Notes on Computer Multiplication

Rui-Jie Fang

We detail computer algorithms that perform $x \cdot y$ with x, y being two n-bit vectors along with polynomial multiplication.

1 Schoolbook Multiplication

Let $x = \sum_{k=0}^{n-1} a_k 2^k$ $y = \sum_{k=0}^{n-1} b_k 2^k$ where $a_k, b_k \in \{0, 1\}$. Schoolbook multiplication starts by computing all partial sums of the form xb_k and ya_k and adding them together. Each 1-bit multiplication for the partial sum can be implemented as a single AND-operation. The time complexity is hence $O(n^2) + cn$ because of the n^2 partial summing operations and the remaining step that adds all partial sums together.

2 Karatsuba Multiplication

Is there an algorithm that can perform multiplication faster than n^2? Kolmogorov thought no; however, Kolmogorov's student Karatsuba went on a quest for a faster algorithm anyway. We start by designing a divide-and-conquer scheme recursion for multiplication. If we express x, y in terms of two $n/2$-bit vectors, one containing the low-bits, and another containing the high-bits, we have:

$$x = x_1 2^{n/2} + x_0$$

$$y = y_1 2^{n/2} + y_0$$

Now the recurrence can be just based upon the expansion of $x \cdot y$:

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + x_1 y_0 2^{n/2} + x_0 y_1 2^{n/2} + x_0 y_0 \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0 \end{aligned}$$

At each level we have four

The runtime recurrence is now expressed as:

$$T(n) = 4T(n/2) + cn$$

(The extra linear factor comes from the cost of addition) Which is $O(n^{\log_2 4}) = O(n^2)$. This is not good enough; can we make the recursion tree less bushy?

Observe:

$$(x_0 + x_1)(y_0 + y_1) = x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1$$

$$x_0y_1 + x_1y_0 = (x_0 + x_1)(y_0 + y_1) - x_0y_0 - x_1y_1$$

Let $A = x_1y_1$, $B = x_0y_0$, $C = (x_0 + x_1)(y_0 + y_1)$. Now we can rewrite our recurrence as

$$\begin{aligned} xy &= x_1y_12^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0 \\ &= x_1y_12^n + ((x_0 + x_1)(y_0 + y_1) - x_1y_1 - x_0y_0)\, 2^{n/2} + x_0y_0 \\ &= A2^n + (C - A - B)2^{n/2} + x_0y_0 \end{aligned}$$

Note how there is now only three recursive calls which are A, B, C; we have successfully traded a single recursive call for two additions and two subtractions.

The runtime for the new "less bushy" algorithm is now:

$$T(n) = 3T(n/2) + cn$$

Which becomes: $O(n^{\log_2 3}) = O(n^{1.59})$. The algorithm's pseudo code is presented below:

Algorithm 1 (Karatsuba Multiplication)**.**

```
Karatsuba(x, y, n):
    if (n = 1): return x ∧ y;
    else:
        Write x = x_1 2^{n/2} + x_0, y = y_1 2^{n/2} + y_0;
        d_0 := x_0 + x_1; d_1 := y_0 + y_1;
        A := Karatsuba(x_1, y_1, n/2);
        B := Karatsuba(x_0, y_0, n/2);
        C :=
        Karatsuba(d_0, d_1, max{sizeof(d_0), sizeof(d_1)}); (: at most n/2+
        1 bits :)
        return A2^n + (C − A − B)2^{n/2} + B;
```

Remark: The multiplications by 2^n and $2^{n/2}$ should be implemented as bitmask operations.

2.1 Subtractive Karatsuba

The main dilemma presented by algorithm 1 is the possibility that d_0 or d_1 may be of $n/2+1$ bit size due to carries. We may solve the dilemma by using a subtractive algorithm. The subtractive algorithm works by computing C in a different way:

$$(|x_1 - x_0|)(|y_1 - y_0|) = \begin{cases} x_1y_1 + x_0y_0 - (x_1y_0 + x_0y_1) & x_1 - x_0 \geq 0, y_1 - y_0 \geq 0 \,(1) \\ x_1y_0 + x_0y_1 - (x_1y_1 + x_0y_0) & x_1 - x_0 \geq 0, y_1 - y_0 < 0 \,(2) \\ x_0y_1 + x_1y_0 - (x_0y_0 + x_1y_1) & x_1 - x_0 < 0, y_1 - y_0 \geq 0 \,(3) \\ x_0y_0 + x_1y_1 - (x_0y_1 + x_1y_0) & x_1 - x_0 < 0, y_1 - y_0 < 0 \,(4) \end{cases}$$

It is not hard to see that cases (1), (4) are equal and (2), (3) are equal. Simplifying, we find that for cases (1), (4), we have:

$$x_1y_0 + x_0y_1 = -(|x_1 - x_0| \cdot |y_1 - y_0|) + x_1y_1 + x_0y_0$$

For cases (2), (3), we have:

$$x_1y_0 + x_0y_1 = (|x_1 - x_0| \cdot |y_1 - y_0|) + x_1y_1 + x_0y_0$$

Thus if we take the sign for $x_1 - x_0$, $y_1 - y_0$ and the sign is stored as a single bit, we simply make the result of $C = |x_1 - x_0| \cdot |y_1 - y_0|$ multiply the product of the two sign bits (Similar to XOR) and subtract the resulting vector from $A + B$ (We want $A + B - |C|$ when C is positive (cases (1), (4)), and $A + B + |C|$ when C is negative (cases (2), (3))).

We therefore have:

Algorithm 2 (Subtractive Karatsuba Multiplication)**.**

```
SubtractiveKaratsuba(x, y, n):
    if (n = 1): return x ∧ y;
    else:
        Write x = x_1 2^{n/2} + x_0, y = y_1 2^{n/2} + y_0;
        k_0 := |x_1 − x_0|; k_1 := |y_1 − y_0|;
        s_0 := sign(x_1 − x_0); s_1 = sign(y_1 − y_0);
        A := SubtractiveKaratsuba(x_1, y_1, n/2);
        B := SubtractiveKaratsuba(x_0, y_0, n/2);
        C := SubtractiveKaratsuba(k_0, k_1, n/2);
        return A · 2^n + B + (A + B − s_0 s_1 C) 2^{n/2};
```

We have now coined most of the details for Karatsuba multiplication, except how we express x, y as x_0, y_0, x_1, y_1 is still vague. For word-sized integers, we can simply express the computation of low-half-word and high-half-word as a fixed-sized bitmask; the example below is for 32bit integers.

$$\mathtt{x_0} \coloneqq \mathtt{x\&0x0000ffff};\ \mathtt{x_1} \coloneqq \mathtt{x\&0xffff0000};$$

$$\mathtt{y}_0 : \mathtt{y\&0x0000ffff}; \mathtt{y}_1 \coloneqq \mathtt{y\&0xffff0000};$$

But for multiprecision arithmetic, we may use regular division and remainder operations (division serves as a high-bit mask; remainder serves as a low-bit mask):

$$x_0 \coloneqq \mathbf{div}(x, \beta^{n/2});\ x_1 \coloneqq x \mod \beta^{n/2};$$

$$y_0 \coloneqq \mathbf{div}(x, \beta^{n/2});\ y_1 \coloneqq y \mod \beta^{n/2};$$

Where β is the base we express our number in. Why? Because division is like right-shifts that take out all “slots” on the right-half of the vector (corresponding to the high-bit mask); If all the slots being taken out have 0’s, then we have no remainder; all remainder are the least-significant bits lying on the right of the $n/2$-th (middle) slot, which is preserved by the remainder operation (corresponding to the low-bit mask). For more efficient implementation we may implement the **div** operation as $n/2$ right shifts (i.e. removing the rightmost $n/2$ words) and mod operation as copying out the $n/2$ least significant words (i.e. copying out the right half of the vector and discarding the left half).

3 Karatsuba with Unequal Sizes

We considered Karatsuba multiplication with two n-bit vectors in section 2. Now we would like to consider the multiplication of an m-bit vector with an n-bit vector with $n \leq m$. There are two ideas: 1) We can split the two vectors into an equal amount of smaller chunks (but of different sizes); 2) We can split the two vectors into an unequal amount of smaller chunks.

TODO: Discuss OddEvenKaratsuba

Algorithm 3 (OddEvenKaratsuba for Unbalanced Multiplication)**.**

```
OddEvenKaratsuba(X, Y, m, n):
Input: X of size m, Y of size n, m ≥ n ≥ 1;
Output: X · Y;

    if (n = 1): return VectorScalarProduct(X, Y, m);
    else:
        k_0 := floor(m/2); k_1 := floor(n/2);
        write x = x_0+
```

4 Speed Up Karatsuba by Accumulation and Mutual Recursion

5 Karatsuba with Improved Space Efficiency

6 Karatsuba with Less Operations

7 Toom-Cook Multiplication

If we look at Karatsuba multiplication, the part that's variable in multiprecision arithmetic is its base. We chose base 2 and represented the number as a high-2^k-bit vector and a low-2^k-bit vector. Toom-Cook, an algorithm invented by Andrei Toom and improved by Stephen Cook, is an algorithm that implicitly also suits polynomial multiplication that generalizes Karatsuba to a base k. Therefore, we say that Karatsuba really is Toom-Cook when $k = 2$. The standard Toom-Cook algorithm is TookCook3, when we split up the numbers x, y by 3.