FORTH-ICS / TR-420 July2011

A GPU-powered computational framework for efficient 3D model-based vision†

Nikolaos Kyriazis, Iason Oikonomidis, Antonis A. Argyros

[0]† This work was partially supported by the IST-FP7-IP-215821 project GRASP.

A GPU-powered computational framework for efficient 3D model-based vision

Nikolaos Kyriazis, Iason Oikonomidis, Antonis A. Argyros

Computational Vision and Robotics Laboratory
Institute of Computer Science
Foundation for Research and Technology — Hellas (FORTH)
N. Plastira 100, Vassilika Vouton
Heraklion, Crete, 700 13 Greece

Web: `http://www.ics.forth.gr/cvrl`
E-mail: {kyriazis | oikonom | argyros}@ics.forth.gr
Tel: +30 2810 391600, Fax: +30 2810 391601

Technical Report FORTH-ICS / TR-420— July2011

Abstract

We present a generic computational framework that exploits GPU processing to cope with the significant computational requirements of a class of model-based vision problems. We study the structure of this class of problems and map the involved processes to contemporary GPU architectures. The proposed framework has been validated through its application to various instances of the problem of model-based 3D hand tracking. We show that through the exploitation of this framework near real-time performance is achieved in problems that are prohibitively expensive to solve on CPU-only architectures. Additional experiments performed in various GPU architectures demonstrate the scalability of the approach and the distribution of the execution time among the involved processes.

1 Introduction

In computer vision, several problems are solved by employing search and optimization methods. For example, in top-down model-based approaches, model instantiations are searched for or fitted in observations, in specific contexts. In this work we are interested in the class of top-down methods that estimate the pose and/or track the articulation of piecewise rigid objects, based on multiple visual cues, such as color images, depth maps etc. Depending on the employed model, the problem can be as simple as the recovery of the 3D position and pose of a rigid object or as complex as the tracking of articulated objects such as the human body or hand.

This broad class of problems is amenable to a generic formulation. For a given object model, candidate hypotheses can be generated. Given a process that quantifies the compatibility of a hypothesis to the observations and a systematic approach for generating the hypotheses, the problem can be solved by searching for the best scoring hypothesis. The scoring process ultimately reduces to comparing model hypotheses and observations at the pixel level, on a pre-defined feature space. The major drawback of this generic formulation is its computational requirements stemming from the significant processing associated to the scoring criterion as well as to the evaluation of mutliple hypotheses. Thus, the definition of an efficient computational framework that addresses the computational requirements of related methods is very important to their deployment in real-world applications. In this work, we propose such a computational framework that is based on the careful mapping of the involved processes to GPU architectures.

2 Relevant Work

Many computer vision tasks are susceptible to GPU acceleration. This is well understood and has lately been availed by a number of researchers, who have proposed efficient solutions for a variety of vision problems. For example, Choudhary et al. [1] were able to accelerate bundle adjustment and perform faster large-scale 3D reconstruction through a CPU/GPU architecture. Fulkerson and Soatto [5] provided a CUDA-based GPU implementation of the exact quick shift algorithm that enables $10 - 50$ times faster image segmentation compared to a CPU implementation. Gwosdek et al. [7] performed fast variational optic flow computations by accelerating the employed Fast Explicit Diffusion Solver on the GPU using CUDA. Stühmer et al. [17] accelerated a Generalized Thresholding Scheme using CUDA, in order to also perform variational optic flow, in real-time. Tzevanidis et al. [18] constructed textured 3D meshes from multicamera observations in real time, by means of an efficient CUDA-based 3D visual hull computation. Zhu et al. [20] studied GPU accelerated dense stereo computations and demonstrated that both accuracy preservation and speed increase can be achieved in a transition from CPU-based to GPU-based algorithms.

GPU-based solutions have also been proposed for instances of the problems of pose estimation and articulated object tracking, that is the class of problems relevant to this work. De La Gorce et al. [2] employed a realistic textured 3D model of a human hand in order to track it in image sequences. GPU acceleration came in the form of a straightforward rendering mechanism that was invoked in a per hypothesis basis. Hammer et al. [8] took GPU acceleration further by considering multiple hypotheses simultaneously in tiled renderings, again with respect to 3D hand tracking. Ganapathi et al. [6] followed a similar approach for the problem of 3D human body pose estimation. Although brief, the described GPU acceleration raises some important issues that are also addressed in this work. Friborg et al. [4] reported a detailed CUDA-based implementation of an inherently parallel likelihood computational scheme, in the context of articulated object tracking.

Despite the significant amount of existing work, there has not been a generic CPU/GPU framework that considers a class of problems rather than a single problem instance. Thus, in this work, we systematically study the structure of a class of 3D model based vision problems and we propose a generic CPU/GPU framework for addressing their significant computational requirements. In spite of the common trend, we select a GPU-independent software architecture, Direct3D, that makes the application of our framework GPU invariant and, simultaneously, demonstrates that former GPU software pipelines are not obsolete. Still, we do consider the potential benefits from graphics-free GPU architectures, such as CUDA and OpenCL and we provide a brief discussion on their applicability and merits to our problem. The usefulness of the adopted approach is documented through experimental results obtained from the application of the proposed framework to various instances of the problem of 3D hand tracking [12–14] which show that the proposed framework achieves almost real-time performance in this type of problems. Additional experiments performed in various GPU architectures demonstrate the scalability of the approach and the distribution of the execution time among the involved processes.

3 Methodology

The high-level outline of the proposed framework is illustrated in Fig. 1. We assume that an *observation* process that is executed on the CPU, feeds the framework with raw input data. These data (acquired from a single or a multiple camera system and/or other sensing modalities) are then transformed into visual cues, that are relevant to the task at hand, through *preprocessing* (e.g., background subtraction, edge detection). Given the resulting visual cues,

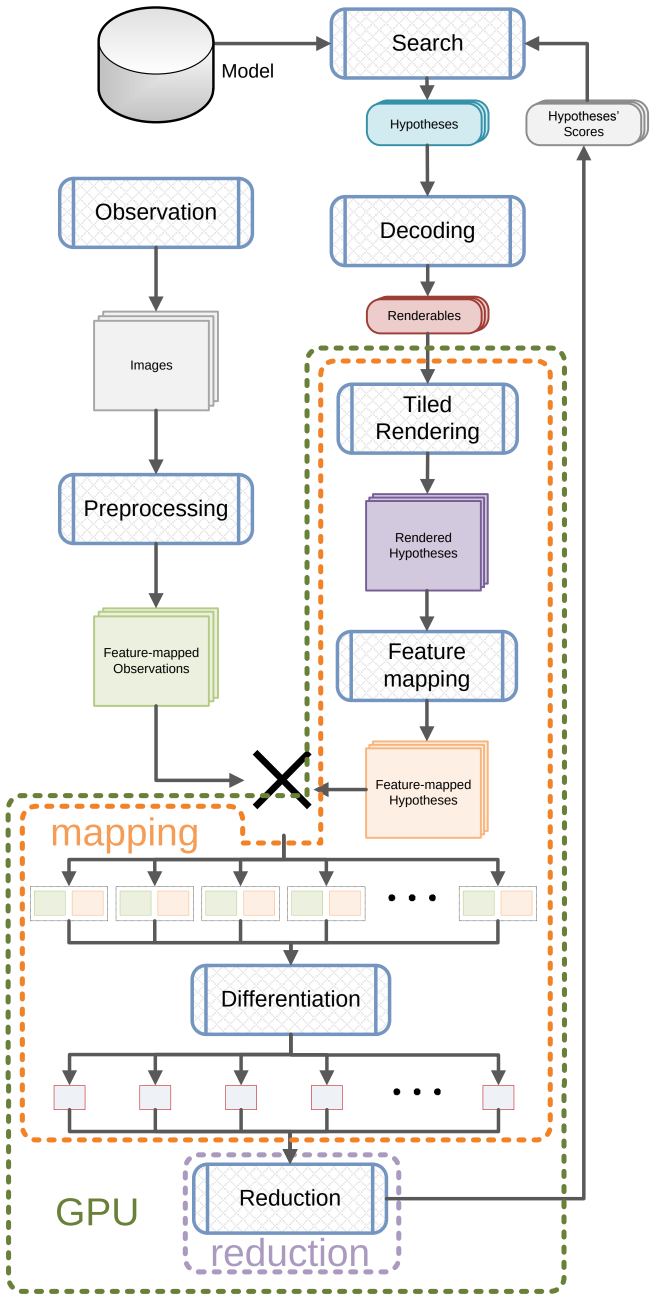


Figure 1: The proposed computational framework. Every iteration of the defined loop amounts to an efficient differentiation of observations and hypotheses, in a top-down, model-based approach.

a *search* process iteratively estimates the parameters of a preselected *model* by testing the compatibility of various hypothesized model instances to the visual cues. The hypothesized model instances are *rendered* and then *feature-mapped*, so as to be comparable to the observations at the pixel level. During each iteration of the *search* process, hypotheses rendering is performed on a big tiled map by a *tiled rendering* process. Since the observed visual cues and the rendered model hypotheses are comparable at the pixel level, their *differentiation* and *reduction* results in a total observation-hypothesis distance. All computed distances are then exploited by the *search* process so that better hypotheses can be further explored. The whole process terminates as soon as a termination criterion (usually related to the achieved accuracy and/or a predefined number of iterations) has been met.

3.1 Accelerated vs non accelerated processes

From the briefly described processes, *observation* is performed on the CPU and *preprocessing* can be executed either on the CPU or on the GPU. It has been observed [12–14] that these processes only consume a small fraction of the total execution time and therefore their acceleration does not have a significant impact on the overall computational performance.

We consider the *Black Box Optimization* paradigm [9], according to which a *search* process/heuristic investigates the hypothesis space of a *model* in order to identify the hypothesis that optimally fits a set of observations. There is only marginal gain in accelerating the core of such processes, as they are usually very fast. However, it is crucial to accelerate the hypothesis evaluation phase, whose execution time is dominant. Search heuristics can be categorized into serial and parallel, according to their evaluation scheme [16]. Serial heuristics are restricted to evaluate a single hypothesis at a time, while parallel heuristics do not pose such restrictions. Parallel heuristics allow for great acceleration gains in their objective function evaluation phase.

After sets of hypotheses have been proposed, they are decoded into renderable entities, i.e. arbitrarily complex 3D geometric instances. *Decoders* are responsible for mapping a multi-dimensional search space into a decomposition over geometries and their transformations (e.g., kinematics). In the proposed architecture decoders are dynamic libraries, that are loaded and selected at run-time. Decoding is worth accelerating since it involves series of matrix multiplications (geometry transformations).

The described iterative process follows the *map-reduce* scheme [3]. During the *mapping* phase, hypotheses are generated and paired to the corresponding observations (cross product notation in Fig. 1). At the *reduction* phase, the differences of the defined pairs are reduced into hypotheses scores that guide the *search* process. Both phases are GPU accelerated. Since the architecture we propose is generic, in the following sections we only describe the design requirements for each of the involved processes, emphasizing issues related to the GPU acceleration and data communication. Still, in Sec.4, we also provide some details on how the proposed framework is instantiated to efficiently solve various instances of the problem of 3D hand tracking.

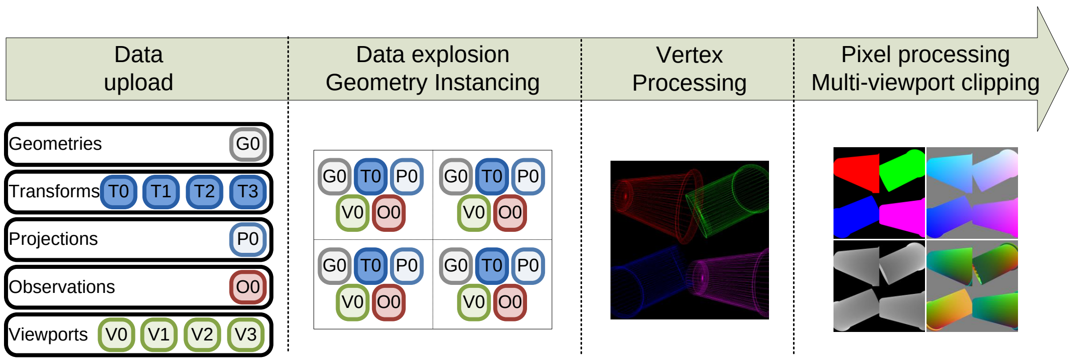


Figure 2: The tiled rendering process. Unique data are uploaded to the GPU, exploded into a tiled plan, processed in the vertex level and output in primary maps for later processing. Although there might be overlap of projected geometry across tiles during vertex processing this is remedied at the pixel-processing stage.

3.2 Tiled rendering

This process receives renderable entities and produces a set of primary outputs for further processing. Renderable entities amount to 3D geometries and geometry transformations and the primary outputs represent view-space 3D information, i.e. per pixel color, 3D position, 3D normal and depth. The input is provided by the *decoder* process. The outputs are post-processed by *feature mapping* and are thus made directly comparable to the respective observations.

Contemporary GPU drivers, GPU architectures and rendering pipelines allow for surprisingly fast parallel processing of massive data. Given proper design, linear increase of data may induce sub-linearly increased execution times, thanks to efficient interleaving of processing/reading/writing instructions [11]. We take parallelization to the limit by processing multiple hypotheses simultaneously. Instead of rendering a single hypothesis at a time we render multiple hypotheses in big *tiled renderings*. We separate the rendering and feature mapping phases in favor of modularity, by employing *deferred shading*. We perform efficient data communication and rendering by employing *geometry instancing*. Proper containment of instantiated geometry in tiles is achieved through *multi-viewport clipping*.

3.2.1 Deferred shading

Deferred shading is a technique commonly used in computer graphics [15]. According to this technique, 3D models are rasterized into a series of *primary outputs* such as color, position, depth and surface normal maps. Essentially, each primary output is an image with each pixel holding 3D information for the rendered models. It is characterized as "deferred" because actual rendering is postponed and at this phase only primary output is produced. An example of *deferred shading* is illustrated in Fig. 3. We employ *deferred shading* for two reasons: (a) we need to isolate the feature mapping process from the rest of the processes for the sake of modularity, (b) primary data may require multiple passes of processing but should not be

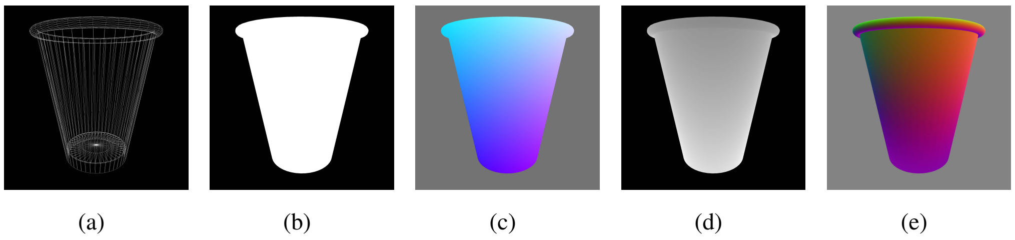


Figure 3: Deferred Shading. (a) the input of 3D model to be rendered and the primary outputs: (b) color, (c) position, (d) depth and (e) normal maps.

multiply computed.

Deferred shading, as a conventional rendering process, involves two successive stages, namely *vertex shading* and *pixel shading*. Given 3D models and view parameters, vertex shading is responsible for transforming every vertex of the input geometry from world coordinates to view coordinates. Then, triangles that are defined by the transformed vertices are rasterized during pixel-processing. Additionally, z-buffering is used in order to produce correct rasterizations of occluded geometry. Each stage is configurable through the integration of appropriate *vertex* and *pixel shaders*, i.e. callback routines, that perform the required processing on the GPU. We implement such shaders in the proposed framework.

3.2.2 Geometry instancing

It is beneficial to reduce communication between the CPU and the GPU in order to eliminate the respective overhead in execution time. For several computational schemes it is common that transferring data across memory spaces is more costly than processing itself. Best performance is achieved when only a few data are transferred to GPU, which are then "exploded" and processed and then "imploded" and transferred back to CPU.

In this work, each tile in a tiled rendering is associated with an actual camera view and a hypothesis. Therefore, for each tile, the following information is required: (a) geometry to be rendered, (b) world transform of this geometry, (c) projection matrix for the actual view and (d) the coordinates of the tile's viewport in the big rendering. However, in a tiled rendering, multiple tiles might refer to the same geometry and/or the same camera view. Therefore, there can be a lot of data reuse in the computations. This is amplified in the quite common case where the geometry itself is so modular that its sub-parts are heavily reused as well (e.g. as in [12], where the hand model consists of appropriate transforms of two geometric primitives). In that case it is highly inefficient to replicate data wherever they are required. Fortunately, in newer *shader models* (3 and above), *hardware instancing* enables an efficient and implicit replication of heavily reused data. Thus, we only upload unique data to the GPU once and explode them during processing by means of indexed referencing. The indexing itself is also implicit as we only define a replication pattern rather than the indices themselves (e.g., repeat datum every n tiles). The corresponding implosion occurs during the reduction phase (Sec. 3.5).

3.2.3 Multi-viewport clipping

If multiple tiles are to be rendered simultaneously it must be guaranteed that the rendered geometry is properly contained therein. That is, we want to avoid rendering the geometry of a given tile over neighboring tiles. However, conventional rendering pipelines do not consider this special case. While geometry clipping is traditionally performed at the vertex processing stage it is complicating to do so in our case. This is because of irregularities in the amount of data that survives clipping which would require an intricate remedy with respect to our target platform.

We chose to perform multi-viewport clipping at the pixel-processing stage where it is both convenient and efficient. After all geometry has been rasterized, a custom pixel shader is invoked to produce the primary map outputs. During geometry instancing, viewport information is attached to every vertex (see Sec. 3.2.2). During rasterization, this information is transferred to the pixels that result from the triangles formed from the processed vertices. Therefore, at pixel-processing each pixel is associated with the viewport in which it should be contained. A custom pixel shader clips pixels that are outside their pre-defined viewports (see Fig. 2).

3.3 Feature mapping

The main task of this process is to make rendered hypotheses comparable to the observations. This is performed by post-processing the 3D information that is contained in the primary outputs of the rendering process. Exemplar *feature mapping* processes can be: (a) the computation of occupancy from the position map, (b) the computation of edges from the normal map, (c) the computation of discrete layers from the depth map, etc. *Feature mapping*, as a straightforward rendering step, is a highly parallel pixel-wise mapping of the primary maps.

3.4 Differentiation

Differentiation is used to quantify the discrepancy between all feature-mapped observations and the corresponding feature-mapped hypotheses. With geometry instancing (Sec. 3.2.2) already one part of the observation-hypothesis pairing has been computed: every hypothesis has been associated to a part of a big tiled rendering which constitutes one operand of the differentiation. The remaining operand needs to be computed in accordance with the aforementioned pairing.

All observations are uploaded to the GPU in the form of multi-channel real-valued 2D textures at appropriate predefined slots. Each texture is assigned to a zero-based slot index that also constitutes its reference. During instancing this information is passed to each tile, so that the contained hypothesis corresponds to an observation. Thus, two big GPU textures are defined: (a) an explicit texture that holds the tiled rendering (first operand) and (b) an implicit tiled texture that is defined by the observation textures (second operand). Those two textures are now susceptible to any pixel-wise differentiation process that quantifies the discrepancy between observations and hypotheses. The result of this differentiation is stored in a multi-channel real-valued 2D texture.

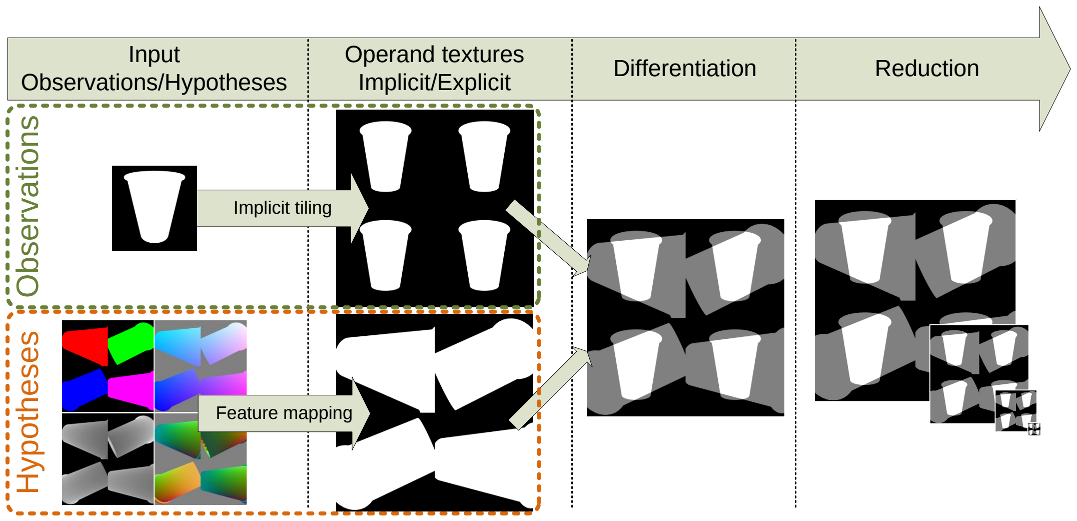


Figure 4: The differentiation process. Primary maps are mapped to the observations' feature space. Observations are implicitly tiled so as to match the tiled rendering of all hypotheses. A pixel wise differentiation is applied and the result is finally summed over the logical tiles by means of subsampling (data implosion).

3.5 Reduction

Once the mapping phase has been completed, the pixel-wise differences (Sec. 3.4) must be reduced to a single value for each tile, that ultimately represents the corresponding hypothesis' distance from the respective observation. We perform this reduction by means of pyramidal computations. Reduction is performed iteratively by subsampling the difference texture with a reduction operator (most commonly the sum operator). This scheme corresponds to the computation of a given MIP (*Multo In Parvo*, i.e., "much in a small space") level of the differences texture [19]. Reduction stops at the level at which the dimensionality of the resulting texture matches the dimensionality of the hypotheses' grid. For the sake of efficiency we select square render and tile sizes that are powers of two. For this process two textures are used that are exchanged in the positions of the input and the output in every subsampling iteration.

4 Applications

The proposed framework has been employed in [12–14], where different instances of the 3D hand tracking problem are treated. In order to exemplify the use of the proposed CPU/GPU architecture, we select the works in [14] and [13] because they differ in the observation model, the employed visual cues and, consequently, in the differentiation and reduction phases. Thus, they are the most diverse from a computational point of view. In both works different variations of the PSO algorithm [10] have been employed as the *search* process. PSO follows a parallel evaluation scheme which is favorable to the employment of the presented framework. The *rendering* process is the same in both cases, thanks to the *decoder* process.

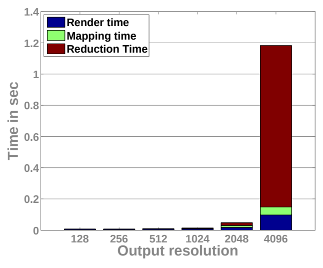
In [14], the *model* accounts for a 3D articulated human hand and a parametric 3D object (ellipsoid, cylinder, cuboid) that amount to a total of $34-35$ DoFs. The observation process produces multi-frames of up to 8 RGB images per invocation. During preprocessing, input multi-frames are transformed into multi-frames of skin detection results m_s and multi-frames of edge detection results m_{DT} that have undergone a Distance Transform. The decoding phase maps the $34-35$ DoFs (depending on the number of DoFs of the object model) to appropriately transformed instances of ellipsoids, cylinders and cuboids. The results of the rendering process are feature-mapped to (a) occupancy images m_c that are produced from the primary color map and correspond to skin detection results and (b) edge detection results m_e, that are produced from the primary normal map and correspond to the edge detection results. The differentiation process generates 4 outputs: (a) $m_s \vee m_c$, (b) $m_s \wedge m_c$, (c) m_e and (d) $m_{DT} \cdot m_e$, where all operations are pixel-wise. Each such output is then reduced, in a per-tile basis, in order to provide tuples of 4 values for each hypothesis, using the sum operator. Each tuple is transformed (more details in [14]) into a single value that represents the distance between a given observation and a hypothesis. The employment of our framework in this application induces a performance of $2fps$ on an Intel Core i7 950 @ $3.07GHz$ with a NVIDIA GeForce GTX 580 GPU. Sample video results are available at `http://youtu.be/N3ffgj1bBGw`.

In [13], the *model* accounts for a 3D articulated hand of 26 DoFs (6 for the hand global position and pose and 20 for hand articulation). The observation process generates a frame set that is composed of a RGB image m_{RGB} and a depth map m_D. During preprocessing, skin detection is applied to m_{RGB} and its result is used to filter out depths, that do not regard the observed hand from m_D, in a new depth map m'_D. The decoding phase maps the 26 DoFs to appropriately transformed instances of ellipsoids and cylinders. The depth primary map that results from the rendering process is subtracted from m'_D, and these differences are summed over each tile to provide the observation-hypothesis distances. The employment of our framework in this application induces a performance of $15fps$ on an Intel Core i7 950 @ $3.07GHz$ with a NVIDIA GeForce GTX 580 GPU. Sample video results are available at `http://youtu.be/Fxa43qcm1C4`.

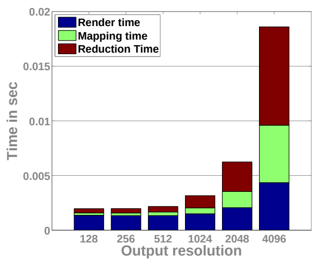
5 Experiments

The accuracy and computational performance of the proposed framework has been evaluated already in the context of the computer vision problems in which it was employed [12–14]. In this paper, further experiments were designed to highlight the strengths and weaknesses of this framework. More specifically, we show that our framework is able to perform in the order of tens of thousands evaluations of complex 3D hypotheses per second, even on mediocre hardware. Moreover, we identify a bottleneck in the pixel-processing stages, which leaves room for further improvement in performance.

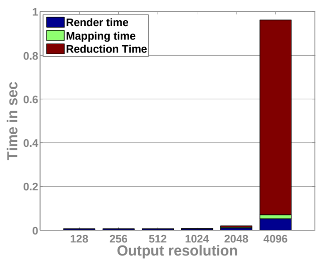
We performed a set of experiments on 3 distinct systems of varied computational power: (a) an Intel Core 2 6600 @ $2.4GHz$ with a NVIDIA GeForce 9600 GT GPU, (b) an Intel Core i7 950 @ $3.07GHz$ with a NVIDIA GeForce GTX 580 GPU and (c) an Intel Core i7 930 @ $2.8GHz$ with a NVIDIA GeForce GTX 295 GPU. The third system has a dual GPU



(a)



(b)



(c)

Figure 5: Performance profiling of the GPU evaluation. Batches of 64 3D hand hypotheses are evaluated over output textures of increasing dimensions. Figures (a), (b) and (c) correspond to systems (a), (b) and (c). Interestingly, in (a) and (c) and for a resolution of 4096×4096, 87.46% and 92.69% of the time is spent on reduction. This suggests that, for these cases, smaller batches are preferable.

but for the experiments only one of them was used. The three systems vary in GPU core count (96 for system (a), 512 for (b) and 240 for (c)) and represent different GPU generations.

During all experiments we used a geometry that represents an articulated hand. The hand consists of 22 homogenous transformations of a spherical mesh and 15 homogenous transformations of a cylindrical mesh. Each sphere consists of $2 \cdot st \cdot sl$ triangles and each cylinder consists of $2 \cdot st \cdot sl + 2 \cdot sl$ triangles, where sl represents slice count and st represents stack count in a geometry sampling grid. Unless otherwise stated, the values of $sl = 10$ and $st = 10$ were used, so, the total triangle count per hypothesis was 7700. For more details the reader is referred to [12]. To consider a challenging scenario regarding rendering computational requirements, wherever a renderable hypothesis was required, one supplied with a hand fully occupying the viewport (open hand facing a virtual camera). For observations we used synthesized renderings of the same hypothesis (the content of the observations does not influence the computational performance). For the mapping and reduction processes, routines similar to those in [12] were used. Each reported measurement is the mean value of 20 successive iterations of the associated process. The timing operations themselves affect performance negatively as they define multiple synchronization points that break pipelining.

In a first line of experimentation we tested the method's computational efficiency under increasing resolution requirements for the tiled rendering, and thus put more weight on the pixel-processing phase. Batches of 64 hypotheses were rendered on a 8×8 grid and on square textures whose dimension was 128, 256, 512, 1024, 2048 and 4096. Respectively, the dimension of the square tile was 16, 32, 64, 128, 256 and 512. We considered the execution times of *rendering*, *mapping* and *reduction* separately. A total of $64 \times 7700 = 492800$ triangles were rendered in each batch. The obtained results are illustrated in Fig. 5. It can be verified that as the resolution increases, the processing time of the pixel-processing stages (rasterization during rendering, mapping, reduction) increases, too. For higher resolutions, reduction becomes the bottleneck, followed by mapping. The graceful degradation of performance for system (b) and for a resolution of 4096, as opposed to systems (a) and (c), is most probably due to

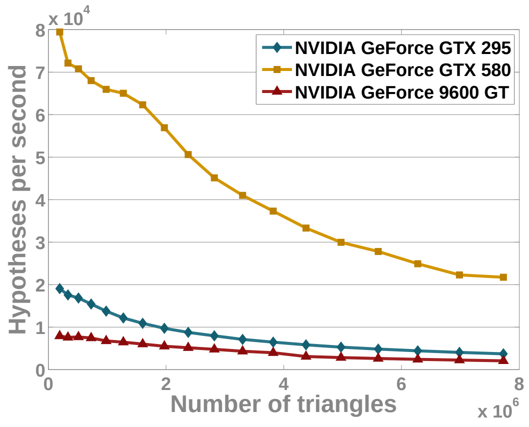


Figure 6: Rendering throughput against geometry complexity. 256 3D hand hypotheses are rendered on a 4096×4096 texture. Although 3D model detail increases with respect to triangle count, performance degradation is graceful due to hardware instancing.

improved memory architecture.

We also tested the rendering performance for increasing complexities of the 3D geometry and thus put more weight in the vertex processing phase. We repeated the previous line of experimentation, excluding mapping and reduction and fixing the tile dimension to 256 and the total dimension to 4096 (hypothesis count is thus 256). We varied the sl and st counts simultaneously in the range $[3, 20]$. This resulted in batches of triangle counts in the range $[193536, 7731200]$. Plots that demonstrate the graceful degradation of the performance are shown in Fig. 6. As the detail of the model increases, more vertices are required to be processed. Also, the tight spacing between small triangles in high levels of detail stresses the rasterization process, which has to also resolve more depth conflicts. Because of the simultaneous issuing of all geometry and due to efficient interleaving, the GPU performs well.

Finally, we tested the scalability of the proposed framework in order to highlight the benefit from considering batches of hypotheses simultaneously. We fixed the dimension of the tile to 128 and considered batches of 1, 4, 16, 64, 256 and 1024 hypotheses. This generated a requirement for output textures of dimensions 64, 128, 256, 512, 1024, 2048 and 4096, respectively. For each batch size we measured the hypothesis evaluation throughput by dividing its size with the required processing time. The results are shown in Fig. 7. Given the requirement for the evaluation of many hypotheses, it is evidently beneficial to consider larger batches for simultaneous evaluation. However, as the batch sizes grow, the required resolution is increased and the whole process becomes pixel-processing bound. Systems (a) and (c) were affected negatively, but system (b) remained unaffected.

In order to cope with batch sizes that are not power-of-two we processed them by sequential decomposition in power-of-two batches. The results of the previous experiment but for non-power-of-two batch sizes are shown in Fig. 8. Performance presents a pattern across sizes, due to the decomposition, and remained good for batch sizes that required few decom-

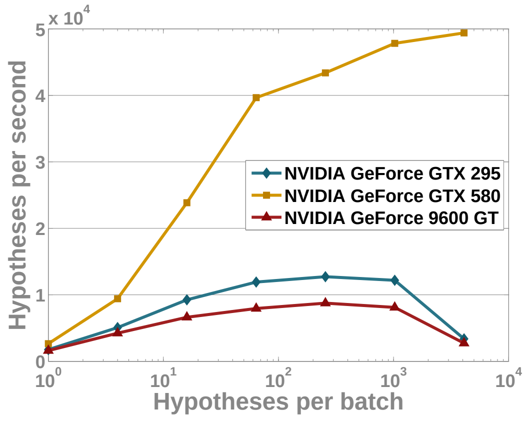


Figure 7: Evaluation throughput against batch sizes. Each hypothesis is evaluated on a 64×64 tile. As the batch size increases the evaluation throughput increases as well. The deterioration of performance for 1024 hypotheses for systems (a) and (c) is due to the requirement for an output texture of 4096×4096, which, as shown in Fig. 5, is problematic.

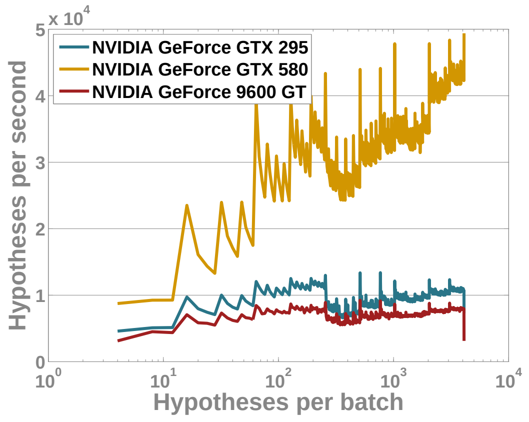


Figure 8: The same experiment as in Fig. 7, but for batch sizes that are not powers of two.

positions. State changes, like the requirement for different sizes of output textures in the same batch, affected performance negatively.

In every case, we were able to evaluate tens of thousands of complex 3D hypotheses per second. This has been most beneficial for the works in [12–14], where near real-time performance was achieved for different challenging instances of the problem of 3D hand tracking.

6 Conclusions

In this paper we provided a detailed description of a CPU/GPU framework that targets a class of computer vision problems. We demonstrated the computational benefits of this framework in a series of experiments performed on various GPU systems. The proposed framework is not restricted to a single application and has already been employed in three similar, yet essentially different problems. It has been shown that an effective GPU-invariant solution is realizable. More specifically, it has been demonstrated that the proposed framework can evaluate tens of thousands of elaborate 3D hypotheses per second. The efficiency of the proposed architecture makes it a likely candidate for several other real-time 3D computer vision applications.

Still, there is room for further improvements. We selected Direct3D as the rendering platform because the class of problems we are interested in optimally maps to the features provided by this platform. However, by also being graphics-oriented, the selected platform induces restrictions and limitations: (a) Direct3D, although GPU-invariant, is not OS-invariant, (b) data-parallel, non-graphics-oriented tasks have to be forced through a graphics-oriented pipeline, which induces unnecessary overheads, (c) numbers need to be powers-of-two in order to achieve optimal performance in a platform that does not favor elaborate gather/scatter operations. Preliminary tests have shown that further speedup can be achieved by complementing or replacing Direct3D with other platforms, such as CUDA, OpenCL etc. For example, a $20\times$ speed-up on the pixel-processing stages (Fig. 5), which was achieved through CUDA, might offer an overall $2-10\times$ speed-up for large resolutions (see Fig. 5).

Acknowledgments

This work was partially supported by the IST-FP7-IP-215821 project GRASP.

References

[1] Siddharth Choudhary, Shubham Gupta, and PJ Narayanan. Practical time bundle adjustment for 3d reconstruction on the gpu. In *ECCV'2010 Workshop on Computer Vision on GPUs (CVGPU2010)*, 2010.

[2] M. de La Gorce, N. Paragios, and DJ Fleet. Model-based hand tracking with texture, shading and self-occlusions. In *CVPR 2008)*, pages 1–8. IEEE, 2008.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] R.M. Friborg, Sø ren Hauberg, and Kenny Erleben. GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking. In *ECCV'2010 Workshop on Computer Vision on GPUs (CVGPU2010)*, 2010.

[5] Brian Fulkerson and Stefano Soatto. Really quick shift: Image segmentation on a GPU. In *ECCV'2010 Workshop on Computer Vision on GPUs (CVGPU2010)*, 2010.

[6] V. Ganapathi, C. Plagemann, D. Koller, and S. Thrun. Real time motion capture using a single time-of-flight camera. In *CVPR 2010*, pages 755–762. IEEE.

[7] Pascal Gwosdek, Henning Zimmer, Sven Grewenig, A. Bruhn, and J. Weickert. A highly efficient GPU implementation for variational optic flow based on the Euler-Lagrange framework. In *ECCV'2010 Workshop on Computer Vision on GPUs (CVGPU2010)*, 2010.

[8] H. Hamer, K. Schindler, E. Koller-Meier, and L. Van Gool. Tracking a hand manipulating an object. In *ICCV 2009)*, pages 1475–1482. IEEE, 2009.

[9] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík. Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 1689–1696. ACM, 2010.

[10] J. Kennedy. Swarm intelligence. *Handbook of Nature-Inspired and Innovative Computing*, pages 187–219, 2006.

[11] Nvidia. Compute unified device architecture programming guide. *NVIDIA: Santa Clara, CA*, 83:129, 2007.

[12] I. Oikonomidis, N. Kyriazis, and A. Argyros. Markerless and efficient 26-dof hand pose recovery. In *ACCV 2010*, pages 744–757. Springer, 2010.

[13] I. Oikonomidis, N. Kyriazis, and A. Argyros. Efficient model-based 3d tracking of hand articulations using kinect. In *BMVC 2011*. BMVA, 2011.

[14] I. Oikonomidis, N. Kyriazis, and A. Argyros. Full dof tracking of a hand interacting with an object by modeling occlusions and physical constraints. In *ICCV 2011*. IEEE, 2011.

[15] M. Pharr and R. Fernando. Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation. 2005.

[16] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[17] J. Stühmer, Stefan Gumhold, and Daniel Cremers. Parallel Generalized Thresholding Scheme for Live Dense Geometry from a Handheld Camera1. In *ECCV'2010 Workshop on Computer Vision on GPUs (CVGPU2010)*, 2010.

[18] K. Tzevanidis, X. Zabulis, T. Sarmis, P. Koutlemanis, N. Kyriazis, and A. Argyros. From multiple views to textured 3d meshes: a gpu-powered approach. In *ECCV'2010 Workshop on Computer Vision on GPUs (CVGPU2010)*, pages 5–11, 2010.

[19] L. Williams. Pyramidal parametrics. In *ACM SIGGRAPH Computer Graphics*, volume 17, pages 1–11. ACM, 1983.

[20] Ke Zhu, Matthias Butenuth, and Pablo Angelo. Comparison of Dense Stereo using CUDA. In *ECCV'2010 Workshop on Computer Vision on GPUs (CVGPU2010)*, 2010.