UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JULIO TOSS

Work Stealing Inside GPUs

Trabalho de Graduação

Nicolas Maillard (Inf / UFRGS)
Orientador

Thierry Gautier (MOAIS / LIG)
Co-orientador

Porto Alegre, dezembro 2011

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

DEQueue	Double Ended Queue
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
HPC	High Performance Computing
LS	List Scheduling
MIMD	Multiple Instruction Multiple Data
MP	Multiprocessor
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SP	Streaming Processor
WS	Work Stealing

LIST OF FIGURES

ABSTRACT

Graphics Processing units have become a valuable support for High Performance Computing (HPC) applications. However, despite the many improvements on the General Purpose GPU, there is still the need of a generic programming model adaptable to the many forms of parallelism that an application can express.

The CUDA programming model is widely used on the GPGPU domain, but is very limited in aspects like load balancing and task parallelism.

This work introduces a new programming model to be used on general purpose graphics processors. We propose an hybrid model combining tasks and data parallelism which extends the domain of applications that can efficiently make use of graphics processors. We implement a work stealing scheduler to efficiently schedule tasks inside a GPU keeping an even load balance between its multiprocessors.

Finally, we evaluate the performance of our work stealing scheduler comparing it with static and list scheduling methods applied to the problems of array transformation and octree partitioning.

Keywords: GPGPU, Scheduling, Dynamic Load Balancing, Work Stealing, Array Transform.

Roubo de Trabalho em Processadores gráficos

RESUMO

Unidades de processamento gráfico (GPU) tornaram-se ferramentas de grande valia no domínio da computação de alto desempenho. Graças às recentes inovações e melhoramentos do hardware é possível utilizar processadores gráficos de propósito genéricos (GPGPUS) em uma ampla gama de aplicações científicas. No entanto, os modelos de programação existentes usados em GPGPU não são ainda suficientemente adaptáveis às diversas formas de paralelismo que uma aplicação possa expressar.

Neste contexto, propomos um modelo híbrido de programação paralela para GPGPU usando paralelismo de tarefas e de dados. Em oposição ao que é advogado pelo modelo de programação CUDA, baseado apenas no paralelismo de dados, mostramos que é possível explorar o paralelismo de tarefas em GPUs e escaloná-las de forma eficiente usando a técnica do roubo de tarefas.

Apresentamos neste trabalho a implementação de um escalonador por roubo de tarefas em CUDA e comparamos seu desempenho aos métodos de escalonamento estático e por lisa aplicados aos problemas de transformação em array e particionamento em octree.

Palavras-chave: GPGPU, Escalonamento, Balanceamento de Carga Dinâmico, Roubo de Trabalho, Transformação em Array.

1 INTRODUCTION

Nowadays, Graphical Processing Units have acquired great importance on the scenario of the High Performance Computing (HPC). Several HPC applications started to 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. New solutions like OpenCL and Nvidia's CUDA allow developers to easily program and exploit parallelism on GPUs. This scenario has contributed and 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 load balancing, synchronization protocols and the need of abstract programming models, are now also present on GPUs.

Despite the great enhancement of the programming capacity 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 matrices. Nevertheless, there are several other kinds of applications where the parallelism is expressed recursively by creating tasks. For these applications it is necessary to provide a suitable runtime system to exploit the different cores present inside the GPUs.

This work arises on the context of the scheduling of recursive programs by work stealing, as implemented in KAAPI (KAAPI, 2011), where an inactive computing resource steals work from active ones.

This research was developed on the MOAIS project at the Laboratory of Informatics of Grenoble (LIG). The MOAIS project focuses on the class of application whose performance enhancements depends mainly on the increase of the number of computation resources. This rises the problematic of how to program a portable application capable of efficiently use large scale resources. Such application program has to be adaptive; and adaptivity is managed by the scheduling. Thus, fundamental researches undertaken in the MOAIS project are focused on this scheduling problem which manages the distribution of the application on the architecture.

1.1 Contribution

This research project studied the implementation of a distributed work queue among multiprocessors (MP) of a GPU. The main contributions found in this work are:

- A bibliographic study presenting the State of Art of load balancing on GPUs.
- An interface for a work-stealing scheduler on GPU.
- Implementation, test and validation of different load balancing methods using a same problem (array transform) with different workload patterns (regular and irregular).
- Proposal of an hybrid programming model for GPUs based on work-stealing (stealable parallel tasks at the MP level + SIMD data-parallel task inside the MP).

2 CONTEXT

This section introduces the background used as basis to develop this research study.

2.1 The GPU Hardware

A Graphics Processing Unit (GPU) has always been a highly parallel processor, traditionally specialized to deliver great throughput performance on the computation of the graphics pipeline (MICHEL, 2006). In the past, stages of the pipeline were implemented as fixed-functions processors barely configurable. More recently, the GPU hardware has evolved toward more flexibility and programmability. On modern GPU architectures, this pipeline was unified into a single programmable parallel processor, thus opening the graphics hardware to the general-purpose computation (GPGPU).

A modern GPU, for instance from the NVIDIA's TESLA architecture, is composed by several *Streaming Multiprocessors* (MP) which are executed in parallel and independently. Each MP is subdivided into several *Streaming Processors* (SP), sometimes also referred as "CUDA cores" (Figure 2.1). Each SP can execute a sequential thread in a way NVIDIA calls SIMT (Single Instruction, Multiple Thread). In practice, this means that all cores in the same SM execute the same instruction at the same time but on different data. The memory sub-system available on GPUs offers almost no flexibility in terms of configuration. Typically, each MP has an internal low latency shared memory but which is only accessible by its own SPs. At higher latencies there is a global memory, which is accessible from all the MPs on the device and is also used to communicate with the CPU. Official documentation states that shared memory latency is roughly 100x lower than global memory latency (NVIDIA, 2010b). Microbenchmarks on a GT200 (PAPADOPOULOU; SADOOGHI-ALVANDI; WONG, 2009) GPU reports latencies of 38 cycles for accesses to shared memory and an average of 441 cycles for global memory. Table 2.1 summarizes the main hardware characteristics of the two NVIDIA architectures that we

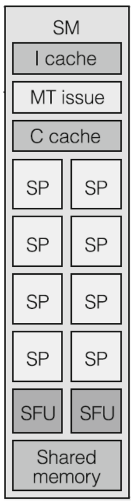


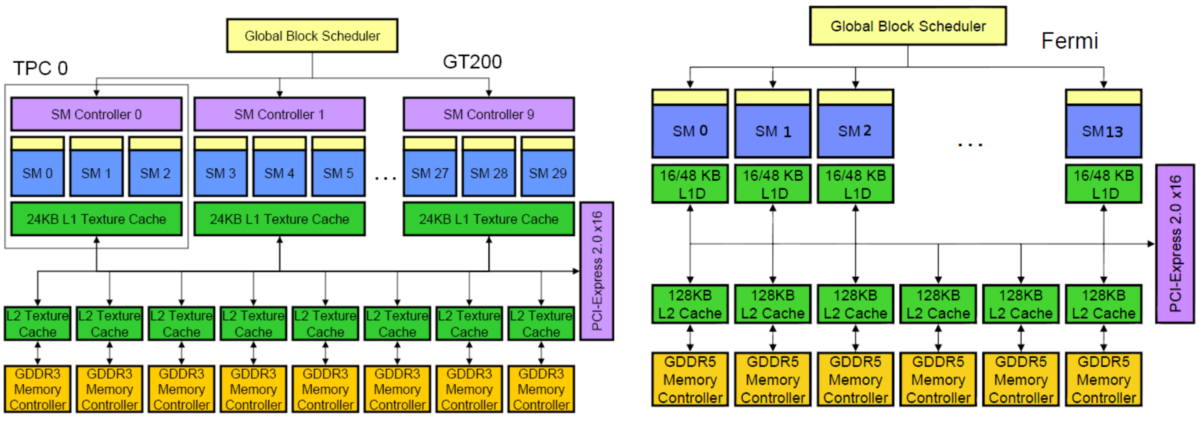
Figure 2.1: A GPU Multiprocessor (MP)

use later in our benchmarks.

GPU	GT200	Fermi
Cuda cores	240	448
Shared Memory (per SM)	16 KB	Configurable 48 KB or 16 KB
Registers (per SM)	16	32
L1 cache (per SM)	None	Configurable 48 KB or 16 KB
L2 cache	None	768 KB
Load/Store Address Width	32-bit	64-bit

Table 2.1: summary table

In Figure 2.2 we see the internal organization of the GT200 and Fermi chipsets. The GT200 chipset (Figure 2.2a) found in graphic cards like the GTX280, is the previous generation of NVIDIA's GPUs which has been substituted by the new Fermi architecture. The Fermi architecture has many significant improvements compared to its predecessors, notably each MP has now a complete memory hierarchy with caches L1 and L2 which speed up the access to global memory.



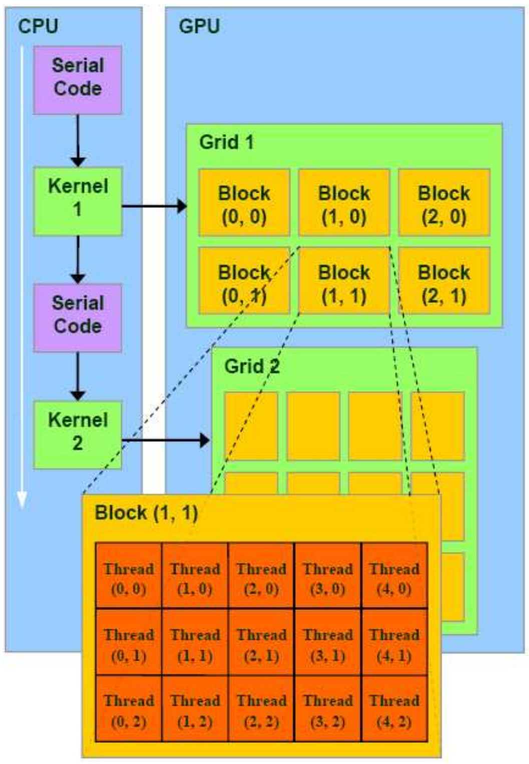
(a) GT200 (b) Fermi

Figure 2.2: GT200 and Fermi Architectures (KANTER, 2008, 2009)

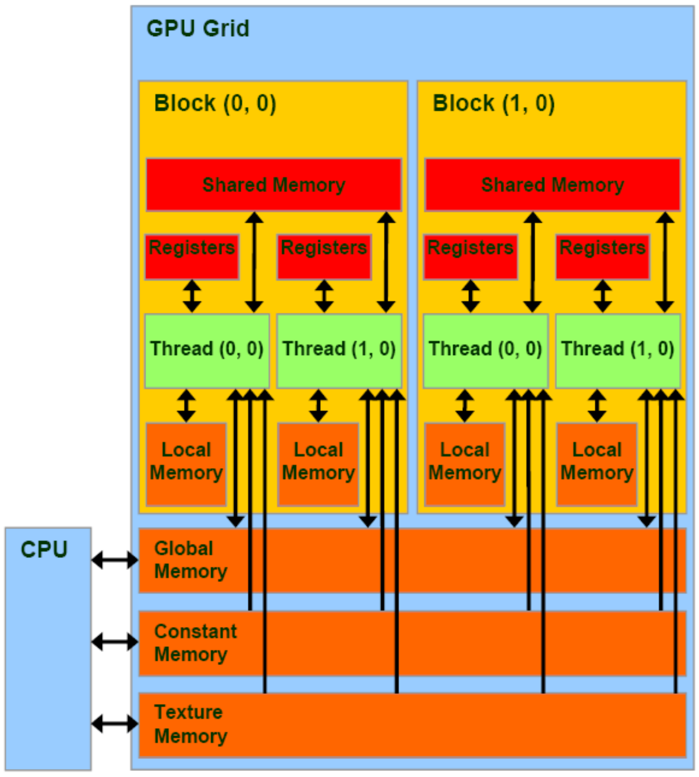
2.2 The CUDA programming model

CUDA is at the same time a scalable parallel programming model and a software environment for parallel computing. It is delivered by NVIDIA and allows GPGPU development on their own graphic cards.

The CUDA programming model is implemented in a hierarchical way in which threads are the base unit of execution. Basic threads are grouped into *blocks* and a set of thread



(a) Thread Hierarchy in CUDA



(b) CUDA Memory Spaces

Figure 2.3: CUDA execution and memory models(KIRK, 2010)

blocks forms a *grid* (figure 2.3a). A GPU always runs special functions called *Kernel*. Each kernel runs at a time on the device and is split into many concurrent threads, which executes the same code over different data based on its thread ID. As the GPU uses a SIMT architecture, the multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Threads within a warp execute one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If the execution path of two threads in the same warp diverges, the warp scheduler serializes and executes each of them at a time.

CUDA threads may access data from multiple memory spaces (figure 2.3b). Each thread has a private local memory. Each block has a shared memory space visible to all threads inside to the block and with same lifetime as the block. Finally, all threads have access to the same global memory.

This thread and memory organization implies in several ”good practices” of programming that should be followed to extract the best performance of the CUDA architecture. Notably, an important performance consideration concerns the access pattern to global memory. Thread accesses to global memory are issued by half or full warp. In order to this access to be performed one a single transaction, all the addresses requested must fit into a segment size of 32, 64 or 128 bytes. Otherwise, multiple transactions will occur resulting in a considerable performance degradation. Effects on misaligned accesses are measured on section 5.5.

More details on optimization and programming guidelines for CUDA developers can be found on the NVIDA's CUDA Programming and Best Practices guides (NVIDIA, 2010a,b).

2.3 Applications

In this work, two classes of applications were considered according to their workload distribution: there are regular and irregular applications. This classes characterize how the parallel parts of an application get distributed among threads.

2.3.1 Regular applications

An application with regular workload means that every task created during its parallel execution have similar computation costs, which makes them easier to schedule. Example of regular algorithms are array Transform and prefix computation.

The standard array Transform is defined as

$$output[i] = f(input[i])$$

meaning that the output array contains each element of the input array transformed by the function $f()$. Figure 2.4 shows the sequential algorithm of an array transform with N elements.

```
for i = 1 to N do
    output[i] = f ( input[i] )
```

Figure 2.4: Array Transform pseudo-code

A classical parallelization scheme of this algorithm, on a multicore architecture, is to cut the input array into p chunks and distribute them between the p available processing units. If the size of the input array is N then each core k computes:

$$output[i] = f(input[i]) \; s.t \; i \in [k * N/P, (k+1) * N/P]$$

This parallel execution performs well if the function f has constant cost over all the elements of *input* and each element can be computed independently.

2.3.2 Irregular applications

Irregular applications are characterized by variable task computation cost. This makes scheduling more difficult and more susceptible to imbalance problems. An array transformation, for example, could use a non-constant function with unknown cost variations thus creating tasks of different costs.

One classical irregular application is the octree partitioning. In this problem, a space containing several particles has to be subdivided (into octants) until each sub-space contains less particles than a threshold. This kind of problem is irregular because each task that creates an octant will cost proportionally to the quantity of particles in the sub-space. Regions of the space with more particles will generate more tasks.

2.4 Scheduling Algorithms

In this section we present three examples of scheduling methods to the transform problem.

2.4.1 Static Scheduling

In this example we consider that the cost of $f()$ is known beforehand but is variable between tasks. First we choose the *grain* of the task that will execute the transform, that is, the size of the segment that will be given to the multiprocessors. For an interval $[a, b)$ the grain is defined as $grain = b-a$. The grain should be defined in a way that the number of tasks N created is big enough compared to the number of processors P to give flexibility to the scheduler. The cost of computing $f()$ over an interval [a,b) is defined as

$$C_F(task)$$

and the average cost per processor as

$$avg_C = \frac{\sum C_F(task_i)}{p}$$

Figure 2.5 shows how to balance the load while mapping tasks to cores.

```
dest_core = 0
cost = 0
task_beg = 0;
for (int i=0; i<N; ++i) {
   cost += cost(task_i);
   if (cost >= average_cost) {
   // all the tasks from task_beg to i are assigned to dest_core
      task_beg = i+1;
      cost = 0;
    }
}
```

Figure 2.5: Static Scheduling whith known task costs

2.4.2 List Scheduling

In this scheduling algorithm, all the tasks are initially in a global queue. Whenever a processor is idle, it is assigned the first task on the list. Upper-bounds on execution time are given by(GRAHAM, 1969) :

$$T_{lpt} <= (2 - \frac{1}{P})T_{optimal}$$

A variation of this method is called LPTF where the tasks are sorted by descending order of cost. Upper-bounds in this case is given by:

$$T_{lpt} <= (\frac{4}{3} - \frac{1}{3P})T_{optimal}$$

2.4.3 Work Stealing

In work stealing, each processing unit has its own queue of tasks to process. Whenever a processing unit completes them, it tries to steal more tasks from another processing unit. Each new task created is added to its own queue.

This scheduling method is particularly interesting in applications where parallelism is expressed by the creation of recursive tasks. For instance, figure 2.6 shows a recursive algorithm of the transform problem where each call to the $transform()$ method generates a new task that can be executed in parallel. Work stealing is an efficient way to schedule such applications. More details on the work stealing method are presented on section 3.1

```
transform( input, output, N, F) {
   if (N <2)
      output[0] = F(input[0])
   else
   {
       transform( input, output, N/2, F);
       transform( input+N/2, output+N/2, N-N/2, F);
     }
}
```

Figure 2.6: A recursive transform

3 PRIOR WORK

In this section we summarize the work done so far on the domain of load balancing and general purpose GPUs.

3.1 Work Stealing Methods on Multi-core Architectures

Load balancing issues have been widely studied and different techniques were documented in the literature. One very popular practical technique is "work stealing". In (BLUMOFE; LEISERSON, 1999) it is given the first provably efficient work-stealing scheduler. It proves that a parallel execution on P processors using their work-stealing scheduler has an expected runtime of

$$\frac{T_1}{P} + O(T_\infty)$$

, where T_1 is the minimum serial execution time of the multi-threaded computation and T_∞ is the minimum execution time with an infinite number of processors.

There are several variants of work-stealing algorithms. These variants can be analyzed regarding the progress conditions (HERLIHY; SHAVIT, 2008, chap. 3.7) which depends on the usage of atomic registers and locking primitives employed by the three work-stealing operations: Pop, Push and Steal. (see section 4.2)

The Cilk-5 runtime system (FRIGO; LEISERSON; RANDALL, 1998) implements a *lock-based* work-stealing scheduler. It employs the Dijkstra's THE protocol for mutual exclusion (DIJKSTRA, 1965) which greatly reduces the lock overhead (FRIGO; LEISERSON; RANDALL, 1998, pg. 8) by only using systematic locks on steal operations at the head of the work queue. In this model, Pop and Push operations only use atomic registers. (ARORA; BLUMOFE; PLAXTON, 2001) presents 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., 2005) presents the first *dynamic memory* work-stealing algorithm, eliminating the potential overflow problems on ABP's algorithm. (CHASE; LEV, 2005) came with a simpler solution to this same problem by implementing unbounded dequeues as dynamic-cyclic-arrays. (HENDLER; SHAVIT, 2002) generalizes 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.

On current mainstream processors, the correctness of all these algorithms depends on the usage of special memory ordering functions (Fences instructions) to enforce the order of writes before reads. (MICHAEL; VECHEV; SARASWAT, 2009) introduces Idempotent Work-stealing which exploits relaxed semantics of some classes of applications to reduce the usage of expensive atomic instruction and store-load memory fences.

(GAUTIER et al., 2010) tries to reduce lock contention in work-stealing by combining steal request of several thieves. In addition, this scheme allows to bias the choice of the victim to steal, which leads to a better load balancing than the random approaches used in previous works.

3.2 Work Stealing on GPU

Recently, with the advent of the use of graphical processors for general purpose computing, scientists started to study those classical CPU load balancing methods on GPUs. (CHEN et al., 2010) uses molecular dynamics simulation to evaluate a centralized method of dynamic load balancing on single an multi-GPUs systems. This method implements a centralized task pool managed on the host side. On the device side, each block pops from its own private work-queue. The host process is able to monitor the state of the device work queues and push new tasks whenever one gets empty. Their results showed that, for unbalanced work-loads, task-based models can utilize the GPU hardware more efficiently than the standard CUDA scheduler.

(CEDERMAN; TSIGAS, 2008) 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 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 task stealing method with distributed work queues.

In (ANGEL; MICHAEL; BIVENS, 2010) 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(BLUMOFE; LEISERSON, 1999). They exploit the performance and synchronization characteristics of the GPU memory hierarchy by using a combination of shared and global memory queues. However, in their scheme, shared memory is used just as a way to speed up warp accesses to their private work-queues and steals are always performed randomly on global memory (i.e. steals do not take into account memory locality). The throughput overheads measured were by a factor of 3 for queues on shared memory and 15 for queues on global memory.

In (TZENG; PATNEY; OWENS, 2010), 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. Their approach combines the load-balancing model to *Uberkernels* [1] and implement the Reyes rendering pipeline entirely on the GPU without the overheads of launching different kernels at each stage of the pipeline.

3.3 Positioning

The literature presents several studies of dynamic load balancing on GPUs applied to irregular problems. However, to our knowledge, there is no reference study about the use of this techniques with regular or semi-regular problems on GPUs. Matrix multiplication and transformation are classical regular problems specially well suited to the data-parallel model of GPUs. Such problems have very efficient implementations using the CUDA programming model (NVIDIA, 2010a). Additionally, libraries like Thrust (HOBEROCK; BELL, 2011) for CUDA provides higher-level interfaces to common parallel algorithms that greatly enhance programmers productivity.

On the other hand, there is still a lack of a generic programming model that could be used for both classes of parallelism: regular and irregular. While work-stealing is proved to be a good scheduling technique for irregular workloads, in this work we also evaluate its performances and overheads when scheduling regular tasks. This study is motivated by the need of a generic programming model that is able to deal with any kind of workload. Such programming model could be implemented in libraries and languages for parallel programming like KAAPI and CILK.

As a result, we present here a simple and generic programming model based on KAAPI. This model is validated using a regular array transformation and then with the octree partitioning problem.

[1] The Uberkernel programming model packs multiple execution paths into one single kernel, this eliminates explicit barrier synchronization and overhead between kernel launches.

4 A PROGRAMMING MODEL FOR GPGPU

In this section, we describe our solution proposed for dynamic load balancing on GPUs. We introduce the work queue interface and the different configurations implemented.

4.1 A programming model based on Work Stealing

Previous work (FRIGO; LEISERSON; RANDALL, 1998; GAUTIER; BESSERON; PIGEON, 2007; PHEATT, 2008) has demonstrated the benefits of task based programming models. In the previous section 3.1 we recall that, thanks to a work stealing scheduling, it is possible to effectively schedule a program that uses tasks in an on-line manner (i.e. on the fly creation of tasks).

However, this is not what is advocated in the traditional programming model of GPUs. The GPU programming model provided by CUDA directly exposes the programmer to the characteristics of the underlying hardware.

We show later on this work how to efficiently execute very regular programs on GPUs using a scheduling algorithm based on work stealing.

In section 5, experimental results actually demonstrate the benefits of such an approach:

- Firstly, it allows regular programs to efficiently run on GPUs, which is to our knowledge the first result in this direction.

- Secondly, this approach allows to execute recursive problems (like octree partitioning) as efficiently as with specific implementations for these problems (Our implementation was compared with a reference code published by (CEDERMAN; TSIGAS, 2008)).

Finally, we propose a programming model at two levels:

1. A program is divided into tasks, possibly recursively, which are scheduled by work stealing.

2. However, in contrast to classic multi-core architectures, each task runs in a SIMD (vector) mode, thereby extracting the computing power of GPUs.

4.2 Work Queue Implementation

The work queue implemented consists of a queue of tasks which is managed by the structure shown on Figure 4.1. This structure consists of two pointers, *beg* to the begging and *end* to the end of a double-ended queue (figure 4.2). Additionally, each workqueue is associated to a *mutex* variable used to control its access in conflicting cases.

```
typedef struct {
  volatile wq_index_t beg ;
  volatile wq_index_t end ;
  volatile atomic_t mutex ;
} workqueue_t;
```

Figure 4.1: The work queue structure

This structure is accessed by tree different methods: Push, Pop and Steal (Figure 4.3). The push operation is called by the thread to add a new task to its own work queue. This operation is always non-blocking and decrements the begging index. The Pop operation is used to get a new task from its own work queue. When there are no more tasks on the thread's own work-queue, it uses the steal operation to get tasks from other threads. The pop operation is non-blocking in the general case, the exception occurs when another thread tries to steal the same task being popped (Listing 4.1, line 10-26). Steal operation are always blocking and removes tasks from the end of the work queue by decrementing the *end* index.

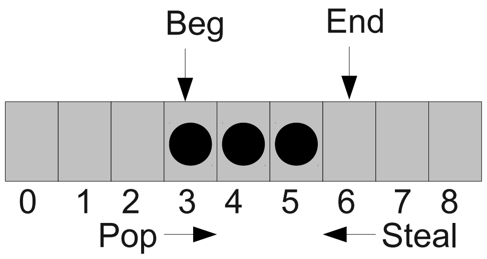


Figure 4.2: Dequeue

The interface that is shown on Figure 4.3 allows to Pop and Steal chunks of the DE-Queue, which corresponds to get more than one task a time. The size of the Pop or Steal is respectively defined by *max_size* and *size* parameters. Please note that a steal operation will fail if there is less than *size* tasks in the requested work-queue. Instead, Pop will return at most *max_size* tasks. Listing 4.3 present a pseudo-code of our work stealing scheduler. The actual code of the CUDA kernel implementing the work stealing scheduler is shown in more details on section 5.2.

```
int push( workqueue_t* kwq, wq_index_t* beg)
int pop( workqueue_t* kwq, wq_index_t* locbeg, wq_index_t* locend, int* max_size)
int steal( workqueue_t* kwq, wq_index_t* locbeg, wq_index_t* locend, int* size)
```

Figure 4.3: Methods interface for Work Stealing

```
Pop( wq, beg, end, smax)
{
  l_beg = wq->beg + smax;
  wq->beg = l_beg;
  if (l_beg < wq->end){
    *end = loc_beg;
    *beg = *end - smax;
    return SUCCESS
  }
  /* conflicting case */
  l_beg -= smax;
  wq->beg = l_beg;
  lock(wq->mutex);
  l_beg = wq->beg;
  size = wq->end - l_beg;
  if (size != 0) {
      if (size > smax)
          size = smax;
      l_beg += size;
      wq->beg = l_beg;
      unlock(wq->mutex);
      *end = l_beg;
      *beg = *end - size;
      return SUCCESS;
  }
   unlock(wq->mutex);
   return FAIL
}
```

Listing 4.1: Pop implementation.

```
Steal( wq, beg, end, size)
{
  l_end = wq->end - size;
  wq->end = l_end;
  if (l_end < wq->beg)
  {
    wq->end = loc_end+size;
    return FAIL; /* false */
  }
  beg = l_end;
  end = beg + size;
  return SUCCESS;
}
```

Listing 4.2: Steal implementation

```
while (1) {
  if( pop(my_wq, beg, end, popSize) ) {
   executeTasks(beg, end)
  }
  else{
    lock(victim_wq.mutex);
    steal(victim , my_wq.beg, my_wq.end,
                                   stealSize )
    unlock(victim_wq.mutex);
  }
}
```

Listing 4.3: Pseudocode of the work stealing scheduler.

4.3 Load Balancing Methods Comparison

Here bellow follows a brief description of the load balancing methods compared. Please note that the DEQueue of tasks and the work-queue structures are in global memory, and so are accessible by all threads on the device.

4.3.1 Static

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 blocks complete its job, the next blocks are scheduled.

4.3.2 Centralized Work Queue

The implementation of the centralized work-queue uses the same interface of Figure 4.3. In this case, there is only one MP that possesses a work-queue (containing the totality of the tasks). All the other MPs steal, with same pop size, on this single centralized work queue. The kernel terminates its execution when the DEQueue is empty. In reality, this work queue basically corresponds to the List Scheduling method (GRAHAM, 1969).

4.3.3 Distributed Work Queues

In this scheme, each multiprocessor has its own work-queue structure in global memory. The use of this work-queue is like in classical work stealing: multiprocessors do pop from their own queues, if there are no more tasks they randomly choose another work queue to steal from (see Listing 4.3). The termination condition of this scheme uses a global counter to control how many tasks have been executed. The multiprocessor stops its execution when it realizes that the counter have reached the total number of tasks. Please note that, in the transform application, there isn't any new tasks created during execution, so the scheduler already knows how many tasks will be executed.

The implementation of this work queue is based on KAAPI (GAUTIER; BESSERON; PIGEON, 2007) and will be detailed in section 5.2.

4.4 Optimizations

The proposed model leaves space to several optimizations. In the current implementation, all the multiprocessors have their work queue stored on global memory which has very high latencies (see section 2.1). The shared memory of the multiprocessor could store a lower level work-queue that would reduce the number of accesses to global memory. However this work-queue would only be stolen by blocs running on the same mutliprocessor. In the benchmarks we present later (section 5) there aren't many concurrent blocks sharing the same multiprocessor. This is because the proposed model favors the use of large blocks (512 threads per block) in order to take benefit of the SIMD parallelism inside a task, and from task parallelism between different MPs. One alternative is to reduce the number of threads per block allowing more concurrent blocks on the same multiprocessor. However, this choice introduces an important trade-off on the amount of task and data parallelism to use. The right choice might depend on the application, in the case of the array transform the tasks are highly data parallel, thus take more benefits from SIMD parallelism.

Another way to enhance performance, is to reduce the lock contention. As mentioned on the section 3.1 lock contention on steal operation can be reduced by combining several steal requests at the same time (GAUTIER et al., 2010). This approach also achieves a faster load balancing as, on the first successful steal attempt, the load stolen is equally divided among the thieves.

The transform operation can be coded in a way to allow idempotent work stealing (MICHAEL; VECHEV; SARASWAT, 2009). In our implementation of transform, the input array do not get modified during execution. It would be possible to relax the semantic of this algorithm and allow the same task to be computed twice by two different processors. This would reduce the utilization of locks in the scheduler but at the price of adding redundant task computation. Once again, this trade-off should be carefully analyzed to conclude about its performance impact on GPUs.

5 EXPERIMENTAL RESULTS AND ANALYSIS

This section first presents the test environment and methodology used for the experiments and then the results obtained with their corresponding analysis.

5.1 Test Environment and Methodology

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 (see table 2.1)

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 (CUDA events, figure 5.1) without counting overheads of the kernel launch nor memory transfers between host (CPU) and device (GPU).

```
cudaEventRecord(timer_begin,0);
arrayinc<<< nBlocks, threadsPerBlock >>>( in_array, out_array, n, size, grain);
cudaEventRecord(timer_end,0);
```

Figure 5.1: Kernel call and timers.

5.2 Implementations

This section presents the different implementations realized on this work.

5.2.1 Lock

A lock mechanism is needed either in the List Scheduling or Work Stealing methods to control the access to a critical section. High-end NVIDIA GPU devices afford the use of some atomic operations. For instance, the atomic compare-and-set (atomicCAS()), which is defined as(NVIDIA, 2010a):

- **atomicCAS(int* addr, int comp, int val)** : Atomically read **old** (located in **addr**); computes (**old==comp ? val : old)**, stores the result in **addr** and returns **old**.

Figure 5.2 corresponds to the protocol of mutual exclusion we used in our implementations. Note the read on lock variable before atomicCAS(). This is an optimization to reduce the number of calls to the atomicCAS() operation.

```
while ( (lock == 1) || !(atomicCAS(lock, 0, 1)== 0) ) ;
   //critical section
lock = 0;
```

Figure 5.2: Spin Lock

5.2.2 Transform function

Our benchmark consists in a simple array transformation. The program applies a function $f(x)$ to each element i of the input array and stores the result on the output array. The reference parallel implementation was taken from the Thrust library (HOBEROCK; BELL, 2011) and is shown in Figure 5.3.

```
__global__ void transform(int* iArray, int* oArray, int n) {
      const int grid_size = blockDim.x * gridDim.x;
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      while(i < n ){
         oArray[i] = f ( iArray[i] );
         i += grid_size;
      }
   }
```

Figure 5.3: CUDA kernel for the Transform function.

```
int increment( data ){
   for (int j=0; j<grain; ++j)
      data++;
   return data;
}
```

Figure 5.4: The $f()$ function applied to each position of the array

5.2.3 Work Stealing Scheduler

Listing 5.1 shows the main kernel that implements the work stealing scheduler. Note the use of the special CUDA variables *blockIdx.x* and *threadIdx.x* that identify, respectively, the block inside the grid and the thread inside the block. As can be seen in line 3 of Listing 5.1, each block has its own stealable work queue. At line 5 the execution path of the thread 0 of each block diverges in order to pop or steal tasks. This execution re-converges at line 16 when all the threads inside the block synchronize before actually executing the SIMD task on line 17.

```
 1  __global__ void
 2  arrayinc(float* g_idata, float* g_odata, int popSize) {
 3   int *my_wq = kwq[blockIdx.x];
 4   while (1) {
 5     if (threadIdx.x == 0) {
 6        if( ! pop(my_wq, locbeg, locend, popSize) ) {
 7           int *victim_wq = kwq[rand()];
 8           int stealSize = lenght(victim_wq)/2;
 9           if ( stealSize >= popSize ) {
10              mutex_lock(victime_wq.mutex);
11              steal(victim_wq, my_wq.beg, my_wq.end, stealSize )
12              mutex_unlock(victim_wq.mutex);
13           }
14        }
15     }
16     __syncthreads();
17     transform(g_idata,g_odata, locbeg, locend);
18     __syncthreads();
19
20     if ( terminate ) break;
21     }
22  }
```

Listing 5.1: Cuda kernel of the work stealing scheduler.

5.3 Reference time

The Thrust implementation of the transform algorithm (figure 5.3) will be used as reference time for the rest of this section. However, as Thrust is intended to facilitate the development in CUDA, their template interface do not allow the programmer to setup the kernel configuration (i.e number of block and block size). Instead these parameters are determined by the Thrust's own heuristic, which tries to maximize the GPU occupation. For benchmark purposes, it is interesting to be able to configure the number and size of Blocks of a kernel execution. For this reason, it was implemented a thrust-equivalent transform that allows to configure the kernel launches. This version will be referred in the rest of this work as the *Static* version.

Figure 5.5 compares the performance of the *Static* and the *Thrust* implementations. Please note that the variation of the number of Blocks do not apply to Thrust. For the

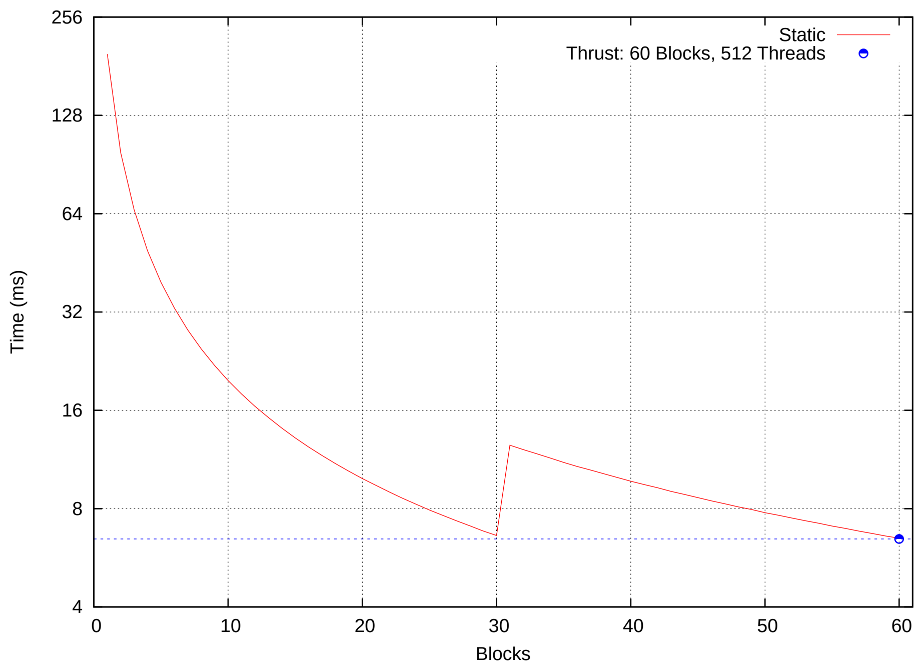


Figure 5.5: Performance comparison between Static and Thrust implementations

transformation function tested (array increment as on figure 5.4), Thrust always configure the kernel launch with 60 Blocks of 512 threads. The Thrust execution time obtained was 6.460ms while, for the same kernel configuration, the Static version achieves 6.497ms. Standard deviation were in the order of μs and were omitted. We conclude that Static has the same performance as Thrust at 60 block but is also very close to this optimum with 30 blocks. Between 30 an 60 blocks we can notice the presence of load imbalance.

5.4 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 Listing 5.1). The Work Stealing version of the transform program is used here to estimate the elementary cost of the pop operation. Figure 5.6 shows the total execution time of a transform applied to an array of 2^{20} elements. This benchmark used a kernel configuration of 1 Block of 512

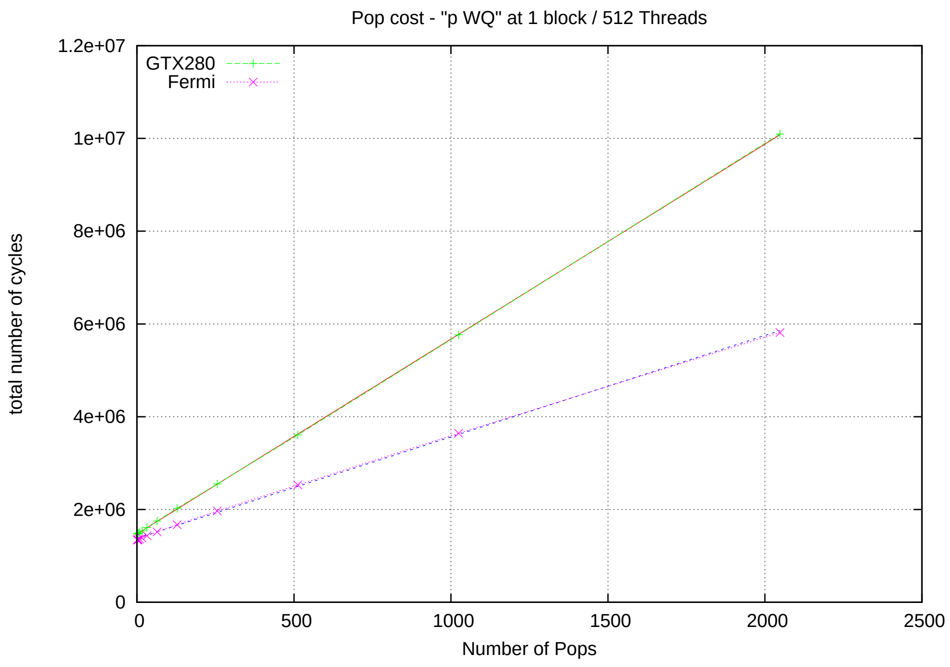


Figure 5.6: Pop cost estimation on a GTX280 and a C2050(Fermi) GPU

threads and varied the pop size from 2^9 to 2^{20}. Please note that the pop operation handles only work queue indexes, therefore the cost of a pop is independent of its size. The cost of the pop can then be estimated with a fit to the line equation $f(x) = a * x + b$ where x is the number of pops done. The obtained pop overhead was 5186.72 cycles (3.52 μs) on the GTX280 and 2187.32 cycles (1.92 μs) on the Fermi GPU.

5.5 Memory alignments

One important constraint on the CUDA architecture are the memory alignments. Global memory is organized in terms of aligned segments of 16 or 32 words. Memory accesses are performed by group of 16 or 32 threads (i.e by half or full warp) depending on the GPU's capability. As a consequence, full memory throughput is only achieved when all the threads of the half(or full)-warp access the same segment. Otherwise, a single access will compile to multiple transactions. In work stealing, to ensure that warp access to global memory are correctly aligned, the size of each pop and steal is adjusted to the closer multiple of 16.

Figure 5.7 shows how misaligned accesses hurt the performance of the transform application. In this example, the number of pops is minimized by using a pop size of approximately $N/Blocks$ where N is the size of the input array. When the result of this division is not a multiple of 16 the performance is compromised. The alignment-aware implementation also shows smaller standard deviations which indicates a more regular execution.

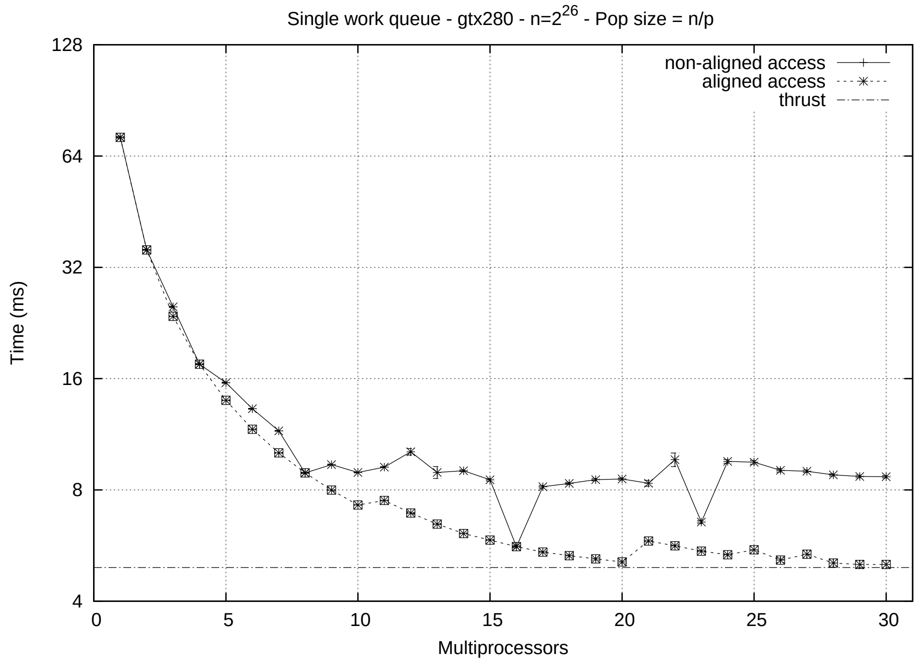


Figure 5.7: Memory alignment impact

5.6 Load Balancing on Regular Workloads

This section compares tree load balancing methods when used to manage regular workloads. As mentioned on section 5.2.2 a 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. In the context of our Transform function, a task consists in incrementing a fixed number of consecutive positions of the input array. More precisely, in the experiments presented here, **a single task contains 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 is fully parallel and makes all the threads of the block to work. This number of thread per block is the one that gives the best performances for a transform on a sufficiently large array. That is to say, an array big enough to express the necessary parallelism to use all the GPU resources.

5.6.1 Single Task Pop

Figure 5.8 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).

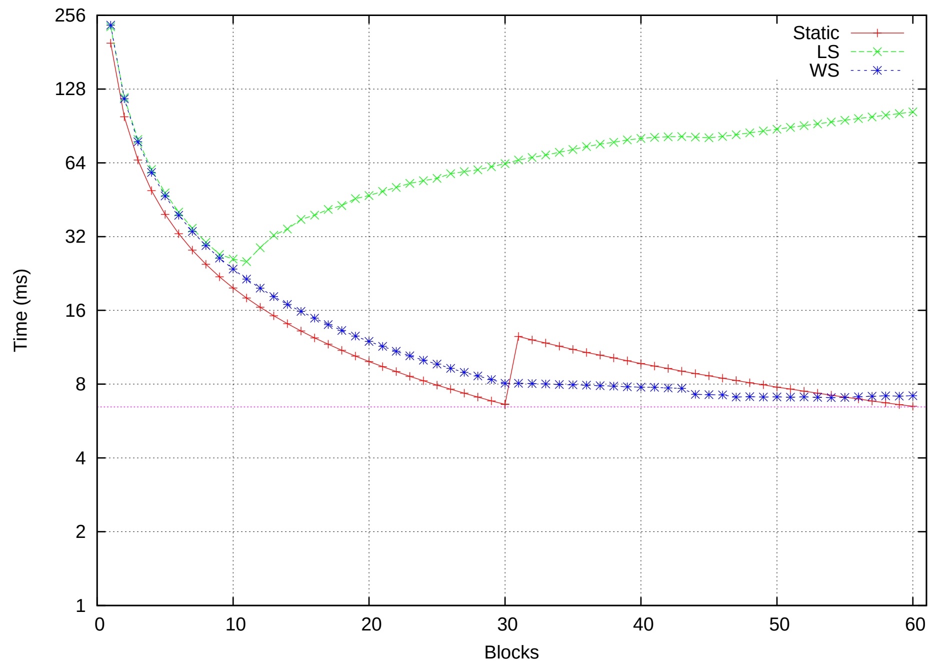


Figure 5.8: Static, List Scheduling (LS) and Work Stealing (WS) comparison for a pop size of a single task

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. Getting a centralized lock requires to spin on an expensive CAS operation (see section 5.2.1). As the time of execution of a single task is very short, blocks are often trying to acquire the lock which increase lock contention.

The static method shows the best absolute performance at 60 blocks, the maximum number of blocks that can run concurrently on the GPU (tests with more blocks didn't enhanced the execution time). However, note that the static method do not scale in a regular way as does work stealing. This can be seen on the performance drop at 30 blocks. This drop is due to the fact that with 31 blocks, one multiprocessor have two active blocks and twice more tasks which causes the load imbalance. Note that in this case the load imbalance was generated exclusively by the CUDA scheduler. This problem is completely solved by the work stealing algorithm.

5.6.2 Reducing Overheads

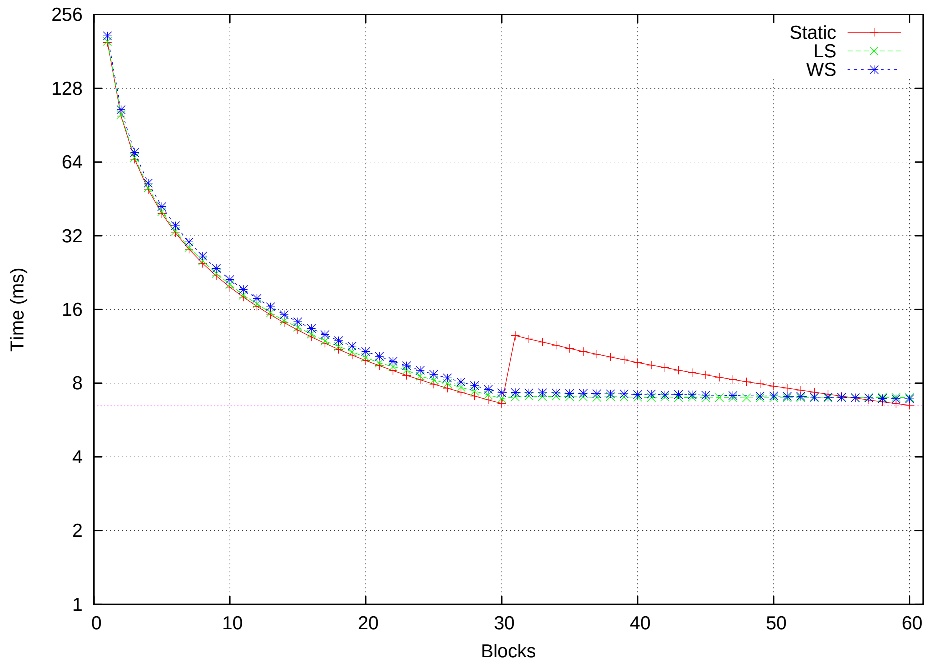


Figure 5.9: Static, List Scheduling (LS) and Work Stealing (WS) comparison on their best pop size

One way to reduce overheads is to amortize the cost to access the work queue. This can be done by popping more tasks at a time. Figure 5.9 show the best performances found for each method when tunning 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.

5.6.3 Pop size variation

Figure 5.10 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 greater 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). .

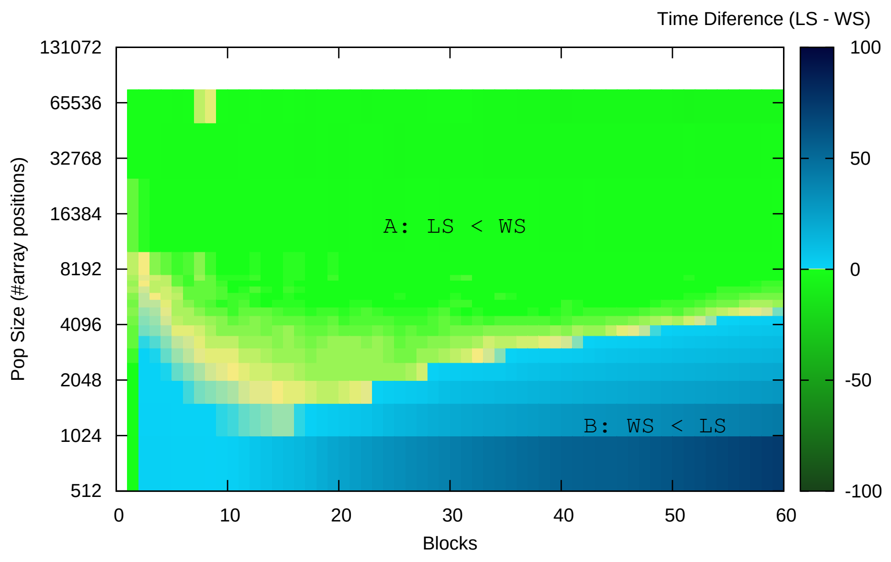


Figure 5.10: Time difference between LS and WS

5.7 Load Balancing on Irregular Workloads

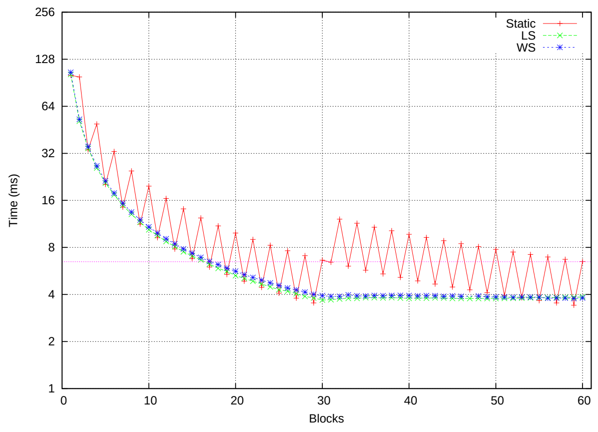
This section evaluates the load balancing methods with two different patterns of irregularity. This is done by nullifying the work of some tasks of the input array by the following patterns:

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

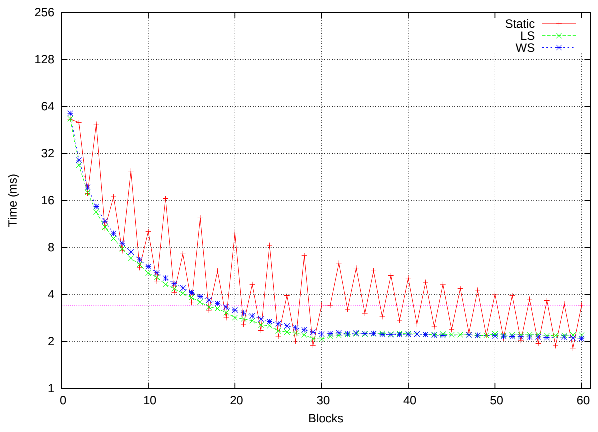
Figure 5.11 show the best results of each method of load balancing for the two patterns of irregularity. These tests represent the best configuration of pop size found for each method. LS uses a pop size of 10240 (20 tasks) with pattern 1 and 25600 (50 tasks) with pattern 2. WS uses pop sizes of 4096 (8 tasks) for both kind of patterns. This results clearly show the instability of the static method for irregular workloads and how dynamic scheduling approaches deal with it.

5.7.1 Octree partitioning

This section compares our work stealing method to ABP's algorithm (ARORA; BLUMOFE; PLAXTON, 2001) applied to the octree partitioning problem (section 2.3.2). The



(a) Pattern 1



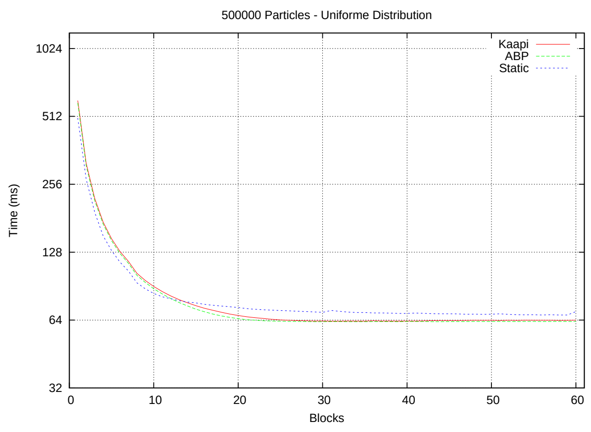
(b) Pattern 2

Figure 5.11: Transform problem on irregular workloads

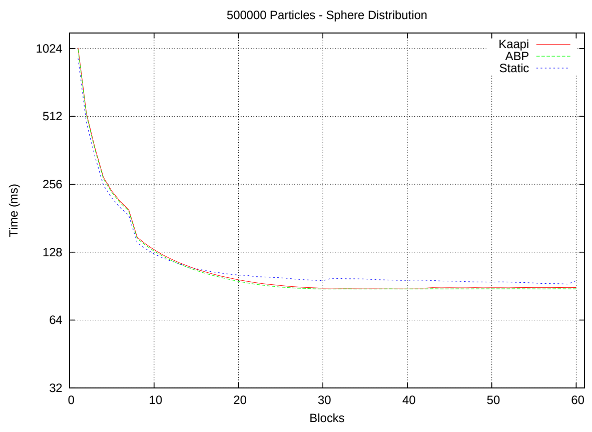
source code of the implementation was retrieved from Tsigas and Cederman's work (CEDERMAN; TSIGAS, 2008) (available in their project website) and adapted to use our interface of work stealing.

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 5.12 shows a comparison between our algorithm (labeled KAAPI) and 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.



(a) Cube Distribution



(b) Sphere Distribution

Figure 5.12: Load Balancing methods on Octree partitioning problem

We found that kaapi and ABP have similar performances which are $63.26ms$ for kaapi with 31 blocks and 62.94 ms for ABP with 35 blocks. Surprisingly the static version also performs very well and do not seems to suffer from load imbalance. This can be attributed to the fact that the number of tasks increases very fast and all the processors quickly get work to do. In fact, for the cubic distribution, on the 3rd step of the partitioning there are already enough tasks to keep all the multi-processors working. On the spheric distribution this happens on the 5th step.

5.8 Conclusions on load balancing methods

The Static scheduling showed to perform quite well for the regular array transform. This is manly due to the cyclic algorithm used to assign array elements (tasks) to threads. The cyclic approach make a good division of the work because it gives to all the blocks the same amount of tasks and also dissolves parts of the array that may contain more expensive tasks. However, this model is vulnerable to specific workload models (see section 5.7). 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 tunning 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 had 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.

6 CONCLUSION

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. Dynamic scheduling on GPGPU has been only studied by researchers for irregular applications such as octree application.

We presented here empirical results that attest the effectiveness of our model in real applications (octree). To the extent of our knowledge this is the first work to evaluate a regular problem (array transform) with dynamic load balancing on a GPU. The results obtained 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.

- We aslo compared our scheduler to another implementation of the well-known work stealing algorithm (ARORA; BLUMOFE; PLAXTON, 2001) and found equivalent performances on GPU on irregular problem.

6.1 Perspectives

In the short the term, we would like 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 5.6), suggest that new hardware capabilities notably, the presence of a full cache memory hierarchy (figure 2.2b), could be better exploited by our workqueue implementation. At long term, we envision the integration of this model in the KAAPI library which lacks the ability of scheduling on GPUs.

REFERENCES

ANGEL, M.; MICHAEL, M. M.; BIVENS, J. A. Dynamic Work Scheduling for GPU Systems. **Memory**, [S.l.], p.57–64, 2010.

ARORA, N. S.; BLUMOFE, R. D.; PLAXTON, C. G. Thread Scheduling for Multiprogrammed Multiprocessors. **Theory of Computing Systems**, [S.l.], v.34, n.2, p.115–144, Jan. 2001.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **Journal of the ACM**, [S.l.], v.46, n.5, p.720–748, Sept. 1999.

CEDERMAN, D.; TSIGAS, P. On dynamic load balancing on graphics processors. In: ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON GRAPHICS HARDWARE, 23. **Proceedings. . .** Eurographics Association, 2008. p.57–64. Code sources available at "`http://www.cse.chalmers.se/research/group/dcs/gpuloadbal.html`".

CHASE, D.; LEV, Y. Dynamic circular work-stealing deque. **Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures - SPAA'05**, New York, New York, USA, n.c, p.21, 2005.

CHEN, L. et al. Dynamic load balancing on single- and multi-GPU systems. **2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)**, [S.l.], p.1–12, 2010.

DIJKSTRA, E. W. Solution of a problem in concurrent programming control. **Commun. ACM**, New York, NY, USA, v.8, p.569–, September 1965.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. **Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98**, New York, New York, USA, p.212–223, 1998.

GAUTIER, T.; BESSERON, X.; PIGEON, L. Kaapi: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PARALLEL SYMBOLIC COMPUTATION, 2007. **Proceedings. . .** ACM, 2007. p.15–23.

GAUTIER, T. et al. **Requests Combining in Work Stealing**. "Private communication".

GRAHAM, R. Bounds on multiprocessing timing anomalies. **SIAM Journal on Applied Mathematics**, [S.l.], v.17, n.2, p.416–429, 1969.

HENDLER, D. et al. A dynamic-sized nonblocking work stealing deque. **Distributed Computing**, [S.l.], v.18, n.3, p.189–207, Dec. 2005.

HENDLER, D.; SHAVIT, N. Non-blocking steal-half work queues. **Proceedings of the twenty-first annual symposium on Principles of distributed computing - PODC '02**, New York, New York, USA, p.280, 2002.

HERLIHY, M.; SHAVIT, N. **The Art of Multiocessor Programming**. [S.l.]: Morgan Kaufmann, 2008. 528p.

HOBEROCK, J.; BELL, N. **Thrust Parallel Template Library**. 2011.

KAAPI. **Kernel for Adaptative, Asynchronous Parallel and Interactive programming**. [accessed on 18-october-2011], "`http://kaapi.gforge.inria.fr/dokuwiki/doku.php`".

KANTER, D. **NVIDIA's GT200**: inside a parallel processor. [accessed on 10-June-2011], `http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1`.

KANTER, D. **Inside Fermi**: nvidia's hpc push. [accessed on 10-June-2011], `http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932&p=1`.

KIRK, D. Nvidia cuda software and gpu parallel computing architecture. **ISMM**, [S.l.], 2010.

LAUTERBACK, C. et al. **Work distribution methods on GPUs**. [S.l.]: University of North Carolina, 2009. (TR009-16).

LINDHOLM, E. et al. NVIDIA Tesla: a unified graphics and computing architecture. **IEEE Micro**, [S.l.], v.28, n.2, p.39–55, Mar. 2008.

MICHAEL, M. M.; VECHEV, M. T.; SARASWAT, V. a. Idempotent work stealing. **ACM SIGPLAN Notices**, [S.l.], v.44, n.4, p.45, Feb. 2009.

MICHEL, B. S. H. S. GPU computing—General-purpose GPU computing. **Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06**, New York, New York, USA, p.233, 2006.

NVIDIA. **NVIDIA CUDA C Programming Guide**. [S.l.]: NVIDIA Corporation, 2010.

NVIDIA. **CUDA C Best Practices Guide**. [S.l.]: NVIDIA Corporation, 2010.

PAPADOPOULOU, M.; SADOOGHI-ALVANDI, M.; WONG, H. Micro-benchmarking the GT200 GPU. **Computer Group, ECE, University of Toronto, Tech. Rep**, [S.l.], 2009.

PHEATT, C. Intel threading building blocks. **J. Comput. Sci. Coll.**, USA, v.23, p.298–298, April 2008.

TZENG, S.; PATNEY, A.; OWENS, J. Task management for irregular-parallel workloads on the GPU. In: CONFERENCE ON HIGH PERFORMANCE GRAPHICS. **Proceedings. . .** Eurographics Association, 2010. p.29–37.