


High-Performance Computation of Bézier Surfaces on Parallel and Heterogeneous Platforms

Rafael Palomar^{1,2}  · Juan Gómez-Luna³ ·
Faouzi A. Cheikh¹ · Joaquín Olivares-Bueno³ ·
Ole J. Elle^{2,4}

Received: 18 December 2016 / Accepted: 25 April 2017 / Published online: 12 May 2017
© The Author(s) 2017. This article is an open access publication

Abstract Bézier surfaces are mathematical tools employed in a wide variety of applications. Some works in the literature propose parallelization strategies to improve performance for the computation of Bézier surfaces. These approaches, however, are mainly focused on graphics applications and often are not directly applicable to other domains. In this work, we propose a new method for the computation of Bézier surfaces, together with approaches to efficiently map the method onto different platforms (CPUs, discrete and integrated GPUs). Additionally, we explore CPU–GPU cooperation mechanisms for computing Bézier surfaces using two integrated heterogeneous systems with different characteristics. An exhaustive performance evaluation—including different data-types, rendering and several hardware platforms—is performed. The results show that our method achieves speedups as high as 3.12x (double-precision) and 2.47x (single-precision) on CPU, and 3.69x (double-precision) and 13.14x (single-precision) on GPU compared to other methods in the literature. In heterogeneous platforms, the CPU–GPU cooperation increases the performance up to 2.09x with respect to the GPU-only version. Our method and the

This work was supported by the European Economic Area (EEA) Grants NILS 004-2BBRR, the Research Council of Norway through the Hypercept project (No. 221073), The Intervention Centre, Oslo University Hospital (Norway), The Ministry of Education of Spain (TIN2013-42253P) and Junta de Andalucía of Spain (TIC-1692).

✉ Rafael Palomar
rafael.palomar@ntnu.no

¹ Department of Computer Science, NTNU, Teknologivegen 22, 2815 Gjøvik, Norway

² The Intervention Centre, Oslo University Hospital, Oslo, Norway

³ Department of Computer Architecture, Electronics and Electronic Technology, University of Córdoba, Córdoba, Spain

⁴ Department of Informatics, University of Oslo, Oslo, Norway

associated parallelization approaches can be easily employed in domains other than computer-graphics (e.g., image registration, bio-mechanical modeling and flow simulation), and extended to other Bézier formulations and Bézier constructions of higher order than surfaces.

Keywords Bézier surfaces · GPU computing · Parallel computing · Computer-graphics · Heterogeneous computing

1 Introduction

Bézier tensor-product surfaces (in the following referred to as Bézier surfaces) are geometric constructions widely used in engineering and computer-graphics. Despite Bézier curves and surfaces have been studied for decades, they are still an active field of research [1,2].

Due to their simplicity and mathematical properties, Bézier surfaces have been employed in applications such as surface reconstruction from clouds of points [3], modeling of free-form deformations [4,5], interactive manipulation of three-dimensional meshes and rendering [6–8], bio-mechanical modeling [9], hybrid volumetric object representation [10], registration in medical imaging [11, 12], and computer games [13] among others.

Computation of tensor-product Bézier constructions—regardless of whether these are surfaces or higher order constructions like volumes—is considered a computationally expensive task. Applications such as shape optimization in aerodynamics [14], flow modeling [15], simulation [16] and non-rigid medical image registration [11] require, indeed, high-degree Bézier formulations to cope with the complexity of the underlying data.

In the last decade, strategies to parallelize the evaluation¹ of Bézier surfaces have been developed (Sect. 3). These strategies, however, circumscribe mostly to the field of computer-graphics as part of tessellation applications (conversion of continuous surfaces to discrete triangle meshes). Furthermore, these strategies are often limited to the computing of bi-cubic Bézier patches widely used in rendering and animation.

New trends in computing like heterogeneous computing systems (HCS), where multi-core processors are integrated (on-chip) with GPUs in the same device, allow new possibilities for improving the performance. These systems, as opposed to traditional computing systems (e.g., CPU and GPU in separate devices) establish cooperation mechanisms across computing units (e.g., CPU+GPU).

In the literature, works evaluating the performance of traditional systems often present their results as a comparison of devices competing to reach the higher performance. In this context, the performance of multi-core CPUs, GPUs and FPGAs have been evaluated in multiple application domain like image processing [17] and computer graphics [18]. As opposed to this approach, HCS evaluate performance results in terms of cooperative work across computing units. These mechanisms range from distribution of the same processing stage among the computing units [19] to distribution

¹ In the line of other related works, we use the term *evaluation* to refer to computation.

of processing stages (from pipelines) based on optimal mapping to the most adequate computing unit [20,21]. To date, organization of new applications using HCS is an active area of research; in order to understand the challenges and opportunities for HCS for leveraging improved performance over traditional computing systems, benchmark suites like Hetero-Mark [22] and CHAI [23] have been recently developed.

Generalized parallel strategies going beyond bi-cubic Bézier schemes, together with techniques to map the parallelization efficiently onto different hardware platforms, including HCS, have consequently the potential to make an impact in the performance of not only computer-graphics, but a broader range of applications.

1.1 Contribution

The aim of this work is computing real-time Bézier tensor-product surfaces that can be employed not only in rendering applications—where bi-cubic Bézier surfaces are predominant—but also in applications requiring high-degree surfaces. The main contribution of this work is threefold:

- A multi-level method (**MLE**) for the computation of parametric non-rational Bézier tensor-product surfaces of arbitrary degree. The use of this method can be further applied to other formulations (e.g., rational Bézier), as well as tensor-products of higher order than surfaces.
- We propose different techniques to map MLE onto different hardware platforms, including central processing units (**CPU**), discrete and integrated graphics processing units (**GPU**) as well as mobile integrated GPUs—these latter ones being poorly explored in the literature.
- As the latest trends in computing move towards hybrid systems (more than one kind of processor present), we also propose CPU–GPU cooperation mechanisms, including the exploitation of (HCS) models with different properties.

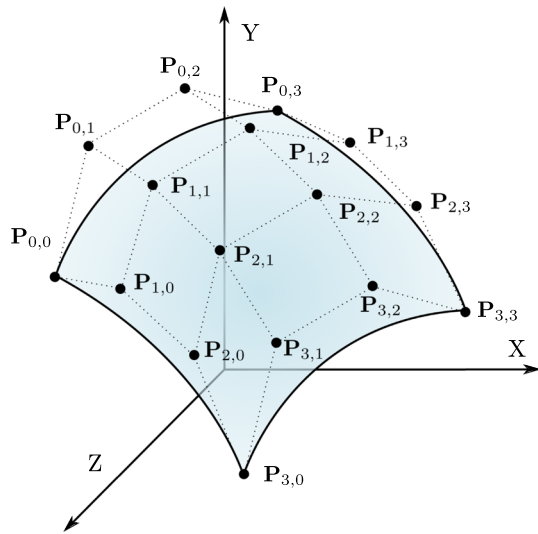
In addition, we review and classify the most important works in the literature concerned with the optimization and acceleration of computation of Bézier surfaces.

The rest of the paper is organized as follows. Section 2 provides fundamental mathematical background on Bézier surfaces. Section 3 lists and shortly reviews relevant works in the literature which accelerate and optimize the computation of Bézier surfaces. In Sect. 4 the proposed method (MLE) is described. Section 5, on other hand, addresses the parallelization and mapping of MLE onto different computing platforms, including CPUs, GPUs and HCSs. In Sect. 6, our experiments and results are described. These results and the most relevant findings are discussed in Sect. 7. Finally, in Sect. 8, some concluding remarks are presented.

2 Background

In this section, a brief description of Bézier surfaces is provided. A deeper description of this type of surfaces and its properties can be found in [24]. For simplicity and clarity reasons, in this work, the focus is on the use of the parametric non-rational formulation of Bézier surfaces. However, the methods presented in this paper are generalizable to

Fig. 1 Bi-cubic Bézier tensor-product surface and its 4×4 net of control points



other Bézier tensor-product formulations (e.g., rational formulations or higher order tensors).

Mathematically, non-rational Bézier tensor-product surfaces $\mathbf{S} : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ are defined as:

$$\mathbf{S}(u, v) = \sum_{i=0}^m \sum_{j=0}^n \mathbf{P}_{i,j} B_{i,m}(u) B_{j,n}(v), \tag{1}$$

where $u, v \in [0, 1]$ form the parametric space of the surface and $\mathbf{P}_{i,j}$ are control points. The m and n values determine the degree of the Bernstein polynomials $B_{i,m}(u)$ and $B_{j,n}(v)$ used as basis functions. These polynomials are generically defined as:

$$B_{i,m}(u) = \binom{m}{i} (1-u)^{(m-i)} u^i, \tag{2}$$

with $0 \leq i \leq m$. $B_{j,n}(v)$ is defined similarly.

The most common case of Bézier surface in the scientific literature is the bi-cubic surface ($m = n = 3$). An example of this type of surface together with its control points is shown in Fig. 1.

Bézier surfaces can also be expressed in terms of the matrix product:

$$\mathbf{S}(u, v) = \mathbf{U}(u)\mathbf{R}(m)\mathbf{P}\mathbf{R}(n)^T\mathbf{V}(v)^T, \tag{3}$$

where the \mathbf{P} is the matrix representing the net of control points. This matrix is given by:

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{0,0} & \mathbf{P}_{0,1} & \dots & \mathbf{P}_{0,n} \\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1} & \dots & \mathbf{P}_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}_{m,0} & \mathbf{P}_{m,1} & \dots & \mathbf{P}_{m,n} \end{bmatrix}.$$

The vectors \mathbf{U} and \mathbf{V}^T are polynomial spaces of degree m and n , associated to the parameterization directions u and v respectively. Generically, these basis vectors take the form $\mathbf{T}(t) = [t^\alpha, t^{\alpha-1}, \dots, t^0]$, where α is the degree of the polynomial space. The matrix of coefficients \mathbf{R} is then defined as:

$$\mathbf{R}(t) = \begin{bmatrix} \binom{t}{0} \binom{t}{t} (-1)^t & \binom{t}{1} \binom{t-1}{t-1} (-1)^{t-1} & \dots & \binom{t}{t} \binom{t-1}{t-1} (-1)^0 \\ \vdots & \vdots & \ddots & \vdots \\ \binom{t}{0} \binom{t}{1} (-1)^1 & \binom{t}{1} \binom{t-1}{0} (-1)^0 & \dots & 0 \\ \binom{t}{0} \binom{t}{0} (-1)^0 & 0 & \dots & 0 \end{bmatrix}.$$

In the literature, some authors like [25] express Eq. (3) in a more compact form:

$$\mathbf{S}(u, v) = \mathbf{U}(u) \mathbf{G} \mathbf{V}(v)^T \tag{4}$$

where $\mathbf{G} = \mathbf{R}(m) \mathbf{P} \mathbf{R}(n)^T$ is constant. This has important implications under the point of view of the implementation and optimization.

3 Related Work

To a great extent, the driving force behind the use and development of Bézier surfaces has been the computer-graphics community. In this context, tessellation algorithms emerged as a mechanism for converting surfaces to triangle meshes [26].

As Bézier surfaces were gaining popularity—eventually becoming the standard for the representation and communication of geometric data—performance became more important. Initially, tessellation was performed in the CPU, then the results were sent to the GPU for further processing and visualization. This transferring process was considered as a bottleneck for high-quality surfaces (which imply high number of triangle transfers). To address this issue, [27] proposed an algorithm and a specific hardware architecture integrated in the GPU.

Later, the GPUs evolved into programmable parallel processors where the graphics pipeline could be redefined by software. These GPUs, included two new programmable units, the *vertex processing unit* dealing with geometry and attributes (i.e., texture coordinates, colors, etc.) and *fragment processing unit* dealing with data stored in textures. User-defined programs, also referred to as *shaders*, were also structured into either *vertex programs* or *fragment programs*. Some works made use of this approach to evaluate and render Bézier surfaces [7, 12, 28–30].

A more contemporary trend to exploit the massive parallelism of graphics hardware is the *general purpose GPU (GPGPU)*, made available through the CUDA [31] and OpenCL [32] programming frameworks. Some works make use of this approach for

the evaluation of Bézier surfaces not only with applications to computer-graphics [4, 8, 33–36].

Regardless of the implementation mechanisms, many of the parallelization strategies can be utilized in both shaders and GPGPU approaches. These strategies can be roughly classified into algorithmic strategies or hardware-specific strategies.

Algorithmic strategies are generally concerned with reducing the number of operations to perform. In this line, [7] uses the matrix formulation in Eq. (3) instead of Eq. (1). Later in [25], the authors make use of Eq. (4), which allows, not only the reduction of the number of operations needed (constant values are pre-calculated), but also the exploitation of spatial coherence of data. Other algorithmic strategies employ numerical approximations, like [8] which is based on forward differencing [37]. However, these methods are subject to error accumulation, and therefore, the generalization to high-degree surfaces is limited.

Hardware-specific strategies are based on providing an efficient mapping of the method on the underlying hardware. The flexibility provided by CUDA and OpenCL allows for a more fine-grained mapping of the method than that obtained by using shaders. [8] make use of a selective transfer of control points to the fast on-chip memory of the GPU (in the following referred to as GPU shared memory), thus providing fast access to those elements frequently accessed during computations. Additionally, [34] utilizes a selective distribution of threads (one GPU thread per control point for evaluation of bi-cubic Bézier patches and one GPU thread per patch for subsequent processing). In [38], the authors present a more generic evaluation (based on non-uniform rational B-splines) approach in which the operations are distributed between CPU and GPU, so that inherently serial operations are carried out by the CPU.

Despite the recent advances of computing in mobile devices, the evaluation of Bézier surfaces in these devices has been given very little attention. To the best of our knowledge, the only work bringing evaluation of Bézier surfaces in mobile platforms is [7]. In this work, the authors highlight the difficulties for real-time tessellation of complex objects.

A summary of all the works considered in this section can be found in Table 1. For each of these works, the table includes the type of Bézier formulation, maximum degree evaluated, employed optimization strategies, programming model used, and whether rendering was the purpose of the application.

4 Multi-Level Evaluation of Bézier Surfaces

On the basis of designing a flexible algorithm able to adapt to different applications, we define the following requirements:

- (A) Update of control points coordinates: this is the most common criterion for real-time evaluation of Bézier surfaces. In this case, the coordinates of the control points change in every evaluation cycle, while the number of control points $[(m + 1) \times (n + 1)]$ and surface resolution $(\rho \times \delta)$ remain invariant. Applications related to evaluation of Bézier surfaces in regular grids, like 3D representation using Bézier patches or deforming surfaces, meet this requirement.

Table 1 Summary of GPU evaluation of Bézier tensor products in the scientific literature

Publication	Bézier formulation	Max. degree evaluated	Optimization strategies		Implementation		Rendering
			Algorithmic	Hardware	Shaders	GPGPU	
[29]	Non-rational	4 × 4			•		•
[27]	Non-rational	3 × 3	•				•
[33]	Non-rational	3 × 3			•		•
[12]	Non-rational	3 × 3			•		•
[8]	Rational	3 × 3	•			CUDA	•
[34]	Rational	3 × 3		•		CUDA	•
[38]	Rational	3 × 3		•	•		•
[30]	Non-rational	N/A		•	•		•
[35]	Rational	N/A				CUDA	
[7]	Non-rational	3 × 3	•		•		•
[25]	Non-rational	3 × 3	•		•		•
[4]	Non-rational	N/A	•			CUDA	•
Our work	Non-rational	12 × 12	•	•		CUDA/OpenCL	*

* Our work does not target rendering applications specifically, however, we provide results with rendering through graphics interoperability

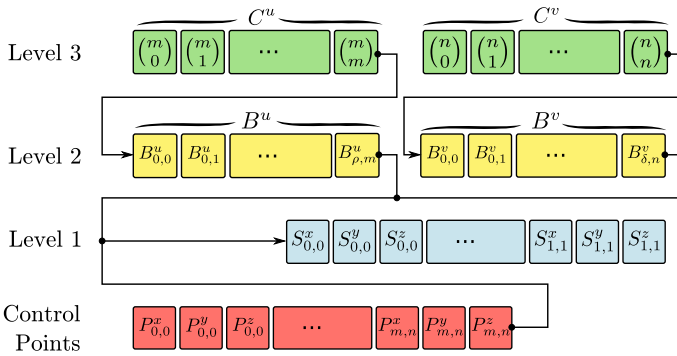


Fig. 2 Decomposition of Bézier formulation in a hierarchy of levels and the associated data items and dependencies

- (B) Variable resolution of surface: subsequent evaluations of the surface present different resolutions ($\rho \times \delta$) (e.g. tessellation applications).
- (C) Variable degree of the surface: this implies a change on the number of control points $[(m + 1) \times (n + 1)]$ in subsequent evaluations (e.g., applications related to degree elevation and surface subdivision).

In order to fulfill these requirements and to reduce the number of operations, we propose an approach based on the a decomposition of the Bézier formulation in a hierarchy of levels. First, we expand Eq. (1) as:

$$S(u, v) = \sum_{i=0}^m \sum_{j=0}^n \underbrace{P_{i,j}}_{C^u} \underbrace{\binom{n}{i} (1-u)^{n-i} u^i}_{B^u} \underbrace{\binom{m}{j} (1-v)^{m-i} v^j}_{C^v} \quad (5)$$

From this formulation, where the different terms, for all points in the surface, are computed and stored in arrays: \mathbf{B}^u and \mathbf{B}^v for arrays of Bernstein basis with lengths $|\mathbf{B}^u| = (m + 1)\rho$ and $|\mathbf{B}^v| = (n + 1)\delta$ for the directions u and v respectively; \mathbf{C}^u and \mathbf{C}^v for arrays of binomial coefficients with lengths $|\mathbf{C}^u| = m + 1$ and $|\mathbf{C}^v| = n + 1$ for the directions u and v respectively.

From a data-dependency standpoint, those arrays can be structured into a hierarchy of levels (Fig. 2) in which each level directly corresponds to the computation of a set of terms:

Level 1 is formed by the coordinates of the set of points \mathbf{S} belonging to the Bézier surface. This level requires the array of control points \mathbf{P} and the Bernstein polynomials corresponding to the u and v directions (\mathbf{B}^u and \mathbf{B}^v respectively), which could have been pre-calculated. A description of the computation of Bézier surfaces with pre-calculated Bernstein basis is given in Algorithm 1. Similarly to Eq. (3), the computation of *level 1* is equivalent to a matrix-vector form of a Bézier tensor-product:

$$S(u, v) = \phi(u)^T \mathbf{P} \psi(v), \quad (6)$$

where ϕ and ψ are subsets of \mathbf{B}^u and \mathbf{B}^v , and \mathbf{P} is the matrix of control points previously described. The complexity of level 1 is characterized by $\mathcal{O}(\rho \times \delta \times m \times n)$.

Algorithm 1 Bézier Surface (level 1)

```

1: function BÉZIER_SURFACE( $m, n, \rho, \delta, \mathbf{B}^u, \mathbf{B}^v, \mathbf{P}$ )
  ▷ Loop over resolution in u direction
2:   for  $i \leftarrow 0$  to  $\rho - 1$  do
  ▷ Loop over resolution in v direction
3:     for  $j \leftarrow 0$  to  $\delta - 1$  do
4:        $\mathbf{S}_{i,j} \leftarrow 0$ 
  ▷ Loop over control points in u direction
5:       for  $k \leftarrow 0$  to  $m$  do
  ▷ Loop over control points in v direction
6:         for  $l \leftarrow 0$  to  $n$  do
7:            $\mathbf{S}_{i,j} \leftarrow \mathbf{S}_{i,j} + \mathbf{P}_{k,l} \times B_{i,k}^u \times B_{j,l}^v$ 
8:         end for
9:       end for
10:    end for
11:  end for
12:  return  $\mathbf{S}$ 
13: end function
  
```

Level 2 represents the basis \mathbf{B}^u and \mathbf{B}^v of the tensor-product (Bernstein polynomials) in the directions u and v respectively, which complexity is characterized by $\mathcal{O}(\rho \times m + \delta \times n)$. Under isotropy conditions (i.e., equal number of control points and resolution for the directions u and v), \mathbf{B}^u equals \mathbf{B}^v , and therefore one of these arrays can be obtained from the other by either memory copy or direct memory addressing. A description of the process, including the *copy/direct-addressing* mechanism, is described in Algorithm 2.

Algorithm 2 Bernstein basis (level 2)

```

1: function BERNSTEIN_BASIS( $m, n, \mathbf{C}^u, \mathbf{C}^v, \rho, \delta$ )
2:   for  $i \leftarrow 0$  to  $\rho - 1$  do
3:      $\mu_i \leftarrow \frac{i}{(\rho-1)}$ 
4:     for  $j \leftarrow 0$  to  $m$  do
5:        $B_{i,j}^u \leftarrow C_j^u \times \mu_i^j \times (1 - \mu_i)^{\rho-1-j}$ 
6:     end for
7:   end for
8:   if  $m \neq n$  or  $\rho \neq \delta$  then
9:     for  $i \leftarrow 0$  to  $\delta - 1$  do
10:       $\mu_i \leftarrow \frac{i}{(\delta-1)}$ 
11:      for  $j \leftarrow 0$  to  $n$  do
12:         $B_{i,j}^v \leftarrow C_j^v \times \mu_i^j \times (1 - \mu_i)^{\delta-1-j}$ 
13:      end for
14:    end for
15:   else
16:      $\mathbf{B}^v \leftarrow \mathbf{B}^u$ 
17:   end if
18:   return  $[\mathbf{B}^u, \mathbf{B}^v]$ 
19: end function
  
```

▷ Isotropy check

▷ Copy/Direct-addressing on isotropy

Level 3 is composed by the array of binomial coefficients \mathbf{C}^u and \mathbf{C}^v which are employed in the computation of the Bernstein basis arrays \mathbf{B}^u and \mathbf{B}^v . Algorithm 3 shows the computation of the binomial coefficients. As in the Bernstein basis (*level 2*), there is no need of duplicated calculation of both \mathbf{C}^u and \mathbf{C}^v under isotropy conditions. The complexity of *level 3* is characterized by $\mathcal{O}(m + n)$.

Algorithm 3 Binomial coefficients (*level 3*)

```

1: function BINOMIALCOEFFICIENTS( $m, n$ )
  ▷ This loop computes the binomial coefficients for u-direction basis
2:   for  $i \leftarrow 1$  to  $m$  do
3:      $C_i^u \leftarrow \frac{(m-1)!}{i!(m-i-1)!}$ 
4:   end for
5:   if  $m \neq n$  then                                     ▷ Isotropy check
  ▷ This loop computes the binomial coefficients for v-direction basis
6:     for  $j \leftarrow 1$  to  $n$  do
7:        $C_j^v \leftarrow \frac{(n-1)!}{j!(n-j-1)!}$ 
8:     end for
9:   else
10:     $\mathbf{C}^v \leftarrow \mathbf{C}^u$                                 ▷ Copy/Direct-addressing on isotropy
11:  end if
12:  return [ $\mathbf{C}^u, \mathbf{C}^v$ ]
13: end function

```

As shown in Fig. 2, the multi-level evaluation of Bézier surfaces opens up the computation (per evaluation cycle) of only those coefficients needed, while reusing all those coefficients remaining invariant. Hence, for evaluations where the number of control points and resolution are constant (i.e., requirement [A]), we can re-utilize pre-computed *level 2* and *level 3*, therefore using computing resources for only *level 1*. Following the same logic, evaluations with variable resolution (requirement [B]) can re-utilize *level 3*, only computing *level 1* and *level 2* for every cycle. Finally, in case the number of control points changes (requirement [C]), all levels need to be computed every cycle.

5 Parallel Implementations

The evaluation of Bézier surfaces is a problem suitable for parallelization due to the lack of data dependencies between output data items (i.e., 3D surface points). The simplest way to parallelize is the use of OpenMP [39] pre-processor directives on multi-core CPUs. For instance, Algorithm 1 can be easily parallelized employing the directive: `#pragma omp parallel for`.

More interestingly, the huge amount of points in a typical Bézier surface matches the availability of computing resources in massively parallel processors such as GPUs. The following section describes our GPU implementation of the evaluation of Bézier surfaces. Afterwards, we explore the cooperation of CPU and GPU on integrated heterogeneous systems in order to attain further acceleration.

5.1 GPU Parallel Computing

Algorithmically, the multi-level evaluation approach proposed in the previous section facilitates the parallelization possibilities of the evaluation of Bézier surfaces. Hence, the parallelization strategy described in this work follows a scheme based on the parallelization of levels, particularly *level 1* and *level 2*. The reader should note that these levels require significantly more operations than *level 3* (e.g. bi-cubic surfaces require the computation of only $2 \times 4 \times 4$ binomial coefficients), even for high-degree surfaces. Furthermore, for most of the applications, *level 3* remains unchanged. Therefore, *level 3* can be computed by the CPU with a negligible impact.

Parallelization of *level 1* (Algorithm 4) and *level 2* (Algorithm 5) is carried out in different kernel functions using a *gather approach* [40], this is, assigning a GPU thread to each output data item (i.e., Bernstein basis coefficient for *level 2* and 3D surface point for *level 1*). This pattern ensures that threads have write access to disjoint memory locations, thus avoiding mutual exclusion and thread synchronization mechanisms, which may introduce serialization. The geometry of the kernels is 2D thread-blocks clustered in a 2D grid², where the block/thread indexes are used to address memory locations related to a particular thread. The parallelization optimizations we apply to these kernel functions can be better understood in terms of existing data dependencies in the computation of an output data item and its neighboring data items within the same block (Fig. 3a), and mapping of data items (basis, surface points and binomial coefficients) onto different processors and memory locations (Fig. 3b).

As shown in Fig. 3a, a GPU thread assigned to the computation of a 3D output surface point (light yellow tile) requires: all the control points, a subspace of the basis \mathbf{B}^u and \mathbf{B}^v , as well as all the binomial coefficients (all these dependencies highlighted in dark red in the figure). Similarly, neighboring GPU threads within a block (dotted tiles) require: all the control points, all the binomial coefficients and neighboring sub-spaces of \mathbf{B}^u and \mathbf{B}^v (in the figure, tiles are highlighted with a dot).

Considering these data dependencies, Fig. 3b shows the distribution of memory in the GPU. Following the line of [8], in our *level 1*, control points are transferred to GPU shared memory (Algorithm 4, lines 2–4), which is a fast on-chip memory that is accessible by all threads within a block. Additionally, in our work, sub-spaces of \mathbf{B}^u and \mathbf{B}^v are transferred to GPU shared memory (Algorithm 4, lines 5–12), since these elements are going to be accessed frequently. Spatial locality of control points and basis ensures coalesced memory accesses, as consecutive threads load consecutive data items. After loading the data items into GPU shared memory, and given that not all threads within a block perform the same amount of memory transfers, intra-block synchronization is necessary (Algorithm 4, line 13).

Computation of Bernstein basis (*level 2*) in GPU can not benefit from using GPU shared memory since the binomial coefficients are accessed only once. However, as in the case of CPUs, it is possible to use the *copy/direct-addressing* mechanism in order to reduce the number of basis elements computed (Algorithm 5, lines 9–19).

² CUDA threads and thread blocks correspond to OpenCL work-items and work-groups respectively. In this work, we use the CUDA terminology.

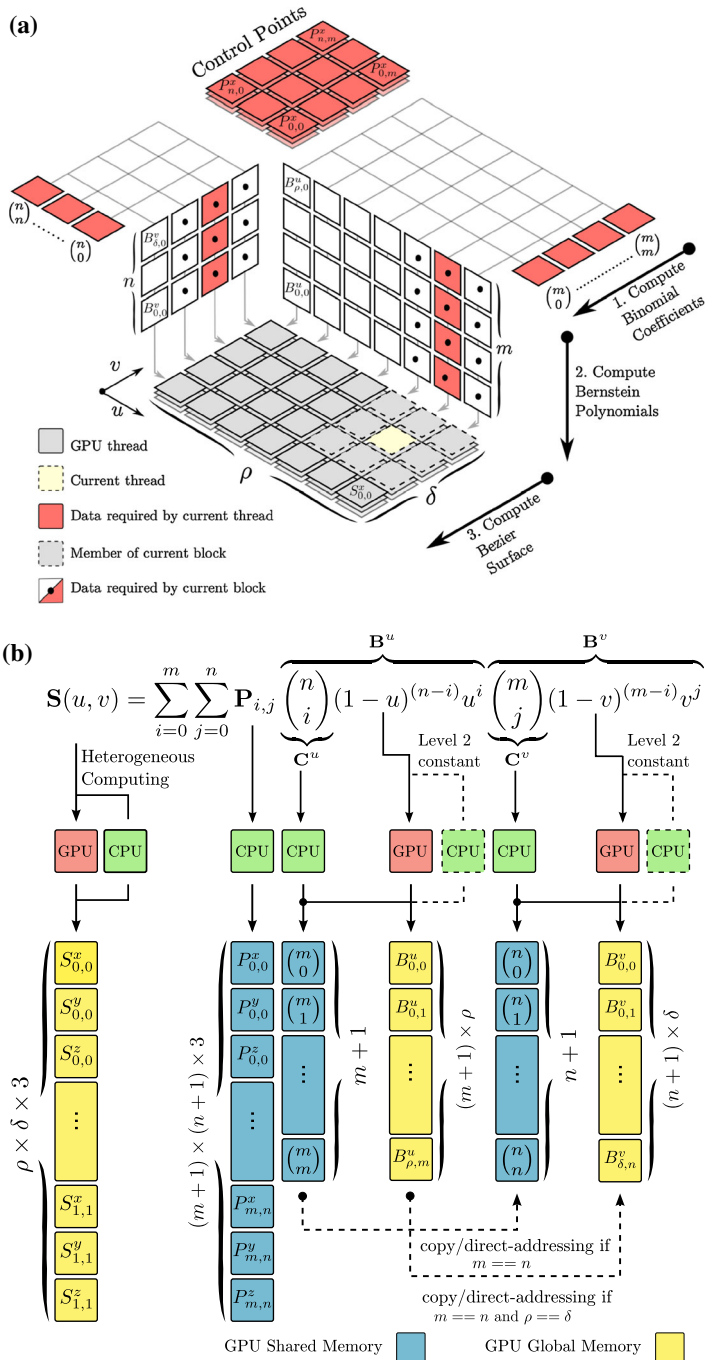


Fig. 3 Parallel computation of Bézier surfaces using MLE under GPU and heterogeneous computing approaches. **a** Geometry of the data and data dependencies at different levels for a GPU thread and its neighboring threads in a thread-block. **b** Distribution of MLE elements across computing units and memory units

Distribution of data items across memory units can be complemented with distribution of computing across processors. To a great extent, the design and distribution of computing is guided by the application. As shown in Fig. 3b, and due to the inherent parallelism, it is indicated that *level 1* and *level 2* are processed by the GPU. Alternatively, and for cases where *level 2* is constant (i.e., constant resolution and number of control points), *level 2* can be pre-computed in CPU. Changes in coordinates of the control points often happen upon user interaction or in a pre-defined way, CPU is therefore adequate. Although these distributions imply cooperation between processors, CPU and GPU do not operate simultaneously, but sequentially one after another. In the next section, we present more elaborated cooperation techniques which allow processors to operate simultaneously in *level 1* (as shown in Fig. 3b), including inter-processor coordination mechanisms.

Algorithm 4 Bézier Surface GPU kernel (*level 1*).

Note: s_x, s_y block sizes in the x and y dimensions. t_x, t_y thread indexes in x and y dimensions. Elements with double-dot accent (e.g. \ddot{B}_i^u) represent allocations in GPU shared memory.

```

1: function BÉZIERSURFACEGPU( $m, n, \rho, \delta, \mathbf{B}^u, \mathbf{B}^v, \mathbf{P}$ )
    ▷ This loop transfers the control points to shared memory
2:   for  $i \leftarrow t_y \times s_x + t_x$  to  $m \times n$  step  $s_x \times s_y$  do
3:      $\ddot{\mathbf{P}}_i \leftarrow \mathbf{P}_i$                                      ▷ Load into shared memory
4:   end for
    ▷ This loop transfers needed basis (u-direction) elements to shared memory
5:   for  $i \leftarrow t_y \times s_x + t_x$  to  $s_y \times m$  step  $s_x \times s_y$  do
6:      $j \leftarrow b_y \times s_y + (i \bmod s_y) + \frac{i}{s_y} \times \rho$ 
7:      $\ddot{B}_i^u \leftarrow B_j^u$                                      ▷ Load into shared memory
8:   end for
    ▷ This loop transfers needed basis (v-direction) elements to shared memory
9:   for  $i \leftarrow t_y \times s_x + t_x$  to  $s_x \times n$  step  $s_x \times s_y$  do
10:     $j \leftarrow b_y \times s_x + (i \bmod s_x) + \frac{i}{s_x} \times \delta$ 
11:     $\ddot{B}_i^v \leftarrow B_j^v$                                      ▷ Load into shared memory
12:  end for
    ▷ Synchronization of threads within block
13:  intra-block_synchronization()
14:   $a \leftarrow b_x \times s_y + t_x$                                ▷ Thread-index of output item (x-coordinate)
15:   $b \leftarrow b_y \times s_x + t_y$                                ▷ Thread-index of output item (y-coordinate)
    ▷ Evaluation of the corresponding surface point
16:  if  $a < \rho$  and  $b < \delta$  then                               ▷ If thread index is within surface
17:     $\mathbf{q} \leftarrow \mathbf{0}$ 
18:    for  $k_i \leftarrow 0$  to  $m$  do
19:       $b_i \leftarrow \ddot{B}_{t_x+k_i \times s_y}^u$ 
20:      for  $k_j \leftarrow 0$  to  $n$  do
21:         $b_j \leftarrow \ddot{B}_{t_x+k_j \times s_x}^v$ 
22:         $\mathbf{q} \leftarrow \mathbf{q} + \ddot{\mathbf{P}}_{k_i+n+k_j} \times b_i \times b_j$ 
23:      end for
24:    end for

```

```

25:    $S_{a \times \delta + b} \leftarrow \mathbf{q}$ 
26:   end if
27:   return S
28: end function

```

Algorithm 5 Bernstein basis GPU kernel (*level 2*).

Note: s_x, s_y block sizes in the x and y dimensions. t_x, t_y thread indexes in x and y dimensions.

```

1: function BERNSTEINBASISGPU( $m, n, \mathbf{C}^u, \mathbf{C}^v, \rho, \delta$ )
2:    $x \leftarrow s_x b_y + t_x$ 
3:   if  $x < (m + 1)\rho$  then
4:      $i \leftarrow x \bmod \rho$ 
5:      $j \leftarrow \frac{x}{\rho}$ 
6:      $\mu \leftarrow \frac{i}{(\rho-1)}$ 
7:      $U_{i+j \times \rho} \leftarrow C_j^u \times \mu^j \times (1 - \mu)^{(m-j)}$ 
8:   end if
9:   if  $m \neq n$  or  $\rho \neq \delta$  then ▷ Isotropy check
10:     $x \leftarrow s_x b_x + t_x$ 
11:    if  $x < (n + 1)\delta$  then
12:       $i \leftarrow x \bmod \delta$ 
13:       $j \leftarrow \frac{x}{\delta}$ 
14:       $\mu \leftarrow \frac{i}{(\delta-1)}$ 
15:       $V_{i+j \times \delta} \leftarrow C_j^v \times \mu^j \times (1 - \mu)^{(m-j)}$ 
16:    end if
17:  else
18:     $V_{i+j \times \delta} \leftarrow U_{i+j \times \delta}$  ▷ Copy/Direct-addressing on isotropy
19:  end if
20:  return [U, V]
21: end function

```

5.2 Heterogeneous Parallel Computing

Heterogeneous computing systems (**HCS**) are composed by hybrid collections of processors (frequently GPUs and CPUs) in the same system [41], and often in the same chip. This trend is intended to satisfy the computational needs of every workload. Inherently sequential or modestly parallel computations are typically executed on the CPU side, while massively parallel phases are executed on the GPU side.

Besides discrete heterogeneous systems where CPU and GPU are connected through *peripheral component interconnect express (PCIe)*, a more recent trend integrates CPU and GPU cores on the same die. The integrated heterogeneous systems solve the bottleneck of the data transfers through the PCIe bus by means of a unified *dynamic random-access memory (DRAM)*.

Current developments try to facilitate communication and concurrency across CPU and GPU cores. The *heterogeneous system architecture (HSA)* [42] provides cache coherence mechanisms [43] and cross-device atomic operations [44]. These systems allow CPU and GPU cores to access the same memory space simultaneously.

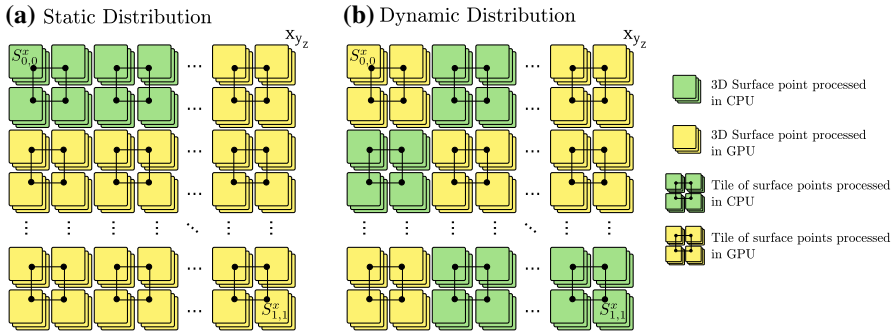


Fig. 4 Heterogeneous parallel computing for a grid of 2×2 tiles of 3D surface elements. **a** SDC where, the first n blocks are statically assigned to the CPU and the remaining blocks are assigned to the GPU. **b** DDC where, blocks are assigned *on-the-fly* (in run-time) to the CPU and the GPU as they finish processing other blocks and become available

A common way to exploit CPU–GPU cooperation is assigning serial tasks to the CPU and parallel tasks to the GPU. However, the regularity of the operations involved in the evaluation of Bézier surfaces makes possible cooperation strategies in which both GPU and CPU can perform the same operations. In this line, we use two CPU–GPU schemes based on the distribution of computation between processors through tiling [45]. The use of tiling is supported by the fact that output surface elements are independent and written in disjoint memory locations.

Two schemes for distributing the computational burden between CPU and GPU are possible depending on the characteristics of the underlying hardware.

Scheme 1: static distribution of computation (SDC). In absence of memory coherence, dynamic communication between CPU and GPU is not possible. Translated to our problem, this means that in run-time none of the processors can verify which part of the 3D surface has already been computed by other processors with concurrency condition guarantees. Despite of that, it is still possible to statically assign the amount of elements that should be processed by the CPU and the GPU. As shown in Fig. 4a, the output surface space is divided into tiles containing neighboring 3D surface points. Each tile is then statically assigned to either a CPU thread or a GPU block. In highly regular problems, the number of tiles processed by the CPU is significantly lower than the number of tiles processed by the GPU. If only *level 1* is computed, offline profiling can help to find a fair workload distribution thanks to the regularity of computations.

Scheme 2: dynamic distribution of computation (DDC). Under memory coherence conditions such as HSA platforms (Fig. 4b), processors assume the computation of non-processed tiles available in a list. This list can be concurrently accessed by CPU and GPU cores through the use of cross-device atomic operations. As opposed to SDC, the assignment of tiles to the CPU threads and the GPU blocks is not known beforehand.

5.3 Rendering and Graphics Interoperability

As previously mentioned, computer-graphics applications and in particular tessellation are the most prominent fields of application for Bézier surfaces. The mechanism by which GPUs can utilize and coordinate computing and graphics capabilities is known as *graphics interoperability*. This mechanism consists of a common buffer of vertices known as *vertex buffer object (VBO)* which is shared by the evaluation of the surface (CUDA/OpenCL) and the rendering. The use of graphics interoperability avoids large data transfers between the CPU and the GPU.

In order to test the performance of MLE in rendering applications, we have implemented an event-driven renderer which employs OpenGL and GLUT [46], together with CUDA graphics interoperability. With the aim to keep simplicity and generality of results, our renderer only considers geometry/topology rendering without coloring and illumination which are not present in every application. The resolution of the output is set to high definition (1920 × 1080) regardless of the underlying hardware. The reader should note that event-driven always present some degree of CPU computing handle the events. This type of renderers is closer to applications where interactions that change the properties of the surface occur (e.g., computer-aided design applications or computer games). Higher performance can be achieved using dedicated engines avoiding events (e.g., animation rendering).

6 Performance Evaluation and Results

In this section, we present the evaluation setup and the results obtained by the proposed method and its parallel mapping onto different hardware platforms, including: one CPU; two discrete GPUs running our CUDA (for NVIDIA) implementation; one mobile integrated GPU (NVIDIA Jetson TK1) running our CUDA implementation; and one integrated GPU (AMD) running our OpenCL implementation. A summary of the employed architectures and the associated implementations are shown in Table 2.

The presented results are based on the comparison of the proposed approach (MLE) with: (1) a “*brute force*” (BRF) iterative approach which computes Eq. (1) and all its elements, including those in Eq. (2) for every evaluation cycle (every frame); and (2) the matrix form in Eq. 4 (MAT) employed by [25]. The reader should note that the latter approach can be seen as an optimized formulation of the matrix form in Eq. (3) previously employed by [7].

Performance evaluation in multi-threading (CPUs) often leads to significant variation of results [47,48] due to memory access mechanisms and operating system scheduling policies. This poses a challenge for benchmarking. In order to reduce the variability of results, our implementations consist of 10 warm-up evaluations followed by 10 measured evaluations which are then averaged. This process is repeated 10 times, providing 10 averaged samples from which the observations exceeding the mean value plus 1.96 standard deviations (95% confidence interval) are removed. After removal of outliers, the resulting observations are used to calculate the mean execution time.

Table 2 GPU/CPU architectures employed for the evaluation of Bézier surfaces in this work

Device	Codename	Type	Year	Implementation
Intel® Core™ i7 930 2.80GHz	Nehalem	CPU	2008	OpenMP
NVIDIA® GTX™ 460	Fermi	Discrete GPU	2010	CUDA
NVIDIA® GTX™ 980	Maxwell	Discrete GPU	2014	CUDA
NVIDIA® Jetson™ TK1	Logan/Kepler	Integrated GPU ^a	2014	CUDA/OpenMP
AMD® A10-7850K	Kaveri	Integrated GPU ^b	2014	OpenCL

^a Heterogeneous mobile architecture. 4-Core ARM Cortex-A15 CPU + Kepler GPU (192 CUDA cores).

^b Heterogeneous architecture 4-core AMD Steamroller CPU + R7 GPU with 8 compute units

Some of our results are expressed in terms of performance increments, measured by a function f in frames per second (**FPS**). Hence, $\Delta\text{MAT} = f(\text{MAT}) - f(\text{BRF})$ represents the difference between MAT and BRF, and $\Delta\text{MLE} = f(\text{MLE}) - f(\text{MAT})$ is the difference between MLE (in *level 1*) and MAT. The results include evaluation (in our results *evaluation*), as well as evaluation followed by a rendering stage (in our results *evaluation+rendering*) through the graphics interoperability mechanisms explained.

In order to widen the applicability of the results, the evaluation takes into consideration the use of single-precision (`float`), as well as double-precision (`double`) data types. For many applications this is an important consideration which may have an impact on the performance, precision and numerical stability. Computer-graphics applications, for instance, are mostly concerned about performance and often use single-precision as the data-type of choice; in simulation applications on the other hand, precision and numerical stability are of paramount importance, and hence, a double-precision data-type is preferred.

6.1 Evaluation on CPU

The evaluation on CPU is performed using a quad-core *Intel® Core™ i7 930 2.80GHz* processor (64-bits architecture) with *Hyper-Threading* technology enabled, thus providing up to 8 virtual cores. The parallel implementation of all the methods evaluated is obtained by means of OpenMP pre-processor directives. The degree of parallelization ranges from 1 to 8 CPU threads.

In a first experiment, the number of control points and the resolution are arranged so that the size of the Bernstein basis arrays meet favorable memory alignment conditions (4×4 control points and 256×256 resolution). Then, evaluations using different methods and different numbers of threads are performed both with and without rendering. For double-precision, the results (Fig. 5a) show a notable performance improvement of MLE over both MAT (2.41x to 3.12x speedup) and BRF (18.98x to 25.47x speedup). The maximum performance is reached by using 4 CPU threads in *evaluation* (1149 FPS), and 8 CPU threads in *evaluation+rendering* (254 FPS). The use of single-precision favors the performance of *evaluation+rendering* (1.62x to 2.82x speedup) over *evaluation*, where there is no significant difference in performance. This is mainly due to the reduction of the CPU–GPU data transfer. Our results show adequate scalability of performance in *evaluation*, this is, a linear increase of performance as the number of threads increases (both for single and double-precision); this behavior holds separately for 1–4 cores (physical cores) and 5–8 (*Hyper-threading*). Similarly, for *evaluation+rendering* linear scaling of results is observed; for double precision, however, linear scaling does not hold for 5–8 threads (*Hyper-threading*) due to memory transfers.

In a second experiment, the number of CPU threads is fixed to 4 and evaluations are performed as combinations of a variable number of control points (4×4 , 8×8 and 12×12) with variable surface resolutions (256×256 , 384×384 and 512×512). As in the first experiment, for double-precision, MLE exhibits higher performance than both MAT (2.18x to 3.15x speedup) and BRF (24.97x to 72.89x speedup). The speedup

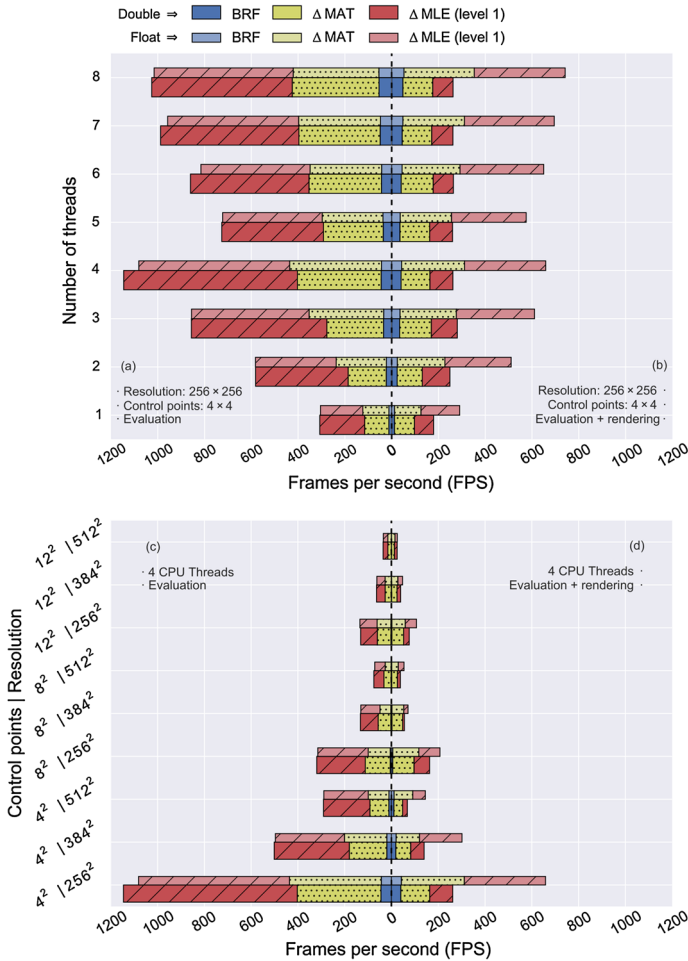


Fig. 5 Parallel evaluation of bi-cubic Bézier surfaces in *Intel® Core™ i7 CPU 930 2.80GHz*. **a, b** Compare computation of the same surface using different number of CPU threads. **c, d** Compare computation of surfaces of variable resolution and number of control points using 4 CPU threads

of MLE over MAT and BRF diminishes as the degree of the surfaces increases—which is further discussed in Sect. 7. For CPUs, including rendering stages not only implies more operations to perform, but also CPU–GPU memory transfers. Thus, the performance of *evaluation+rendering* is within the range 24.4–261.8 FPS for double-precision. As in the first experiment, the use of single-precision can improve the performance significantly (1.02x–2.50x speedup). Scalability of results adheres to the size ($\rho \times \delta$) and degree ($m \times n$) in a linear manner for both parameters according to $\mathcal{O}(\rho \times \delta \times m \times n)$ described in Sect. 1; this phenomenon can be easily observed in Fig. 5c, d where a quadratic increase in either the size or the degree of the surface produces a quadratic performance decrease.

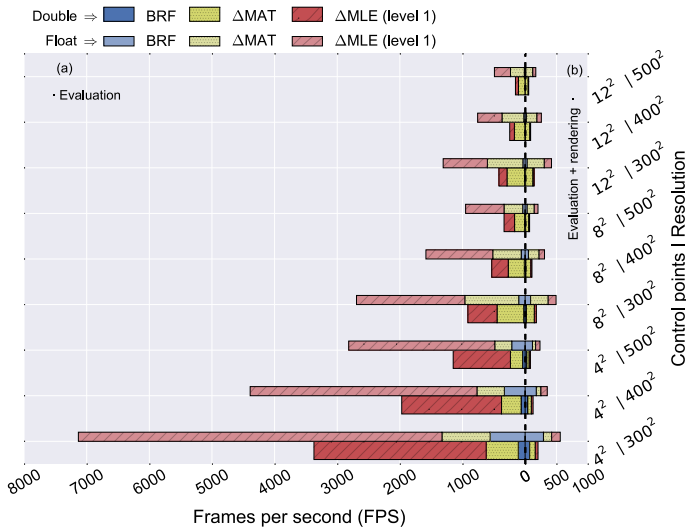


Fig. 6 Parallel evaluation and rendering of Bézier surfaces (variable control points and resolution) in *NVIDIA® GTX™ 460 (Fermi architecture)*

6.2 Evaluation on GPUs

In this section, we test the CUDA implementation of the different methods on two of the three most recent *NVIDIA* architectures (*Maxwell* and *Fermi*). More precisely, the results are obtained from a *NVIDIA® GTX™ 980 4GB* and a *NVIDIA® GTX™ 460 1GB*, both over a *PCIe 2.0* bus. Additionally, we present results on a mobile GPU (*NVIDIA® Jetson™ TK1*). The geometry (size of blocks) of the kernel functions is 16×16 GPU threads, which showed slightly better performance than 8×8 and 32×32 GPU threads, thanks to a higher occupancy value (number of active threads per GPU core).

In a first experiment, the performance of the methods on the older architecture (*GTX 460*) is evaluated. In the case of CPUs, surfaces with variable number of control points (4×4 , 8×8 and 12×12) and resolution (300×300 , 400×400 and 500×500) for evaluation and rendering. As shown in Fig. 6, for double-precision, MLE obtains a significant performance improvement over both MAT (1.43x to 5.42x speedup) and BRF (28.63x to 49.20x speedup). For *evaluation+rendering*, the performance varies between 57 FPS and 206 FPS under double-precision. The use of single-precision favors both *evaluation+rendering* (2.11x to 3.17x speedup) and *evaluation* (2.84x to 2.98x speedup) in a similar manner.

For the most recent architecture (*GTX 980*), the experiment consists of the evaluation and rendering of high-resolution surfaces. The complexity of the surfaces combines variable number of control points (4×4 , 8×8 , and 12×12) with a variable resolution (500×500 , 1000×1000 , and 2000×2000). The results, in Fig. 7, show a performance improvement of MLE over both MAT (1.33x to 3.69x speedup) and BRF (20.68x to 42.62x speedup). For *evaluation+rendering* in double precision, the perfor-

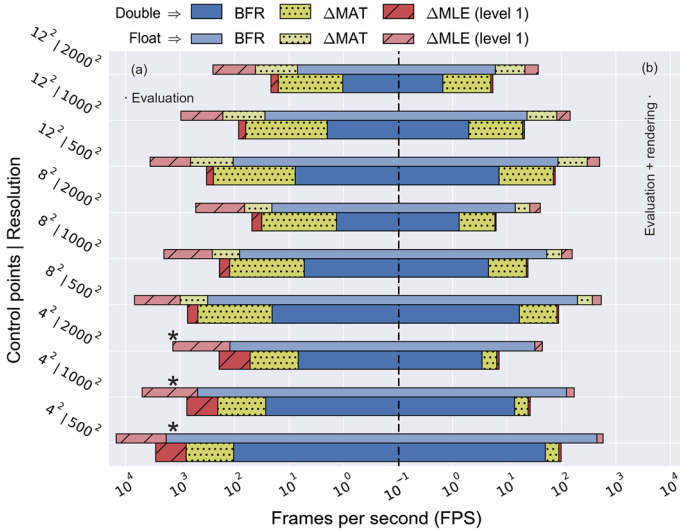


Fig. 7 Parallel evaluation and rendering of Bézier surfaces (variable control points and resolution) in NVIDIA® GTX™ 980 (Maxwell architecture). Results in logarithmic scale. *BRF and MAT show similar performance in both *evaluation* and *evaluation+rendering*

mance varies between 5 FPS and 99 FPS depending on the complexity of the surface. The use of single-precision increases the performance dramatically (5.15x to 11.82x speedup for *evaluation*, and 5.62x to 7.32x speedup for *evaluation+rendering*). Following the same trend as in CPU (Sect. 6.1), the speedup of MLE over MAT diminishes as the degree of the surfaces increases.

The evaluation in mobile GPU was performed using a NVIDIA® Jetson™ TK1. For evaluation purposes, we set the hardware parameters to a high-performance profile (i.e., no CPU down-scaling and GPU frequency at 852MHz). As in previous evaluations, we conduct an experiment consisting of the execution of the methods with the complexity of the surfaces presenting variable number of control points (4 × 4, 8 × 8, and 12 × 12) combined with variable resolution (300 × 300, 400 × 400, and 500 × 500). Figure 8 shows the results in double-precision where *evaluation* on MLE outperforms MAT (1.67x–4.68x speedup) and BFR (29.3x–49.07x speedup). The performance of *evaluation+rendering* varies from 3.7 FPS to 11.5 FPS for double-precision. The use of single-precision over double-precision increases the performance notably (3.72x to 4.80x speedup for *evaluation* and 9.71x to 18.09x for *evaluation+rendering*).

Similarly to our performance results in CPU, performance results in GPU adhere to the linear decrease of performance established by $\mathcal{O}(\rho \times \delta \times m \times n)$ in Sect. 4.

6.3 Evaluation on HCSs

Experiments in this section were designed to demonstrate how the CPU–GPU cooperation can improve the performance of the evaluation of Bézier surfaces in HCSs.

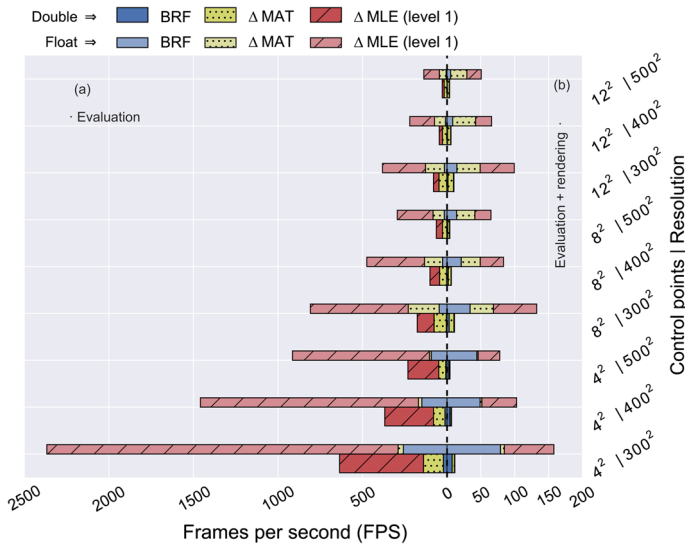


Fig. 8 Parallel evaluation and rendering of single-precision Bézier surfaces (variable control points and resolution) in *NVIDIA® Jetson™ TK1* (Kepler architecture)

Rendering is not subject to cooperation since this is a task carried out by solely the GPU.

For the SDC scheme, we consider the multi-level evaluation approach with pre-defined distribution of computation on a *NVIDIA® Jetson™ TK1* (no memory coherence). In order to obtain the optimal computation distribution, surfaces presenting a different number of control points (4×4 , 8×8 , and 12×12) and different resolutions (300×300 , 400×400 , and 500×500) were evaluated using MLE. For each of these evaluations we assign a CPU load from 0 to 100% in steps of 5%, which determines how many tiles the CPU (and therefore the GPU) will process. We employ 4 CPU threads in order to use the four available cores. The results, in Fig. 9, show how the performance increases when the CPU assumes 10 to 15% of the workload (1.07x to 1.22x speedup compared to a GPU-only approach). Such stable percentages of workload prove that offline profiling has the ability to ensure a reasonably good workload distribution regardless of the number of control points and resolution.

For the DDC scheme, MLE is executed on an *AMD® Kaveri™* (HSA) under DDC. The use of different number of CPU threads (1, 2 and 4) was tested for a variable number of control points (4×4 , 8×8 , and 12×12) and different resolutions (300×300 , 400×400 , and 500×500). Figure 10 shows the benefit of introducing some degree of CPU–GPU cooperation. The positive impact of the CPU–GPU cooperation and the difference between the use of different number of threads diminish as the surface becomes more complex. This increase of performance of the best CPU–GPU cooperation was 1.03x–2.09x speedup compared to the GPU-only approach. In order to compare performance results in different configurations (i.e. 1, 2 or 4 CPU threads cooperating with GPU) careful consideration should be paid to hardware-specific aspects such that power-saving policies; for instance, low complexity surfaces in Fig. 10 show higher

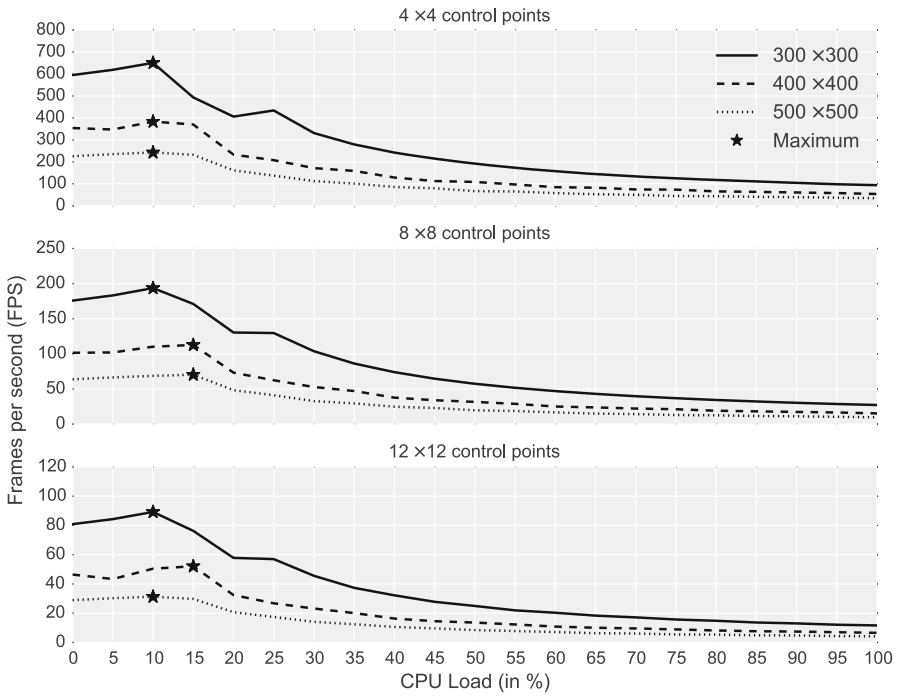


Fig. 9 Parallel evaluation and rendering of Bézier surfaces in *NVIDIA® Jetson™ TK1* under a SDC strategy

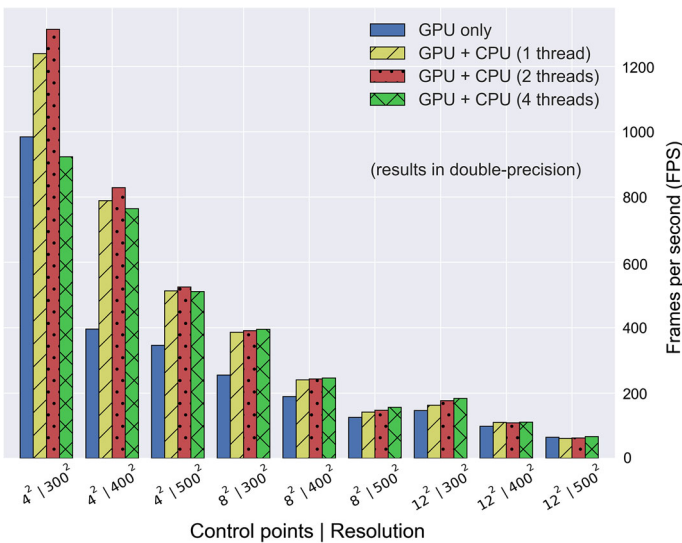


Fig. 10 Parallel evaluation of Bézier surfaces in *AMD® Kaveri™ (HSA)* using different MLE with different CPU–GPU cooperation degrees under a DDC strategy

Table 3 Time results (in ms) for heterogeneous approaches for surfaces of 400×400 and 8×8 control points in AMD® A10-7850K (Kaveri) including *level1* and *level2* computations, memory transfers and allocations

OpenCL	Cooperation	Basis transfer	<i>level2</i>	<i>level1</i>	Surface transfer	Total
2.0*	CPU+GPU (DDC)	0.000	0.052	4.026	0.000	4.078
1.2	CPU+GPU (SDC)	0.293	0.052	4.631	0.278	5.253
2.0*	CPU+GPU (SDC)	0.000	0.054	4.603	0.000	4.657

* Allows the use of memory coherence for CPU–GPU cooperation in the same memory space

performance for the use of 1 and 2 CPU threads compared to 4 CPU threads. This phenomenon, produced by the power-saving policies of the hardware, is contrary to the expected behavior, which is produced for complex surfaces (Fig. 10).

For hardware platforms allowing the use of both SDC and DDC approaches, DDC has greater potential for achieving optimal workload balance between GPU and CPU since the discrete steps on SDC profiling might only find sub-optimal workload balances. This effect is shown in Table 3 where the DDC *level1* computing time is lower than the SDC *level1* time. The use of separate memory spaces for CPU and GPU processing imposes two additional memory transfers: an initial CPU-to-GPU transfer the basis \mathbf{B}^u and \mathbf{B}^v ; and a final CPU-to-GPU transfer of part of the surface to the memory holding the final result (in Table 3 final results are stored in GPU memory).

Theoretical order of complexity, regardless of the cooperation scheme, is preserved as described in Sect. 4, that is, linear decrease of performance is observed when either the degree or the size of the surface increases.

7 Discussion

The use of optimization techniques and parallel computing has been present in modern implementations utilizing Bézier constructions (not only surfaces). As shown in Table 1, the most prominent area of use of parallel computing and optimization techniques is computer-graphics. This explains the relatively low degree of the surfaces employed in the literature (since 3D mesh models can be composed of low-degree sets of Bézier patches). However, due to its interesting properties, Bézier constructions can be found in applications other than computer-graphics. New parallelization and optimization techniques, able to extend the application scope while keeping high-performance results in computer-graphics are, therefore, of great interest. In this line, we propose a method (MLE) and associated parallelization strategies to map the method onto different hardware platforms (CPUs, GPUs, mobile integrated GPUs and HCSs), thus covering a wide spectrum of possible applications.

The idea behind the proposed method is to reduce the number of operations executed. In essence, our approach exploits re-utilization of data items, thus replacing computations by memory accesses. Compared to other methods like MAT in [25]

(some degree of re-utilization) and BRT (no re-utilization), our approach shows a generalized increase of the performance. Using CPUs, the benefit can be as high as 3.12x speedup for double-precision MLE over MAT, and 25.47x speedup over BRF. For discrete GPUs the gain is even larger, making MLE up to 3.69x faster than MAT and up to 42.62x faster than BRF.

The trends observed in the results indicate that as the degree of the surface increases, the speedup of MLE over MAT reduces. This can be explained by looking at the number of operations performed by MAT (Eq. 3) and MLE in *level 1* (Eq. 6). Operationally, MAT needs to compute the basis \mathbf{U} and \mathbf{V} , the products \mathbf{UR} and $\mathbf{R}^T\mathbf{V}^T$, as well the final product with the matrix of control points \mathbf{P} . This imposes an overhead with respect to MLE which only needs to perform a matrix multiplication. In MAT, the size of \mathbf{U} and \mathbf{V} arrays of basis grows linearly with the number of control points while the size of the matrices grows quadratically. Therefore, the overhead takes a larger fraction of the total time for low-degree surfaces than for high-degree surfaces.

The use of single-precision versus double-precision arises as a very important question since our aim is to cover a wide range of applications. Our results reveal a generalized performance gain, that can be as high 11.82x speedup, on using single-precision over double-precision in high-end discrete GPUs (*GTX 980*). Computer-graphics applications, which do not require double-precision are clear targets for choosing single-precision. For scientific applications requiring double-precision (e.g., simulations), the best performance can be achieved by the scientific-class GPU (e.g., *NVIDIA® Tesla™ K20*).

New trends in computing, like HCSs, can leverage higher performance by cooperation mechanisms between processors. In the two strategies for cooperation between processors (SDC and DDC), a speedup as high as 2.09x is observed using DDC, while with the use of SDC a maximum of 1.22x speedup is obtained. The use of SDC and DDC is determined by whether the system possesses memory coherence mechanisms and cross-device atomic operations. For hardware platforms allowing both SDC and DDC, DDC has greater potential to achieve optimal workload balance across processors.

In mobile computing, real-time evaluation and rendering of Bézier surfaces has been considered a difficult task [7]. Our results show that modern mobile processors including an integrated GPU (like *NVIDIA® Jetson™ TK1*), together with parallelization and CPU–GPU cooperation can achieve real-time performance (650.93 FPS in double-precision and 2366 FPS in single-precision) for relatively complex surfaces (bicubic at resolution 300×300). In the absence of rendering, the computation of Bézier surfaces can reach higher performance while presenting better hardware integration possibilities than modern CPUs. In respect to rendering (in which case single-precision is advised), performance can be as high as 157.91 FPS.

With the aim of providing other researchers with a broad coverage of applications, our results include evaluations far beyond the limits of other works found in the literature (Table 1). This, together with the broad set of hardware architectures evaluated, can be used as a guide to establish the limits and scalability of the use of Bézier surfaces in a wide variety of applications including high-degree and high-resolution Bézier surfaces. MLE and the associated strategies proposed, are also easily general-

izable to higher-order Bézier construction and other Bézier formulations (i.e., rational Bézier).

8 Conclusion

In this work, we present a new method (MLE) and the use of different parallel computing techniques to accelerate the computation of Bézier tensor-product surfaces in different hardware platforms (CPUs, discrete GPUs, integrated GPUs, mobile GPUs and HCSs). In line with the latest trends in hybrid computing, we also propose two CPU–GPU cooperation strategies (SDC and DDC) to be exploited by HCS platforms. Our results—which include an exhaustive evaluation using different data-types, different degrees and resolution of surfaces and different computing platforms—show that our method achieves speedups as high as 3.12x (double-precision) and 2.47x (single-precision) on CPU compared to other proposals found in the literature. In GPU computing, the speedup is as high as 3.69x (double-precision) and 13.14x (single-precision). CPU–GPU cooperation strategies employed in this work pose a clear benefit increasing the performance up to 2.09x with respect to GPU-only approaches. MLE, as well as the parallelization and CPU–GPU cooperation strategies are easily generalizable to high-order Bézier constructions (e.g., volumes in medical imaging) and other Bézier formulations (i.e., rational Bézier).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Karim, S.A.A., Saaban, A.: Bézier triangular patches for closed surface. *Appl. Math. Sci.* **8**(8), 355–366 (2014)
2. Yang, H., Wang, G.: Optimized design of Bézier surface through Bézier geodesic quadrilateral. *J. Comput. Appl. Math.* **273**, 264–273 (2015)
3. Akemi, G.: Bezier curve and surface fitting of 3D point clouds through genetic algorithms , functional networks and least-squares approximation. In: *Computational Science and Its Applications–ICCSA 2007*, vol. 4706, pp. 680–693 (2007)
4. Yuan-min, C.U.I.: Real-time accurate free-form deformation in terms of triangular Bézier surfaces. *Appl. Math.* **29**(4), 455–467 (2014)
5. Hilario, L., Falcó, A., Montés, N., Mora, M.C.: A tensor optimization algorithm for Bézier shape deformation. *J. Comput. Appl. Math.* **291**, 264–280 (2015)
6. Concheiro, R., Amor, M., Padrón, E.J., Doggett, M.: Interactive rendering of NURBS surfaces. *CAD Comput. Aided Des.* **56**, 34–44 (2014)
7. Concheiro, R., Amor, M., Padrón, E.J., Gil, M., Martorell, X.: Rendering of Bézier surfaces on handheld devices. In: *21st International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2013)* (2013)
8. Schwarz, M., Stamminger, M.: Fast GPU-based adaptive tessellation with CUDA. *Comput. Gr. Forum* **28**(2), 365–374 (2009)
9. Gilbert, K., Farrar, G., Cowan, B.R., Suinesiaputra, A., Occlshaw, C., Pontre, B., Perry, J., Hegde, S., Marsden, A., Omens, J., et al.: Creating shape templates for patient specific biventricular modeling in

- congenital heart disease. In: Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE, pp. 679–682. IEEE (2015)
10. Zeng, S., Cohen, E.: Hybrid volume completion with higher-order Bézier elements. *Comput. Aid. Geom. Des.* **35–36**, 180–191 (2015)
 11. Soza, G., Bauer, M., Hastreiter, P., Nimsy, C., Greiner, G.: Non-rigid registration with use of hardware-based 3D Bezier functions. In: Proceedings of the 5th International Conference on Medical Image Computing and Computer-Assisted Intervention-Part II, pp. 549–556 (2002)
 12. Li, B., Young, A.A., Cowan, B.R.: GPU accelerated non-rigid registration for the evaluation of cardiac function. *Med Image Comput Comput Assist Interv MICCAI* **11**(Pt 2), 880–887 (2008)
 13. Wang, R., Yang, X., Yuan, Y., Chen, W., Bala, K., Bao, H.: Automatic shader simplification using surface signal approximation. *ACM Trans. Gr.* **33**(6), 226:1–226:11 (2014)
 14. Andreoli, M., Ales, J., Desideri, J.: Free-form-deformation parameterization for multilevel 3D shape optimization in aerodynamics. Technical report, INRIA, FR (2006)
 15. Manzoni, A., Quarteroni, A., Rozza, G.: Shape optimization for viscous flows by reduced basis methods and free-form deformation. *Int. J. Numer. Methods Fluids* **70**(5), 646–670 (2012)
 16. Casquero, H., Liu, L., Bona-Casas, C., Zhang, Y., Gomez, H.: A hybrid variational-collocation immersed method for fluid-structure interaction using unstructured T-splines. *Int. J. Numer. Methods Eng.* **105**(11), 855–880 (2016)
 17. Georgis, G., Lentaris, G., Reisis, D.: Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution. *J. Real-Time Image Process.* **1**, 1–28 (2016)
 18. Jiménez, J., de Miras, J.: Three-dimensional thinning algorithms on graphics processing units and multicore CPUs. *Concurr. Comput. Pract. Exp.* **24**(14), 1551–1571 (2012)
 19. Luo, W., Yang, X., Nan, X., Hu, B.: GPU accelerated 3D image deformation using thin-plate splines. In: Proceedings—16th IEEE International Conference on High Performance Computing and Communications, HPCC 2014, 11th IEEE International Conference on Embedded Software and Systems, ICES 2014 and 6th International Symposium on Cyberspace Safety and Security, pp. 1142–1149 (2014)
 20. Hestness, J., Keckler, S.W., Wood, D.A.: GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU–GPU processors. In: Proceedings—2015 IEEE International Symposium on Workload Characterization, IISWC 2015, pp. 87–97 (2015)
 21. Vilches, A., Navarro, A., Asenjo, R., Corbera, F., Gran, R., Garzarán, M.J.: Mapping streaming applications on commodity multi-CPU and GPU on-chip processors. *IEEE Trans. Parallel Distrib. Syst.* **27**(4), 1099–1115 (2016)
 22. Sun, Y., Gong, X., Ziabari, A.K., Yu, L., Li, X., Mukherjee, S., McCardwell, C., Villegas, A., Kaeli, D.: Hetero-mark, a benchmark suite for CPU–GPU collaborative computing. In: Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016, pp. 13–22 (2016)
 23. Gómez-Luna, J., El Hajj, I., Chang, L.-W., Garcia-Flores, V., de Gonzalo, S., Jablin, T., Pena, A.J., Hwu, W.-M.: Chai: collaborative heterogeneous applications for integrated-architectures. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE (2017)
 24. Piegl, L., Tiller, W.: *The NURBS Book*, 2nd edn. Springer, New York (1997)
 25. Concheiro, R., Amor, M., Bóo, M., Padron, E.J.: Free adaptive tessellation strategy of Bézier surfaces. In: 2014 International Conference on Computer Graphics Theory and Applications (GRAPP), pp. 1–9. IEEE (2014)
 26. Espino, F.J., Bóo, M., Amor, M., Bruguera, J.D.: Adaptive tessellation of NURBS surfaces. *J. WSCG*, **11**(1-3) (2003)
 27. Espino, F.J., Bóo, M., Amor, M., Bruguera, J.D.: Hardware support for adaptive tessellation of Bézier surfaces based on local tests. *J. Syst. Archit.* **53**(4), 233–250 (2007)
 28. Guthe, M., Balázs, A., Klein, R.: GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Trans. Gr.* **24**(3), 1016 (2005)
 29. Loop, C., Blinn, J.: Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Gr.* **25**(3), 664 (2006)
 30. Concheiro R., Amor, M.: Synthesis of Bézier surfaces on the GPU. In: International Conference on Computer Graphics Theory and Applications, pp. 110–115, Angers, France (2010)
 31. NVIDIA: NVIDIA CUDA C Programming Guide (2008)
 32. AMD: Introduction to OpenCL Programming (2010)

33. Dyken, C., Reimers, M., Seland, J.: Real-time GPU silhouette refinement using adaptively blended Bézier patches. *Comput. Gr. Forum* **27**(1), 1–12 (2008)
34. Eisenacher, C., Meyer, Q., Loop, C.: Real-time view-dependent rendering of parametric surfaces. In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games—I3D '09*, pp. 137 (2009)
35. Hanniel, I., Krishnamurthy, A., McMains, S.: Computing the Hausdorff distance between NURBS surfaces using numerical iteration on the GPU. *Gr. Models* **74**(4), 255–264 (2012)
36. Cui, Y., Feng, J.: Real-time B-spline free-form deformation via GPU acceleration. *Comput. Gr.* **37**(1–2), 1–11 (2013)
37. Wallis, B.: Tutorial on forward differencing. In: *Graphics gems*, pp. 594–603. Academic Press Professional, Inc. (1990)
38. Krishnamurthy, A., Khardekar, R., McMains, S.: Optimized GPU evaluation of arbitrary degree NURBS curves and surfaces. *Comput. Aided Des.* **41**(12), 971–980 (2009)
39. OpenMP: OpenMP Application Program Interface (2008)
40. Stratton, J.A., Rodrigues, C., Sung, I.-J., Chang, L.-W., Anssari, N., Liu, G., Hwu, W.W., Obeid, N.: Algorithm and data optimization techniques for scaling to massively threaded systems. *Computer* **45**(8), 26–32 (2012)
41. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. *Sci. Program.* **18**(1), 1–33 (2010)
42. Kyriazis, G.: *Heterogeneous System Architecture : A Technical Review*. Technical report, AMD (2012)
43. Hechtman, B.A., Che, Shuai, Hower, D.R., Tian, Yingying, Beckmann, B.M., Hill, M.D., Reinhardt, S.K., Wood, D.A.: QuickRelease: a throughput-oriented approach to release consistency on GPUs. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 189–200 (2014)
44. Gupta, M., Das, D., Raghavendra, P., Tye, T., Lobachev, L., Agarwal, A., Hegde, R.: Implementing Cross-Device Atomics in Heterogeneous Processors. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pp. 659–668 (2015)
45. Xu, C., Kirk, S.R., Jenkins, S.: Tiling for performance tuning on different models of GPUs. In: *2009 Second International Symposium on Information Science and Engineering*, pp. 500–504 (2009)
46. Wright, R.S., Haemel, N., Sellers, G.M., Lipchak, B.: *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Pearson Education, Upper Saddle River (2010)
47. Wright, N.J., Smallen, S., Olschanowsky, C.M., Hayes, J., Snavely, A.: Measuring and understanding variation in benchmark performance. In: *2009 DoD High Performance Computing Modernization Program Users Group Conference*, pp. 438–443 (2009)
48. Kumar Pusukuri, K., Gupta, R., Bhuyan, L.N.: Thread tranquilizer: dynamically reducing performance variation. *ACM Trans. Archit. Code Optim.* **8**(4), 46 (2012)