Heterogeneous Spline Surface Intersections

Sverre Briseid*
SINTEF ICT, Dept. of Applied Mathematics, Norway

Tor Dokken†
SINTEF ICT, Dept. of Applied Mathematics, Norway

Trond Runar Hagen‡
SINTEF ICT, Dept. of Applied Mathematics, Norway

Abstract

While the PC just a few years ago included only one single-core CPU and a fixed functionality graphics card, the current commodity PC is a heterogeneous system, equipped with both a multi-core CPU and a fully programmable graphics processing unit (GPU). This change has given the commodity PC one to two orders of magnitude increase in computational performance compared with a few years ago [Brodtkorb et al. 2010; Owens et al. 2008]. We will in this paper address the potential of exploiting such a parallel computational capacity for the calculation of intersections and self-intersections of spline represented surfaces. The focus will be on massive parallel spline space refinement by knot insertion, an approach much better adapted to parallel implementations than the knot insertion used in traditional recursive subdivision based intersection algorithms. Rather than presenting a complete algorithm we address the most resource demanding sub-algorithms of surface intersection and self-intersection algorithms and their relative performance on multi-core processors and GPUs for different levels of refinement. Our results show the efficiency of the sub-algorithms on the two types of processors and how this can be used to improve the overall performance on this heterogeneous system.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric Algorithms, Languages, and Systems; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Splines

Keywords: Intersections, splines, GPU, multi-core, OpenMP, CUDA

1 Introduction

In CAD-systems sculptured surfaces are represented as parametric tensor product rational B-spline surfaces, most often denoted NURBS (Non-Uniform Rational B-Spline). As a spline is a piecewise polynomial, a NURBS surface can be decomposed into a set of rational parametric tensor product surfaces (Rational Bézier Surfaces). A rational surface $p(s,t)$ of bi-degree (d_1,d_2) can by elimination of the parametric variables (s,t) be represented as an algebraic surface $q(x,y,z)=0$ of total degree $2d_1d_2$. So the intersection of a bicubic surface $p(s,t)$ with the algebraic representation $q(x,y,z)=0$ of total degree 18, and the bicubic surface $r(u,v)$, can be rewritten to $q(r(u,v))=0$, which is a real algebraic curve of bi-degree $(54,54)$, or total degree 108. It would be nice to know how many branches such a real algebraic curve can have, and which topological structures are possible.

The classification of non-singular plane projective algebraic curves has only been solved for degrees $d \leq 7$ [Viro 2008]. Furthermore the number of connected components (curves) of a plane projective real algebraic curve of degree d is less than or equal to $(d-1)(d-2)/2+1$. Using this we can limit the number of intersection branches in projective space of two bicubic rational surfaces by $107*106/2+1=5672$. When projecting back to the affine spaces some branches can split and further increase the number of branches. In practice intersections of NURBS-surfaces in CAD most often result in one or a few intersection branches. However, when the normal fields of two intersecting surfaces overlap, the number of intersection branches is often experienced to be numerous as the result above indicates.

When current CAD-systems perform intersection of NURBS-surfaces they try to find a sufficiently good solution to a problem where knowledge of the topological complexity is incomplete. To achieve this CAD-systems renounce quality to achieve performance. Approaches employed over the years to find intersection branches include [Dokken and Skytt 2007]:

- Tessellation: Triangulate the surfaces to be intersected, and intersect the triangulations. This works well for transversal intersections, but not for near singular intersections, where the surfaces are near parallel in the area containing the intersection.
- Lattice evaluation: Extract a grid of curves from one surface and intersect with the other surface, and vice versa. Use the detected points to trace out intersection curves. However, this approach can miss intersection branches.
- Recursive subdivision: Recursively subdivide the surfaces to smaller surfaces [Dokken 1985; Sinha et al. 1985]. For each pair of subdivided surfaces check for the possibility of spatial overlap. The nature of the possible intersection is found by analyzing the relation between the normal fields of the surfaces to detect the possibility of internal intersection loops. If internal loops are possible continue the subdivision. As an intersection loop can be infinitesimally the recursion can go on forever unless stopped. In the case where the surfaces intersect tangentially this criteria for detection of loop free intersection will not work.

While tessellation and lattice evaluation will miss considerably more intersections than recursive subdivision, the computation time will be related to the complexity of the intersections found. Recursive subdivision will deliver considerably more complete intersections than the two other approaches, but the computation times can be very long especially in the case of near tangential intersections. Consequently the performance offered by parallel heterogeneous computational resources has a potential to significantly increase the performance and quality of subdivision based intersection algorithms.

*e-mail: sverre.briseid@sintef.no
†e-mail:tor.dokken@sintef.no
‡e-mail:trond.r.hagen@sintef.no

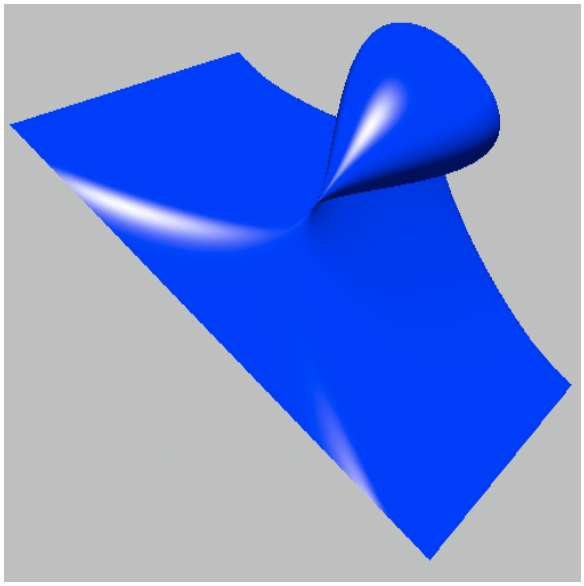

Figure 1: A self-intersecting surface.

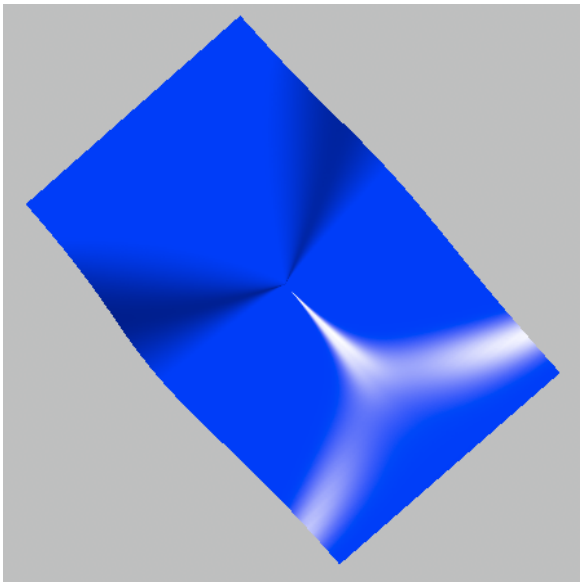

Figure 2: A surface containing a singularity.

In this paper we propose to exploit the computational performance of multi-core processors and GPUs to determine the complexity of the topology of the intersection and self-intersection of NURBS surfaces. The idea is to identify the regions of intersection or self-intersection in the surfaces and classify these. When the classification states that the regions intersect transversally as shown in Figure 3, the actual intersection curve can be traced out by marching or refinement algorithms. For regions where no conclusion can be reached more complex approaches can be employed such as recursive subdivision combined with approximate implicitization [Dokken and Thomassen 2003; Bloomenthal and Wyvill 1997].

For the intersection of two surfaces the classifications will be:

- No intersection occurs.
- All intersections are transversal, points on all branches and loops are identified as shown in Figure 3.
- A region with possible loops, singular or near singular intersections as shown in Figure 4.

For the self-intersection of a surface the classification will be:

- No self-intersection occurs.
- Transversal self-intersection of two parts of the surface.
- Possible singular self-intersection of two parts of the surface.
- Possible locally vanishing surface normal - seed point for surface self-penetration as shown in Figure 1 and 2.

The massive subdivision of the NURBS surfaces is well suited for parallelization. This parallelization can either be on a homogeneous multi-core processor, or in a heterogeneous processor environment combining homogeneous multi-core processors and data stream processors such as GPUs. We will in this paper compare how the massive subdivision and region classification performs on current processors for different levels of subdivision. The idea is to better provide the basis for a future efficient approach for classifying the intersection topology to guarantee quality and direct the recursive intersection algorithms to the challenging regions of the intersection problem.

Until now there has been limited focus on the use of GPUs for CAD type geometry interrogations such as intersections. In [Hoff et al. 2001] collision detection of two 2D objects was addressed, while in [Agarwal et al. 2004] collision detection of polytopes was addressed. Practical algorithms for accelerating geometric queries on models made of NURBS surfaces were addressed in [Krishnamurthy et al. 2009]. [Briseid et al. 2006] showed a method for speeding up surface intersections using a GPU. The results showed a good speedup when compared with a single-core CPU. In this paper we update the method to use the modern programming model for a GPU. We also extend the method to a heterogeneous system by including a multi-core CPU. Additionally we add a module for analyzing the intersection topology.

The paper consists of six sections. In section 2 we describe the programming model we use to instruct the computational resources. In section 3 the intersection problem is described. In section 4 we define the intersection modules to be accelerated. Section 5 gives some results. The concluding remarks are given in section 6.

2 Heterogeneous Programming Model

In our heterogeneous system we use the standard APIs OpenMP and CUDA.

The multi-core CPU is addressed using OpenMP [Chapman et al. 2007]. OpenMP is an API for multi-platform shared memory multiprocessing, supporting C/C++ and Fortran. It was released in 1997, and has gained wide interest since then. The parallelization follows from inserting compiler pragmas into the source code, telling the compiler which code blocks that may be run in parallel. The syntax is easy, giving a low learning threshold for the programmer.

In 2007 NVIDIA released the CUDA (Compute Unified Device Architecture) programming environment [Kirk and Hwu 2010]. Prior to CUDA the classical GPGPU (General Programming on the GPU) approach was to use OpenGL. Although the interface allowed for an extensive use of the GPU, the programming model was limited by the origin of the toolkit. Centered around displaying graphics on a screen the model required the programmer to understand a lot of

Figure 3: Two surfaces with a transversal intersection.

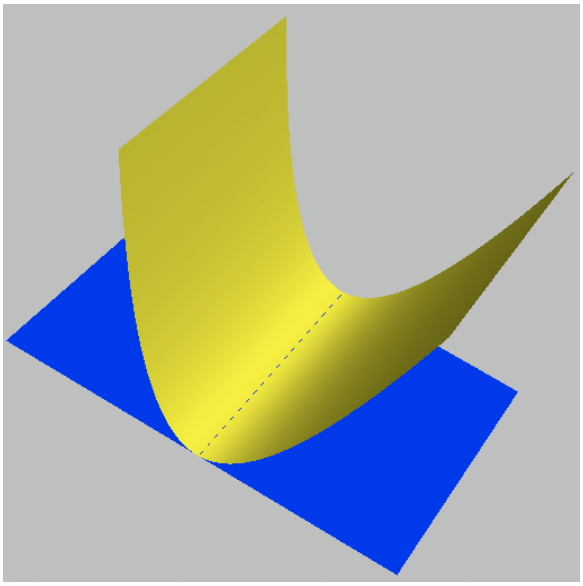

Figure 4: Two surfaces with a tangential intersection.

concepts not connected to the computational problem. With CUDA, NVIDIA removed many of these abstract layers, allowing the programmer more easily to run programs using the GPU. The CUDA programming model is on a higher abstraction layer and resembles C++. The programmer writes small programs to be run on the GPU, called kernels, which are quite similar to function calls carried out on the CPU. This does not mean that any task may be ported to CUDA with little effort and great success. There still are quite a few architectural issues to consider to make for an efficient GPU implementation. But it has become a lot easier to get a program to run on the GPU, even for novice GPU programmers.

3 Spline Surface Intersections

Traditional recursive intersection algorithms will be a potential bottleneck, as these will imply offloading an operation to only one processing core. A natural approach is then to analyze the traditional algorithm and isolate tasks with a high degree of parallelism. These tasks are put into separate modules, giving a good flexibility towards using multiple processing elements.

In our system we test for intersections on tensor product spline surfaces. We are thus given a spline surface of bi-degree (d_1, d_2),

$$S(u,v) = \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} c_{i,j} B_{i,d_1}(u) B_{j,d_2}(v), \tag{1}$$

where $B_{i,d_1}(u)$ and $B_{j,d_2}(v)$ are the B-spline functions and $c_{i,j}$ are the 3-dimensional control points. Our intersection modules will essentially only be working on the control points, with the spline space used for setting up the system.

Central to our approach is a massive subdivision. The subdivision of a spline surface is done by multiplying the coefficient matrix by a refinement matrix, where the former is given by the control points $c_{i,j}$ in equation 1 and the latter constitutes the transition from the original spline space to the refined spline space [Farin 2002]. By using this subdivision strategy we break the problem up into smaller parts which are easier to handle. More than looking for intersecting parts, we enlarge the area which is guaranteed not to self-intersect or contain an intersection. For a trivial example with no intersections we will only need to use a low subdivision level. Surfaces with a complex intersection topology will naturally be more demanding, requiring a closer look. When given an area which may contain an intersection, we use the subdivision to zoom in. Once we have reached a level where any intersection curve is guaranteed to be simple, we are done. Tracing out the intersection curves may then be efficiently performed on the CPU.

In this paper we use single precision rather than double precision in the algorithms. This is not critical as the NURBS algorithms used are extremely stable numerically, and the ambition is to classify the topology of surface intersections and self-intersections, and provide starting points for tracing out intersection curves. Central parts of the classification will be based on the calculation of bounding boxes based on Bézier sub-patches of the surface or its corresponding normal surface. Using single precision gives a relative error of the extent of the bounding boxes of 10^{-6}, while double precision gives a relative error of the extent of the bounding boxes of 10^{-15}. Should double precision be needed, this is supported at half single precision speed by state-of-the-art GPUs. However, in most cases using single precision will have little influence on the classification result.

4 The Intersection Modules

To allow for the various parts of the intersection analysis to be run in parallel, we split the problem into separate modules. The flowchart in Figure 5 shows the dependency of the intersection modules. Our approach consists of two main branches which are totally independent. Additionally, the second branch is split into two branches further down in the pipeline. This indicates a potential for an efficient usage of our heterogeneous system by letting multiple processing elements run independent modules simultaneously. Furthermore, the independence of the branches in the flowchart allows us to use different subdivision levels. This may be in the form of increasing the subdivision level by iterating on specific modules.

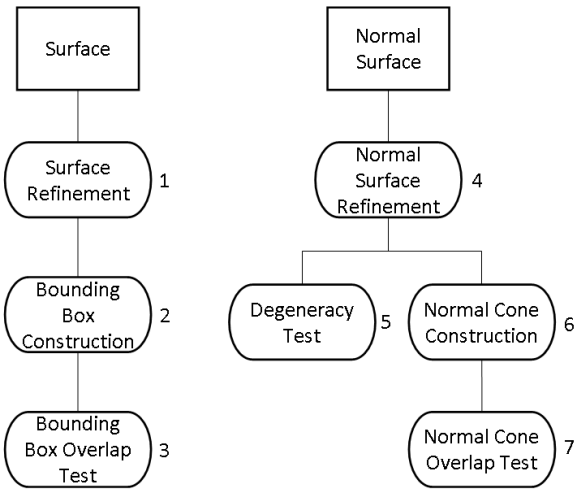

Figure 5: Pipelines of intersection modules.

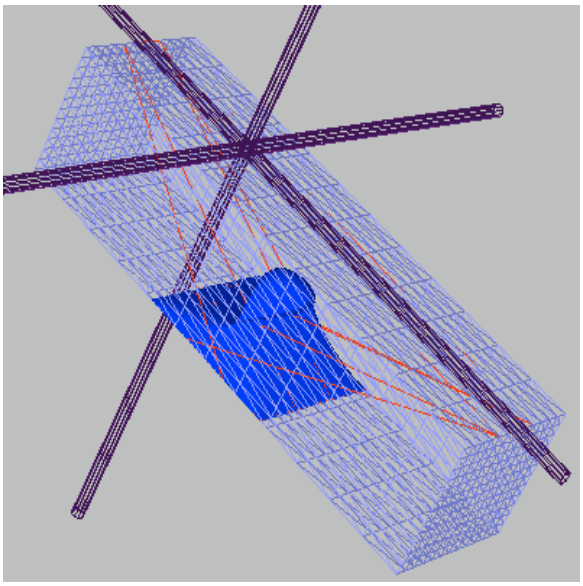

Figure 6: Axis-aligned bounding box enclosing the control point polygon of a spline surface.

4.1 Module 1: Surface Refinement

The input surface is refined to a chosen subdivision level n, corresponding to a grid of $2^n \times 2^n$ Bézier sub-patches. This is done by performing a matrix-matrix multiplication on the control polygon of the surface, highly suitable for massive parallelization. We first refine in one of the parameter directions, followed by refinement in the other parameter direction. When given a bicubic Bézier surface as input, and subdividing to level $n = 8$, for each dimension this corresponds to multiplying a 4×4 matrix by a 4×1024 matrix, then multiplying a 1024×4 matrix by a 4×1024 matrix. The (computational) order of the module is $O(2^{2n})$, where n is the subdivision level.

On the GPU we use CUBLAS, which is a BLAS (Basic Linear Algebra Subprograms) implementation using CUDA. The library is very efficient, typically giving a speedup compared to a single-core CPU version of one order of magnitude.

4.2 Module 2: Bounding Box Construction

When module 1 has completed the surface refinement, we create an axis-aligned bounding box for each Bézier sub-patch, as shown in figure Figure 6. Since a spline surface lies inside the convex hull of its control point polygon, this is a fast and straightforward operation. By choosing the appropriate data structure, giving good data locality, this is well suited for OpenMP. The data locality ensures good cache efficiency, crucial to a method with such a low compute intensity. The order of the module is $O(2^{2n})$, where n is the subdivision level.

On the GPU we use an all-reduce method to extract the minimal and maximal coordinate values from each of the Bézier sub-patches. We do this by comparing pairs of coefficients, for each dimension storing the minimal/maximal value. This reduction process is carried out until we are left with the global minimal/maximal value. It is highly parallel, very well suited for the GPU.

4.3 Module 3: Bounding Box Overlap Test

Once we have created the bounding boxes for all the Bézier sub-patches, we check if pairs of bounding boxes overlap, as shown in Figure 7. For the self-intersection-test this is performed on all the sub-patches in the surface. Adjacent patches are excluded as the shared border trivially gives an overlap. That particular case is handled by module 6, where we look at the normal cone of the sub-patches and the sum of the normal cones of adjacent sub-patches. If we are testing for intersections between two surfaces, we expect both surfaces to already have been analyzed for self-intersections, hence the pair is constructed from both surfaces. If we encounter a pair with overlapping bounding boxes the patches may intersect. Otherwise we know that no such intersection can exist and there is no need for further subdivision. In general we need to check $(m-1)(m-1)/2$ pairs, where $m = 2^{2n}$ is the number of Bézier sub-patches. The test itself is very lightweight as it, for each pair, consists of comparing up to 3 pairs of floating point numbers. However, the order of the module is $O(2^{4n})$, where n is the subdivision level. We should expect this to rapidly become a dominating module for larger n.

4.4 Module 4: Normal Surface Refinement

The module is essentially the same as module 1 (surface refinement). For a surface with bi-degree (d_1, d_2), the bi-degree of the normal surface is $(2d_1 - 1, 2d_2 - 1)$. When given a biquintic Bézier normal surface as input, and subdividing to level $n = 8$, for each dimension the refinement operation corresponds to first multiplying a 6×6 matrix by a 6×1536 matrix, then multiplying a 1536×6 matrix by a 6×1536 matrix. The order of the module is $O(2^{2n})$, where n is the subdivision level.

Similar to the surface refinement module we use CUBLAS for the matrix-matrix multiplications on the GPU.

4.5 Module 5: Degeneracy Test

To identify singular parts of the surface we locate the degenerate parts in the normal surface. By using the refined normal surface we check if any of the Bézier normal sub-patches contain or lie

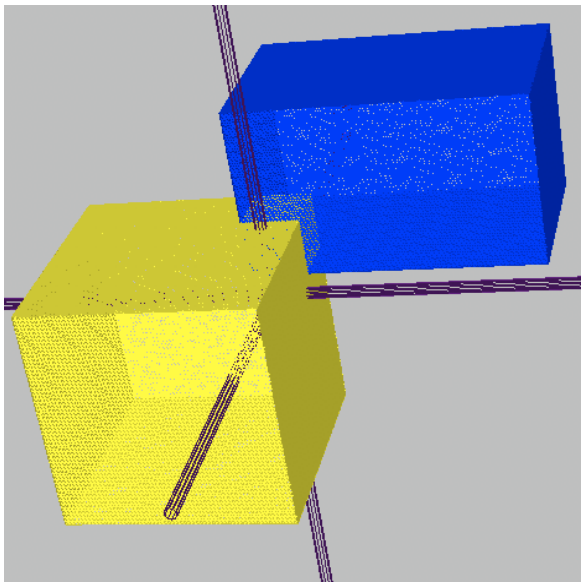

Figure 7: Two overlapping bounding boxes.

close to the origin. This is done by creating the bounding boxes from the refined normal surface and then calculating the distance to the origin. We should subdivide until this distance is greater than the intersection tolerance or the size of the boxes are smaller than the intersection tolerance. This module is not run when we test for intersection between two surfaces as we then expect the input surfaces to already have been analyzed for self-intersections. The order of the module is $O(2^{2n})$, where n is the subdivision level.

On the GPU this is a two-step procedure. We first use an all-reduce method to generate the bounding boxes for the refined normal surface. We then run through these boxes to check for degeneracy.

4.6 Module 6: Normal Cone Construction

The normal cone for a surface represents an upper bound for the span of the normal surface. It consists of a direction and an angle describing the width of the cone. It is used to check for self-intersections in a surface, as well as when classifying whether an intersection may be of a transversal type or a tangential type. The classification test is handled by the next module, the cone overlap module. Assuming that we are looking for self-intersections, we check if the normal cone angle of the sub-patches is greater than π, which allows the sub-patch to contain a self-intersection. We also check whether the span of the normal cones for two adjacent sub-patches is greater than π. This is a solution to the problem in the box overlap module where the shared border between adjacent patches gives a trivial overlap. If any of these tests are positive we should subdivide further to remove the possibility of self-intersecting sub-patches. We focus on speed when we construct the cones, giving us a slightly wider cone. For some cases this may result in the need of a higher subdivision level to rule out self-intersections, but overall our approximation approach gives an increased speed. The order of the module is $O(2^{2n})$, where n is the subdivision level.

On the CPU our approach is linked with the next module, the normal cone overlap test. The number of pairs to check grows rapidly as the subdivision level n increases. Instead of storing the angle of the normal cone, we store the sine and cosine values of the angle. We then use the trigonometric identity $\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$ when testing for overlapping cones, where a and b denote the cone angles in the two normal cones. On the CPU this is a lot faster than computing the angle between the cone centers and then comparing the angle with the cone widths.

On the GPU this module consists of a two-step procedure. We first use an all-reduce method to compute the average value of the control points in the refined normal surface, corresponding to the average normals in the refined input surface. For each Bézier normal sub-patch we then find an upper bound for the distance to the control points. Together with the average direction this defines our approximation of the normal cone. Again, the convex hull property of the spline surface ensures us that this cone encloses the normal cone. On the GPU we may choose between the standard inverse cosine evaluation and a hardware implemented version. The latter is extremely fast at the cost of a slightly lower accuracy. Since we are interested in whether the angle is smaller than π and we thus only need an approximation, the lower accuracy is a fair price to pay for greater speed. This approach, as opposed to the similar CPU version of the module, gave a better GPU speed on both this module and the normal cone overlap module.

4.7 Module 7: Normal Cone Overlap Test

We run through all pairs of normal cones to check if they overlap. When we are testing for intersections between two surfaces we assume that the input surfaces have already been checked for self-intersections, hence each pair is constructed from both surfaces. If the overlap test is false, we are guaranteed that any intersection is of a transversal type. This means that an intersection curve in that area can easily be traced out on the CPU. If the normal cones do overlap, and the result from module 3 shows that the corresponding Bézier patches overlap, the Bézier patches may contain a tangential intersection. We should then subdivide further. Do note that when testing for self-intersections on a surface, the normal cone of adjacent patches will trivially overlap. This was handled in module 6 (normal cone construction), where we tested whether the two sub-patches when viewed as a single surface could include a self-intersection. The order of the module is $O(2^{4n})$, where n is the subdivision level. This is a very dominating module for larger n.

5 Results

Our test system consists of a PC equipped with 6 GB of RAM. The multi-core CPU is a 2.67 GHz quad-core CPU (Core i7 920) and the GPU is a GeForce GTX 470 with 1.25 GB of RAM.

We have tested the intersection modules on a cubic Bézier surface, using subdivision levels from $n = 1$ up to $n = 10$. Since the subdivision is performed uniformly on the input surface, the results are independent of the geometry of the input. The results from running the modules on an intersection case with two surfaces will be very similar, with module 5 (degeneracy) removed from the system.

The benchmarks were performed without timing the communication between the CPU and the GPU. This potential bottleneck will become less of an issue with modern designs, which will decrease the data transfer overhead between the CPU and the GPU. It should be further noted that the tests were performed without timing the overhead in the form of setting up the system. This will imply some additional punishment to a low subdivision level.

As shown in Table 1 and illustrated in Figure 8, the CPU benefits from using more cores at all subdivision levels. But the advantage, measured as the speedup factor compared to the single-core CPU,

Level	1 core	2 cores	3 cores	4 cores	GPU
3	1.61e–04	9.70e–05	7.38e–05	6.48e–05	3.21e–04
4	8.08e–04	4.84e–04	3.76e–04	3.50e–04	3.62e–04
5	5.99e–03	3.86e–03	3.14e–03	2.78e–03	7.64e–04
6	6.64e–02	4.50e–02	3.79e–02	3.42e–02	6.12e–03
7	9.36e–01	6.76e–01	5.81e–01	5.34e–01	8.91e–02
8	1.43e+01	1.03e+01	8.85e+00	8.17e+00	1.41e+00

Table 1: Total timings in seconds for all modules using subdivision levels $n = 3$ up to $n = 8$.

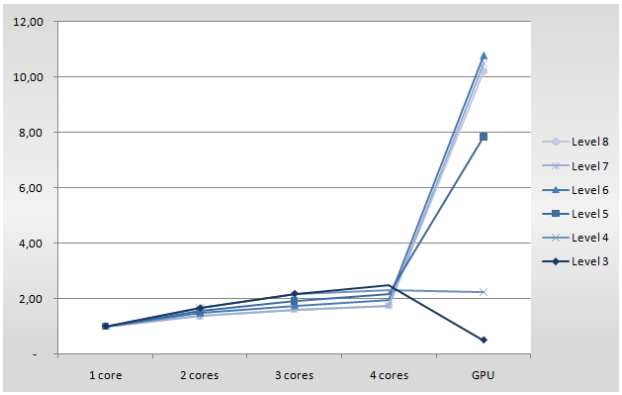

Figure 8: Total speedups for all modules using subdivision levels $n = 3$ up to $n = 8$.

declines as the subdivision level increases. At level $n = 8$, the advantage gained from increasing the number of active CPU cores from 3 to 4 is almost gone. On the GPU the result is quite the opposite. A low subdivision level implies an inefficient GPU due to many idle cores. From subdivision level $n = 6$ and up the task is large enough for the GPU to utilize most of its cores, resulting in a good speedup compared to the single-core CPU.

5.1 Subdivision Level

An important aspect of the setup is choosing an appropriate subdivision level. In our test system we may use a subdivision level up to $n = 10$, limited by the memory on the GPU. Even higher levels are possible, requiring us to use an iterative approach, running the overlap modules in multiple steps on a sub-grid. We do not want to use a high subdivision level on a trivial case as it would imply wasting clock cycles. But if the problem is an intersection with a complex topology, we are better off using a higher subdivision level. We would otherwise have to iterate on the solution, resulting in a lower efficiency due to added overhead from transferring data from the GPU for each step. Still, to achieve good data locality which is vital for an efficient cache usage, it may be beneficial to use a lower initial subdivision level, and then iterate on the solution without wasting time by checking the intermediate results.

In theory, assuming a perfect scalability of the multi-core CPU and the GPU with respect to the problem size, an increased subdivision level should give an increase in run-time for each of the modules according to their respective computational order. To evaluate the efficiency of the modules for the different subdivision levels, we divided the module execution time by 2^{2n} for the modules of order $O(2^{2n})$, and divided by 2^{4n} for the modules of order $O(2^{4n})$. This gives the sub-patch execution time at the various subdivision levels, which indicates whether an iterative strategy on the subdivision

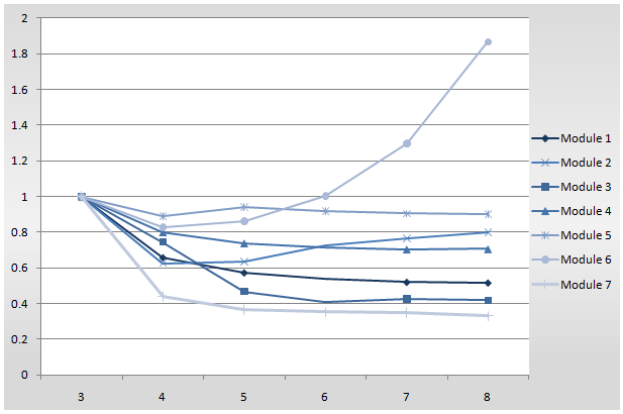

Figure 9: 4-core CPU efficiency at various subdivision levels, measured as a percentage of sub-patch execution time at level $n = 3$.

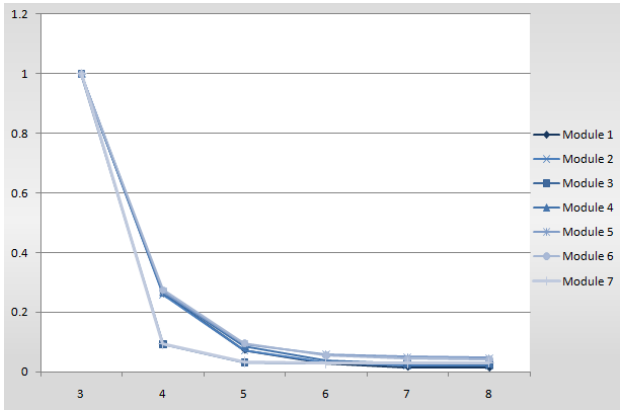

Figure 10: GPU efficiency at various subdivision levels, measured as a percentage of sub-patch execution time at level $n = 3$.

levels may be a good idea. Figure 9 and Figure 10 show the efficiency ratios with respect to level $n = 3$. On the CPU, using four cores, there was clearly a benefit from increasing the subdivision level from $n = 3$ up to $n = 5$, as can be seen in Figure 9. With module 3 and 7 being the most time-consuming modules there is a small advantage in using an even higher subdivision level. Figure 10 confirms the poor GPU performance on low subdivision levels which was shown in Figure 8. From level $n = 6$ and up to $n = 8$ the efficiency is almost the same. There was no additional benefit in further increasing the subdivision level.

5.2 The Intersection Modules

We take a closer look at each of the intersection modules, to see how they perform on the two computational resources. In accordance with the analysis in the previous section we have used subdivision level $n = 8$.

As illustrated in Figure 11, the surface refinement module scales nicely on the CPU. The input data are on a well-ordered layout, allowing an efficient OpenMP implementation. The GPU version of the module gives a very good speedup.

The bounding box module scales well on the CPU. Each Bézier sub-patch is treated by a separate core. With the sub-patch data stored in a continuous stream, we allow for OpenMP to achieve

Module	1 core	2 cores	3 cores	4 cores	GPU
1	1.16e–02	5.80e–03	3.88e–03	2.91e–03	6.87e–04
2	8.23e–03	4.15e–03	2.82e–03	2.15e–03	6.78e–04
3	6.09e+00	6.09e+00	6.10e+00	6.10e+00	6.61e–01
4	4.16e–02	2.08e–02	1.39e–02	1.04e–02	1.42e–03
5	2.38e–02	1.20e–02	8.06e–03	6.06e–03	2.91e–03
6	1.12e–01	5.75e–02	3.86e–02	3.04e–02	3.08e–03
7	7.98e+00	4.11e+00	2.69e+00	2.02e+00	7.36e–01
Sum	1.43e+01	1.03e+01	8.85e+00	8.17e+00	1.41e+00

Table 2: Timings in seconds for the modules using subdivision level $n = 8$.

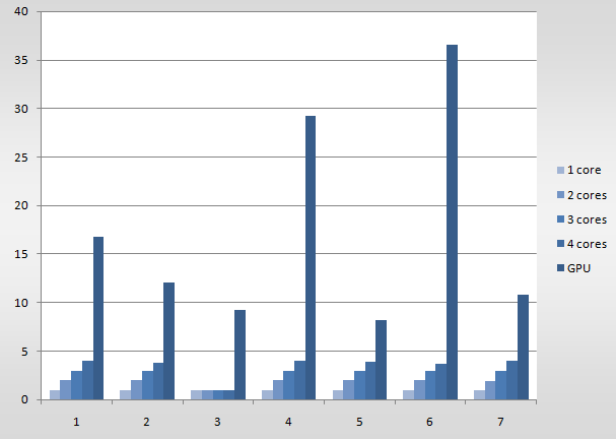

Figure 11: Module speedups using subdivision level $n = 8$, where the numbering is as follows: 1) Surface refinement, 2) Bounding box construction, 3) Bounding box overlap test, 4) Normal surface refinement, 5) Degeneracy test, 6) Normal cone construction, 7) Normal cone overlap test.

good cache efficiency. The number of arithmetic operations is too low for the GPU to utilize all of its computational power. A large part of the speedup is due to the faster GDDR5 RAM on the GPU.

In the box overlap test we noticed an interesting result on the CPU, with no speedup using more cores. The reason for this is that there are too few arithmetic operations carried out. As a result the memory bandwidth is the limiting factor. Due to OpenMP overhead the inclusion of more cores actually leads to a small speed decline. Although there are not enough computations to exploit the GPU, we still get a significant speedup due to the faster GDDR5 RAM.

Since the module for refining the normal surface is essentially the same as the one for refining the input surface, one should expect it to have the same speedup characteristics. The larger matrix system means that the number of arithmetic operations is higher, resulting in a slightly better GPU speedup when comparing to module 1.

The degeneracy test scales well on the CPU. The GPU is not that much faster. Although the size of the computational grid is large enough to utilize all the GPU cores, the number of arithmetic operations are too few to exploit the parallelity of each GPU core.

When generating the normal cone for each Bézier sub-patch, we perform quite many arithmetic operations. This scales well on the CPU, efficiently distributed using OpenMP. This module is perfectly suited for the GPU, giving a very good speedup. The number of GPU cores involved is high, with each core given enough arithmetic operations to utilize its parallelity. It is a well balanced GPU kernel, where all the sub-patches involves the same number of arithmetic operations.

The normal cone overlap test consists of only a few floating point operations for each pair of cones. Although a low number of operations, it is still high enough for the memory bandwidth not to become a limiting factor. This scales well on the CPU. Again the GPU benefits from the much faster GDDR5 RAM. In our first CPU implementation, where we for each normal cone stored the cone angle as opposed to storing the sine and cosine values of the angle, we had to compute the angle between each pair of cones. The slow inverse cosine operation performed on the CPU gave a GPU speedup of approximately 45. By locating the bottleneck in the CPU version and rethinking the mathematics this factor was drastically reduced. A necessary criteria for the cone overlap test to be of any interest is that the corresponding bounding boxes overlap. For most intersection cases the majority of the bounding box pairs will not overlap. Hence only a minority of cone pairs need to be checked for overlap. As a result the computational grid will be unbalanced, reducing the efficiency of the highly parallel GPU. For this reason it is rarely a good idea to run this module massively on all cone pairs. Usually the multi-core CPU will be the natural choice for this module.

5.3 Potential for Heterogeneous Speedup

Even though we must use one thread to control the heterogeneous system and one thread to control the GPU, the Core i7 CPU supports hyper-threading which lets us use all four CPU cores for accelerating the modules.

The speedup factors in Figure 11 show that the multi-core CPU is a good candidate for running module 2 (bounding box construction), module 5 (degeneracy test) and module 7 (cone overlap test). From the flowchart in Figure 5 we see that the GPU may run other modules at the same time. This shows a potential for a good speedup on the heterogeneous system. But if the total run-time is dominated by one or a few of the modules, there may not be much benefit from such an approach.

From Table 2 we see that module 3 and 7 (the overlap tests) are dominating the total run-time. By looking at the computational order of the modules this is to be expected. This means that using the multi-core CPU to run modules 2 and 5 will only give a small speedup. Even though the CPU scales well on module 7, it is evident from Table 2 that it is faster to let the GPU handle that module. By letting the CPU run modules 2 and 5 using all four cores, and let the remaining 5 modules run on the GPU, Table 2 yields a corresponding theoretical speedup of 10.2. Compared to 10.1 on the GPU alone, and taking into account some overhead cost, it is evident that this is not a good solution.

The approach which will give a significantly better speedup is to let the multi-core CPU contribute to the last module, the cone overlap test. That module consists of looking up values in two tables containing the normal cones corresponding to the refined normal surface and then check for overlap. These calculations are totally independent and are well suited to be split among different computational resources. The total heterogeneous system will then consist of the GPU running most of the modules. The multi-core CPU is set to handle module 5 (degeneracy test) and module 7 (cone overlap test), with the GPU joining in on module 7 as soon as it has completed the other modules. From Table 2, this will allow the CPU to complete approximately 33 % of module 7 before the GPU joins in. The combined speedup of the multi-core CPU and the GPU on module 7 is 14.8. The last 67 % of the module should then be done in 0.363 seconds. The theoretical total run-time is 1.03 seconds, with a corresponding heterogeneous speedup of 13.9. This speedup

is a bit optimistic since we in a real test case will not achieve a perfect load balancing on module 7, but the well-organized structure of the data should allow us to come quite close. Since the CPU has to wait for the output from module 4 we do not accomplish to let all the computational resources be active at all times, but the CPU idle time will only constitute a small fraction of the total run-time.

It should be noted that this heterogeneous speedup is referring to a synthetic test where module 7 was run with overlap testing on all the cone pairs. Since this will seldom be necessary the actual speedup will usually be smaller. Still, for those cases the multi-core CPU is even more competitive compared to the GPU for executing module 7.

6 Conclusions and Future Work

An essential part of a heterogeneous system is to analyze the strengths of the different computational elements. One should not only look for ways to split the problem into parallel modules, the modules should also have a good parallelity, preferably allowing them to be shared among multiple processing elements. In order to maximize the total computational power it is important to distribute each module to the architecture which is the most suited, or send tasks to a resource which will otherwise be idle. The results show that this heterogeneous system is well suited for this type of problem. The chosen architectures show a good scalability on most of the modules, with the parallel pipelines allowing an efficient usage of the available resources.

There is a clear trend towards an increased parallelity in modern commodity computers. An interesting concept, backed by AMD through their Fusion project, consists of a multi-core CPU and a GPU on the same chip. Such heterogeneous systems are expected to become increasingly more common in the future, allowing for a wide variety of computational problems to benefit from the diversity in the architecture. As an additional benefit there is a wider communication bus between the separate resources, reducing the overhead. The Cell Broadband Engine is another heterogeneous system. It has shown a great speedup for various tasks. It is used in the IBM Roadrunner supercomputer, which was the first computer to reach 1.0 petaflops.

Although chips with a heterogeneous architecture are gaining attention, there will still be room for specialized entities. Recently NVIDIA announced the Fermi GPU. With that a lot of interesting opportunities have opening up. The GPU is no longer restricted to running only one kernel at a time. It may run up to 16 simultaneous kernels. For our heterogeneous intersection approach this can have a huge impact. As visualized in the flowchart in Figure 5 this will allow the Fermi GPU to execute multiple intersection modules at the same time, possibly operating at different subdivision levels. This will imply an increased GPU efficiency, which may otherwise be restricted by the relatively small size of the intersection problem.

We are planning to set up a heterogeneous intersection system using this acceleration of sub-algorithms in combination with our recursive intersection algorithm. It seems like a good idea to do a fast preprocessing on the CPU using a low subdivision level, and then use the accelerated intersection modules on computationally demanding problems. The results shown in this paper are promising with respect to a good total speedup of the complete intersection algorithm.

References

AGARWAL, P. K., KRISHNAN, S., MUSTAFA, N. H., AND VENKATASUBRAMANIAN, S. 2004. Streaming geometric optimization using graphics hardware. In *Algorithms - ESA 2003*, G. Di Battista and U. Zwick, Eds. Springer Verlag, Berlin Heidelberg New York, 544–555.

BLOOMENTHAL, J., AND WYVILL, B., Eds. 1997. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

BRISEID, S., DOKKEN, T., HAGEN, T. R., AND NYGAARD, J. O. 2006. Spline surface intersections optimized for gpus. In *Computational Science ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, Springer, Berlin / Heidelberg, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., vol. 3994 of *Lecture Notes in Computer Science*, ICCS, 204 – 211.

BRODTKORB, A., DYKEN, C., HAGEN, T. R., HJELMERVIK, J., AND STORAASLI, O. 2010. State-of-the-art in heterogeneous computing. *Scientific Programming*.

CHAPMAN, B., JOST, G. J., AND VAN DER PAR, R. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press.

DOKKEN, T., AND SKYTT, V. 2007. Intersection algorithms and cagd. In *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*, G. Hasle, K.-A. Lie, and E. Quak, Eds. Springer Verlag, Berlin Heidelberg New York, 41–90.

DOKKEN, T., AND THOMASSEN, J. B. 2003. Overview of approximate implicitization. In *Topics in algebraic geometry and geometric modeling*, vol. 334. Amer. Math. Soc., Providence, RI, 169–184.

DOKKEN, T. 1985. Finding intersections of b-spline represented geometries using recursive subdivision techniques. *Comput. Aided Geom. Design 2*, 189–195.

FARIN, G. 2002. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

HOFF, III, K. E., ZAFERAKIS, A., LIN, M., AND MANOCHA, D. 2001. Fast and simple 2d geometric proximity queries using graphics hardware. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 145–148.

KIRK, D. B., AND HWU, W.-M. W. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.

KRISHNAMURTHY, A., MCMAINS, S., AND HALLE, K. 2009. Accelerating geometric queries using the gpu. In *SPM '09: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, ACM, New York, NY, USA, 199–210.

OWENS, J., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J., AND PHILLIPS, J. 2008. Gpu computing. *Proceedings of the IEEE 96(5)*, 879–899.

SINHA, P., KLASSEN, E., AND WANG, K. 1985. Exploiting topological and geometric properties for selective subdivision. In *Symposium on Computational Geometry*. ACM Press, 39–45.

VIRO, O. 2008. From the sixteenth Hilbert problem to tropical geometry. *Japanese Journal of Mathematics 3*, 185–214.