Memory Management Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

Purpose

This tutorial provides information about memory hardware on the PlayStation®Vita as well as related software APIs. It also discusses memory debugging techniques and the performance characteristics of the various memory types and caching modes.

This document includes the following:

- An overview of the memory hardware and related software APIs
- Details about the PTLA unit and libgxm Transfer API
- A discussion of various approaches for debugging memory-related problems
- Comparisons of memory throughput for several different combinations of memory type, caching mode, and copy method
- Additional recommendations for managing memory on PlayStation®Vita

Related Documentation

Any updates or amendments to this guide can be found in the release notes that accompany the release package.

Refer to the following PlayStation®Vita SDK documents for additional details:

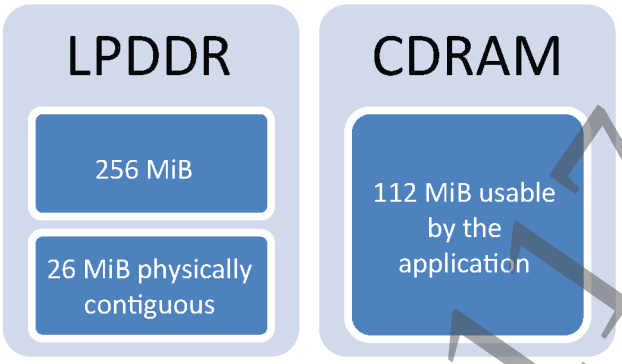
- *Programming Startup Guide*: Provides general information about application memory resources.
- *Kernel Overview*: Provides an overview of the kernel memory management functionality.
- *Kernel Reference*: Provides detailed reference descriptions about each of the kernel memory management functions and data structures.
- *libheap Overview*: Provides an overview of custom allocator functionality.
- *libheap Reference*: Provides detailed reference descriptions about each of the custom allocator functions.
- *Memory Management Function Replacements of the C and C++ Standard Libraries: Reference*: Provides detailed reference descriptions of each of the memory management replacement functions.
- *C and C++ Standard Libraries: Overview and Reference*: Provides information about configuring the libc default heap.
- *libgxm Overview*: Provides an overview of PTLA unit functionality.
- *libgxm Reference*: Provides detailed reference descriptions about each of the libgxm Transfer API functions and data structures.
- *GPU User's Guide*: Provides further information about the PTLA hardware.

1 Hardware Overview

Memory Types

As shown in Figure 1, PlayStation®Vita contains two separate areas of memory.

Figure 1 Memory Types and Sizes



LPDDR

PlayStation®Vita contains a total of 282 MiB of usable LPDDR memory. When using a Development Kit with "Memory Size" set to "Development Tool Size", 512 MiB ordinary memory + 26 MiB physically contiguous memory is available.

Physically contiguous memory is required for Codec Engine I/O buffers – for example when using the AVC and JPEG decoding libraries. It is also required by libcamera image buffers.

Physically contiguous memory is otherwise the same as normal memory.

See the *Programming Startup Guide* for further information.

CDRAM

All of CDRAM is physically contiguous. The libgxm parameter buffer must reside in CDRAM. The default size of the parameter buffer is 16 MiB.

Memory Caching Modes

LPDDR can be allocated as either cached or uncached. However, CDRAM is always allocated as uncached.

CPU and GPU Caching Hardware

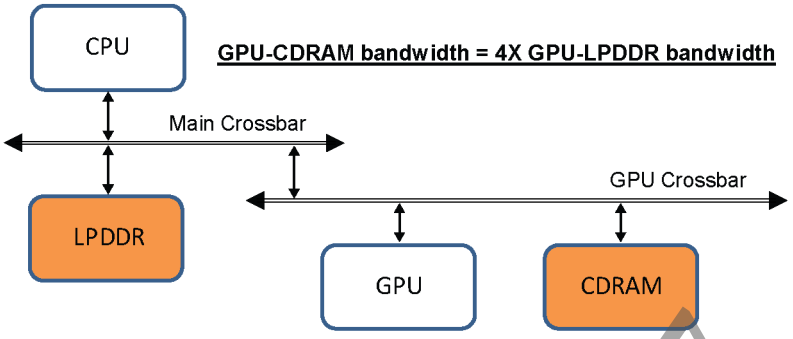
The CPU has L1 and L2 caches, both having a line size of 32 bytes. The L2 cache is used when accessing cached LPDDR.

The GPU has several inner caches. Memory accesses from GPU are cached in the GPU's inner caches, regardless of the caching mode specified when the memory was allocated.

When accessing cached LPDDR, the GPU is also able to snoop the CPU L2 cache.

Memory Bandwidth

Figure 2 Simplified Block Diagram Showing LPDDR and CDRAM Locations



Both LPDDR and CDRAM can be accessed from either CPU or GPU. You may be able to improve performance by locating GPU resources (including surfaces and textures) in CDRAM, because available bandwidth between GPU and CDRAM is much higher than between GPU and LPDDR.

CPU TLB Page Sizes

The CPU MMU (memory management unit) supports the following page sizes:

- 4 KiB, 64 KiB, 1MiB
- 16 MiB (LPDDR only)

TLB (translation lookaside buffer) misses can be reduced by allocating memory blocks that match the MMU supported page sizes.

Virtual Addressing

All user-space addresses are virtual, including those that correspond to memory in a physically contiguous area.

2 Software Overview

Memory Allocation

Kernel Allocation API

`sceKernelAllocMemBlock()` is the main interface for allocating user memory. The maximum number of allocated memory blocks is 2048.

`sceKernelFreeMemBlock()` frees these memory blocks.

See *Kernel Reference* for more information.

Size Alignment Requirements

The size alignment requirements when allocating memory blocks using the kernel allocation API are described below.

Table 1 Size Alignment Requirements When Using the Kernel Allocation API

Memory Block Type	Size Alignment Requirement
LPDDR	4 KiB
LPDDR (physically contiguous)	1 MiB
CDRAM	256 KiB

Memory Usable by Applications

Applications cannot allocate the full LPDDR memory using `sceKernelAllocMemBlock()`. There are various system memory requirements, including memory required to load the program (eboot.bin), memory reserved by the kernel and for the libc heap area.

Memory is also required for stack areas, TLS, and for PRX files that are loaded using libsysmodule.

See the *Programming Startup Guide* for further information.

Allocating Memory for the Codec Engine

Codec engine allocations must be "to order". In other words, you cannot pass a pointer from the middle of a larger pre-allocated memory block to any function that specifies physically contiguous memory.

Allocation Strategies

Due to both execution overhead incurred when allocating memory via the Kernel Allocation API and the size alignment restrictions described above, applications should allocate large memory blocks at startup and perform smaller allocations for application data using a custom allocator.

A heap allocator is not necessarily the best choice because some resources (for example, GPU resources) may have a relatively long lifetime. Therefore, a simple "stack allocator" may work well.

An Example Custom Heap Allocator

libheap is provided as an example implementation of a simple heap allocator. Source code is provided and developers are encouraged to modify it to fit their needs.

See *libheap Overview* for more information.

Configuring the libc Default Heap

The default size of the libc heap is 256 KiB.

You can configure the size of the default heap by defining the global variable `sceLibcHeapSize`.

See *C and C++ Standard Libraries: Overview and Reference* for further information.

Replacing the C and C++ Standard Library Memory Management Functions

You can replace the C/C++ Standard Library memory management functions with user-defined versions. This involves linking in an object file with user versions of each function – for example `user_malloc`, `user_free`, `user_new`, and `user_delete`.

When replacing the memory management functions, all of the functions must be defined regardless of whether they are used.

See *Memory Management Function Replacements of the C and C++ Standard Libraries: Reference* for further information.

Memory Mapping

Mapping Memory for GPU Access

`sceGxmMapMemory()` is used to map memory for use by the GPU. Textures, surfaces, and vertex and index buffers must be placed in mapped memory.

All of main memory can be mapped, with one exception. Render targets require their own unmapped memory blocks. If a memory block ID is supplied to `sceGxmCreateRenderTarget()`, the function will return an error if the memory block is mapped.

However, it is beneficial to only map memory containing GPU resources, to increase the probability of catching out-of-range memory accesses by the GPU. When the GPU accesses unmapped memory, a page fault will be generated.

Mapping Memory for USSE Access

`sceGxmMapVertexUsseMemory()` and `sceGxmMapFragmentUsseMemory()` are used to map code memory for access from USSE. Fragment and vertex shader code must be placed in USSE-mapped memory.

USSE-mapped memory is also required for the initialization of `SceGxmContext` and `GxmShaderPatcher`.

You can map the same memory block for use by fragment and vertex programs. The shader patcher can take advantage of this, sharing USSE-mapped memory blocks between fragment and vertex shader code.

USSE-mapped address space is limited to 32 MiB.

Freeing Mapped Memory

Memory must be unmapped before being freed. Make sure that the GPU is not accessing unmapped or freed memory. To achieve this, use `sceGxmFinish()` to wait for rendering completion before unmapping and freeing memory.

See *libgxm Overview* for more information.

3 PTLA Unit and Transfer API

Introduction

PTLA is a fixed-function transfer unit within the GPU. It operates asynchronously to the GPU cores and CPU. The PTLA unit is used via the libgxm Transfer API.

Supported Features

PTLA supports format conversion, memory layout conversion, downscaling, copying, and filling of 2D images. The downscale factor is fixed at 50%.

The maximum size of a single transfer is 4 MiB. This corresponds to the 2D image sizes shown in Table 2.

Table 2 Maximum Image Sizes Supported by the PTLA Unit

Transfer Format	Maximum Size
RAW128	512x512
RAW64	1024x512
All other formats	1024x1024

PTLA also supports conversion between linear and swizzled formats, and between linear and tiled formats. The source and destination fundamental formats must be the same (RGB, YUV or RAW).

See the *libgxm Overview* and *GPU User's Guide* for more information.

4 Memory Performance

Introduction

PlayStation®Vita supports several different memory types and caching modes:

- LPDDR (cached/uncached)
- Contiguous LPDDR (cached/uncached)
- CDRAM

You can copy data within memory using either CPU-side `memcpy()` or the PTLA unit. With this in mind, the goal of the following test is to collect data that allows conclusions to be made about which memory types and copy method are optimal in terms of memory throughput.

Test Configuration

The test involves repeatedly copying a fixed-size area of memory, using various block sizes, memory types, and caching modes.

The test has the following attributes:

- Copy size: 4 MiB
- Power Configuration: Mode A

This is very much a "best case" test because in each case only CPU, GPU, or PTLA is active. In a real-world application, multiple units will be active at any time and will compete for memory bandwidth.

Test Results

Test results are provided for the following:

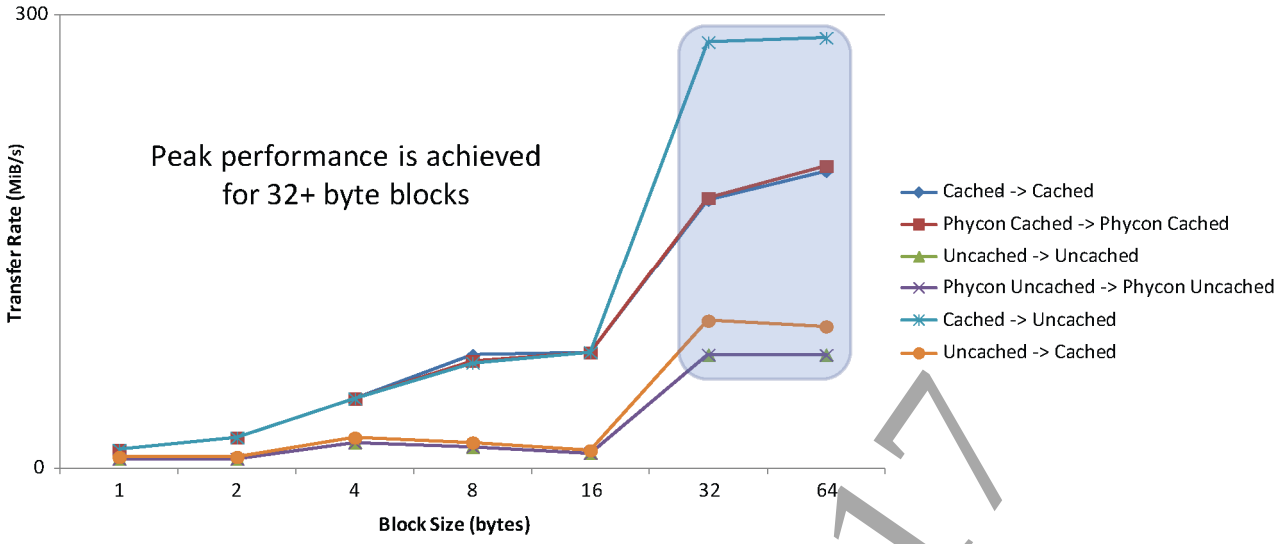
- LPDDR memcpy
- CDRAM memcpy
- LPDDR PTLA
- CDRAM PTLA
- GPU Read
- GPU Write

LPDDR memcpy

This section examines throughput when copying within LPDDR.

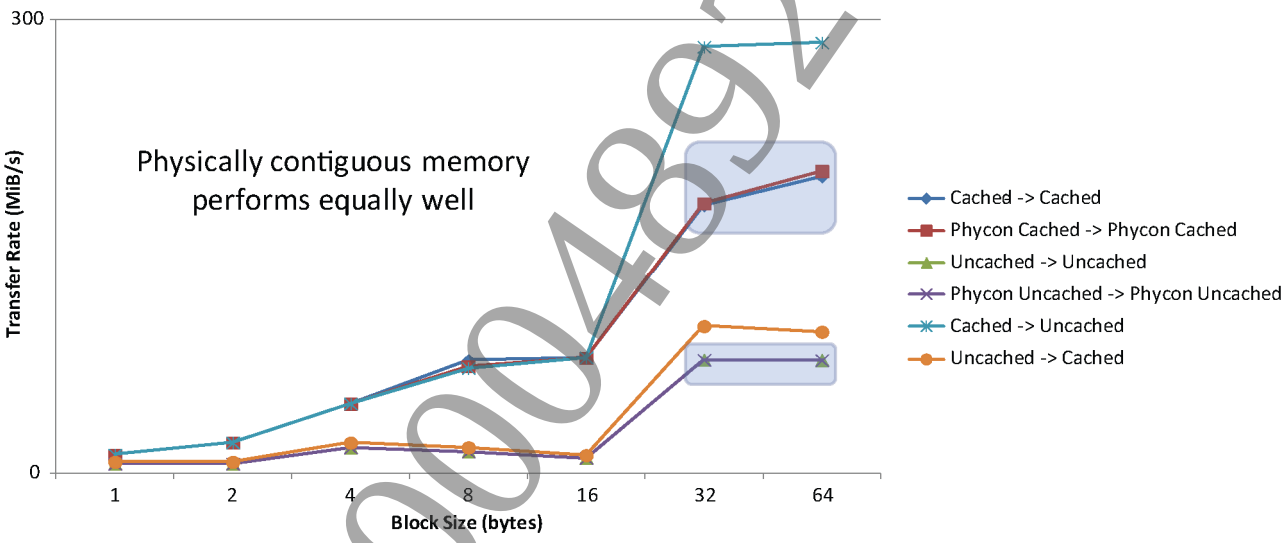
As shown in Figure 3, `memcpy()` reaches peak throughput for 32-byte and larger blocks. This is because for 32-byte and larger blocks, `memcpy()` uses a NEON-optimized inner loop.

Figure 3 Performance by Block Size



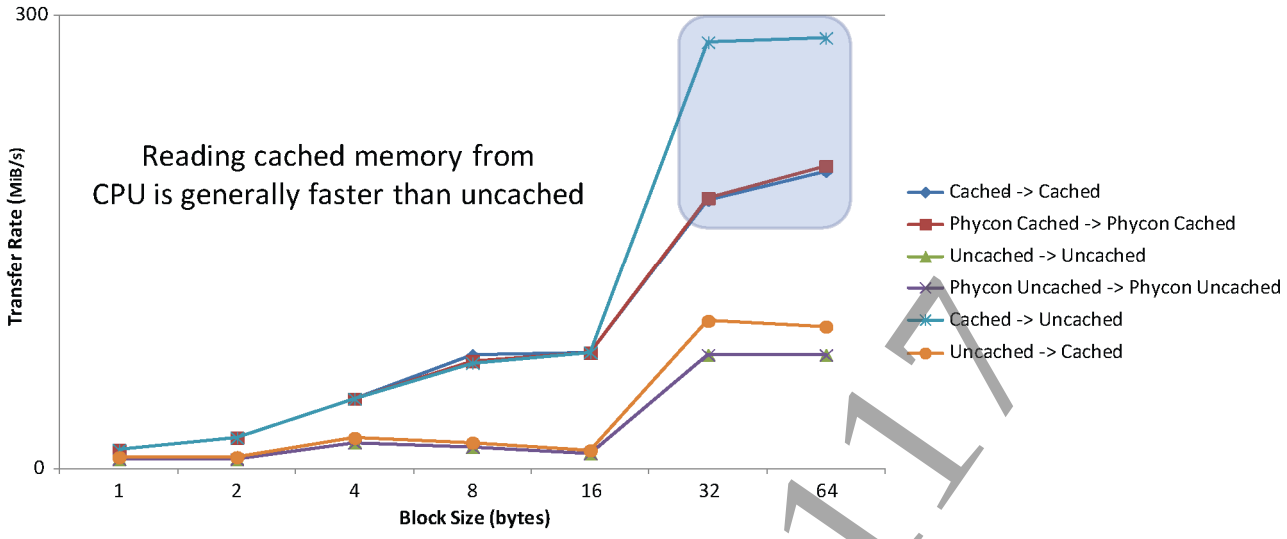
As shown in Figure 4, physically contiguous memory performs identically to non-contiguous memory.

Figure 4 Performance of Physically Contiguous Memory vs Non-contiguous Memory



As shown in Figure 5, reading cached memory from CPU is generally faster than reading uncached memory.

Figure 5 Performance of Reading Cached vs Uncached Memory

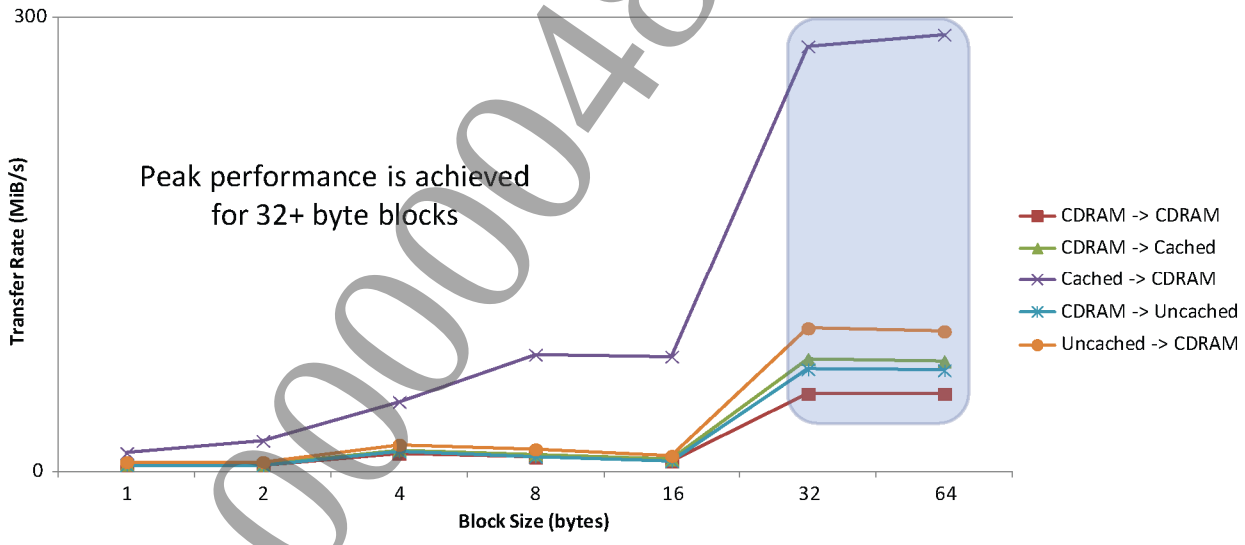


CDRAM memcpy

This section examines throughput when copying within CDRAM and between CDRAM and LPDDR.

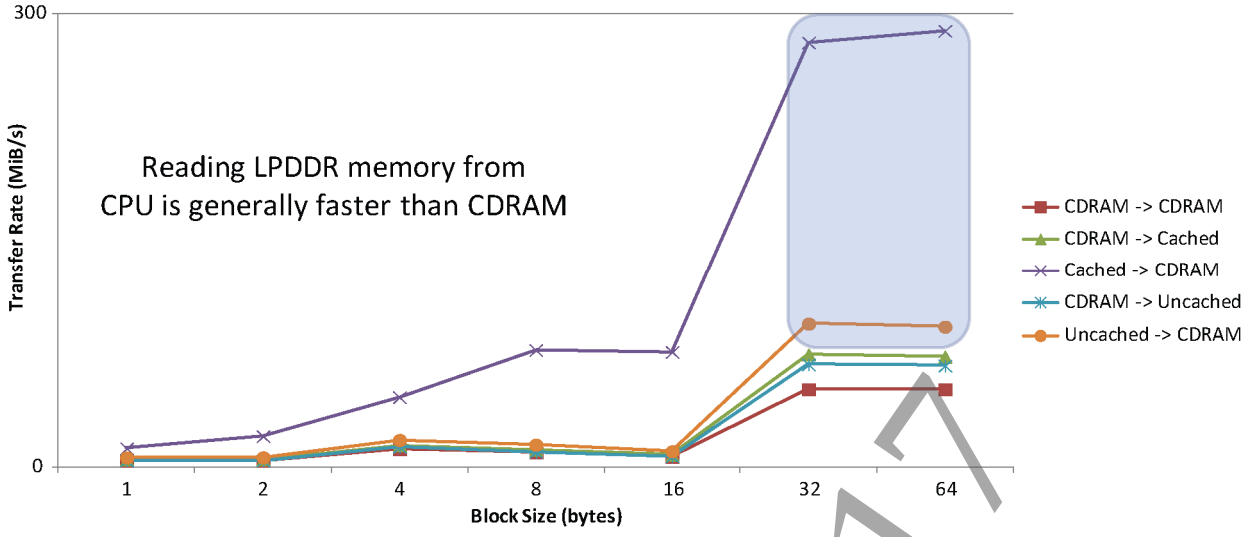
As shown in Figure 6, once again peak throughput is reached for 32-byte and larger blocks.

Figure 6 Performance by Block Size



As shown in Figure 7, reading LPDDR (either cached or uncached) from CPU is generally faster than reading CDRAM.

Figure 7 Performance of Reading LPDDR Memory from CPU vs Reading CDRAM

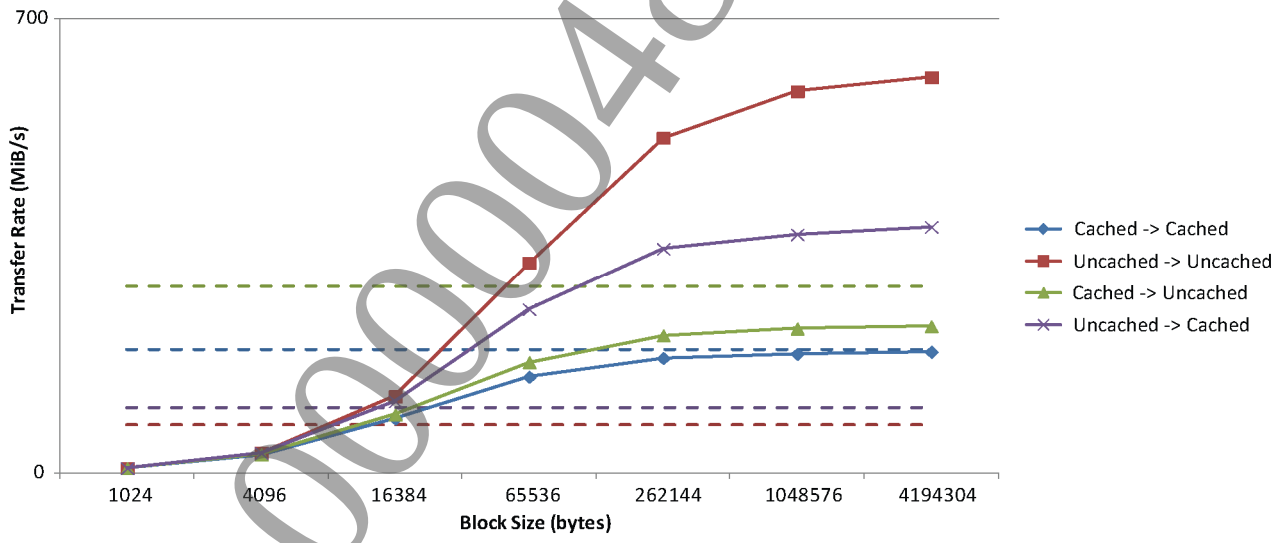


LPDDR PTLA

This section examines throughput for the PTLA unit. Because the PTLA unit is part of the GPU, it benefits from additional bandwidth when accessing CDRAM compared to LPDDR.

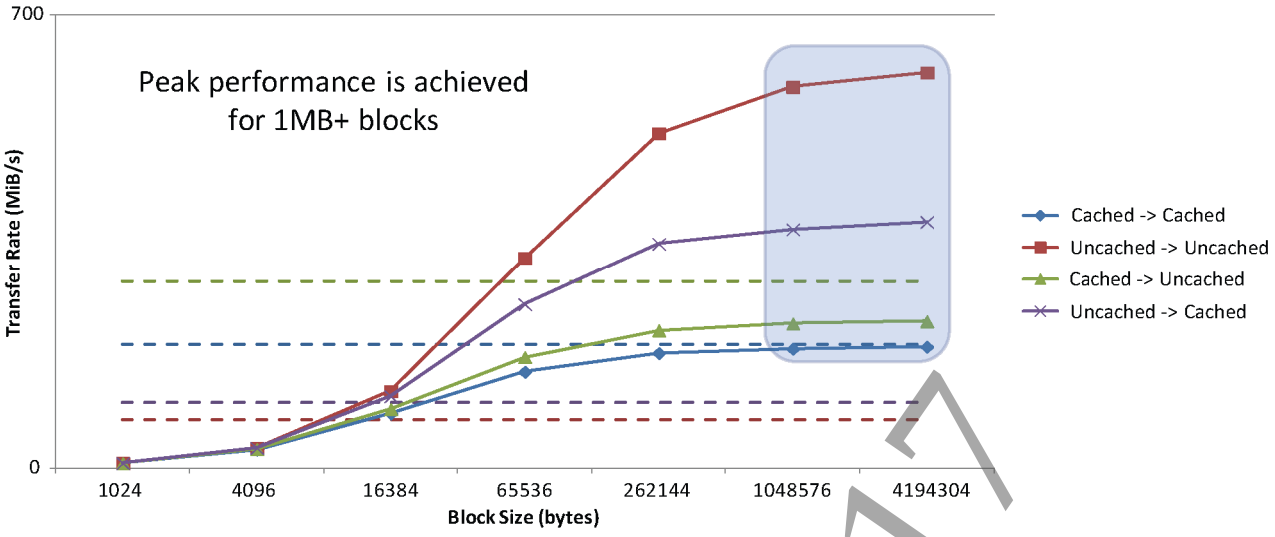
In Figure 8, the dotted lines indicate the throughput of `memcpy()` for each combination of memory type. PTLA throughput only exceeds `memcpy()` throughput when reading from uncached memory in blocks larger than approximately 16 KiB.

Figure 8 Performance by Block Size (1 of 2)



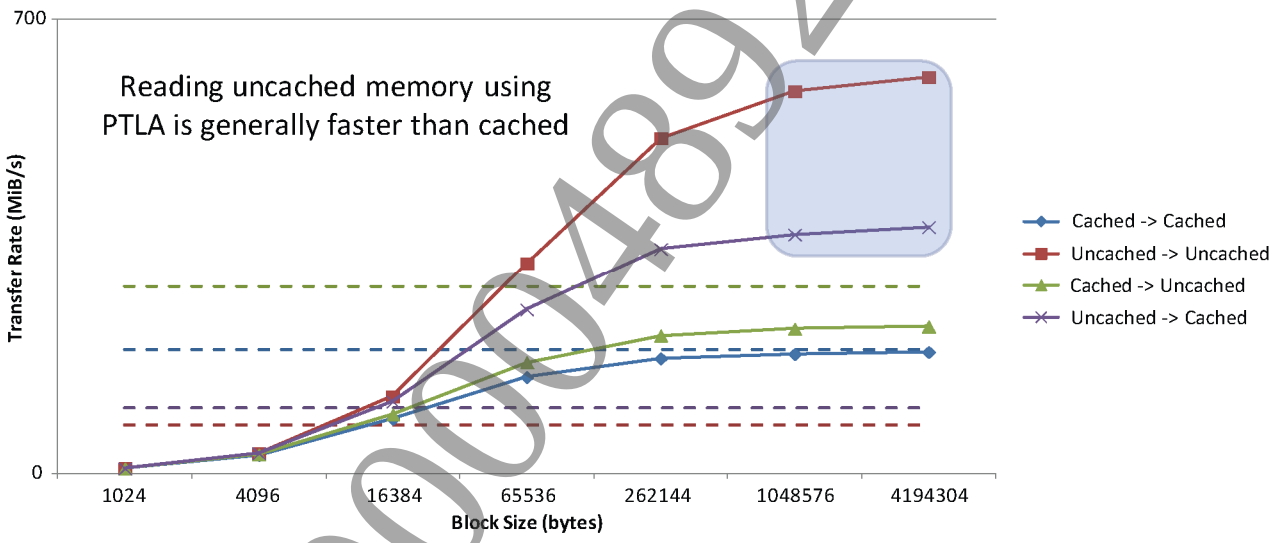
As shown in Figure 9, the PTLA unit reaches peak throughput for 1MiB and larger transfers. This indicates a significant per-transfer overhead.

Figure 9 Performance by Block Size (2 of 2)



As shown in Figure 10, using PTLA to read from uncached memory is generally faster than from cached memory.

Figure 10 Performance of Reading Cached vs Uncached Memory



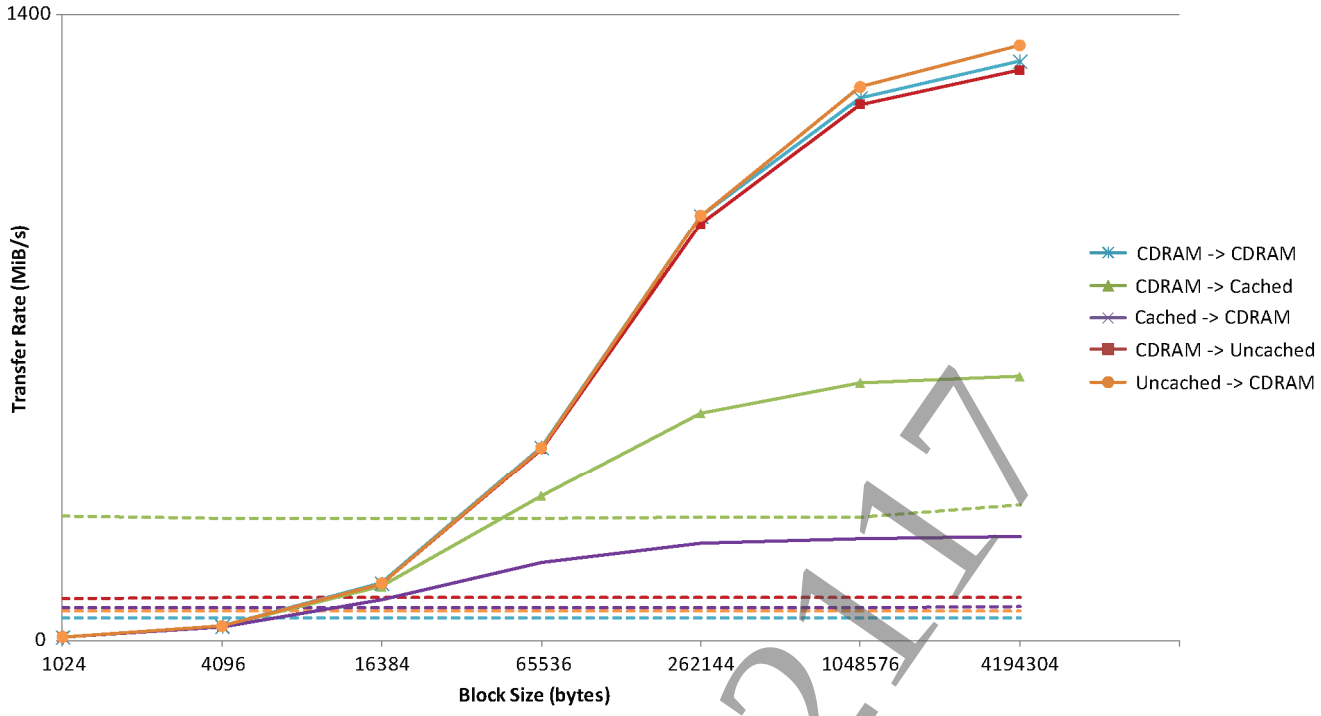
CDRAM PTLA

This section examines throughput for PTLA copies within CDRAM and between CDRAM and LPDDR.

In Figure 11, the dotted lines indicate the throughput of `memcpy()` for each combination of memory type. PTLA throughput exceeds `memcpy()` throughput when reading CDRAM and uncached memory for block sizes greater than 16 KiB.

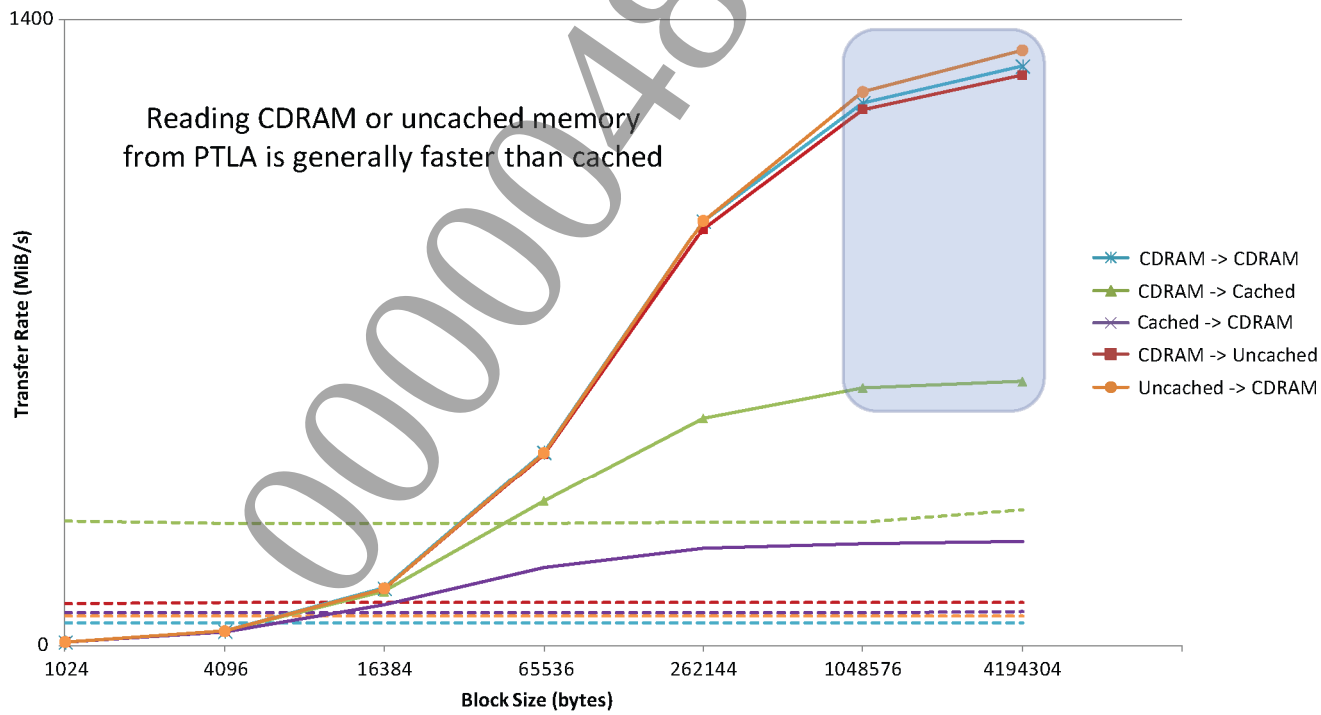
Note that the performance of the transfers involving CDRAM might be reduced if the source reads and the destination writes occasionally conflict with each other on the same memory access channel.

Figure 11 Performance by Block Size (1 of 2)



As shown in Figure 12, using PTLA to read from CDRAM or uncached memory is generally faster than from cached memory.

Figure 12 Performance by Block Size (2 of 2)

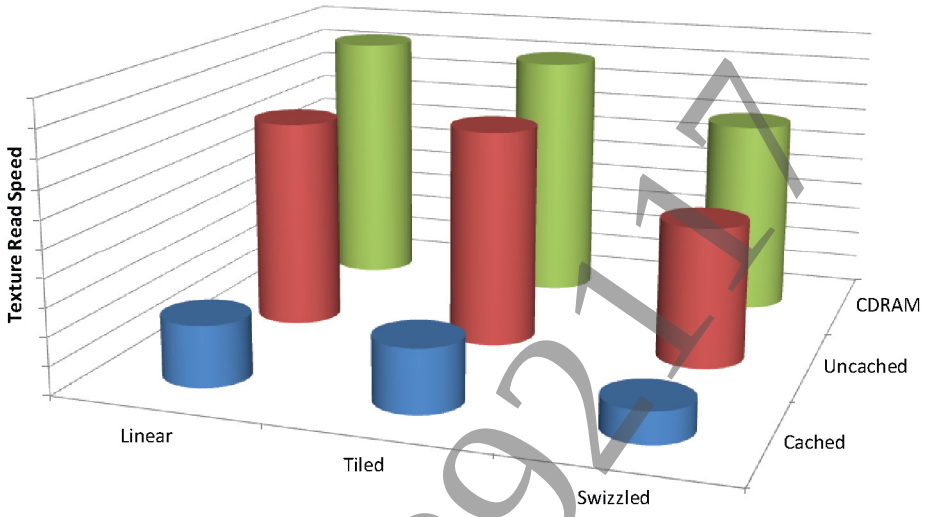


GPU Read

This section examines how texture location and addressing mode affect GPU read speeds.

Figure 13 shows GPU texture read speed when reading a 1024x1024 uncompressed texture. The figure demonstrates that reading a texture in cached memory is significantly slower than reading from CDRAM or uncached memory. In this example, the read speed for linear texture is highest, but in cases where cache access (and so forth) is generally efficient, tiled textures or swizzled textures will be the most optimal.

Figure 13 GPU Texture Read Speed for a 1024x1024 Uncompressed Texture

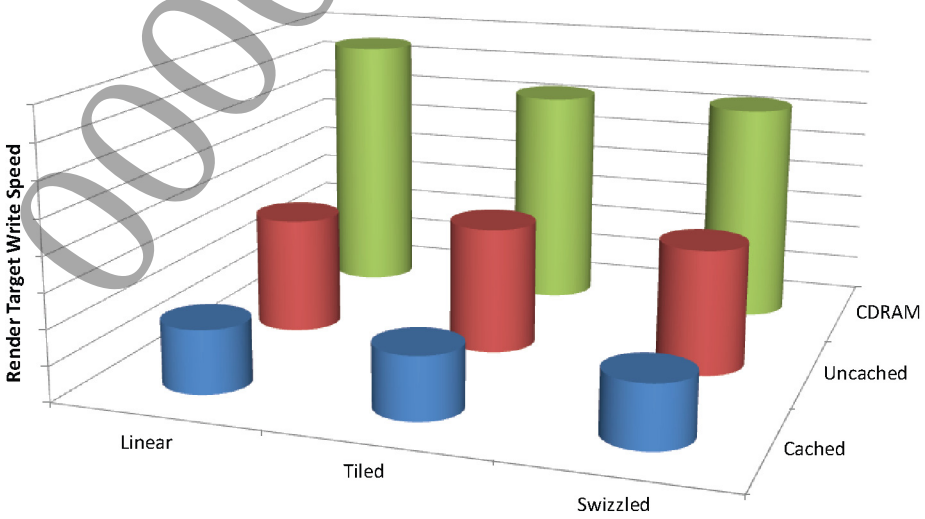


GPU Write

This section examines how render target surface location and addressing mode affect GPU write speeds.

Figure 14 shows GPU target write speed when writing to a 1024x1024 surface. Similar to the previous test, writing to CDRAM is fastest. The drop in performance for uncached memory is quite pronounced, and cached memory is slower still. Writing to linear CDRAM surfaces is generally faster than to swizzled or tiled surfaces.

Figure 14 GPU Target Write Speed for a 1024x1024 Surface



5 Other Recommendations

Optimal memory usage is key to realizing the full potential of PlayStation®Vita. In addition to the recommendations made elsewhere in this tutorial, there are several other points to be aware of.

Resource Location

Unless there is a compelling reason not to do so:

- CPU resources should be located in cached LPDDR
- GPU resources should be located in either CDRAM or uncached LPDDR

Cached Memory Issues

The cached memory access port is narrow compared to accessing uncached memory. The effect of this can be seen in the throughput results, especially for accesses from GPU.

In addition, the GPU will access cached memory through the CPU L2 cache.

However, cached memory access is fastest from CPU. In scenarios where CPU must consume GPU-produced data, it is best to use cached LPDDR, but keep the amount of data to a minimum.

PTLA Unit and LPDDR

If LPDDR is specified as both the PTLA transfer source and destination, the PTLA transfer speed can be slower than for CDRAM because the main memory bandwidth will be a bottleneck.

If cached LPDDR is specified as transfer source or destination (or both), the PTLA transfer speed can be much slower than for CDRAM.

Balance Bandwidth Between LPDDR and CDRAM

By default, GPU resources should be located in CDRAM.

However, it may be beneficial to allocate some graphics resources in uncached LPDDR to balance bandwidth usage or to allow more graphics assets if CDRAM is full.

Use Razor CPU and GPU Together

If in doubt about the most optimal memory location for a given data type, profile your application using Razor CPU and GPU together. This will allow you to see the effect of different memory types on both CPU and GPU performance.