Joint Mobile-Cloud Video Stabilization

Gbolahan S. Adesoye
University of California, Santa Cruz
gadesoye@ucsc.edu

Oliver Wang
Adobe Research
owang@adobe.com

Abstract

In this work we analyze the complex trade-off between data transfer, computation time, and power consumption when a multi-stage data-intensive algorithm (in this case video stabilization) is split between a low power mobile device and high power cloud server. We evaluate design choices in terms of which intermediate representations should be transferred to the server and back to the mobile device, and present a graph-based solution that can update the optimal joint mobile-cloud computation separation as the hardware configuration or user's requirements change. The practices we employ in this work can be extended to other mobile computer vision applications.

1. Introduction

Increasing amounts of visual data is captured, viewed, analyzed, and shared directly on mobile devices. However, these devices are limited in terms of memory and computational power, and are unable to perform many state-of-the-art computer vision and computer graphics algorithms. One way to overcome these limitations is to perform such expensive algorithms on *cloud*-based servers. However, there is an inherent trade-off in terms of latency, power consumption, bandwidth, and connectivity requirements when transferring data to and from cloud servers. In this work, we analyze this trade-off as it applies to the application of video stabilization, and present an optimal division of labor between mobile and cloud computation for a variety of real world configurations.

While mobile devices are becoming more and more powerful, the disparity between mobile and cloud based systems is likely to exist for the foreseeable future due to inherent power and size constraints. In fact, algorithms that split computation across devices are likely to become increasingly useful, as always-on connectivity becomes the norm, and mobile bandwidth and data limits improve. New applications of neural networks show impressive results in many domains, but often require powerful GPUs for computation. Applications that take advantage of large databases, will also benefit from a joint mobile-cloud framework. Cloud servers have an additional benefit in that they can provide power computation independent of the mobile device, so even if a large portion of the world's population is operating on less powerful mobile devices, cloud-based solutions can enable computationally complex applications to have a wider reach.

In this work, we look at the problem of video stabilization as a case study for analyzing different ways to split processing between mobile and cloud computation. We choose video stabilization as it is an important application – hand held mobile video capture is playing an increasingly significant role in our culture, and in most cases, users do not have the luxury of steady-cams or dolly shots for creating smooth motion. As a result, nearly all video sharing platforms offer software-based stabilization as a service, and more and more cameras (iPhone for example) perform software stabilization on every video captured. In addition to the practical significance, video stabilization is a multi-part process with high data requirements, making the division of labor non-trivial.

We present a system that splits the algorithm into a client-server architecture, with the entire pipeline implemented in its entirety on both systems. This allows us to transmit various levels of intermediate data, and make a direct comparison between computation time, bandwidth, and power consumption. The client performs the data acquisition, and is also the target device for consumption of the stabilized video (e.g., viewing and sharing), meaning that the entire video must start and end on the client. While we consider mobile phones as our client, the same technique could be applied to any connected low power compute device, such as wearable cameras. In order to address different hardware setups, we propose a graph-based formulation which can interactively compute the optimal distribution of computation based on the current hardware setup and users preferences. We show that the optimal splitting can in fact be somewhat counter-intuitive, for example, it may be preferable to send the entire input video rather than extract features locally under certain configurations.

The main contribution of this work is a description of

a system for splitting the computation of a state-of-the-art video stabilization method between a mobile device and cloud server, using a graph formulation to compute optimal distributions of work for different scenarios. We explain in detail how we measure the various cost functions to evaluate the trade-off, and provide analysis for different quality levels and hardware configurations. These techniques could be applied to any complex algorithm that we would like to jointly compute on low power mobile devices and a high powered cloud computer. We plan to release our Android test source code which can be used as a framework to implement the same methodology in other problem domains.

2. Related Work

We first discuss prior work related to video stabilization and then existing works on joint mobile-cloud computation.

Video Stabilization Software-based video stabilization is becoming an increasingly important preprocessing step for user created video. Initial solutions to video stabilization [22, 9] modeled motion with a rigid 2D transform, which are fast to compute, but cannot handle real 3D scenes with motion parallax.

When significant compute power is available, video stabilization methods have shown good results by first reconstructing the scene in 3D and then smoothing the resulting 3D camera trajectory. Once this virtual camera path is computed, the result can be rendered either as a warped mesh [18], or using a MRF to select different parts of the input video sequence [15]. While these methods show impressive quality results, the 3D reconstruction step is computationally expensive and fragile.

As a result, the above two methods have both been extended to operate directly in the image domain, for example by smoothing a subspace of 2D features trajectories [19], or joint frame selection and stabilization [13].

Other fast stabilization methods include approaches that perform a linear programming optimization to find an optimal set of similarity transforms [11], a variation of which is used in the cloud-based Youtube stabilizer. It has also been shown to be possible to use the phone's onboard inertial sensors [14] for the analysis stage, as is used in the Instagram Hyperlapse app.

In fact, some of these above approaches have been implemented to run entirely on the mobile device. This has a number of obvious advantages, most importantly the ability to operate in the absence of a network connection. However, these approaches achieve this by limiting the quality in one way or another, most often by restricting the motion models to low-degree of freedom transforms (such as similarity or affine), which means that they cannot correct for things like parallax or rolling shutter effects. As connectivity is rapidly improving, and "offline"-devices become more rare, we focus this work on a joint cloud/mobile solution to achieve a higher quality stabilization.

As a compromise between these global motion models and full 3D reconstruction, a number of works have proposed spatially varying homograpy transforms, which are smoothly interpolated over the image [21, 10]. Similar mesh-warping methods have been shown to be a powerful way of distorting images and videos while preserving realism, and have been successfully used in the past for retargeting [31], perspective correction [3], stereo modification [16], and 360 degree stitching [17]. We choose to implement a recent state-of-the-art approach called Bundled Camera Paths (BCP) [21], which is able to handle some degree of parallax due to the mesh-based warping approach, and demonstrates good results in difficult stabilization cases without requiring a potentially fragile 3D reconstruction. However, we note that many of the conclusions we draw could be easily extend to other approaches and applications.

Joint Mobile/Cloud Processing The rise of mobile devices and cloud compute services such as Amazon's EC2 has lead to the creation of an entirely new research area called mobile cloud computing (MCC). While MCC has received increasing amounts of coverage and interest recently (please refer to the recent surveys [1, 4, 27, 26], there have been relatively few works that describe in detail practical systems for computer vision applications.

Some similarity exists in other domains, for example, [5] describes algorithms that compute compressed feature representations to perform cloud computation, for example audio fingerprinting for song recognition (e.g., Shazam), or compressing images before sending them to the mobile web browser to save bandwidth.

In terms of graphics applications, Ferzli et al. [6] describe a method where entire images are sent to a server for basic image processing routines. Prior work has also looked at gaming and proposes an adaptive approach which varies the rendering quality based on available bandwidth [30]. As opposed to this we look at the benefits of jointly performing different stages of the pipeline on mobile and cloud devices.

Similar to us, Gharbi et al. [8] present an approach where a highly compact low resolution image representation is sent to a server where photometric enhancements are applied and the transform is returned to the mobile device which renders the final output image. Our method is inspired by this approach, and addresses a different set of transformations which allow for geometric warping of video data rather than photometric image transforms.

3. Method

The input video (I) originates on the mobile device, and the output stabilized video (O) is viewed or shared directly on the mobile device as well. All other steps can be

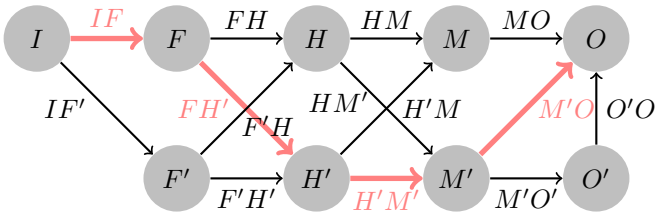

Figure 1: An example cloud-mobile computation graph, in this case for video stabilization. Nodes encode steps of the algorithm, where for example, A indicates mobile and A' indicates cloud computation (for definition of node labels, see Sec 3). Edges, e.g., AB, encode the cost of computing B and transferring the previous state from A to B. We show one example of an optimal computation path for a specific hardware configuration in red.

distributed across mobile and cloud computation devices. Most video stabilization routines operate with a similar set of distinct steps. Feature tracks (F) are computed in the input videos, which are then mapped to a motion model (H), which is then smoothed over time and a deformation model (M) is used to rendered the output video (O).

We consider each of these intermediate steps as nodes in a directed graph, where edges encode the associated costs (Figure 1). The cost of an edge AB is defined as:

$$AB = \lambda_p E_{energy}(A,B) + \lambda_t E_{time}(A,B) + \lambda_d E_{data}(A,B) \quad (1)$$

Where each error function encodes the cost of computing B, and the cost of transmitting the result of A to the platform that computes B. E_{energy} measures the consumption, E_{time} measures the computation time and E_{data} measures the data transfer, and $\lambda_\#$ are application specific weights. This representation allows us to find an optimal computation distribution by searching for shortest paths in this graph from I to O for different hardware configurations and preferences.

We now present a brief overview of each step as it applies to Bundled Camera Paths (BCP) [21] but refer readers to the original paper for full details.

3.1. Bundled Camera Paths

First, features are tracked from frame-to-frame using Lucas-Kanade feature tracking [2]. This produces a set of features F where $F_i(t) = [x, y, x', y']$ is the ith feature correspondence at frame t that indicates the pixel at coordinates (x, y) gets mapped to (x', y') in frame $t + 1$.

In BCP, the motion model H consists of a global homography warp and a subsequent grid of local homographies which model the residual motion, allowing the method to correct for parallax. Without loss of generality, these can be

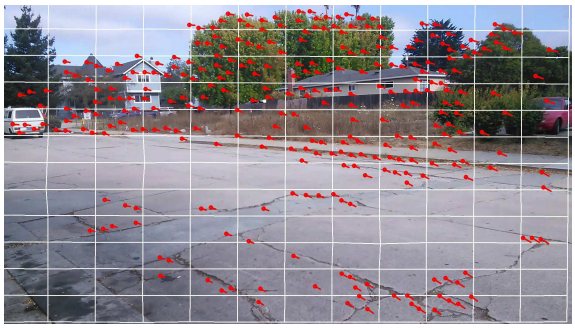

Figure 2: The deformed grid H (white), and features F (red).

combined into a single homography grid where $H_j(t)$ is the jth homography at time t. Fig 2 shows H and F overlaid on an image. From Fig 3, it is clear that the center of the grids follow paths.

Each grid of homographies $H(t)$ is computed by solving a system of linear equations with $4|j| * 2$ unknowns, where $|j|$ is the number of grid cells. This linear system minimizes the error $E(H(t)) = E_d(H(t), F(t)) + \alpha E_s(H(t))$, where E_d is the data term encouraging homographies to match the feature correspondences, E_s is the smoothness term which encourages similar deformation between neighboring homographies, and α is a weighting term.

Once the per-frame motion (H) has been estimated, a second optimization procedure is performed which solves for a new set of local homographies $\hat{H}$ which are smoothed both spatially (across t) and temporally (across j).

$$O(\hat{H}) = E_h(H, \hat{H}) + \lambda_t E_t(\hat{H}) + \lambda_o E_o(\hat{H})) \quad (2)$$

Where $E_h = |H(t) - \hat{H}(t)|^2$ encourages the deformed homographies to stay near the measured ones. For each frame t and nearby frame r, $E_t = |\hat{H}(t) - \hat{H}(r)|$ enforces temporal smoothness in the output path. For each grid cell m and all its neighbor n, the spatial smoothness term $E_o = |\hat{H}_m(t) - \hat{H}_n(t)|^2$ ensures that neighboring homographies are similar. This linear system of equations is solved over all frames simultaneously, and involves $4|j||i| * 2$ unknowns where $|i|$ is the number of frames.

Improvements over BCP We additionally describe a couple small improvements over BCP that we found to improve result quality, and may be useful for re-implementation.

In [21], the similarity constraint is written as:

$$E_s(T, \hat{T}) = \sum_{\hat{t}} |\hat{t} - \hat{t}_1 - \tau R_{90}(\hat{t}_0 - \hat{t}_1)|^2, R_{90} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad (3)$$

for each triangle T in the mesh, where τ is a constant based on T, and $\hat{t}, \hat{t}_0, \hat{t}_1$ form the deformed triangle $\hat{T}$. One issue is that this term leads to a degenerate case; $\hat{t} = \hat{t}_0 = \hat{t}_1$

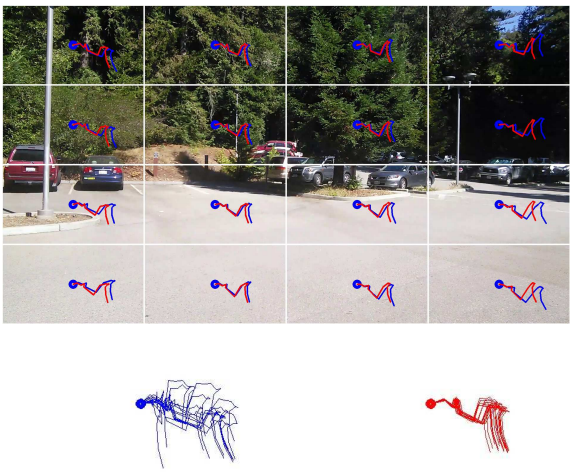

Figure 3: Top: Bundled camera paths overlaid on an input frame. Red paths show un-stabilized trajectories and blue the stabilized ones. The lower figures show all trajectories projected into the middle, which demonstrate that different grid cells require different transformations.

yields zero error. We instead use the distortion energy proposed in Zhang et al. [32], which is defined between an input and deformed quad $Q, \hat{Q}$ as,

$$E_s(Q, \hat{Q}) = min_{s \in S} \sum_{i=1}^{4} |s(q_i) - \hat{q}_i|^2 \qquad (4)$$

where S is the set of all similarity transforms. This equation has a closed form solution, for a full derivation please refer to prior work [32].

Finally, in the original work, after computing final locations $\hat{H}$ there is no guarantee of $C0$ or $C1$ continuity between neighboring homographies, so small gaps were filled by bilinear interpolation [21]. However, we found this approach to be brittle, as it often produces artifacts in-between grid cells. Instead, we simply *re-mesh* the warped homography grid, where each vertex is the average of all of the vertices of the neighboring homographies. This yields a mesh M which can be used to render the warped output video O.

3.2. Implementation

We develop an equivalent video stabilization system on both mobile and cloud systems, which allows us to easily interchange components. Both systems use OpenCV 3.1.0 and FFmpeg 3.0.3 for video encoding and decoding, OpenGL for rendering after re-meshing, and Eigen for matrix computations. We use libcurl to initiate and handle HTTP requests on the mobile, and Abyss server on the server to respond to these requests. In addition, we use FastCV to extract feature points on the client.

On the server side, we run our code on a 16GB RAM, Intel core i7-4980HQ Quad core processor Ubuntu 16.04LTS machine which is capable of 3.7GHZ on each cores. For the mobile, we use a 2GB RAM Motorola Moto G, 4th generation device which has a Octa-core processor (4x1.5 GHz Cortex-A53 and 4x1.2 GHz Cortex-A53) and Adreno 405 GPU, running on android 6.0. We believe that these specification capture the average specifications of smartphones today. For profiling the client, we use App Tune up Kit (ATK) [28] and Trepn [25] which are android applications capable of giving real time information on power consumption and data usage for a particular application. Our system was tested on a WiFi, T-mobile LTE and T-mobile 3G network with average bandwidth values shown in table 3.

The final video was compressed using an MJPEG Codec. Feature points were transferred as packed 32bit floats with accompanying packed 16bit integers for indexing. We also use packed 32bit floats to represent the mesh and optimized homographies.

To cover a wide range of input scenarios, we tested our system on a dataset of videos obtained from different smart phones (Nexus 5X, iPhone 5C, BLU WinLTE, Nexus 7 and Motorola Moto G4), most of which range from 10-20 seconds, and averaged the results over this dataset. The results reported in table 3 show the average values per frame over this dataset. We include the input videos and output stabilizations in the supplemental material.

Data. Given a video with p frames, and a setup that uses $m \times n$ grid cells and maximum of o feature points, we can estimate the resulting sizes in bytes: $F_{data} = (8 \times o \times p) + (2 \times o)$, $H_{data} = (8 \times p \times m \times n)$, $M_{data} = (2 \times p \times m \times n)$, and the size of the compressed video gives the values for uploading/downloading the full video. We split the results into upload and download data $E_{data}(A, B) = E_{data\_u}(A, B) + E_{data\_d}(A, B)$, which can then be computed by adding up the above quantities We verified these measurements experimentally through ATK as well. Because of the limitation of the test device, we set m and n to 12.

Time. $E_{time}(A, B)$ can be measured for each edge on the client and server separately and summed afterwards. We make use of the high resolution time library defined under std::chrono in C++. Again, for paths that require data transfer, we note that $E_{time}(A, B)$ is a function of not only the compute power but also the data transfer rate, which usually fluctuates. We therefore split $E_{time}(A, B)$ into computation and data transfer time, $E_{time\_c}(A, B) + E_{data\_u}(A, B)/\gamma_u + E_{data\_d}(A, B)/\gamma_d$ where $\gamma_{u,d}$ are average download speed and upload speeds, as shown in Table 1. The values in this table were obtained from [24] and [23].

Energy. Similar to computing E_{time}, we split the con-

	3G				4G			
(Mbps)	Tmobile	Verizon	Sprint	AT&T	Tmobile	Verizon	Sprint	AT&T
Download	3.48	0.66	0.64	2.22	21.02	21.11	15.04	18.91
Upload	1.91	0.27	0.10	0.79	11.59	8.22	4.70	6.77

Table 1: Average data speed for the four major carriers estimated from [23] and [24] at the time of submission.

sumption into data and computation related: $E_{energy} = E_{energy_c}(A,B) + \tau_u E_{data_u}(A,B) + \tau_d E_{data_d}(A,B)$. To compute $E_{energy_c}(A,B)$, we perform multiple runs of the application on our device for each edge on different videos, and report the average power consumption used for that stage as measured by ATK [28]. We note that ATK only allows us to measure the power for the entire application, thus, to measure the energy consumption of a particular edge say HM, we simply measure the energy for the path, $IFHM$ and then subtract the energy we measured for IFH. The values obtained were also cross checked with Trepn [25]. τ_u and τ_d correspond to the average power consumption per up- and downloaded MB. We compute this for different connection types (e.g., LTE and 3G) empirically, by measuring the energy consumption of file upload and download at different locations, and averaging out over the entire result, see Table 2. While these measurements are rough, and will not capture the exact energy consumption, which is a function of many uncontrolled variables, such as temperature, packet-loss, and background processes, we found the numbers we obtain are sufficient for computing the splits in computation that we require.

	Upload (mJ/MB)	Download (mJ/MB)
3G	19386.38	4031.95
LTE	7288.81	3628.66
WiFi	15775.13	2148.18

Table 2: Estimated energy consumed in millijoules per megabyte

3.3. Graph-based Solution

After the different cost functions are evaluated as described above, our graph-based representation allows for the user to interactively adjust the importance of different criteria, for example varying the weights based on how much power consumption vs computation time they are willing to expend. This flexibility allows our method to be used in a wide range of application scenarios, with even adaptive computation splitting based on the *current* state of the device, for example when transitioning from a WiFi to a 3G connection.

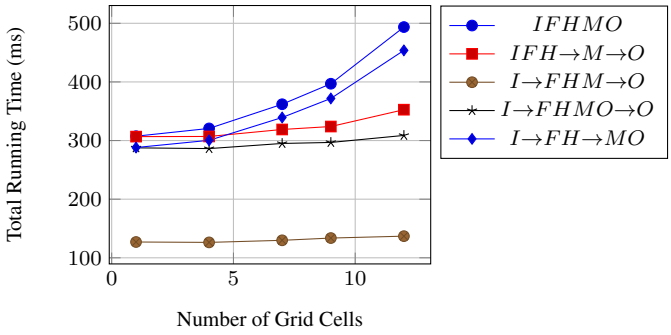

Figure 4: This plot shows how the running time changes for a number of different paths through the cloud-mobile compute graph as the algorithm parameters are adjusted. The effect of the distribution of work largely dominates algorithmic settings, but cloud processing helps reduce the time growth, allowing for more motion parallax correction.

For the results that we present below, we set $\lambda_p = \lambda_t = 1, \lambda_d = 10$, which gives us the best trade-off with our available hardware setup, and encode a roughly equal penalty on each modality. On the other hand, when we set the parameters to be $\lambda_p = 10, \lambda_t = \lambda_d = 1$, for example when we want to run in a *power-saver* mode, we get a completely different path. These resulting paths are shown in Fig 5. These results indicate that fully local, or fully cloud based computer vision solutions are only a subset of the various paths between input and output of the graph, and are often sub-optimal. We note that due to caching, there may be slight differences in total time when compared to the individually computed parts. If more precise measurements are required, it is possible to directly evaluate the performance of the entire path on the graph.

4. Results

Table 3 enumerates all of the choices for which computation to perform on the mobile device and which to perform on the cloud, along with their measurements. In particular, we find that the paths $IF \rightarrow HM \rightarrow O$ and $IFH \rightarrow M \rightarrow O$ to provide a nice balance between running time, data, and energy usage, depending on the hardware configurations. A final production ready optimized system could correspond to different values for each of the edges in our graph, yielding absolute runtime improvements, and possibly a different final path, but the overall process to compute it would be the same.

As the method is largely resolution independent (most computations are performed on the dimensions of the grid), we vary the complexity of the grid (e.g., increasing the number of homographies per frame $|k|$). This allows us to smoothly vary between a single global motion model, and a method that allows for more complex deformations of the

Mobile →	Cloud →	Mobile	Moto G4, T-mobile LTE Time (ms)	Total Data (kB)	Energy (mJ)	Moto G4, WiFi Time (ms)	Total Data (kB)	Energy (mJ)
$IFHMO$	-	-	450.54	0	1301.7	450.54	0	1301.7
IFH	M	O	325.03	4.81	899.35	1133.8	4.81	303.24
IF	H	MO	451.67	14.26	1350.56	443.76	14.26	1004.66
IF	HM	O	322.45	20.55	910.48	124.64	12.15	376.63
I	FHM	O	135.13	160	1166.2	128.3	81.35	1433.85
I	$FHMO$	O	292.25	81.35	1211.5	98.7	160	1264.9
I	FH	MO	421.47	83.46	1651.58	417.82	83.46	1892.93
I	F	HMO	435.37	90.8	1720.15	431.88	90.8	2341.14

Table 3: This table shows some paths through our joint mobile-computation graph and the costs associated with them, per frame. We present two sets of hardware configurations, including a Motorola Moto G4, operating on T-mobile Cellular LTE data and Moto G4 operating on WiFi (Upload and download speed of 23.86Mbps and 23.06Mbps respectively).

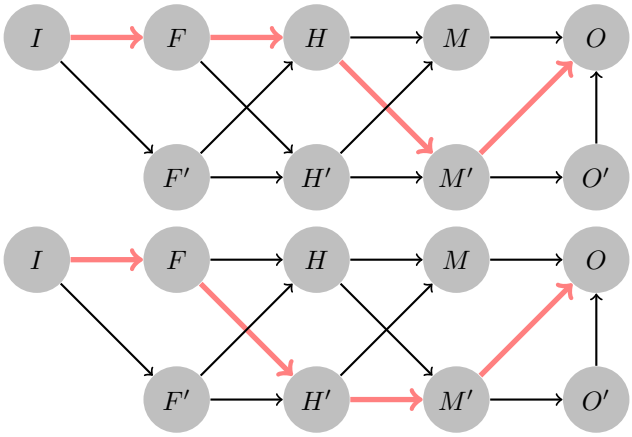

Figure 5: The graph above shows the optimal paths obtained from Moto G4 LTE data, with parameters $\lambda_p = \lambda_t = 1, \lambda_d = 10$. The graph below is for the Moto G4 Wifi data, with parameters $\lambda_p = 10, \lambda_t = \lambda_d = 1$. Data can be found in Table 3

scene. Figure 4 shows the effect of varying this parameter on computation time, and it can be seen that the choice of path plays a much larger role than any of the algorithm settings. We note that the mobile devices we tested on were not able to operate on grids larger than 12×12 due to memory reasons.

Of course, the edge weights in the graph are dependent on the hardware. In the results above, we use the default hardware configuration (Moto G4 LTE and WiFi data), however, the optimal settings may vary with different available bandwidth. Figure 5 shows two different paths through our graph, showing how the work gets redistributed based on hardware configurations and user preferences for example.

5. Conclusions

The main drawback of joint cloud-mobile systems is that it requires a connection to the mobile device in order to work. In many places this requirement is still unrealistic. However, as connectivity grows, situations where cloud-based computation are not possible will likely become rarer. Furthermore, most user-created content is shared with friends, which also requires a network connection, so at some point before sharing cloud-based computation *will* be possible. Despite this, in cases of limited connectivity one solution could be to have a low-quality proxy computed locally which is then updated with the improved version when the phone gets network access.

Similar to prior work addressing photometric modifications [8], we have presented an evaluation of only one application, and while we feel that this is an important application, many of the findings in this work are restricted to similar feature-based image warping approaches, such as 360 degree panorama stitching, image retargeting, and stereo mapping. This is somewhat unavoidable, as each set of algorithms operates on different data, however we hope that the basic graph structure and parameter exploration steps will encourage other joint mobile-cloud applications.

Mobile cloud computing is an emerging area, and we believe there is substantial possibilities for follow up work in regards to vision applications. A recent follow-up work to BCP includes structure from motion (SfM) for better performance in cases with large amounts of parallax[20]. Due to the highly non-convex optimization problem, SfM can be very expensive, and our joint approach should provide *substantially* more benefits in these scenarios, especially as reconstruction must often be performed on low power mobile robotics devices.

Many vision methods involve feature extraction, where the underlying representation has substantially less information. Despite this, the optimal distribution of computation can be somewhat counter-intuitive, so a principled way to determine the best splitting based on the current parameters is a useful feature.

Another family of approaches are those that require databases that are too large to store on mobile devices. These have been used for example, to drive hole filling methods [12]. In particular, deep learning approaches have shown a number promising applications, however most of these approaches require powerful GPUs in order to work. For example, in the popular style transfer application [7, 29], feature maps are extracted from input images, which are then used to constrain a costly optimization procedure. We believe that these approaches are particularly well suited to joint mobile-cloud computation.

References

[1] P. Bahl, R. Y. Han, L. E. Li, and M. Satyanarayanan. Advancing the state of mobile cloud computing. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 21–28. ACM, 2012. 2

[2] J.-Y. Bouguet. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation*, 5(1-10):4, 2001. 3

[3] R. Carroll, M. Agrawal, and A. Agarwala. Optimizing content-preserving projections for wide-angle images. In *ACM Transactions on Graphics (TOG)*, volume 28, page 43. ACM, 2009. 2

[4] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013. 2

[5] N. Fernando, S. W. Loke, and W. Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, 2013. 2

[6] R. Ferzli and I. Khalife. Mobile cloud computing educational tool for image/video processing algorithms. In *Digital Signal Processing Workshop and IEEE Signal Processing Education Workshop (DSP/SPE), 2011 IEEE*, pages 529–533. IEEE, 2011. 2

[7] L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016. 7

[8] M. Gharbi, Y. Shih, G. Chaurasia, J. Ragan-Kelley, S. Paris, and F. Durand. Transform recipes for efficient cloud photo enhancement. *ACM Transactions on Graphics (TOG)*, 34(6):228, 2015. 2, 6

[9] M. L. Gleicher and F. Liu. Re-cinematography: Improving the camerawork of casual video. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 5(1):2, 2008. 2

[10] M. Grundmann, V. Kwatra, D. Castro, and I. Essa. Calibration-free rolling shutter removal. In *Computational Photography (ICCP), 2012 IEEE International Conference on*, pages 1–8. IEEE, 2012. 2

[11] M. Grundmann, V. Kwatra, and I. Essa. Auto-directed video stabilization with robust l1 optimal camera paths. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 225–232. IEEE, 2011. 2

[12] J. Hays and A. A. Efros. Scene completion using millions of photographs. In *ACM Transactions on Graphics (TOG)*, volume 26, page 4. ACM, 2007. 7

[13] N. Joshi, W. Kienzle, M. Toelle, M. Uyttendaele, and M. F. Cohen. Real-time hyperlapse creation via optimal frame selection. *ACM Transactions on Graphics (TOG)*, 34(4):63, 2015. 2

[14] A. Karpenko, D. Jacobs, J. Baek, and M. Levoy. Digital video stabilization and rolling shutter correction using gyroscopes. *CSTR*, 1:2, 2011. 2

[15] J. Kopf, M. F. Cohen, and R. Szeliski. First-person hyper-lapse videos. *ACM Transactions on Graphics (TOG)*, 33(4):78, 2014. 2

[16] M. Lang, A. Hornung, O. Wang, S. Poulakos, A. Smolic, and M. Gross. Nonlinear disparity mapping for stereoscopic 3d. *ACM Transactions on Graphics (TOG)*, 29(4):75, 2010. 2

[17] J. Lee, B. Kim, K. Kim, Y. Kim, and J. Noh. Rich360: optimized spherical representation from structured panoramic camera arrays. *ACM Transactions on Graphics (TOG)*, 35(4):63, 2016. 2

[18] F. Liu, M. Gleicher, H. Jin, and A. Agarwala. Content-preserving warps for 3d video stabilization. *ACM Transactions on Graphics (TOG)*, 28(3):44, 2009. 2

[19] F. Liu, M. Gleicher, J. Wang, H. Jin, and A. Agarwala. Subspace video stabilization. *ACM Transactions on Graphics (TOG)*, 30(1):4, 2011. 2

[20] S. Liu, B. Xu, C. Deng, S. Zhu, B. Zeng, and M. Gabbouj. A hybrid approach for near-range video stabilization. 2016. 6

[21] S. Liu, L. Yuan, P. Tan, and J. Sun. Bundled camera paths for video stabilization. *ACM Transactions on Graphics (TOG)*, 32(4):78, 2013. 2, 3, 4

[22] C. Morimoto and R. Chellappa. Fast 3d stabilization and mosaic construction. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 660–665. IEEE, 1997. 2

[23] ookla. Speedtest Market Report, United States . http://www.speedtest.net/reports/united-states/, 2016. [Online; accessed 11-November-2016]. 4, 5

[24] OpenSignal. State of Mobile Networks: USA (February 2016) . http://www.opensignal.com/reports/2016/02/usa/state-of-the-mobile-network/, 2016. [Online; accessed 11-November-2016]. 4, 5

[25] I. Qualcomm Technologies. Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler. [Online; accessed 11-November-2016]. 4, 5

[26] M. R. Rahimi, J. Ren, C. H. Liu, A. V. Vasilakos, and N. Venkatasubramanian. Mobile cloud computing: A survey, state of art and future directions. *Mobile Networks and Applications*, 19(2):133–143, 2014. 2

[27] Z. Sanaei, S. Abolfazli, A. Gani, and R. Buyya. Heterogeneity in mobile cloud computing: taxonomy and open challenges. *IEEE Communications Surveys & Tutorials*, 16(1):369–392, 2014. 2

[28] Q. Technologies. App Tune-up Kit. https://developer.qualcomm.com/software/app-tune-up-kit. [Online; accessed 11-November-2016]. 4, 5

[29] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. *arXiv preprint arXiv:1603.03417*, 2016. 7

[30] S. Wang and S. Dey. Adaptive mobile cloud computing to enable rich mobile multimedia applications. *IEEE Transactions on Multimedia*, 15(4):870–883, 2013. 2

[31] Y.-S. Wang, H. Fu, O. Sorkine, T.-Y. Lee, and H.-P. Seidel. Motion-aware temporal coherence for video resizing. *ACM Transactions on Graphics (TOG)*, 28(5):127, 2009. 2

[32] G.-X. Zhang, M.-M. Cheng, S.-M. Hu, and R. R. Martin. A shape-preserving approach to image resizing. In *Computer Graphics Forum*, volume 28, pages 1897–1906. Wiley Online Library, 2009. 4