A New Variant of the Barrett Algorithm Applied to Quotient Selection

Niall Emmart*, Fangyu Zheng†‡, Charles Weems*
*College of Information and Computer Sciences, University of Massachusetts, Amherst, MA 01003-4610, USA
†State Key Laboratory of Information Security, Institute of Information Engineering,
Chinese Academy of Sciences, Beijing, China
‡Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
Email: nemmart@yrrid.com, fyzheng@is.ac.cn, weems@cs.umass.edu

***Abstract*—Quotient Selection (QS) is a key step in the classic $O(n^2)$ multiple precision division algorithm. On processors with fast hardware division, it is a trivial problem, but on GPUs, division is quite slow. In this paper we investigate the effectiveness of Brent and Zimmermann's variant as well as our own novel variant of Barrett's algorithm. Our new approach is shown to be suitable for low radix (single precision) QS. Three highly optimized implementations, two of the Brent and Zimmerman variant and one based on our new approach, have been developed and we show that each is many times faster than using the division operation built in to the compiler. In addition, our variant is on average 22% faster than the other two implementations. We also sketch proofs of correctness for all of the implementations and our new algorithm.**

***Index Terms*—multiple precision division, quotient selection, Barrett reduction**

I. Introduction

Quotient Selection (QS) is a simple computation that arises in the implementation of multiple precision division. QS computes the following:

$$q = \left\lfloor \frac{a_1\beta + a_0}{d} \right\rfloor$$
$$\textbf{return} \min(q, \beta - 1)$$

where β is a power of two (typically 2^w, where w is the machine word size), and $0 \le a_0, a_1 < \beta$, and d is a normalized divisor $\beta/2 \le d < \beta$. On processors with a hardware divide instruction, this is a trivial operation. However, it can be quite slow and often a performance limiter on simpler processors, such as GPU cores, vector processors, DSPs, or embedded processors. Since the same divisor is used repeatedly, a common solution to the problem is to use Barrett's algorithm to replace the division with multiplications. The contributions of this paper are two-fold: (1) we present a novel variant of Barrett's algorithm that allows for faster a implementation of QS; (2) we present three highly optimized implementations of Barrett-based QS (BQS), and show that the implementations are correct. These efficient implementations of BQS, using C and assembly language, are non-trivial and we believe they will be of interest to developers working on similar processors.

The rest of this paper is organized as follows. Section I-A presents background information. Section I-B covers prior work on Barrett's algorithm. Section I-C discusses the hardware requirements to support our BQS implementations. Section II covers Brent and Zimmermann's variant ([4] §2.4.1) of the Barrett algorithm and presents the first two implementations of BQS, which are based on it. Section III covers our proposed variant of Barrett's algorithm and the third BQS implementation. Our experiments and results are presented in Section IV, which show that all three BQS implementations are much faster than the built-in compiler-generated long division on NVIDIA GPUs, and that the third implementation of BQS, based on our variant of Barrett's algorithm, is on average 22% faster than the other two variants. Section V gives our conclusions.

A. Background

In multiple precision (MP) division, we wish to compute $\lfloor X/Q \rfloor$, where X and Q are represented using a fixed radix number system, i.e.,

$$X = \sum_{i=0}^{n-1} x_i\beta^i \qquad \text{and} \qquad Q = \sum_{i=0}^{m-1} q_i\beta^i$$

where β is the radix, $0 \le x_i, q_i < \beta$, and $n \ge m$. MP division algorithms can be classified as either slow quadratic algorithms (such as the grade school long division algorithm) or fast sub-quadratic algorithms. The slow algorithms typically compute a fixed number of quotient bits on each iteration. The fast algorithms include Newton-Raphson, Goldschmidt division [9] and Burnikel-Ziegler's divide and conquer algorithm [5]. The fast algorithms typically rely on grade school division, for example, Newton-Raphson uses grade school division to generate an initial seed and Burnikel-Ziegler uses grade school division for the base cases. Thus a good MP library requires an efficient grade school implementation and one of the key steps of the grade school algorithm is Quotient Selection. For more information about division algorithms and QS see [4], [12], [15].

B. Prior Work on Barrett's Algorithm

In 1984, Barrett [1], [2] introduced an algorithm for the modular reduction operation. His basic idea is replacing the expensive division with a multiplication by a pre-computed constant which approximates the inverse of the modulus. Thus

the calculation of the exact quotient $q = \lfloor \frac{a}{d} \rfloor$ is avoided by computing the quotient q^*:

$$q^* = \Big\lfloor \frac{\lfloor \frac{a}{2^{n-1}} \rfloor \lfloor \frac{2^{2n}}{d} \rfloor}{2^{n+1}} \Big\rfloor,$$

where $\lfloor \frac{2^{2n}}{d} \rfloor$ is a pre-computed constant. In the original Barrett reduction, the estimate quotient $q^* \in [q-2, q]$, which implies at most 2 correction steps are required.

In 1994, Dhem [7] provided a generalized version of the Barrett reduction:

$$q^* = \Big\lfloor \frac{\lfloor \frac{a}{2^{n+\delta}} \rfloor \lfloor \frac{2^{n+\gamma}}{d} \rfloor}{2^{\gamma-\delta}} \Big\rfloor$$

.

In particular, with appropriate choices, the error on the quotient can be reduced. Specificly when $\gamma = n+3$ and $\delta = -2$, only one correction step is required.

Many following papers adopted different configurations of γ and δ to achieve different goals. In 2000, Großschädl [10] used the FastMM algorithm proposed in [16] with $\gamma = n$ and $\delta = -n$. When applying it to modular exponentiation, no correction step of the intermediate results is necessary. In 2010, Brent and Zimmermann [4] described the Barrett reduction with $\gamma = n$ and $\delta = 0$. It requires three correction steps but is very efficient for single-precision or low-radix division.

These modifications are mainly related to the configuration of γ and δ. There are also many other variants of the Barrett reduction:

Special-moduli variant. In 2009 and 2010, Knzevic et al. provided two special sets of moduli, one can efficiently avoid the pre-computation step in Barrett reduction [13], the other set can be used in an interleaved modular multiplication based on Barrett reduction to reduce the number of multiplications [14]. In 2016, Géraud et al. [8] announced a method that can double the speed of Barrett reduction by using specific composite moduli. And in 2017 Bos and Friedberger [3] proposed an improved Barrett reduction over $2^x p^y \pm 1$.

More-pre-computation variant. In 2007, Hasenplaugh et al. [11] modified the Barrett reduction with more pre-computation, but reducing the overall amount of multiplication and addition.

Non-2-base variant. In 2013, Cao and Wu [6] proved that the base $b \geq 3$ in Barrett reduction can be replaced by the usual base 2, which solves the data expansion problem in Barrett reduction so as to modestly reduce the cost.

C. Assumptions about the hardware

In the implementations that follow, we rely on a few features that are beyond what is typically provided by C. In particular we assume that the processor has a carry flag for extended precision addition and subtract, and a full multiplier that can compute the low and high words of a product. In Table I we list the functions we use and their meaning. In addition we assume the existence of two types: *uint* for a unsigned integer range from 0 and $\beta - 1$, and *ulong* for unsigned integers that range from 0 to $\beta^2 - 1$.

II. Brent & Zimmermann's Variant of the Barrett Algorithm

The original Barrett algorithm was conceived for high radix multiple precision quotient estimation (and modular reduction), where the size, n, was in machine words. In this setting, we wish to apply it to estimate a single machine word quotient, i.e.,

$$q^* = \Big\lfloor \frac{a_1\beta + a_0}{d} \Big\rfloor$$

where $\beta = 2^w$ and w is the length in bits of a machine word and d is normalized, i.e., $\beta/2 \leq d < \beta$. Barrett's algorithm requires computing the product of a $w+2$ bit value by a $w+1$ bit value, the result of which needs to be divided by 2^{w+1}. Performing these particular product and shift operations does not align well with hardware words and both steps require multiple instructions. There are several variants of Barrett's algorithm that use a less precise inverse approximation, at the expense of more correction steps than the two required in the original Barrett algorithm. Here we look at Brent and Zimmermann's variant, [4] §2.4.1, shown in Algorithm 1, and apply it to quotient selection by adding $\min(q^*, \beta - 1)$ to the return on line 7. The algorithm works as follows. We require a pre-computed approximation to the inverse of the divisor, $\mu = \lfloor \beta^2/d \rfloor$. Then line 1 computes the initial quotient estimate, $q^* = \lfloor (a_1 \cdot \mu)/\beta \rfloor$. q^* will be close to q, in particular, $q^* \leq q \leq q^* + 3$, thus the correction step (lines 4 and 5) will be run at most three times. Then we return the minimum of q^* and $\beta - 1$ as the result.

For the correctness, we use the fact that $\mu = \lfloor \beta^2/d \rfloor$, and thus $\beta^2 - d < \mu d \leq \beta^2$. There are two key parts to the proof. First, we must show that q^* (computed in line 1) is never an

TABLE I
Non-standard C functions used in the implementations

Operation	Meaning
r=add_cc(a, b)	Computes the sum of $a + b$, sets the carry out processor flag
r=addc_cc(a, b)	Computes the sum of $a + b$ with carry in and carry out
r=addc(a, b)	Computes the sum of $a + b$ with carry in, does not set the carry out
r=sub_cc(a, b)	Two's complement subtraction, $a + \sim b + 1$, sets the carry out flag
r=subc_cc(a, b)	Computes $a + \sim b$ with carry in and carry out
r=subc(a, b)	Computes $a + \sim b$ with carry in, does not set carry out
(r0,r1)=mul_wide(a, b)	Computes the full product of a times b, writes the low half to r0, high half to r1
r1=mul_high(a, b)	Computes the high product of a times b

Algorithm 1 Quotient Selection using Brent and Zimmermann's variant of the Barrett Algorithm

Require: $0 \le a_0, a_1 < \beta$, d is normalized, and $\mu = \lfloor \beta^2/d \rfloor$
Ensure: $q^* = \min(\lfloor (a_1\beta + a_0)/d \rfloor, \beta - 1)$
1: $q^* \leftarrow \lfloor (a_1\mu)/\beta \rfloor$
2: $r \leftarrow (a_1\beta + a_0) - dq^*$
3: **while** $r \ge d$ **do**
4: $\quad r \leftarrow r - d$
5: $\quad q^* \leftarrow q^* + 1$
6: **end while**
7: **return** $\min(q^*, \beta - 1)$

over-estimate, i.e., $q^* \le q$:

$$\begin{aligned} q^* = \left\lfloor \frac{a_1\mu}{\beta} \right\rfloor &\le \left\lfloor \frac{a_1\mu}{\beta} + \frac{a_0}{d} \right\rfloor \\ &\le \left\lfloor \frac{a_1\mu d + a_0\beta}{d\beta} \right\rfloor \\ &\le \left\lfloor \frac{a_1\beta^2 + a_0\beta}{d\beta} \right\rfloor = q \end{aligned}$$

Second, the correction step is run at most three times:

$$\begin{aligned} q^* = \left\lfloor \frac{a_1\mu}{\beta} \right\rfloor &\ge \left\lfloor \frac{a_1\mu}{\beta} \right\rfloor + \left\lfloor \frac{a_0}{d} \right\rfloor - 1 \\ &\ge \left\lfloor \frac{a_1\mu d + a_0\beta}{d\beta} \right\rfloor - 1 \\ &\ge \left\lfloor \frac{a_1\beta^2 - a_1 d + a_0\beta}{d\beta} \right\rfloor - 1 \\ &\ge \left\lfloor \frac{a_1\beta + a_0}{d} \right\rfloor + \left\lfloor \frac{-a_1}{\beta} \right\rfloor - 2 \\ &\ge q - 1 - 2 = q - 3 \end{aligned}$$

Algorithm 1 is simple to state, however, there are some challenges in implementing it efficiently, namely, μ and q^* can both exceed $\beta - 1$ and thus need to be represented using multiple machine words. We develop two highly optimized variants, shown in Figure 2 and 3, that allow μ and q^* to be represented by single words, and we improve performance by unrolling the loop and merging the loop comparison with the subtraction (lines 3 and 4 of Algorithm 1). Both implementations use a common routine to compute the approximate inverse, shown in Figure 1, which computes: $\mu' = \mu - (\beta + 1)$. Subtracting $\beta + 1$ from μ means $0 \le \mu' < \beta$. Figure 2 is

```
uint BQS_approx(uint divisor) {
  ulong longdiv=divisor;

  // computes floor(beta^2 / divisor) - (β+1)
  longdiv=(0-longdiv)/divisor;
  return (uint)longdiv;
}
```

Figure 1. Barrett Approximation

```
1  uint BQSv1(uint a0, uint a1,
2             uint divisor, uint μ') {
3    uint qstar, y0, y1, correct;
4
5    (y0,y1) = mul_wide(μ', a1);
6    y0 = add_cc(y0, a1);
7    qstar = addc(y1, a1);
8    (y0,y1) = mul_wide(qstar, divisor);
9
10   //  adjust y0:y1 to simplify correction steps
11   y0 = add_cc(y0, divisor-1);
12   y1 = addc(y1, 0);
13
14   // first correction step
15   y0 = sub_cc(y0, a0);
16   y1 = subc_cc(y1, a1);
17   correct = subc(0, 0);
18   qstar = qstar - correct;
19
20   // second correction step
21   y0 = add_cc(y0, divisor);
22   y1 = addc_cc(y1, 0);
23   correct = addc(correct, 0);
24   qstar = qstar - correct;
25
26   // third correction step
27   y0 = add_cc(y0, divisor);
28   y1 = addc_cc(y1, 0);
29   correct = addc(correct, 0);
30   qstar = qstar - correct;
31
32   if(a1>=divisor)
33     qstar = β-1;
34   return qstar;
35 }
```

Figure 2. Barrett Quotient Selection implementation #1

a relatively straight-forward implementation of Algorithm 1. Lines 5-7 compute:

$$q^* = \left\lfloor \frac{a_1(\mu' + \beta + 1)}{\beta} \right\rfloor = \left\lfloor \frac{a_1\mu}{\beta} \right\rfloor$$

Lines 8-12 compute $(y0, y1) = \text{mul\_wide}(q^*, d) + (d - 1)$. Lines 14-30 are the unrolled correction steps. Lines 32-34 implement the $\min(q^*, \beta - 1)$.

Proof of correctness sketch: Lines 5-7 generate an initial q^* which is less than q, as was proven above. Lines 8-16 compute $(y0, y1) = dq^* + (d - 1) - (a_1\beta + a_0)$. Line 17 sets *correct* to -1 if $(y0, y1)$ is negative and 0 otherwise, and we have:

$$\begin{aligned} \text{correct} = -1 \quad &\Longrightarrow \quad dq^* + (d-1) - (a_1\beta + a_0) < 0 \\ &\Longrightarrow \quad (a_1\beta + a_0) - dq^* > d - 1 \end{aligned}$$

i.e., the remainder is greater than the divisor and we must run the correction step, which is done in line 18. Lines 20-30 similarly implement the other two correction steps. We note that the q^* computations in lines 7, 18, 24, and 30 are executed modulo β, that is, they they throw away any carry out. However, at each step in the computation q^* is guaranteed to be less than or equal to q. Therefore, if there were a carry out, it would require that $q \ge \beta$, in which case, we would have $a_1 \ge d$, since $a_1\beta + a_0 \ge q \cdot d$. Thus, line 33 is triggered

if and only if there is a carry out in the computation of q^* and we can conclude the algorithm is correct.

In Figure 3, we compute q^* as before, and then add three. Thus, $q^* \geq q$ and we approach q from above. If the computation of q^* carries out, then we set q^* to $\beta - 1$ before the correction steps, which fits in a single word. The correction steps are slightly more efficient because they do not have to add $d-1$ to $(y0 : y1)$, and since we are decrementing q^*, we can save an instruction on the last correction.

Correctness proof sketch: The difficult part of this proof is showing that the if statement in line 9 will be triggered if and only if there was a carry out in line 7 and/or line 8. If there were no carry outs in line 7 or line 8, then $q^* < \beta$ and the proof of correctness follows that of Algorithm 1. Next, suppose $a_1 = \beta - 1$, then we will definitely get at least one carry out on lines 7 and 8. However, since $a_1 = \beta - 1$, the if statement is triggered and q^* is set to $\beta - 1$ (as required by the min). Likewise, if $a_1 = \beta - 2$, we will have the carry out, and the if statement is triggered (as required). Thus, for the routine to produce an incorrect result, we would need to have three conditions: $a_1 \leq \beta - 3$, $a_1 < q^*$, and we have one or more carry outs. First, let's assume there is exactly one carry out (in line 7 or line 8), and we have $q^* + \beta = \lfloor (\mu a_1 + 3\beta)/\beta \rfloor$. Thus,

$$\begin{aligned} a_1 < q^* \quad &\Longrightarrow \quad a_1 + \beta < \left\lfloor \frac{\mu a_1 + 3\beta}{\beta} \right\rfloor \\ &\Longrightarrow \quad a_1\beta + \beta^2 < \mu a_1 + 3\beta \end{aligned}$$

which implies $\beta^2 < a_1(\mu - \beta) + 3\beta$. However, since $\mu \leq 2\beta$,

```
1  uint BQSv2(uint a0, uint a1,
2             uint divisor, uint μ') {
3    uint qstar, y0, y1, correct;
4
5    (y0,y1) = mul_wide(μ', a1);
6    y0 = add_cc(y0, a1);
7    qstar = addc(y1, a1);
8    qstar = qstar+3;
9    if(qstar <= a1)
10     qstar = β-1;
11   (y0,y1) = mul_wide(qstar, divisor);
12
13   // first correction step
14   y0 = sub_cc(a0, y0);
15   y1 = subc_cc(a1, y1);
16   correct = subc(0, 0);
17   qstar = qstar + correct;
18
19   // second correction step
20   y0 = add_cc(y0, divisor);
21   y1 = addc_cc(y1, 0);
22   correct = addc(correct, 0);
23   qstar = qstar + correct;
24
25   // third correction step
26   y0 = add_cc(y0, divisor);
27   y1 = addc_cc(y1, 0);
28   qstar = addc(qstar, correct);
29   return qstar;
30 }
```

Figure 3. Barrett Quotient Selection implementation #2

Algorithm 2 Quotient Selection using a proposed new variant of the Barrett Algorithm

Require: $0 \leq a_0, a_1 < \beta$, d is normalized, and $\nu = \lceil \beta^2/d \rceil$
Ensure: $q^* = \min(\lfloor (a_1\beta + a_0)/d \rfloor, \beta - 1)$
1: $q^* \leftarrow \lfloor (a_1\nu)/\beta \rfloor + \lceil a_0/d \rceil$
2: $q^* \leftarrow \min(q^*, \beta - 1)$
3: $r \leftarrow (a_1\beta + a_0) - dq^*$
4: **while** $r < 0$ **do**
5: $\quad q^* \leftarrow q^* - 1$
6: $\quad r \leftarrow r + d$
7: **end while**
8: **return** q^*

we have:

$$\beta^2 < a_1(\mu - \beta) + 3\beta \leq a_1\beta + 3\beta \leq \beta^2$$

Contradiction. Finally, we note that since $a_1 \leq \beta - 3$, we have

$$\mu a_1 + 3\beta \leq 2\beta(\beta - 3) + 3\beta = 2\beta^2 - 3\beta,$$

thus two or more carry outs is also not possible. We can conclude, it is okay to ignore the carry outs on lines 7 and 8. The proof that the correction steps work is similar to above and we omit it to save space.

III. Proposed New Variant of Barrett's Algorithm

In Section II, we noted that it was faster to choose q^* greater than or equal to q and correct downwards, rather than correct upwards. Here we propose a new variant of the Barrett algorithm that selects the initial q^* as

$$q^* = \left\lfloor \frac{a_1\nu}{\beta} \right\rfloor + \left\lceil \frac{a_0}{d} \right\rceil$$

where $0 \leq a_0, a_1 < \beta$, $\nu = \lceil \beta^2/d \rceil$, and we require d to be normalized. Our proposed variant is presented in Algorithm 2.

Correctness proof sketch: We start by noting that since $\nu = \left\lceil \frac{\beta^2}{d} \right\rceil$, we have $\beta^2 \leq \nu d < \beta^2 + d$ and have two parts to show, first, q^* is never an under-estimate of q, thus $q^* \geq q$. Using the fact that $\lfloor x \rfloor + \lceil y \rceil \geq \lfloor x + y \rfloor$, we have:

$$\begin{aligned} \left\lfloor \frac{a_1\nu}{\beta} \right\rfloor + \left\lceil \frac{a_0}{d} \right\rceil \quad &\geq \quad \left\lfloor \frac{a_1\nu d + a_0\beta}{d\beta} \right\rfloor \\ &\geq \quad \left\lfloor \frac{a_1\beta^2 + a_0\beta}{d\beta} \right\rfloor = \left\lfloor \frac{a_1\beta + a_0}{d} \right\rfloor = q \end{aligned}$$

Second, we show that q^* never requires more than two correction steps, thus $q^* \leq q + 2$. Since $\lfloor x \rfloor + \lceil y \rceil \leq \lceil x + y \rceil$, we have:

$$\begin{aligned} \left\lfloor \frac{a_1\nu}{\beta} \right\rfloor + \left\lceil \frac{a_0}{d} \right\rceil \quad &\leq \quad \left\lceil \frac{a_1\nu d + a_0\beta}{d\beta} \right\rceil \leq \left\lceil \frac{a_1\beta^2 + a_1 d + a_0\beta}{d\beta} \right\rceil \\ &\leq \quad \left\lceil \frac{a_1\beta + a_0}{d} + \frac{a_1}{\beta} \right\rceil \leq \left\lceil \frac{a_1\beta + a_0}{d} \right\rceil + 1 \\ &\leq \quad \left\lfloor \frac{a_1\beta + a_0}{d} \right\rfloor + 2 = q + 2 \end{aligned}$$

```
1  uint BQS3_approx(uint divisor) {
2    ulong longdiv=divisor;
3
4    if(divisor == β/2)
5      return β−1;
6
7    // compute ceil(β^2 / divisor) − β
8    longdiv=(−longdiv/divisor);
9    return ((uint)longdiv)+2;
10 }
```

Figure 4. Barrett Approximation

Further, if $a_0 = 0$, then q^* is within one correction step of q:

$$\left\lfloor \frac{a_1\nu}{\beta} \right\rfloor + \left\lceil \frac{a_0}{d} \right\rceil = \left\lfloor \frac{a_1\nu d}{d\beta} \right\rfloor = \left\lfloor \frac{a_1\beta}{d} + \frac{a_1}{\beta} \right\rfloor \leq \left\lfloor \frac{a_1\beta}{d} \right\rfloor + 1 = q+1$$

As with the Brent and Zimmermann variant, we have the issue that ν will be greater than β and thus requires multiple machine words. The pseudocode in Figures 4 and 5 implements Algorithm 2 efficiently.

We begin with the *BQS3_approx* routine in Figure 4, which computes the following function:

$$\nu' = \begin{cases} \beta - 1 & \text{if } d = \beta/2 \\ \left\lceil \frac{\beta^2}{d} \right\rceil - \beta & \text{if } \beta/2 < d < \beta \end{cases}$$

The proof of correctness is straight-foward. Line 4 catches the $\beta/2$ case. The other cases go to line 8, and since $d > \beta/2$ and d is normalized, d must not be a power of two, and therefore does not divide β^2, and we have:

$$\left\lfloor \frac{\beta^2 - d}{d} \right\rfloor + 2 = \left\lfloor \frac{\beta^2}{d} \right\rfloor + 1 = \left\lceil \frac{\beta^2}{d} \right\rceil$$

Line 9 then subtracts β. Thus BQS3_approx returns $\nu - \beta - 1$ when $d = \beta/2$, and $\nu - \beta$ when $d > \beta/2$. This is attractive because the approximation, ν' fits in a single machine word, whose value ranges from 1 to $\beta - 1$.

The *BQSv3* algorithm, shown in Figure 5 uses the ν' approximation to compute something closely related to Algorithm 2. Where they differ, we will show that BQSv3 produces the correct result.

Proof of correctness: the first thing we must show is that even though the algorithm ignores carry outs in line 7, if a carry out had occurred, then it would trigger line 9, which saturates q^* with $\beta - 1$. This proof is similar to the proof in Section II. We begin by noting that if $a_1 = \beta - 1$, then line 7 must carry out (because add is either 1 or 2), and, the if statement on line 8 will be triggered, which will result in q^* being set to $\beta - 1$ as required. Next, let's assume that we get a carry out in line 7, but the if statement on line 8 is not triggered. We have three conditions, $a_1 \leq \beta - 2$, a carry out, i.e., $q^* + \beta = \lfloor \nu' \cdot a_1/\beta \rfloor + a_1 + \text{add}$, and $q^* \leq a_1$:

$$\begin{aligned} a_1 \leq q^* \quad &\Longrightarrow \quad a_1 + \beta \leq \left\lfloor \frac{\nu' a_1}{\beta} \right\rfloor + a_1 + \text{add} \\ &\Longrightarrow \quad \beta^2 \leq \nu' a_1 + \text{add} \cdot \beta \end{aligned}$$

Since $\nu' \leq \beta - 1$, add ≤ 2 and $a_1 \leq \beta - 2$, we have:

$$\beta^2 \leq \nu' a_1 + \text{add} \cdot \beta \leq (\beta - 1)(\beta - 2) + 2\beta = \beta^2 - \beta + 2$$

```
1  uint BQSv3(uint a0, uint a1,
2             uint divisor, uint ν') {
3    uint qstar, y0, y1, add, correct;
4
5    add = (a0<divisor) ? 1 : 2;
6    y1 = mul_high(ν', a1);
7    qstar = y1+a1+add;
8    if(qstar < a1)
9      qstar = β − 1;
10   (y0,y1) = mul_wide(qstar, divisor);
11
12   // first correction step
13   y0 = sub_cc(a0, y0);
14   y1 = subc_cc(a1, y1);
15   correct = subc(0, 0);
16   qstar = qstar + correct;
17
18   // second correction step
19   y0 = add_cc(y0, divisor);
20   y1 = addc_cc(y1, 0);
21   qstar = addc(qstar, correct);
22   return qstar;
23 }
```

Figure 5. Barrett Quotient Selection implementation #3

Contradiction. Finally, we note, when $a_1 \leq \beta - 2$, a double carry out on line 7 is not possible. We can conclude that line 9 is run if and only if there was a carry out on line 7 as required.

For the next part of the proof, there are three cases.

Case 1. Suppose $d > \beta/2$ and $a_0 > 0$, then we have

$$\begin{aligned} \left\lfloor \frac{\nu' a_1}{\beta} \right\rfloor + a_1 + \text{add} &= \left\lfloor \frac{\nu' a_1}{\beta} \right\rfloor + a_1 + \left\lceil \frac{a_0}{d} \right\rceil \\ &= \left\lfloor \frac{a_1(\nu' + \beta)}{\beta} \right\rfloor + \left\lceil \frac{a_0}{d} \right\rceil \\ &= \left\lfloor \frac{a_1\nu}{\beta} \right\rfloor + \left\lceil \frac{a_0}{d} \right\rceil \end{aligned}$$

Thus lines 5-9 compute $q^* = \min\left(\left\lfloor \frac{a_1\nu}{\beta} \right\rfloor + \left\lceil \frac{a_0}{d} \right\rceil, \beta - 1\right)$, exactly as Algorithm 2.

Case 2. Suppose $d > \beta/2$ and $a_0 = 0$. This case is almost the same, except $\left\lceil \frac{a_0}{d} \right\rceil = 0$, whereas add will be 1. Thus, the q^* estimate computed in lines 5-9 will be one greater than the q^* estimate of Algorithm 2. However, we also showed that when $a_0 = 0$, q^* of Algorithm 2 is at most one correction step away from q. Thus the q^* computed in lines 5-9 is at most two correction steps from q, as required.

Case 3. Suppose $d = \beta/2$. In this case, lines 5-9 compute (excluding the min):

$$\begin{aligned} q^* &= \left\lfloor \frac{(\beta - 1)a_1}{\beta} \right\rfloor + a_1 + \text{add} \\ &= 2a_1 - 1 + \text{add} = 2a_1 + ((a_0 < \beta/2)?0 : 1) \end{aligned}$$

since d is a power of two, we can think of this division as an arithmetic shift right by $w - 1$ bits, in which case, the correct

```
// set carry=1 if lo>=d, 0 otherwise
sub_cc(a0, divisor);
// add=hi + 1 + carry
add = subc(a1, β − 2);
y1 = mul_high(ν′, a1);
qstar = y1 + add;
```

Figure 6. Leveraging the carry flag to replace the ternary operation

answer should be

$$\begin{aligned} q &= (a_1\beta + a_0) >> (w-1) \\ &= 2a_1 + (a_0 >> w - 1) \\ &= 2a_1 + ((a_0 < \beta/2)?0 : 1) \end{aligned}$$

in other words, q^* will be exactly q with no correction steps required.

Lines 12-21 perform the two unrolled correction steps in an efficient manner, and the correctness argument is similar to the earlier algorithm. We can conclude that the pseudocode in Figures 4 and 5 is correct.

There is one last trick we can do to improve the performance of *BQS3*. We can replace lines 5-7 with the code in Figure 6, which leverages the carry flag to replace the ternary operator, which further improves the performance on the GPU.

IV. Experimental Setup and Results

To test our Quotient Selection, we run four implementations: the three variants, BQSv1 (see Figure 2), BQSv2 (see Figure 3), BQSv3 with the carry replacement for the ternary operator (see Figures 5 and 6), and the fourth uses the compiler's built-in 64-bit divide. The implementations are each run on four different NVIDIA GPU cards: a GTX Titan Black (Kepler micro-architecture), a GTX 980 (Maxwell), a Tesla P-100 (Pascal), and a Tesla V-100 (Volta). The GTX Titan Black and GTX 980 were hosted on a consumer grade machine (Core i5-7400 at 3 GHz, Gigabyte B250M-DS3H motherboard, 16 GB memory) running 64-bit Ubuntu 16.04.1. The two Tesla cards were hosted on a server grade machine (Xeon E5-2650 v3 at 2.3 GHz, 128 GB memory) running CentOS. The operations listed in Table 1 were carefully optimized in a combination of C and PTX assembly language, to gain access to the carry flags and instructions that generate and make use of them.

To test each algorithm, we generate 100K[1] random 64-bit numbers and one random normalized 32-bit divisor per thread (the GPUs all use 32-bit ALUs) and run 2048 blocks of 128 threads. Each thread runs the 100K random instances divided by its random divisor (and pre-computed approximation). We use a shuffling scheme so that each thread runs the instances in a slightly different order, which ensures that the performance results reflect any warp divergence in the code. The threads each compute a checksum of all the resulting quotients and write the checksum word to global memory. The measured running time is the time to load the 100K numerators from GPU global memory, compute the quotients and checksum, and write the final checksum back to global memory. It does not include the time for the CPU to generate the random data, copy data to/from the GPU or to verify the results. Each implementation (kernel) is run 10 times and we report the average running time (in milliseconds) of each implementation on each of the four GPUs in Table II.

[1]On Volta, we run 1M instances to reduce the run-to-run variability and normalize the running time by dividing by 10.

TABLE II
Running time of the implementations on four GPUs (time in ms)

	GTX Titan Black (Kepler)	GTX 980 (Maxwell)	Tesla P-100 (Pascal)	Tesla V-100 (Volta)
BQSv1	329.9	377.0	216.1	73.7
BQSv2	324.8	358.9	199.9	79.3
BQSv3	265.9	303.2	164.5	63.9
Compiler	2095.4	2027.5	1151.7	317.4

TABLE III
Running time ratios of the implementations on four GPUs

	GTX Titan Black (Kepler)	GTX 980 (Maxwell)	Tesla P-100 (Pascal)	Tesla V-100 (Volta)
BQSv1	1.24	1.24	1.31	1.15
BQSv2	1.22	1.18	1.22	1.24
BQSv3	1	1	1	1
Compiler	7.88	6.69	7.00	4.96

On Kepler, Maxwell and Pascal, BQSv2 is a bit faster than BSQv1. On Volta it's the reverse, with BSQv1 beating BQSv2. All three versions are much faster than the compiler built-in 64-bit divide. Of the variants, BQSv3 is the fastest on all architectures. In Table III, we scale the running times by the fastest implementation on each GPU (BQSv3). We see that BQSv3 is roughly 15% to 30% faster, with an average improvement of 22% over the other two Barrett variants.

V. Conclusion

Quotient selection can be a performance-limiting step in multiprecision division on processors that lack hardware support for division, such as GPUs, DSPs, and some embedded processors. We have presented implementations of variants of Barrett's quotient selection algorithm that significantly outperform the use of compiler-generated division operations on four models of NVIDIA GPU. One of the implementations is based on a direct interpretation of a variant proposed by Brent and Zimmermann. The second adds an optimization we identified that provides a modest improvement to their correction step. We then present a new variant on Barrett's algorithm that corrects downwards, rather than upwards. All three of our highly-optimized C/assembly implementations outperform quotient selection using the compiler-generated division operation. But our new variant of the Barrett algorithm outperforms both versions of the Brent and Zimmerman variation by an average of 22%. We also provide proof sketches to show that all of the implementations are correct. These implementations assume a fairly common set of underlying hardware capabilities that should make the applicable to a wide range of processors other than NVIDIA GPUs.

ACKNOWLEDGMENT

This work was partially supported by National Science Foundation (NSF) under Award No. CCF-1525754 and National Key R&D Program of China under Award No. 2017YFB0802103.

REFERENCES

[1] Paul Barrett. *Communications Authentication and Security Using Public Key Encryption: A Design for Implementation*. PhD thesis, University of Oxford, 1984.

[2] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[3] Joppe W. Bos and Simon Friedberger. Fast arithmetic modulo 2^x p^y $\pm$ 1. In *24th IEEE Symposium on Computer Arithmetic, ARITH 2017, London, United Kingdom, July 24-26, 2017*, pages 148–155, 2017.

[4] Richard P Brent and Paul Zimmermann. *Modern computer arithmetic*, volume 18. Cambridge University Press, 2010.

[5] C. Burnikel and J. Ziegler. Fast recursive division. Technical Report MPI-I-98-1-022, Max-Planck-Institut fuer Informatik, 1998.

[6] Zhengjun Cao and Xiangjia Wu. An improvement of the barrett modular reduction algorithm. *Int. J. Comput. Math.*, 91(9):1874–1879, 2014.

[7] JF Dhem. Modified version of the barrett algorithm. Technical report, Technical report, Jul, 1994.

[8] Rémi Géraud, Diana Maimut, and David Naccache. Double-speed barrett moduli. In *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, pages 148–158, 2016.

[9] Robert E Goldschmidt. Applications of division by convergence. *Massachusetts Institute of Technology*, 1964.

[10] Johann Großschädl. High-speed RSA hardware based on barret's modular reduction method. In *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, pages 191–203, 2000.

[11] William Hasenplaugh, Gunnar Gaubatz, and Vinodh Gopal. Fast modular reduction. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, pages 225–229, 2007.

[12] A. H. Karp and P. Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):561–589, 1997.

[13] Miroslav Knezevic, Lejla Batina, and Ingrid Verbauwhede. Modular reduction without precomputational phase. In *International Symposium on Circuits and Systems (ISCAS 2009), 24-17 May 2009, Taipei, Taiwan*, pages 1389–1392, 2009.

[14] Miroslav Knezevic, Frederik Vercauteren, and Ingrid Verbauwhede. Faster interleaved modular multiplication based on barrett and montgomery reduction methods. *IEEE Trans. Computers*, 59(12):1715–1721, 2010.

[15] Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998.

[16] Wolfgang Mayerwieser, Karl C Posch, Reinhard Posch, and Volker Schindler. Testing a high-speed data path the design of the rsaβ crypto chip. In *J. UCS The Journal of Universal Computer Science*, pages 728–743. Springer, 1996.