Quilk: A Work-Stealing Quicksort Algorithm for GPUs

TOLUWANIMI ODEMUYIWA* and SARMAD SAEED*

Additional Key Words and Phrases: work-stealing, quicksort, GPUs, load balancing

ACM Reference Format:

Toluwanimi Odemuyiwa and Sarmad Saeed. 2020. Quilk: A Work-Stealing Quicksort Algorithm for GPUs. 1, 1 (August 2020), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graphic processing units (GPUs) excel at data parallel tasks. The single instruction multiple data (SIMD) paradigm allows multiple cores to share a control flow in a GPU, giving the hardware more resources for actual computation[1]. NVIDIA's CUDA programming model capitalizes on this by breaking data-parallel tasks into kernels, which are launched as blocks mapped to cores with threads running as executable units. On the other hand, task-parallelism has traditionally been accomplished on multi-core CPUs, which are designed to handle different instruction streams or workloads per processor. On a GPU, task parallelism tends to create load imbalance: some cores end up with more work than other cores. Static load balancing is a method where all possible tasks are scheduled prior to task launch. In programming models such as CUDA, one can attempt to break down a task into multiple sub-tasks ans schedule each sub-task as a separate kernel invocation[2]. However, there is a significant pool of problems where some or all of the following is true:

(1) There is no way to know the total number of tasks that can be generated.
(2) Tasks can generate new tasks dynamically (such as recursive functions).
(3) The run time of certain tasks is unpredictable.

This work addresses points 1 and 2 above, and point 3 is left for future work. On a GPU, static load balancing under the above conditions would leave certain work units idle while others are busy, wasting hardware resources. To mitigate this, dynamic load balancing methods seek to continually schedule tasks to work units, by assigning work in such a way that all units are continually busy. Various methods have previously been explored, most of which involve some variation of a dynamic work pool to which cores can add work or retrieve work. On a GPU, this requires cross-core communication — usually in the form of atomics — to synchronize access. Note that here we are using core in a generic way, where it is simply an

*Both authors contributed equally to this research.

Authors' address: Toluwanimi Odemuyiwa, todemuyiwa@ucdavis.edu; Sarmad Saeed, mssaeed@ucdavis.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.
Manuscript submitted to ACM

execution engine. Depending on the load balancing implementation, a core can be a single thread, a warp, a block, or even a physical processor core.

On CPUs, Cilk has become a popular framework to implement load balancing across multiple cores[3]. It successfully uses *work-stealing* to schedule work across cores. In work-stealing, every core has its own work pool but can steal work from other work pools as its own work supply dwindles. In this way, no core should be left idle while there is still global work available.

In this work, we apply work-stealing to quicksort: an algorithm notorious for generating irregular workloads. We implement three stealing polices and compare the overall work distribution, runtime, and memory overhead to a recursive implementation without work-stealing. The three work stealing policies are listed below:

(1) Always steal from the neighbouring core.
(2) Steal from a random core.
(3) Always steal from a randomly assigned core (assigned prior to kernel launch).

Through this work, we hope to shed light on how work-stealing can dynamically balance irregular workloads, and the parameters that influence its load balancing ability. Specifically, we address the domain of parallel tasks where the total number of tasks generated is unknown prior to runtime and where tasks generate dependent tasks.

This report is organized as follows: We review related work in 2. We describe our base implementation in 3. In 4 we share and discuss our results. Finally we conclude and share future directions in 5.

The software is available on Github at https://github.com/Malloc26/Quilk.

2 RELATED WORK

Alora et al. describe a lock-free work-stealing algorithm that uses atomics to synchronize access[4]. In a comparison of four dynamic load balancing techniques on an irregular workload, Cederman and Tsigas determined that their GPU implementation of the lock-free work-stealing algorithm had the highest performance[5]. However, they use a fixed-array implementation and generate tasks that are not dependent on each other[2]. Tzeng, Patney, and Owens implemented a load balancing technique that uses both work stealing *and* work donation, which reduced the memory overhead of using work-stealing alone[8]. They successfully applied this method to a Reyes renderer, which traditionally has an irregular workload[8].

Quicksort is a popular sorting algorithm, considered to be one of the faster sorting algorithms on CPUs[6]. As a naturally recursive function, it poses an interesting challenge for load balancers. Given an input array and a pivot, at every stage of the algorithm, the array is recursively partitioned into smaller sub arrays. Thus, for an array of size n, in the average case scenario, about $logn$ sub-steps will be generated. Since each subtask can result in different-sized arrays, load imbalance can quickly arise[7]. There are currently two popular GPU implementations: GPU-Quicksort[9] and the CUDA dynamic parallel quicksort (CDP)[12]. GPU-Quicksort uses an iterative approach by dividing the input into sub-partitions. CDP implements quicksort by recursively calling kernels on sub sequences; it only works on GPUs that have dynamic parallelism enabled. A third method, CUDA-Quicksort, based on GPU-quicksort, uses atomic primitives to facilitate communication between blocks and achieves better performance than the previous implementations[13]. To mitigate the irregular workload nature of CDP, all three methods initially partition each sequence across multiple blocks,

and blocks must communicate with each other to determine where to place elements in global memory. Once sequences reach a size small enough to fit on a single block, each block recursively processes a sub-sequence. Depending on the sub-sequence, each block will generate a different amount of total work, introducing load imbalance. None of the three aforementioned methods attempt to load balance the algorithm at this stage.

In this work, we focus specifically on the load balancing problem. This is the central question: how can a core that continually produces new tasks share its work across other cores? We describe our approach and implementation in the following section.

3 THE QUILK IMPLEMENTATION

Since we wanted to focus on work stealing, we did not include the initial phase of work found in GPU-Quicksort, CDP, and CUDA-Quicksort, where a single sequence is partitioned across multiple cores[6, 9, 12, 13]. Instead, we focused on expanding a basic GPU quicksort implementation: a single block containing one thread that recursively calls quicksort on subsequences. This is essentially the traditional serial CPU implementation on a single GPU thread. On NVIDIA GPU devices with compute capability 5.0, dynamic parallelism allows new kernels to be launched within a parent kernel, enabling recursive implementations. Detailed implementation on the simple GPU quicksort implementation can be found in the *cdpSimpleQuicksort* example in CUDA Samples of the CUDA Toolkit Documentation. This implementation is used as the baseline for Quilk, our work-stealing quicksort implementation.

In Quilk, a worker unit is defined as a block with a single thread. This choice was a result of expanding the CUDA Toolkit *cdpSimpleQuicksort* example. A task is defined as the portions of the input data on which to perform quicksort; that is, the process of picking a pivot and moving data elements to their proper positions on either side of the pivot. Task portions are identified by a left pointer and right pointer, marking the start and end of the allotted sections of data, respectively. Tasks are placed in circular queues. A worker adds new tasks to the tail of its queue and removes tasks from the tail. To mitigate race conditions with an owner worker, stealing workers steal always steal from the head of a queue. Figure 1 shows the structure of both tasks and queues.

```
typedef struct __align__(64) task_t
{
    volatile int left; // starting point for sort
    volatile int right; // end point for sort
} task;

typedef struct __align__(128) queue_t
{
  volatile int size; //the current size
  volatile int head; //the head of the queue
  volatile int tail; //the tail of the queue
  volatile task * tasks; //Array of tasks
} queue;
```

Fig. 1. Task and Queue definitions.

Quilk has four stages: the initialization stage, the task assignment stage, the task stealing stage, and the task generation stage.

Initialization: This stage is done on the CPU host. The host allocates space for each worker's queue. The queue for worker 0 is initialized with the single task of sorting the entire input sequence. The CPU launches a certain number of persistent blocks. We varied the number of worker blocks launched to observe how load balancing behaviour changes with available workers.

Task Assignment: This stage — and all subsequent stages — is performed on the GPU device. Each worker attempts to retrieve work from the tail of its own queue. If it succeeds, it continues to the task generation stage. If it fails, it enters the stealing stage.

Task Stealing: Workers that fail to find any work in their own queue enter the task stealing phase. In this stage a worker will use a stealing policy to determine from which worker to steal. Once it has selected its victim, the worker will attempt to steal from the head of the victim queue. If successful, it will move to the task generation stage. If it fails, it will continually attempt to steal. To steal from its victim worker, a worker must first obtain a lock to prevent other workers from stealing from the same victim worker. Moreover, we added a heuristic that the queue size of the victim worker must be greater than three, this is to lessen race conditions we were observing between the owner of the queue and the stealing worker. There are three stealing polices:

- **Policy 0**: Always steal from a neighbour - specifically the one to the right.
- **Policy 1**: Steal from a random worker. If a steal attempt fails, steal from another random worker.
- **Policy 2**: In the initialization stage, the CPU randomly assigns a victim worker to each worker. During task stealing, each worker will continually attempt to steal from its assigned, random victim. If a stealing worker happens to be assigned to itself, then it can randomly steal from any processor.
- **Policy 3**: This policy does not implement work stealing. Each processor works on its own generated or pre-assigned tasks.

In the works by Arora et al. and Cederman and Tsigas, victim workers are randomly chosen[4, 5]. We wanted to explore the effects of other stealing policies on work distribution.

Task Generation: In this stage, a worker has a designated task at hand: move elements within the given boundaries to the left or right of a pivot. In traditional quicksort, this is the phase where recursive calls will be made to the subsequences generated by this task. In Quilk, new tasks are generated and the worker adds these tasks to the tail of its queue.

Quilk was implemented in CUDA. The code is available at https://github.com/Malloc26/Quilk/tree/master/basic.

3.1 Measuring Work Distribution

The primary goal of this project is to provide further insight into how various parameters affect how well work-stealing can distribute work. A key metric is a measure of worker idleness. To this end, we implemented an internal profiling system.

When enabled, the original Quilk implementation runs two streams: an "idleness checker" stream, and the regular, task-stealing Quilk stream. The Quilk stream now includes a debug array, which contains a bin for every worker, representing the number of tasks a worker has completed. Every time a worker reaches the

task generation stage, it will increment its bin count by one. In parallel, the checker stream periodically reads from this debug array. If a bin's value has not increased since the last time it was read, the checker increments an idle count for that worker. Overall, this checker implementation provides a measure of how much work each worker does, combined with a guideline on how many iterations a worker spends idle. Note that the measure of idleness is coarse-grained; if a worker is busy on a task, the checker will identify that worker as idle. Thus, this checker is a measure of how often workers are able to process new work. Future work will focus on adding a measure of the total time each worker spends on processing all its tasks, versus the total program runtime. This should provide another perspective on the idleness of each processor.

The source code with the internal profiling method is available at https://github.com/Malloc26/Quilk/tree/master/Merger.

4 RESULTS AND DISCUSSION

We ran our experiments on three machines: the Quadro 620, Tesla K40C, and the Volta 100. Our experiments aimed to answer the following questions.

How does stealing policy affect work distribution? We observe different work distributions for each policy. In policy 0, an increase in the number items to be sorted results in an increase in the number of busy workers. Worker zero always has the most amount of work, with the work trickling down to the next neighbour. This trend can be seen in figure 2 where 50 workers sort 8192 Elements. On the other hand, policy 1 randomly accesses its neighbours, resulting in the most balanced work distribution amongst workers. Policy 1 can be seen in action in Fig. 3.

In policy 2, work is assigned to a small subset of workers, as in policy 0. Since each worker can only steal from a set worker, it will take a longer time — when compared to policy 1 — for work to be trickle out to all the workers. This can be seen in figure 4. Finally, in policy 3, one worker processes all the work while all the remaining workers are idle, as shown in Fig. 5.

Moreover, as the number of elements to be sorted increases for policy 0, more workers are engaged. Fig. 6, shows this effect as the number of elements to be sorted increases, less workers are idle.

Similarly, as the number of workers increases, but the number of items to be sorted remains constant, more workers remain idle. In figure 7, both the Snake (Quadro 620) and the MC (Tesla K40) machines are executed at their maximum occupancy capacity for the same number of elements (8192). The Tesla K40 has more idle workers than the Quadro 620, since it has more workers for work to fill. Overall, policy 1 has the best work distribution, since tasks are always being picked up because of random stealing. Policy 2 has similar work distribution results across all three machines, since work distribution is clustered around the starting worker.

261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312

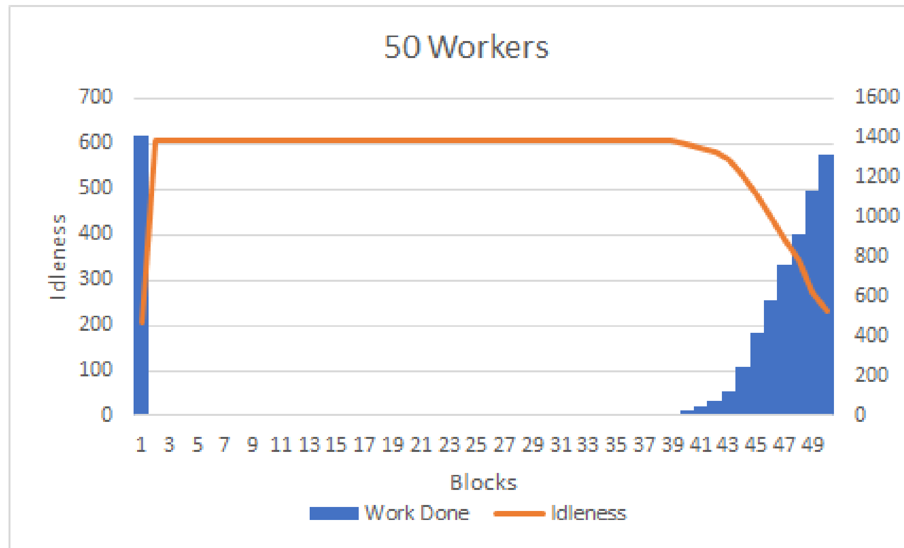

Fig. 2. Work Distribution for Policy 0

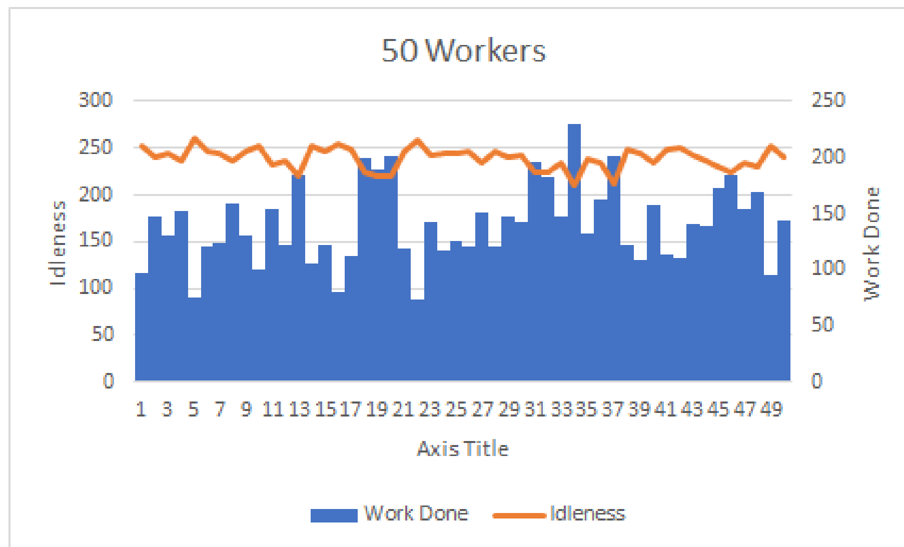

Fig. 3. Work Distribution for Policy 1

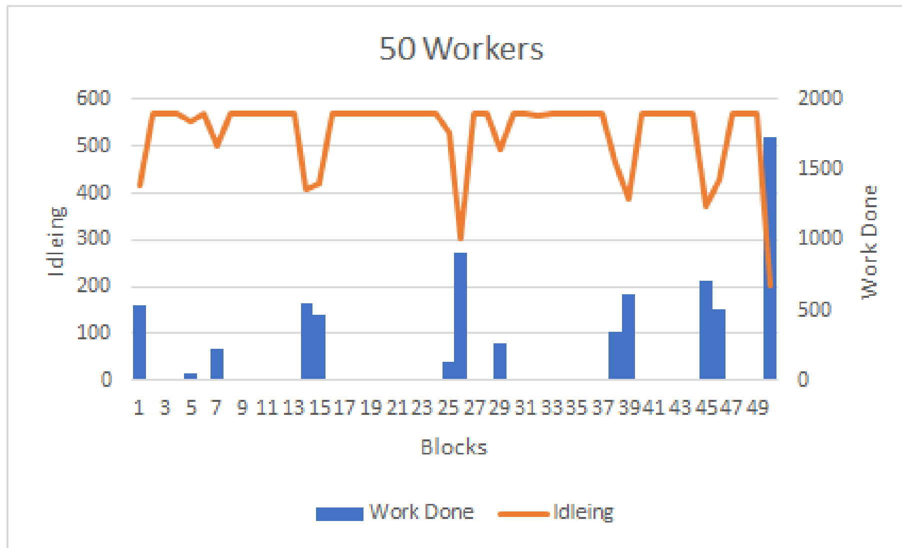

Fig. 4. Work Distribution for Policy 2

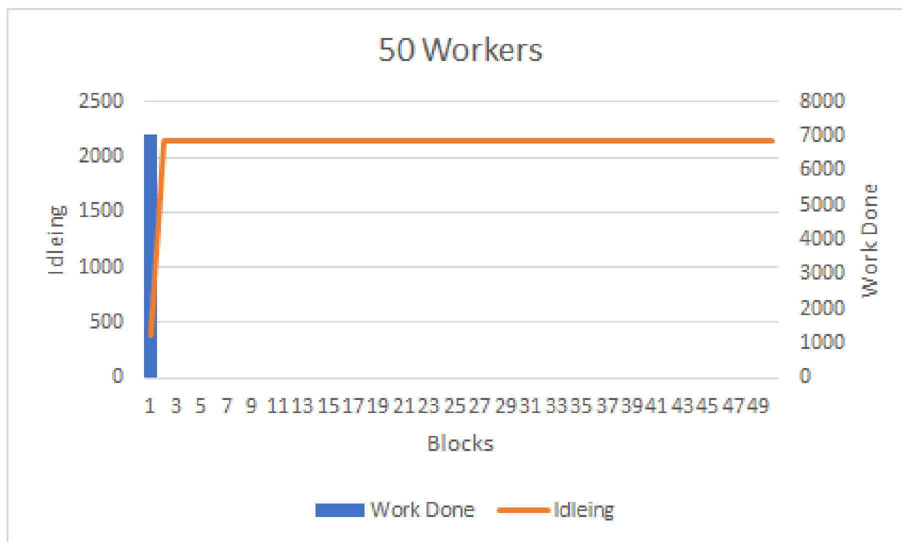

Fig. 5. Work Distribution for Policy 3

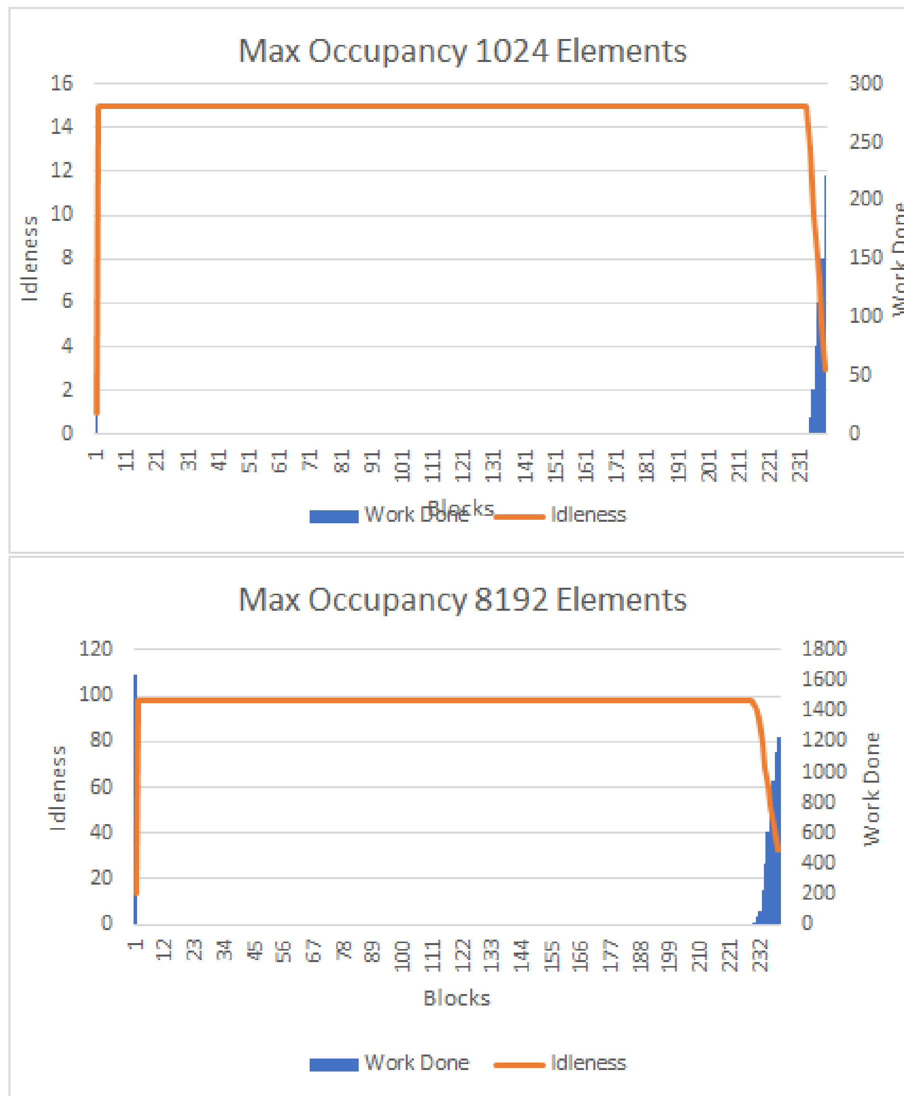

Fig. 6. Increasing the number of work items.

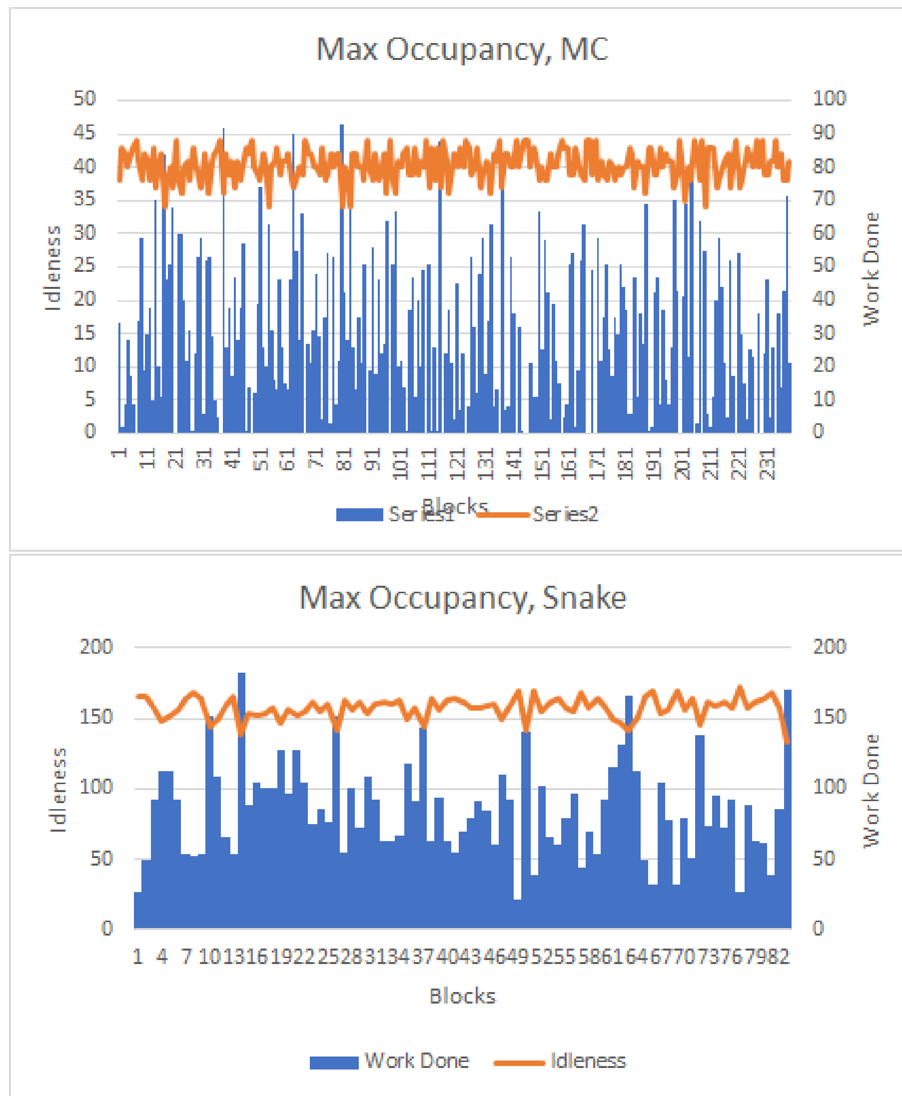

Fig. 7. Performance on Tesla K40c versus Quadro 620.

What is the memory overhead of Quilk?

Figure 8 shows the number of read and write accesses, in bytes, to global memory by each policy, run on the Tesla V100. CDP refers to the original algorithm provided with the CUDA Programming Toolkit. The results indicate policies 1 and 2, the random access and randomly mapped policies, respectively, have a signficant overhead in memory writes. Due to the random nature in which they are accessing queues, they are unable to take advantage of memory coalescing and cache hits. For memory reads, policy 1, the random access policy, has the largest number of transactions. Again, this is due to its random nature. When a worker attempts to steal from a neighbour, it first reads the neighbour's queue to check if it is busy. If it is

not busy, it randomly access another queue. This random access prevents policy 1 from taking advantage of global memory coalescing and cache hits. On the other hand, policy 0 and policy 3 have low memory overhead for both read and write transactions, since workers in each policy access the same queue repeatedly. Policy 2 suffers from a lack of memory coalescing.

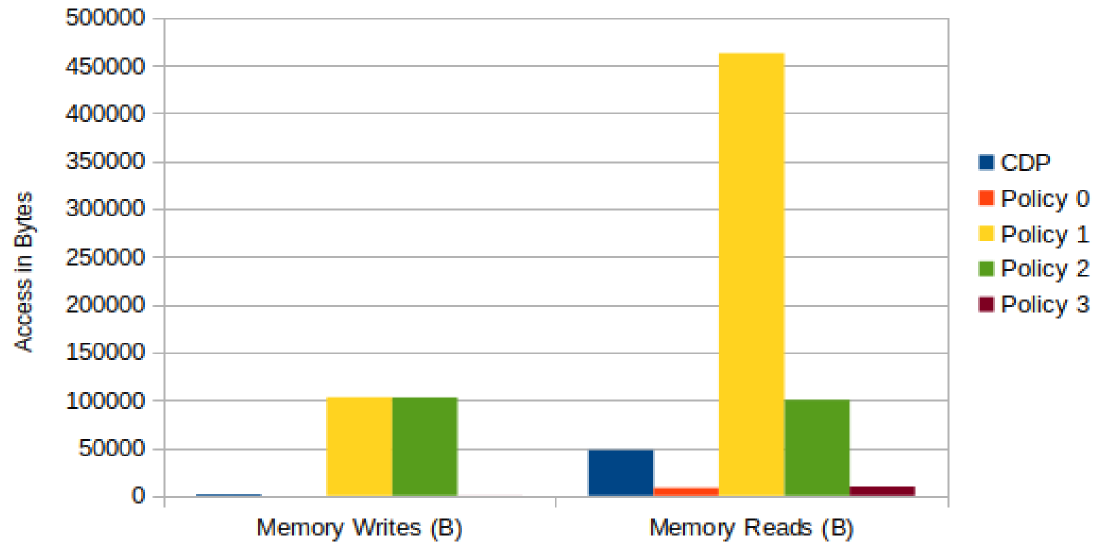

Fig. 8. Memory requests by each Quicksort implementation.

What is the runtime overhead of Quilk? Figure **??** shows the different types of inputs on which Quilk was run. The sdrand distribution refers to the standard C++ library srand function, which generates a pseudo-random sequence of values. The uniform values are generated using the Mersenne Twister algorithm. The sorted distribution is an already sorted sequence, and the constant distribution is a sequence consisting of the same value. Figure **??** shows the runtimes of all five Quicksort implementations. The original, recursive CDP implementation is consistently faster than all other implementations. The increase in runtime by the Quilk algorithm is likely due to the overhead of initializing and managing queue access. Policy 3 in particular, which consists of a single worker processing its queue, has the highest runtime overhead. Directly comparing policy 3 to CDP, where both have single workers, the difference seems to lie in the overhead of accessing global memory for a new task, and running each sequentially, versus launching new parallel dynamic kernels. The random stealing policy has the fastest runtime out of all four work-stealing policies, despite its large memory overhead. Further memory optimizations for these polices needs to be performed to bring the performance up to the level of the original CDP algorithm.

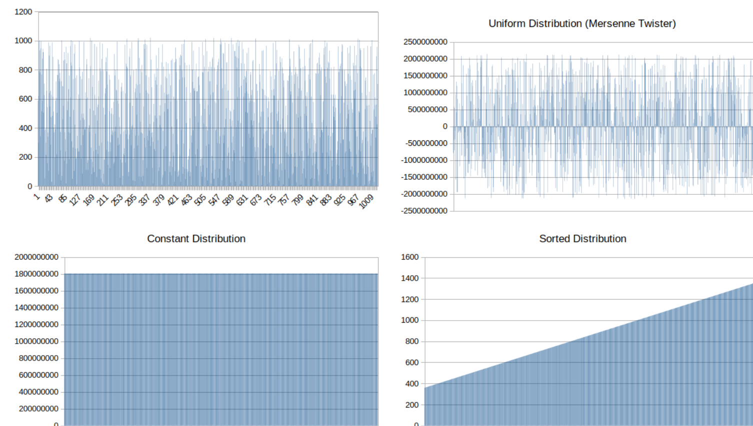

Fig. 9. Input data distribution. From left to right, top to bottom: srand, uniform, constant, and sorted.

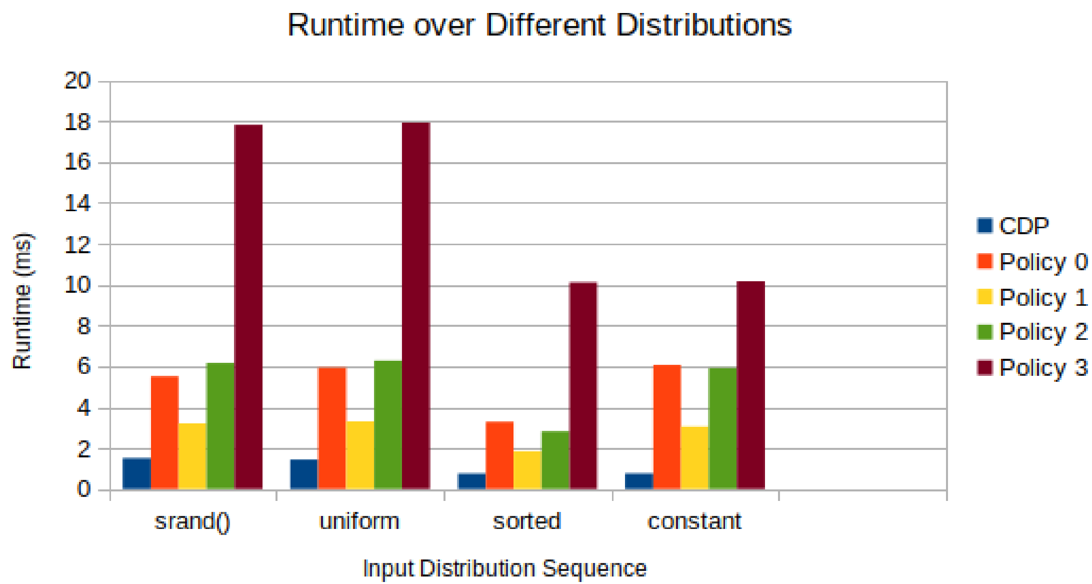

Fig. 10. Runtime overhead of Quilk and CDP for different steal policies.

5 CONCLUSIONS AND FUTURE WORK

In this work, we present Quilk, a basic work-stealing implementation of quicksort on GPUs to analyze how different work stealing policies affect work distribution for recursive algorithms. Three work-stealing policies are implemented: steal from a neighbour, steal from a random processor, and consistently steal from a previously assigned processor. Policy 1, stealing from a random processor, converged the fastest to having

work distributed across all processors. Policies 0 and 1 distributed work, but within a vicinity of processors, leaving several other processors idle.

The current Quilk implementation is a baseline for several future directions. We now list a few of these.

Improving Quilk: The current implementation at times runs into race conditions depending on the architecture. We would like further explore why these race conditions occur on some architectures, but do not exist on others.

Work-Stealing Policies: Additional work-stealing polices should be explored. For example, Tzeng, Patney and Owens found work-donation performed better than work-stealing. Comparing work-donation and its ability to distribute work across workers compared to the existing policies should be explored. Moreover, are there other work-stealing polices that might produce less contention, less memory overhead, or better work distribution in a faster amount of time?

Worker Granularity: The current implementation uses a single thread per worker. One expansion to this work is incrementing the number of threads per block, to increase the amount of parallelism within a task. Worker granularity could be explored with this method: how do 32 threads (a warp) composing a worker compare to a single thread? How do worker-warps compare to entire blocks containing several more threads? We expect performance times to improve with the use of groups of threads as workers rather than the current single worker thread implementation.

Memory Efficiency and Multi-level Work-stealing: Quilk currently uses global memory for tasks and queues. In adding worker units consisting of more than one thread, is it possible to leverage shared memory? One method is to implement multiple levels of work-stealing: work-stealing can be implemented within threads of a block using shared memory, and subsequently within blocks of a grid using global memory.

Work Distribution Metrics: Besides measuring how often a worker picks up a task, there are several other metrics that can be implemented as a measure of work distribution. The following additional metrics will be useful:

(1) A measure of the total amount of time a worker spends processing tasks versus the total amount of time the persistent kernel is active. This will provide a direct idle to busy ratio for a worker.
(2) The waiting time of a task. On average, how long does a task spend waiting in a queue before it is processed by its owner or it is stolen? We posit that a work-stealing policy with lower average task wait times likely has better work distribution capabilities.

Beyond Quilk: The currently available GPU Quicksort algorithms — GPU-Quicksort, CDP, and CUDA-Quicksort — all contain two phases: an initial phase where sequences are partitioned across blocks, and a second phase where each block recursively processes a given subsequence. Quilk should be a viable candidate to replace this second phase. Rather than having each block recursively work on a subsequence and remain idle on completion, each block can now steal work from other blocks. It would be interesting to determine how merging Quilk into these three implementations would improve runtime performance and load balancing.

ACKNOWLEDGMENTS

Thanks to Ahmed Mahmoud for patiently listening to our ideas and providing great feedback! Thanks also for introducing us to the volatile identifier to ensure variables are kept in global memory, thus resolving some race conditions we were facing.

We appreciate Muhammad Osama for spending time explaining some of his load balancing implementations for graph applications.

Our idea on timing tasks in a queue for future work was inspired by discussions with Trivikram Reddy and Simba Nyatsanga. Thank you!

Finally, we appreciate the course instructors for this opportunity! Thanks to Kerry Seitz and Dr. John Owens for listening to our ideas and reminding us to keep things simple, that helped us get back on track and stay focused!

6 REFERENCES

(1) GPU Parallelizable Methods, http://www.oxford-man.ox.ac.uk/gpuss/simd.html

(2) D. Cederman, P. Tsigas, "Dynamic Load Balancing Using Work-Stealing," in GPU Computing Gems, ACM. https://pdfs.semanticscholar.org/5162/8be69ba9418ef293bc379dcec1692a715417.pdf

(3) R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: an efficient multithreaded runtime system, in: R.L. Wexelblat (Ed.), Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, Santa Barbara, CA, 1995, pp. 207–216.

(4) N.S. Arora, R.D. Blumofe, C. Greg Plaxton, Thread scheduling for multiprogrammed multiprocessors, in: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, ACM, Puerto Vallarta, Mexico, 1998, pp. 119–129.

(5) D. Cederman, P. Tsigas, On dynamic load balancing on graphics processors, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware, Eurographics Association, Sarajevo, Bosnia, 2008, pp. 57–64

(6) Daniel Cederman and Philippas Tsigas, GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. In the ACM Journal of Experimental Algorithmics (JEA), Vol. 14, No. 4, ACM press 2009. [doi]

(7) Eriksson, Mattias Kessler, Christoph Chalabine, Mikhail. (2006). Load balancing of irregular parallel divide-and-conquer algorithms in group-SPMD programming environments.. 313-322.

(8) Tzeng, Stanley Patney, Anjul Owens, John. (2010). Task management for irregular-parallel workloads on the GPU. 29-37. 10.2312/EGGH/HPG10/029-037.

(9) GPU Quicksort Algorithm: http://www.cse.chalmers.se/research/group/dcs/gpuqsortdcs.html

(10) A. Duran et al., "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in ICPP'09, 2009, pp. 124–131.

(11) CUDA Toolkit 6.0, documentation. (Available from: http://docs.nvidia.com/cuda/cudasamples/index.html)[accessed on 10 September 2014].

(12) E. Manca, A. Manconi, A. Orro, G. Armano, and L. Milanesi, "Cuda-quicksort: an improved gpubased implementation of quicksort," Concurrency and Computation: Practice and Experience, 2015.

(13) Daniel Cederman and Philippas Tsigas, GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. In the ACM Journal of Experimental Algorithmics (JEA), Vol. 14, No. 4, ACM press 2009. [doi]