My Other Computer is your GPU: System-Centric CUDA Threat Modeling with CUBAR

Nick Black and Jason Rodzik
CS8803SS Project, Spring 2010

"The charm of history and its enigmatic lesson consist in the fact that, from age to age, nothing changes and yet everything is completely different."
—Aldous Huxley

Abstract

Heterogeneous computing has definitely arrived, and graphics processing units (GPUs) in the millions are employed worldwide. History has shown newly programmable domains to be rapidly subjected (and often found vulnerable) to attacks of the past. We enumerate a system-centric threat model using Microsoft's STRIDE process [23]. We then describe an active general purpose programming system, NVIDIA's Compute Unified Device Architecture (CUDA) [1], and explore the threat space. We derive and describe previously-undisclosed memory protection and translation processes in CUDA, successfully mount several attacks on the CUDA system, and identify numerous directions for future work. Our CUBAR suite of tools, especially `cudash` (the CUDA Shell), form a solid platform for research to come.

1 Introduction

Within modern desktops, high-performance workstations, and even laptops there exist sources of tremendous computing power: programmable, massively parallel graphics cards. The hundreds of simple in-order cores found on recent NVIDIA and ATI hardware provide many more peak FLOPS than the x86 processor packages with which they're commonly coupled. Devices such as NVIDIA's Tesla™ have already traded video outputs for onboard memory, giving rise to "desktop supercomputing." Those who would fully utilize their machines can no more ignore heterogeneous processing than multiple cores or SIMD.

But is it safe? GPUs have traditionally accelerated single-instance, interactive tasks (such as video games and CAD) and decomposable, compute-bound tasks (such as batch rendering) for which a deinterleaved schedule is optimal. Hardware facilities for the support of multiple threads or processes are primitive at best. When the GPU was limited to blitting, fogging, and anti-aliasing framebuffers, minimal isolation and assurance capabilities were tolerable. Compromises of scientific data, cryptographic materials and SCADA systems are rather more serious, yet these precise applications drive the adoption of GPGPU programming.

We explored CUDA 3.0 on a 2.6.34-rc2 Linux kernel, using version 195.36.15 of the CUDA software. Hardware included a GTS 360M, a GeForce 8400 GS, and a GT 9600. Our source code can be found on GitHub[2].

2 An overview of CUDA

A GPU-accelerated CUDA application on Linux adds four component parts to a typical process:

- A "fat binary" containing host and guest object code, typically x86 and PTX [20]. Note that PTX is a mere intermediate representation, converted to the native CUBIN format at runtime.
- The CUDA userspace library `libcuda.so`. This closed-source archive provides an interface to the NVIDIA driver, and management of a CUDA instance.
- The NVIDIA driver `nvidia.ko`. This closed-source module handles `ioctl`s issued by `libcuda.so`.
- The hardware, controlled by registers and memory-mapped IO.

No complete open-source CUDA solution yet exists.

The CUDA language is an extension of C++, designed for compilation with NVIDIA's `nvcc` [17] toolchain. NVIDIA states that a given device supports multiple host threads, but that a given host thread can control only one

[1]NVIDIA, CUDA, and GeForce are trademarks or registered trademarks of the NVIDIA Corporation.
[2]`http://github.com/dankamongmen/wdp/tree/master/cs8803ss-project/`

device. Data must either be explicitly copied between system and video RAM or, on devices of Compute Capability 1.2 or higher, restricted to shared memory maps [19]. GPU processes ("kernels") can be launched asynchronously, though (until Compute Capability 2.0) only one kernel can be executed at a time by a given device.

CUDA supports numerous memory "spaces", selected via PTX affixes (until the advent of unified addressing in Compute Capability 2.0, presumably effected via memory translation). Spaces differ according to caching, method of initialization, visibility across the device's multiprocessors, and mutability. Official documentation does not address memory protection or translation.

CUDA kernels are typically distributed as JIT-friendly PTX binaries [10], an intermediate representation suitable for all NVIDIA hardware. Upon being loaded onto the card, dynamic compilation[3] is performed, resulting in a locally-optimized CUBIN blob. These blobs are dispatched to Streaming Multiprocessors across the card, all of which share a common memory. A given system thread can use only one device at a time, but a device may be used by more than one system thread.

In our case, the `nvidia.ko` kernel module and `libcuda.so` library weigh in at thirteen and seven megabytes respectively. We can assume them to contain substantial logic. Ultimately, some of our questions can be answered only via analysis of these binaries. This is a matter of some importance for open source projects seeking to duplicate CUDA functionality, such as the Nouveau Project and `libcudest` [2].

2.1 Existing controls

The `/dev/nvidiaX` device nodes control access to the CUDA hardware and kernelspace component. Under the standard Linux security model, these will be restricted via 0660 permissions to a group (typically `video`). Membership in the owning group is thus necessary and sufficient to satisfy the operating system. `strace` output for CUDA applications, along with NVIDIA's `nvidia-smi` device configuration program, show use of the `geteuid` system call, but no further access controls are exported to the user.

3 Threat model

Well-known threat taxonomies include the "CIA Triad" (extended by the "Parkerian Hexad [3]") and Microsoft's STRIDE. The latter, developed as part of Microsoft's Secure Product Lifecycle, is designed for system-centric threat modeling and suitable for our purposes. We assume that an attacker has access to the NVIDIA device node(s) (by default, `/dev/nvidiaX`); such privileges are necessary to compute on the device, and thus a safe assumption. Escalation to device access is an attack on the operating system's access control, and outside the scope of this paper.

We seek to answer the following questions, for machines with one or more CUDA-capable cards:

3.1 Spoofing

- Is it possible for one CUDA kernel to preëmpt data copies requested from another?

3.2 Tampering

- Can one CUDA kernel manipulate another's data set?
- Is it possible to construct a debugging environment around other CUDA kernels?
- Can a CUDA kernel modify the active display?
- Is it possible to pervert compilation processes, either `nvcc` or JIT?

3.3 Repudiation

- Can a CUDA kernel disassociate itself from the system process which spawned it?
- Can a CUDA kernel spawn new CUDA kernels?
- What forensic data, if any, is created as a result of CUDA computing?

3.4 Information disclosure

- Can a CUDA kernel read another kernel's data set? Need they be simultaneously scheduled for this to occur?
- Is it possible to read another CUDA kernel's code, even if it cannot be controlled?
- Is it possible to read the system memory of another CUDA application via calls through the CUDA intermediary?
- Is it possible to reconstruct the system's video channel from an arbitrary CUDA kernel? What about textures?

3.5 Denial of service

- Can a CUDA kernel monopolize resources in the face of competitors?
- Can a CUDA kernel prevent another from being controlled, or executing data transfers to or from the system?
- Can a CUDA kernel deny resources beyond the GPU?

[3]Likely performed in the driver, not the hardware, though we have not yet verified this.

3.6 Escalation of privilege

- Might the driver be exploited, allowing arbitrary ring 0 code to run?
- If it is possible to return doppelgänger data, might it be leveraged to attack (and hopefully exploit) system-side processes?
- Is it possible to arbitrarily (i.e. without exploitation) manipulate another CUDA kernel's code? Is it possible to construct a CUDA virus?
- Is it possible to arbitrarily (i.e. without exploitation) manipulate a system process's code maps from a CUDA kernel?

4 Methodology

Some of the questions asked by our threat model can be answered via simple experiments (others, such as whether the driver might be exploited, are essentially undecidable). Describing the internal mechanisms implementing these externally-visible properties generally requires disassembly, and is the object of further work.

It is first necessary to develop a model of memory protection, multiple user contexts, and memory layout in CUDA. Few details have been made public regarding these topics; what public knowledge exists takes the form of scattered fora posts, Wikis [14], and mailing lists. Foremost among these is memory protection; a trusted multi-process computing base cannot be constructed in its absence. Memory protection for host accesses of the GPU could be implemented at three levels, each more effective than the last:

- the proprietary CUDA and OpenGL libaries,
- the proprietary `nvidia.ko` kernel driver, and
- on the hardware itself.

Userspace protection can likely be thwarted by userspace code, whereas protection implemented within the kernel module ought be secure against all but ring 0 operations (recall that our threat model does not assume access to ring 0). It is unlikely that protection on the card itself can be generally circumvented. Furthermore, this is the only place to protect memory from CUDA kernels.

As an example of the futility of userspace protection, we were trivially able (contrary to NVIDIA documentation) to control multiple devices from a single host thread. Use of the `ltrace` library call tracer indicated CUDA context association to be performed via `pthread_key_t` thread-specific data. By interpositioning ourselves between the binary and `libpthread.so`, we implemented our own context-multiplexing.

The PTX Reference provides some details. Load and store instructions require a "state space" modifier in addition to an address. General-purpose registers (`r0–r127`) are indexed via 7-bit immediates. Global memory, accessed via one of 16 linear ranges of up to 4GB each (`g0–g15`), is addressed via the contents of a 32-bit general purpose register[4]. Constant memory is referenced through one of 16 linear ranges of up to 64KB each (`c0–c15`), as is a block-shared region of up to 16KB. Per-thread local memory, an abstraction atop the global memory, performs address translation based on block and thread indices.

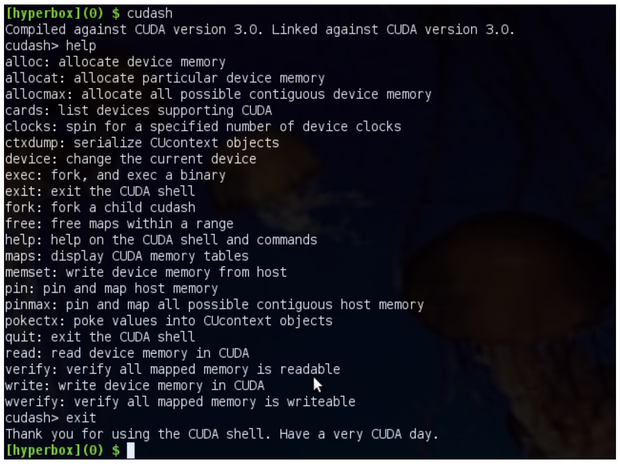
4.1 Tools

Beyond the basic toolchain, we made use of:

- `vbtracetool` [15] to dump video BIOS,
- `cuda-gdb` [18] to debug CUDA programs,
- `strace` [12] to track system calls,
- `nv50dis` [11] to disassemble nv50 binaries,
- `nvtrace` [16] to decipher `ioctls` issued to the NVIDIA proprietary driver, and
- the Linux kernel's MMIOTrace [21] infrastructure to track memory-mapped I/O operations.

We then developed some tools:

- `cudadump` (and its helper binary, `cudaranger`) to discover readable regions in a given virtual memory,
- `cudash`, the CUDA Shell, to perform close-in experiments and prototype attacks, and
- `cudabounder`, `cudaminimal`, `cudapinner`, `cudaquirky`, `cudaspawner`, and `cudastuffer`, a series of single-purpose attack tools.



```
[hyperbox](0) $ cudash
Compiled against CUDA version 3.0. Linked against CUDA version 3.0.
cudash> help
alloc: allocate device memory
allocat: allocate particular device memory
allocmax: allocate all possible contiguous device memory
cards: list devices supporting CUDA
clocks: spin for a specified number of device clocks
ctxdump: serialize CUcontext objects
device: change the current device
exec: fork, and exec a binary
exit: exit the CUDA shell
fork: fork a child cudash
free: free maps within a range
help: help on the CUDA shell and commands
maps: display CUDA memory tables
memset: write device memory from host
pin: pin and map host memory
pinmax: pin and map all possible contiguous host memory
pokectx: poke values into CUcontext objects
quit: exit the CUDA shell
read: read device memory in CUDA
verify: verify all mapped memory is readable
write: write device memory in CUDA
wverify: verify all mapped memory is writeable
cudash> exit
Thank you for using the CUDA shell. Have a very CUDA day.
[hyperbox](0) $
```

[4]Platforms such as the Tesla™ C1060 provide the full 4GB of accessible memory. Compute Capability 2.0 unifies addressing and extends addresses to 64 bits. 40 physical bits are currently supported for up to 1TB of RAM.

5 Attacking CUDA (A dialogue)

We move now beyond the realm of the documented.

5.1 Von Neumann or Harvard architecture?

CUDA appears to be a Harvard architecture. Kernels do occupy video memory, as can be verified by dumping the video RAM. We assert that, by default, a given kernel's code is neither readable nor writeable using the global state space. The former was tested by verifying that global state spaces checksummed to the same value over distinct, subsequent kernels (all of which operated strictly on the shared memory space). The latter was tested by reversing all possible bits in the global memory space, and then performing a series of calculations. It is possible that undocumented opcodes can retrieve or modify code.

This is puzzling: the Harvard architecture's primary advantage is the ability to fetch instructions and data into the CPU simultaneously. We suggest that coherence simplification, combined with the weak memory assurances of the global memory space, motivated this solution.

5.2 Is virtual memory implemented?

CUDA implements virtual memories. We conclusively demonstrated this by examining allocation results in multiple concurrent CUDA contexts. Returned device pointers are equivalent for equivalent allocations in multiple contexts. Allocations begin at 0x101000, honor a minimum alignment of 256 bytes (larger alignments are honored for very large allocations), and otherwise move contiguously through memory. Freed regions can be reclaimed. This suggests multilevel memory allocation, split between user and kernelspace.

Physical memory cannot be aliased or oversubscribed; multiple contexts' maximum allocations cannot add up to more than a single context's maximum possible allocations.

5.3 Does memory protection exist?

CUDA enforces memory protection in hardware. Use of a general-purpose, word-sized register when referencing the global space means kernels' memory accesses cannot be preverified, and the possibility of software-assisted memory protection can be eliminated by analyzing details of memory transactions relative to the device clock [25]. The use of two-level TLBs has been illustrated in previous work [4]. We verified memory protection to be effective beyond the combined capacities of the TLBs, and note a L2 TLB miss to be only about half again as expensive as a L2 TLB hit. Thus we assert that memory protection is performed independently of, and in parallel with, address translation. This requires virtual- (and thus per-context) indexing; together with a memory protection granularity (1MB) distinct from either TLB size (4KB and 8MB), we consider this to imply a backing store in video RAM, and that memory protection is a single bit per entry.

Members of the `cuMemset*()` family of functions, when given an invalid device address, neither return an error nor segfault. Instead, later context functions return a 700 ("previous kernel failed") error. This strongly suggests that memory protection is *not* modeled in software. Further research, especially involving shared, pinned maps, ought investigate this further.

5.4 Does memory protection incorporate contexts?

CUDA hardware considers contexts. To ensure that memory isolation was not being implemented purely through translation, we allocated a large region of memory in one context, and operated on it. We then ensured that a distinct, simultaneous context failed accessing any possible address. This suggests that some manner of unforgeable *capability* [5] lies at the heart of a CUDA context.

Here, we make a controversial conjecture: these capabilities *can* be forged. We draw this conclusion from the interaction of CUDA and the `fork` system call. A CUDA application's system resources are wholly accessible by a child process up until being unmapped via `exec` or some other system call. Furthermore, we verified allocations or kernels executed in either process following the `fork` to be visible in the other. We were unable to forge a context, but think it likely that disassembly of the CUDA stack would make a method plain. It is likely that some memory-mapped I/O is involved independent of the context object itself; `fork`, which preserves maps, thus naturally facilitates context-forging (the `/dev/nvidiaX` maps, as can be seen in `/proc/X/maps`, are mapped `MAP_SHARED` and thus not subject to Copy-on-Write behavior).

5.5 Is memory scrubbed between kernels?

CUDA does not scrub memory. This was demonstrated conclusively by verifying that a large (8MB) random string could be recovered, contiguously and in its entirety, by a subsequent kernel. It was not possible to determine whether code regions were scrubbed, since CUDA does not support indirect branches. The `cudawrapper` [7] tool has been developed to provide scrubbing, but the authors consider it highly dubious that this userspace wrapper could successfully deal with an uncatchable signal.

5.6 Can kernels disassociate from processes?

The CUDA kernelspace tracks processes. This was conclusively demonstrated by sending SIGKILL signals to CUDA processes after they had performed large allocations and launched intense, long-running kernels. There was no opportunity for userspace code to run, yet these resources were properly freed upon process termination.

5.7 Can a kernel fork?

CUDA kernels do not appear capable of forking. A *primum movens* in the form of a host-system CPU is required to launch a CUDA kernel. The kernel-launching interface must, therefore, be visible to the host. There is no compelling reason to then duplicate this interface within the GPU.

Until the mechanism used to launch a kernel is known, however, we cannot be sure that the following loophole does not exist: devices of Compute Capability 1.2 or higher can map host memory into a context's address space. If kernels are launched entirely via memory-mapped I/O transactions, and the map through which this is done is mapped again, this time into shared memory, device code might be able to manipulate the device. At this point, all bets are off.

5.8 Does CUDA produce forensics?

CUDA provides insufficient forensic data. CUDA kernels are not tied into the process accounting mechanism. Exceptions are delivered to klogd, but rate-limited within the kernel itself; it would be trivial to hide a meaningful exception behind a flurry of meaningless ones, as this policy cannot be overridden. No system exists to log who ran what kernel, or for how long.

The `nvidia-smi` tool claims to decode and report exceptions as stored in a hardware buffer, similar to the Machine Check Exception registers of x86. We were not, however, able to generate any informative output using `nvidia-smi`.

5.9 Does CUDA enforce resource limits?

CUDA provides no resource limits. UNIX provides a rich set of resource limits via the `rlimit` mechanism. These are just as important for protecting against programming errors as assuring fair resource distribution. CUDA neither honors the existing infrastructure, nor provides any of its own. Device memory allocations do not count against `RLIMIT_AS`, kernel execution time does not count against `RLIMIT_CPU`, and executing kernels are not bounded by `RLIMIT_NPROC`.

5.10 Can CUDA deny access to system resources?

CUDA allows system RAM to be monopolized. Devices of Compute Capability 1.2 or higher support mapping host memory into a device's address space. For this to be done, the memory must be *pinned* (also known as *locked*), and thus protected against swapping. Since this makes physical memory unusable by the rest of the system, memory pinning is strictly limited using the `RLIMIT_MEMLOCK` limit (the default value on Debian Linux is 64KB). CUDA, allocating system memory from within kernelspace, does not honor this limit. This more than once resulted in Linux's "OOM (Out-of-Memory) killer" delivering SIGKILL to `firefox` or even `sshd` processes during testing.

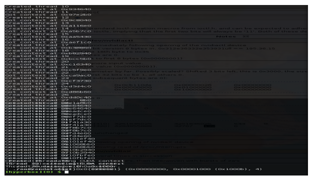
5.11 Any miscellaneous security issues?

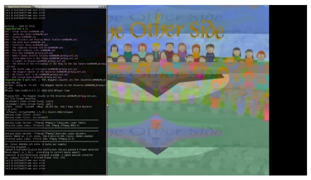
CUDA introduces sundry security issues. The CUDA installer (which must be run as root throughout) contacts an anonymous FTP server to check for updated versions of the driver. This is open to any number of classic man-in-the-middle attacks, leaving the user vulnerable to trojans. Even if the download was signed, running the downloader as root leaves the system vulnerable to user agent exploits.

The `nvcc` compiler does not support a `-pipe` option ala GCC, instead forcing use of temporary files for multi-phase operations. The temporary names used are wholly predictable, suggesting vulnerability to a class of symlink attack. Successfully attacking this scheme via named FIFOs, and thus providing a `-pipe` capability for `nvcc`, is left as an exercise for the reader.

The ability of any CUDA application to monopolize device memory facilitates "use-of-NULL" attacks throughout the graphics stack, much like those which have plagued the Linux kernel in 2009 and 2010. Below, see two corrupted displays. It is doubtful that our CUDA application actively molested the display, since it simply allocated device memory; more likely, the programs ignored some allocation failure.

Behold the grim visage of undefined behavior:


(a) Corrupted AA


(b) Pixellated VDPAU

6 Defending CUDA

Implementors of competing CUDA stacks must ensure that their systems implement the relatively successful memory protections extant in the NVIDIA solution. The following properties ought be considered in designs, and verified:

- CUDA manifests a Harvard architecture atop what is likely a Von Neumann unified memory. It is necessary that code be properly protected, especially under Compute Capability 2.0's concurrent kernels.
- CUDA performs memory translation and protection, almost certainly in hardware. It is necessary that these

mechanisms be properly configured, and that proper invalidations are performed in the face of remappings. Concurrent kernels must not be able to freely read each other's memory.

- CUDA monitors processes from within kernelspace, freeing their device resources on termination. This must be duplicated, and must be performed in kernelspace to be effective.
- It ought be ensured that device control maps are not shared with device code via a secondary mapping. This code must be located in kernelspace.
- It would be valuable for CUDA to make available more forensic information. In particular, the standard UNIX process accounting mechanism ought be honored. Furthermore, rate-limiting ought use the standard, configurable klogd policies.
- CUDA resources ought be accounted for by the `rlimit` infrastructure. It is absolutely critical that this be done for mapped host memory.

7 Conclusions

The recommendations of the Orange Book [13], requirements of the Common Criteria [9], and animadversions of Saltzer and Schroeder [22] have long provided general principles for security-conscious design. These principles have been at least superficially observed along the way to NVIDIA's CUDA system. Reverse engineering of the CUDA software stack and validation of open-source competition will reveal how truly sound the implementations may be. Modern GPUs are some of the world's most powerful devices. Harnessing these mighty processing units is necessary to even approach full machine utilization. This will become only more true:

- Increasing the clock frequency affects cooling and power requirements (dynamic power consumption is proportional to frequency). Furthermore, clock frequency increases tend to force decomposition of pipeline stages, exacerbating branch misprediction delays and pressuring any OOO system [24].
- Increasing the issue width, or duplicating functional units, either requires ISA changes (for VLIW) or massive frontend resources for lookahead, disambiguation, and hazard tracking. Assuming these expenses acceptable, diminishing returns result from limitations of instruction-level parallelism. Furthermore, very wide issues lead to sparse code flows, taxing the instruction store subsystem.
- Further investments in cache — and thus, hopefully, fewer memory delays — serve only to approach more closely a device's theoretical peak. That peak itself is a function of architecture, not of programs or their access patterns.
- Investments in OOO apparatus (reorder buffers, frontend reservation stations, register renaming) yield diminishing returns due to the profound limitations of instruction level parallelism [6]. Like improvements to cache, OOO can only hide delays, not find new FLOPS.
- Denser chip-multithreading similarly serves only to hide latency.
- Larger chips require either pipelined wires or high voltages to operate reliably, and place strict requirements on clock-signaling circuity. Efficient power management requires complex and expensive partial power gating[5].

We see that FLOPS — at any price — can be had only by adding cores or extending SIMD order. The former is obviously easier with small, in-order, SISD cores, such as the unified shaders of a modern GPU; duplicating a Nehalem core is a much more daunting process. As for the latter, note that Intel's "Sandy Bridge" microarchitecture is expected to add the AVX instruction scheme and its 256-bit `YMM` vector registers [8] [1]. The data-based decomposition of CUDA's "SIMT" model can immediately take advantage of their new cores (given sufficiently large problem sets, of course), whereas x86 binaries[6] would require dynamic translation or recompilation to take advantage of the new SIMD order.

Extensive manycoring and huge memory bandwidths are certainly the key to FLOPS. Without security-conscious analysis and verification of infrastructure and policy, they must not be trusted in a multiprocessing context. As GPUs become more and more general-purpose, the attack space will grow; Compute Capability 2.0 already adds significant complexity and danger. The only answer is constant vigilance.

References

[1] AMD CORPORATION. *AMD64 Architecture Programmer's Manual Volume 6: 128-Bit and 256-Bit XOP and FMA4 Instructions 3.04*. No. 43479. November 2009.

[5]For instance, feeding μops from Nehalem's Loop Stream Detector results in power-down of the leading three pipeline stages.

[6]Most of them, anyway.

[2] BLACK, N. libcudest. `http://dank.qemfd.net/dankwiki/index.php/Libcudest`, 2010.

[3] BOSWORTH, S., AND KABAY, M. E. *The Computer Security Handbook*. John Wiley & Sons, Inc., New York, NY, USA, 2002, ch. 5.

[4] DEMMEL, J. W., AND VOLKOV, V. Benchmarking GPUs to tune dense linear algebra. *ACM/IEEE conference on Supercomputing* (2008).

[5] FABRY, R. S. Capability-based addressing. *Communications of the ACM* (1974), 403–412.

[6] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, May 2002.

[7] INNOVATIVE SYSTEMS LAB, NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS. cuda_wrapper. `http://cudawrapper.sourceforge.net/`, 2009.

[8] INTEL CORPORATION. *Intel® Advanced Vector Extensions Programming Reference*. No. 319433-006. July 2009.

[9] ISO 15408-1:2009. Common criteria 3.1: Information technology — security techniques — evaluation criteria for it security.

[10] KERR, A., DIAMOS, G., AND YALAMANCHILI, S. A characterization and analysis of PTX kernels. *IEEE Workload Characterization Symposium 0* (2009), 3–12.

[11] KOŚCIELNICKI, M. W. nv50dis. `http://0x04.net/cgit/index.cgi/nv50dis`, 2009–.

[12] KRANENBURG, P., AND MCGRATH, R. strace. `http://sourceforge.net/projects/strace/`, 1990–.

[13] NATIONAL COMPUTER SECURITY CENTER. *Trusted Computer System Evaluation Criteria (Orange Book)*. No. DOD 5200.28-STD. 1985.

[14] NOUVEAU PROJECT. Nouveau project wiki: Cuda. `http://nouveau.freedesktop.org/wiki/CUDA`, 2009–.

[15] NOUVEAU PROJECT. vbtracetool. `http://nouveau.freedesktop.org/wiki/DumpingVideoBios`, 2009–.

[16] NOUVEAU PROJECT. nvtrace. `http://nouveau.freedesktop.org/wiki/Nvtrace`, 2009.

[17] NVIDIA CORPORATION. *The CUDA Compiler Driver NVCC*. 2007–2010.

[18] NVIDIA CORPORATION. *CUDA-GDB (NVIDIA CUDA Debugger) User Manual 3.0*. No. PG-00000-004. January 2010.

[19] NVIDIA CORPORATION. *NVIDIA CUDA™ Programming Guide 3.0*. 2010.

[20] NVIDIA CORPORATION. *NVIDIA PTX: Parallel Thread Execution ISA Version 2.0*. 2010.

[21] PAALANEN, P., MUIZELAAR, J., INTEL CORPORATION, AND NOUVEAU PROJECT. In-kernel memory-mapped i/o tracing. `http://nouveau.freedesktop.org/wiki/MmioTrace`, 2003–.

[22] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. In *Fourth ACM Symposium on Operating System Principles* (1973).

[23] SHOSTACK, A. Experiences threat modeling at Microsoft. *Workshop on Modeling Security* (September 2008).

[24] SPRANGLE, E., AND CARMEAN, D. Increasing processor performance by implementing deeper pipelines. *Proceedings of the 29th annual international symposium on Computer architecture (ISCA '02) 30* (2002).

[25] WONG, H., PAPADOPOULOU, M.-M., SADOOGHI-ALVANDI, M., AND MOSHOVOS, A. Demystifying GPU microarchitecture through microbenchmarking. *IEEE International Symposium on Performance Analysis of Systems and Software* (March 2010).

A strace(2)d CUDA binary

```
execve("C/bin/linux/release/deviceQueryDrv", ["C/bin/linux/release/deviceQueryD"...], [/* 45 vars */]) = 0
brk(0)                                  = 0x1b29000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc8571000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc856f000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=86224, ...}) = 0
mmap(NULL, 86224, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fdcc8559000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/usr/lib/libcuda.so.1", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\375\7\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=7404990, ...}) = 0
mmap(NULL, 8623832, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fdcc7d1f000
mprotect(0x7fdcc8399000, 1044480, PROT_NONE) = 0
mmap(0x7fdcc8498000, 618496, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x679000) = 0x7fdcc8498000
mmap(0x7fdcc852f000, 169688, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fdcc852f000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/usr/lib/libstdc++.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\243\245\266:\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=1046720, ...}) = 0
mmap(0x3ab6a00000, 3223576, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3ab6a00000
mprotect(0x3ab6af6000, 2097152, PROT_NONE) = 0
mmap(0x3ab6cf6000, 36864, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xf6000) = 0x3ab6cf6000
mmap(0x3ab6cff000, 81944, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x3ab6cff000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libm.so.6", O_RDONLY)        = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p> \253:\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=533472, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc7d1e000
mmap(0x3aab200000, 2625752, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3aab200000
mprotect(0x3aab281000, 2093056, PROT_NONE) = 0
mmap(0x3aab480000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x80000) = 0x3aab480000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libgcc_s.so.1", O_RDONLY)    = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P- \266:\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=93072, ...}) = 0
mmap(0x3ab6200000, 2186360, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3ab6200000
mprotect(0x3ab6216000, 2093056, PROT_NONE) = 0
mmap(0x3ab6415000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x15000) = 0x3ab6415000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\353\241\252:\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1385152, ...}) = 0
mmap(0x3aaaa00000, 3487784, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3aaaa00000
mprotect(0x3aaab4a000, 2097152, PROT_NONE) = 0
mmap(0x3aaad4a000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x14a000) = 0x3aaad4a000
mmap(0x3aaad4f000, 18472, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x3aaad4f000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libpthread.so.0", O_RDONLY)  = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320X‘\253:\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=134033, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc7d1d000
mmap(0x3aab600000, 2208640, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3aab600000
mprotect(0x3aab616000, 2097152, PROT_NONE) = 0
mmap(0x3aab816000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16000) = 0x3aab816000
mmap(0x3aab818000, 13184, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x3aab818000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/usr/lib/libz.so.1", O_RDONLY)    = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\"\240\253:\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=96448, ...}) = 0
mmap(0x3aaba00000, 2188976, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3aaba00000
mprotect(0x3aaba17000, 2093056, PROT_NONE) = 0
mmap(0x3aabc16000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16000) = 0x3aabc16000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libdl.so.2", O_RDONLY)       = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\340\r\340\252:\0\0\0"..., 832) = 832
```

```
fstat(3, {st_mode=S_IFREG|0644, st_size=17504, ...}) = 0
mmap(0x3aaae00000, 2109696, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3aaae00000
mprotect(0x3aaae02000, 2097152, PROT_NONE) = 0
mmap(0x3aab002000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x3aab002000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc7d1c000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc7d1b000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc7d1a000
arch_prctl(ARCH_SET_FS, 0x7fdcc7d1a710) = 0
mprotect(0x3aab002000, 4096, PROT_READ) = 0
mprotect(0x3aab816000, 4096, PROT_READ) = 0
mprotect(0x3aaad4a000, 16384, PROT_READ) = 0
mprotect(0x3aab480000, 4096, PROT_READ) = 0
mprotect(0x3ab6cf6000, 28672, PROT_READ) = 0
mprotect(0x3aa961c000, 4096, PROT_READ) = 0
munmap(0x7fdcc8559000, 86224)           = 0
set_tid_address(0x7fdcc7d1a7e0)         = 4613
set_robust_list(0x7fdcc7d1a7f0, 0x18)   = 0
futex(0x7fff6bcd716c, FUTEX_WAKE_PRIVATE, 1) = 0
futex(0x7fff6bcd716c, FUTEX_WAIT_BITSET_PRIVATE|FUTEX_CLOCK_REALTIME, 1, NULL, 7fdcc7d1a710) = -1 EAGAIN
rt_sigaction(SIGRTMIN, {0x3aab605750, [], SA_RESTORER|SA_SIGINFO, 0x3aab60e990}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {0x3aab6057e0, [], SA_RESTORER|SA_RESTART|SA_SIGINFO, 0x3aab60e990}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
getrlimit(RLIMIT_STACK, {rlim_cur=8192*1024, rlim_max=RLIM_INFINITY}) = 0
sched_get_priority_max(SCHED_RR)        = 99
sched_get_priority_min(SCHED_RR)        = 1
futex(0x3ab6cffb68, FUTEX_WAKE_PRIVATE, 2147483647) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc856e000
write(1, "CUDA Device Query (Driver API) s"..., 58) = 58
open("/proc/stat", O_RDONLY|O_CLOEXEC)  = 3
read(3, "cpu  9758 0 3719 540454 5815 470"..., 8192) = 2079
close(3)                                = 0
brk(0)                                  = 0x1b29000
brk(0x1b4a000)                          = 0x1b4a000
geteuid()                               = 1000
geteuid()                               = 1000
open("/dev/nvidiactl", O_RDWR)          = 3
ioctl(3, 0xc04846d2, 0x7fff6bcd6160)    = 0
ioctl(3, 0xc00446ca, 0x7fdcc85538e0)    = 0
ioctl(3, 0xc60046c8, 0x7fdcc85532e0)    = 0
ioctl(3, 0xc00c4622, 0x7fff6bcd61b0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6190)    = 0
geteuid()                               = 1000
open("/dev/nvidia0", O_RDWR)            = 4
ioctl(3, 0xc048464d, 0x7fff6bcd5f00)    = 0
open("/proc/interrupts", O_RDONLY)      = 5
fstat(5, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc856d000
read(5, "           CPU0       CPU1      "..., 1024) = 1024
read(5, " 97          0   IO-APIC-fasteoi"..., 1024) = 1024
read(5, " 0          0          0        "..., 1024) = 1024
read(5, "7      13730      23433   Local "..., 1024) = 1024
read(5, "    0          0          0     "..., 1024) = 246
read(5, "", 1024)                       = 0
read(5, "", 1024)                       = 0
close(5)                                = 0
munmap(0x7fdcc856d000, 4096)            = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6110)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5f40)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5f40)    = 0
geteuid()                               = 1000
open("/dev/nvidia0", O_RDWR)            = 5
ioctl(3, 0xc048464d, 0x7fff6bcd5df0)    = 0
open("/proc/interrupts", O_RDONLY)      = 6
fstat(6, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc856d000
read(6, "           CPU0       CPU1      "..., 1024) = 1024
read(6, " 97          0   IO-APIC-fasteoi"..., 1024) = 1024
read(6, " 0          0          0        "..., 1024) = 1024
read(6, "7      13730      23433   Local "..., 1024) = 1024
read(6, "    0          0          0     "..., 1024) = 246
read(6, "", 1024)                       = 0
read(6, "", 1024)                       = 0
close(6)                                = 0
munmap(0x7fdcc856d000, 4096)            = 0
ioctl(3, 0xc020462b, 0x7fff6bcd60e0)    = 0
ioctl(5, 0xc0204637, 0x7fff6bcd6140)    = 0
ioctl(5, 0xc0204637, 0x7fff6bcd6140)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6110)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5fb0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5eb0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5eb0)    = 0
geteuid()                               = 1000
open("/dev/nvidia0", O_RDWR)            = 6
ioctl(3, 0xc048464d, 0x7fff6bcd5d60)    = 0
open("/proc/interrupts", O_RDONLY)      = 7
fstat(7, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc856d000
read(7, "           CPU0       CPU1      "..., 1024) = 1024
read(7, " 97          0   IO-APIC-fasteoi"..., 1024) = 1024
read(7, " 0          0          0        "..., 1024) = 1024
read(7, "8      13731      23434   Local "..., 1024) = 1024
read(7, "    0          0          0     "..., 1024) = 246
read(7, "", 1024)                        = 0
read(7, "", 1024)                        = 0
close(7)                                = 0
munmap(0x7fdcc856d000, 4096)            = 0
ioctl(3, 0xc014462d, 0x7fff6bcd6040)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6110)    = 0
ioctl(5, 0xc0144632, 0x7fff6bcd5de0)    = 0
ioctl(5, 0xc0144632, 0x7fff6bcd5de0)    = 0
ioctl(5, 0xc0144632, 0x7fff6bcd5de0)    = 0
ioctl(5, 0xc0204637, 0x7fff6bcd5de0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(5, 0xc0204637, 0x7fff6bcd5de0)    = 0
ioctl(5, 0xc0204637, 0x7fff6bcd5de0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(5, 0xc0204637, 0x7fff6bcd5de0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5db0)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd5f50)    = 0
ioctl(3, 0xc020462b, 0x7fff6bcd6020)    = 0
ioctl(3, 0xc030464e, 0x7fff6bcd6010)    = 0
mmap(NULL, 4096, PROT_READ, MAP_SHARED, 6, 0xf2009000) = 0x7fdcc856d000
ioctl(3, 0xc020462a, 0x7fff6bcd6180)    = 0
write(1, "There is 1 device supporting CUD"..., 34) = 34
ioctl(3, 0xc020462a, 0x7fff6bcd6dc0)    = 0
write(1, "\n", 1)                       = 1
write(1, "Device 0: \"GeForce GTS 360M\"\n", 29) = 29
write(1, "  CUDA Driver Version:          "..., 53) = 53
write(1, "  CUDA Capability Major revision"..., 51) = 51
write(1, "  CUDA Capability Minor revision"..., 51) = 51
ioctl(3, 0xc098464a, 0x7fff6bcd6e10)    = 0
write(1, "  Total amount of global memory:"..., 66) = 66
write(1, "  Number of multiprocessors:    "..., 52) = 52
write(1, "  Number of cores:              "..., 52) = 52
write(1, "  Total amount of constant memor"..., 61) = 61
write(1, "  Total amount of shared memory "..., 61) = 61
write(1, "  Total number of registers avai"..., 55) = 55
write(1, "  Warp size:                    "..., 52) = 52
write(1, "  Maximum number of threads per "..., 53) = 53
write(1, "  Maximum sizes of each dimensio"..., 64) = 64
write(1, "  Maximum sizes of each dimensio"..., 67) = 67
write(1, "  Maximum memory pitch:         "..., 66) = 66
write(1, "  Texture alignment:            "..., 59) = 59
ioctl(3, 0xc020462a, 0x7fff6bcd6c90)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6c90)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6c90)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6d30)    = 0
ioctl(3, 0xc020462a, 0x7fff6bcd6d30)    = 0
write(1, "  Clock rate:                   "..., 58) = 58
write(1, "  Concurrent copy and execution:"..., 53) = 53
ioctl(3, 0xc020462a, 0x7fff6bcd6e20)    = 0
write(1, "  Run time limit on kernels:    "..., 53) = 53
write(1, "  Integrated:                   "..., 52) = 52
write(1, "  Support host page-locked memor"..., 53) = 53
ioctl(3, 0xc020462a, 0x7fff6bcd6e20)    = 0
write(1, "  Compute mode:                 "..., 116) = 116
write(1, "\n", 1)                       = 1
write(1, "PASSED\n", 7)                  = 7
write(1, "\n", 1)                        = 1
write(1, "Press ENTER to exit...\n", 23) = 23
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdcc856c000
read(0, "\n", 1024)                      = 1
exit_group(0)                            = ?
```

B `libcuda.so.195.36.15` 3.0 symbols

```
#generated via nm -S -D -g -a -C --defined-only /usr/lib/libcuda.so
0000000000102120 0000000000000119 T clGetExtensionFunctionAddress
00000000000f03b0 0000000000000005 T clGetPlatformInfo
0000000000111150 0000000000000272 T cuArray3DCreate
0000000000110ed0 0000000000000272 T cuArray3DGetDescriptor
00000000001118a0 0000000000000272 T cuArrayCreate
00000000001113d0 000000000000024d T cuArrayDestroy
0000000000111620 0000000000000272 T cuArrayGetDescriptor
0000000000119c00 0000000000000272 T cuCtxAttach
000000000011a0d0 000000000000028e T cuCtxCreate
0000000000119e80 000000000000024d T cuCtxDestroy
00000000001199b0 000000000000024d T cuCtxDetach
00000000001192c0 000000000000024d T cuCtxGetDevice
0000000000119510 000000000000024d T cuCtxPopCurrent
0000000000119760 000000000000024d T cuCtxPushCurrent
000000000010a6a0 000000000000018c T cuCtxSynchronize
000000000011aaf0 000000000000028e T cuDeviceComputeCapability
000000000011b260 0000000000000272 T cuDeviceGet
000000000011a360 000000000000028e T cuDeviceGetAttribute
000000000011b010 000000000000024d T cuDeviceGetCount
000000000011ad80 000000000000028e T cuDeviceGetName
000000000011a5f0 0000000000000272 T cuDeviceGetProperties
000000000011a870 0000000000000272 T cuDeviceTotalMem
000000000011b4e0 000000000000024d T cuDriverGetVersion
000000000010d1b0 0000000000000272 T cuEventCreate
000000000010c850 000000000000024d T cuEventDestroy
000000000010c5c0 000000000000028e T cuEventElapsedTime
000000000010ccf0 000000000000024d T cuEventQuery
000000000010cf40 0000000000000261 T cuEventRecord
000000000010caa0 000000000000024d T cuEventSynchronize
0000000000111da0 000000000000028e T cuFuncGetAttribute
00000000001122b0 00000000000002b2 T cuFuncSetBlockShape
0000000000111b20 0000000000000272 T cuFuncSetCacheConfig
0000000000112030 0000000000000272 T cuFuncSetSharedSize
000000000011b8c0 000000000000028e T cuGLCtxCreate
000000000011b730 000000000000018c T cuGLInit
000000000011c7a0 000000000000028e T cuGLMapBufferObject
000000000011bdc0 00000000000002aa T cuGLMapBufferObjectAsync
000000000011ca30 0000000000000253 T cuGLRegisterBufferObject
000000000011c070 0000000000000261 T cuGLSetBufferObjectMapFlags
000000000011c540 0000000000000253 T cuGLUnmapBufferObject
000000000011bb50 0000000000000261 T cuGLUnmapBufferObjectAsync
000000000011c2e0 0000000000000253 T cuGLUnregisterBufferObject
000000000010aa90 0000000000000272 T cuGetExportTable
000000000011cf50 000000000000028e T cuGraphicsGLRegisterBuffer
000000000011cc90 00000000000002b2 T cuGraphicsGLRegisterImage
000000000010afa0 0000000000000286 T cuGraphicsMapResources
000000000010b4b0 000000000000028e T cuGraphicsResourceGetMappedPointer
000000000010b230 0000000000000272 T cuGraphicsResourceSetMapFlags
000000000010b740 00000000000002b2 T cuGraphicsSubResourceGetMappedArray
000000000010ad10 0000000000000286 T cuGraphicsUnmapResources
000000000010ba00 000000000000024d T cuGraphicsUnregisterResource
000000000010a830 0000000000000253 T cuInit
000000000010d970 000000000000024d T cuLaunch
000000000010d6e0 000000000000028e T cuLaunchGrid
000000000010d430 00000000000002aa T cuLaunchGridAsync
0000000000117930 0000000000000272 T cuMemAlloc
0000000000116ee0 0000000000000272 T cuMemAllocHost
0000000000117650 00000000000002d4 T cuMemAllocPitch
00000000001173f0 0000000000000253 T cuMemFree
0000000000116c90 000000000000024d T cuMemFreeHost
0000000000117160 000000000000028e T cuMemGetAddressRange
0000000000081ef0 000000000000005f T cuMemGetAttribute
0000000000117bb0 0000000000000272 T cuMemGetInfo
0000000000116a00 000000000000028e T cuMemHostAlloc
0000000000116770 000000000000028e T cuMemHostGetDevicePointer
00000000001164f0 0000000000000272 T cuMemHostGetFlags
0000000000114d10 000000000000024d T cuMemcpy2D
0000000000113850 0000000000000261 T cuMemcpy2DAsync
0000000000114ac0 000000000000024d T cuMemcpy2DUnaligned
0000000000114870 000000000000024d T cuMemcpy3D
00000000001135e0 0000000000000261 T cuMemcpy3DAsync
0000000000114f60 00000000000002d4 T cuMemcpyAtoA
00000000001157c0 00000000000002b1 T cuMemcpyAtoD
0000000000115240 00000000000002b2 T cuMemcpyAtoH
```

```
0000000000113ac0 00000000000002cc T cuMemcpyAtoHAsync
0000000000115a80 00000000000002b2 T cuMemcpyDtoA
0000000000115d40 000000000000028d T cuMemcpyDtoD
0000000000114060 00000000000002aa T cuMemcpyDtoDAsync
0000000000115fd0 000000000000028e T cuMemcpyDtoH
0000000000114310 00000000000002aa T cuMemcpyDtoHAsync
0000000000115500 00000000000002b9 T cuMemcpyHtoA
0000000000113d90 00000000000002cc T cuMemcpyHtoAAsync
0000000000116260 000000000000028d T cuMemcpyHtoD
00000000001145c0 00000000000002aa T cuMemcpyHtoDAsync
00000000001130a0 000000000000029d T cuMemsetD16
0000000000112850 00000000000002d5 T cuMemsetD2D16
0000000000112570 00000000000002d3 T cuMemsetD2D32
0000000000112b30 00000000000002d5 T cuMemsetD2D8
0000000000112e10 000000000000028d T cuMemsetD32
0000000000113340 000000000000029c T cuMemsetD8
0000000000118380 000000000000028e T cuModuleGetFunction
00000000001180c0 00000000000002b9 T cuModuleGetGlobal
0000000000117e30 000000000000028e T cuModuleGetTexRef
0000000000119040 0000000000000272 T cuModuleLoad
0000000000118dc0 0000000000000272 T cuModuleLoadData
0000000000118ae0 00000000000002d4 T cuModuleLoadDataEx
0000000000118860 0000000000000272 T cuModuleLoadFatBinary
0000000000118610 000000000000024d T cuModuleUnload
000000000010e660 0000000000000272 T cuParamSetSize
000000000010dbc0 000000000000028e T cuParamSetTexRef
000000000010e110 00000000000002b5 T cuParamSetf
000000000010e3d0 000000000000028e T cuParamSeti
000000000010de50 00000000000002b9 T cuParamSetv
000000000010c340 0000000000000272 T cuStreamCreate
000000000010bc50 0000000000000245 T cuStreamDestroy
000000000010c0f0 0000000000000245 T cuStreamQuery
000000000010bea0 0000000000000245 T cuStreamSynchronize
0000000000110c80 000000000000024d T cuTexRefCreate
0000000000110a30 000000000000024d T cuTexRefDestroy
000000000010f580 0000000000000272 T cuTexRefGetAddress
000000000010f070 000000000000028e T cuTexRefGetAddressMode
000000000010f300 0000000000000272 T cuTexRefGetArray
000000000010edf0 0000000000000272 T cuTexRefGetFilterMode
000000000010e8e0 0000000000000272 T cuTexRefGetFlags
000000000010eb60 000000000000028e T cuTexRefGetFormat
00000000001104e0 00000000000002b2 T cuTexRefSetAddress
0000000000110220 00000000000002b2 T cuTexRefSetAddress2D
000000000010fd00 000000000000028e T cuTexRefSetAddressMode
00000000001107a0 000000000000028e T cuTexRefSetArray
000000000010fa80 0000000000000272 T cuTexRefSetFilterMode
000000000010f800 0000000000000272 T cuTexRefSetFlags
000000000010ff90 000000000000028e T cuTexRefSetFormat
00000000000d0b30 000000000000003d T cudbgGetAPI
00000000000d0b70 000000000000002a T cudbgGetAPIVersion
00000000000c9b10 000000000000000a T gpudbgDebuggerAttached
```

C `dmesg` output from hung task

```
NVRM: loading NVIDIA UNIX x86_64 Kernel Module  195.36.15  Fri Mar 12 00:29:13 PST 2010
NVRM: Xid (0007:00): 13, 0001 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
NVRM: Xid (0007:00): 13, 0003 00000000 000050c0 00000368 00000000 00000100
INFO: task cudadump:7314 blocked for more than 120 seconds.
"echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
cudadump      D ffff880143d843b0     0  7314   7000 0x00000004
 ffff880146f6f9b8 0000000000000046 ffffffff812ddb8e ffff880146f6fa80
 000000000005e450 0000000000000000 ffff880146f6ffd8 000000000000dea0
 0000000000012940 0000000000004000 ffff880143d84740 ffff8800e2009000
Call Trace:
 [<ffffffff812ddb8e>] ? common_interrupt+0xe/0x13
 [<ffffffff8101d749>] ? __change_page_attr_set_clr+0xed/0x983
 [<ffffffff812dc0a0>] schedule_timeout+0x35/0x1ea
 [<ffffffff81078ae6>] ? __pagevec_free+0x29/0x3c
 [<ffffffff81077839>] ? free_pcppages_bulk+0x46/0x244
 [<ffffffff812dbf26>] wait_for_common+0xc4/0x13a
 [<ffffffff8102f121>] ? default_wake_function+0x0/0xf
 [<ffffffff812dc026>] wait_for_completion+0x18/0x1a
 [<ffffffffa12048bd>] os_acquire_sema+0x3f/0x66 [nvidia]
 [<ffffffffa110c8dc>] _nv006655rm+0x6/0x1f [nvidia]
 [<ffffffffa1114e9d>] ? rm_free_unused_clients+0x5a/0xb7 [nvidia]
 [<ffffffffa1201967>] ? nv_kern_ctl_close+0x93/0xcb [nvidia]
 [<ffffffffa12025b8>] ? nv_kern_close+0xa1/0x373 [nvidia]
 [<ffffffff810a6af3>] ? __fput+0x112/0x1d1
 [<ffffffff810a6bc7>] ? fput+0x15/0x17
 [<ffffffff810a3f71>] ? filp_close+0x58/0x62
 [<ffffffff8103550a>] ? put_files_struct+0x65/0xb4
 [<ffffffff81035594>] ? exit_files+0x3b/0x40
 [<ffffffff81036d1c>] ? do_exit+0x1dc/0x661
 [<ffffffff81037211>] ? do_group_exit+0x70/0x99
 [<ffffffff81040435>] ? get_signal_to_deliver+0x2de/0x2f9
 [<ffffffff81001527>] ? do_signal+0x6d/0x681
 [<ffffffff81046ce0>] ? remove_wait_queue+0x4c/0x51
 [<ffffffff810368bc>] ? do_wait+0x1b2/0x1f5
 [<ffffffff810369a7>] ? sys_wait4+0xa8/0xbc
 [<ffffffff81001b62>] ? do_notify_resume+0x27/0x51
 [<ffffffff81035195>] ? child_wait_callback+0x0/0x53
 [<ffffffff810021cb>] ? int_signal+0x12/0x17
INFO: task cudadump:7314 blocked for more than 120 seconds.
"echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
cudadump      D ffff880143d843b0     0  7314   7000 0x00000004
 ffff880146f6f9b8 0000000000000046 ffffffff812ddb8e ffff880146f6fa80
 000000000005e450 0000000000000000 ffff880146f6ffd8 000000000000dea0
 0000000000012940 0000000000004000 ffff880143d84740 ffff8800e2009000
Call Trace:
 [<ffffffff812ddb8e>] ? common_interrupt+0xe/0x13
 [<ffffffff8101d749>] ? __change_page_attr_set_clr+0xed/0x983
 [<ffffffff812dc0a0>] schedule_timeout+0x35/0x1ea
 [<ffffffff81078ae6>] ? __pagevec_free+0x29/0x3c
 [<ffffffff81077839>] ? free_pcppages_bulk+0x46/0x244
 [<ffffffff812dbf26>] wait_for_common+0xc4/0x13a
 [<ffffffff8102f121>] ? default_wake_function+0x0/0xf
 [<ffffffff812dc026>] wait_for_completion+0x18/0x1a
 [<ffffffffa12048bd>] os_acquire_sema+0x3f/0x66 [nvidia]
 [<ffffffffa110c8dc>] _nv006655rm+0x6/0x1f [nvidia]
 [<ffffffffa1114e9d>] ? rm_free_unused_clients+0x5a/0xb7 [nvidia]
 [<ffffffffa1201967>] ? nv_kern_ctl_close+0x93/0xcb [nvidia]
 [<ffffffffa12025b8>] ? nv_kern_close+0xa1/0x373 [nvidia]
 [<ffffffff810a6af3>] ? __fput+0x112/0x1d1
 [<ffffffff810a6bc7>] ? fput+0x15/0x17
 [<ffffffff810a3f71>] ? filp_close+0x58/0x62
 [<ffffffff8103550a>] ? put_files_struct+0x65/0xb4
 [<ffffffff81035594>] ? exit_files+0x3b/0x40
 [<ffffffff81036d1c>] ? do_exit+0x1dc/0x661
 [<ffffffff81037211>] ? do_group_exit+0x70/0x99
 [<ffffffff81040435>] ? get_signal_to_deliver+0x2de/0x2f9
 [<ffffffff81001527>] ? do_signal+0x6d/0x681
 [<ffffffff81046ce0>] ? remove_wait_queue+0x4c/0x51
 [<ffffffff810368bc>] ? do_wait+0x1b2/0x1f5
 [<ffffffff810369a7>] ? sys_wait4+0xa8/0xbc
 [<ffffffff81001b62>] ? do_notify_resume+0x27/0x51
 [<ffffffff81035195>] ? child_wait_callback+0x0/0x53
 [<ffffffff810021cb>] ? int_signal+0x12/0x17
INFO: task cudadump:7314 blocked for more than 120 seconds.
"echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
cudadump      D ffff880143d843b0     0  7314   7000 0x00000004
 ffff880146f6f9b8 0000000000000046 ffffffff812ddb8e ffff880146f6fa80
 000000000005e450 0000000000000000 ffff880146f6ffd8 000000000000dea0
 0000000000012940 0000000000004000 ffff880143d84740 ffff8800e2009000
Call Trace:
 [<ffffffff812ddb8e>] ? common_interrupt+0xe/0x13
 [<ffffffff8101d749>] ? __change_page_attr_set_clr+0xed/0x983
 [<ffffffff812dc0a0>] schedule_timeout+0x35/0x1ea
 [<ffffffff81078ae6>] ? __pagevec_free+0x29/0x3c
 [<ffffffff81077839>] ? free_pcppages_bulk+0x46/0x244
 [<ffffffff812dbf26>] wait_for_common+0xc4/0x13a
 [<ffffffff8102f121>] ? default_wake_function+0x0/0xf
 [<ffffffff812dc026>] wait_for_completion+0x18/0x1a
 [<ffffffffa12048bd>] os_acquire_sema+0x3f/0x66 [nvidia]
 [<ffffffffa110c8dc>] _nv006655rm+0x6/0x1f [nvidia]
 [<ffffffffa1114e9d>] ? rm_free_unused_clients+0x5a/0xb7 [nvidia]
 [<ffffffffa1201967>] ? nv_kern_ctl_close+0x93/0xcb [nvidia]
 [<ffffffffa12025b8>] ? nv_kern_close+0xa1/0x373 [nvidia]
 [<ffffffff810a6af3>] ? __fput+0x112/0x1d1
 [<ffffffff810a6bc7>] ? fput+0x15/0x17
 [<ffffffff810a3f71>] ? filp_close+0x58/0x62
 [<ffffffff8103550a>] ? put_files_struct+0x65/0xb4
 [<ffffffff81035594>] ? exit_files+0x3b/0x40
 [<ffffffff81036d1c>] ? do_exit+0x1dc/0x661
 [<ffffffff81037211>] ? do_group_exit+0x70/0x99
 [<ffffffff81040435>] ? get_signal_to_deliver+0x2de/0x2f9
 [<ffffffff81001527>] ? do_signal+0x6d/0x681
 [<ffffffff81046ce0>] ? remove_wait_queue+0x4c/0x51
 [<ffffffff810368bc>] ? do_wait+0x1b2/0x1f5
 [<ffffffff810369a7>] ? sys_wait4+0xa8/0xbc
 [<ffffffff81001b62>] ? do_notify_resume+0x27/0x51
 [<ffffffff81035195>] ? child_wait_callback+0x0/0x53
 [<ffffffff810021cb>] ? int_signal+0x12/0x17
INFO: task cudaranger:7349 blocked for more than 120 seconds.
"echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
cudaranger    D ffff88002820dea0     0  7349   7314 0x00000004
 ffff8800663199e8 0000000000000046 0000000000000000 0000000000000000
 ffff8800663199f8 ffffffff8102694c ffff880066319fd8 000000000000dea0
 0000000000012940 0000000000004000 ffff880143caddd0 000000010037459f
Call Trace:
 [<ffffffff8102694c>] ? select_task_rq_fair+0x4eb/0x8a2
 [<ffffffff812dc0a0>] schedule_timeout+0x35/0x1ea
 [<ffffffff8102f10f>] ? try_to_wake_up+0x328/0x33a
 [<ffffffff812dbf26>] wait_for_common+0xc4/0x13a
 [<ffffffff8102f121>] ? default_wake_function+0x0/0xf
 [<ffffffff812dc026>] wait_for_completion+0x18/0x1a
 [<ffffffffa12048bd>] os_acquire_sema+0x3f/0x66 [nvidia]
 [<ffffffffa110c8dc>] _nv006655rm+0x6/0x1f [nvidia]
 [<ffffffffa1114e9d>] ? rm_free_unused_clients+0x5a/0xb7 [nvidia]
 [<ffffffffa12025e2>] ? nv_kern_close+0xcb/0x373 [nvidia]
 [<ffffffff810a6af3>] ? __fput+0x112/0x1d1
 [<ffffffff810a6bc7>] ? fput+0x15/0x17
 [<ffffffff810a3f71>] ? filp_close+0x58/0x62
 [<ffffffff8103550a>] ? put_files_struct+0x65/0xb4
 [<ffffffff81035594>] ? exit_files+0x3b/0x40
 [<ffffffff81036d1c>] ? do_exit+0x1dc/0x661
 [<ffffffffa1204865>] ? os_release_sema+0x47/0x60 [nvidia]
 [<ffffffff81037211>] ? do_group_exit+0x70/0x99
 [<ffffffff81040435>] ? get_signal_to_deliver+0x2de/0x2f9
 [<ffffffff81001527>] ? do_signal+0x6d/0x681
 [<ffffffff812dbc43>] ? schedule+0x9fd/0xaf0
 [<ffffffff810b1fd1>] ? do_vfs_ioctl+0x480/0x4c6
 [<ffffffff81001b62>] ? do_notify_resume+0x27/0x51
 [<ffffffff810b2059>] ? sys_ioctl+0x42/0x65
 [<ffffffff812ddc5a>] ? retint_signal+0x3d/0x83
[recombinator](0) $
```

D dmesg output from wayward OOM killer

```
[41561.362741] NVRM: Xid (0001:00): 13, 0006 00000000 000050c0 00000368 00000000 00000100
[41592.776280] NVRM: Xid (0001:00): 13, 0006 00000000 000050c0 00000368 00000000 00000100
[42842.683531] firefox-bin invoked oom-killer: gfp_mask=0x201da, order=0, oom_adj=0
[42842.683536] firefox-bin cpuset=/ mems_allowed=0
[42842.683538] Pid: 16771, comm: firefox-bin Tainted: P        W  2.6.34-rc2 #1
[42842.683540] Call Trace:
[42842.683547]  [<ffffffff81098a11>] ? T.492+0x5f/0x16f
[42842.683550]  [<ffffffff8109be60>] ? get_page_from_freelist+0x6ea/0x72d
[42842.683553]  [<ffffffff8109895d>] ? badness+0x1d2/0x227
[42842.683555]  [<ffffffff81098b58>] ? T.491+0x37/0xfe
[42842.683558]  [<ffffffff81098d5f>] ? __out_of_memory+0x140/0x157
[42842.683561]  [<ffffffff81098ed1>] ? out_of_memory+0x15b/0x18d
[42842.683563]  [<ffffffff8109c57a>] ? __alloc_pages_nodemask+0x4a0/0x5d5
[42842.683567]  [<ffffffff8109de57>] ? __do_page_cache_readahead+0x93/0x1b3
[42842.683569]  [<ffffffff8109df93>] ? ra_submit+0x1c/0x20
[42842.683572]  [<ffffffff81097807>] ? filemap_fault+0x17e/0x2f3
[42842.683575]  [<ffffffff810adb86>] ? __do_fault+0x52/0x3ba
[42842.683578]  [<ffffffff810ae9bc>] ? handle_mm_fault+0x3ed/0x7aa
[42842.683582]  [<ffffffff81051f21>] ? autoremove_wake_function+0x0/0x2a
[42842.683586]  [<ffffffff81022417>] ? do_page_fault+0x27e/0x29a
[42842.683589]  [<ffffffff812a2945>] ? page_fault+0x25/0x30
[42842.683591] Mem-Info:
[42842.683592] Node 0 DMA per-cpu:
[42842.683594] CPU    0: hi:    0, btch:   1 usd:   0
[42842.683596] CPU    1: hi:    0, btch:   1 usd:   0
[42842.683597] CPU    2: hi:    0, btch:   1 usd:   0
```

```
[42842.683599] CPU    3: hi:    0, btch:   1 usd:   0
[42842.683601] CPU    4: hi:    0, btch:   1 usd:   0
[42842.683602] CPU    5: hi:    0, btch:   1 usd:   0
[42842.683604] CPU    6: hi:    0, btch:   1 usd:   0
[42842.683606] CPU    7: hi:    0, btch:   1 usd:   0
[42842.683607] Node 0 DMA32 per-cpu:
[42842.683609] CPU    0: hi:  186, btch:  31 usd: 172
[42842.683611] CPU    1: hi:  186, btch:  31 usd: 183
[42842.683612] CPU    2: hi:  186, btch:  31 usd: 170
[42842.683614] CPU    3: hi:  186, btch:  31 usd: 133
[42842.683615] CPU    4: hi:  186, btch:  31 usd:  65
[42842.683617] CPU    5: hi:  186, btch:  31 usd:  74
[42842.683619] CPU    6: hi:  186, btch:  31 usd: 164
[42842.683620] CPU    7: hi:  186, btch:  31 usd: 184
[42842.683622] Node 0 Normal per-cpu:
[42842.683623] CPU    0: hi:  186, btch:  31 usd: 169
[42842.683625] CPU    1: hi:  186, btch:  31 usd: 149
[42842.683627] CPU    2: hi:  186, btch:  31 usd: 170
[42842.683628] CPU    3: hi:  186, btch:  31 usd: 146
[42842.683630] CPU    4: hi:  186, btch:  31 usd: 122
[42842.683632] CPU    5: hi:  186, btch:  31 usd:  96
[42842.683633] CPU    6: hi:  186, btch:  31 usd: 167
[42842.683635] CPU    7: hi:  186, btch:  31 usd: 153
[42842.683639] active_anon:64781 inactive_anon:11419 isolated_anon:32
[42842.683640]  active_file:166 inactive_file:472 isolated_file:0
[42842.683640]  unevictable:0 dirty:0 writeback:11229 unstable:0
[42842.683641]  free:11299 slab_reclaimable:3152 slab_unreclaimable:21764
[42842.683642]  mapped:1053702 shmem:493 pagetables:7058 bounce:0
[42842.683644] Node 0 DMA free:15896kB min:20kB low:24kB high:28kB active_anon:0kB inactive_anon:0kB active_file:0kB inactive_fil
[42842.683652] lowmem_reserve[]: 0 3445 7990 7990
[42842.683655] Node 0 DMA32 free:22812kB min:4928kB low:6160kB high:7392kB active_anon:64216kB inactive_anon:12900kB active_file:
[42842.683664] lowmem_reserve[]: 0 0 4545 4545
[42842.683666] Node 0 Normal free:6488kB min:6500kB low:8124kB high:9748kB active_anon:194908kB inactive_anon:32776kB active_file
[42842.683675] lowmem_reserve[]: 0 0 0 0
[42842.683678] Node 0 DMA: 2*4kB 2*8kB 2*16kB 1*32kB 1*64kB 1*128kB 1*256kB 0*512kB 1*1024kB 1*2048kB 3*4096kB = 15896kB
[42842.683685] Node 0 DMA32: 4687*4kB 0*8kB 2*16kB 2*32kB 0*64kB 1*128kB 1*256kB 1*512kB 1*1024kB 1*2048kB 0*4096kB = 22812kB
[42842.683691] Node 0 Normal: 634*4kB 16*8kB 1*16kB 5*32kB 1*64kB 2*128kB 1*256kB 0*512kB 1*1024kB 1*2048kB 0*4096kB = 6488kB
[42842.683698] 12609 total pagecache pages
[42842.683699] 11387 pages in swap cache
[42842.683701] Swap cache stats: add 22707, delete 11320, find 2/2
[42842.683703] Free swap  = 6744792kB
[42842.683704] Total swap = 6835620kB
[42842.707808] 2097151 pages RAM
[42842.707810] 64340 pages reserved
[42842.707812] 1056384 pages shared
[42842.707813] 964448 pages non-shared
[42842.707816] Out of memory: kill process 8555 (cudapinner) score 1057549 or a child
[42842.707820] Killed process 8555 (cudapinner) vsz:4230196kB, anon-rss:744kB, file-rss:15432kB
[42842.718544] NVRM: VM: nv_vm_malloc_pages: failed to allocate a page
[42891.026550] firefox-bin invoked oom-killer: gfp_mask=0x201da, order=0, oom_adj=0
[42891.026559] firefox-bin cpuset=/ mems_allowed=0
[42891.026565] Pid: 16520, comm: firefox-bin Tainted: P        W  2.6.34-rc2 #1
[42891.026569] Call Trace:
[42891.026582]  [<ffffffff81098a11>] ? T.492+0x5f/0x16f
[42891.026588]  [<ffffffff8109be60>] ? get_page_from_freelist+0x6ea/0x72d
[42891.026595]  [<ffffffff8109895d>] ? badness+0x1d2/0x227
[42891.026602]  [<ffffffff81098b58>] ? T.491+0x37/0xfe
[42891.026611]  [<ffffffff81098d5f>] ? __out_of_memory+0x140/0x157
[42891.026614]  [<ffffffff81098ed1>] ? out_of_memory+0x15b/0x18d
[42891.026616]  [<ffffffff8109c57a>] ? __alloc_pages_nodemask+0x4a0/0x5d5
[42891.026619]  [<ffffffff8111d910>] ? ext4_get_block+0x0/0xe1
[42891.026622]  [<ffffffff8109de57>] ? __do_page_cache_readahead+0x93/0x1b3
[42891.026624]  [<ffffffff8109df93>] ? ra_submit+0x1c/0x20
[42891.026626]  [<ffffffff81097807>] ? filemap_fault+0x17e/0x2f3
[42891.026629]  [<ffffffff810adb86>] ? __do_fault+0x52/0x3ba
[42891.026632]  [<ffffffff810ae9bc>] ? handle_mm_fault+0x3ed/0x7aa
[42891.026635]  [<ffffffff810034ce>] ? call_function_interrupt+0xe/0x20
[42891.026638]  [<ffffffff8100948a>] ? read_tsc+0x5/0x16
[42891.026641]  [<ffffffff81022417>] ? do_page_fault+0x27e/0x29a
[42891.026645]  [<ffffffff812a2945>] ? page_fault+0x25/0x30
[42891.026646] Mem-Info:
[42891.026647] Node 0 DMA per-cpu:
[42891.026649] CPU    0: hi:    0, btch:   1 usd:   0
[42891.026650] CPU    1: hi:    0, btch:   1 usd:   0
[42891.026652] CPU    2: hi:    0, btch:   1 usd:   0
[42891.026653] CPU    3: hi:    0, btch:   1 usd:   0
[42891.026655] CPU    4: hi:    0, btch:   1 usd:   0
```

```
[42891.026656] CPU    5: hi:    0, btch:   1 usd:   0
[42891.026658] CPU    6: hi:    0, btch:   1 usd:   0
[42891.026659] CPU    7: hi:    0, btch:   1 usd:   0
[42891.026660] Node 0 DMA32 per-cpu:
[42891.026662] CPU    0: hi:  186, btch:  31 usd: 178
[42891.026663] CPU    1: hi:  186, btch:  31 usd:   0
[42891.026665] CPU    2: hi:  186, btch:  31 usd: 173
[42891.026666] CPU    3: hi:  186, btch:  31 usd:  60
[42891.026667] CPU    4: hi:  186, btch:  31 usd:   0
[42891.026669] CPU    5: hi:  186, btch:  31 usd:   0
[42891.026670] CPU    6: hi:  186, btch:  31 usd:   0
[42891.026671] CPU    7: hi:  186, btch:  31 usd:  57
[42891.026672] Node 0 Normal per-cpu:
[42891.026674] CPU    0: hi:  186, btch:  31 usd: 171
[42891.026676] CPU    1: hi:  186, btch:  31 usd:   0
[42891.026677] CPU    2: hi:  186, btch:  31 usd: 176
[42891.026678] CPU    3: hi:  186, btch:  31 usd: 145
[42891.026680] CPU    4: hi:  186, btch:  31 usd:   0
[42891.026681] CPU    5: hi:  186, btch:  31 usd:   0
[42891.026683] CPU    6: hi:  186, btch:  31 usd:   6
[42891.026684] CPU    7: hi:  186, btch:  31 usd: 136
[42891.026688] active_anon:45494 inactive_anon:8079 isolated_anon:17
[42891.026688]  active_file:264 inactive_file:470 isolated_file:0
[42891.026689]  unevictable:0 dirty:0 writeback:6579 unstable:0
[42891.026690]  free:11235 slab_reclaimable:2986 slab_unreclaimable:21615
[42891.026690]  mapped:1053541 shmem:200 pagetables:7107 bounce:0
[42891.026692] Node 0 DMA free:15896kB min:20kB low:24kB high:28kB active_anon:0kB inactive_anon:0kB active_file:0kB inactive_fil
[42891.026699] lowmem_reserve[]: 0 3445 7990 7990
[42891.026702] Node 0 DMA32 free:22784kB min:4928kB low:6160kB high:7392kB active_anon:52436kB inactive_anon:10560kB active_file:
[42891.026710] lowmem_reserve[]: 0 0 4545 4545
[42891.026712] Node 0 Normal free:6260kB min:6500kB low:8124kB high:9748kB active_anon:129540kB inactive_anon:21756kB active_file
[42891.026720] lowmem_reserve[]: 0 0 0 0
[42891.026722] Node 0 DMA: 2*4kB 2*8kB 2*16kB 1*32kB 1*64kB 1*128kB 1*256kB 0*512kB 1*1024kB 1*2048kB 3*4096kB = 15896kB
[42891.026728] Node 0 DMA32: 765*4kB 386*8kB 204*16kB 139*32kB 56*64kB 13*128kB 1*256kB 1*512kB 1*1024kB 1*2048kB 0*4096kB = 2294
[42891.026734] Node 0 Normal: 315*4kB 54*8kB 56*16kB 5*32kB 2*64kB 2*128kB 1*256kB 0*512kB 1*1024kB 1*2048kB 0*4096kB = 6460kB
[42891.026740] 9590 total pagecache pages
[42891.026741] 8676 pages in swap cache
[42891.026742] Swap cache stats: add 44073, delete 35397, find 892/1022
[42891.026743] Free swap  = 6664808kB
[42891.026744] Total swap = 6835620kB
[42891.056305] 2097151 pages RAM
[42891.056308] 64340 pages reserved
[42891.056309] 1057095 pages shared
[42891.056311] 965272 pages non-shared
[42891.056313] Out of memory: kill process 8614 (cudapinner) score 1057549 or a child
[42891.056316] Killed process 8614 (cudapinner) vsz:4230196kB, anon-rss:748kB, file-rss:15560kB
[42891.073527] NVRM: VM: nv_vm_malloc_pages: failed to allocate a page
[hyperbox](0) $
```