Page Placement Strategies for GPUs within Heterogeneous Memory Systems

Neha Agarwal[‡] * David Nellans[†] Mark Stephenson[†] Mike O'Connor[†] Stephen W. Keckler[†]
University of Michigan[‡] NVIDIA[†]
nehaag@umich.edu, {dnellans,mstephenson,moconnor,skeckler}@nvidia.com

Abstract

Systems from smartphones to supercomputers are increasingly heterogeneous, being composed of both CPUs and GPUs. To maximize cost and energy efficiency, these systems will increasingly use globally-addressable heterogeneous memory systems, making choices about memory page placement critical to performance. In this work we show that current page placement policies are not sufficient to maximize GPU performance in these heterogeneous memory systems. We propose two new page placement policies that improve GPU performance: one application agnostic and one using application profile information. Our application agnostic policy, bandwidth-aware (BW-AWARE) placement, maximizes GPU throughput by balancing page placement across the memories based on the aggregate memory bandwidth available in a system. Our simulation-based results show that BW-AWARE placement outperforms the existing Linux INTERLEAVE and LOCAL policies by 35% and 18% on average for GPU compute workloads. We build upon BW-AWARE placement by developing a compiler-based profiling mechanism that provides programmers with information about GPU application data structure access patterns. Combining this information with simple program-annotated hints about memory placement, our hint-based page placement approach performs within 90% of oracular page placement on average, largely mitigating the need for costly dynamic page tracking and migration.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage management—Main memory

Keywords Bandwidth, Page placement, Linux, Program annotation

1. Introduction

GPUs are now ubiquitous in systems ranging from mobile phones to datacenters like Amazon's elastic compute cloud (EC2) and HPC installations like Oak Ridge National Laboratory's Titan supercomputer. In all of these systems, GPUs are increasingly being used for processing beyond traditional computer graphics, including image processing, computer vision, machine learning, physical dynamics in games, and modeling high energy particle interactions. Regardless of the class of system being considered, GPU/CPU architectures are evolving towards general-purpose cache coherent non-uniform memory access (CC-NUMA) designs with both CPUs and GPUs being able to access a unified globally-addressable memory [18]. While some of these systems may share a single homogeneous pool of memory, an increasing number of systems use heterogeneous memory technologies. Specifically, cost and/or energy concerns are driving memory system architects to provide a pool of high-bandwidth memory as well as a higher-capacity pool of lower-cost and/or lower-power memory.

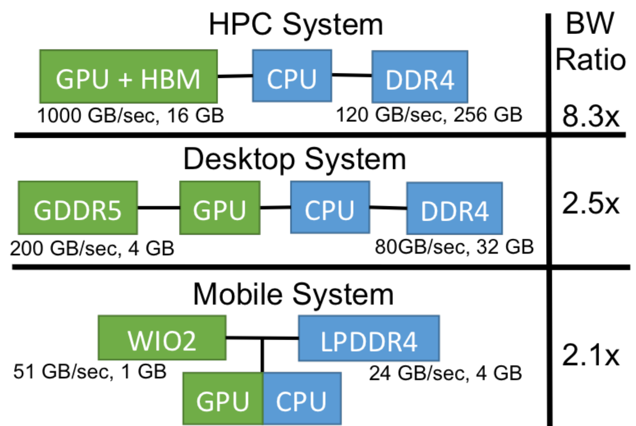


Figure 1: BW-Ratio of high-bandwidth vs high-capacity memories for likely future HPC, desktop, and mobile systems

Figure 1 shows several processor and memory topology options that are likely to be common over the next several years. While traditional systems are likely to continue using commodity DDR3 and soon DDR4, many future GPUs and mobile systems are moving to also include higher bandwidth, but capacity limited, on-package memories such as High Bandwidth Memory (HBM) or Wide-IO2 (WIO2). Regardless the type of machine, both memories will be globally accessible to maximize aggregate capacity and performance, making all systems non-uniform memory access (NUMA) due to variations in latency, bandwidth, and connection topology. Depending on the memories paired the bandwidth ratio between the bandwidth-optimized (BO) and capacity or cost optimized (CO) memory pools may be as low as $2\times$ or as high as $8\times$.

To date, GPU-attached bandwidth optimized (BO) memory has been allocated and managed primarily as the result of explicit, programmer-directed function calls. As heterogeneous GPU/CPU

* This work was done while Neha Agarwal was an intern at NVIDIA.

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 ACM 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.
ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694381

systems move to a transparent unified memory system, the OS and runtime systems will become increasingly responsible for memory management functions such as page placement, just as they are in CPU-only NUMA systems today. In CC-NUMA systems, the notion of local versus remote memory latency is exposed to the operating system via the Advanced Configuration and Power Interface (ACPI). The latency differences between local and remote memory account for the additional latency incurred due to accessing remote memory across the system interconnect. In these systems, latency information, alone, is adequate, as CPUs are generally more performance sensitive to memory system latency, rather than other memory characteristics.

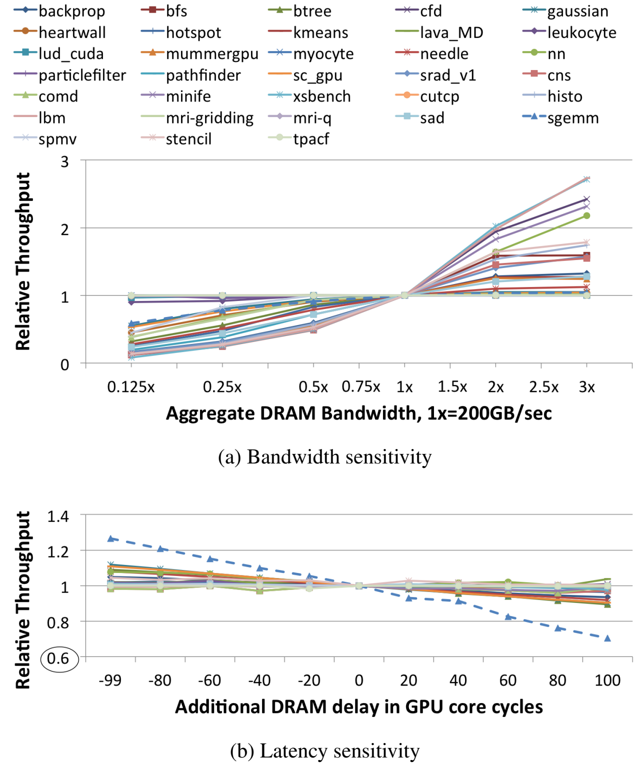
In contrast, massively parallel GPUs and their highly-threaded programming models have been designed to gracefully handle long memory latencies, instead demanding high bandwidth. Unfortunately, differences in bandwidth capabilities, read versus write performance, and access energy are not exposed to software; making it difficult for the operating system, runtime, or programmer to make good decisions about memory placement in these GPU-equipped systems. This work explores the effect on GPU performance of exposing memory system bandwidth information to the operating system/runtime and user applications to improve the quality of dynamic page placement decisions. Contributions of this work include:

1. We show that existing CPU-oriented page placement policies are not only sub-optimal for placement in GPU-based systems, but simply do not have the appropriate information available to make informed decisions when optimizing for bandwidth-asymmetric memory. Exposing additional bandwidth information to the OS, as is done for latency today, will be required for optimized decision making.
2. Perhaps counter-intuitively we show that, placing all pages in the bandwidth optimized memory is not the best performing page placement policy for GPU workloads. We propose and simulate a new bandwidth-aware (BW-AWARE) page placement policy that can outperform Linux's current bandwidth-optimized INTERLEAVE placement by 35% and the default latency optimized LOCAL allocation policy by as much as 18%, when the application footprint fits within bandwidth-optimized memory capacity.
3. For *memory capacity constrained* systems (i.e. bandwidth-optimized memory capacity is insufficient for the workload footprint), we demonstrate that using simple application annotations to inform the OS/runtime of hot versus cold data structures can outperform the current Linux INTERLEAVE and LOCAL page placement policies. Our annotation based policy combined with bandwidth information can outperform these page placement policies by 19% and 12% respectively, and get within 90% of oracle page placement performance.

2. Motivation and Background

2.1 Heterogeneous CC-NUMA Systems

Systems using heterogeneous CPU and GPU computing resources have been widely used for several years. Until recently, the GPU has been managed as a separate accelerator, requiring explicit memory management by the programmer to transfer data to and from the GPU's address space and (typically) the GPU's locally-attached memory. To increase programmer productivity and broaden the classes of workloads that the GPU can execute, recent systems have introduced automatic memory management enabling the CPU and GPU to access a unified address space and transparently migrate data at a page-level granularity [36]. The next step in this evolution is making the GPU a cache-coherent peer to the



(a) Bandwidth sensitivity

(b) Latency sensitivity

Figure 2: GPU performance sensitivity to bandwidth and latency changes.

CPU in the memory system, which is the stated goal of a number of commercial vendors [18].

While some heterogeneous CPU/GPU systems share a single unified physical memory [12], discrete GPUs are already using specialized DRAM optimized to meet their high bandwidth demands. To highlight the sensitivity of GPU performance to memory characteristics, Figures 2a and 2b show the performance variation as memory bandwidth and latency vary for a variety of GPU compute benchmarks from the Rodinia [10] and Parboil [44] suites, as well as a number of recent HPC [8, 17, 35, 46] workloads. Most of these GPU workloads are sensitive to changes in bandwidth, while showing much more modest sensitivity to varying the latency; only `sgemm` stands out as highly latency sensitive among these 33 workloads. Some application kernels are neither bandwidth nor latency sensitive and do not see significant performance variation as modifications are made to the memory subsystem. While GPU-equipped systems generally require bandwidth-optimized memories to achieve peak performance, these memory technologies have significant cost, capacity, and/or energy disadvantages over alternative DRAM technologies.

The most common bandwidth-optimized (BO) memory technology today is GDDR5 [19]. Providing a per-pin data rate of up to 7Gbps, this memory technology is widely used with discrete GPUs used in HPC, workstation, and desktop systems. Due to the high data rates, GDDR5 systems require significant energy per access and are unable to support high-capacity multi-rank systems. In contrast, the roadmap for the next several years of cost/capacity-optimized (CO) DRAM (DDR4 and LPDDR4) provides a per-pin data rate that reaches only 3.2 Gbps. However, these CO DRAM technologies provide similar latency at a fraction of the cost and lower energy per access compared to the BO GDDR5 memories.

Looking forward, systems requiring more bandwidth and/or reduced energy per access are moving to die-stacked DRAM technologies [24, 26]. These bandwidth-optimized stacked memories are significantly more energy-efficient than off-package memory technologies like GDDR5, DDR4, and LPDDR4. Unfortunately, the number of DRAM die that can be economically stacked in a single package is limited, necessitating systems to also provide a pool of off-package capacity-optimized DRAM.

This disaggregation of memory into on-package and off-package pools is one factor motivating the need to revisit page placement within the context of GPU performance. Future GPU/CPU systems are likely to take this disaggregation further and move capacity-optimized memory not just off the GPU package, but across a high speed interconnect where it is physically attached to the CPU rather than the GPU, or possibly further [30]. In a CC-NUMA system, the physical location of this capacity-optimized memory only changes the latency and bandwidth properties of this memory pool – it is functionally equivalent regardless of being CPU or GPU locally attached. A robust page placement policy for GPUs will abstract the on-package, off-package, and remote memory properties into performance and power characteristics based on which it can make optimized decisions.

2.2 Current OS NUMA Page Placement

In modern symmetric multiprocessor (SMP) systems, each socket typically consists of several cores within a chip multi-processor (CMP) that share last-level caches and on-chip memory controllers [22]. The number of memory channels connected to a processor socket is often limited by the available pin count. To increase the available memory bandwidth and capacity in a system, individual sockets can be connected via a cache coherent interconnect fabric such as Intel's Quick Path [21], AMD's HyperTransport [20], or NVIDIA's NVLink [38]. A single socket, the processors within it, and the physically attached memory comprise what an operating system sees as a local NUMA zone. Each socket is a separate NUMA zone. While a processor within any given zone can access the DRAM within any other zone, there is additional latency to service this memory request compared to a locally serviced memory request because the request must be routed first to its own memory controller, across the socket interconnect, and through the remote memory controller.

Operating systems such as Linux have recognized that, unless necessary, it is typically better for applications to service memory requests from their own NUMA zone to minimize memory latency. To get the best performance out of these NUMA systems, Linux learns system topology information from the Advanced Configuration and Power Interface (ACPI) System Resource Affinity Table (SRAT) and memory latency information from the ACPI System Locality Information Table (SLIT). After discovering this information, Linux provides two basic page placement policies that can be specified by applications to indicate where they prefer their physical memory pages to be placed when using standard `malloc` and `mmap` calls to allocate memory.

LOCAL: The default policy inherited by user processes is *LOCAL* in which physical page allocations will be from memory within the local NUMA zone of the executing process, unless otherwise specified or due to capacity limitations. This typically results in allocations from memory physically attached to the CPU on which the process is running, thus minimizing memory access latency.

INTERLEAVE: The second available allocation policy, which processes must specifically inform the OS they would like to use, is *INTERLEAVE*. This policy allocates pages round-robin across all (or a subset) of the NUMA zones within the SMP system to balance bandwidth across the memory pools. The downside of this policy is that the additional bandwidth comes at the expense of increased memory latency. Today, the OS has no knowledge about the relative bandwidth of memories in these different NUMA zones because SMP systems have traditionally had bandwidth-symmetric memory systems.

In addition to these OS placement policies, Linux provides a library interface called *libNUMA* for applications to request memory allocations from specific NUMA zones. This facility provides low-level control over memory placement but requires careful programming because applications running on different systems will often have different NUMA-zone layouts. Additional difficulties arise because there is no performance feedback mechanism available to programmers when making memory placement decisions, nor are they aware of which processor(s) their application will be running on while writing their application.

With the advent of heterogeneous memory systems, the assumptions that operating system NUMA zones will be symmetric in bandwidth, latency, and power characteristics break down. The addition of heterogeneous GPU and CPU computing resources further stresses the page placement policies since processes may not necessarily be migrated to help mitigate performance imbalance, as certain phases of computation are now pinned to the type of processor executing the program. As a result, data placement policies combined with bandwidth-asymmetric memories can have significant impact on GPU, and possibly CPU, performance.

2.3 Related Work

With the introduction of symmetric multiprocessors, significant work has examined optimal placement of processes and memory in CC-NUMA systems [6, 7, 23, 29, 48, 50]. While much of this early work focused on placing processes and data in close proximity to each other, more recent work has recognized that sharing patterns, interconnect congestion, and even queuing delay within the memory controller are important metrics to consider when designing page and process placement policies [2, 5, 11, 13, 27, 45, 53]. Nearly all of these works focus on improving traditional CPU throughput where reduced memory latency is the primary driver of memory system performance. Recent work from Gerofi et al. [16] examines TLB replacement policies for the Xeon Phi co-processor with a focus on highly parallel applications with large data footprints.

Using non-DRAM technologies or mixed DRAM technologies for main memory systems to improve power consumption on traditional CPUs has also been explored by several groups [4, 9, 28, 34, 40–42]. Much of this work has focused on overcoming the performance peculiarities that future non-volatile memory (NVM) technologies may have compared to existing DRAM designs. In addition to mixed technology off-package memories, upcoming on-package memories provide opportunities for latency reduction by increasing the number of banks available to the application [14] and may one day be configurable to balance bandwidth application needs with power consumption [51]. An alternative to treating heterogeneous memory systems as a flat memory space is to use one technology as a cache for the other [25, 32]. While this approach has the advantage of being transparent to the programmer, OS, and runtime systems, few implementations [43] take advantage of the additive bandwidth available when using heterogeneous memory.

In the GPU space, Zhao et al. [52] have explored the affect of hybrid DRAM-NVM systems on GPU compute workloads, making the observation that modern GPU designs are very good at hiding variable memory system latency. Wang et al. [49] explore a mixed NVM-DRAM system that uses compiler analysis to identify near-optimal data placement across kernel invocations for their heterogeneous memory system. While their system does not significantly improve performance, it offers improved power efficiency through

use of NVM memory and shows that software based page placement, rather than hardware caching, can be a viable alternative to managing heterogeneous memory for use with GPUs.

3. BW-AWARE Page Placement

Using all available memory bandwidth is a key driver to maximizing performance for many GPU workloads. To exploit this observation, we propose a new OS page placement algorithm which accounts for the bandwidth differential between different bandwidth-optimized and capacity-optimized memories when making page placement decisions. This section discusses the need, implementation, and results for a bandwidth-aware (BW-AWARE) page placement policy for systems where the application footprint fits within BO memory, the common case for GPU workloads today. Later in Section 5, we discuss an extension to BW-AWARE placement for systems where memory placement decisions are constrained by the capacity of the bandwidth-optimized memory. Both HPC systems trying to maximize in-memory problem footprint and mobile systems which are capacity limited by cost and physical part dimensions may soon encounter these capacity constraints with heterogeneous memories.

3.1 Bandwidth Maximized Page Placement

The goal of bandwidth-aware page placement is to enable a GPU to effectively use the total combined bandwidth of *all* the memory in the system. Because GPUs are able to hide high memory latency without stalling their pipelines, all memories in a system can be used to service GPU requests, even when those memories are off-package or require one or more hops through a system interconnect to access. To exploit bandwidth-heterogeneous memories, our BW-AWARE policy places physical memory pages in the ratio of aggregate bandwidths of the memories in the system without requiring any knowledge of page access frequency. Below we derive that this placement policy is optimal for maximizing bandwidth.

Consider a system with bandwidth-optimized and capacity-optimized memories with bandwidths b_B and b_C respectively, where unrestricted capacity of both memories are available. Let f_B represent fraction of data placed in the BO memory and $1 - f_B$ in the CO memory. Let us assume there are total of N memory accesses uniformly spread among different pages. Then the total amount of time spent by the BO memory to serve $N * f_B$ memory accesses is $N * f_B/b_B$ and that by the CO memory to serve $N(1 - f_B)$ memory accesses is $N(1 - f_B)/b_C$. Since requests to these two memories are serviced in parallel, the total time T to serve the memory requests is:

$$T = max(N * f_B/b_B, N(1 - f_B)/b_C)$$

To maximize performance, T must be minimized. Since, $N*f_B/b_B$ and $N(1 - f_B)/b_C$ are linear in f_b and $N * f_B/b_B$ is increasing function while $N(1 - f_B)/b_C$ is decreasing, the minimum of T occurs when both are equal:

$$T_{opt} = N * f_B/b_B = N(1 - f_B)/b_C$$

Therefore,

$$f_{Bopt} = b_B/(b_B + b_C)$$

Because we have assumed that all pages are accessed uniformly, the optimal page placement ratio is the same as the bandwidth service ratio between the bandwidth-optimized and capacity-optimized memory pools. From this derivation we make two additional observations. First, BW-AWARE placement will generalize to an optimal policy where there are more than two technologies by placing pages in the bandwidth ratio of all memory pools. Second, a practical implementation of a BW-AWARE policy must be aware of the bandwidth provided by the various memory pools available within a system. Hence there is a need for a new System Bandwidth Information Table (SBIT), much like there is already a ACPI System Locality Information Table (SLIT) which exposes memory latency information to the operating system today. We will re-visit the assumption of uniform page access later in Section 4.1.

Simulator	GPGPU-Sim 3.x
GPU Arch	NVIDIA GTX-480 Fermi-like
GPU Cores	15 SMs @ 1.4Ghz
L1 Caches	16kB/SM
L2 Caches	Memory Side 128kB/DRAM Channel
L2 MSHRs	128 Entries/L2 Slice
Memory system	
GPU-Local GDDR5	8-channels, 200GB/sec aggregate
GPU-Remote DDR4	4-channels, 80GB/sec aggregate
DRAM Timings	RCD=RP=12,RC=40,CL=WR=12
GPU-CPU Interconnect Latency	100 GPU core cycles

Table 1: Simulation environment and system configuration for mixed memory GPU/CPU system.

3.2 Experimental Results

While it would be ideal to evaluate our page placement policy on a real CC-NUMA GPU/CPU system with a heterogeneous memory system, such systems are not available today. Mobile systems containing both ARM CPU cores and NVIDIA GPU cores exist today in products such as the NVIDIA Shield Portable, but use a single LPDDR3 memory system. Desktop and HPC systems today have heterogeneous memory attached to CPUs and discrete GPUs but these processors are not connected through a cache coherent interconnect. They require explicit user management of memory if any data is to be copied from the host CPU memory to the GPU memory or vice versa over the PCIe interconnect. Pages can not be directly placed into GPU memory on allocation by the operating system. Without a suitable real system to experiment on, we turned to simulation to evaluate our page placement improvements.

3.2.1 Methodology

To evaluate BW-AWARE page placement, we simulated a heterogeneous memory system attached to a GPU comprised of both bandwidth-optimized GDDR and cost/capacity-optimized DDR where the GDDR memory is attached directly to the GPU. No contemporary GPU system is available which supports cache-coherent access to heterogeneous memories. Commonly available PCIe-attached GPUs are constrained by interconnect bandwidth and lack of cache-coherence; while cache-coherent GPU systems, such as AMD's Kaveri, do not ship with heterogeneous memory.

Our simulation environment is based on GPGPU-Sim [3] which has been validated against NVIDIA's Fermi class GPUs and is reported to match hardware performance with up to 98.3% accuracy [1]. We modified GPGPU-Sim to support a heterogeneous GDDR5-DDR4 memory system with the simulation parameters listed in Table 1. We model a baseline system with 200GB/s of GPU-attached memory bandwidth and 80GB/s of CPU-attached memory bandwidth, providing a bandwidth ratio of $2.5\times$. We made several changes to the baseline GTX-480 model to bring our configuration in-line with the resources available in more modern GPUs, including a larger number of MSHRs and higher clock frequency. With a focus on memory system performance, we evaluate GPU workloads which are sensitive to memory bandwidth or latency from three benchmark suites: Rodinia [10], Parboil [44] and recent HPC [8, 17, 35, 46] workloads; those which are compute-bound see little change in performance due to changes made to the mem-

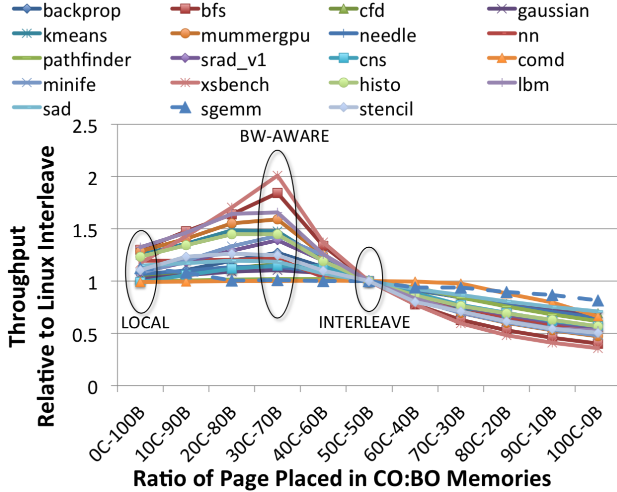


Figure 3: GPU workload performance with different page placement policies. xC-yB policy represents x : y data transfer ratio from CO and BO memory respectively.

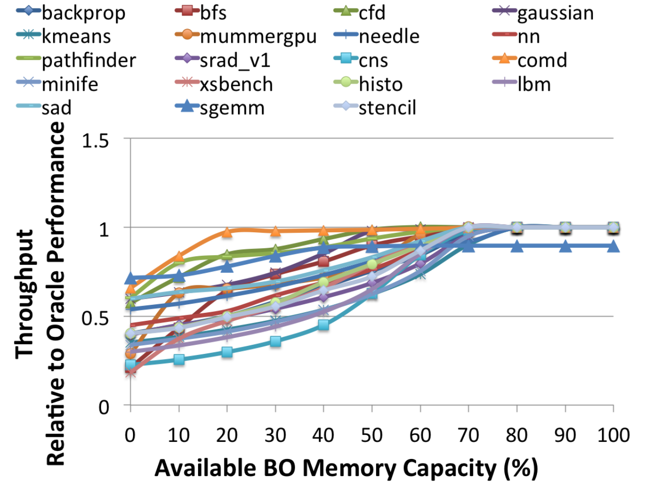


Figure 4: Performance of BW-AWARE placement as application footprint exceeds available high-bandwidth memory capacity.

ory system. For the remainder of the paper we show results for 19 benchmarks, 17 of which are sensitive to memory bandwidth while also providing `comd` and `sgemm` results to represent applications which are memory insensitive and latency sensitive respectively.

As noted in Section 2.1, attaching the capacity-optimized memory directly to the GPU is functionally equivalent to remote CPU attached memory, but with different latency parameters. To simulate an additional interconnect hop to remote CPU-attached memory, we model a fixed, pessimistic, additional 100 cycle latency to access the DDR4 memory from the GPU. This overhead is derived from the single additional interconnect hop latency found in SMP CPU-only designs such as the Intel XEON [22]. Our heterogeneous memory model contains the same number of MSHRs per memory channel as the baseline memory configuration. The number of MSHRs in the baseline configuration is sufficient to effectively hide the additional interconnect latency to the DDR memory in Figure 2b. Should MSHR quantity become an issue when supporting two level memories, previous work has shown that several techniques can efficiently increase MSHRs with only modest cost [33, 47].

Implementing a BW-AWARE placement policy requires adding another mode (`MPOL_BWAWARE`) to the `set_mempolicy()` system call in Linux. When a process uses this mode, the Linux kernel will allocate pages from the two memory zones in the ratio of their bandwidths. These bandwidth ratios may be obtained from future ACPI resources or dynamically determined by the GPU runtime at execution time.

3.2.2 BW-AWARE Performance

We define our BW-AWARE page placement policy xC-yB, where x and y denote the percentage of pages placed in a given memory technology, C stands for capacity-optimized memory and B stands for bandwidth-optimized memory. By definition $x + y = 100$. For our baseline system with 200GB/sec bandwidth-optimized memory and 80GB/sec of capacity-optimized memory the aggregate system bandwidth is 280GB/sec. In this notation, our BW-AWARE policy will then be $x = 80/280 = 28\%$ and $y = 200/280 = 72\%$, represented as $28C$-$72B$. However, for simplicity we will round this to $30C$-$70B$ for use as the placement policy. For processes running on the GPU, the LOCAL policy would be represented as $0C$-$100B$; $50C$-$50B$ corresponds to the bandwidth spreading Linux INTERLEAVE policy.

To achieve the target $30C$-$70B$ bandwidth ratio, we implemented BW-AWARE placement as follows. On any new physical page allocation, a random number in the range $[0, 99]$ is generated. If this number is ≥ 30, the page is allocated from the bandwidth-optimized memory; otherwise it is allocated in the capacity-optimized memory. A LOCAL allocation policy can avoid the comparison if it detects either B or C has the value zero. While this implementation does not exactly follow the BW-AWARE placement ratio due to the use of random numbers, in practice this simple policy converges quickly towards the BW-AWARE ratio. This approach also requires no history of previous placements nor makes any assumptions about the frequency of access to pages, minimizing the overhead for making placement decisions which are on the software fast-path for memory allocation.

Figure 3 shows the application performance as we vary the ratio of pages placed in each type of memory from 100% BO to 100% CO. For all bandwidth-sensitive applications, the maximum performance is achieved when using the correct BW-AWARE $30C$-$70B$ placement ratio. We find that, on average, a BW-AWARE policy performs 18% better than the Linux LOCAL policy and 35% better than the Linux INTERLEAVE policy. However, for latency sensitive applications, such as `sgemm`, the BW-AWARE policy may perform worse than a LOCAL placement policy due to an increased number of accesses to higher latency remote CO memory. The BW-AWARE placement policy suffers a worse case performance degradation of 12% over the LOCAL placement policy in this scenario.

Because the current Linux INTERLEAVE policy is identical to BW-AWARE for a bandwidth-symmetric DRAM $50C$-$50B$ memory technology pairing, we believe a BW-AWARE placement policy could simply replace the current Linux INTERLEAVE policy without having significant negative side affects on existing CPU or GPU workloads. Because maximizing bandwidth is more important than minimizing latency for GPU applications, BW-AWARE placement may be a good candidate to become the default placement policy for GPU-based applications.

3.2.3 Effective Improvement in Problem Sizing

Figure 4 shows the application throughput as we reduce the capacity of our bandwidth-optimized memory pool as a fraction of

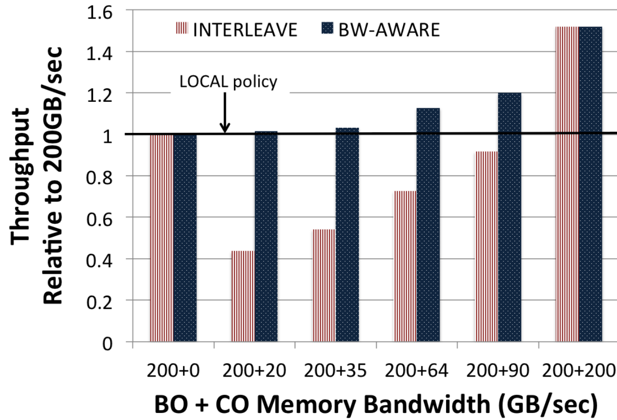


Figure 5: Performance comparison between BW-AWARE, INTERLEAVE, and LOCAL page placement policies while varying the memory bandwidth ratio.

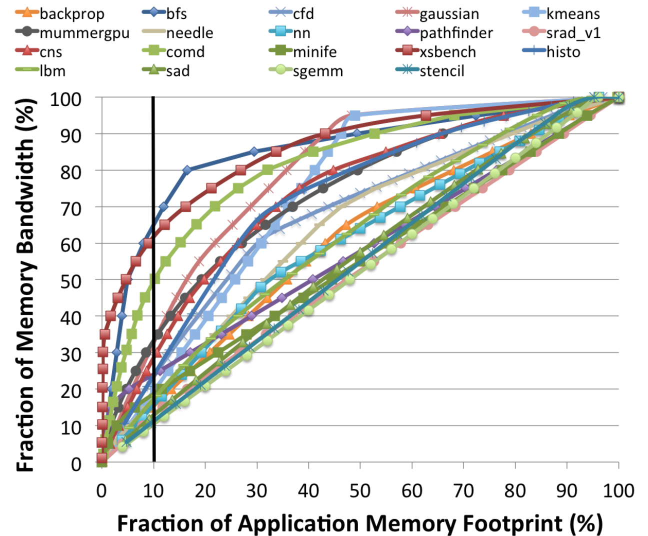


Figure 6: Data bandwidth cumulative distribution function with pages sorted from hot (most memory accesses) to cold (least memory accesses).

the total application footprint. BW-AWARE placement is able to achieve near peak performance even when only 70% of the application footprint fits within the BO memory because BW-AWARE placement places only 70% of pages in BO memory, with the other 30% is placed in the less expensive capacity-optimized memory. Thus, GPU programmers who today tune their application footprint to fit entirely in the GPU-attached BO memory could gain an extra 30% effective memory capacity by exploiting the CPU-attached CO memory with a BW-AWARE placement policy. However, as the bandwidth-optimized memory capacity drops to less than 70% of application footprint, performance begins to fall off. This effect is due to the ratio of bandwidth service from the two memory pools no longer matching the optimal ratio of $30C$-$70B$, with more data being serviced from the capacity optimized ratio than is ideal. Applications which are insensitive to memory bandwidth (shown as having little change in Figure 3), tend to maintain their performance at reduced capacity points (shown as having little change in Figure 4), because the average bandwidth reduction does not strongly affect their performance. Conversely, those applications with strong BW-performance scaling tend to see larger performance reduction as the average bandwidth available is reduced, due to capacity constraints forcing a disproportionate number of memory accesses to the lower bandwidth CO memory. The performance at 70% memory capacity does not exactly match 100% of ideal because the actual ratio of bandwidth in our system is $28C$-$72B$ not $30C$-$70B$.

3.2.4 Sensitivity to NUMA BW-Ratios

Heterogeneous memory systems are likely to come in a variety of configurations. For example, future mobile products may pair energy efficient and bandwidth-optimized Wide-IO2 memory with cost-efficient and higher capacity LPDDR4 memory. Using the mobile bandwidths shown in Figure 1, this configuration provides an additional 31% in memory bandwidth to the GPU versus using the bandwidth-optimized memory alone. Similarly, HPC systems may contain GPUs with as many as 4 on-package bandwidth-optimized HBM stacks and high speed serial interfaces to bulk capacity cost/capacity-optimized DDR memory expanders providing just 8% additional memory bandwidth. While we have explored BW-AWARE placement in a desktop-like use case, BW-AWARE page placement can apply to all of these configurations.

Figure 5 shows the average performance of BW-AWARE, INTERLEAVE, and LOCAL placement policies as we vary the additional bandwidth available from the capacity-optimized memory from 0GB/s–200GB/s. As the bandwidth available from capacity-optimized memory increases, the LOCAL policy fails to take advantage of it by neglecting to allocate any pages in the capacity-optimized memory. The Linux INTERLEAVE policy, due to its fixed round-robin allocation, loses performance in many cases because it oversubscribes the capacity-optimized memory, resulting in less total bandwidth available to the application. On the other hand, BW-AWARE placement is able to exploit the bandwidth from the capacity-optimized memory regardless the amount of additional bandwidth available. Because BW-AWARE placement performs identically to INTERLEAVE for symmetric memory and outperforms it in all heterogeneous cases, we believe that BW-AWARE placement is a more robust default policy than INTERLEAVE when considering bandwidth-sensitive GPU workloads.

4. Understanding Application Memory Use

Section 3 showed that optimal BW-AWARE placement requires the majority of the application footprint to fit in the bandwidth-optimized memory to match the bandwidth service ratios of the memory pools. However, as shown in Figure 1, systems may have bandwidth-optimized memories that comprise less than 10% the total memory capacity, particularly those using on-package memories which are constrained by physical dimensions. If the application footprint grows beyond the bandwidth-optimized capacity needed for BW-AWARE placement, the operating system has no choice but to steer remaining page allocations into the capacity-optimized memory. Unfortunately, additional pages placed in the CO memory will skew the ratio of data transferred from each memory pool away from the optimal BW-AWARE ratio.

For example, in our simulated system if the bandwidth-optimized memory can hold just 10% of the total application memory footprint, then a BW-AWARE placement would end up placing 10% pages in the BO memory; the remaining 90% pages must be spilled exclusively to the capacity-optimized memory. This ratio of $90C$-$10B$ is nearly the inverse of the performance-optimized ratio of $30C$-$70B$. To improve upon this capacity-induced placement problem, we recognize that not all pages have uniform access frequency, and we can selectively place hot pages in the BO memory and cold

pages in the CO memory. In this work we define page *hotness* as the number of accesses to that page that are served from DRAM.

4.1 Visualizing Page Access Distribution

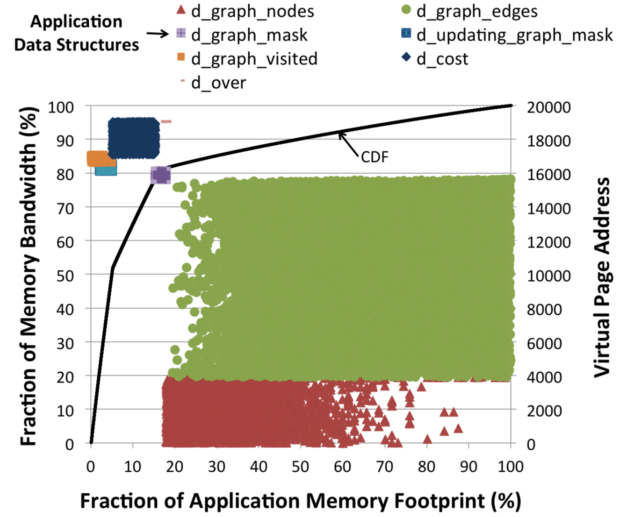
Figure 6 shows the cumulative distribution function (CDF) for memory bandwidth as a fraction of the total memory footprint for each of our workloads. The CDF was generated by counting accesses to each 4kB page, after being filtered by on-chip caches, and then sorting the pages from greatest to least number of accesses. Applications that have the same number of accesses to all pages have a linear CDF, whereas applications in which some pages are accessed more than others have a non-linear CDF skewed towards the left of the graph. For example, we see that for applications like `bfs` and `xsbench`, over 60% of the memory bandwidth stems from within only 10% of the application's allocated pages. Skewing the placement of these hot pages towards bandwidth-optimized memory will improve the performance of GPU workloads which are capacity constrained by increasing the traffic to the bandwidth-optimized memory. However, for applications which have linear CDFs, there is little headroom for improved page placement over BW-AWARE placement.

Figure 6 also shows that some workloads have sharp inflection points within the CDF, indicating that distinct ranges of physical addresses appear to be clustered as hot or cold. To determine if these inflection points could be correlated to specific memory allocations within the application, we plotted the virtual addresses that correspond to application pages in the CDF, and then reverse-mapped those address ranges to memory allocations for program-level data structures, with the results shown in Figure 7. The x-axis shows the fraction of pages allocated by the application, where pages are sorted from greatest to least number of accesses. The primary y-axis (shown figure left) represents the CDF of memory bandwidth among the pages allocated by the application (also shown in Figure 6). Each point on the secondary scatter plot (shown figure right) shows the virtual address of the corresponding page on the x-axis. The points (pages) are colored according to different data structures they were allocated from in the program source.

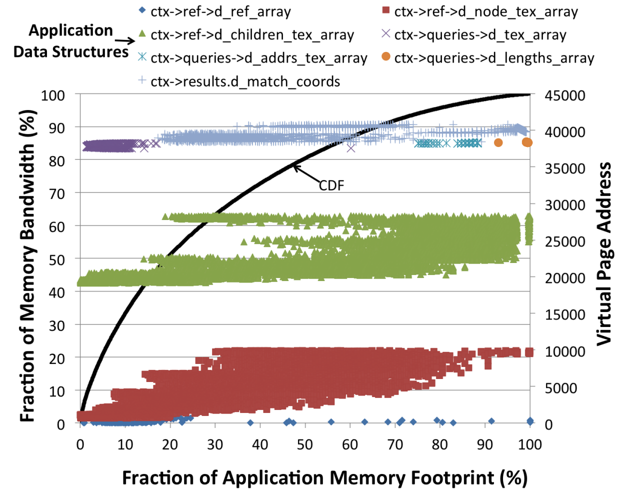
We analyze three example applications, `bfs`, `mummergpu`, and `needle` which show different memory access patterns. For `bfs`, we can see three data structures: `d_graph_visited`, `d_updating_graph_mask`, and `d_cost` consume $\approx$ 80% of the total application bandwidth while accounting for $\approx$ 20% of the memory footprint. However for `mummergpu`, the memory hotness does not seem to be strongly correlated to any specific application data structures. Several sub-structures have similar degrees of hotness and some virtual address ranges (10,000-19,000 and 29,000-37,000) are allocated but never never accessed. In `needle`, which has a fairly linear CDF, the degree of memory hotness actually varies within the same data structure. While we examined all applications with this analysis, we summarize our two key observations.

Individual pages can and often do have different degrees of hotness. Application agnostic page placement policies, including BW-AWARE placement, may leave performance on the table compared to a placement policy that is aware of page frequency distribution. Understanding the relative hotness of pages is critical to further optimizing page placement. If an application does not have a skewed CDF, then additional effort to characterize and exploit hotness differential will only introduce overhead without any possible benefit.

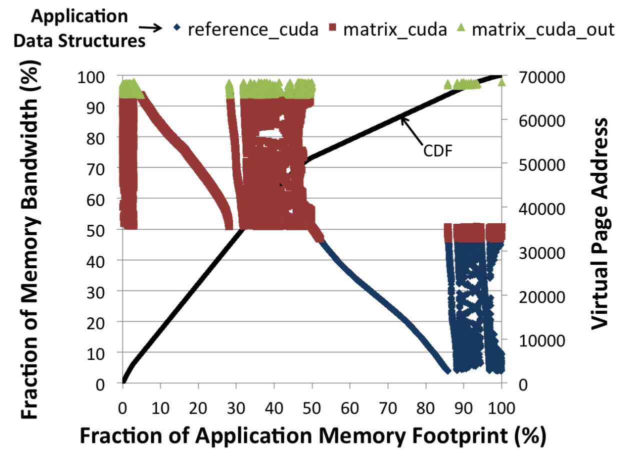
Workloads with skewed cumulative distribution functions often have sharp distinctions in page access frequency that map well to different application data structures. Rather than attempting to detect and segregate individual physical pages which may be hot or cold, application structure and memory allocation policy will likely provide good information about pages which will have similar degrees of hotness.



(a) `bfs`



(b) `mummergpu`



(c) `needle`

Figure 7: Application CDF of data footprint versus virtual address data layout.

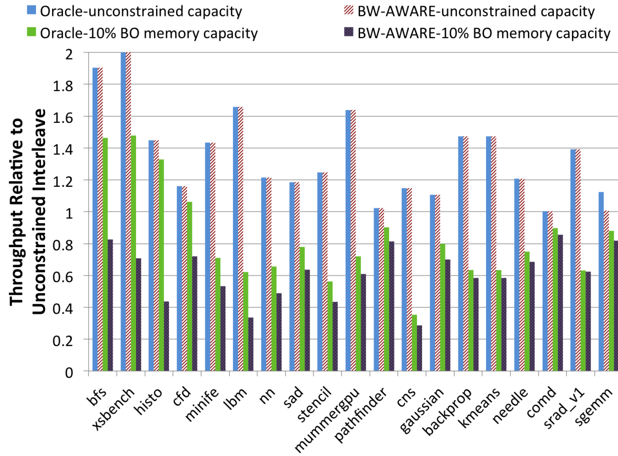


Figure 8: Oracle application performance of a constrained capacity system vs unconstrained capacity system

4.2 Oracle Page Placement

With knowledge that page accesses are highly differentiated for some applications, we implemented an oracle page placement policy to determine how much more application performance could be achieved compared to BW-AWARE placement in capacity constrained situations. Using perfect knowledge of page access frequency, an oracle page placement policy will allocate the hottest pages possible into the bandwidth-optimized memory until the target bandwidth ratio is satisfied, or the capacity of this memory is exhausted. We implemented this policy using two phase simulation. First, we obtained perfect knowledge of page access frequency. Then in a second simulation pass, we used this information to allocate pages to achieve the best possible data transfer ratio under a 10% capacity constraint where only 10% of the application memory footprint fits within the bandwidth-optimized memory.

Figure 8 compares the performance of the oracle and BW-AWARE placement policies in both the unconstrained and 10% capacity constrained configuration. Figure 8 confirms that BW-AWARE placement is near-optimal when applications are not capacity limited. This is because both BW-AWARE and oracle placement are both able to achieve the ideal bandwidth distribution, though the oracle policy is able to do this with a smaller memory footprint by placing fewer, hotter, pages in the BO memory. Under capacity constraints, however, the oracle policy can nearly double the performance of the BW-AWARE policy for applications with highly skewed CDFs and it outperforms BW-AWARE placement in all cases. This because the random page selection of BW-AWARE placement is not able to capture enough hot pages for placement in BO memory, before running out of BO memory capacity, to achieve the ideal bandwidth distribution. On average the oracle policy is able to achieve nearly 60% the application throughput of a system for which there is no capacity constraint. This additional performance, achievable through application awareness of memory access properties, motivates the need for further improvements in memory placement policy.

5. Application Aware Page Placement

Figure 7 visually depicts what may be obvious to performance-conscious programmers: *certain data structures are accessed more frequently than others.* For these programmers, the unbiased nature of BW-AWARE page placement is not desirable because all data structures are treated equally. This section describes compiler toolflow and runtime modifications that enable programmers to intelligently steer specific allocations towards bandwidth- or capacity-optimized memories and achieve near-oracle page placement performance.

To correctly annotate a program to enable intelligent memory placement decisions across a range of systems, we need two pieces of information about a program: (1) the relative memory access hotness of the data structures, and (2) the size of the data structures allocated in the program. To understand the importance of these two factors, let us consider the following scenario. In a program there are two data structure allocations with hotness H1 and H2. If the bandwidth-optimized memory capacity is large enough for BW-AWARE placement to be used without running into capacity constraints, then BW-AWARE page placement should be used irrespective of the hotness of the data structures. To make this decision we must know the application runtime memory footprint. However, if the application is capacity-constrained, then ideally the memory allocation from the hotter data structure should be preferentially placed in the BO memory. In this case, we need to know both the relative hotness and the size of the data structures to optimally place pages.

5.1 Profiling Data Structure Accesses in GPU Programs

While expert programmers may have deep understanding of their application characteristics, as machines become more complex and programs rely more on GPU accelerated libraries, programmers will have a harder time maintaining this intuition about program behavior. To augment programmer intuition about memory access behavior, we developed a new GPU profiler to provide information about program memory access behavior.

In this work we augmented `nvcc` and `ptxas`, NVIDIA's production compiler tools for applications written in CUDA [39] (NVIDIAs explicitly parallel programming model), to support data structure access profiling. When profiling is enabled, our compiler's code generator emits memory instrumentation code for all loads and stores that enables tracking of the relative access counts to virtually addressed data structures within the program. As with the GNU profiler `gprof` [15], the developer enables a special compiler flag that instruments an application to perform profiling. The developer then runs the instrumented application on a set of "representative" workloads, which aggregates and dumps a profile of the application.

When implementing this GPU memory profiling tool, one of the biggest challenges is that `nvcc` essentially generates two binaries: a host-side program, and a device-side program (that runs on the GPU). The profiler's instrumentation must track the execution of both binaries. On the host side, the profiler inserts code to track all instances and variants of `cudaMalloc`. The instrumentation associates the source code location of the `cudaMalloc` with the runtime virtual address range returned by it. The host side code is also instrumented to copy this mapping of line numbers and address ranges to the GPU before each kernel launch. The GPU-side code is instrumented by inserting code before each memory operation to check if the address falls within any of the ranges specified in the map.

For each address that falls within a valid range, a counter associated with the range is incremented, using atomic operations because the instrumentation code is highly multi-threaded. At kernel completion, this updated map is returned to the host which launched the kernel to aggregate the statistics about virtual memory location usage. Our profiler generates informative data structure mapping plots, like those shown in Figure 7, which application programmers can use to guide their understanding of the relative access frequencies of their data structures, one of the two required

pieces of information to perform intelligent near-optimal placement within an application.

5.2 Memory Placement APIs for GPUs

With a tool that provides programmers a profile of data structure hotness, they are armed with the information required to make page placement annotations within their application, but they are lacking a mechanism to make use of this information. To enable memory placement hints (which are not a functional requirement) for where data should be placed in a mixed BO-CO memory system, we also provide an alternate method for allocating memory. We introduce an additional argument to the `cudaMalloc` memory allocation functions that specifies in which domain the memory should be allocated (BO or CO) or to use BW-AWARE placement (BW). For example:

```
cudaMalloc(void **devPtr, size_t size, enum hint)
```

This hint is not machine specific and simply indicates if the CUDA memory allocator should make a best effort attempt to place memory within a BO or CO optimized memory using the underlying OS libNUMA functionality or fall back to the bandwidth-aware allocator. By providing an abstract hint, the CUDA runtime, rather than the programmer, becomes responsible for identifying and classifying the machine topology of memories as bandwidth or capacity optimized. While we have assumed bandwidth information is available in our proposed system bandwidth information table, programmatic discovery of memory zone bandwidth is also possible as a fall back mechanism [31]. In our implementation, memory hints are honored unless the memory pool is filled to capacity, in which case the allocator will fall back to the alternate domain. If no placement hint is provided, the CUDA runtime will fall back to using the application agnostic BW-AWARE placement for unannotated memory allocations. When a hint is supplied, the `cudaMalloc` routine uses the `mbind` system call in Linux to perform placement of the data structure in the corresponding memory.

5.3 Program Annotation For Data Placement

Our toolkit now includes a tool for memory profile generation and a mechanism to specify abstract memory placement hints. While programmers may choose to use this information directly, optimizing for specific machines, making these hints performance portable across a range of machines is harder as proper placement depends on application footprint as well as the memory capacities of the machine. For performance portability, the hotness and allocation size information must be annotated in the program before any heap allocations occur. We enable annotation of this information as two arrays of values that are linearly correlated with the order of the memory allocations in the program. For example Figure 9 shows the process of hoisting the size allocations manually into the `size` array and hotness into the `hotness` array.

We provide a new runtime function `GetAllocation` that then uses these two pieces of information, along with the discovered machine bandwidth topology, to compute and provide a memory placement hint to each allocation. `GetAllocation` determines appropriate memory placement hints by computing the ideal (BO or CO) memory location by first calculating the cumulative footprint of all data structures and then calculating the total number of identified data structures from `[1:N]` that will fit within the bandwidth-optimized memory before it exhausts the BO capacity.

It is not critical that programmers provide annotations for all memory allocations, only large or performance critical ones. For applications that make heavy use of libraries or dynamic decisions about runtime allocations, it may not be possible to provide good hinting decisions because determining the size of data structures allocated within libraries calls is difficult, if not impossible, in many

```
// n:input parameter
cudaMalloc(devPtr1, n*sizeof(int));
cudaMalloc(devPtr2, n*n);
cudaMalloc(devPtr3, 1000);
```

(a) Original code dependent allocations

```
// n: input parameter
// size[i]: Size of data structures
// hotness[i]: Hotness of data structures
size[0] = n*sizeof(int);
size[1] = n*n;
size[2] = 1000;
hotness[0] = 2;
hotness[1] = 3;
hotness[2] = 1;

// hint[i]: Computed data structure placement hints
hint[] = GetAllocation(size[], hotness[]);
cudaMalloc(devPtr1, size[0], hint[0]);
cudaMalloc(devPtr2, size[1], hint[1]);
cudaMalloc(devPtr3, size[2], hint[2]);
```

(b) Final code

Figure 9: Annotated pseudo-code to do page placement at runtime taking into account relative hotness of data structures and data structure sizes

cases.. While this process may seem impractical to a traditional high level language programmer, examining a broad range of GPU compute workloads has shown that in almost all GPU-optimized programs, the memory allocation calls are already hoisted to the beginning of the GPU compute kernel. The CUDA C Best Practices Guide advises the programmer to minimize memory allocation and de-allocation in order to achieve the best performance [37].

5.4 Experimental Results

Figure 10 shows the results of using our feedback-based optimization compiler workflow and annotating our workloads using our new page placement interfaces. We found that on our capacity-limited machine, annotation-based placement outperforms the Linux INTERLEAVE policy performance by 19% and naive BW-AWARE 30C-70B placement by 14% on average. Combining program annotated placement hints and our runtime placement engine achieves 90% of oracular page placement on average. In all cases our program-annotated page placement algorithm outperforms BW-AWARE placement, making it a viable candidate for optimization beyond BW-AWARE placement if programmers choose to optimize for heterogeneous memory system properties.

One of the drawbacks to profile-driven optimization is that data dependent runtime characteristics may cause different behaviors than were seen during the profiling phase. While GPU applications in production will be run without code modification, the data set and parameters of the workload typically vary in both size and value from run-to-run. Figure 11 shows the sensitivity of our workload performance to data input set changes, where placement was trained on the first data-set but compared to the oracle placement for each individual dataset. We show results for the four example applications which saw the highest improvement of oracle placement over BW-AWARE. For `bfs`, we varied the number of nodes and average degree of the graph. For `xsbench`, we changed three parameters: number of nuclides, number of lookups, and number of gridpoints in the data set. For `minife`, we varied the dimensions of the finite element problem by changing the input matrix. Finally,

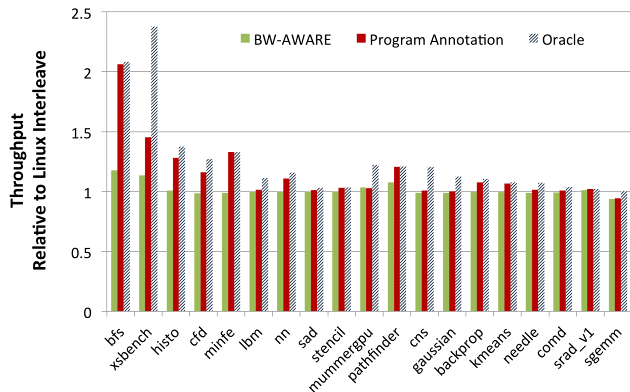


Figure 10: Profile-driven annotated page placement performance relative to INTERLEAVE, BW-AWARE and oracular policies under at 10% capacity constraint.

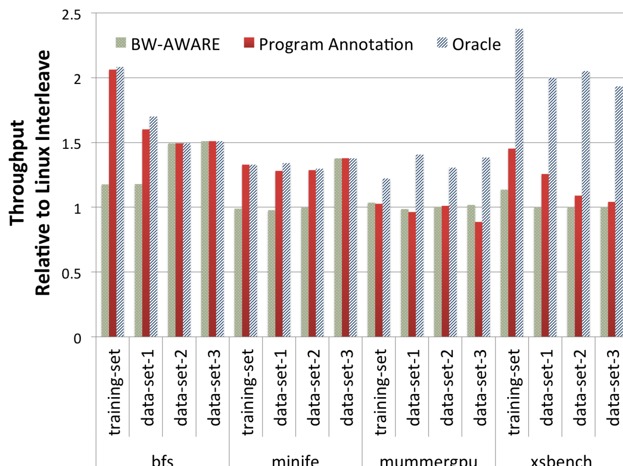


Figure 11: Annotated page placement effectiveness versus data sets variation after training phase under at 10% capacity constraint.

for mummergpu, we changed the number of queries and length of queries across different input data sets.

Using the profiled information from only the training set, we observe that annotated placement performs 29% better than the baseline Linux INTERLEAVE policy, performs 16% better than our own BW-AWARE 30C-70B placement, and achieves 80% of the oracle placement performance. This result indicates that for GPU compute applications, feedback-driven optimization for page placement is not overly sensitive to application dataset or parameter variation, although pessimistic cases can surely be constructed.

5.5 Discussion

The places where annotation-based placement falls short primarily come from three sources. First, our application profiling relies on spatial locality of virtual addresses to determine page hotness. We have shown that this spatial locality holds true for many GPU applications, but this is not guaranteed to always be the case. Allocations within libraries or larger memory ranges the programmer chooses to logically sub-allocate within the program will not exhibit this property. The second shortcoming of our annotation-based approach is for applications which show high variance within a single data structure. For example, when using a hashtable where the application primarily accesses a small, dynamically determined portion of the total key-space, our static hotness profiling will fail to distinguish the hot and cold regions of this structure. Finally, although our runtime system abstracts the programming challenges of writing performance portable code for heterogeneous machines, it is still complex and puts a large onus on the programmer. Future work will be to learn from our current implementation and identify mechanisms to reduce the complexity we expose to the programmer while still making near-ideal page placement decisions.

In this work we have focused on page placement for applications assuming a static placement of pages throughout the application runtime. We recognize that temporal phasing in applications may allow further performance improvement but have chosen to focus on initial page placement rather than page migration for two reasons. First, software-based page migration is a very expensive operation. Our measurements on the Linux 3.16-rc4 kernel indicate that it is not possible to migrate pages between NUMA memory zones at a rate faster than several GB/s and with several microseconds of latency between invalidation and first re-use. While GPUs can cover several hundred nanoseconds of memory latency, microsecond latencies encountered during migration will induce high overhead stalls within the compute pipeline. Second, online page migration occurs only after some initial placement decisions have been made. Focusing on online page migration before finding an optimized initial placement policy is putting the cart before of the horse. With improved default page placement for GPU workloads, the need for dynamic page migration is reduced. Further work is needed to determine if there is significant value to justify the expense of online profiling and page-migration for GPUs beyond improved initial page allocation.

6. Conclusions

Current OS page placement policies are optimized for both homogeneous memory and latency sensitive systems. We propose a new BW-AWARE page placement policy that uses memory system information about heterogeneous memory system characteristics to place data appropriately, achieving 35% performance improvement on average over existing policies without requiring any application awareness. In future CC-NUMA systems, BW-AWARE placement improves the performance optimal capacity by better using all system resources. But some applications may wish to size their problems based on total capacity rather than performance. In such cases, we provide insight into how to optimize data placement by using the CDF of the application in combination with application annotations enabling intelligent runtime controlled page placement decisions. We propose a profile-driven application annotation scheme that enables improved placement without requiring any runtime page migration. While only the beginning of a fully automated optimization system for memory placement, we believe that the performance gap between the current best OS INTERLEAVE policy and the annotated performance (min 1%, avg 20%, max 2x) is enough that further work in this area is warranted as mobile, desktop, and HPC memory systems all move towards mixed CPU-GPU CC-NUMA heterogeneous memory systems.

Acknowledgments

The authors would like to thank the numerous anonymous reviewers and R. Govindrajan for their valuable feedback. This work was supported by US Department of Energy and NSF Grant CCF-0845157.

References

[1] T. M. Aamodt, W. W. L. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, A. Bakhoda, and

H. Jooybar. GPGPU-Sim 3.x Manual. http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual, 2014. [Online; accessed 4-December-2014].

[2] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 319–330, September 2010.

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

[4] R. A. Bheda, J. A. Poovey, J. G. Beu, and T. M. Conte. Energy Efficient Phase Change Memory Based Main Memory for Future High Performance Systems. In *International Green Computing Conference (IGCC)*, pages 1–8, July 2011.

[5] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIX Annual Technical Conference (USENIXATC)*, pages 1–15, June 2011.

[6] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Management. In *Symposium on Operating Systems Principles (SOSP)*, pages 19–31, December 1989.

[7] T. Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 1–18, September 1993.

[8] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf. Software Design Space Exploration for Exascale Combustion Co-design. In *International Supercomputing Conference (ISC)*, pages 196–212, June 2013.

[9] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer. Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access. In *International Symposium on Microarchitecture (MICRO)*, pages 13–24, December 2012.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization (IISWC)*, pages 44–54, October 2009.

[11] J. Corbet. AutoNUMA: the other approach to NUMA scheduling. http://lwn.net/Articles/488709/, 2012. [Online; accessed 29-May-2014].

[12] M. Daga, A. M. Aji, and W.-C. Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, pages 141–149, July 2011.

[13] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394, March 2013.

[14] X. Dong, Y. Xie, N. Muralimanohar, and N. Jouppi. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *International Conference on High Performance Networking and Computing (Supercomputing)*, pages 1–11, November 2010.

[15] Free Software Foundation. GNU Binutils. http://www.gnu.org/software/binutils/, 2014. [Online; accessed 5-August-2014].

[16] B. Gerofi, A.Shimada, A. Hori, T. Masamichi, and Y. Ishikawa. CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores. In *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 73–84, June 2014.

[17] M. Heroux, D. Doerfler, J. Crozier, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009.

[18] HSA Foundation. HSA Platform System Architecture Specification - Provisional 1.0. http://www.slideshare.net/hsafoundation/hsa-platform-system-architecture-specification-provisional-verl-10-ratifed, 2014. [Online; accessed 28-May-2014].

[19] Hynix Semiconductor. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0. http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf, 2009. [Online; accessed 30-Jul-2014].

[20] HyperTransport Consortium. HyperTransport 3.1 Specification. http://www.hypertransport.org/docs/twgdocs/HTC20051222-0046-0035.pdf, 2010. [Online; accessed 7-July-2014].

[21] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009. [Online; accessed 7-July-2014].

[22] Intel Corporation. Intel Xeon Processor E7-4870 . http://ark.intel.com/products/75260/Intel-Xeon-Processor-E7-8893-v2-37_5M-Cache-3_40-GHz, 2014. [Online; accessed 28-May-2014].

[23] R. Iyer, H. Wang, and L. Bhuyan. Design and Analysis of Static Memory Management Policies for CC-NUMA Multiprocessors. *Journal of Systems Architecture*, 48(1):59–80, September 2002.

[24] JEDEC. High Bandwidth Memory(HBM) DRAM - JESD235. http://www.jedec.org/standards-documents/docs/jesd235, 2013. [Online; accessed 28-May-2014].

[25] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Integrating DRAM Caches for CMP Server Platforms. *IEEE Micro*, 31(1):99–108, March 2011.

[26] J. Y. Kim. Wide IO2 (WIO2) Memory Overview. http://www.cs.utah.edu/events/thememoryforum/joon.PDF, 2014. [Online; accessed 30-Jul-2014].

[27] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, May 2008.

[28] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an Energy-efficient Main Memory Alternative. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, April 2013.

[29] R. LaRowe, Jr., C. Ellis, and M. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Transactions on Parallel Distribibuted Systems*, 3(6):686–701, November 1992.

[30] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2009.

[31] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[32] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Computer Architecture Letters*, 11 (2):61–64, July 2012.

[33] A. Minkin and O. Rubinstein. Circuit and method for prefetching data for a texture cache. US Patent 6,629,188, issued September 20, 2003.

[34] J. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 14–18, May 2009.

[35] J. Mohd-Yusof and N. Sakharnykh. Optimizing CoMD: A Molecular Dynamics Proxy Application Study. In *GPU Technology Conference (GTC)*, March 2014.

[36] NVIDIA Corporation. Unified Memory in CUDA 6. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2013. [Online; accessed 28-May-2014].

[37] NVIDIA Corporation. CUDA C Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#allocation, 2014. [Online; accessed 28-July-2014].

[38] NVIDIA Corporation. NVIDIA Launches World's First High-Speed GPU Interconnect, Helping Pave the Way to Exascale Computing. http://nvidianews.nvidia.com/News/NVIDIA-Launches-World-s-First-High-Speed-GPU-Interconnect-Helping-Pave-the-Way-to-Exascale-Computin-ad6.aspx, 2014. [Online; accessed 28-May-2014].

[39] NVIDIA Corporation. Compute Unified Device Architecture. https://developer.nvidia.com/cuda-zone, 2014. [Online; accessed 28-May-2014].

[40] M. Pavlovic, N. Puzovic, and A. Ramirez. Data Placement in HPC Architectures with Heterogeneous Off-chip Memory. In *International Conference on Computer Design (ICCD)*, pages 193–200, October 2013.

[41] S. Phadke and S. Narayanasamy. MLP-Aware Heterogeneous Memory System. In *Design, Automation & Test in Europe (DATE)*, pages 1–6, March 2011.

[42] L. Ramos, E. Gorbatov, and R. Bianchini. Page Placement in Hybrid Memory Systems. In *International Conference on Supercomputing (ICS)*, pages 85–99, June 2011.

[43] J. Sim, G. Loh, H. Kim, M. O'Connor, and M. Thottethodi. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *International Symposium on Microarchitecture (MICRO)*, pages 247–257, December 2012.

[44] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, v.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012.

[45] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors. In *European Conference on Computer Systems (EuroSys)*, pages 47–58, March 2007.

[46] J. Tramm, A. Siegel, T. Islam, and M. Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, September 2014.

[47] J. Tuck, L. Ceze, and J. Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *International Symposium on Microarchitecture (MICRO)*, pages 409–422, December 2006.

[48] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 279–289, September 1996.

[49] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 93–103, September 2013.

[50] K. Wilson and B. Aglietti. Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C. In *International Conference on High Performance Networking and Computing (Supercomputing)*, pages 33–35, November 2001.

[51] J. Zhao, G. Sun, G. Loh, and Y. Xie. Energy-efficient GPU Design with Reconfigurable In-package Graphics Memory. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 403–408, July 2012.

[52] J. Zhao, G. Sun, G. Loh, and Y. Xie. Optimizing GPU Energy Efficiency with 3D Die-stacking Graphics Memory and Reconfigurable Memory Interface. *ACM Transactions on Architecture and Code Optimization*, 10(4):24:1–24:25, December 2013.

[53] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 129–142, March 2010.