Is Data Placement Optimization Still Relevant On Newer GPUs?

Md Abdullah Shahneous Bari[1], Larisa Stoltzfus[2], Pei-Hung Lin[3],
Chunhua Liao[3], Murali Emani[3], Barbara Chapman[1,4]
[1]*Stony Brook University*, Stony Brook, NY 11790, USA
[2]*University of Edinburgh*, Edinburgh, UK
[3]*Lawrence Livermore National Laboratory*, Livermore, CA 94550, USA
[4]*Brookhaven National Laboratory*, Upton, NY 11973, USA
Email: mshahneousba@cs.stonybrook.edu, larisa.stoltzfus@ed.ac.uk, lin32@llnl.gov,
liao6@llnl.gov, emani1@llnl.gov, barbara.chapman@stonybrook.edu

***Abstract*—Modern supercomputers often use Graphic Processing Units (or GPUs) to meet the evergrowing demands for energy efficient high performance computing. GPUs have a complex memory architecture with various types of memories and caches, in particular global memory, shared memory, constant memory, and texture memory. Data placement optimization, i.e. optimizing the placement of data among these different memories, has a significant impact on the performance of HPC applications running on early generations of GPUs. However, newer generations of GPUs implement the same high-level memory hierarchy differently and have new memory features.**

In this paper, we design a set of experiments to explore the relevance of data placement optimizations on several generations of NVIDIA GPUs, including Kepler, Maxwell, Pascal, and Volta. Our experiments include a set of memory microbenchmarks, CUDA kernels and a proxy application. The experiments are configured to include different CUDA thread blocks, data input sizes, and data placement choices. The results show that newer generations of GPUs are less sensitive to data placement optimization compared to older ones, mostly due to improvements to global memory caches.

***Index Terms*—GPU, Data placement, Experiments, Memory, Microbenchmarking, Performance analysis**

I. Introduction

In the past few decades there has been an influx in the use of GPUs to harness computational power as well as address the issue of energy efficiency in the field of high performance computing. For example, three of the top five most powerful supercomputers (Summit, Sierra, and ABCI) as ranked in June 2018 worldwide use NVIDIA Tesla V100 GPUs [1]. However, it is notoriously difficult for developers to obtain optimal performance for applications running on GPUs. One of the major reasons for this is because GPUs have a complex memory hierarchy consisting of different types of memories and caches, such as global memory, shared memory, texture memory, constant memory, L1 cache, and L2 cache (as shown in Fig. 1). Each type of memory or cache is associated with its own characteristics such as capacities, latencies, bandwidths, supported data access patterns, and read/write constraints. Where to put application data, or data placement optimization, was an essential yet challenging step to exploit the abundance of GPU threads which has been reported by previous studies [2]–[4].

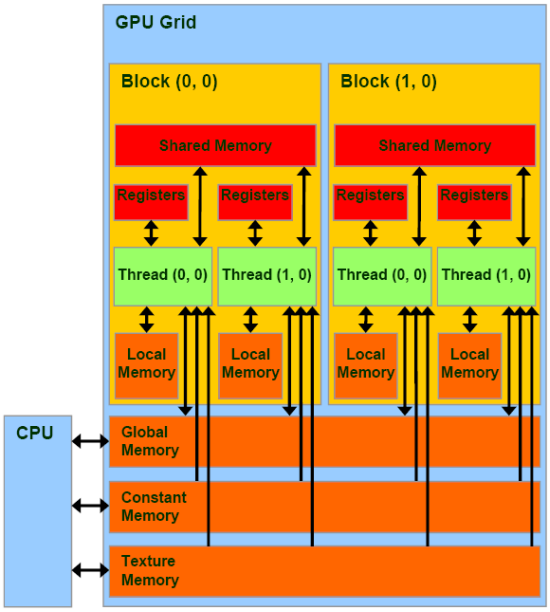


Fig. 1: GPU memory hierarchy (from NVIDIA [5])

Although all NVIDIA GPUs have a similar high-level design, different generations of GPUs introduce new memory properties or implement the same memories using different physical organizations. All these hardware changes may influence the effectiveness of memory optimization, therefore codes optimized for one platform have no guarantee of retaining the same performance on newer platforms. This becomes a problem for maintenance, as targeting optimizations using these different memory types is not a straightforward process. Moreover, optimizations on one platform might slow an application down on another platform.

In this paper, we present a set of experiments designed to explore the impact of data placement optimization on several generations of NVIDIA GPUs (Kepler, Maxwell,

Pascal, and Volta) and different codes, including a set of microbenchmarks, CUDA kernels, and a proxy application. The experiments are configured to include different CUDA thread block sizes, data inputs, and data placement choices. This paper makes the following contributions:

- We show comparisons of memory properties across several generations of GPUs to highlight their similarities and differences,
- Our work uses a range of microbenchmarks and kernels to explore the impact of data placement across generations of GPUs,
- We analyze performance results and summarize the general trends of using special memories across different generations of GPUs.
- To the best of our knowledge, this paper is the first to study the impact of data placement on recent Pascal and Volta GPUs.

The remainder of this paper is organized as follows. Section II gives more information about the GPU memory hierarchy and its special memories. Section III presents our design of a set of experiments to study GPU memory properties and the impact of data placement optimization. Experimental results and analysis are given in Section IV. Section V summarizes related work. We then conclude our findings in Section VI.

II. GPU Memory Hierarchy

GPUs have a highly complex memory hierarchy in order to exploit their massive parallel computing potentials.

Type	Location	Access	Cached	Scope
Global Memory	Off-Chip	Read Write	Y	Global
Shared Memory	On-Chip	Read Write	N	SM
Constant Memory	Off-Chip	Read Only	Y	Global
Texture Memory	Off-Chip	Read Only	Y	Global

TABLE I: GPU memory types

Table I gives a high-level overview of the major types of memories which are exposed to programmers via the CUDA API. Below we describe each in more detail:

- **Global Memory**: Also called device memory, this is the largest off-chip memory on a GPU. It also serves as the main memory. Global memory accesses have long latencies and limited memory bandwidth, when compared with accessing on-chip memory or cache.
- **L1 and L2 Caches**: While old Tesla GPUs (CUDA Compute Capability 1.x) did not have caches for global memory, later GPUs from Fermi onward (CUDA Compute Capability 2.x and later) are equipped with a cache hierarchy to improve the performance of global memory accesses. Each Streaming Multiprocessor (SM) has a dedicated L1 cache while all SMs share a single L2 cache.
- **Shared Memory**: This is a software-managed, on-chip data cache for each SM. It has low-latency (similar to a register access) and high-bandwidth. However, the size of shared memory is very limited. It is visible to only active threads within a SM.
- **Constant Memory**: This is implemented as a predefined part of the global memory space which is set to be read-only. It is cached and globally visible to all threads. The latency of constant memory accesses can be as fast as reading from a register if threads of a warp read the same address from cached data. Otherwise it is the same as accessing the device memory.
- **Texture Memory:** This type of memory is similar to constant memory in that it is an off-chip memory space that is cached and read-only. However, texture memory can be as large as the entire global memory space bound to the texture unit. Texture cache is specially optimized for 2D spatial locality, therefore it is best suited to serve threads accessing the memory addresses that are closer to each other in 2D. Every SM has several texture fetch units.

More than just differing by the various kinds of memories available, different generations of GPUs introduce new memory properties or implement the same memories using a different physical structure and organization. Fermi GPUs introduced a true cache hierarchy for global memory while previous GPUs did not have such a design. In Kelper GPUs, L1 cache and shared memory are combined together and texture cache has its on-chip memory. Volta GPUs have merged L1 cache, shared memory and texture cache into a single unified 128 KB physical memory, while shared memory in Pascal and Maxwell enjoys its own dedicated physical memory. All of these changes to memory design and implementation can directly influence the effectiveness of data placement optimization for GPU applications.

III. Design of Experiments

In this section, we discuss the design of our experiments, including the choices of machines, benchmarks and experimental configurations.

A. GPU Machines

As shown in Table II, we selected four machines for our experiments in order to compare the impact of data placement optimization on different generations of GPUs. In total, they contain four generations, namely Kepler, Maxwell, Pascal and Volta. Two machines are located at Livermore Computing Center of Lawrence Livermore National Laboratory. One machine is provided by the NVIDIA PSG cluster. We have also built a customized Google Cloud machine with a Volta GPU.

Table III summarizes some of the key specifications of the four types of GPUs. As shown in the table, Kepler K40m has a combined 64KB L1 cache and shared memory. The combined cache can be configured as 48KB and 16 KB (or 32 vs. 32, 16 vs. 48) for L1 and shared memory, respectively. Texture cache is dedicated for Kepler. Maxwell and Pascal GPUs also use a different implementation with separate shared memory but combined L1 cache and texture cache. On Volta, all three caches (L1, texture cache and shared memory) are merged together into a 128 KB unified cache, in which shared memory can take up to 96KB.

Name	Location	CPU	Memory	GPU	OS
Surface	Livermore Computing	Intel Xeon E5-2670 @2.60 GHZ	256 GB	K40m	RHEL 7.5
PSG	Nvidia PSG cluster	Intel Xeon E5-2690 v2 @ 3.00GHz	128 GB	M60	CentOS 7.5
Ray	Livermore Computing	IBM Power8 @2.2GHz	256 GB	P100-SXM2-16GB	RHEL 7.5
Custom	Google Cloud	Intel Xeon E5-2699 @2.20GHz	30 GB	V100-SXM2-16GB	Ubuntu 16.04

TABLE II: Experimental machines with GPUs

	Kepler (K40)	Maxwell (M60)	Pascal (P100)	Volta (V100)
Computation capability	3.5	5.2	6.0	7.0
SMs	15	16	56	80
Cores/SM	192 SP cores/64 DP cores	128 cores	64 cores	64 SP cores/32 DP cores
Texture Units/SM	16	8	4	4
Register File Size/SM	256 KB	256 KB	256 KB	256 KB
L1 Cache/SM	Combined 64K L1+Shared	Combined 24KB	Combined 24 KB	128 KB Unified
Texture Cache	48KB			
Shared Memory/SM	Combined 64K L1+Shared	96 KB	64 KB	
L2 Cache	1536 KB	2048 KB	4096 KB	6144KB
Constant Memory	64 KB	64 KB	64 KB	64 KB
Global Memory	12 GB	8 GB	16 GB	16 GB

TABLE III: Key specifications of selected GPUs of different generations

B. Benchmarks

Our goal is to study the impact of data placement optimization within the GPU memory hierarchy, in particular given the choice of utilizing four different types of memories (Constant memory, Shared memory, Texture memory and Global memory). First we attempt to understand the fundamental properties of these GPUs and the impact of using individual memories, using simple CUDA kernels. However, as real applications tend to behave very differently from small kernels, we also test more complex CUDA codes using a mixture of memories. Overall we will showcase four different kinds of benchmarks in our experiments: (1) microbenmarks measuring memory specification (2) simple CUDA kernels evaluating the impact of using one type of memory (3) CUDA codes assesing more complex use of memories (4) a proxy application with multiple kernels using a mixture of data placement choices.

1) Microbenchmarks: We surveyed research literature in order to find available microbenchmarks to measure the memory specification for the selected GPUs. `GPUmembench` used in [6] collects multiple microbenchmarks to measure different memories in previous generations of GPUs. We adopted the microbenchmarks and revised them accordingly to measure the memory bandwidth for the selected GPUs. We also collected the microbenchmark from [7] that revised the traditional pointer-chasing benchmark to use GPU shared memory to store a sequence of data access latencies and eliminate the interference with normal data access.

2) CUDA kernels: As shown in Table IV, we have picked three representative CUDA kernels [8] that are designed to show the advantages of individual special memories (e.g. constant memory, shared memory and texture memory) compared to the global memory.

(a) *Constant memory:* The Ray Tracing benchmark is a kernel which shows the benefits of using constant memory. The kernel simulates light reflecting off three-dimensional spheres on a two dimensional image in a three dimensional cube. The computational complexity of this kernel depends on the number of spheres used and the dimensions of the cube. In the constant memory version, the array representing spheres is put in the constant memory while in global memory version it is put into the global memory. We used three different data sizes for our experimentation: Small, Medium and Large. Details of these data sizes are explained in Table IV. We chose these data sizes in such a way that they utilize the constant memory in different proportion (e.g. data size 'Small' uses only a fraction of constant memory while data size 'Large' uses all 64K available constant memory in the GPU).

(b) *Shared memory:* We present a matrix matrix multiplication kernel (MM) to test the impact of shared memory. The experiments we performed looked at variations of the MM benchmark utilizing global and shared memories. In the shared memory version, blocks of sub-matrices are stored in shared memory and are computed over separately by different threads while in the global version, all values are accessed and stored in global memory. We also used three different data sizes for our experimentation: Small, Medium and Large. Details of these data sizes are explained in Table IV.

(c) *Texture memory* To demonstrate the impact of texture memory we use a simple two-dimensional five-point stencil computation kernel simulating heat transferring. The simulation assumes that a rectangular room is divided into a grid. Inside the grid, a handful of 'heaters' are randomly scattered with various fixed temperature. We use three different types of grids: Small, Medium and Large. For each grid, the kernel is run 90 iterations (time steps) for each grid. Details of these grids are explained in Table IV. Both 1D and 2D texture versions are used to compare their results with respective to global memory versions. However, we only show the results for 2D versions due to space constraint and also the fact that texture memory has more importance in 2D spatial locality.

3) Benchmark Applications: To further evaluate the impact of data placement, we selected three benchmarks used in a previous study [3] to re-run: sparse matrix vector multiplication, matrix matrix multiplication and computational fluid

Kernel	Memory Focused	Small	Medium	Large
Ray Tracing	Constant Memory	No. of spheres: 200 Cube dimension: 2048	No. of spheres: 1000 Cube dimension: 2048	No. of spheres: 2340 Cube dimension: 2048
Matrix Matrix Multiplication	Shared Memory	Matrix size: 512×512	Matrix size: 1024×1024	Matrix size: 2048×2048
Heat Transfer	Texture Memory	Grid size: 1024×1024	Grid size: 4096×4096	Grid size: 8192×8192

TABLE IV: Three representative kernels and their input data sizes

dynamics. Sparse Matrix Vector Multiplication (SpMV) is of the form $y = Ax$ and is the multiplication of a matrix A containing mostly zeros and a vector x, which is a commonly found function in scientific codes. Matrix-Matrix Multiplication (MM) is selected again but with more versions using differing combinations of memories. Computational Fluid Dynamics (CFD) represents a simulation of the behavior of fluid dynamic phenomena, which can be used for prediction in a wide range of areas.

The experiments we performed looked at variations of the SpMV benchmark utilizing global, constant, texture and shared memories. This benchmark is loosely based on the public version which can be found in the SHOC benchmark suite [9]. Nine versions were run using a combination of global, texture, shared and constant memory (shown in Table VIII in the Appendix). This benchmark itself has four arrays (values, columns, vector values and rows) and contains the most broad picture of different memory usage of the three benchmarks we looked at.

For the MM benchmark we ran experiments utilizing global, texture and shared memories. This experiment ran different versions of codes with the two matrices (shown in Table IX in the Appendix). There is one version with only global memory and one with only shared memory. The rest of the versions are combinations shared memory and different types of texture memory - including versions using 2D and surface texture memory.

The CFD benchmark, which is loosely based on the same version found in the Rodinia benchmark suite was tested out using global memory and a variety of texture memories [10]. As this benchmark has a large number of arrays (eight), it would be difficult to show a comparison of all possible combinations of these arrays in different memories. Instead, these experiments (unlike those for MM or SpMV) focused exclusively on global and texture memory. There are eight versions of the codes with varying numbers of different arrays being stored in texture memory (shown in Table X in the Appendix), as well as one version using global memory.

4) Proxy application: LULESH [11] is a shock hydrodynamics code which is a part of the DARPA UHPC effort and is now used in DOE's ExMatEx co-design efforts. LULESH is a hexahedral mesh-based physics code with two centerings: the element centering and the nodal centering. The element centering refers to the center of each hexahedron and stores thermodynamic variables, such as energy and pressure. The nodal centering refers to the corners of hexahedra intersect, and stores kinematics values, such as positions and velocities. The main computation of LULESH happens via time-stepping using a Lagrange leapfrog algorithm followed by a time constraint calculation. We summarize the selected kernels and their array read/write pattern in Table V. The actual data size used in the kernel changes according the problem size. The ranking column shown in the table shows the importance of the selected kernels in terms of the execution time.

We prepared four versions of LULESH. Each version uses only one type of memory for all data in the five kernels. The version using global memory as data storage serves as the baseline version. Other versions using constant memory, shared memory and texture memory are compared to the baseline performance and the speedup between the two is shown The execution time does not include the data transferring time from other memory locations to the selected type of memory. We also use three data sizes (4, 45, and 90) to explore the impact of varying inputs. For data sizes 45 and 90, constant and shared memory are not big enough to hold all the data so only the global and texture memory versions are evaluated.

ID	Kernel	Read-only array count	Write-only array count	Ranking
1	AddNodeForcesFromElems	6	2	4
2	AddNodeForcesFromElems2	6	2	5
3	CalcFBHourglassForceForElems	6	7	2
4	CalcHourglassControlForElems	13	3	1
5	CalcMonotonicQRegionForElems	16	1	8

TABLE V: Properties of selected LULESH kernels

C. Testing Configurations

We present additional information of the configurations of our experiments.

There are several ways to control the available resource utilization in GPUs. The Kepler GPU we use has configurable 64KB caches for both L1 and shared memory. Although there is a CUDA API to control the actual partition sizes between them, we have used the default partition (48 KB shared vs. 16KB L1) on the machine for simplicity. There are also configurations for how to launch a CUDA kernel.

For the ***CUDA kernels*** experimentation (Section III-B2) we used a predefined number of Thread Block Sizes ($4 \times 4 = 16$, $8 \times 8 = 64$, $16 \times 16 = 256$, $32 \times 32 = 1024$). Number of Grids was selected based on the Thread Block Size and Application Data Size (e.g. Application Data Size / Thread Block size). The default number of Warp Size was used. This approach allows us to investigate the performance on the individual SMs (through Thread Block Size) level as well as in the whole GPU level (through Grid Size). Since all the three kernels described in Section III-B2 work on 2D data, we used 2D Thread Blocks and Grids for our experimentation. In Table VI we show an

example of different configurations that were used assuming the application uses a 2048×2048 grid.

Thread Block Size	No. of Grids
$4\ (4 \times 4 = 16)$	$512 \times 512 = 262144$
$8\ (8 \times 8 = 64)$	$256 \times 256 = 65536$
$16\ (16 \times 16 = 256)$	$128 \times 128 = 16384$
$32\ (32 \times 32 = 1024)$	$64 \times 64 = 4096$

TABLE VI: Example thread block and grid configurations

Our experiments were run using a script to collect average values over ten iterations and the median of these values is then reported in Section IV. Each kernel was run five times to warm up the GPU before timings were taken. In order to collect consistent performance times, the performance benchmarks all used `cudaEventRecord` to denote kernel stop and start places and `cudaEventElapsedTime` to calculate the time elapsed. `cudaDeviceSynchronize` was used after these calls and data transfer times are excluded in order to isolate performance differences from memory usage. A value reported as speedup of using a special memory means a ratio relative to the global memory version (ie. the version using only global memory - $speedup = \frac{global\ memory\ version}{new\ memory\ configuration}$).

IV. Results and Analysis

A. Microbenchmarking

All the measured memory specifications of the four GPUs we used are shown in Table VII. The change of global memory from GDDR5 to HBM2 has had a big impact on memory performance as the bandwidth has improved 3.5X from the Kepler GPU to the most recent Volta GPU. For all platforms, our measured global memory bandwidth is roughly 80 to 85% of the theoretical bandwidth. We also observe improvement in all memory specifications, including bandwidth and latency, from the earlier generations to the most recent GPUs. For the latest Volta GPU, the highest bandwidth and lowest latency are measured here. Figure 2 compares the bandwidth of these different GPUs. It is noticeable that L1 bandwidth of Volta is significantly better than previous generations. Another interesting point is that the bandwidth of shared memory of Maxwell is smaller than its predecessor (Kelper).

B. CUDA Kernels

In this section we analyze the experimental results related to the three CUDA kernels. We investigate how the impact of a specific memory has changed over the generations of GPUs. We primarily use two metrics to explain our results: Speedup and Execution time. Speedup is calculated with respect to the default configuration. This default configuration means using global memory and a Thread Block Size of 32. The base speedup is one, denoted by a red dotted line in the graphs. Anything over one is performance improvement compared to default configuration while anything lower is performance degradation.

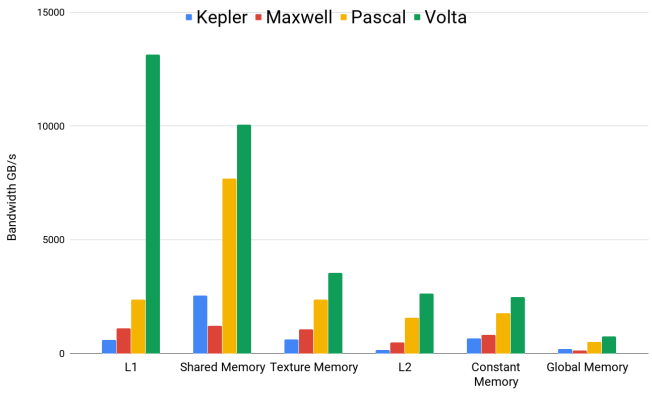


Fig. 2: Bandwidth comparison of different GPUs

1) Constant Memory: Figure 3 shows the speedup using constant memory across different GPUs. The x-axis represents the GPU Thread Block Size used while the y-axis represents the speedup. Each block represents a certain generation of GPU. Figures 3a, 3b and 3c show results for Small, Medium and Large data sizes respectively.

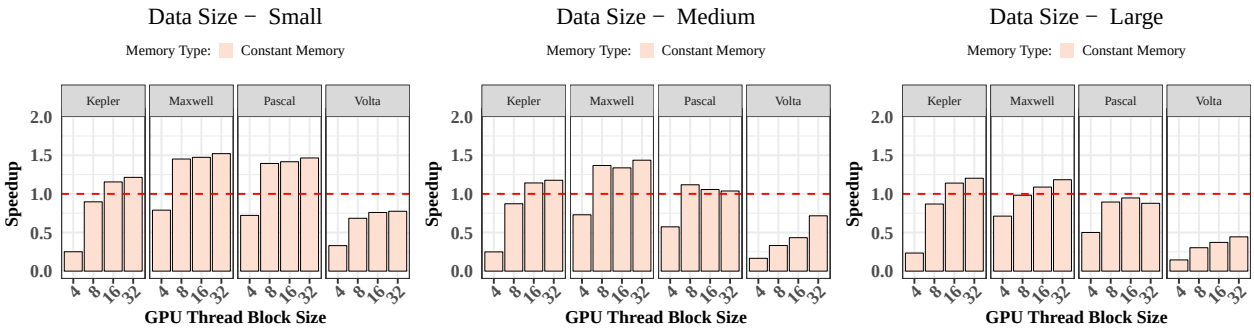
This section of our experiments can be divided into three different parts. First, the impact of constant memory across different generation of GPUs is investigated. Secondly, we analyze the impact of data size. Lastly, we discuss the importance of Thread Block Sizes.

We observe first that the impact of constant memory has started to diminish in newer generations of GPUs since Maxwell. In fact, with Volta, usage of constant memory degrades the performance in all different data sizes. This phenomenon is evident as the speedup using constant memory on Volta is below one in all three data sets. We find high percentage of stalling due to pipeline busy for the constant memory version based on results of the Nvidia profiler. With Pascal we see a similar phenomenon with larger data sets. This is primarily due to the architectural change in the GPUs. Improvement in global memory bandwidth through HBM2 in Pascal and Volta has improved the global memory access which is accompanied by the improvement in L2 and L1 cache performance.

However, this is not to say that the performance of constant memory has not improved across GPU generations. In Figure 4 we compare the execution time across different generations of GPUs using constant memory. We show the *medium* data set size which uses the GPU thread block size of 32. From the Figure, steady execution time improvement can be seen across generations of GPUs. The key take-away from these results is that even though constant memory performance has improved over time, new breakthroughs in global memory along with L1 and L2 cache improvement have come a long way to diminish the impact of dedicated constant memory, at least for some applications. As shown in Figure 2, the significantly enhanced L1 cache on Volta has 5X times higher bandwidth than constant memory does.

Memory/Cache	Properties	Kepler (K40)	Maxwell (M60)	Pascal (P100)	Volta (V100)
L1 data cache	size (KB)	16 ˜48	24	24	32 ˜128
	hit latency	35	82	82	28
	line size	128B	32B	32B	32B
	bandwidth (GB/s)	602	1103	2379	13414
L2 data cache	size (KB)	1536	2048	4096	6144
	line size (B)	32	32	32	64
	hit latency	˜200	˜207	˜234	˜193
	bandwidth (GB/s)	154	488	1579	2629
Shared memory	size per SM (KB)	48	96	64	up to 96
	Banks	32	32	32	32
	Theoretical bandwidth(GB/s)	2912	2410	9519	13800
	Measured bandwidth(GB/s)	2540	1213	7681	10057
Constant memory	Measured bandwidth (GB/s)	665	817	1776	2470
Texture memory	Measured bandwidth (GB/s)	619	1068	2378	3547
Global memory	Memory bus	GDDR5	GDDR5	HBM2	HBM2
	Max clock rate (MHz)	2505	2505	715	877
	Theoretical bandwidth(GB/s)	240	160	732	900
	Measured bandwidth(GB/s)	191	127	510	750

TABLE VII: Measured memory specification



(a) Speedup in Small dataset (b) Speedup in Medium dataset (c) Speedup in Large dataset

Fig. 3: Speedup achieved using constant memory compared to global memory

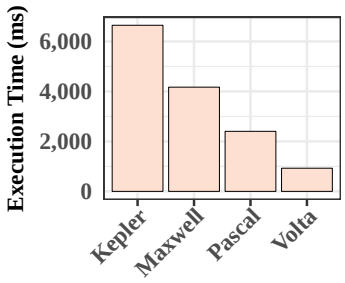


Fig. 4: Execution time using Constant memory

We observe that with the increase in data size, global memory becomes a more viable option compared to constant memory. We see this result probably due to the bandwidth utilization of constant memory. With the increase in data size, utilization of constant memory increases, hence constant memory bandwidth starts to become a bottleneck which in turn degrades performance.

We observe that for this application, speedup (performance) improves with the number of Thread Block Size. We believe, this is primarily due to the improvement in SM utilization. We observe a similar trend across different generation of GPUs and different data sizes.

2) Shared Memory: In Figure 5 we present the speedup achieved using shared memory compared to the default configuration. The x-axis represents the GPU Thread Block Size used and the y-axis represents the speedup. Each block represents a certain generation of GPU. We also show the results for three different data sizes. Figure 5a, 5b and 5c show results for Small, Medium and Large data sizes respectively.

From this figure we see that with the newer generation of GPUs the influence of shared memory is consistently diminishing except for Pascal. We understand this is happening due to the architectural differences in shared memory, L1 cache and texture memory units. Kepler has a configurable shared memory that is shared with the L1 cache while Maxwell has a dedicated shared memory unit. As for Pascal, the memory structure is similar to Maxwell, while in Volta shared memory, L1 cache and texture memory share the same unit.

We see speedup going down from Kepler to Maxwell primarily due to the decrease in shared memory bandwidth. Although in Maxwell there is a dedicated shared memory unit, the application we used never exhausts the available shared memory limit. As a result, we don't see the impact of dedicated shared memory.

However, we do see a dramatic speedup using shared

memory on Pascal, primarily due to its use of HBM2 and high bandwidth of shared memory compared to previous GPUs. One possible reason for this is the fact that shared memory and the L1 cache have different units in Pascal, hence different latencies. While the latency for the L1 cache does not change from Maxwell to Pascal (I), the shared memory latency seems to have improved from Maxwell to Pascal [7]. This also explains the diminishing speedup in Volta, as in Volta shared memory and L1 cache are in the same unit, hence their hit latency is also the same.

We also compare the actual performance of different generation of GPUs in Figure 6. For a representative configuration using the Medium data size and GPU Thread Block Size of 32, we observe steady improvement across the generations of GPUs, except for Maxwell. This is probably due to low bandwidth of the shared memory unit in Maxwell.

3) Texture Memory: Figure 7 shows the results of the 2D texture memory version of the Heat transfer kernel's speedup compared to the default global memory version. The 1D version results are very similar and are therefore omitted here.

We observe that the texture memory slightly outperforms the global memory in Kepler when the thread block size is 16. But with the newer generations of GPUs global memory starts to outperform texture memory, especially with the larger data sizes. This is primarily due to the architectural change in texture memory unit. Among the experimental GPUs, only Kepler has a separate texture cache unit while all other GPUs have the texture cache unit shared with L1.

As for the performance of the texture memory unit across different GPUs, we observe a similar trend as constant memory for the 2D (8) texture version. We see a steady improvement in the newer generation of GPUs.

C. Benchmark Applications

This section describes our results from comparing the three benchmark applications run across the four different platforms with variations of memory configurations. Tables VIII, IX and X have short descriptions of the versions of memory configurations found in these figures.

Figures 9, 10 and 11 shows these separate applications exhibiting very different performance measurements for particular memory types across a range of hardware for two different benchmarks. In Figure 9 it can be seen that choosing different memory for SpMV types generally has some improvement with texture and combinations of texture and shared memories, but as the GPU architectures evolve from Kepler to Volta, the improvement seems to become decrease and even become detrimental. However, Figure 10 paints a somewhat different picture. Here it can be seen that the different data placements can still be relevant as the GPUs evolve, even on the Volta architecture. Finally, in Figure 11 we see an example where the usage of different texture memories seems to have very little effect overall on the benchmark, no matter which generation of GPU.

Figure 9 shows that for some applications, there can be a wide range of performance differences between utilizing different memory versions across these selected platforms. This indicates how important it is for some applications to re-evaluate optimizations that were made for a specific architecture. In particular, version *9* shows a greater than 1.5X speedup on Kepler, but a 20% slowdown on Volta. Yet, alternatively the shared and constant versions maintain relatively similar performance across the platforms.

Figure 10 shows that for some applications, performance gains can still be made even on the newer platforms. On Volta, there is still up to a 1.2X improvement using version *5*, which utilizes texture1D memory for both matrices `A` and `B`. The shared memory version (*6*) also shows marked improvement across all platforms - almost 4X on Pascal and retaining 1.5X on Volta. However, it should be noted that this particular version (and its corresponding global version) are slightly different from the other versions in their use of a different data object for the arrays. This may explain why the speedup is so pronounced - because the original version uses a non-optimal data object to begin with.

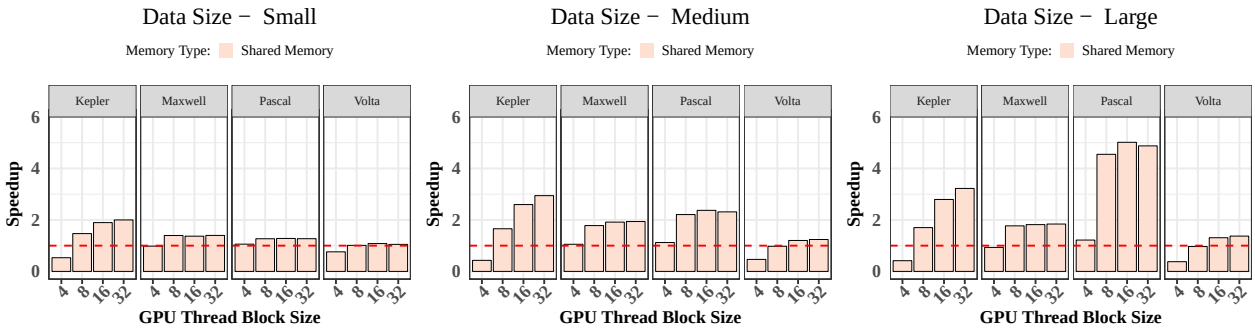
Figure 11 shows that for some applications, performance will stay relatively consistent across generations of GPUs even when utilizing different memory types. For the CFD benchmark, at best there is an 8% speedup on the Kepler architecture. At worst, on Maxwell, there is a 14% slowdown. However, on the newest architecture Volta, the worst slowdown is about 10%. For this application it is clearly still important to get the data placement right, however it is less detrimental than some of the losses clearly shown for SpMV in Figure 9.

Of all the memories looked at, although texture was investigated in the most detail, it also seems to be the one retaining the most improvement. On Volta, using texture memory can still result in improved performance for the MM and CFD (marginally) benchmarks. Shared memory also seems to show improvement for some applications - the MM benchmark retained better performance across all versions and all platforms then others using shared and texture memory in combination. Overall it appears that using different memory optimizations on newer platforms overall does not produce as great of speedup as it did on earlier platforms. However, these applications were all run with the same data size and block size, so the results should be considered only part of an overall picture.

D. Proxy Applications

We performed experiments on four generations of GPUs using four data placements for five selected kernels in the LULESH proxy application. Each variant was tested with three different data sizes: 4, 45 and 90. For data size 4, data is small enough to be placed in all four different memory locations. Data can be placed only in global memory or texture for data size 45 and 90.

Figure 12 shows the speedups comparing the baseline performance using global memory with performances using constant memory, shared memory and texture memory. The corresponding kernel names for Figures 12 and 13 can be found in Table V. Results shown in Figure 12 use data size



(a) Speedup in Small dataset (b) Speedup in Medium dataset (c) Speedup in Large dataset

Fig. 5: Speedup achieved using shared memory compared to global memory

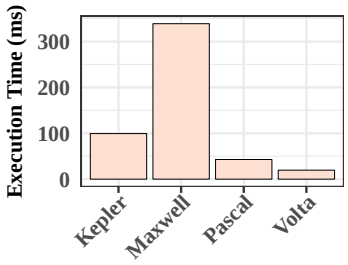


Fig. 6: Execution time using Shared memory

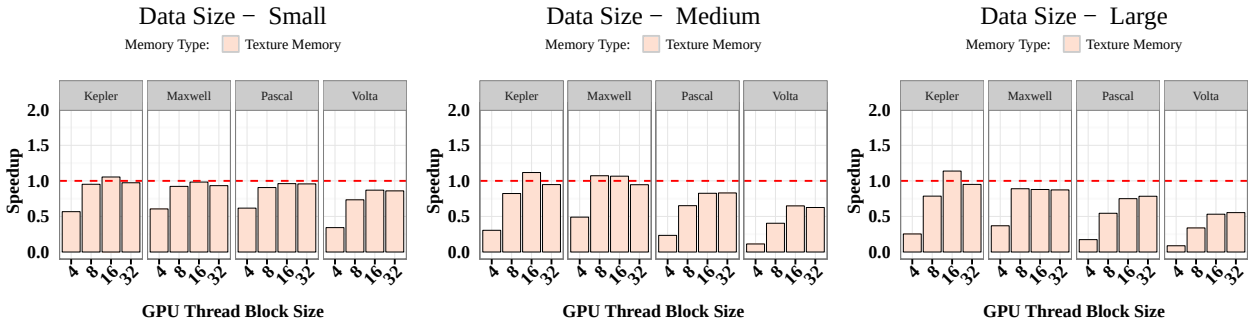
4. The red line is the reference for speedup of one. No performance improvement is observed for data placement using constant memory and texture memory. Using shared memory in the `CalcMonotonicQRegionForElems_kernel` delivers 20% improvement in speedup on the Kepler GPU, marginal speedup on the Maxwell, and no speedup for Pascal and Volta GPUs. Experiments with data size 4 will use a small number of GPU threads. This leads to low achieved GPU occupancy and under used GPU resources. The GPU thread number used in the experiment might not be sufficient to hide the memory latency.

Experiments using data size 45 and 90 are shown in figure 13a and figure 13b respectively. With larger data sizes, the performance factor that GPU resources are under-utilized can be eliminated. Meanwhile however, we can only evaluate texture memory placement compared to the default global memory placement. `CalcHourglassControlForElems_kernel` and `CalcFBHourglassForceForElems_kernel` gain performance improvements with the texture memory placement. However speedups compared to the global memory placement shrink in the newer generations of GPU. Both figures shows less benefit using texture memory placement in the newer generations of GPU. With more advanced memory specification used for global memory in Pascal and Volta GPUs, reading data in the global memory and caching through L1 and L2 cache shows better outcome than reading through the texture memory cache.

The experiment with LULESH proxy application shows only one kernel has a potential performance benefit from using the shared memory in early generations of GPU. The experiment did not exploit cache blocking transformations for the shared memory, however. This could be a more optimal way for shared memory data placement, but it would increase the complexity in programming and tuning. No performance gain is observed with the constant memory placement. However, the bottleneck is low capacity of constant memory and the low GPU occupancy in the experiments. Constant memory is good for read-only data that needs to be repetitively broadcast to all GPU threads. Our implementation with constant memory might not be practical in real applications for the following reasons: (1) low capacity to host data (2) limited read-only data (3) data is allocated at global scope and flexibility to change the data during runtime. We see more benefit using texture memory placement in the early generations of GPU. With the advanced memory adopted as global memory in Pascal and Volta GPUs, reading data from global memory and caching through L1 and L2 caches show higher performance than reading data through the texture cache. Texture memory is designed for its specialized purpose to read 2D or 3D data access patterns. Data stored in texture memory is read-only and would also limit its applicability in real applications. The extra code changes required to exploit texture memory additionally bring in more programming burden to the application developers.

V. Related Work

Previous studies have explored various data placement strategies on early generations of GPUs. For example, PORPLE [3], [12] is a portable GPU data placement approach combining a memory specification, a compiler framework and a run-time library. A lightweight performance model based on reuse distance is used by PORPLE to guide run-time selection of optimal data placement policies. Its effectiveness has been evaluated on Tesla, Fermi, and Kepler GPUs. Huang and Li [4] have proposed a new performance modeling approach for optimal data placement on GPUs. They have analyzed correlations among different data placements and used a sample data placement to predict performance for other data placements. Jang et al. [2] have presented several rules based on data



(a) Speedup with Small dataset (b) Speedup with Medium dataset (c) Speedup with Large dataset

Fig. 7: Speedup achieved using 2D texture memory compared to global memory

Fig. 8: Execution time using 2D texture memory

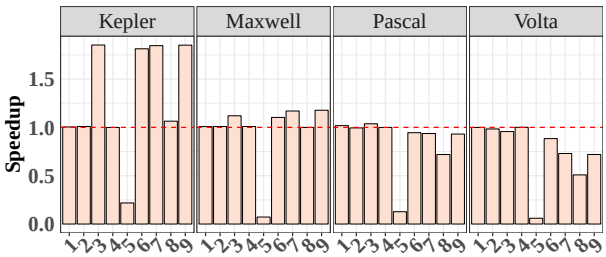


Fig. 9: Speedup of SpMV

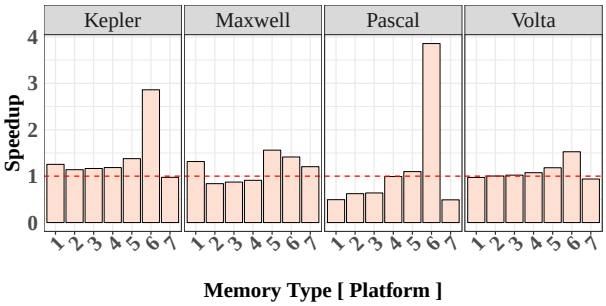


Fig. 10: Speedup of MM

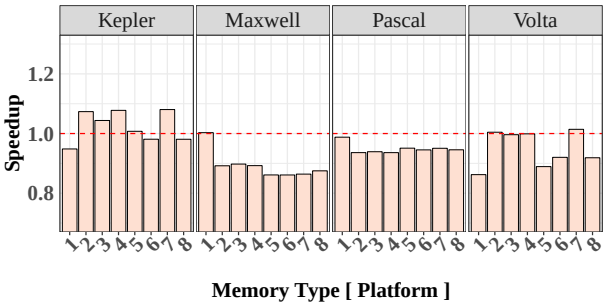
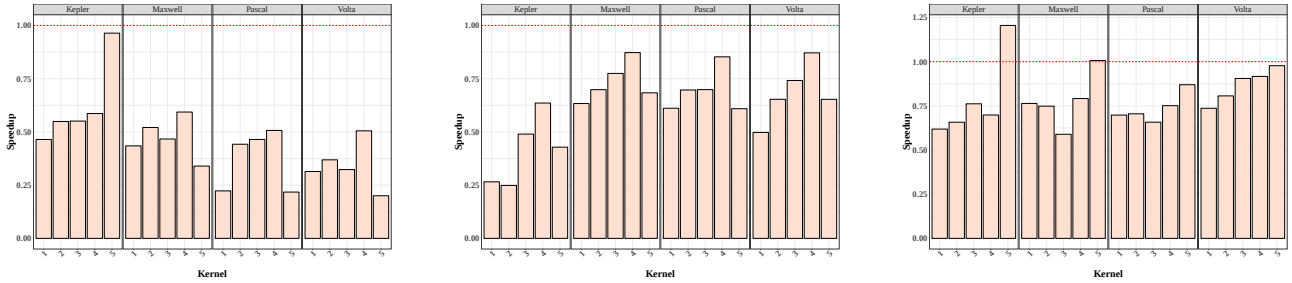


Fig. 11: Speedup of CFD

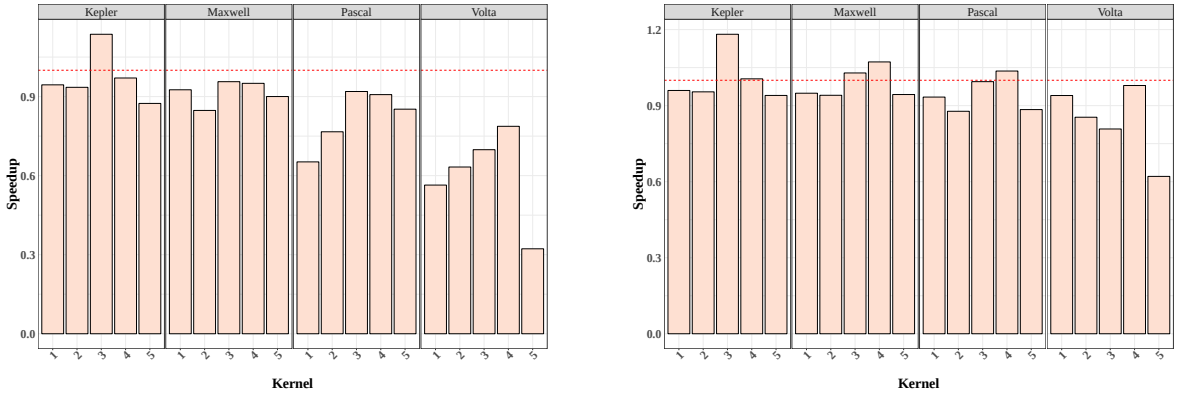
access patterns to guide the memory selection for a Tesla GPU. Yang et al/ [13] proposed compiler-based approach to generate kernel variants for exploiting memory characteristics. We believe our work is the first to study the impact of data placement on more recent generations of GPUs such as Volta.

Researchers have developed several microbenchmark suites to understand various aspects of memory characteristics of different generations of GPUs. For example, GPUBench [14] is one of the early benchmark suites designed to analyze the performance of GPUs. Volkov et al. [15] measured hardware characteristics, including texture caches, of 8800GTX GPU relevant to optimizing dense linear algebra. Fang et al. [16] made a benchmark of a GPU memory system to quantify the capability of parallel accessing and broadcasting. Wong et al. [17] have used microbenchmarking to measure the CUDA-visible architectural characteristics of the early generation Tesla GPU (GT200). BabelStream [18] measures memory transfer rates of global device memory on GPUs. It has multiple implementations using various programming models, such as CUDA, OpenCL, OpenMP, OpenACC, RAJA, and Kokkos. `gpumembench` [6] contains a set of micro-benchmarks to measure bandwidth of on-chip special memories of GPUs. Mei et al. [7] have proposed a new fine-grained microbenchmarking approach to expose unknown cache parameters of three generations of GPUs, including Fermi, Kelper, and Maxwell. More recently, Jia et al. [19] have used microbenchmarking to



(a) Texture memory version speedup (b) Constant memory version speedup (c) Shared memory version speedup

Fig. 12: LULESH kernels using size 4



(a) Texture memory version speedup: size 45 (b) Texture memory version speedup: size 90

Fig. 13: LULESH kernels using size 45 and 90.

analyze the Volta GPU architecture. In comparison, our work use more comprehensive CUDA kernels and applications to study the impact of data placement choices among multiple different memory types across different generations of GPUs.

VI. Conclusion

Previous studies have shown that data placement optimization, i.e. optimizing where to put data into different memories within a GPU, is essential for obtaining optimal performance on older generations of GPUs, such as Fermi and Kelper. In this paper, we have designed a set of experiments to explore the relevance of data placement optimization on newer generations of NVIDIA GPUs. Our experiments include a range of CUDA kernels running on four generations of GPUs including Kepler, Maxwell, Pascal and Volta.

Our key findings in this paper include: 1) All types of memories on newer GPUs have improved performance compared to previous generations. 2) The unified cache design of Volta GPUs helps narrow the performance gap between the default global memory and all other special memories. In particular, constant memory and texture memory seemed to be much less important on recent GPUs. 3) Of those applications that benefited, texture and shared memory showed the most promise of speedup gain through considered data placement optimization. 4) The memory properties of special memories significantly limit their use in real applications. Constant memory and texture memory are read-only; constant memory and shared memory have small capacity. Also significant programming efforts are required for using these special memories, especially for shared memory. Automated code transformation is needed to exploit different types of GPU memory.

In the future, we plan to study data placement optimization's impact on energy consumption. Since all the code variants used in this paper are manually prepared, we are also interested in developing automated code transformation to exploit special GPU memory types.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 18-ERD-006. LLNL-CONF-757796.

REFERENCES

[1] "Top500 list: June 2018," https://www.top500.org/lists/2018/06/.

[2] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel & Distributed Systems*, no. 1, pp. 105–118, 2010.

[3] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 88–100.

[4] Y. Huang and D. Li, "Performance modeling for optimal data placement on gpu with heterogeneous memory systems," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017, pp. 166–177.

[5] C. NVIDIA, "Nvidia cuda programming guide (version 1.0)," *NVIDIA: Santa Clara, CA*, 2007.

[6] E. Konstantinidis and Y. Cotronis, "A quantitative performance evaluation of fast on-chip memories of gpus," in *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*. IEEE, 2016, pp. 448–455.

[7] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[8] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[11] I. Karlin, A. Bhatele, B. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "Lulesh programming model and performance ports overview," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep. LLNL-TR-608824, December 2012.

[12] G. Chen, X. Shen, B. Wu, and D. Li, "Optimizing data placement on gpu memory: A portable approach," *IEEE Transactions on Computers*, vol. 66, no. 3, pp. 473–487, 2017.

[13] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806606

[14] I. Buck, "Gpubench: Evaluating gpu performance for numerical and scientific application," in *Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^ 2'04)*, 2004.

[15] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–11.

[16] M. Fang, J. Fang, W. Zhang, H. Zhou, J. Liao, and Y. Wang, "Benchmarking the gpu memory at the warp level," *Parallel Computing*, vol. 71, pp. 23–41, 2018.

[17] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[18] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Gpu-stream v2. 0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *International Conference on High Performance Computing*. Springer, 2016, pp. 489–507.

[19] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

APPENDIX A
ARTIFACT DESCRIPTION

A. How software can be obtained

The code and scripts related to this work can be downloaded from: https://gitlab.com/sayket/dataPlacementExperiments.

For the benchmark applications, multiple versions using different memory placement configurations are provided, as shown in Table VIII, Table IX and Table X

Version	Description
1	rows in shared
2	rows in constant
3	vector in texture1D, rows in shared
4	matrix values in texture1D
5	vector in constant, rows in texture1D
6	vector in texture
7	matrix values and columns in texture1D, rows in constant
8	matrix values, columns and vector in texture1D
9	matrix values, columns, rows and vector in texture1D

TABLE VIII: SpMV benchmark memory configurations

Version	Description
1	both matrix A and B in texture2D and shared
2	matrix A in texture2D and both matrices in shared
3	matrix B in texture2D and both matrices in shared
4	matrix B in texture1D and both matrices in shared
5	both matrix A and B in texture1D and shared
6	both matrices in shared
7	matrix A in surface texture and both matrices in shared

TABLE IX: MM Benchmark Memory Configurations

Version	Description
1	neighbors in texture1D
2	mx, mz, energy in texture1D
3	mx, my, mz in texture1D
4	energy, my, mz in texture1D
5	mx, my, mz, energy, neighbors and normals in texture1D
6	energy, neighbors and normals in texture1D
7	mx, my, mz, energy, normals and density in texture1D
8	my, mz, energy, normals, neighbors and density in texture1D

TABLE X: CFD Benchmark Memory Configurations

Version	Description
globalall	all data in global memory
texall	all data in texture memory
constantall	all data in constant memory
sharedall	all data in shared memory

TABLE XI: LULESH proxy application Configurations

B. Hardware Dependencies

The details of the machines are provided in Section III-A.

C. Software Dependencies

- As the experiments are based on CUDA programs, you need to have CUDA installed in your system. Also make sure that you have the NVIDIA Management Library (NVML) and CUDA Profiling Tools Interface (CUPTI) installed.
- Data analysis and Graph generation process uses Python and R. Make sure that you have python and R installed with the following packages,
  Python: pandas, numpy, csv, "sys, getopt", matplotlib R: plyr, ggplot2, grid, RColorBrewer, scales, sqldf, "optparse"
- We created an auto-tuning framework called GPUTuner based on active harmony search framework. The code is available in the 'activeharmony-GPUTuner' folder. We use this auto-tuner to exhaustively search all possible combinations of memory and resource configurations. The framework is self-contained.

D. Datasets

All programs used in the experiments have built-in data sets. No additional input files are needed.

E. CUDA Kernel Experimentation Work-flow

1) Code Download and Environment Set Up:

- First do a git clone of the repository (https://gitlab.com/sayket/dataPlacementExperiments.git).
- Load CUDA environment.
- Go to the GPUTuning folder
  cd GPUTuning
- Modify 'make.common' to set up necessary paths, this is very important as scripts and makefiles derive necessary paths based on these variables.
- Modify 'exports.path.common.sh' to export necessary environment variables.

2) Running the Main Experiments: These experiments (in ***Main_Experiments*** folder) are designed to compare the impact of different memory and resource configurations. To run these experiments perform the following steps,

- Go to the 'Main_Experiments' folder
  cd Main_Experiments
- Run the script 'GPUTuner_run_shared_texture_constant.sh'
  source GPUTuner_run_shared_texture_constant.sh
  This script will run the necessary experiments related to CUDA Kernels (e.g. constant_memory, shared_memory, texture_memory) for different types of configurations (e.g. memory types and thread block size).
- The experiments will create a result file called 'exec-time.txt' in each of the application directory. This file contains the result for all the experiments for that application.
- Save these 'exec-time.txt' for a certain machine for future use (e.g. generating plots).

3) Running the CUPTI Experiments: To do a detailed analysis of a certain configuration you can choose to run the CUPTI experiments. These experiments collect CUPTI metrics for a certain machine. We only collect metrics that are common in all four GPUs.

- Go to the 'CUPTI_Experiments' folder
  cd CUPTI_Experiments
- Run the script 'CUPTI_run_shared_texture_constant.sh'
  source CUPTI_run_shared_texture_constant.sh
  This script will run the necessary experiments related to CUDA Kernels (e.g. constant_memory, shared_memory, texture_memory) and collect CUPTI metrics.
- The experiments will create a result file called 'output.csv' in each of the application directory. This file contains the all the CUPTI metrics for that application.

4) Evaluation and Results: To analyze the results of Main Experiments and generate requisite plots, perform the following steps,

- To analyze the results and generate plots for a specific benchmark (e.g. constant_memory, shared_memory, texture_memory), create a directory containing the results (e.g. exec-time.txt) for different GPUs.
- The scripts associated with analysis and plot generation assumes following directory structure,
  .
  ..
  /Volta
  /Pascal
  /Kepler
  /Graph
  Data_Analyzer_to_graph_script.sh
  exec_time_data_normalizer.py
  exec_time_data_summarizer.py
  Graph_script_Mem_Impact_Across_GPUBlock_Across_GPU.R
- Here each GPU named folder contains results for that specific GPU, So just copy 'exec-time.txt' related to a certain GPU to the respective GPU folder (e.g. copy 'exec-time.txt' for Volta to the /Volta folder)
- Run the 'Data_Analyzer_to_graph_script.sh' script with the absolute path to the current directory, and it will create the plots in pdf.
  source Data_Analyzer_to_graph_script.sh -d "path–to–current–directory"

Description of the individual files used for the analysis and plot generation process is described below,

- ***Data_Analyzer_to_graph_script.sh***: Runs necessary files (exec_time_data_summarizer.py, exec_time_data_normalizer.py) to normalize and summarize the data. Then runs the R script Graph_script_Mem_Impact_Across_GPUBlock_Across_GPU.R and Graph_script_Execution_Time_Across_GPU.R to generate the plots.
- ***exec_time_data_summarizer.py***: summarizes the data from 'exec-time.txt' and writes them to 'summary.csv'
- ***exec_time_data_normalizer.py***: normalizes the data from 'summary.csv' based on default GPU configuration and saves them in 'Normalized_Exec_Data-(GPUName)).csv'
- ***Graph_script_Mem_Impact_Across_GPUBlock_Across_GPU.R*** : Takes 'Normalized_Exec_Data-(GPUName)).csv' as input and generates the plots showing the impact of different configurations across different generations of GPUs.
- ***Graph_script_Execution_Time_Across_GPU.R***: Generates plot to compare the execution time of a certain application with certain memory type (e.g. running constant_memory application using constant memory) in different generation of GPUs.

F. Benchmark Application Experimentation Work-Flow

The following steps will report on the values for one platform. To run on multiple platforms, repeat step 1 and consolidate the data directories. All scripts are located within the BenchmarkApplications/ scripts directory.

1) Run benchmarks: Syntax for the running script is:

./ collectNvprofData.sh <DIR_TO_PUT_DATA_IN>

For example:

./ collectNvprofData.sh ../ testdata

2) Consolidate benchmark data: Syntax for the consolidating script is:

./ getPerformanceTimes.sh <DIR_DATA_IN> > <FILE_OUT> or whatever text file, but the R plotting script needs to be updtaed as this is currently hardcoded)

For example:

./ getPerformanceTimes.sh testdata >output.txt

NOTE: due to a bug must also run "sed −i "/ Script /d" <FILE_OUT> before going to step 3.

3) Re-create plots: syntax for the ploting script is:

Rscript plotBenchmarkSpeedupMedian.R <benchmark> <OUT>

For example:

Rscript plotBenchmarkSpeedupMedian.R mm output.txt

After that, plots will be found in analysis / plots /<benchmark>_speedup.pdf.

G. Proxy Application Experimentation Work-Flow

The code variants and the script are located within the ProxyApps directory. To run the experiments, each code variant should be compiled and executed to collect the time measurement. Each variant self-reports the timing after execution for the selected kernels in LULESH.

A compiler script, compile.sh, is provided to compile the code variants. The script takes two arguments: (1) the name of the variant and (2) the GPU architecture (sm_35, sm_60, sm_70). For example: "/compile.sh globalall sm_60" compiles the variant using global memory on the Pascal GPU platform.