UGPU: Dynamically Constructing Unbalanced GPUs for Enhanced Resource Efficiency

Xia Zhao
Defense Innovation Institute
Beijing, China
xiazhao@nudt.edu.cn

Guangda Zhang
Defense Innovation Institute
Beijing, China
zhanggd_nudt@hotmail.com

Lu Wang
Defense Innovation Institute
Beijing, China
734809187@qq.com

Huadong Dai
Defense Innovation Institute
Beijing, China
hddai@vip.163.com

Abstract

Different GPU generations have various numbers of SMs but still keep the balanced idea during the manufacture, i.e., the proportion of compute and memory resources within a single physical GPU is similar. Although GPU applications have different characteristics, it is still uncommon and uneconomic to build unbalanced physical GPUs for customers. With their powerful computational capabilities, GPUs are widely used in the cloud to accelerate diverse workloads from multiple users, creating opportunities to explore the unbalanced GPU concept in multitasking environments.

In this paper, we take the first step in exploring the feasibility and performance benefits of building unbalanced GPUs. Specifically, these unbalanced GPUs, referred to as GPU slices, are dynamically constructed with dedicated compute and memory resources from a single physical GPU to effectively address the diverse demands of co-executing applications, achieving high performance during the execution. However, there are two challenges that must to be solved. First, determining the size of unbalanced GPU slices during execution is challenging, as predicting GPU performance under varying resource allocations is inherently difficult. Second, reallocating memory resources after partitioning requires extensive data migration, with traditional methods leading to unacceptable performance degradation. To address the first challenge, UGPU employs a demand-aware resource partitioning algorithm that partitions resources dynamically without relying on a complex or inaccurate performance model. For the second challenge, UGPU introduces PageMove, a novel mechanism for efficient page migration between different memory dies within an HBM stack. Our key insight is that all memory channels already have physical connections to all through-silicon via (TSV) within a DRAM stack, while different bank groups can transfer data at the same time. PageMove slightly modifies DRAM architecture, uses a customized memory address mapping, designs a new parallel page migration mode (PPMM) and updates the virtual memory management scheme. By doing this, PageMove supports fast entire page migration from one memory die to another memory die which significantly reduces the data migration overhead during the memory resource reallocation. For the heterogeneous workloads with different characteristics, compared to the traditional balanced GPU design, UGPU increases the system performance by 34.3% on average while providing QoS support.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '25, Tokyo, Japan
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731103

Keywords

GPUs, HBM, Multitasking, Unbalanced, PageMove

ACM Reference Format:
Xia Zhao, Guangda Zhang, Lu Wang, and Huadong Dai. 2025. UGPU: Dynamically Constructing Unbalanced GPUs for Enhanced Resource Efficiency. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731103

1 Introduction

In recent year, GPUs feature an increased number of SMs to provide high compute power, e.g., from Fermi to A100, the SM count increases from 14 to 108 [1, 43]. At the same time, the memory bandwidth and LLC capacity also keep increasing to meet the bandwidth demand of the large number of SMs. We call such GPUs as balanced GPU designs since the proportion of compute resources and memory resources within a single physical GPU is maintained during the manufacture to exploit the resource utilization. However, it is well-known that compute-bound applications cannot fully utilize the memory bandwidth while the SMs of memory-bound applications are often stalled due to memory saturation. Even in this case, it is still uncommon and uneconomic to build unbalanced physical GPUs, i.e., a GPU with more SMs and fewer memory resources and a GPU with fewer SMs and more memory resources, since GPUs are bought and used in different scenarios by a wide range of customers.

GPUs, with their strong computational capabilities, are widely deployed as accelerators in cloud computing to meet the high-performance demands of various customers, as seen in platforms like Google Cloud, Amazon Cloud, and Alibaba Cloud [8, 9, 23]. In cloud environments, users can launch a variety of tasks, such as HPC applications and AI workloads, tailored to their specific needs. These applications may be memory-bound or compute-bound, depending on their functionality and implementation. The versatility of GPUs in accelerating diverse workloads from multiple users presents an opportunity to explore the concept of unbalanced GPUs in multitasking environments.

In this paper, we take the first step in exploring the feasibility and performance benefits of building unbalanced GPUs. Our approach involves dynamically constructing unbalanced GPU slices with dedicated and asymmetric allocations of compute and memory resources from a single physical GPU. This enables the system to effectively meet the diverse demands of co-executing applications while delivering high performance. Specifically, a single GPU is partitioned into virtualized GPU slices with unbalanced allocations of SMs and memory channels, optimized to cater to the unique requirements of different workloads. However, there are two challenges that must to be solved in order to get the benefits of the unbalanced GPU idea.

The first challenge is how to decide the size of different GPU slices, i.e., SMs and memory channels, that are assigned to concurrent applications in order to increase the single application performance and overall system throughput. Resource management is not a new topic in CPUs. Most previous work focuses on building a performance model to predict the application performance with different resource allocations and uses techniques such as hill-climbing to find a good resource partition [13, 18, 67]. Predicting the GPU performance with a particular resource allocation is not an easy task due to the high degree of overlap effects caused by hundreds of thousands threads executing on GPUs. Some previous work tries to predict the slowdown when multiple applications co-execute on GPUs but the prediction relies on profiling the architectural information such as the memory bandwidth utilization during the co-execution [25, 72, 73]. In other words, none of them can predict the performance of one particular resource allocation which has not been used yet.

In this paper, based on the key observation that the GPU performance can be maximized by allocating the resources to each GPU application based on its demand, we propose a demand-aware resource partition algorithm instead of building a complex and inaccurate GPU performance model. Starting with the current resource partition, the demand-aware scheme tries to move the SMs from the memory-bound applications to the compute-bound applications while doing the reallocation of memory channels in the opposite direction until the resource allocation become balanced. This actually increases the performance of all applications since the compute-bound applications get SMs that they require and lose memory channels which are under-utilized while the memory-bound applications can have more memory channels and lose some unnecessary SMs. The philosophy of demand-aware scheme matches what Lao Tzu said in his work, TaoTe Ching, "The way of Heaven takes from those in excess to help those in want." [61].

The second challenge lies in efficiently reallocating resources after the demand-aware scheme provides partition suggestions. While techniques such as context switching, draining, and hybrid schemes have been well-studied for reallocating SMs between co-executing applications [47, 59, 74], reallocating memory channels requires significant data migration, which can severely impact system performance and remains unexplored in GPUs. To address this, UGPU introduces PageMove, a novel mechanism for efficient data migration across memory dies within an HBM stack.

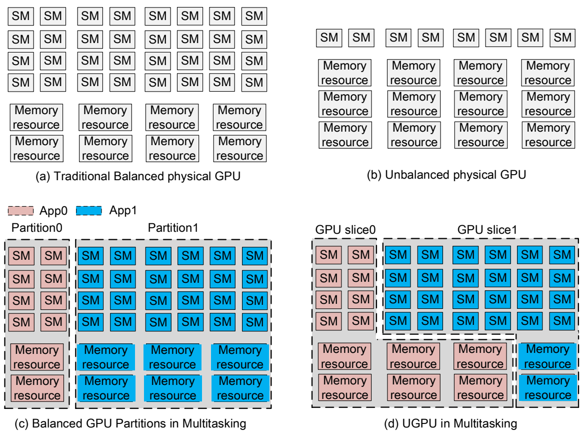

SM
Memory resource
(a) Traditional Balanced physical GPU
(b) Unbalanced physical GPU
App0
App1
Partition0
Partition1
GPU slice0
GPU slice1
(c) Balanced GPU Partitions in Multitasking
(d) UGPU in Multitasking

Figure 1: Traditional Balanced GPUs vs. Unbalanced GPUs.

Our key insight is that modern GPUs use the HBM where all memory channels already have physical connections to all TSVs within a DRAM stack while different bank groups can transfer data at the same time [16, 35, 36, 45]. In particular, PageMove adds simple crossbars inside of DRAM which enables the electric connection between any DRAM dies to any set of TSVs. This allows the row data of different bank groups to be quickly copied from the current channel to target channels. Instead of re-organizing the data across the whole DRAM hierarchy, PageMove uses a customized address mapping to make the data migration only happen between different channels within each DRAM stack. A parallel page migration mode with a new DRAM command called MIGRATION is proposed to enhance the DRAM working mechanism and enable efficient data transfer. Finally, PageMove is embedded in GPU virtual memory management to provide high performance and guarantee the data correctness even during the resource reallocation.

In summary, we make the following major contributions:

- Although building physically unbalanced GPUs is impractical, we demonstrate that dynamically constructing unbalanced GPUs in a multitasking environment, based on application demand, can significantly enhance resource efficiency and deliver high performance.
- UGPU uses a demand-aware resource partitioning algorithm to dynamically construct unbalanced GPU slices, adjusting compute and memory resources to match application demands and optimize performance.
- We propose PageMove to enable efficient data migration and memory reallocation in GPUs. It uses a simple crossbar for concurrent data migration, a customized address mapping to avoid reorganizing data across the DRAM hierarchy, and a parallel migration mode with a new DRAM command.
- The evaluation shows that UGPU increases the system performance by 34.3% on average compared to the traditional balanced GPU design while still providing the QoS support.

2 Background and Motivation

Traditional balanced GPU designs.

In traditional GPUs, as shown in Figure 1(a), SMs are connected to memory resources, such as LLC slices and memory controllers, through the NoC, commonly referred to as the Xbar in commercial

GPU documentation. The Xbar facilitates memory access by transferring requests from SMs to the memory system in one direction and data replies in the opposite direction. While different GPU generations feature varying numbers of SMs, they consistently adhere to a balanced design philosophy during manufacturing, maintaining a similar proportion of compute and memory resources within a single physical GPU. This approach ensures general-purpose applicability across diverse workloads. However, GPU applications often exhibit varied characteristics, making balanced physical GPUs suboptimal for certain scenarios. For example, in memory-bound applications where SMs frequently stall due to memory bandwidth saturation, the abundant SMs in traditional GPUs remain underutilized. In such cases, unbalanced GPUs with fewer SMs and more memory resources could better match application demands. Figure 1(b) illustrates this concept. Despite their potential benefits, manufacturing unbalanced physical GPUs is impractical and uneconomical, as GPUs are designed for diverse scenarios and a wide range of customer needs.

In cloud computing, diverse applications with varying characteristics can be assigned to different GPUs for acceleration, providing an opportunity to explore the feasibility and performance benefits of unbalanced GPUs. However, the traditional balanced GPU approach in multitasking scenarios, where multiple applications co-execute on the same physical GPU, still divides the GPU into smaller, balanced partitions, as illustrated in Figure 1(c). Each partition functions as a balanced virtual GPU, providing isolated resources such as SMs, network ports, LLC slices, and memory channels to each application. We refer to this straightforward approach as balanced partitioning (BP), which is similar to the multiple-instance GPU (MIG) feature provided by NVIDIA's A100 GPUs [1]. While maintaining the balanced GPU philosophy in multitasking scenarios is intuitive and straightforward, it significantly limits resource utilization and results in suboptimal system performance.

Towards Unbalanced GPU designs.

Consider two applications with distinct characteristics, e.g., a compute-bound application and a memory-bound application, co-executing on a GPU with 80 SMs and 32 memory channels. A balanced partition, i.e., MIG technique, allocates each application an equal share of resources, providing 40 SMs and 16 memory channels per application. Alternatively, the compute-bound application could be assigned a larger partition with 60 SMs and 24 memory channels, leaving 20 SMs and 8 memory channels for the memory-bound application. Conversely, the memory-bound application could receive the larger partition, with the compute-bound application taking the smaller one. However, none of these balanced allocation strategies fully exploit the characteristics of the applications. The compute-bound application underutilizes its memory bandwidth, while the memory-bound application experiences frequent SM stalls due to memory saturation. The fundamental issue lies in adhering to the balanced GPU design, which is logical for manufacturing physical GPUs but fails to align with the dynamic demands of multitasking environments.

In this paper, we propose UGPU, a novel approach that dynamically constructs unbalanced GPU slices with a flexible number of SMs and memory channels, as illustrated in Figure 1(d). UGPU enables the creation of GPU slices with dedicated and asymmetric allocations of compute and memory resources from a single physical GPU, effectively addressing the diverse demands of co-executing applications while delivering high performance. Specifically, UGPU partitions a single GPU into virtualized slices with unbalanced allocations of SMs and memory channels, optimized to cater to the unique requirements of different workloads. By moving beyond the traditional balanced GPU philosophy, UGPU provides a more flexible and efficient resource allocation strategy that adapts to the heterogeneity of applications. This design not only maximizes resource utilization but also enhances system throughput, making it well-suited for modern multitasking environments such as cloud computing. Next, we discuss the implementation details of UGPU.

3 Resource Partition Algorithm

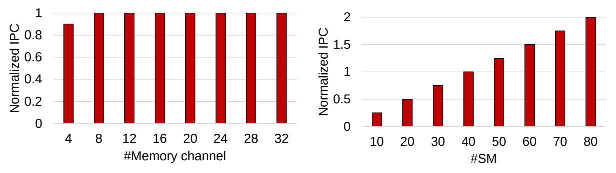
This section presents the demand-aware resource distribution algorithm, designed to enhance individual application performance and overall system throughput in UGPU.

3.1 Previous Work and New Observation

While constructing unbalanced GPU slices improves system performance and maintains resource isolation, the challenge lies in determining how to partition GPU resources among slices. Exhaustively exploring all possible resource partitions offline to find an optimal configuration is impractical and inefficient. In fact, resource management has been well explored in CPUs. Previous work focuses on building a performance model to predict the application performance with different resource allocation decisions and then uses a hill-climbing algorithm to search for an appropriate resource partition [13, 18, 67]. However, building a performance model to predict the GPU performance is not an easy task due to the high degree of overlap effects caused by hundreds of thousands threads executing on GPUs. We note some previous work explores how to predict the slowdown caused by the resource contention in multitasking GPUs. Mathematics models capturing architecture details, machine learning models or hybrid slowdown model are proposed [25, 72, 73]. However, these approaches rely on profiling runtime information during execution to predict the slowdown of the current resource partition. Consequently, they cannot predict the performance of a potential resource partition that has not yet been explored. In this paper, we propose a new resource partitioning algorithm leveraging GPU characteristics, avoiding the need for complex and potentially inaccurate performance models.

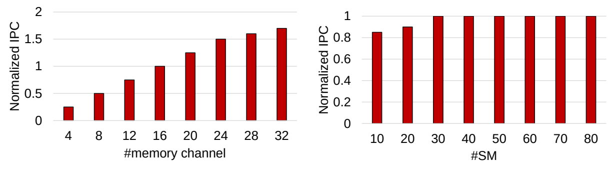
Figures 2 and 3 illustrate the performance of a compute-bound application and a memory-bound application, respectively, under varying SM and MC counts. These two applications are co-executed on a GPU with 80 SMs and 32 memory channels (detailed experimental setup in Section 5). Here, we use a typical compute-bound application (DXTC) and memory-bound application (PVC) as an example. The other workloads have similar trend. We first fix the SM count, i.e., 40 SMs, and evaluate the performance by varying the MC count. Next, we use 16 MCs and change the SM count. The results are normalized to the application performance by using half of the GPU resources, i.e., 40SMs with 16 memory channels.

As shown in Figure 2(a), start with 16 memory channels, increasing the MC count cannot increase the performance since the memory resources are already under-utilized for the compute-bound application. Meanwhile, decreasing the MC count first keeps the



(a) MC count changes (40 SMs)   (b) SM count changes (16 MCs)

Figure 2: Performance changes of a compute-bound application with different MC and SM count.



(a) MC count changes (40 SMs)   (b) SM count changes (16 MCs)

Figure 3: Performance changes of a memory-bound application with different MC and SM count.

performance unchanged but this eventually decreases the performance. It happens when the memory bandwidth demand of the 40 SMs cannot be satisfied by the few memory channels. Figure 2(b) shows the performance changes by varying the SM count with the fixed 16 MCs. It is clear the performance increases linearly since 16 MCs are enough to meet the bandwidth demand of the application even with 80 SMs. Key message to take away, for the compute-bound application which has few memory accesses, as long as its memory bandwidth demand can be satisfied, its performance keeps increasing by getting more SMs and can keep unchanged even if the MC count decreases.

Figure 3 shows the performance changes of a memory-bound application by varying the MC and SM count. With 40 SMs, as shown in Figure 3(a), the performance first increases linearly by increasing the MC count but eventually slowly since 40 SMs cannot fully utilize all the 32 memory channels. Meanwhile, with 16 MCs, the performance keeps unchanged by increasing the SM count from 40 to 80. However, the performance begins to decrease when the application can only use 20 SMs since the memory requests of these SMs cannot fully saturate the memory bandwidth. Key message to take away, for the memory-bound application which has massive memory accesses, as long as the SMs can fully utilize the memory bandwidth, its performance keeps increasing by getting more MCs and can keep unchanged even if the SM count decreases.

Figure 2 and Figure 3 show the performance changes of a single application. Next, we quantify the system performance of the heterogeneous workloads with different GPU slice sizes in UGPU. We use PVC and DXTC to construct a heterogeneous workload, i.e., *PVC_DXTC*, as an example. Figure 4 shows the system performance of the heterogeneous workload by varying the fraction of resources distributed to each application. Since UGPU provides resource isolation, i.e., the co-executing applications do not affect each other, the system performance changes match the performance trend of the singe application. Start with the even partition, i.e., each application has 40 SMs and 16 MCs, the system performance increases by decreasing the SMs and increasing MCs allocated to

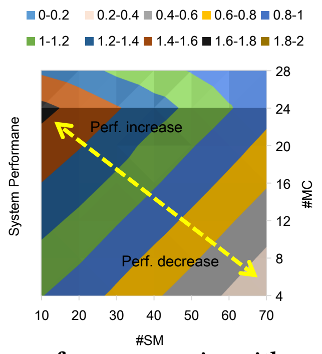


Figure 4: System performance varies with resource distribution among applications in a heterogeneous workload. The X- and Y-axes show the resource distribution for memory-bound app (compute-bound app receives the remaining resources).

```
#define N                    Total number of running apps
#define BW_SM$_X$              Bandwidth (BW) demand of an SM for app X
#define BW_MC                BW supplied by a memory channel (MC)
#define #SM$_X$ , #MC$_X$          SM/MC count allocated to application X
#define max_id(A, n)         return the largest value index in A[0:n]
#define delta                resource partition granularity
compute_list[0:N]            BW demand degree of compute-bound apps
memory_list[0:N]             BW demand degree of memory-bound apps

while (TRUE) {    //invoked at the profiling boundary                                   (a)
//calculate the degree of bandwidth demand for different applications
  for(i = 0, i < N; i++)
    if (BW_SM$_i$ * #SM$_i$ <= BW_MC * #MC$_i$) //compute-bound application
      compute_list[i] = (BW_MC * #MC$_i$)/(BW_SM$_i$ * #SM$_i$);
    else
      memory_list[i] = (BW_SM$_i$ * #SM$_i$)/(BW_MC * #MC$_i$);

  if(max_id(compute_list, N) != empty && max_id(memory_list, N) != empty)           (b)
    //re-allocate resources to re-balance the resource utilization
      Cid = max(compute_list, N); // id of the most compute-bound application
      Mid = max(memory_list, N); // id of the most memory-bound application
        #SM$_{Cid}$= #SM$_{Cid}$ + delta;
        #MC$_{Cid}$ = #MC$_{Cid}$ - delta;
        #SM$_{Mid}$= #SM$_{Mid}$ - delta;
        #MC$_{Mid}$ = #MC$_{Mid}$ + delta;

  else // no resources can be re-allocated                                            (c)
      break;
}
```

Figure 5: Demand-aware resource distribution algorithm. Shaded box (a) calculates the degree of bandwidth demand for all applications (b) chooses the most compute-bound application and increases its SMs while decreasing its MCs; chooses the most memory-bound application and increases its MCs while decreasing its SMs (c) ends the process when no resources can be re-allocated.

the memory-bound application while doing the opposite resource allocation for the compute-bound application. On the other hand, assigning more SMs and less MCs to the memory-bound application while allocating the remaining resources to the compute-bound application degrades the system performance since the performance of both applications decreases. Key messages to take away. Moving SMs from the memory-bound application to the compute-bound application while reallocating MCs in the opposite direction meets the resource demand of both application and brings performance improvement. Note that although we use two applications in the example, this insight is actually not limited to the application count.

3.2 Resource Distribution Algorithm

Building on the previous observations, the proposed demand-aware resource partitioning algorithm evaluates whether the current allocation of compute and memory resources meets application demands and recalculates allocations as needed.

Figure 5 presents the resource distribution algorithm which is used to calculate a new resource partitioning based on profiling. The algorithm consists of three parts. Part (a) (lines 10-15) first classifies the application into compute-bound and memory-bound based on the data bandwidth. If the data bandwidth demand is less than the bandwidth supply, the application is marked as a compute-bound application. Otherwise, the application is memory-bound. We record the degree of bandwidth demand for all applications in order to choose the most compute-bound application and the most memory-bound application. For the memory-bound and compute-bound applications, part (b) (lines 17-23) first assigns more SMs to the most compute-bound application while decreases its MC count. Next, the most memory-bound application gets more MCs and loses some SMs. In the end, part (c) (lines 24-25) ends the resource distribution algorithm when there are no SMs or MCs to be re-allocated, i.e., all compute-bound applications or memory-bound applications achieve balanced resource distribution.

The resource distribution algorithm quantifies the bandwidth demand of each SM and the available bandwidth of each memory channel, both of which vary across applications. The below equations calculate the data bandwidth demand of each SM and the data bandwidth supplied by each memory channel. In Equation 1, BW_{Per_SM} is the bandwidth demand of a single SM in the ideal case where the SM can issue instructions without any stalls. We first multiply the max number of instructions issued per cycle without any stalls IPC_{max} by the LLC accesses per kilo instructions $APKI_{LLC}$ to get the access count per cycle. Then, we multiply the access count, the cache line size $Size_{CacheLine}$ and the SM operating frequency $Freq_{SM}$ together to compute the data bandwidth demand, i.e., the bytes per second. IPC_{max}, $Size_{CacheLine}$ and $Freq_{SM}$ are determined by the GPU hardware, whereas $APKI_{LLC}$ is related to the application and the hardware.

Equation 2 computes the available LLC/DRAM bandwidth that a memory channel can supply. We compute the effective LLC hit bandwidth by multiplying the raw LLC bandwidth B_{LLC} with the LLC cache hit rate H_{LLC}. The effective data bandwidth provided by the memory is related to the LLC miss bandwidth and the maximum bandwidth that can be provided by the memory B_{MEM}. If the LLC miss bandwidth is less than the maximum memory bandwidth, these LLC misses can be totally hidden by the memory system. Otherwise, the miss bandwidth is determined by the maximum bandwidth of the memory system. All values in the above equations are hardware related and are determined ahead of time, except for $APKI_{LLC}$, H_{LLC} and B_{MEM} which are determined through profiling and can be collected using hardware performance counters.

$$BW\_SM = IPC_{max} \times APKI_{LLC} \times Size_{CacheLine} \times Freq_{SM} \quad (1)$$

$$BW\_MC = H_{LLC} \times B_{LLC} + Min\{(1 - H_{LLC}) \times B_{LLC}, B_{MEM}\}. \quad (2)$$

The proposed resource distribution algorithm does not explicitly consider memory capacity, as the datasets of the evaluated applications fit within the allocated memory, even when the number of memory channels is reduced. However, the algorithm can be easily extended to account for memory capacity constraints, ensuring support for workloads with larger memory requirements. An application is classified as compute-bound only when its bandwidth demand is below the available bandwidth, and its memory capacity demand is within the allocated resources. Conversely, if an application's working set exceeds its allocated memory, it is classified as memory-bound, as its performance can benefit significantly from additional memory channels to reduce page fault overhead.

The demand-aware resource distribution algorithm is not restricted by the number of co-locating applications. It iteratively allocates more SMs and fewer memory channels to compute-bound applications, while assigning fewer SMs and more memory channels to memory-bound applications, efficiently approximating optimal resource allocation at a low cost.

3.3 Hardware Cost Discussion

Both the online profilers and the demand-aware resource partition algorithm are implemented in hardware which are the major sources of the implementation cost. Starting with the balanced GPU partition, UGPU divides execution time into epochs and decides the GPU slice size at the end of each epoch. We add 16-bit counters to capture the LLC accesses, LLC hit accesses and memory bandwidth utilization within an epoch. The executed instruction count is recorded in the performance counter which already exists the SMs. The profiling logic uses performance counters to track execution events. These counters, widely used in CPUs and GPUs, increment by one for specific events, such as cache accesses. Since profiling operates off the critical path, it does not impact execution time. The epoch-based profiling mechanism is well-suited for complex cloud scenarios, where diverse GPU applications are launched at varying times. It effectively handles applications with both high-memory and high-compute kernels, provided the kernels run for a sufficient duration. For workloads with many short-execution kernels, the overall behavior within each epoch remains consistent, making resource reallocation unnecessary. In such cases, reallocation overhead could outweigh its benefits, potentially degrading overall performance.

The resource partition algorithm involves a number of arithmetic operations which are implemented in a fixed-function hardware unit for this purpose. It contains a single ALU to perform additions and comparisons in 1 cycle, multiplications in 3 cycles, and divisions in 25 cycles. Here, we assume the algorithm can support 4 applications. First, calculating the bandwidth demand of one application requires four multiplications, and one division. Note the bandwidth supply calculation requires less cycles which can be totally hidden. Next, part(a) requires four multiplications, one division and a compare operation for each application. Finally, part(b) requires six comparisons to get the id number and uses four additions to calculate the current MC and SM count. Overall, this procedure takes 148 cycles to finish the bandwidth demand and supply calculation and uses 162 cycles to do an iteration. If we enforce break if the iterations is larger than 20, the maximum latency of distribution algorithm is 3388 cycles — and we account for this latency in our evaluation. For more applications, since the resource distribution only happens once every epoch (e.g., 5M cycles), the latency can be totally hidden by beginning the procedure before the epoch ends. In this work, to address CPU-GPU communication latency, we implement additional hardware to execute the resource partitioning algorithm, enabling fast resource reallocation decisions without relying on the CPU. However, modern GPUs, such as NVIDIA's, already include MCUs like ARM or RISC-V cores for management.

This allows the resource partitioning algorithm to run on the MCU, enabling adaptability to algorithm updates over time.

After deciding the resources allocated to each GPU slice, UGPU needs to reallocate compute resources, i.e., SMs, and memory resources, i.e., memory channels, between co-executing applications. For the SM reallocation, we adaptively choose SM draining and SM switching which is similar to previous work [47, 59, 74]. The SM draining scheme waits for the TBs on an SM to finish execution before rescheduling the TBs of the newly allocated application to that SM. The SM switching scheme, on the other hand, stores the context information of the currently executing TBs and reschedules the TBs of the newly allocated application once the context switch is complete. We adopt the SM draining scheme if the TB completes within an epoch; otherwise, the SM switching scheme is used. Reallocating memory channels requires data re-organization and migration which can incur large overhead. Traditional memory management and data migration scheme even make UGPU perform worse than balanced GPU partition as shown in the evaluation. In the next section, we propose PageMove to minimize the memory resource reallocation overhead.

4 PageMove Design

This section first gives an example to illustrate memory resource reallocation via page migration and then introduces PageMove for fast migration across memory channels.

4.1 Memory Channel Reallocation Overview

In this subsection, we provide an example to illustrate memory page allocation before and after resource reallocation within an HBM stack consisting of 8 memory channels. Figure 6(a) depicts the initial memory distribution, where each application (App0 and App1) is assigned four memory channels. Each application has its own virtual memory space, with VPNs numbered from 0. The RPN determines the channel in which the physical memory page is located based on the memory address mapping policy, which will be discussed later. After memory resource reallocation, two memory channels (CH4 and CH5) are reassigned from App0 to App1, as shown in Figure 6(b). For simplicity, SM reallocation is not depicted in this example. To accommodate the change, App0 migrates memory pages corresponding to VPN0 and VPN1 to its remaining memory channels, freeing CH4 and CH5 for App1. Meanwhile, App1 migrates some memory pages from CH0 to CH3 into the newly allocated CH4 and CH5, utilizing their available memory bandwidth. The VPN-to-RPN mapping is updated accordingly in the TLBs and page table. Next, we introduce the PageMove design, which enables fast page migration across memory channels.

4.2 Re-architecting HBM

Modern GPUs exploit stacked HBM to provide high memory bandwidth, where GPU dies and HBM stacks are integrated on the same interposer as shown in Figure 7. GPU dies and HBM stacks are interconnected through high-speed links within the interposer. Each HBM stack consists of eight DRAM dies and a logic die which implements the HBM memory controller. The multiple ports of DRAM dies form eight memory channels. Within an HBM stack, these eight different dies or channels are assembled by using TSV technology [16, 35, 36, 45]. Since all DRAM dies are fabricated

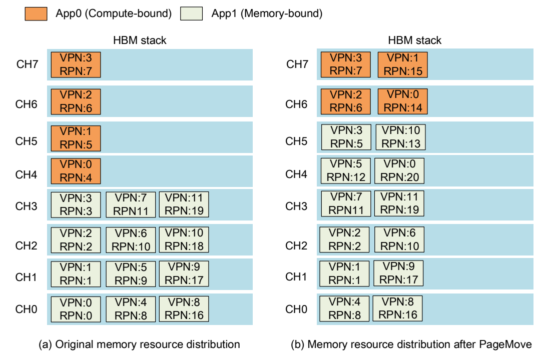


Figure 6: Memory page placement before and after memory resource reallocation. *After determining the resource distribution, memory pages are migrated to the corresponding memory channels to enable memory resource reallocation. Virtual page number (VPN) and real page number (RPN) mappings are updated accordingly.*

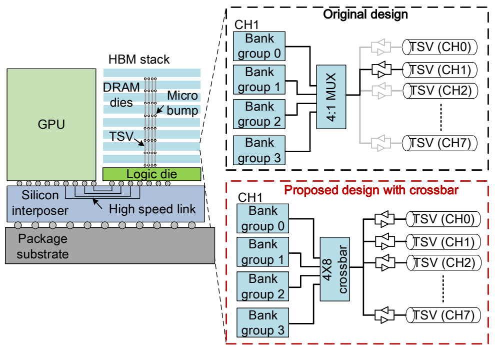


Figure 7: High bandwidth memory (HBM). *PageMove adds a crossbar to link all bank groups to all TSV sets.*

identically, TSVs are actually physically connected to all channels as shown in the figure. During the manufacturing process, a set of TSVs constituting a channel are electrically connected to one DRAM die by using tri-state buffers with the decoder logic. Within each DRAM die, there are multiple bank groups to improve the parallelism and each bank group has its own data bus. Each bank group further consists of several banks where all banks share the same data bus within the bank group. Accesses to different bank groups can be processed closely together.

Based on the insight that there already exists physical connections between TSVs and all DRAM dies while bank group structure supports serving concurrent memory requests, we next show how PageMove supports fast data migration between different memory channels within one HBM stack. The top of Figure 7 shows the original design where different bank groups of one memory channel are electrically connected to a set of TSVs constituting the channel through a 4×1 crossbar. This limits the data migration performance since only one memory request can be served at a time. To solve this problem, as shown in the bottom of Figure 7, within each memory channel, PageMove first adds a fully connected 4×8 crossbar to connect the 4 bank groups to any set of TSVs. In this case, a memory request from any memory channels can be sent to the target bank group by controlling tri-state buffers and the crossbar. Although bank groups of a memory channel still locate on the same die in

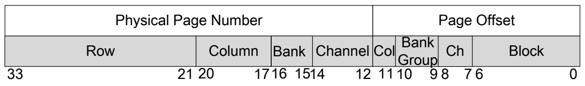


Figure 8: Customized memory address mapping.

PageMove design, they can transfer data to different memory channels through our newly added crossbar and the already existing TSVs at the same time. This enables the concurrent data migration from the four bank groups as shown later. To identify an idle channel, we implement detection logic on the logic die. This logic monitors the number of cycles each channel remains idle following a DRAM access command. The tri-state buffer decoder on the logic die is enhanced to manage the connections between the stack dies and the I/O TSVs, while the input/output connections of the 4×8 crossbar are configured during the data migration as discussed later. The crossbars incur minimal space and timing overheads, as the I/O TSVs of all channels associated with the same bit index are closely positioned. Our estimates, using DSENT [57] and assuming a 22nm chip technology, indicate that the total cost of the newly added crossbar and the enhanced tri-state buffer controller is less than 0.1% of a DRAM die [1].

4.3 Address Mapping and Page Migration

PageMove design enables the concurrent data migration between different memory channels within a HBM stack. To fully exploit PageMove, careful data placement and migration planning are essential. For example, migrating data from multiple bank groups to a single channel or across HBM stacks can degrade performance. To mitigate this, we propose a customized memory address mapping scheme. Figure 8 shows our customized address mapping where the channel, bank group, bank, row, column are indexed by different bits of the memory address. Here we use 4 HBM stacks with 8 memory channels per stack. Each memory channel consists of 4 bank groups and each bank group has 4 banks. We assume 4 KB memory page size which is widely used before. The idea also works with different page sizes as shown in the evaluation.

The low address bits are used to index the memory channels and bank groups which exploits the parallelism in the memory system while working with the PageMove design. In particular, we use address bits [7:8] to index the HBM stack id and use address bits [12:14] to index the memory channel within each HBM stack. We assign at least one memory channel per stack to each application. When the page fault exception happens which leads to a physical page allocation, PageMove can control which channel to use by managing the address bits [12:14]. For example, the address bits [12:14], ranging from '000' to '111', determine the accessible memory channels within each HBM stack, where '000' corresponds to channel 0 and '111' to channel 7. Conversely, setting the address bits [12:14] to '000' restricts the application to a single memory channel, i.e., channel 0 of each HBM stack. In this case, when a physical memory page needs to be migrated, PageMove only needs to do data migration between memory channels within each memory stack. This avoids the high overhead of moving data cross different stacks. Meanwhile, the address bits [10:9] are used to index the bank group id in order to exploit the concurrent data transmission ability provided by multiple bank groups. The GPU driver should have knowledge of which pages map to the same memory channel in DRAM to guarantee the balanced memory allocation in order to maximize the memory bandwidth. In alignment with prior research [52], PageMove utilizes the existing EEPROM present in modern DRAM modules to expose necessary information to the software layer. This EEPROM, which stores critical data about the DRAM chips, is accessed by the memory controller during system bootup. To convey the channel mapping information, only a few additional bytes are required. These bytes effectively communicate the bits of the physical address that correspond to the channel index, facilitating seamless integration and enhancing the overall memory management process.

We next propose the PPMM to facilitate rapid page migration in GPUs. Based on the customized memory address mapping, PPMM operates by directing the memory controller of each HBM stack to copy data from a source row in one bank of each bank group within a channel to the corresponding row and bank in the destination channel. Leveraging the customized memory address mapping, the row, column, and bank indices of the source and destination memory channels are determined by the physical pages located in those channels before and after memory resource reallocation. When page migration is initiated, the GPU driver provides the necessary identifiers, including the source and destination channels, as well as the row, column, and bank information for both channels, ensuring efficient and coordinated data transfer across the channels.

To copy data from a source row in one bank of a bank group to a destination row, PPMM first activates the corresponding rows in both banks. It then controls tri-state buffers with decoder logic and sets the 4×8 crossbars to connect the bank group to the idle TSVs, enabling data transfer. PPMM then puts the source bank in read mode and the destination bank in write mode, transferring data one cache line (128 bytes, i.e., a column of data) at a time. Similar to prior work [52], we introduce a new DRAM command called MIGRATION [2]. The MIGRATION command takes four parameters: 1) idle TSV index, 2) source/dest bank index, 3) source/dest row index, and 4) source/dest column index. It copies the cache line corresponding to the source column index in the activated row of the source bank to the cache line of the activated row in the destination bank with the specified bank index. To minimize changes to the existing DRAM interface, MIGRATION is designed as a two-cycle command. In the first cycle, the controller sends the idle TSV index and source/dest bank index; in the second cycle, it sends the source/dest row index and source/dest column index. One memory page migration incurs 32 MIGRATION commands which operate in different banks groups within the four HBM stacks. By re-architecting the HBM architecture and exploiting the parallel operation capabilities of different bank groups, four bank groups can do the MIGRATION command for a memory page in parallel, when the rows in the banks are activated and idle TSVs are available.

4.4 Virtual Memory Management

Next, we show how to update modern GPUs virtual memory management to enable memory resource reallocation.

[1]Previous work also explored the feasibility of adding a low-cost crossbar to link all bank groups to all TSVs [37, 46]. However, these efforts focus on load balancing, i.e., allowing unused memory channels to handle memory requests, for HBM or hybrid memory systems rather than accelerating page migration as UGPU does.

[2]Unlike RowClone, which facilitates data copying within a bank or between banks within a single memory channel, PPMM enables parallel data copying across multiple memory channels by exploiting the bank groups and utilizing idle TSVs, thereby enhancing memory channel reallocation performance.

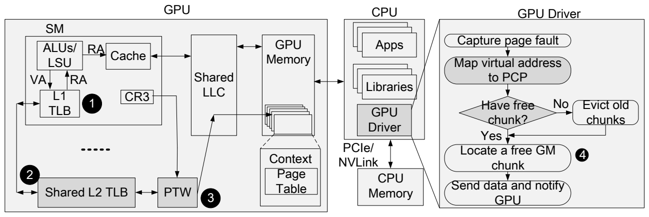


Figure 9: Holistic memory management in PageMove.

As shown in Figure 9, Modern GPUs exploit a virtual management system that is similar to CPUs [58, 76]. Multi-level translation lookaside buffer (TLB) hierarchy is used to avoid the page table walk overhead for every memory access. ❶ Each SM has one L1 TLB that is shared by all the load/store units on that SM. ❷ L1 TLB misses are sent to the L2 TLB that is shared by all SMs in the GPU. These accesses are either serviced by the L2 TLB or are handled by the page table walker (PTW) if the L2 TLB misses. Different applications use different memory page tables identified by CR3 registers to isolate the virtual address space from each other. ❸ For the memory access, if the PTW finds the physical memory page after searching the page tables, the physical page number is inserted into the L1/L2 TLB entry and used to calculate the physical memory address. ❹ Otherwise, a page fault exception incurs and a free physical memory page is allocated to the virtual memory page being accessed by the memory access. The page table is first updated accordingly to map the virtual page to the newly allocated physical page and then the L1/L2 TLBs will be filled with new table entry. Modern GPU drivers maintain a list of free physical memory pages for each application, categorized by the allocated memory channels and the already allocated memory pages. The GPU driver allocates one of them when the page fault exception happens.

At the start of execution, each application is assigned its allocated memory resources, such as specific memory channels. PageMove modifies the GPU virtual memory management system to support dynamic reallocation of these memory channels during execution. The updated components are highlighted in red in Figure 9. When memory resource reallocation occurs, PageMove first flushes the L1 TLBs of all SMs, causing all memory accesses to be routed to the shared L2 TLB for address translation. To guarantee the data is coherent after the page migration, PageMove also needs to flush in-flight instructions in the CU pipeline, in-flight transactions in the caches and the contents of the L1 and L2 caches when memory resource reallocation occurs. For applications with deallocated memory channels, the L2 TLB performs the address translation as usual. If the translation hits in the L2 TLB and the translated physical page number remains within the allocated memory channels, the physical address is returned to the L1 TLB, and a new entry is created and inserted immediately. However, if the translated physical page number falls outside the allocated memory channels, indicating reallocation to another application, PageMove invalidates the corresponding L2 TLB entry and uses the PTW to invalidate the corresponding entry in the page table. PageMove then sends a page fault request to the GPU driver for handling. The GPU driver, aware of the newly allocated or deallocated memory channels, tracks the page count allocated to each channel. It allocates a new free physical page within the still-allocated memory channels and initiates the migration of the memory page from the deallocated channel to another channel by sending the migration request to the memory controller. Once the page migration is complete, the page table, L2 TLB, and L1 TLB are updated as per the traditional design. If the address translation misses in the L2 TLB, the page table is searched. If a page table miss occurs, it is handled normally, with the GPU driver allocating a new physical memory page within the currently allocated memory channels. Otherwise, the physical page number is checked and handled similarly to the process described above.

For applications with newly allocated memory channels, when an L1 TLB miss occurs, the memory access proceeds to the L2 TLB. During L2 TLB access, if the translation hits and the page number recorded in the TLB entry is not in the newly allocated channels, PageMove invalidates the corresponding L2 TLB entry and uses the PTW to invalidate the page table entry. This action triggers a new type of page fault, prompting the GPU driver to allocate a new physical page from the newly allocated memory channels and release the previously allocated memory page back to the free physical memory page list. Concurrently, PageMove migrates the page from the old memory channel to the newly allocated memory channel to leverage the enhanced memory bandwidth. Once the migration is complete, the page fault handling finishes, and the SM resumes execution as usual. If the L2 TLB misses, PageMove invokes the PTW to perform memory address translation. In case of a page fault, the GPU driver handles it by allocating physical memory pages from the least used memory channels. Otherwise, if the page table hits, the physical page number is checked and handled in the manner previously discussed.

A register is used in the L2 TLB to track currently allocated or deallocated memory channels for each application. This register utilizes 2 bits to store the application ID, supporting up to 4 applications, and 1 bit to record the application status, indicating whether the application has more or fewer memory channels after resource reallocation. The remaining 8 bits are used to mark the channel allocation status. For applications with deallocated memory channels, a '1' indicates that the memory channel is still allocated to the application. Conversely, for applications with newly allocated memory channels, a '1' represents that the memory channel is newly allocated to the application. The register is configured based on the resource partition decision. For applications with newly allocated memory channels, the GPU driver sends a request to clear the channel list register in the L2 TLB once the memory page count across all current channels, including both original and newly allocated ones, becomes balanced.

4.5 Page Migration Overhead

The latency overhead of a page migration consists of the communication with the GPU driver and the data migration cost. The GPU driver processing delay includes several components: the initialization delay, the latency for the OS driver to read fault requests from the peripheral page request queue and preprocess them, and the processing delay, which encompasses the time to find a physical page and update the page table. Based on the observations of previous work [64], we assume the software delay to be 1000 cycles, presuming the OS driver is optimized to handle faults synchronously whenever possible. The data migration cost of one bank, i.e., the MIGRATION command latency, mainly consists of transferring the row data using the internal narrow I/O (i.e. 128-bit) bus. Considering the 128 bits/clock transfer rate and 256 byte (per

Table 1: Simulated GPU architecture.

No. SMs	80 SMs
SM resources	1.4 GHz, 32 SIMT width, 96 KB shared memory Max. 2048 threads (64 warps/SM, 32 threads/warp)
Scheduler	2 warp schedulers per SM, GTO policy
L1 data cache	48 KB per SM (6-way, 64 sets), 128 B block, 128 MSHR entries
L1 TLB	64 entries per SM (fully associative), LRU
LLC	6 MB in total (64 slices, 16-way, 48 sets), 120 clock cycles latency
L2 TLB	512 entries total (16-way associative), LRU
NoC	80 × 64 crossbar, 32-byte channel width
Memory stack configuration	440 MHz, 4 memory stacks, open page, FR-FCFS, 64 entries/queue, 8 channels/stack, 4 banks/group, 4 bank groups/channel, 128 I/Os, 900 GB/s
Page Table Walker	Up to 64 concurrent threads, 4-level page tables
HBM Timing [16, 34]	tRC=47, tRCD=14, tRP=14, tCL=14, tWL=2, tRAS=33,tRRDl=6, tRRDs=4, tFAW=20 tRTP=4, tCCDl=2, tCCDs=1, tWTRl=8, tWTRs=3

bank group) then 16 cycles are required to migrate the data of a memory page belonging to one bank to the corresponding bank of the destination memory channel in the ideal case. Considering the delay of closing the current row, activating a new row, and the delay of other operations, the MIGRATION command latency should be less than 50 cycles. Since the GPU clock cycle is 1.25 times slower than memory data transferring clock cycle, we believe 40 GPU clock cycles is a conservative estimation of the MIGRATION latency in GPUs with PageMove. As previously mentioned, the MIGRATION command can operate in parallel across different HBM stacks and bank groups, depending on the status of the HBM stack, such as the availability of idle TSVs and the activation of rows within a bank group for traditional READ/WRITE commands. The MIGRATION command executes without interrupting traditional commands and likewise cannot be interrupted by other commands during its operation. This ensures seamless data migration and efficient utilization of memory resources.

5 Methodology

Simulated System: We modified GPGPU-sim v3.2.2 [12] to support multitasking and integrate GPGPU-sim with a detailed memory simulator, Ramulator [34], to simulate an HBM-based memory subsystem. The SM context switching and traditional page migration incur reading and writing the data into memory which is faithfully modeled. In particular, the data read and data write overhead are modeled in the cycle-accurate Ramulator and data transmission overhead is modeled in Booksim. We model a GPU with 80 SMs and 4 HBM stacks (8 channels per stack), i.e., 32 memory channels in total. We set two 96 KB LLC slices per channel and use a crossbar as the interconnection network. The unified memory behavior is also modeled in GPGPU-sim with added TLBs and a GPU memory management unit. We employ the non-blocking TLB implementation where each SM has a private L1 TLB and different SMs share a L2 TLB [10, 11, 53, 54, 70]. On a TLB miss, the page table walker traverses a 4-level page table that supports up to 64 concurrent threads. A page fault happens if the page is not in the GPU memory which incurs an optimistic 20μs page fault latency similar to previous work [40, 76]. 4KB memory page size is assumed as the baseline [50, 51, 54, 64, 76] and different sizes are evaluated as the sensitivity analysis. UGPU is not limited to GPUs with UVM support. PageMove operates as long as the GPU has virtual memory support, meaning it manages physical memory in memory pages, a feature that has been available even in earlier GPU generations.

Table 2: GPU-compute benchmarks considered in this study.

Benchmark	Abbr.	MPKI	#Knls	#Mem footprint
Page View Count [24]	PVC	4.79	1	3810 MB
Lattice-Boltzmann Method [56]	LBM	6.09	3	389 MB
BlackScholes [2]	BH	1.54	14	48 MB
DWT2D [17]	DWT2D	2.72	1	301 MB
EULER3D [17]	EULER3D	4.39	7	286 MB
FastWalshTransform [2]	FWT	2.23	4	269 MB
Lavamd [17]	LAVAMD	10.45	1	123 MB
Streamcluster [17]	SC	3.42	2	302 MB
Convolution Separable [2]	CONVS	1.14	4	151 MB
Srad_v2 [17]	SRAD	1.09	1	1048 MB
DXTC [2]	DXTC	0.0004	2	20 MB
HOTSPOT [17]	HOTSPOT	0.08	1	130 MB
PATHFINDER [17]	PF	0.06	5	792 MB
Coulombic Potential [56]	CP	0.02	1	40 MB
MRI-Q [56]	MRI-Q	0.01	3	50 MB

Details about the simulated GPU architecture are provided in Table 1. We use GPUWattch to estimate GPU power assuming a 22 nm technology node [38] and update the HBM power model based on previous work [16].

Workloads: To cover different domains, we use a wide range of GPU benchmarks selected from Rodinia [17], Parboil [56], CUDA SDK [2], and Mars [24], which are listed in Table 2. The benchmarks are categorized as memory-bound or compute-bound based on GPU memory bandwidth demand. We construct 105 multi-program workloads, including 50 heterogeneous and 55 homogeneous mixes. Each workload is simulated for 25 million cycles to get stable and representative results, with benchmarks re-launched if they finish early. Performance results are reported from the first run of each benchmark. In our evaluation, we do not include memory-oversubscribed workloads. However, if such workloads were present, they would be classified as memory-bound applications, and additional memory channels would be allocated to reduce page faults and swapping overhead, thus improving performance.

Metrics: We use two multi-program metrics: system throughput (STP) and average normalized turnaround time (ANTT) [22]. STP quantifies overall system performance (higher-is-better). ANTT focuses on per-application performance and quantifies average per-application execution time (lower-is-better). The following equations calculate STP and ANTT metrics — IPC_i^{alone} denotes IPC for benchmark i running alone on the GPU, whereas IPC_i refers to the IPC achieved for benchmark i during multitasking execution.

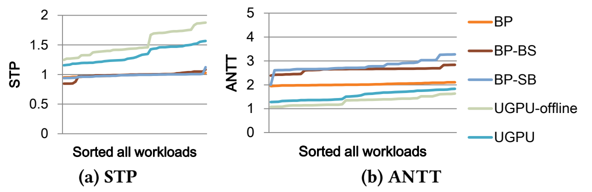
$$STP = \sum_{i=1}^{n} \frac{IPC_i}{IPC_i^{alone}} \tag{3}$$

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{IPC_i^{alone}}{IPC_i} \tag{4}$$

6 Evaluation

This section first compares the performance of following designs:

- **BP:** The balanced partition which divides a GPU into two equal-sized GPU partitions as described in Section 2.
- **BP-BS:** The GPU is divided into a **b**ig GPU partition with 60 SMs and 24 memory channels and a **s**mall GPU partition with 20 SMs and 8 memory channels.
- **BP-SB:** The GPU is divided into a **s**mall GPU partition with 20 SMs and 8 memory channels and a **b**ig GPU partition with 60 SMs and 24 memory channels.
- **UGPU:** UGPU dynamically decides the size of unbalanced GPU slices and reallocates resources during the execution.



(a) STP (b) ANTT

Figure 10: Performance across heterogeneous workloads.

- **UGPU-offline:** The offline mode of U-GPU which decides the size of each GPU slice based on offline profiling. This is used to quantify the resource reallocation overhead.

We next evaluate the performance impact of PageMove design in UGPU. We consider following schemes:

- **UGPU-Ori:** The UGPU with traditional page migration.
- **UGPU:** UGPU with the PageMove design.

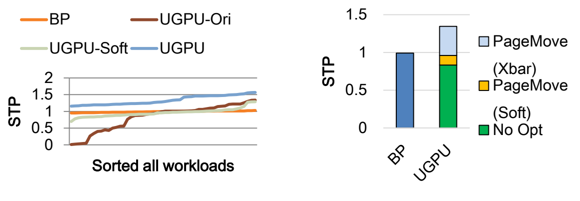
6.1 UGPU performance

Figure 10 shows the performance results of the UGPU and BP with different partition sizes. The x-axis represents the sorted multi-program workloads based on the STP of each workload. This representation provides an overview of the performance comparison across a large number of multi-program workloads and is commonly used in previous studies [3, 62, 73, 74]. It is clear that simply increasing or decreasing the partition size assigned to the GPU application does not help the system performance as shown in Figure 10(a), i.e., BP, BP-BS and BP-SB have similar performance. In contrast to the baseline equally-sized GPU partition, BP-BS and BP-SB assign smaller partitions with fewer compute and memory resources to one application, significantly reducing its performance and increasing ANTT, as shown in Figure 10(b).

Compared to BP, UGPU improves system performance by an average of 34.3%, reaching up to 56.7% for heterogeneous workloads. UGPU achieves this by assigning a GPU slice with many SMs and few memory channels to compute-bound applications, while allocating the remaining resources, i.e., fewer SMs and more memory channels, to memory-bound applications, effectively meeting the demands of both. This performance enhancement also enables UGPU to outperform BP in ANTT by 46.7% on average. Although dynamic resource reallocation during execution typically incurs significant overhead, UGPU's PageMove design minimizes this impact, reducing STP and ANTT by only 12.1% and 13.6%, respectively, compared to UGPU-offline. Dynamic partitioning is exploited as cloud providers offer GPUs to a diverse range of customers, and non-uniform GPU slicing can significantly improve resource utilization. Given the dynamic nature of cloud workloads, offline profiling is often impractical. Here, we use offline UGPU as the ideal design and demonstrate that online profiling introduces minimal overhead while still providing substantial performance benefits.

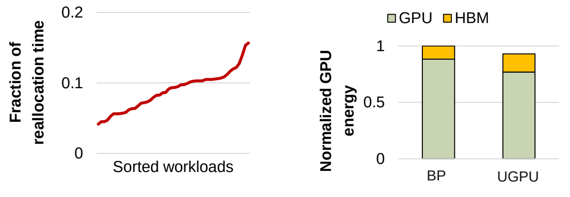
6.2 PageMove Analysis

We now quantify the performance impact of the proposed PageMove which uses a customized memory address mapping, adds a crossbar within each HBM stack and exploits the new PPMM mode to reduce the data migration cost. Figure 11 presents the STP comparison of BP, UGPU-Ori (UGPU without optimizations),



(a) STP (b) Performance breakdown

Figure 11: The performance benefit breakdown of PageMove.



(a) Resource reallocation cost (b) Energy breakdown

Figure 12: Resource reallocation discussion.

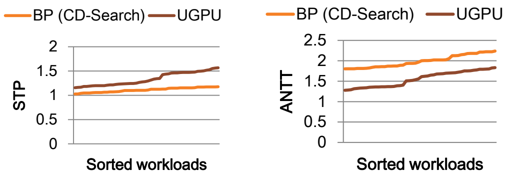
UGPU-Soft (UGPU without hardware modifications), and UGPU. Compared to BP, UGPU-Ori does not use the PageMove design which incurs a massive amount of data migration and decreases the STP seriously for many workloads. On average, UGPU-Ori even decreases the STP by 16.8% compared to the BP. With customized memory address mapping and updated virtual memory management, UGPU-Soft improves performance by 12.7% compared to UGPU-Ori. By further incorporating a crossbar and leveraging data transmission parallelism, PageMove significantly enhances performance, enabling UGPU to outperform BP by an average of 34.3%.

6.3 Resource Reallocation Discussion

Figure 12a shows the fraction of time spent per epoch for both SM migration and data migration. Applications still keep executing when the SMs and memory resource reallocation happens. For workloads with relatively stable behavior, resource reallocation may not occur during certain epochs, resulting in zero migration overhead for those periods. Conversely, for workloads composed of multiple kernels with varying behaviors, repartitioning often occurs between epochs, leading to varying migration overheads depending on the application characteristics. On average, the resource reallocation occupies 8.9 % of an epoch time on average and the fraction is 19.5% in the worst case. This attributes to our proposed PageMove design which enables fast data migration.

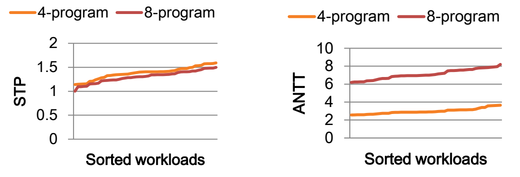
As shown in Figure 12b, the baseline GPU with 80 SMs and the HBM system (900 GB/s) occupies 88.3% and 11.6% of the system energy, respectively, for all the heterogeneous workloads on average [3]. Compared to BP, UGPU incurs data migration during the memory resource reallocation which increases the energy consumption of the memory system by 38% on average. However, memory system only occupies a limited proportion of total GPU system energy which is reported by previous work and also confirmed in our evaluation [31]. UGPU significantly increases the GPU performance which decreases the static and constant energy and brings 13%

[3]We observe the HBM system can occupy up to 30.3% of the system energy for the heterogeneous workload with heavy memory accesses.



(a) STP (b) ANTT

Figure 13: STP and ANTT comparison with previous work.



(a) STP (b) ANTT

Figure 14: STP and ANTT for 4- and 8-program workloads.

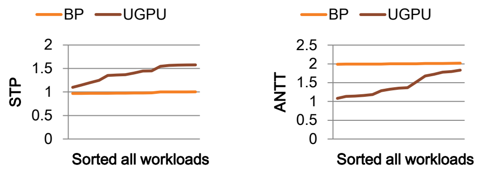
GPU energy reduction. On average, UGPU decreases the energy consumption of the whole GPU system by 7.1%.

6.4 Previous Work Discussion

We compare UGPU to CD-Search, which dynamically reallocates SMs among co-executing applications in multitasking GPUs [74]. In CD-Search, compute and memory resources are shared among applications, lacking resource isolation and QoS support. CD-Search cannot directly operate in BP, as BP ensures applications only use isolated resources within GPU instances to avoid contention. To address this, we combine CD-Search with BP, enabling SM reallocation across GPU instances while maintaining resource isolation. As shown in Figure 13, BP (CD-Search) improves STP by 11.2% over BP by reallocating SMs between GPU instances. UGPU outperforms BP (CD-Search) in STP and ANTT by 22.4% and 43.6%, respectively, by dynamically reallocating both SMs and memory channels. Simultaneously reallocating compute and memory resources increases the complexity of the partitioning algorithm and introduces challenges in minimizing data migration overhead. UGPU addresses these issues by performing resource reallocation based on STP feedback at the end of each epoch and leveraging PageMove to reduce data migration overhead. As the program count increases to four per workload, UGPU's performance advantage over BP (CD-Search) rises further to 25.4% in STP and 56.1% in ANTT, as the increased number of memory-bound and compute-bound applications provides more room for resource reallocation. Detailed results are omitted due to space constraints.

6.5 Multi-program Workload Mixes

UGPU is not limited to the two application workloads. Next, we evaluate the UGPU in the four- and eight-program workload mixes. In this case, at the profiling boundary, the demand-aware resource distribution algorithm iteratively calculates and chooses the most memory-bound application and the most compute-bound application among the four applications and then decides the new resource partition. The iteration ends when no resources can be re-allocated. After that, UGPU migrates SM and memory channels based on the new resource partition. As shown in Figure 14, on average, UGPU improves STP by 38.3% and ANTT by 101.8% for four-program workloads. Compared to the two-program workloads, the STP and ANTT improvement is a higher for the four-program workloads because UGPU can better utilize the system resources with more memory-bound and compute-bound applications. For eight-program workloads, we evaluate performance using 200 randomly selected workloads, each consisting of four compute-bound and four memory-bound applications. On average, UGPU improves STP by 30.3% and ANTT by 89.3%. The performance gains slightly decrease when the application count increases to eight, as the current GPU architecture allocates fewer resources per application, reducing reallocation space and limiting overall improvement.



(a) STP (b) ANTT

Figure 15: STP and ANTT for AI workloads.

6.6 AI Workloads

Next, we evaluate the UGPU performance of AI workloads such as AlexNet, ResNet, SqueezeNet, gated recurrent unit (GRU) and long short time memory (LSTM) [32] combing with the compute-bound benchmarks listed in Table 2. Compared to the traditional balanced partition, UGPU dynamically constructs unbalanced GPU slices to better align with the memory and compute demands of AI workloads during execution. By allocating more memory channels to memory-bound tasks and more SMs to compute-bound tasks, UGPU significantly enhances resource utilization and improves the STP 39.4% and ANTT by 57.6% on average as shown in Figure 15.

As the size of the models being trained increases, multiple GPUs are necessary for deep learning and LLMs. UGPU can be utilized in multi-GPU systems to partition each GPU into unbalanced slices, improving resource utilization. For instance, during LLM training, the kernel executing on the GPU may be either memory-bound or compute-bound, leaving either the compute or memory resources under-utilized. These idle resources can then be allocated to other tasks launched by different users, thus enhancing the utilization of cloud GPU clusters.

6.7 MPS and QoS Support

Multi-Process Service (MPS) enables multiple applications execute on different SMs but the memory resources are still shared between co-executing applications which can easily causes memory contention that can violate the QoS in some cases. On the other hand, BP and UGPU provide compute and memory resource isolation between co-executing applications by assigning each application an independent GPU slice. Next, we compare the QoS support and performance of MPS, BP and UGPU. We denote the compute-bound application as the high-priority application in the heterogeneous mixes and set a QoS target at 0.75 NP for the high-priority application. Figure 16 quantifies QoS and STP of MPS, BP, UGPU

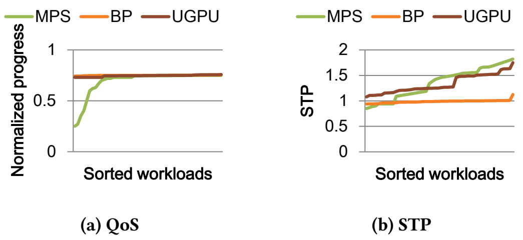


Figure 16: QoS support in MPS, BP and UGPU.

across heterogeneous workloads. MPS exploits offline analysis to identify the high-priority application and assigns more SMs, i.e., 60 SMs, to the application to meet the QoS target. QoS-aware BP executes the high-priority application within a GPU partition with enough SMs and memory channels, i.e., 60 SMs and 24 memory channels, and the compute and memory resources not allocated to the high-priority application are used to create another GPU partition which is allocated to the other application to improve STP. UGPU dynamically constructs unbalanced GPU slices to meet the QoS target while maximizing the resource utilization during the execution. As shown in the figure, BP and UGPU both meet the QoS target for all workloads since the GPU partition provides the high-priority application with isolated resources without any contention with the co-executing application. However, MPS breaks the QoS target for some workloads since the high-priority application performance degrades due to the memory contention of the co-executing application. Compared to the BP, UGPU improves STP by 33.7% on average through providing the memory channels to the low-priority application since the high-priority application can still meet its QoS target with the reduced memory bandwidth. For MPS, it is interesting to note that its STP outperforms UGPU for some workloads due to the higher memory resource utilization by sharing memory resources across all co-exeucitng applications. We want to emphasize that this work is not about arguing which technique is the best choice. Instead, by proposing UGPU, cloud providers can choose UGPU or MPS in different scenarios based on the demand. For example, UGPU can be used when resource isolation is required, and MPS can be considered when resource sharing or contention is allowed to different degrees.

7 Related Work

To our best knowledge, this is the first work exploring the unbalanced GPU design. We show the unbalanced idea which seems crazy in manufacturing physical GPUs works in the multitasking environment. Next, we discuss the related work.

Multitasking GPU optimization: MPS and Simultaneously Multitasking (SMK) are two previously proposed approaches to support multitasking GPUs [44, 68, 69]. Managing SMs among concurrent applications in MPS received significant attention recently [4, 6, 7, 25, 48, 71, 74, 75]. Besides partitioning the SMs, Jog et al. [28, 29] propose an application-aware memory scheduler to manage the memory bandwidth in multitasking GPUs. Wang et al.[65] manage shared cache and memory bandwidth by moderating warp allocation per application within an SM. Other works focus on slowdown prediction in multitasking GPUs, such as DASE[25], which builds an architectural model, and Themis [72], which uses DNNs to learn performance impacts. HSM [73] combines these approaches to reduce model cost while maintaining high accuracy. Existing prediction-based methods, such as HSM, estimate slowdown caused by shared resource contention, particularly when multiple applications access the same memory channel, to ensure fairness. BP avoids this issue due to its resource isolation but still suffers from resource under-utilization within each GPU partition, a challenge that UGPU effectively addresses. Beyond MPS, SMK co-executes applications on one SM. Wang et al.[66] improve throughput and fairness through TB dispatch and warp scheduling optimizations, while Warped-Slicer[69] partitions intra-SM resources across kernels. To reduce contention, Li et al.[41] adjusts TLP and bypasses caches, and Dai et al.[20] limits memory access counts to balance usage. However, none of the previous work has recognized the benefits or explored the feasibility of the unbalanced GPU concept.

Memory Page Allocation: Memory page allocation has been explored in CPUs and GPUs before. RowClone is well-known to accelerate bulk data copy and initialization [52]. However, there is no existing data path between different memory channels within an HBM stack, which PageMove introduces based on our observations. Additionally, RowClone still faces the data reorganization challenge after memory channel reallocation, whereas PageMove addresses this issue through a carefully designed address mapping policy and virtual memory management scheme. In NUMA multiprocessor systems, memory pages are allocated in the local memory near the compute node to reduce access latency [15, 63]. Phadke et al. [49] propose a heterogeneous main memory system with three different memory modules and then place the page in a suitable memory module based on the application requirement. In GPUs with heterogeneous memory systems, Agarwal et al. [5] demonstrate that bandwidth-aware page placement outperforms CPU-oriented policies, as GPUs prioritize memory bandwidth. However, none of them addresses effective memory channel reallocation like UGPU.

CPU Resource Management: With the advent of CMPs, numerous research went into resource management in CPUs. Virtual private machines, job co-scheduling policies, memory bandwidth management, cache sharing are all well explored in multicore CPUs [26, 27, 30, 33, 42, 55, 77]. Meanwhile, many works also focus on the SMT processors, such as dynamic controlled resource allocation, learning-based resource distribution, different instruction fetching policies and L1 cache management [14, 19, 21, 39, 60]. Techniques tailored for CPUs cannot be directly used in multi-tasking GPUs to provide resource isolation, search resource partition and minimize the resource reallocation overhead.

8 Conclusion

While the balanced design is commonly adopted in compute systems, this paper introduces UGPU to dynamically construct unbalanced GPU slices to increase resource efficiency and achieve high performance. UGPU partitions a GPU into multiple unbalanced slices, each with dedicated SMs and memory channels tailored to application demands. UGPU determines GPU slice sizes using a demand-aware resource partition algorithm. To minimize resource reallocation overhead, UGPU leverages PageMove, a software-hardware co-designed mechanism for fast page migration between memory channels. PageMove integrates simple crossbars within

HBM stacks to enable parallel data transmission, employs a customized memory address mapping, and introduces a parallel page migration mode with a new DRAM command to further reduce migration costs. Compared to the traditional balanced GPU partition, UGPU increases the system performance of heterogeneous workloads by 34.3% on average while still providing QoS support.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is supported by the Young Elite Scientists Sponsorship Program and the Beijing Nova Program (No.20220484150).

References

[1] [n. d.]. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[2] [n. d.]. NVIDIA CUDA SDK Code Samples. https://developer.nvidia.com/cuda-downloads.

[3] Almutaz Adileh, Stijn Eyerman, Aamer Jaleel, and Lieven Eeckhout. 2016. Maximizing Heterogeneous Processor Performance Under Power Constraints. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 3, Article 29 (Sept. 2016), 23 pages.

[4] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. 2012. The Case for GPGPU Spatial Multitasking. In *Proceedings of the International Symposium on High-Performance Comp Architecture (HPCA)*. 1–12.

[5] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 607–618.

[6] P. Aguilera, K. Morrow, and N. S. Kim. 2014. Fair Share: Allocation of GPU Resources for Both Performance and Fairness . In *Proceedings of the International Conference on Computer Design (ICCD)*. 440–447.

[7] P. Aguilera, K. Morrow, and N. S. Kim. 2014. QoS-Aware Dynamic Resource Allocation for Spatial-Multitasking GPUs. In *Proceedings of the International Conference on Asia and South Pacific Design Automation Conference (ASP-DAC)*. 726–731.

[8] Alibaba. [n. d.]. Alibaba Cloud. https://www.alibabacloud.com/.

[9] Amazon. [n. d.]. Amazon web services. https://aws.amazon.com/cn/ec2/.

[10] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 136–150.

[11] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 503–518.

[12] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceeding of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174.

[13] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 318–329. https://doi.org/10.1109/MICRO.2008.4771801

[14] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. 2004. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 171–182.

[15] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1994. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 12–24.

[16] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. 2017. Architecting an Energy-Efficient DRAM System for GPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*. 73–84.

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 44–54.

[18] Jian Chen and Lizy Kurian John. 2011. Predictive Coordination of Multiple On-Chip Resources for Chip Multiprocessors. In *Proceedings of the International Conference on Supercomputing (ICS)*. 192–201.

[19] Seungryul Choi and D. Yeung. 2006. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 239–251.

[20] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou. 2018. Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 208–220.

[21] A. El-Moursy and D.H. Albonesi. 2003. Front-end Policies for Improved Issue Efficiency in SMT Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 31–40.

[22] Stijn Eyerman and Lieven Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE micro* 28, 3 (May 2008), 42–53.

[23] Google. [n. d.]. Graphics Processing Unit (GPU) | Google Cloud. https://cloud.google.com/gpu/.

[24] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 260–269.

[25] Q. Hu, J. Shu, J. Fan, and Y. Lu. 2016. Run-Time Performance Estimation and Fairness-Oriented Scheduling Policy for Concurrent GPGPU Applications. In *International Conference on Parallel Processing (ICPP)*. 57–66.

[26] Yunlian Jiang, Xipeng Shen, Chen Jie, and Rahul Tripathi. 2008. Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 220–229.

[27] Yunlian Jiang, Kai Tian, and Xipeng Shen. 2010. Combining Locality Analysis with Online Proactive Job Co-Scheduling in Chip Multiprocessors. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*. 201–215.

[28] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2014. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU)*. 1:1–1:8.

[29] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS)*. 223–234.

[30] Mahmut Kandemir, Sai Prashanth Muralidhara, Sri Hari Krishna Narayanan, Yuanrui Zhang, and Ozcan Ozturk. 2009. Optimizing Shared Cache Behavior of Chip Multiprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 505–516. https://doi.org/10.1145/1669112.1669176

[31] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 738–753.

[32] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Tango: A Deep Neural Network Benchmark Suite for Various Accelerators. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 137–138. https://doi.org/10.1109/ISPASS.2019.00021

[33] Dimitris Kaseridis, Jeffrey Stuecheli, Jian Chen, and Lizy K. John. 2010. A Bandwidth-aware Memory-subsystem Resource Management Using Non-invasive Resource Profilers for Large CMP Systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. 1–11.

[34] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (January 2016), 45–49.

[35] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, Jang Hwan Cho, Ki Hun Kwon, Min Jeong Kim, Jaejin Lee, Kun Woo Park, Byongtae Chung, and Sungjoo Hong. 2014. 25.2 A 1.2V 8Gb 8-channel 128GB/s High-bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods using 29nm Process and TSV. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 432–433.

[36] Jong Chern Lee, Jihwan Kim, Kyung Whan Kim, Young Jun Ku, Dae Suk Kim, Chunseok Jeong, Tae Sik Yun, Hongjung Kim, Ho Sung Cho, Sangmuk Oh, Hyun Sung Lee, Ki Hun Kwon, Dong Beom Lee, Young Jae Choi, Jaejin Lee, Hyeon Gon Kim, Jun Hyun Chun, Jonghoon Oh, and Seok Hee Lee. 2016. High Bandwidth Memory (HBM) with TSV Technique. In *Proceedings of the International SoC Design Conference (ISOCC)*. 181–182.

[37] Sukhan Lee, Kiwon Lee, Minchul Sung, Mohammad Alian, Chankyung Kim, Wooyeong Cho, Reum Oh, Seongil O, Jung Ho Ahn, and Nam Sung Kim. 2018. 3D-Xpath: High-Density Managed DRAM Architecture with Cost-effective Alternative Paths for Memory Transactions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 22, 12 pages.

[38] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 487–498.

[39] H.M. Levy, Jack L. Lo, J.S. Emer, R.L. Stamm, S.J. Eggers, and D.M. Tullsen. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceeding of the International Symposium on Computer Architecture (ISCA)*. 191–191.

[40] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 49–63.

[41] Yun Liang and Xiuhong Li. 2017. Efficient Kernel Management on GPUs. *ACM Transactions on Embedded Computing Systems* 16, 4, Article 115 (MAY 2017), 24 pages.

[42] Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith. 2008. Multicore Resource Management. *IEEE Micro* 28, 3 (2008), 6–16. https://doi.org/10.1109/MM.2008.43

[43] Nvidia. 2009. NVIDIA's Next Generation CUDA Compute Architecture:Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf.

[44] Nvidia. 2021. MULTI-PROCESS SERVICE. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[45] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 41–54.

[46] Byoungchan Oh, Nam Sung Kim, Jeongseob Ahn, Bingchao Li, Ronald G. Dreslinski, and Trevor Mudge. 2018. A Load Balancing Technique for Memory Channels. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia, USA) *(MEMSYS '18)*. Association for Computing Machinery, 55–66.

[47] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 593–606.

[48] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 527–540.

[49] S. Phadke and S. Narayanasamy. 2011. MLP aware Heterogeneous Memory System. In *Design, Automation Test in Europe (DATE)*. 1–6.

[50] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 743–758.

[51] J. Power, M. D. Hill, and D. A. Wood. 2014. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[52] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 185–197.

[53] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 180–192.

[54] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *Proceedings of the International Symposium on Microarchitecture (MICRO) (MICRO-51)*. 352–363.

[55] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2008. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 135–144.

[56] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report.

[57] C. Sun, C. H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. S. Peh, and V. Stojanovic. 2012. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*. 201–210.

[58] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. 2016. GPUvm: GPU Virtualization at the Hypervisor. *IEEE Trans. Comput.* 65, 9 (2016), 2752–2766.

[59] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling Preemptive Multiprogramming on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 193–204.

[60] D.M. Tullsen and J.A. Brown. 2001. Handling Long-latency Loads in a Simultaneous Multithreading Processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 318–327.

[61] Lao Tzu. 1997. *Tao te ching*. Edizioni Mediterranee.

[62] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. 2013. Fairness-Aware Scheduling on Single-ISA Heterogeneous Multi-Cores. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 177–187.

[63] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 279–289.

[64] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. 2016. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[65] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog. 2018. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 247–258.

[66] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog. 2018. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 247–258. https://doi.org/10.1109/HPCA.2018.00030

[67] Xiaodong Wang and José F. Martínez. 2015. Change: A Market-based Approach to Scalable Dynamic Multi-resource Allocation in Multicore Architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 113–125.

[68] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 358–369.

[69] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 230–242.

[70] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang. 2020. HPE: Hierarchical Page Eviction Policy for Unified Memory in GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2461–2474. https://doi.org/10.1109/TCAD.2019.2944790

[71] W. Zhao, Q. Chen, and M. Guo. 2018. KSM: Online Application-Level Performance Slowdown Prediction for Spatial Multitasking GPGPU. *IEEE Computer Architecture Letters* 17, 2 (July 2018), 187–191.

[72] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and Reining in Application-Level Slowdown on Spatial Multitasking GPUs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*.

[73] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. HSM: A Hybrid Slowdown Model for Multitasking GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1371–1385.

[74] Xia Zhao, Zhiying Wang, and Lieven Eeckhout. 2018. Classification-Driven Search for Effective SM Partitioning in Multitasking GPUs. In *Proceedings of the International Symposium on Supercomputing (ICS)*. 65–75.

[75] Xia Zhao, Zhiying Wang, and Lieven Eeckhout. 2019. HeteroCore GPU to Exploit TLP-Resource Diversity. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (2019), 93–106. https://doi.org/10.1109/TPDS.2018.2854764

[76] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[77] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 129–142.