A New Programming Paradigm for GPGPU

Julio Toss[1,⋆] and Thierry Gautier[2]

[1] Institute of Informatics, UFRGS, Porto Alegre - RS, Brasil
jtoss@inf.ufrgs.br
[2] INRIA, MOAIS, LIG, Grenoble, France
thierry.gautier@inrialpes.fr

Abstract. Graphics Processing units (GPU) have become a valuable support for High Performance Computing (HPC) applications. However, despite the many improvements of General Purpose GPUs, the current programming paradigms available, such as NVIDIA's CUDA, are still low-level and require strong programming effort, especially for irregular applications where dynamic load balancing is a key point to reach high performances.

This paper introduces a new hybrid programming scheme for general purpose graphics processors using two levels of parallelism. In the upper level, a program creates, in a lazy fashion, tasks to be scheduled on the different *Streaming Multiprocessors* (MP), as defined in the NVIDIA's architecture. We have embedded inside GPU a well-known work stealing algorithm to dynamically balance the workload. At lower level, tasks exploit each *Streaming Processor* (SP) following a data-parallel approach. Preliminary comparisons on data-parallel iteration over vectors show that this approach is competitive on regular workload over the standard CUDA library Thrust, based on a static scheduling. Nevertheless, our approach outperforms Thrust-based scheduling on irregular workloads.

Keywords: Work Stealing, GPU, Task Parallelism.

1 Introduction

Nowadays, Graphical Processing Units have acquired great importance on the scenario of the High Performance Computing (HPC). Several HPC applications use this kind of hardware support to achieve better performances on parallel algorithms. The hardware is widely available and continues to evolve very fast, adding new capabilities and increasing its programmability. Programming models like OpenCL and Nvidia's CUDA allow developers to program and exploit parallelism on GPUs at the expense of a strong programming effort. Nevertheless, due to their important performances, GPUs have motivated the industry and the scientific community to port increasingly more applications to the GPU platform. At the same time, the generalization of the hardware reveals new challenges to be solved. Classical problems from the multicore-CPU architectures like

⋆ Partially supported by FAPERGS and CNPq grants, through the project "Green-Grid".

load balancing, synchronization and the need of abstract programming models, are now also present on GPUs.

Despite the enhancement of the programming capability provided by existing GPU programming solutions, for instance CUDA, the programming model they propose can only be directly exploited by sufficiently regular applications. Typically for those working over vectors or matrices. Nevertheless, there are several other kinds of applications where the parallelism is expressed recursively by creating tasks. For such applications it is necessary to provide a suitable runtime system to exploit the different cores present inside the GPUs.

The contribution of this paper is to propose and validate by experimentation a new paradigm for GPU programming based on data parallel tasks and work stealing. We show how task parallelism can be supported on graphics processors and how to deal with problems like load balancing and synchronization. Our preliminary results show that our approach is highly competitive with state of the art programming software for either data parallel programming or for pure task parallel programming.

The outline of this paper is the following: in Section 2 we briefly discuss the related works about work stealing algorithms and scheduling on GPUs. Section 3 presents the design and implementation of our approach to support work stealing with CUDA on GPUs. Then, on Section 4, we evaluate our model with several load balance patterns analyzing performance and overheads. On Section 5 we conclude and point future directions to improve the model.

2 Related Works

Task parallel applications are well suited to deal with irregular parallel algorithms. Scheduling of tasks among the computing resources must be effective to balance the workload. If scheduling is not performed well some processing units may be overloaded with work while others stay idle. Additionally, the scheduler implementation must be efficient to minimize overheads that come from the lock contention to access the work queue and to avoid stopping the computation for workload re-balancing.

2.1 Work Stealing

A well-known scheduling technique with several implementations on multi-core processors is work stealing. Here, each processing unit has its own queue of tasks to process. Whenever one gets idle it will, itself, look for tasks from other queues to steal. This technique is particularly interesting for applications that create tasks at execution time in an unpredictable way. Work stealing is, therefore, known as a dynamic load balancing method.

Blumofe et al. [4] give the first provably efficient work stealing scheduler. It proves that a parallel execution, on uniform P processors machine, using their work-stealing scheduler, has an expected run time of $\frac{T_1}{P}+O(T_\infty)$, where T_1 is the serial execution time of the multi-threaded computation and T_∞ is the minimum execution time with an infinite number of processors.

A work-stealing algorithm relies on a work queue data structure owned by each thread of the system. Threads can call three functions: Pop, Push and Steal on a work queue. Push and Pop functions are called only by the owner thread of a work queue to enqueue / dequeue tasks. Steal function is called by an idle thread on a victim work queue to get tasks to execute. The Cilk-5 runtime system [9] implements a *lock-based* work-stealing scheduler. It employs the Dijkstra's THE protocol for mutual exclusion [8] which greatly reduces the lock overhead [9] by only using systematic locks on steal operations to serialize thieves on the same victim. Arora et al. [2] present a completely *lock-free* work-stealing algorithm which uses array-based dequeues and minimizes the need of costly Compare-And-Swap (CAS) operations. Hendler et al. [11] overcame the potential overflow problem on ABP's algorithm [2] with a *dynamic memory* work-stealing algorithm. Chase and Lev [6] came with a simpler solution to this same problem by implementing unbounded dequeues as dynamic-cyclic-arrays. Hendler and Shavit [12] generalize the ABP algorithm to allow the processing units to steal, instead of one, up to half of the items in a given queue at a time. Their algorithm provides better load balancing than ABP while preserving the lock-free and CAS minimization properties.

2.2 Scheduling on GPUs

Recently, with the advent of the use of graphical processors for general purpose computing, those classical CPU load balancing methods started to be studied on GPUs. Chen et al. [7] use molecular dynamics simulation to evaluate a centralized method of dynamic load balancing on single and multi-GPUs systems. Their results showed that, for unbalanced workloads, task-based models can utilize the GPU hardware more efficiently than the standard CUDA scheduler.

Cederman and Tsigas [5] use the task of creating an octree partitioning to compare four different methods for dynamic load balancing. A centralized blocking task queue; a centralized non-blocking task queue; a static list and a distributed task queue using the ABP [2] work stealing protocol. Their results clearly show that centralized blocking methods are not suitable for GPUs as they perform poorly and cannot scale with the increase of processing units. The non-blocking task queue do perform better but, as a centralized approach, scales very poorly. The best performances and scalability were achieved with the work stealing method with distributed work queues.

In Angel et al. [1] the shortest-path problem is used as an irregular application to evaluate a framework for dynamic work scheduling based on Blumofe and Leiserson's work stealing algorithm[4]. They exploit the performance and synchronization characteristics of the GPU memory hierarchy by using a combination of shared and global memory queues. The overhead found was by a factor of 3 for queues on shared memory and 15 for queues on global memory.

In [16], Tzeng et al. propose a dynamic load balancing method based on task-donation, which shows similar performances to previous work-stealing approaches but uses less memory.

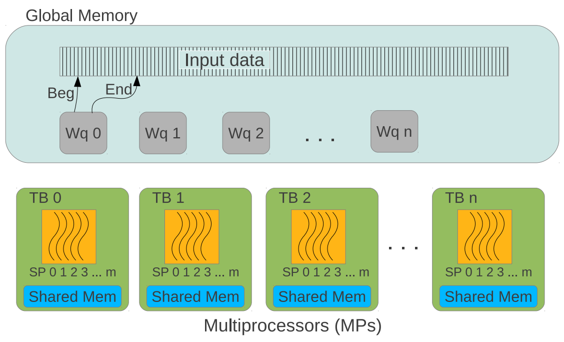


Fig. 1. Scheme of the work stealing scheduler on the GPU: one work queue on global memory for each Tread Block running on the Multiprocessors

Previous work shows that scheduling inside GPUs is necessary to improve performance in task-parallel applications. On the CPUs, several parallel programming tools like CILK+ (Frigo et al. [9]), Intel TBB (Pheatt [13]), KAAPI (Gautier et al. [10]), use work stealing as standard scheduling technique in parallel **for** loops. Additionally, recent architectures like the Intel Many Integrated Core (Intel MIC) use CILK+ as standard programming model reinforcing the trend of work stealing schedulers on massively parallel architectures. On the other hand, GPU programming tools like the Thrust Template library for CUDA (Bell [3]), provides generic templates (e.g. array Transform) to enhance programmer's productivity. However, CUDA does not have a dynamic scheduler, and for some types of workloads it cannot extract the best performance of the GPU. In this context, our model extends the use of work stealing in GPUs to a broader range of parallel applications. We implemented an hybrid programming model combining tasks and data parallelism. Our benchmarks showed comparable performance to State of Art on typical task-parallel problems (Octree Partitioning, Sec. 4.5) and we outperform Thrust on the array transform problem (Sec. 4.4).

3 Mixing Task Parallelism and Data Parallelism on CUDA

As showed before, parallel tasks with work stealing is being used on multi-core architectures and on GPU as standard paradigm for irregular divide and conquer parallel applications, which are the target applications for work stealing.

Here we present an unified programming paradigm for general parallel applications on GPU. This paradigm deals well with irregular and regular workloads on data parallel application as well as task parallel application. The paper focuses on scheduling data parallel GPU application using a novel approach.

CUDA is limited by the absence of a runtime scheduler to support dynamic load balancing. We show next how we can implement an efficient and generic support to load balancing with CUDA based on work stealing. Section 3.3 presents how to do an efficient scheduling of data parallel application. Section 4 provides experimental evaluation with different types of workloads.

3.1 Design of Our Approach

The abstraction provided by the CUDA model allows us to exploit two levels of parallelism. At first level, a program can be divided in coarse sub-problems that can be solved independently in parallel (*Thread Blocks* in CUDA), and then into finer threads that cooperate for the same task.

Independent Thread Blocks (*TB*) are mapped to multiprocessors[1] on the GPU. In practice, CUDA kernels use much more thread blocks than the number of multiprocessors available. The CUDA runtime has a very basic scheduler that assigns thread blocks to MPs. In our approach (Fig. 1), we consider a fixed number of thread blocks, which is independent from the size of the input data. Each thread block stays persistently on a multiprocessor and manages a task queue on the GPU global memory using the work stealing algorithm. When a TB does not have any more tasks to execute, it steals some from another's queue.

3.2 Work Queue Implementation

The work queue implemented consists of a work queue data structure named `workqueue_t`. This structure consists of two integers, `beg` to the beginning and `end` to the end of the interval `[beg,end)`. Additionally, each work queue is associated to a `mutex` variable used to control its access in conflicting cases.

Each Thread Block owns a work queue. This structure is accessed by the following three functions.

Push : `int push( workqueue_t* kwq, int* beg)`
The push function is called by the thread to add a new task to its own work queue. The value `beg` must be less than `kwq->beg`. This operation is always non-blocking and extends the work queue.

Pop : `int pop( workqueue_t* kwq, int* i, int* j, int size)`
The pop function is called by the thread to pop range `[*i, *j)` from its own work queue. The size of the returned range is at most `size`. The function returns a non zero value in case of success, *i.e.* iff the returned range is non empty. The pop increments the field `beg` of the work queue.

Steal : `int steal( workqueue_t* kwq, int* i, int* j, int size)`
The steal function is called by the thread to steal range `[*i, *j)` from another work queue than its own. It decrements the field `end` of the work queue. The size of the returned range is at most `size`. The function returns a non zero value in case of success, *i.e.* iff the returned range is non empty.

The work queue implementation relies on a Dijkstra's protocol and is similar to the T.H.E protocol as described in Frigo et al. [9]. The main difference is

[1] CUDA definition.

```
int *my_wq = workqueue_getown(blockIdx);
while (1) {
  if ( IamTheMasterThread(threadIdx)) {
    if( !workqueue_pop(my_wq, locbeg, locend, popSize) ) {
      workqueue_t *victim_wq = workqueue_getrandom();
      int stealSize = workqueue_size(victim_wq)/2;
      if ( stealSize >= popSize ) {
        workqueue_lock(victime_wq.mutex);
        if (workqueue_steal(victim_wq, stealBeg, stealEnd, stealSize))
          workqueue_push( my_wq, stealBeg, stealEnd );
        workqueue_unlock(victim_wq.mutex);
        continue;
      }
    }
  }
  __syncthreads();
  TASKCall(locbeg, locend);
  __syncthreads();
  if ( terminate ) break;
}
```

Fig. 2. Cuda kernel of the work stealing scheduler loop

that pop or steal functions increment or decrement `beg` and `end` not only by 1 but by `size`. Steal function calls are serialized on the `mutex` lock: the runtime guarantees that the concurrency on a work queue structure is at most 2.

Our work queue data structure allows to steal range of indexes. For task parallelism, the runtime stores tasks into an array container: a task is identified by its index and the work queue can trivially be used to implement work stealing scheduler.

3.3 Data Parallel Application Scheduling

Moreover, our work queue can also be used to lazily create task. Let us consider the `foreach` parallel algorithm where the same functor is applied on each entries of an array. In that case, a task is then only identified by the sub range where it acts on. Stealing a task is equivalent to steal a sub range of the initial interval given by the initial `foreach` call. Our work queue implementation lets, at runtime, the scheduler to steal tasks by simply calling the `steal` function. To apply our method on more complex problems, one should define a linearization of the computations in an interval homomorphic to $[0, N)$, which is the case for almost all STL's algorithms on random access iterators Traore et al. [14].

To mix both task and data parallelisms our runtime implements the concept of the *malleable task* Turek et al. [15]. In our implementation, a *malleable task* exports a function that is called to extract work on work stealing scheduler decision. Data parallel task, such as the `foreach`, is a malleable task that represents its work using our work queue. The exported function calls the Split operation on the work queue. Therefore, after a successful steal operation, an idle TB receives a sub range of the initial range to perform.

3.4 CUDA Work Stealing Algorithm

In work stealing, each multiprocessor has its own work queue structure in global memory. Multiprocessors pop from their own queues, if there are no more tasks they randomly choose another work queue to steal from. Our work stealing scheduler is embedded within a CUDA kernel. The main scheduler loop is sketched in Fig. 2. The loop is executed by all TB on the GPU. While the work queue of a TB has enough local work (line 4), the master thread of the block pops a sub range and all threads ,synchronized at line 16, perform the data parallel task (at line 17). If there is no local work, the master thread selects at random a victim work queue (line 5), tries to steal half of its contents (line 9) and, if the steal successes, populates its own work queue with the theft range `[stealBeg, stealEnd)` (line 10).

4 Evaluation

The experiments were realized on a NVIDIA GTX 280 GPU running at 1.3 GHz with 1GB of global memory. This model of GPU contains 30 Multiprocessors (MP), each one with eight scalar processors (SP) giving a total of 240 cores. All the applications were tested using version 4.0 of the CUDA driver and runtime . Additionally, some experiments were also tested on a Tesla C2050 GPU (Fermi architecture) running at 1.15 GHz with 3GB of global memory.

Every measure presented in the following benchmarks is an average of 10 executions of the kernel. This number showed to be sufficient to get reliable results, with negligible standard deviation (which were omitted on our plots). The time is measured using GPU timers without counting overheads of the kernel launch nor PCI data transfers between host (CPU) and device (GPU).

4.1 Elementary Overhead

Work stealing adds an intrinsic overhead to the program due to the pop operation that is always done before starting the actual computation of the task (line 6 of Fig. 2). This benchmark used a kernel configuration of 1 block of 512 threads and varied the pop size from 2^9 to 2^{20}. Figure 3 reports the total execution time with several numbers of pop operations. Please note that the pop operation handles only work queue indexes, therefore the cost of a pop is independent of its size. By fitting the obtained curves to an affine function, the pop overhead found was 5186.72 cycles (3.52 μs) on the GTX280 and 2187.32 cycles (1.92 μs) on the Fermi GPU.

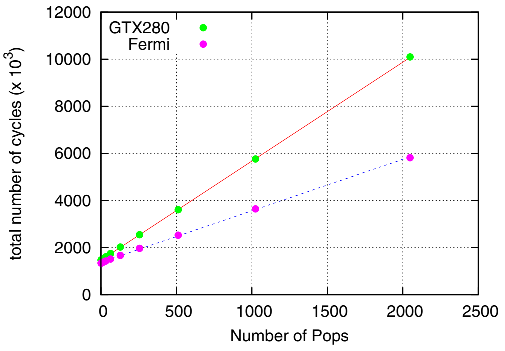


Fig. 3. Pop cost estimation on a GTX280 and a C2050(Fermi) GPU

4.2 Benchmark Application

Our target application consists in a simple transformation over an array of Floats. The program applies a function $x \longrightarrow f(x)$ to each element i of the input array and stores the result in the output array. In this benchmark we consider a task an instance of the `transform` function on a range of the initial interval.

The reference parallel implementation was taken from NVIDIA's Thrust library (Bell [3]). Next sections report experiments with regular then irregular workloads.

4.3 Load Balancing on Regular Workloads

This section compares three load balancing methods when used to manage regular workloads. Our benchmark transform application generates a regular workload when it applies a constant function to every position of the input array.

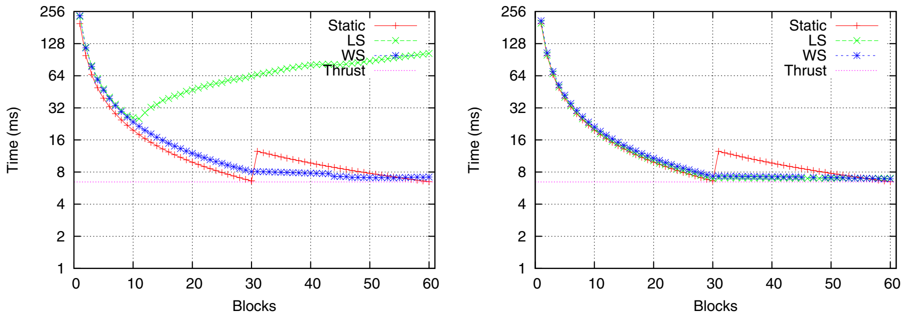
Load balancing is about managing tasks on processors. More precisely, in the experiments presented here, **the data parallel task updates at most 512 positions of the array**. This task size was chosen in conjunction with the block size, which also contains 512 threads. This way each task, except for a few ones at the end of the sub-ranges, is fully parallel and makes all the threads of the block to work. This number of threads per block showed the best performance for a transform on a sufficiently large array. Additionally, the same block size of 512 threads and 60 thread blocks is used by the static transform kernel in Thrust library, whose strategy is to optimize occupancy.

Our work stealing method is compared to two classical scheduling method:

- The *Static Scheduling* is the default load balancing method that CUDA uses when it schedules blocks on multiprocessors. Blocks are evenly distributed among the multiprocessor of the device until they reach the limit of active blocks. When an active block completes its job, the next blocks are scheduled.
- The *List Scheduling* uses a centralized work queue that every processing unit have to access to get new tasks to process. Tasks are assigned in a FIFO manner where idle processors get the task at the beginning of the list. Note that there is a lock on the work queue that serializes every access to it.

Single Task Pop. Figure 4a presents the total execution time of Transform on an array of 5120000 elements (*i.e.* 10000 tasks of 512 array positions). Each curve represents one different load balancing method. In this experiment LS and WS always pop one task (or 512 array elements) at a time which totals 10000 pops operations over the whole execution (one pop by task). WS steal operation steals half of the victim interval.

This graph makes evident the drawback of centralized load balancing methods. Even with very regular workload, the List Scheduling method quickly reaches a limit where it stops scaling. Actually, with more than 10 MPs the performance always gets worse. We attribute this behavior to a lock contention problem. As



(a) Pop size of a single task (b) Best pop size for each method

Fig. 4. Comparison of the Static, the List Scheduling (LS) and the Work Stealing (WS) methods

the time of execution of a single task is very short, blocks are often trying to acquire the lock which increases lock contention.

The static method shows the best absolute performance at 60 blocks, the maximum number of blocks that the GPU scheduler runs concurrently on the hardware (tests with more blocks didn't enhance the execution time). However, note that the static method does not scale in a regular way as does work stealing. This can be seen on the performance drop at 31 blocks. This drop is due to the fact that with 31 blocks, one multiprocessor has two active blocks and twice more tasks which causes the load imbalance. The work stealing (WS) does not suffer of this problem because an idle TB on a multiprocessor can steal task from overloaded multiprocessors.

The overhead of accessing the work queue can be reduced by popping more tasks at a time. Figure 4b shows the best performances found for each method when tuning the number of tasks retrieved by pop (the Pop Size). For work stealing, the optimal pop has a size of 3 tasks (i.e 1536 elements of the array) and the best time, $6.90ms$, is achieved with 60 blocks of 512 threads. List scheduling achieves $6.92ms$ of peak performance when the pop size is equal to 7 tasks (i.e 7168 array elements) with 30 blocks of 512 threads. The best static time is $6.49ms$ at 60 block of 512 threads.

Pop Size Variation. Figure 5 shows how WS and LS behave with the variation of two parameters: pop size and number of blocks. The y axis represents the size of each pop in number of array elements (number of tasks x 512). The values plotted correspond to the difference of execution time between LS (List Scheduling) and WS (Work Stealing). Lighter tonalities means smaller differences.

We can identify two regions A and B. In region A, LS performs better than WS but the biggest difference is only 5.63 ms (37,75% speedup over WS). In region B, WS outperforms LS achieving a gain of 98.53 ms (93,03% of speedup over LS). Therefore, even if LS is simpler to design than WS, it suffers from its

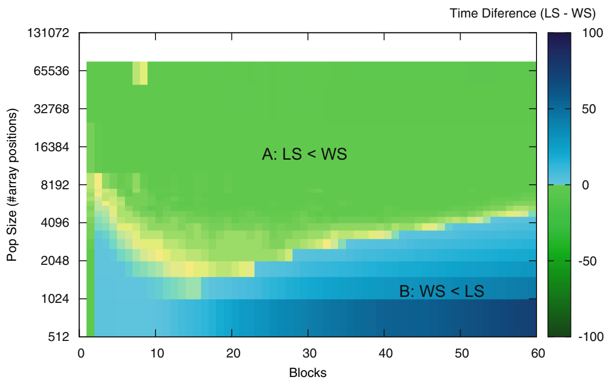


Fig. 5. Time difference between LS and WS

grain size selection where small value may degrade strongly the performance due to contention and where big values limit the parallelism (with LS, once popped a sub-range cannot be stolen anymore).

4.4 Load Balancing on Irregular Workloads

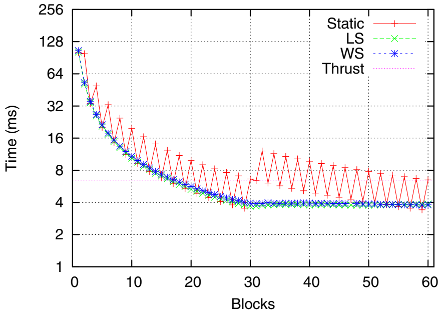
This section evaluates the load balancing methods with two different patterns of workloads. Like on Chen et al. [7], patterns of irregularity were created by nullifying the work of some tasks of the input array. These patterns are:

1. Pattern 1 = 0 1 0 1 0 1 0 1 : one task each other is nullified (50% of workload reduction).
2. Pattern 2 = 0 0 0 1 0 0 0 1 : one on each three task is nullified (75% of workload reduction).

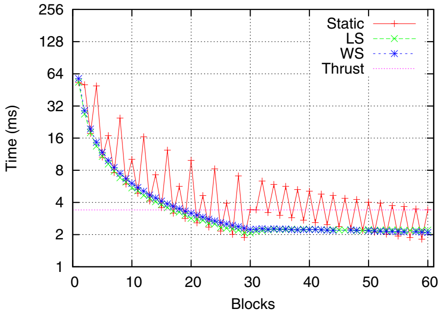
Figure 6 shows the best results of each method of load balancing for the two patterns of irregularity. These tests represent the best configuration of pop size found for each method. LS uses a pop size of 10240 (20 tasks) with pattern 1 and 25600 (50 tasks) for pattern 2. WS uses pop sizes of 4096 (8 tasks) for both patterns. These results clearly show the instability of the static method for irregular workloads and how good dynamic scheduling approaches deal with it.

4.5 Octree Partitioning

The transform benchmark shows how to use our work stealing model to schedule array-based applications. However, this same model can be used for classical task-based problems. For instance, the octree partitioning problem create tasks to recursively separate particles in a 3D space into octants. We used the octree implementation provided by Cederman and Tsigas [5] and adapted it to use our scheduler. We then compared their load balancing method to ours.

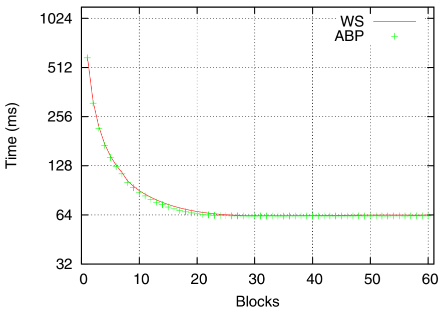


(a) Pattern 1

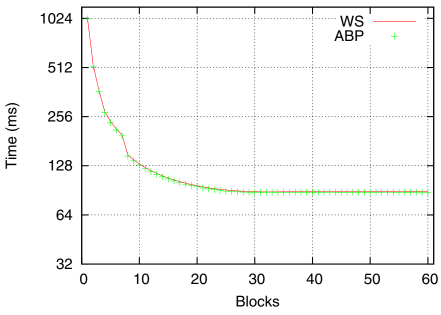


(b) Pattern 2

Fig. 6. Transform problem on irregular workloads



(a) Cube Distribution



(b) Sphere Distribution

Fig. 7. Load Balancing methods on Octree partitioning problem

The following benchmarks consists in creating an octree partitioning of a 3D space containing 500000 particles. The algorithm recursively subdivides the space until the threshold of 20 particles per subspace is reached. Every time a sub-space needs to be split, a new task is created.

Figure 7 shows a comparison between our algorithm (labeled WS) and Cederman et Tsigas (labeled ABP) with two different particles distribution. One where the particles are all randomly picked from a cubic space and other where they are randomly picked from the surface of a sphere. As shown in Fig. 7, WS and ABP presented similar performance. The best times found were 63.26ms with 31 blocks for WS and 62.94 ms for ABP with 35 blocks.

4.6 Discussion

The static scheduling showed to perform quite well for the regular array transform. This is mainly due to the cyclic algorithm used to assign array elements (tasks) to threads. The cyclic distribution makes a good division of the work because it gives to all the blocks the same amount of tasks and at the same time it spreads contiguous parts of the array that may contain more expensive tasks. However, this model is vulnerable to specific workload models (see section 4.4). Additionally, it should be noted that multiprocessors on a single GPU execute at same clock. On a multi-GPU system for example, the speed of multiprocessors may variate creating another source of load imbalance difficult to handle with static scheduling.

List scheduling can achieve good performances, even with irregular work loads. However we found that it is very dependent on the computation time spent between pop operations. Thus an accurate tuning of pop size is mandatory to get good performance.

Work stealing was the method that showed the best adaptability over all of the presented benchmarks. Even if it didn't have the best absolute performance, the difference from the other methods was very small. This method is also less sensible to lock contention than List Scheduling in which pop size has to be carefully tuned to overcome contention.

5 Conclusion and Future Work

In this work we considered a new programming model for general purpose GPUs based on work stealing. This model allows the programmer to express the parallelism of a GPGPU application in a hybrid manner taking benefit, at the same time, from an efficient task scheduling algorithm and from the highly SIMD computation power of graphics processors.

We presented empirical results that attest the effectiveness of our model and, to the extent of our knowledge this is the first work to evaluate a regular problem with dynamic load balancing on GPU. Our results confirm that work stealing is a generic scheduling method and performs well in both regular and irregular problems. We compared our scheduler on regular array transform micro-benchmark with respect to the static implementation of the Thrust well-known GPU library and found little degradation with uniform load, and better performances on unbalanced load.

Ongoing work is to explore in more details how this model behaves on the new Fermi GPU architecture and what optimizations can take favor of it. Preliminary tests (section 3), suggest that new hardware capabilities notably, the presence of a full cache memory hierarchy, could be better exploited by our work queue implementation. At long-term, we envision the integration of this model in the KAAPI library (Gautier et al. [10]) which lacks the ability of scheduling inside GPUs.

References

[1] Angel, M., Michael, M.M., Bivens, J.A.: Dynamic Work Scheduling for GPU Systems. Memory, 57–64 (2010)

[2] Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread Scheduling for Multiprogrammed Multiprocessors. Theory of Computing Systems 34(2), 115–144 (2001), http://www.springerlink.com/openurl.asp?genre=rticle &id=doi:10.1007/s002240011004

[3] Bell, N.: Thrust: A Productivity-Oriented Library for CUDA. sbel.wisc.edu. 359–373 (2012), `http://sbel.wisc.edu/Courses/ME964/Literature/thrustGPUgems2011.pdf`

[4] Blumofe, D.R., Leiserson, E.C.: Scheduling multithreaded computations by work stealing. Journal of the ACM 46(5), 720–748 (1999), `http://portal.acm.org/citation.cfm?doid=324133.324234`

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

[6] Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures - SPAA 2005 (c), vol. 21 (2005), `http://portal.acm.org/citation.cfm?doid=1073970.1073974`

[7] Chen, L., Villa, O., Krishnamoorthy, S., Gao, G.R.: Dynamic load balancing on single- and multi-GPU systems. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–12 (2010), `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470413`

[8] Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM 8, 569 (1965), `http://doi.acm.org/10.1145/365559.365617`

[9] Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation - PLDI 1998, pp. 212–223 (1998), `http://portal.acm.org/citation.cfm?doid=277650.277725`

[10] Gautier, T., Besseron, X., Pigeon, L.: Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, pp. 15–23. ACM (2007), `http://portal.acm.org/citation.cfm?id=1278182`

[11] Hendler, D., Lev, Y., Moir, M., Shavit, N.: A dynamic-sized nonblocking work stealing deque. Distributed Computing 18(3), 189–207 (2005), `http://www.springerlink.com/index/10.1007/s00446-005-0144-5`

[12] Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing - PODC 2002, p. 280 (2002), `http://portal.acm.org/citation.cfm?doid=571825.571876`

[13] Pheatt, C.: Intel threading building blocks. J. Comput. Sci. Coll. 23, 298–298 (2008), `http://portal.acm.org/citation.cfm?id=1352079.1352134`

[14] Traoré, D., Roch, J.-L., Maillard, N., Gautier, T., Bernard, J.: Deque-Free Work-Optimal Parallel STL Algorithms. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 887–897. Springer, Heidelberg (2008)

[15] Turek, J., Wolf, J.L., Yu, P.S.: Approximate algorithms scheduling parallelizable tasks. In: Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1992, pp. 323–332. ACM, New York (1992)

[16] Tzeng, S., Patney, A., Owens, J.D.: Task management for irregular-parallel workloads on the GPU. In: Proceedings of the Conference on High Performance Graphics, pp. 29–37. Eurographics Association (2010), `http://portal.acm.org/citation.cfm?id=1921485`