2019

Performance Evaluation of Blocking and Non-Blocking Concurrent Queues on GPUs

Hossein Pourmeidani
University of Mississippi

PERFORMANCE EVALUATION OF BLOCKING AND NON-BLOCKING CONCURRENT QUEUES ON GPUS

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Hossein Pourmeidani

December 2018

ABSTRACT

The efficiency of concurrent data structures is crucial to the performance of multi-threaded programs in shared-memory systems. The arbitrary execution of concurrent threads, however, can result in an incorrect behavior of these data structures. Graphics Processing Units (GPUs) have appeared as a powerful platform for high-performance computing. As regular data-parallel computations are straightforward to implement on traditional CPU architectures, it is challenging to implement them in a SIMD environment in the presence of thousands of active threads on GPU architectures. In this thesis, we implement a concurrent queue data structure and evaluate its performance on GPUs to understand how it behaves in a massively-parallel GPU environment. We implement both *blocking* and *non-blocking* approaches and compare their performance and behavior using both micro-benchmark and real-world application. We provide a complete evaluation and analysis of our implementations on an AMD Radeon R7 GPU. Our experiment shows that non-blocking approach outperforms blocking approach by up to 15.1 times when sufficient thread-level parallelism is present.

ACKNOWLEDGEMENTS

My first and foremost acknowledgment goes to my family. I'd like to thank my parents who have supported me till I got here. I would like to start with my mother, my role model, who always pursued education and hard work to the last day of her life. I thank her for planting in me the love of learning and believing in my abilities. I would like to thank my father for the full freedom and trust he granted me to navigate through my self-awareness journey the way I choose. My deepest thanks goes to my sisters, brothers, and wife for the endless support and unconditional love. I would like to thank my cousin and best friend Mohammadreza for being my home away from home. I wouldn't have done it without him.

I would like to express my gratitude to my research adviser, Dr. Byunghyun Jang, for encouraging me to think independently and opening for me new doors of knowledge. I am grateful for all the guidance, help and honest advice he has offered me during my Masters journey. I have been fortunate to work with him and gain professional skills that are valuable for my future career.

I am grateful to all my current lab mates David Troendle and Mason Zhao, and previous lab members Ajay Sharma, Tuan Ta and Esraa Gad for their collaboration. I would like to especially thank David Troendle for all the insightful discussions and the foundation on which this thesis was built.

Last but not least, I would like to thank all the professors in the department of computer science for all the help they are always willing to offer. Special thanks to Dr. Conrad Cunningham and Dr. Feng Wang for the valuable discussion and the constructive critique of my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In shared-memory multiprocessors, multiple active threads run simultaneously and communicate and synchronize via data structures in shared memory. As the efficiency of these data structures is critical to performance, designing efficient data structures for multiprocessor machines has been extensively studied. Designing such concurrent data structures is much more difficult than sequential ones since threads running simultaneously may interleave arbitrarily and can result in an incorrect behavior. Furthermore, scalability is a challenge in design of concurrent data structures as contentions among threads can severely undermine scalability [14]. There exist implementations of different concurrent data structures, such as stacks [20], queues [2, 3, 7, 15, 21] and skip-lists [19], but most of them target multi-core CPUs.

Recently, Graphics Processing Units (GPUs) have become one of the most preferred platforms for high-performance parallel computing. This computing model is generally referred to as *General Purpose Computing on GPU (GPGPU)* or *GPU computing*. While the aforementioned concurrent data structures have been implemented and evaluated on many different multi-core CPU architectures but little has been studied on GPU. With the recent introduction of improved memory models including atomic primitives on GPUs, existing concurrent data structures for multi-core CPUs can be ported to GPUs. Regular data-parallel computations with little or no synchronization have been efficiently implemented on the GPUs. However, irregular workloads are known to be difficult to implement due to their

dynamic behavior of control flow and parallelism. Achieving scalable performance of those workloads needs efficient concurrent data structures to use for thread synchronization and communication. In medium-scale parallel machines with tens of active thread contexts, it may be manageable to support synchronizations among them but with thousands of active threads, this would cause significant performance overhead on GPUs. The appearance of OpenCL [16] has made general purpose programming on GPUs easier but the design and implementation of concurrent data structures still remains challenging.

In this thesis, we present the evaluation of blocking and non-blocking implementations of concurrent queue data structure on GPUs. To the best of our knowledge, this is the first attempt to understand their behaviors on GPUs in depth. All of our implementations are written in OpenCL C++ programming model and rely on OpenCL's atomic primitives such as atomic compare-and-exchange and atomic exchange. We evaluate our implementations using several micro-benchmarks and a real-world application. All of our evaluation are carried out on a AMD Radeon R7 GPU.

The rest of this thesis is organized as follows. Chapter 2 presents related work and background. Chapter 3 describes the design and implementation of the proposed blocking and non-blocking concurrent queues. Chapter 4 shows the experimental results. Conclusions are made in Chapter 5.

CHAPTER 2

RELATED WORK AND BACKGROUND

In this chapter, we review previous works on the topic of concurrent data structures (CDS), and provide the background information required to understand this thesis.

2.1 Related Work

Concurrent queues have been studied for three decades. Most of them have targeted multi-core CPUs, and only a few works targeted GPUs. In the section, we review several CPU-based CDS implementations and a few GPU-based studies that we found.

The majority implementations of CDSs are *Compare and Swap (CAS)* based non-blocking. Mellor-Crummey [9] proposed a concurrent queue which is blocking based on *fetch-and-store*. Since enqueue and dequeue operations access both *Front* and *Rear*, enqueuers and dequeuers interfere each other's cache line and therefore results in limited scalability. Min et al. [12] proposed a scalable cache-optimized queue, which is also blocking. They entirely remove *CAS* failure in enqueue operation by replacing *CAS* with *fetch-and-store* and considerably decrease cache line interference among enqueuers and dequeuers. Although the queue shows better performance it includes a *CAS* retry loop in dequeue operation.

Michael and Scott [11] presented the most widely used non-blocking concurrent queue algorithm. It updates *Front*, *Rear*, and *Rear's next* by a non-blocking approach by using *CAS*. If the *CAS* fails, the thread is repeated until it succeeds in *CAS*. However, beyond a rather low concurrency level, the frequent *CAS* retries result in a complete loss of scalability [1],

[4]. Ladan-Mozes and Shavit [6] proposed a new concurrent lock-free queue with reduction in number of *CAS* operations from two to one in an enqueue operation. The fewer number of required *CAS* operations results in less possibility of *CAS* failure and better scalability. In [13], pairs of concurrent enqueue and dequeue operations have the ability to alter values without accessing the shared queue itself. Unfortunately, this approach is applicable to only small queues since the enqueue operation cannot be eliminated until all former values have been dequeued in order to preserve the correct FIFO queue semantics. Hoffman et al. [5] decreased the possibility of *CAS* retries in an enqueue operation by replacing baskets of mixed-order entities with the standard totally ordered list. Unfortunately, using a basket in the enqueue operation causes a new overhead in the dequeue operation because linear search among *Front* and *Rear* is needed to find the first non-dequeued node. In addition, a contention restriction scheme between losers who failed the *CAS* is required. As a result, in some architectures, the baskets queue performs worse than the Michael and Scott's queue [4].

Xiao and Feng introduced inter-block synchronization that synchronizes threads across blocks on a GPU by communicating through global memory [22]. Stuart and Owens presented the implementations of barriers, mutexes, and semaphores on GPUs [18] and Michael presented lock-free hash tables [10]. Our evaluation of the blocking and non-blocking queues considered in this thesis is the first attempt to gain a detailed understanding of the performance of concurrent queues on GPUs.

2.2 Background

This section surveys topics on concurrent data structures, OpenCL, and atomic operations related to our work.

2.2.1 Concurrent Data Structures

Most data structures being designed are a kind of conventional sequential data structures. In concurrent data structures, the semantics of conventional data structures are relaxed in order to get simpler and more efficient and scalable implementations. For example, when traversing a graph through BFS algorithm by using a concurrent queue, it might be enough to allow each thread do the enqueue operation to add their values in the queue, and not necessarily in the same order. As shown in Figure 2.1, threads *T1*, *T2* and *T3* can add their elements to the queue in any orders.

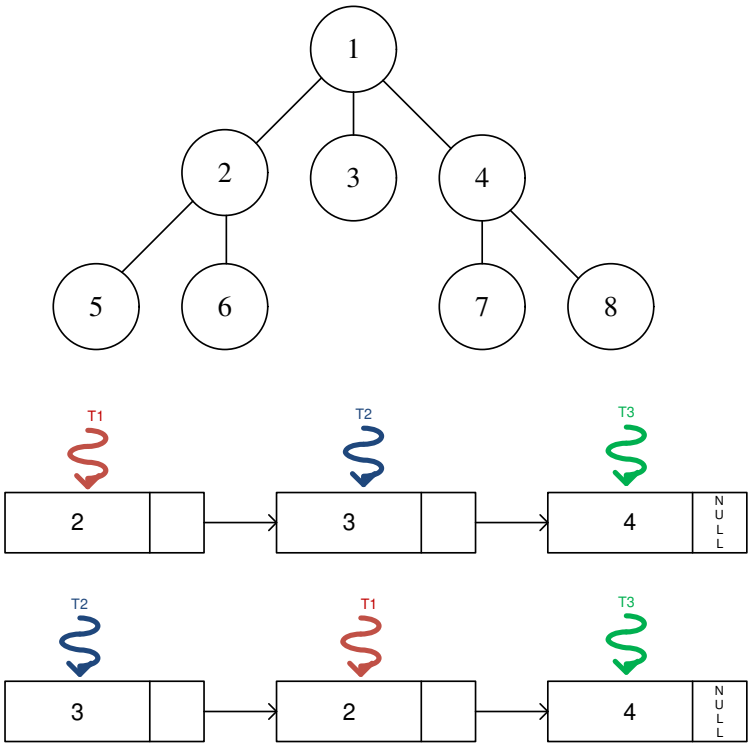


Figure 2.1. Relaxed queue for BFS.

Designing concurrent data structures for multicore systems exhibits several challenges in terms of performance and correctness. On today's machines, the layout of cores and memory, the layout of data in memory, and the communication load on the different elements of the multicore architecture all affect performance. Algorithmic improvements that seek to enhance performance often make it more difficult to design and verify a correct data

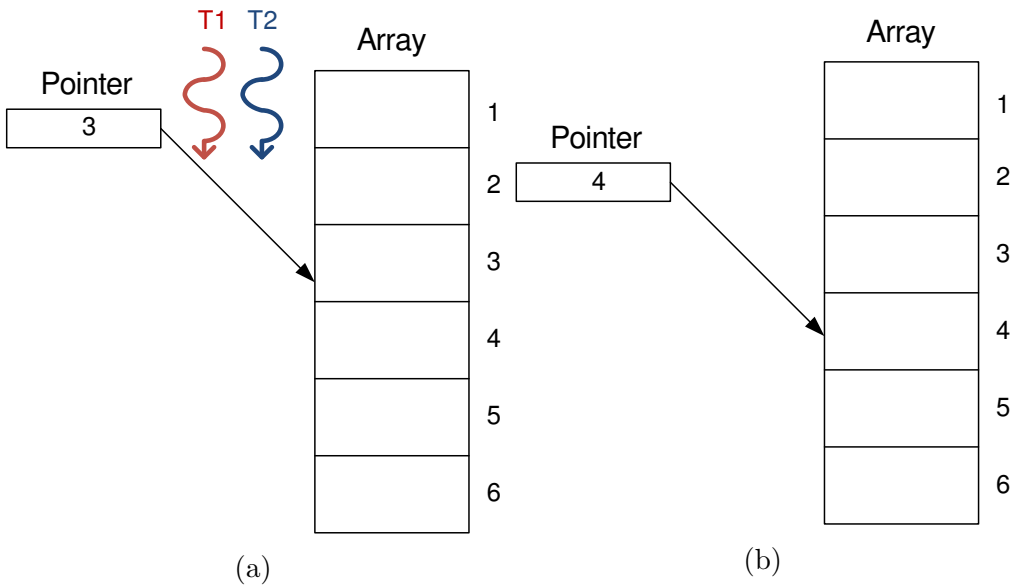


Figure 2.2. Bad interleaving: (a) Two threads want to increment the pointer (b) Two threads updated the pointer but the result is incorrect.

structure implementation. Figure 2.2 shows an example of incorrect behavior in concurrent data structures that is called *bad interleaving*. Suppose we wish to increment a pointer to refer to the next element in a shared array. If we allow concurrent increments of the pointer by multiple threads, this implementation behaves incorrectly. Suppose that the pointer initially refers to the element number 3, and two threads run on different cores concurrently want to increment the pointer as shown in Figure 2.2a. Then there is a risk that both threads read 3 from the pointer, and therefore both store 4. As you can see in Figure 2.2b, this is clearly incorrect because the pointer must refer to the element number 5 instead of 4 at the end.

Based on the synchronization mechanism, concurrent data structures are categorized into two strategies: *Blocking* and *Non-blocking*. Blocking approaches prevent bad interleavings by using a *mutual exclusion lock* (also known as a *mutex* or a *lock*). A *lock* is a construct that, at any point in time, is unowned or is owned by a single work-item. If a work-item W1 wishes to acquire ownership of a lock that is already owned by another work-item W2, then W1 must wait until W2 releases the ownership of the *lock*. While it is easy to achieve a

correct shared data structure this way, this simplicity comes with performance degradation because the *lock* suffers from *sequential bottleneck* and *memory contention*. *Sequential bottleneck* means that at any point in time, at most one operation is doing useful work. In order to reduce *sequential bottleneck*, we need to decrease the number and length of sequentially executed code sections that means decreasing the number of *locks* acquired, and decreasing *lock* granularity, a measure of the number of instructions executed while holding a *lock*. If the lock protecting our data structure is implemented in a single memory location, as many simple locks are, then in order to acquire the lock, a work-item must repeatedly try to modify that location that causes *memory contention*. Blocking concurrent data structures needs to be designed efficiently and correctly in order to avoid *deadlocks*. Also, no completion is guaranteed in blocking approach [14].

For non-blocking approach, there are several different types of completion guarantees that can be assured. The two well-known ones are *wait-free* and *lock-free*. Wait-free synchronization ensures that all the operations finally complete after a finite number of processing steps. Lock-free synchronization guarantees that some of the operations will complete after a finite number of processing steps. Wait-free is a stronger non-blocking guarantee of progress than lock-free, and lock-free in turn is stronger than blocking. As stronger progress conditions seem desirable, implementations that make weaker guarantees have generally easier design and verification. Non-blocking algorithms for several work-items need the use of atomic primitives, such as *Compare-And-Swap (CAS)*. The CAS operation atomically reads from a memory location, compares the value read to a given value, and if the comparison succeeds then swaps the old value with the new value. A non-blocking approach has many of the same disadvantages that the blocking approach has like *sequential bottleneck* and *memory contention* for a single location. Many non-blocking algorithms may suffer from *ABA* problem. The *ABA* problem occurs when a work-item reads a location twice and another work-item runs between the two reads and modifies the data structure, does other work, then modifies the data structure back, thus the first thread thinks that nothing has been modified.

As a scenario, suppose multiple concurrent work-items all attempt a dequeue operation that removes the first element, located in node A, from the queue by using a CAS to redirect the front pointer to point to a previously-second node B. The problem is that it is possible for the queue to change completely just before a specific dequeue operation attempts its CAS, so that by the time it does attempt it, the queue has the node A as the first node as before, but the rest of the queue including B is in a completely different order. This CAS of the front pointer from A to B may now succeed, but B might be anywhere in the queue and the queue will behave incorrectly [14].

2.2.2 OpenCL

All our code is written in OpenCL C++ programming language. A detailed introduction to OpenCL can be found in [16]. OpenCL targets a parallel computing platform for heterogeneous systems consisting of CPUs, GPUs, and other processors. The OpenCL platform model includes a host connected to one or more OpenCL devices each of which is composed of certain number of Compute Units (CUs) and further Processing Elements (PEs) as shown in Figure 2.3. An OpenCL application begins its execution on a host and puts device commands in the queue to communicate with device. The PEs in a CU run a single stream of instructions as SIMD units. The OpenCL program is composed of two parts: a host program that runs on the host and kernels that run on the devices. The declaration of kernel functions must be preceded by *__kernel*. The host program describes the context for the kernels and controls their execution. When the host launches a kernel for execution, a thread index space (called an NDRange) is configured. An instance of the kernel is mapped to each thread (called work-item) in the NDRange. The command *get_global_id (dim)* returns the unique global work-item ID value for dimension specified by *dim*. Each work-item runs the same code but uses possibly different data. The scheduler assigns each workgroup (a group of work-items defined by programmer) to a CU until all work-items have been executed as shown in Figure 2.4. Workgroups are composed of work-items. AMD GPUs execute on

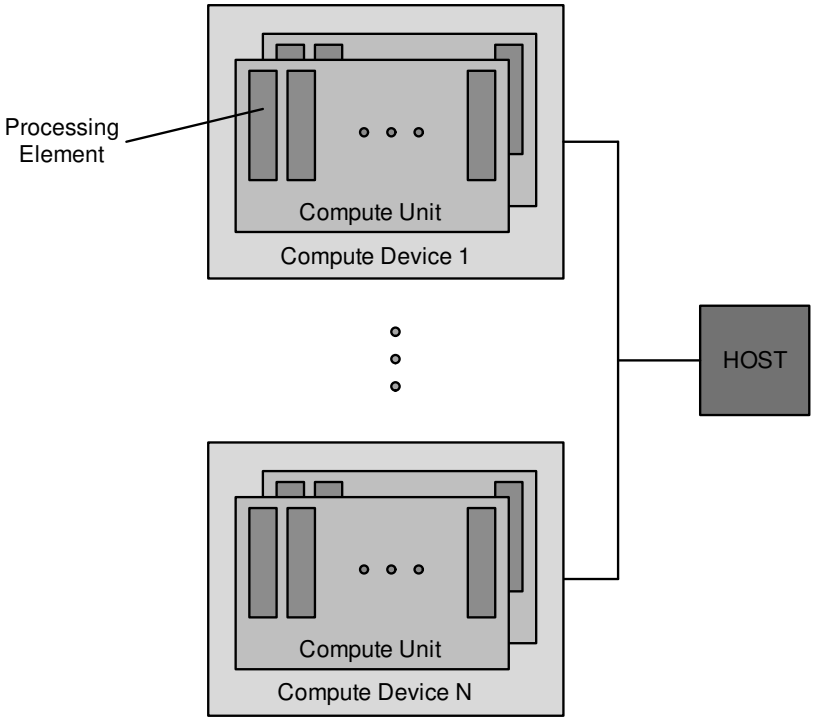


Figure 2.3. OpenCL platform [17].

wavefronts (group of work-items) while each workgroup consists of an integer number of wavefronts as shown in Figure 2.5. Work-items in the same wavefront executed in lock-step in a compute unit. If a conditional branch causes some of threads to diverge from the rest, the remaining threads must wait for the divergent threads to finish. Different workgroups must communicate with each other through global memory. The only method to implement synchronization among arbitrary threads in a NDRange is through the atomic operations that are running in global memory.

2.2.3 Atomic Operations on GPUs

We end this section with a short explanation on atomic operations that we use in this thesis. We use two atomic operations offered by OpenCL, namely, *atomic_cmpxchg* and *atomic_xchg*, to implement our blocking and non-blocking queues. The *atomic_cmpxchg* function takes three arguments, namely, a pointer p, a comparable value *cmp*, and a new value *val*. It reads the value *old* at location pointed by p. If *old* equals *cmp*, it stores *val* at location pointed by

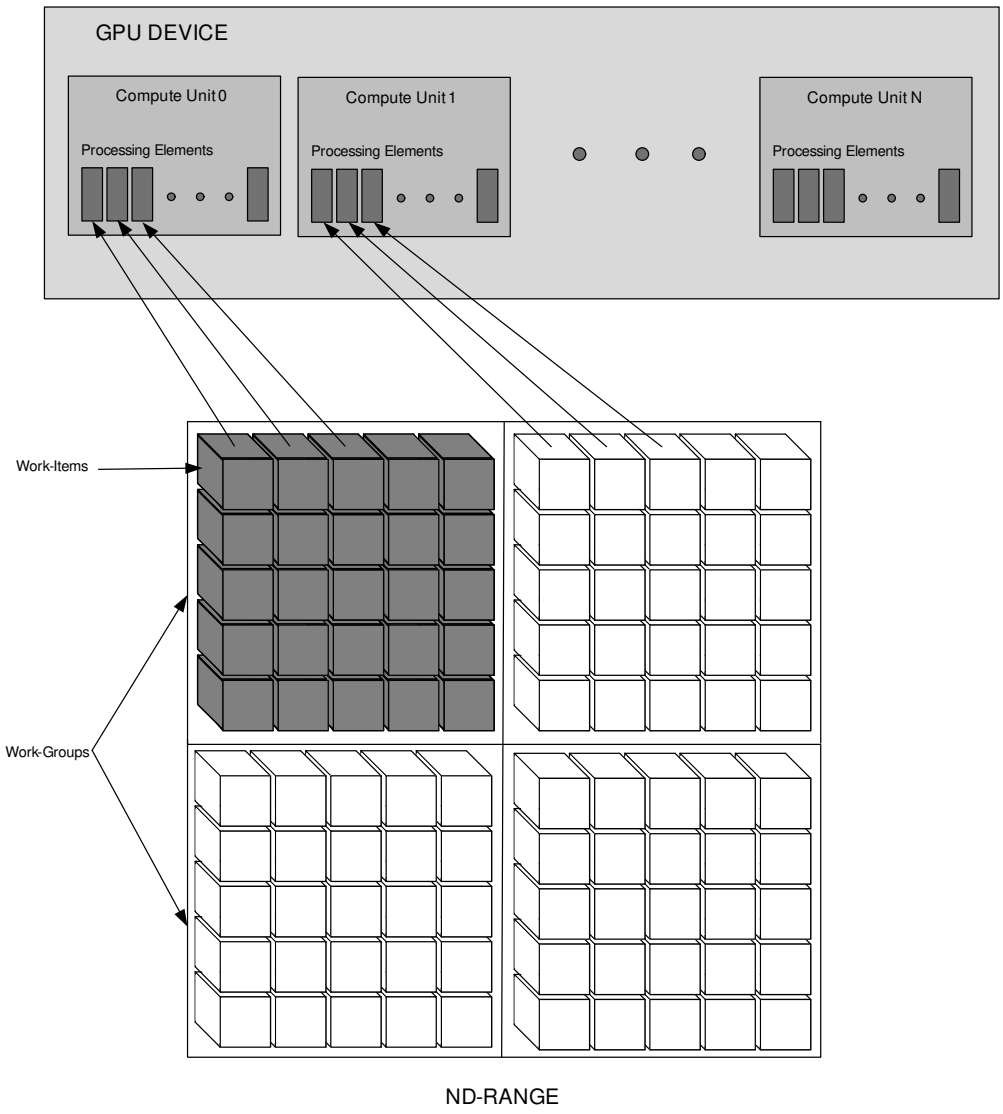


Figure 2.4. GPU thread scheduler assigns each workgroup to a CU [17].

p; otherwise it leaves the contents of p unchanged. It always returns *old*. By comparing the return value with *cmp*, one can check if the execution of *atomic_cmpxchg* has successfully stored *val*. An *atomic_cmpxchg* function of a work-item W_1 to pointer p may fail if some other work-item W_2 updates the contents of p with a value different from *cmp* of W_1. The *atomic_xchg* function takes two arguments, namely, a pointer p and a new value *val*. It swaps atomically the value *old* at location pointed by p with *val*. It always returns *old*.

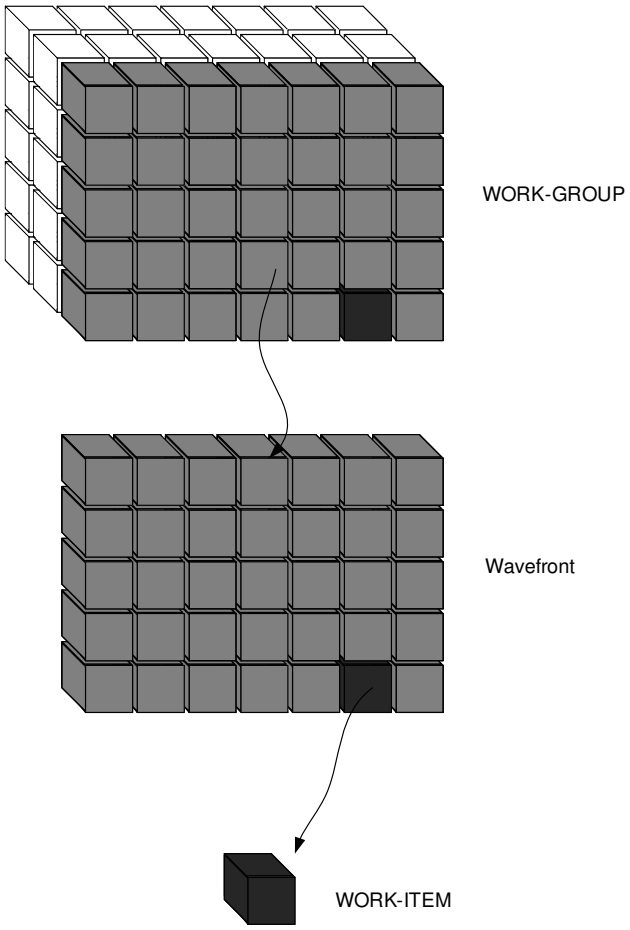


Figure 2.5. Work-group, Wavefront and Work-item [17].

CHAPTER 3

DESIGN AND IMPLEMENTATION

In this chapter we present our blocking and non-blocking concurrent queues that are based on linked list data structures. It is written in OpenCL and designed for executing on modern GPUs supporting atomic compare-and-swap operations. The queue in both implementations is created by nodes, each including two fields: *next*, a pointer to the next node in the queue, and *value*, the data value stored in the node. Two global pointers, *Front* and *Rear*, point to the front and rear nodes on the list that are used to locate the correct node when dequeuing and enqueuing, respectively. Figure 3.1 shows an example of *bad interleaving* in concurrent queues. Suppose we wish to increment the *Front* pointer to refer to the next element in a shared queue. If we allow concurrent increments of *Front* pointer by multiple work-items, this implementation behaves incorrectly. Suppose that the *Front* pointer initially refers to the first element, and two threads run on different cores concurrently want to increment the *Front* pointer as shown in Figure 3.1a. Then there is a risk that both threads read 0 from the *Front* pointer, and therefore both store back 1. As you can see in Figure 3.1b, this is clearly incorrect because the pointer must refer to the element number 2 instead of 1 at the end.

3.1 Blocking Algorithm

A general way to implement a concurrent queue is to use a *lock*. At any point in time, a *lock* is unowned or owned by a single work-item in order to guarantee mutually exclusive access to

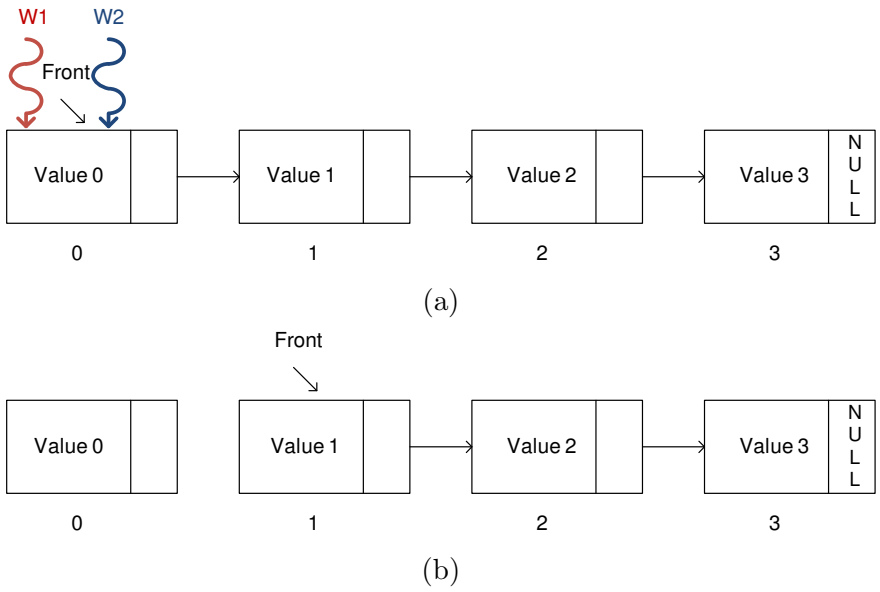


Figure 3.1. Bad interleaving in queues: (a) Two threads want to increment the *Front* pointer (b) Two threads updated the *Front* pointer but the result is incorrect.

a queue. If a work-item W_1 wants to acquire ownership of a *lock* that is already is owned by another work-item W_2, then W_1 must wait until W_2 releases ownership of the *lock*. We must be careful when using *locks* in GPUs because they can easily result in a *SIMD Deadlock* easily. *SIMD Deadlock* is due to a structural conflict among work-item synchronizations and SIMD-lockstep execution when the work-items are from the same wavefront. In this kind of deadlock, the work-item that acquired the lock will wait at the convergence point for the remaining work-items to join in order to proceed to execute the unlock instruction, whereas the remaining work-items are waiting to acquire the lock before they can step to the convergence point and this inter-waiting causes a deadlock.

We implement two functions *Acquire* and *Release* by using the two synchronization primitives *atomic_cmpxchg* and *atomic_xchg* to atomically change the *lock* from unowned to owned and vice versa. In the *Acquire* function, each work-item reads the value of *lock*. If the value of *lock* equals 0, it means that the *lock* is unowned. Then, it stores 1 at location pointed by *lock* and changes the *lock* from unowned to owned. Otherwise, it leaves the contents of *lock* unchanged. In both cases, it returns the old value of *lock* that can be 0 or

1. By comparing the return value with 0, we can check whether the work-item could acquire the *lock* or not. In the *Release* function, the work-item that acquired the *lock* changes the *lock* from owned to unowned by atomically swapping the value of *lock* with 0. The following pseudo-code shows these functions:

```
Acquire (*Lock) {
  return atomic_cmpxchg (*Lock, 0, 1) == 0;
}
```

```
Release (*Lock) {
  atomic_xchg (*Lock, 0);
}
```

We present a blocking queue by having separate *locks* for the *Front* and *Rear* pointers of a linked-list-based queue. Separate *locks* allow *enqueue* and *dequeue* operations to run simultaneously. In this approach, we need a *dummy* node in order to prevent acquiring both *Front* and *Rear* locks when the queue is empty and therefore it avoids *deadlock*. *Front* always points to the *dummy* node. We support three operations on the queue, namely, *Initialize*, *Enqueue* and *Dequeue*. The *Initialize* function creates a queue with a dummy node. The responsibility of *Enqueue* and *Dequeue* functions are the addition and removal of entities to and from the rear and front positions respectively. A work-item inside *Enqueue* and *Dequeue* tries to acquire the *LockR* and *LockF* respectively. If the work-item fails, it repeatedly tries to acquire the lock since the lock will be released soon by the work-item that acquired the lock. The variable q shows the index value after *Rear* pointer. The following pseudo-code shows our blocking queue implementation.

```
__kernel void Initialize (Q, q) {
  Q [0].next = NULL;
  Front = &Q[0];
  Rear = &Q[0];
  q = q+1;
}
```

```
__kernel void Enqueue (Q, q, value) {
  int idx = get_global_id(0);
  do {
      if (Acquire (&LockR)) {
        Q [q].data = value [idx];
        Rear->next = &Q[q];
        Rear = &Q[q];
        FlagR [idx] = 1;
        q = q+1;
        Release (&LockR);}
  }while (FlagR [idx] != 1);
}
```

```
__kernel void Dequeue (Q, pvalue) {
  int idx = get_global_id(0);
  do {
      if (Acquire (&LockF)) {
       if (Front->next == NULL) {
        FlagF [idx] = 1;
        error;}
       else {
        pvalue = Front->next->data;
        Front = Front->next;
        FlagF [idx] = 1;}
       Release (&LockF);}
  }while (FlagF [idx] != 1);
}
```

For example, Figure 3.2 shows two work-items that want to add their values to the queue simultaneously. At the beginning, the queue has two nodes and q shows the index value after Rear pointer (q=*2*). The enqueue always starts by checking that *lock* is free or not. Suppose that *W1* runs the atomic operation in the *Acquire* function before *W2* in order to acquire the lock. *W1* could acquire the lock because the lock is free. At the same time, *W2* could not acquire the lock and has to wait for *W1* because the lock is held by *W1*. Then, *W1* adds its value to the queue in a node that is identified by q, updates the q and releases the lock as shown in Figures 3.2b and 3.2c. Now, *W2* could acquire the lock by calling the *Acquire* function since lock is not held by any work-items. As shown in Figures 3.2d and 3.2e, *W2* first acquires the lock, adds its value in a node that is identified by q (q=*3*), updates the q and releases the lock.

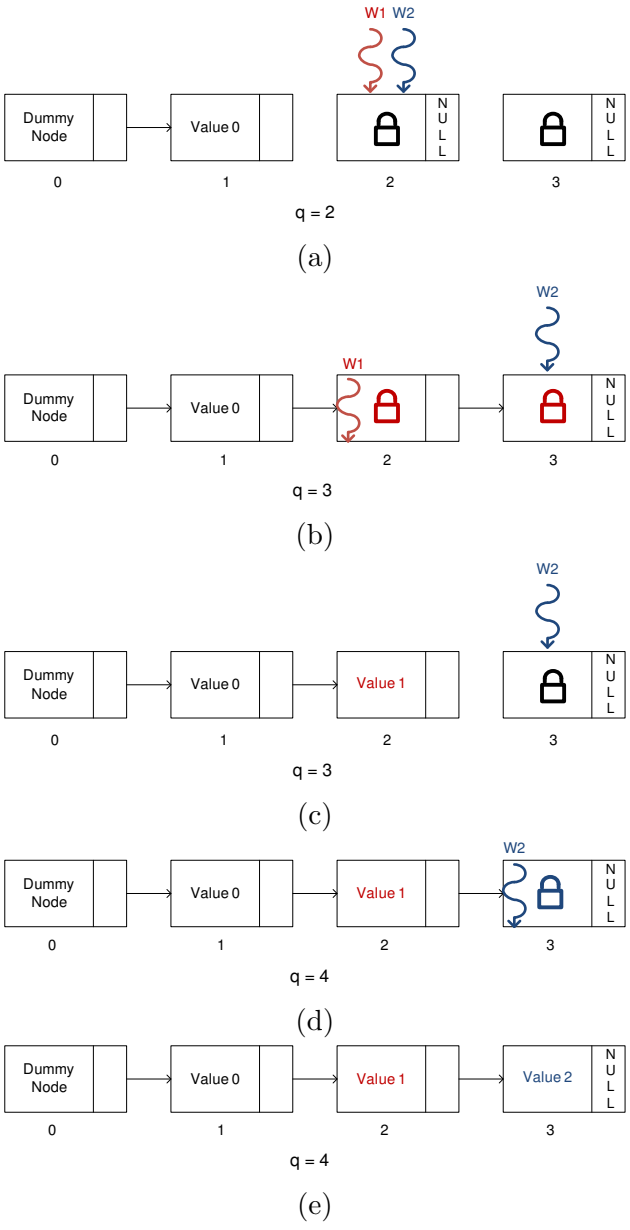


Figure 3.2. Two threads are trying to add their values simultaneously in a blocking approach: (a) Lock is free (b) Lock is acquired by *W1* (c) *W1* adds its value and update q (d) Lock is acquired by *W2* (e) *W2* adds its value and update q.

3.2 Non-blocking Algorithm

In this section, we present a non-blocking queue by using only one atomic primitives in an enqueue operation. Like our blocking algorithm, we have a dummy node at the front of queue that guarantees both *Front* and *Rear* always point at a node on the linked list. Therefore, preventing problems that occur when the queue is empty or contains just a single entity and also removes contention among enqueuing and dequeuing processes even when there is just a single entity in the queue. The variable q shows the index value of *Rear* pointer.

Like our blocking algorithm, the *Initialize* function creates a queue with a dummy node. In the *Enqueue* function, our algorithm first stores the value of general variable q in a private variable *xp*, then checks the consistency of *xp*. If the private variable *xp* of work-item W_2 was consistent, then W_2 links the new node to the end of queue and updates the *Rear* pointer. Otherwise, it means that another work-item W_1 added an entity after W_2 stores the general variable q in its private variable *xp* and W_2 needs to update its private variable *xp*. The following pseudo-code shows our non-blocking enqueue implementation.

```
__kernel void Initialize (Q) {
  Q [0].next = NULL;
  Front = &Q[0];
  Rear = &Q[0];
}
```

```
__kernel void Enqueue (Q, q, value) {
  int idx = get_global_id(0);
  do {
      xp = q;
      if (atomic_cmpxchg (&q, xp, q+1) == xp) {
       Q [xp+1].data = value [idx];
       FlagR [idx] = 1;
       Q [xp].next = &Q [xp+1];
       Rear = &Q [q];}
  }while (FlagR[idx] != 1);
```

For example, Figure 3.3 shows two work-items that want to add their values to the queue in non-blocking approach. At the beginning, the queue has two nodes, *q* shows the index value of Rear pointer (*q=1*) and two private variables of *xp* for each work-item (*xp(W1)=xp(W2)=1*). The enqueue always starts by checking the consistency of *xp*. Suppose that *W1* runs the atomic operation before *W2* in order to check the consistency of its *xp*. *W1* could come inside the if statement because both *q* and its *xp* are equal to 1 and updates the value of *q* to 2. At the same time, *W2* could not come inside the if statement because the value of *q* and its *xp* are not equal. Then, *W1* adds its value to the queue in a node that is identified by *xp(W1)+1* as shown in Figures 3.3b and 3.3c. Now, *W2* needs to update its *xp*. As shown in Figures 3.3d and 3.3e, *W2* first updates its *xp*, updates the *q* and adds its value in a node that is identified by *xp(W2)+1*.

In the *Dequeue* function, *Front* always points at the last node that was dequeued. Like the *Enqueue* function, our algorithm first stores the address of general pointer *Front* in a private variable *t*, then checks the consistency of *t*. If the private variable *t* of work-item W_2 was consistent, then W_2 updates the *Front* pointer. Otherwise, it means that another work-item W_1 removed an entity after W_2 stores the address of general pointer *Front* in its

private variable t and W_2 needs to update its private variable t. The following pseudo-code shows our non-blocking dequeue implementation.

```
__kernel void Dequeue (Q, pvalue) {
  int idx = get_global_id(0);
  do {
      t = &Front;
      if (Front->next == NULL) {
       FlagF [idx] = 1;
       error;}
      else {
       pvalue = Front->next->data;
       if (atomic_cmpxchg (&Front, t, t+1) == t){
        FlagF [idx] = 1;}
      }
  }while (FlagF [idx] != 1);
}
```

For example, Figure 3.4 shows two work-items that want to delete some values from the queue in non-blocking approach. At the beginning, the queue has four nodes and two private variables of t for each work-item ($t(W1)=t(W2)=0$). The dequeue always starts by checking the consistency of t. Suppose that *W1* run the atomic operation before *W2* in order to check the consistency of its t. *W1* could come inside the if statement because both *Front* and its t mention to the node with index 0 and updates the *Front* pointer. At the same time, *W2* could not come inside the if statement because the *Front* pointer and its t are not mention to the same node. Then, *W1* deletes a value from the queue as shown in Figures 3.4b and 3.4c. Now, *W2* needs to update its t. As shown in Figures 3.4d and 3.4e, *W2* first updates its t, updates the *Front* pointer and deletes a value from the queue.

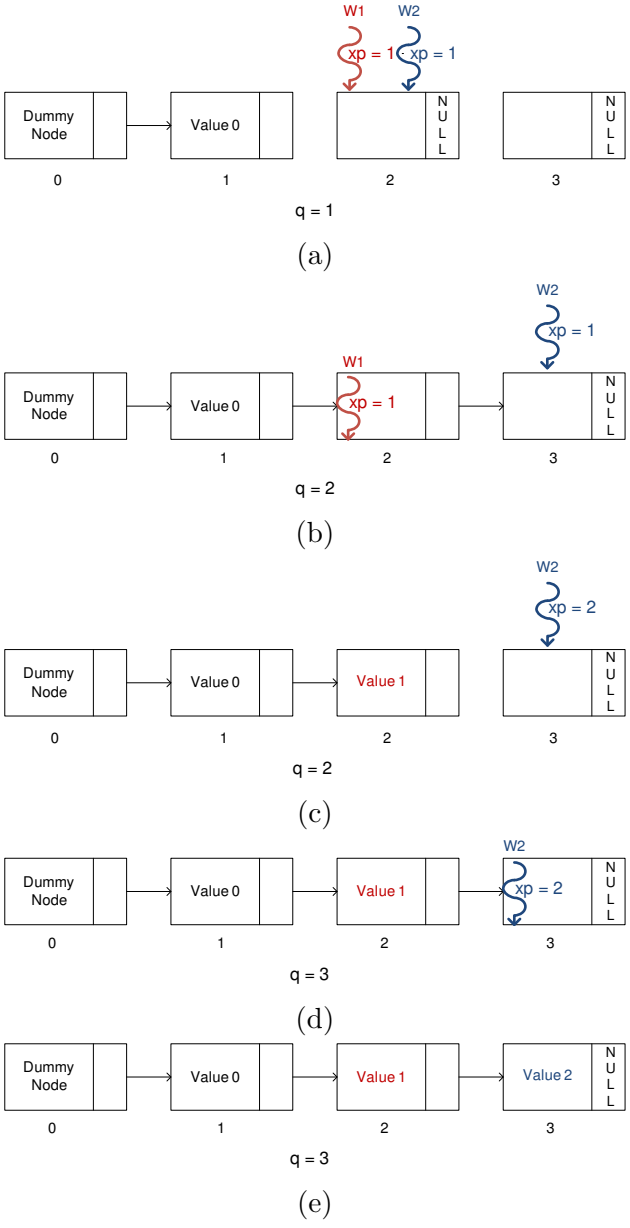


Figure 3.3. Two threads are trying to add their values simultaneously in a non-blocking approach: (a) *xp(W1)* is consistent (b) *q* is updated by *W1* (c) *W1* adds its value (d) *xp(W2) is consistent (e) W2 adds its value.*

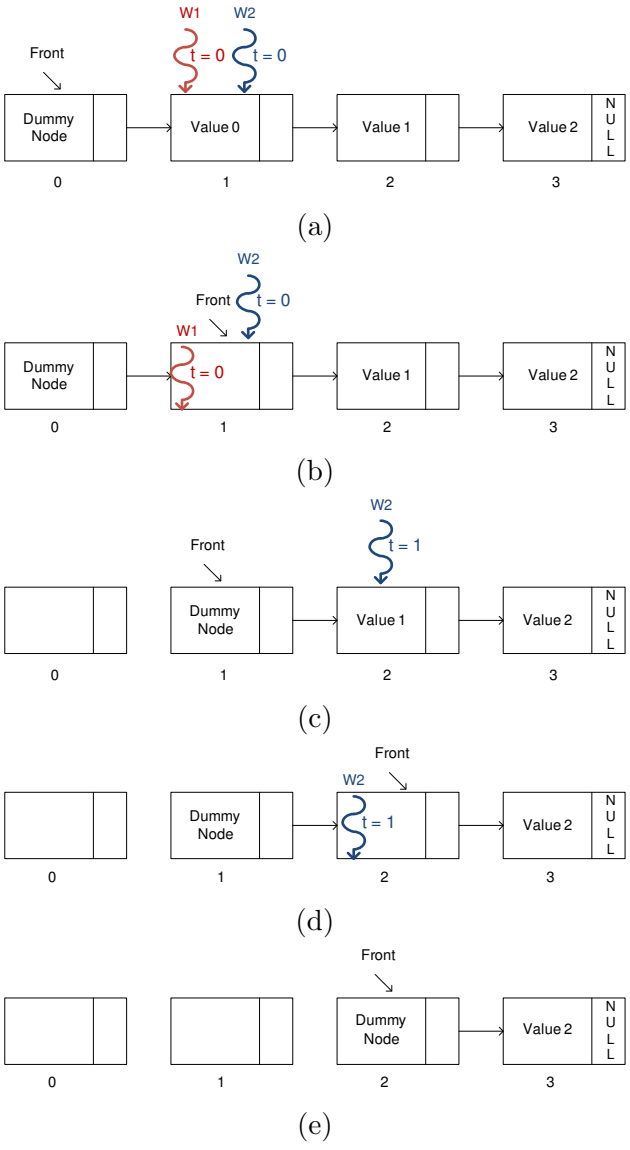


Figure 3.4. Two threads are trying to delete some values simultaneously in non-blocking approach: (a) *t(W1)* is consistent (b) *Front* is updated by *W1* (c) *W1* deletes a value (d) *t(W2) is consistent (e) W2 deletes a value.*

Our non-blocking algorithm does not suffer from ABA problem because when a work-item reads a location twice and another work-item runs between the two reads and modifies the data structure, does other work and been modifies the data structure back, then the first thread observes that the data structure has been modified. Let's consider the aforementioned scenario. Suppose multiple concurrent work-items all attempt a dequeue operation that removes the first element, located in node A, from the queue by using an atomic_cmpxchg to redirect the front pointer to point to a previously-second node B. It is possible for the queue to change completely just before a specific dequeue operation attempts its atomic_cmpxchg, so that by the time it does attempt it, the queue has the node A as the first node as before, but the rest of the queue including B is in a completely different order. This atomic_cmpxchg of the front pointer from A to B does not succeed because the private variable t and front pointer do not match and the work-item has to update its private variable t. Therefore, the queue will behave correctly.

CHAPTER 4

EXPERIMENTAL RESULTS

4.1 Experiments Setup

We test our blocking and non-blocking concurrent queue implementations on a AMD Radeon R7 APU. The APU has 12 compute units including 8 compute units on the GPU device and 4 compute units on the CPU device. The maximum clock frequencies of the GPU and CPU devices are 720 MHz and 3.5 GHz, respectively. The maximum work group size is 256 for the GPU device and 1024 for the CPU device. The OpenCL C programming implements the atomic operations on 32-bit signed and unsigned integers to locations in *__global* and *__local* memory spaces.

4.2 Performance Evaluation and Analysis

We use a micro-benchmark and real-world application as a benchmarking workload.

4.2.1 Micro-benchmark

The performance of blocking and non-blocking queues for a fixed number of work-items may depend on the combination of operations and the total number of operations. We evaluate each approach for a number of different combination of operations. In our micro-benchmark, we demonstrate each different operation combination as a pair [x, y], where the operation stream has x% *add* and y% *delete* operations. For each operation combination, we change the total number of operations from 10,000 to 100,000 in steps of 10,000. Also, we examine

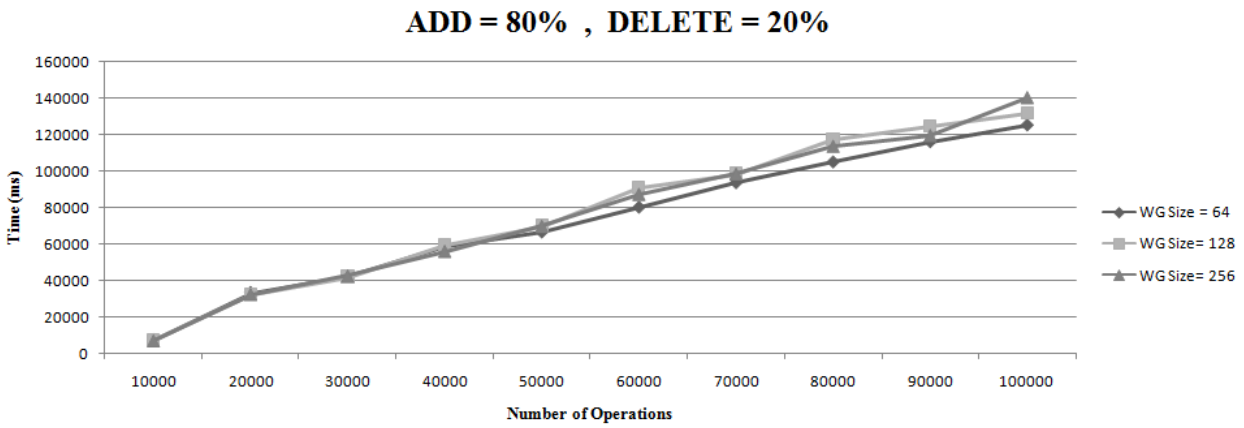
different number of work-items per work group, 64, 128 and 256, to show the influence of work group size on execution time.

We measure the performance of blocking and non-blocking concurrent queues on two different types of operation combinations. One is unbiased and has 50% add and 50% delete operations, while the other one is add-dominated and has 80% add and 20% delete operations. The performance of each approach is measured on three different work group sizes and ten different operation counts.

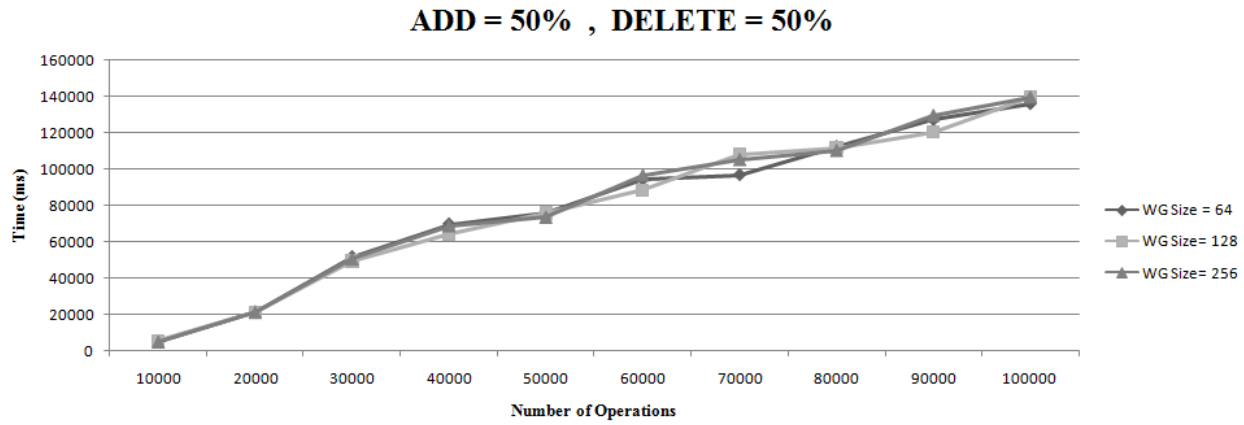
Figure 4.1 shows the performance results for our blocking concurrent queue on GPU. Figure 4.1a shows the results for an input operation with 80% add and 20% delete operations, while Figure 4.1b has equal combination of add and delete. Also, Figure 4.2 shows the performance results for the non-blocking approach while Figure 4.2a and 4.2b show the result for 80% add, 20% delete and 50% add, 50% delete combination respectively. Results show that non-blocking implementation outperforms blocking implementation significantly across different number of operations.

As the operation increases, more add and delete operations involve more atomic operations. As a result, the speedup diminishes due to the overhead of atomic operations and complicated control flow of the implementation. Nonetheless, the [80, 20] and [50, 50] combinations still benefit a speedup of nearly 7 and 5 respectively with hundred thousand operations as shown in Table 4.1 and 4.2. Also, we observe that work group size does not affect the speedup much due to the sequential bottleneck problem in both blocking and non-blocking algorithms and we cannot benefit from a bigger work group size with more thread parallelism.

Interestingly, we see that as the percentage of add operations increases, the speedup also increases (compare the upper panels with the lower panels in Figure 4.1 and 4.2). This is because with more add operations, the required number of control flow for queue modification decreases resulting relatively less number of thread divergence. This is the reason for better performance in 80% add, 20% delete scenario. Overall, the best speedup obtained by the

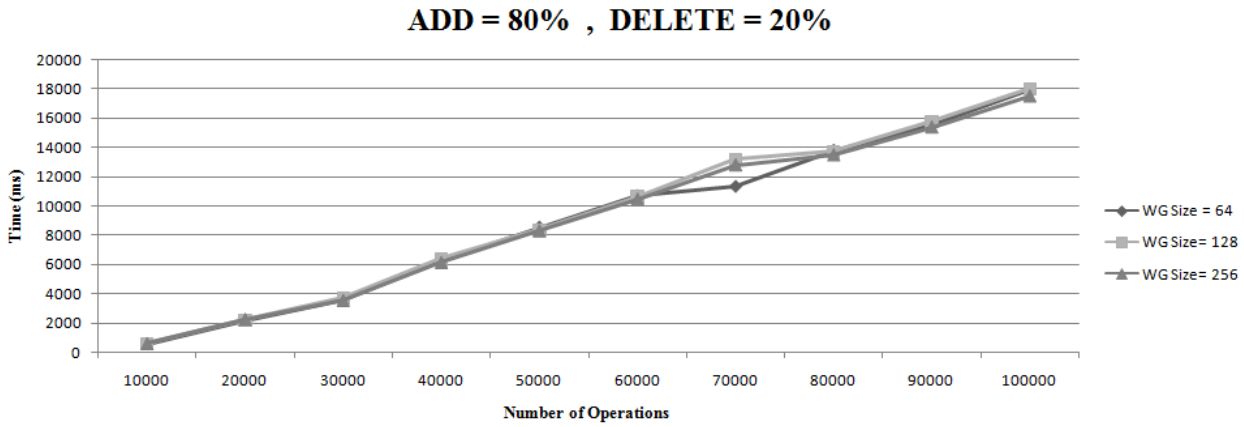


(a)

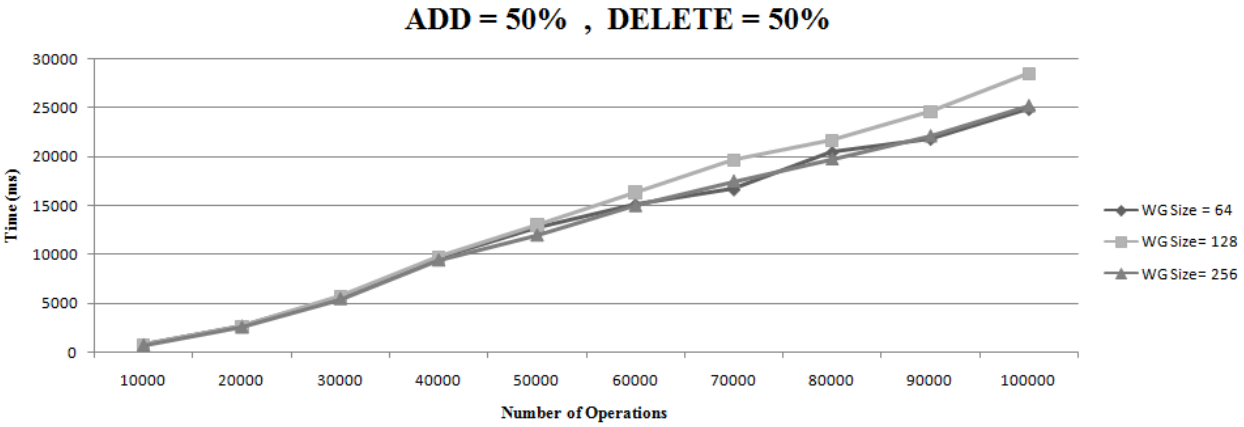


(b)

Figure 4.1. Blocking Algorithm: (a) Add = 80%, Delete = 20% (b) Add = 50%, Delete = 50%.



(a)



(b)

Figure 4.2. Non-blocking Algorithm: (a) Add = 80%, Delete = 20% (b) Add = 50%, Delete = 50%.

Table 4.1. Speedup for 80% Add, 20% Delete.

Number of Operations	WG Size = 64	WG Size = 128	WG Size = 256
10,000	13.2	12.5	12.1
20,000	15.1	14.9	14.5
30,000	11.6	11.2	12
40,000	9.3	9.3	9.1
50,000	7.7	8.3	8.3
60,000	7.4	8.5	8.3
70,000	8.2	7.4	7.6
80,000	7.5	8.5	8.3
90,000	7.4	7.8	7.7
100,000	6.9	7.2	7.9

Table 4.2. Speedup for 50% Add, 50% Delete.

Number of Operations	WG Size = 64	WG Size = 128	WG Size = 256
10,000	6.8	7.1	7.2
20,000	7.9	8.1	8.4
30,000	9.1	8.6	9.2
40,000	7.2	6.5	7.3
50,000	6	5.8	6.1
60,000	6.2	5.4	6.4
70,000	5.8	5.4	5.9
80,000	5.4	5.1	5.5
90,000	5.8	4.8	5.8
100,000	5.4	4.8	5.5

non-blocking implementation is 15.1 higher compared to the blocking implementation.

4.2.2 Breadth First Search (BFS)

Queue is used when data do not need to be processed right away, but need to be processed in FIFO order like Breadth First Search (BFS). For a graph G = (V, E) and a root vertex s, BFS traverses the edges of G to explore every vertex that is reachable from s. BFS proceeds in the following steps:

- Step 1: Visit the adjacent unvisited vertex and enqueue it in a queue.
- Step 2: If no adjacent vertex remains, dequeue the first vertex from the queue.
- Step 3: Repeat Step 1 and Step 2 until the queue is empty.

Table 4.3. Number of nodes and edges in each road networks.

Road Network	Number of Nodes	Number of Edges
Pennsylvania	1,088,092	1,541,898
Texas	1,379,917	1,921,660
California	1,965,206	2,766,607

BFS is a graph algorithm that has wide applications in different fields and can benefit from GPU acceleration. Therefore, concurrent queues play a significant role in BFS algorithm on GPUs. In this section, we want to compare our blocking and non-blocking concurrent queues by using BFS as an application while our evaluation is done on GPU.

4.2.2.1 Input graph data

We need a high performance system for analysis and manipulation of large networks as an input graph data. The system must be optimized for maximum performance and compact graph representation and easily scales to massive networks with hundreds of millions of nodes, and billions of edges. It needs to efficiently manipulate large graphs, calculates structural properties, generates regular and random graphs, and supports attributes on nodes and edges. Moreover, edges and attributes in a graph or a network need to be changed dynamically during the computation.

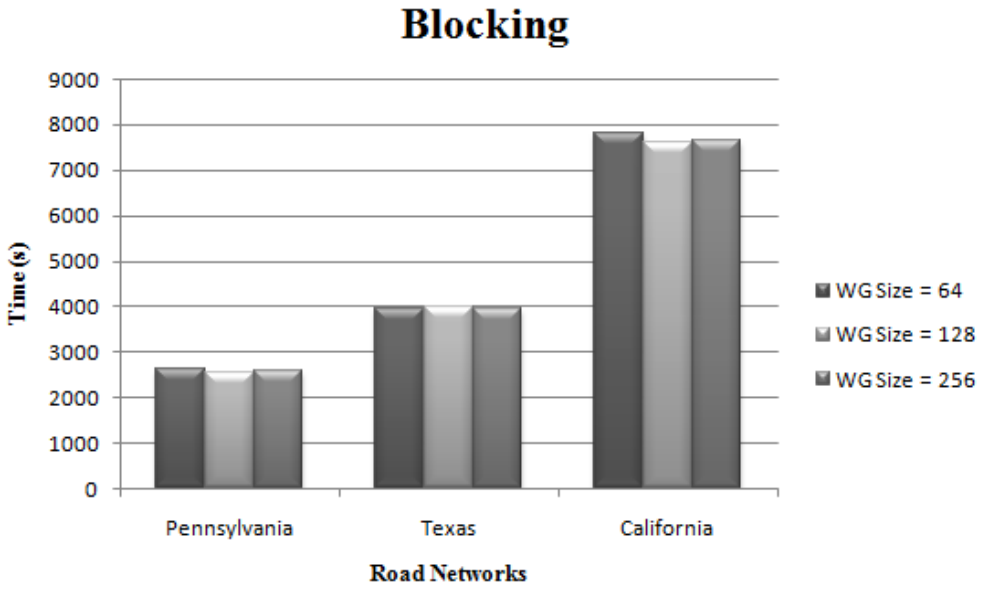
We use the Stanford Large Network Dataset Collection (SNAP) library that is developed as a result of some research in analysis of large social and information networks. We measure the performance of blocking and non-blocking concurrent queues on road networks [8] consisting of Pennsylvania, Texas and California road networks as an input graph for our BFS. The road network indicates intersections and edges roads connecting the intersections. Intersections and endpoints are indicated by nodes and the roads connecting these intersections or road endpoints are indicated by undirected edges. Table 4.3 shows the number of nodes and edges of each road network.

4.2.2.2 Performance results

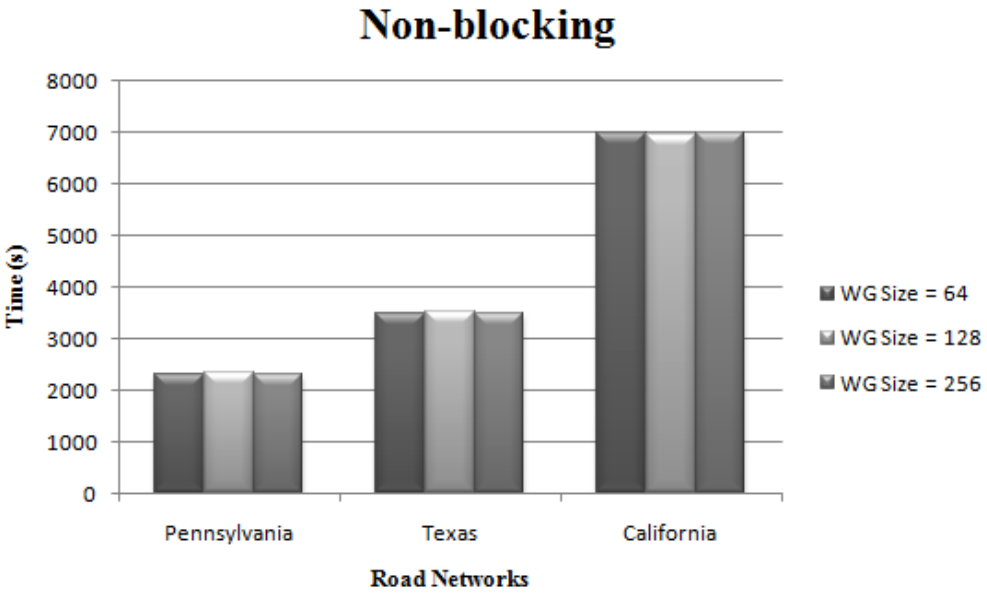
Figure 4.3a and 4.3b show the performance results for our blocking and non-blocking concurrent queues on GPU. Our queue does not show a considerable speedup on the non-blocking implementation in comparison to the blocking implementation for all road networks as much as we saw earlier in the micro-benchmark. This is because the non-blocking implementation could show better performance than blocking just when there is sufficient data-level parallelism during each addition and deletion while data-level parallelism in BFS is low. Also, the non-blocking implementation shows the same scalability compared to the blocking one. As mentioned earlier, work group size does not affect the speedup too much due to the sequential bottleneck problem in blocking and non-blocking algorithms and we cannot benefit from a bigger work group size with more thread parallelism. Overall, the best speedup obtained by the non-blocking implementation is just around 1.1x compared to the blocking implementation in our BFS algorithm.

Table 4.4. Speedup for BFS Algorithm.

Number of Operations	WG Size = 64	WG Size = 128	WG Size = 256
California	1.12	1.09	1.10
Pennsylvania	1.14	1.09	1.12
Texas	1.13	1.13	1.12



(a)



(b)

Figure 4.3. BFS Result for Road Networks: (a) Blocking (b) Non-blocking.

CHAPTER 5

CONCLUSION

This study evaluates the performance of blocking and non-blocking concurrent queues on AMD Radeon R7 GPU. Both implementations are built upon the array based linked list implementation. The non-blocking implementation consistently shows better performance compared to the blocking implementation for carrying out addition, deletion, and search operations on various number of operations. Our evaluation shows that for sufficient thread-level parallelism, the non-blocking implementation outperforms (up to 15.1) the blocking implementation. For insufficient thread-level parallelism, a concurrent queue does not benefit much from the non-blocking implementation due to the underutilization of hardware resources. The non-blocking concurrent queues obtain higher speed up (up to 13.2) with the presence of sufficient thread-level parallelism compared to the insufficient thread-level parallelism.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012.

[2] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52. ACM, 2008.

[3] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. *Principles of Distributed Systems*, pages 302–317, 2010.

[4] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[5] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. *Principles of Distributed Systems*, pages 401–414, 2007.

[6] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. *Distributed Computing*, 20(5):323–341, 2008.

[7] Patrick PC Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 78–79. ACM, 2009.

[8] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[9] John M Mellor-Crummey. Concurrent queues: Practical fetch-and-phi algorithms. Technical report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1987.

[10] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[11] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[12] Changwoo Min, Hyung Kook Jun, Won Tae Kim, and Young Ik Eom. Scalable cache-optimized concurrent fifo queue for multicore architectures. *IEICE TRANSACTIONS on Information and Systems*, 95(12):2956–2957, 2012.

[13] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262. ACM, 2005.

[14] Mark Moir and Nir Shavit. Concurrent data structures., 2004.

[15] Thomas Preud'Homme, Julien Sopena, Gael Thomas, and Bertil Folliot. Batchqueue: Fast and memory-thrifty core to core communication. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 215–222. IEEE, 2010.

[16] AMD Accelerated Parallel Processing. Software development kit (sdk). *URL http://developer. amd. com/sdks/amdappsdk*.

[17] AMD OpenCL Programming. User guide 2. *URL http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf*.

[18] Jeff A Stuart and John D Owens. Efficient synchronization primitives for gpus. *arXiv preprint arXiv:1110.4623*, 2011.

[19] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[20] R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[21] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143. ACM, 2001.

[22] Shucai Xiao and Wu-chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

VITA

HOSSEIN POURMEIDANI

hpourmei@go.olemiss.edu

https://www.linkedin.com/in/hossein-pourmeidani-b3a701126/

- **Education**

  * **Master's of Science**, Computer Science, University of Mississippi, November 2018

  * **Master's of Science**, Computer Engineering, Islamic Azad University, September 2012

  * **Bachelor's of Science**, Computer Engineering, Islamic Azad University, September 2010

- **Publications**

  Conference Papers

  * H. Pourmeidany, Partial Way-Access Set-Associative Cache for Low Energy Consumption, International Conference on Computer and Information Technology, 2011.

  * M. Habibi, H. Pourmeidani, Hierarchical TCAM/TMR Defect Tolerance Technique for Nanodevice RAM Repairing, International Conference on Applied Electronics, 2012.

  * H. Pourmeidani, M. Habibi, Hierarchical Defect Tolerance Technique for NRAM Repairing with Range Matching CAM, The 21st Iranian Conference on Electrical Engineering, 2013.

  * H. Pourmeidani, M. Habibi, A Range Matching CAM for Hierarchical Defect Tolerance Technique in NRAM Structures, International Conference on Applied Electronics, 2013.

  Journal Papers

∗ M. Habibi, H. Pourmeidani, A Hierarchical Defect Repair Approach for Hybrid Nano/CMOS Memory Reliability Enhancement, Microelectronics Reliability, Vol. 54, Issue 2, pp. 475-484, 2014.

∗ H. Pourmeidani, A. Sharma, K. Choo, M. Hasan, K. Kim, M. Choi, B. Jang, Dynamic Temperature Aware Scheduling for CPU-GPU 3D Multicore Processor with Regression Predictor, Journal of Semiconductor Technology and Science, Vol. 18, No. 1, pp. 115-124, 2018.

- **Teaching Experience**

  ∗ **Teaching Assisstant** 08/2017 –12/2018

  Courses: Computer Organization and Assembly Language and Models of Computation

  ∗ **Instructor** 08/2012 –08/2016

  Course: Assembly Language, Hardware 2, Computer Programming, Graphic Laboratory, Computer Networks, Computer Networks Laboratory, Visual Basic Programming

- **Research Experience**

  ∗ **Research Assistant** 08/2016 –08/2017

To mitigate high temperatures on CPU-GPU 3D heterogeneous processors, a novel dynamic temperature-aware task scheduling approach for compute workloads using OpenCL framework is proposed in this project. Our experimental results demonstrate that the proposed scheduling technique is a viable solution to address the hotspots and heat dissipation issue of 3D stacked heterogeneous processors under reasonable performance tradeoffs.

- **Skills**

  * Softwares: Cadence IC Design, Pspice & Hspice, Ledit, Matlab, Simplescalar & Simwattch, Xilinx ISE, Gem5, Flex & Bison, Multi2Sim, McPAT, HotSpot, JFLAP

  * Languages: C/C++, OpenCL, Haskell, Python

  * Op. Systems: Linux, Windows

- **Honors**

  * Member of Iranian Microelectronics Association (IMA), 2013

  * Member of Irans Center for Integrated Circuits (ICIC), 2011

  * Ranked in Top 2% of Applicants for Entrance to Master of Computer Architecture, 2010

  * Ranked in Top 15% of Applicants for Entrance to Bachelor of Computer Engineering, 2005

  * Member of Isfahan Mathematics House, 2002

  * Taking Part in the Second Stage of Math Olympiad for Iranian Middle School Students, 2000