# Morpheus: Extending the Last Level Cache Capacity in GPU Systems Using Idle GPU Core Resources

*Sina Darabi[†] *Mohammad Sadrosadati[§] Joël Lindegger[§] Negar Akbarzadeh[†] Mohammad Hosseini[‡]
Jisung Park[§∇] Juan Gómez-Luna[§] Hamid Sarbazi-Azad[†‡] Onur Mutlu[§]

[†]*Sharif University of Technology* [§]*ETH Zürich*
[‡]*Institute for Research in Fundamental Sciences (IPM)* [∇]*POSTECH*[*]



*Graphics Processing Units (GPUs) are widely-used accelerators for data-parallel applications. In many GPU applications, GPU memory bandwidth bottlenecks performance, causing underutilization of GPU cores. Hence, disabling many cores does* not *affect the performance of memory-bound workloads. While simply power-gating unused GPU cores would save energy, prior works attempt to better utilize GPU cores for other applications (ideally compute-bound), which increases the GPU's total throughput.*

*In this paper, we introduce* Morpheus*, a new hardware/software co-designed technique to boost the performance of memory-bound applications. The key idea of Morpheus is to exploit unused core resources to extend the GPU last level cache (LLC) capacity. In Morpheus, each GPU core has two execution modes:* compute mode *and* cache mode*. Cores in* compute mode *operate conventionally and run application threads. However, for the cores in* cache mode*, Morpheus invokes a software helper kernel that uses the cores' on-chip memories (i.e., register file, shared memory, and L1) in a way that extends the LLC capacity for a running memory-bound workload. Morpheus adds a controller to the GPU hardware to forward LLC requests to either the conventional LLC (managed by hardware) or the extended LLC (managed by the helper kernel). Our experimental results show that Morpheus improves the performance and energy efficiency of a baseline GPU architecture by an average of 39% and 58%, respectively, across several memory-bound workloads. Morpheus' performance is within 3% of a GPU design that has a quadruple-sized conventional LLC. Morpheus can thus contribute to reducing the hardware dedicated to a conventional LLC by exploiting idle cores' on-chip memory resources as additional cache capacity.*

## 1. Introduction

Graphics Processing Units (GPUs) are widely-used accelerators, especially for data-parallel applications with high arithmetic intensity (i.e., arithmetic instructions executed per byte accessed from memory). GPUs rely on managing execution resources for a large number of Single-Program-Multiple-Data (SPMD) threads to exploit this arithmetic intensity and overlap the long memory access latencies with computation [1-3].

Unfortunately, the maximum performance of a GPU is often limited by the available memory bandwidth [4], causing considerable underutilization of GPU cores, i.e., GPU cores are frequently idle (waiting for memory accesses) during application execution time. This is the case for many important general-purpose GPU applications, such as `kmeans` [5], `mri-gri` [6], and `cfd` [5], which are memory-bound in nature due to their low arithmetic intensity [5, 6]. As a result, we do *not* need to use all available GPU cores to saturate the performance of these applications. To demonstrate this, we experimentally study (using an NVIDIA RTX 3080 GPU [7] and Accel-Sim [8]) the performance of 14 memory-bound applications as we scale the number of GPU cores. Our experiments reveal that, on average across the 14 applications, only 56% of the GPU cores are enough to saturate performance (see §3 for more details). Hence, the remaining 44% of the GPU cores, on average, can remain *unused* (i.e., no threads scheduled onto them) without hurting performance of memory-bound applications.

Several prior works (e.g., [9-18]) make similar observations and propose to have *only* a *subset* of the available GPU cores execute memory-bound application threads, and leverage the remaining GPU cores in one of three ways: (1) *power-gating* these cores to save energy [9], (2) using them for *redundant execution* of the *already running* memory-bound application for better reliability [10, 11], and (3) *co-scheduling* additional *compute-bound* applications to these cores to increase GPU's total throughput [12-18]. In contrast, **our goal** is to boost the performance of a memory-bound application using a number of GPU cores that are not useful for executing application threads.

To this end, we propose ***Morpheus***.[1] **The key idea of Morpheus** is to reserve a number of GPU cores to use their on-chip memory (i.e., register files, shared memory, and L1 cache) as an extension of the shared last level cache (LLC, e.g., the shared L2 cache in NVIDIA GPUs). Doing so can improve the performance of memory-bound applications due to two main reasons. First, a larger LLC reduces the number of off-chip memory accesses, thereby enabling more threads to run effectively since the system is not memory bandwidth bottlenecked any more. Second, a larger LLC reduces memory access latency, which can improve performance.

Morpheus introduces two *execution modes* (compute and

[*]Sina Darabi and Mohammad Sadrosadati are co-primary authors.

[1] *"Morpheus"* because we dynamically *morph*, parts of GPU hardware (in a logical sense) to meet memory-bound applications' needs.

cache) for every GPU core. A core in *compute mode* behaves like a regular core in conventional GPUs, i.e., it executes application threads. A core in *cache mode* lends its on-chip memory space to extend the effective shared LLC size via a hardware/software co-designed technique. A GPU core in cache mode runs a software helper kernel, called the *extended LLC kernel*, that stores the extended LLC tag/data arrays inside the GPU core's local register file, shared memory, and L1 cache. We add a hardware controller, called the *Morpheus controller*, to support access to the extended LLC. The Morpheus controller performs three main tasks: it (1) forwards each LLC request to either the conventional LLC or the extended LLC, depending on the requested memory address (i.e., LLC set number), (2) tracks outstanding extended LLC requests, and (3) predicts the outcome of an extended LLC lookup (hit/miss), so that it forwards *only* the requests that are predicted to be hit in the extended LLC to GPU cores in cache mode, which mitigates the overhead of extended LLC misses. The Morpheus controller uses Bloom filters [19] for hit/miss prediction, providing *zero* false-negative and negligible false-positive rates.

To improve the effectiveness of Morpheus, we employ two optimization techniques on top of our basic design. First, we increase the effective capacity of the extended LLC by employing a cache compression technique in the extended LLC kernel. Second, we accelerate the data array access in the extended LLC by adding a new specialized instruction to the GPU instruction set architecture. This new instruction enables *indirectly* addressing a register, i.e., reading from a register whose index is determined by accessing the value in another register.

To evaluate the effectiveness of Morpheus, we first measure the bandwidth, access latency, and energy consumption of the extended LLC via real-system experiments using an NVIDIA RTX 3080 GPU (§5). We then use the measured performance and energy numbers for the extended LLC in the AccelSim simulator [8] to estimate the effect of Morpheus on overall GPU performance and energy consumption. Our experimental results show that Morpheus improves overall GPU performance and energy efficiency by an average of 39% and 58%, respectively, compared to the baseline NVIDIA RTX 3080 GPU architecture, across 14 memory-bound applications. Morpheus performs within 3% of a conventional GPU design that has a quadruple-sized conventional LLC (assuming *no* latency overhead for the larger conventional LLC).

We make the following contributions:

- We propose Morpheus, the first technique that leverages some GPU cores' on-chip memories to extend the total GPU last-level cache capacity.
- We introduce two new mechanisms that Morpheus employs: (1) a software helper kernel to extend the LLC using the on-chip memory resources of GPU cores that are in cache mode, and (2) a hardware controller that enables accesses to *both* the conventional LLC and the extended LLC.
- We evaluate Morpheus and show that it significantly improves performance and energy. It also enables a 4× larger cache without requiring new hardware resources for it.

## 2. Background

A GPU program is composed of a number of kernels that are executed using Single-Program-Multiple-Data (SPMD) threads [20]. These threads are partitioned into multiple blocks, or Cooperative Thread Arrays (CTAs) [20]. CTAs are then assigned to Single-Instruction-Multiple-Data (SIMD) cores for execution, called *Streaming Multiprocessors (SMs)* on NVIDIA GPUs. Each SM contains multiple CUDA cores, special-function units (e.g., cos, sin and tan), shared memory, L1 cache and thousands of registers. SMs are connected to several memory partitions using an on-chip interconnection network. Each memory partition includes one or multiple LLC banks, a memory controller, and a main memory (GDDRx/HBMx). Threads inside each CTA are grouped into *warps*. Threads within a warp execute the same instruction on different data items in a lock-step manner. The *warp scheduler* time-multiplexes warps, and assigns warps to the execution units.

## 3. Motivation

Memory-bound applications cannot fully utilize the compute throughput of GPUs as they are bottlenecked by the limited memory bandwidth. This causes long memory access latencies, which cannot be hidden via thread-level parallelism, causing core idleness and performance saturation [4, 21-29]. To illustrate this observation, we experimentally study the performance of 17 applications (14 memory-bound and 3 compute-bound) as we scale the number of GPU cores using AccelSim [8] (see §6 for our methodology). Figure 1 illustrates the results for a baseline GPU architecture that resembles an NVIDIA RTX 3080 [7]. The x-axes correspond to the number of GPU cores (SMs), ranging from 10 (chosen empirically) to 68 (the total SM count in an RTX 3080). The y-axes correspond to the normalized performance for each application. We normalize the performance of each application to its performance when using 10 SMs for readability.

We make two key observations. First, performance *gradually saturates* (i.e., stops increasing) as the number of SMs increases for 9 of the memory-bound applications (`p-bfs`, `cfd`, `dwt2d`, `stencil`, `r-bfs`, `bprob`, `sgem`, `nw`, `page-r`). In contrast, the performance of the compute-bound applications continues to increase with more SMs. Second, performance *decreases* sharply after a certain number of SMs for 5 of the memory-bound applications (`kmeans`, `histo`, `mri-gri`, `spmv`, `lbm`). For example, after more than 20 SMs are used for the kmeans application, performance drops greatly and the performance of the application with 68 SMs is almost the same as it is with 10 SMs, which is 50% lower than with 20 SMs. We conclude that limiting the number of SMs running a memory-bound application (after some SM count) not only does *not* significantly hurt performance but can even *improve* performance in some cases.

Leveraging our key observations, we aim to reserve a number of SMs and use their on-chip memory (i.e., register file, shared memory, and L1 cache) to extend the overall LLC capacity and, thus, improve GPU performance for memory-bound

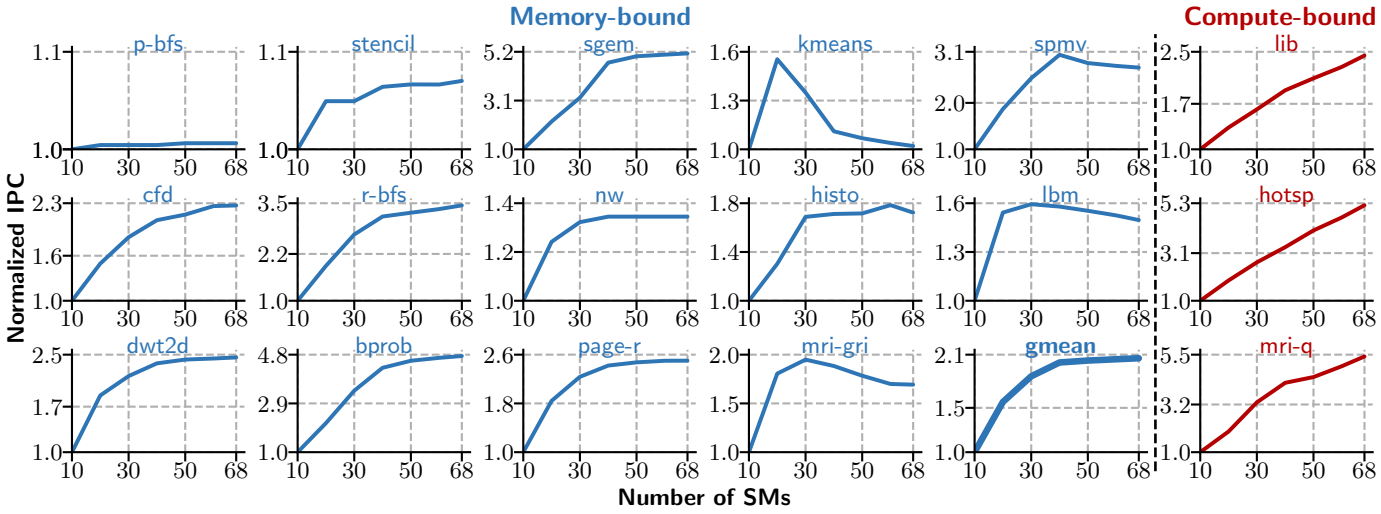


**Figure 1: Normalized GPU performance (IPC) of 14 memory-bound and 3 compute-bound applications**

applications. To quantify the benefits of potentially having a larger LLC for memory-bound applications, we repeat our previous experiment with 2× and 4× LLC sizes, compared to the baseline 5-MiB LLC of an NVIDIA RTX 3080. We vary the number of GPU cores in these two evaluated GPU designs (2× and 4× larger LLCs) and measure overall performance for each case. Figure 2 shows the maximum performance that we observe while varying the number of GPU cores for 14 representative memory-bound applications. We normalize the performance results of each application to the case of a baseline 5-MiB LLC. We observe that both 2× and 4× LLC sizes improve the performance of *all* evaluated memory-bound applications. In particular, a 4× larger LLC improves performance by as much as 2.34× (kmeans), and by 1.57× on average (geometric mean). We conclude that a larger LLC effectively and consistently improves the performance of memory-bound applications.

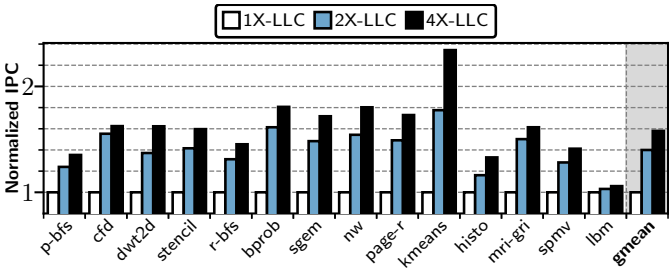


**Figure 2: Effect of larger LLC sizes (2× and 4× the size of the baseline LLC) on performance (normalized IPC relative to the baseline LLC) of 14 memory-bound applications**

**Our goal** in this work is to design a mechanism that can leverage the on-chip memory units of a number of GPU cores (that are otherwise not beneficial), to effectively extend the overall LLC capacity for memory-bound GPU applications, so as to increase performance.

## 4. Morpheus

We introduce Morpheus, a hardware/software co-designed technique to alleviate the memory bottleneck on GPUs. The key idea of Morpheus is to reserve a number of GPU cores and use their on-chip memory units (i.e., register file, shared memory, and L1 cache) as an extension of the GPU's LLC. In Morpheus, each GPU core has two working modes, *compute mode* and *cache mode*. Cores in compute mode behave exactly like the cores in existing GPUs and execute application threads. In contrast, cores in cache mode execute a software helper kernel (called the *extended LLC kernel*) to extend the LLC capacity by exploiting the storage capacity of their local on-chip memory units. This additional LLC capacity provided by the cores in cache mode is called the *extended LLC*.

Figure 3 gives a conceptual overview of the LLC lookup procedure in Morpheus. An LLC request in a Morpheus-enabled GPU first arrives at the *Morpheus controller* (❶ in Figure 3). The Morpheus controller forwards the LLC request to either the conventional LLC (which works exactly as in existing GPUs) or the *extended LLC* (which we propose). The forwarding decision is based on a static address mapping scheme, called *address separation* ❷. An access to the extended LLC is served by the *extended LLC controller* ❸ using the on-chip memory units of cores in cache mode. The extended LLC controller either *forwards* a given request to the core's L1 cache ❹, or *directly queries* the register file ❺ or shared memory ❻. The forwarding or querying decision is based on the same static address separation principle as in the Morpheus controller.

Morpheus implements the Morpheus controller as a new hardware unit per LLC partition and the extended LLC controller as software (i.e., the extended LLC kernel) running on cores in cache mode. Figure 4 shows a pictorial example of a Morpheus-enabled GPU with these new components. An LLC request originates from a core in compute mode (① in Figure 4) and moves through the interconnection network to an LLC partition ② based on a static address mapping scheme, similar to the one used in a conventional GPU. In

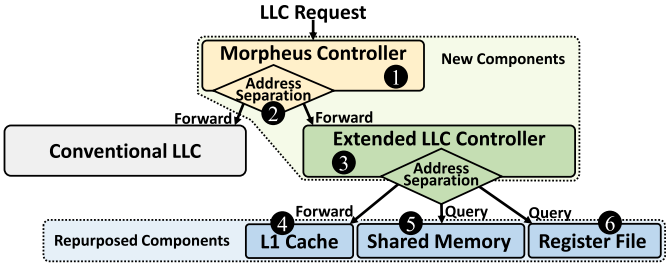


**Figure 3: Conceptual overview of Morpheus**

a Morpheus-enabled GPU, a hardware implementation of the Morpheus controller local to the LLC partition ③ then either forwards the request to the local conventional LLC, or through the interconnection network to the responsible GPU core running in cache mode ④ based on a static address separation scheme (§4.1.1). The responsible cache mode GPU core is determined by the address separation scheme. The extended LLC controller is implemented as multiple instances of a software helper kernel (the extended LLC kernel) running on cores in cache mode ⑤. This kernel queries one of the local memory units ⑥ for each incoming request, based on a static address separation scheme (§4.1.1), and sends the response over the interconnection network back to the Morpheus controller ③. If the request is a hit in the extended LLC, the cache block is sent over the interconnection network to the GPU core that initially issued the LLC request ①. If the request is a miss in the extended LLC, it is treated exactly like an LLC miss in a conventional GPU by the LLC partition.

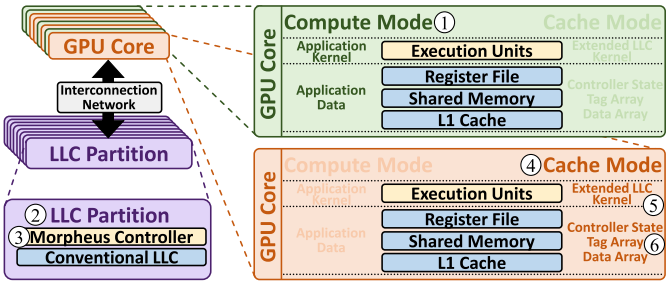


**Figure 4: GPU structure with Morpheus**

The rest of this section explains Morpheus' key mechanisms in detail. §4.1 describes our hardware implementation of the Morpheus controller. §4.2 describes our software implementation of the extended LLC controller using the extended LLC kernel. §4.3 introduces two optimization techniques on top of our basic design to improve the effectiveness of Morpheus.

## 4.1. Morpheus Controller

This unit has three main tasks: (1) separating LLC requests between the conventional LLC and the extended LLC, (2) handling communication between the extended LLC and the LLC partition, and (3) predicting the outcome of the extended LLC lookup (hit/miss), so that the Morpheus controller forwards only the requests that are predicted to be hits in the extended LLC to GPU cores in cache mode, which mitigates the overhead of extended LLC misses. Figure 7 shows the main components in the Morpheus controller. In this section, we describe each component in detail.

**4.1.1. Address Separation.** Since a Morpheus-enabled GPU employs two distinct LLCs (i.e., the conventional LLC and the extended LLC), there should be a mechanism to map each cache block to one of these LLCs. To this end, Morpheus divides the memory address space statically into two partitions proportional in size to the conventional and extended LLC capacity. The conventional LLC is responsible for caching the first partition, while the extended LLC is responsible for caching the second. When the Morpheus controller receives an LLC request, a unit called *address separator* checks whether or not the set number is in the range of memory addresses served by the extended LLC. If so, the unit forwards the LLC request to the next unit in the Morpheus controller, the *hit/miss predictor*. Otherwise, the conventional LLC handles the request exactly the same way as done in the conventional GPU architectures.

**4.1.2. Hit/Miss Prediction.** Misses in the extended LLC are more expensive in terms of latency compared to misses in the conventional LLC. Figure 5 breaks down the latencies of hits and misses in both the conventional and extended LLC (see §6 for our methodology). We observe that misses in the conventional LLC take 608ns to be served, while misses in the extended LLC take 773ns (i.e., 27% longer). This is due to two main reasons. First, an extended LLC miss adds a round trip interconnection network latency (two grey boxes in Figure 5) compared to a conventional LLC miss, to move the request and response between the Morpheus controller and a GPU core in cache mode. Second, the latency for accessing the extended LLC is longer than for the conventional LLC, because in the extended LLC, tag lookups and data array accesses have to be managed by the extended LLC kernel (i.e., in software).

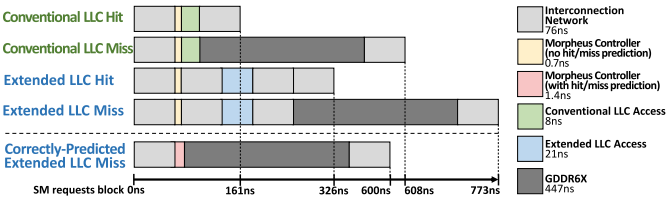


**Figure 5: Timelines for LLC hits, misses, and predicted misses on a Morpheus-enabled GPU**

We mitigate the overhead of extended LLC misses using a *hit/miss predictor*. After the address separation mechanism in the Morpheus controller determines that a given request falls into the extended LLC's address space, the hit/miss predictor decides if the request is likely to be a hit in the extended LLC. If the request is predicted to be a hit, the Morpheus controller forwards the request to the extended LLC. Otherwise, the Morpheus controller directly accesses the off-chip DRAM to serve the memory request.

Figure 5 shows the timeline for a correctly-predicted extended LLC miss. If a miss is predicted correctly, it avoids the unnecessary latency of (1) an interconnection network roundtrip, and (2) querying the tag array with the extended

LLC kernel. Thus, correctly-predicted extended LLC misses can be serviced as fast as conventional LLC misses.

To maintain correctness in the presence of the hit/miss predictor, we must understand if and which types of mispredictions are acceptable. We observe that it is acceptable to falsely predict a request as an LLC hit when in reality the request is a miss. This is because the predicted LLC hit causes a lookup in the extended LLC, at which point the misprediction will be discovered. Nevertheless, such *false positives* are undesirable, because they increase the latency of extended LLC misses to the same latency as if there were no hit/miss prediction. In contrast, falsely predicting a request as an LLC miss even though the requested address is in the LLC can violate both coherence and consistency guarantees. For example, if a request to a dirty cache block is falsely predicted as an LLC miss, the requesting core will receive an out-of-date value from main memory, which violates basic cache correctness. Thus, for correctness, the hit/miss predictor must not produce such *false negatives*, or otherwise it should recover from its mispredictions such that no correctness problems appear.

We design our hit/miss predictor using Bloom filters [19]. Bloom filters are a good fit for our requirements because they provide fast and low-cost set membership queries without false negatives. In our hit/miss predictor, a Bloom filter represents the LLC blocks which are currently in a given extended LLC set. To avoid increasingly frequent false positives, the Bloom filters must be cleared regularly,[2] after which there is a risk for *false negatives*. We next describe an algorithm using two Bloom filters, which are cleared alternately, thus avoiding false negatives.

The key idea of our extended LLC hit/miss prediction algorithm is to track the blocks currently in the LLC set using two Bloom filters per extended LLC set, BF1 and BF2. BF1 and BF2 are updated upon every access to the extended LLC set, such that at any given time, (1) BF1 contains at least all the cache blocks currently in the extended LLC set, and (2) BF2 contains the $n$ most recently used cache blocks in the extended LLC set. Invariant (1) guarantees the absence of false negatives when querying BF1 with requested extended LLC addresses, i.e., BF1 can be used to safely predict if a request will hit in the extended LLC set. Invariant (2) enables eventually replacing BF1 with BF2, when BF2 contains all cache blocks in the extended LLC set ($n \geq$associativity). The benefit of eventually replacing BF1 with BF2 is that BF2 does not (yet) contain any evicted cache blocks, thus producing fewer false positives than the old BF1. In contrast, BF1 may contain multiple evicted cache blocks. We explain in detail how the hit/miss predictor queries BF1, and how BF1 and BF2 are updated upon every access to the extended LLC set to maintain invariants (1) and (2).

Figure 6(a) shows a flow diagram of how the Morpheus controller with the hit/miss predictor serves an extended LLC request. To make a hit/miss prediction for some LLC request, the hit/miss predictor queries BF1 with the LLC request's address (❶ in Figure 6). It predicts a hit when the address is found in BF1, and a miss otherwise. This cannot produce false negatives, since invariant (1) guarantees that BF1 contains all blocks currently in the extended LLC set, and Bloom filters do not produce false negatives. Upon a predicted extended LLC hit, the algorithm queries the address in the extended LLC ❷. Upon a (predicted or actual) extended LLC miss, the Morpheus controller accesses the requested block in DRAM instead ❸. Finally, the response is sent from the Morpheus controller to the core that issued the request ❹.

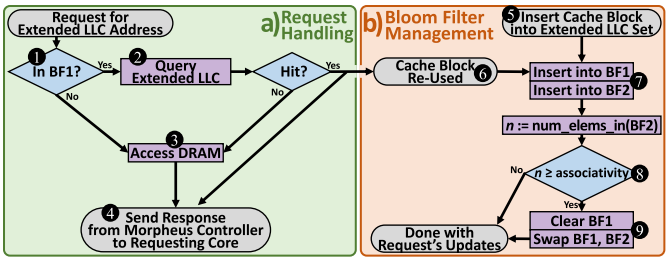


**Figure 6: Flowchart of the extended LLC hit/miss predictor**

Figure 6(b) shows a flow diagram of how the hit/miss predictor updates BF1 and BF2 for a given extended LLC access to maintain invariants (1) and (2). Upon an access (i.e., when a cache block is inserted into the set (❺ in Figure 6), or a cache block in the set is re-used ❻), the accessed cache block is inserted into both Bloom filters ❼. This trivially maintains invariant (1), because any inserted extended LLC block will also be in BF1. Invariant (2) is maintained because before the insertion, BF2 contained the $n_{old}$ most recently used blocks, and after the insertion BF2 contains either the $n$=$n_{old}$+1 most recently used blocks (if the used block was not in BF2 before) or the $n$=$n_{old}$ most recently used blocks (if the used block was already in BF2). After $n$ becomes as large as the set's associativity ❽, all future predictions can be made by querying BF2 instead of BF1, without risking false negatives. This is because the extended LLC uses the LRU (least recently used) replacement policy and BF2 contains the $n$ most recently used blocks, thus if $n$ is as large as the set's associativity, BF2 is guaranteed to contain *all* blocks that are in the LLC set. Thus, the contents of BF1 are cleared, BF1 and BF2 are swapped, and the scheme repeats ❾.

**Cost.** Assuming each Bloom filter is 32 bytes in size, and up to 256 extended LLC sets per LLC partition, the hit/miss predictor requires 32B×2×256=16KiB of Bloom filter storage in each LLC partition.

**4.1.3. Extended LLC Query Logic Unit.** The Morpheus controller includes a hardware unit called the *extended LLC query logic unit* to track and manage outstanding extended LLC requests. Figure 7 shows the four main components in the extended LLC query logic unit, namely the *request queue*, the *warp status table*, the *read data buffer*, and the *write data buffer*. We explain each component in this section.

**Request Queue.** For simplicity, each extended LLC kernel warp serves only a single extended LLC request at a time in Morpheus. To avoid clogging the interconnection network

[2] Counting Bloom filters [30] would support individual element removal instead, but require more bits compared to standard Bloom filters.

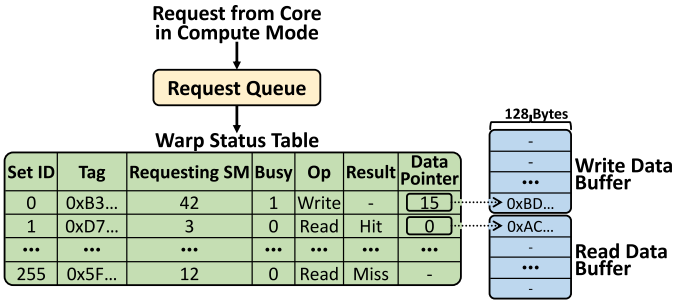


**Figure 7: Extended LLC Query Logic Unit**

with backlogged request bursts, we introduce a request queue, which buffers any requests to the extended LLC. A given request is de-queued as soon as the warp assigned to the request's extended LLC set is ready to serve a new request. When dequeued, the request's metadata is written to the corresponding row in the warp status table. If the request is a write, the payload data is written to the write data buffer.

**Warp Status Table.** The warp status table has one row per set in the current LLC partition, tracking the status of the extended LLC kernel warp that is assigned to each set. Each row has fields for the current request's tag, the origin of the request, a *busy* bit (indicating if the warp is currently serving a request), an *op* field (indicating if the request is a read or write), a *result* field (indicating if the request is a hit or a miss in the extended LLC), and a pointer to either the read or write data buffer entry for the payload data. The warp status table is memory-mapped, thus the extended LLC kernel warps can read from and write to it with regular load/store instructions.

We size the warp status table based on the maximum number of sets in the extended LLC. We assume an NVIDIA RTX 3080 GPU as the baseline, which has 10 LLC partitions, 68 SMs, and 48 warps per SM (and thus, up to 48 extended LLC sets per SM in cache mode). We assume that up to 75% of all SMs can be in cache mode, based on Figure 1, where GPU performance starts to saturate after using at least 25% of the SMs for computation. Under these assumptions, Morpheus can provide up to 2448 extended LLC sets, i.e., about 256 sets per LLC partition. As a result, the warp status table has 256 rows in each Morpheus controller of a Morpheus-enabled NVIDIA RTX 3080 GPU.

**Read and Write Data Buffers.** The read and write data buffers hold payload data from requests to the extended LLC. For example, when a write request arrives at the extended LLC query logic unit, it includes a dirty cache block (e.g., 128 bytes) to write to the extended LLC. These 128 bytes are written to an entry in the write data buffer, and the data pointer in the warp status table is updated to point to that entry. Read requests work analogously, with the difference that the extended LLC kernel writes the requested cache block to the read data buffer entry instead. Like the warp status table, the read and write data buffers are memory-mapped, and thus extended LLC kernel warps can read from and write to them with simple load/store instructions.

## 4.2. Extended LLC Controller

Morpheus implements the extended LLC controller as a software helper kernel, called the *extended LLC kernel*, that performs three main tasks: (1) storing and updating the extended LLC tags and data in the local memory units of a GPU core operating in cache mode (§4.2.1-4.2.2), (2) executing simple operations (e.g., increment) on the extended LLC blocks needed for *atomic* instructions (§4.2.3), and (3) identifying the correct memory unit to perform these operations on, based on a static address separation mechanism.

Morpheus schedules one copy of the extended LLC kernel on each GPU core that is operating in cache mode. The extended LLC kernel uses the maximum number of warps in each GPU core, and each warp handles *exactly* one extended LLC set. The extended LLC kernel effectively uses all of the local memory units of GPU cores operating in cache mode to store the extended LLC data and metadata (e.g., tags, valid bits, dirty bits), except for a number of registers that the extended LLC kernel reserves as *auxiliary registers* for its own operation. In this section, we explain only the first and second tasks of the extended LLC kernel since the third task, i.e., address separation, is analogous to the address separation in the Morpheus controller (§4.1.1), with the only difference being that the address space is divided proportionally to the respective *memory units'* capacities instead.

**4.2.1. Extended LLC via Register File.** In this section, we describe how the extended LLC kernel (1) lays out the extended LLC blocks' tags and data in the register file, and (2) accesses and updates them.

**Extended LLC Layout in the Register File.** Figure 8 shows how the extended LLC kernel lays out 48 sets of a 32-way set-associative extended LLC in the register file of a GPU core operating in cache mode. This example assumes a baseline NVIDIA RTX 3080 [7] GPU, with up to 48 active warps per SM, and 42 registers per warp.

The extended LLC kernel uses the register file to store a number of extended LLC sets (e.g., 48) ①, for each set a number of cache blocks (e.g., 32 blocks of 128 bytes each) ②, and for each block a metadata block ③ containing the block's LRU counter, dirty bit, valid bit, and tag ④. Each extended LLC set ① is stored in the registers of exactly one warp ⑤. Each data block of a set ② is stored in exactly one warp register, called a *data-array register* ⑥. The metadata blocks ③ are coalesced into a single warp register, called the *metadata register* ⑦, such that thread $i$ holds block $i$'s metadata block. The remaining *auxiliary registers* are used for extended LLC kernel execution ⑧.

In this configuration of 48 extended LLC sets per GPU core operating in cache mode, 32 blocks per set, and 128 bytes per block, each GPU core operating in cache mode adds $48 \times 32 \times 128\text{B} = 192\text{KiB}$ capacity to the extended LLC by using the space in the GPU core's register file.

**Extended LLC Tag Lookup.** When a warp executing the extended LLC kernel receives an extended LLC request for its set, the warp executes a tag lookup procedure to determine if

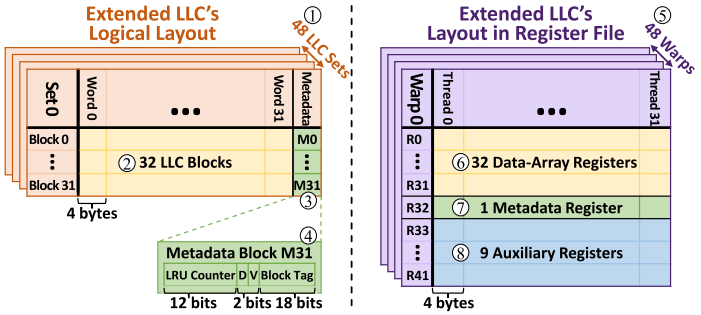


**Figure 8: Each out of the 48 warps in an SM in cache mode implements one fully associate cache set (e.g., set 0) with several cache blocks (e.g., 32).**

the request is a hit in the set, and if so, which cache block in the set was hit. To this end, the warp compares the tag of the request address to the tags of extended LLC blocks stored in its metadata register $R_M$ (e.g., $R_{32}$ in Figure 8).

Algorithm 1 shows the pseudo code for the tag lookup procedure in the extended LLC kernel. The procedure receives the tag of the extended LLC request as the input and returns the outcome of the lookup ("HIT=True" and the "BLOCK_INDEX" of the corresponding cache block, or "HIT=False") as the output. Algorithm 1 assumes that the request's tag is in the auxiliary register $R_{aux_0}$ (e.g., $R_{33}$ in Figure 8) for all threads, i.e., there are 32 copies of the tag.

Each thread of the warp is assigned to a cache block (Block 0-31 in Figure 8) and corresponding metadata block (M0-M31 in Figure 8), such that all following operations run in parallel for all cache blocks. First, each thread ensures that its assigned metadata block in $R_M$ (e.g., $R_{32}$ in Figure 8) is valid by checking the valid bit (line 2). Second, each thread compares the tag in its assigned metadata block to the request's tag in $R_{aux_0}$, and stores the result in an auxiliary register (e.g., $R_{aux_1}$) (line 3). Third, the comparison result of each thread is shared among all threads as a 32-bit bitvector and written into $R_{aux_2}$ using the *ballot_sync* instruction [20] (line 4). If $R_{aux_2}$ is non-zero, one of the tags in the metadata blocks must have matched the request's tag, i.e., the request is a hit (lines 5-6). In this case, the 0-based index of the first 1-bit in $R_{aux_2}$ is obtained using the *ffs* instruction [20] (line 7). This index is the index of the cache block whose metadata matched the request's tag. Finally, the LRU counters are updated (lines 8-12).

**Handling Extended LLC Hits.** After detecting an extended LLC hit, the corresponding warp in the extended LLC kernel should move the cache block from the register file to the *read data buffer* in the Morpheus controller (§4.1.3). In this section, we explain how the warp in the extended LLC kernel reads the register that contains the requested cache block.

After the *tag lookup* procedure, the corresponding extended LLC kernel warp has the index of the matching cache block available (e.g., in $R_{aux_3}$). To retrieve the matching cache block's data, the extended LLC kernel warp should access the register whose index equals the value of $R_{aux_3}$. For example, if $R_{aux_3}$ is equal to 5, $R_5$ should be accessed. This register file access is *indirect*, i.e., it requires reading from a register whose

**Algorithm 1** Extended LLC Tag Lookup – Register File

**Input: Extended LLC Request's Tag ($R_{aux_0}$)**
**Output: HIT:bool, and if HIT=True, BLOCK_INDEX:int ($R_{aux_3}$)**

```
1:  procedure TAG LOOKUP//executed by an extended LLC kernel warp of 32 threads
2:      R_aux1 ← Valid(R_M) //ensure the block is valid
3:      R_aux1 ← R_aux1 && (R_aux0 == Tag(R_M)) //match request tag to metadata
4:      R_aux2 ← __ballot_sync(0xffffffff, R_aux1) //share R_aux1 between all
        threads as a 32-bit vector
5:      if (R_aux2) then //one of the bits is non-zero because there was a hit
6:          R_aux3 ← __ffs(R_aux2) − 1 //get the 0-based index of the non-zero bit
7:          HIT ← True
8:          BLOCK_INDEX ← R_aux3
9:          if (thread_idx == R_aux3) then //reset the LRU counter of the hit block
10:             LRU_Counter(R_M) ← 0xfff
11:         else//decrement the LRU counters of all other blocks
12:             LRU_Counter(R_M) ← LRU_Counter(R_M) − 1
13:         end if
14:     else
15:         HIT ← False
16:     end if
17: end procedure
```

index is determined by accessing the value in another register. An indirect register file access is not straightforward, because many existing GPU ISAs (e.g., [20]) only provide instructions for accessing the register file with an immediate (constant) index.

To enable indirect register accesses, we define a procedure called *Indirect-MOV*. The key mechanism is to implement a *switch-case* structure in the procedure. The procedure 1) allocates each case to access a specific register index, and 2) selects the corresponding case using the target register index (e.g., the value of $R_{aux_3}$). Algorithm 2 illustrates how we implement the *Indirect-MOV* procedure using the instructions already present in an existing GPU ISA [20].[3] The procedure uses the *brx.idx* instruction [20]. This instruction is a branch instruction that gets a list of branch targets (i.e., $T_{List}$) and an index as input. The control flow jumps to the branch targets at the specified index. The *Indirect-MOV* procedure uses the the target LLC block index (e.g., the value of $R_{aux_3}$) as the input to the *brx.idx* instruction. The procedure defines 32 branch targets ($L_0$-$L_{31}$), each is allocated to access the corresponding data-array register. For example, the branch targets *L0* and *L31* are to access the cache blocks in registers $R_0$ and $R_{31}$, respectively.

**Handling Extended LLC Misses.** The warp in the extended LLC kernel that services the extended LLC request handles an extended LLC miss in four steps. First, the warp accesses main memory to bring the requested cache block. Note that main memory accesses from the extended LLC kernel bypass the conventional LLC. Second, to free space in the extended LLC for the to-be-inserted block, the warp selects the victim extended LLC block based on the *LRU* replacement policy. To this end, the warp determines which extended LLC block has the lowest LRU counter. Third, the warp checks the dirty bit of the victim extended LLC block and writes it back to the main memory if it is dirty. Fourth, the warp writes the new extended LLC block into the register file using the *Indirect-MOV* procedure (Algorithm 2).

[3] We optimize *Indirect-MOV* by adding a new instruction to the ISA in §4.3.

**Algorithm 2** Indirect-MOV Algorithm

**Input: BLOCK_INDEX:int ($R_{aux_3}$)**
**Output: Requested Extended LLC Block ($R_{aux_0}$)**

```
1:  procedure INDIRECT-MOV //The goal is to implement register indirect access, reading from a register whose index is determined by accessing the value in another register. This procedure is critical for accessing data-array registers in the extended LLC kernel.
2:      T_list : .Branch Targets L_0, L_1, L_2, ..., L_31; //Define 32 branch targets, each is allocated to access a specific register index.
3:      @p brx.idx R_aux3, T_list; //Branch to label L_i specified by the target LLC block index i=R_aux3
4:      L_0 :
5:          MOV R_0, R_aux0 //Access data-array register R_0 if target LLC block index is 0
6:          return
7:      L_1 :
8:          MOV R_1, R_aux0 //Access data-array register R_1 if target LLC block index is 1
9:          return
10:         ...
11:     L_31 :
12:         MOV R_31, R_aux0//Access data-array register R_31 if target LLC block index is 31
13:         return
14: end procedure
```

**4.2.2. Extended LLC via Unified L1/Shared-memory.** We explain how the extended LLC kernel uses the L1 cache and shared memory as the extended LLC.

**L1 cache.** Morpheus assigns a portion of the extended LLC to the L1 cache of each GPU core that is in cache mode. When a warp executing the extended LLC kernel receives a request that should be serviced in the L1 cache, the warp simply forwards the request to the L1 cache by executing GPU load and store instructions. If the request hits in the L1 cache, the responsible warp in the extended LLC kernel responds to the Morpheus controller through the extended LLC query logic unit. Otherwise, the L1 cache accesses the main memory to service the miss request. Note that the Morpheus controller ensures that an L1 cache miss from a GPU core that is in cache mode bypasses the conventional LLC and directly accesses main memory.

**Shared Memory.** Morpheus assigns a portion of the extended LLC to the shared memory. Since shared memory does not have a hardware unit for storing the tags (unlike the L1 cache), the extended LLC kernel stores the tags of the extended LLC blocks assigned to the shared memory inside the register file instead. The advantage of this approach is that a register file access is faster than shared memory access, which accelerates the tag lookup procedure. The tag lookup procedure is similar to the procedure shown in Algorithm 1. To access an extended LLC block, the extended LLC kernel calculates the address of the block's data in shared memory based on the extended LLC set number and the cache block index from the tag lookup procedure.

**4.2.3. Supporting Atomic Instructions in the Extended LLC.** Modern GPUs execute global memory atomic operations via single SASS instructions that run on atomic units in the conventional LLCs [31, 32], which is critical for performance of GPU applications with inter-CTA synchronization. Morpheus supports global memory atomic operations in the extended LLC, as we explain next. First, the warp handling an extended LLC request performs atomic operations using the functional units inside the SMs in cache mode, no matter which on-chip memory inside an SM in cache mode has the corresponding block. Second, the Morpheus controller guarantees the atomicity in the extended LLC since several threads cannot access the same extended LLC block at the same time. This is because: (1) each cache block in the extended LLC is assigned to exactly one warp in the extended LLC kernel, and (2) each warp completes one extended LLC request before starting to service another request.

### 4.3. Optimizing Morpheus

We describe two optimization techniques on top of the basic Morpheus design we introduced in §4-4.2. First, the flexibility of the extended LLC kernel enables low-cost implementation of cache optimization techniques, such as cache compression, resizing cache blocks, and online modification of the replacement policy. As a case study, we discuss how to use cache compression in Morpheus (§4.3.1). Second, the extended LLC needs the *Indirect-MOV* procedure that we implement using the *brx.idx* instruction in basic Morpheus (see Algorithm 2). However, having architectural support for this operation can accelerate the data array access. We provide new architectural support for the *Indirect-MOV* instruction in §4.3.2.

**4.3.1. Cache Compression.** Queries to, insertions to, and evictions from the register file and shared memory partitions of the extended LLC *always* go through the extended LLC kernel[4]. Hence, the extended LLC kernel can manipulate them in a way that is transparent to the rest of the system. Our goal is to leverage this opportunity for increased extended LLC capacity. To this end, we propose a cache compression scheme on top of Morpheus. The key idea is to use the extended LLC kernel to store *compressed* (where possible) versions of the extended LLC blocks, thereby increasing the number of blocks in each extended LLC set, and thus the effective capacity of the extended LLC. The extended LLC kernel compresses any inserted extended LLC block, stores the compressed version in the register file or shared memory, and serves requests by decompressing the blocks upon a hit. In this section, we describe our mechanism in detail.

An inserted or updated (i.e., written-to) 128-byte extended LLC block is grouped into one of three *compression levels* that we define as follows: (1) the *high* compression level includes extended LLC blocks that can be compressed 4-fold into 32 bytes, (2) the *low* compression level includes extended LLC blocks that can be compressed 2-fold into 64 bytes, and (3) the *uncompressed* level includes extended LLC blocks that could not be compressed. Figure 9 shows how logical 128-byte extended LLC blocks ❶ are laid out in the register file, depending on their respective compression level ❷❸❹. Blocks in the *high* ❷ and *low* ❸ compression levels are laid out across reg-

[4]Note that insertions to and evictions from the L1 partition of the extended LLC do *not* go through the extended LLC kernel, because the L1 cache handles them in hardware.

isters in a strided and interleaved manner with strides 4 and 2, respectively. Blocks in the *uncompressed* level are laid out exactly like they were without our compression scheme ❹. For example, for the *high* compression level, four extended LLC blocks ❶ are stored in a single warp register of 32×4 bytes ❷, such that they occupy the first, second, third and fourth bytes of each thread, respectively.

Since the compression levels of cache blocks cannot be known ahead of time, the number of warp registers allocated to each compression level should be adapted dynamically. Our mechanism initially assigns all registers to the *uncompressed* level. Then, over epochs of $n$ cycles (we empirically choose $n$=10,000), the number of cache blocks in the *high*, *low* and *uncompressed* levels are counted. At the end of each epoch, the number of registers assigned to each compression level is updated based on the counter values.

We employ the Base-Delta-Immediate (BDI) compression algorithm [33] due to its simplicity and good cache compression ratios. The BDI algorithm works as follows: First, the input cache block is divided into segments (e.g., 4 bytes each). Second, one of these segments (e.g., the first) is designated as the *base* segment and copied to the output. Third, only the *deltas* (arithemtic differences) of the remaining segments from the base segment are copied to the output [33]. The achieved compression ratio depends on how large the deltas are. For example, if all segments are very similar to the base segment, the deltas are small and can be stored in only a few bits for each delta. We consider 4-byte segments in our implementation and store the base segments of compressed blocks in auxiliary registers.

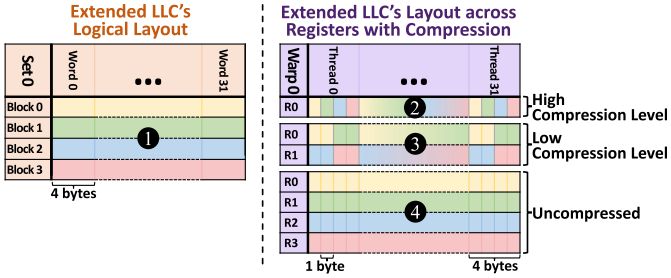


**Figure 9: Layout of compressed cache blocks across registers**

**4.3.2. Indirect-MOV Instruction.** The Indirect-MOV procedure (§4.2.1) is required for indirectly addressing registers in the extended LLC kernel. Our software implementation of the Indirect-MOV procedure (Algorithm 2) is very portable and flexible because it uses only instructions from NVIDIA's existing PTX ISA [20]. However, it is inefficient and slow for two reasons. First, it executes three instructions (i.e., *brx.idx*, *MOV*, and *return*) to perform a single indirect register access. Second, two of these instructions (i.e., *brx.idx* and *return*) are branches, which cause irregular control flow.

To improve the efficiency of the Indirect-MOV procedure, we introduce a new instruction in the GPU ISA that can perform Indirect-MOV natively via minor modifications to existing hardware. Like the software implementation of Indirect-MOV, the new Indirect-MOV instruction conceptually (1) accesses the register file to read the source register $R_{src}$, (2) re-accesses the register file to read the indirectly addressed register $R[R_{src}]$, and (3) moves the value from $R[R_{src}]$ to the destination register $R_{dest}$.

To support this instruction in GPU hardware, we modify the register file architecture. The key idea is to support two *sequential* register file reads for the Indirect-MOV instruction. We slightly modify the operand collectors in the register file to support these sequential reads. The operand collector first accesses the register file using the register number specified in the instruction to read a 1024-bit warp register. Then, the operand collector uses the first eight least significant bits of the read register value as the next register number. The operand collector then re-accesses the register file with this new register number. The read value is then written to the destination register using the regular MOV instruction's data path in the pipeline. Figure 10 illustrates our changes to the baseline operand collector. We add a single multiplexer per operand collector to select between the two different sources of the register number: (1) the immediate source register number from the instruction, (2) the value loaded from a register. The multiplexer is controlled by the ready bit of the register number loaded from a register, i.e., the indirect register number is used as soon as it is available.

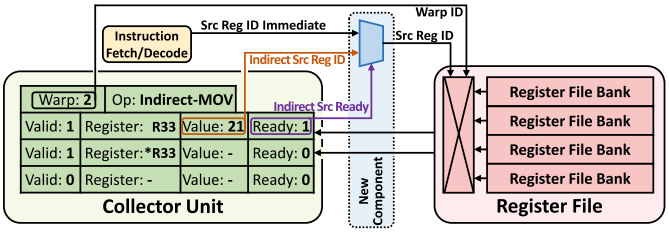


**Figure 10: Native hardware implementation of Indirect-MOV**

## 5. Characterization of the Extended LLC Kernel

We implement the extended LLC kernel, as described in §4.2, and evaluate it on a real GPU. Our goal is to obtain relevant metrics that characterize the extended LLC kernel and the different implementation alternatives (i.e., combinations of register file, L1, and shared memory). We evaluate four relevant metrics for the extended LLC: (1) storage capacity, (2) access latency, (3) access bandwidth, and (4) energy per byte. We use these metrics to (i) determine the implementation alternative that provides the best tradeoff, and (ii) properly configure our simulation of the extended LLC in our cycle-level GPU simulator (see §6), to evaluate the effectiveness of Morpheus at boosting GPU performance of real-world memory-bound applications (§7).

**Methodology.** We implement and evaluate the extended LLC kernel on a real NVIDIA RTX 3080 GPU [7]. Our implementation faithfully follows the description in §4.2. However, since there is no actual Morpheus controller (§4.1) in a real state-of-the-art GPU, we emulate the warp status table (§4.1.3) by

placing a similar data structure, which contains the addresses of the extended LLC to access in our evaluation, in the conventional LLC. The latency of an access to this emulated warp status table is similar to an access to the warp status table in the Morpheus controller, since the Morpheus controller sits inside the LLC partition (see Figure 4).[5]

We implement three variants of the extended LLC kernel: (1) extended LLC via register file, (2) extended LLC via L1, and (3) extended LLC via shared memory. For each of them, we experiment with different numbers of warps of the extended LLC kernel on a single GPU core. Each warp is in charge of one set of the extended LLC. Thus, the larger the number of warps, the smaller the extended LLC sets are (because of the fixed size of memory storage). Each extended LLC variant and number of warps offers a different tradeoff in terms of capacity, latency, bandwidth, and energy/byte.

To measure the extended LLC capacity, we calculate the available space for the extended LLC data array per GPU core in cache mode. This depends on the size of the actual storage (i.e., register file, shared memory, L1) and the space needed for *auxiliary* purposes (e.g., the execution context of extended LLC kernel). To measure the extended LLC access latency, we use the *Nsight* profiler tool [35] and the *cudaEventElapsedTime* API [34]. To measure the extended LLC access bandwidth, we first measure the number of *accesses per second* by dividing the total number of extended LLC accesses (100 million accesses in our experiments) by the total time takes to service all extended LLC accesses. Second, we multiply the resulting accesses per second by 128 (the extended LLC block size in bytes) to calculate the extended LLC bandwidth in bytes per second (B/s). To measure the extended LLC energy per byte, we first measure the average GPU power consumption using *nvidia-smi* [20] while servicing 100 million accesses to the extended LLC. Second, we calculate the total energy consumption by multiplying the measured power consumption with the total time it takes to service these extended LLC requests. Third, we divide the total energy consumption by the number of extended LLC accesses to obtain the energy per access. Fourth, we divide the energy per access by 128 (the extended LLC block size in bytes) to calculate the extended LLC energy per byte (J/B).

**Extended LLC Capacity.** Figure 11(a) reports the extended LLC capacity per GPU core in cache mode for different implementations using various numbers of warps, i.e., 1, 8, 16, 32, and 48. We make four key observations. First, the extended LLC capacity is substantial per GPU core in cache mode. For example, when using 8 warps for the extended LLC via register file (providing 239 KiB capacity) and 8 warps for the extended LLC via L1 (providing 128 KiB capacity), the extended LLC capacity is 367 KiB per GPU core in cache mode.[6] Second, the capacity of the extended LLC via register file varies with the number of warps. This is due to two main reasons. First, using fewer than eight warps, the extended LLC kernel cannot utilize the total register file capacity since the extended LLC capacity is limited to the maximum number of registers per thread (i.e., 256). Third, using eight warps results in the maximum extended LLC capacity via register file (i.e., 239 KiB). Using more than eight warps leads to smaller extended LLCs via register file due to allocating a higher number of registers for *auxiliary* purposes (e.g., the execution context of the extended LLC kernel warps). Fourth, the capacity of the extended LLC via L1 and via shared memory does *not* change with the number of warps. This is because the extended LLC kernel allocates the whole space of each of these two memories (L1 and shared memory) to the extended LLC data array no matter how many warps the kernel uses.

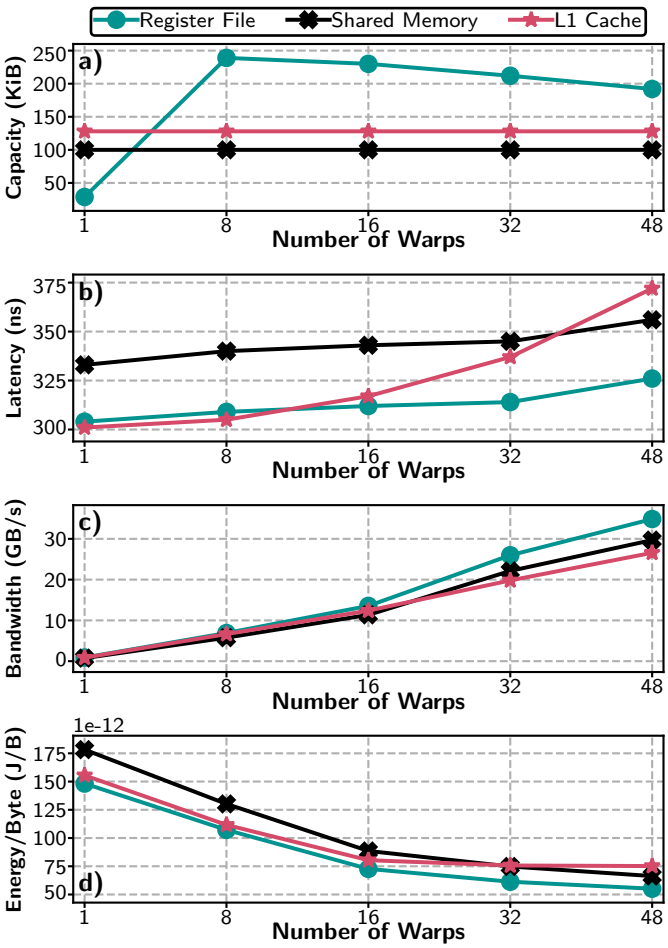


**Figure 11: Characterization of the extended LLC using a real GPU [7]. a) extended LLC capacity, b) extended LLC access latency, c) extended LLC access bandwidth, and d) extended LLC energy per byte.**

[5]We ensure that (1) the data structure that emulates the warp status table resides in the conventional LLC (and not L1 cache) using the *ld.global.cg* instruction [34], and (2) the data structure fits completely in the conventional LLC so that all accesses (after the initialization phase of the extended LLC kernel) to this data structure hit in the conventional LLC.

[6]Note that using shared memory would not further increase the extended LLC capacity in this case. This is because the L1 and shared memory are unified in modern NVIDIA GPUs, i.e., the sum of L1 and shared memory space in a core is at most their total unified storage capacity (e.g., 128 KiB in an NVIDIA RTX 3080 [7]).

**Extended LLC Access Latency and Bandwidth.** Figures 11(b) and 11(c) report the extended LLC access latency and bandwidth, respectively, for different implementations using various numbers of warps, i.e., 1, 8, 16, 32, and 48. We make five key observations. First, the extended LLC access latency (≥300 ns) is almost two times longer than the conventional LLC access latency (∼160 ns [31, 36]). The longer access latency of the extended LLC compared to the conventional LLC is mainly because of the round trip interconnect latency from the Morpheus controller to the GPU core in cache mode and from the GPU core in cache mode to the Morpheus controller. However, the extended LLC access latency is still approximately 2× faster than accessing off-chip memory (∼600ns [31, 36]). Second, the extended LLC access bandwidth per GPU core in cache mode is 37 GB/s using the register file implementation and 48 warps. The bandwidth of each conventional LLC partition is around 300 GB/s [31, 36], and thus eight GPU cores in cache mode can collectively provide the same bandwidth as one conventional LLC partition. Third, increasing the number of warps of the extended LLC kernel in all three implementations results in higher extended LLC bandwidth at the cost of longer access latency. Increased access latency with more warps is mainly due to the fact that the corresponding warp servicing a request to the extended LLC needs to wait until its scheduling slot, and the waiting time becomes longer when using more warps. Fourth, the extended LLC via register file has both lower latency and higher bandwidth for almost all warp counts, compared to the extended LLC via shared memory and via L1. This is because the register file has a lower access latency and a higher access bandwidth compared to shared memory and L1.[7] Fifth, the bandwidth of the extended LLC in the best case (i.e., the extended LLC via register file using 48 warps) is still less than 40 GB/s, which is one order of magnitude lower than the bandwidth of the register file (1 TB/s). This is because the interconnection network connecting LLC partitions to GPU cores significantly bottleneck the bandwidth of the extended LLC via register file (and similarly via shared memory and via L1).

To further analyze the effect of the interconnection network, we *ideally exclude* the interconnection network from the extended LLC accesses. To this end, we (1) assume the address of the extended LLC request is already ready in an *auxiliary* register of the SM operating in cache mode and (2) let the SM operating in cache mode discard the response instead of sending it over the interconnection network. We observe that the access bandwidth of the extended LLC via register file, shared memory, and L1 using 48 warps becomes 290 GB/s, 106 GB/s, and 97 GB/s, which is 7.8×, 3.4×, and 3.5× higher than the *non-ideal* versions, respectively. Hence, a better interconnect design could significantly improve the performance of the extended LLC.

**Extended LLC Energy per Byte.** Figure 11(d) reports energy per byte results for different implementations using various numbers of warps, i.e., 1, 8, 16, 32, and 48. Note that the energy per byte results take into account the energy consumed by all the components included in the extended LLC accesses, i.e., GPU cores in cache mode (executing the extended LLC kernel), interconnect, and the conventional LLC banks. We make three key observations. First, The extended LLC energy per byte in the best case (extended LLC via register file and using 48 warps) is 53 pJ which is approximately 5.3× the energy per byte of the conventional LLC (∼10 pJ [37]). Despite its high energy consumption compared to the conventional LLC, the extended LLC can reduce GPU energy consumption by reducing the number of energy-hungry off-chip memory accesses. Second, increasing the number of warps in the extended LLC kernel significantly reduces energy per byte for all implementations. This is because using more warps increases the extended LLC's throughput, without significantly increasing power consumption, leading to lower energy per byte. Third, the extended LLC via register file has a lower energy per byte for all warp counts compared to the extended LLC via L1 and via shared memory. This is mainly due to the fact that a register file access consumes less energy compared to an access to the L1 or shared memory [37].

**Combining different extended LLC versions.** Our characterization of the extended LLC shows that the extended LLC via register file outperforms other implementations in terms of capacity, access latency, access bandwidth, and energy per byte. However, to utilize all on-chip memories of a GPU core in cache mode and thus enable larger extended LLC capacities, the extended LLC kernel aims to combine the implementation via register file with the implementation via L1 and/or via shared memory. To this end, the extended LLC kernel allocates a number of warps to the implementation via register file, and the remaining number of warps to the implementations via L1 and shared memory. We only consider the combination of the extended LLC via register file and via L1 since L1 and shared memory are unified in our evaluated GPU system (i.e., combining the extended LLC via L1 and via shared memory does *not* provide a larger extended LLC capacity).

We use our characterization results to optimize the number of warps the extended LLC kernel should allocate to each of the two extended LLC kernel implementations (via register file and via L1) to design a high-capacity and high-performance extended LLC. (1) The extended LLC kernel should allocate more than 8 warps to the implementation via register file to better utilize the register file capacity (Figure 11(a)). (2) The extended LLC kernel should allocate fewer than 48 warps to the implementation via register file to be able to allocate a number of warps to the implementation via L1. (3) Among the remaining options (8, 16, or 32 warps), using 32 warps in the extended LLC kernel via register file results in higher access bandwidth (Figure 11(c)) and lower energy per byte (Figure 11(d)), while the access latency is almost equal to the case of using 8 warps (Figure 11(b)). Hence, the extended LLC kernel combines the extended LLC via register file and via L1

[7] The access latency of the register file, shared memory, and L1 are 2, 25, and 34 ns, respectively [8]. The access bandwidth of the register file, shared memory, and L1 are 1 TB/s, 170 GB/s, and 170 GB/s, respectively [8, 31, 36].

by allocating 32 and 16 warps to each of these two implementations, respectively. Using our characterization, we observe that for each GPU core operating in cache mode, the extended LLC via *register file+L1* has 328 KiB capacity, 185ns average access latency, 34GB/s average access bandwidth, and 61pJ average energy per byte. We use this extended LLC implementation to evaluate the effectiveness of Morpheus to boost the performance of memory-bound applications.

## 6. Methodology

**Simulation methodology.** We evaluate Morpheus using the AccelSim [8] cycle-level simulator. We model our baseline after the NVIDIA Ampere 3080 GPU [7]. Table 1 shows the simulation parameters modeling our baseline. To evaluate energy consumption, we use AccelWattch [37] embedded in AccelSim as the state-of-the-art GPU energy model.

**Table 1: Baseline GPU configuration**

| Parameter | Value |
|---|---|
| Number of SMs | 68 |
| Scheduler | Two-Level [18, 38] |
| GPU Memory Interface | 320-bit GDDR6X [39] |
| GPU Memory Capacity | 10 GiB |
| Conventional LLC Capacity | 5 MiB |
| L1/Shared-Memory Capacity | 128 KiB per SM |
| Register File Capacity | 256 KB per SM |

**Applications.** We randomly choose 14 *memory-bound* and 3 *compute-bound* applications from four benchmark suites, Rodinia [5], Parboil [6], Pannotia [40] and ISPASS [41]. Our methodology to categorize applications into memory-bound and compute-bound groups is based on our experiment in §3. The performance of the compute-bound applications to increases (linearly) with more GPU cores. In contrast, the performance of the memory-bound applications either saturates or decreases sharply after a certain number of GPU cores. Table 2 shows the applications we choose, their names, and their types (memory-bound or compute-bound). We run each application either entirely, or until the application reaches two billion executed instructions.

**Table 2: Evaluated Applications**

| Application | Name | Type |
|---|---|---|
| Breadth-First Search [6] | p-bfs | Memory-bound |
| Computational fluid dynamics [5] | cfd | Memory-bound |
| Discrete Wavelet Transform (2D) [5] | dwt2d | Memory-bound |
| Stencil [6] | stencil | Memory-bound |
| Breadth-First Search [5] | r-bfs | Memory-bound |
| Back Propagation [5] | bprob | Memory-bound |
| sgemm [6] | sgem | Memory-bound |
| Needleman-Wunsch [5] | nw | Memory-bound |
| Page Rank [40] | page-r | Memory-bound |
| K-means [5] | kmeans | Memory-bound |
| Histogram [6] | histo | Memory-bound |
| Magnteic Resonance Imaging-Gridding [6] | mri-gri | Memory-bound |
| Sparse-Matrix Dense-Vector Multiplication [6] | spmv | Memory-bound |
| Lattice-Boltzmann [6] | lbm | Memory-bound |
| LIBOR Monte Carlo [41] | lib | Compute-bound |
| HotSpot [5] | hotsp | Compute-bound |
| Magnetic Resonance Imaging - Q [6] | mri-q | Compute-bound |

**Evaluated Systems.** We evaluate six systems. (1) *BL*: baseline system that models a GPU architecture with the parameters reported in Table 1. *BL* employs all available GPU cores (i.e., 68) for application execution. To provide a fair comparison, we add the extra on-chip storage in Morpheus, the 16-KiB Bloom filters (§4.1.2) and the 5-KiB extended LLC query logic unit (§4.1.3) per LLC partition (overall; 21 KiB×#partitions = 210 KiB), to the conventional LLC capacity. (2) *IBL*: improved baseline system where we use the number of GPU cores that provides the maximum performance for each application and power-gate the remaining cores. Table 3 (second row) reports the number of GPU cores we use for each application in *IBL*. (3) *IBL-4×-LLC*: *IBL* with 4× the LLC size. We increase the total LLC capacity by increasing the number of the LLC banks (without adding any latency and power impact). (4) *Frequency-Boost*: *IBL* with higher frequency memory system components, including the interconnection network, conventional LLC, and off-chip DRAM channels. This system uses the energy saved by power-gated GPU cores in *IBL* to increase the clock frequency of the aforementioned components in the GPU memory system by 10%-20% depending on the number of power-gated cores. (5) *Unified-SM-Mem*: *IBL* that has a larger L1 data cache capacity by using methods from prior works on unifying L1 data cache, shared memory, and the register file [42, 43]. Our baseline architecture (*BL*) already unifies L1 data cache and shared memory. In *Unified-SM-Mem*, we add the amount of unused register file space to the L1 data cache (without additional latency impact). (6) Different versions of Morpheus with and without our various optimizations, namely *Morpheus-Basic*, *Morpheus-Indirect-MOV*, *Morpheus-Compression*, and *Morpheus-ALL*. For all Morpheus variants, we determine the number of GPU cores in cache mode that results in the highest performance per application offline.[8] Table 3 (third and fourth rows) reports the number of GPU cores in compute mode for *Morpheus-Basic* and *Morpheus-ALL*. We observe that *Morpheus-ALL* uses a larger number of GPU cores in compute mode because it employs cache compression (§4.3.1), which enables larger extended LLC capacities per SM in cache mode, which in turn enables higher performance with more application threads.

## 7. Evaluation

We evaluate the effectiveness of Morpheus compared to different baselines. §7.1 shows the overall effect of Morpheus on GPU performance. §7.2 evaluates the effect of Morpheus on GPU energy efficiency. §7.3 analyzes the effectiveness of our Bloom filter-based hit/miss predictor design. §7.4 shows the effect of Morpheus on on-chip and off-chip bandwidth utilization. §7.5 studies the storage and power overheads of Morpheus.

[8] This is a static version of Morpheus in which we adjust the number of SMs in cache mode before running the application. If the best configuration per application is not known prior to execution, we could use online profiling techniques similar to prior work [44-46] on top of Morpheus, to learn the best configuration for a running application and dynamically adjust the number of cores in cache mode. We leave the use of online profiling techniques with Morpheus to future work.

**Table 3: Number of GPU cores executing application threads for different evaluated systems (#available GPU cores is 68).**

| Application | p-bfs | cfd | dwt2d | stencil | r-bfs | bprob | sgem | nw | page-r | kmeans | histo | mri-gri | spmv | lbm | lib | hotsp | mri-q |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **IBL** | 68 | 68 | 68 | 68 | 68 | 68 | 68 | 68 | 68 | 24 | 53 | 34 | 42 | 34 | 68 | 68 | 68 |
| **Morpheus-Basic** | 32 | 42 | 42 | 50 | 34 | 39 | 48 | 18 | 42 | 37 | 47 | 36 | 44 | 32 | 68 | 68 | 68 |
| **Morpheus-ALL** | 40 | 55 | 54 | 56 | 37 | 41 | 54 | 26 | 46 | 47 | 52 | 43 | 47 | 36 | 68 | 68 | 68 |

### 7.1. Performance Analysis

To study the effectiveness of Morpheus at improving GPU performance, we measure application execution times on *nine* different systems, namely *BL*, *IBL*, *IBL-4×-LLC*, *Frequency-Boost*, *Unified-SM-Mem*, *Morpheus-Basic*, *Morpheus-Compression*, *Morpheus-Indirect-MOV*, and *Morpheus-ALL* (see §6 for details of each evaluated system). Figure 12 (top) shows the results. The x-axis shows applications in two groups, memory-bound and compute-bound. The y-axis shows the application execution time (the lower the better) normalized to the baseline (*BL*) system.

We make five key observations. First, Morpheus (i.e., *Morpheus-ALL*) improves performance greatly with all its optimizations over all real baselines and across all memory-bound applications. Specifically, Morpheus significantly improves GPU performance by an average of 27% over the best real baseline (i.e., *Unified-SM-Mem*) and by 39%, 32%, and 29% over *BL*, *IBL*, and *Frequency-Boost*, respectively. Second, Morpheus performs within 3% of an ideal baseline with 4× the LLC (i.e., *IBL-4×-LLC*) at only small overhead (see §7.5). Third, *Morpheus-Compression* improves performance by an average of 9% over *Morpheus-Basic*, by providing a larger extended LLC capacity. Fourth, *Morpheus-Indirect-MOV* improves performance by an average of 4% over *Morpheus-Basic*, by providing a lower extended LLC access latency. Fifth, Morpheus does *not* affect the performance of compute-bound applications since all GPU cores stay in compute mode for such applications. We conclude that Morpheus is highly effective at improving the performance of memory-bound GPU applications.

### 7.2. Energy Efficiency Analysis

To study the effect of Morpheus on GPU energy efficiency, we calculate GPU *performance/watt* by dividing the overall GPU IPC by GPU average power consumption for each of the nine systems we evaluate. Figure 12 (bottom) reports the results, where the y-axis shows performance/watt (the higher the better). We normalize the results to the performance/watt of the baseline (*BL*) system.

We make four key observations. First, Morpheus (i.e., *Morpheus-ALL*) improves GPU energy efficiency greatly over all real baselines and across all memory-bound applications. Morpheus provides 58%, 38%, 35%, and 33% better energy efficiency compared to *BL*, *IBL*, *Unified-SM-Mem*, and *Frequency-Boost*, on average, respectively. Morpheus' energy efficiency improvement is due to (1) reducing the number of energy-hungry off-chip memory accesses and (2) Morpheus' large speedups over all real baselines (§7.1). Second, the energy efficiency of the *Morpheus-ALL* system is within 6% of that of the *IBL-4×-LLC* system, for which we *ideally* assume *no* power and latency impact while using a 4× larger LLC. Third, *Morpheus-Compression* and *Morpheus-Indirect-MOV* improve the energy efficiency of *Morpheus-Basic* by 8% and 5%, on average, respectively. Fourth, Morpheus slightly reduces the performance/watt of compute-bound applications (less than 1%) compared to the baseline (*BL*) system due to the power consumption overhead of the Morpheus controller (see §7.5 for our overhead analysis). A more optimized Morpheus can power-gate the Morpheus controller completely for compute-bound applications to avoid the power overhead for such workloads [47-50]. We conclude that Morpheus is highly effective at improving the energy efficiency of memory-bound GPU applications.

### 7.3. Effect of Hit/Miss Prediction

To study the effectiveness of Morpheus' hit/miss prediction technique at improving Morpheus' performance, we compare the execution time of the 14 memory-bound applications on a *Morpheus-Basic*-enabled GPU with three hit/miss prediction designs: (1) our *Bloom-Filter* design (§4.1.2), (2) *No-Prediction*, where we disable the hit/miss prediction technique and immediately forward all extended LLC requests to the extended LLC, and (3) *Perfect-Prediction*, where we assume 100% accuracy for the hit/miss prediction technique. Figure 13 reports the execution time results normalized to the baseline system (*BL*). We make two key observations. First, *No-Prediction* has a 9% higher execution time compared to the *Bloom-Filter* design, on average. Second, *Bloom-Filter*'s results are within 1% of *Perfect-Prediction*. We conclude that hit/miss prediction is important for Morpheus' performance and that our predictor design is very effective.

### 7.4. On-Chip & Off-Chip Bandwidth Analysis

To further analyze the sources of Morpheus' performance improvement, we study the effect of Morpheus on (1) *LLC throughput*, (2) *interconnect performance*, and (3) *off-chip bandwidth utilization*.

**LLC throughput.** In this study, we aim to measure (1) by how much Morpheus improves the LLC throughput and (2) determine the reasons for the LLC throughput increase. To this end, we measure the LLC (both the conventional LLC and extended LLC) throughput for four systems, *BL*, *IBL*, *Morpheus-ALL*, and *larger-LLC*. The goal of evaluating the *larger-LLC* system is to distinguish between two different benefits of Morpheus on LLC (larger LLC capacity and higher number of LLC banks). In *larger-LLC*, we increase the conventional LLC size to be exactly the same as the total LLC capacity in *Morpheus-ALL* (i.e., conventional LLC + extended LLC) without increasing the number of conventional LLC banks. This is to isolate the

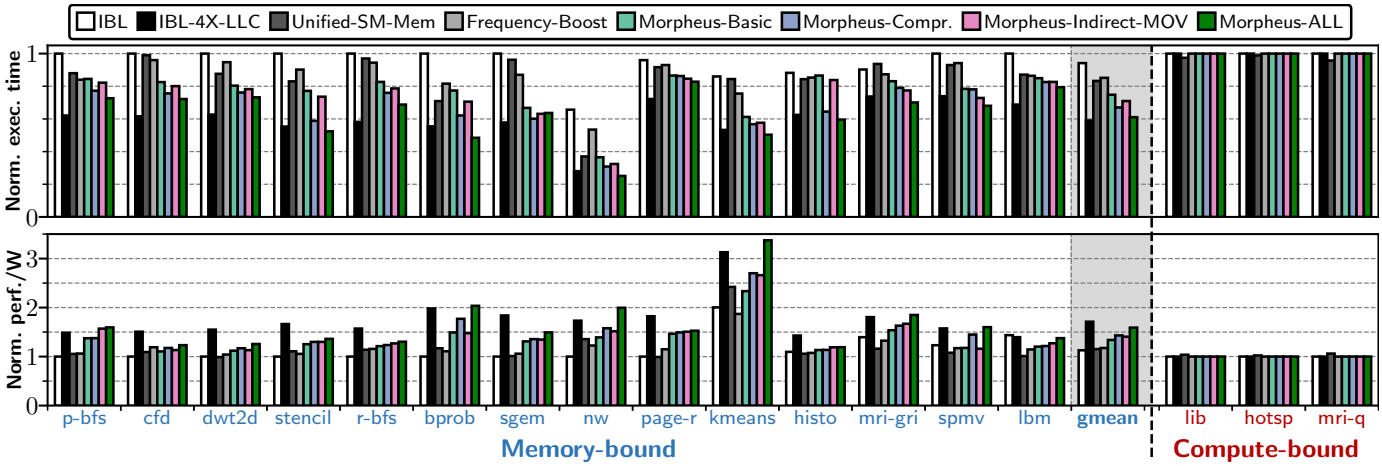


**Figure 12: Comparison of eight GPU systems' execution time (top) and performance/watt (bottom) for 14 memory-bound and 3 compute-bound applications, normalized to the baseline system (BL)**

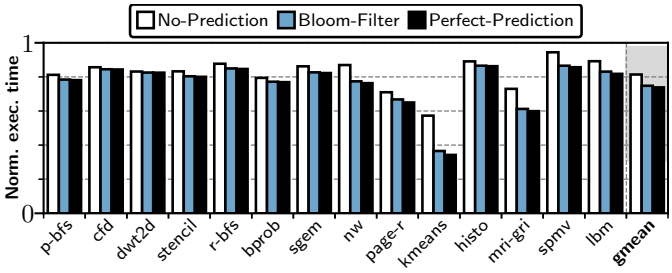


**Figure 13: Effect of hit/miss prediction on execution time for 14 memory-bound applications with *Morpheus-Basic*, normalized to the baseline system (BL)**

effect of an increased LLC capacity (relative to *BL*) from the effect of an increased number of banks. The LLC capacity of *larger-LLC* varies between applications, as it depends on the number of cores operating in cache mode in *Morpheus-ALL*, which is determined on a per application basis.

We make two key observations. First, *Morpheus-ALL* improves the LLC throughput by an average of 75% and 68% (up to 374% and 236%) compared to *BL* and *IBL*, respectively. Second, the *larger-LLC* system improves LLC throughput by an average of 42% compared to *BL*. We conclude that *Morpheus-ALL*'s higher throughput comes from *both* (1) increasing the LLC capacity and (2) increasing the number of banks.

**Interconnect Performance.** Morpheus increases the load on the interconnection network by servicing the extended LLC requests through it. We study the performance of the GPU interconnection network in Morpheus. We measure the overall network injection rate, network throughput, and average interconnect latency for two designs, *BL* and *Morpheus-ALL*. We make three key observations. First, *Morpheus-ALL* increases the load of the on-chip network by 97% compared to *BL*, on average. Second, both network injection rate and throughput increase in *Morpheus-ALL* by the same amounts, showing that the GPU interconnection network does *not* saturate due to handling more traffic. Third, the higher load causes 7% longer average network latency compared to *BL*. We observe *no* overall application performance loss due to the longer average network latency.

**Off-chip Bandwidth Utilization.** To study the effect of Morpheus on off-chip bandwidth utilization, we measure the off-chip bandwidth utilization for two systems, *IBL* and *Morpheus-ALL*. We observe that *Morpheus-ALL* reduces off-chip bandwidth utilization by an average of 17% compared to *IBL*. This is mainly due to the fact that the larger LLC enabled by Morpheus reduces the number of off-chip main memory requests. To further analyze the reason behind the bandwidth utilization reduction, we measure the LLC MPKI (misses per kilo instructions) for both *IBL* and *Morpheus-ALL*. We observe that *Morpheus-ALL* reduces the LLC MPKI by 47% compared to *IBL*.

### 7.5. Overhead Analysis

We analyze the storage and power overheads of the additional hardware required by Morpheus (i.e., the Morpheus controller).
**Storage cost.** The Morpheus controller has two main storage components: the hit/miss prediction unit (§4.1.2) and the extended LLC query unit (§4.1.3). The hit/miss prediction unit has 16-KiB of Bloom filter storage per LLC partition. The extended LLC query unit has a 5-KiB on-chip storage per LLC partition to store the request queue, warp status table, and read/write data buffers (§4.1.3). Overall, Morpheus adds 21 KiB per LLC partition; which is approximately 4% of the conventional LLC capacity per LLC partition in the NVIDIA RTX 3080 GPU.

**Power Consumption.** We measure the power consumption of the additional hardware required by Morpheus using (1) CACTI 6.5 [51] for storage units and (2) the synthesized Verilog HDL models with the NanGate 45nm open cell library [52] for logic units. We observe that the Morpheus controller imposes a 0.93% overhead to the total GPU power consumption. Note that we already take into account Morpheus' power consumption overhead for the energy efficiency analysis in §7.2.

## 8. Related Work

To our knowledge, this is the first work to propose extending the GPU last-level cache capacity by repurposing the on-chip memory units (i.e., register files, L1 caches, scratchpad memories) of otherwise unused GPU cores. In this section, we briefly review related work in four GPU domains: (1) *increasing cache capacity*, (2) *increasing interconnection network performance*, (3) *controlling cache contention*, and (4) *helper threads*.

**Increasing Cache Capacity.** Prior works increase the capacity of the GPU L1 cache (e.g., [22, 33, 42, 43, 53-71]) by (1) utilizing unused registers in the register file [22, 43], (2) unifying on-chip memories in the GPU core [42, 53, 54], (3) applying cache compression techniques [33, 55-59, 70, 72], (4) caching cooperatively using multiple instances of the L1 cache [60-62, 64-66], and (5) using dense emerging memory technologies, e.g., Domain Wall Memory (DWM) [63]. Since the L1 cache is one of the on-chip memory units that Morpheus repurposes to increase the GPU LLC capacity, we expect these works can further increase the benefits of Morpheus by providing a higher extended LLC capacity.

Prior works increase the GPU LLC capacity (e.g., [63, 73-76]) by using dense emerging memory technologies, such as STT-MRAM and DWM, to build a conventional LLC with a larger capacity. Compared to these works, Morpheus increases the GPU LLC capacity without (1) adding any extra on-chip memory, and (2) depending on emerging memory technologies.

Some existing GPUs have high LLC capacity. For example, the NVIDIA A100 features a 40-MiB LLC [77]. Although returns diminish, Morpheus can still improve the performance of such a GPU by further increasing the LLC capacity. Since such a large conventional LLC costs significant silicon area (e.g., about 27% of the total chip area in the A100 [77]), Morpheus can help to 1) *reduce* the conventional LLC size in such large-LLC architectures and 2) enable allocating more hardware resources to compute units.

**Improving Interconnection Network Performance.** Prior works (e.g., [78-90]) increase the GPU LLC throughput by improving interconnection network performance, e.g., by improving network resources [78-86], better distributing the LLC banks inside the network topologies (e.g., 2D mesh) [87, 88], and duplicating frequently-accessed cache blocks to reduce network contention [89]. Morpheus relies on the GPU's interconnection network to enable the extended LLC, and therefore, Morpheus' effectiveness can be improved by increasing the performance of the interconnection network.

**Controlling Cache Contention.** Prior works improve the effectiveness of GPU caches by controlling cache contention (e.g., [1, 24, 91-107]). These works either (1) bypass some levels of the cache hierarchy for some memory accesses [1, 91-98], or (2) throttle threads to control cache thrashing [24, 99-106, 108]. Morpheus can be combined with these techniques to enable even higher performance.

**Helper Threads.** Helper threads assist the execution of the main applications threads, by using idle cores or idle cycles for various purposes. Prior works (e.g., [4, 109-130]) propose helper threads to implement optimization techniques for both CPUs and GPUs, such as data prefetching, pre-computing branch outcomes, managing the caches, or increasing the effective cache bandwidth and capacity by compressing the cache contents. In particular, CABA [4] enables each GPU warp to launch an *assist* warp on the same GPU core to perform data compression, exploiting fine-grained idleness of execution units. Morpheus uses helper threads for a new purpose: extending the capacity of the LLC by exploiting on-chip memories of otherwise unused cores.

## 9. Conclusion

We introduce Morpheus, the first hardware/software co-designed technique to repurpose otherwise unused GPU cores' on-chip memories to extend the total GPU last-level cache capacity. Morpheus introduces two execution modes for GPU cores: (1) compute mode, where the core behaves exactly like in a conventional GPU, and (2) cache mode, where the core lends its on-chip memory space (register file, L1 cache, shared memory) to extend the effective total LLC size. Morpheus reuses the on-chip memory resources of a core in cache mode using a software helper kernel. Morpheus improves the performance and energy efficiency of a baseline NVIDIA RTX 3080 architecture by an average of 39% and 58%, respectively, across 14 memory-bound applications. Morpheus performs within 3% of an ideal baseline with 4× the LLC, while increasing GPU power consumption by only 0.93%. We hope that Morpheus can help researchers and system designers to rethink how the large on-chip memory resources of GPUs and other accelerators are managed by software and hardware cooperatively.

## Acknowledgments

We thank the anonymous reviewers of MICRO 2022 for their encouraging feedback. We thank HPCAN and SAFARI Research Group members for their feedback. SAFARI Research Group acknowledges the generous gifts provided by our industrial partners: Google, Huawei, Intel, Microsoft, and VMware.

## References

[1] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient GPU computing," in *MICRO*, 2014.

[2] N. Nematollahi, M. Sadrosadati, H. Falahati, M. Barkhordar, and H. Sarbazi-Azad, "Neda: Supporting direct inter-core neighbor data exchange in GPUs," *IEEE CAL*, 2018.

[3] N. Nematollahi, M. Sadrosadati, H. Falahati, M. Barkhordar, M. P. Drumond, H. Sarbazi-Azad, and B. Falsafi, "Efficient nearest-neighbor data sharing in GPUs," *ACM TACO*, 2020.

[4] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps," in *ISCA*, 2015.

[5] 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 *IISWC*, 2009.

[6] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[7] NVIDIA, "A102 (RTX 3080) architecture whitepaper - NVIDIA file downloads," 2021. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[8] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *ISCA*, 2020.

[9] X. Zhu, M. Awatramani, D. Rover, and J. Zambreno, "Onac: Optimal number of active cores detector for energy efficient GPU computing," in *ICCD*, 2016.

[10] J. Tan and X. Fu, "RISE: Improving the streaming processors reliability against soft errors in GPGPUs," in *PACT*, 2012.

[11] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," in *ISCA*, 2014.

[12] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, "Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs," in *IPDPS*, 2019.

[13] P. Aguilera, K. Morrow, and N. S. Kim, "QoS-aware dynamic resource allocation for spatial-multitasking GPUs," in *ASP-DAC*, 2014.

[14] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *HPCA*, 2012.

[15] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou, "Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls," in *HPCA*, 2018.

[16] X. Zhao, Z. Wang, and L. Eeckhout, "HeteroCore GPU to exploit TLP-resource diversity," *IEEE TPDS*, 2019.

[17] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting core criticality for enhanced GPU performance," in *SIGMETRICS*, 2016.

[18] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *MICRO*, 2011.

[19] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970.

[20] NVIDIA, "CUDA C++ programming guide," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[21] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, "LTRF: Enabling high-capacity register files for GPUs via hardware/software cooperative register prefetching," in *ASPLOS*, 2018.

[22] Y. Oh, G. Koo, M. Annavaram, and W. W. Ro, "Linebacker: Preserving victim cache lines in idle register files of GPUs," in *ISCA*, 2019.

[23] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-register parameter caching for dynamic neural nets with virtual persistent processor specialization," in *MICRO*, 2018.

[24] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *MICRO*, 2015.

[25] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in GPUs," in *MICRO*, 2016.

[26] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an energy-efficient dram system for GPUs," in *HPCA*, 2017.

[27] M. Zhu, Y. Zhuo, C. Wang, W. Chen, and Y. Xie, "Performance evaluation and optimization of HBM-enabled GPU for data-intensive applications," *IEEE TVLSI*, 2018.

[28] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *HPCA*, 2015.

[29] J. Zhao and Y. Xie, "Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration," in *ICCAD*, 2012.

[30] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM TON*, 2000.

[31] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVidia Turing T4 GPU via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[32] M. Mantor, "AMD Radeon™ HD 7970 with graphics core next (GCN) architecture," in *HCS*, 2012.

[33] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012.

[34] NVIDIA. (August 2022) CUDA toolkit documentation, v11.7. [Online]. Available: https://docs.nvidia.com/cuda/index.html

[35] K. Iyer and J. Kiel, "GPU debugging and Profiling with NVIDIA Parallel Nsight," *Game Development Tools*, 2016.

[36] D. Yan, W. Wang, and X. Chu, "Optimizing batched winograd convolution on GPUs," in *PPoPP*, 2020.

[37] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "AccelWattch: A power modeling framework for modern GPUs," in *MICRO*, 2021.

[38] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *ISCA*, 2011.

[39] Micron. (May 2021) 8Gb: 2 Channel x16/x8 GDDR6X SGRAM. [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/gddr/gddr6/gddr6x_sgram_8gb_brief.pdf?rev=161547726f0b45239d3da37ef29b09bf

[40] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *IISWC*, 2013.

[41] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[42] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *MICRO*, 2012.

[43] N. Jing, J. Wang, F. Fan, W. Yu, L. Jiang, C. Li, and X. Liang, "Cache-emulated register file: An integrated on-chip memory architecture for high performance GPGPUs," in *MICRO*, 2016.

[44] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming," in *ISCA*, 2016.

[45] L.-W. Chang, H.-S. Kim, and W.-m. W. Hwu, "DySel: Lightweight dynamic selection for kernel-based data-parallel programming model," in *ASPLOS*, 2016.

[46] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic resource management for efficient utilization of multitasking GPUs," in *ASPLOS*, 2017.

[47] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural techniques for power gating of execution units," in *ISLPED*, 2004.

[48] O. Kayiran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "$\mu$c-states: Fine-grained GPU datapath power management," in *PACT*, 2016.

[49] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for GPGPUs," in *MICRO*, 2013.

[50] M. Sadrosadati, S. B. Ehsani, H. Falahati, R. Ausavarungnirun, A. Tavakkol, M. Abaee, L. Orosa, Y. Wang, H. Sarbazi-Azad, and O. Mutlu, "ITAP: Idle-time-aware power management for GPU execution units," *ACM TACO*, 2019.

[51] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP laboratories*, 2009.

[52] J. Knudsen, "Nangate 45nm open cell library," *CDNLive, EMEA*, 2008.

[53] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," in *ISCA*, 2015.

[54] S. Darabi, E. Yousefzadeh-Asl-Miandoab, N. Akbarzadeh, H. Falahati, P. Lotfi-Kamran, M. Sadrosadati, and H. Sarbazi-Azad, "OSM: Off-chip shared memory for GPUs," *IEEE TPDS*, 2022.

[55] S. Sardashti, A. Seznec, and D. A. Wood, "Yet another compressed cache: A low-cost yet effective compressed cache," *ACM TACO*, 2016.

[56] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *MICRO*, 2013.

[57] A. Arelakis and P. Stenstrom, "SC 2: A statistical compression cache scheme," in *ISCA*, 2014.

[58] A. Ghasemazar, P. Nair, and M. Lis, "Thesaurus: Efficient cache compression via dynamic clustering," in *ASPLOS*, 2020.

[59] A. Ghasemazar, M. Ewais, P. Nair, and M. Lis, "2DCC: Cache compression in two dimensions," in *DATE*, 2020.

[60] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *ISCA*, 2006.

[61] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *HPCA*, 2009.

[62] S. Dublish, V. Nagarajan, and N. Topham, "Cooperative caching for GPUs," *ACM TACO*, 2016.

[63] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "STAG: Spintronic-tape architecture for GPGPU cache hierarchies," in *ISCA*, 2014.

[64] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.

[65] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *ISCA*, 2006.

[66] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *ICS*, 2007.

[67] B. Li, J. Sun, M. Annavaram, and N. S. Kim, "Elastic-Cache: GPU cache architecture for efficient fine- and coarse-grained cache-line management," in *IPDPS*, 2017.

[68] M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh, and A. Jog, "Analyzing and leveraging decoupled L1 caches in GPUs," in *HPCA*, 2021.

[69] B. Li, J. Wei, J. Sun, M. Annavaram, and N. S. Kim, "An efficient GPU cache architecture for applications with irregular memory access patterns," *ACM TACO*, 2019.

[70] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and HPC workloads on GPUs," in *ISCA*, 2020.

[71] D. Wang and W. Xiao, "A reuse distance based performance analysis on GPU L1 data cache," in *IPCCC*, 2016.

[72] A. Arunkumar, S.-Y. Lee, V. Soundararajan, and C.-J. Wu, "LATTE-CC: Latency tolerance aware adaptive cache compression management for energy efficient GPUs," in *HPCA*, 2018.

[73] M. H. Samavatian, H. Abbasitabar, M. Arjomand, and H. Sarbazi-Azad, "An efficient STT-RAM last level cache architecture for GPUs," in *DAC*, 2014.

[74] N. Jing, L. Jiang, T. Zhang, C. Li, F. Fan, and X. Liang, "Energy-efficient eDRAM-based on-chip storage architecture for GPGPUs," *IEEE TC*, 2015.

[75] M. H. Samavatian, M. Arjomand, R. Bashizade, and H. Sarbazi-Azad, "Architecting the last-level cache for GPUs using STT-RAM technology," *ACM TODAES*, 2015.

[76] J. Zhan, O. Kayıran, G. H. Loh, C. R. Das, and Y. Xie, "OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures," in *MICRO*, 2016.

[77] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, 2021.

[78] X. Zhao, Y. Liu, A. Adileh, and L. Eeckhout, "LA-LLC: Inter-core locality-aware last-level cache to exploit many-to-many traffic in GPGPUs," *IEEE CAL*, 2016.

[79] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo, and L. Eeckhout, "Get out of

the valley: Power-efficient address mapping for GPUs," in *ISCA*, 2018.
[80] J. Wang, L. Jiang, J. Ke, X. Liang, and N. Jing, "A sharing-aware L1. 5D cache for data reuse in GPGPUs," in *ASP-DAC*, 2019.
[81] A. Mirhosseini, M. Sadrosadati, B. Soltani, H. Sarbazi-Azad, and T. F. Wenisch, "BiNoCHS: Bimodal network-on-chip for CPU-GPU heterogeneous systems," in *NOCS*, 2017.
[82] A. Mirhosseini, M. Sadrosadati, F. Aghamohammadi, M. Modarressi, and H. Sarbazi-Azad, "BARAN: Bimodal adaptive reconfigurable-allocator network-on-chip," *ACM TOPC*, 2019.
[83] X. Zhao, S. Ma, Y. Liu, L. Eeckhout, and Z. Wang, "A low-cost conflict-free NoC for GPGPUs," in *DAC*, 2016.
[84] X. Zhao, D. Kaeli, Z. Wang, L. Eeckhout *et al.*, "Intra-cluster coalescing and distributed-block scheduling to reduce GPU NoC pressure," *IEEE TC*, 2019.
[85] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for manycore accelerators," in *MICRO*, 2010.
[86] H. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Providing cost-effective on-chip network bandwidth in GPGPUs," in *ICCD*, 2012.
[87] M. Sadrosadati, A. Mirhosseini, S. Roozkhosh, H. Bakhishi, and H. Sarbazi-Azad, "Effective cache bank placement for GPUs," in *DATE*, 2017.
[88] H. Jang, J. Kim, P. Gratz, K. H. Yum, and E. J. Kim, "Bandwidth-efficient on-chip interconnect designs for GPGPUs," in *DAC*, 2015.
[89] X. Zhao, M. Jahre, and L. Eeckhout, "Selective replication in memory-side GPU caches," in *MICRO*, 2020.
[90] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in *PACT*, 2012.
[91] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal, "Adaptive and transparent cache bypassing for GPUs," in *SC*, 2015.
[92] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *MICRO*, 2012.
[93] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *ICS*, 2012.
[94] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *HPCA*, 2015.
[95] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing," in *ICS*, 2015.
[96] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *HPCA*, 2014.
[97] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting inter-warp heterogeneity to improve GPGPU performance," in *PACT*, 2015.
[98] M. Rhu, M. B. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *MICRO*, 2013.
[99] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and fair multiprogramming in GPUs via effective bandwidth management," in *HPCA*, 2018.
[100] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO*, 2012.
[101] T. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *MICRO*, 2013.
[102] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *PACT*, 2013.
[103] B. Wang, Y. Zhu, and W. Yu, "OAWS: Memory occlusion aware warp scheduling," in *PACT*, 2016.
[104] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *HPCA*, 2015.
[105] M. Mao, J. Hu, Y. Chen, and H. Li, "VWS: A versatile warp scheduler for exploring diverse cache localities of GPGPU applications," in *DAC*, 2015.
[106] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU concurrency in heterogeneous architectures," in *MICRO*, 2014.
[107] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-aware CTA clustering for modern GPUs," in *ASPLOS*, 2017.
[108] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in GPU," in *ISCA*, 2017.
[109] S. Darabi, N. Mahani, H. Baxishi, E. Yousefzadeh-Asl-Miandoab, M. Sadrosadati, and H. Sarbazi-Azad, "NURA: A framework for supporting non-uniform resource accesses in GPUs," *ACM POMACS*, 2022.
[110] T. M. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. P. Shen, "Hardware support for prescient instruction prefetch," in *HPCA*, 2004.
[111] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen, "Speculative precomputation on chip multiprocessors," *Proc. of MTEAC*, 2002.
[112] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," in *ISCA*, 1999.
[113] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Microarchitectural support for precomputation microthreads," in *MICRO*, 2002.
[114] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *MICRO*, 2001.
[115] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *ISCA*, 2001.
[116] M. Dubois, "Fighting the memory wall with assisted execution," in *Proceedings of the 1st Conference on Computing Frontiers*, 2004.
[117] M. Dubois and Y. Song, "Assisted execution," *University of Southern California CENG Technical Report*, 1998.
[118] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg, "Slipstream execution mode for cmp-based multiprocessors," in *HPCA*, 2003.
[119] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *ASPLOS*, 2011.
[120] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-Execution," in *ASPLOS*, 2002.
[121] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham, "Dynamic helper threaded prefetching on the Sun UltraSPARC/spl reg/CMP processor," in *MICRO*, 2005.
[122] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *ISCA*, 2001.
[123] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003.
[124] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for efficient processing in runahead execution engines," in *ISCA*, 2005.
[125] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of GPU architectures," in *ISCA*, 2015.
[126] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and adapting precomputation threads for effcient prefetching," in *HPCA*, 2007.
[127] C. B. Zilles, J. S. Emer, and G. S. Sohi, "The use of multithreading for exception handling," in *MICRO*, 1999.
[128] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *ISCA*, 2001.
[129] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: optimizing GPU memory bandwidth via warp specialization," in *SC*, 2011.
[130] T. M. Aamodt and P. Chow, "Optimization of data prefetch helper threads with path-expression based statistical modeling," in *ICS*, 2007.