Performance Evaluation and Optimization of Random Memory Access on Multicores with High Productivity

Vaibhav Saxena, Yogish Sabharwal
IBM Research - India, New Delhi,
{*vaibhavsaxena,ysabharwal*}*@in.ibm.com*

Pramod Bhatotia
Max Planck Institute for Software Systems, Germany,
bhatotia@mpi-sws.org

Abstract

The slow progress in memory access latencies in comparison to CPU speeds has resulted in memory accesses dominating code performance. While architectural enhancements have benefited applications with data locality and sequential access, random memory access still remains a cause for concern. Several benchmarks have been proposed to evaluate the random memory access performance on multicore architectures. However, the performance evaluation models used by the existing benchmarks do not fully capture the varying types of random access behaviour arising in practical applications.

In this paper, we propose a new model for evaluating the performance of random memory access that better captures the random access behaviour demonstrated by applications in practice. We use our model to evaluate the performance of two popular multicore architectures, the Cell and the GPU. We also suggest novel optimizations on these architectures that significantly boost the performance for random accesses in comparison to conventional architectures. Performance improvements on these architectures typically come at the cost of reduced productivity considering the extra programming effort involved. To address this problem, we propose libraries that incorporate these optimizations and provide innovatively designed programming interfaces that can be used by the applications to achieve good performance without loss of productivity.

1. Introduction

Memory access latencies have been unable to keep up with the fast pace at which CPU speeds have been increasing. This growing disparity of speed between the CPU and the memory has resulted in memory accesses dominating code performance. Commonly referred to as the "Memory Wall," this has become a challenging problem in the HPC domain. Several architectural enhancements such as bigger caches, wider line sizes, complex pre-fetch and cache replacement policies tend to improve the performance of applications with data locality and sequential access. However, these enhancements are of no use or may even have a negative impact on the performance of applications that access the memory in random fashion.

Applications with random access such as histogram update, ray tracing, forward and backward projections in image reconstruction may be categorized based on several factors:
– *Operations performed on the random data:* While applications such as histogram update and back-projection for image reconstruction involve update of memory locations, other applications such as ray-tracing and forward-projection for image reconstruction involve only random reads.
– *Random stream generation:* In multi-core environments, applications can also be categorized based on the way the random stream of updates is generated. On the one hand, in some applications such as ray tracing, forward and backward projection for image reconstruction, the random updates/addresses can be programmatically generated in parallel; on the other hand, in applications such as histogram update, the update stream is determined from some centralized I/O device or file system.
– *Error tolerance:* In some applications, for instance in histogram update of word counts in documents, it is not critical for every update to be accurately accounted. However, in other applications such as projections in medical imaging reconstruction, errors may not be tolerable.

There are several benchmarks to evaluate random memory access performance on multicore architectures, such as the HPC Challenge RandomAccess benchmark [6]. However, the performance evaluation models used by existing benchmarks do not fully capture the varying characteristics of random memory access applications as described in the above categorizations. We propose a new model to evaluate the performance of such applications that models the characteristics of random memory access behavior arising in practical applications.

We use our model to evaluate the performance of two popular architectures, the Cell [5] and the GPU [9]. These two architectures have contrasting approaches for dealing with the "Memory Wall" issue. On the one hand, the Cell processor features DMA based memory access coupled with a user-managed local store that encourages overlapping of memory accesses with computations thereby, hiding the memory access latencies. The list-based DMAs in Cell allow

978-1-4244-8520-8/10/$26.00

for scatter/gather operations that are useful when accessing memory randomly. On the other hand, GPUs feature an architecture designed for parallel throughput. In this case, the "average" memory access cost is reduced due to sheer parallelism using a large number of processing cores and low-overhead scheduling of light-weight threads that amortizes the memory access costs. We suggest novel optimizations for improving the performance of random access applications on both architectures and evaluate the improvement with these optimizations using our proposed model.

While these architectures offer impressive performance, this comes at the cost of reduced productivity. For obtaining the desired performance, considerable effort is required in programming and optimizing for these architectures. To handle this, we propose libraries that incorporate these optimizations and provide innovatively designed programming interfaces that can be used by the applications to derive better performance without loss of productivity.

Our contributions are summarized as follows:

• *Performance evaluation model.* We propose a new model to evaluate the performance of random access applications. In this model, a large table resides in the system memory. A stream of random numbers is generated. The stream may be generated from a single processing unit (*streaming mode*) or the stream may be divided into multiple sub-streams, each generated on a separate processing unit (*parallel mode*). For each random number that is generated, a table offset is suitably extracted from the random number and the corresponding entry is fetched from the table. Two different kinds of operations may be applied to the entry. In the *read-only mode*, an operation is performed on the entry using the random number and accumulated into a running variable. In the *update mode*, an operation is performed on the entry using the random number and the resulting value is then updated back in the table. This leads to four different combinations – *streaming read-only mode*, *streaming update mode*, *parallel read-only mode* and *parallel update mode*. Moreover, depending on whether errors can be tolerated or not, the *parallel update mode* and *streaming update mode* may be divided into two more subcases - *error-tolerant* and *error-intolerant* leading to 6 combinations in all. These combinations are illustrated in Figure 1.

• *Increased performance.* We suggest novel optimizations to improve the performance of random memory accesses on the Cell and the GPU using the proposed model. For the Cell processor, we propose a multi-buffering scheme (generalization of the double buffering technique) for error-tolerant applications that groups the updates. For error-intolerant applications, we propose a data partition based locking scheme that does not exhibit too much loss of performance in comparison to the error-tolerant case. For the GPU, we study the performance of random accesses using NVIDIA CUDA [9] and suggest techniques using which the threads can co-operate to boost performance for large updates even

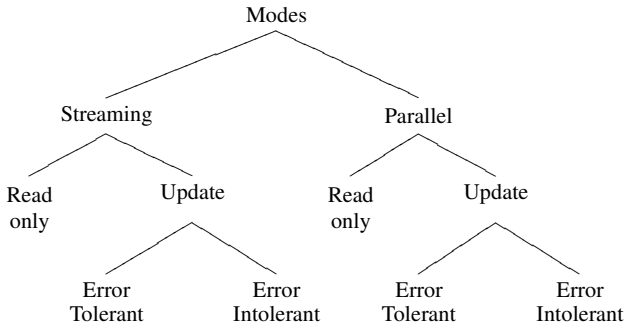


Figure 1. Different modes of operation

though they are working on independent update streams. This is achieved using the on-chip shared memory. Our analysis of the performance on these specialized architectures demonstrates that they offer considerable performance boost in comparison to conventional architectures. We find that while the GPU is more suitable when there are multiple update streams being generated from different threads, the Cell performs better when there is a single update stream generated from the PPU.

• *Increased productivity.* We present innovatively designed libraries that incorporate these optimizations and provide generic interfaces that can be used across a large number of applications thereby enhancing programmer productivity, reducing software development effort and cost. Our libraries are similar in spirit to the software cache library on the Cell. However, while the software cache library is advantageous for applications that exhibit locality of reference and/or data reuse, our library is targeted at applications that exhibit random memory access behavior.

Many applications exhibiting random memory access behavior have been optimized on the Cell and GPU [3], [11], [12]. However most of these optimizations are application specific. Balart et al. [2] and Chen et al. [4] propose runtime libraries combined with compile time code transformations to optimize irregular memory accesses in the applications. Our optimizations for the RandomAccess benchmark achieve performance improvement of 66% in comparison to that reported by Balart et al. [2].

The rest of this paper is organized as follows. Section 2 provides a brief overview of our model for measuring random memory access performance based on the HPC Challenge RandomAccess benchmark. Our optimizations for the Cell and GPU architectures are described in Sections 3 and 4 respectively. In Section 5, we evaluate and analyze the performance of our optimizations on these architectures. Finally Section 6 concludes the paper.

2. Proposed performance evaluation model

As discussed in the previous section, existing benchmarks for random memory access fall short of capturing the

characteristics of practical applications that display random access behavior. In this section, we describe a new model for studying random accesses to the system memory that fills the gap by capturing the varying characteristics displayed by random access applications in practice. Our model is adapted from the HPC Challenge RandomAccess benchmark [6]. We briefly describe this benchmark and then present our proposed extensions.

The HPC Challenge RandomAccess benchmark measures the capability of the memory subsystem while performing random updates to the system memory. It has received considerable attention [7], [1], [10] and has been optimized on many architectures. In the single node version of the Randomaccess benchmark, there is a large table T of size 2^k, called the *update table*, that occupies approximately half the memory. The processor generates a pseudo-random sequence of 64 bit integers. For each random number (say a_i), the least significant k bits are selected to index into the table T. The selected entry in the table is updated using a bit-wise *xor* with the random number a_i. The stream of random numbers is generated iteratively, with each random number generated from the previous one, using the primitive polynomial over the Galois Field of order 2 (GF(2)). The number of such updates performed is four times the table size. To capitalize on multi-threading and multicore features, the benchmark allows for the pseudo-random sequence to be generated and applied in parts (sub-streams), with different threads/cores generating and applying the updates for the different parts. The benchmark specifications allow for look-ahead and storage of at most 1024 updates before they are applied to the table. The performance of the system is measured by the number of *giga updates per second* (GUPS) performed by the system. The benchmark allows 1% errors in updates. This allows for inconsistencies that may arise when performing the updates non-atomically.

The shortcoming of the conventional random-access benchmark is that it only supports the *parallel update mode with error tolerance*. We extend the benchmark in several ways to model the characteristics of random memory accesses that arise in practical applications.

– *Read-only mode.* In this mode, the value from the table entry is retrieved and XORed with the value of a running variable. The table entry is not updated.

– *Streaming mode.* In this mode, the random updates are generated on the processor/core that is capable of performing I/O, file operations or interactions with other nodes in a distributed memory system. The updates are then streamed to different processors/cores to perform the update or accumulate operations.

– *Error Tolerance.* For the *error-intolerant* mode, we modify the verification process to check that there are no errors. The existing verification process to check for at most 1% errors is used for the *error-tolerant* mode.

– *Varying sizes.* We also vary the update sizes to study the memory access performance for different lengths. The update sizes are varied from 8 to 128 bytes in powers of 2.

In Sections 3 and 4, we discuss our optimizations for random memory accesses under this new model on the Cell and the GPU respectively.

3. Optimization of Random Access on Cell

The Cell processor combines a conventional high-power PowerPC core (PPE) with eight simple SIMD cores, called Synergistic Processing Elements (SPEs) in a heterogeneous multi-core offering. It offers extremely high compute-power on a single chip combined with a power-efficient software-controlled memory hierarchy. Each SPE has 256KB of user-managed Local Store (LS) for code and data. An SPE explicitly issues Direct Memory Access (DMA) requests to transfer data between the off-chip main memory and its local store. Access to the external memory is handled via a 25.6 GB/s XDR (DDR2 for PowerXCell 8i) memory controller. The PPE, eight SPEs, DRAM controller and I/O controllers are connected via the high bandwidth Element Interconnect Bus (EIB) [8].

We now discuss our design choices for optimizing random memory accesses on the Cell.

3.1. Parallel mode

We design an optimized SPE library to support random access updates targetted for applications that operate in parallel mode wherein the SPEs generate the sequence of addresses and update them. The library incorporates optimizations to perform the updates efficiently. Our discussions will first focus on the optimizations for the *update mode* with *error intolerance*. We will then discuss the modifications for handling *read-only mode* and *error-tolerance*.

The Optimized Library. In designing a random access library that allows error-free updates, locking mechanisms need to be used to synchronize across the SPEs. Locking is very costly on the Cell processor as it is performed using DMA operations in the main memory. Associating locks with individual updates is very expensive as the locking operations have to be performed for every udpate. Moreover, this does not allow for grouping of updates and use of DMA lists. On the other hand, associating a lock with the table results in serialization effects across the SPEs as the lock can only be with one SPE at a time. In order to address this, we partition the update table into k partitions and associate a lock with each partition. The partitions are determined using the higher order bits of the table index. The SPEs maintain update buffers (array of pending addresses and values) of size m for each partition. The SPEs perform DMA operations on the group of updates stored in the update buffers (m updates at a time) using DMA lists. This results in efficient bandwidth utilization as memory latencies

are amortized over more updates. The SPEs also employ double-buffering to overlap the update computations with the memory read/write (DMA) operations.

The most important method offered by the SPE library is *r_update(u64Int addr, const void* uvalue)* where *addr* is the address of the table entry to be updated and *uvalue* points to the value to be used for the update. This method is invoked for every random access update that is to be performed. The pseudocode for this method is shown in Algorithm 1. Whenever the *r_update* method is invoked, the update is placed into one of the buffers based on the higher order bits of the table index that determines the partition. When some buffer becomes full (contains m pending updates), the corresponding partition is locked and a DMA read operation is initiated to get the coresponding values of the table from the main memory into the local store. When the DMA read completes, the updates are applied to the fetched values and then a DMA write is initiated. Once the DMA write completes, the lock is released on the partition. These operations are performed asynchronously and the library operates on the data for the partitions independently. It keeps track of the DMAs that are in progress and does not wait for the DMA to complete. Instead it returns back to the SPE application. Note that the library does not ensure that the update is completed on return. This allows library flexibility to perform updates in batches and perform the DMA operations efficiently. Moreover, the library requires that this method be invoked frequently as the library may be holding certain locks when control is returned to the user. The library frequently checks for DMA completion when the *r_update* method is invoked. On completion of the DMA operation for some partition, it proceeds to perform the subsequent operation (performing updates and initiating a DMA write in case a DMA read has completed or freeing up the buffer in case a DMA write has completed). When the application anticipates that it requires control for a long period of time and will not be invoking the *r_update* method frequently or when all the updates have been performed, it may invoke the *r_flush()* method. This method is provided to commit all the pending updates and release all the locks.

We use 2 buffers for every partition for double buffering. The number of updates in each buffer is restricted to m such that $2 \cdot m \cdot k \cdot P \leq \mathcal{L}$ where k is the number of partitions, P is the number of SPEs and $\mathcal{L}$ is the look-ahead limit.

While adding updates to the buffers, a *deduplication* process needs to be performed in order to collate updates to the same memory location. It can be very expensive to walk through the buffer list to search for duplicates. We use a hashing based mechanism to reduce the comparisons to be performed. We associate a bitmap of 256 bits with each buffer. We hash an update address to a bit of the the bitmap (simply by looking at 8 lower order bits). Before dispatching a DMA read for a buffer, we scan the list of updates in the buffer and update the corresponding bit in the bitmap. Before doing so, we check if the bit is already set. If so, there may be a duplicate for this update and we need to scan against the other updates in the buffer. If not, we avoid any scanning. With the number of updates per buffer being much smaller than 256, we avoid substantial number of comparisons. The Algorithm 1 includes this *deduplication* process.

For the *error-tolerant* case, the library simply disables the locking mechanism. However, updates are still separated into different buffers based on the address bits. This allows the SPE to work on more independent streams of updates in a multi-threaded manner. This approach is similar to *Software-Hyperthreading* mentioned in [3]. The *read-only mode* is also handled similarly. We omit the interface details due to space limitations.

The Optimized Benchmark. We now discuss how the benchmark is optimized using the optimized library for random access described above. Recall that in the *parallel mode*, the random sequence generation is distributed across the SPEs. The PPE acts as the master – it initializes the *Update Table* and distributes the RandomAccess workload to the SPEs, which act as workers. The work partitioning is done by dividing the stream of updates to be performed into substreams, as many as the number of SPEs and each SPE generates the pseudorandom subsequence for its own substream. The SPE thereafter use the optimized SPE library to perform the actual updates on its substream. These steps are shown in Algorithm 2.

3.2. Streaming mode

In the *streaming mode*, the PPE generates the updates and passes them to the SPEs for applying them to the *table*.

The Optimized Library. For this case, we design a PPE based library. The library internally uses the SPEs in order to optimize the random accesses to the memory. It does this by creating buffers containing groups of updates for each SPE to process. The PPE maintains w buffers for each of the P SPEs. Each of these buffers can store m updates. On invokation of the *r_update* method on the PPE, the library fills up buffers for the SPEs. When any of the buffer fills up, it sends a mailbox message to the corresponding SPE. An SPE on receiving a mailbox message, initiates a DMA get to retrieve the updates from the corresponding buffer. On completion of the DMA get, they send back a mailbox message to the PPE so that it can fill in more updates in the corresponding buffer in the main memory. Once the SPE completes the DMA to fetch the buffer contents, it can process it as in the case of *parallel mode* by reusing the SPE library routines discussed in Section 3.1. The total number of updates in the buffers is $m \cdot w \cdot P$. As there can be an equal number of updates pending with the SPEs at any time, we ensure that $2 \cdot m \cdot w \cdot P \leq \mathcal{L}$ to adhere to the look-ahead limit.

Algorithm 1: SPE update routine *r_update*

```
r_update(u64Int addr, const void* uvalue)
//fill buffer - Index of first buffer of the partition that is currently being filled
//dma buffer - Index of second buffer of the partition that is being used for DMA
Determine the partition of the table, ipartition, this addr belongs to
// Check if this addr is already present in the dma list
Hash the addr to bit b in the bitmap
if bit b is not set then
    Set bit b in the bitmap
    Add the addr to the dma list for fill buffer of ipartition and also copy the update value, uvalue, to the SPE LS buffer
else
    Scan through the current set of addresses in the dma list to see if we already have addr
    if duplicate address found in the dma list then
        Update (e.g. XOR) the existing uvalue in the dma list with incoming uvalue
    else
        Add the addr to the dma list for fill buffer of ipartition and also copy the update value, uvalue, to the SPE LS buffer
    end
end
if fill buffer got full then
    if dma buffer is free (no DMA/compute in progress) then
        Mark the fill buffer to be used for DMA read and swap the fill and dma buffers
    else
        Set the waitflag to wait until dma buffer gets free
    end
end
if dma buffer marked for DMA read then
    Lock the ipartition and initiate the DMA read request for dma buffer
end
/* Frequently check status of in-progress DMA/compute operations for each partition and proceed to the next set of operation for the corresponding partition */
if DMA read is complete then
    update the fetched value and start the DMA write
else if DMA write is complete then
    mark the dma buffer as free
    unlock the ipartition
end
```

Note that streaming mode involves additional DMAs from the SPEs to fetch the buffer of updates which it was generating earlier. However, this does not impact the performance much as these are large contiguous buffers and can be fetched efficiently.

4. Optimization of Random Access on GPU

GPUs are highly parallel, multi-threaded, many-core co-processors to the CPU. A GPU can be modeled as a set of SIMD multiprocessors (SMs) each consisting of a set of scalar processor cores (SPs). The SPs of a multiprocessor execute the same instruction simultaneously but on different data. The GPU has a large high bandwidth device memory with high latency. Each SM also contains a very fast, low-latency on-chip shared memory shared among its SPs. The *CUDA* programming model[9] is amongst the most popular programming models on the GPU. In this programming model, a *host* program runs on the CPU and launches a *kernel* program to be executed on the GPU device in parallel. The kernel executes as a grid of one or more thread blocks, each of which is dynamically scheduled to be executed on a single SM. Each thread block consists of a group of threads that cooperate with each other by synchronizing their execution and efficiently sharing resources on the multiprocessor such as shared memory and registers. Threads within a thread block get executed on a multiprocessor in scheduling units of 32 threads called a *warp*. A *half-warp* is either the first or second half of a warp.

We now discuss out design considerations for optimizing random memory accesses on the GPUs.

Algorithm 2: Optimizing Benchmark on Cell in *parallel update mode*

```
Get the number of updates to be performed per SPE
while SPE has updates for the table do
    Generate the next pseudorandom number in the sequence
    Compute the address, addr, of the table entry to be updated using the random number
    Compute the update value, uvalue, for updating the table entry
    Call r_update(addr, uvalue) for performing update
end
Call r_flush to commit all pending updates and release locks
```

4.1. Parallel mode

In the parallel mode, the random sequence generation is distributed amongst the threads. We start with a discussion of a basic implementation of the benchmark using the GPU. We then discuss our optimizations and libraries incorporating these optimizations.

Basic Implementation. The basic implementation is driven by the host CPU. It allocates and initializes the *update table* in the GPU global memory. Accesses to the table are performed by multiple threads that are created on the GPU by launching a *randomaccess kernel*. The update stream is divided into substreams, as many as the number of threads. Each thread is responsible for handling one of these substreams. For each generated update, a thread performs read or read-modify-write operation on the corresponding update table entry depending on the *mode* of operation. For

update sizes larger than 16 bytes, the threads update the table entry iteratively, working on 16 bytes in each iteration.

The Optimized Library. We now discuss the optimizations incorporated in our library:

(1) Memory Coalescing. Our library enables large contiguous memory read and write operations to be performed collectively by the threads of a half-warp. It is based on a hardware feature provided by the GPUs for higher memory throughput, known as *memory coalescing*. For instance, when threads of a half-warp (group of 16 threads) access consecutive words of an aligned segment in global memory, the GPU hardware is able to coalesce these accesses into a single memory transaction, improving memory bandwidth utilization. The main idea in our library design is for the threads of a half-warp to collaborate and utilize the memory coalescing feature when all of them have to perform large memory accesses. This is feasible since threads in a warp (or half-warp) execute the same stream of instructions.

The library provides an interface for a GPU thread to read 128 bytes of data from a global memory location and place it into the low latency on-chip shared memory available on the GPU multiprocessor. The method *load128(void* shmem, void* glmem, void** saddr)* reads 128 contiguous bytes from the global memory region pointed to by *glmem* and writes it to the shared memory buffer pointed to by *shmem* at an offset of $htid \times 128$ bytes, where *htid* is the index (0-15) of the caller thread within its half-warp. This method is a collective operation and is required to be called by all the threads of the half-warp together. To perform this read efficiently, the threads of a half-warp first save their global memory addresses pointed to by *glmem* to *saddr* in the shared memory. Then all the 16 threads of a half-warp work on these addresses, one at a time. They work iteratively on these addresses, collectively reading 128 bytes from one of the global memory address stored in *saddr* to shared memory buffer *shmem* in each iteration. There are 16 such iterations for a half-warp. While reading each 128 byte value, the 16 threads of a half-warp read consecutive 8 bytes of the value. The benefits of memory coalescing are observed as the threads of a half-warp read consecutive parts of the same 128 byte entry at the same time. This is illustrated in Figure 2(a).

A similar method *store128(void* shmem, void* glmem, void** saddr)* is provided for performing large memory writes collectively. This method reads 128 bytes from the shared memory and writes to the global memory in a similar manner to *load128* described above. The library also provides similar interfaces *load64* and *store64* for 64 bytes. In this case, the 16 threads of a half-warp read/write consecutive 4 bytes of a 64-byte value in the memory.

(2) Reducing Bank Conflicts. The on-chip shared memory on a multiprocessor is banked for achieving high memory bandwidth. Therefore its effective bandwidth is inversely proportional to the number of bank conflicts between the shared memory accesses by threads of a half-warp. When using the shared memory approach described above, each thread of a half-warp on returning from the load method would perform some operations (read, modify, write) on its 128-byte value fetched into the shared memory buffer. However this causes 16-way bank conflict as each thread of a half-warp accesses the same bank resulting in reduced effective bandwidth. To reduce conflicts, we add an extra 8-byte element per 128-byte (or 64-byte) update size in shared memory buffer. A thread with index *tid* now updates its 128-byte value stored at an initial offset of $tid \times (128 + 8)$ bytes instead of $tid \times 128$ bytes, thereby reducing bank conflicts.

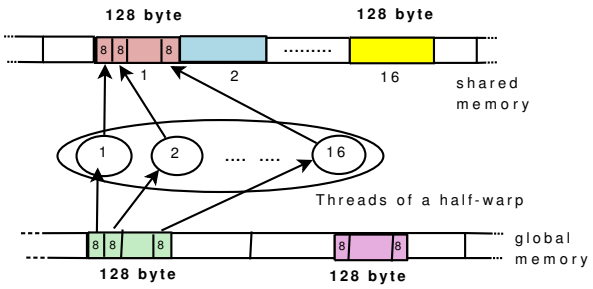
This general purpose library is also useful in many other applications that require access to large contiguous memory.

The Optimized Benchmark. The optimized benchmark uses the library described above to perform large updates efficiently. The threads in a half-warp co-ordinate to collectively perform the updates that they generate. Each thread works on its own substream. Thus the update stream is divided into as many substreams as the number of threads. The threads perform the updates in three steps iteratively. In the first step, each thread within a half-warp generates a random number in its own substream and invokes the optimized library method *load128* to collectively read the updates into shared memory. In the second step as illustrated in Figure 2(b), each thread of a half-warp reads its 128-byte value fetched into the shared memory and updates (XORs) it with a generated 128-byte value. In the third step, the threads of a half-warp store the updated 128-byte values using the *store128* methods collectively, similar to how they read the values. In the *read-only mode*, the value is accumulated in the second step and the third step is not performed.

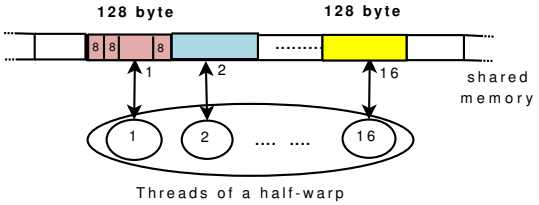
Note that, there are multiple half-warps that are operating in parallel. Therefore the shared memory requirement is proportional to the number of threads in a thread-block.

4.2. Streaming mode

In the *streaming mode*, the host generates the updates and streams them to the GPU for updating the entries in the update table. The host maintains a buffer, *h_AddrBuf*, of size $BSize$ to cater to streaming of the updates to the GPU. It alternately fills up the two halves of the buffer, each of size $BSize/2$. While one of the halves is being processed by the GPU, the other half gets filled with updates by the host. Therefore, the host first generates $BSize/2$ updates to fill one half of the buffer, and then transfers the generated half to the GPU and invokes the *stream randomaccess kernel* to perform updates to the *update table* stored in the device global memory using the optimized library as above. The kernel call is asynchronous and doesn't block the host. The host simultaneously generates the next set of updates, filling the other half of *h_AddrBuf*. Therefore, kernel execution on the GPU is overlapped with update generation on the host.



(a) Co-ordinated loads from the global memory with *load128*



(b) Updates of 128-byte values by threads of a half-warp

Figure 2. Memory loads and updates using shared memory on GPU

This update scheme ensures there are no more than $BSize$ updates pending at anytime. We set $BSize \leq \mathcal{L}$, where $\mathcal{L}$ is the look-ahead limit.

The error-intolerant mode can be handled on modern GPUs by using CUDA atomic functions to implement locks. However, this is not a recommended approach on the GPU as it introduces dependencies amongst the CUDA threads.

Version	Basic	Memory Coalescing	Shared Memory	Reduced Bank Conflicts	Streaming
GPU-RA-BA	√	-	-	-	-
GPU-SRA-BA	√	-	-	-	√
GPU-RA-SM	-	√	√	-	-
GPU-RA-SMB	-	√	√	√	-

Table 2. Optimization versions for GPU

5. Performance Results and Analysis

In this section, we present experimental results to evaluate the performance of the proposed optimizations for random memory accesses in *parallel* and *streaming modes* on the Cell processor and GPU.

Experimental Setup. A summary of specifications of evaluated Cell and GPU systems is shown in Table 1. The Quadro FX 4600 GPU consists of 96 processor cores (SPs). It is organized as set of 12 SMs each consisting of 8 SPs running at 1.19 GHz. It has 768 MB of off-chip device memory providing a peak memory bandwidth of 67.2 GB/s. Each SM has 8192 registers and 16KB of local on-chip *shared memory*, shared between its scalar cores. QS22 IBM BladeCenter consists of two PowerXCell 8i processors running at 3.2 GHz connected in a NUMA configuration. The BladeCenter consists of 4GB DRAM per processor with a total of 8GB main memory. We also used an Intel Xeon processor based system. The system consisted of two dual-core Xeon 5148 processors with a total of 4 cores running at 2.33 GHz. It has 4 GB of main memory. The GCC 3.2.3 compiler (with -O3) was used to compile the source code.

5.1. Discussion and Analysis

Table 2 summarizes different optimizations performed for the GPU along with the optimization parameters. We use the same naming convention for the optimizations when discussing performance results.

The RandomAccess benchmark measures the performance in units of GUPS (Giga-updates per second). As we vary the update size as well, we use a slightly different unit, Giga-bytes updated per second (GBUPS), to measure the random access performance. This is defined as $GUPS \times$ *update-size* and therefore measures the total number of bytes updated per second in *update mode* and total number of bytes read per second in *read-only mode*.

Error-tolerant update mode. This section discusses the performance obtained for the *error-tolerant update mode*.

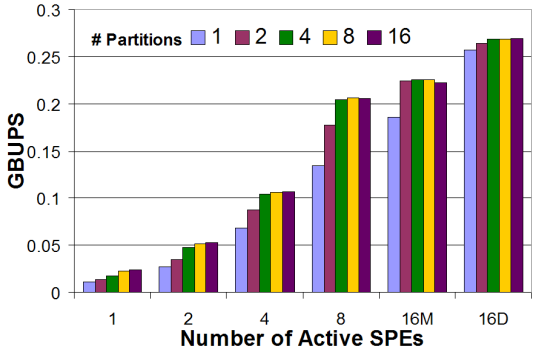


Figure 3. Partitioning approach for Cell (*error tolerant parallel update mode*)

• *Scaling:* Figure 3 shows the performance on Cell for varying number of SPEs and partitions with an update size of 8-bytes. For 8 or fewer SPEs, a single Cell chip of the BladeCenter is used. For more SPEs, we use two approaches of memory allocation - $16M$ represents the case when entire table is allocated from a single memory node. For $16D$ the table allocation is interleaved across the two memory nodes.

We observe that for a fixed number of partitions, the

System	Arch. Type	Clock (GHz)	Sockets	Cores per Socket	Memory b/w (GB/s)	Memory Size	Local Store (KB)	SDK	Compiler
QS22 PowerXCell 8i BladeCenter	multicore SIMD	3.2	2	8(+1)	51.2 25.6/socket	8 GB	256	Cell SDK 3.1	ppu/spu-gcc 4.1.1
Quadro FX 4600	multithread SIMD(SIMT)	1.19	1	96 (12x8)	67.2 (device) 4 (PCIe x16)	768 MB	16 per SM	CUDA 2.1	nvcc 0.2.1221

Table 1. Specification of evaluated systems

performance scales almost linearly with increasing number of SPEs upto 8 SPEs. Performance improves only marginally beyond 8 SPEs, scaling better for the $16D$ case. This is expected as the memory contention becomes high and distributing the table across the nodes results in load-balancing of the requests.

Figure 3 also shows the performance variation with different number of partitions with 8-byte updates. Performance improves with increasing the number of partitions (independent buffers) and nearly peaks when using 4 partitions. Thereafter, the gains are very small. For 8 SPEs, performance with 4 partitions is 52% better compared to 1 partition.

These results highlight that for memory intensive applications, working with multiple buffers independently (in a multi-threading fashion) can significantly improve performance compared to the traditional double buffering approach where operations on the buffers are tightly coupled.

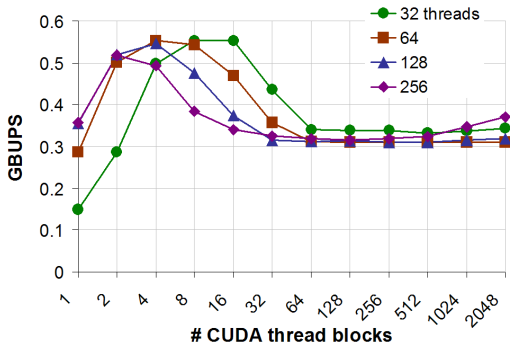


Figure 4. GPU performance scaling for basic implementation (*parallel update mode*)

Figure 4 shows scaling of basic GPU implementation (GPU-RA-BA) performance with 8-byte update size for different number of CUDA threads and thread blocks. We observe that performance peaks when the total number of threads ranges between 256 to 512. Each thread-block gets scheduled on one multiprocessor comprised of 8 processing cores on the GPU. With 64 threads, the best performance is observed with 4 thread-blocks. This only utilizes 32 processing cores – about one third of the available processing cores. No performance gains are observed by increasing the number of thread-blocks (trying to utilize more processing cores). This is expected as more threads lead to high memory contention due to an increased number of requests.

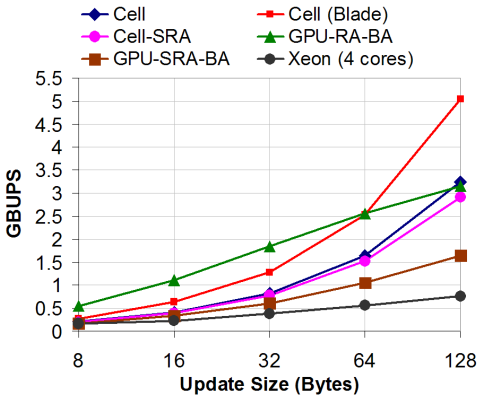


Figure 5. Cell and GPU performance with different update sizes for *error tolerant update mode*

• *Varying Update Size:* Figure 5 shows the performance with varying update sizes for traditional Intel Xeon processor using 4 cores, Cell and GPU. The Intel Xeon performance is taken with 4 threads (1 thread/core). Each core generates its own substream for performing updates. The Xeon performance is much lower than Cell and GPU performance especially with large update sizes.

For Cell, the *parallel update mode* performance scales well with increasing update sizes on both 8 SPEs (Cell) as well as 16 SPEs with interleaved data (Cell(Blade)). The best performance is observed for 128-byte updates, showing an increase of close to 16x over the performance for 8-byte updates. This is because 128 byte updates involve full cacheline transfers between main memory and local store while the number of memory requests remains the same. Smaller updates only transfer a partial cacheline but still consume 128 bytes (cacheline size) worth of bandwidth.

For the basic implementation on the GPU (GPU-RA-BA) with 8-byte updates, performance does not scale well with increasing update sizes. The performance with 128-byte updates is limited to about 5.6 times better than the performance for 8-bytes. This is primarily because memory requests larger than 16 bytes are serviced iteratively. For

64 and 128-byte updates, performance can be improved by using memory coalescing effectively. This is discussed later.

The results for *streaming update mode* (Cell-SRA and GPU-SRA-BA) are also shown in Figure 5. As can be seen, the Cell performance (Cell-SRA) achieves similar performance as *parallel mode* (Cell) and is largely unaffected in the distributed-memory setting. The performance for the GPU (GPU-SRA-BA) on the other hand is much lower than the *parallel mode* (GPU-RA-BA) performance. This can be attributed to the host buffer size (Bsize) being restricted to 1024 (see Section 4.2) in order to adhere to the benchmark limit. By relaxing this constraint and increasing the buffering to 1M (2^{20}) updates, the performance improved to within 80% of the performance in *parallel mode*. This is because the host sends large number (0.5M) of updates in each iteration to the GPU increasing the data transfer rate and the kernel works on larger number of updates in every iteration.

Operation	Update Mode		Read-Only Mode	
Update Size	64 bytes	128 bytes	64 bytes	128 bytes
GPU-RA-BA	2.567	3.146	3.448	4.612
GPU-RA-SM	3.199	4.651	4.854	5.161
GPU-RA-SMB	4.182	7.507	4.87	8.978

Table 3. GPU Performance (in GBUPS)

• *Memory Coalescing:* The performance results for 64 and 128 byte updates with memory coalescing using shared memory are shown in Table 3 for different optimization strategies. Memory coalescing using shared memory (GPU-RA-SM) leads to a factor 1.5 improvement in performance over the basic version (GPU-RA-BA) for 128-byte update size. However, with the optimizations related to avoiding bank conflict in shared memory (GPU-RA-SMB), performance further improves by about 61% over GPU-RA-SM. This results in an overall factor of 2.4 improvement with GPU-RA-SMB over GPU-RA-BA. Usage of shared memory for this optimization limits the number of threads per thread block that can be used on the GPU. This suggests that having larger shared memory on the GPUs will be beneficial.

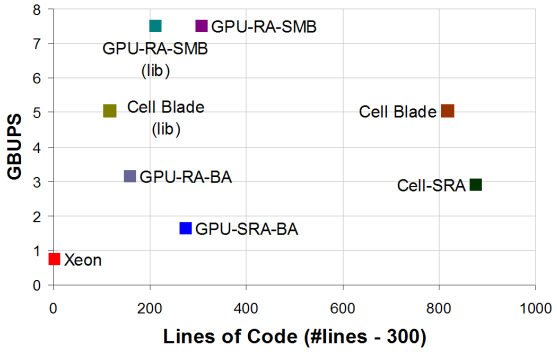


Figure 6. Productivity (*error-tolerant update mode*)

• *Productivity:* Figure 6 measures the productivity for different optimizations on the Xeon, Cell and GPU. It plots lines of code and corresponding performance for various optimizations for 128-byte updates in *update mode*. We note that the Cell and GPU significantly boost the performance for random accesses in comparison to conventional architectures, such as Xeon. However the improved performance comes at the cost of reduced productivity due to extra programming effort involved. The GPU shows the best performance/lines of code ratio. It requires considerably less programming effort compared to the Cell. The Cell performs better than GPU for the *streaming mode* (Cell-SRA and GPU-SRA-BA) with small look-ahead limit but also requires more programming effort.

To study the productivity gains when using our libraries, we compared the library version with the non-library version of our implementation. GPU-RA-SMB(lib) and GPU-RA-SMB show the performance with and without using the library on GPU for large update sizes (section 4.1) respectively. Similarly, Cell-Blade and Cell-Blade (lib) show the *parallel update mode* performance with and without using the library on Cell (section 3.1). This demonstrates that our libraries significantly improve programmer productivity.

Read-Only Mode. The GPU performance scaling with *read-only mode* shows similar trends as with the *update mode* shown in Figure 4. The performance peaks when the total number of threads goes above 256 and the best performance is achieved with 4 thread-blocks each containing 64 threads.

Using multiple partitions on the Cell improves performance in *read-only mode* as well but the gain is less than what was obtained in *update mode*. For 8 SPEs, performance with 4 partitions is 39% better in comparison to 1 partition.

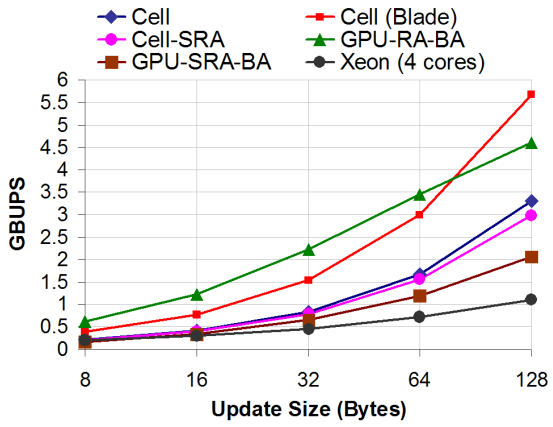


Figure 7. Cell and GPU performance with different update sizes for *read-only mode*

Figure 7 shows the Cell, GPU and Xeon performance with varying update sizes for *read-only mode*. The performance numbers are better in this case compared to *update mode* as expected and show similar trends.

Table 3 also shows performance numbers for *read-only mode*. GPU-RA-SMB leads to almost $\times 2$ improvement over GPU-RA-BA for 128-bytes.

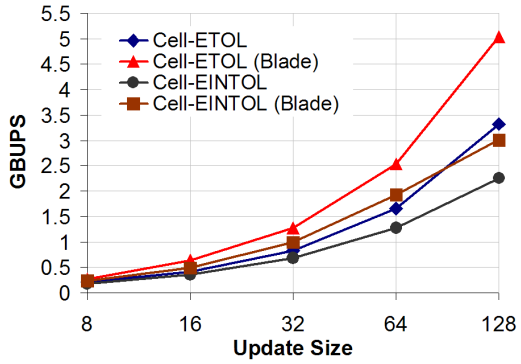


Figure 8. Cell performance with error tolerant and intolerant modes (*parallel update mode*)

Error-intolerant update mode. Figure 8 compares the performance of *error-tolerant* (ETOL) and *error-intolerant* (EINTOL) update modes for varying update sizes on 8 SPEs (Cell) as well as 16 SPEs with interleaved data (Cell(Blade)). For EINTOL, the partition based approach (section 3.1) benefits with increasing number of partitions and updates per partition buffer. Therefore, for comparing performance, we relaxed the look-ahead limit to 32K allowing for more partitions and updates per buffer. The EINTOL performance is more than 76% of the ETOL performance up to update sizes of 64 bytes, showing effectiveness of the optimizations incorporated in the optimized SPE library. The performance with 128 update size is lower as SPE Local Store size limits the number of partitions and updates per buffer.

6. Conclusion

We proposed a new model for evaluating the performance of random memory accesses as exhibited by applications in practice. Our model better captures the varying characteristics of random access behavior in practical applications when compared to traditional benchmarks. This is demonstrated by the fact that the performance results show significant variation in different modes, such as *parallel update* and *streaming update* modes.

We used our model to evaluate the random access performance on Cell and GPU. Our optimized numbers on GPU are better by a factor of 2.4 over the base version for 128-byte updates in *parallel error-tolerant update mode*. The Cell performance in *streaming mode* is similar to the *parallel mode*, however, the GPU performance in *streaming mode* is much lower compared to *parallel mode* when host stream buffer size is small. We conclude that the GPU is more suitable when multiple update streams are being generated from different cores. However, in the case where the update stream is generated from a single host/processor, the Cell performs better when the buffer size is small.

We also showed that the Cell and GPU significantly boost the performance of random memory accesses in comparison to conventional architectures. However this performance improvement comes at the cost of reduced productivity due to the extra programming effort involved. We showed that by providing carefully designed optimized libraries, this effort can be reduced without compromising performance.

Acknowledgements

We would like to thank Rahul Garg for his expert opinion and useful discussions.

References

[1] V. Aggarwal, Y. Sabharwal, R. Garg and P. Heidelberger, *HPCC RandomAccess Benchmark for Next Generation Supercomputers*, IPDPS 2009.

[2] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'Brien, and K. O'Brien, *A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor*, LCPC 2007.

[3] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, *Ray Tracing on the Cell Processor*, IEEE RT 2006.

[4] T. Chen, T. Zhang, Z. Sura, and M. G. Tallada, *Prefetching irregular references for software cache on cell*, CGO 2008.

[5] T. Chen, R. Raghavan, J. Dale and E. Iwata, *Cell Broadband Engine Architecture and its first implementation*, http://www.ibm.com/developerworks/power/library/pa-cellperf.

[6] J. Dongarra and P. Luszczek, *Introduction to the HPC Challenge Benchmark Suite,* TR:ICL-UT-05-01, ICL, 2005.

[7] R. Garg and Y. Sabharwal, *Software Routing and Aggregation of Messages to Optimize the Performance of HPCC Randomaccess Benchmark*, SuperComputing 2006, pp 109.

[8] IBM Corp., *Cell B.E. Programming Handbook v 1.1*.

[9] NVIDIA CUDA Programming Guide 2.1, http://www.nvidia.com/object/cuda_get.html.

[10] S. J. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis, *A Simple Synchronous Distributed-Memory Algorithm for the HPCC RandomAccess Benchmark*, Cluster Computing, 2006, pp. 1-7.

[11] H. Scherl, B. Keck, M. Kowarschik, J. Hornegger, *Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)*, Nuclear Science Symposium Conference Record. 2007.

[12] R. Shams and R. A. Kennedy, *Efficient histogram algorithms for NVIDIA CUDA compatible devices*, ICSPCS 2007, pp. 418-422.