

Intelligent Data Placement on Discrete GPU Nodes with Unified Memory

Tanzima Sultana
 tanzima_sultana@txstate.edu
 Texas State University
 San Marcos, Texas

Blake Allen*
 blake.m.allen@l3harris.com
 L3 Harris Technologies
 Greenville, Texas

Apan Qasem
 apan@txstate.edu
 Texas State University
 San Marcos, Texas

ABSTRACT

With increasing heterogeneity, the importance of data organization within a compute node has grown immensely. Recently, industry vendors have introduced technology that can present a unified shared address space for multiple physical pools of memory. In this paper, we leverage unified memory technology and characterize the performance trade-offs of host and device placement across a range of hybrid application design patterns. We perform a Roofline analysis to establish fundamental performance bounds in collaborative applications and then develop an analytical model that makes profitable placement decisions at the individual data structure level. We integrate the placement model into a runtime system and enable transparent data placement in CUDA/C++ applications. Preliminary experiments yield the following results: (i) placement policies have significant performance impact across hybrid application design paradigms (ii) placement decisions are impacted by the sparsity of data access, page re-migration, amount of latency hiding opportunities and design-specific attributes such as the number of pipeline stages, and (iii) intelligent data placement can improve node performance by up to 5× on applications with sparse access patterns.

CCS CONCEPTS

- General and reference → Performance; • Software and its engineering → Runtime environments.

KEYWORDS

heterogeneous computing, unified memory, data placement, performance modeling, Roofline analysis

ACM Reference Format:

Tanzima Sultana, Blake Allen, and Apan Qasem. 2020. Intelligent Data Placement on Discrete GPU Nodes with Unified Memory. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3410463.3414651>

*the work was done while author was an undergraduate at Texas State University

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.

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
 ACM ISBN 978-1-4503-8075-1/20/10...\$15.00
<https://doi.org/10.1145/3410463.3414651>

1 INTRODUCTION

HPC vendors have recently introduced technology that presents a unified view of multiple physical pools of memory contained within a compute node [4, 18, 45]. Unified Memory (UM)¹ systems offer an obvious programmability benefit. Pointers can be freely used between different memory regions, relieving developers from the burden of explicitly managing data between host and device. Programmability benefits are particularly compelling for collaborative design patterns in which a shared data structure is accessed by both the CPU and GPU and for applications that oversubscribe GPU memory.

The performance benefits of UM are less obvious. Accessing host-resident data via demand paging can be expensive. Data is fetched over a high-latency, low-bandwidth channel during kernel execution and page migrations incur additional overhead due to fault handling. Given that on current systems, GPU memory bandwidth is an order-of-magnitude higher than that of the CPU-GPU interconnect [33], a device-only placement policy would appear to be the natural choice when programmability is not a concern. Notwithstanding, placement decisions are more nuanced in practice for several reasons. First, the massively multi-threaded GPU kernels can potentially hide some fraction of the latencies and mitigate the costs associated with host placement. Second, with demand paging, only the data requested by the GPU is transferred from host memory. As a result, for applications that exhibit sparse or irregular data access patterns, the actual volume of traffic over the interconnect can be a lot smaller with host placement than with device placement. Third, for GPU oversubscribed applications, the need to strip-mine the code is eliminated when data is placed in host memory. This can provide an indirect benefit by reducing the overhead of repeated kernel launches. The above issues give rise to an intriguing performance challenge with respect to data placement: *when is it profitable to place data in host memory?*

Although there has been extensive work in data placement on NUMA and other classes of emerging hybrid memory systems [5, 13, 15–17, 41, 42, 47, 50, 53, 55], the state-of-the-art, as yet, has not addressed the specific performance challenges described above. As we discuss in § 2.1, there are subtle, yet significant, differences in data access methods on a heterogeneous node with a discrete GPU than what is observed in other hybrid memory systems. These differences dictate that a data placement policy factor in a new set of attributes both at the application and system layers than what has been considered in prior work. Furthermore, the recently proposed collaborative design paradigms, which emphasize tight

¹here, this term refers to Nvidia's UM, IBM's CAPI, and AMD's ROCm technologies, all of which follow the same underlying principles

coupling of CPU-GPU tasks [22, 30, 46], must also be taken into consideration in making placement decisions.

This paper takes an in-depth look at the performance trade-offs in data placement on a CPU-GPU compute node with UM support. First, we construct a Roofline model [51] to characterize the inherent performance limitations of applications running on such nodes. We derive new performance ceilings in the Roofline that factor in the interconnect bandwidth, cost of data copy, access sparsity and data migration under demand paging. This augmented Roofline uncovers performance cross-over points for host and device-placed data as a function of operational intensity.

Next, we build on the insight gained from the Roofline to develop an analytical model for selecting individual data structures for host placement. We show that placement decisions are impacted by a range of architecture and system parameters, program characteristics, and at a high level, the choice of the application design paradigm. The model synthesizes the inter-related factors at different layers and determines the suitability of host placement using a single metric called the *host access penalty*. This metric represents the net cost of fetching a data structure via demand paging as opposed to bulk-copy. In addition, in cases where a portion of the data must be placed in host memory to avoid GPU oversubscription, *host access penalty* can be used to evaluate the relative merits of host placement between two candidate data structures. The required information to estimate *host access penalty* is collected in two phases (i) system-level attributes such as peak effective bandwidth and average page fault service cost are gathered at install-time via hardware performance counter-based microbenchmarking and (ii) program characteristics such as the size of a data structure and kernel launch bounds are extracted via runtime analysis. Both sets of parameters are passed into the model at runtime which returns a binary decision on placement for each data structure in the application. The model has been integrated into a runtime system to enable transparent data placement in CUDA/C++ applications. The main contributions are summarized below.

- we develop a new Roofline model for hybrid memory systems that characterizes fundamental performance limitations under host and device placement
- we formalize the notion of page re-migration and establish (i) page re-migration (ii) access sparsity and (iii) latency hiding opportunities as three primary considerations for reasoning about data placement; we describe methods to quantify their effects on data placement
- we show that contemporary hybrid applications exhibit characteristics that make certain data structures amenable to host placement, yielding 1.85 speedup on average on a suite of 14 applications and up to 5× on codes with sparse data access patterns

2 BACKGROUND

2.1 Hybrid Memory Systems

Recent innovations in memory technology [29, 32, 36] have given rise to hybrid memory systems (HMS), which combine multiple memory technologies with varying characteristics into a single unified system. A typical HMS consists of a *fast* and a *slow* memory component. The *fast* component has either lower latency or higher

bandwidth or both, while the *slow* memory has much larger capacity. For instance, Intel's Knight Landing combines an off-package DRAM with an on-package high-bandwidth memory [27].

UM was first introduced in CUDA 6.0 [38] but hardware support for fault handling and migration was added more recently with the release of Pascal and Volta [39, 40]. UM allows the previously discrete memory units in a CPU-GPU node to be treated as a single hybrid memory system (HMS-DGPU). In a UM system, a GPU page fault occurs when a request is issued to data that is not resident in device memory. On a page fault, the UM driver allocates new pages on the GPU, requests the corresponding pages from CPU memory and the pages are migrated from host to device memory. During page fault handling, the GPU TLBs are locked to ensure each SM's view of memory is consistent. Typically, page faults are serviced in *groups*. When the GPU generates multiple page faults concurrently the UM driver removes duplicates, coalesces the requests and then transfers the data for all requests simultaneously. Once data is migrated, the CPU pages are freed and CPU page faults occur for subsequent requests to migrated data.

Although the underlying principles of a conventional HMS apply to HMS-DGPU, there are also significant differences.

(i) On HMS-DGPU, the GPU must fetch all data from the *slow* memory over the CPU-GPU interconnect (e.g., PCIe, NVlink). The interconnect bandwidth is typically an order-of-magnitude lower than DRAM's. Therefore, the access cost of the *slow* memory is dominated by the interconnect rather than the memory unit itself.

(ii) Unlike HMS, input data placed in *fast* memory needs to be explicitly copied from *slow* memory before it can be accessed by the GPU. The cost of this copy must be factored in when calculating the benefits of placing data in *fast* memory.

(iii) On conventional HMS, both the *fast* and *slow* memory are accessible by all compute engines and all compute engines enjoy the same bandwidth and latency to the *fast* (and *slow* memory). This is not the case for HMS-DGPU. Device-mapped data structures must be accessed over the PCIe by the CPU. Thus, from the CPU perspective, the roles of the memory units are reversed. Therefore, when considering overall performance of hybrid applications, as we do in this paper, one must consider not only the trade-offs of *fast* vs *slow* placement on the GPU side but also the implications of the data being accessed on the CPU side.

2.2 Collaborative Design Patterns

The emergence of unified memory has paved the way for collaborative, tightly-coupled application design patterns [22, 30, 46]. The choice of the design pattern can have a major impact on placement decisions. In this work, we consider the performance implications of data placement in seven collaborative design paradigms: CPU-to-GPU (C2G), CPU-to-GPU Iterative (C2GI), CPU-and-GPU Iterative (C&GI), Workload Partition (WP), CPU-Producer-GPU-Consumer (CPGC), GPU-Producer-CPU-Consumer (GPCC) and CPU-GPU Pipelined (CGP)².

Fig. 1 shows how CPU-GPU data access patterns can vary across different paradigms. Clearly, data transfer between CPU-GPU is heavily influenced by application design parameters. For iterative applications the same data structure may be copied many times

²terminology is taken from [46]

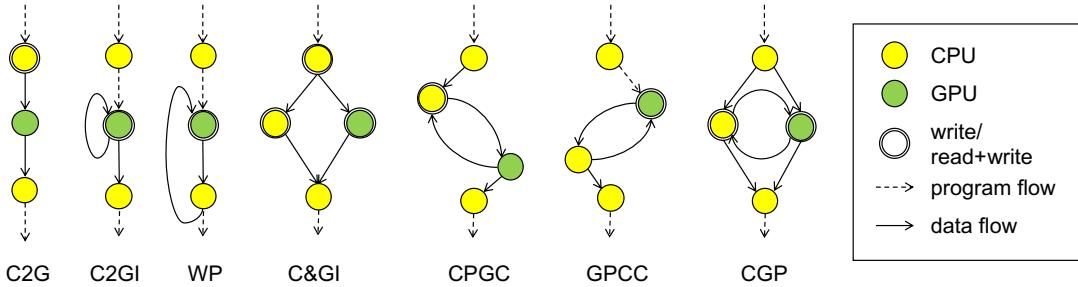


Figure 1: Data access patterns across collaborative design paradigms. Access patterns are shown for a single data structure from an application written in the paradigm. Variations exist with the same paradigm.

between the CPU and GPU. The number of copies depends on the iteration count and the length of a pipeline stage which may only be determined at runtime. Further, the actual volume of data transferred may depend on placement. For example, under host placement for a sparsely accessed data structure only the data requested by the kernel will be transferred while under device placement the data structure will be copied in its entirety. For C&GI applications, typically a small amount of data (e.g., a scalar) is copied back to the CPU after each kernel execution. This transfer can incur a high copy overhead with device placement. Conversely, with host placement this pattern can give rise to page thrashing behavior. For pipelined patterns the same data structure may be accessed concurrently or in close temporal proximity. With host placement, this has implications with respect to CPU-GPU synchronization between kernel invocations. Finally, iterative patterns are often chosen to avoid GPU oversubscription. Under host placement, the oversubscription problem is eliminated. This implies that certain iterative applications can be executed in a non-iterative mode or at least with fewer iterations if a subset of the data is placed in host memory.

2.3 Definitions and Terminology

This section introduces definitions and terminology used in the remainder of the paper. Operational intensity and effective bandwidth are not new concepts but we clarify their use in this context.

Host and Device Memory refer to the CPU and GPU memories, respectively. A *host-placed* data structure is resident in host memory at the time of kernel launch and is transferred to device memory via demand paging. *Device-placed* data is bulk-copied to device memory prior to kernel launch and is copied back to host memory (if touched by CPU) after kernel execution.

Operational intensity (α) ratio of arithmetic operations to the number of bytes accessed from memory. Memory is device memory unless otherwise noted. The definition includes both integer and floating-point operations and hence we use the term *operational* instead of *arithmetic* intensity.

Effective bandwidth (β^e) maximum attainable sustained bandwidth. $\beta^e \leq$ pin bandwidth. β^e can be reasonably estimated using a STREAM-like benchmark [14, 37].

Required bandwidth (β^r) the rate at which data needs to be transferred from memory for a kernel to achieve peak performance. β^r is inversely related to α .

Copy-to-computation ratio (γ) is computed as the ratio of data copy time (t_{cp}) to the kernel execution time (t). Intuitively, γ represents the cost to the application due to bulk-copying of data from host memory. For instance, if $t = 10$ and $t_{cp} = 2$ then copy-adjusted execution time, $t' = 12$ and $\gamma = 0.2$ which implies a 20% slowdown.

Access sparsity (σ) denotes the fraction of data that is explicitly requested by the compute elements. If every element in the data structure is touched at least once (i.e., dense access) then $\sigma = 1$. The lower the value of σ the higher the access sparsity. Data reuse is not factored into our calculation of sparsity. If the same element is accessed many times but not all elements are requested from the data structure then sparsity is still < 1 .

Page re-migration rate (ρ) A page may need to be migrated more than once if it is evicted from memory before its reuse. In the traditional C2G paradigm, re-migration only occurs from CPU to GPU. But in collaborative applications, re-migration can be bi-directional. For a given data structure, the page re-migration rate, ρ is measured as the ratio of the total number of page migrations to the number of distinct pages requested by the compute elements. Each requested page is migrated at least once. Hence, ρ is always ≥ 1 .

3 ROOFLINE ANALYSIS

Existing Roofline models [7, 11, 51] are not readily applicable to hybrid applications running on HMS-DGPU. In particular, they do not account for (i) host-to-device interconnect bandwidth (ii) cost of data copy (iii) access sparsity and (iv) data migration effects. We discuss the need for incorporating these factors and show the derivation of new performance ceilings for each.

3.1 Interconnect Bandwidth

The maximum sustained BW on host-placed data is bounded above by β_C^e , the effective BW of the communication channel between host and device. Fig. 2, which plots β^r as a function of α , identifies the region where this performance bound comes into play and host vs device placement decisions become consequential. To the right of the zone of interest, β^r is low enough to be met by both host and device memory. To the left of the zone, neither host or device memory can sustain the BW required to achieve peak performance. For an application with an α that falls outside the zone of interest, performance bounds can be modeled using canonical Roofline. A performance bound for host-placed data within the zone of interest

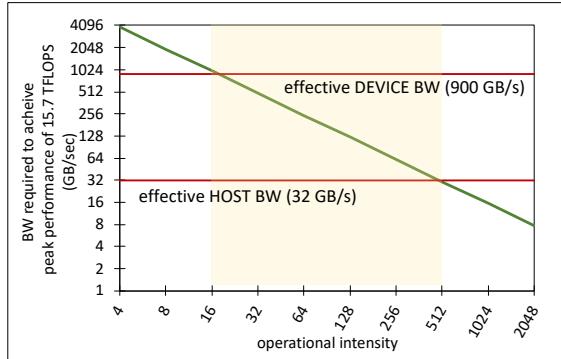


Figure 2: Effective bandwidth and its relationship to attainable peak performance [Nvidia V100 + PCIe Gen4]

can be obtained as follows

$$P_H = \min(\alpha \times \beta_C^e, P) \quad (1)$$

where P is the maximum performance obtained via canonical Roofline and $\alpha \times \beta_C^e < P$ represents the zone of interest.

3.2 Data Copy

The copy cost is a one-time cost that is incurred *before* the computation takes place on the device. It is independent of the rate at which data is being requested by the kernel. To estimate the impact of the data copy cost, we introduce in our model the notions of copy-to-computation ratio (γ) and access sparsity (σ) and make the following observations.

1. *For a given operational intensity and a fixed access sparsity level, copy-to-computation ratio remains constant, irrespective of the total volume of data copied.*

Explanation Intuitively, if the data set size increases, then to maintain the same level of intensity there needs to be a proportional increase in number of operations performed by the kernel. Thus, the ratio of copy time to execution time will remain same for all data set sizes.

Let A and B be two kernels with $\alpha = i$ and $\sigma = k$. Let the volume of data copied by A and B be b and $m \times b$, respectively. Let $\gamma_A = t_{cp}/t_A$. Then $\gamma_B = (m \times t_{cp})/t_B$. We need to show that $\gamma_A = \gamma_B$ or $t_B = m \times t_A$.

Let n_A and n_B be the total operations executed by A and B , respectively. Then, we have, $\alpha = n_A/(k \times b) = n_B/(k \times m \times b)$ or, $n_A \times m = n_B$. From the canonical Roofline, we get maximum performance $P = P_A = P_B$. Therefore, $n_A/t_A = n_B/t_B = (n_A \times m)/t_B$ or $t_B = m \times t_A$. ■

2. *For a fixed access sparsity level, copy-to-computation ratio remains constant for different operational intensity levels until the stage when performance is no longer bounded by device memory bandwidth; beyond this point copy-to-computation ratio will increase proportionally.*

Explanation [*BW bound*] When the performance of a kernel is BW bound, its performance will increase proportionally with increases in α . That is, if at $\alpha = i$, maximum performance is P then at $\alpha = a \times i$, performance is $a \times P$. Intuitively, since the kernel is BW bound, it is able to perform additional operations, in parallel,

without increasing the overall execution time, thus achieving a proportional increase in performance.

Let A and B be two kernels with $\alpha_A = i$ and $\alpha_B = a \times i$, respectively. We need to show that $P_B/P_A = \alpha_B/\alpha_A = a$.

In the BW bound region, the maximum attained BW by a kernel is fixed at β_C^e . Therefore, from the Roofline we get, achieved BW, $\beta^a = P_A/\alpha_A = P_B/\alpha_B$. Then $P_B/P_A = \alpha_B/\alpha_A = a$. Now, from the above, we can show that in the region where performance is BW bound, $\gamma_A = \gamma_B$. We consider two cases.

Case 1: α increases due to an increase in the number of operations only. If number of operations for A is n then number of operations in B is $a \times n$. Then $P_A = n/t_A$ and $P_B = (a \times n)/t_B$. Therefore, we get $(a \times n/t_A)/(n/t_A) = a$ or $a \times t_A/t_B = a$ or $t_A/t_B = 1$ or $t_A = t_B$. Since the increase is only in number of operations and σ is fixed, the amount of data copied is the same for both A and B . Therefore, $\gamma_A = t_{cp}/t_A = t_{cp}/t_B = \gamma_B$. ■

Case 2: α increases due to a decrease in the amount of data transferred. If amount of data transferred in A is b then amount of data transferred in B is b/a . Number of operations in A and B is n . Then $P_A = n/t_A$ and $P_B = n/t_B$. Therefore, we get $n/(t_B)/(n/t_A) = a$ or $t_A/t_B = a$ or $t_A = a \times t_B$. Let, copy time in A be t_{cp} . Then copy time in B is t_{cp}/a . Then, $\gamma_A = t_{cp}/t_A = (t_{cp}/a)/(a \times t_B) = t_{cp}/t_B = \gamma_B$. ■

[*Non BW bound*] When the kernel becomes non BW bound (meets the eave of the roof), performance is capped at peak effective computation throughput. At this stage, increasing operational intensity will increase execution time but not copy time, thus increasing γ . Again, we consider two cases for increases to α , as in the BW-bound region.

Case 1: If number of operations for A is n then number of operations in B is $a \times n$. Then $P = n/t_A = a \times n/t_B$. Therefore, $t_B/t_A = a$ or $t_B = a \times t_A$. Since the increase is only in number of operations and σ is fixed then volume of data copied is the same for both A and B . Then $\gamma_A = t_{cp}/t_A$ and $\gamma_B = t_{cp}/t_B = a \times t_{cp}/t_A = a \times \gamma_A$, showing a proportional increase in γ . ■

Case 2: If amount of data transferred for A is b then amount of data transferred in B is b/a . Number of operations in A and B is n . Then $P = n/t_A = n/t_B$ or $t_A = t_B$. Let copy time in A be t_{cp} . Then copy time in B is t_{cp}/a . Then, $\gamma_B = (t_{cp}/a)/t_B = (t_{cp}/a)/t_A = a \times \gamma_A$. ■

We can now derive a new performance ceiling that accounts for data copy under device placement. Let P be the maximum performance from the canonical Roofline with $\alpha = i$. Let B be size of data copied from host to device. Then the total number of operations, $Q = i \times B$. Let t be the execution time in seconds at P . Then $t = Q/P$. Let t_u = time to copy one byte of data from host to device memory. Then copy time, $t_{cp} = t_u \times B$. Therefore, copy-adjusted performance ceiling under device placement is given by

$$P_D = Q/(t + t_{cp}) \quad (2)$$

3.3 Access Sparsity

Under demand paging, data movement over the CPU-GPU interconnect is sensitive to the size of the data structure and sparsity of access. If a host-placed data structure is sparsely accessed then, ignoring prefetching effects, only the fraction of data that is explicitly requested is transferred from host to device. By contrast, access sparsity plays no role in determining the volume of data

transferred in the copy-then-execute model. Since bulk-copy takes place prior to kernel launch, the entire data structure is copied irrespective of the actual data access pattern. Therefore, everything else being equal, access sparsity puts device placement at a relative disadvantage due to the increased traffic over the interconnect. The higher the sparsity the bigger the disadvantage. This leads us to the following observation.

For a given operational intensity, increased access sparsity will lower the performance ceiling under device placement

Explanation This is a direct consequence of the cost incurred due to copying additional data at higher sparsity levels. Let A be a kernel with operational intensity $\alpha = Q/M$, where Q is the number of operations and M is the total bytes transferred from device memory. Let, σ_1 and σ_2 be two distinct sparsity levels for the kernel with $\sigma_1 < \sigma_2 \leq 1$. We need to show that $P_{\sigma_2} > P_{\sigma_1}$, where P_{σ_1} and P_{σ_2} are performance ceilings at sparsity levels σ_1 and σ_2 .

According to our definition of access sparsity, the volume of data copied at σ_1 and σ_2 is b/σ_1 and b/σ_2 for some b. Let t_{cp} be the time to copy b bytes. Then the copy times at σ_1 and σ_2 are t_{cp}/σ_1 and t_{cp}/σ_2 with $t_{cp}/\sigma_1 > t_{cp}/\sigma_2$. Let P be the performance ceiling derived from the canonical Roofline. Then kernel execution time, $t = Q/P$. Then $P_{\sigma_1} = Q/(t + t_{cp}/\sigma_1)$ and $P_{\sigma_2} = Q/(t + t_{cp}/\sigma_2)$ and $P_{\sigma_2} > P_{\sigma_1}$.

Now, from (2), we can derive a new performance ceiling for device placement at sparsity level σ

$$P'_D = Q/(t + t_{cp}/\sigma) \quad (3)$$

Access sparsity does not impact the performance ceiling of host placement. Even with high degrees of sparsity performance is still bounded by 1.

3.4 Page Re-migration

Under an ideal page migration scheme, each page is migrated once from host to device on first access and any subsequent requests to the page are serviced in device memory. Pages only need to be re-migrated if the application oversubscribes GPU memory and the oversubscription has not been dealt with via strip-mining. In practice, however, pages may be evicted from device memory and then re-migrated more frequently because of the implemented migration policy and the page table limitations of the driver. Re-migration can also occur in collaborative paradigms such as C&GI and CGP, in which a data structure may be repeatedly accessed in concurrent or nearby phases by the CPU and GPU. In the extreme case this can cause a thrashing behavior where the same page is migrated back-and-forth between host and device many times. The page re-migration rate (ρ) captures these negative migration effects under host placement. $\rho > 1$ indicates that some pages are being migrated more than once and a higher value of ρ indicates a higher rate of eviction in device memory. We can use the page re-migration rate to make the following claim about the performance bounds under host placement.

For a given operational intensity, increased page re-migration rate will lower the performance ceiling under host placement

Explanation Under host placement, the effective operational intensity is the ratio of the number of operations and the number of bytes transferred from host memory rather than device memory. A higher value of ρ indicates increased traffic over the interconnect

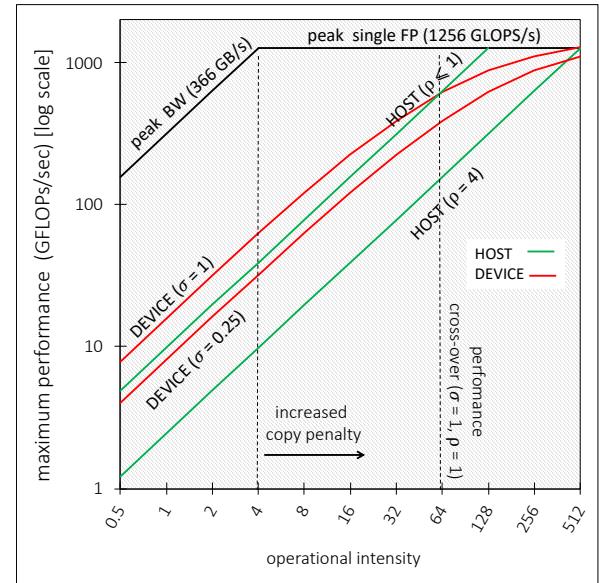


Figure 3: Augmented Roofline with new performance ceilings [Nvidia P100 + PCIe Gen3]

for the same number of operations performed by the kernel which reduces the effective operational intensity. Let α be the operational intensity of the kernel under perfect page migration. Then effective operational intensity is α/ρ . Let P_H be the performance bound obtained from 1 at operational intensity α . Then the performance bound, accounting for page evictions, is simply P_H/ρ which is $< P_H$ for any $\rho > 1$.

The performance bound with page migration effects can be derived as follows

$$P'_H = \alpha/\rho \times \beta_C^e \quad (4)$$

Fig 3 places the derived performance ceilings on the canonical Roofline plot for Nvidia P100 with a PCIe Gen3 interconnect. In this Roofline, we use the effective peak rather than theoretical peak as it is a more realistic reflection of the actual performance bounds. Effective peak single-precision FP performance is obtained from the SHOC benchmark [12]. Effective peak bandwidth is obtained via BabelStream [14]. Copy times are estimated at effective peak bandwidth. The penalty associated with data copy begins to increase at $\alpha = 2$ and performance difference between host and device placement starts to narrow (2). A cross-over point is observed at $\alpha = 64$ under dense access ($\sigma = 1$) and ideal page migration ($\rho = 1$). The Roofline plot illustrates the significant impact that access sparsity and page migration can have on the relative performance of host and device placement. When $\sigma \leq 0.25$, host placement achieves higher performance ceilings for all operational intensity. On the other hand, if pages are evicted from device memory at a rapid rate (e.g., $\rho \geq 4$) then device placement with dense access outperforms host for all α .

4 A COST MODEL FOR DATA PLACEMENT

Although insightful, the Roofline cannot be used directly to make placement decisions for a given application. We build on the Roofline

to develop a policy for data placement under UM. For each data structure touched by at least one GPU kernel, the model quantifies the net penalty for accessing that data structure via demand paging as opposed to bulk-copy. We label this quantity as host access penalty (ϕ). ϕ is determined at runtime and data structures with $\phi < 0$ are selected for host placement. To estimate ϕ , we require both system-level and program-level information. System-level attributes that do not depend on program characteristics (e.g., average cost of servicing a page fault) are collected ahead of time via microbenchmarking and profiling and hard-coded into the model per target platform (§ 5.2). Program-level attributes are fed into the model at runtime. We begin with an estimate of the volume of traffic over the interconnect. We then calculate the exposed latency based on this estimated volume of data migration. We factor in costs associated with page fault handling and the effects of design parameters and finally, compare this cost with the cost of data copy to determine the net host access penalty. The model is designed to be conservative in that it will err on the side of overestimating the host access penalty.

4.1 Data Movement

To determine ϕ , we first need to determine the volume of data that will be transferred over the interconnect under host placement. As discussed in § 3, the expected volume of data transfer is primarily impacted by two factors, access sparsity and page re-migration. For a given data structure, access sparsity will reduce the volume of traffic over the interconnect while re-migration will increase it. Let d be a data structure of size d then the volume of data transferred can be estimated as

$$V_d = \text{size}_d \times \sigma \times \rho$$

We analyze the data access patterns in regular data structures to calculate the expected sparsity. Access sparsity cannot be estimated statically for irregular data structures as it heavily depends on the program input. We adopt the techniques proposed in [3] to get a reasonable estimate of sparsity based on the structure of the input graph. We get an estimate of the page re-migration rate based on the (i) size of the data structure (ii) amount of page-level data reuse (iii) concurrent or consecutive phase access by CPU and GPU, and (iv) device memory capacity.

4.2 Latency Hiding

We estimate total latency for accessing a host-placed data structure from its required bandwidth. We then factor in available concurrency and hardware prefetching effects to determine the exposed latency.

Total Latency. For a host-allocated data structure d , memory latency includes the time to transfer the data item from host memory. A conservative estimate of total latency for accessing d can be obtained from β_{SM}^r , the required bandwidth at each SM. β_{SM}^r is calculated from number of memory instructions issued per cycle and the average size of a request. If the combined required bandwidth from all SMs is less than β_C^e then d is transferred at the requested rate, at β_C^e otherwise. Now, assuming uniform access to d from all SMs, the volume of data requested at each SM is V_d/SMs . Since memory latency across SMs are overlapped, the total latency for

the kernel with respect to d is the observed latency at each SM. From the above, we can estimate total latency for d as follows

$$L = \min(\beta_{SM}^r, \beta_C^e) \times V_d/SMs$$

Concurrency. To quantify latency hiding opportunities in a kernel, we estimate the available concurrency as a function of operational intensity (α) and occupancy (\cdot). Increased α implies that more instructions are available to hide the latency of each memory operation. Let, $|w|$ be the size of a warp. Then for a given kernel, If $\alpha = 1/|w|$, then the kernel is performing one arithmetic operation for every memory request (without factoring in coalescing). Furthermore, a $1/|w|$ increase in α implies that in every cycle, there is one additional instruction available for scheduling from the same warp.

Achieved occupancy, which is defined as the ratio of the average number of active warps to the maximum number of warps supported by the system per SM, can also increase opportunities for latency hiding. A higher number of active warps implies availability of a larger pool of instructions per issue cycle, indicating higher tolerance for longer latency memory operations. An active warp is one that has at least one instruction available for issue on a given cycle. If a system supports a maximum of W active warps then on average at least $W \times \cdot$ instructions are available for issue in each cycle. From this, we derive the following estimate for available concurrency

$$C = \lfloor (\alpha - 1/|w|)/(1/|w|) \rfloor + W/SM \times \cdot$$

Now, let l be the average memory latency created in each cycle in each SM, as derived from β_{SM}^r and the average size of a request. Then the fraction of latency hidden is given by (C/l) and total hidden latency due to concurrency,

$$H_C = (C/l) \times L$$

Prefetching. Latency of successfully prefetched pages are not exposed to the kernel. The Page Migration Engine (PME) in the Nvidia UVM driver³, implements a tree-based prefetcher that employs the following algorithm: prefetching is triggered when a strided pattern is detected in GPU page faults, emitted from all thread blocks on a single SM. Once activated, the number of pages migrated on each subsequent fault is doubled with a maximum of 256 pages prefetched on a single fault. The prefetcher is reset to prefetch a single page when the upper bound is reached. For regular data structures, with strided and dense access pattern, we estimate the volume of data prefetched to device memory based on the above heuristic. We assume no prefetching on irregular data structures. From this we estimate the latency hidden by prefetching as

$$H_{pre} = \text{prefetch_hidden_latency}(V_d, \beta_{SM}^r, \text{PME})$$

Then exposed memory latency in d is given by

$$\text{exposed_latency} = L - H_C - H_{pre} \quad (5)$$

4.3 Fault Handling and System Effects

To account for GPU page faults, we need to determine the number of page faults incurred by the data structure and the cost of servicing each fault. We first get a baseline estimate for the number of pagefaults by dividing data structure size by GPU page size. The

³available under MIT open-source license

actual number of observed page faults, however, is influenced by many factors [58]. We consider three of these

(i) *Fault concurrency*: Requests to distinct pages from warps running on the same SM are coalesced and serviced together. This is known as fault concurrency. We estimate fault concurrency by counting the number of distinct pages accessed by the thread blocks scheduled on the same SM and then dividing by the coalescing unit.

(ii) *Prefetching*: Prefetching not only hides the latency of host-accessed pages but it also reduces the number of page faults incurred as prefetched pages do not incur a fault. We calculate the effect of prefetching using the method described in (§ 4.2) but in this case we estimate the number of pagefaults avoided due to prefetching.

(iii) *Page Trashing*: In collaborative applications, when the same data structure is repeatedly accessed (read + write) by the CPU and GPU in concurrent or nearby phases, a page thrashing behavior may be observed. We identify such data structures by analyzing data access patterns and application phases. Since thrashing has a severe negative impact on performance and dominates other parameters, we consider it as a binary choice. Any data structure suspected of thrashing is eliminated from consideration for host placement.

The cost associated with servicing GPU page faults for a host-allocated data structure d is captured as follows

$$\text{fault_penalty} = ((V_d/\text{page_size}) - \text{fault_concurrency} - \text{prefetch}) \times \text{service_cost} + \text{thrashing} \quad (6)$$

4.4 Design Paradigm Considerations

As discussed in § 2.2, the choice of the application design paradigm and the parameters thereof can have a profound impact on placement choices. We consider two of these in our model.

First-touch. Access patterns to a shared data structure, d can be classified as (i) **host-first**: d is touched by a CPU thread and then by device kernel (typically input data) and (ii) **device-first**: d is touched by device kernel and then by CPU thread (typically, output data). The UVM driver employs a first-touch allocation policy where pages are allocated at the time of first-access, not at allocation. As a result, host-placed data structures with device-first access incur page faults both on the CPU and GPU sides when they are fetched via demand paging. Although CPU pages faults count towards the overall execution time no matter *when* they occur, the application pays a heavier penalty if they occur at the time of request from the GPU. A kernel with a higher occupancy (under device placement) will pay a higher penalty due to the additional stalls introduced because of the CPU page faults. We use microbenchmarking to derive a cost/byte estimate of this penalty as a function of kernel occupancy. The first-touch penalty for a host-placed, device-first data structure is then calculated as

$$\text{first_touch_penalty} = (\text{CPU_fault_penalty}(')) \times V_d \quad (7)$$

Oversubscription and Strip Mining. Iterative design patterns address the GPU oversubscription problem by strip mining the application. Placing a data structure in host memory can reduce the required number of strips or, if it is large enough, completely eliminate the need for strip mining. Reducing the strip count reduces the number of required kernel invocation. Since the kernel launch overhead can go up to microseconds on current systems [25], even a few saved invocations can have a substantial impact on performance.

To account for the savings in kernel launch overhead, we calculate the number of iterations saved due to host-placement of d . If d is larger than device memory then placing it in host memory implies that the amount of data that would have been bulk-copied is reduced by at least the size of device memory. This means that the number of required kernel invocations is reduced by at least one. In the general case, the number of saved iterations is given by $\lfloor \text{size}_d / \text{dev_mem_cap} \rfloor$. We then derive the average cost of a kernel launch via microbenchmarking and obtain the savings in kernel launch overhead as

$$\text{kernel_launch_savings} = \lfloor \text{size}_d / \text{dev_mem_cap} \rfloor \times \text{kernel_launch_overhead} \quad (8)$$

4.5 Data Copy Cost

Quantifying the cost of bulk-copy is less complicated as it does not depend on the data access patterns in the kernel. A baseline estimate of data copy cost can be derived from $\beta_C^e \times \text{size}_d$. We then adjust this baseline as follows.

(i) *Asynchronous copy*: In iterative applications, asynchronous copies are used to overlap host-to-device data transfer with kernel execution. A reasonable estimate of these savings can be derived from the number of asynch copy calls and the size of data transfer in each call.

(ii) *Copy call overhead*: The copy call overhead is incurred *per call*, irrespective of the volume of data transferred. Thus, cost/byte is higher for smaller buffers as they incur the same copy overhead of a larger buffer.

The total cost of copy is then estimated as

$$\text{copy_cost} = (\beta_C^e \times \text{size}_d) - \text{asynch_savings} + \text{copy_call_overhead} \quad (9)$$

Now, from (5)-(9), we can derive an estimate of ϕ as

$$\phi = \text{exposed_latency} + \text{fault_penalty} + \text{first_touch_penalty} - \text{kernel_launch_savings} - \text{data_copy_cost} \quad (10)$$

5 IMPLEMENTATION

The implementation of the placement strategy consists of a source-to-source transformation tool, a hardware performance counter based profiler and a runtime system.

5.1 Source-to-source Transformation

Each application goes through a sequence of three source-level transformations.

(i) *Parameterize placement of each data structure*: For each data structure accessed by a GPU kernel, we identify the allocation and de-allocation sites. Each site is bracketed with a placement-dependent clause and appropriate calls to the host allocator and de-allocator are inserted in the host path. We wrap each copy site with a condition such that copy calls are only invoked when the data structure is allocated to device memory. Alternate launch configurations are added to the kernel launch site. For a kernel with n arguments, $2^n - 1$ alternate configurations are added.

(ii) *Insert calls to the runtime system*: The parameterized source is instrumented with calls to the runtime system. A single call is inserted for each data structure to obtain its placement location.

These calls are inserted at a point where the size of the data structure and the launch bounds of the kernel that accesses it can be determined.

(iii) *Re-structure application design pattern:* This step inserts alternate configurations for iterative applications. Only one configuration is selected at runtime.

5.2 Micro-benchmarking and Profiling

Since current profilers do not support the probing of Unified Memory events on a *per data-structure* basis, we developed our own profiling tool, Reverse Page Fault Mapper (RPFM), for this purpose. RPFM monitors GPU page fault and data migration activity via hardware performance counters and then correlates the events back to source-level data structures. RPFM (i) instruments the CUDA source to gather address range information for each data structure (ii) profiles the instrumented code to create a trace of UM events and (iii) analyzes the collected profiles to map UM events to data structures. First, a CUDA source file is instrumented to extract the starting address and size of each data structure. The instrumented code is then profiled to collect a trace and the trace along with data structure memory address information is fed into the *mapper* module. The mapper first establishes the address range for each data structure from its starting address and size. The trace is analyzed to filter out page faults that are serviced at the same time (i.e., one page fault per page fault group is retained). The mapper then loops through page fault addresses collected from the trace and determines if a page fault has occurred within a given range. In addition to extracting the number of page faults per data structure, RPFM also calculates a number of synthesized metrics, such as page faults per data structure per CPU page that can provide useful feedback to an autotuning system, to guide the autotuner.

At install time, system attributes are estimated by repeatedly profiling a series of synthetic micro-benchmarks (as in [49]). To derive the system attributes used in (5)-(10), 16 microbenchmarks were profiled and a total of 28 events and metrics were collected across different runs. Prior to invoking the model on a specific application, a portion of each kernel is profiled by RPFM with checkpointing. These profiles extract the per-kernel attributes needed by our model (e.g., Fault concurrency in (6)) and map them to the data structures under consideration.

5.3 MemMap: A Runtime for Data Placement

MemMap derives estimates of the model parameters defined in (5)-(10). System attributes derived via microbenchmarking are hard-coded into the runtime system, per installation. Program attributes, which are sensitive to input data and launch bounds, are passed in at runtime via a call to `get_placement()`, the only exposed function in the MemMap API.

6 EXPERIMENTAL RESULTS

We evaluate our strategy on 11 heterogeneous applications from the Hetero-Mark benchmark suite [46]. Since Hetero-Mark benchmarks are implemented primarily with regular data structures, we include three irregular graph applications from LonestarGPU [48] (Table 1). We conduct experiments on a node featuring a POWER8 system with 128 logical cores connected to two NUMA sockets.

Table 1: Heterogeneous applications and design patterns

Name	Description	Design Pattern	Placement Configs
AES	Advanced Encryption	C2G	8
BE	Background Extraction	CPGC	256
BS	Black-Scholes	WP	8
BST	BST Insertion	CGP	8
EP	Evolutionary Program	CGP	128
FIR	Finite Impulse Response	C2GI	32
GA	Gene Alignment	GPCC	128
HIST	Color Histogramming	C2G	16
KM	KMeans Clustering	C&GI	256
KNN	K-Nearest Neighbor	WP	256
PR	PageRank	C2GI	32
BFS	Breadth-first Search	C&GI	8
SSSP	Single-source shortest path	C&GI	8
MST	Minimum spanning tree	C&GI	256

The node has four NVIDIA Tesla P100 GPUs connected to the CPU via NVLink. Experiments are conducted with CUDA 10.0 runtime and driver. All applications are compiled with nvcc at the highest optimization level. Hetero-Mark programs are run with the largest available data set. LonestarGPU codes are run with 20 graphs from the Koblenz database [31] and only the average numbers are reported. Each experimental run is repeated five times.

6.1 Roofline Validation

We validated the Roofline model using the Empirical Roofline Tool (ERT) [56]. We re-wrote the micro-kernels in ERT to run them under host and device placement. We then parameterized the kernels such that they can be invoked at different levels of access sparsity (σ) and page re-migration rates (ρ). We also modified the performance measurement module in ERT to include data copy time when the kernels are run under device placement. Fig 4 shows the results of the empirical validation on a P100 GPU connected to the CPU via a PCIe Gen3 interconnect. The GPU has a peak effective single-precision performance of 1256 GFLOPS/s and effective memory bandwidth of 362 GB/s. We observed that the empirical results closely follow the predicted peak performance for different values of σ and ρ . Although the micro-kernels under-perform in a few cases (e.g., very low operational intensity), the predicted performance ceilings are never exceeded, attesting to the validity of the Roofline as an instrument for determining performance bounds.

6.2 Performance Sensitivity

We conducted an exhaustive exploration of the space of all possible placement configurations to get a better understanding of the potential impact of placement choices. Fig. 5 shows the data placement performance space across 14 applications. Performance is expressed as a percentage of the maximum in each application's space (i.e., boxes should not be compared across applications). We see that data placement can cause wide variations in performance with integer-factor performance differences between the best and

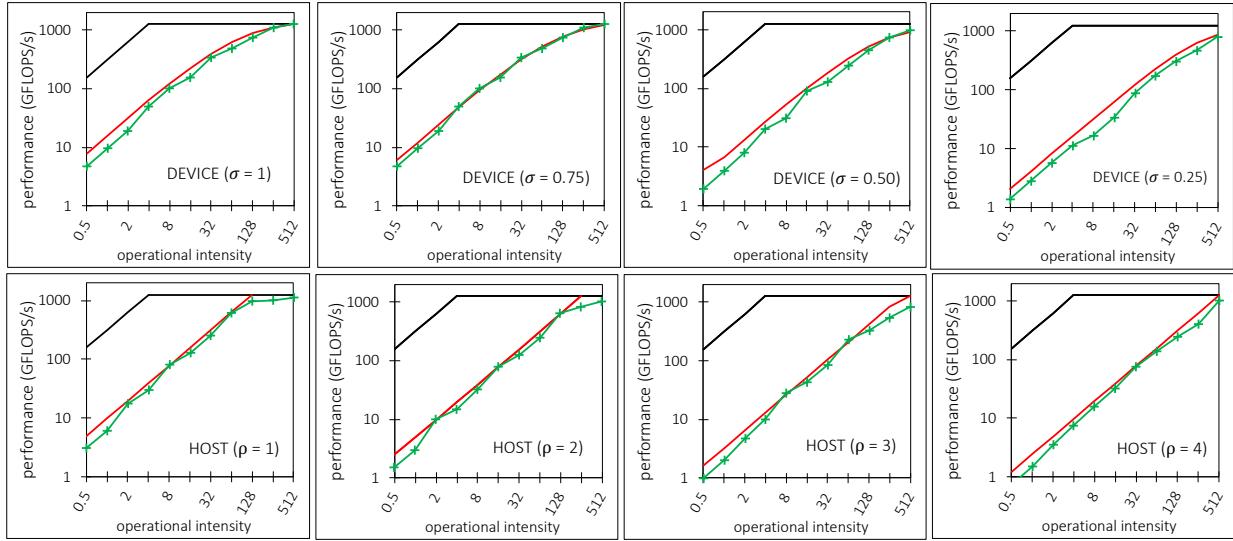


Figure 4: Empirical validation of the Roofline. Top set of figures shows access sparsity and bottom set shows re-migration. Hardware limits are shown in black; Roofline predicted ceilings are shown in red and empirical results are shown in green.

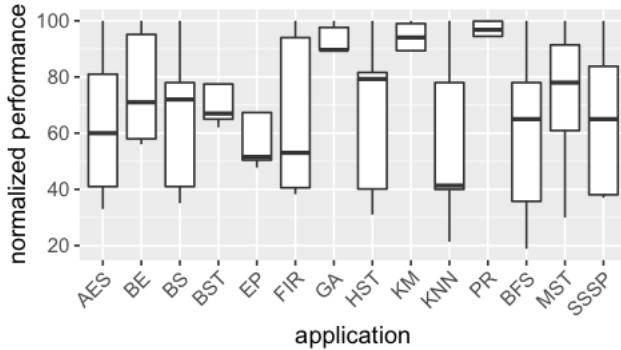


Figure 5: Data placement and performance sensitivity. Box denotes upper and lower quartile boundaries. Whiskers are drawn at $1.5 \times \text{IQR}$. Outliers are not shown.

worst configurations. One exception is PR, which proves to be more-or-less insensitive to placement decisions. For eight applications Q4 starts at or around 80%, indicating there exist many configurations that yield poor performance. BST and EP do not show a top whisker, which stipulates that there are very few configurations that achieve close to the max performance.

6.3 Data Movement

As discussed, access sparsity and page re-migration can have a direct impact on data movement under device and host placement, respectively. Fig. 6 shows the access sparsity levels of the principal data structures in the 14 applications. Access sparsity is measured by tracking the number of distinct pages accessed under host placement and dividing by the number of pages copied under device placement. As expected, irregular data structures (graphs in BFS, MST, SSSP and binary tree in BST) exhibit access sparsity. The PR

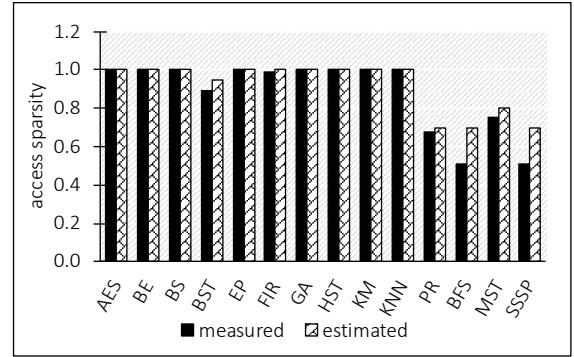


Figure 6: Measured access sparsity across 14 applications. Numbers are reported for the principal data structure.

implementation uses a matrix but is accessed sparsely because of the nature of the algorithm. The model correctly identifies each data structure as having dense or sparse access but it underestimates the sparsity levels for the irregular data structures. This is not surprising as we designed the model to be conservative when it comes to host placement and our method for estimating sparsity in irregular data structures is not too sophisticated. Nonetheless, as we will see in § 6.5, this level of accuracy proved sufficient in making the right placement decisions.

Fig. 7 shows the page re-migration rate of the principal data structures in the 14 applications. To measure the page migration rate, we utilize hardware performance counters to count the total number of migrations and also track the number of requests to distinct host-resident pages. For most applications, the page migration policy works quite well with no re-migration necessary. Predictably, however, re-migration is a problem for the tightly-coupled pipelined applications. The principal data structures in EP,

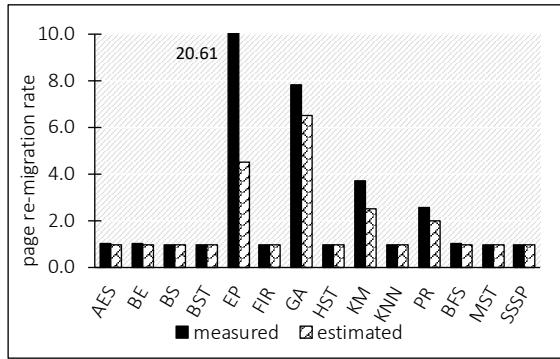


Figure 7: Observed and estimated page re-migration rates in 14 applications

GA, KM and PR are all being accessed in close proximity by the CPU and GPU, resulting in some pages being evicted from the device memory before being reused. Except for EP, the model is able to reasonably estimate the re-migration rates. Again, this inaccuracy did not lead to an incorrect decision as we discuss in § 6.5.

6.4 GPU Faults and Prefetching

Fig. 8 shows variations in GPU page fault rates (PFR) in host-placed data structures for four applications. PFR is impacted both by data access patterns and PME prefetch heuristics. Data structures with no reuse suffer a single page fault when the pages containing them are migrated for the first time. But if the access pattern is amenable to prefetching then the PFR can be significantly lower (e.g., *ciphertext*). Data structures whose pages are touched multiple times but the reuse is not exploited in the GPU cache, have a PFR > 1. Even in those cases prefetching can lower the PFR if the access pattern is regular. For example, the reuse in *histogram* is not exploited in cache but prefetching drives down its PFR to 0.64. By contrast *graph*, which is accessed in an irregular pattern has a PFR of 2.1. Scalar data structures, whose reuse is typically serviced in cache have a PFR of 1 (e.g., *num_points* in KM), unless the scalar resides in the same page as another host-placed data structure. In such cases PFR can be 0. We also observe that data structures like *clusters* in KM, exhibit page thrashing behavior, making them unsuitable for host placement.

6.5 End-to-End Performance

We compare our strategy (*model*) with the following *baseline*: original implementation in which all data structures touched by the GPU are placed in device memory.

mem_advise: Nvidia's `cudaMemAdvise()` API sets the initial location of managed data based on hints provided by the programmer. We invoke `cudaMemAdvise()` with appropriate hints such as `cudaMemAdviseSetReadMostly` to allow *mem_advise* to use its internal heuristic to make placement decisions. Note, *mem_advise* makes decisions on managed data only and thus is not directly comparable to *model* but it is the closest to the state-of-the-art in data placement for CUDA applications.

optimal: performance obtained with optimal placement, derived via exhaustive exploration of the placement space.

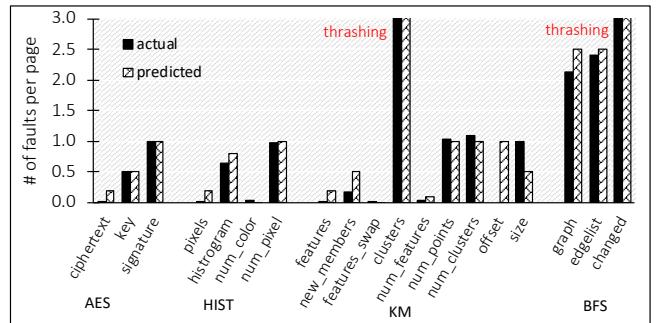


Figure 8: GPU page fault rates (PFR) of individual data structures (PFR = # of faults per 4K CPU page)

Table 2: Data structures selected for host-placement

Name	Data Structure
AES	ciphertext_
BE	_bg_, _fg_
BS	_rand_array_, call_price_
BST	tree_buffer
EP	island1, island2
FIR	input_buffer, output_buffer
GA	
HIST	d_pixels
KM	
KNN	h_locations
PR	
BFS	graph
MST	graph
SSSP	graph

Optimal Placement. Fig. 9 shows that for 11 of the 14 applications, *optimal* performance is obtained when at least some of the data is placed in host memory. *model* is able to identify the optimal configuration in all but one instance. In KNN, *model* underestimated the prefetching benefits of the *locations* data structure, an array of objects with two floats per object. This led the model to make a sub-optimal choice in placing *locations* in device memory. On further inspection, we found that a structure-splitting optimization [44] was being applied by the nvcc compiler which improved the prefetching for *locations*. The effects of aggregate data structure layout is not addressed in our analysis and is a shortcoming of the current model. We found that *mem_advise* takes a very conservative approach to host placement, allocating only a single data structure to host in BST, HIST and EP and PR. As a result, it finds the optimal configuration in only two cases. Furthermore, in some cases (EP and PR), it incorrectly identified a data structure for host placement which resulted in worse performance than *baseline*.

Criteria for Host Placement. Table 2 shows the data structures that were more profitable with demand fetching. Although the model considers many parameters, in our experiments we found page thrashing, access sparsity and the re-migration to be the most dominant criteria. Any data structure with thrashing behavior can be immediately eliminated from consideration for host placement. We

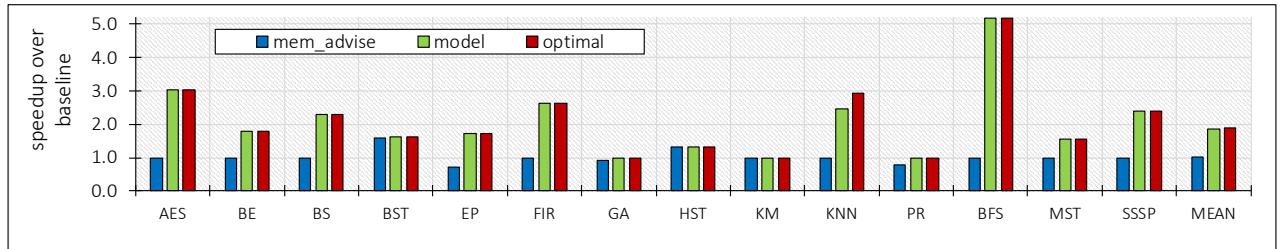


Figure 9: Performance comparison of different placement strategies.

encountered four such data structures in our experiments and although some of them did exhibit high degrees of prefetch activity and fault concurrency (e.g., *clusters*), it was no where close to offsetting the costs of thrashing. Irregular data structures in BFS, MST and SSSP exhibited poor prefetching and the kernels did not offer much latency hiding opportunities. In spite of that, these data structures were selected for host placement because sparsity dominated all other factors. Finally, several data structures were selected to be placed in device memory because of a high page re-migration rate. In our experiments, we found that a re-migration rate of > 2 is sufficient to negate any benefits of fault concurrency and the latency hiding opportunities in the kernel.

6.6 Overhead

The *online* overhead of our strategy is minimal: 1.43% on average (Fig. 9 numbers are inclusive of this overhead). The source-to-source transformation that applications are subject to does not re-order instructions and hence no heavy compiler machinery is required. The runtime overhead is dominated by calls to `MemMap()`. Average time for a call to `MemMap()` is 780 ms, which is 9.35% higher than a call to `cudaMalloc()`. Notwithstanding, `MemMap()` is called only once per data structure and thus the overhead is amortized over long running programs. The system incurs a one-time *offline* overhead at installation, during which micro-kernels are repeatedly executed and profiled. On our experimental platform, installation took 1 hour 16 minutes.

7 RELATED WORK

Recent research has looked at data partitioning between CPU and GPU [2, 6, 23, 24]. This work focuses on data placement, post partition. We consider this to be an orthogonal problem.

NUMA: Data movement issues on NUMA systems is tackled primarily by moving tasks closer to the data [5, 13, 15, 16, 47, 55]. This approach is not suitable for HMS since the observed differences in latency and bandwidth apply equally to all processing units (e.g., KNL). Furthermore, on HMS-DGPU, tasks are designed to work on a particular compute engine and moving them can have other performance implications (e.g., moving a GPU task to CPU).

GPU Memory: Analytical and profile-guided methods have been proposed for data placement within the GPU memory hierarchy [8, 9, 28, 57]. These techniques can be implemented in conjunction with our method to further improve data access in the GPU.

HMS: Work on data placement in HMS have either focused on NVM+DRAM [17, 26, 34, 52–54] or DRAM+HBM systems [41, 42,

50]. Dulloor *et al.* develop a placement policy where data structures are placed in DRAM based on access patterns on an emulated DRAM+NVM platform [17]. Several works have considered data placement on Intel’s KNL platform. Peng *et al.* present a policy in which objects are classified based on size, frequency of access and life span [42]. Wen *et al.* present ProfDP which incorporates HW performance counter based metrics such as bandwidth and latency sensitivity [50]. Chen *et al.* present a tool for data placement in graph applications, which also leverages performance counters [10]. This work targets HMS-DGPU which presents distinct performance challenges (§ 2.1).

Several recent works have proposed system-level solutions for data migration and prefetching on HMS-DGPU [1, 19–21, 35, 43, 58]. System-level approaches primarily focus on improving access to host-resident data and is complementary to the methods described here. For instance, CU execution during page faulting [58] will reduce the fault handling penalty in our model while the data management techniques in [35] will increase the performance of data structures selected for host placement.

8 CONCLUSIONS

This paper describes a new approach to data placement on hybrid memory systems with discrete GPUs. At the center of the framework is an analytical model that quantifies the performance trade-offs of placement decisions.

This work shows that placement decisions on HMS-DGPU can have a significant impact on performance, particularly for applications written in collaborative paradigms. Application characteristics such as page re-migration and access sparsity, and system attributes such as effective host-to-device bandwidth are important considerations for reasoning about these performance differences.

The experimental results reveal that in some instances it is profitable to map only a segment of a data structure to host memory. This advocates for a more fine-grained approach to data placement, which is part of our future work. This work also shows that performance of host-resident data can be further improved through compiler-level code transformation, which is another avenue for future exploration.

ACKNOWLEDGMENTS

The authors wish to thank Nuwan Jayasena and Mike Chu for their valuable input. This work was supported by the National Science Foundation through awards CNS-1253292 and OAC-1829644, and equipment grants by IBM and NVIDIA.

REFERENCES

- [1] 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 Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. 607–618.
- [2] J.I. Agulleiro, F. Vazquez, E.M. Garzon, and J.J. Fernandez. 2012. Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction. *Ultramicroscopy* 115 (2012), 109 – 114.
- [3] Masab Ahmad, Halit Dogan, Christopher J. Michael, and Omer Khan. 2019. HeteroMap: A Runtime Performance Predictor for Efficient Processing of Graph Analytics on Heterogeneous Multi-Accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24–26, 2019*. 268–281. <https://doi.org/10.1109/ISPASS.2019.00039>
- [4] AMD. [n. d.]. Radeon Open Compute Manifest 1.6. <https://rocm.github.io/>. Accessed: 2019-04-09.
- [5] Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. 2018. Reducing data movement on large shared memory systems by exploiting computation dependencies. In *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 207–217.
- [6] Michela Becchi, Surendra Byna, Srihari Cadambi, and Srimat Chakradhar. 2010. Data-aware Scheduling of Legacy Kernels on Heterogeneous Platforms with Distributed Memory. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. 82–91.
- [7] Victoria Caparrós Cabezas and Markus Püschel. 2014. Extending the roofline model: Bottleneck analysis with microarchitectural constraints. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 222–231.
- [8] Guoyang Chen and Xipeng Shen. 2016. Coherence-Free Multiview: Enabling Reference-Discerning Data Placement on GPU. In *Proceedings of the 2016 International Conference on Supercomputing (ICS16)*.
- [9] Guoyang Chen, Xipeng Shen, Bo Wu, and Dong Li. 2017. Optimizing Data Placement on GPU Memory: A Portable Approach. *IEEE Trans. Comput.* 66, 3 (March 2017).
- [10] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 293–304. <https://doi.org/10.1145/3368826.3377922>
- [11] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. 2013. A Roofline Model of Energy. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. Washington, DC, USA, 661–672.
- [12] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tippuraj, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 63–74.
- [13] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaise, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 381–394.
- [14] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262.
- [15] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. 2014. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 3 (2014), 30.
- [16] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 125–137.
- [17] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 15.
- [18] Denis Foley and John Danskin. 2017. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro* 37, 2 (2017), 7–17.
- [19] Debasish Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*. 224–235.
- [20] Debasish Ganguly, Z Zhang, J Yang, and Rami Melhem. 2020. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In *Proc. of the Int. Conf. on Parallel and Distributed Processing (IPDPS)*.
- [21] V. GarcÃ±a-Flores, E. Ayguade, and A. J. Pena. 2017. Efficient Data Sharing on Heterogeneous Systems. In *2017 46th International Conference on Parallel Processing (ICPP)*. 121–130.
- [22] Juan Gomez-Luna, Izzat El Hajj, Li-Wen Chang, Victor Garcia-Flores, Simon Garcia de Gonzalo, Thomas B Jablin, Antonio J Pena, and Wen-mei Hwu. 2017. Chai: collaborative heterogeneous applications for integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. 43–54.
- [23] Chris Gregg, Jeff Brantley, and Kim Hazelwood. 2010. Contention-aware scheduling of parallel code for heterogeneous systems. In *2nd USENIX Workshop on Hot Topics in Parallelism (HotParâž10)*.
- [24] Dominik Grewe and Michael F. P. O'Boyle. 2011. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC'11/ETAPS'11)*. 286–305.
- [25] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar)*, 2012. IEEE, 1–14.
- [26] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2015. Software-managed energy-efficient hybrid DRAM/NVM main memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 23.
- [27] Intel. 2016. MCDRAM (High Bandwidth Memory) on Knights Landing : Analysis Methods and Tools.
- [28] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2010. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2010), 105–118.
- [29] Joonyoung Kim and Younsu Kim. 2014. HBM: Memory Solution for Bandwidth-Hungry Processors. In *A Symposium on High Performance Chips (HOTCHIPS)*.
- [30] Konstantinos Krommydas, Wu-chun Feng, Christos D Antonopoulos, and Nikolaos Bellas. 2016. OpenDwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems* 85, 3 (2016), 373–392.
- [31] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. 1343–1350.
- [32] Martijn HR Lankhorst, Bas WSMM Ketelaars, and Robertus AM Wolters. 2005. Low-cost and nanoscale non-volatile memory concept for future silicon chips. *Nature materials* 4, 4 (2005), 347.
- [33] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 94–110.
- [34] Dong Li, Jeffrey S Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. 2012. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 945–956.
- [35] Lingda Li and Barbara M. Chapman. 2019. Compiler assisted hybrid implicit and explicit GPU memory management under unified address space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17–19, 2019*. 51:1–51:16. <https://doi.org/10.1145/3295500.3356141>
- [36] Gabriel H Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *ACM SIGARCH computer architecture news*, Vol. 36. IEEE Computer Society, 453–464.
- [37] John D McCalpin et al. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2, 19–25 (1995).
- [38] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. 2014. Unified memory in CUDA 6.0: a brief overview of related data access and transfer issues. Technical Report TR-2014-09.
- [39] Tesla NVIDIA. 2017. *NVIDIAâž Teslaâž P100 âž The Most Advanced Data Center Accelerator Ever Built*. Technical Report WP-08019-001.
- [40] Tesla NVIDIA. 2017. *NVIDIAâž Teslaâž V100 GPU architecture*. Technical Report WP-08608-001_v1.
- [41] M Ben Olson, Tong Zhou, Michael R Jantz, Kshitij A Doshi, M Graham Lopez, and Oscar Hernandez. 2018. MemBrain: Automated Application Guidance for Hybrid Memory Systems. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–10.
- [42] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: A tool for data placement on hybrid memory system. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 82–91.
- [43] Apan Qasem, Aswhin Aji, and Gregory Rodgers. 2017. Characterizing Data Organization Effects on Heterogeneous Memory Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.
- [44] Probir Roy and Xu Liu. 2016. StructSlim: A Lightweight Profiler to Guide Structure Splitting. In *Proceedings of the 2016 International Symposium on Code Generation*

- and Optimization (CGO '16)*. Association for Computing Machinery, New York, NY, USA, 364–346. <https://doi.org/10.1145/2854038.2854053>
- [45] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development* 59, 1 (2015), 7–1.
- [46] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David R. Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25–27, 2016*. 13–22.
- [47] Masahiro Tanaka and Osamu Tatebe. 2012. Workflow scheduling to minimize data movement using multi-constraint graph partitioning. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 65–72.
- [48] University of Texas at Austin. [n. d.]. LonestarGPU. <http://iss.ices.utexas.edu/~p-projects/galois/lonestargpu>.
- [49] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*.
- [50] Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. 2018. Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems. In *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 263–273.
- [51] Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).
- [52] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM, 58.
- [53] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 31.
- [54] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S Vetter, and Sparsh Mittal. 2016. Algorithm-directed data placement in explicitly managed non-volatile memory. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 141–152.
- [55] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2009. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 172–187.
- [56] Charlene Yang, Rahulkumar Gayatri, Thorsten Kurth, Protonu Basu, Zahra Ronagh, Adedoyin Adetokunbo, Brian Friesen, Brandon Cook, Douglas Doerfler, Leonid Oliker, et al. 2018. An empirical roofline methodology for quantitatively assessing performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 14–23.
- [57] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 155–164.
- [58] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.