Faster arithmetic for number-theoretic transforms

David Harvey

University of New South Wales

20th December 2011, FLINT/Sage Days, University of Warwick

Plan for talk

1. Review number-theoretic transform (NTT)
2. Discuss typical butterfly algorithm
3. Improvements to butterfly algorithm
4. Performance data

The number-theoretic transform (NTT)

NTT $=$ discrete Fourier transform (DFT) over a finite field.

We will assume:

- transform length is $N = 2^n$.
- base field is $\mathbf{F}_p$ where p is prime and $p = 1 \pmod{N}$, so that $\mathbf{F}_p$ contains N-th roots of unity.
- p fits into a single machine word, i.e. $p < \beta$, where β describes the word size, for example $\beta = 2^{64}$.

The number-theoretic transform (NTT)

Definition of NTT:

Input vector $(a_0, \ldots, a_{N-1}) \in \mathbf{F}_p^N$.

Let ω be an N-th root of unity in $\mathbf{F}_p$.

Output is the vector $(b_0, \ldots, b_{N-1})$ where

$$b_j = \sum_{0 \leq i < N} \omega^{ij} a_i.$$

Computing NTT is equivalent to evaluating the polynomial

$$f(x) = a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$$

simultaneously at the points

$$1, \omega, \omega^2, \ldots, \omega^{N-1}.$$

The number-theoretic transform (NTT)

Naive algorithm for NTT has complexity $O(N^2)$.

(Complexity $=$ number of ring operations in $\mathbf{F}_p$.)

Fast Fourier transform (FFT) has complexity $O(N \log N)$.

Applications:

- Fast polynomial multiplication in $\mathbf{F}_p[X]$.
- Fast polynomial multiplication in $\mathbf{Z}[X]$ (via Chinese remainder theorem), $(\mathbf{Z}/n\mathbf{Z})[X]$, $\mathbf{F}_q[X]$, etc.
- Other polynomial operations: reciprocal, division, GCD, square root, composition, factoring, etc.

Real-life example: Victor Shoup's NTL library, very popular in computational number theory and cryptography, uses the fast NTT as the building block for all of these operations.

The number-theoretic transform (NTT)

```
Algorithm 1: Simple FFT pseudocode
Input: $N = 2^n$, $(a_0, \ldots, a_{N-1}) \in \mathbf{F}_p^N$
1 for $i \leftarrow 0, 1, \ldots, n-1$ do
2     for $0 \le j < 2^i$ do
3         for $0 \le k < 2^{n-i-1}$ do
4             $s \leftarrow j2^{n-i} + k$
5             $t \leftarrow j2^{n-i} + k + 2^{n-i-1}$
6             $w \leftarrow (\omega^{2^i})^k$
7             $\begin{bmatrix} a_s \\ a_t \end{bmatrix} \leftarrow \begin{bmatrix} a_s + a_t \\ w(a_s - a_t) \end{bmatrix}$    (butterfly)
```

Output is in-place, in bit-reversed order.

Butterflies

Consider the butterfly operation

$$\begin{bmatrix} X \\ Y \end{bmatrix} \mapsto \begin{bmatrix} X + Y \\ W(X - Y) \end{bmatrix}.$$

Algorithm performs $O(N \log N)$ of these.

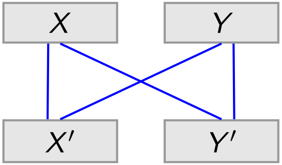
$O(N)$ of them have $W = 1$; we will concentrate on $W \neq 1$ case.

We will assume indexing and locality is taken care of, and assume W comes from table lookup.

Our focus is the following problem: given X, Y and W, how to most efficiently compute $X + Y$ and $W(X - Y)$?

Butterflies

$\cong$



$$X' = X + Y$$
$$Y' = W(X - Y)$$

Butterflies

Primitive operations allowed (modelled on typical modern instruction sets):

- addition/subtraction of single words (modulo β)
- comparison of single words
- multiplication of single words (modulo β)
- wide multiplication, i.e. given $U, V \in [0, \beta)$, compute UV in the form $UV = P_1\beta + P_0$ where $P_0, P_1 \in [0, \beta)$.

For expository purposes I'll also temporarily assume we have:

- double-word division, i.e. given $U \in [0, \beta^2)$ and $M \in [0, \beta)$, compute $Q = \lfloor U/M \rfloor$ and $R = U - QM$

Butterflies

Algorithm 2: Simple butterfly routine

Input: $X, Y, W \in [0, p)$, assume $p < \beta/2$

Output: $X' = X + Y \bmod p$

$Y' = W(X - Y) \bmod p$

```
1 $X' \leftarrow X + Y$
2 if $X' \geq p$ then $X' \leftarrow X' - p$    (now $X' = X + Y \bmod p$)
3 $T \leftarrow X - Y$
4 if $T < 0$ then $T \leftarrow T + p$    (now $T = X - Y \bmod p$)
5 $U = U_1\beta + U_0 \leftarrow TW$    (wide multiplication)
6 $Y' \leftarrow U \bmod p$    (double-word division)
```

This is inefficient because hardware division is *slow*.

Modular multiplication

How can we compute $TW \bmod p$ more efficiently?

There are several well-known methods that replace the division by multiplication(s).

Basic strategy:

- Estimate a ‘quotient’ Q.
- Multiply Q by p.
- Subtract Qp from TW to obtain candidate remainder R.
- Add/subtract small multiple of p to adjust remainder into standard interval $[0, p)$.

Modular multiplication

Algorithm 3: Shoup's modular multiplication algorithm

```
Input: $T, W \in [0, p)$, assume $p < \beta/2$
       precomputed $W' = \lfloor W\beta/p \rfloor$
Output: $R = TW \bmod p$
1 $Q \leftarrow \lfloor W'T/\beta \rfloor$    (high part of product $W'T$)
2 $R \leftarrow (WT - Qp) \bmod \beta$    (two low products)
3 if $R \geq p$ then $R \leftarrow R - p$
```

Note: W is invariant. This is reasonable for the NTT, since each transform uses the same roots of unity.

W' is a scaled approximation to W/p.

Q is an approximation to WT/p.

Claim: $WT - Qp \in [0, 2p)$. Thus the R computed in line 2 is exactly $WT - Qp$. The last line adjusts the remainder into $[0, p)$.

Modular multiplication

Proof of claim: we have

$$0 \le \frac{W\beta}{p} - W' < 1 \qquad \text{and} \qquad 0 \le \frac{W'T}{\beta} - Q < 1.$$

Multiply by Tp/β and p respectively, and add:

$$0 \le WT - Qp < 2p.$$

In other words, Q is either the correct quotient or too large by 1.

Modular multiplication

Algorithm 4: Shoup butterfly

Input: $X, Y, W \in [0, p)$, assume $p < \beta/2$
precomputed $W' = \lfloor W\beta/p \rfloor$

Output: $X' = X + Y \bmod p$, $Y' = W(X - Y) \bmod p$

```
1 $X' \leftarrow X + Y$
2 if $X' \geq p$ then $X' \leftarrow X' - p$
3 $T \leftarrow X - Y$
4 if $T < 0$ then $T \leftarrow T + p$
5 $Q \leftarrow \lfloor W'T/\beta \rfloor$
6 $Y' \leftarrow (WT - Qp) \bmod \beta$
7 if $Y' \geq p$ then $Y' \leftarrow Y' - p$
```

This is essentially the algorithm used in NTL.

Removing adjustment steps

Our goal: to remove as many adjustment steps

$$\text{“}\textbf{if}\text{ (some condition) }\textbf{then}\ Z \leftarrow Z \pm p\text{”}$$

as possible.

Each adjustment requires several machine instructions: a conditional move, and several other instructions to set it up.

These adjustments can account for a significant proportion of the total execution time.

Especially on modern processors with very fast multipliers!

Removing adjustment steps

One adjustment is easy to remove.

In Shoup's algorithm for computing $TW \bmod p$, we assumed that $T \in [0, p)$.

But in fact the algorithm works perfectly well for any $T \in [0, \beta)$.

So we can simply skip the adjustment for T.

(I don't know where this was first noticed, but certainly Fabrice Bellard knew this in 2009 when he computed π to 2.7 trillion decimal places using a souped-up desktop machine. NTL does not use this trick.)

Removing adjustment steps

Algorithm 5: Shoup butterfly, one adjustment removed

Input: $X, Y, W \in [0, p)$, assume $p < \beta/2$
precomputed $W' = \lfloor W\beta/p \rfloor$

Output: $X' = X + Y \bmod p$, $Y' = W(X - Y) \bmod p$

```
1 $X' \leftarrow X + Y$
2 if $X' \geq p$ then $X' \leftarrow X' - p$
3 $T \leftarrow X - Y + p$    (now $T \equiv X - Y \bmod p$, and $T \in [0, 2p)$)
4 $Q \leftarrow \lfloor W'T/\beta \rfloor$
5 $Y' \leftarrow (WT - Qp) \bmod \beta$
6 if $Y' \geq p$ then $Y' \leftarrow Y' - p$
```

Removing adjustment steps

What about the last adjustment for Y'?

Apparently the only way to avoid it is to somehow get the quotient $\lfloor TW/p \rfloor$ correct the first time.

But I don't know of any way to get the correct quotient efficiently.

(This is part of the reason that hardware division is so slow!)

Removing adjustment steps

But there is another way... **don't perform the adjustment**!

Then the butterfly outputs lie in $[0, 2p)$.

Relax the algorithm to allow the *inputs* to lie $[0, 2p)$.

In other words, the entire FFT algorithm operates on 'non-canonical residues'. Each element of $\mathbf{F}_p$ has two possible representatives, one in $[0, p)$ and one in $[p, 2p)$.

If desired, a final pass at the end reduces the output into $[0, p)$.

We need $p < \beta/4$ for this scheme to work.

Removing adjustment steps

Algorithm 6: Shoup butterfly, two adjustments removed

Input: $X, Y \in [0, 2p)$, $W \in [0, p)$, assume $p < \beta/4$
precomputed $W' = \lfloor W\beta/p \rfloor$

Output: $X' \equiv X + Y \bmod p$, $Y' \equiv W(X - Y) \bmod p$
$X', Y' \in [0, 2p)$

```
1 $X' \leftarrow X + Y$
2 if $X' \geq 2p$ then $X' \leftarrow X' - 2p$
3 $T \leftarrow X - Y + 2p$
4 $Q \leftarrow \lfloor W'T/\beta \rfloor$
5 $Y' \leftarrow (WT - Qp) \bmod \beta$
```

I don't know how to remove the adjustment in line 2.

The inverse butterfly

Consider the 'inverse butterfly'

$$\begin{bmatrix} X \\ Y \end{bmatrix} \mapsto \begin{bmatrix} X + WY \\ X - WY \end{bmatrix}.$$

This is the inverse of the ordinary butterfly (after replacing W by W^{-1}, and dividing by 2).

The inverse butterfly appears naturally in the inverse FFT algorithm.

One can also implement a forward FFT using the inverse butterfly by switching from decimation-in-frequency to decimation-in-time.

The inverse butterfly

A similar trick applies to the inverse butterfly, but now we use representatives in $[0, 4p)$:

Algorithm 7: Shoup inverse butterfly, two adjustments removed

Input: $X, Y \in [0, 4p)$, $W \in [0, p)$, assume $p < \beta/4$
Output: $X' \equiv X + WY \bmod p$, $Y' \equiv X - WY \bmod p$
$X', Y' \in [0, 4p)$

```
1 if $X \geq 2p$ then $X \leftarrow X - 2p$    (now $X \in [0, 2p)$)
2 $U \leftarrow WY \bmod p$ with $0 \leq U < 2p$
  (Shoup multiplication without adjustment)
3 $X' \leftarrow X + U$
4 $Y' \leftarrow X - U + 2p$
```

Montgomery multiplication

Algorithm 8: Montgomery's modular multiplication algorithm

Input: $T, W \in [0, p)$, assume $p < \beta/2$
precomputed $J = p^{-1} \bmod \beta$ and $W' = \beta W \bmod p$

Output: $R = TW \bmod p$

```
1 $U = U_1\beta + U_0 \leftarrow TW'$    (wide product)
2 $Q \leftarrow U_0 J \bmod \beta$    (low product)
3 $H \leftarrow \lfloor Qp/\beta \rfloor$    (high product)
4 $R \leftarrow U_1 - H$
5 **if** $R < 0$ **then** $R \leftarrow R + p$
```

In this algorithm Q is a 2-adic approximation to TW'/p.

Proof of correctness: we have $Qp = H\beta + U_0$, so

$$U_1 - H = (U - Qp)/\beta \equiv TW'/\beta \equiv TW \bmod p.$$

Moreover $U_1, H \in [0, p)$, so the first guess for R lies in $(-p, p)$.

Montgomery multiplication

Algorithm 9: Montgomery butterfly

Input: $X, Y, W \in [0, p)$, assume $p < \beta/2$
precomputed $J = p^{-1} \bmod \beta$ and $W' = \beta W \bmod p$

Output: $X' = X + Y \bmod p$, $Y' = W(X - Y) \bmod p$

```
1 $X' \leftarrow X + Y$
2 if $X' \geq p$ then $X' \leftarrow X' - p$
3 $T \leftarrow X - Y$
4 if $T < 0$ then $T \leftarrow T + p$
5 $U = U_1\beta + U_0 \leftarrow TW'$
6 $Q \leftarrow U_0 J \bmod \beta$
7 $H \leftarrow \lfloor Qp/\beta \rfloor$
8 $Y' \leftarrow U_1 - H$
9 if $Y' < 0$ then $Y' \leftarrow Y' + p$
```

Montgomery multiplication

Just as in the Shoup case, we can remove two adjustments:

Algorithm 10: Montgomery butterfly, two adjustments removed

Input: $X, Y \in [0, 2p)$, $W \in [0, p)$, assume $p < \beta/4$
precomputed $J = p^{-1} \bmod \beta$ and $W' = \beta W \bmod p$

Output: $X' \equiv X + Y \bmod p$, $Y' \equiv W(X - Y) \bmod p$
$X', Y' \in [0, 2p)$

```
1 $X' \leftarrow X + Y$
2 **if** $X' \geq p$ **then** $X' \leftarrow X' - 2p$
3 $T \leftarrow X - Y + 2p$    (adjustment skipped)
4 $U = U_1\beta + U_0 \leftarrow TW'$
5 $Q \leftarrow U_0 J \bmod \beta$
6 $H \leftarrow \lfloor Qp/\beta \rfloor$
7 $Y' \leftarrow U_1 - H + p$    (adjustment skipped)
```

Performance data

Next slide compares performance of several algorithms on AMD Opteron (K8 model 8218) and Intel Core 2 Duo (Penryn SL9600):

- NTL FFT routine (version 5.5.2, after running tuning wizard). Excludes bit-reversal and generating root tables. NTL uses a 50-bit modulus for historical reasons.
- C/C++ implementation of Barrett, Shoup, Montgomery algorithms, both ‘plain’ (performs all three reductions) and ‘modified’ (only one reduction). Modulus is as close to 64 bits as allowed. Wide multiplication uses inline assembly macros.
- Assembly implementation of ‘modified Shoup’, one optimised for each processor.

Transforms are length $2^{12} = 4096$, all in L1 cache.

Table shows cycles per butterfly, assuming exactly $\frac{1}{2}N \lg_2 N$ butterflies.

Performance data

	AMD K8 (Opteron)	Intel Core 2
NTL	16.3[†]	16.5
Plain Barrett	16.8	20.7
Modified Barrett	12.7	12.4
Plain Shoup	13.4	11.8
Modified Shoup	11.2	10.5
Plain Montgomery	12.7	13.2
Modified Montgomery	10.7	11.5
Assembly*	6.0	8.0

[†] Transform length only 2^{11}, seemed to be slightly faster.

* Assembly cycle counts are exact, based on measurements of inner loop. Cycles per butterfly are slightly higher.

Performance data

Personal communication from Niels Möller, Torbjörn Granlund, Tommy Färnquist (GMP developers, highly experienced assembly programmers):

The fastest they can make ‘plain Shoup’ on AMD K8 is about 8.5 cycles per butterfly in the inner loop (more precisely 8.0 if the root of unity is invariant over the loop, 9.0 if it varies over the loop).

So here we are 1.4x faster!

Performance data

The cycle counts for the assembly implementations are optimal in the following sense:

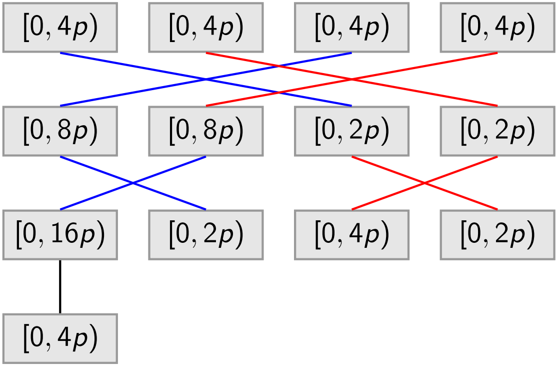
On the AMD chip, maximum integer multiply throughput is 2 cycles per multiply. Each butterfly has 3 multiplications, so this implementation saturates the multiplier.

On the Intel chip, maximum throughput is 2 cycles for the low word of a product, and 4 cycles for the wide product. The Shoup algorithm needs one wide multiply and two low multiplies: total is 8 cycles per butterfly, again we saturate the multiplier.

This also suggests that on chips with even faster multipliers (relative to other operations), such as newer Nehalem or Sandy Bridge, it might be worth trying to eliminate even more adjustments.

Radix-4

Example: assume $p < \beta/16$, represent values in $[0, 4p)$. Use a radix-4 decomposition of FFT, i.e. decompose into transforms of length 4, composed of 4 ordinary butterflies each. Then we only need one adjustment for each such transform:



Integer multiplication

Another application for NTTs is large integer multiplication.

GMP developers have been working on this for a while, nothing released.

I wrote my own from scratch, in pure C, optimised for multicore performance and low memory usage (inputs are overwritten!).

Performance data for 1.2GB inputs ($2 \times$ quad-core 3GHz Intel Xeon X5472, 16 GB RAM):

	Wall time (s)	Peak memory (GB)
GMP 5.0.2	266	11.1
MPIR 2.4.0	340	9.9
NTT 0.1.0 (1 core)	246	7.0
NTT 0.1.0 (8 cores)	45	7.0

Thank you!