DTM-NUCA: Dynamic Texture Mapping-NUCA for Energy-Efficient Graphics Rendering

David Corbalán-Navarro
Univ. de Murcia
Murcia, Spain
david-corbalan2@um.es

Juan L. Aragón
Univ. de Murcia
Murcia, Spain
jlaragon@um.es

Joan-Manuel Parcerisa
Univ. Politècnica de Catalunya
Barcelona, Spain
jmanel@ac.upc.edu

Antonio González
Univ. Politècnica de Catalunya
Barcelona, Spain
antonio@ac.upc.edu

***Abstract*—Modern mobile GPUs integrate an increasing number of shader cores to speedup the execution of graphics workloads. Each core integrates a private Texture Cache to apply texturing effects on objects, which is backed-up by a shared L2 cache. However, as in any other memory hierarchy, such organization produces data replication in the upper levels (i.e., the private Texture Caches) to allow for faster accesses at the expense of reducing their overall effective capacity. E.g., in a mobile GPU with four shader cores, about 84.6% of the requested texture blocks are replicated in at least one of the other private Texture Caches.**

This paper proposes a novel dynamically-mapped Non-Uniform Cache Architecture (NUCA) organization for the private Texture Caches of a mobile GPU aimed at increasing their effective overall capacity and decreasing the overall access latency by attacking data replication. A block missing in a local Texture Cache may be serviced by a remote one at a cost smaller than a round trip to the shared L2. The proposed Dynamic Texture Mapping-NUCA (DTM-NUCA) features a lightweight mapping table, called Affinity Table, that is independent of the L2 cache size, unlike a traditional NUCA organization. The best owner for a given set of blocks is dynamically determined and stored in the Affinity Table to maximize local accesses. The mechanism also allows for a certain amount of replication to favor local accesses where appropriate, without hurting performance due to the small capacity loss resulting from the allowed replication. DTM-NUCA is presented in two flavors. One with a centralized Affinity Table, and another with a distributed Affinity Table. Experimental results show first that the L2 pressure is effectively reduced, eliminating 41.8% of the L2 accesses on average. As for the average latency, DTM-NUCA performs a very effective job at maximizing local over remote accesses, achieving 73.8% of local accesses on average. As a consequence, our novel DTM-NUCA organization obtains an average speedup of 16.9% and overall 7.6% energy savings over a conventional organization.

***Index Terms*—mobile devices, GPUs, graphics pipeline, cache, NUCA, energy efficiency.**

I. Introduction

Mobile devices such as smartphones, tablets or smartwatches, have dramatically evolved in the last decade and nowadays are very complex devices able to perform a huge variety of tasks such as playing video games, watching high-resolution videos, doing video calls, messaging and surfing the internet. The more tasks a device executes, the more energy is consumed and, hence, the more heat is generated. In this context, energy efficiency has become a key design challenge, playing a critical role in the mobile arena, in which the user experience heavily depends on battery autonomy, portability and low heat. Regarding the video games segment, smartphones integrate increasingly powerful GPUs able to render complex graphics scenes in high resolutions. Users also demand higher-quality graphics to play on their devices, leading to more complex scenes with more elaborated shading and lighting effects. In this highly demanding scenario, the energy efficiency of mobile GPUs has become a central design challenge. Prior works have pointed out that the GPU is one of the major contributors in energy consumption in a mobile SoC because of their powerful computing capabilities [1]–[4]. Current mobile GPUs can feature more than 8 processing units with their own local caches. These processing units are usually known as Vertex or Fragment Processors, depending on the elements they process. This work focuses on the Fragment Processors (also known as Shader Cores) as they are the most power-hungry processing units within the Graphics Pipeline [5] (68.1% of the total GPU power in our experiments) and presents a novel and energy-efficient organization of their private Texture Caches, whose purpose is to store recently used texture data.

As the number of processing elements increases in any system, the complexity to efficiently interconnect them becomes more challenging. When it comes to GPUs, the more Fragment Processors the more replication in their private Texture Caches because of the data locality and the increasing sharing degree[1] which significantly reduces their effective capacity. Figure 1 shows the average replication degree in the Texture Caches of the Fragment Processors for all the texture block requests right after they have been serviced, broken down into how many of the Texture Caches hold each block. For the sake of visibility, only 4 Fragment Processors have been considered. It can be seen that about 50% of the serviced blocks are fully replicated in all the Texture Caches, whereas 15.4% of the serviced blocks reside in exactly one of the caches (i.e., *unique* blocks, as opposed to *replicated* blocks that are present in more than one cache). Overall, it is observed that 84.6% of the blocks are replicated up to some extent, which leads to a significant waste of the Texture Caches' effective capacity.

To tackle this storage waste and improve the overall access

[1]Texture block replication naturally appears as a result of the massive parallelism in graphics workloads where scene objects are divided into triangles, which are disaggregated into independent fragments –that share the same texture– and are simultaneously shaded in multiple Fragment Processors.

2377-5750/22/$31.00 ©2022 IEEE
DOI 10.1109/PDP55904.2022.00030

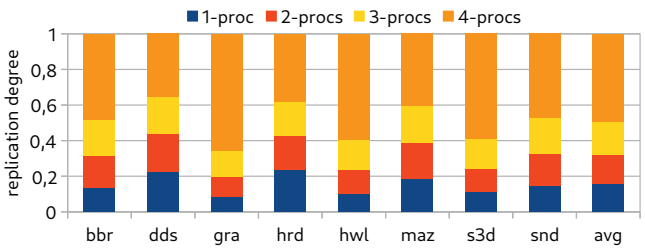

Fig. 1. Replication degree of the evaluated benchmarks (see Table II) broken down into the number of Fragment Processors keeping the serviced block.

latency to textures, this paper proposes a novel NUCA organization for the private Texture Caches, called Dynamic Texture Mapping-NUCA (DTM-NUCA), that exploits the peculiarities of texture access patterns. With this organization we pursue a 2-fold goal: 1) increase the effective overall capacity of Texture Caches by reducing data replication; and 2) shorten the overall access latency to textures. In the proposed NUCA approach, a block missing in a *local* Texture Cache can be serviced by a *remote* one at a latency cost smaller than a round trip to the shared L2. This is accomplished by using a very lightweight dynamic scheme to map sets of texture blocks to particular Fragment Processors, and by implementing a 2D-mesh interconnection network to drive all the traffic among the different Texture Caches.

It is important to note that, unlike conventional private caches on general-purpose multicore systems, Texture Caches do not need a directory to track coherence since textures are read-only data which do not suffer from read-modify-write access patterns. However, DTM-NUCA uses a small Affinity Table to track the ownership of texture blocks. For a given texture block address, the Affinity Table returns the Fragment Processor who owns the block at that particular moment. This owner dynamically changes over time to better adapt to the access pattern behavior with the aim of maximizing local accesses over remote ones. Additionally, our approach introduces a mechanism to allow for a small amount of replication to favor local accesses where appropriate with a minimal impact on performance despite the small capacity loss from the allowed replication. DTM-NUCA is presented in two flavors. One design option uses a centralized Affinity Table aimed at maximizing the overall capacity of the Texture Cache, at the expense of increasing the remote accesses. A second design option relies on a distributed Affinity Table which also aims to reduce the access latency to texture blocks by further allowing local accesses.

The main contributions of this work are:

- We propose a novel NUCA organization targeting the private Texture Caches of a GPU which is able to interconnect many nodes with a very small ownership mapping table to reduce overheads.
- We reduce the replication degree of Texture Caches which results in an average 8× increase of the overall effective capacity (for the case of 32 Fragment Processors).
- We reduce the pressure over the shared L2 by eliminating 41.8% of its accesses on average, significantly reducing the access latency to textures.
- Overall, this results in an average speedup of 16.9% and a reduction of the energy consumption of 7.6%.

The rest of the paper is organized as follows. Section II provides some background on mobile GPUs and NUCA organizations. Section III describes the proposed DTM-NUCA whereas Section IV presents two practical hardware implementations. Section V explains the evaluation methodology and Section VI reports the experimental results of our proposal. Section VII summarizes the main conclusions of this work.

II. Background and Related Work

A. The Graphics Pipeline

The graphics pipeline in a typical GPU consists of two parts: the Geometry Pipeline, a front-end that transforms the vertices of the input primitives (commonly triangles) from the local coordinates to the perspective view coordinates; and the Raster Pipeline, a back-end that assigns color to each screen space pixel covered by these primitives, applies textures, resolves visibility through a depth test and writes colors to the frame buffer in main memory.

There are two main rendering modes: Immediate-Mode Rendering (IMR) and Tile-Based Rendering (TBR). In IMR, each triangle transformed in the Geometry Pipeline is immediately sent down the Raster Pipeline for further pixel processing. Graphical objects that are overlapped by others generate a lot of pixels that are not visible in the final scene. Part of them are culled during a visibility stage like the Z-test, but many of them are not, which produces a large amount of useless work (a problem known as $overdraw$). In IMR, the problem is even worse because the colors of those pixels are not only computed but also written multiple times into memory, which increases memory traffic and wastes energy. In contrast, TBR completely avoids this useless memory traffic by splitting the screen into equally-sized small square regions called tiles, and processing them one at a time. Since each tile is small enough so that all its pixels may be stored in local on-chip memory, each pixel color is not transferred to main memory until the whole tile is rendered, hence, each pixel is written only once. Overall, TBR reduces off-chip memory accesses and is more energy efficient which make it more suitable for mobile GPUs.

B. Tile-Based Rendering (TBR)

Figure 2 depicts the pipeline of a TBR architecture. The first stage corresponds to the Vertex Fetcher which loads the vertices that compose the scene from main memory. A private on-chip Vertex Cache, connected to the shared L2, is used to speed up the process. A vertex is defined by a set of attributes, including position, color, normal vector and texture coordinates. Vertices are then sent to the Vertex Processors which apply a *vertex shader*, a user-defined program to transform them from model-space coordinates to screen-space coordinates. The transformed vertices are sent to the Primitive Assembly stage where they are appropriately grouped to form

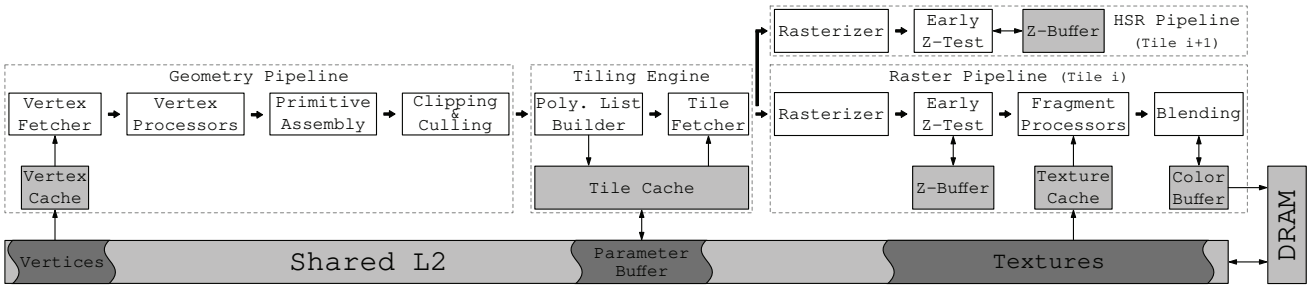

Fig. 2. Overview of the graphics pipeline of a TBR GPU, also adding a Hidden Surface Removal (HSR) phase.

primitives (typically triangles). These primitives are clipped and culled, to discard back-facing ones and those falling outside the view volume, and passed over to the Tiling Engine.

The Tiling Engine stores the primitives and their attributes into the so-called Parameter Buffer, a big structure held in main memory. The Polygon List Builder identifies the screen tiles overlapped by each primitive to generate, for each tile, a list of overlapped primitives. These lists are stored in the Parameter Buffer as well. Once all the geometry has been processed and stored in the Parameter Buffer, the Tiling Engine starts scheduling the screen tiles in sequence. For each tile, its primitives are fetched from the Parameter Buffer, cached in the L2 and Tile Cache, and passed over to the Raster Pipeline.

In the Rasterizer, each primitive is discretized into *fragments* by interpolating the attributes of its vertices. A fragment is the set of attributes associated with a pixel in the screen. Fragments are checked for visibility in the Early Z-Test stage based on depth information stored in a tile-sized on-chip Z-Buffer. Visible fragments are processed in the Fragment Processors by executing a *fragment shader*, another user-defined program that applies a shading and a lighting model to produce its final color. Fragments are processed in groups of 2x2 called $quads$ to facilitate texture filtering and computation of gradients. The shading process accesses textures through a private on-chip Texture Cache attached to each Fragment Processor to reduce access latency and save DRAM bandwidth. Finally, output colors are processed in the Blending Unit, which also manages translucent pixels, using a tile-sized on-chip Color Buffer which is eventually flushed into DRAM.

C. Tile-Based Deferred Rendering (TBDR)

TBDR is an evolution of TBR in which overdraw is totally eliminated. TBDR adds a Hidden Surface Removal (HSR) phase at the beginning of the Raster Pipeline to eliminate occluded fragments and avoid their costly rendering. While working on a given tile, the HSR phase processes the primitives of the *next* tile to generate its Z-Buffer beforehand, thus completely eliminating any overdraw that would appear by otherwise failed Z-Tests. To that end, a Rasterizer and an extra Early-Z Test unit with its own Z-Buffer are used.

D. Background and Related Work on NUCA Organizations

A conventional NUCA cache is characterized by four policies that determine its behavior: 1) a bank *placement* policy to decide where to place an incoming block from a set of possible banks; 2) a bank *search* policy to locate the block within the set of banks in which can be placed, or raise a miss if not found; 3) a bank *replacement* policy that determines what to do with an evicted block; and 4) a bank *migration* policy to determine where to move a frequently accessed block.

The placement policy is crucial as it determines the operational behavior and complexity of the NUCA design. The simplest approach, S-NUCA [6], uses a static mapping scheme in which the cache's bank/slice is solely determined by the memory address (using a fixed function that assigns memory addresses to a particular bank/slice; e.g., using the lower bits of the block's address). A static mapping does not incur storage overhead, however, each block is always assigned to the same bank, regardless of whether the data is more frequently accessed by other cores. Alternatively, D-NUCA [6] allows for a higher placement flexibility since a block can be dynamically migrated into any of the bank sets at the expense of a costly locating overhead (each bank set must be searched either through a centralized tag store or by broadcasting the tags to all the banks in the set).

There are more complex schemes like R-NUCA [7] that applies a different placement policy depending on each access class (i.e., to private or to shared data) and performs a rotational interleaving within a cluster of cores. Other works [8]–[12] allow for different placement flexibility. In [13] a NUCA organization for CMPs is presented, which reduces the access latency by using a cost-effective function that places the data closer to their owners. In [14] the same mechanism is improved by allowing some victims and replicas. In [15] replicas and victims are also leveraged to increase data locality by using an adaptive method based on the cache's bank/slice pressure. In [16] a NUCA with a dynamic partitioning scheme at the shared level is presented, which dynamically controls how much available capacity a cache's slice has. NuRAPID [17] decouples data placement from tag placement, and places frequently-accessed data into the fastest slices to reduce the data migration rate. Despite the high number of proposed NUCA schemes, only S-NUCA is implemented in commercial processors and it continues to be an active research topic [18].

Our proposed DTM-NUCA differs from the aforementioned works in that it is the first NUCA organization targeting the private L1 Texture Caches of GPUs (which have the peculiarity

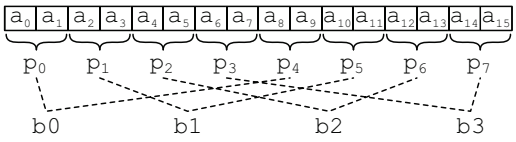

Fig. 3. Grouping scheme of texture blocks into buckets. This simple example shows an address space with 16 blocks at addresses (a_0-a_{15}) that belong to 8 pages (p_0-p_7) which are grouped (modulo 4) into 4 buckets (b_0-b_3).

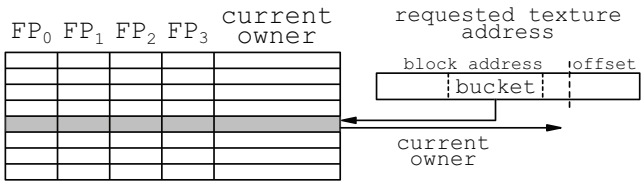

Fig. 4. Affinity Table overview for 4 Fragment Processors and 8 buckets.

that store read-only data, thus, not involving a coherence protocol) and primarily focuses on reducing block replication with a much simpler mapping scheme based on the temporary ownership of texture blocks.

III. A NUCA organization for Texture Caches

The main goal of the proposed DTM-NUCA organization is to increase the overall effective cache size of the Fragment Processors by reducing the replication degree, so both the number of L2 accesses and overall access latency are reduced. To do so, we organize the private Texture Caches as a NUCA where each Fragment Processor has a temporary exclusive access to a specific range of texture blocks. Another important feature of DTM-NUCA is that it uses a dynamic mapping scheme based on how Fragment Processors access textures.

A. Bucket-based Ownership of Texture Blocks

Our approach uses a dynamic mapping scheme, similarly to a conventional D-NUCA. However, instead of using a big directory with the size of the shared (L2/L3) level, we use a small buffer, called Affinity Table, to track the ownership of blocks. Storing the bank owner for each block in memory would result in a prohibitive Affinity Table that could not be kept on-chip. To reduce the storage overhead, instead of tracking the bank owner for *each* memory block, our approach groups a number of contiguous blocks into a smaller number of *buckets* so that tracking the ownership of these buckets significantly reduces the storage needs of the Affinity Table.

Grouping all blocks into n buckets could use a simple modulo function of the block's address (a) such as "address a belongs to bucket b, where $b = a \bmod n$". However, we found that keeping a small number of adjacent blocks within the same bucket tends to favor locality. Therefore, our grouping function considers pages of m consecutive blocks instead of individual blocks (the page address p is simply derived from the block address a as: $p = a \ / \ m$). Therefore, the grouping of all pages into n buckets follows a similar modulo function "page p belongs to bucket b, where $b = p \bmod n$". Figure 3 illustrates this grouping of memory blocks into buckets.

As depicted in Figure 4, the Affinity Table is implemented as a small buffer with one entry per bucket, containing one saturating counter per Fragment Processor plus a field to identify the current owner. In our evaluation we have used 4-bit saturating counters, and 5 bits to identify the owner (as our baseline setting has 32 Fragment Processors). We have experimentally determined that 32 buckets suffice to achieve a good storage and performance trade-off. This 32-entry Affinity Table results in a total size of 4256 bits (or 0.51 KiB). It works as follows. When a request from a Fragment Processor arrives in the Affinity Table, the bucket identifier is used to index it and the current owner for that bucket is retrieved (see Figure 4). Additionally, the saturating counter of the requester Fragment Processor is incremented.

B. Dynamic Ownership Tracking

Fragment Processors access to their local Texture Caches to perform the texturing process. However, as the scene changes from one frame to the next (e.g., because of graphics objects movements and/or camera movements) each Fragment Processor accesses different texture blocks. Therefore, a static mapping leads to performance degradation because of the huge amount of remote accesses. To mitigate this issue, we allow the ownership of texture blocks to dynamically change. To make this feasible, we define a coarse-grained ownership assignment policy based on the use of *buckets*. To determine a bucket's owner (i.e., which Fragment Processor a bucket belongs to), we account the number of accesses of each Fragment Processor to each bucket. Every time a Fragment Processor accesses a texture block, its bucket is calculated and its corresponding counter is incremented. These counters inform what Fragment Processor has accessed each bucket more times so far, and hence, it is the best owner for it.

Initially, all the counters are set to zero and the first Fragment Processor that accesses a bucket acquires the ownership. Whenever a counter saturates, a reset event happens consisting of halving (a 1-bit right-shift operation) all the counters, including the one that is saturated. Then, the ownership changes to the processor whose counter is saturated (the owner could remain the same). To avoid ownership ping-ponging effects, because two or more Fragment Processors perform a similar number of accesses to a bucket, the saturating counter must be greater than the one from the current owner plus a percentage.

Even though DTM-NUCA aims to have a unique owner for each bucket, due to the high sharing degree that some particular texture blocks exhibit among Fragment Processors, it could be beneficial if such blocks could be served by more than one owner. For such cases, our approach allows for a certain amount of replication to favor local accesses and improve performance. Due to the mentioned affinity changes, there is a point in which a texture block can be replicated in both an old and a new owner. During this time, the block from the old owner can still be utilized since local Texture Caches are probed first, and the block is served locally in case of a hit, regardless of a recent ownership change. However, in

order to consolidate the new owner and avoid replication for long periods of time, the LRU counters are not updated when the local cache of a just-exchanged owner serves the block. This way, the old owner will eventually evict the block from its Texture Cache in a natural way.

A final related issue has to do with the *inertia* for changing the ownership due to the use of saturating counters. If a Fragment Processor frequently accesses a particular bucket, and later another Fragment Processor starts doing the same, it will pass some time until the second Fragment Processor gets the ownership of the bucket. This inertia exacerbates with bigger counters. There are two ways of mitigating it: 1) reduce the size of the saturating counters, but this would also reduce the accuracy of the owner assignment; 2) make a reset of all the counters and recompute the ownership for the buckets every certain amount of time, called *epoch*. An epoch is measured in terms of texture accesses, so every time the total amount of accesses exceeds the epoch length, a counter reset event takes place. Determining an appropriate epoch length is important since long epochs will make the inertia problem remain, whereas short epochs will lead to lots of affinity changes. In this work, we have determined the epoch duration experimentally to find a good trade-off between remote accesses and affinity changes. Different epoch lengths were evaluated (100, 1K, 10K, 20K, 50K and 100K texture accesses) and the best trade-off was obtained for an epoch length of 20K texture accesses.

IV. DTM-NUCA Design Options

The Affinity Table can be physically implemented in two alternative options, either as a centralized structure or distributed over the different nodes (Fragment Processors).

A. Centralized Affinity Table

In the centralized approach, all the nodes access the same table. The advantage is that all the nodes share up-to-date information regarding the ownership of each bucket. However, a centralized approach 1) leads to contention when multiple nodes try to access it concurrently, 2) imposes serialization when updating the ownership information, and 3) incur extra latency and energy per access for the nodes that are further away from the Affinity Table. All of that makes the centralized scheme a potential bottleneck.

In the baseline architecture each Fragment Processor can only access its private Texture Cache, which is connected by a common bus to the shared L2 cache. In order to directly interconnect Fragment Processors to each other and allow accessing to remote Texture Caches, the proposed DTM-NUCA uses a 2D-mesh network composed of routers that redirects the requests to the destination Fragment Processor using an X-Y routing mechanism. Figure 5-(a) shows this interconnection network. The latency of routing a request is one cycle.

B. Distributed Affinity Table

As mentioned above, a centralized Affinity Table implemented as another node in the interconnection network leads to contention and higher access latencies. To avoid that, we propose a distributed approach for the Affinity Table, as depicted in Figure 5-(b), in which each node has its own local version which is later synchronized with the rest, at the very large granularity of *epochs* (defined as 20K texture accesses, as stated in Section III-B) by using a simple broadcasting protocol which works as follows.

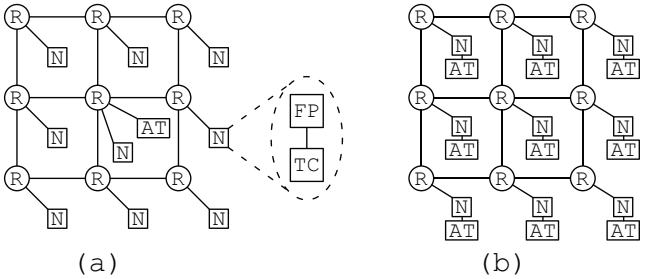

Fig. 5. Interconnection networks for 9 nodes (where each node is a Fragment Processor plus a Texture Cache). (a) Centralized scheme with the Affinity Table in the center. (b) Distributed scheme with a per-node Affinity Table.

Initially, all Fragment Processors own all the buckets since there has been no communication among them yet. At every epoch, the Affinity Tables get synchronized by broadcasting all the *owned* buckets over the interconnection network. It is important to note that a synchronization event does not impose any stall in the operation of the Fragment Processors. The broadcasting process is made in the background and each Fragment Processor updates its local Affinity Table on-the-fly, as packets arrive, by updating the ownership of each bucket with the most up-to-date owner. In the special case of two counters having exactly the same value, the ownership is assigned to the node with the lowest identifier. Therefore, Fragment Processors continue working without waiting until the synchronization is completed. A minor side effect of this *lazy* synchronization approach is that, right at the beginning of an epoch, the ownership information is not fully updated but this has a negligible effect in comparison with the epoch length. It will simply happen that a Fragment Processor will retrieve the same owner as right before the epoch change. In any case, it is guaranteed that the ownership information will be fully propagated to all Fragment Processors after a delay.

V. Evaluation methodology

A. Simulator infrastructure

In order to validate and evaluate the performance of our proposal, we have used the Teapot framework [19], which includes a cycle-accurate simulator that models a modern mobile GPU based on the Mali Bifrost architecture. Teapot also models a TBDR architecture which constitutes our baseline GPU configuration. To model the memory subsystem, Teapot uses DRAMSim2 [20], a cycle-accurate simulator for modeling DRAM and the memory controllers. The GPU power and area is modeled by integrating McPAT [21], a tool that calculates the power, area and timing of each component within the architecture. To obtain the simulator traces, a modified version

TABLE I
SIMULATION PARAMETERS.

Baseline GPU Parameters	
Frequency	600 MHz
Voltage	1.0 V
Technology node	22 nm
Screen resolution	2160x1080
Tile size	32x32 pixels
Main Memory	
Frequency	400 MHz
Voltage	1.5 V
Technology node	32 nm
Latency	50-100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Size	1 GiB, 8 banks
Queues	
Vertex (Input & Output)	16 entries, 136 bytes/entry
Triangle & Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Color	64 entries, 24 bytes/entry
Caches	
Block size	64 bytes (all caches)
Vertex Caches (× # VPs)	4 KiB, 1-cycle lat, 2-way assoc
Texture Caches (× # FPs)	16 KiB, 2-cycle lat, 4-way assoc
Tile Cache	32 KiB, 2-cycle lat, 2-way assoc
L2 Cache	1 MiB, 8 banks, 18-cycle lat, 8-way assoc
Color Buffer	1 KiB, 1-cycle latency
Depth Buffer	1 KiB, 1-cycle latency
Non-programmable stages	
Primitive assembly	1 vertex/cycle
Rasterizer	1 attribute/cycle
Early Z-Test	8 in-flight quad-fragments
Programmable stages	
Vertex processing stage	4 Vertex Processors (VPs)
Fragment processing stage	32 Fragment Processors (FPs)
DTM-NUCA parameters	
Number of buckets	32 buckets
Access counters size	4 bits
Page size	8 blocks
Affinity Table size	0.51 KiB
Routing latency	1 cycle/hop
Epoch length	20K texture accesses

of GAPID [22] has been used, a Graphics API Debugger developed by Google that, among other things, extracts OpenGL traces from a mobile device. GAPID allows to capture an OpenGL trace while running a game and then replay the trace over a modified version of the Gallium Softpipe renderer [23] (from the Mesa 3D project) that gives us the final trace to be used with Teapot. Table I shows the simulation parameters and the baseline GPU configuration used to evaluate the proposed DTM-NUCA organization. Regarding the area overhead, the centralized Affinity Table occupies 0.02% of the total GPU area, while the distributed version occupies 0.54%.

B. Benchmarks

To evaluate DTM-NUCA we have chosen a set of commercial and very popular games (based on their number of downloads) available from the Google Play Store. The evaluated scenes have been selected to get a representative and realistic use-case scenario for each game. Table II shows some characteristics of each working set.

VI. EXPERIMENTAL RESULTS

In addition to the baseline configuration (consisting of a private Texture Cache on each Fragment Processor), we have evaluated three NUCA architectures. The first one is a conventional D-NUCA that instead of replicating the data between caches and having a directory with a list of sharers, it completely avoids the replication by allowing only a single sharer per directory entry. Note that such a scheme requires a directory whose size depends on the L2 cache size. Against this implementation, we compare our two proposals: DTM-NUCA with a centralized Affinity Table, and DTM-NUCA with a distributed Affinity Table (labeled as DTM-NUCA and DTM-NUCA-Dist, respectively). In all cases, we consider 32 Fragment Processors and 32 Texture Caches.

Since the aim of our proposal is to increase the effective capacity of the Texture Caches, our first experiment measures the amount of L2 accesses that have been reduced due to this aggregated capacity. Figure 6 shows the amount of L2 accesses normalized to the baseline configuration for the three NUCA architectures. As expected, D-NUCA produces the lowest number of accesses to L2 (46.2% average reduction) because it completely eliminates replication and maximizes the effective capacity of the Texture Cache, so we can consider it as our upper bound. On the other side, DTM-NUCA reaches a 41.8% reduction, close to the optimum capacity, while DTM-NUCA-Dist achieves a more modest reduction of 33.3%. This is due to the delayed synchronization between Affinity Tables, which leads to some accuracy loss since until the synchronization takes place, some Affinity Tables may differ and assign the same bucket to different Fragment Processors.

Figure 7 further illustrates the benefits of DTM-NUCA by comparing its total amount of L2 accesses against a baseline Texture Cache ranging from 16KiB to 512KiB. As shown, a 16KiB DTM-NUCA (red line) has almost the same L2 accesses as the baseline with 8× more capacity (128KiB), while a 16KiB DTM-NUCA-Dist (yellow line) achieves the same L2 accesses as a baseline 4× bigger (64KiB).

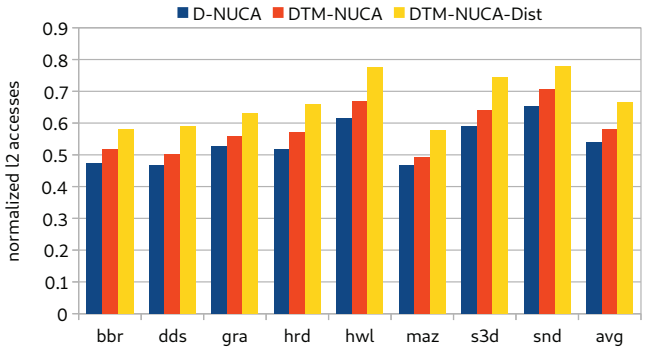

Fig. 6. Normalized L2 accesses for the different NUCA designs with respect to the baseline.

TABLE II
EVALUATED BENCHMARKS FROM THE GOOGLE PLAY STORE.

Benchmark	Alias	Description	Downloads (Mill.)	Vertex shader instr. (Mill.)	Fragment shader instr. (Mill.)	Execution Time (Mill. cycles)
Beach Buggy Racing	bbr	Racing	100-500	96	2052	749
Derby Destruction Simulator	dds	Racing & Battle royale	10-50	165	4993	1140
Gravity	gra	Action	1-5	74	355	144
Hellrider	hrd	Racing	1-5	112	3534	868
Hot Wheels	hwl	Racing	50-100	431	2073	950
Maze 3D	maz	Labyrinth	10-50	131	4420	1112
Sniper 3D	s3d	Shooter	100-500	144	1600	684
Sonic Dash	snd	Adventure arcade	100-500	87	4154	1219

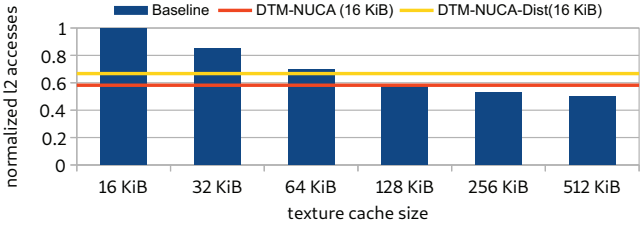

Fig. 7. Normalized L2 accesses of DTM-NUCA with respect to the baseline setting with varying sizes.

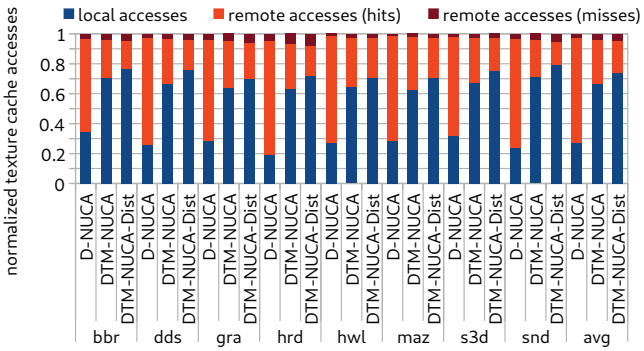

Fig. 8. Local and remote accesses to the Texture Caches for the evaluated NUCA organizations.

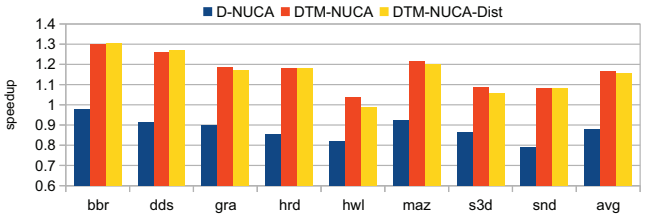

Fig. 9. Speedup of NUCA organizations with respect to the baseline.

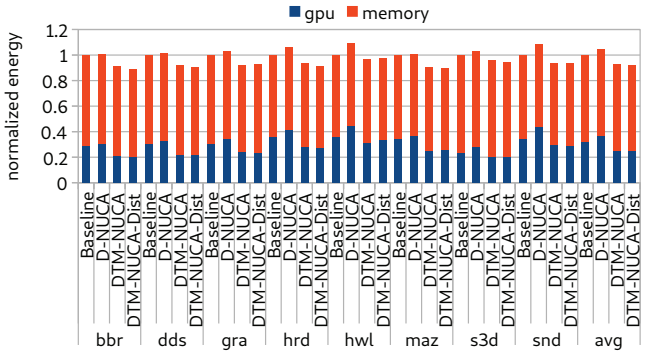

Fig. 10. Normalized energy for the evaluated NUCA organizations.

However, our aim is to also increase the local accesses as a measure to reduce the total access latency. This is done by allowing a small degree of replication on the local caches. The downside is a reduction of the overall capacity. Figure 8 shows the accesses to textures broken down into local and remote accesses, which in turn are broken down into L2 hits and misses. As it can be seen, DTM-NUCA and DTM-NUCA-Dist produce slightly more L2 misses than D-NUCA (as discussed before) but have an impressive number of local hits (an average of 66.3% and 73.9%, respectively), much more than the 27.5% achieved by D-NUCA, which shows a large amount of remote hits. This is because the replication-free scheme of D-NUCA completely neglects whether an access is made to the local cache or to a remote one. Moreover, D-NUCA lacks a data migration mechanism to maximize local accesses on-the-fly, unlike DTM-NUCA and DTM-NUCA-Dist, thanks to the Affinity Table. Since DTM-NUCA-Dist allows more replication due to the synchronization delay of the Affinity Tables, it gets more local accesses than the centralized DTM-NUCA, in detriment to the overall effective capacity. This demonstrates the goodness of the dynamic assignment of buckets performed with the Affinity Table.

The performance of DTM-NUCA is reported in Figure 9. We can observe performance improvements of up to 30.2% and 30.4% in the case of `bbr` using DTM-NUCA and DTM-NUCA-Dist respectively. On average, DTM-NUCA and DTM-NUCA-Dist get a speedup of 16.9% and 15.7%, respectively. However, D-NUCA suffers an average slowdown of 22% because of its poor migration policy which only allows migration whenever a block is evicted from the owner cache. This causes a lot of remote accesses that increase the access latency.

Figure 10 shows the energy consumption of the evaluated architectures broken down into memory and GPU energy. Note that the main sources of energy savings are the dynamic energy because of the speedup, and the reduction of accesses to L2.

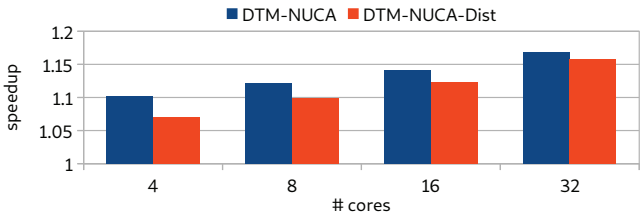

Fig. 11. Scalability of DTM-NUCA for different number of nodes.

The best results are obtained in `bbr` and `maz` for the case of DTM-NUCA-Dist, achieving reductions of up to 11.1% and 10.3% respectively. On average, DTM-NUCA and DTM-NUCA-Dist obtain respective energy savings of 6.9% and 7.6%. For the same reason as for speedup, D-NUCA is worse than the baseline, increasing the energy a 4% on average.

Finally, we focus on the scalability of the proposed approaches. Figure 11 shows the speedup of both flavors ranging from 4 to 32 Fragment Processors. As it can be observed, with only 4 Fragment Processors DTM-NUCA gets an average speedup of 10.2% whereas DTM-NUCA-Dist achieves 7.0%. As the number of nodes grows, the speedup (normalized to the baseline with the same number of nodes) increases, but the increment lowers on each step. Although DTM-NUCA gets the best performance, DTM-NUCA-Dist tends to converge with DTM-NUCA as the number of cores increases.

VII. Conclusions

This paper proposes DTM-NUCA, a dynamically-mapped NUCA organization targeting the private Texture Caches of mobile GPUs with the goal of increasing their overall effective capacity by reducing block replication. Unlike traditional NUCA organizations, this scheme incorporates a very small table (Affinity Table) whose size is independent of the L2 size. Some replicas are deliberately allowed in order to improve the local access to Texture Caches without penalizing performance. DTM-NUCA presents two variants regarding the implementation of the Affinity Table: a centralized and a distributed design. Our proposal is able to reduce the L2 pressure by 41.8% on average. Local accesses are maximized, reaching 73.8% on average. Overall, these improvements lead to an average speedup of 16.9% and energy savings of 7.6% over a traditional private cache configuration.

Acknowledgments

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, the ICREA Academia program and a research fellowship from the University of Murcia's "Plan Propio de Investigación".

References

[1] E. De Lucas, "Reducing redundancy of real time computer graphics in mobile systems," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2018.

[2] S. Patil, Y. Kim, K. Korgaonkar, I. Awwal, and T. S. Rosing, "Characterization of user's behavior variations for design of replayable mobile workloads," in *International Conference on Mobile Computing, Applications, and Services*. Springer, 2015, pp. 51–70.

[3] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 3, p. 26, 2014.

[4] J. Pool, "Energy-precision tradeoffs in the graphics pipeline," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2012.

[5] T. Akenine-Moller and J. Strom, "Graphics processing units for handhelds," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 779–789, 2008.

[6] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002, pp. 211–222.

[7] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "R-nuca: Data placement in distributed shared caches," *Computer Architecture Lab at Carnegie Mellon Technical Report*, 2009.

[8] J. Lira, C. Molina, A. González *et al.*, "Hk-nuca: Boosting data searches in dynamic non-uniform cache architectures for chip multiprocessors," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 419–430.

[9] N. Muralimanohar and R. Balasubramonian, "Interconnect design considerations for large nuca caches," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 369–380, 2007.

[10] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. Prete, "Way adaptable d-nuca caches," *International Journal of High Performance Systems Architecture*, vol. 2, no. 3-4, pp. 215–228, 2010.

[11] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. A. Prete, "A power-efficient migration mechanism for d-nuca caches," in *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2009, pp. 598–601.

[12] A. Arora, M. Harne, H. Sultan, A. Bagaria, and S. R. Sarangi, "Fp-nuca: A fast noc layer for implementing large nuca caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 9, pp. 2465–2478, 2014.

[13] J. Merino, V. Puente, P. Prieto, and J. Á. Gregorio, "Sp-nuca: a cost effective dynamic non-uniform cache architecture," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 2, pp. 64–71, 2008.

[14] J. Merino, V. Puente, and J. A. Gregorio, "Esp-nuca: A low-cost adaptive non-uniform cache architecture," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 2010, pp. 1–10.

[15] A. Huang, J. Gao, W. Guo, W. Shi, M. Zhang, and J. Jiang, "Psa-nuca: A pressure self-adapting dynamic non-uniform cache architecture," in *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. IEEE, 2012, pp. 181–188.

[16] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 2–12.

[17] Z. Chishti, M. D. Powell, and T. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 55–66.

[18] M. Rapp, A. Pathania, T. Mitra, and J. Henkel, "Neural network-based performance prediction for task migration on s-nuca many-cores," *IEEE Transactions on Computers*, 2020.

[19] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International ACM Conference on Supercomputing*. ACM, 2013, pp. 37–46.

[20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 469–480.

[22] "Gapid," https://developers.google.com/vr/develop/unity/gapid, accessed September 2021.

[23] "Gallium3d," https://www.freedesktop.org/wiki/Software/gallium, accessed July 2021.