Appears in the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)
Austin, Texas, April 2013

Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips

Blake A. Hechtman and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

Although current homogeneous chips tightly couple the cores with cache-coherent shared virtual memory (CCSVM), this is not the communication paradigm used by any current heterogeneous chip. In this paper, we present a CCSVM design for a CPU/GPU chip, as well as an extension of the pthreads programming model for programming this HMC. We experimentally compare CCSVM/xthreads to a state-of-the-art CPU/GPU chip from AMD that runs OpenCL software. CCSVM's more efficient communication enables far better performance and far fewer DRAM accesses.

1 Introduction

The trend in general-purpose chips is for them to consist of multiple cores of various types—including traditional, general-purpose compute cores (CPU cores), graphics cores (GPU cores), digital signal processing cores (DSPs), cryptography engines, etc.—connected to each other and to a memory system. Already, general-purpose chips from major manufacturers include CPU and GPU cores, including Intel's Sandy Bridge [6][4] and AMD's Fusion [2], as well as Nvidia Research's Echelon [5].

Perhaps surprisingly, the communication paradigms in emerging heterogeneous multicores (HMCs) differ from the established, dominant communication paradigm for homogeneous multicores. The vast majority of homogeneous multicores provide tight coupling between cores, with all cores communicating and synchronizing via *cache-coherent shared virtual memory* (CCSVM). Despite the benefits of tight coupling, current HMCs are loosely coupled and do not support CCSVM.

We develop a tightly coupled CCSVM architecture and microarchitecture for an HMC consisting of CPU cores and GPU cores. *We do not claim to invent CCSVM for HMCs; rather our goal is to evaluate one strawman design in this space.* We also present a programming model that we have developed for utilizing CCSVM on an HMC. The programming model, called *xthreads*, is a natural extension of pthreads. In the xthreads programming model, a process running on a CPU can spawn a set of threads on GPU cores in a way that is similar to how one can spawn threads on CPU cores using pthreads.

2 CCSVM Chip Design

The chip consists of CPU cores and GPU cores that are connected together via some interconnection network. Each CPU core and each GPU core has its own private cache (or cache hierarchy) and its own private TLB and page table walker. All cores share one or more levels of globally shared cache. This cache is logically shared and CPU and GPU cores can communicate via loads and stores to this cache.

We illustrate the organization of our microarchitecture in Figure 1. The key aspect of our architecture is to extend CCSVM from homogeneous to heterogeneous chips. The CCSVM design is intentionally simple and conservative. The architecture provides sequential consistency using an unoptimized cache coherence protocol that treats CPU and GPU cores identically.

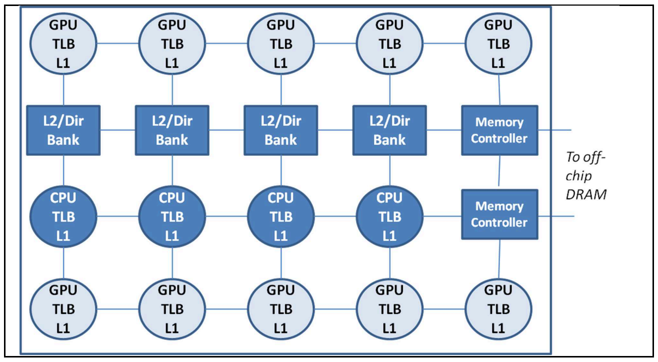

Figure 1. System Model.

3 Xthreads Programming Model

We developed xthreads with the goal of providing a pthreads-like programming model that exploits CCSVM and is easy to use. The xthreads API extends pthreads by enabling a thread on a CPU core to spawn threads on non-CPU cores. The two primary mechanisms for synchronization are wait/signal and barrier. A CPU thread or a set of GPU threads can wait until signaled by CPU or GPU threads. The wait/signal pair operates on condition variables in memory. The barrier function is a global barrier across one CPU thread and a set of GPU threads.

When a process begins, the CPU cores begin executing threads as they would in a "normal" pthreads application. The differences begin when a CPU initiates a GPU task with *N* threads. For each task, the library performs a write syscall to the GPU interface device (GIFD). The GIFD then assigns incoming tasks to available GPU cores. A GPU core with a task begins executing from the program counter it receives. When it reaches an Exit instruction, it halts and waits for the GIFD to send it a new task.

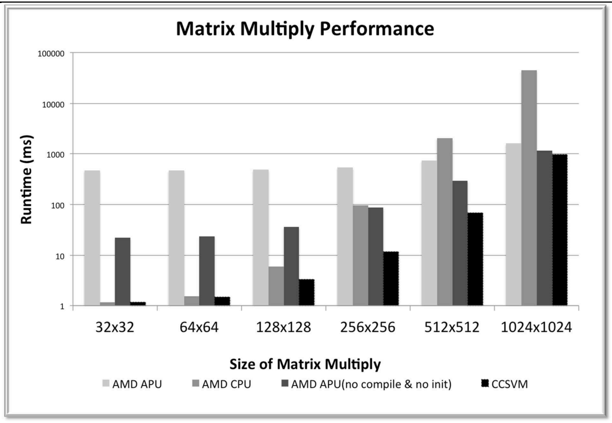

Figure 2. Performance on Matrix Multiply

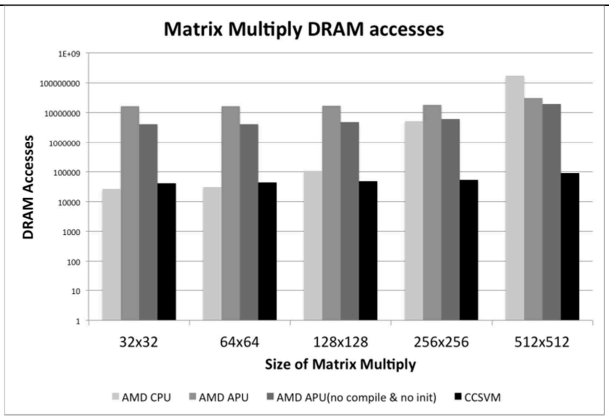

Figure 3. DRAM Accesses for Matrix Multiply

4 Experimental Evaluation

We have implemented our chip design in the gem5 full-system simulator [1]. Our extensions to gem5 enable it to faithfully model the functionality and timing of the entire system. The simulated CPU cores are in-order x86 cores that run unmodified Linux 2.6 with the addition of our simple GIFD driver (~30 lines of C code). The GPU cores are SIMT cores that have an Alpha-like ISA that has been modified to be data parallel.

We purchased an AMD "Llano" system based on its Quad-Core A8-3850 APU [3], and we use this real hardware running OpenCL software for comparisons to CCSVM running xthreads software.

To experimentally demonstrate the potential benefits of tighter coupling between the CPU and GPU cores, we compare the execution of a (dense) matrix multiplication kernel that is launched from a CPU to as many GPU cores as can be utilized for the matrix size. Intuitively, the overhead to launch a task will be better amortized over larger task sizes, and the benefit of CCSVM will be highlighted by how it enables smaller tasks to be profitably offloaded to the GPU cores. In **Figure 2**, we plot the *log-scale* runtimes of the AMD APU running OpenCL code and CCSVM running xthreads code, relative to the AMD CPU core (i.e., just using the CPU core on the APU chip), as a function of matrix sizes.

The results are striking: CCSVM/xthreads greatly outperforms the APU, especially for smaller matrix sizes. Eventually, as the matrices reach 1024x1024, the APU's performance catches up to CCSVM/xthreads, because the APU's raw GPU performance exceeds that of our simulated GPU cores. The results dramatically confirm that optimizing the communication between the CPU and GPU cores offers opportunities for profitably offloading smaller units of work to GPU cores, thus increasing their benefits.

CCVSM avoids the vast amount of off-chip traffic that current chips require for CPU-GPU communication. We plot the memory traffic results in Figure 3. As with the performance results, the differences between the APU/OpenCL and CCSVM/xthreads are dramatic. Furthermore, as the problem size increases, that ratio remains roughly the same, and the number of DRAM accesses from the AMD CPU core increases greatly as the working set outgrows the CPU core's caches.

These DRAM access results have two implications. First, the results help to explain the performance results. The APU requires far more DRAM accesses, and these long-latency off-chip accesses hurt its performance, relative to CCSVM and its largely on-chip communication. Second, given both the importance of using DRAM bandwidth efficiently and the energy consumed by DRAM accesses, the results show that CCSVM/xthreads offers large advantages for system design.

5 Conclusions

We have demonstrated that the tight coupling of cores provided by CCSVM can potentially offer great benefits to an HMC. Our CCSVM architecture with the xthreads programming model is a functional and promising starting point for future research into new features (e.g., programming language extensions) and optimizations for performance and efficiency.

Acknowledgment

This material is based on work supported by the National Science Foundation under grant CCF-1216695.

References

[1] N. Binkert et al., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, Aug. 2011.
[2] B. Burgess, B. Cohen, M. Denman, J. Dundas, D. Kaplan, and J. Rupley, "Bobcat: AMD's Low-Power x86 Processor," *IEEE Micro*, vol. 31, no. 2, pp. 16–25, Mar. 2011.
[3] D. Foley et al., "AMD's 'Llano' Fusion APU," in *Hot Chips 23*, 2011.
[4] Intel, "Intel® OpenSource HD Graphics Programmer's Reference Manual (PRM), Volume 1, Part 1: Graphics Core (SandyBridge)." May-2011.
[5] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Oct. 2011.
[6] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A Fully Integrated Multi-CPU, GPU and Memory Controller 32nm Processor," in *2011 IEEE International Solid-State Circuits Conference*, 2011, pp. 264–266.