Outperforming Sequential Full-Word Long Addition with Parallelization and Vectorization

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

10-01-2022 / 13-01-2022

CITATION

Chusov, Andrey (2022): Outperforming Sequential Full-Word Long Addition with Parallelization and Vectorization. TechRxiv. Preprint. <https://doi.org/10.36227/techrxiv.18129191.v1>

DOI

[10.36227/techrxiv.18129191.v1](https://doi.org/10.36227/techrxiv.18129191.v1)

Outperforming Sequential Full-Word Long Addition with Parallelization and Vectorization

Andrey Chusov

Abstract—The paper 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 Kogge-Stone method. Essentially, the paper 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.

Index Terms—Long arithmetic, full-word arithmetic, carry-lookahead adders, SIMD, SMP, CUDA, GMP, parallel arithmetic, vectorization, AVX-512, mask registers, high-performance computing.

1 INTRODUCTION

USE of big-integral arithmetic is required in many fields of technology and science such as cryptography, digital signal processing and high-precision scientific computations. Additive operations provide a foundation for most of the operations of big-int arithmetic. Yet addition (and subtraction) is known to be poorly parallelizable due to data dependencies which occur during correct carry propagation between digits of big integers. 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 [1]. 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 [2] and [3] 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 [4], [5], [6].

The disadvantage of carry-lookahead methods is the complexity of their implementation. For instance, in [7] the authors combine the two approaches, i.e. Kogger-Stone [2] and Brent-Kung [3], 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 [6], in which case the performance gain may take place.

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 [1] and borrow-save [8] 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 implemen-

tation of adders and employ bit-level parallelism in order to improve performance and energy-efficiency of addition. This paper on the other hand provides ways to implement parallel carry-lookahead addition of 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.

Throughout the paper the following notations are used. Vectors which represent long integers of n words are in the group $\langle Z_W, + \rangle$ which is isomorphic to $\langle Z_W, + \rangle$. Particularly, each scalar v_i ($0 \leq i < n$) can be either an unsigned integer from $\langle Z_W, + \rangle$ 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 $\langle Z_{2^n}, \oplus \rangle$ (i.e. addition homomorphic to one of polynomials in $\{p \in Z_2[x] \mid \deg(p) < n\}$) and a vector addition $\boxplus$ of n -length vectors in Z_W^n (with respective scalars added up modulo W) to form an additive group $\langle Z_W^n, \boxplus \rangle$. The latter is isomorphic to a group of polynomials in $\langle Z_W[x], \boxplus \rangle$ 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. VPCMPUD and CMHS) 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

Fig. 1. Detection of carry flag during addition of signed integers in two's complement notation

a pair of n -element vectors $V = (v_0 \dots v_{n-1})^T$ and $U = (u_0 \dots u_{n-1})^T$ as well as a function $\text{CMPGT}(V, U) = ((v_0 > u_0) \dots (v_{n-1} > u_{n-1}))^T$ which maps $(Z_W^n)^2 \rightarrow Z_2^n$, 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. Secondly, 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 $((0 > x) \wedge (0 > y)) \vee ((0 > x) \wedge (x + y \geq 0)) \vee ((0 > y) \wedge (x + y \geq 0)) \equiv (0 > x \wedge 0 > y) \vee \neg(0 > x + y) \wedge ((0 > x) \vee (0 > y))$. Similarly to the unsigned case, 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 CMPGT (although for signed integers),

$$C(V, U) = (c_1 \text{ VAND } c_2) \text{ VOR } (c_s \text{ VANDN } (c_1 \text{ VOR } c_2)), \quad (1)$$

where $c_1 = \text{CMPGT}(O, V)$, $c_2 = \text{CMPGT}(O, U)$, $c_s = \text{CMPGT}(O, V + U)$, 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 $\lceil n/b \rceil$ 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 $\lceil n/b \rceil = 1$, which is the case for many vector processors. Otherwise, e.g. in parallel implementation of addition, the Kogge-Stone method [2] 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 [9]), 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 $I = (W-1 \dots W-1)^T$ using the approach similar to that used above can be applied here to obtain a mask

$$I(V, U) = \text{CMPEQ}(V + U, I), \quad (2)$$

where $\forall V, U \in \mathbb{Z}_W^n : \text{CMPEQ}(V, U) = ((v_0 = u_0) \dots (v_{n-1} = u_{n-1}))^T \in \mathbb{Z}_2^n$.

In order to formally associate scalar values with their vector counterparts we introduce the bijection $\nu_m^n : \mathbb{Z}_m^n \rightarrow \mathbb{Z}_{m^n}$ (together with its inverse ν_m^{-n}) between the respective naturally isomorphic sets as well as the injection $\gamma_m^n : \mathbb{Z}_2^n \rightarrow \mathbb{Z}_m^n$.

Theorem 1. Given two vector addends $V, U \in \mathbb{Z}_W^n \subset \mathbb{Z}_W^{n+1}$ representing respectively the unsigned integers

$\nu_W^n(V) = \sum_{k=0}^{n-1} v_k W^k$ and $\nu_W^n(U) = \sum_{k=0}^{n-1} u_k W^k$ as well as a bit $\zeta \in \mathbb{Z}_2$, the sum $S = \sum_{k=0}^n s_k W^k$ of the two integers and ζ equals the vector sum

$$\begin{aligned} S &= \nu_W^n(V) + \nu_W^n(U) + \zeta \\ &= \nu_W^{n+1} \left(V \boxplus U \boxplus (\gamma_W^{n+1} \circ \nu_2^{-(n+1)})(\varepsilon) \right), \end{aligned} \quad (3)$$

where

$$\varepsilon = (i + 2c + \zeta) \oplus i \in \mathbb{Z}_{2^{n+1}}, \quad (4)$$

$i = \nu_2^n(I(V, U))$, and $c = \nu_2^n(C(V, U))$.

Proof: First, let $n = 1$. Then, to take into account a possible carry flag, consider both $\nu_W^1(V) = v_0$ and $\nu_W^1(U) = u_0$ to be in the sets $\mathbb{Z}_W \times \{0\} \subset \mathbb{Z}_W \times \mathbb{Z}_2$. Then adding up $\nu_W^1(V)$ and $\nu_W^1(U)$ produces a sum

$$\begin{aligned} S' &= \nu_W^1(V) + \nu_W^1(U) = v_0 + u_0 \\ &= (v_0 + u_0) \bmod W + \left\lfloor \frac{v_0 + u_0}{W} \right\rfloor W \\ &= \nu_W^2 \left(\left((v_0 + u_0) \bmod W \right) \right) = \nu_W^2 \left(V \boxplus U \boxplus \begin{pmatrix} 0 \\ c \end{pmatrix} \right) \end{aligned}$$

in which the vector $\begin{pmatrix} 0 & c \end{pmatrix}^T = (\gamma_W^2 \circ \nu_2^{-2}) (2\nu_W^1(C(V, U)))$. Then adding ζ to the sum yields

$$S = S' + \zeta = \nu_W^2 \left(V \boxplus U \boxplus \begin{pmatrix} 0 \\ c \end{pmatrix} \boxplus \begin{pmatrix} \zeta \\ i' \end{pmatrix} \right) \quad (5)$$

where i' is a carry bit resulting from adding ζ to the first element $s'_0 \in \mathbb{Z}_W$ of S' and equals $\zeta \wedge (s'_0 = W-1) \equiv \zeta \wedge i$. Here $c + i' \in \mathbb{Z}_2$ because either $c = 1$, i.e. $s'_0 < v_0$ and $s'_0 < u_0$, or $i' = 1$, i.e. $\zeta \wedge (s'_0 = W-1)$, because there are no values in $\mathbb{Z}_W$ that are greater than $s'_0 = W-1$, i.e. $c \wedge i' = c \wedge i = 0$. Therefore,

$$\begin{pmatrix} 0 \\ c \end{pmatrix} \boxplus \begin{pmatrix} \zeta \\ i' \end{pmatrix} = \begin{pmatrix} \zeta \\ c \oplus i' \end{pmatrix} = \begin{pmatrix} \zeta \\ c \oplus (\zeta \wedge i) \end{pmatrix}. \quad (6)$$

Fixing given values of s'_0 and W (and thus $i = (s'_0 = W-1)$) consider the additive group G_1 with the set

$$\{t \mid (\exists \zeta, c \in \mathbb{Z}_2) [\neg(i \wedge c) \wedge (t = (i + 2c + \zeta) \oplus i)]\} \subseteq \mathbb{Z}_{2^2}$$

along with bitwise XOR as a group operation. Note that

$$\begin{aligned} \forall \zeta, c, i \in \mathbb{Z}_2 \wedge \neg(i \wedge c) : \\ (i + 2c + \zeta) \oplus i &= \left\lfloor \frac{(i + 2c + \zeta) \oplus i}{2} \right\rfloor \cdot 2 + ((i + 2c + \zeta) \oplus i) \bmod 2 = \\ &= \left\lfloor \frac{i + 2c + \zeta}{2} \right\rfloor \cdot 2 + ((\zeta \oplus i) \oplus i) = (c \oplus \zeta \wedge i) \cdot 2 + \zeta. \end{aligned}$$

Therefore, G_1 is closed under bitwise XOR:

$$\begin{aligned} ((i + 2c + \zeta) \oplus i) \oplus ((i + 2c + \zeta) \oplus i) &= \\ ((c_1 \oplus \zeta_1 \wedge i) \cdot 2 + \zeta_1) \oplus ((c_2 \oplus \zeta_2 \wedge i) \cdot 2 + \zeta_2) &= \\ ((c_1 \oplus c_2) \oplus (\zeta_1 \oplus \zeta_2) \wedge i) \cdot 2 + (\zeta_1 \oplus \zeta_2) &= \\ (i + 2(c_1 \oplus c_2) + (\zeta_1 \oplus \zeta_2)) \oplus i, \end{aligned}$$

where $(\neg(i \wedge c_1) \wedge \neg(i \wedge c_2)) \implies \neg(i \wedge (c_1 \oplus c_2))$, and thus G_1 is isomorphic to a group G_2 of linear polynomials in $\mathbb{Z}_2[x]$ with the set

$$\{t \mid (\exists \zeta, c \in \mathbb{Z}_2) [\neg(i \wedge c) \wedge (t = (c \oplus \zeta \wedge i)x + \zeta)]\} \subseteq \mathbb{Z}_2[x]$$

and a map $G_1 \rightarrow G_2 : a \cdot 2 + b \mapsto ax + b$ which is a bijective homomorphism because $(a_1x + b_1) + (a_2x + b_2) =$

$(a_1 \oplus a_2)x + (b_1 \oplus b_2) \mapsto 2(a_1 \oplus a_2) + (b_1 \oplus b_2)$, with zeroes of both groups only possible when c and ζ are both zero because $(c \oplus \zeta \wedge i)x + \zeta = 0 \iff (\zeta = 0) \wedge ((c \oplus (\zeta \wedge i)) = 0) \iff (\zeta = 0) \wedge ((c \oplus 0) = 0) \iff (\zeta = 0) \wedge (c = 0)$ and $(2c + \zeta + i) \oplus i = 0 \iff 2c + 2(i \wedge \zeta) + \zeta = 0 \iff \zeta = 0 \wedge c = 0$.

Therefore, applying the bijection $\nu_2^{-2} = \gamma_2^2 \circ \nu_2^{-2}$ to ε in (4) yields (5), and thus (3) holds for $n = 1$.

Then from (5) and (6) it follows that $S \bmod W = (v_0 + u_0 + \zeta) \bmod W = \nu_W^1(V \boxplus U \boxplus \zeta)$, and $\lfloor S/W \rfloor = c \oplus (\zeta \wedge i) = \left\lfloor \frac{i+2c+\zeta}{2} \right\rfloor \in \mathbb{Z}_2$.

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, $\nu_W^{n+1}(V)$ and $\nu_W^{n+1}(U)$, as respectively $v_1W + v_0$ and $u_1W + u_0$ (with $v_0, u_0 \in \mathbb{Z}_W$ and $v_1, u_1 \in \mathbb{Z}_{W^n}$) yields the sum:

$$\begin{aligned} S &= (v_1 + u_1)W + (v_0 + u_0 + \zeta) = \\ &\left(v_1 + u_1 + \left\lfloor \frac{v_0 + u_0 + \zeta}{W} \right\rfloor \right) W + (v_0 + u_0 + \zeta) \bmod W = \\ &\left(v_1 + u_1 + \left\lfloor \frac{i_0 + 2c_0 + \zeta}{2} \right\rfloor \right) W + (v_0 + u_0 + \zeta) \bmod W, \end{aligned}$$

where $i_0 = \nu_2^1(I(v_0, u_0))$, and $c_0 = \nu_2^1(C(v_0, u_0))$.

Note that $\forall v \in \mathbb{Z}_W^n$

$$\nu_W^n(v) \cdot W = \nu_W^{n+1} \begin{pmatrix} 0 \\ v \end{pmatrix},$$

and $\forall \varepsilon \in \mathbb{Z}_{2^n}$

$$\left(\begin{pmatrix} 0 \\ \gamma_w^n \circ \nu_2^{-n}(\varepsilon) \end{pmatrix} \right) = \gamma_W^{n+1} \left(\begin{pmatrix} 0 \\ \nu_2^{-n}(\varepsilon) \end{pmatrix} \right) = (\gamma_W^{n+1} \circ \nu_2^{-(n+1)})(2\varepsilon).$$

Then, based on the induction hypothesis and on the proof for the base case $n = 1$,

$$\begin{aligned} S &= W \cdot \nu_W^{n+1} \left(\nu_W^{-(n+1)}(v_1) \boxplus \nu_W^{-(n+1)}(u_1) \boxplus \right. \\ &\quad \left. (\gamma_W^{n+1} \circ \nu_2^{-(n+1)}) \left(\left(i_1 + 2c_1 + \left\lfloor \frac{i_0 + 2c_0 + \zeta}{2} \right\rfloor \right) \oplus i_1 \right) \right) \\ &\quad + \nu_W^1(\nu_W^{-1}(v_0) \boxplus \nu_W^{-1}(u_0) \boxplus \\ &\quad (\gamma_W^1 \circ \nu_2^{-1})(((i_0 + 2c_0 + \zeta) \bmod 2) \oplus i_0)) \\ &= \nu_W^{n+2} \left(\left(\begin{pmatrix} \nu_W^{-1}(v_0) \\ \nu_W^{-(n+1)}(v_1) \end{pmatrix} \boxplus \begin{pmatrix} \nu_W^{-1}(u_0) \\ \nu_W^{-(n+1)}(u_1) \end{pmatrix} \right) \right. \\ &\quad \left. \boxplus \gamma_W^{n+2} \left(\begin{pmatrix} \nu_2^{-1}(((i_0 + 2c_0 + \zeta) \oplus i_0) \bmod 2) \\ \nu_2^{-(n+1)} \left(\left(i_1 + 2c_1 + \left\lfloor \frac{i_0 + 2c_0 + \zeta}{2} \right\rfloor \right) \oplus i_1 \right) \end{pmatrix} \right) \right) \\ &= \nu_W^{n+2} (V \boxplus U \boxplus \\ &\quad (\gamma_W^{n+2} \circ \nu_2^{-(n+2)}) \left(2 \left(\left(i_1 + 2c_1 + \left\lfloor \frac{i_0 + 2c_0 + \zeta}{2} \right\rfloor \right) \oplus i_1 \right) \right. \\ &\quad \left. + (((i_0 + 2c_0 + \zeta) \oplus i_0) \bmod 2) \right)). \end{aligned} \quad (7)$$

Since $\forall a, b, k \in \mathbb{N}_0 : (a \oplus b) \cdot 2^k = a \cdot 2^k \oplus b \cdot 2^k$, and $\forall a, b \in \mathbb{N}_0, \forall d, c \in \mathbb{Z}_{2^n}, \forall k \geq n : (a \oplus b) \cdot 2^k + (c \oplus d) =$

$(a \cdot 2^k + c) \oplus (b \cdot 2^k + d)$, the last term of (7) can be rearranged:

$$\begin{aligned} &\left(\left(i_1 + 2c_1 + \left\lfloor \frac{i_0 + 2c_0 + \zeta}{2} \right\rfloor \right) \oplus i_1 \right) \cdot 2 \\ &\quad + ((i_0 + 2c_0 + \zeta) \oplus i_0) \bmod 2 = \\ &\left(2i_1 + 2 \cdot 2c_1 + 2 \left\lfloor \frac{i_0 + 2c_0 + \zeta}{2} \right\rfloor + (i_0 + 2c_0 + \zeta) \bmod 2 \right) \\ &\quad \oplus (2i_1 + i_0) = \\ &((2i_1 + i_0) + 2(2c_1 + c_0) + \zeta) \oplus (2i_1 + i_0) = (i + 2c + \zeta) \oplus i. \end{aligned}$$

Substitution of this to (7) yields (3). $\square$

Computation (3) provides a way to parallelize addition as described in the following sections. This can provide performance increase, if, firstly, ν_W^{n+1} is cheap (e.g. it is the case when the map is just a re-representation of the same data and does not, for instance, involve data transfers which can be expensive as is the case with CUDA); and, secondly, if computations of ε and $(\gamma_W^{n+1} \circ \nu_2^{-(n+1)})(\varepsilon)$ 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, $X, Y \in \mathbb{Z}_{W^n}$ with $W > 2$ and $n > 0$, involves parallel carry-less addition $\nu_W^n(X) \boxplus \nu_W^n(Y)$ using T threads of execution in time $O(n/T)$. Consequently, the carry propagation has to be performed using (4). Computation of $\varepsilon \in \mathbb{Z}_B$ (where $B = 2^b$) also requires possibly long addition of the bit masks i and $2c + \zeta$ 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 $2c + \zeta$ also requires no carry propagation because ζ is a one-bit value, and $2c$ is simply a linear bit shift to the left (towards the most significant position) by one, therefore $2c + \zeta = (c \ll 1) \oplus \zeta$. But the computation of the sum $i + (2c + \zeta)$ 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 $\lceil n \log_W 2 \rceil$ sized values of i and $2c + \zeta$. Likewise, the addition of i and $2c + \zeta$ is reduced to the addition of $\lceil n \log_W 2 \rceil^2$. This reduction continues up until the size of addition is reduced to 1 which requires one scalar addition in $\mathbb{Z}_W$, i.e. $\lceil n \log_W 2 \rceil^{\max k} = 1$ for $\max k \in \mathbb{N}_0$. Therefore, the total number of base- W digits to add up is

$$2 \sum_{k=0}^{\max k} \left\lceil \frac{n}{(\log_2 W)^k} \right\rceil = O \left(\frac{n \log_2 W - 1}{\log_2 W - 1} \right) \quad (8)$$

for all k such that $k < \max k$ it follows that $\left\lceil \frac{n}{(\log_2 W)^k} \right\rceil > 1$, that is

$$\max k = \left\lceil \log_{\log_2 W} n \right\rceil = \left\lceil \frac{\log_2 n}{\log_2 \log_2 W} \right\rceil.$$

Since parallel addition (without ε) can be performed in parallel by T threads, and the same parallel addition is performed to addends for each k , total time required

Fig. 2. Addition of X and Y from $\mathbb{Z}_{W^n}$ modulo W^n using (3) and parallel reduction.

to perform the addition with T threads can therefore be estimated as

$$\sum_{k=0}^{\max k} \left\lceil \frac{n}{T(\log_2 W)^k} \right\rceil = O\left(\frac{n \log_2 W - 1}{T(\log_2 W - 1)}\right). \quad (9)$$

because

$$\begin{aligned} \sum_{k=0}^{\max k} \left\lceil \frac{n}{T(\log_2 W)^k} \right\rceil &= O\left(\sum_{k=0}^{\max k} \frac{n}{T(\log_2 W)^k}\right) = \\ &= O\left(\frac{n}{T} \frac{1 - (\log_2 W)^{\max k+1}}{1 - \log_2 W}\right) = \\ &= O\left(\frac{n}{T} \frac{1 - (\log_2 W)^{\lceil \log_{\log_2 W} n \rceil + 1}}{1 - \log_2 W}\right) = \\ &= O\left(\frac{n}{T} \frac{1 - (\log_2 W)^{\log_{\log_2 W} n + 1}}{1 - \log_2 W}\right) = \\ &= O\left(\frac{n}{T} \frac{1 - n \log_2 W}{n(1 - \log_2 W)}\right) = O\left(\frac{n \log_2 W - 1}{T(\log_2 W - 1)}\right). \end{aligned}$$

Note that in order to achieve (9), 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 $\left\lceil \frac{n}{T(\log_2 W)^k} \right\rceil$ 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 $\mathbb{Z}_{W^n}^n$, then $\left\lceil \frac{n}{T(\log_2 W)^k} \right\rceil$ should be a multiple of $\lceil \log_2 W \rceil$ 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 $\varepsilon \bmod 2^n$) can be estimated with the upper bound:

$$\begin{aligned} 3 \sum_{k=1}^{\max k} \left\lceil \frac{n}{(\log_2 W)^k} \right\rceil &= 3 \sum_{k=1}^{\max k} \left\lceil \frac{n + (\log_2 W)^k - 1}{(\log_2 W)^k} \right\rceil \\ &= 3 \left(\max k + \sum_{k=1}^{\max k} \left\lceil \frac{n-1}{(\log_2 W)^k} \right\rceil \right) \\ &\leq 3 \left(\max k + \sum_{k=1}^{\max k} \frac{n-1}{(\log_2 W)^k} \right) \\ &= 3 \left(\max k + \frac{n-1}{\log_2 W} \frac{(\log_2 W)^{\max k} - 1}{\log_2 W - 1} \right) \\ &\leq 3 \left(\max k + \left\lceil \frac{n-1}{\log_2 W} \frac{(\log_2 W)^{\max k} - 1}{\log_2 W - 1} \right\rceil \right) \\ &= 3 \left(\max k + \left\lceil 1 + \frac{((\log_2 W)^{\max k} - 1)(n-1) - 1}{1 - \log_2 W} \right\rceil \right) \\ &= 3(\max k + \left\lceil 1 + \frac{((\log_2 W)^{\max k} - 1)(n-1) - (\log_2 W)^{\max k}}{(\log_2 W)^{\max k+1} - (\log_2 W)^{\max k}} \right\rceil) \\ &\leq 3 \left(\max k + \left\lceil 1 + \frac{n-2}{\log_2 W - 1} - \frac{n-1}{n(\log_2 W - 1)} \right\rceil \right) \\ &= 3 \left(\max k + \left\lceil 1 + \frac{n^2 - 3n + 1}{n(\log_2 W - 1)} \right\rceil \right). \end{aligned}$$

5 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. ν_W^n (as well as the inverse ν_W^{-n}) and γ_W^n (or the composition $\gamma_W^n \circ \nu_W^{-n}$). 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 ν_m^n and γ_m^n 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 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 $\mathbb{Z}_{2^{64}}^8$ and mask values to be in $\mathbb{Z}_{2^{16}}$ (to take extra carry bit of (4) into account).

```

start:
xor rax, rax
; zmm2 := {-1, -1, -1, -1, -1, -1, -1, -1},
; i.e. set all bits of zmm2
vpternlogq zmm2, zmm2, zmm2, 0FFh
kxorw k1, k1, k1 ; k1 := 0
loop_start:
; zmm0 := {x0, x1, x2, x3, x4, x5, x6, x7}
vmovdqu64 zmm0, [a_addr + rax]
; zmm1 := {y0, y1, y2, y3, y4, y5, y6, y7}
vmovdqu64 zmm1, [b_addr + rax]
; vector addition: zmm0 := zmm0 [+] zmm1
vpaddq zmm0, zmm0, zmm1
; k0 :=  $v_w^8(\text{CMPGT}(zmm1, zmm0)) = c$ 
vpcmpuq k0, zmm0, zmm1, 1
kaddw k0, k0, k0 ; k0 := k0 + k0
korw k0, k0, k1 ; k0 := k0 +  $\zeta$ 
; k1 :=  $v_w^8(\text{CMPEQ}(zmm0, zmm2)) = i$ 
vpcmpeqq k1, zmm0, zmm2
kaddw k0, k0, k1 ; k0 := k0 + k1
kxorw k1, k0, k1 ; k1 := k0  $\oplus$  k1 =  $\varepsilon$ 
; zmm0 := zmm0 [+]  $(\gamma_2^8 \circ v_2^{-8})(\varepsilon \bmod 2^8)$ 
vpsubq zmm0 {k1}, zmm0, zmm2
; k1 := floor(k1 / 8) = floor( $\varepsilon$  / 8)
kshiftrw k1, k1, 8
; {z0, z1, z2, z3, z4, z5, z6, z7} := zmm0
vmovdqu64 [result + rax], zmm0
add rax, 64
cmp rax, r9
jc loop_start

```

Fig. 3. Vectorized addition of X and Y from $\mathbb{Z}_{W^n}$ modulo W^n based on (3) using AVX-512 vector and mask instructions.

In other cases implementations of the maps $\nu_2^n(I(V, U))$, and $\nu_2^n(C(V, U))$ is rather straightforward using widespread vector comparison instructions (see section 2). Such are, for instance, (V)PCMPEQ and (V)PCMPU on Intel [10, Section 5.2] (Volume 2C) and CMEQ and CMHI on ARM [11] which set bits of elements of a destination register according to how the respective elements of the operands meet the comparison predicate.

On the other hand, the inverse transformation, $(\gamma_W^n \circ \nu_2^{-n})$, 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 vectors (and $n = 4$) exhibited performance decrease (see Fig.8) compared to a sequential adder.

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 ε of the sum of the most significant elements of the subvectors) and i (which can be obtained from accumulated bitwise AND of $\nu_2^n(I(v_i, u_i))$ of elements v_i and u_i of the subvectors with the result equal to $2^n - 1$, if i should be set), write those bits to respective vectors of words, such that each word is occupied

Fig. 4. Time taken to perform addition of two long values using the parallel algorithm shown in Fig.2 performed using 36 logical cores (dashed line) compared to time taken to perform the same parallel algorithm but such that each thread performs vectorized addition (Fig.3) producing a pair of bits i and c to add-up later recursively using (4) (dot-dash line) as well as times of sequential addition using the ADC instruction (solid line).

by one bit, then use parallel reduction to pack those bits together: $\mathbb{Z}_w^n \rightarrow \mathbb{Z}_2^n \rightarrow \mathbb{Z}_w^{\lceil n / \log_2 W \rceil}$. Elements of the result can then again be combined to obtain ε 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).

6 EXPERIMENTS

Measurements of performance of the algorithms were done on a system with Intel Core i9-10980XE CPU with 18 physical cores and Hyper-Threading enabled, with one NVIDIA Tesla K40m 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.

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 (Fig.2) shows increase in performance of long addition (Fig.4).

However, measurements show little scalability of the parallel algorithm, both with and without vectorization (Fig.5), 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. Nonetheless, there is a steady performance gain when using the parallel adder compared to performance of the

Fig. 5. Speedups achieved using the parallel adder without (solid line) and with (dashed line) AVX-512 vectorization on a machine with 18 physical (36 logical) CPU cores.

Fig. 6. Time taken to perform addition of two long values, up to 8MiB of random data per addend, using AVX-512 implementation of the vector algorithm in Fig.3 (dashed line) compared to times of sequential addition using ADC instruction (solid line).

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, which happens at about 8MiB of data size per each addend (Fig.6).

Additionally, CUDA implementation of the parallel addition yields performance boost shown in Fig.7 which does not take into account time for data transfer via the PCI-E bus. However, as soon as data transmissions via the bus begin to take place, e.g. when the CUDA device does not have enough memory to hold all data at once (i.e. both addends, the sum as well as sets of bits i , c and ε in (4) during each iteration of the parallel algorithm (Fig.2), the performance of addition quickly drops.

Also, every implementation of long addition presented

Fig. 7. Measured performance of CUDA implementation (dashed line) of addition using NVIDIA Tesla K40m compared to sequential addition on CPU using the ADC instruction (solid line). CUDA times do not take into account time taken to exchange data between host and the device.

Fig. 8. Performance of long addition with GNU Multiple Precision arithmetic library (GMP) [12] compared to other implementations. Notice ADC and SMP performance to see the relationship with the other results.

in the paper, as well as other used SIMD-based adders), significantly outperform addition using GNU multiprecision library (GMP) version 6.2.0, even if the measurements do not take into account data transfers performed with `mpz_import` and `mpz_export`. This is shown in Fig.8.

7 CONCLUSION

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 an increase of complexity of the algorithm. Nonetheless, there are various applications, e.g. in high-precision

scientific computations, including those based on floating-point arithmetic, 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 [3], [6] and [7].

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-lookahead full adders (described, for instance, in [5]) from the statement of the theorem 1. The described parallel algorithm shown in Fig.2 corresponds to one presented in [2] and [7].

Experiments conducted to measure performance of the results agree with the prediction (9) in the paper. Notably, AVX-512 with its hardware implementation of γ_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.

REFERENCES

- [1] V. Awasthi and K. Raj, "Hybrid signed digit arithmetic in efficient computing: A comparative approach to performance assay," *Novel Perspectives of Engineering Research* Vol. 2, pp. 47–58, 10 2021. [Online]. Available: <https://stm.bookpi.org/NPER-V2/article/view/4372>
- [2] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, 1973.
- [3] Brent and Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982.
- [4] G. Paci, P. Marchal, and L. Benini, "Exploration of low power adders for a simd data path," in *2007 Asia and South Pacific Design Automation Conference*, 2007, pp. 914–919.
- [5] S. Srikanth, I. T. Banu, G. V. Priya, and G. Usha, "Low power array multiplier using modified full adder," in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, 2016, pp. 1041–1044.
- [6] S. Kumar and B. P. Konduru, "High-speed, low area and energy efficient 32bit carry skip adder using verilog hdl," in *International Conference on Emerging Trends in Engineering, Science and Technologies (ICETEST 2017)*, 03 2017.
- [7] M. J. Ahammed and M. M. Parvez, "Fast performance of parallel adders using VLSI," *International Journal of Engineering Research & Technology (IJERT)*, vol. 3, no. 10, pp. 356–360, 2014.
- [8] C. Selsi Aulvina and R. Kabilan, "Low power and area efficient borrow save adder design," in *2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2018, pp. 339–342.
- [9] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. [Online]. Available: <http://portal.acm.org/citation.cfm?id=270146>
- [10] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2021. [Online]. Available: <https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>
- [11] Arm Limited, *Arm® Architecture Reference Manual Armv8, for A-profile architecture*, ARM Limited, 2021. [Online]. Available: <https://developer.arm.com/documentation/ddi0487/gb/>
- [12] Free Software Foundation. (2020) Integer arithmetic (GNU MP 6.2.1). [Online]. Available: <https://gmplib.org/manual/Integer-Arithmetic>

Andrey Chusov graduated from Far-Eastern Federal University in Vladivostok, Russia in 2008. In 2012 he received the Ph.D. degree having defended the thesis on the subject of high-performance scalable simulation of acoustical fields. The research interests are in applications of high-performance computing for scientific research, including numerical approximations with controllable precision in models of physical fields and solutions to wave equations, scalable parallel algorithms and their implementation on parallel and distributed platforms, SPMD and SIMD parallelism and vectorization applied in physics.

Currently, he is an Associate Professor in the Far-Eastern Federal University teaching on topics of parallel programming and data processing, security in telecommunications, low-level programming and probabilistic models of systems.