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.