

Big Prime Field FFTs on the GPU

Liyangyu Chen¹ Svyatoslav Covanov²
Davood Mohajerani³ Marc Moreno Maza^{3,4}

¹East China Normal University, China

²University of Lorraine, France

³ORCCA, University of Western Ontario, Canada

⁴IBM Center for Advanced Studies, Markham, Canada

13th Workshop on Challenges For Parallel Computing

CASCON 2018

October 29, 2018

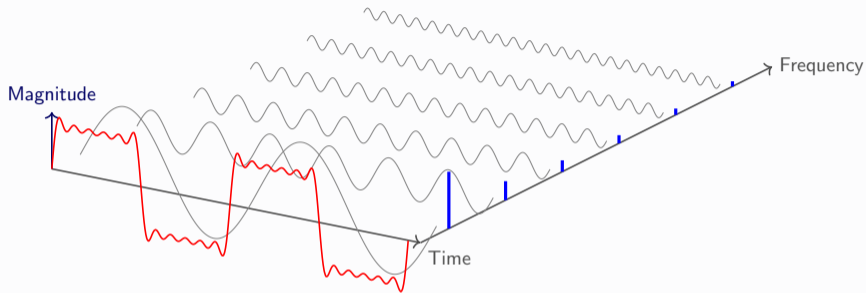
Outline

- ① Fourier transforms
- ② Fürer's trick: beyond Cooley-Tukey factorization
- ③ Implementation challenges
- ④ Experimental Comparison

- 1 Fourier transforms
- 2 Fürer's trick: beyond Cooley-Tukey factorization
- 3 Implementation challenges
- 4 Experimental Comparison

Fourier transform

What does Fourier transform do?



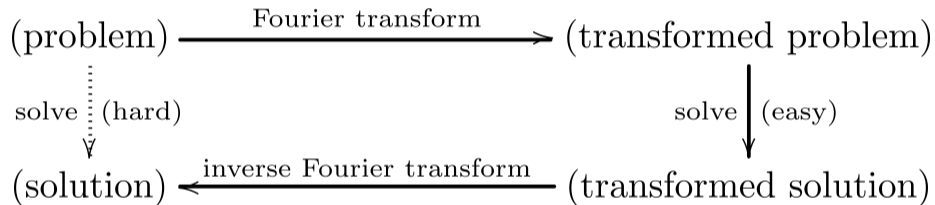
Examples of a function sampled over a finite time interval:

- pressure of a sound wave,
- a radio signal,
- daily temperature readings.

- An extension of the Fourier series when the period approaches infinity.
- It can be studied for complex values of ξ :

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} e^{-2\pi i \xi t} f(t) dt$$

How does FT help solving a problem?

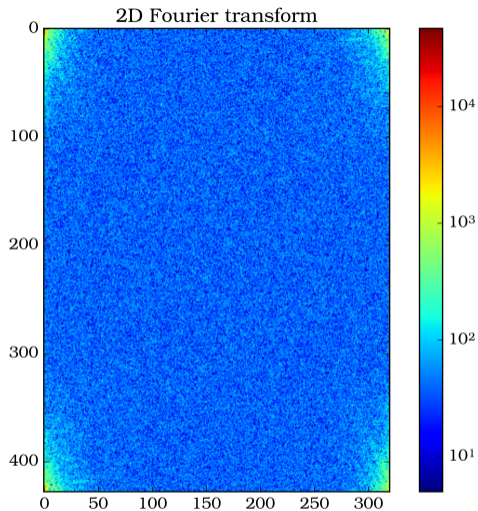
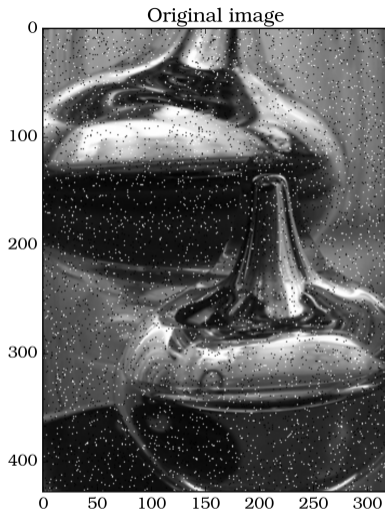


DFT transforms a sequence of N complex numbers $\{\mathbf{x}_n\} := x_0, x_1, \dots, x_{N-1}$ into another sequence of complex numbers, $\{\mathbf{X}_k\} := X_0, X_1, \dots, X_{N-1}$, :

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2\pi i}{N} kn} = \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi kn}{N}\right) - i \cdot \sin\left(\frac{2\pi kn}{N}\right) \right]$$

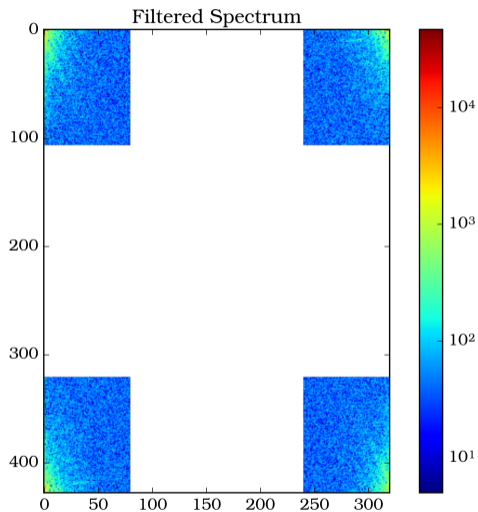
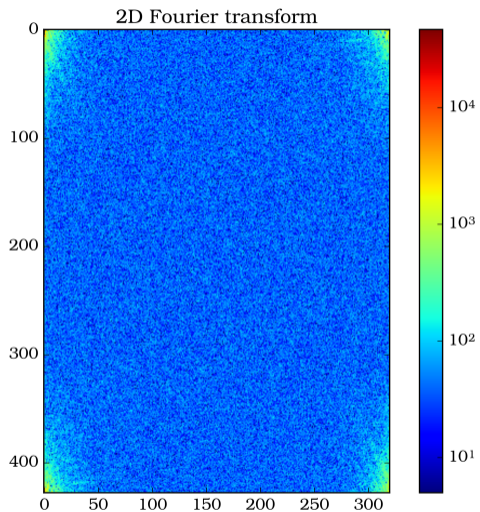
- Digital signal processing,
- Solving partial differential equations,
- Fast polynomial multiplication,
- Multiplying large integers.

Example: Denoising image (1/4)

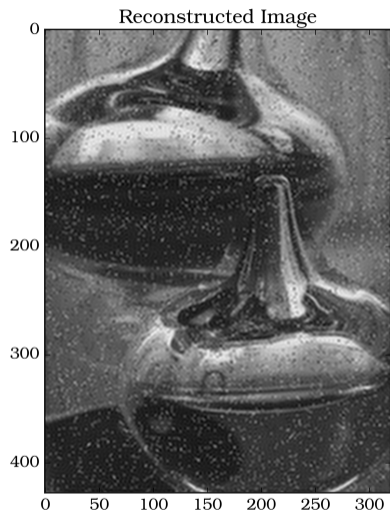
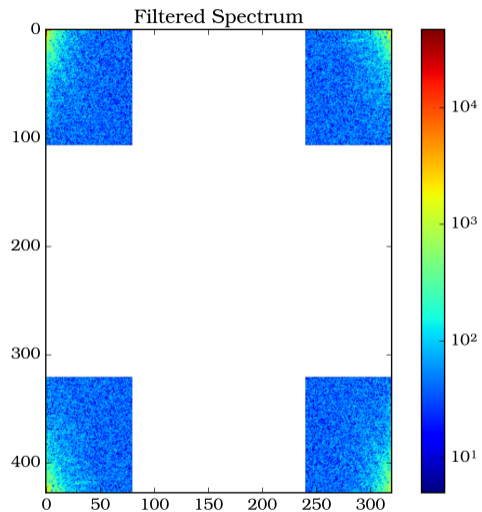


https://www.scipy-lectures.org/intro/scipy/auto_examples/solutions/plot_fft_image_denoise.html

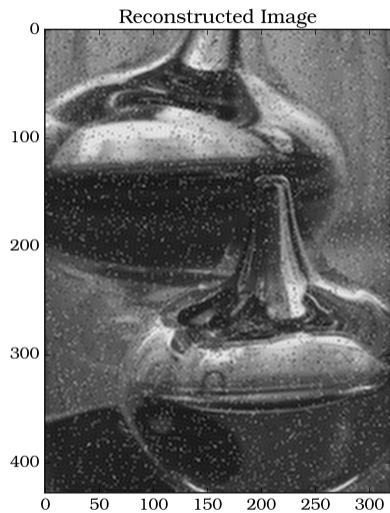
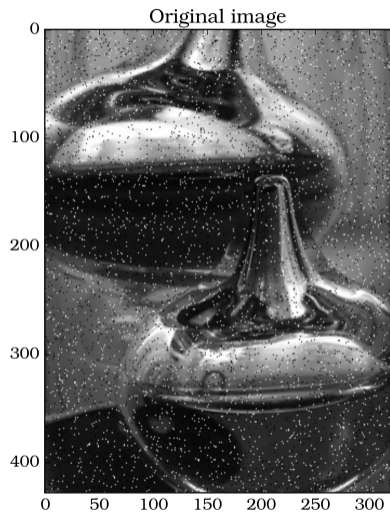
Example: Denoising image (2/4)



Example: Denoising image (3/4)



Example: Denoising image (4/4)



- DFT deals with a finite amount of data, so it can be implemented in computers
- In practice, it is realized both as a software, or through dedicated hardware.

Fourier transform over \mathbb{C} :

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} e^{-2\pi i \xi t} f(t) dt$$



Discret Fourier transform (DFT) over \mathbb{C} :

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi kn}{N}\right) - i \cdot \sin\left(\frac{2\pi kn}{N}\right) \right]$$



DFT over finite field

- We now turn our focus to finite fields.
- Then, the next question is **what are finite fields?**

Field

A set on which addition, subtraction, multiplication, and division are defined.

Examples of field

- \mathbb{C} is a field
- \mathbb{Z} is not a field as inverse is not defined for every element!

Finite field

A field with a finite number of elements.

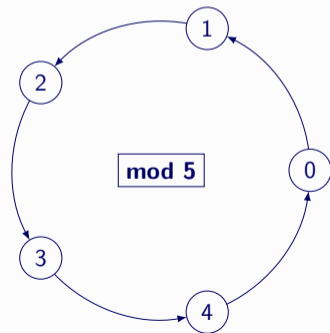
Prime field

Prime field

- $\mathbb{Z}_p/\mathbb{Z} = \{0, 1, \dots, p-1\}$
- The sum, the difference, and the product are computed, then, reduced modulo p .
- The modular inverse is computed using “extended Euclidean algorithm”.

Example: $\mathbb{Z}_5/\mathbb{Z} = \{0, 1, 2, 3, 4\}$

- $3 + 4 \equiv 2 \pmod{5}$
- $3 - 4 \equiv -1 \pmod{5} \equiv 4 \pmod{5}$
- $3 * 2 \equiv 1 \pmod{5}$
- $3^{-1} \equiv 2 \pmod{5}$



DFT over a prime field

Driving applications of prime fields

Coding theory, cryptography, and solving systems of polynomial equations.

Can we compute DFT over a prime field?

It can be proven that most attributes of DFT over \mathbb{C} also hold over prime fields.

Definition

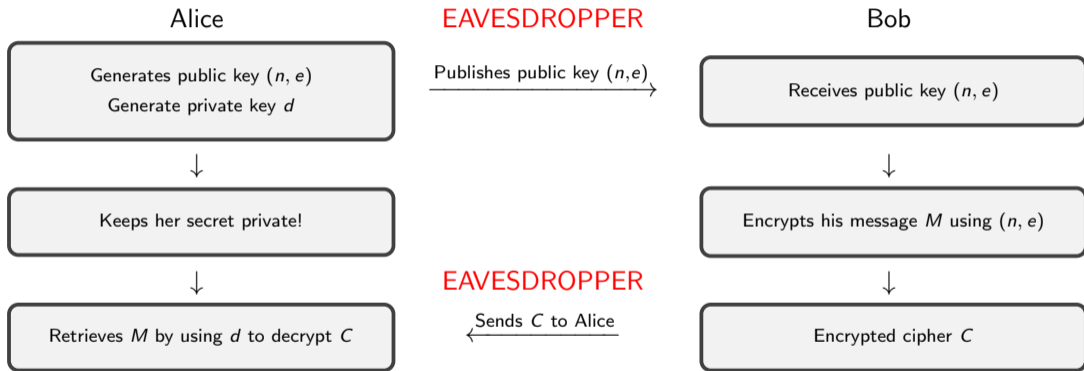
For prime field \mathbb{Z}_p , with $\omega \in \mathcal{R}$ a N -th root of unity, the following **linear map** is the DFT_N at ω :

$$\vec{a} = (a_0, \dots, a_{N-1})^T \xrightarrow{\Omega} \vec{b} = (b_0, \dots, b_{N-1})^T$$

with **matrix** Ω defined as follows:

$$\Omega = (\omega^{jk})_{0 \leq j, k \leq N-1} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

Example: RSA cryptosystem (1/3)



Example: RSA cryptosystem (2/3)

- Find three very large positive integers e , d and n such that for $0 \leq m < n$ we have:

$$\underbrace{(m^e)^d \equiv m \pmod{n}}_{\text{modular exponentiation}}$$

- The value of n is product of two randomly generated primes p and q .
- Encryption: $\underbrace{C \equiv m^e \pmod{n}}_{\text{modular exponentiation}}$
- Decryption: $\underbrace{C^d \equiv (m^e)^d \equiv m \pmod{n}}_{\text{modular exponentiation}}$

Example: RSA cryptosystem (3/3)

Alice

Generate public key (n, e) :
 $n = 7 * 11 = 77$ and $e = 13$
Generate private key $d = 7$



Keeps her secret private!



Retrieve M :
 $C^d \equiv (m^e)^d \equiv 64^7 \equiv 15 \pmod{n}$

EAVESDROPPER

Publishes public key (n, e) →

Bob

Receives public key (n, e)



Encrypt $M = 15$:
 $C \equiv m^e \equiv 15^{13} \equiv 64 \pmod{77}$



Encrypted cipher C

EAVESDROPPER

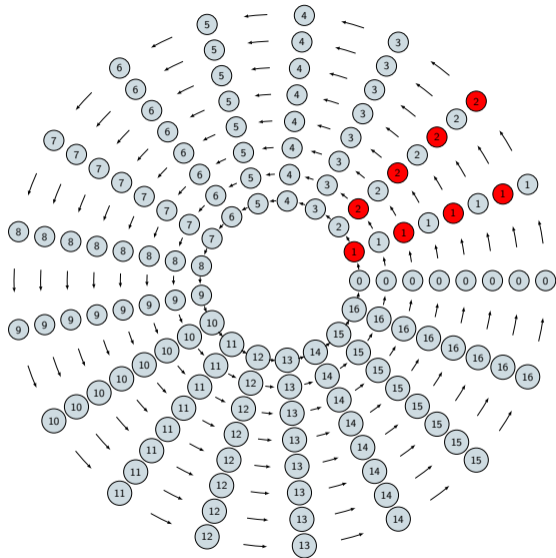
← Sends C to Alice

Example: computing DFT over \mathbb{Z}_{17} (I)

Computing DFT_8 for

$$\vec{v} = [1, 2, 1, 2, 1, 2, 1, 2]$$

- $n = 8$ and $n^{-1} = 15$,
- $\omega = 2$ (as $\omega^8 \equiv 1 \pmod{17}$), and
- $\omega^{-1} = 9$ (as $\omega\omega^{-1} \equiv 1 \pmod{17}$).

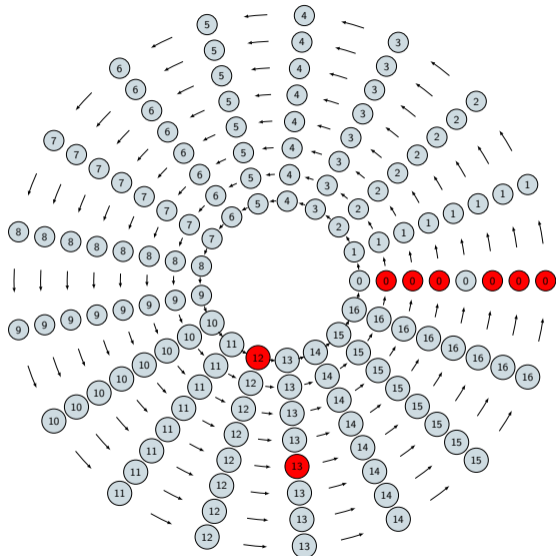


Example: computing DFT over \mathbb{Z}_{17} (II)

$$\Omega = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10} & \omega^{12} & \omega^{14} \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \omega^{15} & \omega^{18} & \omega^{21} \\ \omega^0 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \omega^{20} & \omega^{24} & \omega^{28} \\ \omega^0 & \omega^5 & \omega^{10} & \omega^{15} & \omega^{20} & \omega^{25} & \omega^{30} & \omega^{35} \\ \omega^0 & \omega^6 & \omega^{12} & \omega^{18} & \omega^{24} & \omega^{30} & \omega^{36} & \omega^{42} \\ \omega^0 & \omega^7 & \omega^{14} & \omega^{21} & \omega^{28} & \omega^{35} & \omega^{42} & \omega^{49} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 15 & 13 & 9 \\ 1 & 4 & 16 & 13 & 1 & 4 & 16 & 13 \\ 1 & 8 & 13 & 2 & 16 & 9 & 4 & 15 \\ 1 & 16 & 1 & 16 & 1 & 16 & 1 & 16 \\ 1 & 15 & 4 & 9 & 16 & 2 & 13 & 8 \\ 1 & 13 & 16 & 4 & 1 & 13 & 16 & 4 \\ 1 & 9 & 13 & 15 & 16 & 8 & 4 & 2 \end{bmatrix}$$

$$\begin{aligned} DFT_8(\vec{v}) &= \Omega \vec{v} \\ &= \Omega([1, 2, 1, 2, 1, 2, 1, 2]) \\ &= [12, 0, 0, 0, 13, 0, 0, 0] \end{aligned}$$



Example: DFT as an evaluation-interpolation scheme

$$f(x) = 1x^0 + 2x^1 + 3x^2 + 4x^3 + 5x^4 + 6x^5 + 7x^6 + 8x^7$$

$$y_0 = f(\omega^0) = 2$$

$$y_1 = f(\omega^1) = 8$$

$$y_2 = f(\omega^2) = 14$$

$$y_3 = f(\omega^3) = 6$$

$$y_4 = f(\omega^4) = 13$$

$$y_5 = f(\omega^5) = 3$$

$$y_6 = f(\omega^6) = 12$$

$$y_7 = f(\omega^7) = 1$$

$$\left. \begin{array}{l} y_0 = f(\omega^0) = 2 \\ y_1 = f(\omega^1) = 8 \\ y_2 = f(\omega^2) = 14 \\ y_3 = f(\omega^3) = 6 \\ y_4 = f(\omega^4) = 13 \\ y_5 = f(\omega^5) = 3 \\ y_6 = f(\omega^6) = 12 \\ y_7 = f(\omega^7) = 1 \end{array} \right\} \text{INTERPOLATE}((y_0, y_1, \dots, y_7)) = \text{DFT}^{-1}((y_0, y_1, \dots, y_7)) = [1, 2, 3, 4, 5, 6, 7, 8]$$

$$\begin{array}{rcccl}
 f(x) = \sum_{i=0}^{n-1} a_i x_i & \longrightarrow & \vec{a} & \xrightarrow{\text{DFT}_n(\omega)} & \text{DFT}(\vec{a}) \\
 g(x) = \sum_{i=0}^{n-1} b_i x_i & \longrightarrow & \vec{b} & \xrightarrow{\text{DFT}_n(\omega)} & \text{DFT}(\vec{b}) \\
 \downarrow & & & & \downarrow \\
 f(x)g(x) \equiv \sum_{i=0}^{n-1} c_i x_i \pmod{x^n - 1} & \longrightarrow & \vec{c} & \xleftarrow{\text{DFT}_n^{-1}(\omega^{-1})} & \text{DFT}(\vec{a}) * \text{DFT}(\vec{b})
 \end{array}$$

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

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

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

$$\begin{aligned}\vec{a} &= [1, 2, 3, 4, 5, 6, 7, 8] & \xrightarrow{\text{DFT}_8(\omega)} & \text{DFT}_8(\vec{a}) &= [2, 8, 14, 6, 13, 3, 12, 1] \\ \vec{b} &= [8, 7, 6, 5, 4, 3, 2, 1] & \xrightarrow{\text{DFT}_8(\omega)} & \text{DFT}_8(\vec{b}) &= [2, 9, 3, 11, 4, 14, 5, 16] \\ \vec{c} &= [6, 3, 8, 4, 8, 3, 6, 0] & \xleftarrow{\text{DFT}_8^{-1}(\omega^{-1})} & (\text{DFT}_8(\vec{a}) * \text{DFT}_8(\vec{b})) &= [4, 4, 8, 15, 1, 8, 9, 16]\end{aligned}$$

What do we achieve by computing DFT via FFT?

Fast Fourier transform (FFT): a fast divide-and-conquer algorithm

- Naively computing DFT over a vector of size n takes $O(n^2)$.
- Efficient DFT computation is done via **fast Fourier transform (FFT)** which takes $O(n \log n)$.
- First mentioned by Gauss (1805), popularized by IBM fellows Cooley and Tukey (1965).
- Using FFT, DFT-based polynomial multiplication leads to faster division, gcd, and factorization!

Problem

- Some computations over prime fields need high **accuracy**.
- This can only be achieved if computing is done **directly over prime fields of a large characteristic**.

Our work

- CUDA implementation of **arithmetic over $\mathbb{Z}/p\mathbb{Z}$** for p of size of at least 8 or 16 machine words.
- CUDA implementation of **FFT over $\mathbb{Z}/p\mathbb{Z}$** .
- **Theoretical and practical comparison** of our approach vs. an approach based on small prime fields.

- ① Fourier transforms
- ② Fürer's trick: beyond Cooley-Tukey factorization
- ③ Implementation challenges
- ④ Experimental Comparison

Outline

- ① Fourier transforms
- ② Fürer's trick: beyond Cooley-Tukey factorization
- ③ Implementation challenges
- ④ Experimental Comparison

Cooley-Tukey factorization formula

- Expressed in tensor notation, for $J, K > 1$ and $n = JK$, we have:

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

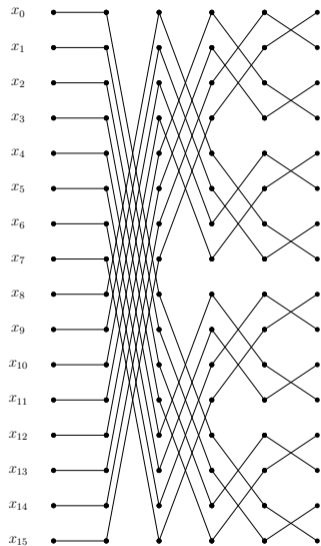
where

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

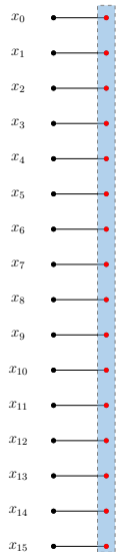
$$L_J^{JK} = \mathbf{x}[iJ + j] \mapsto \mathbf{x}[jJ + i], (0 \leq j < J, 0 \leq i < K.)$$

- Various **fast Fourier transform** algorithms can be derived from this formula.
- We can greatly benefit from **sparsity of factorized matrices**.

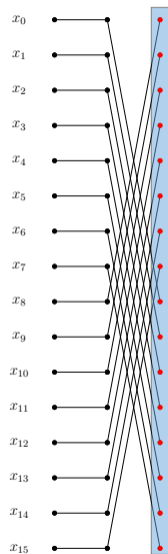
Example: FFT algorithm and data locality



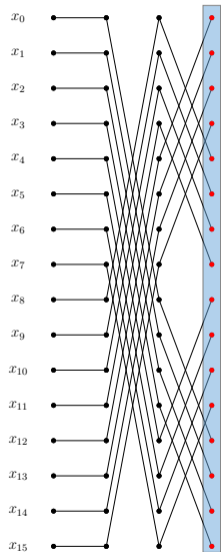
Example: FFT algorithm and data locality: radix-2 (1/5)



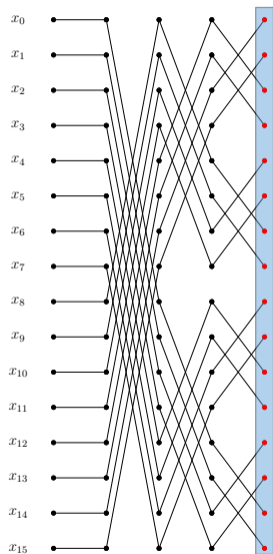
Example: FFT algorithm and data locality: radix-2 (2/5)



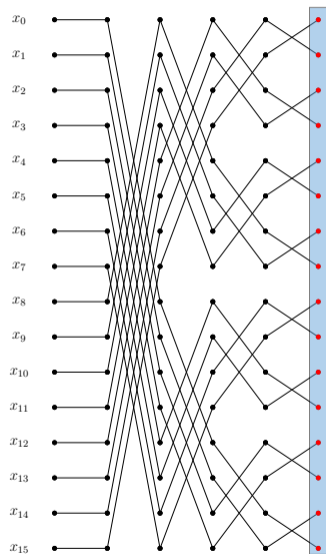
Example: FFT algorithm and data locality: radix-2 (3/5)



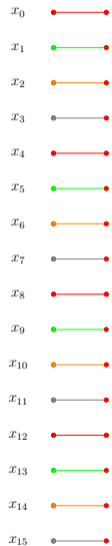
Example: FFT algorithm and data locality: radix-2 (4/5)



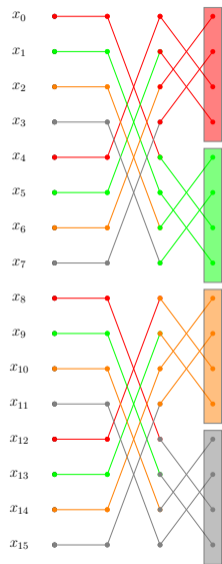
Example: FFT algorithm and data locality: radix-2 (5/5)



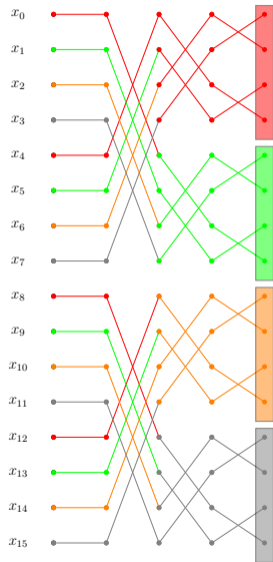
Example: FFT algorithm and data locality: blocking (1/5)



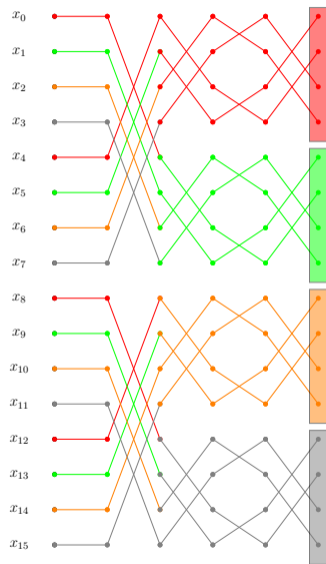
Example: FFT algorithm and data locality: blocking (2/5)



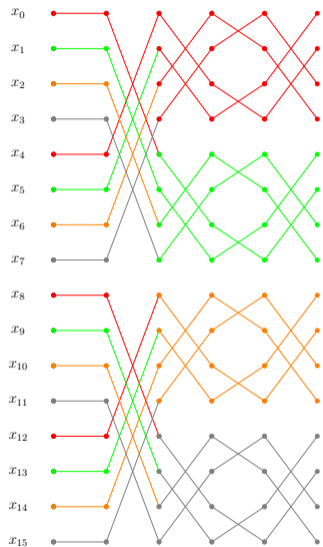
Example: FFT algorithm and data locality: blocking (3/5)



Example: FFT algorithm and data locality: blocking (4/5)



Example: FFT algorithm and data locality: blocking (5/5)



Analysis of FFT over a prime field

$$\text{DFT}_{JK} = \underbrace{(\text{DFT}_J \otimes I_K)}_{K \text{ DFT's of size } J} \underbrace{D_{J,K}}_{N \text{ mults by a power of } \omega} \underbrace{(I_J \otimes \text{DFT}_K)}_{J \text{ DFT's of size } K} L_J^{JK}$$

- For $f \in \mathbb{Z}/p\mathbb{Z}[x]$ of degree at most $N - 1$, computing $\text{DFT}_N(f)$ 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, the cost of each addition remains linear ($\mathcal{O}(k)$ word ops.), but multiplication by a power of ω becomes a bottleneck as k grows ($\mathcal{O}(M(k))$ word ops).
- Can we reduce cost of some multiplications to a cost of an addition?

Fürer's trick

- Let $N = K^e$ for some “small” K , $J = \frac{N}{K} = K^{e-1}$ and $\eta = \omega^{N/K}$.
- Assume that multiplying any element of $\mathbb{Z}/p\mathbb{Z}$ by a power of η is as cheap as an addition.
- Radix r fits in one machine word (32 or 64 bits wide depending on the device).
- This latter result holds whenever p is a prime of the form $p = r^k + 1$ (*a generalized Fermat number*).

Applying Fürer's trick to CT factorization

- Using $p = r^k + 1$, with $K = 2k$, $\text{DFT}_N(\omega)$ amounts to:

$$O(N \log_2(N) k + N \log_k(N) M(k)) \text{ word ops.}$$

- Without our assumption, the same DFT would run in $O(N \log_2(N) M(k))$ word ops.
- Using $p = r^k + 1$ results in a speedup factor of $\log(K)$.

- ① Fourier transforms
- ② Fürer's trick: beyond Cooley-Tukey factorization
- ③ Implementation challenges
- ④ Experimental Comparison

Outline

- 1 Fourier transforms
- 2 Fürer's trick: beyond Cooley-Tukey factorization
- 3 Implementation challenges
- 4 Experimental Comparison

- A **platform** developed by NVIDIA corporation.
- Provides an API for writing **scalable** parallel programs on GPUs

The CUDA execution and programming model

- **Streaming multiprocessors** (SMs): building blocks of GPUs
- **Kernel**: The function that is executed on the GPU.
- **Device**: the GPU that executes kernels.
- **Warp**: The GPU scheduler deploys every 32 threads together for execution.

The CUDA memory model

On-chip Memory:

- Registers
- L1 cache
- Shared memory

Off-chip memory:

- Global memory
- L2 cache
- Local memory

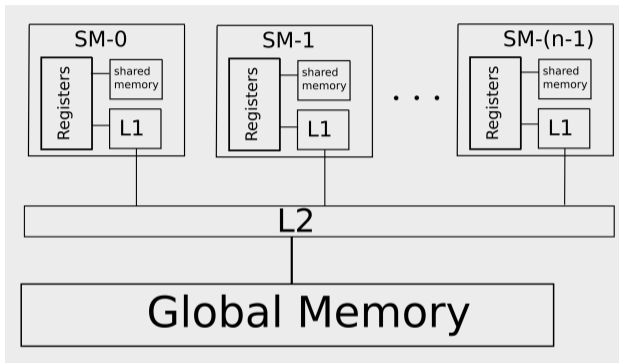


Figure: The CUDA memory model for CC 2.x and higher.

Choice of FFT algorithm:

- Using GPU as a block processor (i.e., lots of SM's working together).
- Block parallelism can be realized by $I_J \otimes \text{DFT}_K$.
- We use the six-step recursive FFT algorithm

$$\text{DFT}_N = L_K^N (I_J \otimes \text{DFT}_K) L_J^N D_{K,J} (I_K \otimes \text{DFT}_J) L_K^N.$$

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

Implementation challenges (2/6)

Parallelization of arithmetic operations.

- Initial idea: using multiple threads for computing one operation!
- High overhead will not improve the performance before a threshold! (think of $p > 2^{2048}$)
- Solution: we need to **compute operations in batches**; one big number handled by one thread.

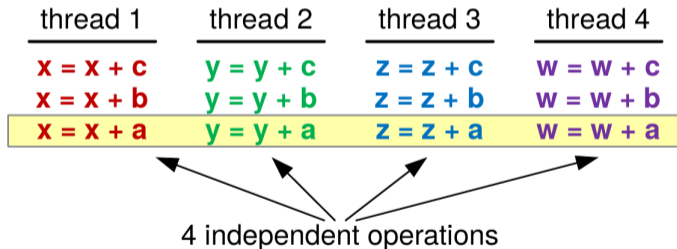


Figure: Better performance at lower occupancy, Vasily Volkov

Implementation challenges (2/6, continued)

Maximizing global memory efficiency.

- Consider \vec{X} as a vector of N elements of $\mathbb{Z}/p\mathbb{Z}$
- Assume that consecutive digits of each element are stored in adjacent memory addresses.
- View it as the row-major layout of a $N \times k$ matrix.
- Problem: performance is hurt due to increased memory overhead.
- Solution: perform a stride permutation (transposition) L_k^{kN} on all input vectors.
- Result: increasing memory (load/store) efficiency.

$$\begin{bmatrix} \vec{X}_0 \\ \vec{X}_1 \\ \vdots \\ \vec{X}_{N-1} \end{bmatrix} = \begin{bmatrix} \vec{X}_{(0,0)} & \vec{X}_{(0,1)} & \vec{X}_{(0,2)} & \cdots & \vec{X}_{(0,k-1)} \\ \vec{X}_{(1,0)} & \vec{X}_{(1,1)} & \vec{X}_{(1,2)} & \cdots & \vec{X}_{(1,k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vec{X}_{(N-1,0)} & \vec{X}_{(N-1,1)} & \vec{X}_{(N-1,2)} & \cdots & \vec{X}_{(N-1,k-1)} \end{bmatrix}_{(N \times k)} \xrightarrow{L_k^{kN}} \begin{bmatrix} \vec{X}_{(0,0)} & \vec{X}_{(1,0)} & \vec{X}_{(N-1,0)} \\ \vec{X}_{(0,1)} & \vec{X}_{(1,1)} & \vec{X}_{(N-1,1)} \\ \vec{X}_{(0,2)} & \vec{X}_{(1,2)} & \vec{X}_{(N-1,2)} \\ \vdots & \vdots & \vdots \\ \vec{X}_{(0,k-1)} & \vec{X}_{(1,k-1)} & \vec{X}_{(N-1,k-1)} \end{bmatrix}_{(k \times N)}$$

Memory-bound kernels.

- Problem: Performance is limited by frequent accesses to memory.
- Solutions:
 - minimizing memory latency (through buffering),
 - maximizing occupancy (#active warps on each SM) to hide latency, and
 - maximizing IPC (instructions per clock cycle): exploiting ILP.
 - avoiding use of shared memory (to keep occupancy high),
 - turn off L1 cache for operations that do not reuse data: all add, sub
 - keep all data on global memory

Register spilling

- Problem: Using many registers per thread can lower the occupancy,
- Solution: register-intensive kernels are broken into multiple smaller ones.

Maximizing occupancy

- Problem: For the same application, different GPUs need different kernel parameters for achieving peak performance.
- Solution: Design kernels that are oblivious to the size of a thread block.
- We choose size of thread blocks for maximizing bandwidth-related performance metrics (read and write throughput).

Effect of GPU instructions on performance

- Initial idea: implementation based on 64-bit instructions (64-bit radix should be better!)
- Problem: Compiler converts all instructions to a sequence of 32-bit equivalents, this conversion can have negative impact on the overall performance (specially, 64-bit multiplication!).
- Solution: Using 32-bit arithmetic provides more opportunities for optimization such as ILP

Argument for NOT using 64-bit arithmetic

- Let's see how GPU behaves in each case.
 - There is a cost for converting between 64-bit types to and from all other 32-bit types.
 - All 64-bit instructions have a significantly lower IPC count compared to their 32-bit counterparts.
- Table 1 is taken from Pages 85-86 of [CUDA C PROGRAMMING GUIDE](#).

Table: Number of Results per Clock Cycle per Multiprocessor

Instruction /CC	3.0, 3.2	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.0
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	160	128	128	64	128	128	64
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	32	Multiple inst.	Multiple inst.	Multiple inst.	Multiple inst.	Multiple inst.	64
32-bit bitwise AND, OR, XOR	160	160	128	128	64	128	128	64
Type conversions from and to 64-bit types	8	32	4	4	16	4	4	16
32-bit floating point add, multiply, multiply-add	192	192	128	128	64	128	128	64
64-bit floating point add, multiply, multiply-add	8	64	4	4	32	4	4	32
32-bit integer shift	32	64	64	64	32	64	64	64
compare, minimum, maximum	160	160	64	64	32	64	64	64

“Forward looking GPU integer performance”

One of the answers verified by NVIDIA on this subject:

```
" Yes, 64-bit arithmetic is accomplished via instruction sequences generated by the compiler (on all current CUDA GPUs). There is no native 64 bit integer add or multiply instruction ... "
```

Posted 06/29/2016 04:52 AM, by txbob

<https://devtalk.nvidia.com/default/topic/948014/forward-looking-gpu-integer-performance/?offset=14>

Profiling addition for $N = 2^{16}$ (random input) for $P_3 = (2^{63} + 2^{34})^8$ ($K = 16$)

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

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

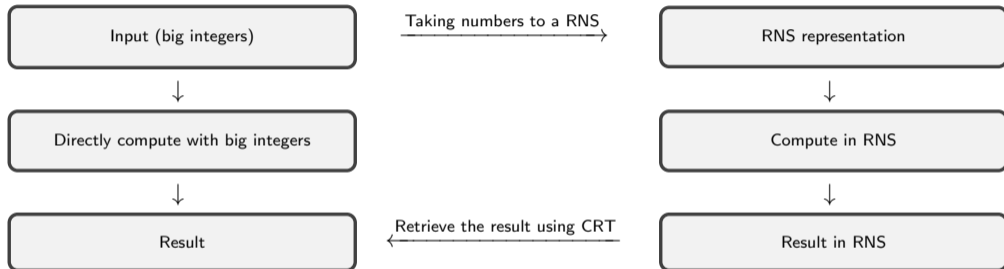
Table: Profiling results for transposed and non-transposed $\times r^s$ in $\mathbb{Z}/p\mathbb{Z}$

- ① Fourier transforms
- ② Fürer's trick: beyond Cooley-Tukey factorization
- ③ Implementation challenges
- ④ Experimental Comparison

Outline

- ① Fourier transforms
- ② Fürer's trick: beyond Cooley-Tukey factorization
- ③ Implementation challenges
- ④ Experimental Comparison

Big prime vs. small prime: RNS and CRT (1/2)



Example of RNS-CRT scheme for 4-bit numbers

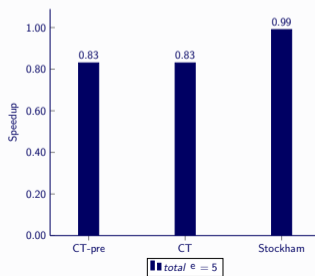
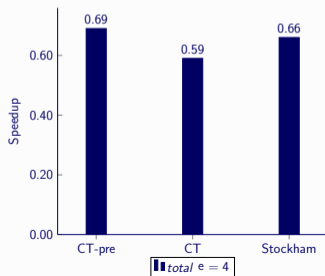
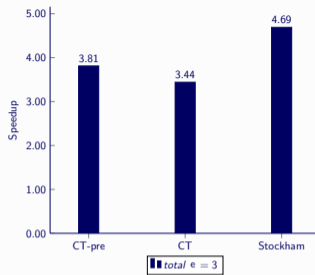
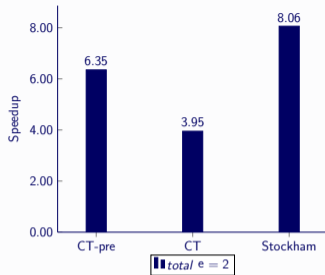
- If M is the maximum absolute value that will be used in computation, a RNS/CRT scheme needs primes $M < p_1 \times \dots \times p_e$.
- For $0 \leq x, y < B$, the $M = \max(x * y) = (2^B - 1)^2$.
- With $B = 4$, $M = (2^4 - 1)^2 = 225$, then, we can pick $p_1 = 3, p_2 = 7, p_3 = 11$ (as $225 < 3 \times 7 \times 11$).

$$\begin{array}{rcccl}
 x & = & 14 & \xrightarrow{\text{RNS } (3,7,11)} & x' & = & (2, 0, 3) \\
 y & = & 10 & \xrightarrow{\text{RNS } (3,7,11)} & y' & = & (1, 3, 10) \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 xy & = & 140 & \xrightarrow{\text{RNS } (3,7,11)} & x'y' & = & (2, 0, 8) \\
 & & & \xleftarrow{\text{CRT } (3,7,11)} & & &
 \end{array}$$

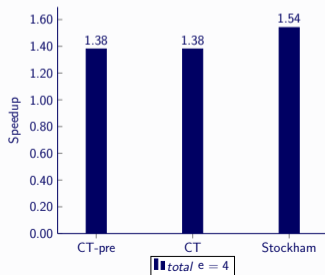
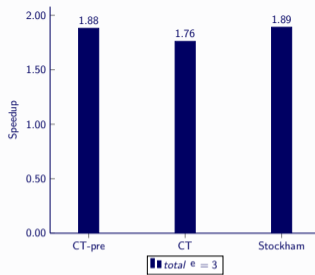
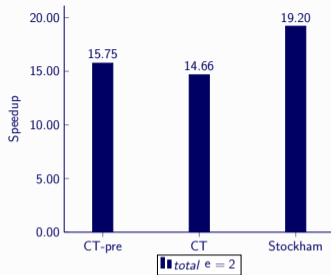
Computing FFT: big prime vs. small prime

- Small prime approach: pairwise different primes p_1, \dots, p_k
 1. compute image f_i of f in $\mathbb{Z}/p_1\mathbb{Z}[x], \dots, \mathbb{Z}/p_k\mathbb{Z}[x]$ (*projection*)
 2. compute $\text{DFT}_N(f_i)$ at ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$
 3. combine the results using the CRT (*recombination*)
- The small primes are $\frac{\text{machine-word size}}{2}$, it is fair to use $2k$ of them!
- 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 NVIDIA GTX-1080Ti (3584 CUDA cores @1.5 GHZ, Memory @5 GHZ)

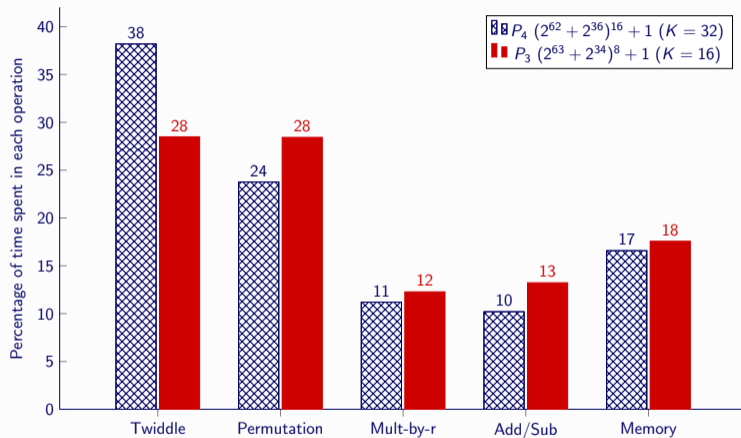
Benchmarking for $P_3 = (2^{63} + 2^{34})^8$ ($K = 16$)



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



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



Performance analysis:

- Larger p in $\mathbb{Z}/p\mathbb{Z} \Rightarrow$ a higher number of **cheap multiplications!**
- Beyond a certain point, we have more **expensive multiplications!**

Addressing performance bottlenecks

- Multiplication algorithm
- Suboptimal use of device due to 64-bit arithmetic (emulated on CUDA GPUs).
- Difficulty in adapting the code for new primes (mostly due to difficulties in multiplication).

Concluding remarks

Work in progress

- Moving to a complete implementation based on 32-bit arithmetic.
- Improving multiplication in $\mathbb{Z}/p\mathbb{Z}$ is work in progress.
- By choosing a larger prime, say $k = 32$, or $k = 64$, we hope to cover other ranges of sizes.
- We have been working on a multi-core implementation and have managed to use FFT for multiplying elements of $\mathbb{Z}/p\mathbb{Z}$ for k large enough ($k \geq 64$).

Conclusions

- Big prime field arithmetic is required by many advanced algorithms.
- Arithmetic modulo a big prime can be efficiently computed on GPUs.
- We have been the first in putting Fürer's ideas into practice and experimentally verify them.
- 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 mult by a power of r) remains a bottleneck.

Thank You!

Your Questions?

Appendix

- [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

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 $\vec{x} = (x_{k-1}, x_{k-2}, \dots, x_0)$ such that:
$$x \equiv x_{k-1} r^{k-1} + x_{k-2} r^{k-2} + \dots + x_0 \pmod{p}.$$
 1. either $x_{k-1} = r$ and $x_{k-2} = \dots = x_1 = 0$, or
 2. $0 \leq x_i < r$ for all $i = 0 \dots (k-1)$

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

Add(sub) with carry for $x, y \in \mathbb{Z}/p\mathbb{Z}$ represented by \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 , $f_x f_y \pmod{T^k + 1}$ 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:

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

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. $\omega^{N/2^k} = r$

procedure PROOT(N, r, k, g)

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

$\beta := \alpha$

$j := 1$

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

$\beta := \alpha\beta$

$j := j + 1$

end while

$\omega := g^j$

return (ω)

end procedure

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

Input:

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

Output:

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

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

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

KernelGeneralOperation $\lll \frac{N}{b}, b \ggg(\vec{X}, \vec{Y}, \vec{U}, N, k, r)$

return \vec{U}

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

local: stride := N

local: offset:=0

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

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

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

 offset:=tid +i*stride

$\vec{x}[i] := \vec{X}[\text{offset}]$

▷ Reading the digit with the index i of element \vec{X}_{tid} .

$\vec{y}[i] := \vec{Y}[\text{offset}]$

▷ Reading the digit with the index i of element \vec{Y}_{tid} .

end for

$\vec{u} := \text{DeviceGeneralOperation}(\vec{x}, \vec{y}, k, r)$.

▷ each thread computing one element of the final result.

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

 offset:=tid +i*stride

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

end for

return

▷ End of Kernel

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

```
1: local:  $m := e$  where  $N = K^e$ ,  $j := 0$ 
2: if  $e \bmod 2 = 1$  then
3:   HostGeneralStridePermutation( $\vec{X}, \vec{Y}, K^1, N, k, s, b$ )
4: end if
5: for ( $0 \leq i < m$  by 2) do
6:   HostDFTK2( $\vec{X}, \vec{\Omega}, N, K, k, s, r$ )
7:   KernelTwiddleMultiplication( $\vec{X}, \vec{\Omega}, N, K, k, s := 2, r$ )
8:   HostGeneralStridePermutation( $\vec{X}, \vec{Y}, K^2, N, k, s, b$ )
9:    $\vec{X}[0 : kN - 1] := \vec{Y}[0 : kN - 1]$ 
10:  HostDFTK2( $\vec{X}, \vec{\Omega}, N, K, k, s, r$ )
11:  HostGeneralStridePermutation( $\vec{X}, \vec{Y}, K^2, N, k, s, b$ )
12:   $\vec{X}[0 : kN - 1] := \vec{Y}[0 : kN - 1]$ 
13: end for
```

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

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

Mixed-radix representation

- For pairwise different primes p_1, \dots, 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, \dots, b_s) is *mixed-radix representation* of $n \in \mathbb{Z}$:

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

Reconstructing n from (b_1, b_2, \dots, b_s)

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

MRR or CRT?

- The CRT defines a ring isomorphism:

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

- 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.

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. $\rightarrow \mathcal{O}(sN \log_2(N))$
- Compute a MRR on results of small prime FFTs $\rightarrow \mathcal{O}(sNk^2)$

Example: computing DFT-16 based on DFT-2

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

$$\text{DFT}_{16} = L_2^{16}(I_8 \otimes \text{DFT}_2)L_8^{16}D_{2,8}^{16}(I_2 \otimes \text{DFT}_8)L_2^{16},$$

$$\text{DFT}_8 = L_2^8(I_4 \otimes \text{DFT}_2)L_4^8D_{2,4}^8(I_2 \otimes \text{DFT}_4)L_2^8,$$

$$\text{DFT}_4 = L_2^4(I_2 \otimes \text{DFT}_2)L_2^4D_{2,2}^4(I_2 \otimes \text{DFT}_2)L_2^4,$$

- Following multiplications by twiddle factors are required:

- DFT_{16} with $\omega_0 = \omega^{N/K} = r$,
- DFT_8 needs $\omega_1 = \omega^{(N/K)^2} = r^2$,
- DFT_4 needs $\omega_2 = \omega^{(N/K)^4} = r^4$.

- We have:

$$D_{2,8}^{16} = (1, 1, 1, 1, 1, 1, 1, 1, r^0, r^1, r^2, r^3, r^4, r^5, r^6, r^7),$$

$$D_{2,4}^8 = (1, 1, 1, 1, r^0, r^2, r^4, r^6),$$

$$D_{2,2}^4 = (1, 1, r^0, r^4).$$

- $x, y \in \mathbb{Z}/p\mathbb{Z}$ 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 $i = 0, \dots, k - 1$,
- 2: let $z_k = 0$,
- 3: for $i = 0, \dots, k - 1$, 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}, \dots, z_0) ,
- 5: if $z_k = 1$ and $z_{k-1} = \dots = z_0 = 0$, then let $z_{k-1} = r$ and return (z_{k-1}, \dots, 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 $z_0 = \dots = z_{i_0-1} = r - 1$ and return (z_{k-1}, \dots, 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 $f_x f_y \bmod T^k + 1$ 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 \vec{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

Example: RNS and CRT

p_i	$N_i = \frac{p_1 p_2 p_3}{p_i}$	$N_i \pmod{p_i}$	$M_i \equiv \frac{1}{N_i} \pmod{p_i}$	$N_i M_i$
3	77	2	2	154
7	33	5	3	99
11	21	10	10	210

$$\begin{array}{rcl}
 x = 14 & \xrightarrow{\text{RNS } (3,7,11)} & x' = (2, 0, 3) \\
 y = 10 & \xrightarrow{\text{RNS } (3,7,11)} & y' = (1, 3, 10) \\
 \downarrow \quad \downarrow & & \downarrow \quad \downarrow \\
 xy = 140 & \xrightarrow{\text{RNS } (3,7,11)} & x'y' = (2, 0, 8) \\
 & \xleftarrow{\text{CRT } (3,7,11)} &
 \end{array}$$