Outperforming Sequential Full-Word Long Addition With Parallelization and Vectorization

Andrey Chusov

Abstract—The article presents algorithms for parallel and vectorized full-word addition of big unsigned integers with carry propagation. Because of the propagation, software parallelization and vectorization of non-polynomial addition of big integers have long been considered impractical due to data dependencies between digits of the operands. The presented algorithms are based upon parallel and vectorized detection of carry origins within elements of vector operands, masking bits which correspond to those elements and subsequent scalar addition of the resulting integers. The acquired bits can consequently be taken into account to adjust the sum using the proposed generalization of the Kogge-Stone method. Essentially, the article formalizes and experimentally verifies parallel and vectorized implementation of carry-lookahead adders applied at arbitrary granularity of data. This approach is noticeably beneficial for manycore, CUDA and vectorized implementation using AVX-512 with masked instructions. Experiments show that the parallel and vectorized implementations of the proposed algorithms can be multiple times faster compared to a sequential ripple-carry adder or adders based on redundant number systems such as one used in the GNU Multiple Precision library.

Index Terms—AVX-512, CUDA, full-word addition, Long arithmetic, parallel arithmetic, SIMD, SMP, vectorization

1 INTRODUCTION

Use of big-integral arithmetic is required in many fields of technology and science such as cryptography [1], digital signal processing [2] and high-precision scientific computations [3].

Additive operations provide a foundation for most of the operations of big-int arithmetic. Although most of modern published research is focused on big-integer multiplication, addition plays a significant role in it. For instance, every integral multiplication algorithm in [4] and [5], i.e., the schoolbook multiplication, the Comba’s method, the Toom-Cook multiplications (including the Karatsuba’s method), as well as asymptotically more efficient Fourier transform based algorithms over rings or floating-point algebras (e.g., Schönhage–Strassen [6] and Fürer multiplication [7]) rely on a separation of multiplicands to smaller values, which in turn have to be multiplied, using the same or different algorithm, and then recombined using long addition to produce a product.

Specifically, the family of Toom-Cook algorithms compute values of polynomials, which represent the multiplicands and the product, at several different points on the curves defined by the polynomials. Then the final product is computed as a value of a polynomial which is a result of interpolation given smaller products at the chosen set of points. Given the right values of the chosen points the problem of multiplication is reduced in size. The algorithm thus involves long addition during computation of the polynomials, during their smaller pointwise multiplications as well as interpolation of the result to yield the final product.

The Fourier transform based multiplication algorithms [6], [7] and [8] rely on the convolution theorem to compute the product. Depending on the algorithm, the multiplicands are represented using a ring of either integer or complex values. The choice of the ring as well as its principal root of unity used in the Fourier transform defines the choice of the discrete (DFT) or fast (FFT) Fourier transform algorithm as well as implementation of elementwise multiplication, addition and remainder computations. First, the DFT by itself, whether direct or using one of the FFT algorithms, requires at best $O(n \log n)$ (in case of radix-2 Cooley-Tukey FFT) and at worst $O(n^2)$ additions. The size of additions again depends on the choice of the principal root of unity and the underlying ring of integers or the required precision of floating-point representation of complex values with the latter formally described in [7] and [2]. Particularly, the rings of integers used in [6] and [7] are equipped with arithmetic operations, including addition, modulo $2^n + 1$. In order to implement those, one could use an approach described in [9] particularly for such moduli. The approach represents modulo $2^n + 1$ addition and subtraction in terms of additive operations modulo $2^n$ which can be implemented using the algorithms proposed in this paper. Furthermore, the acquired product, which results from the inverse Fourier transform of the pointwise multiplication of the transformed multiplicands, needs to be recombined into a final scalar product by adding up parts of the pointwise products to perform the multiplicative carry operation which again requires long addition.

Therefore, importance of long addition is at least comparable to one of long multiplication both of which provide the foundation for many algorithms and implementations as stated above. One way to improve performance of the operations is to develop a way to compute parts of the initial problem in parallel. While in many respects multiplication
can be conducted in parallel, such as evaluation of polynomials at different points or components of the Fourier transform of the inputs, the addition operation with the requirement to propagate excessive higher-order digits make the addition (as well as subtraction and additive steps of multiplication) poorly parallelizable due to data dependencies.

This, in particular, hinders vectorization of addition, and implementers of long addition (e.g., the GNU multiprecision library, as of version 6.2.0) choose scalar ripple-carry addition (e.g., ADD/ADC instructions in the case of x86 and x86-64 architectures) in favor of SIMD. Indeed, a number of operations required to correctly propagate and take into account the carry flag after addition of each element of the addends is at least equal to the number of elements within the addends, and thus vectorization and parallelization might yield no performance benefits.

Yet detection of digits which can produce carry flags can partially be performed in parallel, which provides an opportunity to increase performance and energy efficiency of adders at a cost of increasing complexity of implementation.

A detailed overview of methods to implement adders at a bit-level granularity can be found in [10]. The most straightforward way to implement addition of long integers is based upon a ripple-carry adder which adds up digits of addends sequentially propagating carry bits. Many implementations rely on this method because it is cheap, easy to implement both as a program and as a system of logic gates with minimal wiring and respective latencies. This adder can easily outperform (see Section 6) other more complicated methods especially when size of addition is small.

Yet detection of digits which will or might produce carry bits is easily parallelizable when no carry propagation resulting from adding up less significant digits is performed. General approach to such operation is given in [11] and [12] both of which provide algorithms commonly known as carry-lookahead adders. These adders precompute sums without carry propagation and then perform parallel reduction to iteratively adjust the result to account for carry propagation. This approach is widely used in implementations of addition when performance and energy efficiency is important [13], [14], [15] and [16].

The disadvantage of carry-lookahead methods is the complexity of their implementation. For instance, in [17] the authors combine the two approaches, i.e., Kogger-Stone [11] and Brent-Kung [12], to find a balance between sparsity of implementations of the former and latency of the latter caused by an increased number of sequential operations. Another way to find a balance between performance and complexity of addition is to use a tree-like structure of addition algorithm which conditionally adjusts elements of the carry-less sum based on whether less-significant digits yield carry; if they do not, the adjustment can be skipped [15], in which case the performance gain may take place. Carry-select adders operate similarly regarding the selection of either a sum or its incremented value based on a value of an inbound carry bit produced as a result of lower-order addition. Works such as [18] and [19] propose a design of such adders with the primary purpose of reducing complexity of adders.

Another approach to implement parallel addition is to express the sum in a redundant number system in which a given value can be represented in more than one way. Such are carry-save [10] and borrow-save [20] adders, both are roughly equivalent performance-wise. These allow addition with no interdependence of digits provided that the representation of at least one of the addends and of the sum remains in the respective redundant number system. Though fast and scalable by themselves, conversion between the redundant and non-redundant number systems will at best make their performance equivalent to one of ripple-carry adders because the conversion requires adjustments of digits in the non-redundant number system, which is linear in time, as well as extra memory (and IO operations) to store the values in the redundant-number system.

These works focus primarily on hardware implementation of adders and employ bit-level parallelism in order to improve performance and energy-efficiency of addition. Also, in existing literature the discussed adders mostly implemented for short operands, and addition of arbitrary-length integers is represented poorly [21]. This paper on the other hand provides ways to implement parallel carry-lookahead addition of arbitrarily long integers at a machine word level using a set of common arithmetic instructions on word-sized scalars and vectors thereof.

For that, the paper formalizes and presents two algorithms for parallel addition of long full-word unsigned integers using widespread scalar multiprocessors and vectorization. Essentially, results are a generalization of carry-lookahead adders, especially the Kogger-Stone adder, to add up numbers represented as arbitrary-lengths vectors of machine words under the assumption that a pair of words can be added up in a constant time by the underlying hardware. Capabilities required from a vector processor in order to implement such addition with performance gains are also investigated.

The paper concludes with experimental evaluation of performance of the proposed adders implemented for several underlying architectures and compares the results with the sequential implementation of long addition using the Intel ADC instruction. The latter can be thought of as either a ripple-carry adder (and regarded as such in the paper) or, given an equivalence between a multiplexer of a carry-select adder and the ADC instruction both of which sequentially select/produce a sum of their operands adjusted given a carry bit.

The results show significant performance boost of the proposed parallel and vectorized algorithms for addition arbitrarily long full-word integers. Besides parallelism, the performance is gained because there is no need to convert data between different number systems or to perform unnecessary data movement, the latter of which is shown to be highly impactful on overall performance. The other main contribution of the paper is that it is agnostic about a (full-word) number system used and does not limit or fix sizes of the addition problem which makes the results applicable in wide range of use cases.

Throughout the paper the following notations are used. Vectors which represent long integers of \(n\) words are in the group \((\mathbb{Z}_W, +)\) which is isomorphic to \((\mathbb{Z}_W, +)\). Particularly, each scalar \(v_i\) \((0 \leq i < n)\) can be either an unsigned integer from \((\mathbb{Z}_W, +)\) or a signed integer provided that the latter has \(W\)’s complement representation. Additionally, the branchless
carry detection algorithm for signed integers presented in the Section 2 relies on bit representation of \( v_i \). In this (and only in this) case, \( W \) is assumed to be a power of two.

Also, separate \( b \)-bit scalars are used by the presented algorithms to perform carry propagation. Therefore, the underlying platform is assumed to provide a way to implement integral \( b \)-bit arithmetic, i.e., addition in \( \langle Z_B, + \rangle \), where \( B = 2^b \).

There are three addition operations used in the paper, those are: ordinary scalar addition "+" defined for an additive group \( \langle Z, + \rangle \) which contains the long integers being added up; bitwise XOR, denoted as \( \oplus \), of elements in \( \langle Z_{2^a}, \oplus \rangle \) (i.e., addition homomorphic to one of polynomials in \( \{ p \in Z_2[x] \mid \deg(p) < n \} \) and a vector addition \( \oplus \) of \( n \)-length vectors in \( Z_W \) (with respective scalars added up modulo \( W \)) to form an additive group \( \langle Z_W^n, \oplus \rangle \). The latter is isomorphic to a group of polynomials in \( \langle Z_W[x], \oplus \rangle \) of degrees less than \( n \).

## 2 Detection of Carry Origin

The first step of the addition algorithm is to detect origins of set carry flags resulting from addition of vector elements. This detection is easily performed using a comparison of vector values. For this, two cases need to be considered with respect to signed and unsigned comparison provided by vector processors. For instance, Intel AVX-512 and ARM SIMD extensions provide instructions (e.g., VPCMAMDU and CMHIS) for unsigned comparison. On the other hand, some processors only allow comparison of signed integers, as is the case with SSE-SSE4.2 and AVX2.

Addition of two unsigned scalars, \( x \) and \( y \), overflows if and only if \( \forall x, y \in Z_W : x + y < x \) (where addition is performed modulo \( W \)). Since there is no dependency here on anything other than \( x \) and \( y \), this detection, when applied to multiword integers, can be done in parallel using elementwise unsigned comparison. That is, given a pair of \( n \)-element vectors \( V = (v_0, \ldots, v_{n-1})^T \) and \( U = (u_0, \ldots, u_{n-1})^T \) as well as a function \( \text{CMPGT}(V, U) = (v_0 > u_0, \ldots, v_{n-1} > u_{n-1})^T \) which maps \( (Z_W^n)^2 \to Z_W^n \), the detection of the carry flag is then performed by the call \( C(V, U) = \text{CMPGT}(V, V + U) \) (or alternatively \( C(V, U) = \text{CMPGT}(U, V + U) \)).

In the case of signed vector comparison, the detection is a little more complicated as it requires to consider three cases (Fig. 1). As stated in the Section 1, the rest of this section assumes that elements of the vectors are represented with a set of bits. If this condition is met, one can proceed as follows. First, if the most significant bits of both addends, \( x \) and \( y \), are reset, then their addition will never yield a set carry flag, because addition of lower-order digits will yield a carry of at most 1 which, added to zero highest-order bit of the sum, will never produce a new carry. Second, if the most significant bits of both addends are set, then their arithmetic sum will always be either 2 (binary 10) or 3 (binary 11) with the left bit to become the value of the carry flag, and the right bit equals the value of the incoming (resulting from addition of less significant bits) carry. Otherwise, i.e., when most-significant bits of \( x \) and \( y \) differ, the resulting carry flag will always be equal to the carry produced by addition of lower-order bits and complement to the sum of most-significant bits of the operand modulo 2. Indeed, if the most-significant bit of one operand is set and the most-significant bit of the other operand is reset, then the result of arithmetic sum will equal one plus the carry of the lower-order bits, that is 1 (binary 01) or 2 (binary 10) with the left bit becoming the new carry flag which is the opposite of the right bit. Assuming that the most-significant bit is a sign bit, the latter check can be performed by an arithmetic comparison of the sum with zero.

To avoid branching, this detection can be represented in terms of bitwise operations upon \( x \) and \( y \); for signed integers \( x \) and \( y \) the value of the carry flag equals \( ((0 > x) \land (0 > y)) \lor ((0 > x) \land (x + y > 0)) \lor ((0 > y) \land (x + y > 0)) \equiv (0 > x \land 0 > y) \lor (0 > x + y) \lor (0 > x) \land (0 > y) \lor (0 > x + y) \lor (0 > x) \land (0 > y) \lor (0 > x) \land (0 > y)) \). Consequently, the similarly unsigned, these bitwise operations do not introduce data dependencies and thus can be applied to SIMD vectors. In this case, given a similarly defined vector comparison \( \text{CMPGT} \) (although for signed integers),

\[
C(V, U) = (c_1 \text{VAND} c_2) \text{VOR}(c_1 \text{VANDN} c_2),
\]

where \( c_1 = \text{CMPGT}(O, V), c_2 = \text{CMPGT}(O, U), c_1 = \text{CMPGT}(O, \text{VOR} U), O \) is a zero \( n \)-element vector, and VAND, VOR and VANDN are respectively bitwise AND, OR and AND-NOT operations upon respective elements of the operands. AND-NOT complements the first operand and evaluates bitwise conjunction of the result with the second operand.

## 3 Carry Propagation

The main problem which hinders efficient parallelization and vectorization of addition is the need for carry propagation, because adding up separate pairs of digits can cause additional carry flags to appear which cause data dependency of higher-order digits of the result upon the lower-order digits.

Nonetheless, once carry flags which stem from the addition of vector elements (as well as possibly a carry flag which originates from prior addition of less significant parts of the original long integers which the vectors are part of) are determined, the following approach can be used to replace \( n \) checks and increments of vector elements with \( [n/b] \) operations to accumulate bits produced by the function \( C(V, U) \) (given above) in an integral value stored in \( b \)-bit registers which allow arithmetic addition or subtraction. Any general purpose integral registers (or mask registers K0-K7 of AVX-512) can be used for it.

For brevity, in this section it is assumed that \( [n/b] = 1 \), which is the case for many vector processors. Otherwise, e.g., in parallel implementation of addition, the Kogge-Stone method [11] can be used to perform parallel reduction of \( b \)-bit values as shown in the Section 4.
Since the result of addition of any amount of lower-significant digits of a number can never produce a value of carry greater than 1 (the proof of that can be found in many classical sources, e.g., in [22]), the mask produced by the function \( C(V, U) \) can never have the bits set at the same positions as the positions of sum elements which need to be incremented during carry propagation. This allows to unite (using bitwise OR) the mask resulting from \( C(V, U) \), shifted once to the left, with the bit mask which corresponds to the position of the sum elements which generate carry flags as a result of carry propagation, and no information loss will take place.

For that, one first needs to determine elements of the vector sum \( V + U \) which may produce a carry flag as a result of carry propagation. Since the carry value is at most one, those elements have a defined value of \( W - 1 \), i.e., all bits of those elements are set. Therefore, vectorized comparison of the sum with a vector \( I = (W - 1 \cdots W - 1)^T \) using the approach similar to that used above can be applied here to obtain a mask

\[
I(V, U) = \text{CMPEQ}(V + U, I),
\]

where \( \forall V, U \in Z_W^n : \text{CMPEQ}(V, U) = ((v_0 = u_0) \cdots (v_{n-1} = u_{n-1}))^T \in Z_W^m \).

In order to formally associate scalar values with their vector counterparts we introduce the bijection \( v_{n-1}^W : Z_W^n \rightarrow Z_m^n \) (together with its inverse \( v_{n-1}^{W^{-1}} \)) between the respective naturally isomorphic sets as well as the injection \( p_{n-1}^W : Z_m^n \rightarrow Z_m^n \).

**Theorem 1.** Given two vector addends \( V, U \in Z_W^n \subset Z_W^{n+1} \) representing respectively the unsigned integers \( V = \sum_{k=0}^{n-1} v_k W^k \) and \( U = \sum_{k=0}^{n-1} u_k W^k \) as well as a bit \( \xi \in Z_2 \), the sum \( S = \sum_{k=0}^{n-1} s_k W^k \) of the two integers and \( \xi \) equals the vector sum

\[
S = v_{W}^W(V) + v_{W}^W(U) + \xi
= v_{W}^{n-1}
\left(V \boxplus U \boxplus (v_{W}^{n-1} \circ v_{W}^{(n+1)}(\xi))\right),
\]

where

\[
\varepsilon = (i + 2c + \xi) \oplus i \in Z_{2^{n+1}},
\]

\[
i = v_{2}^W(I(V, U)), \quad \text{and} \quad c = v_{2}^W(C(V, U)).
\]

**Proof.** First, let \( n = 1 \). Then, to take into account a possible carry flag, consider both \( v_{W}^W(V) = v_0 \) and \( v_{W}^W(U) = u_0 \) to be in the sets \( Z_W \times \{0\} \subset Z_W \times Z_2 \). Then adding up \( v_{W}^W(V) \) and \( v_{W}^W(U) \) produces a sum

\[
S' = v_{W}^W(V) + v_{W}^W(U) = v_0 + u_0
= (v_0 + u_0) \mod W + \left[\frac{v_0 + u_0}{W}\right] W
= v_{2}^W \left(\left(v_0 + u_0 \mod W, \left[\frac{v_0 + u_0}{W}\right]\right) \equiv v_{W}^{n-1}(V \boxplus U \boxplus \left(0 \oplus \left(\xi \oplus i\right)\right))\right)
\]

in which the vector \( (0 \ c)^T = (\gamma_{W}^2 \circ v_{W}^{-2}) (2v_{W}^W(C(V, U))) \). Then adding \( \xi \) to the sum yields

\[
S = S' + \xi = v_{W}^{n-1}(V \boxplus U \boxplus \left(0 \oplus \left(\xi \oplus i\right)\right))
\]

where \( \epsilon' \) is a carry bit resulting from adding \( \xi \) to the first element \( s_0' \in Z_W \) of \( S' \) and equals \( \xi \land (s_0' = W - 1) = \xi \land i \). Here \( c + \epsilon' \in Z_2 \) because either \( c = 1 \), i.e., \( s_0' < v_0 \) and \( s_0' < u_0 \) or \( \epsilon' = 1 \), i.e., \( \xi \land (s_0' = W - 1) \), because there are no values in \( Z_W \) that are greater than \( s_0' = W - 1 \), i.e., \( c \land \epsilon' = c \land i = 0 \). Therefore,

\[
\left(\begin{array}{c}
0 \\
0
\end{array}\right) \oplus \left(\begin{array}{c}
\xi \\
\epsilon'
\end{array}\right) = \left(\begin{array}{c}
\xi \\
\epsilon' \oplus i
\end{array}\right) = \left(\begin{array}{c}
\xi \\
\epsilon \oplus (\xi \land i)
\end{array}\right)
\]

(6)

Fixing given values of \( s_0' \) and \( W \) (and thus \( i = (s_0' = W - 1) \)) consider the additive group \( G_1 \) with the set \( \{t | (3 \xi, c \in Z_2) \land (t = (i + 2c + \xi) \land i)) \subseteq Z_2 \).

along with bitwise XOR as a group operation. Note that

\[
(i + 2c + \xi) \oplus i \Rightarrow (i + 2c + \xi) \land i = (c + \xi \land i) \cdot 2 + \xi.
\]

Therefore, \( G_1 \) is closed under bitwise XOR:

\[
(i + 2c + \xi) \land i = \left((c_1 + \xi \land i) \cdot 2 + \xi_1 \right) \land \left((c_2 + \xi \land i) \cdot 2 + \xi_2 \right) = \left((c_1 + \xi \land i) \cdot 2 + (\xi_1 + \xi_2) \right) \land i = (i + 2(c_1 + c_2) + (\xi_1 + \xi_2) + i),
\]

where \( (\neg (i \land c) \land \neg (i \land c_2)) \Rightarrow \neg(i \land (c_1 + c_2)) \), and thus \( G_1 \) is isomorphic to a group \( G_2 \) of linear polynomials in \( Z_2[x] \) with the set

\[
\{t | (3 \xi, c \in Z_2) \land (t = (i + \xi \land i) x + \xi) \subseteq Z_2 \}
\]

and a map \( G_1 \rightarrow G_2 : a \cdot b \mapsto ax + b \) which is a bijective homomorphism because \( (a_1 x + b_1) + (a_2 x + b_2) = (a_1 + a_2) x + (b_1 + b_2) \rightarrow (2(a_1 + a_2) + (b_1 + b_2), \) with zeroes of both groups only possible when \( c = 0 \) and \( \xi = 0 \) because \( (c + \xi \land i) \land i = (c = 0) \land (\xi = 0) \land (i = 0) \land (c = 0) \land (\xi = 0) \land (i = 0) \land (c = 0) \land (\xi = 0) \land (i = 0) \land (c = 0) \land (\xi = 0).

Therefore, applying the bijection \( v_{2}^W = v_{2}^W \circ v_{2}^W \rightarrow \epsilon \) in (4) yields (5), and thus (3) holds for \( n = 1 \).

Then from (5) and (6) it follows that \( S \mod W = (v_0 + u_0 + \xi) \mod W = v_{W}^{n}(V \boxplus U \boxplus \xi) \), and \( [S/W] = c \oplus (\xi \land i) = \left[\frac{v_{2}^W\xi}{W}\right] \subseteq Z_2 \).

Now, if \( n > 1 \), the proof can proceed by induction: if (3) and (4) hold for \( n \), then they also hold for \( n + 1 \).

Representing the addends, \( v_{W}^{n+1}(V) \) and \( v_{W}^{n+1}(U) \), as respectively \( v_{W}^{n+1}(V) \) and \( v_{W}^{n+1}(U) \) which yields the sum:

\[
S = (v_1 + u_1) W + (v_0 + u_0 + \xi) = v_{1} + u_{1} + \left[\frac{v_0 + u_0 + \xi}{W}\right] W + (v_0 + u_0 + \xi) \mod W = v_{1} + u_{1} + \left[\frac{i_0 + 2c + \xi}{W}\right] W + (v_0 + u_0 + \xi) \mod W,
\]

where \( i_0 = v_{1}(v_0, u_0) \), and \( c_0 = v_{1}(C(v_0, u_0)) \).
Note that $\forall v \in Z_W^n$
\[ v_W^n(v) \cdot W = v_{W}^{n+1}(0) \]
and $\forall \varepsilon \in Z_2^n$
\[ \left( y_W^{n+1}(v) - v_2^{(n)}(v) \right)(\varepsilon) = y_W^{n+1}(v_2^{(n)}(v)) \left( y_W^{(n+1)} - v_2^{(n)}(v) \right)(2\varepsilon). \]

Then, based on the induction hypothesis and on the proof for the base case $n = 1$,
\[ S = W \cdot v_W^{n+1}(v_1) \oplus v_W^{(n+1)}(u_1) \oplus \left( y_W^{n+1}(v_2^{(n)}(v)) \right)(\varepsilon) + v_W^{(n+1)}(v_2^{(n)}(v_1)) \right( y_W^{(n+1)} - v_2^{(n)}(v_1) \right) \]
\[ = v_W^{n+1}(v_2^{(n+1)}(v_1)) \right( y_W^{n+1}(v_2^{(n)}(v_1)) \right) \]
\[ = v_W^{n+1}(v_2^{(n+2)}(v_1)) \right( y_W^{n+1}(v_2^{(n+1)}(v_1)) \right) \]
\[ + \left( \left( (i_0 + 2c_0 + \varepsilon) \right) \oplus i_0 \right) \right) \oplus (2i_1 + i_0) = \left( (i_0 + 2c_0 + \varepsilon) \right) \oplus i_0 + (2i_1 + i_0) = (i_1 + 2c_0 + \varepsilon) \oplus i. \]

Substitution of this to (7) yields (3).

\[ \Box \]

Computation (3) provides a way to parallelize addition as described in the following sections. This can provide performance increase, if, first, $v_W^{n+1}$ is cheap (e.g., it is the case when the map is just a re-identification of the same data and does not, for instance, involve data transfers which can be expensive as is the case with CUDA); and, second, if computations of $\varepsilon$ and $\left( y_W^{n+1}(v_2^{(n+1)}(v_1)) \right)(\varepsilon)$ are efficient. Impacts of these computations essentially determine overall efficiency of the adders as shown below.

4 Parallelization

Using (3) one can add up two long integers in parallel as follows.

Addition of two integers, $X, Y \in Z_W^n$ with $W > 2$ and $n > 0$, involves parallel carry-less addition $v_W^n(X) \oplus v_W^n(Y)$ using $T$ threads of execution in time $O(n/T)$. Consequently, the carry propagation has to be performed using (4). Computation of $\varepsilon \in Z_B$ (where $B = 2^a$) also requires possibly long addition of the bit masks $i$ and $2c + \varepsilon$ followed by a bitwise XOR with $i$. The latter is a carry-less addition of binary digits which requires no special effort for parallelization. The sum $2c + \varepsilon$ also requires no carry propagation because $\varepsilon$ is a one-bit value, and $2c$ is simply a linear bit shift to the left (towards the most significant position) by one, therefore $2c + \varepsilon = (c < < 1) \oplus \varepsilon$. But the computation of the sum $i + (2c + \varepsilon)$ again requires carry propagation which, in turn, can also be performed using (3). Then the original addition of size $n$ is reduced to addition of $\lceil n \log_2 W \rceil$ sized values of $i$ and $2c + \varepsilon$. Likewise, the addition of $i$ and $2c + \varepsilon$ is reduced to the addition of $\lceil n \log_2 W \rceil^2$. This reduction continues up until the size of addition is reduced to 1 which requires one scalar addition in $Z_W$, i.e., $\lceil n \log_2 W \rceil^2$. Therefore, the total number of base-$W$ digits to add up is
\[ \sum_{k=0}^{\max} \left[ \frac{n}{(\log_2 W)^k} \right] = O\left( \frac{n \log_2 W - 1}{\log_2 W - 1} \right) \]
for all such that $k < \max$ it follows that $\left[ \frac{n}{(\log_2 W)^k} \right] > 1$, that is
\[ \max = \left[ \log_{\log_2 W} n \right] = \left[ \log_{\log_2 W} \right] \frac{n}{\log_2 W}. \]

Since parallel addition (without $\varepsilon$) can be performed in parallel by $T$ threads, and the same parallel addition is performed to addends for each $k$, total time required to perform the addition with $T$ threads can therefore be estimated as
\[ \sum_{k=0}^{\max} \left[ \frac{n}{T (\log_2 W)^k} \right] = O\left( \frac{n \log_2 W - 1}{T (\log_2 W - 1)} \right). \]

Note that in order to achieve (10), one should exclude race conditions and limit the use of blocking synchronization of threads accessing different parts of generated $i$ and $c$. This can be achieved by choosing (possibly increasing) values of
to be a multiple of a size of words which \(i\) and \(c\) are composed of. In particular, if \(i\) and \(c\) are implemented as vectors in \(\mathbb{Z}_W^m\), then \(\frac{n}{T(\log_2 W)^k}\) should be a multiple of \(\lfloor \log_2 W \rfloor\) for every thread to set bits within a subset of its own words.

The above derivation can also be applied to (8) with \(T = 1\). More precisely, space required by this algorithm excluding \(3n\) words of the addends and the sum (i.e., space required to store values of \(i\), \(c\) and \(\varepsilon \mod 2^u\)) can be estimated with the upper bound:

\[
S_f = 3 \sum_{k=1}^{\max k} \left\lfloor \frac{n}{(\log_2 W)^k} \right\rfloor = 3 \sum_{k=1}^{\max k} \left\lfloor \frac{n(\log_2 W)^k - 1}{(\log_2 W)^k} \right\rfloor
\]

\[
= 3 \left( \max k + \sum_{k=1}^{\max k} \frac{n - 1}{(\log_2 W)^k} \right)
\]

\[
= 3 \left( \max k + \sum_{k=1}^{\max k} \frac{n}{(\log_2 W)^k} \right)
\]

\[
= 3 \left( \max k + \frac{n - 1}{(\log_2 W)^{\max k} - 1} \right)
\]

\[
= 3 \left( \max k + \left[ 1 + \frac{(n - 1)(\log_2 W)^{\max k} - 1}{1 - (\log_2 W)^{\max k}} \right] \right)
\]

\[
= 3 \left( \max k + \left[ 1 + \frac{-n}{(\log_2 W)^{\max k} - 1} \right] \right)
\]

\[
= 3 \left( \max k + \left[ 1 + \frac{-n}{(\log_2 W)^{\max k} + 1} \right] \right)
\]

\[
= 3 \left( \max k + \left[ 1 + \frac{-n}{(\log_2 W)^{\max k} + 1} \right] \right)
\]

This bound is particularly useful to assess maximal \(n\) such that the totally occupied space \(S\) does not exceed available memory \(G\), i.e.

\[
S \leq 2n + (n + 1) + S_f + \max k \leq G
\] (11)

which is composed of \(2n\) words of the addends, \(n + 1\) words of the result with extra word for carry bit, \(S_f\) words of \(c\), \(i\) and \(\varepsilon \mod 2^u\) as well as \(\max k\) words for holding carry bits of \(\varepsilon\) on each iteration of the algorithm (cf. Fig. 2). Let \(L = \log_2 W\). Then

\[
S \leq 3n + 4\max k + 4 + 3 \left[ \frac{n^2 - 3n + 1}{n(L - 1)} \right]
\]

\[
= 3n + 4\max k + 4 + 3 \left[ \frac{n(L - 1) + 3}{n(L - 1) - L - 1} \right]
\]

\[
V \vdots n \rightarrow V \equiv U \quad (v_w^U + v_w^C(U)) \bmod W^*
\]

\[
U \vdots n \rightarrow \gamma_W^{(n)}(U)
\]

\[
i \vdots \pi \rightarrow (i \equiv 2c) \bmod W \bmod \gamma_W^{(n)}
\]

\[
i_{\text{max}} \vdots \pi \rightarrow (i_{\text{max}} + 2c_{\text{max}}) \bmod W \bmod \gamma_W^{(n)}
\]

Fig. 2. Addition of \(X = v_w^U(V)\) and \(Y = v_w^C(U)\) from \(\mathbb{Z}_W^m\) modulo \(W^n\) using (3) and parallel reduction.

Here \(\frac{1}{n(L - 1)} - \frac{3}{L - 1} \leq \frac{2}{L - 1}\), therefore

\[
S \leq 3n + 4\max k + 4 + 3 \left\lfloor \frac{n - 2}{L - 1} \right\rfloor
\]

\[
\leq 3n + 4\max k + 4 + 3 \frac{n - 2}{L - 1} = 3 \frac{nL - 2}{L - 1} + 4k + 4.
\]

Now, if \(n \in \mathbb{Z}_N\), then from (9) it follows that \(\max k \leq \lfloor \log_L(N - 1) \rfloor\). Thus if

\[
3 \frac{nL - 2}{L - 1} + 4k + 4 \leq G
\]

then

\[
n \leq \frac{(G - 4\lfloor \log_L(N - 1) \rfloor)(L - 1) - 4L + 10}{3L}.
\] (12)

\[5 \text{ Vectorization}\]

As shown in the paper, scalar operations on bit masks \(i\) and \(c\), as if those masks were ordinary word-sized integers, gives a way to increase performance of vectorized addition and outperform existing scalar ripple-carry implementations.

This relies on the capability of the underlying parallel (vector) platform to efficiently implement both maps used in the theorem statement, i.e., \(v_w^U\) (as well as the inverse \(v_w^C\)) and \(\gamma_w^V\) (or the composition \(\gamma_w^V \circ v_w^C\)). This is the case with the new masking forms of AVX-512 instructions which consequently provides performance boost to carry-lookahead adders implemented using (3).

The Equation (3) can also be used to improve performance of addition with vectorization if, besides the addition, the maps \(v_w^C\) and \(\gamma_w^V\) are or can be implemented efficiently with a vector processor. While support for the former applied to results of comparisons (1) and (2) is widespread (e.g., PCMPEQ/PCMPGT in SSE2 and CMEQ/ CMHI in ARM), support for the latter map is noticeably limited, and while its implementation with lookup tables is possible, this requires access to external memory and thus can be inefficient. Experimentally, significant performance
start:
  xor rax, rax; memory index
; zmm2 := (-1 -1 -1 -1 -1 -1 -1 -1 -1),
; i.e. set all bits of zmm2
vpvtniogq zmm2, zmm2, zmm2, 0FFh
kxorw k1, k1, k1 ; k1 := 0
  loop_start:
; zmm0 := (V0 V1 V2 V3 V4 V5 V6 V7)
  vmovdq64 zmm0, [V + rax]
; zmm1 := (U0 U1 U2 U3 U4 U5 U6 U7)
  vmovdq64 zmm1, [U + rax]
; vector addition: zmm0 := zmm0 \oplus zmm1
  vpaddq zmm0, zmm0, zmm1
; k0 := w0(CMPGT(zmm0, zmm1)) = c
  vpcmpps k0, zmm0, zmm1, 1
  kaddw k0, k0, k0 ; k0 := k0 + k0
  korw k0, k0, k1 ; k1 := k0 + c
  j3 := w3(CMPEQ(zmm0, zmm2)) = i
  vpcmpeqq k1, zmm0, zmm2
  kaddw k0, k0, k1 ; k0 := k0 + k1
  kxorw k1, k0, k1 ; k1 := k0 \oplus k1
; zmm0 := zmm0 \oplus (w6 \oplus w7)(\bmod 2^4)
  vpsubq zmm0 \{k1\}, zmm0, zmm2
; j3 := \bmod (floor(k1 / 2^3) = \bmod (\bmod k1 / 2^3)
  kshiftw k1, k1, 8
; (s0 s1 s2 s3 s4 s5 s6 s7) := zmm0
  vmovdq64 [S + rax], zmm0
  add rax, 64
  cmp rax, 9
  jcc loop_start

Fig. 3. Vectorized addition $S = v_{W}^{(V)} + v_{U}^{(V)}$ from $Z_{W}$ modulo $W$ based on (3) using AVX-512 vector and mask instructions.

boost was only achieved with masked operations of Intel AVX-512 upon integral mask registers K0-K7. The implementation of the critical loop using AVX-512 in MASM syntax is shown in Fig. 3 in which 512-bit vector values are considered to be in $Z_{2^{512}}$ and mask values to be in $Z_{2^{64}}$ (to take extra carry bit of (4) into account).

In the other hand, the inverse transformation, $(y_{W}^{(V)} \circ v_{2}^{(V)})$, used in (3), is not widespread which makes its implementation less efficient. Since lengths of vector registers are usually small, the transformation can in many cases be implemented with a lookup table which maps short scalar integers to respective vector values. The obvious disadvantage of this is the need for a data structure in external memory, access to which can be expensive, though such a table can be small enough to reside in cache.

Such implementation for AVX-2 with 256-bit vector registers (and $n=4$) exhibited performance (see Fig. 10) comparable to the sequential adder.

AVX-2, as well as SSE-4.2, only provide for signed comparison of integers by means of the (V)PCMPGTx instructions. Therefore, in order to implement $C(V, U)$ in (4) one has to employ (1) for signed carry detection.

The workaround for a lack of hardware means to easily implement $(y_{W}^{(V)} \circ v_{2}^{(V)})$ can be used lookup tables (LUT) due to shortness of ymm and xmm registers. The AVX-2 implementation for this paper (Fig. 4) uses the lookup table which can be represented as a matrix, in $Z_{2^{64}}$, composed of values $m_{y_{W}^{(V)}} = -(\lceil \varepsilon \cdot 2^{k} \rceil \bmod 2)$ with $\varepsilon$-th row loaded into an AVX-2 register, where $\varepsilon$ is a result of evaluation of (4). The $\varepsilon$-th row contains a binary representation of the number $\varepsilon$ using the alphabet $\{0, -1\} \bmod 2^{k}$. Likewise, SSE-4.2 can use a similar matrix in $Z_{2^{64}}$. In general, a LUT to implement the map $(y_{W}^{(V)} \circ v_{2}^{(V)})$ would require $n \cdot 2^{n}$ elements of the set $Z_{W}$.

If vector registers are small, such as 128 bit SSE registers which can fit in two 64 bit scalars, carry detection and propagation, i.e., computation and addition of $(y_{W}^{(V)} \circ v_{2}^{(V)})(\varepsilon)$ in (3) and (4), can be done without a LUT using horizontal shuffling of scalars. However, this would make computation of the adjustment $(y_{W}^{(V)} \circ v_{2}^{(V)})(\varepsilon)$ for the vector sum sequential and thus is neither scalable nor efficient. Indeed, such implementation (cf. Fig. 8) showed no performance gain compared to the sequential scalar implementation using the ADC instruction nor to the implementation which utilizes a LUT similarly to the AVX-2 implementation described above.

The most beneficial way to combine the vector algorithm with one shown in Fig. 2 is by a thread to reduce its whole subvector to a set of singular bits, $c$ (obtained from $\varepsilon$ of the sum of the most significant elements of the subvectors) and $i$ (which can be obtained from accumulated bitwise AND of $v_{2}^{(U)}(0, i_{j})$) of elements $v_{j}$ and $y_{j}$ of the subvectors with the result equal to $2^{n} - 1$, should $i$ be set), write those bits to respective vectors of words, such that each word is occupied by one bit, then use parallel reduction to pack those bits together: $Z_{W} \rightarrow Z_{2^{n}} \rightarrow Z_{W}$. Elements of the result can then again be combined to obtain $\varepsilon$ using the parallel algorithm (Fig. 2) and/or the vector algorithm (Fig. 3). If the latter is used, one would need to include a way to double the value of integers representing the bits $c$ which can be accomplished by using bit-shift instructions (e.g., (V) PSLL in x86-64 or SHL in ARM) together with element permutation across lanes (respectively, (V)PSHUF and VEXT).
Measurements of performance of the algorithms were conducted using a system with Intel Core i9-10980XE CPU with 18 physical cores and Hyper-Threading enabled, with one NVIDIA Tesla P100 CUDA card operated by Ubuntu Workstation 20.04.3 LTS. The measurements were taken as a difference between two values of a CPU counter returned by the RDTSC instruction as well as differences between clock values of the underlying system (chrono::steady_clock library of C++ and CUDA events, the latter were used to measure performance of CUDA kernel code). Source code of a program which performs all the measurements, generates data tables and every plot provided in this section is available in the repository [25].

Different implementations of the parallel and vector addition algorithms were given randomized data sets representing addends of sizes from 64 bytes to 8 gigabytes. Measurements were taken ten times per experiment and then averaged. The results presented in the paper are produced arithmetic means.

Measurement of performance of the parallel algorithm (cf. Fig. 2) shows increase in performance of long addition (Fig. 5).

However, measurements show little scalability of the parallel algorithm, both with and without vectorization (Fig. 6), respectively achieving speedups of about 2.1 and 2.2 when the algorithm is executed to add up two 8 GiB values with 11 threads on a machine with 18 physical cores. When sizes of addends are close to cache sizes, the results are slightly better without vectorization: the maximal measured speedup of 16 MiB addition (which is close to L2 cache size) performed using a parallel scalar adder is 4.41 when executed using 16 threads which is close to a number of physical CPU cores in the machine.

This demonstrates that the limited scalability of the implementation is due to contention of threads accessing global memory (using quad-channel configuration) even if the threads write to memory addresses far enough to exclude false sharing. Indeed, assuming the thread contention which implies hardware blocking of threads accessing the memory with performance impact proportional to the number of threads, one can apply the model (11) and (12) in [26] for critical sections as shown in [27] to achieve asymptotic limits for speedups \( \sigma \). Assuming no asymptotically impactful sequential part of parallel the algorithm (due to (10)) the following simplified upper limit of speedup can be derived from the model in [26]:

\[
\lim_{T \to \infty} \sigma = \frac{1}{f_{\text{cont}} p_{\text{cont}}},
\]

where \( f_{\text{cont}} \) is a fraction of time spent by threads during blocking access to memory, and \( p_{\text{cont}} \) is the probability of contention.

Even though the model (11) in [26] is purely theoretical as it is described in terms of base core equivalents, which model CPU cores as defined in [28], it can be applied to a real-world manycore CPU and the parallel algorithm. For
that one has take into account the way threads (and, respectively, CPU cores) access words of the vectors $V$, $U$ and $V_w^{-1}(S)$. In order to reduce contention of the threads whose access to global memory and cache can be serialized, the vectors are split onto subvectors with its adjacent elements, if possible, processed in a single thread. This causes the elements of $V$ and $U$ to reside in a cache line read by a single CPU-thread and take advantage of the cache. If the addends are $n$ words in size, the thread $t$, $1 \leq t \leq T$, is thus assigned $n_t$ elements of the vectors starting from the index

$$
\left\{ \begin{array}{ll}
\lceil \frac{n}{T} \rceil t + t, & \text{if } t < n \mod T \\
\lceil \frac{n}{T} \rceil t + (n \mod T), & \text{if } t \geq n \mod T,
\end{array} \right.
$$

where

$$
n_t = \left\{ \begin{array}{ll}
\lceil \frac{n}{T} \rceil + 1, & \text{if } t < n \mod T \\
\lceil \frac{n}{T} \rceil, & \text{if } t \geq n \mod T.
\end{array} \right.
$$

Provided that the cache line is 64 bits in size, a thread reading an 8-byte word from the global memory causes the respective CPU core to cache the word together with seven adjacent words which are reused later. In the case of AVX-512 adder, the thread reads 64 bytes, i.e. one cache line, of $V$ or $U$ into a ZMM register. If $T > 4$, the threads contend for cache lines read from the global memory using four channels of the platform in question. Therefore, in this case the probability of contention is the probability of five or more threads, out of $T$, accessing the global memory which adheres the binomial distribution

$$
p_{\text{cont, } R} = \frac{T}{5} p_R(1 - p_R)^{(T-5)},
$$

where $p_R$ is a probability of a single thread reading $V$ or $U$ from global memory at a random point in time during the parallel addition process.

As soon as a cache line of a vector addend has been read (and seven of the eight 64-bit words of the corresponding vector sum $V \oplus U$ are acquired), the cache line can be discarded as the words no longer participate in the addition algorithm. Instead the thread writes the vector sum into a memory buffer to be adjusted using the evaluated $\varepsilon$ later on. This introduces a point in which cache misses occur as well as a thread contention for using cache of limited size: ideally the vector sum would remain in cache until after it is adjusted. However, if the vector sum is sufficiently large, as in the bottom plot of the Fig. 6, a part of the sum will have to be replaced with $S_f$ words of $i$, $c$ and $\varepsilon$ (see Section 4). One can verify that $S_f < 2n$ whenever $n \geq 10$, i.e. if the size of an addend processed by a thread is greater or equal to 80 bytes, the entirety of the corresponding vectors $i$, $c$ and $\varepsilon$, for all $k$, fit in cache whenever $n$ fits in the cache, which in turn means that evaluation of $\varepsilon$ does not have to involve interactions with the global memory. However, one has to take into account thread contention for accessing cache on all three levels when HyperThreading is enabled, because threads of a single CPU core share the cache. In this example for simplicity the assumption is made that no such contention takes place. To take into account cache contention, one can employ the model of L2 cache described in [29].

Also, the reduction of the parallel algorithm (cf. Fig. 2) requires a barrier synchronization at every level $k$ which is usually implemented such that each of $T$ threads blocks at a mutex, and one of the $T$ threads signals a monitor associated with the mutex. Thus probability of a thread blocking at the barrier is taken as 1.

After a thread has evaluated $\varepsilon$, it reads the value of formerly obtained vector sum, partially from the global memory and partially from cache, adjusts it with respect to $\varepsilon$ and finally writes the result into the global memory with probability $p_{\text{cont, } W}$ of contention for quad-channel memory. Again, $p_{\text{cont, } W}$ is distributed binomially based on the probability $p_W$ of a thread writing the result at a random point in time.

Given that, the speedup can be estimated using the following derivation from (11) in [26] (considering time taken by the adder run in a single thread as 1):

$$
\sigma = \left( f_{\text{bar}} + f_{R} p_{\text{cont, } R} + f_{W} p_{\text{cont, } W} \right) + \left( f_{R} + f_{W} \right) \frac{(1 - p_{R} p_{\text{cont, } R})(1 - p_{W} p_{\text{cont, } W})}{T} + \frac{1 - f_{\text{bar}} - f_{R} - f_{W}}{T},
$$

where $f_{R}$, $f_{\text{bar}}$ and $f_{W}$ are respectively fractions of time spent by a thread reading the addends from memory, at the barrier and writing the result into global memory.

This shows that the speedup and efficiency of parallelization significantly depends on the ability of the underlying platform to perform data transactions in parallel with minimal contention as demonstrated in Fig. 6.

Nonetheless, there is a steady performance gain when using the parallel adder compared to performance of the sequential adder and, likewise, when using parallel addition with AVX-512 vectorization compared to all other adders.

As for vectorization on one CPU, the increase in performance is visible until the impact of interactions with memory overtakes performance gains.

To compare performance provided by the AVX-512 implementation to one of the sequential adder, the relative measurement $1 - t_{\text{AVX-512}}^{-1} - t_{\text{ADC}}^{-1}$ (where $t_{\text{ADC}}$ is time taken by the ADC-based adder, and $t_{\text{AVX-512}}$ is time of the AVX-512 based adder) is used in Fig. 7. The clock frequency is fixed at 1.5 GHz. The peaks correlate with cache sizes (level 1 data cache is 1.125 MiB in size, level 2 cache size is 18 MiB and level 3 size is 24.75 MiB) of the CPU.

Additionally, an implementation of the parallel algorithm (Fig. 2) was developed for CUDA devices as follows. First, input data is separated onto subvectors to fit available global memory on a device. In order to determine size of each subvector, one can employ (12) in which $G$ is available device global memory, in words, $N$ is $2^{64}$ (assuming $n$ is a 64-bit integer), $W$ is $2^{32}$ (a device word is considered 32-bit in size) which yields the limit $n \leq (31G - 1730)/96$ for $n$ per device. The word size of the device ($log_2 W$) is chosen to be 32-bit because, first, a word size of CUDA shared memory is 32-bit, second, because the number of shared memory banks is 32 (or 16 for devices of compute capability 1.x) and, third, because CUDA warp size is 32 threads. Therefore, a 32-bit word allows employment of fast shared memory, individual for each CUDA streaming multiprocessor, to efficiently evaluate $i$, $c$ and $\varepsilon$ in (4), as described below, avoiding bank conflicts.
as opposed to the SSE4.2 implementation (in terms of Fig. 2). Since each of the bits of a block of 1024 threads per block completes in (3). On the other hand, if there are

\[ n = \frac{W}{32} \]

words. Provided that the capacity of the shared memory is sufficient to hold both addends, the sum as well as the values of \( S \) in (3) are recursively computed upon elements of \( i_1 \) and \( c_1 \) located within the shared memory, rather than global memory, until 32 words of 32-bit size are obtained. The latter are written to the global memory. The CUDA kernel stops after having completed the evaluation of the subvector of \( V \oplus U \) composed of 1024 word-sized elements per block as well as 32 words per block per each of \( i_1 \) and \( c_1 \).

Consequently, the host performs parallel evaluation of \( e \) recursively as follows (cf. Fig. 2). Given \( k \) such that \( 1 \leq k < maxk \), the host first invokes CUDA kernel to perform the carryless evaluation of

\[
\varepsilon_k = \gamma_W^{[n/32]} (i_k) \gamma_W^{[n/32]} (2c_k + \zeta_k),
\]

as described in the Section 4, as well as bits of \( i_{k+1} \) and \( c_{k+1} \). The latter two are evaluated in the process involving the shared memory and the parallel bitwise-OR-based reduction described above. After the kernel is complete, the host recursively invokes kernels to compute \( \varepsilon_{k+1} \) and adjust the value of \( \varepsilon_k \) using (3) and (4) to obtain

\[
\varepsilon_k = (i_k + (2c_k + \zeta_k)) \oplus i_k
\]

In this process, given the global (grid-based) index \( t \) of a CUDA thread, the value for \( \varepsilon_k \) is either the input bit \( \zeta \) (if \( k = 1 \) and \( t = 0 \), i.e., the thread 0 is processing the first words of \( i_1 \) and \( c_1 \) resulting from operations upon the first respective words of \( V \) and \( U \) or zero (if \( k > 1 \) and \( t = 0 \)) or (otherwise) the value of the most-significant bit of the lower-order word (concurrently read by the thread \( t - 1 \)) of \( c_k \). The map

\[
\gamma_W^{[n/32]} (i_k) \gamma_W^{[n/32]} (2c_k + \zeta_k) \oplus i_k
\]

is implemented as a test of the bit \( t \bmod 32 \) of the word \( [t/32] \) of \( \varepsilon_{k+1} \) resided in the global memory of CUDA device. The value \( k = maxk \) represents the recursion base case, in which the computation of \( \varepsilon_{maxk} \) only involves operations upon word-sized \( \hat{i}_{max} \) and \( \hat{c}_{max} \). Finally, the computation upon the subvectors of \( V \) and \( U \) completes when \( \varepsilon \) has been evaluated, and the value of \( V \oplus U \) has been adjusted with respect to (3) to compute the result.

If memory capacity of an employed CUDA device is sufficient to hold both addends, the sum as well as the values of \( i_k, c_k \) and \( e \) for each \( k, 1 \leq k \leq maxk \), then one invocation of a CUDA kernel which implements the adder is enough to obtain the final sum \( S \) in (3). On the other hand, if there are multiple subvectors, their addition, there has to be a loop which processes the corresponding subvectors on every iteration. This loop carries dependency on respective carry bits, and on every iteration it is required to exchange data with host which significantly impacts the performance (Fig. 9). However, given multiple CUDA devices, the theorem and the parallel algorithm (Fig. 2) can be used again for parallelization. The efficiency of the approach has not been
Addition of full-word long integers in parallel, and especially using vectorization, is widely believed to be impractical due to challenging carry propagation which establishes data interdependence between digits of addends and the sum. On the one hand, it can be true because parallelization and vectorization can be applied only partially at the cost of a decrease of the algorithm simplicity. Nonetheless, there are various applications, e.g., in high-precision scientific computing, digital signal processing and cryptography, where addition can become performance-critical. In this case complexity of the implementation can become acceptable, or it can be mitigated as shown in [12], [15] and [17].

However, the existing sources focus on hardware implementation of adders and employ bit-level parallelism requiring specialized hardware to perform addition with low latency. The current paper on the other hand focuses on using unspecialized scalar and vector processors with a minimal set of common arithmetic instructions to achieve similar performance gains. Hence, the proposed algorithms and their formal analysis. Particularly, if the word length is one bit, i.e., \( W = 2 \), one can easily derive bit-level carry-look-ahead full adders (described, for instance, in [14]) from the statement of the theorem 1. The described parallel algorithm shown in Fig. 2 corresponds to one presented in [11] and [17].

Experiments conducted to measure performance of the results agree with the prediction (10) in the paper. Notably, AVX-512 with its hardware implementation of \( y_W^0 \circ y_W^n \) via masked operations helps to increase performance of addition significantly, especially when the addition is implemented on a manycore CPU. The parallel implementations without use of vectorization, although not very scalable, provide significant performance boost compared to sequential scalar addition implementing ripple-carry algorithm, even when optimization techniques such as loop unrolling are enforced upon the latter.

Performance gains also take place when addition is performed on a CUDA device as long as transfers of parameters with the host are not involved.

Surprisingly, all of the implementations significantly outperformed addition implemented in the GNU Multiple Precision library (version 6.2.0) even if import and export of the operands and the result are not taken into account. Brief analysis of the GMP source code as well as its disassembly showed that much time is taken for restructuring of data (with reallocations) and performing addition in several different ripple-carry loops.

Compared to adders described in literature, the proposed adders are agnostic to sizes of addends, easily scale with respect to problem size and full-word data representation and provide noticeably better performance if parallelism and/or vectorization are employed.
REFERENCES

[1] C. Chang, S. Yao, and D. Yu, “Vectorized big integer operations for cryptosystems on the Intel MIC architecture,” in Proc. IEEE 22nd Int. Conf. High Perform. Comput., 2015, pp. 194–203.

[2] V. Aulvina and R. Kabilan, “Low power and area efficient borrow save adder design,” in Proc. Int. Conf. Smart Syst. Inventive Technol., 2018, pp. 339–342.

[3] C.-P. Jeannerod, P. Kornerup, N. Louvet, and J.-M. Muller, “Error bounds on complex floating-point multiplication with an FMA,” Math. Comput., vol. 86, pp. 881–898, 2017.

[4] J. Grootendorst, R. M. Avanzi, E. Savaš, and S. Tillich, “Energy-efficient software implementation of floating point modular arithmetic,” in Proc. Int. Workshop Cryptographic Hardware Embedded Syst., 2005, pp. 75–90.

[5] L. Hars, “Fast truncated multiplication for cryptographic applications,” in Proc. Int. Workshop Cryptographic Hardware Embedded Syst., 2005, pp. 211–225.

[6] K. Millar, M. Lukowiaik, and S. Radziszowski, “Design of a Flexible Schönhage-Strassen FFT polynomial multiplier with high-level synthesis to accelerate HE in the cloud,” in Proc. Int. Conf. Reconfigurable Comput. FPGAs, 2019, pp. 1–5.

[7] M. Fürrer, “Faster integer multiplication,” in Proc. 39th Annu. ACM Symp. Theory Comput., 2007, pp. 57–66.

[8] R. Chandran and B. Fagin, “Discrete weighted transforms and large integer arithmetic,” Math. Comput., vol. 62, no. 205, pp. 305–324, Jan. 1994.

[9] P. A. V. Mohan, Residue Number Systems. Cham, Switzerland: Birkhäuser, 2016, pp. 16–24.

[10] V. Avasthi and K. Raj, “Hybrid signed digit arithmetic in efficient computing: A comparative approach to performance assay,” Novel Perspectives Eng. Res., vol. 2, pp. 47–58, 2021. [Online]. Available: https://stm.bookpi.org/NPER-V2/article/view/4372

[11] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations,” IEEE Trans. Comput., vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[12] Brent and Kung, “A regular layout for parallel adders,” IEEE Trans. Comput., vol. C-31, no. 3, pp. 260–264, Mar. 1982.

[13] G. Paci, P. Marchal, and L. Benini, “Exploration of low power adders for a SIMD data path,” in Proc. Asia South Pacific Des. Autom. Conf., 2007, pp. 914–919.

[14] S. Srikanth, I. T. Banu, G. V. Priya, and G. Usha, “Low power array multiplier using modified full adder,” in Proc. IEEE Int. Conf. Eng. Technol., 2016, pp. 1041–1044.

[15] S. Kumar and B. P. Konduru, “High-speed, low area and energy efficient 32bit carry skip adder using verilog HDL,” Proc. Int. Conf. Emerg. Trends Eng. Sci. Technol., vol. 4, pp. 205–209, 2017.

[16] F. Jalarzadephour, A. S. Molahosseini, A. Emrany, and L. Sousa, “A new energy-efficient hybrid wide-operand adder architecture,” IET Circuits Devices Syst., vol. 13, no. 8, pp. 1221–1231, Nov. 2019.

[17] M. J. Ahlham and M. M. Parvez, “Fast performance of parallel adders using VLSI,” Int. J. Eng. Res. Technol., vol. 3, no. 10, pp. 356–360, 2014.

[18] S. Sarabdeep and K. M. Dilip, “Design of area and power efficient modified carry select adder,” Int. J. Comput. Appl., vol. 33, pp. 14–18, 2011.

[19] B. Mohanty and S. Patel, “Area-delay-power efficient carry-select adder,” IEEE Trans. Circuits Syst., II, Exp. Briefs, vol. 61, no. 6, pp. 418–422, Jun. 2014.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.