mathbb{Z}$ as $\vec{x} = (x_{k-1}, x_{k-2}, \ldots, x_0)$ such that:

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

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

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

- $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by vectors $\vec{x}, \vec{y}$ with k coefficients

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 fast algorithms
- For small k, say $k \leq 16$, using plain multiplication is reasonable.

Multiplication by r^i for some $0 < i < 2k$ in $\mathbb{Z}/p\mathbb{Z}$

- 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:

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

- Computing the product $x\,r^i$ reduces to a negacyclic shift.

Outline

Outline

Implementation notes (1)

- We use NVIDIA CUDA, a platform for writing scalable parallel programs on GPUs
- Our code is tuned for SRGFN's $P_3 := (2^{63} + 2^{34})^8 + 1$ and $P_4 := (2^{62} + 2^{36})^{16} + 1$.

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

$$\mathrm{DFT}_N = L^N_K (I_J \otimes \mathrm{DFT}_K) L^N_J D_{K,J} (I_K \otimes \mathrm{DFT}_J) L^N_K.$$

- We expand $I_K \otimes \mathrm{DFT}_J$ to turn all DFT computations to base-case DFT_K.

Implementation notes (2)

Parallelization.

In our implementation, each arithmetic operation is computed by one thread.

Memory-bound kernels.

- Performance is limited by frequent accesses to memory.
- We have considered solutions for:
  - minimizing memory latency,
  - maximizing occupancy (no. of active warps on each streaming multiprocessor) to hide latency, and
  - maximizing IPC (instructions per clock cycle).
- Location of data: keep all 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.

Implementation notes (3)

Size of thread blocks.

- Kernels do not depend on the size of a thread block.
- 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).

Effect of GPU instructions on performance.

- Current implementation relies on 64-bit instructions (due to 64-bit radix).
- NVIDIA GPUs support 64-bit integer instructions, however, compiler converts all instructions to a sequence of 32-bit equivalents.
- This conversion might have a negative impact on the overall performance (specially, 64-bit multiplication!).
- Finally, using 32-bit arithmetic provides more opportunities for optimization, e.g. instruction level parallelism.

Implementation notes (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.
- In practice, this data structure will hurt performance due to increased memory overhead, caused by non-coalesced accesses to global memory.
- An effective solution is to perform a stride permutation L_k^{kN} on all input vectors
- Therefore, accesses to global memory will be coalesced, increasing memory load and store efficiency, and lowering the memory overhead.

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

Profiling addition for $N = 2^{17}$ (random input)

Metric Description	Avg. non-transposed	Avg. transposed
Achieved Occupancy	92%	87%
Executed IPC	0.13	0.72
Global Memory Load Efficiency	25%	99%
Global Memory Store Efficiency	25%	100%
Instruction Replay Overhead	3.22	0.36
Global Memory Replay Overhead	1.78	0.10
Device Memory Read Throughput	26 GB/s	30 GB/s
Device Memory Write Throughput	13 GB/s	15 GB/s
Total Kernel Time	2.395ms	0.504ms

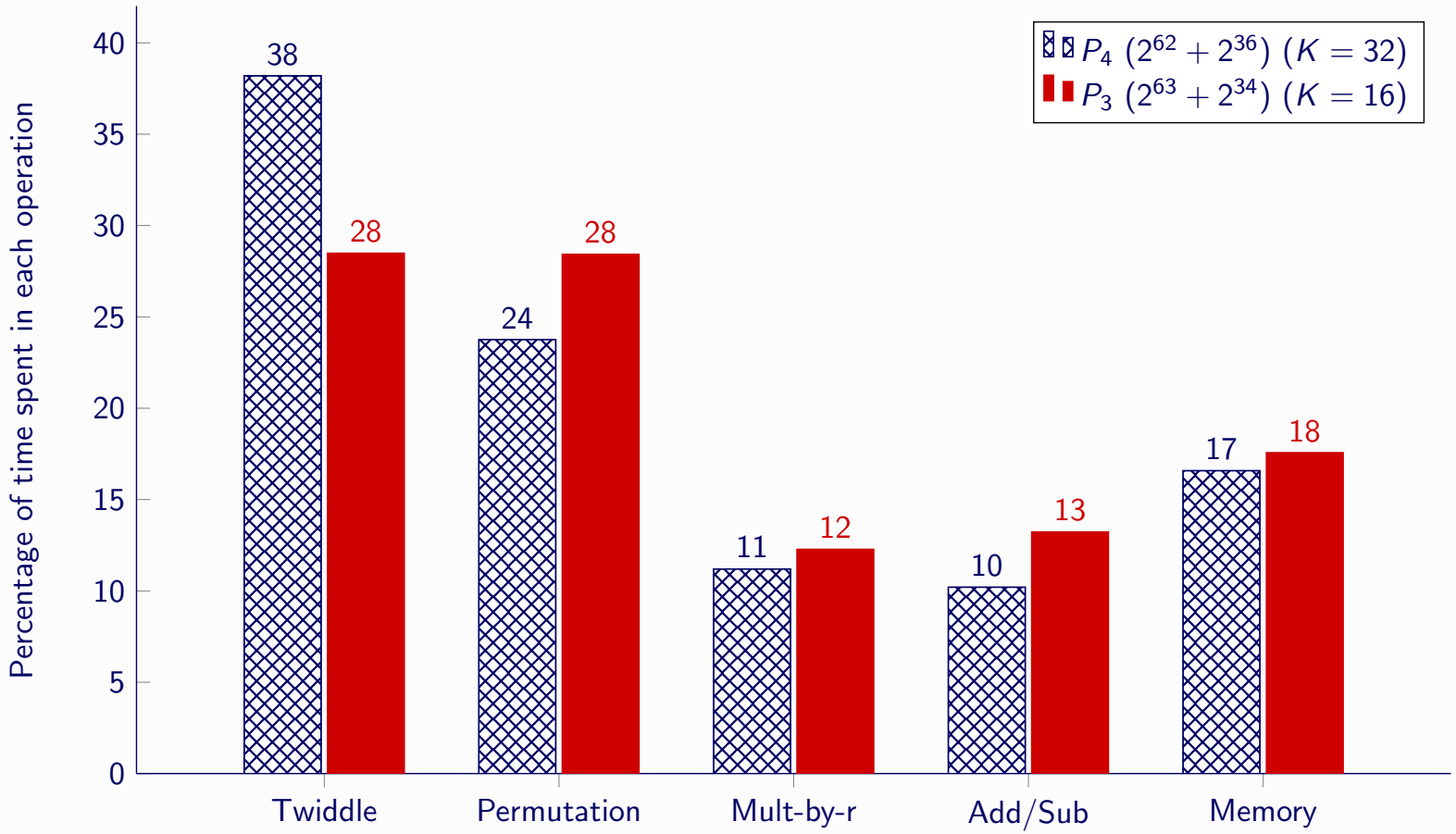
Table: Profiling results for transposed and non-transposed addition in $\mathbb{Z}/p\mathbb{Z}$

Profiling $x\ r^s$ for $N = 2^{17}$ (random input)

Metric Description	Avg. non-transposed	Avg. transposed
Achieved Occupancy	91%	57%
Executed IPC	0.18	4.61
Global Memory Load Efficiency	25%	86%
Global Memory Store Efficiency	25%	87%
Instruction Replay Overhead	3.19	0.04
Global Memory Replay Overhead	1.22	0.01
Device Memory Read Throughput	14 GB/s	20 GB/s
Device Memory Write Throughput	18 GB/s	20 GB/s
Total Kernel Time	3.331ms	0.380ms

Table: Profiling results for transposed and non-transposed $x\ r^s$ in $\mathbb{Z}/p\mathbb{Z}$

Profiling results for DFT_{K^4} with P_3 $(K = 16)$ and P_4 $(K = 32)$



Outline

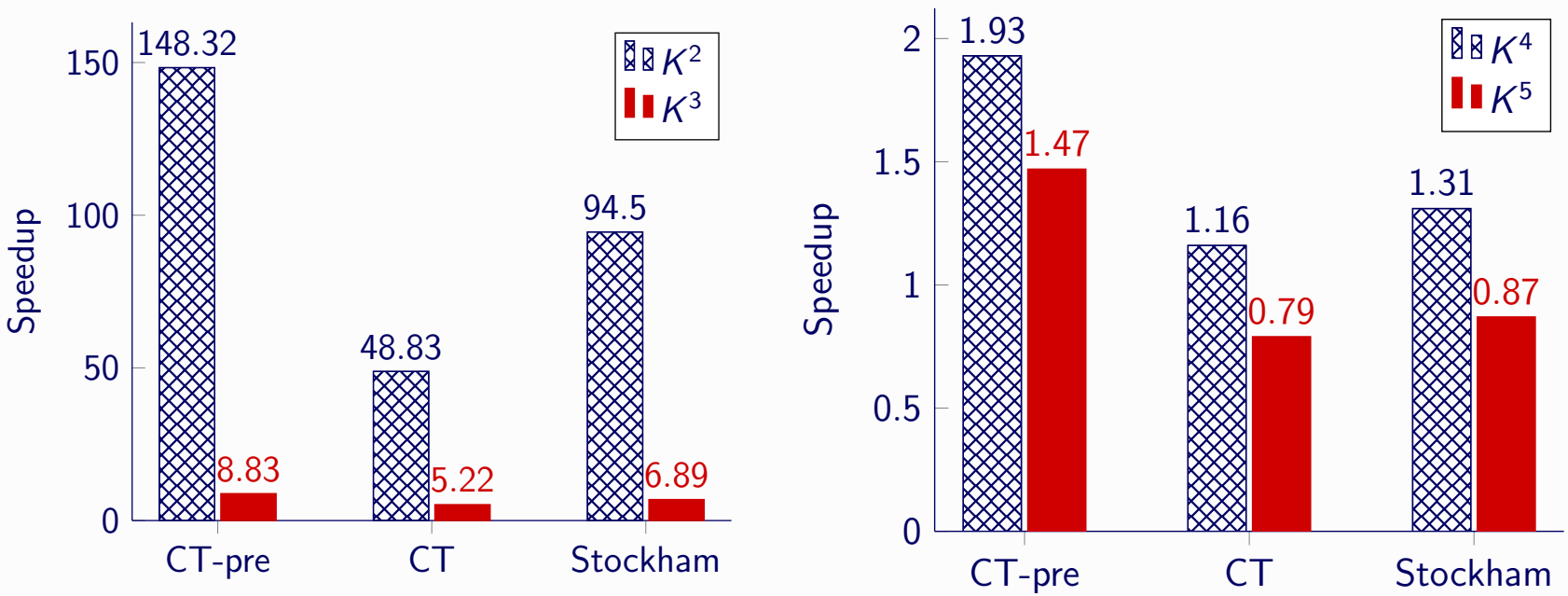
Outline

Big prime vs. Small prime

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 ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$
  3. combine the results using the CRT (*recombination*)
- The small primes are $\dfrac{\text{machine-word size}}{2}$, it is fair to use $2k$ of them!
- For comparing against $P_3 = (2^{63} + 2^{34})^8 + 1$ and $P_4 = (2^{62} + 2^{36})^{16} + 1$ we pick 16 and 32 small primes, respectively.
- Small prime FFTs from the CUMODP library compute DFT_{2^n} for $8 \leq n \leq 26$:
  - the Cooley-Tukey FFT,
  - the Cooley-Tukey FFT with pre-computed powers of ω,
  - the Stockham FFT.
- Tests completed on a NVIDA GeforeGTX760M card

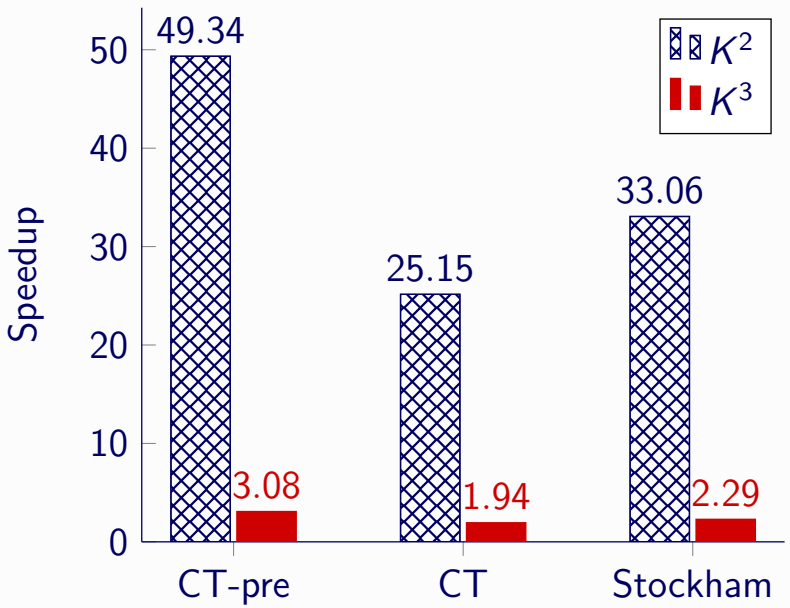
Benchmarking for $P_3 = (2^{63} + 2^{34})^8$ $(K = 16)$

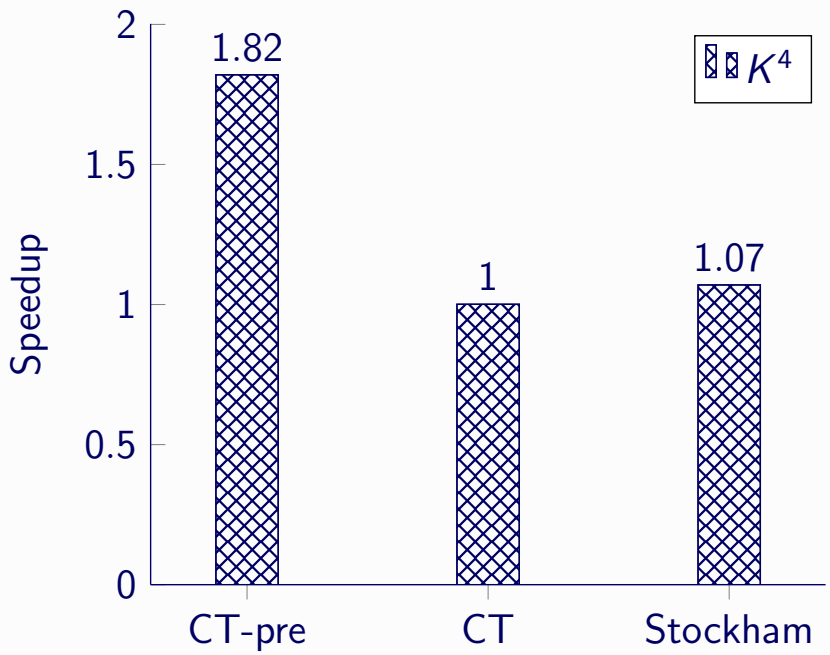


Performance analysis:

- 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 $P_4 = (2^{62} + 2^{36})^{16}$ $(K = 32)$





CPU vs. GPU implementations I

Table: Running time of computing the benchmark for $N = K^e$ using sequential C code on CPU (timings in milliseconds).

Measured on Intel Xeon X5650 @ 2.67GHz CPU		
Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		
e	NTL Small FFT	C Big FFT
2	2.51	1.85
3	23.19	35.08
4	372.19	750.40
Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)		
e	NTL Small FFT	C Big FFT
2	14.94	12.91
3	384.10	692.16
4	11303.76	33351.29

Measured on AMD FX(tm)-8350 @ 2.40GHz CPU		
Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		
e	NTL Small FFT	C Big FFT
2	4.06	4.13
3	16.06	20.01
4	296.00	528.00
Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)		
e	NTL Small FFT	C Big FFT
2	12.00	8.00
3	296.00	396.00
4	10128.00	22992.00

Measured on Intel Core i7-4700HQ @ 2.40GHz CPU		
Computing the benchmark for $N = K^e$ for $P_3 := (2^{63} + 2^{34})^8 + 1$ ($K = 16$)		
e	NTL Small FFT	C Big FFT
2	3.12	0.73
3	14.19	21.06
4	232.76	505.96
Computing the benchmark for $N = K^e$ for $P_4 := (2^{62} + 2^{36})^{16} + 1$ ($K = 32$)		
e	NTL Small FFT	C Big FFT
2	12.48	9.79
3	233.26	496.03
4	7573.65	26089.53

CPU vs. GPU implementations II

Table: Speedup 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).

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**

Concluding remarks

Conclusions

- Big prime field arithmetic is required by advanced algorithms in computer algebra (like polynomial system solving).
- Arithmetic modulo a big prime can be efficiently computed on GPUs
- Transposing input data increases memory usage efficiency
- Computing FFT in $\mathbb{Z}/p\mathbb{Z}$ is competitive with a CRT-based approach for a range of sizes
- Nevertheless multiplication in $\mathbb{Z}/p\mathbb{Z}$ (except for the case of a multiplication by a power of r) remains a computational bottleneck.

Work in progress

- Improving multiplication in $\mathbb{Z}/p\mathbb{Z}$ is work in progress.
- By choosing a larger prime, say with $k = 32$ instead of $k = 8$, or $k = 16$, we hope to cover other ranges of sizes.

Main References

- Faster integer multiplication by Fürer
- Modern computer algebra 3rd edition by von zur Gathen and Gerhard
- Fast polynomial arithmetic by Moreno Maza and Pan
- How to write fast numerical code by Chellappa, Franchetti, and Puschel
- Fast Fourier transforms by Franchetti and Puschel
- CUDA C programming guide 8.0 by NVIDIA Corporation

Thank You!

Your Questions?

Appendix

Finding primitive roots of unity in $\mathbb{Z}/p\mathbb{Z}$

Computing ω:

- **Goal**: speed up $x\omega^i$ for $x \in \mathbb{Z}/p\mathbb{Z}$.
- Let $N = 2^\ell$, s.t. $N \mid p-1$
- Assume that $g^N = 1$ for $g \in \mathbb{Z}/p\mathbb{Z}$
- Write $p = qN + 1$.
- Pick a random $\alpha \in \mathbb{Z}/p\mathbb{Z}$
- Let $\omega = \alpha^q$.
- Fermat's little theorem:
  - if $\omega^{N/2} = 1 \Rightarrow \omega^N = 1$,
  - if $\omega^{N/2} = -1$, pick another α.

Algorithm 1 Primitive N-th root $\omega \in \mathbb{Z}/p\mathbb{Z}$ s.t. $\omega^{N/2k} = r$

procedure $\textsc{PRoot}(N, r, k, g)$

$\alpha := g^{N/2k}$

$\beta := \alpha$

$j := 1$

while $\beta \neq r$ **do**

$\beta := \alpha\beta$

$j := j + 1$

end while

$\omega := g^j$

return (ω)

end procedure

Host entry point function

Template HostGeneralOperation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \mathtt{N}, \mathtt{k}, \mathtt{r}, \mathtt{b}$)

Input:

- an integer b giving the size of 1D thread block,
- Positive integer k, r, and N
- Vectors $\vec{\mathtt{X}}$ and $\vec{\mathtt{Y}}$, each of size N

Output:

- vector $\vec{\mathtt{U}}$ storing the result ($\vec{\mathtt{U}} :=$ operation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}$)).

$\vec{\mathtt{X}} :=$ HostTranspose($\vec{\mathtt{X}}, \mathtt{N}, \mathtt{k}$)

$\vec{\mathtt{Y}} :=$ HostTranspose($\vec{\mathtt{Y}}, \mathtt{N}, \mathtt{k}$)

KernelGeneralOperation$<<<\frac{\mathtt{N}}{\mathtt{b}}, \mathtt{b} >>>(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \mathtt{N}, \mathtt{k}, \mathtt{r})$

return $\vec{\mathtt{U}}$

Template KernelGeneralOperation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \vec{\mathtt{U}}, \mathtt{N}, \mathtt{k}, \mathtt{r}$)

local: `stride` := `N`

local: `offset:=0`

local: vectors $\vec{\mathtt{x}}$, $\vec{\mathtt{y}}$, $\vec{\mathtt{u}}$ each storing `k` digits, all initialized to zero.

local: `tid := blockIdx.x*blockSize.x+threadIdx.x`

for ($0 \leq \mathtt{i} < \mathtt{k}$) **do**

    `offset:=tid +i*stride`

    $\vec{\mathtt{x}}[\mathtt{i}] := \vec{\mathtt{X}}[\mathtt{offset}]$ ▷ Reading the digit with the index `i` of element $\vec{\mathtt{X}}_{\mathtt{tid}}$.

    $\vec{\mathtt{y}}[\mathtt{i}] := \vec{\mathtt{Y}}[\mathtt{offset}]$ ▷ Reading the digit with the index `i` of element $\vec{\mathtt{Y}}_{\mathtt{tid}}$.

end for

$\vec{\mathtt{u}}$:= DeviceGeneralOperation($\vec{\mathtt{x}}, \vec{\mathtt{y}}, \mathtt{k}, \mathtt{r}$). ▷ each thread computing one element of the final result.

for ($0 \leq \mathtt{i} < \mathtt{k}$) **do**

    `offset:=tid +i*stride`

    $\vec{\mathtt{U}}[\mathtt{offset}] := \vec{\mathtt{u}}[\mathtt{i}]$

end for

return ▷ End of Kernel

Host entry point for DFT_{K^e}

Algorithm 4 HostDFTGeneral($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}, \mathtt{b}$)

local: $\mathtt{m} := \mathtt{e}$ where $N = K^e$, $\mathtt{j} := 0$
if $\mathtt{e} \mod 2 = 1$ **then**
    HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^1, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)
end if
for ($0 \leq \mathtt{i} < \mathtt{m}$ by 2) **do**
    HostDFTK2($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}$)
    KernelTwiddleMultiplication($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s} := 2, \mathtt{r}$)
    HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^2, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)
    $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$
    HostDFTK2($\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}$)
    HostGeneralStridePermutation($\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^2, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b}$)
    $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$
end for

Host entry point for DFT_{K^e}

Algorithm 5 HostDFTGeneral$(\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s}, \mathtt{r}, \mathtt{b})$

1: **if** ($\mathtt{e} \mod 2 = 1$) **then**
2: KernelTwiddleMultiplication$(\vec{\mathtt{X}}, \vec{\Omega}, \mathtt{N}, \mathtt{K}, \mathtt{k}, \mathtt{s} := 2, \mathtt{r})$
3: HostGeneralStridePermutation$(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^1, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b})$
4: $\mathtt{X}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$
5: KernelBaseDFTKAllSteps$(\vec{\mathtt{X}}, \mathtt{N}, \mathtt{K}, \mathtt{r})$
6: HostGeneralStridePermutation$(\vec{\mathtt{X}}, \vec{\mathtt{Y}}, \mathtt{K}^1, \mathtt{N}, \mathtt{k}, \mathtt{s}, \mathtt{b})$
7: $\vec{\mathtt{X}}[0 : \mathtt{kN} - 1] := \vec{\mathtt{Y}}[0 : \mathtt{kN} - 1]$
8: **end if**
9: **return** $\vec{\mathtt{X}}$

Block parallelism

$\mathrm{I_N} \otimes \mathrm{A}$: Block parallelism for N blocks

$$\mathrm{I}_4 \otimes \mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 & & & & & & \\ 1 & -1 & & & & & & \\ & & 1 & 1 & & & & \\ & & 1 & -1 & & & & \\ & & & & 1 & 1 & & \\ & & & & 1 & -1 & & \\ & & & & & & 1 & 1 \\ & & & & & & 1 & -1 \end{bmatrix}$$

Mixed-radix conversion

Mixed-radix representation

- For pairwise different primes $p_1, \ldots, p_s$:

$$b_i \in \mathbb{Z}/p_i\mathbb{Z}, \quad 0 \leq b_i < p_i, \quad (1 \leq i \leq s).$$

- Then $(b_1, b_2, \ldots, b_s)$ is *mixed-radix representation* of $n \in \mathbb{Z}$:

$$\begin{cases} n = b_1 + b_2 p_1 + b_3 p_1 p_2 + \cdots + b_s p_1 \cdots p_{s-1} \\ 0 \leq n < p_1 p_2 \cdots p_s \end{cases}$$

Reconstructing n from $(b_1, b_2, \ldots, b_s)$

- pre-compute $m_1 := p_1$, $m_2 := p_1 p_2$, $\ldots$, $m_s := m$,
- compute $u_i := b_i m_i$ (stored in i machine-words),
- compute the sum $n := u_1 + u_2 + \ldots + u_s$

Recombination step

MRR or CRT?

- The CRT defines a ring isomorphism:

$$\mathbb{Z}/p_1\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}/p_s\mathbb{Z} \cong \mathbb{Z}/(p_1 \times \ldots \times p_s)\mathbb{Z}$$

- The MRR defines a bijection:

$$\mathbb{Z}/p_1\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}/p_s\mathbb{Z} \mapsto [0, p_1 p_2 \cdots p_s[$$

  which preserves the order (mapping lex-order to $<$)

- Both take $\Theta(k^2)$ machine-word operations.
- The MRR is interesting for modular methods for real numbers
- We use MRR map instead of the CRT.

Benchmarking FFTs

Computing equivalent results

- Compute a DFT_N in $\mathbb{Z}/p\mathbb{Z}$ with $p = r^k + 1 \rightarrow \mathcal{O}(N \log_K(N) k^2)$
- Compute s DFT_N over small prime fields. $\longrightarrow \mathcal{O}(sN \log_2(N))$
- Compute a MRR on results of small prime FFTs $\longrightarrow \mathcal{O}(sNk^2)$

Example: computing DFT-16 based on DFT-2

- Expanding DFT_{16} based on the six-step FFT algorithm:

$$\begin{aligned}
\mathrm{DFT}_{16} &= L_2^{16}(I_8 \otimes \mathrm{DFT}_2)L_8^{16}D_{2,8}^{16}(I_2 \otimes \mathrm{DFT}_8)L_2^{16}, \\
\mathrm{DFT}_8 &= L_2^{8}(I_4 \otimes \mathrm{DFT}_2)L_4^{8}D_{2,4}^{8}(I_2 \otimes \mathrm{DFT}_4)L_2^{8}, \\
\mathrm{DFT}_4 &= L_2^{4}(I_2 \otimes \mathrm{DFT}_2)L_2^{4}D_{2,2}^{4}(I_2 \otimes \mathrm{DFT}_2)L_2^{4},
\end{aligned}$$

- Following multiplications by twiddle factors are required:
  1. DFT_{16} with $\omega_0 = \omega^{N/K} = r$,
  2. DFT_8 needs $\omega_1 = \omega^{(N/K)^2} = r^2$,
  3. DFT_4 needs $\omega_2 = \omega^{(N/K)^4} = r^4$.

- We have:

$$\begin{aligned}
D_{2,8}^{16} &= (1,1,1,1,1,1,1,1,r^0,r^1,r^2,r^3,r^4,r^5,r^6,r^7), \\
D_{2,4}^{8} &= (1,1,1,1,r^0,r^2,r^4,r^6), \\
D_{2,2}^{4} &= (1,1,r^0,r^4).
\end{aligned}$$

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

- $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by $\vec{x}, \vec{y}$

Algorithm 6 Computing $x + y \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

procedure BigPrimeFieldAddition$(\vec{x}, \vec{y}, r, k)$

1: compute $z_i = x_i + y_i$ in $\mathbb{Z}$, for $i = 0, \ldots, k-1$,

2: let $z_k = 0$,

3: for $i = 0, \ldots, k-1$, compute the quotient q_i and the remainder s_i in the Euclidean division of z_i by r, then replace (z_{i+1}, z_i) by $(z_{i+1} + q_i, s_i)$,

4: if $z_k = 0$ then return $(z_{k-1}, \ldots, z_0)$,

5: if $z_k = 1$ and $z_{k-1} = \cdots = z_0 = 0$, then let $z_{k-1} = r$ and return $(z_{k-1}, \ldots, z_0)$,

6: let i_0 be the smallest index, $0 \leq i_0 \leq k$, such that $z_{i_0} \neq 0$, then let $z_{i_0} = z_{i_0} - 1$, let $z_0 = \cdots = z_{i_0 - 1} = r - 1$ and return $(z_{k-1}, \ldots, z_0)$.

end procedure

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

- Associate $x, y \in \mathbb{Z}/p\mathbb{Z} \longrightarrow f_x, f_y \in \mathbb{Z}[T]$
- For large k, can compute $f_x f_y \mod T^k + 1$ in $\mathbb{Z}[T]$ by fast algorithms
- For small k, say $k \leq 8$, using plain multiplication is reasonable.

Algorithm 7 Computing $xy \in \mathbb{Z}/p\mathbb{Z}$ for $x, y \in \mathbb{Z}/p\mathbb{Z}$

procedure BIGPRIMEFIELDMULTIPLICATION(f_x, f_y, r, k)

1: We compute the polynomial product $f_u = f_x f_y$ in $\mathbb{Z}[T]$ modulo $T^k + 1$.

2: Writing $f_u = \sum_{i=0}^{k-1} u_i T^i$, we observe that for all $0 \leq i \leq k-1$ we have $0 \leq u_i \leq kr^2$ and compute a representation $\overrightarrow{u_i}$ of u_i in $\mathbb{Z}/p\mathbb{Z}$ explained in Section **??**.

3: We compute $u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using the method of Section **??**.

4: Finally, we compute the sum $\sum_{i=0}^{k-1} u_i r^i$ in $\mathbb{Z}/p\mathbb{Z}$ using Algorithm 6.

end procedure