Texture Cache Approximation on GPUs

Mark Sutherland Joshua San Miguel Natalie Enright Jerger

Dept. of Electrical and Computer Engineering, University of Toronto

{suther68,enright}@ece.utoronto.ca, joshua.sanmiguel@mail.utoronto.ca

Abstract

We present texture cache approximation *as a method for using existing hardware on GPUs to eliminate costly global memory accesses. We develop a technique for using a GPU's texture fetch units to generate approximate values, and argue that this technique is applicable to a wide variety of GPU kernels. Applying texture cache approximation to an image blur kernel on an NVIDIA 780GTX, we obtain a 12% reduction in kernel execution time while only introducing 0.4% output error in the final image.*

1. Introduction

Applications from domains such as computer vision and machine learning are highly amenable to approximate computing; they operate on noisy data and produce outputs whose precision can be reduced without noticeable impact on output quality [2, 3]. Such applications are also well-suited for the massive parallelism of graphics processing units (GPUs). While approximation techniques have already been proposed for GPUs [6, 7], we observe that these techniques do not fully utilize the existing hardware capabilities of the GPU, specifically, the texture hardware. We believe that this hardware is well-suited to load value approximation, a technique that approximates data values to avoid costly memory accesses [8]. This paper proposes *texture cache approximation* on GPUs as a means of eliminating global memory accesses and improving performance with low application error.

2. Texture Hardware Background

Modern GPUs dedicate an increasing fraction of their resources to a fast and flexible memory system. For example, NVIDIA's Fermi microarchitecture now includes a CPU-like multi-level cache hierarchy, where each streaming multiprocessor (SM) has its own private L1 cache, backed by a write-back shared L2 that services all loads and stores to global memory. In this work, we primarily focus on the 12kB private texture cache associated with each SM; each SM also has 4 texture fetch units which are designed to exploit the spatial locality of rendering 2D/3D textures and surfaces [4].

Textures are special objects that contain colour information meant to give an on-screen object a particular appearance. Although support for texture rendering is not strictly necessary, using a texture allows graphics artists to draw an entire object using far fewer polygons, saving critical draw time. The texture fetch units that interface between threads and the on-chip texture cache have the following advanced functionality that we utilize in this paper: the ability to interpolate between two or more adjacent cache entries, and automatic wrap-around for accesses that are out-of-bounds. Interpolation is the most important feature in this work, as it allows us to use floating point data as indexes for the texture cache. Without the ability to interpolate in hardware, we would be restricted to only integer benchmarks.

3. Texture Cache Approximation

Building on our previous work on load value approximation [8], this paper proposes to use the texture hardware to generate approximate values, which replace the exact values read from global memory and speed up execution by reducing memory stalls. Although many types of approximations are possible (such as taking the last value, or the average of the last N values), we choose to use *delta approximations*, as they provide the best output accuracy. In this work, we define a delta approximation as *the difference observed between two consecutively read global memory values*; this stems from our observation that input data that is either temporally or spatially correlated also exhibits strong *value locality*. The approximate value X_{apx} is the sum of the last exact data value and a delta approximation read from the texture cache: $X_{last}+D_{apx}$. Since the Kepler memory architecture defines texture memory as read-only, we cannot update any delta approximations at runtime; therefore we pre-load the texture cache with a training set of delta approximations that is generated by analyzing the data access patterns of each thread.

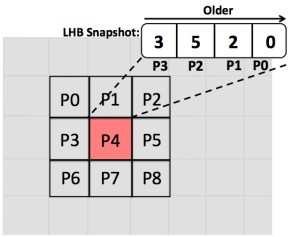
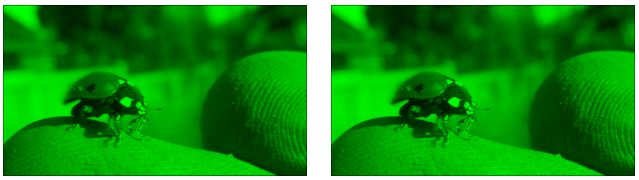


Figure 1: Texture cache approximation example using a local history buffer.

3.1 Offline Training

To generate our training set, we analyze the different deltas that appear throughout the lifetime of one thread. We give each thread its own local history buffer (LHB) in shared memory that stores the last *m* values read from global memory in FIFO fashion. Upon observing a memory access, we record the delta between the two most recent elements in the LHB, paired with the delta between the most recent LHB element and the value returned by the observed memory access. Figure 1 illustrates this process for an image processing kernel, where each thread reads a central pixel and the 8 pixels surrounding it. The current pixel under consideration is P4; assuming that it has a value of 4, then the *previous* delta (between the most recent LHB elements), would be *3 - 5 = -2*, and the *new* delta would be *4 - 3 = 1*.

The use of delta approximations provides good accuracy for applications whose data tends to be distributed inside a fixed range; if the input's data range is unbounded, it is possible for the training set to contain very large delta approximations that may introduce significant fluctuations in the final output. Our approach works well when the application target naturally exhibits data-level parallelism, where each thread processes independent items such as pixels in an image or synapses in a neural network. Data-parallel applications are naturally more error-robust as there is very little value sharing between threads. This prevents the case where a small number of poor approximations have a ripple effect on the data accessed by a large number of future computations. In general, texture cache approximation is well suited to data-intensive workloads such as image processing, physics and computational simulations,



(a) precise (b) approximate

Figure 2: Comparison of precise and approximate blur filter outputs (replacing 40% of global memory accesses).

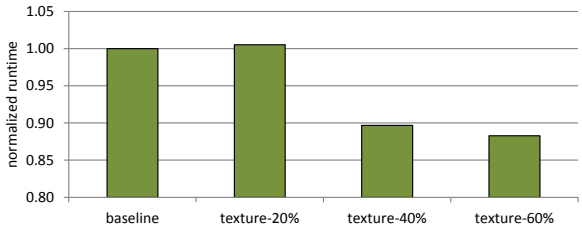


Figure 3: Runtime using texture approximations, normalized to precise execution baseline.

neural networks, and large graph queries. It is not recommended for applications that require exact solutions (eg. dense matrix operations), since a small discrepancy in the values of a matrix may result in an unsolveable problem and incorrect application control flow.

3.2 Online Approximations

In this work, we focus on kernels that contain a *characteristic loop*, which is the outermost loop that defines an identical operation being performed on separate members of the input set. For example, a financial options pricing benchmark might have a characteristic loop that solves the same equation for many different option parameters. We manually peel off N iterations of the characteristic loop to create an *epilogue* loop, in which all global memory accesses are replaced with approximate values generated via the texture cache. All iterations of the original characteristic loop that remain serve as a warmup period where the local history buffers are populated before generating approximations.

During the online execution of the approximate kernel, we keep the LHB in shared memory; the LHB stores the past values observed by this thread. On a texture cache reference, the index is calculated from the most recent two values in the LHB, and normalized to the minimum delta present in the training set. The texture cache is then accessed with this index, D_{apx} is returned, and then used to generate an approximate value. Our approach makes use of the fact that the texture cache index does not have to exactly match one contained in the training set, due to linear interpolation inside the fetch units. From the example in Figure 1, each time that a previous delta of *-2* is observed, a thread loads the *next* delta as 1 from texture memory and adds this delta to the most recent LHB value to generate the value used in place of a global memory read.

4. Evaluation

As a proof of concept that texture cache approximation can be successful applied on real hardware, we explore its use in a blur kernel derived from the San Diego CV Benchmark Suite [9], executing on an NVIDIA 780GTX. Our input was 16 HD frames (resolution 1920×1080), taken from a short stock film clip [1]. To obtain the kernel's execution time, we use the NVIDIA CUDA Toolkit Profiler [5], and obtain the number of cycles the kernel executed on an SM. Figure 3 displays the execution time (normalized to a precise kernel) of our approximate blur kernel, varying the number of texture cache approximations, while Figure 4 shows the per-frame error when we use the training set generated from the first frame to process all 16 images. We used mean pixel difference across all 3 RGB channels as our error metric, which has been used in prior work on approximation-induced image error [6].

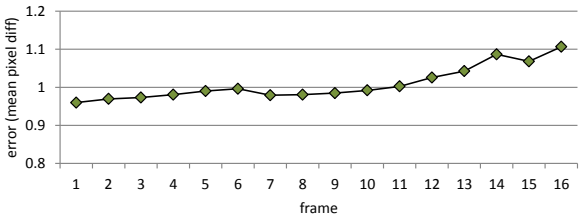


Figure 4: Per-frame error using texture approximations for 40% of global memory accesses.

By replacing 40% of the global memory accesses with texture cache approximations, we obtain a 12% decrease in kernel execution time, with a mean pixel error of 0.4%. This corresponds to a maximum mean pixel difference of 1.1 out of a maximum 8-bit channel value of 255. We observe errors of 0.1% and 1.3% when replacing 20% and 60% of memory accesses respectively. The average texture cache hit rate exceeds 99%, demonstrating that our training set is small enough to reside in the 12kB texture cache of each SM while still resulting in accurate images. Figure 2 compares the visual output of precise and approximate blur kernels to show that our technique generates output that appears consistent with precise kernel execution.

Since we use the same training set for all 16 frames, the mean pixel error increases for frames further away from the frame used to generate the training set as shown in Figure 4. Therefore, a real-time algorithm would necessitate re-training when the output error exceeds a certain threshold. Generating the new training sets could be easily overlapped with already executing kernels to avoid stalling the GPU.

Speedup is also possible by simply using a blur kernel that has less coefficients, without using our texture cache approach. However, depending on the application, it is possible that the output would change by a much greater degree than the slight errors in accuracy conceded by using value approximation. Additionally, this approach would not be valid for a more complex access pattern, such as those appearing in computational fluid dynamics, where many differential equations must be solved for each unit volume in the region of interest. Hardware value approximation is in fact harmonious with reduced precision algorithms, and ultimately the choice to use one instead of the other depends on the application and the goals of the systems designer.

5. Conclusion

In this paper, we present *texture cache approximation*, a technique that utilizes the on-chip texture cache to speed up general-purpose GPU workloads. By evaluating a specific image processing kernel, we show that the features of the texture cache are well-suited to the generation of approximate values. This implementation allows us to accelerate a standard image blur by 12%, with low output error despite the fact that we use the same training set for all frames in the application. Our work demonstrates that existing underutilized hardware structures on GPUs are suitable for use in approximation; with minor modifications, these structures can further support the approximate computing paradigm with little to no additional cost.

References

[1] Nature stock footage archive. `http://downloadnatureclip.blogspot.ca/p/download-links.html`. Accessed: 2015-03-28.

[2] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference (DAC)*, 2013.

[3] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *IEEE International Symposium on Parallel Distributed Processing*, 2009.

[4] *NVIDIA Cuda C Programming Guide*. NVIDIA Corp., . URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[5] *NVIDIA nvprof Users Guide*. NVIDIA Corp., . URL `http://docs.nvidia.com/cuda/profiler-users-guide/nvprof-overview`.

[6] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[7] M. Samadi, J. Lee, D. A. Jamshidi, and S. Mahlke. Paraprox: pattern-based approximation for data parallel applications. In *Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[8] J. San Miguel, M. Badr, and N. Enright Jerger. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[9] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego vision benchmark suite. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009.