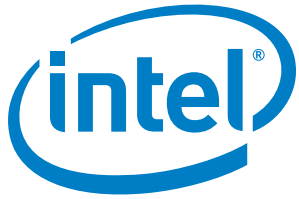

Atomic Read Modify Write Primitives for I/O Devices

White Paper

Atomic Read Modify Write Primitives

Abstract

New I/O usage models have emerged recently. There is a trend towards offloading compute-intensive applications to specialized engines/accelerators. Many such applications today are in the high performance computing domain, examples of such are financial options modeling, seismic exploration, game physics, and bio-informatics. This paper illustrates the need for synchronization primitives for I/O accelerators to help these emerging usage models. It shows examples from real-world applications and the associated performance benefits of such primitives.

Abhishek Singhal
Rob Van der Wijngaart
Peter Barry
Intel Corporation

August 2008

Table of Contents

Introduction

In recent years there has been a trend towards offloading computing-intensive workloads to specialized I/O devices (example: math acceleration and physics acceleration). Two notable changes have contributed to this trend. First, the accelerators have benefited from Moore's Law which has given rise to extremely powerful engines which are increasingly multi-core/multi-thread/multi-card devices. Second, the accelerators are becoming increasingly programmable. Another trend emerging today is that of a collaborative execution-model where by both CPU and accelerator resources are combined to achieve minimum execution time (for example, double precision units, branch prediction hardware).

Many such applications today are in high performance computing domain for example, options modeling, seismic exploration, and game physics. These applications and associated libraries such as the Intel® Math Kernel Library (Intel® MKL) have traditionally been run in multi-threaded environments (over multiple cores/CPUs) and hence make extensive use of synchronization primitives such as semaphores, mutexes, and barriers. The challenges involved in porting existing applications to accelerators are reduced if existing algorithms and their associated interactions could be relocated to the accelerator(s). Because of the prevalence of synchronization primitives like semaphores, barriers, etc. in these algorithms, it is natural that these primitives be extended to the accelerators. In this paper, we examine some use cases for such primitives. In section two we define what Read Modify Write (RMW) primitives are. In section three we examine the usage of I/O device side RMWs for traditional (single-threaded) I/O devices and in section four we show their usage in multi-accelerator environments.

2. What are Read Modify Writes?

Read Modify Write (RMW) operations are hardware-assisted operations that atomically update a variable at its memory location. These operations have a long history and their usage in a wide range of synchronization algorithms is well published. The operations are required for efficient implementation of synchronization primitives such as semaphores, mutexes, and barriers. Examples of RMWs that we have referred to in this paper are:

1. **Compare and Exchange (addr, value1, value2):** Read the value at *addr* and compare it with *value1*, write *value2* to *addr* if *value1* is equal to the value obtained at *addr.*
2. **Atomic_Add (addr, value):** Atomically increment (or decrement) the variable at memory location *addr.*

In this paper we have examined the need and benefit of having these primitives available on the I/O device(s).

3. Synchronization Primitives for Traditional CPU-I/O Device[1] Interactions

In this section we examine the use of RMWs for traditional I/O devices. One can partition the interaction between host/CPU and I/O device into two categories:

1. Bulk data transfer
2. Control information exchange

In the next two sections we examine the key characteristics of these exchanges and also what role RMWs play in each of these categories.

3.1. Bulk Data Transfer

Bulk data transfer can be further divided into two subcategories based on direction of transfer. The first subcategory is bulk data transfer from Host/CPU to the I/O device. This primarily depends on the bandwidth available to the I/O device. Since a large proportion of algorithms being executed on I/O devices are data-parallel, programmers have been able to hide I/O device data access latencies with great success through techniques such as double buffering. A programmer can improve performance of this data flow by a number of optimization techniques such as pipelining data transfers, maximizing transfer sizes to achieve higher link efficiencies, and by placing data structures to improve memory page efficiency.

For the other direction of bulk-data transfer (I/O device to Host/CPU), data access latency, and not bandwidth alone, is critical to performance. The code executing on the host is dominated by control (branches) and thus has very high sensitivity to latency for data accesses. In applications where the I/O device serves as the producer of data that would be consumed by the Host/CPU (example: where accelerator collaborates with Host to utilize its double precision units) the access to system-memory can be a performance limitation.

Another example is that of high-speed network links where small network packets arrive at about the rate of memory access time. Host/CPU caches have been traditionally utilized to improve the performance of CPU latency. Another paper *(Merits of Data Reuse Hints)* examines the extension of CPU caches for use by accelerators to improve performance of this data flow. I/O device side RMWs are focused on improving synchronization rather than improving raw bandwidth or latency, and are thus not critical to performance of bulk data transfer.

3.1.1 Control Flow Between Host/CPU and I/O Device

Control between Host/CPU and the I/O device is required for a wide range of uses such as:

- Setting up application parameters
- Initiating computation on the I/O device. Often generically known as "Do Work"
- Checking on the status bits on the accelerator
- Sending done signal to the Host/CPU, that is, *Work complete indications*
- To indicate error conditions from the I/O device to the Host/CPU

In traditional-I/O usage scenarios the existing mechanisms such as Memory Mapped I/O (MMIO) writes (doorbells), interrupts, and polling, etc. are utilized to exchange control information and facilitate synchronization between the Host/CPU and I/O device.

These mechanisms typically provide for a pre-defined number of discrete communications mechanisms between the I/O device and the CPU complex. For example a network interface card might be designed to have a separate descriptor ring per CPU core to facilitate TCP/IP flow to processor core affinity. The trend to facilitate direct use of an I/O device by user space applications brings with it scalability concerns with the existing mechanisms, as the product designer must predefine the number of resources dedicated to Host/CPU communications. Some platform architectures provide architecture enhancements such as Monitors, but these capabilities are also typically limited in availability.

However, I/O device side RMWs address some of the limits in scaling mentioned above and also allow development of new usage models which can improve the Host CPU-I/O interaction. For example, having RMWs available to the I/O device facilitates the creation of lock-free queues which are shared between I/O devices and the CPU(s). This mechanism provides a scalable solution whereby each CPU thread can submit work to an I/O device using a shared queue resident in system memory. The lock-free queue can be read by the I/O device without the need for costly bus-locks *(Optimized Lock-Free FIFO Queue, Fober et. al).*

It is important to recognize that the proposed RMW primitives equip I/O devices with the capabilities hitherto available only to CPU/Host. This capability is a basic requirement for an environment where algorithms can be easily (and in a cost-effective manner) migrated between the CPU and I/O domain.

4.Synchronization Primitives in Multi-threaded Environments

Next we examine the use of I/O-side RMW semantics in multi-threaded I/O environments.

4.1 Multiple Accelerators Accessing Host Defined Work Queue

In our first example we examine the utilization of work queues in a multi-threaded environment. Work queues are a common way of extracting coarse-grain parallelism in a dynamic environment where load balancing cannot be done a priori, or where it is not efficient to do so. A good example is the implementation of FLAME numerical linear algebra library (*Extracting SMP Parallelism for Dense Linear Algebra Algorithms from High-Level Specifications,* Tze Meng Low, Robert A. van de Geijn, Field G. Van Zee). Even though the sizes of tasks to be executed change continually, the load remains balanced as workers (accelerators) acquire new work once they have finished a task. Synchronization is required to make sure no task gets executed more than once, and that no starvation occurs. This is accomplished using RMW operations on shared variables in system memory, as illustrated in the example work-queue implementation below.

Consider the case where multiple accelerators (or accelerator threads) N_a are executing tasks (N_t) of various sizes (in cycles). A master thread on Host/CPU acting as a *producer,* fills a queue of tasks that is maintained as a (circular) array of size L ($L < N_t$) in system memory. It ensures new tasks are added to the end of the queue, and that no existing tasks are overwritten prematurely. To this end it maintains the *producer_index* (see code below) variable that points to the next entry in the task array to be filled, and checks the status of the task in that entry to make sure that it has been completed before overwriting it with a new one. The accelerators *(consumers)* acquire tasks strictly in the order they were generated. L must equal at least N_a to allow full concurrency of consumers. The situation is illustrated in figure 1, which depicts an intermediate state of the work queue.

4.1.1 Non-RMW-based Implementation

Without RMW the producer and consumers have to communicate strictly through interrupts, as the producer coordinates access to shared-data structure. Each consumer has a driver which communicates with the producer master driver which ensures atomicity of operations in system memory. The consumers submit their *ready to execute another task* status by sending an interrupt to the Host. This is logged in the device driver. The master driver arbitrates between multiple *requests for task,* chooses a winner, and signals the winner to start processing. This model carries the overhead of interrupt based communication.

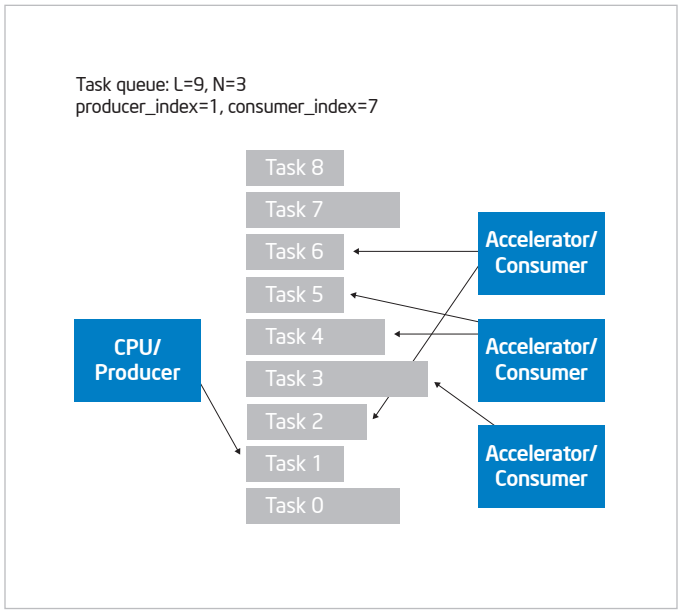

Figure 1a. Shared work-queue example.

```
Producer (CPU)
  while (task [producer_index].status ≠ completed){ };
  task[producer_index].status = busy;
  create task[producer_index];
  task[producer_index].status = ready;
  producer_index++; (periodic)

                                                  atomic RMW
Consumer
  while (task_being_acquired ≠ false) { };
  task_being_acquired = true;
  while (task[consumer_index].status ≠ ready){ };
  task[consumer_index].status = busy;
  index = acquire task [consumer_index];
  consumer_index++; (periodic)
  task_being_acquired = false;
  execute task[index];
  task[index].status = completed;
```

Figure 1b. Pseudo-code for work-queue sharing.

4.1.2 Atomic RMW-based Implementation

With RMWs available to I/O devices, producer and consumers can read and update shared variables atomically in system memory, thus avoiding the need for interrupts. Some of the coordination can be handled by the I/O devices without involving the producer, which increases concurrency (see figure 1a). Acquisition order of tasks is enforced by the shared *consumer_index* variable that indicates the next available task in the queue. Idle consumers spin on the shared *task_being_acquired* variable for the right to access the next task in the queue *(consumer_index).* Once a consumer wins access rights, it waits for the producer to finish creating the task by spinning on its shared *status* variable. Next, it changes the status so that the producer will not overwrite its contents before the task is completed and increments (modulo *L*) the *consumer_index.* Then it releases the *task_being_acquired* variable, which implements a critical section among the multiple consumers. The individual task status variables implement critical sections among the actively acquiring consumer and the producer. Execution of tasks takes place outside all critical sections. Figure 1b shows the pseudo-code for the RMW implementation for producer and consumers. Each shaded box is implemented using a single atomic RMW primitive.

4.1.3 Performance Analysis

We use the following definitions and assumptions. All cycles refer to CPU cycles. Average time to acquire and execute a task is C_e cycles, latency from accelerator to LLC is C_{LLC} cycles, time an interrupt takes to travel to the device driver plus time to handle the interrupt is C_i cycles, mutex variables (RMW) remain in Host cache, each task acquisition takes four non-pipelined cache accesses (RMW), or one interrupt (no RMW). Because we focus on synchronization, we ignore the time to create a task or to generate an interrupt.

We study two different scenarios.

1. All tasks are the same size, and are executed by the accelerators in lockstep, or the tasks are all very small. In that case cache accesses to *task_being_acquired* (RMW) cannot be overlapped with computations by other accelerators. The one cache access to a non-globally shared variable that occurs within the critical section guarded by *task_being_acquired* also cannot be overlapped with computations or cache accesses by other accelerators. Hence, three accesses per task are exposed (serialized), while one can be overlapped. In the non-RMW case, interrupts are all fully exposed.

2. Tasks are of different sizes and sufficiently large that they lead to staggered cache accesses of synchronization variables (RMW), or to stagger interrupts (no RMW). In that case, interrupts or cache accesses related to synchronization variables can occur concurrently with computations by other accelerators.

We compute the time C_{tot} to execute all tasks for both scenarios, with and without RMW.

1. *RMW:* $C_{tot} = N_t*((C_e+C_{LLC})/N_a + 3*C_{LLC})$
   No RMW: $C_{tot} = N_t*(C_e/N_a + C_i)$

2. *RMW:* $C_{tot} = N_t*(C_e+4*C_{LLC})/N_a$
   No RMW: $C_{tot} = N_t*(C_e + C_i)/N_a$

We plot the RMW performance benefit P_G, expressed as the ratio of completion times in both scenarios, for the case that C_{LLC}~300, C_i ~4300, as a function of the task size. The results are shown in figure 2.

Obviously, the performance benefit of RMW operations versus interrupt-based synchronization for both scenarios is largest for small task sizes, and becomes negligible for tasks exceeding a quarter million cycles. For task sizes in the range of a few hundred to a few thousand cycles (very fine grain), RMW benefit is more than 100%.

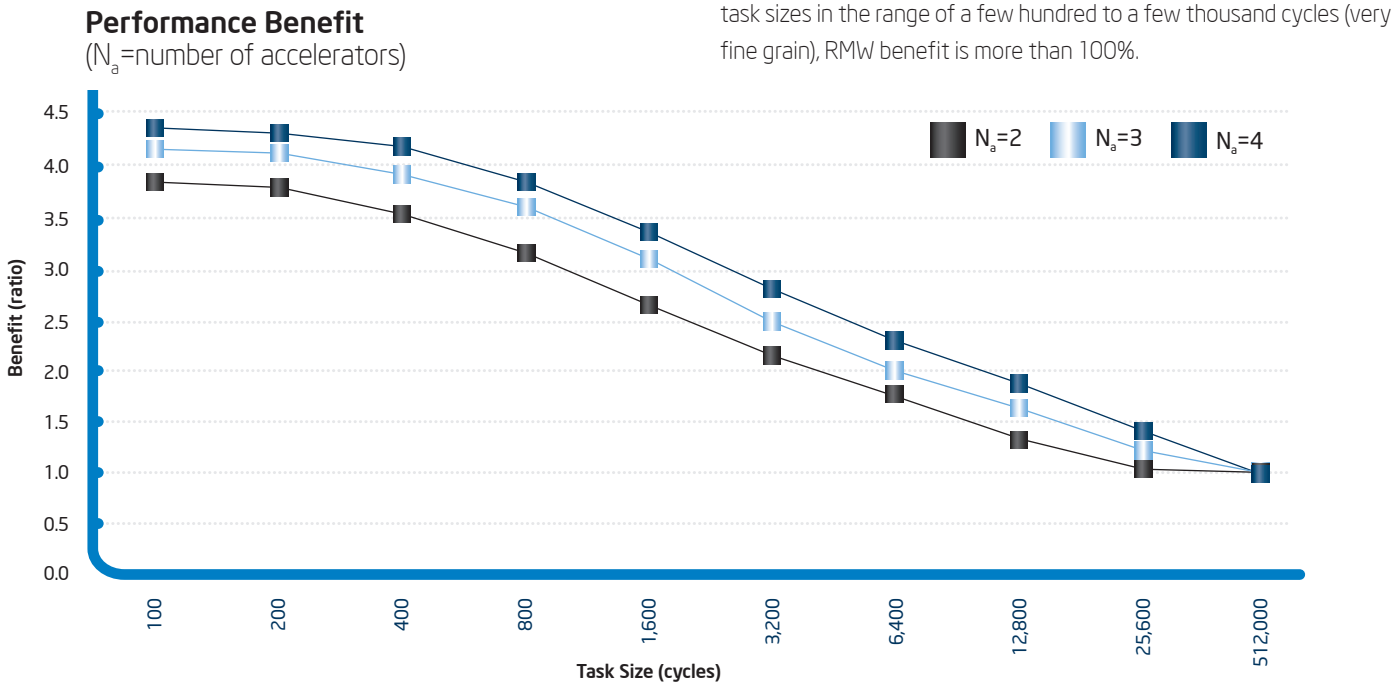

Figure 2. Performance benefit of RMWs for shared work queue.
Source: Intel Corporation.

4.2 Multiple Accelerators Synchronizing with Host

Next we examine a use-case where multiple accelerators are working collaboratively with the host. Our example illustrates the collaborative usage of CPU and accelerator resources through barriers. Barriers, which find very high usage in traditional multi-threaded application, provide a mechanism by which no process can go beyond the synchronization point until all process have reached the barrier. The barrier transaction has three distinct phases, arrival, wait for release, and release of the barrier. There are a number of published variations of barriers such as Centralized Barrier, Software Combining Trees, and Tree Barrier with local spinning. A simple and prevalent example implementation of the barrier is shown below.

```
Barrier Init:
        Set counter value to 0; Set lock to 0;
Barrier:
        Acquire(lock); [Uses SWAP atomic] - RMW
                if counter is zero set flag to zero.
                increment counter

        Release(lock) ; [ Uses SWAP atomic] - RMW

        if counter equals number of waiting processes
                set counter to zero
                set flag to one
        else
                repeat:
                        until flag is one
        end if
```

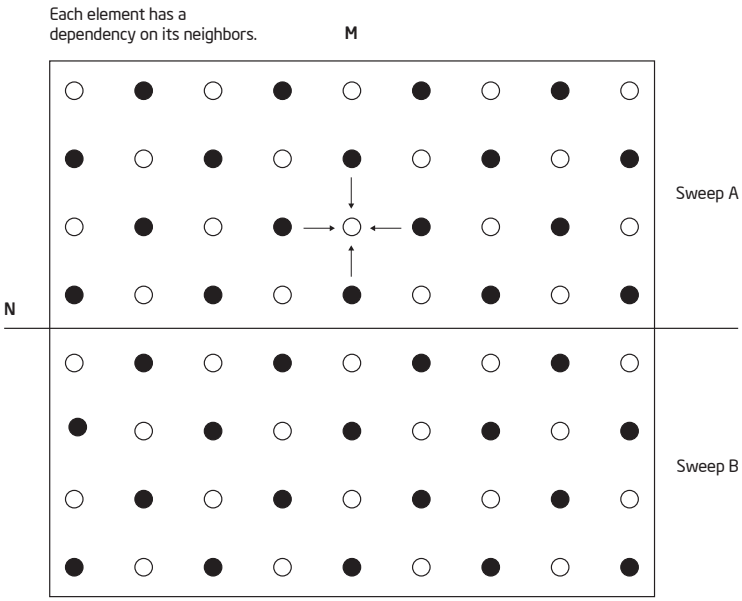

Figure 3. Data partitioning for Gauss-Seidel Algorithm.

An example application, which may make use of barriers to synchronize between worker threads, is the parallel execution of Gauss-Seidel equation solver. The Gauss-Seidel is an iterative solver that continues to run until it converges to a solution within a target error range.

The algorithm consists of dependencies between surrounding elements within the matrix. In order to solve elements of the algorithm in parallel, data dependent groups are grouped a number of ways. The first data partitioning is known as red-black. This allows the algorithm to be separated into two separate phases which do not have data dependences. The second method is to divide the array into separate sweeps that may occur in parallel. The data points adjacent to each sweep must not be calculated once Sweep A and B have completed.

Figure 4 shows an example association of accelerator worker threads to the data partitioned matrix. There are a number of global synchronization points introduced which can be implemented as barriers. A barrier is required at the end of processing between each sweep to synchronous processing of the elements adjacent to each sweep.

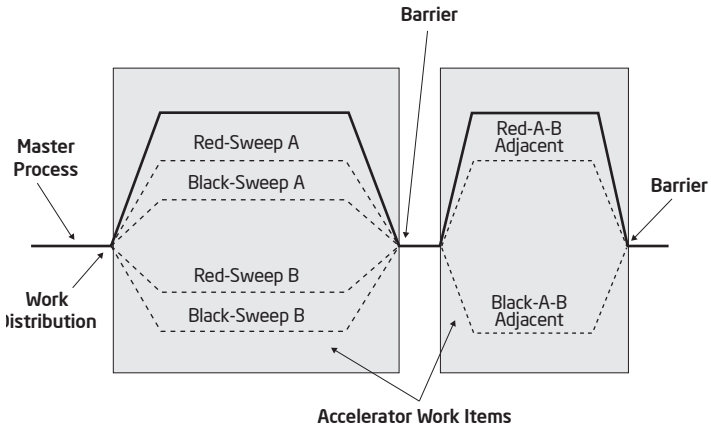

Figure 4. Example thread partition for Gauss-Seidel algorithm.

4.2.1 Non-RMW-Based Implementation

In the traditional (without I/O-side RMWs), the accelerators must signal the IA processor when each of the work tasks complete. This signaling is typically an interrupt raised by the accelerator. A device drive interrupt handler would execute and decrement the barrier counter on behalf of the accelerator. Atomicity of updates is achieved by the shared interrupt-queue. When the counter reaches zero the barrier is synchronized and the master thread that is waiting on the barrier is released. As the reader would note interrupts (alternatively bus-locks) present a substantial overhead.

4.2.2 RMW-Based Implementation

With RMW primitives accelerators can replace the interrupts with atomic increments the shared variable in the system memory (or the system cache if appropriate hardware is available). The next section shows performance benefits achievable as a consequence. It also highlights the nature of applications that will benefit from these primitives. The potential usage of RMWs is highlighted in the grayed section of the code reference above. Essentially, the barrier-count variable can be updated in-place by the use of RMW primitives.

4.2.3 Performance Analysis

The graph below shows the benefit of using RMW operations for our use case. It is apparent from the graph below that main savings come from replacing costly interrupts with much faster RMW operations. It is also to be noted that the benefits are mostly for applications which have small problem sizes that are offloaded to accelerators.

Conclusion

We have examined several application models in this paper. We believe that to best service the emerging application domain of multiple-accelerators and multi-threaded applications which span both Host and I/O devices, I/O-side RMWs are a key element. These primitives not only provide ease of programming but also by over-coming the limitation of legacy CPU-I/O mechanisms (locks, interrupts) they provide a significant performance boost. For some representative example parallel workloads (work-queues) we showed that RMW operations can give performance benefits of more than a factor of two compared to interrupt-based synchronizations. The gains are typically significant in use when there is need for fine-grained synchronization in presence of multi-threaded accelerators.

Contact Information:

Abhishek Singhal	abhishek.singhal@intel.com
Peter Barry	peter.barry@intel.com
Rob Van der Wijngaart	rob.f.van.der.wijngaart@intel.com

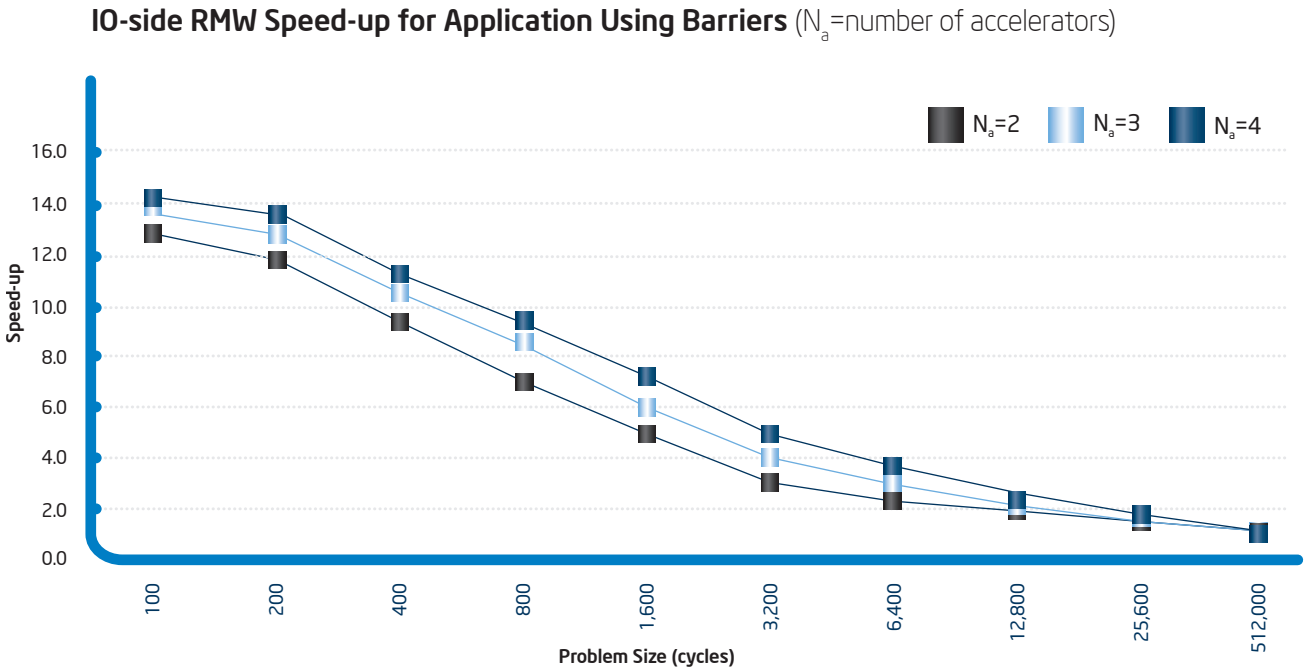

Figure 5. Performance benefit of RMWs for applications with barriers.
Source: Intel Corporation.

Engage with Intel on Next Generation PCI Express* (PCIe*) product development, visit www.intel.com/technology/pciexpress/devnet for more information.

1. In this paper we define traditional CPU-Device model to be one which has one I/O thread. this could have one or more CPU threads.

Printed in USA 0808/VP/HBD/PDF ♻Please Recycle