Fractional GPUs: Software-based Compute and Memory Bandwidth Reservation for GPUs

Abstract—**GPUs are increasingly being used in real-time systems, such as autonomous vehicles, due to the vast performance benefits that they offer. As more and more applications use the GPU, more than one application may need to use the GPU in parallel. However, real-time systems also require predictable performance from each individual applications which GPUs do not fully support in a multi-tasking environment. Nvidia recently added a new feature in their latest GPU architecture that allows limited resource provisioning. This feature is provided in the form of a closed-source kernel module called the *Multi-Process Server* (MPS). However, MPS only provides the capability to partition the compute resources of GPU and does not provide any mechanism to avoid inter-application conflicts within the shared memory hierarchy. In our experiments, we find that compute resource partitioning alone is not sufficient for performance isolation. In the worst case, due to interference from co-running GPU tasks, read/write transactions can observe a slowdown of more than 10x.**

In this paper, we present Fractional GPUs (FGPUs), a software-only mechanism to partition both compute and memory resources of a GPU to allow parallel execution of GPU workloads with performance isolation. As many details of GPU memory hierarchy are not publicly available, we first reverse engineer these details through various micro-benchmarks. We find that the GPU memory hierarchy is different from that of the CPU, making it well-suited for page coloring. Based on our findings, we were able to partition both the L2 cache and DRAM for multiple Nvidia GPUs. Furthermore, we show that a better strategy exists for partitioning compute resources than the one used by MPS. An FGPU combines both this strategy and memory coloring to provide superior isolation.

We compare our FGPU implementation with Nvidia MPS. Compared to MPS, FGPU reduces the average variation in runtime from 135% to 9%. To allow multiple applications to use FGPUs seamlessly, we ported Caffe, a popular framework used for machine learning, to use our FGPU API.

Index Terms—**GPGPU, CUDA, partitioning, page coloring, memory hierarchy, cache structure, Program Co-Run**

I. Introduction

GPUs are becoming more powerful with each new generation and architecture. Also, real-time systems are increasingly deploying applications that use the GPU. This is driven by the increase in popularity of machine-learning applications, especially in domains such as autonomous vehicles, which exploit the massive parallelism provided by GPU. A single application may not be able to use the entire GPU, while multiple applications can benefit from using the GPU. These two trends makes it important to allow a GPU to run multiple applications. As real-time applications have strict deadlines, GPUs simultaneously need to provide predictable application performance even in worst case scenarios, especially for safety-critical applications.

To support these demands, Nvidia provides MPS [5] which allows multiple applications to co-run on GPU. They recently even added a new QoS feature in MPS that allows programmers to specify an upper limit on the number of GPU threads available for each application to limit available compute bandwidth on a per-application basis. The idea is that capping the portion of available threads will reduce destructive interference between applications. However, there are three issues with this MPS QoS feature: 1) It is only available on the latest Nvidia GPUs, 2) Its source code is not available making it a black-box and potentially unreliable, and 3) MPS only allows to partition the compute resources of GPU and does not provide any mechanism to avoid inter-application conflicts within the memory hierarchy.

Prior works [11] [17] [18] [24] [28] [29] [33] have shown that two applications, running on different cores on a CPU, can still affect each other's runtime due to conflicts in the memory hierarchy (mainly in shared cache and DRAM) due to the following reasons:

1) Cache set conflicts in the shared cache,
2) Miss Status Holding Registers (MSHR) contention in the shared cache,
3) Reordering of requests in the memory controller,
4) Row buffer conflicts in DRAM, and
5) DRAM bus contention.

[28] presents a comprehensive set of studies that investigate these sources of conflicts on the CPU. Each of these papers, including [28], design mechanisms to remove one or more of these sources of conflicts on CPU. But no such work has been attempted for GPU. In this paper, we first investigate the hierarchy of memory on GPU and then propose a page coloring mechanism to resolve these conflicts and run applications with isolation on GPU.

The main contributions of our paper are as follows:

1) We implemented a software-based GPU partitioning mechanism to run multiple applications in parallel with predictable performance. Specifically, we combine both compute and memory bandwidth isolation mechanisms to split a single GPU into smaller fractional GPUs. We show that it is possible to run multiple applications in parallel, each within a fractional GPU with a high degree of isolation.
2) We construct micro-benchmarks to reverse-engineer the publicly unknown architectures details of the L2 cache and DRAM in Nvidia GPU.

   Prior studies [15] [19] [20] have attempted to dissect the memory hierarchy of various Nvidia GPUs but no work has been able to explain the structure of L2 cache/DRAM (apart from some work on early simpler GPUs [30] [31]). Through micro-benchmarks, we reverse-engineer the architecture of the L2 cache and DRAM on multiple Nvidia GPUs. We show that the Nvidia GPU L2 cache is not structured in the same manner as a traditional CPU L2 cache, and is more sophisticated.

3) To the best of our knowledge, we are the first to implement page coloring on a GPU.
We show that the L2 cache/DRAM architecture of Nvidia GPU is well-suited for page coloring. Despite the Nvidia device driver being partially-closed source, we are able to implement page coloring.
4) We port Caffe [14] to use our FGPU abstraction.
Caffe is a widely-used GPU framework for *Deep Neural Networks* (DNN). Porting Caffe to FGPUs allows multiple DNNs to run on a single GPU in parallel with predictable runtimes.

This paper focuses on Nvidia GPUs (and corresponding SDK i.e. CUDA [21]) because they are the leading platforms for high-performance computing. Specifically, we choose GTX 1070, GTX 1080 and Tesla V100. We believe our reverse engineering algorithms are general enough to be applied to other GPUs. We leave this for future work.

The remainder of the paper is organized as follows. Section II provides some necessary background on CPU and GPU architectures. In Section III, we define the term *Compute Isolation* and then describe how we achieve it on Nvidia GPU. Similarly, in Section IV, we define *Memory Bandwidth Isolation*, reverse-engineer various details about Nvidia GPUs and then use these findings to implement memory bandwidth partitioning. Section V reports our experimental results, followed by related work, and conclusions.

II. Background

In this section we give some necessary background on GPU architecture using terminology from Nvidia's popular CUDA platform. We use the CUDA terminology for the rest of the paper for consistency.

A. GPU Compute Hierarchy

A GPU consists of tens of *streaming multiprocessors (SM)*, each in turn having a large number of hardware threads. But an SM is abstracted away from the programmer. Instead, CUDA provides a software abstraction called *blocks*, each of which is a collection of threads. To execute code on the GPU, a CUDA programmer specifies 1) a *kernel* (a set of instructions), 2) the number of blocks to be used, and 3) the number of threads within a block. CUDA then launches specified number of blocks, each having the same number of threads and each thread executing the same kernel (although operating on different data). A hardware GPU scheduler decides the mapping between blocks and SMs, with a single SM executing multiple blocks. However, since an SM can execute only a fixed number of blocks at a time, some blocks might be queued until the required hardware resource become available at any SM. The details about the Nvidia GPU hardware scheduler are not known publicly. To the CPU, the GPU looks like a co-processor on which it can launch CUDA kernels via the CUDA API, but it has no control over the scheduling algorithm.

For the rest of the paper, a *kernel* refers to a CUDA kernel, *device drivers* refer to OS kernel modules, *threads* refer to GPU hardware threads and a *process* refers to an OS process.

B. GPU Memory Hierarchy

A GPU today has three main levels in its memory hierarchy: 1) L1 cache 2) L2 cache and 3) DRAM. The L1 cache is private to an SM and shared by all the threads in that SM. All SMs share the L2 cache and DRAM. Nvidia does not reveal the details about the memory architecture of the GPU. Hence, we briefly discuss the traditional hierarchy of these memories on a CPU.

Caches are memory modules that are faster to access than DRAM but smaller, and used to cache frequently-accessed data. A cache is divided into m *sets*, and each set has n *cache lines* with a cache line being a collection of consecutive words. The address associated with a memory transaction dictates the cache set in which it might potentially lie. If the requested memory block resides within one of the cache lines in this set, it is a *cache hit*. Otherwise, a *cache miss* occurs, a cache line is chosen to be evicted for this set and the request is forwarded to DRAM. After the data is fetched from DRAM, the new cache line is placed in this set in lieu of the evicted cache line.

DRAM is organized as a set of *chips*, each further divided into *banks* and each bank has multiple *rows*. Each bank has a *row buffer* that is used to cache the last read row. When a request is issued to DRAM, based on the physical address, it is routed to a specific bank inside a specific chip. If the requested address lies on the row that is cached, the data is fetched from the row buffer. Otherwise, the row in the row buffer is evicted, and the new row that is being accessed is fetched into the buffer. The request is then fulfilled using the data in the row buffer. A row buffer miss is much slower than a row buffer hit due to the additional overhead of row buffer eviction.

The mapping from an address to a cache set or DRAM chip/bank/row is done via separate hash functions which are unique to the hardware under consideration. Nvidia reports that memory addresses are hashed on new GPUs [9] to spread them across DRAM but does not report these hash functions.

In the rest of this paper, *DRAM* refers to GPU DRAM, *cache sets* refer to GPU cache sets and *banks* refers to GPU DRAM banks.

III. Compute Isolation on GPU

A. Definition

Given two partitions, P_1 and P_2, on a single GPU and two sets of kernels, K_1 and K_2, compute isolation is said to be achieved between the partitions if, for any $k_1 \in K_1$ and

$k_2 \in K_2$, the run-time of k_1 (k_2) remains unchanged in the following scenarios:

1) k_1 (k_2) is running on P_1 (P_2) and P_2 (P_1) is empty.
2) k_1 is running on P_1 and k_2 is running on P_2 in parallel.

i.e. k_2 has no effect on k_1's run-time (k_2 does not take away k_1's compute resources) and vice-versa. While considering compute isolation, we do not consider the impact of k_2's memory transactions (reads/writes) on k_1's run-time through any conflicts in the memory hierarchy.

B. Implementation

A single GPU is split into multiple partitions P_i ($1 \leq i \leq n$, where n is the number of partitions). For each application, all the kernels in that application are assigned to a single kernel set K_i. All kernels $k_i \in K_i$ run on P_i, ensuring that all kernels within an application run in a single GPU partition. Compute isolation can be achieved between each pair of partitions by assigning disjoint sets of SMs to each partition P_i. For example, if a GPU has 10 SMs, 1 to 5 SMs can be assigned to P_1 and 6 to 10 SMs can be assigned to P_2. When k_1 runs, it will be forced to use only 1 to 5 SMs. Different kernels from the same kernel set will run one after another in a time-interleaved fashion, but kernels in different sets can run in parallel. It is possible to assign multiple applications to the same partition, in which case, the application running on different partitions will be compute-isolated from each other but applications running on the same partition will be affected by one another. The maximum number of compute partitions is equal to the total number of SMs as SM is the smallest compute unit allocated to a partition. Currently, Nvidia's GPU does not have hardware support for assigning specific SMs to a kernel. Hence, we implement compute isolation using a software partition. Our implementation is based on [32], with some extensions to improve performance. We now briefly describe the overall mechanism to achieve *SM affinity*:

1) Each application, during initialization, identifies the partition on which it wishes to run on.
2) At kernel launch, instead of launching nb blocks with nt threads in each block as specified by the programmer, we launch npb *persistent blocks* with nt threads. npb is calculated such that npb blocks occupy all the threads in all the SMs.
   npb =(Threads in SM / nt) * Number of SM[1].
3) Each persistent block identifies the SM it is running on by fetching its ID. If this SM is not in the set of the SMs assigned to the application's partition, the block exits without doing any work. Remaining persistent blocks, say npb', are left running on all of the correct SMs.
4) A centralized queue of block indexes, from 0 to $nb-1$, is implemented on GPU using atomic operations. The npb' persistent blocks execute the original nb blocks by popping block indexes from this centralized queue.

Original Code

```
// CUDA Kernel
// Implements c[i] = a[i] + b[i]
__global__ void vectorAdd(float *a, float *b, float *c) {
    int i = blockIdx * blockDim + threadIdx;
    c[i] = a[i] + b[i];
}
...
// CPU code
nt = 256;                // Number of threads
nb = num_elements / nt;  // Number of blocks
vectorAdd<<<nb, nt>>>(A, B, C); // Run nb blocks
....
```

Modified Code

```
__global__ void FGPU_DEFINE_KERNEL(vectorAdd, float *a,
    float *b, float *c) {
    int _blockIdx;
    FGPU_BLOCK_INIT();

    FGPU_FOR_EACH_BLOCK(_blockIdx) {
        int i = _blockIdx * blockDim + threadIdx;
        c[i] = a[i] + b[i];
    }
}
...
fgpu_init(partition_id)
...
FGPU_LAUNCH_KERNEL(vectorAdd, nb, nt, A, B, C)
...
```

Fig. 1: Comparison of original vector addition code and modified code to support compute isolation using FGPU API. *blockIdx*, *blockDim*, *threadIdx* are CUDA-supported keywords

Figure 1 shows an example application, vector addition, that adds two vectors, A and B, and returns the result into output vector C. We show the original code and the modified code that uses the FGPU API. In the original code, the CPU launches a *vectorAdd* kernel with nb blocks and nt threads in each block. Inside the kernel, each block calculates the index of a single element and calculates the sum for corresponding elements of the input vectors. *blockIdx* and *threadIdx* are CUDA-provided primitives that return the index of the current block and thread, respectively. *blockDim* specifies the number of threads in a block.

In the modified code, the kernel launch code and kernel code are modified to use macros provided by the FGPU API. At the start of the application, on the CPU, using *fgpu_init()*, the programmer indicates the GPU partition to be associated with the current application. At kernel launch, the *FGPU_LAUNCH_KERNEL()* macro takes a kernel, its launch parameters and kernel arguments as inputs. Instead of launching nb blocks, npb persistent blocks of *vectorAdd* are launched. Within each block, *FGPU_BLOCK_INIT()* performs two functions: 1) It exits the current persistent block if the SM is not in the partition, and 2) It initializes the queue containing the block indexes. Each remaining persistent block keeps fetching one block index at a time from the centralized queue using *FGPU_FOR_EACH_BLOCK()* and executes that block.

Though our current implementation requires modification of an application's source code, we believe that, because the changes required are minimal, it is possible to automate the process using compiler-assisted code transformation tech-

[1] Calculation of npb is more involved as apart from threads, blocks also compete for other SM resources (such as registers). We take care of this in our implementation.

niques. We leave this as future work.

Nvidia, via its normal API, does not allow kernels from different applications to run together. Hence, [32] requires modifying applications to merge them into a single application. In our implementation, we use the Nvidia MPS and bypass this limitation without the need to merge applications. This makes our approach easier to use.

MPS also has a QoS feature that allows programmers to specify an upper limit on the number of threads available for each application. However, FGPU does not use MPS for compute isolation. Because MPS do partitioning at thread granularity, it allows multiple applications to run on the same SM which can cause the applications to conflict over limited resources within the SM (e.g. threads registers). We show SM affinity based compute isolation is better than MPS QoS feature in Section V-A.

IV. Memory Bandwidth Isolation on GPU

A. Definition

Given two compute isolated-partitions, P_1 and P_2, on a single GPU and any two sets of kernels, K_1 and K_2, memory bandwidth isolation is said to be achieved between the partitions if, for any $k_1 \in K_1$ and $k_2 \in K_2$, the run-time of k_1 (k_2) remains unchanged in the following scenarios:

1) k_1 (k_2) is running on P_1 (P_2) and P_2 (P_1) is empty.
2) k_1 is running on P_1 and k_2 is running on P_2 in parallel.

i.e. memory transactions (reads/writes) issued by k_2 have no effect on time taken by k_1's memory transactions, and vice-versa. With only compute isolation and no memory bandwidth isolation, k_1's and k_2's reads/writes can conflict due to the factors mentioned in Section I.

To understand how to partition memory hierarchy to avoid conflicts, it is required to understand the Nvidia GPU memory hierarchy, especially the L2 cache and DRAM, the chief memory components that are shared among SMs. Unfortunately, these details are not publicly available. In the following sections, we formulate experiments and algorithms to reverse-engineer the L2 cache and the DRAM hierarchy of the Nvidia GPU using GTX 1080 as an example. We later verify that a similar memory hierarchy also exists in other Nvidia GPU chipsets.

An Nvidia GPU has other types of memory [22], namely *L1 cache*, *shared memory*, *constant memory*, and *textured memory* that can be optionally used by applications. The L1 cache and shared memory are private to each SM and hence are not relevant for memory bandwidth reservation as each partition has disjoint sets of SMs. Constant memory is small in size (64 KB in GTX 1080) and cached. Hence, we do not expect much contention over constant memory. We leave exploring the effects of conflicts over texture memory as future work (No application in our evaluation used texture memory).

B. Reverse Engineering of DRAM Bank Addressing

If two applications, try to access the same DRAM bank, they can evict each other's cached data in the row buffer. Ideally, each memory bandwidth-isolated partition should have their own disjoint set of DRAM row buffers and by extension, disjoint sets of DRAM banks. This can be achieved by ensuring that addresses accessed by the applications in different partitions lie on different banks. This requires first understanding the relationship between physical addresses and the banks that they lie on. We describe an algorithm to discover the hash function that maps physical address to a bank. The algorithm exploits the fact that accessing two addresses that lie on the same bank but different rows (which causes row buffer eviction) will be noticeably slower than accessing two addresses that lie on different banks.

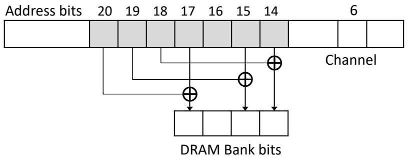


Fig. 2: Mapping from physical address to DRAM bank for Intel Core i7-2600 [28]

We make the following assumptions about the GPU memory hierarchy:

1) For a given physical address, bits of index of the corresponding bank are derived from the XOR sum of two or more bits of that physical address.
2) The GPU caches use *Least Recently Used* (*LRU*) as their eviction policy.

Assumption (1) is based on prior work [28] [17] on finding DRAM bank bits on CPU. XORing of bits is done to randomize the mapping function to decrease the likelihood of bank collisions. Figure 2 shows the mapping function for Intel Core i7-2600 as reverse-engineered in [28]. Also, we assume that the mapping is based on physical addresses, rather than virtual addresses, because we expect that by the time a request arrives at DRAM, address translation should have already taken place. To witness a row buffer eviction, read/write transactions must arrive at DRAM. For this to occur, caches need to be bypassed. We could not find any CUDA instruction that either disabled the L2 cache or flushed it. Hence, if we assume that caches in GPU are LRU (Assumption (2)), we can implicitly clear the cache by reading enough spurious data. The amount of spurious data read must be at least equal to the size of the cache. This is unlike CPU where explicit instructions exist that flush cache (e.g. *clflush* on x86). In the GPUs we tested, this assumption seems to hold strongly.

Algorithm 1 gives the pseudo code for reverse-engineering the hash function for bits of bank index. As per Assumption 1, we need to know the physical address for finding the hash function. Since CUDA does not provide any API to query the physical address from a given virtual address, we modified the Nvidia device driver[2] to (a) add a new API to allocate a contiguous physical chunk of GPU memory, (b) perform virtual address to physical address translation for this chunk,

[2] The released code for Nvidia's Linux device driver is partially closed. Fortunately, some of the memory management code is publicly available. Modules such as the GPU scheduler are completely-closed source.

Algorithm 1 Hash function to reverse-engineer GPU DRAM bank bits

```
1:  /* Code Executed by CPU */
2:  function PrintBankMappingFunctions
3:      /* Find size of free memory on GPU */
4:      $S \leftarrow GetGpuFreeMem()$;
5:      /* Allocate contiguous physical memory chunk */
6:      $\langle VirtStart, PhyStart\rangle \leftarrow AllocGpuPhyMem(S)$
7:      $Pairs \leftarrow \emptyset$
8:      /* Collect all pairs */
9:      for $Offset \leftarrow 1$ to $S$ do
10:         $Vaddr \leftarrow Offset + VirtStart$
11:         $Paddr \leftarrow Offset + PhyStart$
12:         /* Virtual Primary and Secondary Addresses */
13:         if $IsGpuRowEvicted(VirtStart, Vaddr)$ then
14:             $Pairs.append(PhyStart, Paddr)$
15:         end if
16:     end for
17:     $F \leftarrow GenerateAllHashFunctions()$
18:     /* Test all permutations of mapping function */
19:     for each $func \in F$ do
20:         if $IsValidMapping(func, Pairs)$ then
21:             $Print(func)$
22:             return
23:         end if
24:     end for
25: end function
26: /* Code Executed by GPU */
27: function IsGpuRowEvicted($PrimaryAddr, SecondaryAddr$)
28:     /* Clear cache by reading enough spurious data */
29:     $ClearGpuCache()$
30:     $StartTime \leftarrow GetClock()$
31:     /* Access Primary and Secondary Addresses */
32:     $data \leftarrow PrimaryAddr$
33:     $data \leftarrow SecondaryAddr$
34:     $AccessTime \leftarrow GetClock() - StartTime$
35:     if $AccessTime \geq Threshold$ then
36:         return $True$
37:     else
38:         return $False$
39:     end if
40: end function
```

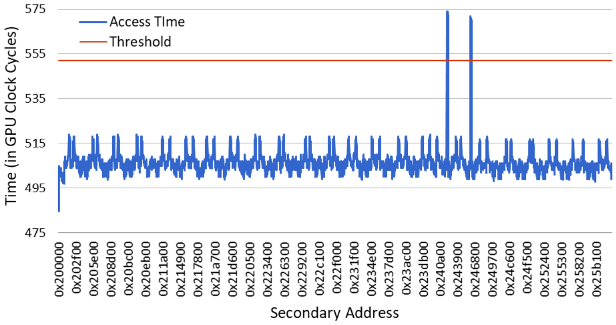


Fig. 3: DRAM access time of pair of addresses on DRAM. Secondary address is varied keeping primary address fixed. Only 3K samples are shown here. Threshold is set to 1.1 times the average access time.

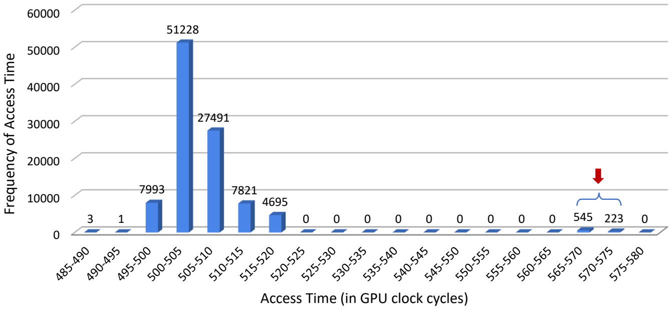


Fig. 4: Histogram of DRAM access time of pairs of addresses. Only 100K samples are shown here. As highlighted, there are only few addresses that have high latency (more than 1.1 times the average).

and (c) return the starting virtual and physical address (shown as *AllocGpuPhyMem()* in the pseudo code). This allows us to calculate the physical address for any virtual address (lines 1-11). In the loop (lines 9-16), we vary the secondary address, keeping the primary address fixed at the start address and find multiple pairs of addresses that cause row buffer eviction by identifying spikes in access time. We then find the valid hash function (i.e. a hash function that assigns the same bank to all the pairs of addresses) by searching over all possible XOR based functions (based on Assumption (1)). If all pairs are tested, only one hash function is identified as valid.

We can further improve this naive brute-force approach by intelligently identifying pairs of addresses to test so as to eliminate candidate functions and stop when we are left with just one function. (Interested readers can see our implementation[3]). All of our reverse engineering experiments ended within 1 hour on all of the GPUs that we tested.

Figure 3 shows the access times for a subset of pairs of addresses and Figure 4 shows the histogram. Most of the access times lie between 495-520 cycles. Few pairs have access time more than the threshold (552 cycles). This bi-modal distribution verifies that the row buffer miss causes significant access time penalty to make it possible to differentiate it from row buffer hit on GPU. Figure 6(b) details the hash function we found experimentally for the GTX 1080. We will verify this hash function experimentally in Section IV-E.

C. Reverse Engineering of L2 Cache set Addressing

If two partitions access addresses that lie on the same cache set in a shared cache, they can evict each other's cache data due to a conflict. To avoid this scenario, for memory bandwidth-isolated partitions, we can divide the cache sets between the partitions so that each partition has disjoint sets of cache sets. To find the mapping among a physical address and its corresponding L2 cache set, we follow an approach similar to Algorithm 1. The algorithm is described in detail in Appendix A, and reveals that the Nvidia GPU L2 cache is unlike a traditional CPU L2 cache. Figure 5 shows a typical cache set addressing mechanism on a CPU. In contrast, we found that cache set addressing for the GPU L2 cache is similar to bank addressing. Figure 6(a) details the mapping function we found

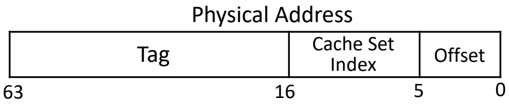
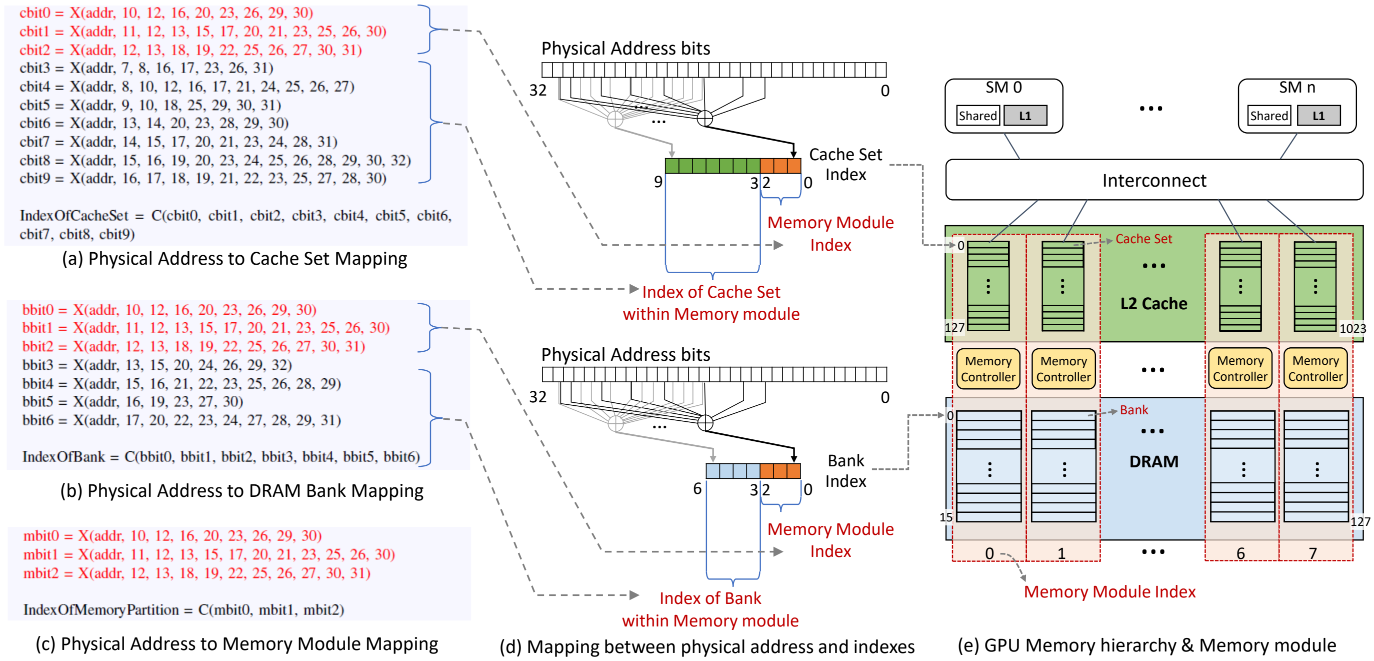


Fig. 5: A typical cache set addressing using a physical address on the CPU

[3] Our source code will be available through the track chair upon request.



cbit0 = X(addr, 10, 12, 16, 20, 23, 26, 29, 30)
cbit1 = X(addr, 11, 12, 13, 15, 17, 20, 21, 23, 25, 26, 30)
cbit2 = X(addr, 12, 13, 18, 19, 22, 25, 26, 27, 30, 31)
cbit3 = X(addr, 7, 8, 16, 17, 23, 26, 31)
cbit4 = X(addr, 8, 10, 12, 16, 17, 21, 24, 25, 26, 27)
cbit5 = X(addr, 9, 10, 18, 25, 29, 30, 31)
cbit6 = X(addr, 13, 14, 20, 23, 28, 29, 30)
cbit7 = X(addr, 14, 15, 17, 20, 21, 23, 24, 28, 31)
cbit8 = X(addr, 15, 16, 19, 20, 23, 24, 25, 26, 28, 29, 30, 32)
cbit9 = X(addr, 16, 17, 18, 19, 21, 22, 23, 25, 27, 28, 30)

IndexOfCacheSet = C(cbit0, cbit1, cbit2, cbit3, cbit4, cbit5, cbit6, cbit7, cbit8, cbit9)

(a) Physical Address to Cache Set Mapping

bbit0 = X(addr, 10, 12, 16, 20, 23, 26, 29, 30)
bbit1 = X(addr, 11, 12, 13, 15, 17, 20, 21, 23, 25, 26, 30)
bbit2 = X(addr, 12, 13, 18, 19, 22, 25, 26, 27, 30, 31)
bbit3 = X(addr, 13, 15, 20, 24, 26, 29, 32)
bbit4 = X(addr, 15, 16, 21, 22, 23, 25, 26, 28, 29)
bbit5 = X(addr, 16, 19, 23, 27, 30)
bbit6 = X(addr, 17, 20, 22, 23, 24, 27, 28, 29, 31)

IndexOfBank = C(bbit0, bbit1, bbit2, bbit3, bbit4, bbit5, bbit6)

(b) Physical Address to DRAM Bank Mapping

mbit0 = X(addr, 10, 12, 16, 20, 23, 26, 29, 30)
mbit1 = X(addr, 11, 12, 13, 15, 17, 20, 21, 23, 25, 26, 30)
mbit2 = X(addr, 12, 13, 18, 19, 22, 25, 26, 27, 30, 31)

IndexOfMemoryPartition = C(mbit0, mbit1, mbit2)

(c) Physical Address to Memory Module Mapping

(d) Mapping between physical address and indexes

(e) GPU Memory hierarchy & Memory module

Fig. 6: (a-b) Mapping from Physical Address to DRAM Bank, Cache Set and Memory Module for GTX 1080, where, X($addr$, x_1, x_2 .., x_n) = $\bigoplus_{i=1}^{n} addr[x_i]$ i.e. X takes a physical address and n bit indices and returns XOR sum over all those bits of address ($addr[x_i]$ refers to x_i^{th} bit of $addr$). And C(v_1, v_2, ..., v_n) = $v_n \parallel v_{n-1} \parallel \cdots \parallel v_1$ i.e. C concatenates bits together. There are 2^7 banks and 2^{10} cache sets. (c-d) There are 3 bits common in the indexes of bank and cache set which have been highlighted. These bit define a memory module. Mapping for GTX 1070 is identical to GTX 1080 as they are of the same architecture. Mapping for Tesla V100 is similar but different in number of cache sets, banks, memory module, and hash functions. (e) Our understanding of Nvidia GPU memory hierarchy based on experiments. L2 cache is split into sets of cache sets and DRAM is split into sets of banks. We call corresponding sets of cache sets and banks together as a "Memory Module".

experimentally for the GTX 1080. To verify the hash function, we use a Nvidia *nvprof* [8], a GPU profiler capable of listing the number of L2 cache hits and misses. We generate sets of addresses that lie on same and different cache sets using the hash function. Our experiments show that $nvprof$ reports a high cache miss rate when addresses in same cache sets are accessed and low miss rates when addresses in different cache sets are accessed (Some cache misses occur due to cold cache misses).

D. GPU Memory Hierarchy

We ran our experiments (Algorithm 1 and 2) on GTX 1070, GTX 1080 and Tesla V100 GPU and found the hash functions to be similar to those shown in Figure 6(a) and (b). For these all GPUs, we found that the bank index and cache index have d LSB bits common, where d depends on GPU architecture. For GTX 1070 and GTX 1080, d is 3 and for Tesla V100, d is 5. Based on these measurements, we conclude that the overall GPU architecture is designed as shown in Figure 6(e).

We define a *Memory module* as a partition of the GPU memory hierarchy that has a set of cache sets and banks such that no two memory modules share any cache sets or banks. All the cache sets and banks that have the same common d bits in their indexes lie on the same memory module. Hence, the total number of memory modules is $2^3 = 8$ in GTX 1070/GTX 1080 and $2^5 = 32$ in Tesla V100. Our findings match with Figure 7 which shows the architecture of Nvidia GPU GTX 970. This further validates our results. Based on Figure 7, we also suspect that each memory module has an independent memory controller and DRAM bus.

Our micro-benchmark on GTX 1080 also revealed some other interesting properties of the L2 cache (such as cache line size, set associativity, and cache replacement policy) and DRAM which we state in Appendix B as they are not relevant for the rest of the paper.

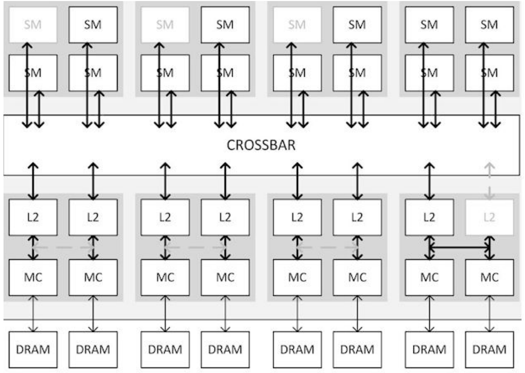


Fig. 7: Architecture of Nvidia GPU GTX 970. *MC* refers to a memory controller.

E. Page Coloring

The idea behind page coloring is to assign specific physical pages to different GPU partitions so as to limit memory interference between any two GPU partitions. Physical pages can be allocated such that no two GPU partitions access intersecting cache sets and therefore cannot evict each other's cache lines. Similarly, they can be allocated such that GPU partitions do not share DRAM banks, thereby avoiding row buffer conflicts. For a Nvidia GPU, as noted in Section IV-D, it is also possible to avoid both cache conflicts and DRAM bank conflicts together by assigning pages lying on different memory modules to different partitions.

To evaluate which approach gives better isolation, we conduct a series of experiments. In each experiment, we launch n CUDA blocks on GTX 1080, each containing one thread, where n varies from 1 to 50. We refer to the thread in the first block as *Primary Thread* and all the other threads as *Secondary Threads*. All the threads are assigned distinct sets of addresses that they read in a loop. However, all the addresses assigned to a single thread lie on the same cache set and the same bank. The addresses accessed by the primary thread are kept constant while the addresses accessed by the secondary threads are varied according to five different cases:

1) Addresses lying on the same cache set and the same bank as addresses accessed by the primary thread (SCSB).
2) Addresses lying on a different cache set but the same bank (DCSB).
3) Addresses lying on the same cache set but a different bank (SCDB).
4) Addresses lying on a different cache set and a different bank but the same memory module (DCDB).
5) Address lying on a different memory module (and hence, implicitly, on a different cache set and a different bank) (DM).

Under these configurations, we measure the average time taken by the primary thread to access a single word. More interference by secondary threads will lead to a proportionally larger increase in time taken by primary thread. The aim of the experiment is to find the case in which the primary thread is most isolated from interference due to the secondary threads. We take the following precautions in constructing the experiment:

1) To avoid any interference other than memory interference (i.e compute interference), using a technique similar to the computation isolation implementation we discussed in Section III-B, we ensure that the secondary threads do not run on the same SM as the primary thread.
2) To ensure that we see bank conflicts, each thread accesses enough addresses to spill the L2 cache so that all the accesses reach DRAM (i.e. accessing more cache lines than set associativity).

Figure 8 shows the average time taken by the primary thread in all five scenarios. As expected, the worst interference is seen when both the primary thread and the secondary threads access the same cache set and bank (*SCSB*). This is closely followed by the case when the primary thread and the secondary threads access different cache set but the same bank (*DCSB*). In both these cases, the interference is mainly coming from bank conflicts. For *SCDB* and *DCDB*, we believe that the interference comes from conflicts over *Miss Status Holding Register (MSHR)* in the L2 cache. MSHRs are limited registers that track the pending cache misses. One MSHR is allocated for each request that causes a cache miss and freed only after the data is fetched from DRAM. Further requests which cause cache misses are blocked till one of the MSHRs is freed. Based on the knee of the curve for *DCDB*, the number of MSHRs for an L2 cache partition within a single memory module appears to be 16. So, even when the primary and secondary threads access different cache sets and banks, we see interference because they access the same memory module. The best case with respect to isolation is *DM* where the primary thread sees no interference. Comparing the worst and the best cases, we see that memory bandwidth partitioning can potentially reduce interference by a factor of more than 10x.

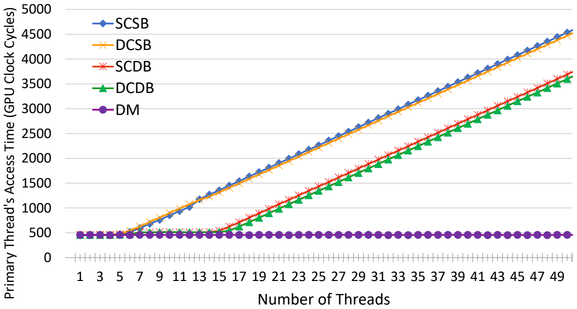


Fig. 8: Variation in the average word access time by primary thread due to cache set/bank conflict on GTX 1080. Legend indicates the relation between address accessed by primary and secondary threads.

In these experiments, we used the hash function we found in Section IV-B to assign addresses to primary and secondary threads. We then see a sharp interference between the threads when they access the same bank. Hence, we are confident that Algorithm 1 produced the correct hash function.

F. Implementing Page Coloring

Based on the discussion in Section IV-E, to achieve memory bandwidth isolation, the best methodology is to assign addresses from different memory modules to different GPU partitions. Furthermore, by looking at Figure 6(e) and Figure 7 , we can see that assigning each partition its own memory module allows each partition to have an independent memory controller and DRAM bus, apart from a disjoint sets of L2 cache set and banks, which should give a high degree of isolation. Taking the example of GTX 1080 which has 8 memory modules, we can have up to 8 GPU partitions, each assigned a distinct memory module or *color*. All applications running in a GPU partition will be allocated addresses from only that memory module. The finest granularity of allocation of physical memory to an application is a page. Hence, for

Name	Source	Description
MM	CUDA SDK	Matrix Multiplication
SN	CUDA SDK	Sorts Array
VA	CUDA SDK	Vector Addition
SP	CUDA SDK	Scalar Product
FWT	CUDA SDK	Fast Walsh Transform
CFD	Rodinia	Computational Fluid Dynamics

TABLE I: Applications used for Evaluation

page coloring, it is required that all the addresses within a single page must reside in the same memory module. By inspecting the code of the Nvidia device driver, we find that the latest Nvidia GPU architectures (Pascal [3] and Volta [7]) support multiple page sizes: 4 KB, 64 KB and 2 MB (All architectures support 4 KB pages). By looking at Figure 6(c), we see that for GTX 1080, the only possible page size that we can use for page coloring is 4 KB. All the addresses within a 4 KB page will either lie on $M_0 - M_3$ or on $M_4 - M_7$, where M_i is the i^{th} memory module. This is because the third bit of the memory module index (*mbit2*) does not use any bit of physical address below 12 (and 2^{12} = 4 KB). A page of size 64 KB or 2 MB will have its addresses lying across all the memory modules. So, for GTX 1080, even though we have 8 memory modules, only 2 memory colors are possible ($M_0 - M_3$ and $M_4 - M_7$). Similarly, for Tesla V100, 8 memory colors are possible from 32 memory modules.

We next implemented page coloring code in the Nvidia device driver. Each GPU maintains one free-page list per memory color that contains all the free pages of that color. On GPU initialization, all the pages are placed in a corresponding free-page list. An application, before allocating any GPU memory, indicates to the device driver which color it wishes to use. All future allocations of GPU memory for that application are fulfilled using the appropriate free-page list.

V. Evaluation

In this section, we evaluate the effectiveness of our compute and memory bandwidth isolation using various applications.

A. Micro-benchmark Experiments

Table I lists the applications we use for micro-benchmark study. This set contains a mix of computation and memory-intensive applications taken from the CUDA SDK [2] and Rodinia benchmark suite [10]. We conduct the evaluation on two Nvidia GPUs: GTX 1080 and Tesla V100. Table II presents the specifications of the platforms tested. We partition each GPU into two equal partitions, P_1 and P_2, using the following approaches:

1) Compute Partitioning only (*CP*).
   Each partition is assigned disjoint sets of SM.
2) Both Compute and Memory Partitioning (*CMP*).
   Each partition is assigned disjoint sets of SM and different memory colors.
3) Nvidia MPS QoS feature [4] (*MPS*).
   Each partition can use only up to 50% of total threads in the GPU. Only GPUs with Volta architecture have this QoS feature [5] hence we omit this approach from the GTX 1080 evaluation.

Components	System 1	System 2
GPU platform	GTX 1080	Tesla V100
GPU Arch.	Nvidia Pascal	Nvidia Volta
GPU SMs	20 SMs	80 SMs
GPU L2 cache	2 MB	6 MB
GPU DRAM	8 GB GDDR5X 320GB/s	16 GB HBM2 900GB/s
OS & CPU	x86 ubuntu16.04 3.6Ghz Intel I7-7700	AWS Cloud 2.3Ghz Intel Xeon E5

TABLE II: System Specifications for Evaluation

For each application, A, we run multiple scenarios:

1) A running on P_1. P_2 is idle.
2) A running on P_1 and kernels of compute-intensive Matrix Multiplication (MM) running on $P2$.
3) A running on P_1 and kernels of memory-intensive Fast Walsh Transform (FWT) running on $P2$.
4) A running on P_1 and kernels of memory-intensive Vector Addition (VA) running on $P2$.

We measure the average runtime of A's kernels in the above scenarios for all different partitioning approaches. With a perfect partitioning technique, the runtimes in all scenarios should be equal (i.e. there will be no interference from co-running kernels). The amount of variation in the runtime defines the amount of interference. So, we define the following metric for measuring isolation:

$$Variation_A = (\frac{Max(T_{A,MM}, T_{A,FWT}, T_{A,VA})}{T_{A,-}} - 1) * 100$$

where $T_{A,B}$ refers to the average runtime of A's kernels with B's kernels running on other partition (And $T_{A,-}$ indicates that P_2 is idle). A lower $variation$ signifies more predictable runtimes.

As a baseline, we measure the runtime of A's kernels when it runs alone on a GPU without any partitioning. We normalize all the runtimes by the factor of $2 * T_{baseline}$ (since we expect the runtime of A on 50% partition of the GPU to increase by 2x). Figure 9(a) shows the results for GTX 1080 and 9(b) shows the results for Tesla V100. Table III reports the average and max $variation$ across all applications.

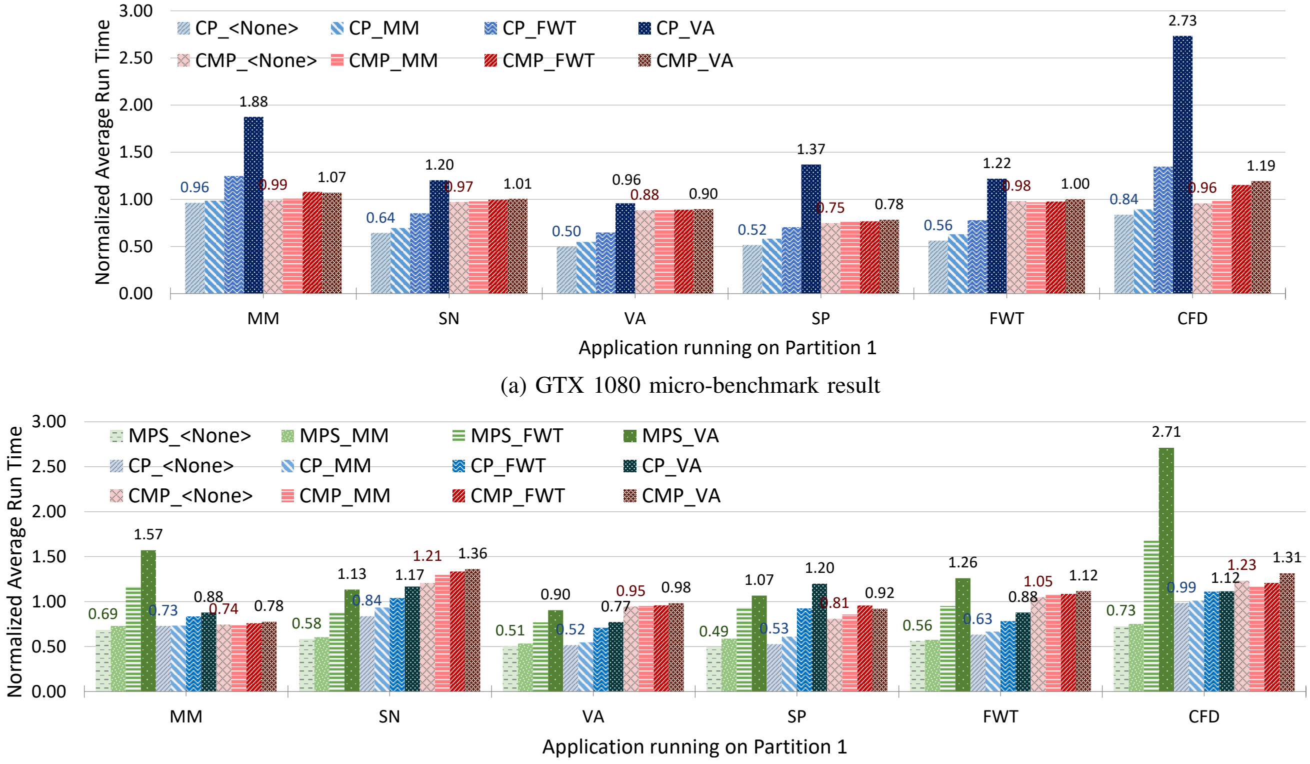
Key observations from the evaluation on GTX 1080:

1) CMP is much better than CP for predictable performance.
   Computation isolation is clearly not sufficient for predictable performance. Without memory isolation, a memory-intensive application can cause high interference. This is the reason why vector addition (VA) causes the most slowdown as it is highly memory-intensive. Adding memory bandwidth isolation reduces the $variation$ from 130.4% to 7.5% on average.

GPU / Technique	Average Variation	Max Variation
GTX 1080 / CP	130.4%	226.7%
GTX 1080 / CMP	7.5%	24.6%
Tesla V100 / MPS	135.8%	273.3%
Tesla V100 / CP	48.1%	127.7%
Tesla V100 / CMP	8.7%	18.3%

TABLE III: Variation in Runtime

[4] MPS QoS feature explained in Section III-B



(a) GTX 1080 micro-benchmark result

(b) Tesla V100 micro-benchmark result

Fig. 9: Results of micro-benchmark study. Legend is of the form <Partitioning Method>_<Application running on Partition 2>. Normalized average runtimes are reported for kernels of application running on Partition 1. 1000 samples are collected for each application.

2) There is a tradeoff between predictability and performance.
   The runtimes of all applications are shorter for the case of CP_<None> as compared to CMP_<None>. This effect is important and is because memory bandwidth partitioning splits the bandwidth between the GPU partitions, reducing the memory bandwidth available to a single partition even when the other partition is *idle*. This effect naturally has more impact on memory-intensive applications which are sensitive to the memory bandwidth available.
3) Some applications under-utilize GPU resources.
   All applications that have normalized runtimes less than 1 indicate that they are under-utilizing the GPU. For example, for the case of CP_<None> for vector addition, its normalized runtime is 0.5 even though it is getting half the compute resources as compared to the baseline. This means that its runtime is the same when running on a single partition as when running on the whole GPU (since runtimes are normalized with $2 * T_{baseline}$). The reason is that VA being a memory-intensive application spends most of the time being stalled on reads/writes. Hence, a reduction in compute resources does not lead to an increase in runtime.

The results for Tesla V100 are similar to those for GTX 1080 with a few differences:

1) CP is better than Nvidia MPS for predictable performance.
   The average $variation$ for MPS is 135.8% whereas it is considerably lower for CP at 48.1%. This is because MPS partitions at thread granularity whereas CP partitions at SM granularity. Hence, in MPS partitioning, two applications can run on the same SM causing them to share SM resources, leading to potential conflicts.
2) CP performs better on Tesla V100 than on GTX 1080. This is because Tesla V100 has higher memory bandwidth. It has 3x more DRAM bandwidth, a 3x larger L2 cache, and 4x more DRAM banks. These extra resources reduce the probability of conflicts in the L2 cache/DRAM. However, CMP is still better in terms of average and max $variation$. Since Tesla V100 is the largest Nvidia GPU currently [6], we believe the gap between CMP and CP/MPS to only be larger for other GPUs (such as in case of GTX 1080).
3) CP and CMP have high overheads for SN and CFD.
   This is evident when comparing MPS_<None> with CP_<None> for SN and CFD. The implementation of compute isolation has a fixed overhead per kernel which becomes relatively large for small kernels. As Tesla V100 is more powerful than GTX 1080, all kernel runtimes reduce. Both SN and CFD have multiple small kernels. Hence, for SN and CFD, there is a relatively

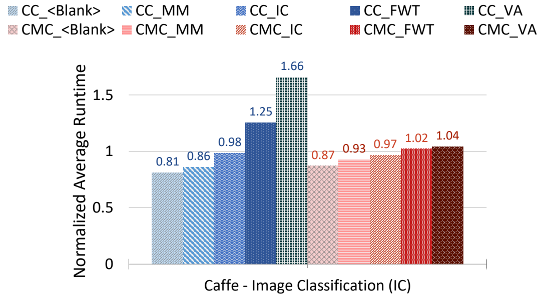


Fig. 10: Results for experiment using Caffe on GTX 1080. Normalized average runtimes are reported for IC which is running on Partition 1. 1000 samples are collected.

large overhead on Tesla V100 for compute partitioning. Since, for the other applications, MPS_<None> and CP_<None> are comparable, it indicates that compute partitioning has relatively small overhead for them.

We also ran the same micro-benchmarks on Tesla V100 using 4 memory partitions instead of 2. The results are similar to Figure 9(b) and presented in Appendix C.

B. Caffe Framework Experiments

We ported *Caffe* [14] to use our FGPU API. Caffe uses the *cuBLAS* library [1], a CUDA implementation of the *Basic Linear Algebra Subroutines (BLAS)*, which is closed source. Since our current FGPU implementation requires kernels to be modified slightly, we replaced cuBLAS routines with native CUDA kernels. Porting Caffe makes it possible for a vast number of existing applications to enjoy the benefits of FGPU without needing any modifications.

To evaluate Caffe performance, we ran an image classification *(IC)* application on Caffe that uses an AlexNet [16] model trained on the ImageNet [12]. We ran the same experiments as in Section V-A with IC running on the first partition. Also, we add another scenario where IC also runs on the second partition, acting as an interfering task. Instead of just measuring kernel execution time, we measure the total time taken by IC. IC is GPU-intensive with the CPU being used mostly for transferring data between the CPU and GPU DRAM. Figure 10 shows the results for GTX 1080. $Variation_{IC}$ for compute partitioning is 104.6% whereas adding memory partitioning reduces it to 19.5%. While the compute and memory partitioning of our FGPU reduces $variation$ dramatically relative to their absence, there is still non-negligible interference. Safety-critical systems need to be aware of this limitation and use appropriate safety margins.

C. Related work

Prior studies [15] [19] [20] [30] [31] have attempted to dissect the memory hierarchy of various Nvidia GPUs across different architectures. Work in [30], [31] evaluated early simpler Nvidia GPUs (pre-2010) which had L2 cache structure similar to CPU. In [19], the authors were unable to identify the structure of L2 cache on newer architectures, and they state that none of the GPU that they evaluated used traditional memory addressing. [15] used the techniques introduced in [19] to find set associativity of a GPU with Volta architecture, but they also did not find the cache structure. Authors in [20] tried to micro-benchmark C2070 GPU but got noisy data and stated that C2070's L2 cache had on an average 13.7 set associativity. To the best of our knowledge, no prior work has attempted to dissect the structure of DRAM on GPU. In this work, we focused on newer Nvidia GPU and implemented page coloring by reversed engineering the structures of both L2 cache and DRAM.

Page coloring is a well-know technique on CPU. [18] implemented cache coloring on CPU for real-time systems and [17] implemented DRAM bank coloring. [28] combined both cache and bank coloring on multi-core CPU systems. Since page coloring does not solve the problem of DRAM bus contention on CPU, other works [11] [24] [33] attempted to analyze or limit its impact. [29] showed that cache coloring did not solve the problem of conflict over MSHR in shared cache and implemented hardware extensions in a simulator to show how to reduce the effect of these conflict. FGPU uses the technique of page coloring for splitting memory resources of GPU. We show that the way GPU memory hierarchy is structured makes the page coloring effective solution to reduce interference between co-running tasks.

Regarding partitioning GPU compute resources, our implementation is based on the work done in [32]. Their work is focused on how to control scheduling on GPU through *SM-centric* kernel transformation and leverage this control to increase system throughput and decrease average turnaround time. [13] uses SM-centric kernel transformation to improve the throughput of tasks in Cholesky Factorization. Authors for [26] came up with a heuristic for dividing SM between co-running task using SM-centric transformation to increase the total number of tasks that can be scheduled in the system and looks into schedulability analysis. In this work, we do not focus on scheduling algorithms for running multiple application on GPU but instead, focus on how to best isolate applications running in different partitions from each other. We show that apart from compute partitioning, memory bandwidth isolation is necessary for this. Authors in [23] also observed resource contention between co-running workloads on GPU but did not explain which resources were causing the contention.

VI. Conclusions

This paper presents a software-based mechanism to allow multiple applications to run in parallel on a GPU while still maintaining isolation by partitioning compute and memory resources among these co-running tasks. Partitioning a single large GPU into smaller fractional GPUs opens up more options for real-time system architects regarding scheduling GPU resources. Our evaluations show that applications using FGPUs have significantly more predictable runtimes irrespective of other applications running in parallel. We also present various details about Nvidia GPU which were previously unknown in the public literature. We hope these insights can help programmers better understand the GPU architecture and optimize the performance of their program.

References

[1] NVIDIA cuBLAS. https://developer.nvidia.com/cublas.

[2] NVIDIA CUDA Toolkit. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html.

[3] NVIDIA GP100 Pascal Whitepaper. http://www.nvidia.com.

[4] NVIDIA GTX 1080. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[5] NVIDIA MPS. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[6] NVIDIA Tesla V100 Tensor Core. https://www.nvidia.com/en-us/data-center/tesla-v100/.

[7] NVIDIA V100 Volta Whitepaper. http://www.nvidia.com.

[8] NVIDIA Visual Profiler User's Guide. http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[9] Optimizing Matrix Transpose in CUDA. http://developer.download.nvidia.com/compute/DevZone/C/html64/6_Advanced/transpose/doc/MatrixTranspose.pdf.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[11] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *8th IEEE International Conference on Embedded Software and Systems*, pages 1068–1075. IEEE, 2011.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[13] J. Janzén, D. Black-Schaffer, and A. Hugo. Partitioning gpus for improved scalability. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on*, pages 42–49. IEEE, 2016.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

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

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 367–376. ACM, 2012.

[18] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.

[19] X. Mei and X. Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[20] R. Meltzer, C. Zeng, and C. Cecka. Micro-benchmarking the c2070. In *GPU Technology Conference*. Citeseer, 2013.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[22] C. Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[23] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 353–364. IEEE, 2017.

[24] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 741–746. IEEE, 2010.

[25] R. H. Saavedra-Barrera. *CPU performance evaluation and execution time prediction using narrow spectrum benchmarking*. PhD thesis, University of California, Berkeley, 1992.

[26] S. K. Saha. *Spatio-Temporal GPU Management for Real-Time Cyber-Physical Systems*. PhD thesis, UC Riverside, 2018.

[27] A. J. Smith and R. H. Saavedra. Measuring cache and tlb performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, (10):1223–1235, 1995.

[28] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 685–692. IEEE, 2013.

[29] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–12. IEEE, 2016.

[30] 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*, pages 1–11. IEEE, 2008.

[31] 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*, pages 235–246. IEEE, 2010.

[32] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130. ACM, 2015.

[33] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308. IEEE, 2012.

Appendix A
Discovering GPU Cache Set Hash Function

In this section, we describe the algorithm to find out the mapping between a physical address and the index of the corresponding cache set for L2 cache in GPU. Our algorithm relies on two facts 1) Cache lines that lie on the same cache set can evict each other and 2) Accessing a cached word is much faster than accessing a word from DRAM.

We make following assumptions about the GPU memory hierarchy (similar as in Section IV-B):

1) For a given physical address, bits of the index of the corresponding cache set are derived from XOR sum of two or more bits of that physical address.
2) GPU caches use LRU as their eviction policy.

Algorithm 2 outlines the pseudo code. They key idea is same as in Algorithm 1. We collect enough pair of addresses that we know lie on same L2 cache set and then find the valid hash function. To find the pairs, we use P-chase (pointer chase) [25], [27] for our algorithm. P-chase list is essentially a linked list traversal. The key is that the elements of the linked list are placed at addresses that we wish to read. Since P-chase list traversal is data dependent (address of next element is found by reading current element), only one read is pending at a time even on out-of-order execution core. Hence P-chase list gives a guarantee on the order of execution of reads without using memory barriers (Nvidia GPU do not have global memory barriers).

Function *GpuSetPchase()* implements the P-chase list with elements lying on consecutive addresses and function *TraversePchase()* is used to traverse it, which is basically a linked list traversal. In line 12-25, we traverse the P-chase list multiple times, each time incrementing the end address but keeping the starting address fixed. As we traverse the P-chase list, all the elements will be cached (assuming LRU cache[5]) as they are accessed, including the starting address. At the end of each traversal, we measure the time taken to access the data at starting address, let say D. There are two possibilities:

1) Access time is smaller than or equal to the threshold.
   D was not evicted. We continue with the next iteration.
2) Access time is larger than the threshold.
   This indicates that D has been evicted from the cache. Since the previous traversal did not evict D and the only additional element accessed between previous and current traversal is the end address of current traversal, we conclude that the start and the end address of current traversal lie in the same cache set. We remove this end address from P-chase (line 21) so that it does not evict D in future P-chase traversals.

As in Section IV-B, we collect multiple pairs and then find the valid mapping (line 26-32). Figure 6(a) details the mapping function we found experimentally for GTX 1080.

Algorithm 2 Reverse engineering GPU Cache-line bits

```
 1: /* Code Executed by CPU */
 2: function PrintCachelineMappingFunctions
 3:     /* Find size of free memory on GPU */
 4:     $S \leftarrow GetGpuFreeMem()$;
 5:     /* Allocate contiguous physical */
 6:     $\langle VirtStart, PhyStart> \leftarrow AllocGpuPhyMem(S)$
 7:     $Pairs \leftarrow \emptyset$
 8:     /* Set up p-chase from start to end */
 9:     $GpuSetPchase(VirtStart, VirtStart + S)$
10:     $CurVaddr \leftarrow VirtStart$
11:     $PrevVaddr \leftarrow CurVaddr$
12:     /* Collect all pairs */
13:     for $Offset \leftarrow 1$ to $S$ do
14:         $CurVaddr \leftarrow Offset + VirtStart$
15:         $CurPaddr \leftarrow Offset + PhyStart$
16:         $TraversePchase(VirtStart, CurVaddr)$
17:         /* Tests if cached data at VirtStart is evicted */
18:         if $IsGpuCLEvicted(VirtStart)$ then
19:             $Pair.append(PhyStart, CurPaddr)$
20:             /* Remove CurVaddr from P-chase */
21:             $*PrevVaddr \leftarrow CurVaddr + 1$
22:         else
23:             $PrevVaddr \leftarrow CurVaddr$
24:         end if
25:     end for
26:     $F \leftarrow GenerateAllHashFunctions()$
27:     /* Test all permutations of mapping function */
28:     for each $func \in F$ do
29:         if $IsValidMapping(func, Pairs)$ then
30:             $Print(func)$ return
31:         end if
32:     end for
33: end function
34:
35: /* Code Executed by GPU */
36: /* Sets a pchase from $BaseAddr$ to $EndAddr$ */
37: function GpuSetPchase($BaseAddr$, $EndAddr$)
38:     $Addr \leftarrow BaseAddr$
39:     /* Each address points to next address*/
40:     while $Addr \neq EndAddr$ do
41:         $*Addr \leftarrow Addr + 1$
42:     end while
43:     $*Addr \leftarrow 0$
44: end function
45: /* Traverse pchase from $BaseAddr$ to $EndAddr$ */
46: function TraversePchase($BaseAddr$, $EndAddr$)
47:     $Addr \leftarrow BaseAddr$
48:     while $Addr \neq EndAddr$ do
49:         $Addr \leftarrow *Addr$
50:     end while
51: end function
52: /* Check if data in $BaseAddr$ is in cache */
53: function IsGpuCLEvicted($BaseAddr$)
54:     /* Measure time to read Base address */
55:     $StartTime \leftarrow GetClock()$
56:     $data \leftarrow BaseAddr$
57:     $AccessTime \leftarrow GetClock() - StartTime$
58:     if $AccessTime \geq Threshold$ then
59:         return $True$
60:     end if
61:     return $False$
62: end function
```

[5] Nvidia compiler can drop hints to L2 cache regarding eviction policy through specific instructions. We had to take precautions against this in our implementation as our implementation assumes cache is LRU.

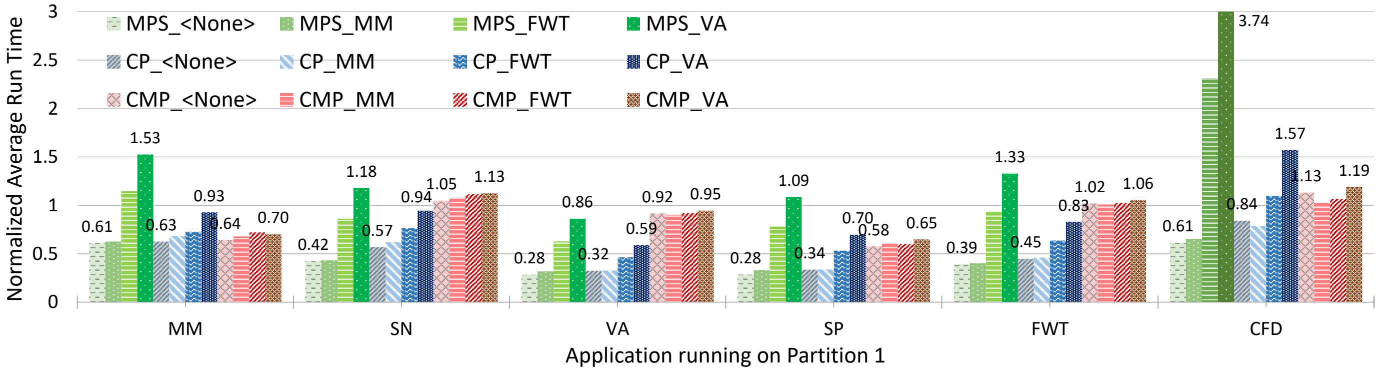


Fig. 11: Results of micro-benchmark study on V100 Tesla using 4 memory colors. Legend is of the form <Partitioning Method>_<Application running on Partition 2,3,4>. Normalized average runtimes are reported for kernels of application running on Partition 1. 1000 samples are collected for each application.

Appendix B
Miscellaneous GTX 1080 L2 cache and DRAM properties

Following are some properties of L2 cache that we found on GTX 1080:

1) The cache line size is 128 bytes.
   As seen in Figure 6(a), the least significant bit of the physical address that is used by the cache set hash function is bit 7. This suggests that the cache line is smaller than or equal to 2^7 = 128 bytes because all the physical addresses with only the 7 LSB bits differing will lie in the same cache set. During our experiments, we found that, when an eviction happens, contiguous 128 bytes are evicted. Hence, we conclude that a cache line is of 128 bytes.[6]
2) The set associativity is 16.
   As seen in Figure 6(a), since cache set index has 10 bits, the L2 cache has 2^{10} = 1024 sets. As the L2 cache size is 2 MB in GTX 1080 [4] and the cache line size is 128 bytes, Set associativity = Number of cache lines in a set = Cache Size / Cache line size / Number of sets = 16. We experimentally verified this relationship to be true. At minimum, 17 different cache lines are needed to be accessed to witness a conflict miss in the cache, suggesting that a cache set can hold a maximum of 16 cache lines.
3) The cache is LRU but it is asymmetric for reads and writes.
   Data that is cached on a write miss is less likely to be evicted than data that is cached on a read miss[7].
4) We did not find any evidence of pre-fetching.

And following are some properties of DRAM of GTX 1080:

1) Size of DRAM row is 2 KB.
   We base this on the fact that we found sets of addresses that lied on the same bank (based on hash function found for DRAM bank in Section IV-B) but did not cause row buffer conflict when any two addresses in these sets were accessed together. The maximum of these sets contained 2 KB addresses.
2) Each DRAM chip has bus of 32 bits.
   GTX 1080 specification states it has 256-bit memory interface width [4]. Since we have 8 memory modules in GTX 1080, this means that memory controller in each memory module is connected to DRAM via $256/8 = 32$ bit bus.

Appendix C
Tesla V100 Micro-benchmark results using four partitions

Figure 11 shows the results of running the same micro-benchmarks as in Section V-A on Tesla V100 but instead of using only 2 partitions, in this experiment the GPU is split into 4 partitions. (a) In MPS, we assign 25% of total threads to each partition, (b) in CP we assign 20 SMs to each partition (Tesla V100 has total of 80 SMs), and (c) in CMP we assign 20 SMs and a different memory color to each partition. The primary application runs on first partition and 3 replicas of an interfering task runs on all the other partitions. For example, when MM is ran with VA, MM runs on P_1 and VA runs on P_2, P_3, and P_4. Correspondingly, we normalize all the runtimes by the factor of $4 * T_{baseline}$ (since we expect the runtime on 25% partition of the GPU to increase by 4x) where baseline is the runtime of primary application's kernels when it runs alone on a GPU without any partitioning. The observations are similar to those reported in Section V-A. Table IV reports the $variation$. CMP performs slightly better as compared to the case of 2 partitions whereas CP and MPS perform worse.

GPU / Partitions / Technique	Average Variation	Max Variation
Tesla V100 / 4 / MPS	260.6%	510.0%
Tesla V100 / 4 / CP	79.0%	106.8%
Tesla V100 / 4 / CMP	7.3%	12.3%

TABLE IV: Variation in Runtime

[6] We also observed a peculiar behaviour. When data is populated into the L2 cache, only 32 bytes are fetched at a time. We suspect that four consecutive 32 bytes share the same tag in a cache line. This cache architecture is also different from a typical CPU L2 cache.

[7] We took precaution against this behaviour in our implementation of Algorithm 2 which assumes a typical LRU cache