IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

A shared file system abstraction for heterogeneous architectures

Mark Silberstein and Idit Keidar

CCIT Report #782
January 2011

Electronics
Computers
Communications

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #782 January 2011

A shared file system abstraction for heterogeneous architectures

Mark Silberstein, Idit Keidar
Technion

Abstract

We advocate the use of high-level OS abstractions in heterogeneous systems, such as CPU-GPU hybrids. We suggest the idea of an *inter-device shared file system* (IDFS) for such architectures. The file system provides a unified storage space for seamless data sharing among processors and accelerators via a standard well-understood interface. It hides the asymmetric nature of CPU-accelerator interactions, as well as architecture-specific inter-device communication models, thereby facilitating portability and usability. We explore the design space for realizing IDFS as an in-memory inter-device shared file system for hybrid CPU-GPU architectures.

1 The case for better abstractions

Recent years have seen increasingly *heterogeneous* system designs featuring multiple hardware accelerators. These have become common in a wide variety of systems of different scales and purposes, ranging from embedded SoC, through server processors (IBM PowerMP), and desktops (GPUs) to supercomputers (GPUs, ClearSpeed, IBM Cell). Furthermore, the "wheel of reincarnation" [8] and economy-of-scale considerations are driving toward fully *programmable* accelerators with *large memory capacity*, such as today's GPUs[1].

Despite the growing programmability of accelerators, developers still live in the "medieval" era of explicitly asymmetric, low-level programming models. Emerging development environments such as NVIDIA CUDA and OpenCL [1] focus on the programming aspects of the accelerator hardware, but largely overlook its interaction with other accelerators and CPUs. In that context they ignore the increasing self-sufficiency of accelerators and lock the programmers in an asymmetric CPU-centric model with accelerators treated as co-processors, second-class citizens under CPU control.

We argue that this idiosyncratic asymmetric programming model has destructive consequences on the programmability and efficiency of accelerator-based systems. Below we list the main constraints induced by this asymmetric approach.

[1]NVIDIA GPUs support up to 64GB of memory.

Problem: coupling with CPU process. An accelerator needs a *hosting* CPU process to manage its (separate) physical memory, and invoke computations; the accelerator's state is associated with that process.

Implication 1: no standalone applications. One cannot build accelerator-only programs, thus making modular software development harder.

Implication 2: no portability. Both the CPU and the accelerator have to match program's target platform.

Implication 3: no fault-tolerance. Failure of the hosting process causes also state loss of the accelerator program.

Implication 4: no intra-accelerator data sharing. Multiple applications using the same accelerator are isolated and cannot access each others' data in the accelerator's memory. Sharing is thus implemented via redundant staging of the data to a CPU.

Problem: lack of I/O capabilities. Accelerators cannot *initiate* I/O operations, and have no direct access to the CPU memory[2]. Thus, the data for accelerator programs must be explicitly staged to and from its physical memory.

Implication 1: no dynamic working set. The hosting process must pessimistically transfer all the data the accelerator would potentially access, which is inefficient for applications with the working sets determined at runtime.

Implication 2: no inter-device sharing support. Programs employing multiple accelerators need the hosting process to implement data sharing between them by means of CPU memory.

Problem: no standard inter-device memory model. Accelerators typically provide a relaxed consistency model [1] for concurrent accesses by a CPU and an accelerator to its local memory. Such a model essentially forces memory consistency at the accelerator invocation and termination boundaries only.

Implication: no long-running accelerator programs. Accelerator programs have to be terminated and restarted by a CPU before they can access newly staged data.

Implication: no forward compatibility. Programs using explicit synchronization between data transfers and accelerator invocations will require significant programming efforts to adapt to more flexible memory models likely to become available in the future.

[2]NVIDIA GPUs enable dedicated write-shared memory regions in the CPU memory, but with low bandwidth and high access latency.

Despite its shortcomings, asymmetry appears to be essential at the hardware level and will probably long remain so. But does that mean we are destined to struggle with this restrictive programming model dictated by the hardware? We argue to the contrary; we believe that asymmetry is yet another low-level hardware property that can be hidden behind higher-level OS abstractions. The key to a solution is in providing efficient mechanisms for data sharing among accelerators and CPUs.

We propose the abstraction of an ***Inter-Device shared File System*** (IDFS) with a single name space spanning all accelerators and CPUs in a single computer. IDFS provides a standard open/read/write/close interface to all participating devices, with well-defined data access semantics. Any device can create new files, or read and write files created by others. Thus, IDFS effectively eliminates the accelerator's dependence on a CPU hosting process to organize and transfer data for accelerator applications. Data location and structure are completely hidden, allowing for system-wide data transfer optimizations, which is impossible in today's application-specific solutions.

To highlight the benefits of IDFS for programmers we next consider a number of usage scenarios common in accelerator systems today. We shall speculate about future usage in Section 6.

2 Usage scenarios

We use a simple online system for image processing as a running example. The system executes a typical four-stage computing cycle which includes storing the uploaded data to a file, adding the processing task to a task queue for deferred processing, execution on accelerator, and reporting results back. This application can benefit from IDFS in a number of ways.

Standard CPU tools for post-processing the output are trivially enabled by mounting IDFS as any other VFS-compliant file system. Moreover, NFS or SMB can be used expose IDFS data to remote machines, avoiding explicit multi-hop data transfers, and allowing thrid-party transfers between accelerators in different machines.

Simple I/O, such as event logging from accelerator programs, can be trivially implemented, unlike the state-of-the-art today. Logs on IDFS can be read by standard monitoring tools thus facilitating standalone accelerator program integration into larger systems.

Legacy-accelerator program cooperation can be achieved without requiring a CPU wrapper for data staging. The web server simply writes the uploaded files directly to IDFS. The accelerator program is invoked later by the queuing system and uses IDFS I/O to read them and write the results back.

Intra- and inter-accelerator sharing allows extending the existing processing pipeline with new modules, such as accelerated encryption, without changing existing modules or staging data to a CPU. Modules communicate via files on IDFS, which automatically optimizes the data transfers among the devices.

Persistent state of accelerator programs can be maintained on IDFS across multiple invocations of processing tasks. This may be required, e.g., for machine learning algorithms that update the state and use it in later invocations. Without IDFS, the state would have to be moved from and to a CPU for each accelerator invocation, or a long-running CPU service would have to be implemented to avoid loss of the accelerator's state at the module's boundaries.

All of these scenarios illustrate the strong need for a data sharing abstraction. In the following, we justify our particular choice of a file system to serve that purpose.

3 Why file system?

Our main goal is to break the hardware coupling between accelerators and CPUs by treating accelerators as self-contained processors capable of producing and consuming data independently of other devices.

This goal is translated into the following design requirements:

- same name space, identical I/O interface and access semantics for all devices;
- data persistence; and
- high performance parallel I/O operations.

A file system abstraction naturally meets these requirements. However we also considered several design alternatives:

SMP abstraction implies emulating a single homogeneous SMP virtual machine over all processors in the system and presenting each accelerator as one or more standard CPUs [7]. This approach is currently infeasible in accelerators because of their loose coupling with the main CPU, low serial program performance, and lack of support for I/O and programmable virtual memory.

Shared memory abstraction is convenient for tightly coupled parallel applications but is often too low level, and cannot be seamlessly integrated with today's accelerators. A fine grain symmetric shared memory implementation is impractical because of the slow communication and relaxed memory consistency model of today's devices. Recently an *asymmetric* CPU-centric shared memory for GPUs has been proposed [5], but its asymmetric programming model is exactly what we want to avoid.

Unix pipes abstraction. Pipes facilitate data sharing in streaming applications with multiple concurrent data filters, potentially enabling integration of accelerators in the flow [4]. Pipes, however, do not provide persistence or random selective data access, nor do they enable temporal decoupling of data generation and consumption.

Our choice of a file system abstraction has also been inspired by the UNIX everything-is-a-file principle, which has proved extremely useful for providing a unified, well-defined interface to a variety of heterogeneous systems. Avoiding new interfaces enables all existing tools for file I/O on a CPU to be used in conjunction with accelerators.

The file system's high-level interface with explicit data access enables a variety of implementation options. It can be exposed to accelerator programs by the device drivers, and to CPU applications via a standard interface such as VFS layer implemented in the OS FS driver. Alternatively, either one of these can be realized via a user-mode library to allow gradual adoption.

Notably, a file system does not substitute and may complement other solutions, including those above. However the actual functionality, extensibility, performance and programming appeal are greatly influenced by the chosen data access semantics, which will be discussed next.

4 Access semantics

Data sharing designs with strongly consistent monolithic data images are destined for extinction. Emerging software and hardware systems abandon this approach [3, 2]. IDFS is no exception.

IDFS provides *session semantics* [6, 10] (*aka* fork-join [2]) across different devices *at the whole-file granularity*. Specifically, a session starts (ends) upon successful termination of open (close) call. The result of concurrent updates to the same file performed from different devices is a full version of one of them (one writer wins). All the writes performed by a process on a device become visible only when *the file is opened* on a reading device *after being closed successfully* by the writing device.

There are several reasons for choosing such explicit, coarse-grain semantics.

Usability. Accelerator applications with producers-consumers, master-worker style workloads are naturally supported. In fact, all the examples in Section 2 follow this pattern. Such workloads are likely to dominate accelerator applications as long as the accelerators remain loosely coupled with CPUs.

Feasibility. Data sharing is practical even in the face of weak inter-device consistency. Thus, session semantics will also enable sharing in future systems comprising a mixture of tightly and loosely-coupled devices.

Performance and fault tolerance. Coarse-grain semantics lends itself to efficient implementation using *caching* (or *replication*) of local operations to minimize the synchronization overhead over slow communication channels. In fact, this has been one of the original motivations for such semantics in AFS and Coda [6, 10]. Furthermore, using state replication inside a single machine, as advocated by Barrelfish [3], contributes to scalability and fault tolerance.

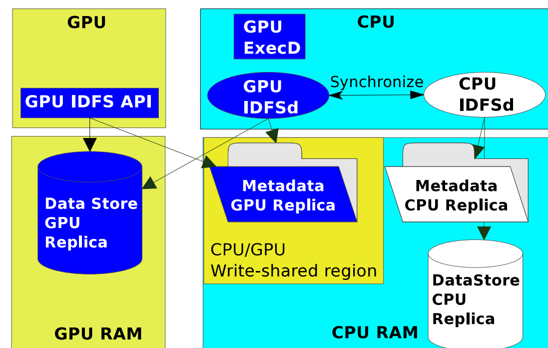

Figure 1: High level design of IDFS on CPU-GPU hybrids.

Determinism. Well-defined explicit session boundaries allow one to reason about concurrent accesses, enabling deterministic behavior as was demonstrated by Determinator OS [2].

Our initial implementation of IDFS in a CPU-GPU system illustrates its feasibility and relies heavily on replication, as we next discuss.

5 Implementing IDFS for GPUs

The focus of our ongoing work is the implementation of IDFS in a real CPU-GPU hybrid system running Linux. Here we briefly outline the main design choices. We also discuss the limitations of current technology and how we plan to cope with them.

A GPU is a massively parallel processor for executing mostly SPMD-style parallel programs called *kernels* with thousands of fine-grain threads. The program is executed directly on the hardware and managed by the GPU driver.

Figure 1 presents the main components of IDFS. The file system is replicated: replicas occupy a part of the physical memory of each device. Such design is dictated by the GPU's inability to initiate I/O during the kernel execution, and also avoids slow CPU-GPU communications by using local replica for all I/O operations.

A replica comprises the file system data and metadata. File system daemons (IDFSd) for all replicas run on a CPU and synchronize the replicas to ensure IDFS' session semantics.

GPU programs that require file system access are invoked via execution daemon (Execd) and use IDFS API. CPU programs use standard I/O API, as explained below. Note that GPU does not (and cannot) run any daemon.

The IDFS implementation can be logically divided into four layers (top-down): *API layer* for standard I/O functions, *OS layer* for managing open files, *IDFS driver*, and *data store* for internal memory management.

The CPU IDFS driver implements a standard VFS interface. Thus, IDFS can be mounted to the user space as

any other VFS-compliant file system, and take advantage of the standard OS implementation of higher layers. In contrast, for GPU all four layers have to be implemented.

5.1 IDFS I/O implementation on GPU

In GPUs thousands of concurrent threads, some running in lock step, may open, read, write and close the same file. Such massively parallel nature raises several issues.

Intra-device semantics. What should the semantics be for the threads running on the same device? Even weak session semantics may not always be feasible. Only recent NVIDIA GPUs added support for memory fences to make updates by one thread visible to others, and these incur high overhead. So we opt to provide session semantics only at the inter-device level and leave intra-device semantics undefined.

Open. In our design, GPU threads share the same file descriptor table. Intuitively, massive contention when the same file is being opened by all threads could be avoided by letting only one thread to actually open it. Unfortunately, the execution order of the threads in a GPU is non-deterministic, and no global barrier exists to synchronize all the running threads. Thus, all threads intending to work with a file must explicitly open it. We nevertheless apply some techniques to reduce contention.

Close. According to the session semantics, the close function should push an update to other devices. However if the file is concurrently opened in many threads, when should this update be propagated? We choose to do so once the file is no longer open on any of the GPU threads. We maintain the number of threads holding the file open, and when it reaches zero the update is propagated.

Parallel I/O. GPU applications will not do well to heavily rely on standard read and write operations because of the contention this would induce on the file pointer. To alleviate this contention, we support read_at and write_at operations, allowing quick random access to the data.

5.2 State replication

The lack of I/O capabilities implies that an accelerator's local IDFS replica must be up-to-date by the time the file is opened by the accelerator program. The CPU replica, on the other hand, can be synchronized on demand. Thus, the replica synchronization is performed by a CPU upon its open and close operations. Namely, the changes in the GPU's replica are retrieved when the file is opened on a CPU, and the updated data is pushed back to the GPU when the file is closed. On the other hand, GPU open and close involve no CPU-GPU communication.

In order to comply with the session semantics, closing the file on the CPU must be delayed until the GPU completes the kernel execution. Otherwise, the GPU may have an inconsistent view of its replica.

Note that *the protocol is intrinsically asymmetric*. But this asymmetry is hidden by the IDFS implementation. Also, it is *the only place* where the file system design distinguishes between accelerators and CPUs.

This design allows for various performance optimizations, such as caching read-only data, incremental updates, and eliminating redundant transfers of data generated on the GPU and never accessed by a CPU.

The protocol works well with a single accelerator. However a straightforward multi-GPU extension is inefficient. Lazy synchronization becomes impossible because *all* GPUs must have their replicas synchronized before they start executing their kernels. We are working on alleviating this limitation by updating only the files that will be actually used. Such files can be determined through the kernel code analysis or provided by the programmer.

The metadata for the GPU replica resides in the write-shared region of the CPU memory, and is thus accessible to both CPU and GPU. By placing it in the CPU memory we speed up frequent CPU operations such as directory listing that can thus be performed without high-latency GPU memory transfers. On the other hand, the GPU performance remains almost unaffected as the first open call caches the file metadata in the GPU memory and updates it only when the file is closed.

5.3 Data store

Files in IDFS can grow and shrink dynamically after creation, making the use of contiguous memory chunks for each file problematic. Instead, we use a block-structured memory pool thus enabling files to span multiple blocks.

Achieving fast data synchronization with other replicas is quite challenging. If the blocks were allocated to files without coordinating allocation between the replicas, each block in a replica would have to be synchronized separately. This would be detrimental to performance because a CPU does not have scatter/gather capabilities for transferring data to a GPU. Larger blocks would ease the overhead per data unit but would increase fragmentation, which is particularly undesirable in small-sized FS. On the other hand, coordinating each block allocation between the replicas is not practical. Our choice is to partition the blocks between the devices (statically or dynamically), so that each device uses its own memory allocation pool. Concurrent updates of the same block in several replicas are overwritten by data from a single winning replica. Files may grow on any device, hence they may occupy blocks belonging to different devices. This complicates the deletion process and file shrinking: we do not commit freed blocks of other replicas until the next replica synchronization.

The design could be simplified if only the device which created a file would be allowed to update it. However, be-

sides restricting the functionality, such design would encourage creation of small files thereby increasing space fragmentation.

Memory mapped I/O. File I/O operations effectively copy data between a user buffer and the IDFS memory space, which could be avoided by using memory mapping. Unfortunately GPU does not expose any programmable virtual memory, thus requiring IDFS implementation to provide this functionality. Memory mapped files are allocated a contiguous fixed set of blocks, reshuffling other blocks if necessary.

5.4 Open questions

Access control is imperative for any sharing mechanism in a multiuser system. The VFS-compatible interface of IDFS enables standard OS mechanisms for CPU processes. Accelerators today have no programmable memory protection mechanisms, thus providing no way for preventing unauthorized access to the FS data directly in memory. It is not clear whether such mechanisms will become available.

Large files cannot be stored on IDFS if their size exceeds the physical memory of the devices. Similarly, scaling to multiple accelerators is problematic because full IDFS replica should fit the physical memory of every one of them. This problem has no efficient solution today because accelerators cannot initiate I/O, but is likely to be resolved in more tightly coupled systems when such become available.

6 Discussion

The file system concept has been one of the cornerstones of computer systems since the early sixties, and its simplicity and versatility make it indispensable in a variety of applications. We have shown the benefits it brings to programmer for today's hybrids by abstracting away their asymmetric hardware. But *will it hold promise for inter-device data sharing in future hardware systems?*

The answer, to a large extent, depends on the degree of coupling between the devices. For the future systems-on-chip with the accelerators and CPUs residing on the same chip, IDFS may not be an ideal solution. Instead, it may become easier to fully integrate accelerators into the OS, thus providing standard I/O services to stand-alone accelerator programs. The advantage of IDFS in such case, however, is that it can be implemented in a thin layer with very modest requirements from the underlying hardware.

Furthermore, systems will likely feature both tightly- and loosely-coupled devices. This is because CPU-accelerator hybrid chip designs put significant power and memory bandwidth constraints on accelerators, and thus pay a non-negligible performance cost. Already today there are systems featuring both integrated and stand-alone powerful devices (e.g., NVIDIA Optimus technology). At any moment the one with the best power-performance balance is selected for a given workload, and the other is powered off. IDFS may be particularly convenient to transparently share data between integrated and stand-alone devices.

Finally, peripheral devices such as programmable NICs, cameras and even persistent storage controllers may benefit from direct I/O to accelerators, and such hardware is becoming available (e.g., NVIDIA direct I/O with Infiniband cards). IDFS may provide the missing glue software layer to take advantage of this functionality.

A second question one may ask is: *Will the choice of session semantics remain unchanged?* The above reasoning suggests an affirmative answer. Tighter coupling will make more frequent data updates possible, but the advantage of coarse-grain explicit consistency control is that it leaves the decision about when to synchronize in the programmers' hands. While the implementation will most likely change, the interface will remain intact.

To conclude, we believe that the file system abstraction for data sharing between devices in a single system holds high potential to drastically simplify the programming in heterogeneous systems by hiding their asymmetry. Also, it may naturally serve a glue layer between different components in the future system. Our proof-of-concept implementation for CPU-GPU hybrids is under way.

References

[1] Open standard for parallel programming of heterogeneous systems. `http://http://www.khronos.org/opencl`.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI '10*, 2010.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. SOSP '09, 2009.

[4] W. de Bruijn and H. Bos. Pipesfs: fast linux i/o in the unix tradition. *SIGOPS Oper. Syst. Rev.*, 42, July 2008.

[5] I. determinism osdi10, Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. ASPLOS '10, 2010.

[6] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6, February 1988.

[7] R. McIlroy and J. Sventek. Hera-jvm: a runtime system for heterogeneous multi-core architectures. OOPSLA '10, 2010.

[8] T. H. Myer and I. E. Sutherland. On the design of display processors. *Commun. ACM*, 11, June 1968.

[9] M. Rosenblum and J. K. Ousterhout. The lfs storage manager. In *Proceedings of the 1990 Summer Usenix*, 1990.

[10] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *Transactions on Computers*, 39(4), 1990.