A FAST AND EFFICIENT SIFT DETECTOR USING THE MOBILE GPU

Blaine Rister, Guohui Wang, Michael Wu and Joseph R. Cavallaro

Department of Electrical and Computer Engineering, Rice University, Houston, Texas
Email: {blaine.rister, wgh, mwb2, cavallar}@rice.edu

ABSTRACT

Emerging mobile applications, such as augmented reality, demand robust feature detection at high frame rates. We present an implementation of the popular Scale-Invariant Feature Transform (SIFT) feature detection algorithm that incorporates the powerful graphics processing unit (GPU) in mobile devices. Where the usual GPU methods are inefficient on mobile hardware, we propose a heterogeneous dataflow scheme. By methodically partitioning the computation, compressing the data for memory transfers, and taking into account the unique challenges that arise out of the mobile GPU, we are able to achieve a speedup of 4-7x over an optimized CPU version, and a 6.4x speedup over a published GPU implementation. Additionally, we reduce energy consumption by 87 percent per image. We achieve near-realtime detection without compromising the original algorithm.

Index Terms— computer vision, mobile computing, feature detection, graphics processing unit (GPU), OpenGL for Embedded Systems (OpenGL ES)

1. INTRODUCTION

The recent development of low-cost, high-quality cameras in mobile devices has generated enormous interest in mobile computer vision applications, such as face detection and augmented reality [1, 2, 3]. Scale-invariant interest points, or features, are essential to many computer vision tasks, such as object recognition and tracking, and will continue to gain relevance in the realm of mobile computing [4]. The Scale-Invariant Feature Transform (SIFT) is a practical algorithm for detecting and describing features that are invariant to scaling and rotation, and partially invariant to affine transformation, illumination, noise, and partial occlusion [5].

However, computer vision algorithms such as SIFT are computationally complex, making it difficult to meet the demands of emerging mobile applications. The limitations of mobile hardware and the lack of programming tools prevent feature detection algorithms from being implemented in realtime applications. In addition, reliance on the relatively small batteries in mobile devices dramatically increases the concern of power consumption in computer vision applications, which can occupy the power-hungry CPU for long periods of time. Seeking to accelerate the process, we enlist the help of general-purpose computing on graphics processing units (GPGPU). Graphics processing units (GPUs) are cost-effective processors that far exceed CPUs in computational throughput on parallel tasks [4, 6]. By methodically partitioning the workload between the GPU and CPU and organizing the data for efficient computation, we are able to achieve significant speedup over an optimized CPU version and related GPU work, and drastically reduce total energy consumption.

2. PRIOR WORK

GPUs have been shown to provide great speedup on a wide range of computer vision algorithms, including SIFT [7, 8, 9, 10]. However, this success has mostly been confined to desktop GPU applications. Firstly, mobile hardware has a host of limitations that make GPGPU development considerably more challenging. Traditional GPGPU methods call for storage and processing of as much data as possible on the GPU [11], but this is not practical on mobile devices. Without support for dynamic branching or asynchronous readback, we must rethink the way that we process data on the GPU [12]. Secondly, programming models supporting GPGPU such as CUDA and OpenCL are unavailable on most mobile devices, forcing developers to use OpenGL for Embedded Systems (OpenGL ES), which was designed for graphics [13]. Although some algorithms, expressing a high degree of parallelism, have been successfully accelerated using OpenGL ES [2, 3, 14], the few mobile GPU implementations of SIFT of which we are aware were not fast enough to support emerging applications [12], due to frequent branching in the later stages of the algorithm, and the considerable overhead of CPU-GPU memory transfers.

To overcome these obstacles, we rework the algorithm's dataflow to perform the most time-consuming and inherently parallel tasks on the GPU, leaving the rest for the CPU and minimizing memory transfers. Additionally, we utilize an efficient data packing scheme that reduces the amount of data that must be transferred, simultaneously maximizing the efficiency of OpenGL ES rendering operations. Because many of the competing algorithms express a similar degree of parallelism, our methods could easily be extrapolated to a wide range of algorithms, allowing developers to tailor our tech-

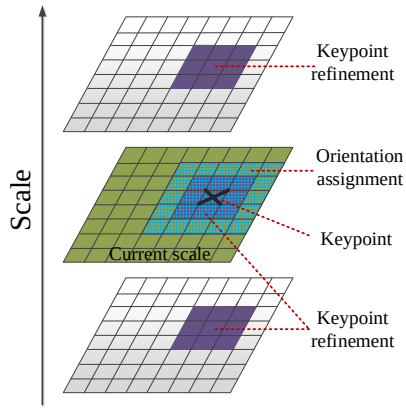

Fig. 1. SIFT detector memory access pattern. Top scale: Extrema detection, Middle: Extrema, GSS and DoG, Bottom: Extrema and DoG.

niques to specific applications [9, 15].

3. SIFT OVERVIEW

There are two major steps to extracting features from an image: detection and description. Feature detection identifies the spatial locations and dominant orientations of interest points, while feature description provides criteria for matching. We focus on the feature detection phase of the algorithm because its inherent data parallelism makes it a good candidate for GPU acceleration. We now describe the major stages of the SIFT detector:

In order to achieve scale invariance, we generate multiple copies of the input image through repeated Gaussian smoothing, each representing the scene at a different scale, which are collectively known as the Gaussian Scale Space (GSS) pyramid [5]. Once we have smoothed to twice the original scale, we downsample the result and begin the next octave of the pyramid.

We then construct the Difference of Gaussians (DoG) pyramid by subtracting pairs of successive images in the GSS Pyramid pixel-by-pixel. This approximates the well-studied Laplacian of Gaussian function [16]. Keypoints are identified as local extrema in the DoG pyramid, by comparison with 26 neighbors in 3×3 regions at the current and neighboring scales. Finally, some keypoints are rejected by local contrast and edge detection thresholds, and the remaining locations are refined to sub-pixel accuracy by Gaussian elimination.

With keypoint locations known, we determine their orientations to provide rotational invariance. The gradient pyramid is constructed by taking the grayscale intensity gradient of the relevant GSS pyramid levels at each pixel. Local orientations are assigned to each keypoint by converting the gradients in a neighborhood around each keypoint to polar form, and accumulating their weighted magnitudes into a histogram of orientations, where up to four histogram maxima become keypoint orientations.

Table 1. Profiling results of major stages of the algorithm on CPU and GPU, on the Google Nexus 7 with Tegra 3 system-on-chip, with an average of 88 keypoints per image. Our implementation is traced in bold and italicized.

Type of task	Stage of algorithm	Time (ms)	
		On CPU	On GPU
Computation	GSS Pyramid	734.3	***28.2***
	Diff. of Gaussians	***3.6***	4.5
	Extrema detection	***5.1***	32.8
	Keypoint Refinement	***0.8***	N/A[†]
	Polar Gradient	***45.0***	33.2
Memory transfer	Load image to GPU	***0.5***	
	GSS Pyramid readback	***23.5***	
	Keypoint readback	0.8	
	Gradient readback	21.7	

[†] Keypoint refinement is not implemented on GPU, due to the lack of support for dynamic looping.

4. IMPLEMENTATION DETAILS

We present an efficient SIFT detector for mobile devices, using a heterogeneous methodology. Code was written for the Android Native Development Kit with OpenGL ES 2.0.

Our main contributions consist of methodical partitioning of the workload between the CPU and GPU, efficient packing of image data into GPU texture memory, and on-the-fly code generation for branch-free convolution in graphics hardware. Each of these design decisions allows us to improve upon the work of Kayombya in frame rate and energy efficiency [12].

4.1. Efficiently-partitioned heterogeneous computation

Careful partitioning of the workload between the CPU and GPU significantly improves processing ability over other SIFT implementations [12]. Fig. 1 shows the memory access pattern for key stages of the algorithm. Table 1 shows that of the major stages of the SIFT detector, Gaussian smoothing is by far the most expensive, and the only one with the speedup to warrant the overhead of GPU memory transfers. Extrema detection performed very poorly on the GPU due to scalar comparisons and the lack of support for dynamic branching in mobile GPUs. While the gradient was accelerated by the GPU, the readback time was prohibitive. All other operations were cheap on both processors, and would not warrant the memory transfer overhead of GPU acceleration.

These data led us to compute only the GSS pyramid on the GPU, and leave the rest for the CPU, as seen in Fig. 2. The decision is motivated by the unique challenges of mobile GPUs, in which data readback stalls the rendering pipeline, and thus cannot be hidden by concurrent computation, resulting in the significant overheads seen in Table 1 [13]. We further reduce the number of memory transfers by downsampling pyramid levels on the GPU, rather than performing a CPU downsample and subsequent memory transfer. The resulting program

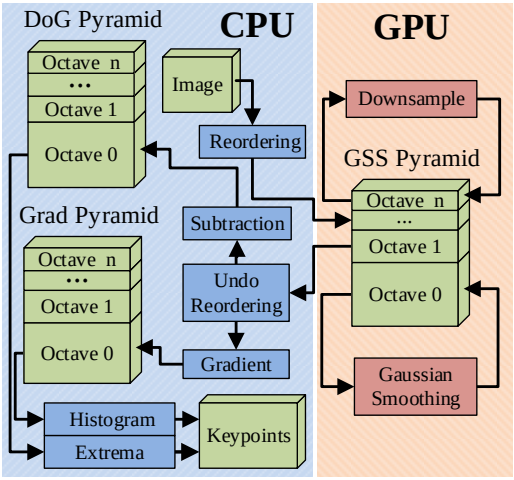

Fig. 2. Proposed dataflow. Extrema and refinement combined in diagram.

completes the SIFT detector with only one GPU to CPU transfer per pyramid level, and one CPU to GPU transfer in total.

4.2. GPU data compression by pixel reordering

We implement an efficient pixel reordering scheme that tightly packs data into GPU memory, reducing memory transfer time and accelerating OpenGL ES rendering operations. Images are stored as textures in OpenGL ES, where each pixel of a texture is called a "texel." The standard texture format is RGBA, where each texel contains separate red, blue, green, and alpha channels [13]. We reorder the input image so that every 2×2 grayscale square is stored as one RGBA texel, with one pixel per channel, as seen in Fig. 3.

We define arithmetic intensity as follows [6]:

$$\text{arithmetic intensity} = \frac{\text{No. of additions and multiplications}}{\text{No. of texture fetches}}$$

Packing the data in this way reduces the total number of texture fetches in both Gaussian smoothing and downsampling by a factor of four. Using this scheme, the arithmetic intensity of the Gaussian blur is increased from 2 to 8, resulting in an efficient program that can keep the fragment processors of the GPU busy between memory accesses. Perhaps more importantly, the amount of data that must be read back from the GPU is reduced by the same factor.

Packing by 2×2 squares, rather than some other geometry, is efficient because it reflects the memory access pattern of SIFT. Data that will be processed together are returned by the same texture fetch, optimizing memory operations. If the image were packed without reordering, as in Kayombya's implementation, the arithmetic intensity would depend on the dimension that we traverse, resulting in a large performance hit in vertical filtering [12]. Our tests show that the overhead of reordering the input image, and then reversing the reordering on the CPU, is negligible. Thus, the pixel reordering scheme reduces memory transfer time and accelerates computation at almost no cost.

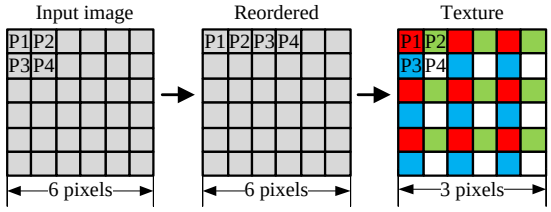

Fig. 3. Pixel reordering to compress data.

4.3. Branchless convolution through on-the-fly code generation

We generate GPU shader programs at runtime to reflect the parameters of the algorithm, accelerating computation by unrolling filtering loops. The width of the Gaussian smoothing kernel changes for each successive pyramid level, and is often based on user-defined parameters, so a single set of filtering programs would have to evaluate loop conditions to determine which pixels to process. Because the mobile GPU lacks dynamic branching, this would incur serious performance penalties. Instead, we generate and compile filter programs with all loops unrolled, eliminating branching from the program. We do not even need to branch to check image boundaries, as this is done for us by OpenGL ES. The same binary programs can be used for all of the images in a video stream, assuming that the parameters of the algorithm do not change, so the overhead of switching programs is not a performance issue. These techniques allow fine-tuning by the user without loss of performance.

4.4. Other optimizations

The original algorithm calls for doubling of the initial image in both width and height, to detect more features, but this is not conducive to realtime processing. We omit this step, reducing the amount of data that must be processed by a factor of two. This is a common step to take in high-framerate implementations of SIFT [7]. In our tests, the number of features is reduced by sixty-four percent, but we should note that the ones missing are low-scale features, which are less stable to matching [5]. If a high feature count is required, upsampling can be reintroduced at the expense of frame rate. Overall, this is a necessary tradeoff to bring SIFT close to realtime speeds.

Finally, we utilize the separability of Gaussian smoothing with separate shader programs for horizontal and vertical passes, reducing the total number of texture fetches from Nw^2 to $2Nw$, where N is the number of pixels in the image, and w is the width of the one-dimensional filter kernel. Separable filtering allows for a linear memory access pattern, which is especially efficient on the GPU [6]. In addition to these optimizations, our main contributions of efficient workflow partitioning, pixel reordering, and on-the-fly code generation are essential to providing the frame rates we desire.

5. EXPERIMENTAL RESULTS

We tested the performance of our implementation against an optimized CPU version running in a single thread. For

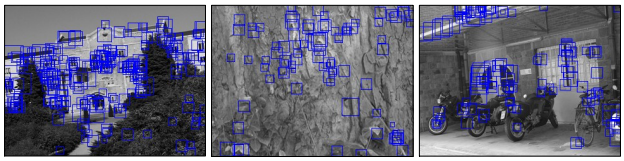

Fig. 4. Interest points detected in 320×240 images. Box size proportional to scale, orientation not shown.

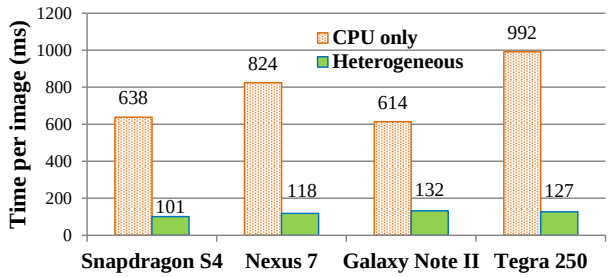

Fig. 5. Execution time observed on different devices.

the most informative comparison, we benchmarked on four different devices, representing a wide range of current mobile hardware. The test platforms are: Qualcomm Snapdragon S4 APQ8064 Mobile Development Platform, with Android 4.1.2; Google Nexus 7, with NVIDIA Tegra 3, Android 4.2; Samsung Galaxy Note II, with Samsung Exynos Quad, Android 4.1.1; and NVIDIA Tegra 250 Development Board, with NVIDIA Tegra 2, Android 2.2. Benchmarks were performed on a popular dataset of 320×280 pixel images, yielding an average of 88 keypoints over three octaves per image [17]. The small image size is appropriate for our goal of realtime processing. As expected, the number of features varies widely with the image size and the parameters of the algorithm. Some results are shown graphically in Fig. 4.

All devices we tested show considerable acceleration with heterogeneous computation, as seen in Fig. 5. The speedup ranges from 4.7x on the Galaxy Note II to 7.0x on the Nexus 7. Table 2 shows that the readback time varies drastically across different devices, mostly accounting for the slower execution time in the Note II's Mali-400 GPU. ARM admits this drawback of their design [18]. The Snapdragon S4 processed 9.9 images per second, elevating the SIFT detector to the realm of near-realtime mobile applications.

Next, we compare our heterogeneous implementation with the mobile GPU implementation of SIFT in Kayombya [12]. We reintroduce upsampling to the algorithm, and reduce the image size to 224×224 pixels for an accurate comparison. The author reports execution time only up to keypoint refinement, so we benchmark our implementation performing the same tasks. This comparison is not in our favor, as a complete implementation of the author's dataflow scheme would read orientation data back to the CPU, which we have shown to be a costly operation. Furthermore, the author's implementation uses an approximate method of GPU extrema detection to reduce complexity. We compare our results from the Tegra 250 development board to Kayombya's, from a Qualcomm FFA device with a Snapdragon S2 system-on-chip, so that the testing platforms are nearly contemporary. We show significant improvement upon the other design, averaging 148 ms per image, to Kayomba's 952 ms, for a 6.4x speedup.

Table 2. Profile of heterogeneous implementations, in ms.

Task	S4	Nexus 7	Note II	Tegra 250
GSS Pyramid	43.5	28.2	38.7	52.8
Readback	8.4	23.5	43.1	5.0
Orientation	43.6	58.0	49.5	67.4
Misc. SIFT	5.8	8.3	0.8	1.3

Table 3. Power and energy consumption on NVIDIA Tegra 250 development board.

Test items	Results	
	CPU-only	Heterogeneous
Power (mW)	3186	3383
Energy per image (mJ)	3161	430

Finally, we test the power and energy consumption of our implementation on the NVIDIA Tegra 250 development board. Measurements were taken as an average of 50 seconds of continuous iteration over the dataset. The development board takes a 15-volt input, resulting in higher power consumption than on a phone or tablet. Although Table 3 shows that the two implementations consume comparable power, the heterogeneous implementation reduces energy consumption by 87 percent per image compared to the CPU version, by occupying the processor for a shorter period of time.

6. CONCLUSIONS

We introduced an efficient implementation of the SIFT feature detector algorithm utilizing mobile GPU acceleration. Profiling of the major stages of the algorithm on both the GPU and CPU led us to develop a fast dataflow scheme for heterogeneous computation. Several key optimizations, such as reordering and compression of the input image, minimized the communication overhead and accelerated GPU computation. Considerable speedup was achieved over an optimized CPU version and related GPU work, resulting in near-realtime processing. Additionally, energy consumption was greatly reduced on an image-by-image basis. These techniques can be used to improve framerates and save energy in emerging mobile applications, such as object recognition, panorama construction, and augmented reality. In the future, we plan to explore what additional speedup CPU multi-threading and OpenCL might have to offer [19, 20].

Acknowledgments

This work was supported in part by Samsung, and by the US National Science Foundation under grants EECS-1232274, EECS-0925942 and CNS-0923479.

7. REFERENCES

[1] S. E. Lee, Y. Zhang, Z. Fang, S. Srinivasan, R. Iyer, and D. Newell, "Accelerating mobile augmented reality on a handheld platform," in *IEEE International Conference on Computer design (ICCD)*, 2009, pp. 419–426.

[2] M. Bordallo Lopez, H. Nykanen, J. Hannuksela, O. Silven, and M. Vehvilainen, "Accelerating image recognition on mobile devices using GPGPU," in *SPIE Parallel Processing for Imaging Applications*, vol. 7872, 2011, pp. 78 720R–78 720R–10. [Online]. Available: http://dx.doi.org/10.1117/12.872860

[3] K.-T. Cheng and Y. Wang, "Using mobile GPU for general-purpose computing - a case study of face recognition on smartphones," in *IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, April 2011, pp. 1 –4.

[4] S. Srinivasan, Z. Fang, R. Iyer, S. Zhang, M. Espig, D. Newell, D. Cermak, Y. Wu, I. Kozintsev, and H. Haussecker, "Performance characterization and optimization of mobile augmented reality on handheld platforms," in *IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 128 –137.

[5] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[6] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (GPU Gems)*. Addison-Wesley Professional, 2005.

[7] S. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," *Machine Vision and Applications*, vol. 22, pp. 207–217, 2011.

[8] S. Warn, W. Emeneker, J. Cothren, and A. Apon, "Accelerating SIFT on parallel architectures," in *IEEE International Conference on Cluster Computing and Workshops*, Sept. 2009, pp. 1–4.

[9] H. Xie, K. Gao, Y. Zhang, J. Li, and Y. Liu, "GPU-based fast scale invariant interest point detector," in *IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, March 2010, pp. 2494 –2497.

[10] D. Connors, K. Dunn, and J. Wiencrot, "Adaptive OpenCL (ACL) execution in GPU architectures," in *ACM International Conference on High-Performance and Embedded Architectures and Compilers*, Jan. 2013.

[11] S. Heymann, B. Fröhlich, F. Medien, K. Müller, and T. Wiegand, "SIFT implementation and optimization for general-gurpose GPU," in *In Winter School of Computer Graphics*, 2007.

[12] G.-R. Kayombya, "SIFT Feature Extraction on a Smartphone GPU using OpenGL ES 2.0," Master's thesis, MIT, Cambridge, MA, 2010.

[13] The Khronos Group, *The OpenGL ES 2.0 Specification*. [Online]. Available: http://www.khronos.org/opengles

[14] A. Ensor and S. Hall, "GPU-based image analysis on mobile devices," *CoRR*, vol. abs/1112.3110, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1112.html#abs-1112-3110

[15] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-Up Robust Features (SURF)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, Jun. 2008.

[16] T. Lindeberg, "Scale-space theory: A basic tool for analysing structures at different scales," *Journal of Applied Statistics*, pp. 224–270, 1994.

[17] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 10, pp. 1615 –1630, Oct. 2005.

[18] ARM, *Mali GPU Application Optimization Guide*. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0555a/index.html

[19] The Khronos Group, *The OpenCL Specification*. [Online]. Available: http://www.khronos.org/opencl

[20] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2013.