Parker Software Optimization Guide

Application Note

DA-07334-001 | July 2018

Document History

DA-07334-001

Version	Date	Description of Change
01	July, 27, 2018	Initial release

Table of Contents

Overview

This document contains recommendations for optimizing CPU workloads for the Nvidia Parker mobile processor. This document is primarily focused on optimizing on the Denver-2 cores. The guidelines provided in the attached A57 SW Optimization Notes document should also be followed, since the Parker processor includes both A57 and Denver-2 Cores.

Single-threaded Workload Optimization

This section provides guidance on optimizing for single-threaded workloads.

Avoid Unnecessary Data Movement Between Floating-Point and Integer Registers

Performance characteristics degrade when you move data from integer registers to floating-point registers for floating-point operations, and then transfer that data back to integer registers before storing in memory.

Guidance

- Reduce the number of floating point and integer register interactions by using native `float` and `double` data types and reducing casts between types in the software. This does not require the use of intrinsics or compiler changes.
- If there are both integer and floating point versions of the data, use integer operations provided by the Advanced SIMD instruction set, which operate on the floating-point register file.
- Use direct loads and stores to the floating-point registers (VLD/VST in `AArch32`; FLD/FST in `AArch64`.)
- If integer versions are not required, use direct loads and stores to the floating-point registers (VLD/VST in `AArch32`; FLD/FST in `AArch64`.) with VFP or floating-point flavors of Advanced SIMD operations. This might require usage of intrinsics. Alternatively, an improved compiler can help.

Background

This behavior causes poor performance on all processors, but the integer to floating-point register transfer in Denver-2 does not have a dedicated transfer path, and so it might be slower than other ARM microarchitectures.

Avoid or Minimize Load or Store Operations Crossing Cache-Line Boundaries

Memory access operations that cross cache-line boundaries are slower than load or store operations that remain within a single cache line. The cost of these boundary crossings might be up to 20 extra cycles. In common cases, the cost is usually 1 extra cycle for integer load and store operations, or 2 extra cycles for floating-point loads and stores.

Guidance

- For wide operations, like those commonly performed in `memcpy` routines, try to keep every static ARM load or store operation aligned.
- Keep the stack pointer cache-line aligned and ensure that stack stores and loads are cache-line aligned.

Background

Memory accesses that cross cache-lines requires loading or storing to two separate cache lines, which incurs a performance penalty. The dynamic code optimizer detects common cache-line crossers and emits code that executes unaligned load and store operations in a performant manner. But unaligned load and store operations require more operations and consequently consume more cycles and power.

Avoid or Minimize MCR/MRC/MSR/MRS/SYS Instructions

The following system register access instructions can be slower than in other processor implementations:

- MRC
- MCR
- MCRR
- MRRC
- MSR
- MRS

Guidance

Minimize usage of these system register access instructions.

Background

System register access instructions are implemented in microcode. Because the branches are data-dependent, there is significant potential for incorrect multiple branch predictions.

Avoid or Minimize Loads from Device Memory Types

Uncached (both device and non-streaming normal non-cacheable) loads in Denver-2 are strongly serializing. There is only one outstanding at a time, similar to UC loads in x86 processors. When the data cache is turned off, all loads are either Device or normal-non-cacheable. When the MMU is off, all loads are Device loads. Because of this, and due to their strongly serializing nature, they might cause performance degradation and should be reduced to a minimum.

Guidance

Turn the MMU and data cache on as quickly as possible. Use normal memory types only when accessing DRAM. Either use cacheable normal memory when accessing DRAM, or use streaming (non-temporal) loads when accessing normal non-cacheable memory.

Background

The Denver-2 memory system is essentially a fully out-of-order memory system, which does not preserve order between independent loads. For cacheable loads, this is not problematic because the coupling between the commitment logic and coherence snoops resolves potential ordering issues. However, for uncached access operations (Device and normal non-cacheable), there are no coherence snoops, and hence you must preserve order by other means. In Denver-2 (as in x86 micro-architectures), this is done by completely serializing such uncached loads. In this way there is only one outstanding at a time. If there were multiple outstanding uncached loads they could lose order.

Streaming (non-temporal) loads to normal non-cacheable memory have relaxed ordering constraints. There can be multiple such loads outstanding. Thus, for bulk data transfers, either use cacheable loads or streaming (non-temporal) loads to normal non-cacheable memory.

Batch Non-Load/Store Operations Requiring DMB/DSB/ISB

Barriers that are used to guarantee ordering and/or completion of operations other than load and store operations might be expensive. This includes:

- DMB barriers used to order cache maintenance operations with memory access operations.
- DSB barriers used to order cache maintenance operations with memory access operations, or complete broadcast TLB invalidations.
- DSB barriers which are blocked by a MMIO device access on another core, where the MMIO access takes a long time due to a timeout or other long-latency operation.
- ISB barriers used to guarantee that system register access operations take effect.

DMB and DSB barriers used to order just load and store operations are relatively inexpensive. ISB barriers used to flush the instruction fetch pipeline are also relatively inexpensive.

Guidance

Barriers with additional ordering or completion properties can degrade performance. Also, DSBs that complete TLB invalidations cause time to be spent on another core to complete the invalidation. Such barriers should be batched as far as possible.

Background

Barriers that have additional ordering or completion properties cause an internal micro fault, and their additional ordering or completion properties are handled in microcode. DMB and DSB barriers that merely order load and store operations, and ISB barriers that merely flush the instruction fetch pipeline, do not cause such micro faults. In fact, DMB and DSB barriers, when used only to order cacheable memory access operations, are effectively NOPs.

Avoid or Minimize Breakpoints and Watchpoints as They Might Interfere with Optimizations

Some breakpoints and watchpoints logic may potentially lead to unexpected speed decreases, even when no breakpoint or watchpoint related code is executed.

Guidance

Use of Breakpoints and Watchpoints should be minimized. If used, it should be done so with the understanding that they can cause performance losses even if the code at which they are placed isn't executed.

Background

Some breakpoint and watchpoint logic is implemented on 4k sized granules, potentially leading to unexpected slowdowns even when no breakpoint or watchpoint produces a match. This is because optimizations can fail, resulting in code being executed through the less aggressive HW decoder, even if the actual code path executed doesn't contain the breakpoints/watchpoints.

Avoid Counting Architectural Instructions outside of Code Profiling

Counting architectural instructions might cause a small (~1%) reduction in program execution speed.

Guidance

Turn on architectural-instruction counting only when profiling code during application development. When turned on, the impact to the profile is minimal, and the profile is accurate.

Background

Architectural instructions are counted using microarchitectural operations that occupy functional units.

Use LDRD/STRD Instead of LDM/STM Where Possible

LDM and STM operations are not particularly advantageous on the Denver-2 microarchitecture, especially when compared to LDRD/STRD.

Guidance

Compilers should use LDRD/STRD in preference to LDM/STM and try to use 8-byte-aligned addresses for LDRD/STRD.

Background

LDM/STM might cause performance problems when the memory region being loaded or stored to crosses a cache line boundary, or the starting address is not 8-byte aligned.

Avoid Modifications to Pages Which Contain Executable Code

On Denver-2, each time code is modified dynamically, that code might temporarily incur a significant reduction in speed. If any part of a 4k page is modified, any code on that page is treated as though it is modified, incurring the same speed reduction.

Guidance

In general, avoid this type of self-modifying code. Try to place mutable constant pools on a separate page from executable code. Constant pools that are immutable should remain on the same page as the code.

Avoid placing mutable data on the same page as executable code. In particular, JIT compilers should evaluate strategies that avoid allocating generated code on the same page as mutable data, including garbage-collection bits.

JITs should batch writes of dynamically generated code on a single page, so that the cost of modifying the page is not incurred repeatedly by a series of separate modifications to the page. This guideline applies both to the initial generation of code and to any later modifications.

Background

Self-modifying code may render dynamically optimized code invalid. The protection to detect modified code operates at the granularity of a 4k page. If the code has not actually changed, the dynamically optimized code may be kept, at the cost of reading the ARM code and validating that it remains unchanged. If the validation fails, the optimized code must be regenerated at a significant cost.

Use L2 Cache Clean and Invalidate by Address Instructions

Cache Invalidate operations by set and way cause not only the L2 cache to be completely flushed but also all dynamic code optimization flows to be invalidated, and subsequently re-validated on demand.

Guidance

Cache Invalidate operations by machine virtual address are preferred over operations by set/way. If you must use invalidate by set/way, use ROC flush instructions which do not invalidate instructions (CCPLEX cache flush only vs cache flush trbits). If the regions are too large, consider using normal non-cacheable memory and `memcpy` to and from cacheable memory.

Background

Cache operations by set and way is imprecise and might result in loss of all coherence between data and dynamically optimized code. As such, it causes all dynamically optimized code flows to be invalidated, and they need to be re-validated on demand, causing performance degradation.

Avoid Code reuse that Accesses Uncacheable Memory or Uses Unaligned Accesses

As explained elsewhere in this document, accessing uncacheable memory performs more slowly than accessing normal, cacheable memory. Similarly, a memory access operation that crosses a cacheline boundary performs more slowly than a memory access operation that is aligned to remain within a single cacheline. If the same piece of code is used for both the cases that are expected to be slow and for the cases that are expected to be fast, the performance of the code frequently resembles the performance of the slow case, even when executing the fast case.

Guidance

There should be few pieces of code that actually need to be applied to the slow cases. Create separate copies of that code and use them only in those cases. For example, there should be a separate version of `memcpy` used only for device memory. Never apply the version of `memcpy` that is applied to normal, cacheable memory to device memory.

Generally, this code duplication should be necessary only in a limited set of functions that need to access uncacheable memory. Avoid unaligned access operations where possible, so that code duplication is unnecessary for that case.

Background

The dynamic code optimizer changes the code that it generates to mitigate performance losses from cacheline crossing memory access operations and from access operations to uncacheable memory. But while those changes improve performance for unaligned memory access operations and uncacheable access operations, they result in reduced performance when applied to properly aligned memory addresses and cacheable memory.

This tension can be avoided, for example, by duplicating the ARM code and applying one version exclusively to uncacheable memory and the other to cacheable memory. This duplication works because the dynamic code optimizer tailors the generated code appropriately to each version of the duplicated code.

Emit Aligned Registers When Using Floating-Point "S" or "D" Registers

The Denver-2 processor implements a high-performance, 128-bit FP datapath that executes best when operating on aligned FP registers in AArch32 mode. In some cases NVIDIA has measured improvements of 5% for floating-point latencies in AArch32 floating-point code.

Guidance

For AArch32 code, use aligned floating-point "S" and "D" register allocation in your compiler or JIT.

For D registers, source and destination operands for a given operation should be all even or odd, i.e., (src1_reg % 2) == (src2 % 2) == (dest_reg % 2). Thus, VADD D0, D2, D4 is preferred over VADD D0, D1, D2.

For S registers, source and destination operands for a given operation should be aligned in the same 32b sublane of the 128b wide register, i.e. (src1_reg % 4) == (src2 % 4) == (dest_reg % 4). Thus, VADD S1, S5, S9 is preferred over VADD S0, S1, S2.

Background

The Denver-2 CPU cores implement a 128 bit-wide floating-point data path which provides performance benefits for AArch64 and full width advanced SIMD, i.e., 4-wide vector single precision code. However, due to the ARM architectural layout of the AArch32 floating-point register file, source operands that do not use the full width of the register file might be misaligned and require extra latency for alignment.

Set FPCR.FTZ For FP Workloads That Do Not Need Precise Values in the Subnormal Range

Setting FPCR.FTZ (flush to zero) will improve performance on certain operations whose operands or results are in the subnormal range.

Guidance

If an application does not need precise values in the subnormal range, performance can be improved by setting FPCR.FTZ. One way to achieve this with the GCC compiler is to use the GCC flag -ffast-math. This may vary across GCC versions

Background

There are a few FP operations whose performance is slower with denormal inputs or outputs. Setting FPCR.FTZ prevents denormal inputs or outputs by causing the processor to treat subnormal (also called denormal) FP values as zero. In many applications, gradual underflow is not important, so setting this bit will help FP performance.

Use Floating-point “Fused-Multiply Add” Instead of “Floating-Point Chained-Multiply Add”

Enable fused-multiply add instructions in compiled code, or use them in hand written assembly.

Guidance

Compile or write assembly in programs such that multiply followed by add instructions can use a fused-multiply add instructions (e.g. AArch32: VFMA, AArch64: FMADD, FMLA) rather than chained multiply add (AArch32:VMLA) or separate multiply and add instructions. One way to achieve this with the GCC compiler is to use the GCC flags “–mfpu=vfpv4” and/or “–mfloat-abi=hard”. This may vary across GCC versions. Note that chained multiply add is only available in AArch32.

Background

Fused-multiply add instructions use a single rounding step for a multiply and add. In contrast, chained-multiply add instructions require rounding after the multiply and add. As such, the fused versions of these instructions are faster on the Denver-2 processor. The fused versions also use less code space than a multiply followed by an add instruction.

Unroll Loops No More Than Four Times

Excessive loop unrolling can result in degraded performance.

Guidance

Unroll loops a limited number of times. A significant benefit from unrolling more than four times is unlikely.

Background

The dynamic code optimizer unrolls loops further if there is an expected performance improvement. But if the loop has already been unrolled in software, the dynamic optimizer cannot undo the unrolling, which might result in inferior performance due to:

- The dynamic optimizer might not form a region large enough to capture the loop, resulting in missed optimization opportunities.
- For performance reasons, the dynamic optimizer might not allow a transaction long enough to capture the entire body of an excessively unrolled loop, resulting in missed optimization opportunities.
- Excessive unrolling wastes icache space, resulting in stalls and reduced performance.

Multi-threaded Workload Optimization

This section provides guidance on optimizing for multi-threaded workloads.

Use Power-Efficient Locking Code That Respects Cache Lines

Spin locks can consume large amounts of power. Allocating multiple locks on the same cache line causes false contention, resulting in degraded performance and power waste.

Guidance

Follow the lock conventions specified by ARM: if lock acquire code fails to acquire the lock, it should execute a `WFE` before trying again; lock-release code should execute a `SEV` when the lock is released. Allocate each lock on its own cache line, with nothing else on the cache line.

Background

False-sharing due to contended data on a shared cache line wastes time and power on any processor; the cost of contention for cache lines on the Denver-2 processor might be slightly higher than other single-cluster ARM processors.

Maintain LDREX and STREX Instructions as Near to Each Other as Possible

The ARM architecture does not guarantee that any particular thread using exclusives makes forward progress, because its split transaction nature (separate load and store operations) allow other cores to intervene between LDREX and STREX instructions. Note that this is a general issue in all ARM architecture based processors. But the problem could be exacerbated in the Parker processor due to the different performance characteristics of the Denver-2 and A57 CPU's.

Guidance

To reduce the likelihood of this kind of thread starvation occurring, the LDREX and STREX instructions should be as close as possible, both in the code and in time of execution.

In general, only use LDREX and STREX instructions to implement very simple atomic read-modify-write instructions such as atomic-ADD or 'compare-and-exchange, placing the LDREX and STREX as closely as possible both in the code and in time of execution. Accomplish this by reducing the number of instructions dynamically executed between them.

There is no starvation issue with the LDREX/WFE pattern and you can use it freely. Only LDREX/STREX patterns must be carefully constructed.

Be careful not to repeatedly call LDREX/STREX in a tight loop, where no work is done before calling LDREX/STREX again, since it can cause starvation to another core. If your application needs a forward progress guarantee for each thread when using LDREX/STREX, then it must be addressed by the software itself, just like for any other ARM processor.

Background

Forward progress of exclusives in the ARM architecture is difficult to guarantee for microarchitectures that speculatively and aggressively prefetch exclusive/unique ownership. This is because speculative execution in other cores might cause starvation of a core that is attempting an architecturally non-contended atomic read-modify-write operation using exclusives.

The fewer the number of dynamic instructions between the LDREX and the STREX, the smaller the window of vulnerability.

In addition, the smaller (and more sequentially-laid out) the actual code is, the less likely that a capacity eviction from the inclusive L2 (that also includes the coherent I-caches) will evict the monitored line.

Avoid Frequent Lock Contention Due to LDREX/STREX Instructions

Since the probability of each core-type in the Parker processor winning a lock is not the same, excessive lock contention between threads on different cores can degrade performance.

Guidance

Avoid lock-contention with LDREX/STREX instructions where one or more threads are frequently (roughly once every 1000 cycles) requesting the lock with these instructions.

Background

Since there are differences in the micro-architectures of the Denver-2 and A57 cores in the Parker processor, the probability of winning a lock through LDREX/STREX is significantly higher for the Denver-2 cores than the A57 cores. A situation can arise where one thread is on a Denver-2 core and the other is on a A57 core, and the A57 thread wins the lock a low percentage of the time.

Core Selection

This section provides guidance on selecting cores for applications in an optimal manner

General Guidance on Core Selection for Workloads

The two core types in the Parker processor have differing performance and power characteristics. This section gives guidelines on how to determine which type of core is best to run a workload on, if the user/developer has specific performance and power requirements for these workloads.

Guidance

On Unix/Linux/Android based kernels, the *taskset* utility is available and can be used to pin an application to a given core ID or set of core ID's. Core ID information can be obtained with the command *cat /proc/cpuinfo* in Unix/Linux/Android based kernels.

1. Measure performance individually on the Denver-2 and A57 cores using *taskset* or some other similar utility to peg the application to the specified core type.

2. If the primary consideration is performance and power is not a concern, then run the application on the processor that gives either the higher performance, or adequate performance.

3. If the primary consideration is power and performance is not a concern, then run the application on the A57 core(s).

4. If both Perf and Power are important considerations, it is best to run the application on the core-type with the best Perf/Watt. To estimate this, look at the performance difference between Denver-2 and A57 cores. If the Denver-2 cores are faster by 25% or more, then running on Denver-2 will likely be better for Perf/Watt. If the performance benefit is less than 25%, then running on A57 will likely be better for Perf/Watt. In the latter case, it would be preferable to let the task run on A57 if its lower performance is acceptable.

Background

In general, the Denver-2 cores deliver higher performance than the A57 cores, but at a higher power cost. The A57 cores generally consume less power, but deliver lower performance than the Denver-2 cores. However, there may be some specific cases where the A57 cores have better or equal performance. The guidelines above have been provided based on these considerations.

Do Not Use Core Detection Mechanism That Treats Parker as big.LITTLE

The Parker processor has two Denver-2 cores and 4-A57 cores, all of which can deliver high performance. As such, it should not be treated as a big.LITTLE system where some of the cores are designed only for low-power use-cases and have limited performance capabilities.

Guidance

When applications get a core count from the Kernel, it should be ensured that the related code does not identify the Parker processor as a big.LITTLE system with fewer than 6 “Big” cores.

Background

Applications may determine the number of threads that should be spawned based on the number of cores in the system. Further, they may try to determine if the system is big.LITTLE and only count the “Big” cores, since the “Little” cores have limited performance. Doing so will limit the performance potential of the Parker processor, as both the Denver-2 and A57 clusters are capable of delivering high performance. For example, code that only recognizes the Denver-2 cores as “Big” will spawn threads based on a core count of 2, while the full performance potential of the system will only be achieved with a core count of 6.

Software Scheduling on Parker Cores

One of the following two approaches to scheduling may be suitable, depending on the use case and application requirements.

Static Scheduling

The Linux operating system exposes several features that can be used to explicitly partition the CPUs in the system into different sets. Notably, these are:

- isolcpus - This kernel command line parameter isolates a subset of cores from the general system. No tasks will be scheduled on these cores, unless explicitly assigned by the user.

- cgroups/cpusets - More flexible runtime configuration mechanism that allows the creation of 'sets' of cpus and groups of tasks, and bind tasks to specific cpusets.

- sched_setaffinity() - System call to specify what subset of cores an individual task is allowed to run on.

The section General Guidance on Core Selection provides guidelines on how to profile applications to determine which core they are best suited to be scheduled on.

A static scheduling approach allows a lot of flexibility for certain applications such as embedded applications, that desire deterministic behavior when it comes to mapping tasks to particular CPU types. However, they require some form of manual profiling of the workload (in terms of performance and power requirements) to determine the correct mapping.

Static scheduling is the default on L4T on Jetson TX2 where all tasks by default are scheduled on A57. The Denver-2's are kept free as 'accelerator' cores for specific workloads. The user / application developer can use sched_setaffinity()/cpusets to map specific tasks to the Denver-2 cores.

Dynamic Scheduling

The Linux kernel in the Parker BSP contains a SoC-specific extension to the scheduler: CAPACITY_AWARE scheduling. With this feature enabled, tasks will be assigned to either the Denver-2 or A57 clusters depending on the number of CPU cycles consumed by individual tasks. The model assumes that tasks will always see better performance (in terms of Instructions Per Cycle) on the Denver-2 cluster, and the policy uses a fixed ratio to convert between Denver and A57 clock cycles. Specifically:

- If a task consumes less than 80% of cycles available on an A57 core at maximum frequency, we assume that task is receiving enough compute capacity on A57 and will schedule it on the A57 cluster; to conserve power and leave Denver-2's available for more demanding tasks.
- If a task consumes more than 80% of cycles available on an A57 core at maximum frequency, we identify this task as 'compute-heavy' and try to place it on Denver-2 cores to optimize for full performance.
- Both rules are subject to the availability of sufficient capacity on the desired cluster (i.e. if there are 3 heavy tasks, we can place only 2 on Denver-2).

CAPACITY_AWARE scheduling is enabled by default on Android for Jetson TX2.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

Arm

Arm, AMBA and Arm Powered are registered trademarks of Arm Limited. Cortex, MPCore and Mali are trademarks of Arm Limited. All other brands or product names are the property of their respective holders. ”Arm” is used to represent Arm Holdings plc; its operating company Arm Limited; and the regional subsidiaries Arm Inc.; Arm KK; Arm Korea Limited.; Arm Taiwan Limited; Arm France SAS; Arm Consulting (Shanghai) Co. Ltd.; Arm Germany GmbH; Arm Embedded Technologies Pvt. Ltd.; Arm Norway, AS and Arm Sweden AB.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com