Speeding up the computation of uniform bicubic spline surfaces

Viliam Kačala
Institute of Computer Science,
Faculty of Science,
P. J. Šafárik University in Košice
Jesenná 5,
040 01 Košice, Slovakia
viliam.kacala@student.upjs.sk

Lukáš Miňo
Institute of Computer Science,
Faculty of Science,
P. J. Šafárik University in Košice
Jesenná 5,
040 01 Košice, Slovakia
lukas.mino@upjs.sk

ABSTRACT

Approximation of surfaces plays a key role in a wide variety of computer science fields such as graphics or CAD applications. Recently a new algorithm for evaluation of interpolating spline surfaces with C^2 continuity over uniform grids was proposed based on a special approximation property between biquartic and bicubic polynomials. The algorithm breaks down the classical de Boor's computational task to reduced tasks and simple remainder ones. The paper improves the reduced part's implementation, proposes an asymptotic equation to compute the theoretical speedup of the whole algorithm and provides results of computational experiments.

Both de Boor's and our reduced tasks involves tridiagonal linear systems. First of all, a memory-saving optimization is proposed for the solution of such equation systems. After setting the computational time complexity of arithmetic operations and clarifying the influence of modern microprocessors design on the algorithm's remainder tasks, a new expression is suggested for assessing theoretical speedup of the whole algorithm. Validity of the equation is then confirmed by measured speedup on various microprocessors.

Keywords

Spline interpolation, Bicubic spline, Hermite spline, Biquartic polynomial, Uniform grid, Tridiagonal systems, Speedup

1 INTRODUCTION

The paper is devoted to effective computation of tridiagonal systems. Since evaluation of such systems belongs to challenges of computer science and numerical mathematics, designing fast algorithms for their computation is a fundamental task. One of many applications of tridiagonal linear systems is construction of spline curves and surfaces that pass through the pre-set input points.

Our *reduced* algorithm is based on an interrelation between bicubic and biquartic polynomials that has been proved in [Sza16a], [Min16a] and its application was thoroughly described in [Min16a], [Min15a], [Min15b]. This interrelation was inspired by a similar property between cubic and quartic polynomials uncovered in [Tor14a]. A proof of this interrelation both in 2D and 3D is based on the IZA representation [Tor13a], [Sza13a] which incorporates both interpolation and approximation. The IZA representation was obtained using an r-point transformation that is a generalization of three point model introduced by Dikoussar [Dik97a]. A three point transformation was successfully applied to various approximation problems such as the assessment of unknown degrees in regression polynomials [Tor00a], [Mat05a] or a method for detecting piecewise cubic approximation segments for data with moderate errors [Dik06a].

An idea, based on which the IZA representation has been ultimately derived, appeared in [Rev07a]. The paper [Tor09a] showed how to properly use the IZA representation's reference points for segment connection and their relation to derivatives. Papers [Dik07a], [Sza13a] contain results on approximating 3D data utilizing the reference point approach. The basis for the quartic-cubic interrelation makes up a two-part model, which was first thoroughly studied in [Rev13a] and [Tor13a]. These works proved the validity of the two-part approximation model, which led first to approximation of a quartic polynomial by two cubic ones in [Tor14a] and then to approximation of a biquartic polynomial by two bicubic ones in [Min16a]. The reduced system approach to spline curve construction was proposed in [TorTA] and afterwards it was generalized for case of spline surface construction in [Min15b]. The main goal of this work is both the theoretical and practical confirmation that the reduced algorithm for spline surfaces is faster than the de Boor's original algorithm which we refer to as *full algorithm*.

The structure of this article is as follows. Section 2 is devoted to a problem statement. To be self-contained, Section 3 briefly describes de Boor's algorithm and our recent algorithm based on reduced systems. Section 3.1 shows the standard way of solving tridiagonal linear systems and proposes a modified approach to solve such systems with lesser memory requirements. The next section analyses the architecture of microprocessors in search of a way to speedup the algorithms. The assessed speedup stated in Section 5 is confirmed by real-world measurements summarized in Section 6.

2 PROBLEM STATEMENT

This section defines inputs for the spline surface and requirements, based on which it can be constructed.

Consider a uniform grid

$$[u_0, u_1, \dots, u_{I-1}] \times [v_0, v_1, \dots, v_{J-1}], \qquad (1)$$

where

$$u_i = u_0 + ih_x, \quad i = 1,2,\dots,I-1, \quad I = 2m+1, m \in \mathbb{N},$$
$$v_j = v_0 + jh_y, \quad j = 1,2,\dots,J-1, \quad J = 2n+1, n \in \mathbb{N}.$$

According to [Boo62a], a spline surface is defined by given values

$$z_{i,j}, \quad i = 0,1,\dots,I-1, \quad j = 0,1,\dots,J-1 \qquad (2)$$

at equispaced grid-points, and given first directional derivatives

$$d^x_{i,j}, \quad i = 0, I-1, \quad j = 0,1,\dots,J-1 \qquad (3)$$

at boundary verticals,

$$d^y_{i,j}, \quad i = 0,1,\dots,I-1, \quad j = 0, J-1 \qquad (4)$$

at boundary horizontals and cross derivatives

$$d^{x,y}_{i,j}, \quad i = 0, I-1, \quad j = 0, J-1 \qquad (5)$$

at four corners of the grid.

The task is to define a quadruple $[z_{i,j}, d^x_{i,j}, d^y_{i,j}, d^{x,y}_{i,j}]$ at every grid-point $[u_i, v_j]$, based on which a uniform bicubic clamped spline surface S of class C^2 can be constructed with properties

$$S(u_i,v_j) = z_{i,j}, \qquad \frac{\partial S(u_i,v_j)}{\partial y} = d^y_{i,j},$$
$$\frac{\partial S(u_i,v_j)}{\partial x} = d^x_{i,j}, \qquad \frac{\partial^2 S(u_i,v_j)}{\partial x \partial y} = d^{x,y}_{i,j},$$

where the adjacent spline segments are twice continuously differentiable.

3 FULL AND REDUCED ALGORITHMS

This section is devoted to the description of two algorithms for computing the unknown first derivatives of a C^2-class uniform spline surface's unknown first derivatives. The classic de Boor's algorithm is based on solving tridiagonal linear systems of equations that are further described in [Boo62a]. Henceforward we will refer to the de Boor's algorithm as the *full* algorithm. The recently proposed *reduced* algorithm based on the special approximation property between biquartic and bicubic polynomials breaks down the classical de Boor's computational task to reduced tasks and simple remainder ones as proposed in [Min15a], [Min15b]. The central part of the reduced algorithm comprises three new model equations and five new explicit formulas. Cross derivatives at the boundaries are computed using the classic de Boor's approach. Both algorithms are described in Appendix, thus the paper is self contained and the reader can count the number of mathematical operations for precise comparison.

3.1 Tridiagonal LU factorization

The standard way of solving tridiagonal linear systems

$$\underbrace{\begin{bmatrix} b_1 & 1 & 0 & & \\ 1 & b_2 & 1 & & \\ 0 & 1 & b_3 & & \\ & \ddots & \ddots & \ddots & \ddots \\ & & & & b_K \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_K \end{bmatrix}}_{\boldsymbol{d}} = \underbrace{\begin{bmatrix} r_1 - d_0 \\ r_2 \\ r_3 \\ \vdots \\ r_K - d_{K+1} \end{bmatrix}}_{\boldsymbol{r}}$$

uses the LU factorization $\boldsymbol{A}\,\boldsymbol{d} = \boldsymbol{L}\,\underbrace{\boldsymbol{U}\,\boldsymbol{d}}_{\boldsymbol{y}} = \boldsymbol{r}$, where

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & & & \\ l_2 & 1 & & & \\ 0 & l_3 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots \\ & & & l_K & 1 \end{bmatrix},$$

$$\boldsymbol{U} = \begin{bmatrix} u_1 & 1 & 0 & & \\ 0 & u_1 & 1 & & \\ & & u_2 & & \\ & & \ddots & \ddots & \ddots \\ & & & & u_K \end{bmatrix},$$

the u_i and l_i elements are computed as, see [Bjo15a],

$$\boldsymbol{LU}: \quad u_1 = b, \{l_i = \frac{1}{u_{i-1}}, u_i = b - l_i\}, i = 2, ..., K, \qquad (6)$$

and the forward (Fw) and backward (Bw) steps of the solution are

$$\text{Forward:} \quad \boldsymbol{L}\boldsymbol{y} = \boldsymbol{r}, \qquad (7)$$

where $y_1 = r_1$, $\{y_i = r_i - l_i y_{i-1}\}$, $i = 2,\dots,K$;

$$\text{Backward:} \quad \boldsymbol{U}\,\boldsymbol{d} = \boldsymbol{y}, \qquad (8)$$

where $d_K = \frac{y_K}{u_K}$, $\{d_i = \frac{1}{u_i}(y_i - d_{i+1})\}$, $i = K-1, \dots, 1$.

```
**Input:** $b$, $r[1..K]$
**Output:** $d[1..K]$
$l[2..K]$
$u[1..K]$
$y[1..K]$
$u[2] \leftarrow b$
$y[1] \leftarrow r[1]$
**for** $i$ **from** $2$ **to** $K$ **do**
    $l[i] \leftarrow 1/u[i-1]$
    $u[i] \leftarrow b - l[i]$
    $y[i] \leftarrow r[i] - l[i] \cdot y[i-1]$
$d[K] \leftarrow y[K]/u[K]$
**for** $i$ **from** $K-1$ **downto** $1$ **do**
    $d[i] \leftarrow (y[i] - d[i+1])/u[i]$
```

Algorithm 1: LU factorization

```
**Input:** $b$, $r[1..K]$
**Output:** $r[1..K]$
$p[1..K]$
$m \leftarrow 1/b$
$p[1] \leftarrow m$
$r[1] \leftarrow m \cdot r[1]$
**for** $i$ **from** $2$ **to** $K-1$ **do**
    $m \leftarrow 1/(b - p[i-1])$
    $p[i] \leftarrow m$
    $r[i] \leftarrow m \cdot (r[i] - r[i-1])$
$m \leftarrow 1/(b - p[K])$
$p[K] \leftarrow m$
$r[K] \leftarrow m \cdot (r[K] - r[K-1])$
**for** $i$ **from** $K-1$ **downto** $1$ **do**
    $r[i] \leftarrow r[i] - p[i] \cdot r[i+1]$
```

Algorithm 2: LU factorization

Tridiagonal systems of equations for full and reduced algorithms are solved by LU factorization. All the systems of these algorithms are diagonally dominant with elements 1, 4, 1 or 1, −14, 1.

The process of computing a tridiagonal system is indicated in Algorithm 1. Since $b_1 = b_2 = \cdots = b_{T-1}$, where $T = K$ in case of the full algorithm or $T = K-1$ in case of the reduced algorithm, this method can been improved, see Algorithm 2, requiring less memory as stated in the lemma below.

Lemma 1 *Let K be the number of equations in a linear tridiagonal system with constant diagonals. Then Algorithms 1 and 2 require $5K$ and $2K$ of memory space, respectively.*

4 MICROPROCESSOR'S DESIGN INFLUENCE

In Section 3 we described the full and reduced algorithms for computing the unknown derivatives of spline surfaces. Their time complexity is $O(IJ)$. When determining the asymptotic time complexity it is common to ignore the speed of the algorithm's individual steps, arithmetic operations, etc. Since the asymptotic time complexity is equal for both aforementioned algorithms, it is vital to consider the influence of their individual steps.

One should also keep in mind that larger numbers of operations don't necessarily mean slower completion times as computation time also depends on the type of performed operations. In case of floating point operations it holds that additions and multiplications are similarly fast, but divisions are multiple times slower, see Table 1 and Table 2. In this section we briefly discuss some technical principles how modern CPUs work with data and how one can utilize this in implementation of algorithms. Results of computational experiments are presented in the last section.

Nowadays a performance increase cannot be achieved by just increasing the clock speed. The architectures of modern CPUs use other ways to improve performance, such as superscalar designs, pipelined instructions or thread parallelism.

4.1 Caching

One of the most important ways for a programmer to optimize the algorithm's implementation is the choice of proper data structures. To store input and output values of the two suggested algorithms we use matrices. A matrix can be represented as a jagged array or as a single continuous array where the element of the i-th row and j-th column has an index $n \cdot i + j$, where n is the number of columns.

The main system memory is slower than the CPU which has to wait tens or hundreds of machine cycles to load a value from the main memory. To address this latency issue, modern microprocessors are equipped with small and fast caches that preloads both data and machine instructions of programs from the main memory. Caching is an automated process controlled by the CPU [Pat15a].

Consider an $m \times n$ matrix represented by jagged arrays. When element $a_{0,0}$ is loaded from the matrix, the microprocessor might also cache elements $a_{0,1}$, $a_{0,2}$, …, but not element $a_{1,0}$. Therefore an evaluation of $a_{0,0} + a_{0,1}$ will be faster than the one of $a_{0,0} + a_{1,0}$.

In our case the jagged array representation proved to be more effective as most operations are evaluated on rows and so we do not need to cache the entire matrix which can be unfeasible considering large values of m and n.

4.2 Evaluation time of arithmetic operations

Microprocessor cores consist of several execution units specialized in different types of operations with varied instruction latencies. Values of latencies and throughputs can be found in CPU micro-architecture documentations. The x86 instruction set has many extensions and because it is not practical to include all instruction sets to this test, we chose the most commonly used instruction set extension, namely the Streaming SIMD Extensions 2 (SSE2) as all 64 bit x86 microarchitectures supports these. The sets got a more modern replacement in Advanced Vector Extensions (AVX) whose main advantage lies in the improved vector operations. However vector operations require independent calculations on each particular vector element [Pat15a] and this isn't the case of the considered full and reduced algorithms.

The speedup measurement of the reduced algorithm compared to the full one was conducted on five x86 CPUs covering the generations from AMD K10 to Intel Skylake. In the Table 1 we present four basic arithmetic instruction speeds on these four micro-architectures. The first column contains the name of the architecture and the year of its release to the market. The architectures are ordered alphabetically by the manufacturer and then by the year of release. Instruction **latency** is the number of CPU clocks it takes for an instruction to have its data available. Instruction **throughput** is the number of CPU clocks it takes for an instruction to execute. Some instructions have greater latency than throughput, meaning that the execution unit can process another instruction before the data from a current one are available for further processing. This is referred as pipelining which is one form of the instruction parallelism. The table confirms the expectation that addition and subtraction are equally fast. Therefore these operations will be jointly denoted as ±. Hereafter when we mention the operation of addition we are meaning the subtraction as well. It is clear from the table that division is the slowest operation.

	Latency/Throughput			
Architecture (year)	+	−	×	÷
AMD K10.5 Llano (2011)	4/1	4/1	4/1	20/15
Intel Westmere (2010)	3/1	3/1	5/1	7-22/7-22
Intel Sandy Bridge (2011)	3/1	3/1	5/1	16-22/22
Intel Haswell (2013)	3/1	3/1	5/0,5	14-20/13
Intel Skylake (2015)	3/1	4/0,5	3/0,5	14/4

Table 1: Number of machine cycles for SSE2 double precision floating point arithmetic operations on different x86 generations by [Int16a], [Amd11a] and [Fog16a].

In Table 2 operations were measured in an array containing 512 random elements with the calculations repeated 500 000 times. The last two columns represent measured time ratio of multiplication to addition and ratio of division to addition. For the last two columns we define the following notations:

Definition 1

- *Value $\gamma^{\times}$ is the execution time ratio between multiplication and addition. It means the performance of one multiplication is equivalent to $\gamma^{\times}$ additions.*
- *Value $\gamma^{\div}$ is the execution time ratio between division and addition. It means the performance of one division is equivalent to $\gamma^{\div}$ additions.*

Operations were in the form of $a[i] = a[i] \circ a[i-1]$, where $\circ \in \{\pm, \times, \div\}$ to simulate the form of calculations in Algorithm 2.

A reason to perform measurements instead to rely on processor documentation is the fact, that given latencies and throughput for division of some microarchitectures depends on the input values which are not usually described in documentations. The rows of Table 2 corresponds to the rows of Table 1 but instead of architecture they indicate names of concrete microprocessors.

CPU	±	×	÷	$\gamma^{\times}$	$\gamma^{\div}$
A6-3420M	368	336	1747	0.91	5.20
Core i5 430M	253	347	544	1.37	2.15
Core i3 2350M	227	341	907	1.50	4.00
Core i7 4790	144	207	488	1.44	3.39
Core i7 6700K	135	136	422	1.01	3.13

Table 2: The speed of arithmetic operations on specific CPUs measured in milliseconds.

Comparing the second and fourth columns of Table 1 with the second and third columns of Table 2 we can say that addition and multiplication are similarly fast.

4.3 Parallelism of arithmetic operations

It remains to emphasize another property of the microprocessor's architecture called the *instruction level parallelism* (ILP). Modern processors are pipelined, superscalar and support vectorized computations as we briefly mentioned in the part 4.2. While the vectorization is not the concern for us due to form of Algorithm 2, the superscalar pipelined nature of modern CPU's is to be considered.

Consider the equations in Lemmas 3 and 4 in Section 9. The right-hand sides of the equation contain more than one arithmetic operation. Such expressions are broken automatically into more mutually independent subexpressions and evaluated automatically in parallel [Pat15a].

Table 3 shows how the increase in number of operations will extend the calculation time. For example from the second column in Table 3 it follows that evaluating

$a[i] = a[i] + a[i-1] + a[i-2]$ will be 1.61 times slower than $a[i] = a[i] + a[i-1]$. Operations were measured in an array containing 512 elements with the calculations repeated 500 000 times. Table 4 then shows similar values also for other CPU's, but for the sake of readability only for expressions containing ten numeric operators.

Num. of ops.	$\pm$	$\times$	$\div$
2	1.61	1.61	2.00
3	2.15	2.16	3.00
4	2.59	2.6	4.01
5	3.03	3.04	5.01
6	3.43	3.45	6.02
7	3.85	3.86	7.02
8	4.25	4.27	8.03
9	4.68	4.69	9.03
10	5.09	5.11	10.04

Table 3: Evaluation times of arithmetical operation on Intel Core i7 6700K depending on the number of operations.

The results of this section will be used in Section 6.

CPU	$\pm$	$\times$	$\div$	$\beta^{\pm}$	$\beta^{\times}$
A6-3420M	6.56	7.19	2.40	1.52	1.39
i5 430M	3.84	5.31	10.03	2.60	1.88
i3 2350M	5.10	5.93	9.99	1.96	1.67
i7 4790	6.43	6.17	10.00	1.56	1.62
i7 6700K	5.09	5.11	10.04	1.96	1.96

Table 4: Evaluation times multiples for mathematical expressions containing ten operations of said type compared to those expressions containing only a single operation.

Following the results of the Tables 3 and 4 we define the following notation:

Definition 2

- *Value $\beta^{\pm}$ denotes performance effect of instruction level parallelism on expressions containing more than one addition or subtraction. It means that an expression containing enough number of said operations will be evaluated in $\frac{1}{\beta^{\pm}}$ time compared to a CPU without such a feature.*
- *Analogous, the value $\beta^{\times}$ denotes performance effect of instruction level parallelism on expressions containing more than one multiplication.*

Remark 1 *Since both considered algorithms doesn't contain expressions with more than one floating point division, value $\beta^{\div}$ is not necessary.*

5 THEORETICAL SPEEDUP OF THE ALGORITHM

In this section we count the number of operations and their cost for full and reduced algorithms and provide our main result about the speedup of the latter. In Table 5 we have the cost of arithmetic operations for LU factorization of tridiagonal systems covering both de Boor's and the reduced algorithms. In summary we have the cost of operations for solving one tridiagonal system with K being the size of a matrix. The cost is defined as the sum of arithmetic operations where values for multiplications and divisions are multiplied by their execution time ratios of $\gamma^{\times}$ or $\gamma^{\div}$ respectively. The expressions containing more than one addition or multiplication operand were also multiplied by $\frac{1}{\beta^{\pm}}$ or $\frac{1}{\beta^{\times}}$ to accommodate the ILP effect on such expressions.

	Expression	$\pm$	$\times$	$\div$
Full	LU (6) + Fw (7) + Bw (8)	$3K$	$2\gamma^{\times}K$	$\gamma^{\div}K$
	RHS (11), (12), (13), (14)	K	$\gamma^{\times}K$	0
	Summary full al.	$4K$	$3\gamma^{\times}K$	$\gamma^{\div}K$
Reduced	LU (6) + Fw (7) + Bw (8)	$3K$	$2\gamma^{\times}K$	$\gamma^{\div}K$
	RHS d^x, d^y (15), (17)	$\frac{3}{\beta^{\pm}}K$	$\frac{2}{\beta^{\times}}\gamma^{\times}K$	0
	Summary d^x, d^y	$3(1+\frac{1}{\beta^{\pm}})K$	$2(1+\frac{1}{\beta^{\times}})\gamma^{\times}K$	$\gamma^{\div}K$
	RHS $d^{x,y}$ (21)	$\frac{31}{\beta^{\pm}}K$	$\frac{17}{\beta^{\times}}\gamma^{\times}K$	0
	Summary $d^{x,y}$	$(3+\frac{31}{\beta^{\pm}})K$	$(2+\frac{17}{\beta^{\times}})\gamma^{\times}K$	$\gamma^{\div}K$

Table 5: Cost of operations for LU factorization of full and reduced algorithms in regards to the number of unknowns K.

Full	$\pm$	$\times$	$\div$
d^x (11)	$4IJ$	$3\gamma^{\times}IJ$	$\gamma^{\div}IJ$
d^y (12)	$4IJ$	$3\gamma^{\times}IJ$	$\gamma^{\div}IJ$
$d^{x,y}$ (13)	$8I$	$6\gamma^{\times}I$	$2\gamma^{\div}I$
$d^{x,y}$ (14)	$4IJ$	$3\gamma^{\times}IJ$	$\gamma^{\div}IJ$
Summary	$12IJ$	$9\gamma^{\times}IJ$	$3\gamma^{\div}IJ$

Table 6: Cost of operations for the full algorithm.

Reduced	$\pm$	$\times$	$\div$
d^x (15)	$\frac{3}{2}(1+\frac{1}{\beta^{\pm}})IJ$	$(1+\frac{1}{\beta^{\times}})\gamma^{\times}IJ$	$\frac{1}{2}\gamma^{\div}IJ$
d^x (16)	$\frac{3}{2\beta^{\pm}}IJ$	$\frac{1}{\beta^{\times}}\gamma^{\times}IJ$	0
d^y (17)	$\frac{3}{2}(1+\frac{1}{\beta^{\pm}})IJ$	$(1+\frac{1}{\beta^{\times}})\gamma^{\times}IJ$	$\frac{1}{2}\gamma^{\div}IJ$
d^y (18)	$\frac{3}{2\beta^{\pm}}IJ$	$\frac{1}{\beta^{\times}}\gamma^{\times}IJ$	0
$d^{x,y}$ (19), (20)	$8(I+J)$	$6\gamma^{\times}(I+J)$	$2\gamma^{\div}(I+J)$
$d^{x,y}$ (21)	$\frac{1}{4}(3+\frac{31}{\beta^{\pm}})IJ$	$\frac{1}{4}(2+\frac{17}{\beta^{\times}})\gamma^{\times}IJ$	$\frac{1}{4}\gamma^{\div}IJ$
$d^{x,y}$ (22), (23), (24)	$\frac{21}{4\beta^{\pm}}IJ$	$\frac{8}{4\beta^{\times}}\gamma^{\times}IJ$	0
Summary	$(\frac{15}{4}+\frac{19}{\beta^{\pm}})IJ$	$(\frac{5}{2}+\frac{41}{4\beta^{\times}})\gamma^{\times}IJ$	$\frac{5}{4}\gamma^{\div}IJ$

Table 7: Cost of operations for the reduced algorithm.

Remark 2 *Let us consider I being the number of grid-points along the x-axis, J is the same along the y-axis. The cost of operations has the form $aIJ + bI + cJ + d$, but we provide the count for IJ only.*

Tables 6 and 7 show the cost of operations for individual steps and imply the following result.

Lemma 2 *Consider a uniform grid of size I and J. The total costs of operations are*

$$\left(12 + 9\gamma^{\times} + 3\gamma^{\div}\right)IJ$$

for the full algorithm and

$$\left(\frac{15}{4}+\frac{19}{\beta^{\pm}}+\left(\frac{5}{2}+\frac{41}{4\beta^{\times}}\right)\gamma^{\times}+\frac{5}{4}\gamma^{\div}\right)IJ$$

for reduced algorithm.

We are ready to provide our main result about the speed increase achieved by the reduced algorithm in comparison with the classical full algorithm.

For measuring the expected speedup of the reduced algorithm with respect to the full one, the next theorem proposes an asymptotic expression.

Theorem 1 *Consider a uniform grid of size I and J. If $I,J\to\infty$, then the expected asymptotic speedup of the reduced algorithm is*

$$\frac{12+9\gamma^{\times}+3\gamma^{\div}}{\frac{15}{4}+\frac{19}{\beta^{\pm}}+\left(\frac{5}{2}+\frac{41}{4\beta^{\times}}\right)\gamma^{\times}+\frac{5}{4}\gamma^{\div}} \qquad (9)$$

Proof. Consider a uniform grid of size I and J. From Lemma 2 we get the following costs of operations

- $(12+9\gamma^{\times}+3\gamma^{\div})IJ$ for the full algorithm,
- $\left(\frac{15}{4}+\frac{19}{\beta^{\pm}}+\left(\frac{5}{2}+\frac{41}{4\beta^{\times}}\right)\gamma^{\times}+\frac{5}{4}\gamma^{\div}\right)IJ$ for reduced algorithm.

The speedup is expressed as a cost ratio of both algorithms

$$\frac{(12+9\gamma^{\times}+3\gamma^{\div})IJ}{\left(\frac{15}{4}+\frac{19}{\beta^{\pm}}+\left(\frac{5}{2}+\frac{41}{4\beta^{\times}}\right)\gamma^{\times}+\frac{5}{4}\gamma^{\div}\right)IJ},$$

what completes the proof.

We underline that the asymptotic expression was derived in accordance with Remark 2.

6 MEASURED SPEEDUP

In the previous section, an asymptotic expression for the theoretical speedup has been derived. In this one we show the results of real measurements.

The tested data sets comprises of uniform grid $[u_0,u_1,\ldots,u_{2000}]\times[v_0,v_1,\ldots,v_{2000}]$ where $u_0=-20$, $u_{2000}=20$, $v_0=-20$, $v_{2000}=20$ and values $z_{i,j}$, $d^x_{i,j}$, $d^y_{i,j}$, $d^{x,y}_{i,j}$, see (2) – (5), are given from function $sin\sqrt{x^2+y^2}$ at equispaced grid-points. The speedup values were gained averaging 50 measurements of each algorithm.

The benchmark was implemented in C++14 and compiled with a 64 bit GCC 6.3 using *-Ofast* optimization level. Tests were conducted on five different computers with microprocessors from Tables 1 and 8, all equipped with 8 – 32 GB of RAM and Windows 10 operating system. The tests were conducted on freshly booted PCs after 10 minutes of idle time without running any non-system services or processes like browsers, database engines, etc.

The two $\gamma^{\times}$ and $\gamma^{\div}$ columns of Table 8 contains the execution time ratios of arithmetic operations with respect to addition taken from Table 2. For assessing the theoretical speedup from Theorem 1 we consider the instruction parallelism values $\beta^{\pm}$ and $\beta^{\times}$ from Table 4. The last column holds the values for the real measured speedup of the sequential reduced algorithm with respect to the full one.

	Ratios				Speed-up	
CPU	$\gamma^{\times}$	$\gamma^{\div}$	$\beta^{\pm}$	$\beta^{\times}$	Asses.	Meas.
A6-3420M	0.91	5.20	1.52	1.39	1.13	1.05
i5 430M	1.37	2.15	2.60	1.88	1.24	1.24
i3 2350M	1.39	3.43	1.96	1.67	1.15	1.15
i7 4790	1.44	3.39	1.56	1.62	1.07	1.10
i7 6700K	1.01	3.13	1.96	1.96	1.21	1.23

Table 8: Comparison of assessed and measured speedups on a 2001×2001 grid.

As we can see, the theoretical speedup is comparable with the measured one on the chosen CPU architectures.

7 DISCUSSION

Let us discuss the results from the numerical and experimental point of view. The reduced algorithm works with five types of tridiagonal system, see (15), (17), (19), (20) and (21), that differ from each other with the right hand sides similarly to the de Boor's full systems. Since three of these systems contains two times less equations then the corresponding full systems and their diagonal elements equal -14 instead of 4, from the theoretical view the reduced systems are diagonally dominant and therefore computationally stable [Bjo15a]. The second half of unknowns are computed from simple explicit formulas, see (16), (18), (22), (23), (24) and therefore do not present any issue from the computational view. The model equations of the reduced system were derived to fulfil the requirement of class C^2. The maximal error between the full and reduced system solutions is 10^{-12}, so we can conclude that the proposed reduced method yields numerically accurate results.

Although the reduced one contains twice as many cheap addition, subtraction and multiplication operations, it also contains less than half of expensive divisions giving to new approach a speed increase of factor 1.05 to 1.24 on the rest of tested CPU micro-architectures.

In the future we aim to improve the reduced algorithm for spline surfaces on a uniform grid with simpler equations and formulas for cross derivatives and so further increase the speedup.

8 CONCLUSION

We achieved performance increase of derivatives computation at uniform grid-points for spline surfaces and halved the memory space requirements.

The achieved speedup can be attributed to two interesting facts. Firstly, the reduced algorithm contains less than half the number of divisions. Depending on the CPU microarchitecture a floating-point division is several times slower than addition while floating point multiplication and addition are similarly fast.

Secondly, microprocessor cores are pipelined and superscalar. The reduced algorithm contains many expressions containing more than one arithmetic operation that can be and are evaluated in parallel on most modern x86 CPUs.

Although the reduced one contains twice as many cheap addition, subtraction and multiplication operations, it also contains less than half of expensive divisions giving to new approach a speed increase of factor 1.05 to 1.24 on the tested CPU micro-architectures.

9 APPENDIX

The full and reduced algorithms are given by two lemmas:

Lemma 3 (Full) *If the z values and d derivatives are given, see (2) – (5), then the values*

$$\begin{array}{lll} d^x_{i,j}, & i=1,\dots,I-2, & j=0,\dots,J-1,\\ d^y_{i,j}, & i=0,\dots,I-1, & j=1,\dots,J-2,\\ d^{x,y}_{i,j}, & i=1,\dots,I-2, & j=0,\dots,J-1,\\ & \textit{and } i=0,\dots,I-1, & j=1,\dots,J-2 \end{array} \tag{10}$$

are uniquely determined by the following $2I+J+2$ linear systems of altogether $3IJ-2I-2J-4$ equations:
for $j=0,\dots,J-1$,

$$d^x_{i+1,j}+4d^x_{i,j}+d^x_{i-1,j}=\frac{3}{h_x}(z_{i+1,j}-z_{i-1,j}), \tag{11}$$

where $i=1,\dots,I-2$;
for $i=0,\dots,I-1$,

$$d^y_{i,j+1}+4d^y_{i,j}+d^y_{i,j-1}=\frac{3}{h_y}(z_{i,j+1}-z_{i,j-1}), \tag{12}$$

where $j=1,\dots,J-2$;
for $j=0,J-1$,

$$d^{x,y}_{i+1,j}+4d^{x,y}_{i,j}+d^{x,y}_{i-1,j}=\frac{3}{h_x}(d^y_{i+1,j}-d^y_{i-1,j}), \tag{13}$$

where $i=1,\dots,I-2$;
for $i=0,\dots,I-1$,

$$d^{x,y}_{i,j+1}+4d^{x,y}_{i,j}+d^{x,y}_{i,j-1}=\frac{3}{h_y}(d^x_{i,j+1}-d^x_{i,j-1}), \tag{14}$$

where $j=1,\dots,J-2$.

Lemma 4 (Reduced) *If the z values and d derivatives are given, see (2) – (5), then the values $d^x_{i,j}$, $d^y_{i,j}$, $d^{x,y}_{i,j}$ from (10) are uniquely determined by the following $\frac{3I+2J+5}{2}$ linear systems of altogether $\frac{5IJ-I-J-23}{4}$ equations and $\frac{7IJ-7I-7J+7}{4}$ formulas:*
for $j=0,1,\dots,J-1$,

$$\begin{aligned} &d^x_{i+2,j}-14d^x_{i,j}+d^x_{i-2,j}=\\ &=\frac{3}{h_x}(z_{i+2,j}-z_{i-2,j})-\frac{12}{h_x}(z_{i+1,j}-z_{i-1,j}), \end{aligned} \tag{15}$$

where $i=2,4,\dots,I-3$;

$$d^x_{i,j}=\frac{3}{4h_x}(z_{i+1,j}-z_{i-1,j})-\frac{1}{4}(d^x_{i+1,j}+d^x_{i-1,j}), \tag{16}$$

where $i=1,3,\dots,I-2$, $j=1,3,\dots,J-2$;
for $i=0,1,\dots,I-1$,

$$\begin{aligned} &d^y_{i,j+2}-14d^y_{i,j}+d^y_{i,j-2}=\\ &=\frac{3}{h_y}(z_{i,j+2}-z_{i,j-2})-\frac{12}{h_y}(z_{i,j+1}-z_{i,j-1}), \end{aligned} \tag{17}$$

where $j=2,4,\dots,J-3$;

$$d^y_{i,j}=\frac{3}{4h_y}(z_{i,j+1}-z_{i,j-1})-\frac{1}{4}(d^y_{i,j+1}+d^y_{i,j-1}), \tag{18}$$

where $i=1,3,\dots,I-2$, $j=1,3,\dots,J-2$;
for $j=0,J-1$,

$$d^{x,y}_{i+1,j}+4d^{x,y}_{i,j}+d^{x,y}_{i-1,j}=\frac{3}{h_x}(d^y_{i+1,j}-d^y_{i-1,j}), \tag{19}$$

where $i=1,\dots,J-2$;
for $i=0,I-1$,

$$d^{x,y}_{i,j+1}+4d^{x,y}_{i,j}+d^{x,y}_{i,j-1}=\frac{3}{h_y}(d^x_{i,j+1}-d^x_{i,j-1}), \tag{20}$$

where $j=1,2,\dots,J-2$;
for $i=2,4,6,\dots,I-3$,

$$\begin{aligned} &d^{x,y}_{i,j+2}-14d^{x,y}_{i,j}+d^{x,y}_{i,j-2}=\\ &=\frac{1}{7}(d^{x,y}_{i-2,j+2}+d^{x,y}_{i-2,j-2})-2d^{x,y}_{i-2,j}+\\ &+\frac{3}{7h_x}(d^y_{i-2,j+2}+d^y_{i-2,j-2})+\frac{3}{7h_y}(-d^x_{i-2,j+2}+d^x_{i-2,j-2})+\\ &+\frac{9}{7h_x}(d^y_{i,j+2}+d^y_{i,j-2})+\frac{9}{7h_xh_y}(-z_{i-2,j+2}+z_{i-2,j-2})+\\ &+\frac{12}{7h_x}(-d^y_{i-1,j+2}-d^y_{i-1,j-2})+\frac{12}{7h_y}(d^x_{i-2,j+1}-d^x_{i-2,j-1})+\\ &+\frac{3}{h_y}(d^x_{i,j+2}-d^x_{i,j-2})+\frac{27}{7h_xh_y}(-z_{i,j+2}+z_{i,j-2})+\\ &+\frac{36}{7h_xh_y}(z_{i-1,j+2}-z_{i-1,j-2}+z_{i-2,j+1}-z_{i-2,j-1})-\\ &-\frac{6}{h_x}d^y_{i-2,j}+\frac{12}{h_y}(d^x_{i,j+1}+d^x_{i,j-1})+\frac{108}{7h_xh_y}(z_{i,j+1}-z_{i,j-1})-\\ &-\frac{18}{h_x}d^y_{i,j}+\frac{144}{7h_xh_y}(-z_{i-1,j+1}+z_{i-1,j-1})+\frac{24}{h_x}d^y_{i-1,j}, \end{aligned} \tag{21}$$

where $j = 4, 6, \ldots, J-5$*;*

$$\begin{aligned} d^{x,y}_{i,j} = {} & \frac{1}{16}(d^{x,y}_{i+1,j+1} + d^{x,y}_{i+1,j-1} + d^{x,y}_{i-1,j+1} + d^{x,y}_{i-1,j-1}) - \\ & - \frac{3}{16h_y}(d^{x}_{i+1,j+1} - d^{x}_{i+1,j-1} + d^{x}_{i-1,j+1} - d^{x}_{i-1,j-1}) - \\ & - \frac{3}{16h_x}(d^{y}_{i+1,j+1} + d^{y}_{i+1,j-1} - d^{y}_{i-1,j+1} - d^{y}_{i-1,j-1}) + \\ & + \frac{9}{16h_xh_y}(z_{i+1,j+1} - z_{i+1,j-1} - z_{i-1,j+1} + z_{i-1,j-1}), \end{aligned} \tag{22}$$

where $i = 1, 3, \ldots, I-2,\ j = 1, 3, \ldots, J-2$*;*

$$d^{x,y}_{i,j} = \frac{3}{4h_y}(d^{x}_{i,j+1} - d^{x}_{i,j-1}) - \frac{1}{4}(d^{x,y}_{i,j+1} + d^{x,y}_{i,j-1}), \tag{23}$$

where $i = 1, 3, \ldots, I-2,\ j = 2, 4, \ldots, J-3$*;*

$$d^{x,y}_{i,j} = \frac{3}{4h_y}(d^{x}_{i,j+1} - d^{x}_{i,j-1}) - \frac{1}{4}(d^{x,y}_{i,j+1} + d^{x,y}_{i,j-1}), \tag{24}$$

where $i = 2, 4, \ldots, I-3,\ j = 1, 3, \ldots, J-2$.

10 ACKNOWLEDGEMENT

This work was partially supported by the research grants VEGA 1/0073/15 and VVGS-PF-2015-477.

11 REFERENCES

[Bjo15a] A. Björck: Numerical Methods in Matrix Computations, Springer, 2015.

[Boo62a] C. de Boor: Bicubic Spline Interpolation, Journal of Mathematics and Physics, 41(3), 1962, pp. 212–218.

[Dik97a] N. D. Dikoussar: Function parametrization by using 4-point transforms, Comput. Phys. Commun. 99 (1997), pp. 235–254.

[Dik06a] N. D. Dikoussar, Cs. Török: Automatic Knot Finding For Piecewise Cubic Approximation, Mat. Model., 2006, T-17, N.3.

[Dik07a] N. D. Dikoussar, Cs. Török: On one approach to local surface smoothing, Kybernetika 43 (4), N.4, pp. 533-546, 2007.

[Fog16a] A. Fog: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, Technical University of Denmark, 1996 – 2016, Last updated 2016-12-01, `http://www.agner.org/optimize/instruction_tables.pdf`

[Int16a] Intel 64 and IA-32 Architectures Optimization Reference Manual, Intel Corp., 2016 `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`

[Mat05a] A. Matejčiková, Cs. Török: Noise Suppression in RDPT, Forum Statisticum Slovacum, 3/2005, Bratislava, ISSN 1336-7420, pp. 199–203.

[Min15a] L. Miňo: Efficient Computational Algorithm for Spline Surfaces, ITAT 2015, pp. 30–37.

[Min16a] L. Miňo, I. Szabó, Cs. Török: Bicubic Splines and Biquartic Polynomials, Open Computer Science, Volume 6, Issue 1, Pages 1–7, ISSN (Online) 2299–1093, February 2016.

[Min15b] L. Miňo, Cs. Török: Fast Algorithm for Spline Surfaces, Communication of the Joint Institute for Nuclear Research, Dubna, Russian Federation, E11-2015-77, (2015), pp. 1–19.

[Amd11a] Software Optimization Guide for AMD Family 10h and 12h Processors, 2011 `http://support.amd.com/TechDocs/40546.pdf`

[Pat15a] J. R. C. Patterson: Modern Microprocessors - A 90-Minute Guide, 2001-2015 `http://www.lighterra.com/papers/modernmicroprocessors/`

[Rev07a] M. Révayová, Cs. Török: Piecewise Approximation and Neural Networks, Kybernetika, Vol. 43 (4), No. 4, 2007, pp. 547–559.

[Rev13a] M. Révayová, Cs. Török: Reference Points Based Recursive Approximation, Kybernetika, Vol. 49, No. 1, 2013, pp. 60–72.

[Sza16a] I. Szabó: Approximation Algorithms for 3D Data Analysis, PhD Thesis, P. J. Šafárik University in Košice, Slovakia, 2016.

[Sza13a] I. Szabó, Cs. Török: Smoothing in 3D with Reference points and Polynomials, 29th Spring Conference on Computer Graphics SCCG 2013, Smolenice - Bratislava, Comenius University, ISBN 9788022333771, pp. 39–43.

[Tor00a] Cs. Török: 4-point transforms and approximation, Comput. Phys. Commun., 125 (2000) pp. 154–166.

[Tor14a] Cs. Török: On reduction of equations' number for cubic splines, Matematicheskoe modelirovanie, vol. 26 (2014), no. 11, ISSN 0234-0879, pp. 33–36.

[Tor09a] Cs. Török: Piecewise smoothing using shared parameters, Forum Statisticum Slovacum, 7/2009, pp. 188–193.

[Tor13a] Cs. Török: Reference Points Based Transformation and Approximation, 2013, Kybernetika, Vol. 49, No. 4, 2013, `http://www.kybernetika.cz/content/2013/4/644/paper.pdf`

[TorTA] Cs. Török: Speed-up of Interpolating Spline Construction, to appear.