

Enable back memory and global synchronization on LLC buffer

Licheng Yu¹ · Yulong Pei¹ · Tianzhou Chen¹ ·
Xueqing Lou¹ · Minghui Wu² · Tiefei Zhang³

Published online: 15 June 2017
© Springer Science+Business Media, LLC 2017

Abstract The last-level cache (LLC) shared by heterogeneous processors such as CPU and general-purpose graphics processing unit (GPGPU) brings new opportunities to optimize data sharing among them. Previous work introduces the LLC buffer, which uses part of the LLC storage as a FIFO buffer to enable data sharing between CPU and GPGPU with negligible management overhead. However, the baseline LLC buffer's capacity is limited and can lead to deadlock when the buffer is full. It also relies on inefficient CPU kernel relaunch and high overhead atomic operations on GPGPU for global synchronization. These limitations motivate us to enable back memory and global synchronization on the baseline LLC buffer and make it more practical. The

✉ Licheng Yu
yulicheng@zju.edu.cn

Yulong Pei
pyl88429@zju.edu.cn

Tianzhou Chen
tzchen@zju.edu.cn

Xueqing Lou
xqlou@zju.edu.cn

Minghui Wu
mhwu@zucc.edu.cn

Tiefei Zhang
tfzhang@mail.zjgsu.edu.cn

- ¹ College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China
- ² School of Computer and Computing Science, Zhejiang University City College, Hangzhou, Zhejiang, China
- ³ School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou, Zhejiang, China

back memory divides the buffer storage into two levels. While they are managed as a single queue, the data storage in each level is managed as individual circular buffer. The data are redirected to the memory level when the LLC level is full, and are loaded back to the LLC level when it has free space. The case study of n-queen shows that the back memory has a comparative performance with a LLC buffer of infinite LLC level. On the contrary, LLC buffer without back memory exhibits 10% performance degradation incurred by buffer space contention. The global synchronization is enabled by peeking the data about to be read from the buffer. Any request to read the data in LLC buffer after the global barrier is allowed only when all the threads reach the barrier. We adopt breadth-first search (BFS) as a case study and compare the LLC buffer with an optimized implementation of BFS on GPGPU. The results show the LLC buffer has speedup of 1.70 on average. The global synchronization time on GPGPU and CPU is decreased to 38 and 60–5%, respectively.

Keywords LLC buffer · Synchronization · Heterogeneous system · GPGPU

1 Introduction

1.1 Background

The heterogeneous multicore system with general-purpose graphics processing unit (GPGPU) is able to deliver high performance for various applications. Shared last-level cache (LLC) between CPU and GPGPU enables fine grain data sharing in these heterogeneous cores and avoids data copying between discrete GPGPU memory and system memory. However, the CPU–GPGPU system differs from the symmetry multiple CPU cores due to programming model and execution model of GPGPU.

A GPGPU program is called a *kernel*. The programming languages of GPGPU such as CUDA [21] and OpenCL [8] allow the programmers to write code for a single thread of a kernel, which is instantiated to thousands of threads at run time to provide high data parallelism. Threads are organized into *blocks*¹ and can synchronize with each other as long as they are in the same block. This requirement ensures lower interthread communication overhead for GPGPU. Though threads in different blocks can synchronize with global atomic operations, the overhead is much higher. As a result, the global synchronization is usually done via relaunching the kernel by CPU after the previous kernel exits [10].

Meanwhile, the GPGPU exhibits distinctive patterns in both thread scheduling and memory access due to its unique architecture. A GPGPU consists of multiple stream multiprocessors (SMs), which are connected to memory interfaces (to LLC) by an on-chip network. Each SM runs one or more thread blocks, and threads in each of which are further grouped into *warps*. Threads of the same warp run in lock step, so a memory access instruction of a warp can produce multiple memory requests. Memory requests to the same cache line are coalesced in SM to reduce memory system overhead. However, with thousands of threads running on GPGPU, its memory

¹ For simplicity, we use CUDA terminologies.

bandwidth requirement is still much larger than a CPU. The different memory access patterns make the cooperation in data exchange between CPU and GPGPU non-trivial.

1.2 Baseline LLC buffer and its limitations

To facilitate cooperation between CPU and GPGPU with a shared LLC, we introduce the LLC buffer [18]. It replaces part of the LLC storage with a hardware-managed FIFO buffer. The data exchanged between processors are in *elements*, each of which can be read once and written once. Each element is further divided into fixed length *atoms*. An atom is the data transmission unit of a memory request. The element-atom organization enables arbitrary atom access order and arbitrary element length. The buffer architecture keeps the order of atoms in each element and manages the queue storage. When a request fails due to writing full buffer or reading empty buffer, it is rejected and the processor can resend the request later. The buffer storage organization is also optimized for coalesced memory accesses from GPGPU. More details about the proposed LLC buffer design are presented in Sect. 2.

The hardware-managed queue access eliminates global atomic operations in software queue management. Therefore, it enables simple and efficient data exchange between CPU and GPGPU. However, the baseline LLC buffer is just a proof of concept. One problem is that an element must be stored in the LLC until it is consumed. Although, in this way, the overhead of memory system below LLC is reduced, in a real application such as n-queen problem, all threads may be writing the buffer before they can consume another elements. Deadlock is raised when the buffer is full and all of the threads are blocked at writing.

On the other hand, due to lack of efficient global synchronization in GPGPU, applications involving global data synchronization or barrier such as breadth-first search (BFS) are required to synchronize with CPU by relaunching the kernel at each global barrier. In this way, the finish condition can be checked and the global barrier is ensured via kernel launch serialization in CPU side. Meanwhile, in GPGPU side, optimization such as local buffer [19] may also be used for more efficient data transmission. However, the optimization itself can incur high overhead. For the baseline LLC buffer, kernel relaunching and GPGPU code optimization overhead are still inevitable to implement global synchronization. As a result, the performance benefit brought by the LLC buffer is eliminated in applications require global synchronization.

1.3 Contributions

To this end, we introduce a pragmatic LLC buffer that aims to solve the problems of storage deadlock and inefficiency of global synchronization in the baseline LLC buffer. The contributions of this paper are:

- We reveal the limitations of the baseline LLC buffer: limited buffer storage and no global synchronization support.
- Back memory is added to the baseline LLC buffer to save LLC space and avoid deadlock due to limited buffer storage. It redirects the element writings to the main

memory when the LLC-level storage is full and loads them back to the LLC level when it has free space. It takes advantage of the FIFO buffer access to hide the memory access latency.

- Global synchronization is supported in the LLC buffer to alleviate both CPU and GPGPU overhead of software implementation. The LLC buffer finds the global barrier according to the data in the buffer and only allows element reads before the barrier until all threads reach the barrier. The same infrastructure is also used to control threads execution in heterogeneous multicore system for graceful exiting.
- We implement n-queen and BFS as case studies to demonstrate the ability of back memory and global synchronization. We show that n-queue with the back memory has a comparative performance with an infinite storage baseline LLC buffer and BFS with global synchronization supported LLC buffer has an average speedup of 1.70 compared with an optimized GPGPU implementation [19].

The paper is organized as follows. Section 2 briefly describes the baseline LLC buffer and reveals the limitations that make the baseline not so useful in real-world applications. Sections 3 and 4 introduce back memory and global synchronization on the LLC buffer, respectively, to address the problems mentioned in Sect. 2. Case studies with n-queen and BFS are presented in Sect. 5 to demonstrate the enhanced LLC buffer. Section 6 gives the related work, and Sect. 7 concludes the paper.

2 The baseline LLC buffer

2.1 Element-atom organization

In the LLC buffer, we assume that data exchanged between processors are of fixed size and the unit is called *element*. To support arbitrary element size, we propose the baseline LLC buffer in [18]. It organizes element in fixed size *atoms* with unique atom IDs. An element access is done by accessing each atom of that element iteratively. In case of GPGPU, threads of the same warp execute in lock step; thus, a coalesced memory request holds requests to the atoms with same ID of multiple elements. For example, for warp with 32 threads accessing a buffer of two-atom element (i.e., atom-0 and atom-1), an element access is divided into two coalesced requests, the first one contains requests to atom-0 of 32 elements, while the second one is for atom-1 of the same batch of 32 elements.

The LLC buffer data organization for two-atom element is demonstrated in Fig. 1. To enable efficient coalesced request servicing, atoms of the same ID that a warp access together are stored back to back in *block*, which is of the same size with the LLC cache line. Blocks of the same batch of elements form a *stream*. For example, *WIB1* in Fig. 1 contains atom-0 of 32 elements, which are accessed by one coalesced memory request from one 32-thread warp. The block *WIB1* is aligned to one LLC line (128 bytes), while *WIB2* stores atom-1 of the same batch of 32 elements in the next cache line. *WIB1* and *WIB2* belong to the same stream. Finally, an element consists of all atoms of the same offset from blocks of one stream. In Fig. 1, two first atoms of *WIB1* and *WIB2* (shown in blue and orange, respectively) are of the same offset in block (offset = 0) and form the first element of the stream.

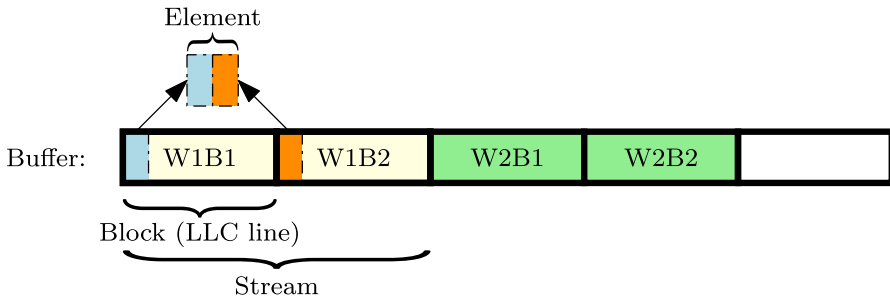


Fig. 1 Element-atom organization of the data buffer (two-atom element)

2.2 Access protocol

An element is accessed an atom a time. However, accesses to atoms of different elements probably interleave each other in case of multicore or multiple SMs trying to access the same buffer and lead to race condition. Although locks such as mutex are traditionally used in CPU programs to resolve this problem, overhead becomes significant when the currency increases. Thus, they are not suitable in the shared LLC system, and lockless queue is adapted in the LLC buffer. Instead of transparent to the processors, the LLC buffer has to be accessed with dedicated API or access protocol.

It is possible that the buffer access fails when read an empty buffer or write a full buffer. In these cases, a processor has to repeatedly send the access request until it is fulfilled. To avoid redundant data transmission due to these unsuccessful requests, a reserve request with minimal payload is issued to check and reserve enough readable elements or writable empty slots before the actually atom data accesses. The minimal payload the reserve request contains the buffer ID and the number of element to access, which can be larger than one in case of GPGPU's coalesced memory access. The response of a successful reserve request returns address of the reserved elements in the buffer that the request is going to access. A valid address consists of stream ID and offset within the target stream. After that, the processor can issue atom access requests of the given elements in any order. Besides the address, an atom access request also provides atom ID and number of accessed atoms. On the other hand, when the reserve request fails, an invalid address is returned. The processor retries until the reserve request returns with a valid address.

The code for typical element read (`ReadElement`) of a n -atom element is shown in Fig. 2. The argument *buffer* indicates which buffer will be accessed. The processor first issues the read reserve request via `bufresread` repeatedly until it gets a valid *address* (lines 2–4). Then, all the atoms of the element are accessed though `bufread` with the *address* and corresponding atom IDs (lines 5–7). Since there is no dependence among the atoms, `bufreads` can execute in parallel after the successful `bufresread`. Therefore, instruction-level parallelism can be exploited with the proposed access protocol. The code is for a single thread, so buffer access requests from threads of a same warp in

Fig. 2 Baseline LLC buffer read (n -atom element)

Input: *buffer* the target buffer
Output: *atoms*[n] a n -atom element

```

1: procedure READELEMENT(buffer)
2:   repeat
3:     address = bufresread(buffer)
4:   until address is VALID
5:   for  $i = 0$  to  $n - 1$  do
6:     atoms[ $i$ ] = bufread(buffer, address,  $i$ )
7:   end for
8:   return atoms
9: end procedure

```

GPGPU will always be coalesced as long as they target the same buffer. Each thread only accesses one element in ReadElement, so element count, which equals the threads executing bufresread and bufread in a warp, is calculated by SM. Element write API (WriteElement) is similar to read.

2.3 LLC buffer management

The LLC buffer manager on the LLC side serves the buffer requests. Besides the data buffer, each stream has meta-data that record how many atoms are written and read. The stream meta-data are also managed in a circular FIFO buffer named *stream queue*. For low management overhead, each stream in stream queue consists of following data:

- R : elements reserved for read;
- W : elements reserved for write;
- S : atoms already been read;
- F : atoms already been written.

For each stream, W and R are updated by reserve requests, while F and S are updated by atom access requests.

Supposing that an element consists of n atoms and a stream can hold at most m elements, a stream has $m - W$ empty slots reservable for write and $W - R$ elements readable (provided all atoms of reserved elements are written). Besides, the following conditions hold for stream management:

- $W = m$: all empty slots are reserved for write (stream full);
- $R = W$: all written elements are read (stream empty);
- $F = nW$: all atoms of reserved (write) elements are written;
- $S = nR$: all atoms of reserved (read) elements are read.

Since the atoms of reserved elements can be written to the stream in any order, a stream is readable only when all atoms of reserved elements are written ($F = nW$). Similarly, a stream can be freed only when all atoms of its written elements are read ($R = W \wedge S = nR$). For example, in Fig. 3, stream 0 has all its element written ($16_F = 2_n * 8_W$), so the stream is readable and three elements are already read. On the other hand, stream 1 is reserved two elements for read, but one of their atoms is not written. As a result, stream 1 is not readable until the missing atom arrives.

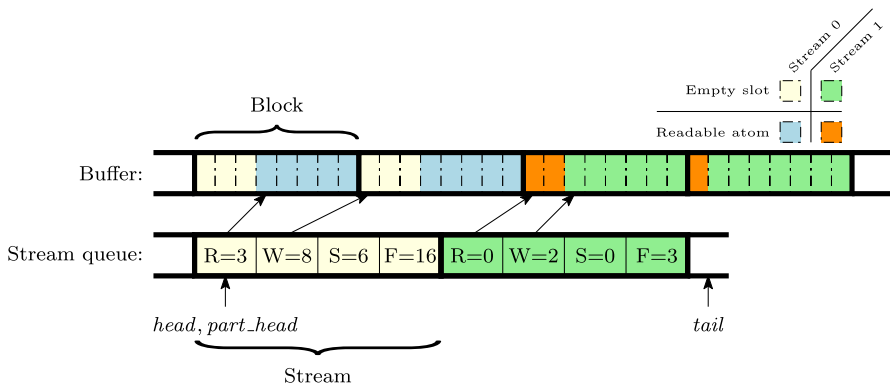


Fig. 3 LLC buffer management (two-atom element)

The LLC buffer keeps pointers to the *head* stream and *tail* stream of each stream queue (as shown in Fig. 3). The *head* points to the first stream (oldest) with unread atoms and the *tail* points to the last stream with empty slots in queue. While *head* moves forward when the first stream is entirely read, *tail* moves forward when empty slots of the tail stream are all reserved. Due to the arbitrary order of atom accesses, the stream pointed by *head* may be fully reserved for read but their atoms are not entirely read. To avoid searching for readable elements, *part_read* points to the first stream with unread elements ($W > R$). The *part_read* moves forward when written elements of the stream are all reserved for read. For simplicity in management, the internal fragmentation caused by read in a stream is not reused until the whole stream is freed.

Since a coalesced request does not always carry the same number of atoms (for example, when some threads are masked off), not all requests access the whole block or start at the block boundary. As a result, it is possible that a coalesced request cannot be fulfilled with the remaining atoms or slots in one stream. Due to the limited request size (never larger than one block), a coalesced request accesses at most two successive streams. This is transparent to the processors and is handled by the LLC buffer manager, which issues two accesses to corresponding LLC lines when a request spans two streams. It can take advantage of the LLC bank parallelism to eliminate the time overhead when the two LLC lines are in different banks.

The storage of both the buffer and stream queue is reserved in the cache lines when the buffer is allocated in the LLC. Each block in the buffer occupies one LLC line, and each stream takes up insignificant space. Given that the LLC line is 128 bytes and each atom is 4 bytes, each of the R , W , S and F of a stream has to refer up to 32 elements and needs 5 bits; thus, a stream occupies 20 bits. To avoid unaligned stream spanning two LLC lines, a stream is rounded up to 4 bytes. Thus, the management overhead of a stream is $4/(128n)$ of the data buffer for n -atom element (less than 2% when $n > 1$). Meanwhile, stream queue and data buffer are directly mapped, so they are allocated and freed together. As a result, there is no extra overhead to manage the data buffer.

Fig. 4 Typical use case of the LLC buffer

```

1: while next do
2:   element = ReadElement(buffer)
3:   resultset = ProcessElement(element)
4:   for each result in resultset do
5:     WriteElement(buffer, result)
6:   end for
7: end while

```

2.4 Use case and limitations

A typical use case of the LLC buffer is that all threads start together and read the same buffer. For each thread, after processing an element, results are stored back to the buffer. The code of each thread is shown in Fig. 4. Note that the results produced by ProcessElement may consist of more than one element. The procedure continues until the data are in a final state when ProcessElement does not generate any new elements.

There are some problems in the use case. The saved elements are only consumed until one of the threads finishes writing and starts consuming them again. However, the buffer storage occupies the LLC, whose limited storage space can lead to deadlock when there are no more empty slots in buffer, but all threads are blocked on buffer writing. This problem is exacerbated when a thread writes more elements in each iteration in Fig. 4.

On the other hand, global synchronization is not considered in the baseline LLC buffer. The stop condition *next* flag is a global atomic variable in our earlier implementation, which can be set by ProcessElement. Nevertheless, atomic operation is expensive and thus inefficient in GPGPU threads, so this is not scalable. Meanwhile, it relies on CPU for global barriers. For example, BFS has a similar code pattern as the use case, but requires the next level of nodes start after the previous level done visiting. In the baseline LLC buffer, it is not supported without multiple kernel launches by CPU.

3 Back memory

3.1 Overall design

In order to prevent the deadlock caused by the limited buffer storage, back memory is added to the LLC buffer. Part of the main memory is allocated to serve as the back memory. The LLC buffer now consists of two levels: the LLC level and the memory level. When the LLC level is full, the extra data are stored in the memory level. It therefore allows the threads to finish writing and continue to consume the buffer data. Thus, the deadlock described in the previous section is prevented.

A naïve solution is treating the data buffers in two levels as a single buffer logically by the LLC buffer manager. This eventually results in directly reading the main memory by the processors when the *head* points to memory level. A read request is usually sensitive to latency because of the data dependence of the following instructions. As

a result, the naïve solution leads to longer latency, which goes against the motivation of the LLC buffer.

On the contrary, a memory write instruction is usually independent with its following instructions. Therefore, instruction-level parallelism can be exploited to cover the latency of the memory write instructions. To avoid high read latency, it is better that LLC buffer reads only access the LLC level. When the buffer is full, the further writes are redirected to the back memory without performance degradation caused by write latency. When there is empty space in LLC level, the data at memory level are loaded back to the LLC level, so that reads can hopefully hit the LLC level.

Similar to the traditional cache design, the back memory of LLC buffer is transparent to the processors and programmers. However, traditional cache scheme is prone to cache thrashing, which happens when a cache line is repeatedly writing to and reading from back memory. It is not trivial to eliminate cache thrashing due to the unpredictable cache access pattern. Fortunately, the FIFO property of the LLC buffer ensures that an element can be written to or loaded from back memory at most once and thus avoids buffer thrashing.

3.2 Stream mapping

Both the memory level and the LLC level are managed by the LLC buffer manager in one stream queue. As shown in Fig. 5, the stream queue is made up of three parts. The first part (yellow) contains streams in the LLC level, the second part (blue) is streams loading from memory to LLC, and the third part (green) contains streams in the memory level. The loading part exists only when there are loading streams, and storage of these streams is actually duplicated in both LLC data buffer and memory data buffer. All the three parts are seen as a single queue both by the processors and by the LLC manager when managing the stream queue. Similar to the baseline buffer, the *head* and *tail* point to the first stream and the first non-full stream, respectively, no matter where the data are located. To keep a simple design, the whole stream queue is always stored in the LLC even for the streams in the memory level. The buffer capacity is not likely limited by the stream queue even with large back memory, since they are relatively small.

Despite the single queue view of the stream queue, to avoid accessing the memory level when the LLC level is not full, the manager ensures that the first part of the queue is always in the LLC level. The LLC-level data buffer and the memory-level data buffer

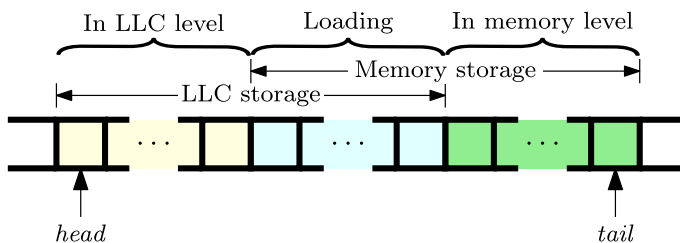


Fig. 5 Stream queue with back memory

Input: *streamId* offset of the stream in the stream queue
Output: *llcId* if the given stream is in LLC level or *memId* if it is in memory level
 1: **if** *memHead* is INVALID **or** *streamId* < *memHead* **then**
 2: *llcId* = *streamId* - *head* + *llcStart*
 3: **else**
 4: *memId* = *streamId* - *memHead* + *memStart*
 5: **end if**

Fig. 6 Mapping from queue stream ID to stream offset in data buffer

are managed as two individual circular buffers. As a result, the stream queue and the data buffer are not directly mapped as in the baseline LLC buffer. Besides the base memory address of the memory-level buffer, more meta-data are added to manage these buffers for each queue:

- *llcStart*: the head stream of the LLC-level data buffer;
- *memStart*: the head stream of the memory-level data buffer;
- *memHead*: the first loading or memory stream in the stream queue;
- *loadings*: the number of streams being loading from the memory level.

With the additional meta-data, the LLC manager can determine whether a stream is in the LLC level or is in the memory level. The stream ID of the returned address in the reserved response is still the offset in the stream queue. The buffer manager can get the exact address of its data buffer location by mapping the stream queue offset to the local offset of either LLC level or memory level. The mapping also ensures that the returned stream ID constantly points to the same batch of elements during their life time, even after the stream has been moved from the memory level to the LLC level.

Figure 6 shows the mapping logic, where the add, minus and compare operations are assumed to take circular buffer into consideration. *head*, *tail* and *memHead* point to the stream queue. When there are streams locating in the memory level, the *memHead* is set between *head* and *tail*. Therefore, when a given stream ID is between *head* and *memHead*, it is in the LLC level. Otherwise, the stream is in the memory level. In either case, the local offset is the difference between the stream ID and the corresponding head (*head* or *memHead*). The stream offset in the data buffer is then achieved by adding the start offset of the data buffer (*llcStart* or *memStart*) to the local offset. Similar to the baseline, the atom ID and atom offset are then used to calculate the final address.

An example snapshot of the stream queue and data buffers with back memory is presented in Fig. 7, where each data buffer can store five streams. The number inside each stream is the stream ID in the stream queue (valid stream IDs are from 1 to 8). The first stream in LLC level is at *llcStart* (2) and the first stream in the memory level is at *memStart* (3). For example, the given stream ID 2 is between the *head* (1) and *memHead* (5). Therefore, it is in the LLC level and the stream offset inside the LLC level is 3 (=2 - 1 + 2). On the other hand, the stream ID 7 is between *memHead* and *tail* and locates in memory level. The stream offset inside the memory level is 0 [(7 - 5 + 3) mod 5]. Note that the circular buffer warps around to the beginning of the memory-level data buffer.

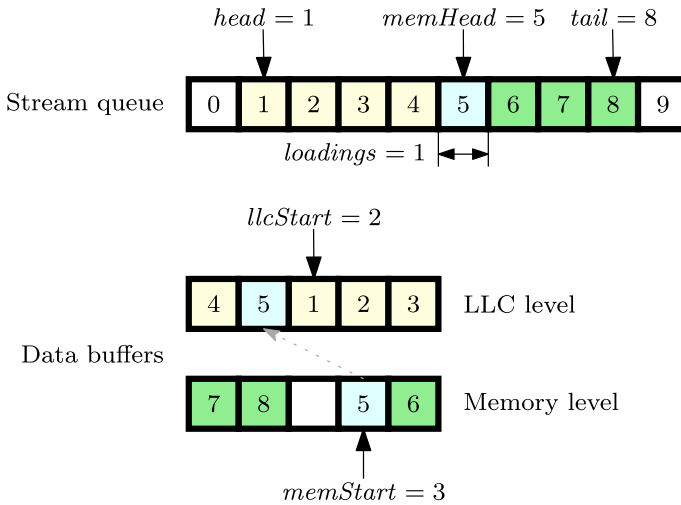


Fig. 7 An example snapshot of stream queue and data buffers

3.3 Data buffer management

The LLC level data buffer is initially used in the same way as the baseline LLC buffer except that the storage is offset by $llcStart$. When the LLC level is full, the $memHead$ is set to the first stream in the memory level ($memHead = head + llcLevelSize$). Further data write requests to the streams in memory level are redirected to memory by the LLC buffer manager, while reads are not allowed until they are loaded to the LLC-level data buffer. Since the redirections and loadings are all traditional memory access requests issued by the LLC buffer manager, there is no modification required in the memory level.

The first stream in the memory level is loaded to the LLC level as soon as the head stream advances when a stream is freed in the LLC level. Each block of the stream requires one reading request. Access to the data of a loading stream is not allowed due to possible data hazard. The LLC buffer manager uses $loadings$ to track how many streams are being loaded from the memory. As a result, the manager can tell that a given stream is loading when $streamId \in [memHead, memHead + loadings)$. For example, in Fig. 7, stream 5 is loading from memory to LLC level, and therefore, it exists in both data buffers. Memory overflow is prevented naturally, because the LLC manager understands that there are $tail - memHead + 1$ streams in the memory data buffer, which includes any loading streams. For example, there are two free streams (0 and 9) in the stream queue in Fig. 7, while the memory data buffer only presents one free stream. The write is blocked when the memory level is fully filled. Fortunately, large memory level renders the blocking uncommon.

Reserve writes only access meta-data in the stream queue, and therefore, no special action except memory full checking is taken when the corresponding data are in the memory level. When a data write request with a target stream in the memory level arrives, the buffer manager generates traditional memory write requests with memory

address calculated by the mapping discussed in the previous section. These generated memory requests are issued along with normal LLC miss requests. Consequently, the original data write may be rejected when the LLC memory queue is full or the corresponding stream is loading from memory. Since the memory queue is expected to drain quickly, the rejection is done with any traditional low-level method such as credit-based flow control.

In case of reads, they are not allowed on the memory-level streams. A reserve read to the stream in the memory level is rejected by the LLC buffer manager with an invalid stream ID. Reads are not rejected with low-level method as in the data write requests because it is possible that the streams in the LLC are not fully consumed and the loading will not happen in the near future. Since a stream in LLC level will never write to memory, data read requests following a successful reserve read always hit the LLC level.

4 Global synchronization

4.1 Finish signal

The design so far does not provide any way to send the finish signal at the end of data processing. All threads will eventually wait on element read when no one is producing any more elements. In other words, the *next* flag in Fig. 4 has to be updated at the end. In fact, without further synchronization, it seems that only the LLC buffer manager is able to know that all threads are stuck. The LLC buffer manager can send a special address as the finish signal in the reserve response. The *next* flag in Fig. 4 is therefore set in `ReadElement` accordingly.

The problem is how to tell the LLC buffer manager which threads are stuck. A simple solution would be to attach the thread ID with the reserve request, while the LLC buffer manager keeps a map to record whether each thread is blocked by the buffer. A bit corresponding to each thread in the map is either set or cleared when the last reserve request from that thread is rejected or fulfilled, respectively. When all bits are set, a reserve response will be sent with the finish signal to each thread. Eventually, all threads stop after they get the finish signal. It requires that the LLC buffer manager knows the total number of running threads from the beginning and all reserve requests are augmented with a thread ID field. The former requirement is trivial, since the starting CPU knows exactly how many GPGPU or CPU threads it starts. On the other hand, the latter tends to introduce non-negligible overhead due to the extra data field. Meanwhile, a fixed field is not scalable with different number of threads.

Fortunately, the LLC buffer just has to know when all threads are stuck instead of knowing exactly which one. A thread is stuck when it is looping endlessly for a reserve request with valid address in the reserve loop (lines 2–4 in Fig. 2). The LLC buffer manager can count the stuck threads with a counter instead of keeping a thread bitmap. The counter increases when the first reserve request in the loop from a thread is rejected, and decreases when a reserve request from a stuck thread is fulfilled later. Therefore, when the counter reaches the preset total threads, the finish signal is sent.

```

Input: first flag of the first request in the reserve loop
Input: address the address returned to the response
Input: n the number of threads coalesced in the request
Input: stuckthreads number of total stuck threads
Input: totalthreads number of total running threads
1: if address is invalid then
2:   if first then
3:     stuckthreads = stuckthreads + n
4:   end if
5:   if stuckthreads = totalthreads then
6:     stuckthreads = stuckthreads - n
7:     totalthreads = totalthreads - n
8:     set address to finish signal
9:   end if
10: else
11:   if not first then
12:     stuckthreads = stuckthreads - n
13:   end if
14: end if

```

Fig. 8 Finish signal support when a reserve request is served by the LLC buffer manager

Fig. 9 Finish signal support in reserve loop of ReadElement

```

1: first = 1
2: repeat
3:   address = bufresread(buffer, first)
4:   if address is finish signal then
5:     exit
6:   end if
7:   first = 0
8: until address is valid

```

Figure 8 demonstrates the process when a reserve request is served by the LLC buffer manager. For a rejected request, if the request is the first one of the reserve loops (with the *first* flag), the counter is increased to indicate *n* more threads are waiting from now, where *n* is the count of atoms in the request in case of a coalesced request from GPGPU. At a point that all threads are stuck (lines 5–9), the invalid address is replaced by a finish signal to all the further responses. Since these threads are supposed to exit after they get the signal, the total counter is also decreased. As a result, other threads will get the exit signal when they resend the reserve request. On the other hand, when the reserve request is successfully served, but it is not the first one in the reserve loop, the counter is decreased (lines 11–13) since they have been accumulated earlier.

For the reserve request, instead of carrying the exact thread ID, a single bit is added to indicate whether the request is the first request in the reserve loop. This can be simply achieved by replacing the reserve loop in Fig. 2 with the code in Fig. 9. The *first* flag is sent with the buffer ID. It is set before the reserve loop and cleared in the following loops. Meanwhile, when the returned address is the finish signal, the thread simply exits.

For a coalesced read reserve request, when the readable elements in the buffer are less than it requests, it is rejected. As a result, when all threads are stuck, it is not necessarily that all elements in the LLC buffer are read. With the previous logic, the whole application may exit prematurely and the buffer is not completely drain. To address this problem, the finish signal is only sent when the buffer is empty. Meanwhile, when all the threads are stuck while the LLC buffer has unread elements, a coalesced request is allowed to be *partly served*. To this end, the processor that issues a coalesced request is modified to check the size of the response. When the returned data are less than it required, only part of the coalesced request is successfully served, while the other part is kept stuck. For GPGPU, it creates control divergence since some threads of the same warp are moving on. It will converge as long as they go back to the reserve loop. At this point, all threads are stuck and no readable element is in the LLC buffer, the previous logic can therefore decently end all threads.

4.2 Global barrier

Highly optimized GPGPU application usually includes sophisticated management code for efficient data transmission, such as stashing to local buffer before writing to global buffer with coalesced requests. Nonetheless, the optimization itself incurs overhead in GPGPU thread. This is resolved by the baseline LLC buffer, which removes data management code for sharing among processors. However, other synchronizations such as global barrier among consumers are still not supported. For example, in BFS, it must ensure that all nodes at the current level are visited before it goes to the next level. That is, the node consumers should be synchronized at the start of each level. Traditionally, as in a GPGPU implementation [10] of BFS, the same kernel is launched repeatedly, one for each level. In each launch, the kernel reads from the source buffer and generates the next level nodes to the destination buffer. The buffers (pointers) are swap between iterations. In this case, kernel launch by CPU works as a global barrier between each of the levels. However, kernel launch overhead can be significant. Therefore, if the global barrier can be supported by the LLC buffer, both overheads incurred by optimization in GPGPU code and global synchronization in CPU code can be alleviated.

Each thread is required to write the LLC buffer before the barrier and read the same buffer just after the barrier. Due to the FIFO property of the LLC buffer, all data before the barrier are in the first part. As long as all threads write different data across the barrier, but keep the data same between two barriers, the barrier point is just before the element with different data from the previous element. As a result, the LLC buffer can throttle the threads by rejecting the read reserve requests when they trying to read the elements after the barrier. When all threads are reading these elements, they all reach the barrier. At this point, the elements become readable and all threads go across the barrier. The data to indicate its barrier of each element are supposed to be embedded in the element data. For example, in BFS, one atom of each element is the level number, which increases across each barrier. Since the LLC buffer only cares about the difference before and after a barrier, one bit that flips between them also works.

```

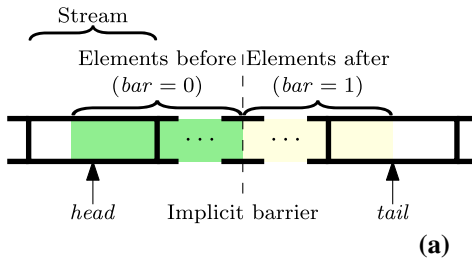
1: bar = 0
2: repeat
3:   (barrier here)
4:   in = ReadElement(...)
5:   out = ProcessElement(in)
6:   WriteElement(out|bar)
7:   bar = ~ bar
8: until finish

```

▷ 1-bit flag for barrier

▷ toggle between barrier

Fig. 10 Code using global barrier with LLC buffer



Input: *request* the reserve read request

Input: *first* flag of the first request

Input: *stuckthreads* number of total stuck threads

Input: *totalthreads* number of total running threads

```

1: bar = Sample(request)
2: if bar = cbar then
3:   (Normal read)
4: else if stuckthreads = totalthreads then
5:   cbar = bar
6:   (Normal read)
7: else
8:   (Reject request)
9: end if

```

▷ all threads reach the barrier

▷ some threads reach the barrier

(b)

Fig. 11 Global barrier support in LLC buffer. **a** Elements in LLC buffer and **b** LLC buffer management

Figure 10 shows the code pattern that uses global barrier with LLC buffer. An element is read from the buffer, and output elements are produced. The *bar*, which flips before each barrier, is attached to each output element before write. As shown in Fig. 11a, the elements before barrier are read from *head* stream and written elements are stored in *tail* stream until read reaches an implicit barrier, which can be detected when the next element has different *bar* value. The LLC buffer ensures ReadElement works as a global barrier and returns only when all threads reach the barrier.

When the global barrier is enabled in LLC buffer by management API, the global barrier processing is shown in Fig. 11b. The *bar* bit position of an element is also set via management API. The *bar* bit can be sampled during either element read or write. While sampling in reading, the data buffer has to be accessed by reserved read request. The next readable element can be sampled once and cached to avoid resampling for further reads until the next readable element changes. The current *bar* value *cbar* is also kept in the manager. An element can be read only when *cbar* equals to *bar* of the

element. Otherwise, all reserve reads are rejected, until the LLC buffer receives at least one reserve read request from all threads (when $stuckthreads = totalthreads$, both of them are managed by finish signal processing as shown in Fig. 8). At this moment, $cbar$ is set to bar of the first element in the buffer, and all threads are released to read.

5 Case study

5.1 Methodology

We evaluate the system with a simulator that combined the Gem5 [6] and the GPGPU-Sim v3.2 [5]. The former is a full system simulator for memory system and CPU, and the latter is a modern GPGPU simulator that mimics the NVIDIA Fermi architecture. Both of them are cycle accurate and widely used. We implement the baseline LLC buffer as well as the improved LLC buffer supporting back memory and global synchronization in the simulator. The GPGPU is configured similar to NVIDIA GTX480. The configuration parameters are listed in Table 1.

We implement the baseline LLC buffer manager and the LLC buffer manager with back memory and global synchronization support in Verilog. The design is synthesized by Yosys 0.7 [29] and mapped to Nangate 45-nm open cell library. The baseline LLC buffer (including the storage for meta-data of 16 buffers) and the one with back memory and global synchronization support occupy 0.016 and 0.029 mm² chip area, respectively. As an 8 MB cache in 45 nm technology node requires 51.7 mm² (from CACTI 5.3 [27]), the LLC buffer manager has a negligible 0.056% chip area overhead.

Table 1 Configuration parameters

CPU	Core frequency	2 GHz
	Data L1 cache	64 KB
	L2 cache	2 MB
GPGPU	SMs	15
	SM frequency	1400 MHz
	L1 cache	16 KB, write-evict, private to each SM
	NoC	Butterfly
	L2 cache	6 banks, 128 KB each, shared by all SMs
LLC	Block size	128 bytes
	Associative	8 ways
	Size	8 MB
	Hit latency	20 ns
	Replacement	LRU
Memory	Bandwidth	12.8 GB/s
	Latency	50 ns

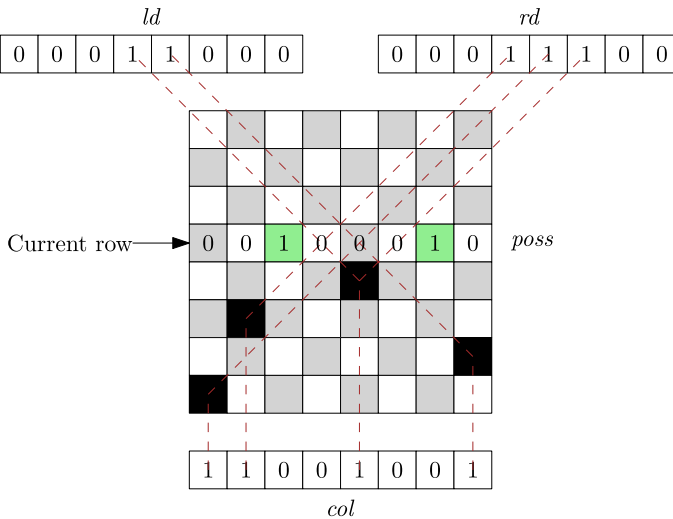


Fig. 12 Board state represented by three bitmaps (queens are placed in black blocks) [25]

5.2 N-queen

The n-queen problem tries to find the number of solutions of queen placing, each of which places n queens without any two of them sharing the same row, column or diagonal on an $n \times n$ chessboard. We use an efficient backtracking algorithm [25] that generates possible solutions row by row of the chessboard. In each row, it takes the state of the board to decide which positions a queen can set. The board state is represented by three bitmaps, which give the confliction on the left diagonal (*ld*), right diagonal (*rd*) and column (*col*) of each position in the current row. Figure 12 from [25] shows a board state indicated by these bitmaps. Therefore, the possible positions of queens in the current row (*poss*) are calculated by $poss = \sim(ld|col|rd)$. The bitmaps of the next row are then generated according to the position selected in *poss*, and it recursively does the above steps until all rows are decided.

We implement the above algorithm on the GPGPU with the LLC buffer enabled. An element of the buffer used by n-queen consists of three 4-byte words (*ld*, *rd*, *col*). The initial bitmap set (0, 0, 0) is pushed to the LLC buffer by the CPU before the kernel is launched. The kernel code is shown in Fig. 13. In each iteration of the outer loop, a bitmap set is read from the LLC buffer (line 3). When all queens are placed, a solution is found and the counter is increased (line 5). Otherwise, the bitmap of current row is calculated, and each 1 bit in it is extracted as a possible solution path (lines 9–10). After that, the bitmaps for the next row are generated with the selected bit and the possible solution path is written back to the LLC buffer (line 14). The outer loop continues until all solutions are found.

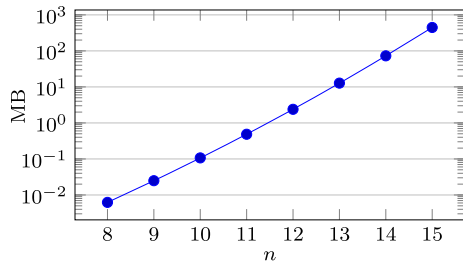
The kernel can also run with the baseline LLC buffer. However, the limited LLC buffer tends to overflow as more threads generate more states. The kernel configuration is 15 blocks/grid and 32 threads/block in this section. Figure 14 gives the maximum used LLC buffer storage in different n of n-queen. We find that the storage requirement

```

Input: buffer LLC buffer
Input: all bitmap of possible position of a row
Input: counter result counter
1: procedure NQUEEN
2:   repeat
3:     (ld, col, rd) = ReadElement(buffer)
4:     if col = all then
5:       atomicInc(counter)
6:     else
7:       poss =  $\sim (ld|col|rd) \& all$ 
8:       while poss  $\neq 0$  do
9:         bit = poss & ( $-poss$ )
10:        poss = poss - bit
11:        nld = (ld|bit) << 1
12:        nrd = (rd|bit) >> 1
13:        ncol = col|bit
14:        WriteElement(buffer, (nld, ncol, nrd))
15:      end while
16:    end if
17:  until exit
18: end procedure
    
```

Fig. 13 Kernel for n-queen

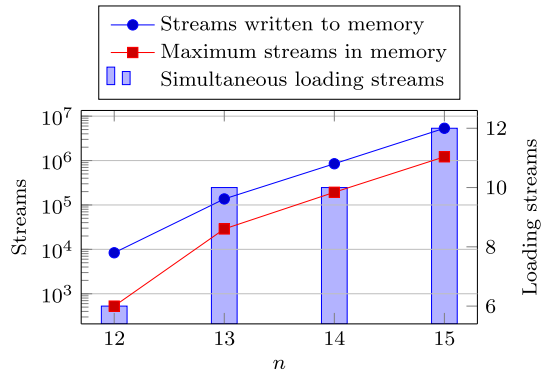
Fig. 14 Buffer storage requirement of n-queen



grows exponentially as the n increases. When $n = 8$, only 6528 bytes is used. When $n = 15$, 447MB of the LLC buffer is required simultaneously. With very limited LLC buffer, deadlock may appear when all threads try to write to the buffer before any of them can consume the existed elements. The back memory is therefore used to augment the buffer storage in main memory to prevent the deadlock in the baseline LLC buffer.

We assume that the LLC level takes 2MB from the 8MB LLC storage, and an infinite back memory. When the streams in buffer are more than 2MB, new elements are written to the back memory. The number of streams written to the memory and the maximum streams simultaneously occupied in the memory are pictured in Fig. 15. Since for $n < 12$, all the streams can be served in the LLC level, no stream is ever written to the back memory in these cases. When $n = 15$, 5340481 streams have been written to the back memory, and they carry 1.9 GB data. Meanwhile, due to the FIFO management, at most 1214827 streams (444 MB) are saved in the memory at the same time. Figure 15 also gives the maximum streams that are simultaneously loading back

Fig. 15 Back memory accessing of n-queen



to the LLC level. When $n = 15$, at most 12 streams are loading to the LLC level at the same time. Since 2-MB LLC level gives more than 5000 streams, only about 0.2% streams are not readable due to loading, and therefore, the loading overhead is insignificant.

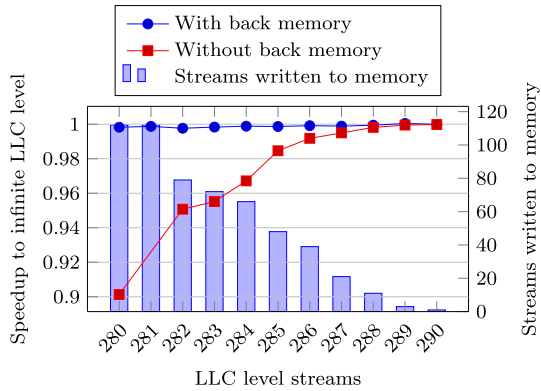
We explore the performance in execution time when the back memory is disabled. The lines in Fig. 16 give the speedup under different LLC-level capacities compared with infinite LLC level when $n = 10$. As shown in Fig. 14, at most 291 streams (109 KB) are used simultaneously. Without the back memory, when the storage capacity in the LLC level is less than 280 streams, deadlock happens when all warps in the SM try to write the LLC buffer under our kernel configuration. Although it is able to finish when the LLC level has more streams without back memory, the performance degrades because some warps are waiting to write the LLC buffer when it is full. For example when 280 streams are provided, the baseline LLC speedup is 0.9 compared with infinite LLC buffer. On the other hand, we find that the total overhead incurred by back memory is almost negligible. Since the memory level is not read directly and the streams are loaded back timely, the back memory access never becomes a bottleneck of n-queen. As a result, although more than 400 MB data are written to and read from the back memory when $n = 15$, the performances from $n = 8$ to $n = 15$ in the 2M LLC level with infinite memory-level storage are almost identical to an infinite LLC level (these results are not given in the figure for simplicity).

5.3 BFS

Breadth-first search (BFS) is widely used in graph theory and is applied in many fields. Given a graph where nodes are connected by a set of edges, BFS visits nodes in levels, and each node is only visited once. For a given level, all unvisited neighbors of the nodes are visited first and constitute the next level. This is usually achieved with a *frontier* queue that contains nodes of the same level. The queue is visited, and the frontier of the next level is composed meanwhile.

To ensure the level traverse, parallel BFS has to be synchronized. For example, a simple BFS implementation [10] assigns each node in the graph to one thread to avoid a software-managed queue. In the frontier array that is as large as all nodes of the

Fig. 16 Performance in different LLC-level streams when $n = 10$



graph, nodes that belong to the current level are marked. The barrier between levels takes place on CPU, where the kernel is repeatedly launched without overlapping to visit the nodes in each level until no nodes in the frontier array are marked. Obviously, this simple BFS implementation is not scalable when there are not enough threads to cover all the nodes in a large graph. Besides, each kernel launch incurs overhead, which is not acceptable when the graph level is higher.

We therefore consider a more sophisticated implementation [19] of BFS on GPGPU as our baseline. It introduces hierarchical queues to alleviate the overhead of global frontier queue management. The hierarchical queue consists of three levels, each of which is used by a kernel to deliver the best performance for different scale of nodes. When the nodes are less than threads of a single block, kernel of *one block level* synchronization only uses barrier in shared memory. Whenever the nodes exceed threads of one block, the kernel of *SM level* synchronization does the job with atomic operation in global memory to enable the interblock synchronization [30]. The number of block is limited to the number of SM available (15 in our configuration). Further, when this limit is exceeded, a kernel of *wavefront level* synchronization similar to the method in [10] is used, which requires a kernel launch for each level. All the three kernels first write to the local queue in shared memory and save to global memory with highly coalesced memory requests.

The LLC buffer with global barrier can implement BFS with a single kernel and relieves the programmer’s burden to deal with different level of frontier queues. The kernel of BFS with LLC buffer is shown in Fig. 17. The buffer element consists of two atoms: the node ID to be visited and its level. The main loop reads one element from the buffer (line 3), and each of its connected node is checked if it has been visited (line 8). Each unvisited node and its level is written back to the buffer (line 9). The kernel does not contain any code for global barrier, because it is done automatically by the LLC buffer during element read. The LLC buffer is set to monitor the least significant bit of the first atom in each element (the *level* atom, which is updated in each new level). The LLC buffer allows ReadElement to continue only when all threads are reading new elements. As a result, the BFS traverse is ensured.

The evaluation dataset we used consists of two randomly synthesized graphs (100 nodes and 10K nodes) and BFS dataset from Parboil benchmark [26]. Table 2 gives

```

Input: buffer LLC buffer
Input: nodevec node data
Input: edgevec edge data
Input: levels level for each node
1: procedure BFS(buffer)
2:   repeat
3:     (level, nid) = ReadElement(buffer)
4:     node = nodevec[nid]
5:     visit(node)
6:     for i = start(node) to end(node) do
7:       cid = edgevec[i]
8:       if atomicCAS(levels[c], 0, level) = 0 then
9:         WriteElement(buffer, (level + 1, cid))
10:      end if
11:    end for
12:  until exit
13: end procedure

```

Fig. 17 BFS kernel using LLC buffer

Table 2 BFS dataset

Benchmark	Nodes	Edges	Levels	Source
100	98	424	6	Synthesized
10K	9895	44,882	10	Synthesized
UT	248,730	590,269	817	Parboil
NY	270,926	740,426	632	Parboil
1M	1,000,000	5,992,002	1999	Parboil
SF	1,257,001	7,239,016	1003	Parboil

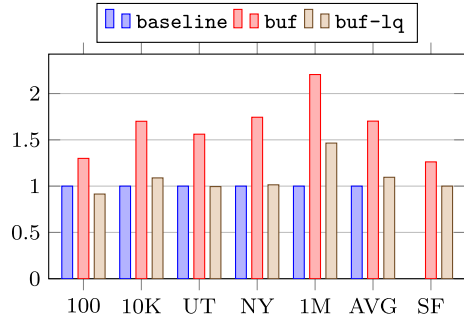
Table 3 Launch times and levels processed by kernels of *baseline*

Benchmark	One block level		SM level		Wavefront level	
	Lunched	Levels	Launched	Levels	Launched	Levels
100	1	6	0	0	0	0
10K	2	6	1	4	0	0
UT	7	679	6	138	0	0
NY	6	415	5	217	0	0
1M	2	1024	1	975	0	0

the details of the dataset. The LLC is 8 MB for *baseline* and is 6 MB for kernel using LLC buffer, which occupies the other 2 MB. The kernel using LLC buffer is launched in 32 blocks/grid and 64 threads/block. Table 3 gives the number of launches and the level traversed of each *baseline* kernel under our configuration.

We compare the *baseline* and two implementations of the LLC buffer: *buf* directly writes to LLC buffer as shown in Fig. 17, and *buf-lq* writes to local queues first and coalesces them as more as possible when writes to the LLC buffer (as in

Fig. 18 BFS speedup (normalized to *baseline*, SF is normalized to *buf-lq* and is excluded from AVG)



baseline). The SF is not finished in *baseline* due to local queue overflow. To support more nodes in one block level and SM level, maximum threads (512) must be set to the block and it is prone to overflow. On the contrary, while *buf* does not use local queue, *buf-lq* is able to reuse a thread in the queue and runs with moderate threads per block (64). As a result, both implementations with LLC buffer do not suffer from the overflow.

The speedup of *baseline* and our implementations with LLC buffer are shown in Fig. 18. Since SF is not finished in *baseline*, it is listed solely and is not included for average. The LLC buffer achieves 1.70 speedup in average compared with *baseline*. As the input graph scales up, the LLC buffer brings more speedup, which is not only caused by eliminating multiple kernel launches, but also due to the efficient global synchronization (for example, 1M has only three kernel launches in *baseline* as presented in Table 3, but it exhibits lower performance in *baseline*). Meanwhile, the LLC buffer with local queue has an average speedup of 1.10, which is the result of overhead from local buffer management and memory access conflict. The entire local queue is sent to the LLC buffer to coalesced more requests as possible. It helps increase the efficiency in *baseline*, but does not contribute much with the LLC buffer. The requests to LLC buffer are always coalesced but sending local queue concentrates the memory access at the end of each block’s iteration. As a result, the local queue incurs more overhead and degrades the performance with the LLC buffer.

To understand the synchronization overhead, we collect the time spent on the global barrier. Since the iteration with LLC buffer does not in unit of BFS level as in the *baseline*, we divide the average time of each warp spent for synchronization by the BFS level it processed. Meanwhile, only SM level in *baseline* does the level synchronization, as shown in Table 3. Figure 19 gives the result, where 100 is absent because of zero-launch of SM level kernel. We find that the LLC buffer uses 38.03% of *baseline* SM level synchronization time for each level it processed. This is due to that the LLC buffer avoids the high overhead incurred by the global atomic operation and other optimization code (such as local queue and global queue management) in implementing the global barrier and data sharing.

Another overhead of global barrier is incurred by CPU processing. In the *baseline*, the CPU is involved to check the current level queue and launch the proper kernel for further processing. With the LLC buffer, the kernel is only launched once and the CPU is not used until the end of BFS. Figure 20 depicts the percentage

Fig. 19 Average time spent per BFS level for global synchronization with LLC buffer (buf, normalized to baseline)

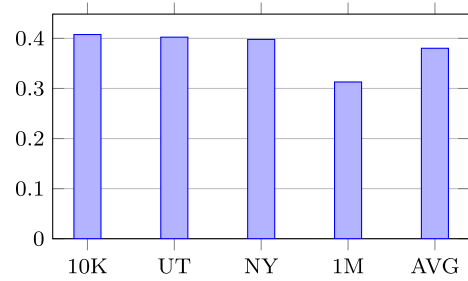
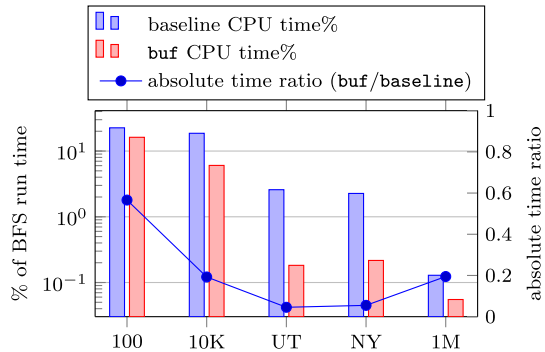


Fig. 20 Time spent by CPU for global synchronization



of time used by CPU during their kernel running and the CPU absolute time ratio of buf and baseline. Although 100 has only one kernel launch of baseline, due to simpler management code, the buf CPU time is 56.60% of baseline. The CPU overhead of baseline increases when more kernels are launched. Meanwhile, the absolute CPU overhead of buf is fixed. For UT with 11 kernel launches, the buf CPU time is only 4.56% of the baseline. As a result, the LLC buffer alleviates the burden of CPU for global synchronization.

6 Related work

Heterogeneous platforms are widely studied in recent years. Efficient communication among heterogeneous cores especially CPUs and GPGPUs is a rather important problem in these platforms. Similar to our motivation, [9] finds that communicating through cache or traditional scratchpad memory either is constrained by memory wall or requires specific optimization by programmers. [9] separates memory access from computing and makes them communicate through scratchpad memory with compiler. [13, 14] enable automatic GPU parallelization by managing and optimizing CPU–GPU communication during run time. Since GPGPU’s programming and memory models differ dramatically from conventional CPUs, many automatic mapping algorithms are introduced to improve the performance [23], as well as to select the best memory allocation and communication policy [17, 20]. [24] focuses on linearization of non-uniform dimensions for vectorization without modifying memory accesses. Meanwhile, the data transfer between the these processors is also scheduled to achieve better

performance [3]. [16] directly maps a device's I/O address space to the host memory and allows direct data transfer without copying. Most of these methods can also build on the LLC buffer. With global synchronization enabled, the processors can efficiently communicate with each other via the LLC buffer.

The GPGPU is more likely to be directly used by operating systems or high-level programming languages. For example, [15] presents an OS-level GPU resource management system that allows the user and the OS to treat the GPU as a first-class computing resource. [4] introduces a hardware and OS co-design to improve the performance in heterogeneous processors, which are hidden with a unified interface managed by a microkernel OS to enable flexible task scheduling. High-level languages such as Java [7] are also extended to target heterogeneous systems and are able to generate optimized GPGPU code with a JIT compiler [11, 12]. OpenMP is also automatically translated to OpenCL with data transformation when it is worth running the code on GPGPU [28]. With the help of the LLC buffer, there are more opportunities for interprocessor communication to be more efficient and convenient in these solutions.

On the other hand, cache coherence between CPU and GPGPU seems promising for data sharing. But it is not widely adopted due to different private protocols of them. Compiler [22] and GPU cache [2] are used to identify potential stale accesses to avoid redundant transfers between CPU and GPGPU as well as to eliminate the tedious work from the programmers. [1] finds that memory coalescing contributes to most of the performance and most shared data are read only to GPGPU. They introduce selective cache, which turns off coherence of the shared data that are cached by CPU to avoid a unified complex coherence protocol. However, the cache coherence protocol inevitably incurs non-trivial overhead to the system. The massive data transfers are more efficient with the LLC buffer, and thus, they are complementary to our work.

7 Conclusion

The baseline LLC buffer is a proof of concept to improve the data exchange of shared LLC CPU–GPGPU system. However, the limited LLC buffer storage may lead to deadlock. To support global synchronization, it relies on CPU kernel relaunch, which eliminates the performance improvement in the baseline LLC buffer. In this paper, we enhance the baseline LLC buffer to support back memory and global synchronization.

The back memory is added to enlarge the LLC buffer and avoid deadlock caused by limited space. The data storage is divided into the LLC level and the memory level. They are managed in one stream queue, while each data buffer is individually managed circular buffer in each level. Requests written to the buffer are redirected to the memory level when the LLC level is full. Streams in the memory level are loaded to the LLC level when the LLC level has free space. As a result, the back memory is transparent to the processor. We use n-queen as a case study to demonstrate the overhead compared with both limited and infinite LLC-level buffers. The results show that the back memory eliminates the performance overhead incurred by limited LLC buffer even when a lot of elements are written to the memory level.

Another enhancement is the global synchronization on LLC buffer. It avoids the kernel relaunch or complicated manual code optimization for synchronization. The

element data are accessed by the LLC buffer manager to distinguish whether it is before or after the global barrier. The elements after the barrier are unreadable only when all threads reach the barrier. Consequently, the global synchronization is done seamlessly in the LLC buffer without any performance overhead to either CPU or GPGPU. We compare the LLC buffer with an optimized implementation for global synchronization in the case study of BFS. The results give speedup of 1.70 in average. The global synchronization time on GPGPU and CPU is decreased to 38% and 60–5%, respectively.

Acknowledgements This project is supported by the National Natural Science Foundation of China (Grant No. 61379035), the National Natural Science Foundation of Zhejiang Province, China (Grant No. LY14F020005) and the National Natural Science Foundation of Zhejiang Province, China (Grant No. LQ14F02001).

References

1. Agarwal N, Nellans D, Ebrahimi E, Wenisch TF, Danskin J, Keckler SW (2016) Selective gpu caches to eliminate cpu-gpu hw cache coherence. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 494–506, doi:[10.1109/HPCA.2016.7446089](https://doi.org/10.1109/HPCA.2016.7446089)
2. Al-Saber N, Kulkarni M (2015) Semcache++: Semantics-aware caching for efficient multi-gpu offloading. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ACM, New York, ICS '15, pp 79–88, doi:[10.1145/2751205.2751210](https://doi.org/10.1145/2751205.2751210)
3. Amini M, Coelho F, Irigoien F, Keryell R (2013) Static Compilation Analysis for Host-Accelerator Communication Optimization, Springer Berlin Heidelberg, Heidelberg, pp 237–251. doi:[10.1007/978-3-642-36036-7_16](https://doi.org/10.1007/978-3-642-36036-7_16)
4. Asmussen N, Völp M, Nöthen B, Härtig H, Fettweis G (2016) M3: A hardware/operating-system co-design to tame heterogeneous manycores. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, ASPLOS '16, pp 189–203, doi:[10.1145/2872362.2872371](https://doi.org/10.1145/2872362.2872371)
5. Bakhoda A, Yuan GL, Fung WWL, Wong H, Aamodt TM (2009) Analyzing cuda workloads using a detailed gpu simulator. In: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pp 163–174, doi:[10.1109/ISPASS.2009.4919648](https://doi.org/10.1109/ISPASS.2009.4919648)
6. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 simulator. SIGARCH Comput Archit News 39(2):1–7. doi:[10.1145/2024716.2024718](https://doi.org/10.1145/2024716.2024718)
7. Dubach C, Cheng P, Rabbah R, Bacon DF, Fink SJ (2012) Compiling a high-level language for gpus: (via language support for architectures and compilers). In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, PLDI '12, pp 1–12, doi:[10.1145/2254064.2254066](https://doi.org/10.1145/2254064.2254066)
8. Group KOW et al. (2008) The opencl specification. 1(29):8
9. Ham TJ, Aragón JL, Martonosi M (2015) Desc: Decoupled supply-compute communication management for heterogeneous architectures. In: Proceedings of the 48th International Symposium on Microarchitecture, ACM, New York, MICRO-48, pp 191–203, doi:[10.1145/2830772.2830800](https://doi.org/10.1145/2830772.2830800)
10. Harish P, Narayanan PJ (2007) High Performance Computing – HiPC 2007: 14th International Conference, Goa, India, December 18–21, 2007. Proceedings, Springer Berlin Heidelberg, Heidelberg, chap Accelerating Large Graph Algorithms on the GPU Using CUDA, pp 197–208
11. Hayashi A, Ishizaki K, Koblents G, Sarkar V (2015) Machine-learning-based performance heuristics for run time cpu/gpu selection. In: Proceedings of the Principles and Practices of Programming on The Java Platform, ACM, New York, PPPJ '15, pp 27–36, doi:[10.1145/2807426.2807429](https://doi.org/10.1145/2807426.2807429)
12. Ishizaki K, Hayashi A, Koblents G, Sarkar V (2015) Compiling and optimizing java 8 programs for gpu execution. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp 419–431, doi:[10.1109/PACT.2015.46](https://doi.org/10.1109/PACT.2015.46)
13. Jablin TB, Prabhu P, Jablin JA, Johnson NP, Beard SR, August DI (2011) Automatic cpu-gpu communication management and optimization. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, PLDI '11, pp 142–151, doi:[10.1145/1993498.1993516](https://doi.org/10.1145/1993498.1993516)

14. Jablin TB, Jablin JA, Prabhu P, Liu F, August DI (2012) Dynamically managed data for cpu-gpu architectures. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, ACM, New York, CGO '12, pp 165–174, doi:[10.1145/2259016.2259038](https://doi.org/10.1145/2259016.2259038)
15. Kato S, McThrow M, Maltzahn C, Brandt S (2012) Gdev: First-class gpu resource management in the operating system. Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12). USENIX, Boston, pp 401–412
16. Kato S, Aumiller J, Brandt S (2013) Zero-copy i/o processing for low-latency gpu computing. In: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems, ACM, New York, ICCPS '13, pp 170–178, doi:[10.1145/2502524.2502548](https://doi.org/10.1145/2502524.2502548).
17. Lee H, Brown KJ, Sujeeth AK, Rompf T, Olukotun K (2014) Locality-aware mapping of nested parallel patterns on gpus. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, Washington, MICRO-47, pp 63–74, doi:[10.1109/MICRO.2014.23](https://doi.org/10.1109/MICRO.2014.23).
18. Licheng Y, Yulong P, Tianzhou C, Xueqing L, Minghui W, Tiefei Z (2016) LLC buffer for arbitrary data sharing in heterogeneous systems. In: High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on, IEEE, pp 260–267
19. Luo L, Wong M, Hwu Wm (2010) An effective gpu implementation of breadth-first search. In: Proceedings of the 47th Design Automation Conference, ACM, New York, DAC '10, pp 52–55, doi:[10.1145/1837274.1837289](https://doi.org/10.1145/1837274.1837289)
20. Margiolas C, O'Boyle MFP (2014) Portable and transparent host-device communication optimization for gpgpu environments. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, New York, CGO '14, pp 55:55–55:65, doi:[10.1145/2544137.2544156](https://doi.org/10.1145/2544137.2544156)
21. Nvidia C (2008) Cuda programming guide
22. Pai S, Govindarajan R, Thazhuthaveetil MJ (2012) Fast and efficient automatic memory management for gpus using compiler-assisted run time coherence scheme. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ACM, New York, PACT '12, pp 33–42, doi:[10.1145/2370816.2370824](https://doi.org/10.1145/2370816.2370824)
23. Phothilimthana PM, Ansel J, Ragan-Kelley J, Amarasinghe S (2013) Portable performance on heterogeneous architectures. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, ASPLOS '13, pp 431–444, doi:[10.1145/2451116.2451162](https://doi.org/10.1145/2451116.2451162).
24. Ren B, Ravi N, Yang Y, Feng M, Agrawal G, Chakradhar S (2016) Automatic and Efficient Data Host-Device Communication for Many-Core Coprocessors, Springer International Publishing, Cham, pp 173–190. doi:[10.1007/978-3-319-29778-1_11](https://doi.org/10.1007/978-3-319-29778-1_11)
25. Richards M (1997) Backtracking algorithms in MCPL using bit patterns and recursion. Citeseer
26. Stratton JA, Rodrigues C, Sung I, Obeid N, Chang L, Anssari N, Liu G, Hwu W (2012) The parboil technical report. Tech. rep., IMPACT Technical Report (IMPACT-12-01), University of Illinois Urbana-Champaign
27. Thoziyoor S, Muralimanohar N, Ahn JH, Jouppi NP (2008) Cacti 5.1. Tech. rep., Technical Report HPL-2008-20, HP Labs
28. Wang Z, Grewe D, O'boyle MFP (2014) Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. ACM Trans Archit Code Optim 11(4):42:1–42:26, doi:[10.1145/2677036](https://doi.org/10.1145/2677036)
29. Wolf C, Glaser J, Kepler J (2013) Yosys-a free verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)
30. Xiao S, c Feng W (2010) Inter-block gpu communication via fast barrier synchronization. In: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp 1–12, doi:[10.1109/IPDPS.2010.5470477](https://doi.org/10.1109/IPDPS.2010.5470477)