Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator

John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago,
William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, Sanjay J. Patel
University of Illinois at Urbana-Champaign
Urbana, IL 61801

ABSTRACT

This paper considers Rigel, a programmable accelerator architecture for a broad class of data- and task-parallel computation. Rigel comprises 1000+ 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 straightforward target for a broader set of applications.

We perform a design analysis of Rigel to quantify the compute density and power efficiency of our initial design. We find that Rigel can achieve a density of over 8 single-precision $\frac{GFLOPS}{mm^2}$ in 45nm, which is comparable to high-end GPUs scaled to 45nm. We perform experimental analysis on several applications ported to the Rigel low-level programming interface. We examine scalability issues related to work distribution, synchronization, and load-balancing for 1000-core accelerators using software techniques and minimal specialized hardware support. We find that while it is important to support fast task distribution and barrier operations, these operations can be implemented without specialized hardware using flexible hardware primitives.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms

Design, Performance

Keywords

Accelerator, Computer Architecture, Low-level programming interface

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

1. INTRODUCTION

An accelerator is a hardware entity designed to provide advantages for a specific class of applications. Accelerators exploit characteristics of the target domain to deliver some combination of higher performance, lower power, and lower unit cost compared to general-purpose processors. Depending on the domain, accelerators utilize architectural features such as stream-based data paths, vector processing, exotic memory systems, and special-purpose functional units tailored to the computation and communication patterns of target workloads.

Domains including 3D graphics, audio, image, and video processing find fixed-function logic beneficial because it can offer significant power, area, and throughput advantages over programmable logic [7]; programmability is less essential when the workload can be described by a small, fixed set of algorithms or encapsulated in an API. Some domains require programmability for other reasons, such as generality within a single domain, lack of standards, diversity of computation, or complexity of a fixed-function solution.

Recent commercial and academic activities in GPU computing [20, 21], stream computing [13], and many-core systems [25] have created an awareness of the application possibilities for programmable compute accelerators that exploit large degrees of parallelism. Due to their programmability, such accelerators can be applied to a variety of domains that require high computational performance provided the domains exhibit large amounts of data parallelism. For example, applications such as MRI image reconstruction [28], molecular dynamics simulation [27], and protein folding [5] have demonstrated 10x or greater acceleration over conventional high-end multi-core CPUs using GPU computing.

Accelerators are architected to maximize throughput, or $\frac{operations}{sec}$, while their general-purpose counterparts are designed to minimize latency, or $\frac{sec}{operation}$. Accelerators rely less on latency-based optimizations such as caching, high-frequency operation, and speculation to achieve high performance. Accelerators are able to achieve an order of magnitude higher throughput/area and throughput/watt compared to CPUs by limiting programmability, supporting specific parallelism models, and implementing special-purpose memory hierarchies and functional units. While restricting the programming model yields high performance for data-parallel applications that have regular computation and memory access patterns, it presents a difficult target for applications that are less regular. GPUs, for example, achieve high compute density with specialized memories and data paths

optimized for processing vectors with hundreds or thousands of elements. GPUs thus require the programmer to manage the memory hierarchy and minimize control flow divergence within groups of parallel threads in order to obtain high performance. Generally speaking, existing compute accelerators provide higher throughput than CPUs via architectural choices that often compromise the programming model.

In this paper we provide a rationale, design overview, and early evaluation of Rigel, an architecture and programming interface for a 1000+ core fine-grained parallel accelerator. The Rigel design strikes a balance between performance and programmability by adopting programming interface elements found in conventional general-purpose processor interfaces and adapting them for high-throughput execution. We demonstrate that compromises in development of the programming interface can be made in a principled manner so that the end product provides high compute density, scalability, and high performance for a broad class of applications, while maintaining a general-purpose programming model that programmers are accustomed to.

In Section 2, we provide a top-down motivation for the Rigel architecture by deriving a set of *elements* that we identify as requirements for the low-level programming interface to Rigel. We derive these elements based on experimental observations we have made in the design and development of the Rigel architecture and software targeting it, on anecdotal experience writing codes for other compute accelerators, and on constraints placed on the architecture and microarchitecture by physical design.

In Section 3, we describe the microarchitectural execution model for Rigel. We discuss the hierarchical core and cluster architecture and provide an estimate of the compute throughput and power based on the mapping of a 1024-core Rigel design onto 45nm technology. Our analysis is based on area estimates derived from synthesis of an early RTL implementation using commercial standard cell libraries, IP blocks, and a memory compiler. We also describe the memory model and the cache hierarchy, support for coherence, and motivate the operations supported by the Rigel instruction set architecture (ISA) to provide fast synchronization, globally coherent and locally coherent memory accesses, and locality management.

In Section 4, we describe the implementation of the low-level programming interface for Rigel, which we refer to as the Rigel Task Model. The low-level programming interface supports enqueing of work into queues, which are resident in memory, that dynamically distribute units of work across the chip.

In Section 5, we provide experimental studies of a diverse set of computation kernels extracted from applications that require high performance. We show the performance and scalability of these codes on simulated Rigel configurations and evaluate the utility of hardware primitives for our design. We demonstrate that with a modest amount of specialized hardware for certain elements, we support a dynamic task-parallel programming model that is flexible and efficient.

2. MOTIVATION AND OBJECTIVES

In this section, we provide a top-down motivation of the Rigel programming interface which includes the set of functionality to be supported by the architecture and low-level software of a programmable compute accelerator. Programmable accelerators span a wide spectrum of possible architectural models. At one end of the spectrum are FPGAs, which can provide extremely high compute density and fine-grained configurability at the cost of a very low-level native application programming interface (API). The gate-level orientation of the FPGA interface, i.e., netlist, creates a large semantic gap between traditional programming languages, such as C or Java, and the low-level programming interface (LPI). The semantic gap requires that the programmer make algorithmic transformations to facilitate mapping or bear the loss of efficiency in the translation and often, both. The other end of the spectrum is represented by hardware accelerators and off-load engines tightly-coupled to general-purpose processors. Examples include TCP/IP and video codec accelerators incorporated into systems-on-a-chip. Here the LPI is an extended version of the traditional CPU LPI, i.e., the ISA, and thus makes an easier target for programmers and programming tools.

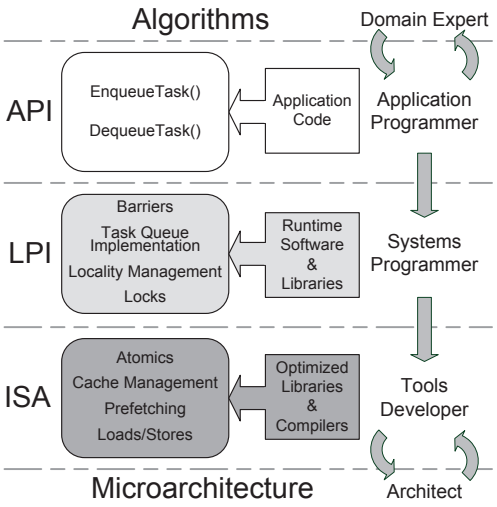

Figure 1: Software stack for accelerator architectures. API: Application-level Programming Interface, LPI: Low-level Programming Interface, ISA: Instruction Set Architecture.

Akin to an instruction set architecture, the LPI is the interface between the applications development environment and the underlying software/hardware system of the accelerator. The LPI subsumes the ISA: as with any uniprocessor interface, the accelerator interface needs to provide a suitable abstraction for memory, operations, and data types. Given that programmable accelerators provide their performance through large-scale parallel execution, the LPI also needs to 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.

What is desirable from a software development point of view is a programmable accelerator with an LPI that is a relatively small departure from a conventional programming interface. The LPI should also provide an effective way to exploit the accelerator's compute throughput. In this section, we motivate the tradeoffs made in Rigel between generality in the LPI and accelerator performance. To that end, we

describe the *elements* that we identify as necessary for supporting these objectives. The elements described include: the execution model, the memory model, work distribution, synchronization, and locality management.

2.1 Element 1: Execution Model

The execution model is the mapping of the task to be performed, specified by the application binary, to the functional units of the processor. The choice of execution model is ultimately driven by characteristics of the application domain and its development environment. The overarching goal for accelerators is for the execution model to be powerful enough to efficiently support common concurrency patterns, yet be simple enough for an implementation to achieve high compute density. The execution model encompasses the instruction set, including its level of abstraction and use of specialized instructions, static versus dynamic instruction-level parallelism, e.g., VLIW versus out-of-order execution, and SIMD execution versus MIMD.

The goal for Rigel is to develop a general-purpose execution model suitable for compact silicon implementation. The choice of the SPMD execution model is backed by previous studies and experience that shows that the SIMD model imposes undue optimization costs for many irregular applications. Mahesri et al. [17] show that even considering the area benefit of SIMD, some parallel applications scale poorly on long vector architectures, reducing the *effective* compute density of the accelerator.

2.2 Element 2: Memory Model

The design of a memory model for a parallel programmable system involves a choice of memory hierarchy, including software-managed memories such as those found in the Cell Processor [10] or multiple specialized address spaces found in GPUs [16], as well as choices regarding explicit versus implicit interprocessor communication and allowable memory orderings. Tradeoffs between these choices are hard to quantify, but it is understood that one can generally reduce hardware complexity, thus increasing compute throughput, by choosing simpler, software-controlled mechanisms, albeit at additional complexity in software development.

2.3 Element 3: Work Distribution

When an application reaches a section of code suitable for parallel acceleration, work is systematically distributed to available chip resources, ideally in a fashion that maximizes the throughput of the accelerator. With Rigel, we adopt a task-based work distribution model where parallel regions are divided into parallel tasks by the programmer, and the underlying LPI provides mechanisms for distributing tasks across the parallel resources at runtime in a fashion that minimizes overhead. Such an approach is more amenable to dynamic and irregular parallelism than approaches that are fixed to parallel loop iterations.

In Section 4 we discuss the actual programmer interface for the Rigel Task Model, an API for enqueing and dequeuing tasks, supported by a small number of primitives in the underlying LPI. We show in Section 5 that the Rigel Task Model can support fine-grain tasks at negligible overhead at the scale of 1000 cores.

2.4 Element 4: Synchronization

Selection and implementation of synchronization primitives abounds in the literature. Blelloch [2] describes the generality of reduction-based computations. The implementation of barriers in particular has been accomplished with cache coherence mechanisms [18], explicit hardware support such as the Cray T3E [24], and more recently, a combination of the two on chip multiprocessors [23]. Using message passing networks to accelerate interprocess communication and synchronization was evaluated on the CM-5 [15]. Interprocessor communication using in-network combining in shared-memory machines such as in the NYU Ultracomputer [9] and using fetch-and-ϕ operations as found in the Illinois CEDAR computer [8] have also been studied. These designs give relevant examples that influence our work as we reevaluate the tradeoffs of past designs in the context of single-chip, 1000+ core, hierarchical accelerators.

The ability to support fine-grained tasks, and thus a high degree of parallelism, requires low-latency global synchronization mechanisms. Limiting the scope to data- and task-parallel computation focuses the support required for Rigel to two classes of global synchronization: global barrier support, which is required to synchronize at the end of a parallel section, and atomic primitive support, which is useful for supporting shared state, such as updating a global histogram using the atomic increment primitive.

2.5 Element 5: Locality Management

Locality management involves the co-location of tasks onto processing resources with the goal of increased local data sharing to reduce the latency and frequency of communication and synchronization amongst co-located tasks. Locality management can be performed by a combination of programmer effort, compiler tools, runtime systems, and hardware support. In programming parallel systems, performing locality-based optimization constitutes a significant portion of the application tuning process. An example of locality management is blocked dense matrix multiply, in which blocking factors for parallel iterations increase the utility of shared caches by maximizing data reuse and implicit prefetching across threads while amortizing the cost of cache misses.

Accelerator hardware and programming models also rely heavily on locality management. Modern GPUs such as the NVIDIA G80 make use of programmer-managed local caches and provide implicit barrier semantics, at the warp-level, using SIMD execution [16]. The CUDA programming model allows for the programmer to exploit the benefits of shared data using the shared memories of the GPU, fast synchronization across warps using `__syncthreads` primitives, and the implicit gang scheduling of threads through warps and thread blocks. Models such as Sequoia [6] and HTA [11] demonstrate examples of how to manage locality on accelerators such as the Cell and for clusters of workstations.

Memory bandwidth has historically lagged available compute throughput; thus, the memory bandwidth a single chip can support limits achievable performance [3]. The cost of communicating across the chip has grown to where it takes hundreds of cycles to perform cross-chip synchronization or memory operation between two cores [1]. Because they are optimized for compute throughput on kernels, accelerators

tend to have smaller amounts of on-chip cache per core. The fraction of per-core cache allocated to each processing element in modern accelerators, which can be on the order of kilobytes [16], is a fraction of the megabytes per core available on a contemporary multicore CPU. The communication latency, synchronization overheads, and limited per-core caching are all indicative that the locality management interface is a critical component of an LPI moving forward.

2.6 Low-level Programming Interface

We conclude this section with an overview of the Rigel LPI, summarizing the points raised in the earlier subsections. The low-level programming interface to Rigel supports a simple API for packaging up tasks that are managed using a work queue model. The individual tasks are generated by the programmer, who uses the SPMD execution model and single global address space memory model in specifying the tasks. It is the responsibility of the work distribution mechanism, the Rigel Task Model implementation, to collect, schedule, and orchestrate the execution of these tasks. Execution of these tasks is based on the prevalent Bulk Synchronous Parallel (BSP) [31] execution model, which is also the de facto model for many other accelerator platforms such as CUDA-based GPUs. With BSP, a parallel section of tasks is followed by global synchronization, followed by the next parallel section.

The Rigel LPI supports task queues as a means to distribute tasks. Global synchronization is provided by an implicit barrier when all tasks for a given phase of the computation have completed, forming an intuitive model for developers. The Rigel LPI also provides a means to implicitly (at barriers) or explicitly (under software control) make updates to shared state globally visible before entering a barrier to provide a coherent view of memory to programmers.

Locality management at the low-level programming interface is provided via a combination of mechanisms to co-locate groups of tasks to clusters of cores on chip and to manage the cache hierarchy. Much of the locality management is provided implicitly by hardware-managed caches that exploit temporal and spatial locality, as with a typical CPU. A programmer can tune the effectiveness of these implicit structures through co-location of tasks to increase reuse of shared data. To that end, the Rigel LPI supports grouping of tasks that have similar data access streams, thus increasing the effectiveness of local caches for co-located tasks. Similarly, tasks that require local synchronization can be co-located onto the same cluster of cores, thus synchronizing through the local caches with less overhead than with global synchronization. To provide explicit control when necessary, the Rigel LPI supports cache management instructions, explicit software-controlled flushes, memory operation that bypass local caches, and prefetch instructions for explicit control for performance-minded programmers to extract higher performance from the accelerator when desired.

With the LPI for Rigel, we choose to present application software a general-purpose memory model typical of multicore CPUs: a single global address space across the various cores of the accelerator. The address space can be cached and is presented to the programmer in a coherent way; however the actual hardware does not provide coherence directly. With such a model, managing the memory hierarchy can be done implicitly by the software. Interprocessor communication is implicit through memory, reducing the semantic gap between high-level programming and the LPI. Providing implicit support for the memory model creates an implementation burden on the underlying LPI: if the address space is cached, which is required to conserve memory bandwidth, then one needs to consider the overheads of caching and also coherence, discussed with respect to hardware and software in Sections 3 and 4, respectively.

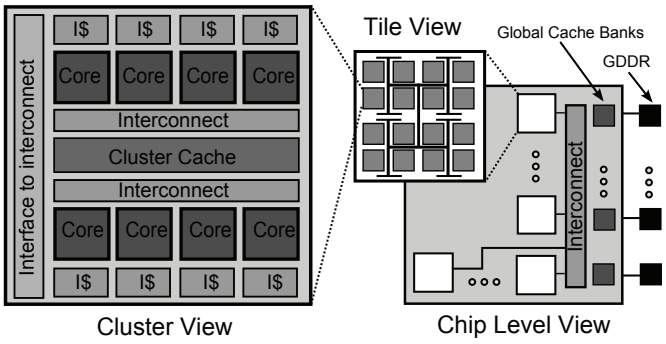

Figure 2: Diagram of the Rigel processor.

3. THE RIGEL ARCHITECTURE

The architectural objective of Rigel is to provide high compute throughput by minimizing per-core area while still supporting a SPMD parallel model and a conventional memory heirarchy. Density is improved by focusing on the LPI, identifying which elements of the LPI for Rigel should be supported directly in hardware versus those that can be supported by low-level software.

A block diagram of Rigel is shown in Figure 2. The fundamental processing element of Rigel is an area-optimized dual-issue in-order processing core. Each core has a fully-pipelined single-precision floating-point unit, independent fetch unit, and executes a 32-bit RISC-like instruction set with 32 general-purpose registers. Cores are organized as *clusters* of eight cores attached to a shared write-back data cache called the *cluster cache*. The cores, cluster cache, core-to-cluster-cache interconnect and the cluster-to-global interconnect logic comprise a single Rigel cluster. Clusters are connected and grouped logically into a tile. Clusters within a tile share resources on a tree-structured interconnect. Tiles are distributed across the chip and are attached to *global cache* banks via a multi-stage crossbar interconnect. The global caches provide buffering for high-bandwidth memory controllers and are the point of coherence for memory.

In this section, we provide a description and an analysis of the Rigel Architecture. We find that in 45nm technology, a 320mm^2 Rigel chip can have eight GDDR memory channels, 32 global cache banks (4MB) and eight tiles of 128 clusters resulting in 1024 cores across the chip. At a frequency of 1.2 GHz, a peak throughput of 2.4 TFLOPS is achievable. We show that the peak performance of the chip is comparable to commercial accelerators scaled to the 45nm process generation. We show that the achievable performance for a variety of accelerator kernels enables Rigel to strike a good balance between a flexible programming interface and high compute throughput.

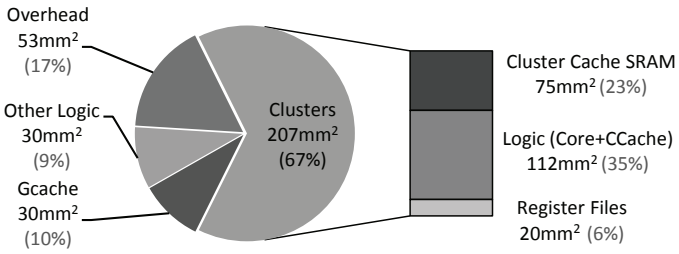

Figure 3: Area estimates for the Rigel Design

3.1 Caching and Memory Model

Cores comprising the Rigel processor share a single address space. Cores within a cluster share the same components of the memory hierarchy (except register files) and are thus coherent by design. Cores in separate clusters, in contrast, have different cluster caches, and thus require some coherence protocol if the memory hierarchy is to be kept consistent. We describe the mechanism for maintaining coherence of the cluster caches and the primitive low-level operations required for supporting the Rigel memory model.

Rigel cores have access to two classes of memory operations: *local* and *global*. Local memory operations are akin to standard memory operations and are fully cached by a core's cluster cache. They constitute the majority of memory operations and support low latency and high throughput for data accesses. Values evicted from the cluster cache are written back to the global cache, which also services cluster cache misses. The cluster and global cache are not kept coherent in hardware and are non-inclusive, non-exclusive. Local loads that miss in both the cluster and global cache are cached in the global cache to facilitate read sharing and are brought into the cluster cache of the requesting core. Local stores are not made visible to the global cache until an eviction occurs or the data is explicitly flushed by software. Local operations are used for accessing read-only data, private data, and data shared intra-cluster. Per-word dirty bits are used to merge updates at the global cache and eliminate the performance penalty caused by false sharing and lost updates due to partial writes to shared cache lines.

Global operations always bypass cluster caches without updating its contents. Global memory operations on Rigel complete at the global cache which is the point of full-chip coherence. Memory locations operated on solely by global memory operations are kept inherently coherent across the chip. Global memory operations are used primarily by the low-level software to construct synchronization mechanisms and enable fine-grained inter-cluster communication through memory, i.e., they are *used to implement the LPI*. The cost of global operations is high relative to local operations due to the greater latency of accessing the global caches versus the local cluster caches. Furthermore, the achievable global memory throughput is limited by the global cache port count and on-chip global interconnect bandwidth.

Software must enforce coherence in the scenarios when inter-cluster read-write sharing exists. This may be done by co-locating sharers on a single (coherent) cluster, by using global memory accesses for shared data, or by forcing the writer to explicitly flush shared data before allowing the reader to access it.

Memory ordering on Rigel is defined separately for local and global memory operations. Global memory operations are kept coherent across the chip with respect to other global memory operations by forcing all global memory operations to complete at the global caches. The ordering between local and global operations from a single core can be enforced by using explicit memory barrier operations. A memory barrier forces all outstanding memory operations from a cluster to complete before allowing any memory operations after the memory barrier to begin.

3.2 Coherence and Synchronization

The algorithm used for enforcing cache coherence on Rigel is not implemented in hardware, but instead exploits the sharing patterns present in accelerator workloads to enforce coherence in software using a form of lazy write-through at barriers. Mutable data shared across clusters on Rigel could be kept coherent between sharers by forcing all memory accesses to be made using global operations; however the cost of using only global memory operations is high and strains global network and cache resources. One of the key motivations for Rigel is that many accelerator workloads have a low frequency of inter-core write-shared data *between* two consecutive barriers. As an example, Mahesri et al. [17] demonstrate the relative lack of inter-core shared data on a set of visual computing workloads similar to what Rigel targets. Instead, most read-write sharing occurs *across* barriers, in the form of *write-output data*. Rigel exploits this fact by lazily writing back data prior to barriers, avoiding long-latency global memory operations.

The sharing patterns present in our target workloads allow Rigel to leverage local caches for storing write-output data between barriers before lazily making modifications globally-visible. Lazy updates can be performed as long as coherence actions performed to write-output data are complete before a barrier is reached. Rigel enables software management of cache coherence in two ways. One is by providing instructions for explicit cluster cache management that include cache flushes and invalidate operations at the granularity of both the line and the entire cache. Explicit cluster cache flushes update the value at the global cache, but do not modify nor invalidate copies that may be cached by other clusters. The second is broadcast invalidation and broadcast update operations that allow software to implement data synchronization and wakeup operations that rely on invalidation or update-based coherence in conventional cache coherent CMP designs; these operations are discussed further in the evaluation section of the paper.

3.3 Area and Power Estimates

A goal of programmable accelerators is to provide higher performance compared to a general-purpose solution by maximizing compute density. With an initial RTL implementation of the Rigel cluster, we provide an area and power estimate on 45nm technology to understand the impact of our choices on compute density. Our estimates are derived from synthesized Verilog and include SRAM arrays from a memory compiler and IP components for parts of the processor pipeline. For large blocks, such as memory controllers and global cache banks, we use die plot analysis of other 45 nm designs to approximate the area that these components will consume for Rigel. Figure 3 shows a breakdown of prelimi-

Architecture	Power ($\frac{W}{mm^2}$)	Perf. ($\frac{GOPS}{mm^2}$)	Machine Balance ($\frac{GBPS_{PEAK}}{GOPS_{PEAK}}$)
CellBE	.3	1.8	.13
Intel Quad-core	.5	.4	.25
NVIDIA GTX280	.3–.4	3.3	.14
ATI R700	.55–.9	6.4	.1
Rigel	.3	8	.05

Table 1: Power, area, and performance comparison of Rigel to accelerators normalized to 45nm.

nary area estimates for the Rigel design. Cluster caches are 64kB each (for a total of 8MB) and global cache banks are 128kB each (for a total of 4MB) and are constructed from a selection of dual-ported (one-read/one-write) SRAM arrays chosen for optimal area. Cluster logic includes estimates for core area, including FPU, and the cluster cache controller. Other logic includes interconnect switches as well as memory and global cache controller logic. Register files have four read and two write ports and are synthesized from latches. Our initial area estimate totals 266mm^2. For a more conservative estimate, we include a 20% charge for additional area overheads. The resulting area of 320mm^2 is quite feasible 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 core and cluster components, and estimates for interconnect and I/O pin power. A 20% charge for additional power overhead is included. Peak power consumption beyond 100W is possible for Rigel. The figure is similar to contemporary GPUs, such as the GTX8800 from NVIDIA which has a stated power consumption of 150W [16], and CPUs, such as Intel's 8-core Xeon processor which can reach 130W [22].

In Table 1, we compare our initial area and power estimates to those of comparable accelerators scaled to match the process generation of the Rigel implementation. The numbers provided are meant to lend context for our estimates and are subject to parameter variation, such as clock speed. Early estimates indicate that a Rigel design could potentially surpass accelerators such as GPUs in compute density; this is partially due to the lack of specialized graphics hardware. GPUs also spend a substantial portion of their area budget on graphics-related hardware for texture, framebuffer, and raster operations that take considerable area, but do not improve the performance of general-purpose computation. GPUs also incorporate high levels of multithreading which increase utilization, but reduce peak compute throughput. Rigel recovers this area and puts it towards additional compute and cache resources. As expected, Rigel and other accelerators hold a significant advantage in compute density compared to general-purpose CPUs, such as those from Intel [22] and Sun [30].

The Rigel core area estimates are comparable to those of other simple core designs. Tensilica cores with 8kB SRAM scaled to 45nm are .06-.18 mm^2 [29], approximating a cluster area of .5 to 1.6 mm^2. Higher performance MIPS soft-cores consume .42mm^2 scaled to 45nm, and if used to build

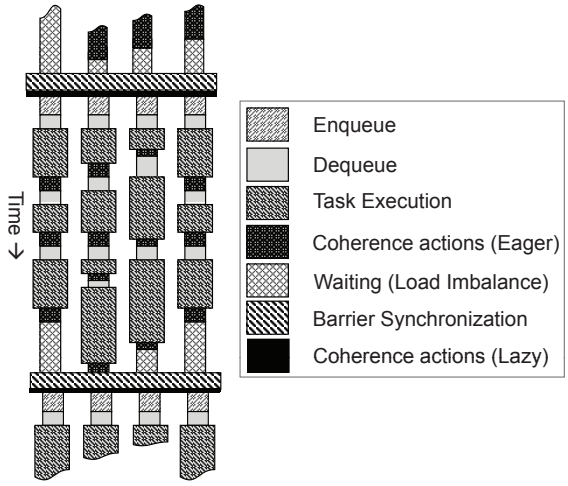

Figure 4: Rigel Task Model execution.

8-core clusters, would approximately occupy 3.5mm^2 [19]. Neither match the features of a Rigel core exactly, but both contain additional features that are not required such as debug ports, MMU components, and peripheral bus interfaces. Frequency and area depend on many parameters, including enabled features, synthesis, process, and cell libraries.

4. RIGEL PROGRAMMING MODEL

In this section we describe the task-based API used by programmers, the *Rigel Task Model*, and how it maps to the Rigel architecture.

4.1 The Rigel Task Model API

The software API of the Rigel Task Model is how the applications developer accesses the LPI. The API is composed of basic operations for (1) managing the resources of queues located in memory, (2) inserting and removing units of work at those queues, and (3) intrinsics, such as atomic primitives, that are supported by the LPI. Applications are written for the Rigel Task Model using a SPMD/MIMD execution paradigm where all cores share a single address space and application binary, but arbitrary control flow among threads of execution from each core is allowed.

The programmer defines parallel work units, which we refer to as *tasks*, that are inserted and removed from queues between barrier operations. We refer to the period between two barriers as an *interval* of computation. The barriers thus provide a partial ordering of tasks. In the Rigel Task Model, barriers are used to synchronize the execution of all cores using a queue. Barriers also define a point at which all locally-cached non-private data modified after the last barrier will be made globally visible using invalidate and flush operations. From the programmer's perspective, tasks that are inserted between two barriers should not be assumed to be ordered and any inter-barrier write-shared data between barriers must be explicitly managed by the programmer. Figure 4 shows the actions as they would occur during an interval.

4.1.1 Queue Management

The Rigel Task Model provides the following set of basic API calls: `TQ_Create`, `TQ_EnqueueGroup`, `TQ_Dequeue`, `TQ_Enqueue`. Each queue has a logical queue ID associated

with it. `TQ_Create` is called once for each queue generated in the system. The call to `TQ_Create` allocates resources for the queue and makes it available to all other cores in the system. Once a queue has been created, any core in the system can enqueue tasks on the queue or can attempt to dequeue tasks from the queue. Each basic enqueue and dequeue action operates on a single task descriptor. The `TQ_EnqueueGroup` operation provides a single operation to enqueue a DO-ALL-style parallel loop similar to the loop operation available in Kumar et al. [14]. We extend the model adding the notion of a task group to encompass locality.

An initialized queue can be in one of four states: `full`, `tasks-available`, `empty`, and `completed`. Any initialized task queue without available tasks but without all cores blocking on dequeue, will be in the `empty` state. Attempts to dequeue to an `empty` queue block. Enqueuing tasks transitions the state of the queue from `empty` to `tasks-available`. When tasks are available, dequeue operations return tasks without blocking. If cores are blocking on the task queue during a transition to `tasks-available`, newly available tasks are allocated and cores become unblocked. If the queue exceed its defined size limit, the queue becomes `full` and any enqueue operation returns notifying the core attempting the enqueue of the queue's `full` status.

The `completed` state is used to provide an implicit barrier in the Rigel Task Model and requires special consideration. When all cores participating in an interval have finished executing all tasks, they will all begin blocking on the task queue and the task queue will transition to the `completed` state. When the `completed` state is reached, a barrier is executed and all cores are returned a notification that a barrier has been reached. The semantics of the `completed` state allow work to be generated between barriers and work creation is not constrained to only occur at the start of an interval. An example of where this may be useful is in the traversal of a tree structure where sibling subtrees can be processed in parallel, but the number of tasks generated is not known a priori.

4.1.2 Scheduling and Locality

Each task is tracked by a *task descriptor*. We define a *task group* as a set of tasks that are guaranteed by the Rigel Task Model to execute on a single Rigel cluster. The shared cluster cache enables low-cost fine-grained communication amongst the tasks within a task group; a task group can be thought of as logically executing on a coherent eight-processor SMP. The number of tasks in a task group can be tuned by the programmer. Enforcing concurrent execution of tasks within a task group is possible using cluster-level barrier operations inserted by the programmer.

As part of the API, we provide performance-minded programmers with mechanisms to manage locality and work allocation. We provide an API call to partition work statically among all cores and have them execute tasks in a data-parallel fashion. The goal of the data-parallel mode is to keep the same API, but to allow the LPI to take advantage of application characteristics to reduce task management costs known statically, e.g., that an application is highly regular and does not benefit from dynamic scheduling. Other API calls allow the programmer to adjust the granularity at which blocks of tasks are fetched from the various levels of queue. The hierarchy is otherwise transparent to the programmer.

4.1.3 Atomic Primitives

Atomic primitives are used extensively in the implementation of the Rigel Task Model runtime where mutual exclusion of accesses to the shared queues must be maintained. The primitives are also exposed to the programmer as part of the Rigel Task Model API. The primitives can be used to implement data structures that require shared updates during an interval. Operations for atomic increment, decrement, integer addition, and exchange are available to the programmer using intrinsics in the code. These operations are perfromed at the global cache and are thus atomic with respect to other global memory and atomic accesses issued from all cores across the chip. A global broadcast operation is provided that allows for one core to update the value of a word cached in any cluster cache and at the global cache. The broadcast reduces the traffic requirement for supporting polling by allowing cores to poll locally at the cluster cache and receive a new value from the broadcast when it becomes available thus avoiding the need to use global memory operations to implement polling. Primitives for load-link and store-conditional are provided by the cluster cache for implementing low-latency synchronization at the cluster. Cluster-level atomic operations are not kept coherent across clusters.

4.1.4 Implementation

The Rigel Task Model is presented to the programmer as a monolithic global queue, but is implemented with the Rigel LPI using hierarchical task queues [4]. The hierarchy of local and global queues reduces contention and latency compared to a single global queue implementation as well as the load imbalance that would be a concern if private task queues alone were adopted. Queues are resident in memory and can be cached in the global cache when existing in the global task queue and at the cluster cache when in the local task queue. Atomic add operations are used to enqueue and dequeue tasks. Task descriptors are inserted into the global task queue after being made globally visible, i.e., by either being flushed to the global cache or by using global stores to avoid caching in local caches. As an optimization, the programmer can specify local enqueing for reduced task overhead at the cost of potential load imbalance. The barrier mechanism is integrated as part of the task queue system by using the same multi-level hierarchy used by the queues to track completion status.

While task queues in general are not novel, the implementation of task queues on Rigel presents a relevant case study in the co-optimization of hardware mechanisms (broadcast and atomic primitives), a runtime system, and software techniques for applications targeting SPMD hierarchical accelerators. The value of atomic operations and the broadcast mechanism are evaluated in Section 5 to demonstrate the benefit of our minimalist approach to hardware support for task management on an accelerator such as Rigel.

5. EVALUATION AND ANALYSIS

In this section we evaluate design choices made for the Rigel architecture using a set of accelerator benchmarks representing a variety of compute, communication, and memory access patterns. We demonstrate scaling up to 1024 cores using software-managed work distribution and synchronization while maintaining a conventional programming model.

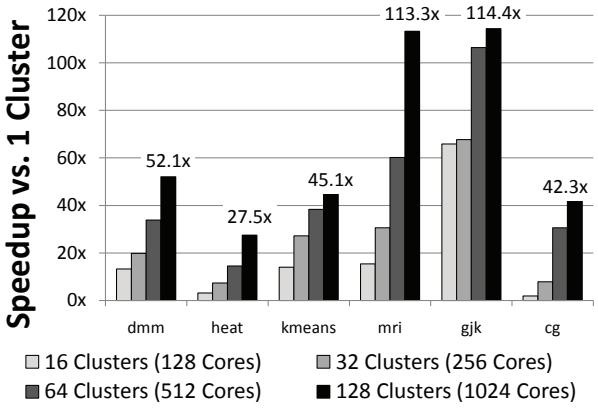

Figure 5: Speedup of one, two, four, and eight (1024 core) tile configurations over execution on a single 8-core Rigel cluster. There are sixteen clusters (128 cores) per tile.

In particular, we find that our task-based work queueing system can support dequeue and enqueue operations in tens to a few hundred cycles on average, but the scalability of most workloads is robust with increased task overhead and that minimizing these overheads does not always represent the best tradeoff for performance. We demonstrate that atomic operations that complete at the global caches, as opposed to at the cores, and a broadcast notification mechanism for supporting fast barriers to be useful in supporting fine-grained irregular applications.

5.1 Benchmark Overview and Scalability

Optimized versions of benchmarks targeting the Rigel Task Model API are used throughout our evaluation. The list of benchmarks includes: A conjugate gradient linear solver (`cg`) performed on sparse matrices from the Harwell-Boeing collection of dimension 4884 (147,631 non-zero elements) representing a real-word physical system; GJK collision detection (`gjk`) for a scene consisting of 512 randomly-generated convex polyhedra of varying size and complexity; An iterative 2D stencil computation that performs a heat transfer simulation (`heat`) on a 4096x512 element grid; A computer vision kernel, k-means clustering (`kmeans`) performed on 16k element 18-dimensional data sets; A 1024x1024 blocked dense-matrix multiply (`dmm`); And a medical image reconstruction kernel (`mri`) derived from the work of Stone et al. [28]. All simulations are executed for one to five billion instructions after initialization. The results are from an execution-driven timing model of the design described in Section 3, including the network, caches, memory controllers, and a GDDR4 DRAM model.

Figure 5 shows the scaling of the Rigel benchmark suite for 128- to 1024-core configurations relative to execution on an optimistically configured single cluster of eight cores. The baseline has global cache and memory bandwidth equal to that of a full tile of 128 cores. Otherwise, global cache and memory bandwidth scale with the number of tiles. A cluster of eight cores can achieve roughly 16 GFLOPS (peak) at 1.2 GHz in 2 mm^2 at 45 nm. For comparison, a single core from a contemporary quad-core Intel i7 processor [12] can support 25 single-precision GFLOPS (peak) using 4-wide SIMD units at roughly ten times the area and three times the clock frequency. Therefore, our scalability numbers demonstrate Rigel's ability to perform one to two orders of magnitude better than a conventional multicore with similar power and area constraints.

Figure 6 displays a histogram of global cache traffic, i.e., requests from the clusters. The figures show that each benchmark makes use of the caching system in a distinct manner. Visualizing global cache traffic patterns helps in understanding the tradeoffs in our design and has suggested optimization techniques. As an example, the figure for `gjk` shows one interval, initially dominated by loading data, then becoming limited by global task queue interactions. Using task prefetching from the global queue and a mechanism for enqueing directly into the local queue, we were able to achieve scaling and avoid the pathology demonstrated in the figure. Due to space limitations, we select a subset of our workloads, `cg`, `kmeans`, and `dmm`, that exemplify patterns found across all of our benchmarks and explain how they map to the Rigel architecture to illustrate Rigel as a programmable accelerator.

5.1.1 Conjugate Gradient Linear Solver

The `cg` benchmark has an interesting memory access pattern for accelerator platforms due to frequent global barriers and reduction operations. The algorithm comprises a sparse-matrix vector multiply (SMVM), constituting 85% of the sequential execution time, followed by simple vector-vector scaled additions and dot products separated by barriers.

Each element in the large, read-only data array is accessed only once per iteration while performing the SMVM. Although the matrix is accessed in a streaming fashion, the vectors that are generated each iteration can be shared by cores within a cluster to amortize read latency and increase effective cluster cache size through sharing. The ability to efficiently exchange the vector modifications each iteration through the global cache, shown in Figure 6 by the periodic writebacks from each iteration, significant performance benefit. However, the vectors conflict with the read-once sparse matrix in the global cache. We make use of a prefetch operation in the Rigel ISA that allows for data to bypass the global cache thus avoiding pollution for touch-once data not shared across clusters.

We find that after managing locality and maximizing memory bandwidth utilization, the ultra-fine granularity of tasks in `cg` and the small number of tasks between barriers stresses the LPI's ability to not only dequeue tasks, but also enqueue work fast enough to keep up with the rate at which tasks complete, and can limit scalability. We find that trading off a small degree of load imbalance for coarser task granularity in `cg` reduces the rate of task management operations translating into reduced contention and better enqueue efficiency (fewer operations enqueue a greater amount of work) resulting in higher achieved performance for the benchmark; A similar pattern is found in `gjk`.

5.1.2 K-Means Clustering

The `kmeans` benchmark iteratively converges on the set of k bins in an n-dimensional space that minimize the aggregate error in the selection of mapping N points in n-dimensions to the k bins. The distributed reduction operation of `kmeans` exploits our ability to perform efficient atomic operations at the global caches, interleaving updates to the shared histograms with other compute, instead of performing a global

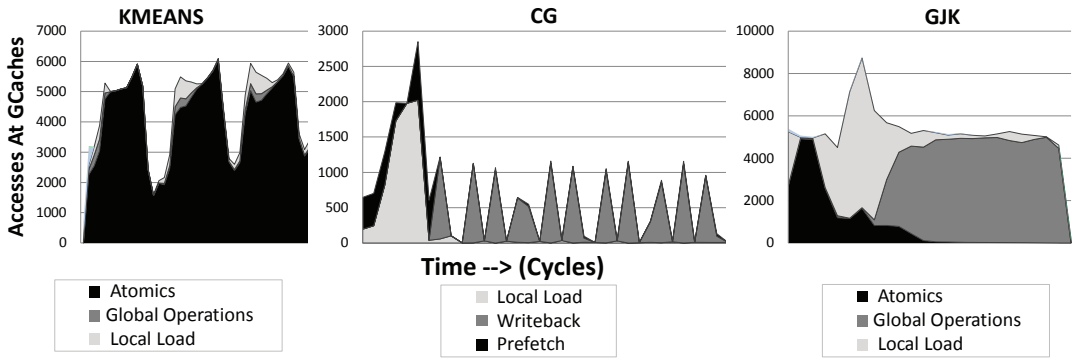

Figure 6: Three global cache traffic patterns from: `kmeans`, where atomic operations dominate the global cache traffic due to a histogram optimization; `cg`, where writebacks of shared data dominate, but do not saturate the global cache bandwidth; and `gjk`, showing high levels of load balancing resulting in global task queue actions dominating global cache traffic. With tuning we have found most of this overhead can be removed.

reduction at the end of the parallel section. The `kmeans` benchmark makes heavy use of atomic operations to remove a reduction at the end of each parallel region and the pattern is clear from Figure 6. However, due to the high arithmetic intensity of the benchmark and high reuse in cluster caches, the increased global cache traffic does not adversely impact performance. In fact, even with atomics dominating global cache traffic, the tradeoff provides better performance at 1024 cores compared to our experiments with using a global reduction at the end of an interval instead of atomics at the end of each task. The benchmark also makes use of software prefetching for object data and the position of the current set of bins used for computing the next set. The set of bins is also read-shared by the tasks that execute on a single cluster. Grouping of multiple points to execute as part of a task group, i.e., mapped to a single cluster, and combined to make tasks comprising multiple points is found to be beneficial from a cache management standpoint as well.

5.1.3 Dense-Matrix Multiply

The dense-matrix multiply benchmark (`dmm`) has a very regular data-access pattern with high arithmetic intensity. The value of dense matrix multiply is that it demonstrates Rigel's ability to maximize its effective use of prefetching, cache management through blocking, read sharing at the cluster cache using task group formation, staging of data in the global cache using a combination of prefetching and added synchronization. Most importantly, `dmm` shows that we can support applications amenable to static partitioning efficiently, though do not make use of the dynamic mechanisms provided by the LPI.

5.2 Rigel Task Model Evaluation

The cost of dequeue and enqueue operations limits the minimum exploitable task length. Similarly, barrier operations can be viewed as an overhead limiting the minimum amount of parallel work that can be done between two global synchronization points. The load imbalance represents the ability of the LPI implementation, in both hardware and software, to compensate for task length variability and the impact of dynamic system behavior.

Task length is a parameter that can be adjusted by the programmer. The choice of task granularity is constrained from below by the minimum size parallel unit of work inherently supported by the application and the marginal cost of generating, scheduling, and synchronizing a task. The choice to increase task length is constrained from above by load imbalance brought on by task length variance and by the lack of parallelism should too few tasks be available to saturate the processor. Table 2 shows statistics related to task length used for our study. The task lengths are generally tens to hundreds of thousands of cycles across all benchmarks, but can vary greatly from hundreds to millions of cycles (even within a single benchmark). The variety of task lengths supports the choice of dynamic task allocation and MIMD execution model. The ability to support 1000-way parallelism with tasks that are thousands of cycles supports our choice of software task management with limited hardware support.

	Task Length (1000 Cycles)		
Benchmark	Mean	Max	Min
dmm	24	511	4.4
heat	124	454	54
kmeans	98	173	41
mri	1607	1664	1585
gjk	23	58	2.6
cg	17	41	0.38

Table 2: Task length statistics (1024 cores)

The high average task lengths, as is found in `mri` and `heat`, lead to infrequent barriers and increased barrier wake up costs due to the code and data responsible for exiting the barrier and beginning the next interval often being evicted from the caches during task execution. The length of tasks relative to these costs makes the effect on overall performance minimal. We measure barrier wake up as the number of cycles, on average, between the last core entering a barrier until the other cores begin executing after the barrier. Load imbalance is measured as the average number of cycles from when one core performs a dequeue operation with the task queue being empty until the last core enters the barrier. The cost of load imbalance and barrier synchronization for each core during one interval of computation are plotted in Figure 7 as a fraction of a task since the fixed cost is not borne by each marginal task.

Enqueue and dequeue overheads are a function of parameters chosen by the runtime and/or the programmer for the Rigel Task Model. The minimum per-task cost of an enqueue and a dequeue can be as little as 44 and 66 cycles, respectively, while the actual overheads that we find pro-

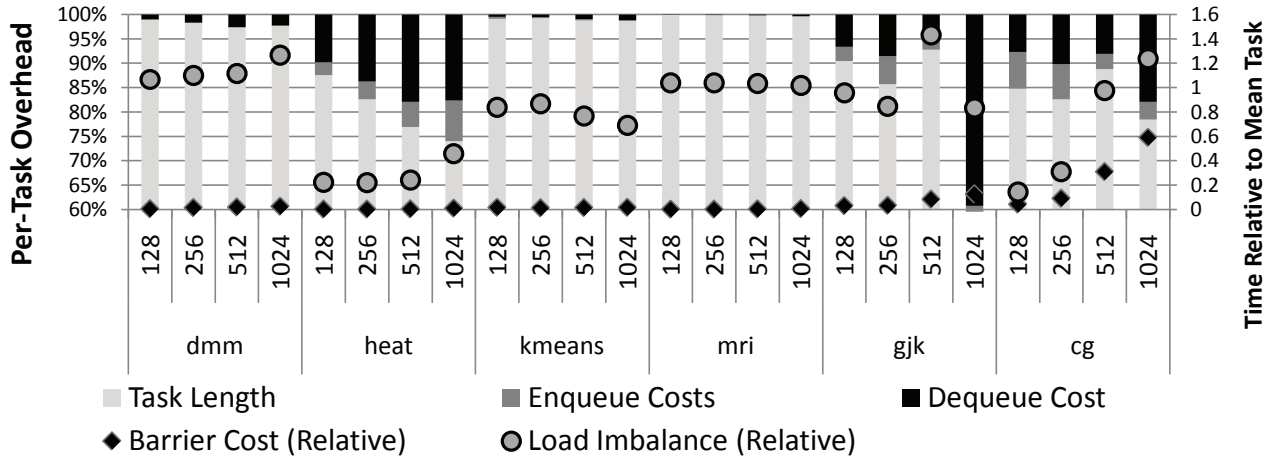

Figure 7: The per-task cost of task execution, barriers, enqueue, and dequeue operation are plotted as a percentage of the average task length in cycles.

vide the best tradeoff for performance and load balancing are generally higher by a factor of three to ten. Nonetheless, as Figure 7 shows, the overhead of barriers, enqueue, and dequeue as a percentage of task length is not more than a few percent for all runs other than `cg`. The `cg` benchmark is sensitive to task management overhead due to the short and irregular nature of its tasks, limiting the enqueing cores' ability to distribute work as fast as the dequeuing cores can consume it. The contention for cluster-level task queue locks and access to the cluster caches exacerbates the problem.

Figure 7 also shows that without explicit support for global synchronization, Rigel is able to achieve low load imbalance and implement efficient barriers in software. Our experience tuning both the runtime and benchmarks has demonstrated that minimizing the absolute costs of task overheads often leads to lower overall performance. The lower performance is due to load imbalance and the inability of the runtime to perform task queue optimizations such as pre-loading of tasks into the local task queues and intelligent prefetching from the global task queues.

5.3 Accelerator Primitives

We examine two mechanisms provided by the Rigel architecture to support the LPI described in Section 2. In our work porting applications to Rigel, it has become apparent that components of the LPI can benefit greatly from hardware acceleration in the form of primitives exposed to low-level systems programmers via the ISA. Figure 8 shows the use of two such operations that are particularly helpful in supporting the task distribution mechanism and synchronization elements of the LPI: broadcast update and global atomic operations.

The broadcast update operation replaces a function served by invalidate messages in multi-processors with hardware cache coherence. The `bcast.update` instruction reduces the contention for the barrier completion flag. Without a broadcast mechanism, the value of the flag would be polled at the global cache by the cores already in the barrier. When the last core enters the barrier, it flips the sense of the global barrier thus allowing other cores polling on that value to proceed past the barrier. On Rigel, instead of writing the value with a global store, the last-entering core issues a `bcast.update` with the new global sense. The data is sent from the core to the global cache where it is updated and then a notification message is sent to all clusters via the cache interconnect. Each router in the interconnect replicates the update notification message to all cores it is connected to. The broadcast operations may also be used to provide fine-grained sharing between barriers and to support other operations served by invalidate messages in conventional coherence schemes; however, such studies are left to future work.

We find that atomic operations, integer accumulate in particular, are useful in implementing efficient algorithms for work allocation and scheduling, histogramming, reductions, barriers, and other synchronization mechanisms. The exposed serial latency is minimized by executing atomic operations at the global caches instead of at the core. The overhead is a small amount of area for functional units and some added complexity at each global cache bank. A conventional cache coherent design would require cores issuing atomic operations to obtain ownership of a line, then to perform the modification, and if there are other cores waiting to access that line, the line must then be sent to another core.

Coherence is used to guarantee atomicity in conventional multiprocessors at the cost of exposing the latency of highly-contended values across the chip. Rigel guarantees atomicity by executing the operation at the point of coherence (the global cache on Rigel). Thus multiple read-modify-write operations can be pipelined through the network instead of being serialized by cache coherence state transitions. Atomics that execute at the global cache also reduce the number of network messages for each atomic operation issued, thus minimizing the latency experienced by a single core and the contention at the global caches and interconnect.

Figure 8 shows the results of our study for benchmarks most affected by the broadcasts and global atomics; results are normalized to the baseline configuration where both are enabled. The relative benefit of broadcast updates is evaluated by replacing them with polling loops that use global loads/stores to perform the polling/updates at the global cache. Performance gained from implementing atomics that execute at the global cache is shown by modifying the im-

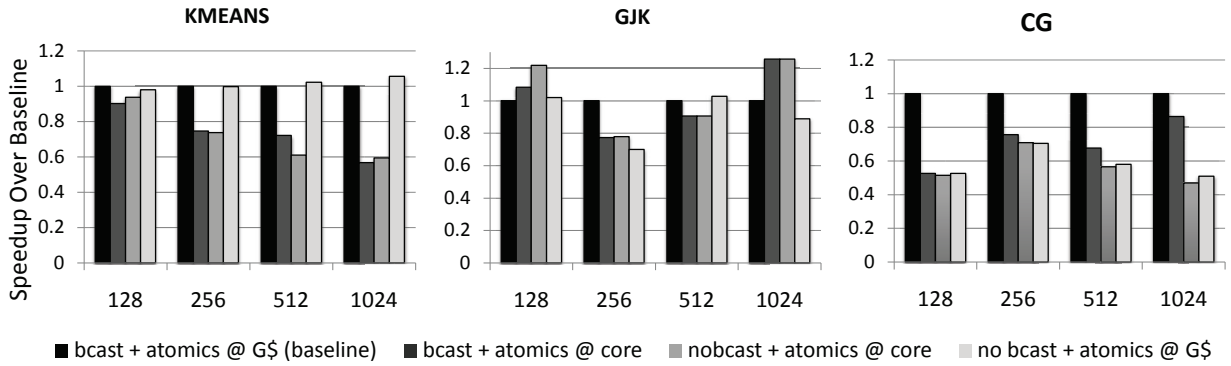

Figure 8: Execution time relative to using both broadcast-update instructions for barriers and operations that complete at the global cache (baseline).

plementation of all atomics to lock values on read from the global cache, perform the update locally at the core, and then store that value back to the global cache and unlock the value. The implementation is similar to atomic operations implemented with full-empty bits on the HEP multiprocessor [26]. For coarse-grained data-parallel workloads with few reductions and long intervals between barriers, less than 1% performance difference is seen and those results are omitted for brevity.

For `kmeans` we find that without fast atomic support, performance is degraded by as much as 40% with the difference from the baseline being more pronounced as the number of cores scales. `cg` is most affected by the primitives as it does a large number of barriers and has a large number of interactions with global state due to the finer granularity of tasks in the benchmark. The `gjk` benchmark provides a counter intuitive result: with accelerator hardware lower performance is observed. The reduced performance is due to the flood of messages generated by the broadcast operations and the stampeding-herd effect a broadcast wakeup can generate. The latter is an interesting effect observed in some situations where all cores begin requesting the same data or code from the global cache in a short period of time. The correlation of accesses results in heavily loaded areas of the cache interconnect, global cache, and memory controller buffers. Staggering accesses either from the natural throttling effects of polling on the global caches or from programmer intervention, tend to spread out the traffic yielding higher performance.

6. CONCLUSION

In this paper we motivate, describe, and evaluate a programmable accelerator architecture targeting a broad class of data- and task-parallel computation. The Rigel Architecture consists of 1000+ hierarchically-organized cores that use a fine-grained single-program, multiple-data (SPMD) execution model. The concept of a low-level programming model is described in the context of a set of elements that we have found to be requirements for supporting scalable programmable accelerator applications on our architecture. These elements include the execution, memory, and work distribution models as well as synchronization and locality management.

Our initial findings and evaluation show that, ultimately, Rigel can achieve a compute density of over 8 single-precision $\frac{GFLOPS}{mm^2}$ in 45nm with a more flexible programming interface compared to conventional accelerators. Rigel thus offers promise for higher computational efficiency on a broader class of applications. In support of scalability beyond 1000 cores, we demonstrate experimental results and analysis of several applications ported to the low-level programming interface for Rigel, the Rigel Task Model. We examine scalability issues with work distribution, locality management, synchronization, load-balancing, and other overheads associated with massively parallel execution. We find that it is important to support fast task enqueue and dequeue operations and barriers, and both can be implemented with a minimalist approach to specialized hardware.

7. ACKNOWLEDGEMENTS

The authors acknowledge the support of the Focus Center for Circuit & System Solutions (C2S2 and GSRC), two of the five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program. The authors thank the Trusted ILLIAC Center at the Information Trust Institute for their contribution of use of their computing cluster. The authors also wish to thank Wen-mei Hwu, the IMPACT Team, and the anonymous referees for their input and feedback. John Kelm was partially supported by a fellowship from ATI/AMD.

8. REFERENCES

[1] J. Balfour and W. J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *ICS'06*, 2006.

[2] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11), 1989.

[3] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *ISCA'96*, 1996.

[4] S. P. Dandamudi and P. S. P. Cheng. A hierarchical task queue organization for shared-memory multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(1), 1995.

[5] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande. Poster session – N-body simulation on GPUs. In *SC'06*, 2006.

[6] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC'06*, 2006.

[7] K. Fatahalian and M. Houston. GPUs: a closer look. *Queue*, 6(2):18–28, 2008.

[8] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar: a large scale multiprocessor. *SIGARCH Comput. Archit. News*, 11(1):7–11, 1983.

[9] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer. In *ISCA'82*, 1982.

[10] M. Gschwind. Chip multiprocessing and the Cell broadband engine. In *CF'06*, pages 1–8, New York, NY, USA, 2006.

[11] J. Guo, G. Bikshandi, D. Hoeflinger, G. Almasi, B. Fraguela, M. Garzaran, D. Padua, and C. von Praun. Hierarchically tiled arrays for parallelism and locality. In *Parallel and Distributed Processing Symposium*, April 2006.

[12] Intel. Intel microprocessor export compliance metrics, Februrary 2009.

[13] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8), 2003.

[14] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA'07*, pages 162–173, New York, NY, USA, 2007.

[15] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the connection machine CM-5. *J. Parallel Distrib. Comput.*, 33(2), 1996.

[16] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2), 2008.

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

[18] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[19] MIPS. *MIPS32 24K Family of Synthesizable Processor Cores*, 2009.

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

[21] J. D. Owens, D. Luebke, N. Govindaraju, mark 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.

[22] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, R. V. J. Chang, M. Ratta, and S. Kottapalli. A 45nm 8-core enterprise Xeon processor. In *ISSCC'09*, Februrary 2009.

[23] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *MICRO'06*, 2006.

[24] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS'96*, pages 26–36, 1996.

[25] 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(3):1–15, 2008.

[26] B. Smith. The architecture of HEP. In *On Parallel MIMD computation*, pages 41–55. Massachusetts Institute of Technology, 1985.

[27] 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.

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

[29] Tensilica. *570T Static-Superscalar CPU Core PRODUCT BRIEF*, 2007.

[30] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In *ISSCC'08*, Feb. 2008.

[31] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.