# Analyzing Locality of Memory References in GPU Architectures

Saurabh Gupta, Ping Xiang, Huiyang Zhou
North Carolina State University
Raleigh, NC, USA
{sgupta12, pxiang, hzhou}@ncsu.edu

## ABSTRACT

In this paper we advocate formal locality analysis on memory references of GPGPU kernels. We investigate the locality of reference at different cache levels in the memory hierarchy. At the L1 cache level, we look into the locality behavior at the warp-, the thread block- and the streaming multiprocessor-level. Using matrix multiplication as a case study, we show that our locality analysis accurately captures some interesting and counter-intuitive behavior of the memory accesses. We believe that such analysis will provide very useful insights in understanding the memory accessing behavior and optimizing the memory hierarchy in GPU architectures.

## Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Heterogeneous (hybrid) systems

## General Terms

Algorithms, Performance.

## Keywords

GPGPU, matrix multiplication, tiling, locality of reference.

## 1. INTRODUCTION

Although graphics processing units (GPUs) rely on thread-level parallelism to hide long memory access latencies, the memory hierarchy, including on-chip caches, remain critical for many GPGPU (general purpose computing on GPUs) applications. In this paper, we advocate formal locality analysis to understand the nature of the memory access patterns of GPGPU applications and how they interact with the GPU memory hierarchy. We will first discuss our approach for locality analysis and then use matrix multiplication as a case study to show that interesting and somewhat unexpected memory access behavior can be revealed from our locality analysis.

## 2. APPROACH

To get a quantitative measure of locality, we use our probability based approach [2]. It defines locality as a conditional probability: given the condition that a memory address, X, is accessed, how likely the same address X or a neighbor address Y, where Y is within the neighborhood of X, will be accessed in the near future. With N being defined as the near future window size (N unique addresses) and K as the neighborhood size ($|Y-X| < K$), the spatial locality is represented as LS(N,K), which can be readily computed from an address stream for different Ns and Ks, resulting in a 3D mesh.

In current GPU memory hierarchy, each stream multiprocessor (SM) has an L1 cache and multiple SMs share an L2 cache, which is then connected to off-chip memory using multiple memory controllers. Since each L1 cache is shared by all the warps running on the SM and the threads are also managed in thread blocks, we analyze the locality behavior at the warp-level, the thread block level and the SM level to understand the impact on the L1 cache. Then, the L2 cache access stream is used to compute the locality at the L2 cache level. The L2 miss stream is also used to examine the locality behavior at the off-chip DRAM level.

In our study, GPGPU-Sim 3.1.1 [1] is used to generate various address streams of interests. Our GPU model is similar to NVIDIA GTX 480GPUs [3]. The L1 cache has the capacity of 16kB, the block size of 128 bytes and the set associativity of 4. The shared L2 cache has the capacity of 768kB, the block size of 128 bytes and the set associativity of 16. The L2 cache is shared among 15 SMs. The round-robin warp scheduling policy is used for all the warps in an SM.

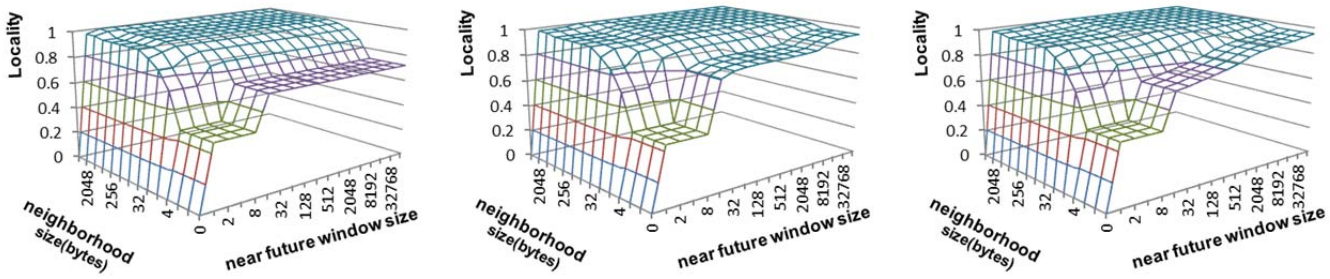
## 3. LOCALITY ANALYSIS ON MATRIX MULTIPLICATION

In this case study, we use the tiled matrix multiplication (MM) kernel from CUDA SDK [4]. We modified the kernel to use the L1 cache rather than the shared memory. The tile size is 16x16. Therefore each thread block contains 8 warps and 4 thread blocks can run concurrently in an SM. The locality curves computed at the warp level, the thread-block level, and the SM level are shown in Figure 1. The locality curves at different levels capture the reuse patterns accurately and can reveal interesting and somewhat unexpected results.

**At the warp level,** within a warp, the temporal locality varies as shown by curve for neighborhood size of 0 in Figure 1a. Since tile A's element A[0][0] gets multiplied with 16 elements from tile B (i.e. B[0][0~15]), the element A[0][0] is reused 15 times within warp 0. This extremely near reuse distance is shown in Figure 1a with a quick increase in temporal locality (LS(1,0)= 0.46). Since re-accesses to tile B occur after 16 accesses, the temporal locality jumps to 0.71 at the near-window size of 16 (LS (16, 0) = 0.71). There is also significant spatial locality present in the accesses to matrix B which shows up in the figure as increase in the locality as we move towards bigger neighborhood sizes (keeping near future window size fixed at say 1 or 2). This is because consecutive accesses to array B are from the same row (note that we assume row major data layout for 2D arrays).

**At the thread block level** (containing 8 warps), access patterns form several warps are interleaved, shown as follows.


*MSPC'13*, June, 2013, Seattle, Washington.




(a) Warp level (b) Thread-block level (c) SM-level

**Figure 1: Locality of Matrix Multiplication kernel at warp, thread-block level, SM level**

A[0][0]*B[0][0 ~ 15] – warp 0
A[1][0]*B[0][0 ~ 15] – warp 0
A[2][0]*B[0][0 ~ 15] – warp 1
…
A[14][0]*B[0][0 ~ 15] – warp 7
A[15][0]*B[0][0 ~ 15] – warp 7
---
A[0][1]*B[1][0 ~ 15] – warp 0
A[1][1]*B[1][0 ~ 15] – warp 0
…
A[15][1]*B[1][0 ~ 15] – warp 7
---
…
---
A[0][2]*B[2][0 ~ 15] – warp 0
…
….
A[0][15]*B[15][0~15] – warp 7

Since elements of matrix A again see multiple accesses causing the first jump in temporal locality in Figure 1b. It is interesting to see that all 16 elements in the row from tile B are again reused while they get multiplied by newer elements from tile A by subsequent warps. This causes accesses to matrix B have even higher temporal locality across warps at the thread block level (shown by LS(16, 0) = 0.80).

**At the SM level,** multiple thread blocks (4 in our case) are scheduled together and they share the L1 cache capacity. This can disrupt the data locality due to interference of data access patterns of multiple thread blocks sharing the same cache. In other words, if the cache in minimally sized to hold only the data accessed by one thread block, multiple co-scheduled thread blocks may end up thrashing the cache. Figure 1b and 1c compare the locality behavior of only one of the thread blocks executing in the SM and 4 thread blocks executing together. From the figures, we cannot see any significant difference in the locality plots and hence we can conclude that for MM, there is not any significant impact from multiple thread blocks being co-scheduled on the same SM. Clearly, this is counter intuitive as our initial expectation is that each of the four thread blocks will need 1 tile for A and 1 tile for B. Even with sharing among thread blocks when they have the same X position, 1 tile for A and 4 tiles for B will be needed by the 4 thread blocks. Therefore, the locality curve at the SM level should be quite different from the locality curve at the thread block level. So, what did we miss here?

From the access patterns at the thread block level, we can see that accesses to tile A enjoy spatial locality, i.e., if the cache is large enough to hold 16 elements from tile A (i.e. A[0][0], A[1][0], … A[15][0]) and the cache block is big enough to hold multiple data elements, the access A[0][1], A[1][1], … A[15][1] will be hits due to spatial locality. Consider the capacity requirement for achieving good performance from the tiled implementation in this case, only 16 cache blocks from tile A need to be cached. On the other hand, the 16 contiguous elements need to be cached from tile B to get the performance. In our GPU, the cache block size is 128-bytes and therefore all 16 elements (4-bytes each) fit in the cache block. Therefore, only 16 cache blocks for tile A + one cache block for tile B = 17 cache blocks in L1 cache is enough to get performance from the tiled MM on GPU.

Now we move on to analyzing the locality of the memory reference stream observed by the L1 cache of an SM in the GPU when tiled MM is used. The GPU in our experiments uses round-robin scheduling policy to schedule thread blocks on SMs and within a thread block it uses round robin scheduling as well for scheduling warps. As multiple thread blocks are sharing the same L1 cache, the requirement of L1 cache capacity is increased as well. Though depending upon which thread blocks get co-scheduled, there is a possibility of sharing one of the tiles (either A or B) among the multiple co-scheduled thread blocks. In our experiments we observed that the co-scheduled thread blocks share the same tile A. Interestingly, this causes the capacity requirement to increase only by very small amount. The reason is that we only need to cache 16 cache blocks of tile A and only 1 cache block for each thread block scheduled for their respective tile B. This is the fundamental reason why the locality curves in Figure 1b and Figure 1c do not show any significant difference.

# 4. CONCLUSIONS AND FUTURE WORK

In this paper, we advocate locality analysis on memory references of GPGPU applications and showcase the locality study on the matrix multiplication kernel. We show that our locality curves capture the nature of the access patterns accurately and also reveal interesting and somewhat unexpected results, regarding multiple thread blocks sharing the L1 cache in an SM. In our future work, we plan to optimize the memory hierarchy based on the insights revealed from the locality analysis. We also plan to extend this work to understand the interactions of locality when multiple kernels share the last level of cache in a GPU.

# 5. REFERENCES

[1] A. Bakhoda, et al. Analyzing CUDA workloads using a detailed GPU simulator. In IPASS 2009.

[2] S. Gupta et al. Locality Principle Revisited: A Probability-Based Quantitative Approach. In IPDPS 2012.

[3] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture. 2009; http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[4] NVIDIA GPU Computing SDK 3.1; https://developer.nvidia.com/cuda-toolkit-31-downloads.