Parallel Vectorized Algorithms for Computing Trigonometric Sums Using AVX-512 Extensions

Przemysław Stpiczyński[0000−0001−8661−414X]

Maria Curie–Skłodowska University, Institute of Computer Science
Akademicka 9/519, 20-033 Lublin, Poland
przemyslaw.stpiczynski@umcs.pl

Abstract. The aim of this paper is to show that Goertzel and Reinsch algorithms for computing trigonometric sums can be efficiently vectorized using Intel AVX-512 intrinsics in order to utilize SIMD extensions of modern processors. Numerical experiments show that the new vectorized implementations of the algorithms using only one core achieve very good speedup over their sequential versions. The new algorithms have been parallelized using OpenMP in order to utilize multiple cores. For sufficiently large problem sizes, the parallel implementations of the algorithms achieve reasonable speedup against the vectorized ones.

Keywords: Trigonometric sums · Goertzel and Reinsch algorithms · Vectorization · AVX-512 · Intrinsics · OpenMP

1 Introduction

For given real numbers $b_0, \ldots, b_n$, and x, let us consider the problem of computing trigonometric sums of the following forms

$$C(x) = \sum_{k=0}^{n} b_k \cos kx \quad \text{and} \quad S(x) = \sum_{k=1}^{n} b_k \sin kx, \tag{1}$$

which appear in many numerical applications. For example, finding (1) is the central part of Talbot's algorithm [13, 18, 31] for computing the numerical inverse of the Laplace Transform. The sums are also used to compute individual terms of the Discrete Fourier Transform [32]. It is not recommended to use (1) directly due to large number of arithmetic operations and poor numerical properties [23]. Instead, one can calculate the sums $C(x)$ and $S(x)$ using the Goertzel algorithm [8, 23], which is a special case of Clenshaw's algorithm [4] used for the summation of orthogonal polynomial series [2, 3]. Actually, the main computational parts of both Goertzel's and Clenshaw's algorithms are the same. The Goertzel algorithm has also some other technical applications [29], especially in signal processing [10, 12, 15, 20, 30], thus its efficient implementations is highly desired [6, 19]. Unfortunately, it can be numerically unstable for $|x| \ll 1$ [7]. Then it is better to use the algorithm introduced by Reinsch [23], which is a little bit more complicated but has better numerical properties.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

Although implementations of the Goertzel (Clenshaw) and Reinsch algorithms seem to be simple, their direct parallelization is not possible due to their recursive form. Papers [2,3] showed how to use Clenshaw's algorithm in order to develop a new method for the evaluation of the Chebyshev polynomials of the first kind and how to parallelize it using special properties of the polynomials. Numerical experiments performed on Cray T3D showed that the introduced approach allowed to achieve limited speedup and the efficiency up to 38%. Another approach [25, 26] assumes that the Goertzel and Reinsch algorithms reduce to the problem of solving special narrow-banded systems of linear equations and introduces new *divide and conquer* parallel algorithms for solving such systems. The implementations of these algorithms achieve rather limited speedup on Intel processors: Pentium III, Pentium 4 and Itanium 2. Moreover, the new version of the Reinsch algorithm applied for finding the numerical inverse of the Laplace transform scales very well on Cray X1, but it cannot utilize vector extensions of its processors. Therefore, the approaches to parallelizing these algorithms described in the literature enable the use of parallel processors but do not allow to utilize vector extensions of modern multicore processors, which is crucial for achieving high performance [1, 5, 24, 27, 28, 33].

In this paper we show how to modify and implement the *divide and conquer* parallel Goertzel and Reinsch algorithms [25, 26] in order to develop almost fully vectorized versions of both algorithms. The new implementations utilize advantages of AVX-512 vector extensions [9] and can also be parallelized. The rest of the paper is organized as follows. Section 2 presents the ordinary Goertzel and Reinsch algorithms for computing (1). In Section 3 we briefly recall the *divide and conquer* versions of the algorithms introduced in [25]. Section 4 discusses the details of vectorized and parallel implementations of the algorithms based on AVX-512 intrinsics [9] and OpenMP [14] constructs. Section 5 shows the results of experiments performed on a machine with modern Intel multicore processors that confirm the efficiency of our new implementations. Finally we present some concluding remarks and plans for future studies.

2 Goertzel and Reinsch Algorithms

First let us observe that we can restrict our attention to the case where $x \neq k\pi$. If $x = k\pi$ then $S(x) = 0$ for all x and $\cos kx = \pm 1$, thus $C(x)$ can be computed using a simple summation algorithm. In case of the Goertzel algorithm [8, 23] for finding (1), we need to compute two last entries (namely S_1 and S_2) of the solution of the following linear recurrence system with constant coefficients:

$$S_k = \begin{cases} 0, & k = n+1, n+2 \\ b_k + 2S_{k+1}\cos x - S_{k+2}, & k = n, \ldots, 1 \end{cases} \tag{2}$$

and then we have

$$\begin{aligned} C(x) &= b_0 + S_1 \cos x - S_2 \\ S(x) &= S_1 \sin x. \end{aligned} \tag{3}$$

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

To avoid the influence of rounding errors on the final computed solution, when x is closed to 0 [7], one can use the Reinsch algorithm [23]. In this case we set $S_{n+2} = D_{n+1} = 0$ and if $\cos x > 0$, then we solve the following linear recurrence system

$$\begin{cases} S_{k+1} = D_{k+1} + S_{k+2} \\ D_k = b_k + \beta S_{k+1} + D_{k+1} \end{cases} \tag{4}$$

for $k = n, n-1, \ldots, 0$, where $\beta = -4\sin^2 \frac{x}{2}$. If $\cos x \leq 0$, we solve

$$\begin{cases} S_{k+1} = D_{k+1} - S_{k+2} \\ D_k = b_k + \beta S_{k+1} - D_{k+1} \end{cases} \tag{5}$$

where $\beta = 4\cos^2 \frac{x}{2}$. Finally, we compute

$$S(x) = S_1 \sin x \text{ and } C(x) = D_0 - \frac{\beta}{2} S_1. \tag{6}$$

3 *Divide-and-conquer* approach

Both algorithms presented in the previous section are very simple but it is clear that loops corresponding to (2) and (4–5) have obvious data dependencies, thus compilers cannot vectorize or parallelize them in order to utilize the underlying hardware of modern multicore processors. Now let us consider the *divide and conquer* approach [25] that forms the basis of new efficient implementations of the algorithms that will be discussed in Section 4. In this paper we will use a slightly different notation than that used in [25], but all given properties are equivalent to it.

3.1 Goertzel Algorithm

Let us consider the following approach for parallelizing the Goertzel algorithm. Equation (2) is equivalent to the following system of linear equations

$$\begin{bmatrix} 1 & -c & 1 & & & \\ & 1 & -c & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & & 1 & -c \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}, \tag{7}$$

where $c = 2\cos x$ and $x_i = S_i$, $i = 1, \ldots, n$. For the sake of simplicity let us assume that $n = m \cdot r$, where integers $m, r \geq 2$. Then (7) can be rewritten as the following block system

$$\begin{bmatrix} U & L & & & \\ & U & L & & \\ & & \ddots & \ddots & \\ & & & & L \\ & & & & U \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_r \end{bmatrix}, \tag{8}$$

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

where $\mathbf{x}_j = [x_{(j-1)m+1}, \ldots, x_{jm}]^T \in \mathbb{R}^m$, $\mathbf{b}_j = [b_{(j-1)m+1}, \ldots, b_{jm}]^T \in \mathbb{R}^m$, $U \in \mathbb{R}^{m\times m}$ is of the same form as the matrix of the system (7), and

$$L = \begin{bmatrix} 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & & \vdots \\ & & \ddots & \vdots \\ 1 & 0 & \ddots & \vdots \\ -c & 1 & \cdots & 0 \end{bmatrix} \in \mathbb{R}^{m\times m}. \tag{9}$$

It is obvious that L can be rewritten as $L = \mathbf{e}_{m-1}\mathbf{e}_1^T - c\mathbf{e}_m\mathbf{e}_1^T + \mathbf{e}_m\mathbf{e}_2^T$, where $\mathbf{e}_k$ is k-th unit vector from $\mathbb{R}^m$. The solution to (7) can be expressed as

$$\begin{cases} \mathbf{x}_r = U^{-1}\mathbf{b}_r \\ \mathbf{x}_j = U^{-1}\mathbf{b}_j - U^{-1}L\mathbf{x}_{j+1}, \quad j = r-1, \ldots, 1, \end{cases} \tag{10}$$

Assuming that $U\mathbf{z}_j = \mathbf{b}_j$, $j = 1, \ldots, r$, and $\mathbf{y}_{m-1} = U^{-1}\mathbf{e}_{m-1}$, $\mathbf{y}_m = U^{-1}\mathbf{e}_m$, we get

$$\mathbf{x}_j = \mathbf{z}_j - x_{jm+1}\mathbf{y}_{m-1} + (cx_{jm+1} - x_{jm+2})\mathbf{y}_m. \tag{11}$$

It can be proved [25] that for $x \neq k\pi$, $k \in \mathbb{Z}$, entries of $\mathbf{y}_m = [y_1, y_2, \ldots, y_{m-1}, 1]^T$ satisfy

$$y_j = \frac{\sin(m+1-j)x}{\sin x}, \quad j = 1, \ldots, m-1. \tag{12}$$

Moreover, $\mathbf{y}_{m-1} = [y_2, y_3, \ldots, y_{m-1}, 1, 0]^T$, where all y_j also satisfy (12). Let $\mathbf{x}'_j$, $\mathbf{z}'_j$, $j = 1, \ldots, r$, denote first two entries of $\mathbf{x}_j$, $\mathbf{z}_j$, respectively, and

$$M = \begin{bmatrix} -y_1 & cy_1 - y_2 \\ -y_2 & cy_2 - y_3 \end{bmatrix} \in \mathbb{R}^{2\times 2}. \tag{13}$$

Then we get

$$\begin{cases} \mathbf{x}''_r = \mathbf{z}''_r \\ \mathbf{x}''_j = \mathbf{z}''_j + M\mathbf{x}''_{j+1}, \quad j = r-1, \ldots, 1 \end{cases} \quad \text{and} \quad \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} = \mathbf{x}''_1. \tag{14}$$

The presented approach makes it possible to formulate a parallel algorithm. First we should find all vectors $\mathbf{z}_j$ in parallel, solving $U\mathbf{z}_j = \mathbf{f}_j$, and then we use (14) to find S_1, S_2.

3.2 Reinsch Algorithm

Now let us observe that equations (4–5) are equivalent to the following system of linear equations

$$\begin{bmatrix} 1 & -\beta & \delta & & & & \\ & 1 & -1 & \delta & & & \\ & & 1 & -\beta & \delta & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & 1 & -1 & \delta \\ & & & & & 1 & -\beta \\ & & & & & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ x_{2n} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ f_{2n} \end{bmatrix}, \tag{15}$$

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

where

$$\delta = \begin{cases} -1, \cos x > 0 \\ 1, \cos x \le 0 \end{cases}, \quad x_k = \begin{cases} D_{\lfloor k/2 \rfloor}, k = 1, 3, \ldots, 2n-1 \\ S_{k/2}, \quad k = 2, 4, \ldots, 2n \end{cases} \tag{16}$$

and

$$f_k = \begin{cases} b_{\lfloor k/2 \rfloor}, & k = 1, 3, \ldots, 2n-3 \\ 0, & k = 2, 4, \ldots, 2n-2 \\ b_{n-1} - \delta b_n, & k = 2n-1 \\ b_n, & k = 2n. \end{cases} \tag{17}$$

Similarly to the Goertzel algorithm, under the assumption that $n = m \cdot r$, the system (15) can be written in the following block form

$$\begin{bmatrix} U & L & & \\ & U & L & \\ & & \ddots & L \\ & & & U \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_r \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_r \end{bmatrix}, \tag{18}$$

where all $\mathbf{x}_j$, $\mathbf{f}_j \in \mathbb{R}^{2m}$, and $U \in \mathbb{R}^{2m \times 2m}$ is of the same form as the matrix of the system (15), and

$$L = \begin{bmatrix} 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & & \vdots \\ \delta & 0 & \ddots & \vdots \\ -1 & \delta & \cdots & 0 \end{bmatrix} \in \mathbb{R}^{2m \times 2m}. \tag{19}$$

This yields

$$\begin{cases} U\mathbf{x}_r = \mathbf{f}_r \\ U\mathbf{x}_j = \mathbf{f}_j + L\mathbf{x}_{j+1}, \quad j = r-1, \ldots, 1, \end{cases} \tag{20}$$

so

$$\mathbf{x}_j = U^{-1}\mathbf{f}_j - U^{-1}L\mathbf{x}_{j+1} = \mathbf{z}_j - x_{jm+1}\mathbf{y}_1 - x_{jm+2}\mathbf{y}_2, \tag{21}$$

where $\mathbf{y}_1$, $\mathbf{y}_2$ satisfy $U\mathbf{y}_1 = [0, \ldots, 0, \delta, -1]^T$ and $U\mathbf{y}_2 = [0, \ldots, 0, \delta]^T$, respectively. Let us define

$$M = \begin{bmatrix} y_1^{(1)} & y_1^{(2)} \\ y_2^{(1)} & y_2^{(2)} \end{bmatrix} \in \mathbb{R}^{2 \times 2}, \tag{22}$$

where $y_1^{(1)}$, $y_2^{(1)}$, and $y_1^{(2)}$, $y_2^{(2)}$ denote first two entries of $\mathbf{y}_1$, $\mathbf{y}_2$, respectively. It can be proved [25] that the entries of M satisfy

$$\begin{aligned} y_1^{(1)} &= -\cos mx + (\beta/2)y_2^{(1)} \\ y_2^{(1)} &= -\sin mx / \sin x \\ y_1^{(2)} &= \delta \cos mx + \cos(m-1)x + (\beta/2)y_2^{(2)} \\ y_2^{(2)} &= (\delta \sin mx + \sin(m-1)x) / \sin x \end{aligned} \tag{23}$$

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

Finally, we get

$$\begin{cases} \mathbf{x}''_r = \mathbf{z}''_r \\ \mathbf{x}''_j = \mathbf{z}''_j - M\mathbf{x}''_{j+1}, \quad j = r-1, \ldots, 1 \end{cases} \quad \text{and} \quad \begin{bmatrix} D_0 \\ S_1 \end{bmatrix} = \mathbf{x}''_1. \tag{24}$$

Similarly to the Goertzel algorithm, the parallel Reinsch algorithm consists of two parts: the parallel one, where we find all $\mathbf{z}_j$, and the sequential based on (24). It should be noted that increasing the value of r, i.e. potentially more processors operating in parallel, increases the number of operations in the sequential part. In Section 4 we will also show how to omit the assumption that $n = m \cdot r$.

It should be noted that good numerical properties of the parallel version of the Reinsch algorithm were confirmed empirically. The algorithm was used as the main part of the parallel version of Talbot's method for computing the numerical inverse of the Laplace transform [26]. No decrease in accuracy was observed compared to the sequential version of the algorithm.

4 Implementation of the Algorithms

Although the algorithms presented in Section 3 have potential parallelism, they do not explicitly utilize vector extensions of modern processors, like AVX-512 [9], what is crucial to achieve high performance [1,5,24,27,28,33]. Simply, the parallel (i.e. *divide*) parts of both algorithms are still based on recurrence computations.

In order to utilize advantages of AVX-512 vector extensions [9] and develop vectorizable implementations of the considered algorithms we will use intrinsics for SIMD instructions which allow to write constructs that look like C/C++ function calls corresponding to actual AVX-512 instructions. Such calls are automatically replaced with assembly code inlined directly into programs. Moreover, the use of intrinsics is a good choice to make sure that compilers do exactly what we want, especially in the case of complex nested loops that may prevent automatic vectorization.

Now let us consider the way how we can use these algorithms to obtain their efficient implementations that would operate on 512-bit vector registers as basic data structures. The `__m512d` datatype defined in AVX-512 can be used to store eight double precision floating point numbers. When we set $r = 8$ in (8), then the first *divide* part of the Goertzel algorithm is equivalent to the problem of solving the block system of linear equations $UZ = B$, where

$$B = \begin{bmatrix} b_1 & b_{m+1} & \cdots & b_{7m+1} \\ b_2 & b_{m+2} & \cdots & b_{7m+2} \\ \vdots & \vdots & \cdots & \vdots \\ b_m & b_{2m} & \cdots & b_{8m} \end{bmatrix} \in \mathbb{R}^{m \times 8}. \tag{25}$$

Then, the solution Z can be found using the following sequence of vector operations

$$Z_{k,*} \leftarrow B_{k,*} + cZ_{k+1,*} - Z_{k+2,*} \quad \text{for } k = m, \ldots, 1, \tag{26}$$

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

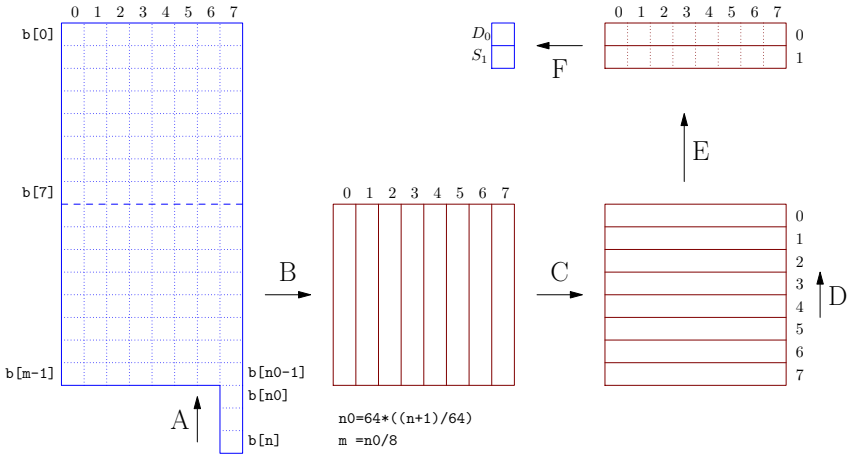

Fig. 1. Stages of the vectorized Reinsch algorithm using AVX-512

where $Z_{m+1,*}$ and $Z_{m+2,*}$ are zero vectors. Moreover, in order to use the *conquer* part of the algorithm (14), we only need to find the vectors $Z_{1,*}$ and $Z_{2,*}$, thus all necessary calculations based on (26) can be carried out using 512-bit vector registers filled with data loaded from memory.

In case of the Reinsch algorithm, we work similarly. We have to find $Z_{1,*}$ and $Z_{2,*}$, i.e. first two rows of the matrix Z defined by $UZ = F$, where

$$F = \begin{bmatrix} b_0 & b_m & \cdots & b_{7m} \\ 0 & 0 & \cdots & 0 \\ b_1 & b_{m+1} & \cdots & b_{7m+1} \\ \vdots & \vdots & \cdots & \vdots \\ b_{m-1} & b_{2m-1} & \cdots & b_{n-1} - \delta b_n \\ 0 & 0 & \cdots & b_n \end{bmatrix} \in \mathbb{R}^{2m\times 8}, \tag{27}$$

using the following sequence of vector operations

$$\begin{cases} Z_{k,*} \leftarrow Z_{k+1,*} - \delta Z_{k+2,*}, & k = 2m, 2m-2, \ldots, 2 \\ Z_{k,*} \leftarrow B_{k,*} + \beta Z_{k+1,*} - \delta Z_{k+2,*}, & k = 2m-1, 2m-3, \ldots, 1. \end{cases} \tag{28}$$

Note that the assumption $n = m{\cdot}r$ can be omitted. Then, for both algorithms, it is necessary to perform a number of sequential steps defined by (2) or (4–5) to initialize last entries of the vectors $Z_{m+1,*}$ and $Z_{m+2,*}$ (Goertzel), or $Z_{2m+1,*}$ and $Z_{2m+2,*}$ (Reinsch), respectively.

Stages of the Reinsch algorithm are shown in Figure 1. The sequence of coefficients $b_0, \ldots, b_n$ stored in the array `b` of double precision numbers can be divided into two parts. The first one can be treated as a collection of $m/64$ square 8×8 blocks. Thus it contains the coefficients $b_0, \ldots, b_{n_0-1}$, $n_0 = 64\cdot\lfloor (n+1)/64 \rfloor$, that should be processed using AVX-512 vector extensions. The second part, i.e.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

the numbers $b_{n_0}, \ldots, b_n$, should be processed first (Stage A) using (4–5). Then each column of the last block (i.e. eight consecutive numbers) is loaded into a vector register using the intrinsic `_mm512_load_pd()` (Stage B). Then (Stage C), such a block stored in eight registers is transposed so that each register contains one row of the block. During Stage D, we use the intrinsic `_mm512_fmadd_pd()` which performs *a fused multiply–add*, and depending on the sign of δ, the intrinsic `_mm512_add_pd()`, or `_mm512_sub_pd()` to process all rows of the block according to (28). Stages B–D are repeated for all blocks (Stage E). Finally (Stage F), we use (24) on the first two rows of the first block in order to get D_0 and S_1.

The vectorized Goertzel algorithm can be implemented similarly but with one difference. The intrinsic `_mm512_load_pd()` loads a 512-bits vector composed of eight packed double-precision floating-point numbers from memory but on condition that the memory address is aligned on a 64-byte boundary or a general-protection exception may be generated. In case of the Goertzel algorithm, the coefficient b_0 is only used in (3). Thus, the vectorized part will operate on blocks containing the coefficients $b_{64}, \ldots, b_{n_0-1}$, and the coefficients $b_1, \ldots, b_{63}$ should be processed sequentially using (2).

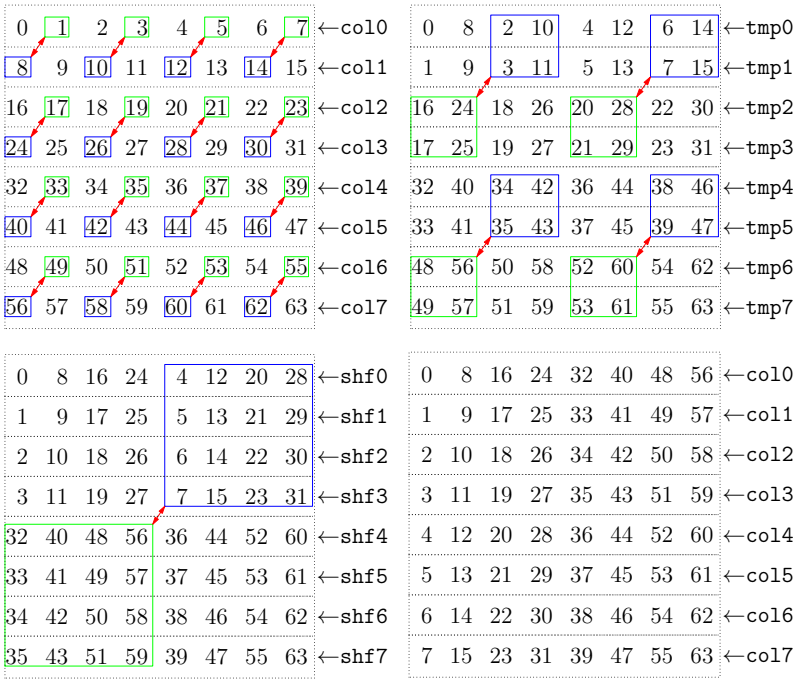

Fig. 2. Three steps and the final result of the transposition of 8×8 block stored in eight `__m512d` registers: red arrows indicate blue and green blocks that should be swapped

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

It should be noticed that Stage C, i.e. transpositions of 8×8 blocks stored columnwise in vector registers, can be performed in three steps using only AVX-512 intrinsics. First, we have to swap single entries between two consecutive vectors (Figure 2, top-left). Then, 2×2 blocks are swapped between pairs of vectors (Figure 2, top-right). Finally, we swap 4×4 blocks stored in first and second halves of quads of vectors (Figure 2, bottom-left) in order to get the result (Figure 2, bottom-right). The details can be found in Figure 3.

```
_inline void vec8x8_transpose(__m512d *col0,__m512d *col1,__m512d *col2,__m512d *col3,
                              __m512d *col4,__m512d *col5,__m512d *col6,__m512d *col7)

{  __m512d tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
   __m512d shf0, shf1, shf2, shf3, shf4, shf5, shf6, shf7;

// Step 1: Unpack and interleave double-precision floating-point elements
//         from the low and high halves of each 128-bit lane of pairs of columns
     tmp0 = _mm512_unpacklo_pd(*col0,*col1);
     tmp1 = _mm512_unpackhi_pd(*col0,*col1);
     tmp2 = _mm512_unpacklo_pd(*col2,*col3);
     tmp3 = _mm512_unpackhi_pd(*col2,*col3);
     tmp4 = _mm512_unpacklo_pd(*col4,*col5);
     tmp5 = _mm512_unpackhi_pd(*col4,*col5);
     tmp6 = _mm512_unpacklo_pd(*col6,*col7);
     tmp7 = _mm512_unpackhi_pd(*col6,*col7);
// Step 2: Shuffle 128-bits (composed of 2 double-precision floating-point elements) selected from pairs,
//         store the results (elements are copied when the corresponding mask bit is not set)
     shf0 = _mm512_mask_shuffle_f64x2(tmp0,0b11001100,tmp2,tmp2,0b10100000);
     shf1 = _mm512_mask_shuffle_f64x2(tmp1,0b11001100,tmp3,tmp3,0b10100000);
     shf2 = _mm512_mask_shuffle_f64x2(tmp2,0b00110011,tmp0,tmp0,0b11110101);
     shf3 = _mm512_mask_shuffle_f64x2(tmp3,0b00110011,tmp1,tmp1,0b11110101);
     shf4 = _mm512_mask_shuffle_f64x2(tmp4,0b11001100,tmp6,tmp6,0b10100000);
     shf5 = _mm512_mask_shuffle_f64x2(tmp5,0b11001100,tmp7,tmp7,0b10100000);
     shf6 = _mm512_mask_shuffle_f64x2(tmp6,0b00110011,tmp4,tmp4,0b11110101);
     shf7 = _mm512_mask_shuffle_f64x2(tmp7,0b00110011,tmp5,tmp5,0b11110101);
// Step 3: Shuffle 128-bits (composed of 2 double-precision floating-point elements)
//         selected from pairs, and store the results
    *col0 = _mm512_shuffle_f64x2(shf0,shf4,0b01000100);
    *col1 = _mm512_shuffle_f64x2(shf1,shf5,0b01000100);
    *col2 = _mm512_shuffle_f64x2(shf2,shf6,0b01000100);
    *col3 = _mm512_shuffle_f64x2(shf3,shf7,0b01000100);
    *col4 = _mm512_shuffle_f64x2(shf0,shf4,0b11101110);
    *col5 = _mm512_shuffle_f64x2(shf1,shf5,0b11101110);
    *col6 = _mm512_shuffle_f64x2(shf2,shf6,0b11101110);
    *col7 = _mm512_shuffle_f64x2(shf3,shf7,0b11101110);
}
```

Fig. 3. The source code corresponding to Figure 2

Both vectorized algorithms can easily be parallelized using OpenMP in order to utilize multiple cores. The idea of the parallel vectorized Reinsch algorithm is shown in Figure 4. First, the sequential part is responsible for processing the tail of coefficients $b_{n_0}, \ldots, b_n$, where $n_0 = 64 \cdot P \cdot \lfloor (n+1)/(64 \cdot P) \rfloor$, and P is the number of threads that will be assigned to cores. Then, each OpenMP thread performs the vectorized Reinsch algorithm on one column of $q = n_0/(64 \cdot P)$ blocks and stores first two rows of its processed top block to shared memory. Finally, again during another sequential part, the numbers D_0, S_1 are evaluated using (24). Note that in this case $r = 8P$. The parallel vectorized Goertzel algorithm works similarly, but during its parallel part we process $q \cdot P$ blocks, where q is the number of blocks processed by each thread. Finally, during another sequential part, we use (14) and then process the coefficients $b_1, \ldots, b_{63}$ using (2) in order to evaluate S_1 and S_2.

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

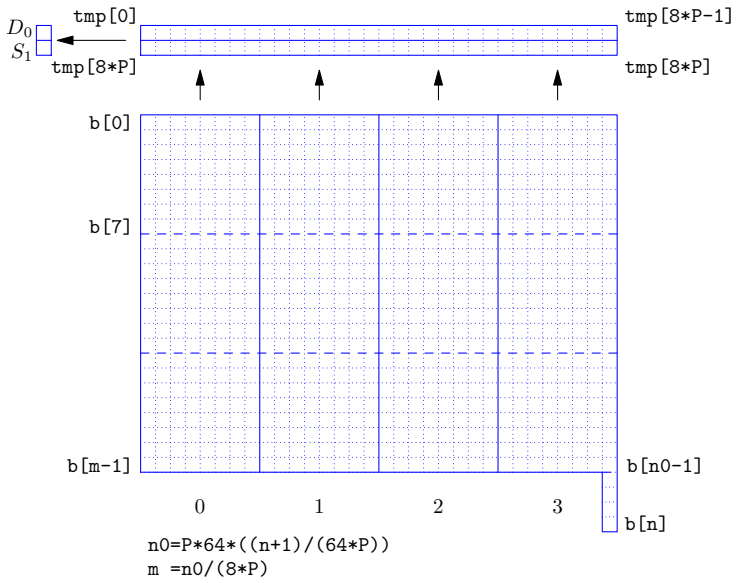

Fig. 4. Stages of the parallel vectorized Reinsch algorithm using OpenMP

It is clear that the use of the vectorized Reinsch algorithm is possible only for $n \geq 64$. The vectorized Goertzel algorithm requires $n \geq 127$, i.e. when vectorization can be applied. Moreover, the use of the parallel implementations is reasonable when a sufficient number of blocks can be processed in parallel.

All source codes of the discussed implementations have been made available in our GitHub repository `https://github.com/pstpicz/trigsums`. It contains sequential, vectorized, and parallel versions of both Goertzel and Reinsch algorithms, as well as test programs that can be used to evaluate performance and accuracy of the algorithms.

5 Results of Experiments

All considered implementations have been tested on a machine with two Intel *Xeon Platinum 8358* processors (totally 64 cores with hyperthreading, 2.6 GHz, 48 MB of cache memory), 256 GB RAM, running under Linux with Intel OneAPI version 2023 containing the C/C++ compiler.

Table 1 shows the execution time of both sequential and vectorized implementations of our versions Goertzel and Reinsch algorithms, as well as the speedup of the vectorized implementations compared to their sequential counterparts, obtained for various problem sizes. It can be observed that both vectorized algorithms' implementations are faster than the sequential ones, even for very small problem sizes, and the speedup increases as the problem size increases, up to 6.4

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

Table 1. Execution time $[s]$ and speedup of sequential and vectorized algorithms

	Goertzel			Reinsch		
n	sequential $[s]$	vectorized $[s]$	speedup	sequential $[s]$	vectorized $[s]$	speedup
$2 \cdot 10^2$	8.8215e-06	2.8610e-06	3.08	1.1921e-05	7.8678e-06	1.52
$2 \cdot 10^3$	1.5020e-05	4.7684e-06	3.15	1.5974e-05	6.9141e-06	2.31
$2 \cdot 10^4$	5.7220e-05	1.0967e-05	5.22	8.0109e-05	1.5974e-05	5.01
$2 \cdot 10^5$	4.8995e-04	8.2970e-05	5.91	7.1883e-04	1.0204e-04	7.04
$2 \cdot 10^6$	5.0120e-03	7.8297e-04	6.40	7.1170e-03	9.5701e-04	7.44
$2 \cdot 10^7$	4.9497e-02	8.2841e-03	5.97	7.1515e-02	9.1901e-03	7.78
$2 \cdot 10^8$	5.0171e-01	9.3292e-02	5.38	7.6957e-01	9.2379e-02	8.33
$2 \cdot 10^9$	5.035e+00	9.5067e-01	5.30	7.882e+00	9.1985e-01	8.57

for Goertzel, and 8.57 for Reinsch. It is worth adding that the sequential implementation of the Goertzel algorithm has also been implemented in the *Boost* library[1] as as a part of the function `chebyshev_clenshaw_recurrence()` and its performance is comparable to the performance of our implementation of this sequential algorithm based on (2–3).

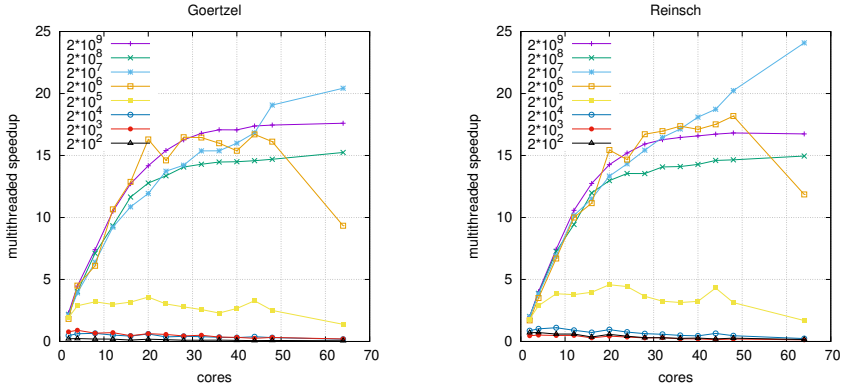

Fig. 5. Speedup of parallel Goertzel and Reinsch algorithms against vectorized ones

Figure 5 illustrates the speedup of the parallel vectorized algorithms over their vectorized counterparts calculated using $s_p(n) = t_v(n)/t_p(n)$, where $t_p(n)$ and $t_v(n)$ denote the execution time of the parallel vectorized and the vectorized algorithms, respectively, and p is the number of cores. Timing results have been collected using various numbers of sockets, cores, and threads per core, specified

[1] `https://www.boost.org/doc/libs/1_83_0/boost/math/special_functions/chebyshev.hpp`

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

using the `KMP_HW_SUBSET` environment variable. We have also used `KMP_AFFINITY` to tell the OpenMP runtime how threads should be assigned to cores. The best results for $P = 2 \cdot X$ threads have been obtained for `KMP_HW_SUBSET=2s,Xc,1t` and `KMP_AFFINITY=compact`, what means that the best choice is to use one thread per core, distributing threads evenly between cores of two processors.

It can be observed that the qualitative behavior of both algorithms is almost the same. The use of multiple cores is not reasonable for smaller problem sizes. Then, most of cores remain idle during the parallel part of the execution. Speedup over the vectorized versions can be observed for $n > 10^5$. However in most cases, we can observe a peak for a number of cores for which the best performance is achieved. When we use more cores, the speedup does not increase or even becomes worse. The best speedup can be observed for $n = 2 \cdot 10^7$. It is approximately 20.4 for Goertzel and 24 for Reinsch, respectively. This means that the largest observed speedups of the parallel vectorized implementations achieved on 64 cores over the sequential implementations of the Goertzel and Reinsch algorithms are 122 and 187, respectively.

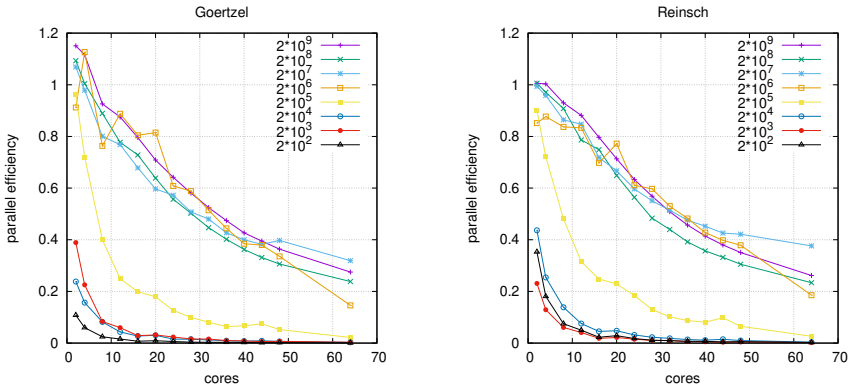

Fig. 6. Parallel efficiency of parallel Goertzel and Reinsch algorithms

Figure 6 shows the parallel efficiency of both parallel implementations calculated as $e_p(n) = s_p(n)/p$. For sufficiently large problem sizes, the parallel efficiency is very good but decreases as the number of cores increases. For the Goertzel algorithm, when a small number of cores is used (i.e. $p = 2, 4$) one can observe $e_p(n) > 1$, what is a fine example of the cache effect [21], when the use of multiple cores speeds up memory transfers over the sequential algorithm [17].

6 Conclusions and Future Works

We have demonstrated that both Goertzel and Reinsch algorithms, examples of recurrence computations, can be efficiently vectorized and parallelized when we

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

apply vectorization and parallelization techniques to the algorithmic approach based on *divide and conquer* methods for solving narrow banded linear systems that arise for linear recurrences. It is clear that such approach can also be applied to develop fast implementations of other problems that reduce to narrow banded systems of equations. It should be noted that although the use of intrinsics significantly limits portability, the places where they are used are crucial for the performance and can easily be ported to other ISA extensions such as scalable vector extensions on ARM [22]. In the future we plan to develop portable implementations of the algorithms in SYCL [16] and study its performance portability on various CPU and GPU platforms [11].

References

1. Amiri, H., Shahbahrami, A.: SIMD programming using intel vector extensions. Journal of Parallel and Distributed Computing **135**, 83–100 (2020). https://doi.org/10.1016/j.jpdc.2019.09.012
2. Barrio, R.: Parallel algorithms to evaluate orthogonal polynomial series. SIAM Journal on Scientific Computing **21**(6), 2225–2239 (2000). https://doi.org/10.1137/S1064827598340494
3. Barrio, R., Sabadell, J.: A parallel algorithm to evaluate Chebyshev series on a message passing environment. SIAM Journal on Scientific Computing **20**, 964–969 (1998). https://doi.org/10.1137/S1064827596312857
4. Clenshaw, C.W.: A note on the summation of Chebyshev series. Mathematical Tables and other Aids to Computation **9**, 118–120 (1955)
5. Dmitruk, B., Stpiczyński, P.: Improving accuracy of summation using parallel vectorized Kahan's and Gill-Møller algorithms. Concurrency and Computation Practice and Experience pp. 1–13 (2023). https://doi.org/10.1002/cpe.7763
6. Dulik, T.: An FPGA implementation of Goertzel algorithm. In: Field-Programmable Logic and Applications, 9th International Workshop, FPL'99, Glasgow, UK, August 30 - September 1, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1673, pp. 339–346. Springer (1999). https://doi.org/10.1007/978-3-540-48302-1_35
7. Gentleman, W.M.: An error analysis of Goertzel's (Watt's) method for computing fourier coefficients. Computer Journal **12**(2), 160–164 (1969). https://doi.org/10.1093/COMJNL/12.2.160
8. Goertzel, G.: An algorithm for the evaluation of finite trigonometric series. The American Mathematical Monthly **65**, 34–35 (1958). https://doi.org/10.2307/2310304
9. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High-Performance Programming. Knights Landing Edition. Morgan Kaufman, Cambridge, MA, USA (2016)
10. Kececioglu, O., Gani, A., Sekkeli, M.: A performance comparison of static VAr compensator based on Goertzel and FFT algorithm and experimental validation. SpringerPlus **5**, 391 (2016). https://doi.org/10.1186/s40064-016-2034-7
11. Marowka, A.: Reformulation of the performance portability metric. Software: Practice and Experience **52**(1), 154–171 (2022). https://doi.org/10.1002/spe.3002
12. Martinez-Roman, J., Puche-Panadero, R., Terron-Santiago, C., Sapena-Bano, A., Burriel-Valencia, J., Pineda-Sanchez, M.: Low-cost diagnosis of rotor asymmetries

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

of induction machines at very low slip with the Goertzel algorithm applied to the rectified current. IEEE Transactions on Instrumentation and Measurement **70**, 1–11 (2021). https://doi.org/10.1109/TIM.2021.3115216
13. Murli, A., Rizzardi, M.: Algorithm 682: Talbot's method for the Laplace inversion problem. ACM Trans. Math. Soft. **16**, 158–168 (1990)
14. van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP – The Next Step. Affinity, Accelerators, Tasking, and SIMD. MIT Press, Cambridge MA (2017)
15. Regnacq, L., Wu, Y., Neshatvar, N., Jiang, D., Demosthenous, A.: A Goertzel filter-based system for fast simultaneous multi-frequency EIS. IEEE Transactions on Circuits and Systems II: Express Briefs **68**, 3133–3137 (2021). https://doi.org/10.1109/TCSII.2021.3092069
16. Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., Tian, X.: Data Parallel C++. Apress Berkeley, CA (2021). https://doi.org/10.1007/978-1-4842-5574-2
17. Ristov, S., Prodan, R., Gusev, M., Skala, K.: Superlinear speedup in HPC systems: why and when? In: Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016. Annals of Computer Science and Information Systems, vol. 8, pp. 889–898. IEEE (2016). https://doi.org/10.15439/2016F498
18. de Rosa, M.A., Giunta, G., Rizzardi, M.: Parallel Talbot's algorithm for distributed memory machines. Parallel Computing **21**, 783–801 (1995)
19. Seshadri, R., Ramakrishnan, S., Kumar, J.: Knowledge-based single-tone digital filter implementation for DSP systems. Personal and Ubiquitous Computing **26**, 319–328 (2022). https://doi.org/10.1007/s00779-019-01304-2
20. Singh, B., Reddy, C.C.: Fast Goertzel algorithm and RLS-adaptive filter based reference current extraction for grid-connected system. In: 2020 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe). pp. 156–160 (2020). https://doi.org/10.1109/ISGT-Europe47291.2020.9248955
21. Speckenmeyer, E., Monien, B., Vornberger, O.: Superlinear speedup for parallel backtracking. In: Supercomputing, 1st International Conference, Athens, Greece, June 8-12, 1987, Proceedings. Lecture Notes in Computer Science, vol. 297, pp. 985–993. Springer (1987). https://doi.org/10.1007/3-540-18991-2_58
22. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Prémillieu, N., Reid, A., Rico, A., Walker, P.: The ARM scalable vector extension. IEEE Micro **37**, 26–39 (2017). https://doi.org/10.1109/MM.2017.35
23. Stoer, J., Bulirsh, R.: Introduction to Numerical Analysis. Springer, New York, 2nd edn. (1993)
24. Stojanov, A., Toskov, I., Rompf, T., Pueschel, M.: SIMD intrinsics on managed language runtimes pp. 2–15 (2018). https://doi.org/10.1145/3168810
25. Stpiczyński, P.: Fast parallel algorithms for computing trigonometric sums. In: 2002 International Conference on Parallel Computing in Electrical Engineering (PARELEC 2002), 22-25 September 2002, Warsaw, Poland. pp. 299–304. IEEE Computer Society (2002). https://doi.org/10.1109/PCEE.2002.1115276
26. Stpiczyński, P.: A note on the numerical inversion of the Laplace transform. In: Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3911, pp. 551–558. Springer (2006). https://doi.org/10.1007/11752578_66

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12

27. Stpiczyński, P.: Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus. The Journal of Supercomputing **74**(4), 1461–1472 (2018). https://doi.org/10.1007/s11227-017-2231-3
28. Stpiczyński, P.: Algorithmic and language-based optimization of Marsa-LFIB4 pseudorandom number generator using OpenMP, OpenACC and CUDA. Journal of Parallel and Distributed Computing **137**, 238–245 (2020). https://doi.org/10.1016/j.jpdc.2019.12.004
29. Sysel, P., Rajmic, P.: Design of high-performance parallelized gene predictors in MATLAB. BMC Res Notes **5**, 183 (2012). https://doi.org/10.1186/1756-0500-5-183
30. Sysel, P., Rajmic, P.: Goertzel algorithm generalized to non-integer multiples of fundamental frequency. EURASIP Journal on Advances in Signal Processing **56** (2012). https://doi.org/10.1186/1687-6180-2012-56
31. Talbot, A.: The accurate numerical inversion of Laplace transforms. J. Inst. Maths. Applics. **23**, 97–120 (1979)
32. Vitali, A.: The Goertzel algorithm to compute individual terms of the discrete Fourier transform (DFT). Tech. Rep. DT0089 Rev1, STMicroelectronics (2017)
33. Wang, H., Wu, P., Tanase, I.G., Serrano, M.J., Moreira, J.E.: Simple, portable and fast SIMD intrinsic programming: Generic SIMD library pp. 9–16 (2014). https://doi.org/10.1145/2568058.2568059

ICCS Camera Ready Version 2024
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-63778-0_12