CampProf: A Visual Performance Analysis Tool for Memory Bound GPU Kernels

Ashwin M. Aji, Mayank Daga, Wu-chun Feng
Dept. of Computer Science
Virginia Tech
Blacksburg, USA
{aaji, mdaga, feng}@cs.vt.edu

***Abstract*—Current GPU tools and performance models provide some common architectural insights that guide the programmers to write optimal code. We challenge these performance models, by modeling and analyzing a lesser known, but very severe performance pitfall, called 'Partition Camping', in NVIDIA GPUs. Partition Camping is caused by memory accesses that are skewed towards a subset of the available memory partitions, which may degrade the performance of memory-bound CUDA kernels by up to *seven-times*. No existing tool can detect the partition camping effect in CUDA kernels.**

We complement the existing tools by developing 'CampProf', a spreadsheet based, visual analysis tool, that detects the degree to which any memory-bound kernel suffers from partition camping. In addition, CampProf also predicts the kernel's performance at all execution configurations, if its performance parameters are known at any one of them. To demonstrate the utility of CampProf, we analyze three different applications using our tool, and demonstrate how it can be used to discover partition camping. We also demonstrate how CampProf can be used to monitor the performance improvements in the kernels, as the partition camping effect is being removed.

The performance model that drives CampProf was developed by applying multiple linear regression techniques over a set of specific micro-benchmarks that simulated the partition camping behavior. Our results show that the geometric mean of errors in our prediction model is within 12% of the actual execution times. In summary, CampProf is a new, accurate, and easy-to-use tool that can be used in conjunction with the existing tools to analyze and improve the overall performance of memory-bound CUDA kernels.

***Keywords*-CUDA; Partition Camping; Analysis; Optimization; NVIDIA GPU's**

I. Introduction

Graphics Processing Units (GPUs) are being increasingly adopted by the high-performance computing (HPC) community due to their remarkable performance-price ratio, but a thorough understanding of the underlying architecture is still needed to optimize the GPU-accelerated applications [1]. Several performance models have recently been developed to study the organization of the GPU and accurately predict the performance of the GPU-kernels [2]–[5]. Our paper challenges and complements the existing performance models, by modeling and analyzing a lesser known but extremely severe performance pitfall, called *Partition Camping*, in NVIDIA GPUs.

Partition Camping is caused by memory accesses that are skewed towards a subset of the available memory partitions, which may severely affect the performance of memory-bound CUDA kernels [6], [7]. Our studies show that the performance can degrade by up to *seven-times* because of partition camping (Figure 3). While common optimization techniques for NVIDIA GPUs have been widely studied, and many tools and models guide programmers to perform common intra-block optimizations, neither the current performance models nor existing tools can discover the partition camping effect in CUDA kernels.

We develop *CampProf*, which is a new, easy-to-use, and a spreadsheet based visual analysis tool for detecting partition camping effects in memory-bound CUDA kernels. The tool takes the kernel execution time and the number of memory transactions of any one kernel configuration as the input, and displays a range of execution times for all the other kernel configurations (Figure 1). The upper and lower bounds indicate the performance levels with and without the partition camping problem respectively. The relative position of the actual execution time with respect to the performance range will show the *degree* to which the partition camping problem exists in the kernel. In addition, CampProf also predicts the exact execution times of the kernel at *all* the other execution configurations. We recommend CampProf to be used in conjunction with the other existing tools, like CUDA Occupancy Calculator [8] and CUDA Visual Profiler (CudaProf) [9], to analyze the overall performance of memory-bound CUDA applications.

CampProf uses a performance prediction model, which we developed by first creating several micro-benchmarks that captured the performance of all the different memory transaction types, with and without the partition camping behavior. Based on the execution times of the micro-benchmarks and the different memory transactions, we used multiple linear regression to model the performance range of actual CUDA kernels. To demonstrate the utility of CampProf in real applications, we analyze three very different memory-bound CUDA kernels, and show how our tool and model can be used to discover the partition camping problem. We also demonstrate how the tool can be used to monitor the performance of the kernel, where the execution time progresses towards the best case after the partition camping effect has been reduced. Also, we show that our performance prediction model has a geometric mean error of less than 12% when validated against the actual kernel execution times.

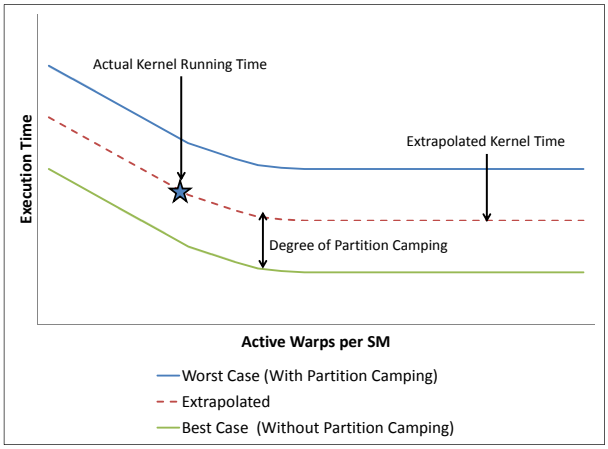


Fig. 1. Conceptual Output Chart of CampProf.

The rest of this paper is organized as follows: Section II provides background on the NVIDIA GPU, the CUDA architecture and the partition camping problem. Section III presents the related work. Section IV described the CampProf tool in detail, followed by performance modeling techniques using micro-benchmarks and statistical analysis tools. Section V explains the experimental setup. Section VI discusses the experimental results. Section VII concludes the paper and proposes some future work.

II. Background on the NVIDIA GPUs

In this section, we explain the basic architecture of the general NVIDIA GPU, the CUDA programming model, common optimization techniques and the lesser known partition camping problem.

A. The NVIDIA GPUs and CUDA

The NVIDIA GPU (or *device*) consists of a set of Single Instruction Multiple Data (SIMD) streaming multiprocessors (SMs), where each SM consists of eight scalar processor (SP) cores, two special function units and a double precision processing unit with a multi-threaded instruction unit. The actual number of SMs vary depending on the different GPU models.

The SMs on the GPU can simultaneously access the *device memory*, which consists of read-write global memory, 64 KB of read-only constant memory and read-only texture memory. However, all the device memory modules can be read or written to by the *host* processor. Each SM has on-chip memory, which can be accessed by all the SPs within the SM and will be one of the following four types: a set of registers; 16 KB of 'shared memory', which is a software-managed data cache; a read-only constant memory cache; and a read-only texture memory cache. The global memory space is not cached by the device.

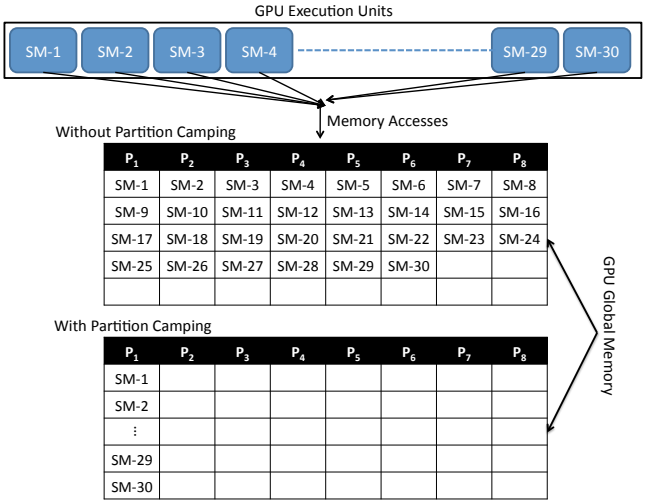
CUDA (Compute Unified Device Architecture) [6] is the parallel programming model and software environment provided by NVIDIA to run applications on their GPUs, programmed via simple extensions to the C programming language. CUDA follows a code off-loading model, i.e. data-parallel, compute-intensive portions of applications running on the host processor are typically off-loaded onto the device. The *kernel* is the portion of the program that is compiled to the instruction set of the device and then off-loaded to the device before execution.

Execution Configuration of a CUDA Kernel: The threads in the kernel are hierarchically ordered as a logical grid of thread blocks, and the CUDA thread scheduler will schedule the blocks for execution on the SMs. When executing a block on the SM, CUDA splits the block into groups of 32-threads called *warps*, where the entire warp executes one common instruction at a time. CUDA schedules blocks (or warps) on the SMs in batches, and not all together, due to register and shared memory resource constraints. The blocks (or warps) in the current batch are called the *active* blocks (or warps) per SM. The CUDA thread scheduler treats all the active blocks of an SM as a unified set of active warps ready to be scheduled for execution. In this way, CUDA hides the memory access latency of one warp by scheduling another active warp for execution [6], [7]. In short, a kernel with an arbitrary number of blocks will perform only as good as the kernel with a configuration equal to set of active warps. In this paper, we have chosen 'active warps per SM' as the metric to describe the execution configuration of any kernel, because it is much simpler to be represented in only a single dimension.

There are some hardware restrictions imposed on the NVIDIA GPUs with compute capability 1.3 that limits the possible number of active warps that can be scheduled on each SM. The warp size for the current GPUs is 32 threads. The maximum number of active threads per multiprocessor can be 1024, which means that the maximum number of active warps per SM is 32. Also, the maximum number of threads in a block is 512, and the maximum number of active blocks per multiprocessor is 8 [6]. Due to a combination of these restrictions, the number of active warps per SM can range anywhere from 1 to 16, followed by even-numbered warps from 18 to 32.

B. The Partition Camping Problem

Optimization techniques for NVIDIA GPUs have been widely studied, and many proprietary tools, like CUDA Visual Profiler (CudaProf) and the CUDA Occupancy Calculator spreadsheet tool, guide programmers to perform common intra-block optimizations. These include optimizing arithmetic instruction throughput, efficiently accessing global memory, and avoiding bank conflicts in shared memory. In this paper, we study a lesser known performance pitfall, which NVIDIA calls 'partition camping', where memory requests across blocks get serialized by fewer memory controllers on the graphics card (Figure 2). Just as shared memory is divided into multiple banks, global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200- and 10-series GPUs) of 256-byte width. The partition camping problem is similar to shared memory bank conflicts,



P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
SM-1	SM-2	SM-3	SM-4	SM-5	SM-6	SM-7	SM-8
SM-9	SM-10	SM-11	SM-12	SM-13	SM-14	SM-15	SM-16
SM-17	SM-18	SM-19	SM-20	SM-21	SM-22	SM-23	SM-24
SM-25	SM-26	SM-27	SM-28	SM-29	SM-30		

P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
SM-1							
SM-2							
⋮							
SM-29							
SM-30							

Fig. 2. Partition Camping effect in 200- and 10-series NVIDIA GPUs. Note: The column P_i denotes the i^{th} partition.

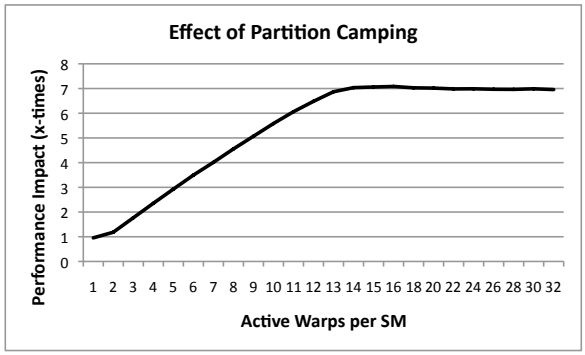


Fig. 3. Partition Camping in Micro-Benchmark

but experienced at a macro-level where concurrent global memory accesses by all the active warps in the kernel occur at a subset of partitions, causing requests to queue up at some partitions while other partitions go unused [7]. Our micro-benchmarks simulate the partition camping effect; one benchmark accesses the memory uniformly while the other accesses just one partition of the memory. They reveal that the performance of memory-bound kernels can degrade by up to *seven-times* if memory request suffer from partition camping, as shown in Figure 3.

Discovery of the partition camping problem in CUDA kernels is a difficult problem. There is existing literature on static code analysis for detecting bank conflicts in shared memory [10], but the same logic cannot be extended to detecting the partition camping problem. Bank conflicts in shared memory occur among threads in a warp, where all the threads share the same clock, and an analysis of the accessed address alone is sufficient to detect conflicts. However, the partition camping problem occurs when multiple active warps queue up behind the same partition *and at the same time*. This means that a static analysis of just the partition number of each memory transaction is not sufficient, and its timing information should also be analyzed. Each SM has its own private clock, which makes the discovery of this problem much more difficult and error prone. To the best of our knowledge, there are no tools that can diagnose the partition camping problem in CUDA kernels.

III. Related Work

There have been analytical models developed to help the programmer understand bottlenecks and achieve optimum performance on the GPU. In [2], Baghsorkhi et al. have developed a compiler front end which analyses the kernel source code and translates it into a Program Dependence Graph (PDG) which is useful for performance evaluation. The PDG allows them to identify computationally related operations which are the dominant factors affecting the kernel execution time. With the use of symbolic evaluation, they are able to estimate the effects of branching, coalescing and bank conflicts in the shared memory.

In [3], Hong et al. propose an analytical model which dwells upon the idea that the execution of any kernel is bottlenecked by the latency of memory instructions and multiple memory instructions are executed to successfully hide this latency. Hence, calculating the number of parallel memory operations (memory warp parallelism) would enable them to accurately predict performance. Their model relies upon the analysis of the intermediate PTX code generated by the CUDA compiler. However, the PTX is just an intermediate representation which is further optimized to run on the GPU [11]. PTX not being a good representation of the actual machine instructions introduces some error in their prediction model.

Recently Ryoo et al. proposed two metrics; efficiency and utilization to prune the optimization space of general purpose applications on the GPU [12]. Their model, however, does not work for memory bound kernels. Boyer et al. present an automated analysis technique to detect race conditions and bank conflicts in a CUDA program. They do so by instrumenting the program to track the memory locations accessed which is done by analyzing the PTX code [10]. Schaa et al. focus on the prediction of execution time for a multi-GPU system, knowing the time for execution on a single GPU [13]. They do so by introducing models for each component of the multi-GPU system; the GPU execution, PCI-Express and the RAM and the Disk.

Micro-benchmarks have been extensively used to reveal the architectural details of the GPUs. In [14], Volkov et al. benchmark the GPUs to tune dense linear algebra. They created detailed benchmarks to reveal the kernel bottlenecks like access patterns of the GPU Shared memory and kernel launch overhead. Their benchmarks also portrayed the structure of the GPU memory system including the access latencies. Wong et al. also use micro-benchmarks to understand the micro-architecture of the GT200 GPU, the one used in this current work [4]. Both these works, used decuda which is a disassembler for NVIDIA's machine level instructions, to understand the mapping of various instructions on the GPU [15].

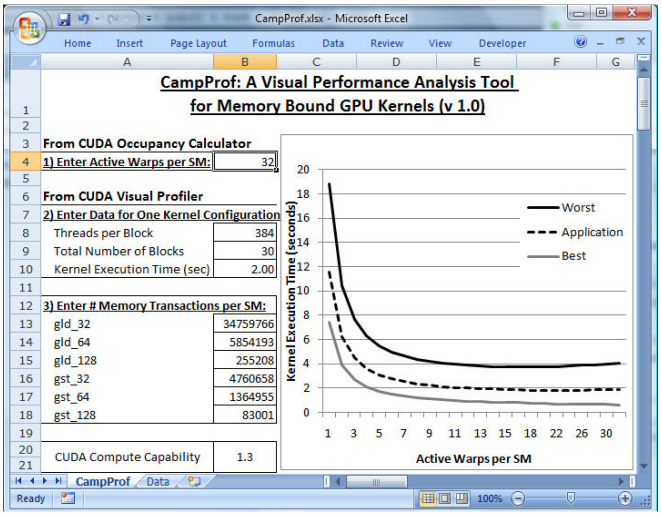


Fig. 4. CampProf: Screenshot

In [5], Hong et al. propose an Integrated Power and Performance Model for GPUs where they use intuition that once an application reaches the optimum memory bandwidth, increasing the number of cores would not help the performance of the application and hence, power can be saved by switching off the additional cores of the GPU. Nagasaka et al. make use of statistical tools like regression analysis and CudaProf counters for power modeling on the GPU [16]. Their approach is very similar to ours, however, they model power and we model performance. Bader et al. have developed automated libraries for data re-arrangement to explicitly reduce partition camping problem in the kernels [17].

In the present work, we develop a performance model which reveals the extent of partition camping in an application as well as predict the execution time for all kernel configurations if the time for one configuration is provided. Micro-benchmarks and multiple linear regression formed the basis of our model. We simplify performance prediction by developing a spreadsheet like tool which takes CudaProf counters as input parameters and envisions the worst and best possible execution times for any memory bound application.

IV. Performance Prediction with CampProf

In this section, we first describe CampProf and its several utilities at a high level, and also mention the class of CUDA kernels that can benefit from this tool. Next, we show how we have used micro-benchmarks and statistical analysis tools to develop CampProf. We then discuss how our ideas can be applied to develop similar tools for the newer GPU architectures.

A. Overview

CampProf is the visual analysis tool that we have developed, and it offers the following utilities to the programmer of memory bound CUDA kernels –

1) *Partition Camping Detection:* CampProf takes the kernel execution time, and the number of memory transactions of any one kernel configuration as inputs, and displays a range of execution times for all the other kernel configurations. The bounds range from the best case running times (assuming no partition camping) to the worst case running times (assuming partition camping). The relative position of the actual execution time with respect to the performance range will show the *degree* to which the partition camping problem exists in the kernel. This is in contrast to the other existing performance models that predict just a single kernel execution time, but those models can be applied only to the kernels that are known to not suffer from partition camping. By detecting the effect of partition camping, the programmer can estimate the target for performance improvement, if the memory access patterns in the kernel are carefully managed. We are not aware of any tool that provides this utility.
2) *Performance Prediction:* In addition to the previous utility, CampProf inputs the actual execution time and the execution configuration of the given kernel, and predicts the exact running times of the same kernel at *all* the other execution configurations.

Note that CampProf provides insights into this largely ignored partition camping problem, and *not* the other common performance pitfalls that the CUDA programming guide describes, like non-coalesced global memory accesses, shared memory bank conflicts, low arithmetic intensity, etc. The performance counter values from the CudaProf [9] tool (`gld_32/64/128b`, `gst_32/64/128b`, `instructions`, `warp serialize` (for bank conflicts in shared memory), etc) can be used to understand and optimize the common performance bottle-necks of the kernel. However, these values describe the kernel's behavior either within a single SM or a single TPC (depending on the profiler counter), and does not provide any details of the overall system. On the other hand, CampProf helps understand the memory access patterns among all the active warps in the entire kernel. We recommend CampProf to be used by the CUDA programmer along with CudaProf and the CUDA Occupancy Calculator, to tune their program to the optimum.

CUDA Kernel Classification: We now categorize the CUDA kernels into two general groups and describe the class of kernels that CampProf supports. We classify CUDA kernels as either being compute-bound or memory-bound. Compute-bound kernels typically have a high arithmetic intensity, and so most of the memory transactions by the warps take negligible time when compared to the overall kernel time, i.e. most of the memory transactions are hidden by other active warps being scheduled for execution on the SM by the CUDA thread scheduler. On the other hand, memory-bound kernels have low arithmetic intensity, where the kernel's execution time is mainly dominated by the memory accesses. The core computations are negligible and will be mostly hidden by the active warps that are accessing the memory [6]. The partition camping problem significantly affects the performance of the

kernel's memory transactions, which mainly affects memory-bound kernels. So, CampProf can be used to analyze all the memory-bound CUDA kernels.

But, how do we determine if a kernel is memory or compute-bound? Our solution is to analyze trends in the kernel execution time when the GPU's core-clock and memory frequencies are changed. If the execution time varies significantly when the core-clock frequency is changed, then the kernel is likely to be compute-bound. On the other hand, if there is a significant change in the running time when the memory frequencies are varied, then the kernel is likely to be memory-bound. It is hard to strictly classify a kernel as compute-bound or memory-bound, and some kernels can be a bit of both. We will show in Section V-B that the performance of our case-study applications shows significant changes when the memory frequency of the GPU card is changed from 800MHz to 1300MHz, and so can be considered to be memory-bound kernels and good candidates for our experiments. We also realize that our tool cannot be used to analyze the partition camping effects in kernels that are not memory-bound. But, in such cases, the effect of memory transactions will not even be significant when compared to the total execution time of the kernel.

The CampProf Tool: CampProf is a spreadsheet tool similar to the CUDA Occupancy Calculator [8] and its screenshot is shown in Figure 4. The spreadsheet consists of some input fields and the output chart, which shows the partition camping effects. The inputs to CampProf are the following values that can be easily obtained from the CudaProf and the CUDA Occupancy Calculator tools: `gld 32b/64b/128b`, `gst 32b/64b/128b`, grid and block sizes, and active warps per SM. The active warps per SM is obtained from the Occupancy Calculator, and it depends on the shared memory usage per block, registers per thread and threads per block. Note that inputs from just a single set of threads and blocks are enough for CampProf to predict the kernel's performance range for any other execution configuration. We deviate from the other static analysis work that has been done so far in this field, but provide more realistic performance estimates. CampProf passes the input values to our performance model, which is a multiple linear regression model, which then generates the following two sets of predicted execution times for *all* the possible kernel execution configurations[1] – (1) best case, assuming that all the memory transactions by the system's active warps are uniformly distributed amongst the available memory partitions, and there is no partition camping, and (2) worst case, if all the active warps access the same memory partition, which leads to the partition camping problem. CampProf then plots these two sets of predicted execution times as two lines in a chart in the spreadsheet. The best and worst case execution times form a band between which the actual execution time lies.

To detect the partition camping problem in the kernel, the user can simply collect the actual kernel execution time (`GPU Time` from CudaProf) for the same set of input parameters, and compare it with CampProf's execution band. If the actual running time is almost touching the worst case line, it means that all the memory transactions of the kernel (reads and writes of all sizes) are partition camped, and the warps of the kernel should be re-mapped to the data to avoid this effect. Likewise, the kernel is considered to be optimized with respect to partition camping if the actual execution time is very close to the lower best case line. If the actual running time is somewhere in the middle of the two lines, it means that performance can be potentially improved, and there is a subset of memory transactions (reads or writes) that is queuing up behind the same partition. For example, while processing two matrices, the kernel might read one matrix in the row major format (has no partition camping) and the other matrix might be read or written into in the column major format (has partition camping). This means that only a part of the kernel suffers from camping, and the actual execution time will lie somewhere between the two extremities of CampProf's execution band. Detailed results will be explained in Section VI. The only remedy to the partition camping problem is careful analysis of the CUDA code and re-mapping the thread blocks to the data, as explained in 'TransposeNew' example of the CUDA SDK [7].

CampProf further predicts the execution time of the kernel at all the possible execution configurations, by extrapolating the actual execution time of the kernel at the input configuration proportionally to the execution band. This will give the user an estimate of the effect of the partition camping problem at different thread and block configurations.

B. Performance Prediction via Micro-benchmarks

We developed micro-benchmarks to predict the performance range of the memory-bound CUDA kernel, given the number of memory reads (`gld_32, gld_64, gld_128`) and memory writes (`gst_32, gst_64, gst_128`) for all the execution configurations, i.e. for all possible active warps per SM. We first write different micro-benchmarks that each trigger a different type of memory transaction, where the number of transactions is arbitrary, but fixed and known. More specifically, one set of benchmarks will contain three benchmarks for the reads and three for the writes, where the active warps are distributed across all the memory partitions uniformly and have no partition camping. For each of these benchmarks, we write another set of benchmarks that explicitly access a single memory partition, to mimic the partition camping effect. For all the above benchmarks, we vary the active warps per SM from 1 to 32 and record the kernel execution times. We then use a multiple linear regression model over the entire collected data set and formulate *four* model equations. We use the model equations to calculate the running times of any memory-bound kernel having all the different memory transaction type, for any word length and execution configuration, with and without partition camping. We use these micro-benchmarks and prepare a set of the model

[1] As mentioned in Section II-B, the 'number of active warps per SM' is our chosen metric of kernel configuration.

equations for each of the supported GPU architectures. Currently, we support the GPUs with compute capability 1.3. Once the user enters their application specific input data, which they collect from CudaProf and the occupancy calculator, CampProf will aggregate the predicted running times for all the memory access types (reads and writes of different word lengths) and present the results in CampProf's chart tool. As mentioned before, the upper line will represent the worst case with partition camping, and the lower line will be without partition camping.

1) Micro-benchmarks: Figures 5 and 6 show the CUDA kernel of the micro-benchmarks for memory reads, without and with partition camping respectively. The micro-benchmarks for memory writes are very similar to the memory reads, except that `readVal` is written to the memory location inside the for-loop (line numbers 28 and 21 in the respective code snapshots). When all the threads in a half warp read 1 or 2 byte words, and the memory request is coalesced, then it results in a 32 byte memory transaction to global memory. Similarly, 4-byte and 8-byte coalesced memory requests per thread will translate to 64-byte and 128-byte transactions to global memory respectively [6]. So, we modify the data types in the benchmarks from `short int` (to represent 2 bytes) to `int` (4 bytes) and `double` (8 bytes), in order to trigger 32 byte, 64 byte and 128 byte memory transactions respectively to the global memory. The set of benchmarks that mimic the partition camping effect carefully access memory from a single partition. Note that the above global memory transactions can also be triggered in a multitude of other ways, but their performance will not be different. For example, a 32-byte transaction can also be invoked if all the threads in a half warp access the same memory location (char/short int/int/double). The performance of any memory transaction does not depend on its invocation method, and we tested this fact by writing a simple micro-kernel, which we have not included in this paper for brevity. So, our benchmarks form a good representation of real memory-bound kernels.

Validity Analysis: We now check if the micro-benchmarks are truly memory bound before providing the data to the statistical tools. We again analyze trends in the kernel execution time when the GPU's core-clock and memory frequencies are changed. Figures 7 shows that the performance decreases steadily with the increase in the core-clock frequency until about 550 MHz. But, the change in the running times become insignificant for higher frequencies including and around the default core-clock frequency of 602 MHz [18]. Moreover, we see a large variation in performance for different memory frequencies, proving the memory-boundedness of our benchmarks.

2) Multiple Linear Regression: We perform multiple linear regression analysis to fully understand the relationship between the execution time of our micro-benchmarks and its parameters. The independent variables (predictors) that we chose are: (1) the active warps per SM (w), and (2) the word-lengths that are read or written per thread. The dependent variable (response) is the execution time (t). The word-length predictor takes only three values (2, 4 or 8 bytes)[2], and so

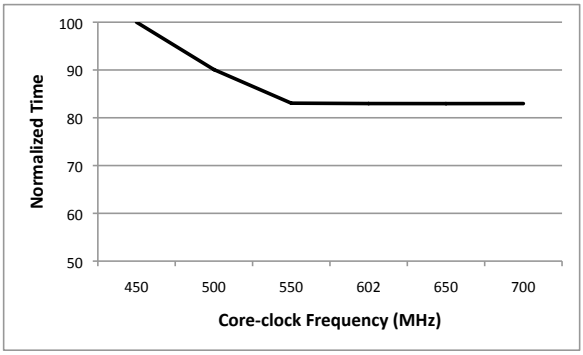
```
1
2  // TYPE can be short int, int, or double
3  typedef int TYPE;
4
5  __global__ void readBenchmark(TYPE *d_arr)
6  {
7      // assign unique partitions to blocks,
8      // where number of partitions is 8
9      int curPartition = blockIdx.x % 8;
10     // size of each partition: 256 bytes
11     int elemsInPartition = 256 / sizeof(TYPE);
12     // jump to unique partition
13     int startIndex = elemsInPartition
14                             * curPartition;
15
16     TYPE readVal = 0;
17     /* ITERATIONS = 8192, but it can be any
18         fixed and known number. Loop counter 'x'
19         ensures coalescing */
20     for(int x = 0; x < ITERATIONS; x += 16)
21     {
22         /* offset guarantees to restrict the
23            index to the same partition */
24         int offset = ((threadIdx.x + x)
25                         % elemsInPartition);
26         int index = startIndex + offset;
27         // Read from global memory location
28         readVal = d_arr[index];
29     }
30     /* Write once to memory to prevent the above
31        code from being optimized out */
32     d_arr[0] = readVal;
33 }
```

Fig. 5. Code Snapshot of the Read Micro-benchmark for the NVIDIA 200- and 10-series GPUs (Without Partition Camping).

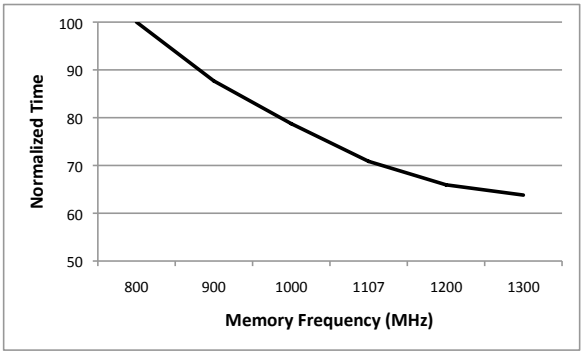
```
1
2  //   TYPE can be short int, int, or double
3  typedef int TYPE;
4
5  __global__ void readBenchmark(TYPE *d_arr)
6  {
7      // size of each partition: 256 bytes
8      int elemsInPartition = 256 / sizeof(TYPE);
9
10     TYPE readVal = 0;
11     /* ITERATIONS = 8192, but it can be any
12         fixed and known number. Loop counter 'x'
13         ensures coalescing */
14     for(int x = 0; x < ITERATIONS; x += 16)
15     {
16     /* all blocks read from a single partition
17         to simulate Partition Camping */
18         int index = ((threadIdx.x + x)
19                         % elemsInPartition);
20         // Read from global memory location
21         readVal = d_arr[index];
22     }
23     /* Write once to memory to prevent the above
24        code from being optimized out */
25     d_arr[0] = readVal;
26 }
```

Fig. 6. Code Snapshot of the Read Micro-benchmark for the NVIDIA 200- and 10-series GPUs (With Partition Camping).

[2]1- and 2-byte word lengths will both result in 32-byte global memory transactions.



(a) Micro-benchmark: Time vs GPU Core-clock Frequency



(b) Micro-benchmark: Time vs GPU Memory Frequency

Fig. 7. Normalized Execution Time at Various Core-clock and Memory Frequencies

we treat it as a group variable (b). This means, we first split the data-type variable into two binary variables (b_2 and b_4), where their co-efficients can be either 0 or 1. If the co-efficient of b_2 is set, it indicates that the word-length is 2-bytes. Likewise, setting the co-efficient of b_4 indicates a 4-byte word-length, and if co-efficients of both b_2 and b_4 are not set, it indicates the 8-byte word-length. Having decided our model variables, we use SAS, a popular statistical analysis tool to perform the regression analysis for our data. SAS helps us to derive a model such as the equation 1, where α_i denotes the contribution of the different predictor variables to our model, and β is the constant intercept.

$$t = \alpha_1 w + \alpha_2 b_2 + \alpha_3 b_4 + \beta \quad (1)$$

Significance Test: The output of SAS also shows us the results of some statistical tests, which describe the significance of our model, and how well our chosen model variables are contributing to the overall model. In particular, R^2[3] ranges from 0.953 to 0.976 and $RMSE$ (Root Mean Square Error) ranges from 0.83 to 5.29. Moreover, we also used parameter selection techniques in SAS to remove any non-significant variable, and choose the best model. This step did not deem any of our variables as insignificant. These results mean that the response variable (execution time) is strongly dependent on the predictor variables (active warps per SM, data-types), and each of the predictors are significantly contributing to the response, which proves the strength of our performance model. We evaluate our prediction model for real applications in detail in Section VI.

[3] R^2 is a descriptive statistic for measuring the strength of the dependency of the response variable on the predictors

C. Limitations of CampProf

CampProf can be used to discover the effect of partition camping in GPU architectures with compute capability 1.2 or 1.3. We do not support devices with a lower compute capability. The architectural changes in the newer Fermi [19]cards pose new problems for memory-bound kernels. The memory access patterns are significantly different for Fermi, because of the introduction of L1/L2 caches and having only 128-byte memory transactions that occur only at the cache line boundaries. The partition camping problem may still exist in the newer cards, but its effect may be somewhat skewed due to cached memory transactions. As a preliminary investigation, we tried to disable the caches in Fermi to test the partition camping problem, but we found options to disable only the L1 cache and not the L2 cache.

CampProf may not support Fermi, but our visual analysis technique will still work, though in a different scenario. For example, we could use our ideas of displaying a performance band for the Fermi cards, and visually show the effect of cache misses and larger memory transactions for a given execution configuration. This can then help in understanding the gains that can be achieved by improving the locality of memory accesses in the kernel. Our immediate goal is to extend CampProf to *CacheProf* for the the Fermi architecture.

V. Experimental Setup

In this section, we describe our hardware setup, and describe the applications and their execution profiles, which we use to validate CampProf.

A. Hardware

The host machine consists of an E8200 Intel Quad core running at 2.33 GHz with 4 GB DDR2 SDRAM. The operating system on the host is a 64-bit version of Ubuntu 9.04 distribution running the 2.6.28-16 generic Linux kernel. The GPU was programed via the CUDA 3.1 toolkit with the NVIDIA driver version 256.40. We ran our tests on a GeForce GTX 280 graphics card (GT200 series). The GTX 280 has 1024 MB of onboard GDDR3 RAM as global memory. The card has the core-clock frequency of 602 MHz, processor (shader) clock frequency of 1296 MHz and memory frequency of 1107 MHz [18]. This card belongs to compute capability 1.3. For the sake of accuracy of results, all the processes which required graphical user interface (GUI) were disabled to limit resource sharing of the GPU.

B. Applications

In order to test the utility of CampProf, and the efficacy of our performance prediction model, we chose the

following three applications, which are very different from each other in their execution profiles – (1) GEM (Gaussian Electrostatic Model) [20] is a molecular modeling application, (2) GALib (Graph Analysis LIBrary) contains several graph related applications, out of which we chose the Clique-Counter application, and (3) the matrix transpose application. GEM and Clique-Counter (of GALib) are complete applications, while the matrix transpose is part of the NVIDIA CUDA SDK. We now briefly describe the execution profiles of these applications, and then show that each of these is a memory-bound application, which is a necessary condition necessary for CampProf.

1) GEM: GEM is a molecular modeling application which allows one to visualize the electrostatic potential along the surface of a macro-molecule [20]. GEM belongs to the N-Body class of applications (or dwarfs[4]). The goal of this application is to compute the electrostatic potential at all the surface points due to the molecular components. GEM uses an approximation algorithm to speed up the electrostatic calculations by taking advantage of the natural partitioning of the molecules into distant higher level components. The algorithm is presented below:

Algorithm: GEM with the Approximation Algorithm

```
for v = 0 to #Surface Points do
  for c = 0 to #Higher Level Components do
    if (dist(v, c) >= threshold) then
      potential += calcPotential(v, c)
    else
      for a = 0 to #Atoms in c do
        potential += calcPotential(v, a)
      end for
  end for
end for
```

Each GPU thread is assigned to compute the electrostatic potential at one surface point, which is later added together to calculate the total potential for the molecule. The approximation algorithm requires the distance information between each surface point and the higher level components. To compute this distance, each thread needs to access the component coordinates, which are stored in the GPU's global memory. Therefore, every computation results in multiple memory accesses, contributing to GEM being memory-bound, as proved later.

2) GALib: We have chosen the *Clique-Counter* application from the GALib library for our validation. In graph theory, a clique is a set of vertices in a graph, where every two vertices in the set are connected by an edge of the graph. Cliques are one of the basic concepts of graph theory and are one of the fundamental measures for characterizing different classes of graphs. We identify the size of a clique by the number of vertices in the clique. Clique Counter is a program, which as the name suggests, counts the number of cliques of user-specified size in a graph. This is an NP-complete problem, with respect to the size of the clique that must be counted.

[4]A dwarf or a motif is a common computation or communication pattern in parallel algorithms [21].

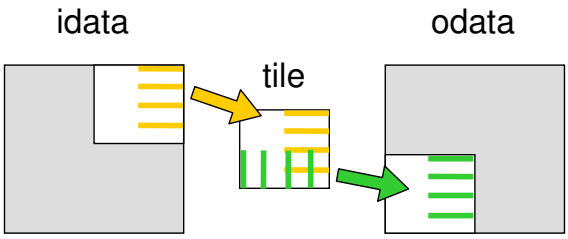


Fig. 8. Matrix Transpose (Source: [7])

The vertices of the input graph are distributed among the CPU threads in a cyclic fashion for load balancing purposes, where the entire graph is stored in the GPU's global memory in the form of adjacency lists. Each thread counts the number of cliques of the given size that can be formed from its set of vertices, followed by a reduction operation that sums up the individual clique counts to get the final result. Larger cliques are formed from smaller cliques by incrementally adding common vertices to the clique. The Clique Counter application belongs to the 'backtracking' class of algorithms (or dwarfs), where set intersection operations are repeatedly performed between the vertex set of the current clique and each of their adjacency lists. This means, each thread has to fetch adjacency lists of several vertices from the GPU's global memory in no particular order. Moreover, the set intersection operation does not have any arithmetic intensive operations, and so Clique-Counter can be a memory-bound application.

3) Matrix Transpose: Matrix Transpose is a simple kernel, which is part of the NVIDIA CUDA SDK. The kernel performs an out-of-place transpose of a matrix of floating point numbers, which means that the input and output matrices are stored at different locations in the GPU's global memory. The input matrix is divided into square 16×16 tiles, so that the loads are coalesced. Each tile is assigned to a thread block, which performs the following operations – (1) load the tile from the input matrix (global memory), (2) re-map the threads to tile elements to avoid uncoalesced memory writes, and (3) store the tile in the output matrix (global memory). Thread synchronization is required between the above steps to make sure that the global memory writes take place only after all the reads have finished. Also, this application does not have any arithmetic intensive operations, and so matrix transpose can be considered to be a memory-bound application. Figure 8 describes the matrix transpose kernel.

Proof of Memory-Boundedness: To check if our chosen applications are compute-bound or memory-bound, we analyzed the change in their execution times by varying the GPU's core-clock and memory frequencies. Higher core-clock frequency abets an improved performance of arithmetic instructions, while a higher memory frequency helps faster loads and stores from/to the GPU's global memory. We changed these frequencies by using *Coolbits*, a utility which allows the tweaking of these features via the NVIDIA

driver control panel. The default core-clock frequency for the GTX 280 is 602 MHz, while the default memory frequency is 1107 MHz [18]. Coolbits enables over-clocking and under-clocking of the underlying graphics card, where the core-clock frequencies can be changed from 450 MHz to 700 MHz and the memory frequencies can be varied from 800 MHz to 1300 MHz. When we tried to set the frequencies outside these ranges, they would be automatically reset back to the default frequency values.

For all the chosen applications, we plot the execution times at various frequencies and observe the trends. We first keep the GPU's memory frequency constant at the default value of 1107 MHz and vary the core-clock frequency. Figure 9a shows that the execution times decrease steadily with the increase in the core-clock frequency until about 550 MHz. But, the change in the running times become insignificant for higher frequencies, including over-clocking. Therefore, from the graph, we can infer that some of the applications are compute-bound until the core-clock frequency of 550 MHz. However, the applications become memory-bound at higher frequencies, including the default core-clock frequency of 602 MHz.

To further support our claim, we plot Figure 9b, where the core-clock frequency is kept constant at the default value of 602 MHz, while the memory frequency is varied. We can see that the execution times decrease steadily with the increase in memory frequency. In this case, even over-clocking of the GPU helps in the reduction of execution time. Thus, we can convincingly state that the chosen applications are memory-bound at the default core-clock and memory frequencies of the GPU.

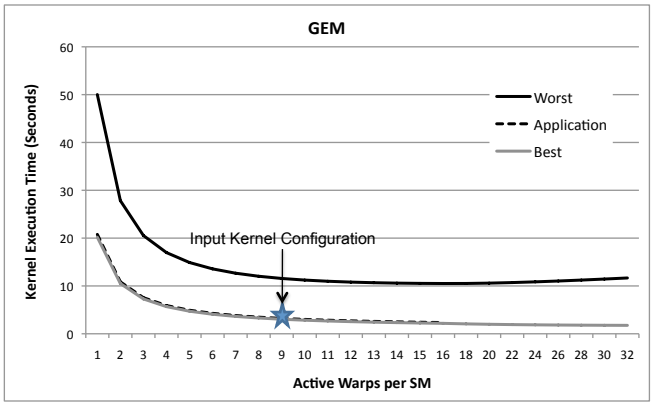
VI. Results and Discussion

In this section, we first demonstrate the utility of CampProf for detecting partition camping in the chosen applications, followed by validating the performance prediction model that serves as the back-end to the tool.

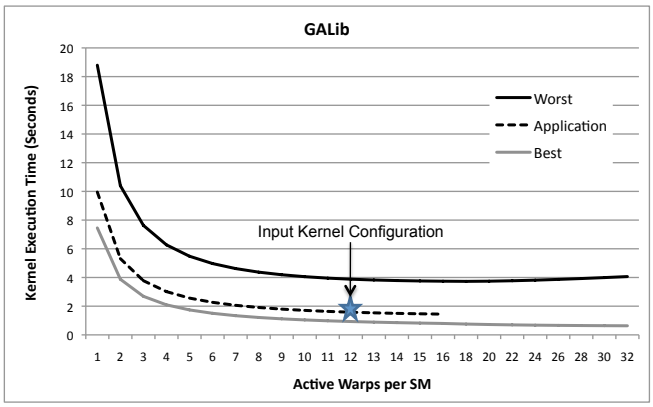
A. Partition Camping Detection

In this section, we demonstrate how CampProf can be used to visually analyze the partition camping effect in GEM, GALib and the matrix transpose applications. We then use matrix transpose as a case study to show how CampProf is used to monitor the performance of the kernel, where the execution time progresses towards the best case after the partition camping effect has been reduced, i.e. the NVIDIA SDK provides two versions of the transpose example – one with partition camping and another which is supposed to be free from partition camping. The CampProf output can be used to support NVIDIA's claim as well.

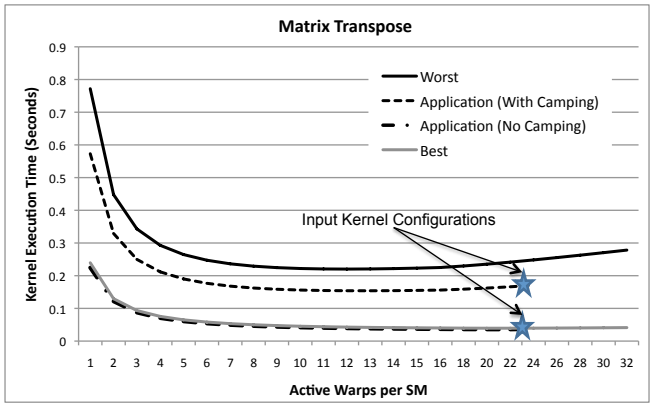
Figure 10 is the CampProf output depicting the partition camping effect in all the three applications. The input to the tool is the number of memory transactions and the kernel execution time for one execution configuration (denoted by the $\star$). It shows the worst and best execution times for all the execution configurations, as predicted by our model. The



(a) GEM: Performance Prediction



(b) GALib: Performance Prediction

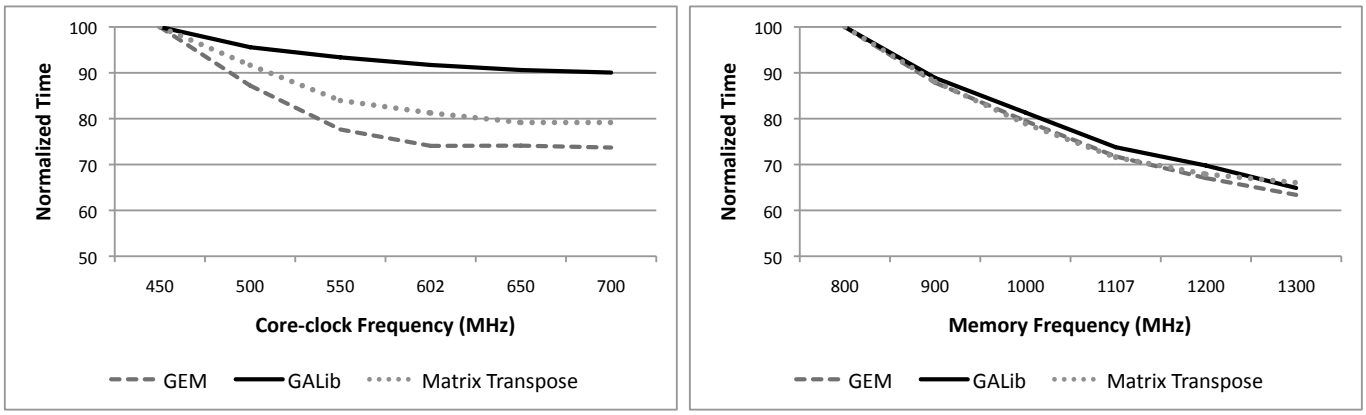


(c) Matrix Transpose: Performance Prediction

Fig. 10. Performance Bands

'Application' line in the graphs is extrapolated from the actual execution time that was provided as input. We now discuss application specific output details.

1) GEM: The predicted application execution times are presented for only up to 16 active warps, because this is the maximum active warps per SM for GEM, as computed by the CUDA Occupancy Calculator (based on the shared memory and register usage for GEM). The predicted performance can be seen to be extremely close to the 'Best' line of CampProf, which suggests that GEM does not suffer from partition



(a) Time vs GPU Core-clock Frequency

(b) Time vs GPU Memory Frequency

Fig. 9. Normalized Execution Time at Various Core-clock and Memory Frequencies

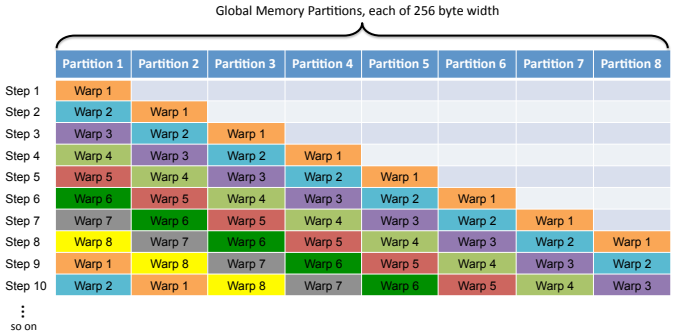


Fig. 11. GEM: Memory Access Pattern

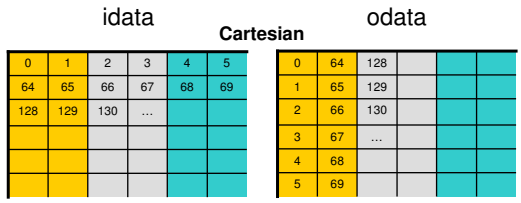
camping.

We now analyze the memory access patterns in GEM to validate CampProf's output that there is no partition camping. For computing the distance between the surface point and the higher level component, every thread accesses the coordinates of the components in the same order, i.e, from the first component to the last. This indicates that all the active warps are queued up behind the same memory partition at the beginning of the algorithm. But only one warp can go through to access that global memory partition, stalling the other warps. Once the first warp finishes accessing the elements in the first partition, it would move on the next partition, while the first partition is accessed by the next warp in the queue. Partition access will be pipelined, as shown in the figure 11. After eight such iterations (eight partitions are in the GTX 280), the memory accesses will be distributed across all the available memory partitions. It can be assumed that the pipelined memory access will have no further stalls, because the workload for all warps is the same, i.e, computing the distance to 64 higher level components. This indicates that GEM does not suffer from partition camping.

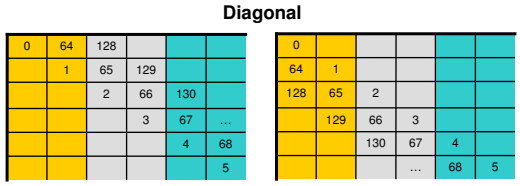
2) GALib: In the Clique-Counter application (of the GALib library), each thread has to fetch adjacency lists of several vertices from the GPU's global memory in an undefined order. This indicates that the memory access patterns are neither uniformly distributed across all the memory partitions, nor are they accessing the same memory partition. So, the Clique-Counter application can neither be completely free from partition camping, nor is it near the worst case scenario, as indicated in the figure 10b. The predicted line is shown only until 16 active warps per SM, because of the application specific resource usage, as explained previously for GEM.

3) Matrix Transpose: The NVIDIA CUDA SDK provides various versions of matrix transpose, but we specifically chose two of them for our experiments – one with moderate partition camping (called Transpose Coalesced) and the other without partition camping (called Transpose Diagonal). The only difference between the two applications versions is their global memory access pattern. Using CampProf, we determine the 'Worst' and the 'Best' execution times for the application as shown in Figure 10c. We use application specific input parameters from Transpose Coalesced and determine the extrapolated application times, which is shown in the figure as 'Application (with Camping)'. Again, the application's resource usage restricts the maximum active warps per SM for this application to 24 (deduced from the Occupancy Calculator). We note that this application time lies somewhere in the middle of the prediction band, implying that the application suffers from partition camping although it is not completely camped, i.e. the predicted times are not too close to the 'Worst' line.

To validate the fact that Transpose Coalesced is not free from partition camping, we analyze its global memory access pattern as shown in figure 12a. Different colors imply different partitions in the global memory and the numbers denote the blocks on the GPU. The figure shows that the blocks are evenly distributed among the partitions while reading the input matrix, and so partition camping is not a problem for the read operation. But, while writing into the output matrix, all blocks write to the same memory partition, making the Transpose Coalesced version of the application suffer from moderate partition camping. We can see that the CampProf output agrees



(a) Transpose Coalesced



(b) Transpose Diagonal

Fig. 12. Matrix Transpose: Memory Access Patterns [7]

with this argument.

In another version of the application called Transpose Diagonal, this problem has been rectified by rearranging the block scheduling. Now, the blocks are diagonally arranged which means that subsequent blocks are assigned diagonal tiles rather than sequential ones. This helps in making the application free from partition camping for both reads and writes, because the blocks *always* access different memory partitions uniformly as shown in Figure 12b. This is in agreement with the output of CampProf as depicted by the line 'Application (no Camping)' in figure 10c, which is close to the 'Best' predicted times in the figure.

Thus, we have demonstrated how CampProf can be used to monitor the performance of the kernel, where the execution time progresses towards the best case after the partition camping effect has been reduced. This is a great utility to use to verify if the programmer's optimizations are solving the partition camping problem or not.

B. Model Accuracy

In this section, we validate the accuracy of our performance model, by validating the accuracy of each predicted line in CampProf, i.e. the 'Best' line, 'Worst' line and the 'Application' line, against actual application execution times. We choose to use geometric mean as the error metric, because it suppresses the effect of outliers and so, is a better estimate for accuracy in the model.

Validating the 'Best' line: To verify the accuracy of the predicted 'Best' line, we should compute the error between the actual execution times for an application that is known to be free of partition camping and the predicted 'Best' time by our model for the same application. GEM has been shown to be free of partition camping in the previous section and hence, its execution times is expected to be in the vicinity of the lower time line of our prediction band.

In Figure 13a, the actual execution times and the predicted 'Best' times for GEM have been shown for all the possible configurations. As expected, the predicted times ('Best' line of the CampProf output) are in the vicinity of the actual execution times, and the geometric mean of error for the predicted time was found out be 11.7%.

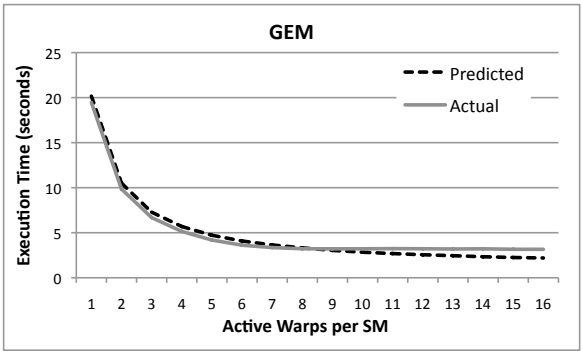
Validating the 'Worst' line: To verify the accuracy of the predicted 'Worst' line of the prediction band, we should compute the error between the actual execution times for an application that is known to have maximum partition camping effects and the predicted 'Worst' time by our model for the same application. This is a non-trivial task for two reasons – (1) It is rare to find applications, other than our micro-benchmarks, which have the maximum partition camping effects, and (2) there is no other tool or model available against which we can verify our predicted times. Therefore, it is not possible to verify the accuracy of the upper line of the prediction band. We can only use the predicted 'Worst' case line as a loose upper bound for the kernel's performance.

Validating the 'Application' line: The 'Application' line of CampProf is predicted by extrapolation based on the input execution time provided for one kernel configuration. We now estimate the accuracy of this extrapolation performed by our model for the Clique-Counter application (of the GALib library), because this application is shown to have moderate partition camping effects, and so cannot be used to validate either the 'Best' or the 'Worst' predicted lines.

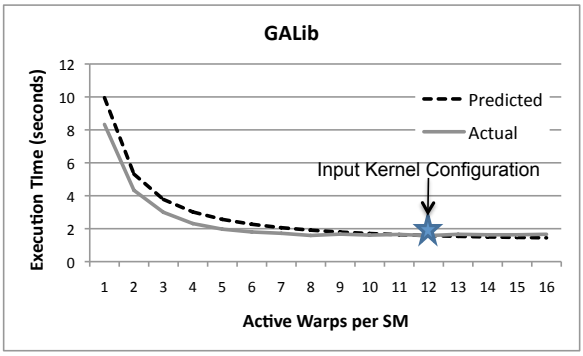
In Figure 13b, we present the extrapolated application times and the actual times for all possible kernel configurations for GALib, where the starting point for extrapolation is at 12 active warps per SM. But, there are 16 starting points (16 possible execution configurations) from which one can extrapolate to verify the prediction model. We chose *all* the 16 execution configurations as starting points for extrapolation for getting the best estimate of our model accuracy. The extrapolated times shown in the figure is for one such starting point (at 12 active warps per SM). The geometric mean of errors due to all such predicted times was found out to be 9.3%, thereby, suggesting that our performance model is accurate in the case of performance prediction via extrapolation.

VII. Conclusions and Future Work

Key understanding of the GPU architecture is imperative to obtain optimum performance. Though there are certain prevalent architectural features of the GPU, there exists a fairly obscure feature; division of the GPU Global memory into various partitions. This feature of the GPU results in what is known as the Partition Camping problem which can adversely impact the performance of any memory bound application by up to seven folds. In this paper, we have explored this partition camping problem and have developed a performance model which not only detects the extent to which a memory bound application is partition camped but also predicts the execution times for all kernel configurations if the time for one configuration is known. The performance model was formed using multiple linear regression on the results of our micro-benchmarks which exercise the partition camping effects on the GPU. We have also developed a simple, spreadsheet based tool called CampProf which inherently uses the indigenous



(a) GEM



(b) GALib

Fig. 13. Model Accuracy.

model for visual performance analysis. To the best of our knowledge, CampProf is the first tool of its kind which takes in as input parameters, easily available CudaProf values and predicts a range of execution times for an application.

Our model, at present works only for memory bound applications. Therefore, in near future, we would like to come up with such a performance model for compute bound applications as well. The newer Fermi architecture is known not to suffer from the partition camping problem, however, with its cache hierarchy, it makes GPU programming more challenging. For the Fermi cards, idea of visual performance analysis can be used to portray the effect of cache misses and to understand the gains of improved data locality. Hence, we would like to develop *"CacheProf"* for the next generation GPU architecture.

REFERENCES

[1] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, and W. mei Hwu, "Program Optimization Study on a 128-Core GPU," in *Workshop on General Purpose Processing on Graphics Processing*, 2008.

[2] Sara S. Baghsorkhi and Matthieu Delahaye and Sanjay J. Patel and William D. Gropp and Wen-mei W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *PPPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2008.

[3] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *ISCA '10: Proceedings of the 37th International Symposium of Computer Architecture*, 2009.

[4] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi and Andreas Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *ISPASS '10: Proceedings of the 37th IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[5] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *ISCA '10: Proceedings of the 37th International Symposium of Computer Architecture*, 2010.

[6] NVIDIA, "NVIDIA CUDA Programming Guide-2.3," 2009, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

[7] ——, "Optimizing Matrix Transpose in CUDA," 2009.

[8] ——, "CUDA Occupancy Calculator," 2008, http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[9] ——, "CUDA Visual Profiler," 2009, http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/cudaprof_1.2_readme.html.

[10] Michael Boyer, Kevin Skadron and Westley Weimer, "Automated Dynamic Analysis of CUDA Programs," in *Proceedings of 3rd Workshop on Software Tools for MultiCore Systems*, 2010.

[11] NVIDIA, "The CUDA Compiler Driver NVCC," 2008, http://www.nvidia.com/object/io_1213955090354.html.

[12] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008, pp. 195–204.

[13] Dana Schaa and David Kaeli, "Exploring the Multi-GPU Design Space," in *IPDPS '09: Proc. of the IEEE International Symposium on Parallel and Distributed Computing*, 2009.

[14] V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[15] W.J. van der Laan, "Decuda," 2008, http://wiki.github.com/laanwj/decuda.

[16] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka, "Statistical Power Modeling of GPU Kernels Using Performance Counters," in *IGCC '10: Proceedings of International Green Computing Conference*, 2010.

[17] Michael Bader, Hans-Joachim Bungartz, Dheevatsa Mudigere, Srihari Narasimhan and Babu Narayanan, "Fast gpgpu data rearrangement kernels using cuda," in *HIPC '09: Proceedings of High Performance Computing Conference*, 2009.

[18] NVIDIA, "GeForce GTX 280 Specifications," http://www.nvidia.com/object/product_geforce_gtx_280_us.html.

[19] ——, "NVIDIA Fermi Compute Acrhitecture," 2008, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[20] John C. Gordon, Andrew T. Fenley, and A. Onufriev, "'An Analytical Approach to Computing Biomolecular Electrostatic Potential, II: Validation and Applications," *Journal of Chemical Physics*, 2008.

[21] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html