2015 IEEE 22nd Symposium on Computer Arithmetic

Efficient divide-and-conquer multiprecision integer division

William Hart
Dept. of Mathematics
TU Kaiserslautern
Kaiserslautern, Germany
Email: goodwillhart@googlemail.com

***Abstract*—We present a new divide-and-conquer algorithm for mid-range multiprecision integer division which is typically 20-25% faster than the recent algorithms of Möller and Granlund implemented in the GNU Multi Precision (GMP) library.**

I. Introduction

The classical algorithm for integer division has $O(n^2)$ complexity for a multiprecision $2n \times n$ word division. This can be improved asymptotically using a divide-and-conquer approach.

In this paper we present such a divide-and-conquer algorithm for speeding up multiprecision integer division. Our algorithm builds on ideas from the work of David Harvey [8].

We obtain speedups typically in the range of 20–25%, in the range of applicability for this algorithm, over the recent divide-and-conquer algorithms of Möller and Granlund as implemented in GMP [4].

The idea to use divide-and-conquer for integer division has been around since at least 1997 when Tudor Jebelean described such an algorithm [9]. The top half Q_H of the quotient Q of A by D is computed approximately (recursively using the same algorithm if optimal). This quotient is then corrected with high probability by doing three word updates to the dividend using a method due to Krandick. Karatsuba multiplication is then used to update the dividend by subtracting $D_L Q_H$, where D_L is the low half of the divisor. Next the low half Q_L of the quotient is computed (recursively if optimal). The dividend is again adjusted using Karatsuba multiplication by subtracting $Q_L D_L$. The final remainder may be too large, but this is corrected with some multiprecision subtractions.

Jebelean's approach may fail with very low probability, but this can be detected and classical division can be used in that case. Thus the average complexity is that of Karatsuba multiplication, i.e. $O(n^{\log_2 3})$.

In particular, for a $2n \times n$ word division, two $n/2 \times n/2$ multiplications are performed at the top level of Jebelean's algorithm with Karatsuba complexity. Of course, this is repeated at smaller sizes for each recursive call to the function.

In 2009, David Harvey developed an integer middle product algorithm with Karatsuba complexity [8]. Given a $2n \times n$ word product, the middle product computes the contribution to the middle n words of the product.

Harvey applied this algorithm to Hensel division and also produced proof-of-concept code for approximate Euclidean division. The latter took the straightforward approach of computing an additional word of each recursive quotient which can be discarded, so that the next word of the quotient is either correct or off by at most one. It also manually computed additional words of the middle product which are required to compute the intermediate remainder. This effectively allows the first recursive quotient to be corrected, though in practice this step is merged with the second recursive quotient.

In section 2 of this paper we describe a new divide-and-conquer algorithm for approximate quotient based on Harvey's middle product. In this approach, an *exact* correction to the intermediate remainder is computed in linear time so that additional intermediate words of data do not have to be computed.

Our divide-and-conquer algorithm is based on a specialised approximate quotient basecase which computes a quotient along with a partial remainder, but where the latter has exact arithmetic significance. We make use of the exact arithmetic guarantee for the basecase algorithm to ensure correctness of the divide-and-conquer algorithm.

In section 3 we provide benchmarks of our implementation and compare with the fastest algorithm and implementation known to us (that of Möller and Granlund as implemented in GMP [4]).

II. Notation and preparations

Throughout this paper we use the notation $\langle a_{m-1}, \ldots, a_1, a_0\rangle$ to denote a non-negative bignum integer consisting of m digits/words in base B (we will always have $B = 2^{64}$ or 2^{32}), with the most significant word at the left.

We will usually denote the three leading words of the current dividend A by $u_2 = a_{m-1}$, $u_1 = a_{m-2}$, $u_0 = a_{m-3}$ and the leading two words of D by $v_1 = d_{n-1}$, $v_0 = d_{n-2}$.

Recall that the Euclidean *divrem* problem is as follows: given bignums $A = \langle a_{m-1}, \ldots, a_1, a_0\rangle$ and $D = \langle d_{n-1}, \ldots, d_1, d_0\rangle$ with $m \geq n$, find bignums Q and R such that $A = QD + R$ with $0 \leq R < D$.

To simplify matters, we will assume that D is normalised so that $B/2 \leq v_1 < B$, i.e. so that the most significant bit is set. If this is not the case, the dividend and divisor can be shifted so that this is so. We also assume throughout that the divisor D has at least two words.

1063-6889/15 $31.00 © 2015 IEEE
DOI 10.1109/ARITH.2015.19

Our algorithm also requires $n-2 < B/2$, which is not a problem in practice if FFT based algorithms are used for large operands.

III. Divide-and-conquer approximate division

One problem with division with remainder is that its complexity depends both on the size of the quotient and the size of the remainder. For, if only the quotient is required and that is much smaller than the remainder, then much time is wasted computing the latter.

A solution to this, implemented by the authors of the GMP library, is to compute an approximate quotient without remainder. An approximate quotient might compute the correct quotient or be perhaps one off.

It is possible to use an approximate quotient function to compute an exact quotient. One more quotient word than requested is computed. This extra word is then examined to see if a correction to it could possibly cause a carry into the low word of the quotient. If not, the quotient must be correct, even if the extra word is not.

If a carry from the extra word could change the quotient, then the full remainder can be computed to see if the quotient was correct or not. This is expensive, however this happens with very low probability.

Given an approximate quotient it is also possible to compute a quotient with remainder using a mullo and subtraction, developing an extra word of the remainder to see if the approximate quotient might need correction.

Therefore, it only remains to develop an efficient algorithm for the approximate quotient and the other division functions can also be made efficient.

In this section we describe a divide-and-conquer algorithm for approximate quotient. However it relies recursively on the fact that it also computes an algebraically significant partial remainder.

Firstly, we give the definition of our approximate quotient stating the required properties.

Definition 3.1: (divapprox) Given multiprecision inputs $A = \langle a_{m-1}, \ldots, a_1, a_0\rangle$ and $D = \langle d_{n-1}, \ldots, d_1, d_0\rangle$ with $m \geq n > 2$ and $d_{n-1} \geq B/2$, compute $Q^\star = \langle q_{m-n}, q_{m-n-1}, \ldots, q_1, q_0\rangle$ and $R^\star = A - \sum_{i+j\geq n-2} q_i d_j B^{i+j}$, with $q_{m-n} = 0, 1$ such that $R^\star$ is non-negative, but as small as possible.

We will see that $Q^\star$ will either be correct or one too large.

In practice, we update A in-place so that the partial remainder $R^\star$ is stored in A at the end of the algorithm. We note that the low $n-2$ words of A are not altered.

Throughout, we define $s = m - n + 1$. Initially, this is the number of words in the final quotient Q (the leading quotient word q_{m-n} may be zero), however we view s as a function of the current size m of the dividend as it changes throughout the algorithm.

Algorithm 1:

Basecase divapprox

Input: $A = \langle a_{m-1}, \ldots, a_1, a_0\rangle$, $D = \langle d_{n-1}, \ldots, d_1, d_0\rangle$ with $m > n > 2$ with $B^2 > d \geq B^2/2$, $s = m - n + 1$, and $\nu = \lfloor B^3/(d+1)\rfloor - B$ where $d = \langle d_{n-1}, d_{n-2}\rangle$.
Output: $Q^\star = \langle q_{m-n-1}, \ldots q_1, q_0\rangle$ and $R^\star = \langle r_n, \ldots, r_1, r_0\rangle$ with $R^\star = A - \sum_{i+j\geq n-2} q_i d_j B^{i+j}$, such that $R^\star \geq 0$ is as small as possible. We store $R^\star$ in-place in A.

while $s > n$ **do**
  $s_h \leftarrow \min(n, s-n)$
  $A_h \leftarrow \langle a_{m-1}, \ldots, a_{m-n-s_h}\rangle$ {In-place}
  $\langle q_{s-2}, \ldots, q_{s-s_h-1}\rangle \leftarrow$ divrem(A_h, D, ν)
  $s \leftarrow s - s_h$, $m \leftarrow m - s_h$
end while
while $s \geq 2$ **do**
  $D' \leftarrow \langle d_{n-1}, \ldots, d_{n-s}\rangle$ {Truncate divisor to length s}
  if $\langle a_{m-1}, \ldots, a_{m-s}\rangle \geq D'$ **then**
    while $s \geq 2$ **do**
      $q_{s-2} = B - 1$ {Special case: overflow}
      $A \leftarrow A - q_{s-2} \times \langle d_{n-1}, \ldots, d_{n-s}\rangle \times B^{n-2}$
      $s \leftarrow s - 1$
    end while
    return {Terminate algorithm}
  end if
  $q_{s-2} =$ divappr32$(a_{m-1}, a_{m-2}, d_{n-1}, d_{n-2}, \nu)$
  $A \leftarrow A - q_{s-2} \times D' \times B^{n-2}$
  if $a_{m-1} > 0$ or $\langle a_{m-2}, \ldots, a_{m-s-1}\rangle \geq D'$ **then**
    $q_{s-2} = q_{s-2} + 1$ {Overflow cannot occur}
    $A \leftarrow A - D' \times B^{n-2}$
  end if
  $m \leftarrow m - 1$
  $s \leftarrow s - 1$
end while

end

In order to present a divide-and-conquer divapprox we first require a basecase algorithm which performs this computation. This is important, as the divide-and-conquer algorithm will rely on the fact that the bottom $n-2$ words of the dividend have not been touched, and that the partial remainder has been computed correctly.

Our approach to basecase divapprox is similar to a standard basecase divrem algorithm, with two main differences. Firstly, we truncate the divisor as we proceed, so that the adjustment step does not affect the low $n-2$ words of the dividend. Secondly, we must deal with the case where truncating the divisor results in overflow of the quotient word.

In algorithm 1 we present our basecase divapprox algorithm.

Before our algorithm is called, we assume a preparation step has been called to compute q_{m-n}, so that only $m-n$ quotient words need to be computed by the algorithm. Because the divisor is normalised, $q_{m-n} = 0, 1$ and this step can be completed with a single subtraction in linear time.

Note that we require $m > n$, however, the cases $m \leq n$ can be dealt with trivially if required.

Firstly in the algorithm, steps of the usual basecase divrem algorithm are applied that do not affect the bottom $n-2$ words of the dividend, i.e. until $s \leq n$.

Once $s < n$ the algorithm uses only the leading words of the dividend and divisor, treating it as a $(2s-1) \times s$ division at each step. This also avoids modifying the low $n-2$ words of the dividend.

The algorithm makes use of a function divappr32$(u_2, u_1, d_{n-1}, d_{n-2}, \nu)$ (see algorithm 2) which performs an approximate 3×2 division using the precomputed inverse ν. By an argument that is essentially the same as the proof of theorem 2 in [10] this computes an approximate quotient word $q^\star$ which is either the correct multiprecision quotient word at each step, or one too small.

Algorithm 2:

divappr32

Input: $A = \langle u_2, u_1, 0\rangle$, $d = \langle d_1, d_0\rangle$ with $A < dB$, $B^2 > d \geq B^2/2$, and $\nu = \lfloor B^3/(d+1)\rfloor - B$

Output: $q^\star$ where $(A+1)B^n - 1 < (q^\star + 1)dB^n$ and $AB^n \geq q^\star((d+1)B^n - 1)$ for all $n \geq 0$

$\langle q^\star, q', q''\rangle \leftarrow \langle u_2, u_1\rangle \times \nu$
$\langle q^\star, q'\rangle \leftarrow \langle q^\star, q'\rangle + \langle u_2, u_1\rangle$

end

Theorem 3.2: The quotient words q_i do not overflow, and if $Q^\star = \langle q_{s-2}, \ldots q_0\rangle$ is the quotient computed by *basecase divapprox* then it is either the correct quotient Q of A by D or one too large. The partial remainder $R^\star$ is computed correctly and the low $n-2$ words of the dividend are not modified.

Proof: It is clear that at each iteration of the algorithm, truncation of the divisor to s words could lead to a quotient word q_{s-2} which is one too large. However, this is only possible in the first instance if the leading s words of the dividend are at least D'. But the algorithm deals with this case separately by setting all remaining words of the quotient to $B-1$, which clearly do not overflow.

In all other cases, computing a quotient word with D' as divisor does not overflow the quotient word.

It is clear that the partial remainder is computed at each iteration by subtracting the current quotient word times the appropriate words of the divisor from the leading words of A. It is easy to see that the final partial remainder is as given.

We also see that each adjustment of the dividend does not touch the low $n-2$ words of the dividend, by construction.

Note that after each step, except possibly the special overflow case, the leading word of the dividend is set to zero and the next s words end up with a value that is less than D'. This is because divapprox32 and correction compute precisely the quotient word of the leading $s+1$ words of the dividend by D'. Thus the remainder of this division must be less than D'.

As the largest possible quotient word is chosen at each step throughout the algorithm, it is clear that the final $R^\star$ could not be made any smaller by increasing one of the quotient words. By construction, $R^\star$ is never negative.

It remains to show that $Q^\star$ is either the correct quotient or one too large.

We will first deal with the case where the special overflow case never happens.

Because the partial remainder $R^\star$ is larger than $A - Q^\star D$, and increasing any of the quotient words would make $R^\star$ negative, we see that $Q^\star$ is either correct or too large.

To see that $Q^\star$ is at most one too large, we must show that $R' = A - Q^\star D \geq -D$. The difference between $R^\star$ and R' is the contribution $V = \sum_{i+j \leq n-3} q_i d_j B^{i+j}$. Thus it suffices to show that $V \leq D$.

But breaking the expression for V into a double sum, the first over i and the second over j, we see that each term of the outer sum is less than B^{n-1}. Thus $V < (n-2)B^{n-1} < D$ (assuming $n - 2 < B/2$), as required.

In the special overflow case, increasing $Q^\star$ by 1 would replace trailing words of $Q^\star$ that had been $B-1$ with 0 and then increase the next word, q_k say, by 1.

Clearly $R^\star$ becomes negative if we do this, otherwise q_k would have been chosen larger in the first place. The remainder $R' = A - Q^\star D$ is at most $R^\star$, and so $Q^\star$ is either correct or too large.

It remains to show that $R^\star$ is not negative in the special overflow case. Then a similar argument to the above will apply. But this is clear, as at each iteration of the special case we have that the leading $s+1$ words of the dividend are at least BD'. Thus, subtracting $(B-1)D'$ from the leading $s+1$ words of the dividend leaves a result which is at least D'. After truncating the divisor by one word, the same argument applies at the next iteration, and so on. ∎

Our basecase algorithm also has the following important property.

Lemma 3.3: Truncating the dividend and divisor of an $m \times n$ division to the leading $m-k$ words of the dividend and $n-k$ words of the divisor will not affect the result of divapprox basecase, so long as $n - k \geq s = m - n + 1$.

Proof: The algorithm treats both as a $(2s-1) \times s$ division and performs the same steps. ∎

Corollary 3.4: At the start of an iteration of the main *while* loop in the divapprox basecase algorithm, the steps that remain are precisely those that would be performed if the divapprox basecase algorithm were called with the current dividend and divisor as operands.

Algorithm 3:

divapprox_divconquer

Input: $A = \langle a_{m-1}, \ldots, a_1, a_0\rangle$, $D = \langle d_{n-1}, \ldots, d_1, d_0\rangle$ with $m > n > 2$ with $B^2 > d \geq B^2/2$, and $\nu = \lfloor B^3/(d+1)\rfloor - B$ where $d = \langle d_{n-1}, d_{n-2}\rangle$, and $\langle a_{m-1}, \ldots, a_{m-n}\rangle < D$, $4 < S_{\min} \in \mathbb{Z}$.

Output: $Q^\star = \langle q_{m-n-1}, \ldots q_1, q_0\rangle$ and $R^\star = \langle r_n, \ldots, r_1, r_0\rangle$ with $R^\star = A - \sum_{i+j \geq n-2} q_i d_j B^{i+j}$ (in-place in A), with $R^\star \geq 0$ and as small as possible.

$s \leftarrow m - n + 1$
if $s < S_{\min}$ or $n \leq 5$ **then**
  $Q^\star \leftarrow$ divapprox_basecase(A, D, ν)
  return

end if
while $s > n$ **do**
  $s_h \leftarrow \min(n, s-n)$
  $A_h \leftarrow \langle a_{m-1}, \ldots, a_{m-n-s_h}\rangle$ {In-place}
  $\langle q_{s-2}, \ldots, q_{s-s_h-1}\rangle \leftarrow$ divrem(A_h, D, ν)
  $s \leftarrow s - s_h$, $m \leftarrow m - s_h$
end while
if $\langle a_{m-1}, \ldots, a_{m-s}\rangle \geq \langle d_{n-1}, \ldots, d_{n-s}\rangle$ **then**
  $A_h \leftarrow \langle a_{m-1}, \ldots, a_{m-s-1}\rangle$ {In-place}
  $D_h \leftarrow \langle d_{n-1}, \ldots, d_{n-s}\rangle$
  $Q^\star \leftarrow$ divapprox_helper $(A_h, D_h, s-1)$
  return
end if
$s_h \leftarrow \lfloor (s-1)/2 \rfloor$, $s_l \leftarrow s - 1 - s_h$, $i \leftarrow 0$
$A_h \leftarrow \langle a_{m-1}, \ldots, a_{s_l}\rangle$ {In-place}
$Q_h \leftarrow$ divapprox_divconquer(A_h, D, ν)
$D_h \leftarrow \langle d_{n-3}, \ldots, d_{n-s}\rangle$
$A_m \leftarrow \langle a_{m-s+s_l+1}, \ldots, a_{m-s}\rangle$ {In-place}
$A_m \leftarrow A_m -$ mulmid(D_h, Q_h)
if $a_{m-s+s_l+1} < 0$ **then**
  $Q_h \leftarrow Q_h - 1$ {Ensure quotient is not too large}
  $A_m \leftarrow A_m + \langle d_{n-1}, \ldots, d_{n-s_l-2}\rangle$
  while $i < s_h - 1$ and $q_{s_l+i} = B - 1$ **do**
    $A_m \leftarrow A_m + d_{n-s_l-3-i}$ {Middle product correction}
    $i \leftarrow i + 1$
  end while
end if
if $A \geq DB^{s_l}$ **then**
  $A_l \leftarrow \langle a_{m-s+s_l}, \ldots, a_{m-s-1}\rangle$ {In-place}
  $D_l \leftarrow \langle d_{n-1}, \ldots, d_{n-s_l-1}\rangle$
  $Q_l \leftarrow$ divapprox_helper(A_l, D_l, s_l)
else
  $Q_l \leftarrow$ divapprox_divconquer(A, D, ν)
end if
$Q^\star \leftarrow Q_h B^{s_l} + Q_l$

end

We now discuss a divide-and-conquer version of the divapprox algorithm, which switches to the divapprox basecase when the quantity s falls below a heuristically chosen cutoff $S_{\min}$.

We make use of an algorithm which we denote $Q =$ divapprox_helper(A, D, r). It performs the steps described in the "special overflow" *while* loop of Algorithm 1, namely to set each of the remaining r words of the quotient Q to $B-1$ and to adjust the dividend A by subtracting the appropriate multiple of D.

Also we use the middle product algorithm. Given inputs $D = \langle d_{m-1}, \ldots, d_0\rangle$ and $Q = \langle q_{n-1}, \ldots q_0\rangle$, with $m + 1 \geq n \geq 2$, the middle product computes the value mulmid$(Q, D) = \sum_{i=0}^{n} q_i \times \langle d_{m-1-i}, \ldots, d_{n-1-i}\rangle$. The result has $m - n + 3$ words.

In order for our divide-and-conquer algorithm to be asymptotically faster than the basecase, we require the middle product to be asymptotically faster than classical multiplication. For this purpose we use the integer middle product of David Harvey [8], which has Karatsuba complexity.

As for the basecase, we assume that the input has been prepared so that the leading part of the dividend is less than the divisor. Thus if the dividend has m words and the divisor n, the quotient will only have $m - n$ words.

Note that our algorithm modifies parts of the dividend A in-place rather than working on a copy (noted in comments in the algorithm).

Theorem 3.5: Algorithm 3 computes divapprox correctly. The computed quotient is either the correct quotient of A by D or it is one too large.

Proof: We will show that the divide-and-conquer algorithm computes the same thing as the basecase divapprox. This implies that the algorithm yields either the correct quotient or one too large. We will also check that all of the conditions of both the basecase and divide-and-conquer algorithm are met recursively.

In the case that $s > n$ an $(n + s_h) \times n$ division is carried out by performing steps of the divrem algorithm. By the input conditions, this will compute s_h quotient words. As no truncation of the divisor occurs, the quotient words are computed exactly. If $s \geq 2n$ we choose $s_h = n$, performing a $2n \times n$ division, otherwise we set $s_h = s - n$, performing an $s \times n$ division. These steps are obviously the same as the corresponding steps in the basecase divapprox algorithm above.

Now we have $s \leq n$. From now on, the algorithm implicitly truncates dividend and divisor so that $s = n$.

Thus we need to perform a $(2s-1) \times s$ division and compute $s - 1$ quotient words. As the new s is n, the new m is $2n - 1 > n$ (as $n > 2$).

The next step of the algorithm is to perform the preparation step that is required before calling the basecase algorithm, namely to ensure that the leading s words of the dividend are less than the s words of the divisor. If this is not the case, we terminate the algorithm after setting all remaining words of the quotient to $B - 1$ using divapprox_helper, which also adjusts the $s+1$ words of A_h. We see that this adjusts all but the low $n - 2$ words of A.

In broad outline, the strategy of the rest of the algorithm is to compute a total of $s-1$ quotient words in two parts. We first compute s_h of the quotient words, then adjust the dividend and finally compute the remaining s_l of the quotient words, where $s_h + s_l = s - 1$. The two parts of the quotient are computed by recursive calls to divapprox_divconquer. Note that as $S_{\min}$ is at least 5, we have that s is at least 5, s_h is at least 2 and s_l is at least 2.

The adjustment of the dividend between the recursive calls is partially handled by the first recursive call, which computes a partial remainder. However, this does not update sufficiently many words of the dividend, so we extend the adjustment by subtracting the middle product. However, this may cause the quotient to need adjustment. This in turn changes the value that would have been obtained when computing the middle product and first recursive divapprox_divconquer, so we need to adjust for this.

We now discuss the details. The first recursive call to divapprox_divconquer is an $(m - s_l) \times n$ division. It will

compute s_h words of the quotient, either exactly, or one too large. It will not alter the low $n-2$ words of A_h, i.e. it will not alter the low $n+s_l-2$ words of A.

We see that A_h has $m-s_l = 2s-s_l-1 > s$ words and D has $s > 2$ words. We have already ensured that the preparation step for an s word divisor has been completed, so the input requirements of divapprox_divconquer are recursively met.

We now need to ensure that all contributions to the partial remainder $R^\star$ involving Q_h are computed. So far we have not touched the low $n+s_l-2$ words of A. However the contribution to $R^\star$ coming from Q_h requires the product by an additional s_l words of D for each word of Q_h over what was computed in the recursive call to divapprox_divconquer. This missing contribution is the middle product.

The middle product of D_h and Q_h will compute $(s-2)-s_h+3 = s+1-s_h = s_l+2$ words. The leading two words of the middle product are subtracted from words already adjusted by the recursive call. Thus an additional s_l words of A are adjusted. Doing this, we have adjusted all but the low $n-2$ words of A, as required for the partial quotient.

It is easy to check that the input requirements for the middle product are met.

Now it may be that the quotient computed so far is one too large, in which case it may be that the remainder becomes negative. This would result in the now leading word of A to be negative.

In this case, we decrease Q_h by 1. However, if subtracting 1 from Q_h causes a borrow from the next word of Q_h, and so on, more than one word of Q_h may be modified. In this case the lower words of Q_h will change from 0 to $B-1$ and the next non-zero word of Q_h will be decreased by 1. We must therefore adjust the middle product for each word of Q_h that has been altered. This is simple to test for, as the words of Q_h will now be $B-1$.

In addition to this, if Q_h were one smaller, the partial remainder computed by the first recursive call to divapprox_divconquer would also be different.

To make matters simpler, let's temporarily relabel A_m as $\langle a_{s_l+1}, \ldots, a_0\rangle$ and Q_h as $\langle q_{s_h-1}, \ldots, q_0\rangle$. The contribution of $Q_h D$ to the partial remainder A_m, taking into account both the middle product and partial remainder of the first recursive call is

$$\sum_{i=0}^{s_h-1} q_i \langle d_{n-1}, \ldots, d_{n-s_l-2-i}\rangle.$$

We need to evaluate the change in such an expression if we change the low words of Q_h from 0 to $B-1$ and then decrement the next word of Q_h. If there are k terms of Q_h which change from 0 to $B-1$, then the total adjustment to the partial remainder should be

$$\sum_{i=0}^{k-1}(1-B)\langle d_{n-1}, \ldots, d_{n-s_l-2-i}\rangle + \langle d_{n-1}, \ldots, d_{n-s_l-2-k}\rangle.$$

But this can be rewritten

$$\langle d_{n-1}, \ldots, d_{n-s_l-2}\rangle - \sum_{i=0}^{k-1} B\langle d_{n-1}, \ldots, d_{n-s_l-2-i}\rangle + \sum_{i=0}^{k-1}\langle d_{n-1}, \ldots, d_{n-s_l-3-i}\rangle.$$

It's easy to see that most of the terms in the sums cancel for each i, leaving precisely the adjustment terms given in the algorithm.

At this point it is clear that we have computed the same thing as is computed in the basecase algorithm. A final call to divapprox_divconquer completes the remaining steps. Corollary 3.4 guarantees that a recursive call at this point completes precisely the steps that would be computed by the divapprox basecase algorithm from this point.

In fact, what remains is an $(m-s+s_l+1)\times n = (n+s_l)\times n$ division. The input must be prepared so that $A < DB^{s_l}$. In case it is not, we can set the remaining quotient words to $B-1$ and adjust the partial remainder by updating A_l with divapprox_helper.

For the final call to divapprox_divconquer we see that A has $m-s+s_l+1 = n+s_l$ words, and D has $n > 2$ words. Thus the input conditions are met for this recursive call. The partial remainder of the final recursive call updates all but the low $n-2$ words of A, which is just what is required.

Finally, we write out the approximate quotient $Q^\star$. As the same thing has been computed as in the basecase, we have that the computed quotient is either correct or one too large. ■

IV. Benchmarks

We implemented the algorithms above in both BSDNT [7] and MPIR [3]. For the MPIR divide-and-conquer algorithm, we made use of David Harvey's implementation of his Karatsuba complexity middle product which he generously made public.

Our timings (see Table 1 below) are in nanoseconds per iteration. We used a single core of a 64 bit 2.2GHz AMD Opteron 6174.

We performed sufficiently many iterations at each size for a total time of a few seconds. Times reported are per iteration. Uniformly random data was regularly regenerated so that the final timings didn't depend on the properties of the numbers, but not so often that random generation was a significant part of the time.

Our timing comparison was made against GMP version 6.0.0a, the current version at the time of benchmarking. GMP's crossover to divide-and-conquer is much higher than ours (around 230 words on the test hardware vs around 50 words for our algorithm). Above 1000 words the FFT based algorithms for division start to become effective, so we stop our table at this point.

These improvements carry over to exact division and division with remainder. For these we made use of the code already in GMP. Their code makes use of a divapprox not

TABLE I. MULTIPRECISION DIVIDE-AND-CONQUER DIVAPPROX $2n \times n$ WORDS

n	GMP	NEW	RATIO
46	287	219	1.31
51	332	261	1.27
57	395	308	1.28
63	456	365	1.25
70	544	437	1.25
77	637	525	1.21
85	755	619	1.22
94	869	725	1.20
104	1061	838	1.27
115	1260	1010	1.25
127	1490	1230	1.21
140	1710	1440	1.19
154	2020	1640	1.23
170	2350	1920	1.22
188	2810	2350	1.20
207	3270	2660	1.23
228	3800	3110	1.22
251	4620	3810	1.21
277	5320	4310	1.23
305	6160	4930	1.25
336	7040	5950	1.18
370	8580	7090	1.21
408	9780	7980	1.23
449	11590	9100	1.27
494	13930	11080	1.26
544	15540	13150	1.18
599	18160	15970	1.14
659	21280	17410	1.22
725	25420	21030	1.21
798	29040	24540	1.18
878	33130	28960	1.14
966	40630	33280	1.22

based on the middle product. We substituted their divapprox with our own.

We have glossed over one implementation detail which caused us some trouble. The precomputed inverse used by our algorithm is not that used by GMP. As the divide-and-conquer and basecase divapprox algorithms need to pass this inverse to the divrem basecase algorithm, we are not able to use the GMP basecase division directly.

One obvious solution is to simply compute both precomputed inverses. However this is a hidden expense that we wished to avoid. We found two solutions to this problem, both of which we implemented.

The first solution is to compute one precomputed inverse from the other. In fact this can be done at almost no cost, as they are the same except in very rare cases which can be tested for very cheaply.

The second solution is to implement a divrem basecase based on our precomputed inverse. In fact this turns out to even be slightly faster than the GMP basecase algorithm for very small input sizes.

In both cases, the basecase times can be made to match those of GMP almost precisely, and even beat GMP for very small sizes on some platforms.

V. CONCLUSION

We have described a divide-and-conquer division algorithm for midsized multiprecision integer division and demonstrated that it beats the recent algorithms implemented in the GMP library, typically by 20-25%.

ACKNOWLEDGMENT

The author would like to thank Fredrik Johansson for reading and commenting on an earlier draft of this paper, and for the numerous comments and suggestions of the anonymous reviewers.

REFERENCES

[1] P. Barret, *Implementing the Rivest Shamir and Adelman public key encryption algorithm on a standard digital signal processor*, Advances in Cryptology – Crypto '86, LNCS 263, Springer-Verlag (1987), pp.311–323.

[2] P. W. Beame, S. A. Cook, H. J. Hoover, *Log depth circuits for division and related problems*, SIAM J. Comput. 15 (1986), no. 4, pp. 994-1003.

[3] B. Gladman, T. Granlund, W. Hart, J. Moxham and the MPIR and GMP development teams *The Multiple Precision Integers and Rationals Library version 2.7.0* http://mpir.org/

[4] T. Granlund and the GMP development team *The GNU Multi Precision Arithmetic Library version 6.0.0a* http://gmplib.org/

[5] T. Granlund, N. Möller, *Division of Integers Large and Small*, (August 2009, to appear)

[6] T. Granlund, P. Montgomery *Division by invariant integers using multiplication* Proc. SIGPLAN Conf. Programming Language Design and Implementation (PLDI '94), June 1994, SIGPLAN Notices (ACM) 29, (6), pp. 61-72.

[7] W. Hart and the BSDNT development team *BSDNT version 1.0* https://github.com/wbhart/bsdnt/

[8] D. Harvey, *The Karatsuba middle product for integers*, J. Symb. Comp. 47 (2012), pp. 954-967.

[9] T. Jebelean, *Practical integer division with Karatsuba complexity*, Küchlin, W. W. (ed.), Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC '97), (45), pp. 339-341.

[10] N. Möller, T. Granlund, *Improved division by invariant integers*, IEEE Transactions on Computers, vol. 60, no. 2, February, (2011), pp. 165–175.

[11] C. S. Wallace, *A suggestion for a fast multiplier*, IEEE Trans. Electronic. Computers, vol. 13, (1964), pp. 14-17.