Rigel: A Scalable Architecture for 1000+ Core Accelerators

Daniel R. Johnson, John H. Kelm, Neal C. Crago, Matthew R. Johnson, William Tuohy,
Wojciech Truty, Stephen Kofsky, Steven S. Lumetta, Wen–mei W. Hwu, Matthew I. Frank, Sanjay J. Patel
University of Illinois at Urbana-Champaign

Abstract

We describe Rigel, an architecture for 1000+ core MIMD accelerators, and its Low-level Programming Interface (LPI). We describe Rigel's cached single address space memory hierarchy, motivated by dominant BSP-like application characteristics. We provide analysis of Rigel in the form of kernel scalability results as well as area and power estimates for a 1024-core design. We find that Rigel can achieve a density of 8 single-precision $\frac{GFLOPS}{mm^2}$ in 45nm, comparable to high-end GPUs scaled to 45nm.

1 Introduction

Contemporary general-purpose chip multiprocessor (CMP) development is driven by the need to support multitasking operating systems, legacy code, and a broad spectrum of applications. In contrast, *compute accelerators* are hardware entities designed to improve performance and reduce power for a specific class of applications by exploiting characteristics of the target domain. Accelerators are optimized for a narrower class of workloads and programming styles and architected to maximize throughput ($\frac{operations}{sec}$) while general-purpose processors minimize latency ($\frac{sec}{operation}$). Examples of compute accelerators include graphics processing units [7] (GPUs), Cell [2], and Larrabee [9]. GPU [6, 8] and stream computing [1] applications such as MRI reconstruction [11] and molecular dynamics simulation [10] have shown considerable speedups over high-end conventional CMPs.

Current accelerators provide restricted programming models which yield high performance for data-parallel applications with regular computation and memory access patterns, but present a more difficult target for less regular applications. Generally, existing compute accelerators provide higher throughput via architectural choices that compromise the generality of the programming model.

We describe Rigel [3], a programmable MIMD accelerator architecture for a broad class of data- and task-parallel computation, especially visual computing workloads. Rigel comprises 1024 hierarchically-organized cores that use a fine-grained, dynamically scheduled single-program, multiple-data (SPMD) execution model. Rigel's low-level programming interface adopts a single global address space model where parallel work is expressed in a task-centric, bulk-synchronized manner using minimal hardware support. Compared to existing accelerators, which contain domain-specific hardware, specialized memories, and restrictive programming models, Rigel is more flexible and provides a more straightforward target for a broader set of applications.

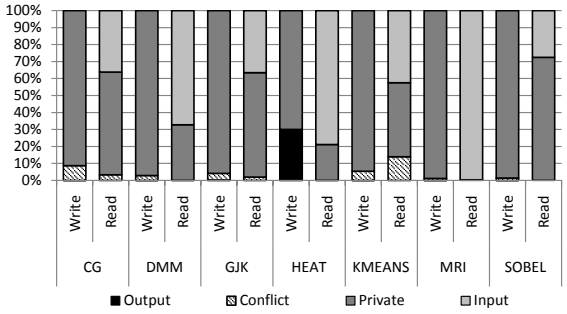

Figure 1. Read and write sharing between independent tasks in our benchmarks

We observe that data sharing and communication patterns in visual computing workloads can be leveraged in the design of memory systems for future 1000-core processors. Based on these insights, we propose a memory model [4] that uses a software protocol, working in concert with hardware caches, to maintain a coherent, single-address space view of memory without the need for hardware coherence.

2 Application Characteristics

Accelerators place different constraints on caches and coherence management relative to CMPs. Opportunity exists to exploit these differences due to characteristics of visual computing workloads. To enlarge the space of applications accelerators target, they must not only support the data-parallel execution model prevalent today, but also irregular data- and task-parallel computation not well-suited to contemporary SIMD accelerators.

We propose the *task-centric memory model* [4], a hard-

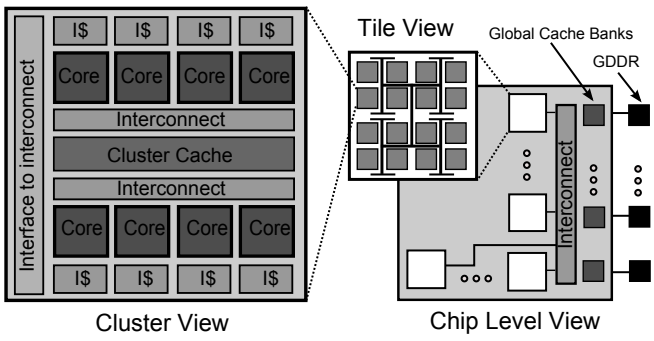

Figure 2. Diagram of the Rigel processor.

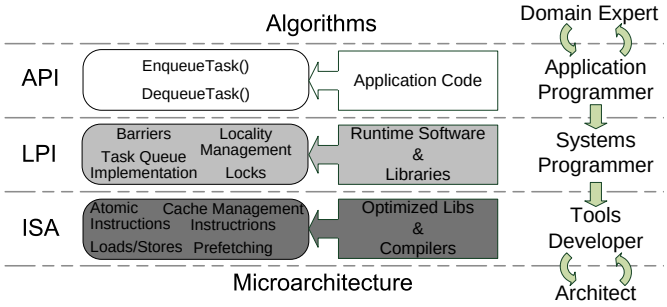

Figure 3. Accelerator software stack

ware/software protocol for maintaining a coherent view of shared memory for accelerators. The model exploits sharing patterns we observe in visual computing workloads that are developed using a form of bulk synchronous processing (BSP) [12] in which parallel work units (*tasks*) execute independently between barriers, a period that we denote an *interval*.

The data access properties of these workloads include: a high degree of read-sharing among tasks within an interval, a private working set with updates that need only be made globally visible at the end of an interval (if at all), and a small amount of data that is shared amongst tasks and must be kept coherent within an interval. Figure 1 shows data sharing for our kernels; conflicts indicate read-write sharing within an interval, while inputs and outputs indicate sharing between intervals. System task management code is excluded. More detailed workload analysis may be found in [4] and [5].

Collectively, these characteristics demonstrate that little coherence management is required within an interval, indicating the potential to push coherence management into software (to be logically performed at the end of an interval). At the same time, mechanisms must be present to allow small amounts of fine-grained synchronization and data sharing within an interval.

3 Rigel Architecture

Overview Rigel [3] is a MIMD compute accelerator targeting task- and data-parallel visual computing workloads that scale up to thousands of concurrent tasks. The design objective of Rigel is to provide high compute density while enabling an easily targeted conventional programming model. A block diagram of Rigel is shown in Figure 2.

The fundamental processing element of Rigel is an area-optimized dual-issue in-order core with a RISC-like ISA, a single-precision floating-point unit, and an independent fetch unit. Eight cores and a unified cache comprise a single Rigel *cluster*. Clusters are grouped logically into a *tile* using a bi-directional tree-structured interconnect. Eight tiles of 16 clusters each are distributed across the chip, attached to global cache banks via a multistage crossbar interconnect. The global caches provide buffering for multiple high-bandwidth memory controllers. Our initial design incorporates 8 GDDR memory controllers and 32 global cache banks.

LPI The Low-level Programming Interface (LPI) is an extended version of the traditional CPU LPI, i.e., the ISA, and is intended as an easier target for programmers and programming tools. The LPI is the interface between the applications development environment and the underlying software/hardware system of the accelerator, as illustrated in Figure 3. Given that programmable accelerators provide their performance through large-scale parallel execution, the LPI must include primitive operations for expressing and managing parallelism. The accelerator LPI needs to be implemented in a scalable and efficient manner using a combination of hardware and low-level system software. The LPI should provide an effective way to exploit the accelerator's compute throughput.

Tradeoffs are made in Rigel's LPI between generality and accelerator performance. The elements that we identify as necessary for supporting these objectives include: the execution model, the memory model, work distribution, synchronization, and locality management.

The Rigel execution model omits complex ILP-oriented cores in favor of simple in-order cores. A more flexible MIMD model is chosen over a potentially denser SIMD model. A degree of multithreading is beneficial in improving throughput and is under investigation. We choose to present application software with a single global address space, a general-purpose memory model typical of CMPs. The Rigel LPI supports task queues for work distribution based on the prevalent BSP execution model, the de facto model for many other accelerator platforms such as CUDA-based GPUs. The Rigel LPI supports two classes of global synchronization: global barrier support (software) and atomic primitives (hardware). Rigel supports a variety of memory operations to aid in locality management, including prefetches, local and global memory operations, and explicit cache management instructions.

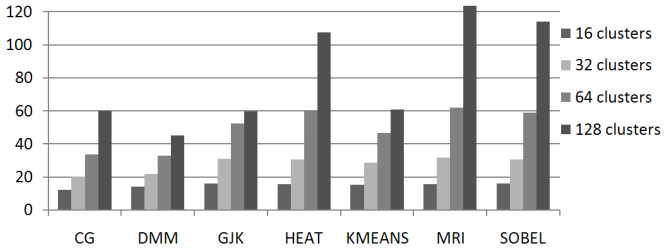

Figure 4. Kernel scalability (speedup)

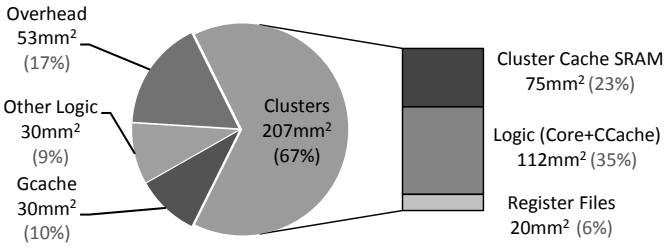

Figure 5. Area estimates for the Rigel design

Cache Management All cores share a single global address space. Cores within a cluster have the same view of memory via the shared cluster cache, while global coherence is not explicitly maintained by hardware between clusters. When serialization of accesses is necessary between clusters, the global cache is the point of coherence. Rigel implements two classes of memory operations: *local* and *global*.

Local memory operations are intended to constitute the majority of memory operations. Local memory read operations are cacheable at the cluster cache, but are not kept coherent between clusters by hardware. Local memory operations are used for accessing read-only data, private data, and data shared intra-cluster.

Global loads, stores, and atomics on Rigel bypass the cluster caches and complete at the global cache. Memory locations operated on solely by global memory operations are kept coherent across the chip. Global operations are key to supporting system resource management, synchronization, and fine-grained inter-cluster communication. The cost of global memory operations is high relative to local operations due to increased latency.

For more detail on Rigel's memory system, see [4].

4 Results

Performance Figure 4 shows scalability relative to one cluster for our selection of benchmark kernels. Scaling is achieved across a spectrum of design sizes without recompilation. In addition, the vast majority of runs suffer less than 15% software task management overhead. More details on benchmarks and analysis of Rigel appear in [3].

Area and Power To demonstrate feasibility of Rigel, we provide area and power estimates on 45nm technology. Our estimates are derived from synthesized Verilog, compiled SRAM arrays, IP components, and die plot analysis of other 45nm designs. Figure 5 shows a breakdown of preliminary area estimates for the Rigel design. Cluster caches are 64KB each and global cache banks total 4MB. "Other Logic" includes interconnect as well as memory and global cache controller logic. For a conservative estimate, we include a 20% charge for additional overheads. The resulting 320mm^2 is reasonable for current process technologies.

Typical power consumption of the design with realistic activity factors for all components at 1.2GHz is expected to be in the range of 70–99W. Our estimate is based on power consumption data for compiled SRAMs, post-synthesis power reports for logic, leakage, and clock tree of cluster components, and estimates for interconnect and I/O pin power.

References

[1] U. J. K. et al. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[2] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF'06*, pages 1–8. ACM, 2006.

[3] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *ISCA'09*, June 2009.

[4] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. J. Patel. A task-centric memory model for scalable accelerator architectures. In *PACT'09*, September 2009.

[5] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *MICRO'08*, 2008.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2), 2008.

[7] NVIDIA. NVIDIA GeForce 8800 GPU architecture overview, November 2006.

[8] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[9] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27, 2008.

[10] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28:2618–2640, 2007.

[11] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, B. P. Sutton, and Z. P. Liang. Accelerating advanced mri reconstructions on gpus. *J. Parallel Distrib. Comput.*, 68(10):1307–1318, 2008.

[12] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.