COLLECTIVE COMMUNICATION AND BARRIER SYNCHRONIZATION ON NVIDIA CUDA GPUs

September 10, 2009

Contents

1 Introduction **6**

1.1 Identification of the problem 6

1.2 Motivation and proposed solution 7

1.3 Related Work 7

1.4 Thesis outline 8

2 NVIDIA GPU architecture **10**

3 Methods of synchronization within GPUs **13**

3.1 NVIDIA Provided Operations 13

3.1.1 NVIDIA atomic native functions 14

3.1.2 Wait-free and t-resilient 15

3.1.3 Built-in barrier synchronization 16

3.2 Constant Time Race Resolution (CTRR) 17

3.2.1 Native Kernels 19

3.2.2 Reduction operations 21

3.3 Global Block synchronization (GBS) 24

3.3.1 Global State Restoring 27

3.3.2 GBS Library 27

4 Performance results **28**

4.1 Native kernel `p_any()` using atomic operations 29

4.2 CTRR native kernels 31

4.3 GBS kernels 36

4.4 Using GBS to implement some AFAPI functions 44

5 Conclusion and future work **48**

References **50**

List of Tables

1 Atomic Operations available in NVIDIA CUDA GPUs 14

2 Test Machines Information 28

3 GPU specifications 28

List of Figures

1 CUDA application stack . . . 10

2 Threads, Blocks and Grids in NVIDIA GPUs . . . 11

3 NVIDIA GPU hardware model . . . 12

4 `atom_p_any()` Execution Time . . . 29

5 p_atom_any() Early vs Late finish with inputs 0 . . . 30

6 p_atom_any() Early vs Late finish with inputs 1 . . . 30

7 p_selectOne() Execution Time . . . 31

8 p_any() Execution Time . . . 32

9 p_any() Early vs Late finish with inputs 0 - GTX 280 . . . 33

10 p_any() Early vs Late finish with inputs 0 - GTX 280 . . . 34

11 p_any() Early vs Late finish with inputs 0 - 8800GTS . . . 34

12 p_any() Early vs Late finish with inputs 1 - 8800GTS . . . 35

13 p_all() Execution Time . . . 35

14 __syncblocks() Execution Time . . . 36

15 reduceAdd() using global block synchronization . . . 37

16 reduceOr() using global block synchronization . . . 38

17 reduceAnd() using global block synchronization . . . 39

18 reduceMax() using global block synchronization . . . 39

19 reduceMin() using global block synchronization . . . 40

20 NVIDIA reductionAdd() Execution Time [1] . . . 40

21 reduceAnd() using CTRR and global block synchronization . . . 42

22 reduceOr() using CTRR and global block synchronization . . . 42

23 reduceMax() using CTRR and global block synchronization . . . 43

24 reduceMin() using CTRR and global block synchronization . . . 44

25 p_bcast() Execution Time . . . 45

26 p_count() Execution Time . . . 46

27 p_first() Execution Time . . . 46

28 p_quantify() Execution Time . . . 47

29 p_vote() Execution Time . . . 47

Abstract

GPUs (Graphics Processing Units) employ a multi-threaded execution model using multiple SIMD cores. Compared to use of a single SIMD engine, this architecture can scale to more processing elements. However, GPUs sacrifice the timing properties which made barrier synchronization implicit and collective communication operations fast.

This thesis demonstrates efficient methods by which these aggregate functions can be implemented using unmodified NVIDIA CUDA GPUs. Although NVIDIA's highest "compute capability" GPUs provide atomic memory functions, they have order N execution time. In contrast, the methods proposed here take advantage of basic properties of the GPU architecture to make implementations that are both efficient and portable to all CUDA-capable GPUs. A variety of coordination operations are synthesized, and the algorithm, CUDA code, and performance of each is discussed in detail.

KEYWORDS: GPU, barrier synchronization, CUDA, constant time race resolution, global block synchronization

1 Introduction

With the explosive growth of new high-end GPU systems and architectures, the use of GPUs is no longer restricted to graphic processing applications. General-purpose parallel computation using a GPU is now a common technique due to the good price to performance ratio, the large number of processing elements they offer, and the emergence of high level GPU programming languages and tools such CUDA [2], Brook+ [3], and BSGP [4]. Unfortunately, there is not yet a clear path to implementing complex applications on a GPU; there are a variety of issues associated with each of the alternative approaches. The primary issue that we address is common to nearly all of these approaches: the lack of efficient barrier synchronization and collective communications.

1.1 Identification of the problem

Recent GPU architectures are no longer designed as pure Single Instruction Multiple Data (SIMD) systems, but as a set of multiple SIMD multiprocessors glued together. Each one is capable of performing a different instruction at the same time. Each SIMD multiprocessor also is heavily multi-threaded to hide memory access latency. Thus, this model implicitly results in a lack of synchronization and communication between multiprocessors and traditional coordination primitives between SIMD cores which are not always available.

On the other hand, GPUs are now widely used to execute high performance applications, hence it is highly desirable that efficient methods for synchronization and communication are provided. Different GPU architectures provide different tools to reach synchronization. For example, ATI GPUs are equipped with data and instruction cache memory which, at least in theory, would allow the user to restore the global state of the system easily.

NVIDIA CUDA GPUs support for explicit atomic operations has gradually evolved with higher compute capability [1] devices. Thus, while GPUs with compute capability 1.0 do not support any of them, devices with compute capability 1.1 supports atomic functions operating on 32-bit words in global memory, and devices with compute capability 1.2 and 1.3 support atomic functions operating on memory shared within a SIMD core and atomic functions operating on 64-bit words in memory shared across SIMD cores as well [2].

Although including support for atomic operations as a tool to reach synchronization in general purpose parallel computation using GPUs was desired and expected, it is still not clear how to

[1] Compute capability is defined by NVIDIA [2] as the version of the core architecture presented in their systems and the minor improvements of new features added to new GPUs. Thus, the most basic device has compute capability 1.0 and the highest performance device has compute capability 1.3

achieve efficient synchronization between processing elements across different SIMD multiprocessors. The lack of portability associated with use of these constructs was a major motivation for finding alternative software solutions upon which the desired high-level operations could be efficiently implemented.

1.2 Motivation and proposed solution

GPUs offer the best price to performance ratio to execute parallel computation, either graphics applications, or general purpose. However, GPU architectures are not designed as purely SIMD machines nor MIMD, but rather something in between, and target applications to be optimized by the use of GPUs are mostly SIMD oriented, despite the fact that most of the emerging programming models and languages[2] are based in MIMD programming style like C [4].

This mixing of styles involved in the whole GPU system environment should make it feasible to execute MIMD programs in those new SIMD like machines more specifically the GPU. Researchers such as H. Dietz et al. demonstrated [5] this is possible. The concept is called MOG - MIMD On GPU.

But to make the MIMD environment run efficiently inside the GPU, it is necessary to provide tools to the new instruction set, compiler, or interpreter. Such tools include methods of threads synchronization like barrier synchronization and atomic functions, algorithms for reduction operations like Min, Max, Sum, And, Or, and algorithms for Voting and scheduling operations.

The solution proposed by this thesis document attempted to design, efficiently implement, and evaluate the performance of algorithms to:

1. Reach synchronization between threads within the GPU even if they exist in different SIMD engines.

2. Execute reduction and voting - scheduling operations within a GPU.

The algorithms are intended to provide solution to the synchronization problem independently of the hardware capabilities of the systems. They are intended to be as simple and efficient as possible.

1.3 Related Work

The highly evolving architecture of GPUs offers the best computational power per dollar ratio systems [6]. However, since most of the transistors on a GPU are dedicated to data processing and

[2]CUDA, ATI CAL, Brook+, OpenCL

few of them deal with cache and control flow [2], some features had to be compromised. One of those features is the synchronization between processing elements (PEs) or threads (virtual PEs) within different SIMD engines on a GPU.

The lack of synchronization mechanisms in recent GPUs and the need of synchronization in parallel applications was also identified in other publications [7], where a pair of synchronization mechanisms were suggested, both of them based in atomic operations. Each thread of a SIMD core accesses the shared memory across cores using atomic read/write operations and then a wait-free and t-resilient synchronization objects are created. More of this paper is discussed further in Section 3.1.

It was also found that it is possible to reach synchronization between SIMD cores, using the coalesced memory access of GPU [8] and taking advantage of the GPU memory distribution and the fact that multiple SIMD engines have access to a shared piece of memory (which NVIDIA CUDA refers as Global Memory). Handling the architectural restrictions, H. Phuong Hoaihave et al. [8] have shown that threads on different SIMD engines can reach synchronization. Three different access memory models were built by the researchers and the corresponding synchronization capabilities were evaluated.

The main contribution of this thesis is the algorithm design, testing, and performance analysis of several key global communication operations. Barrier synchronization between PEs across different SIMD engines in the GPU is a fundamental algorithm by itself, and also within the implementation of other operations. Various reduction and voting-scheduling algorithms were also implemented. All of these implementations are designed to work on any NVIDIA CUDA GPU regardless the compute capability, even if they do not have support for atomic operations; everything was tested using a compute capability 1.0 system.

1.4 Thesis outline

Chapter 2 of this thesis document presents the technical background and architecture information available for the most important high-end GPU vendors. It clarifies the concepts and the vocabulary used by them and shows the key points explored in the development of this research. Chapter 3 introduces a summary of some commonly used methods of synchronization between processing elements (PEs) and also presents the theory of the different synchronization methods proposed as an alternative multi-thread solution when using GPUs for general purpose computation. The results obtained when running kernels implementing a variety of synchronization methods are presented in

Chapter 4. Graphs of the execution time are shown there. The final Chapter presents the conclusions and suggestions for future work in this research area.

CUDA code for all the kernels used as benchmarks to produce the results presented in Chapter 4 are included in Appendix 5. Although they are not structured for release at this time, the intent is that these routines will be packaged and released as open source from Aggregate.Org in the near future.

2 NVIDIA GPU architecture

This Chapter introduces the most important architectural features found in the NVIDIA GPUs. It is not intended to be a complete description of the NVIDIA architecture; it serves primarily as a review of the model used here to develop the synchronization and reduction mechanisms proposed in this thesis. For a more detailed description of all the features of the NVIDIA CUDA GPUs, please refer to NVIDIA CUDA official programming guide[2].

Before introducing the architecture details, it is necessary to explain some concepts and terminology used with CUDA and the programming environment.

As defined by NVIDIA [2], CUDA is a general-purpose computing architecture which includes a programming environment that allows developers to use simple extensions to the standard C and C++ languages to develop applications for parallel computing on GPUs. In the future, it is claimed that CUDA will support other programming interfaces, such as FORTRAN. There are other non-CUDA environments as well [9], but this thesis restricts itself to the CUDA environment because it is the officially native environment. Figure 2 [3] shows the application stack layers used in a CUDA environment.

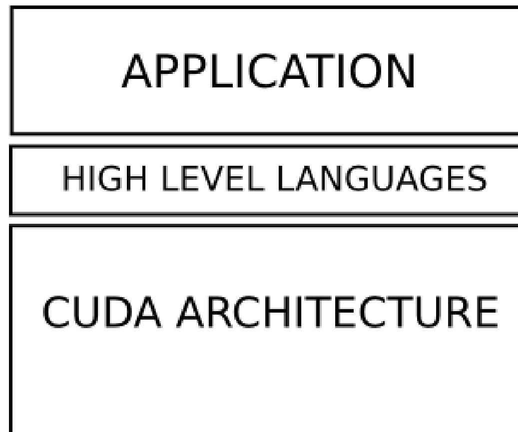

Figure 1: CUDA application stack

The way that the CUDA programming environment identifies that a function must be executed by the GPU is by defining a kernel function inside a C program. A kernel is the equivalent of a C function that declares the variables and computations that a GPU must execute for the current program on the top of the application stack. The kernel invocation is used by the system in order to differentiate between the two environments (CPU and GPU), so when a kernel is invoked it is executed simultaneously by every active thread within the GPU. Each thread is mapped to a virtual

[3] Figure 2 is based on the Application Stack graph described by NVIDIA in the CUDA programming guide [2].

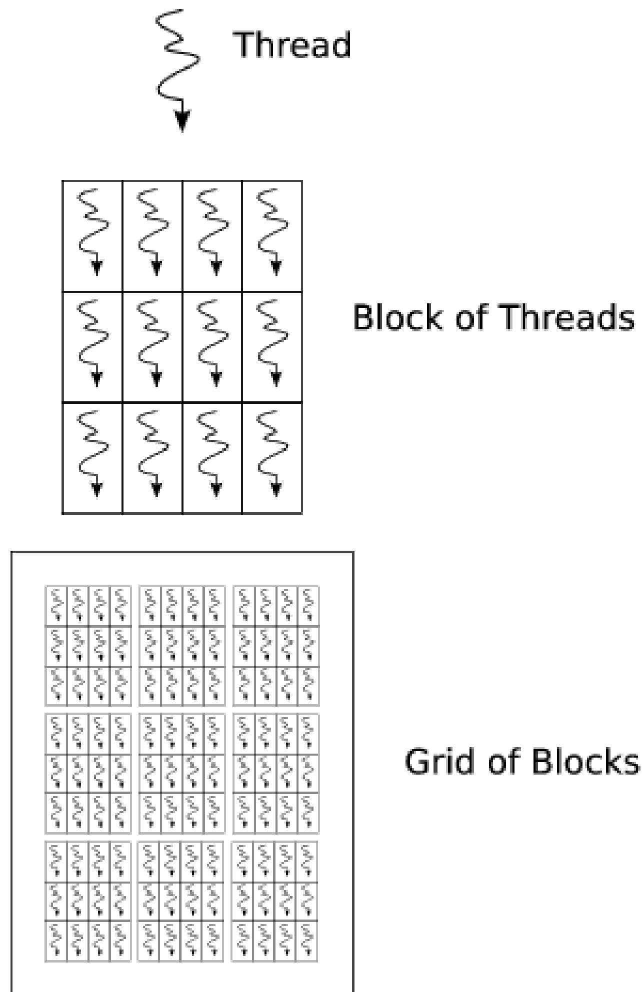

Figure 2: Threads, Blocks and Grids in NVIDIA GPUs

processing element within the GPU for execution.

Thus, treads in a GPU are seen as the basic unit of computation. As said above, every instruction declared inside a kernel is executed simultaneously by multiple threads and threads can be organized such that they form a one-, two-, or three-dimensional block. In this thesis, only one-dimensional blocks are analyzed. At the same time, multiple blocks are put together in a one or two dimensional grid. Figure 2 clarifies the relationship between threads, blocks, and grids.

With respect to the hardware architecture, NVIDIA GPU systems were designed as an array of multi-threading processors or SIMD engines, each one having eight scalar processor cores and each core having four single processing elements which are the physical fundamental computational units. Beside the scalar processors, each SIMD engine also includes an on-chip shared memory, a set of 32-bit registers per processor, a read only constant cache, and a read only texture cache.

Figure 2 shows the GPU hardware model. Every SIMD engine creates, manages, schedules, and

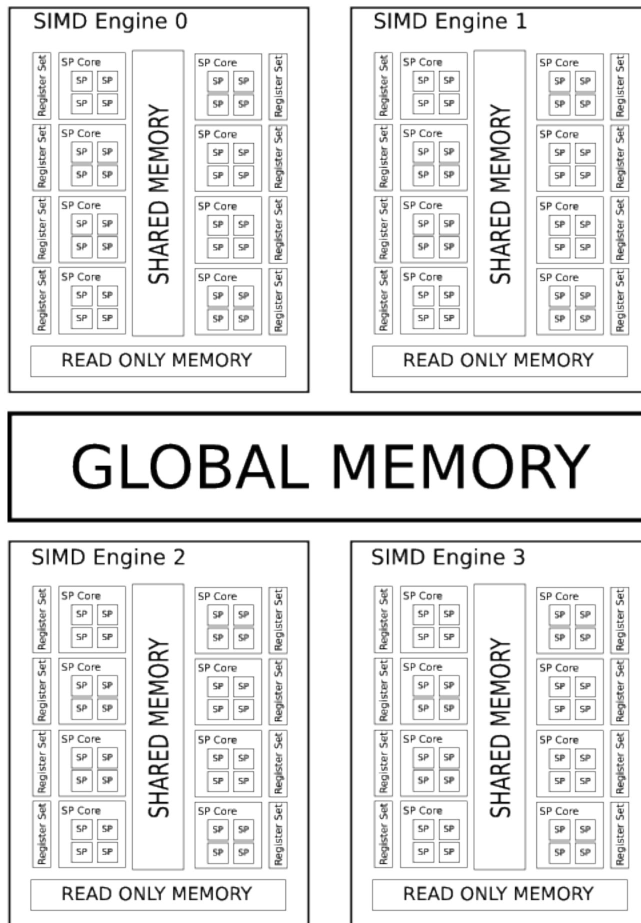

Figure 3: NVIDIA GPU hardware model

executes threads in groups of 32 threads each. To be precise, a group of 32 threads is actually performed by multi-threading over 8 physical processing elements within a SIMD engine rather than being executed truly simultaneously, but this minimum level of multi-threading cannot be avoided. Each group of 32 threads is called a warp. A warp executes one common instruction at a time – like a pure SIMD machine – so peak performance is reached when all threads within the warp agree in the instruction path. Hence, if one thread within a warp wants to execute a branch instruction, then this thread is put in a hold and the diverged instructions are executed serially. Once every branch path has been completed, the entire warp is released.

This document follows the terminology used by NVIDIA with respect to the different types of memory. The term global memory refers to the memory shared between SIMD engines. In contrast, the term shared memory refers to the memory that is associated with a single SIMD engine, and hence shared only between processing elements within that SIMD engine.

3 Methods of synchronization within GPUs

In order to achieve efficient parallel execution, synchronization between processing elements (PEs) has to be reached either by software native functions or by hardware capability. Barrier synchronization is one of the most common coordination strategies used to synchronize multiple PEs within a parallel system. A barrier forces all PEs to wait until every PE reaches the synchronization point and then releases all of them [10]. The barrier could be defined as a synchronization mechanism where no PE is allowed to proceed beyond the barrier until all other PEs have reached it. When the barrier is hardware implemented, PEs have to be somehow physically connected with each other to perform the synchronization as in KAPERS – Kentucky Adapter for Parallel Execution and Rapid Synchronization – [11]. When the barrier is software implemented, the synchronization is achieved as the compound effect of a set of primitive hardware operations.

Beside barriers, another popular type of synchronization used specially in operating systems to coordinate between processes is the *atomic* transaction model. This model comes from the concept of mutual exclusion in critical transactions performed by processes that may request access to a shared resource simultaneously; only one process should be allowed access at any given time. Basically, an atomic transaction is an indivisible transaction unit that forces concurrently-issued instructions to behave as if they were performed serially if they would interfere, thereby maintaining atomicity for each action. The next Section presents an overview of the barrier synchronization and atomic function operations present in NVIDIA hardware[4].

3.1 NVIDIA Provided Operations

Various synchronization tools have been proposed and widely implemented to coordinate PEs or multiprocesses entities such as those within the operating system or multi-threaded computations within a GPU, either directly using primitive atomic operations or through the use of different strategies based on higher-level atomic operations such as locks or semaphores. But what exactly is an atomic operation? Silberschatz et al. [12] define an atomic operation as a program unit that must be executed atomically. That is, either all the operations associated with it are executed to completion or none of them are performed.

[4]Atomic operations are available in devices with compute capability 1.3

Atomic Function	Description	Restriction
`atomicAdd()`	Add	64-bit word only supported for global memory
`atomicSub()`	Subtraction	Only supported for 32-bit word
`atomicExch()`	Exchanges the value in memory with a new value	64-bit word only supported for global memory
`atomicMin()`	Finds the min between value in memory and a new value	Only supported for 32-bit word
`atomicMax()`	Finds the max between value in memory and a new value	Only supported for 32-bit word
`atomicInc()`	`((val_in_mem >= new_val)` `? 0 : (val_in_mem+1))`	Only supported for unsigned 32-bit word

`atomicDec()`	`(((val_in_mem == 0) |`
	`(val_in_mem > new_val))`
	`? new_val : (val_in_mem-1))`
		Only supported for unsigned
		32-bit word

`atomicCAS()`	Compare and Swap	64-bit word only supported for global memory
`atomicAnd()`	Bitwise And	Only supported for 32-bit word
`atomicOr()`	Bitwise Or	Only supported for 32-bit word
`atomicXor()`	Bitwise Xor	Only supported for 32-bit word

Table 1: Atomic Operations available in NVIDIA CUDA GPUs

3.1.1 NVIDIA atomic native functions

Although the first release of NVIDIA CUDA GPUs – Compute Capability 1.0 – actually has a number of operations it performs atomically, it explicitly did not include support for any of the best known atomic operations. The second generation of GPUs – compute capability 1.1 (and higher[5]) – includes a set of atomic operations aimed to be used as synchronization tools. This evolution was driven by NVIDIA's desire to lead in general-purpose computing on GPUs; these atomic operations were frequently requested by users.

Atomic operations within an NVIDIA GPU perform a Read-Modify-Write uninterruptible cycle for 32-bit or 64-bit words in shared or global memory depending upon the compute capability supported. The atomicity of those operations lean on the fact that it is guaranteed that no other thread can read or write on the memory cell used by the atomic operation until it is completed. Table 1 presents a summary of the available atomic operations along with its restrictions in NVIDIA CUDA GPUs.

Based on the built-in functions presented in Table 1, it is possible to implement an atomic version of the `p_any()` kernel. It is called `p_atom_any()`. The goal of this function is to determine if at least one of the threads carries the value `datum`, or in other words, the goal is to implement a global OR using atomic functions.

[5] [2]describes the specifications for 1.2 compute capability systems, however they are not available in the market.

Algorithm 1 `p_atom_any()` CUDA code

```
if (g_odata[blockIdx.x] == 0){
        while( i<n && sdata[0] == 0){
            sdata[tid] |=  g_idata[i]
                | g_idata[i + blockDim.x];
atomicOr(&sdata[0],sdata[tid]);
      i += gridsize;
    }
  g_odata[blockIdx.x] = sdata[0];
}
```

The kernel `p_atom_any()` was implemented using the built-in function `atomicOr()`. Each thread within a block executes a bitwise atomic OR operation having as inputs its own shared memory cell and the designated cell of shared memory with the accumulated result. This way the value returned in global memory by the kernel its guaranteed to be logic high when *any* location of the input data stream analyzed by the treads has a true (non-zero) flag value. In the kernel implemented, for example, the flag true value is represented by an integer value of 1. Listing 3.1.1 presents the CUDA kernel of the `p_atom_any()` code.

There have been various methods published that would allow an operation like `atomicOr()` to execute without blocking resources such as Fetch-and-Op [13] which permits highly concurrent execution of synchronization primitives. Fetch-and-Op also has the property that when multiple processing elements simultaneously fetch and write a variable in memory, the result of these operations is the same as if they would be executed serially. Unfortunately, those implementations are somewhat hardware intensive, and hence less likely to be used in a GPU. Current implementations of NVIDIA CUDA GPUs cause the atomic operations to lock system resources while executing operations to guarantee that no other thread will interfere with the memory read-modify-write process. The result is that this function has order N execution time, much slower than the constant-time race resolution concept proposed in Section 3.2 which has constant time execution time. Chapter 4 presents a summary of the performance test results achieved and the comparison between `p_atom_any()` and the constant time race resolution version of ANY as well.

3.1.2 Wait-free and t-resilient

Other proposed solutions to reach coordination and synchronization between SIMD cores in NVIDIA GPUs have been proposed before by using atomic operations [7], where the GPU was modeled as N-SIMD machines sharing a shared memory. Each SIMD machine can process M threads in one clock cycle, each of the M threads inside a core can read/write one memory location in one atomic

step, then each core can read/write M memory location per atomic step.

After identifying the atomic capability of the model, researchers constructed a set of wait-free and t-resilient synchronization objects aimed to provide synchronization tools to programmers of parallel applications. Those objects are:

- Wait-free Long-lived Consensus Object
- Wait-free Read-Modify-Write Object
- t-Resilient Read-Modify-Write Object

The terminology used by the researchers to explain their theory is presented here as it was presented in the original paper [7].

Consensus number refers to the maximum number of PEs for which the object can solve a consensus problem. An n-consensus number allows n-PEs to propose their values and return only one of these values back to all of the n-PEs. *Long-lived* consensus object is a consensus object in which the variables are used more than once during the object life time. An object implementation is *wait-free* if any PE can complete any operation on the object in a finite number of steps regardless the execution speed of the other PEs. An object implementation is *t-resilient* if non-faulty PEs can complete their operations as long as no more than t PEs fail.

3.1.3 Built-in barrier synchronization

As part of the synchronization tools available in GPU systems, NVIDIA CUDA GPUs include a built in primitive called `__syncthreads()` used to synchronize threads within a block 2. Basically, the primitive `__syncthreads()` acts as a barrier synchronization point where no thread is allow to proceed before the `__syncthreads()` is over. Once this point is reached, the threads' execution continues normally. This synchronization primitive was built to coordinate thread communication within a block and make it the way to avoid data hazards such as read-after-write, write-after-read, or write-after-write when multiple threads are wanting to access the same address in shared or global memory.

However, since `__syncthreads()` works only within a block and NVIDIA GPUs allocate one block per SIMD multiprocessor, the primitive cannot be used to synchronize threads residing in different blocks or across different multiprocessors.

The description of the synchronization mechanisms implemented by using the hardware and software features presented in the GPUs are described within the next two Sections.

3.2 Constant Time Race Resolution (CTRR)

What happens when multiple threads want to write simultaneously a value in memory? According to the CUDA programming guide, it is not possible to know the number of serialized writes but it is guaranteed that at least one of the threads writes the value in memory. Empirically we found that the execution time of such kernels is constant rather than order N.

Although it is not exactly clear how the hardware handles the simultaneous writes to memory and which thread gains the right to do it – the fastest, the slowest, or a random thread –, this fact lead us to better understand the functionality of the NVIDIA CUDA GPUs and was the key to open a new optimal method of synchronization between threads for simple operations such `p_any()`, `p_all()`, used to determine the global AND or global OR state of the system, and `p_selectOne()` which returns the thread ID number (IPROC) of any active thread. The motivation to implement these native function kernels in NVIDIA CUDA GPUs aroused from the fact that they were available on the MasPar MP-1 SIMD machine [14] as part of the support for running MIMD code on SIMD hardware [15]. As expressed in Chapter 1, part of the motivation to research the functionality of high end GPUs was to be able to use them to implement a MIMD environment.

However, in the beginning it was not clear how NVIDIA CUDA GPUs managed the process of multiple threads writing simultaneously to a single shared memory cell. Since the CUDA programming guide [2] does not mention how the hardware handles this, at least two possibilities were assumed. The first possibility was that the hardware enforced every single thread to write its content into the memory cell, just as the original design of the MasPar MP-1. As one can imagine, following this method to write data into memory would be extremely inefficient and would hurt the overall performance of any process calling any of those simple functions. The second possibility was that the hardware somehow identifies multiple writers and handles the situation. Fortunately, that was the case and when a number of elements within a block try to write data to the same memory location, the hardware guarantees that one of them will have access to do it and somehow picks a thread to write its value in memory. This property is what we have called constant time race resolution - CTRR.

The fact that the hardware identifies when multiple threads are trying to access data at the same location is the key to making this mechanism function in constant time independent of the number of threads involved in the operation. The documentation makes it clear that this filtering is applied to convert multiple simultaneous reads from the same location into a single read and broadcast. However, it is easy to use this same hardware to detect write races and pick a winner for each race,

so that each race takes only one unit of time to complete. Empirically, this is what NVIDIA seems to have done.

Both writing and reading constant time race resolution are used to implement the `p_any()` kernel. The logic behind the function kernel is simple. As soon as a thread identifies that its value is equal to the passed argument `d`, it immediately decides to write the output value in memory. So it or any of the other threads (but only one) for whom the comparison with `d` was true is selected by the memory hardware to write the output value in the corresponding cell. When a thread reads that the value held in memory correspond to the solution value, the process is interrupted. In other words, when the solution is reached, it is broadcast to the other threads preventing them from continuing to work toward a solution. Appendix 5 presents the CUDA kernel codes.

The CTRR is an extremely fast mechanism when the nature of the solution can be adapted to picking a winner thread who carries a solution value. Questions like "Does any thread have the value D?" can be solved easily by using CUDA kernels implementing CTRR. Following this logic, common global functions like global Or, global And, and first IPROC efficiently could be executed by CUDA GPUs regardless of the compute capability. Such algorithms are discussed later in this Chapter.

Using CTRR to execute simple functions like `p_any(), p_all()`, or `p_selectOne()` is fast enough in terms of execution time, even taking into account the restriction imposed by the GPU architecture of being designed with multiple SIMD cores and claiming that it is better to use virtualization to divide the input data stream into multiple blocks of data, each block executed by an available SIMD engine, which forces it to recursively execute the kernel. That is, since CTRR acts so fast within a block of threads and the solution of such kernels can be reached without actually processing the whole array of input data, CTRR is the most efficient mechanism known to process such functions using CUDA GPUs.

However, CTRR is not suitable for every kind of algorithm. For example, in those functions where the kernel needs to make sure that every single thread had been evaluated, it is necessary to include a control mechanism. This extra mechanism inside the kernel implies more conditional instructions adding more clock cycles to the execution. It also could serialize thread execution when the control generates divergence between threads, further degrading performance.

In general, CTRR is very likely to be efficient with functions that often can take advantage of an early termination condition like ALL, ANY, MAX or MIN. Although verifying the early termination condition implies the use of an extra conditional instruction, it does not hurt the overall performance of the program because when an application needs to evaluate the kernels `p_any()` or `p_all()` it is

Algorithm 2 `p_selectOne()` algorithm

```
selectOne(volatile int *g_odata) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __syncthreads();
    g_odata[0] = i;
}
```

very common to find that at least a subset of elements meet the primitive condition, that is a group of elements are zeros or ones. In opposition, the early finish condition would hurt the performance of those functions that require arithmetic execution over all the stream elements or those where the early finish condition would not occur very often, e.g. 32-bit OR, or 32-bit AND. It is not very likely to find all 32 bits being zero or all 32 bits being one. In this case, evaluating the extra branch condition to verify the early finish state would cost more clock cycles to execute without gaining any advantage.

3.2.1 Native Kernels

Functions implemented using the CTRR concept are the simplest aggregate functions used in GPUs to read special features of the global state of the system at a determined point and are called native kernels. The main characteristic of those functions is that as soon as one of the threads reaches a positive solution for the function and writes the result back to the designated space in memory, all other threads stop evaluating the input data parameters. In most cases, they do not have to evaluate the whole stream of input data they receive, making them very efficient and suitable to the architecture of NVIDIA GPUs.

Based on the CTRR concept introduced in the last Section, it is possible to construct code for the native kernels `p_selectOne(), p_any(), p_all(),` and `p_warp_any()` successfully. As an example, a description of the algorithms and highlights of the code for such functions is presented here as well. However, the complete sequence of code is shown in Appendix 5.

The easiest way to see how CTRR works on NVIDIA GPUs is by understanding how the `p_selectOne()` native kernel works. Although it is a fairly simple CUDA kernel, it practically shows how threads race for access to memory when they need to write data there.

Algorithm 2 shows the kernel function. The idea is to identify the first thread that will write in memory. The way in which this is done is by letting the threads write the thread Id number. The value left in memory is the thread Id of the GPU chosen thread.

Algorithms 3 and 4 show the CUDA code for the `p_any()` and p_all() native kernels respectively.

Algorithm 3 `p_any()` kernel

```
if (g_odata[blockIdx.x]==0){
    while(sdata[0]==0 && i < n) {
        sdata[tid] |= g_idata[i] | g_idata[i + blockIdx.x];
        if (sdata[tid]==datum) sdata[0] = datum;
    i += gridsize;
    }
    g_odata[blockIdx.x]= (sdata[0]==datum);
}
```

Algorithm 4 `p_all()` kernel

```
while(sdata[0] == 1 && i < n){
     sdata[tid] &= g_idata[i] & g_idata[i + blockIdx.x];
     if (sdata[tid] != datum) sdata[0] = 0;
     i += gridsize;
  }
g_odata[blockIdx.x] = sdata[0];
```

These functions were implemented following four basic steps. First, an initial condition verifies that any previous thread has not written the solution to the designated cell of global memory. If no thread has reached the solution, then it is safe for the threads to keep searching for one. In the second step, every thread verifies that no other thread within the block has reached the solution before. It is also verified if the thread needs to process more data from the input stream. If both conditions are met, then input stream data from global memory is loaded into the block shared memory. An optimization suggested by NVIDIA [1] is that the first level of reduction could be executed while loading data from global to shared memory. The third step executes the kernel function by making each thread compare the value held in shared memory with the reference value. If a match is found, then the thread tries to write the solution in shared memory. The final step is writing the solution value in global memory. The kernel is executed recursively until a final solution is found or the whole input data was analyzed.

Both `p_any()` and `p_all()` kernels follow the same structure, the only difference between them is the comparison with the reference value done in step three. For `p_any()` the solution is found when the value in shared memory is equal to the reference value, while for `p_all()` the solution is found when the value in shared memory is not equal to the reference value.

Algorithm 5 implements a version of the global OR kernel by using the NVIDIA built in function `__any()`[6]. The `__any()` function, compares the reference value passed as a parameter against the

[6] According to NVIDIA [2], the function `__any()` is only supported by devices with compute capability 1.2 or higher. As from the best of our knowledge, compute capability 1.2 devices are not available in the market.

Algorithm 5 `p_warp_any()` algorithm

```
if (g_odata[blockIdx.x]==0){
    while(i<n && sdata[0]==0){
          sdata[tid] |= g_idata[i] | g_idata[i + blockDim.x];
          sdata[0] = (__any(1) != 0);
          i+=gridsize;
          }
    g_odata[blockIdx.x]=sdata[0];
}
```

value held by all the threads of the warp and returns a non zero value if the comparison is true for any of them. Having the `__any()` function available makes it easy to implement a `p_warp_any()` kernel.

Basically, the algorithm follows the same steps implemented in `p_any()` or `p_all()`, except that the comparison made in step 3 is made by the `__any()`, which implies that the evaluation is made warp by warp instead of block by block. Although the function works fine and could be seen as a viable solution, the overhead of working warp by warp and not block by block, plus the fact that the function `__any()` is not available for all the devices, makes this approach undesirable.

3.2.2 Reduction operations

Reductions are aggregate operations which have a single output value which is the result of applying an associative function to the input data. As opposed to the CTRR native kernels, reduction operation kernels have to process the whole input data stream in order to reach an accurate solution and in most cases including an early finish condition will affect the performance rather than make the execution faster. This is because for some reduction operations there are required arithmetic operations, while for some others the early finish condition rarely occurs. However, the principle over CTRR is based on the random race access that threads have when reading from or writing to memory. This characteristic makes the execution of the kernels easier and faster in the Section above because the threads carrying the solution run to write the right value in the corresponding cell of memory.

When implementing the reduction algorithms, it is necessary to know which thread has already taken into account the operation to avoid duplicity and which thread is pending to be included for the operation to make sure that the final result includes everybody's data. Adding the thread participation control into the kernel code reduces the overall performance of the reduction operation under CTRR.

The aggregate reduction operations implemented using the CTRR technique are `reduceOr()`, `reduceAnd()`, `reduceMax()`, and `reduceMin()`. As in the above Section, a description of the algorithms and the highlights of the CUDA code are presented here. The complete code is shown in Appendix 5.

Or reduction using CTRR can be achieved by letting each thread provide their data value to a specific cell in memory, `sdata[0]` in this case. Since CTRR is used here, the mechanism to implement the reduction operation is almost identical to that used with the native kernels. However, there is a major difference. Native kernels do not require all of the input data stream to be analyzed and most of the time as soon as one thread reaches the solution, the rest of them stop processing. With the reduction operations every thread has to be analyzed in order to get the final solution, hence a thread control must to be added to the kernel in order to deactivate threads that have already provided the information or those who carry redundant data already written to memory.

This control was implemented by comparing the data stored previously in `sdata[0]` with the data stored by each thread in a local register through the XOR operator. If the result of the value in register XOR the value in `sdata[0]` is not equal to 0, then the thread still needs to write some information to memory. Once every thread has written its value in memory, then a solution is reached. Algorithms 6 and 7 presents the CUDA code for the `reduceOr()` and `reduceAnd()` kernels respectively.

Again and as suggested by NVIDIA [1], the first level of reduction was executed during the initial load of data from global memory to local registers.

The overhead of having to implement the thread control and the repeated call of the kernel (simulating recursive application) to reach the solution, makes this mechanism a viable but not an optimal choice. However, it is possible to find positive results by mixing the CTRR algorithms with the Global Block Synchronization (GBS) mechanism explained later in this document. Timing results for the execution of reduction kernels are found later in Chapter 4.

Finding the minimum and maximum of a data stream is also a kind of reduction algorithm implemented in CUDA using the CTRR mechanism. Again, every single element of the input data array has to be analyzed to reach the solution, so thread control has to be implemented.

This time, the control was implemented using a flag which was turned off every time the thread attempted to write the value in memory and turned back on again when the thread finds out that the value written in memory was not its own. Another difference with the reductions explained before is that the reduction executed in the initial load step requires a temporary variable to be able to compare if the new loaded value is greater (smaller in the case of minimum reduction) than the

Algorithm 6 CTRR `reduceOr()` algorithm

```
unsigned int t = 0xFFFFFFFF;
sdata[0] = 0;
while (i<n){
    reg_thread |= g_idata[i] | g_idata[i + blockSize];
    i += gridSize;
}
while ((reg_thread & t)!=0){
    sdata[0] |= reg_thread;
   __syncthreads();
   t = sdata[0] ^ reg_thread;
}
```

Algorithm 7 CTRR reduceAnd() algorithm

```
unsigned int t = 0xFFFFFFFF;
unsigned int temp = 0xFFFFFFFF;
while (i<n){
    temp &= g_idata[i] & g_idata[i + blockSize];
    i += gridSize;
}
while ((~temp & t)>0){
    sdata[0] &= temp;
    __syncthreads();
    t = sdata[0] ^ temp;
}
```

Algorithm 8 CTRR reduceMax()

```
int flag = 1;
int temp = 0;
sdata[tid]=0;
while (i<n){
   temp = sdata[tid];
   sdata[tid] = (g_idata[i] > g_idata[i + blockSize]) ?
                 g_idata[i] : g_idata[i + blockSize];
   sdata[tid] = (sdata[tid] > temp) ? sdata[tid] : temp;
   temp = 0;
   i += gridSize;
}
while(flag == 1){
   sdata[0] = (sdata[0] < sdata[tid])? sdata[tid]: sdata[0];
   __syncthreads();
   flag = 0;
   flag = (sdata[tid] > sdata[0]);
}
flag = 1;
```

previous value loaded for this particular thread.

Algorithms 8 and 9 present the highlights of the CUDA code written for maximum and minimum reductions respectively.

3.3 Global Block synchronization (GBS)

The benefits of using barrier synchronization in parallel systems are widely known, but it is not clear how this concept could be applied to multicores SIMD GPUs. NVIDIA suggests [2, 1] the use of multiple thread blocks to maximize the performance of the system and synchronize blocks within the GPU by making the host CPU simulate recursive calls to CUDA kernels. Therefore, communications between thread blocks is possible at block invocation boundaries.

The problem with this type of interaction between blocks is simply that returning control to the CPU is expensive and, even more significantly, doing so flushes all local state in the processing elements. The cost of reloading registers and per-block shared memory can be very significant. Thus, the ability to interact across blocks without ending a code fragment often will be critical in achieving acceptable performance. In particular, barrier synchronization between blocks is very useful when executing operations like reduction of very large arrays, voting functions, or even scans (parallel prefix) as well; without barrier synchronization, the CPU must get involved.

The concept of global block synchronization implemented within GPUs was derived from a previous design of a barrier synchronization function included in the AFAPI library [16] and used to

Algorithm 9 CTRR reduceMin() algorithm

```
int flag = 1;
int temp = 0x7FFFFFFF;
sdata[tid]=0x7FFFFFFF; // sdata is type int
while (i<n){
   temp = sdata[tid];
   sdata[tid] = (g_idata[i] < g_idata[i + blockSize]) ?
                 g_idata[i] : g_idata[i + blockSize];
   sdata[tid] = (sdata[tid] < temp) ? sdata[tid] : temp;
   temp = 0x7FFFFFFF;
   i += gridSize;
}
while(flag == 1){
   sdata[0] = (sdata[0] <= sdata[tid])? sdata[0]: sdata[tid];
   __syncthreads();
   flag = 0;
   flag = (sdata[tid] < sdata[0]);
}
flag = 1;
```

perform multiple aggregate computation algorithms including reduction, scans, voting, scheduling, and communications functions implemented for the SHared Memory Adapter for Parallel Execution and Rapid Synchronization (SHMAPERS) [17]. AFAPI [16] is an abstract program interface library which provides multiple aggregate functions and operations intended to be run in Unix/Linux multiprocessor systems with shared memory.

In the barrier synchronization primitive included in SHMAPERS AFAPI, every time a barrier is reached each processing element places his barrier counter value and the data value in two separate buffers in the same dedicated cache line. Then, each processing element grabs data from cache and linearly performs local computations for the corresponding operation (reduction, scans, voting, scheduling or communications) grabbing data from cache after the barrier counter and data value are loaded in shared memory. The barrier is executed by the `p_wait()` primitive which basically assigns a barrier counter to each processing element. This counter is incremented every time a barrier is reached and no PE is able to perform computation unless it determines that all other PEs have reached the value. If a PE finds a barrier counter value greater that of its own, it means that another PE found the barrier was already reached hence it is safe to continue with computations. Since the cache lines fetched to detect the barrier hold the data as well, execution of the aggregate functions can be started while reading the barrier synchronization values [17].

Following the AFAPI concept, a block synchronization kernel called `__syncblocks()` was implemented for NVIDIA CUDA GPUs even though the same concept could be applied generically for

Algorithm 10 `__syncblocks ()` CUDA kernel

```
__syncblocks(register volatile unsigned int *barvec) {
/* Make all threads sync within each block */
__syncthreads();

/* Only one PE represents the block */
if (threadIdx.x == 0) {
        register int i = ((blockIdx.x + 1) % gridDim.x);
        register int barno = barvec[blockIdx.x] + 1;
        register int hisbarno;
        barvec[blockIdx.x] = barno;

        /* Keep looking at others until all are here
           or somebody else is past                  */
        do {
        /* Wait for this one to arrive */
        do {
                        hisbarno = barvec[i];
                } while (hisbarno < barno);
        /* Bump to next, wrapping if needed */
        if (++i >= gridDim.x) i = 0;
        } while ((hisbarno == barno) && (i != blockIdx.x));

        /* We're past...  tell everybody */
        barvec[blockIdx.x] = barno + 1;
        }
}
```

other cards. The `__syncblocks()` kernel and the logic behind it is presented now.

By using `__syncblocks()`, every block running the kernel is assigned a barrier counter which is incremented every time a barrier is passed. Thus, when all the counters hold the same value means a barrier is completed. However, if a block sees that the value held by another block is greater than its own value then another block is currently waiting at the next barrier. The actual block understands that the current barrier has completed and proceeds to update its counter without having to read every other block's counter. This logic prevents a block from looping all the way around when a barrier was already passed. Algorithm 10 presents the code for `__syncblocks().`

How exactly are `__syncblocks()` and the block synchronization used? The idea is to execute the computation within each block, and once each block reaches its own result, then it loads the value to global memory. Next, `__syncblocks()` is invoked to set the synchronization point. When the block synchronization is completed and every block has written its value in memory, then the first block is in charge of collecting the temporary results previously found by all blocks to perform the final computation and find the global solution. It should be noted that CUDA can be forced to

maintain memory access order within a thread by using the `volatile` keyword in declarations.

As an example, algorithm `p_reduceAdd()` follows the optimization path for the summation reduction operation using CUDA suggested by NVIDIA [1], and a final step is inserted calling the `__syncblocks()` function, resulting on better execution times for large arrays and slightly less efficient for small arrays of data. The description of `p_reduceAdd()` is included in Appendix 5.

3.3.1 Global State Restoring

The GBS mechanism restores the data held in global memory at a determined time which is equivalent to restoring the global state. In order to compare the performance of the GBS mechanism, a global state restoring algorithm was implemented.

This algorithm allows each thread in a block to read the value from global memory and transfer it to a local set of registers, where the set of data would be available for computation. Then, data is loaded back to global memory.

3.3.2 GBS Library

By using the kernel `__syncblocks()` as a barrier synchronization mechanism and using the same concept used when implementing AFAPI in SHMAPERS [17], one could implement and execute within the GPUs most of the AFAPI library algorithms. Here, we introduce a fraction of the AFAPI library version implemented using the GBS for NVIDIA GPUs.

The first kernel implemented was `__syncblocks()`, which implements barrier synchronization between blocks. The `reduce_Or(), reduce_And(), reduce_Max(),` and `reduce_Add()` calls perform the corresponding types of aggregate reductions. The `p_bcast()` function corresponds to the broadcasting function, `p_count()` counts how many elements voted for a given value, and `p_vote()` returns an array of bits with bit 2^{k}set if only if element k voted for the given value. Finally, `p_quantify()` returns 0 if no element voted for the given value, returns 1 if only one element voted for the given value and returns 2 if two or more elements voted for the given value.

Chapter 4 presents the performance results found after running the following algorithms using the GBS mechanism and Appendix 5 presents the detail of the CUDA code.

4 Performance results

To verify the effectiveness of the concepts presented in this thesis, a set of micro-benchmarks were run in two different machines and the execution times for each of the programs were measured. Each of the machines have a different type of GPU system. The first test machine, uses a high end NVIDIA GPU GTX-280 with compute capability 1.3 while the second test machine uses a low end NVIDIA GPU 8800-GTS with compute capability 1.0. Both machines run under Linux operation system Ubuntu 8.04 “hardy” with a 2.6.24 kernel.

Table 2 presents a summary of the CPU and Memory specs for both test machines and Table 4 presents the specifications of the GPU cards included in the test machines.

	Test Machine Specifications
Processor	AMD Athlon(tm) 64 X2 Dual Core Processor 4200+
CPU MHz	1,000
Cache Size	512 KB
Cores	2
RAM Memory	1 GB

Table 2: Test Machines Information

	Machine 1	Machine 2
GPU System	GTX-280	Geforce 8800 GTS
Compute Capability	1.3	1.0
Global Memory	1 GB	320 MB
Shared Memory per Block	16 KB	16 KB
Memory Clock	1107 MHz	800 MHz
Memory Bandwidth	141.7 GB/Sec	64 GB/Sec
Multiprocessors (SIMD Engines)	30	12
Cores (Scalar Processors)	240	96
Core Clock	1296 MHz	500 MHz
Registers per Block	16384	8192
Max Threads per Block	512	512
Clock Rate (card)	1.3 GHz	1.19 GHz

Table 3: GPU specifications

After having described the technical specifications of the test machines and the GPU systems they included, it is necessary to describe how the benchmark programs were implemented and executed to measure the execution time of the CTRR and GBS algorithms. In order to obtain the simulation graphics and make the results comparable with each other some restrictions were set. The number of threads per block and the number of blocks were fixed to 128 and 64 respectively. Since the main objective of this thesis was to probe the functionality of the new CTRR and GBS concepts and not to find the optimal parameters under which these concepts reach the peak optimal, fixing

the values of threads and blocks seemed like a valid procedure. Although the parameters for the number of threads per block and the number of blocks per grid were chosen arbitrarily, they are still representative numbers and they match the parameters used by NVIDIA to present their simulation results [1].

The number of elements in the input data stream was used as a variable parameter to run the benchmark programs. Eleven different values were used from 4K to 4M and the programs were executed 50 times for each parameter value in the two test machines. The CUDA program determines the average execution time using the `cutGetAverageTimerValue()` CUDA function [2] then, the mean average time value out of the 50 results was chosen.

4.1 Native kernel `p_any()` using atomic operations

Figures 4.1, 5, and 6 show the execution time of the `atom_p_any()` kernel when it was run on test machine 1 using a GTX-280 GPU. The input data for this test was a random vector with values 0 or 1 and the kernel included an early finish termination condition.

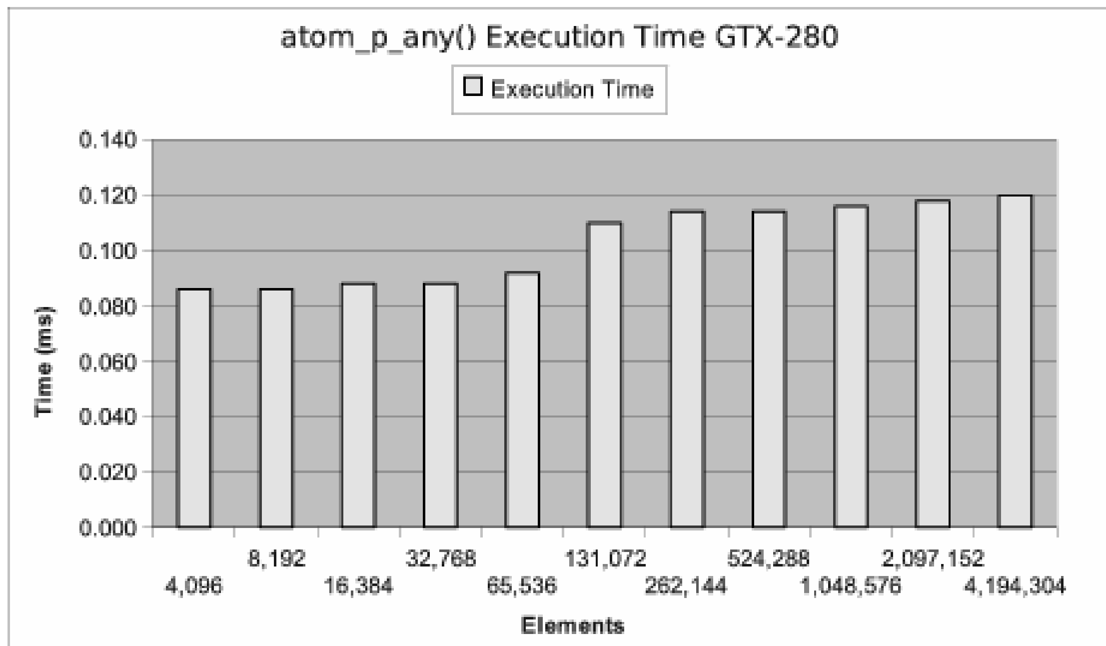

Figure 4: `atom_p_any()` Execution Time

To validate the improvement achieved with the early finish condition, four more benchmarks were run combining early and late termination when the input stream was all 1 or all 0. The results are presented in figures 5 and 6. This set of tests could not be performed in test machine 2 because the 8800 GTS system does not include atomic operation support (it is compute capability 1.0).

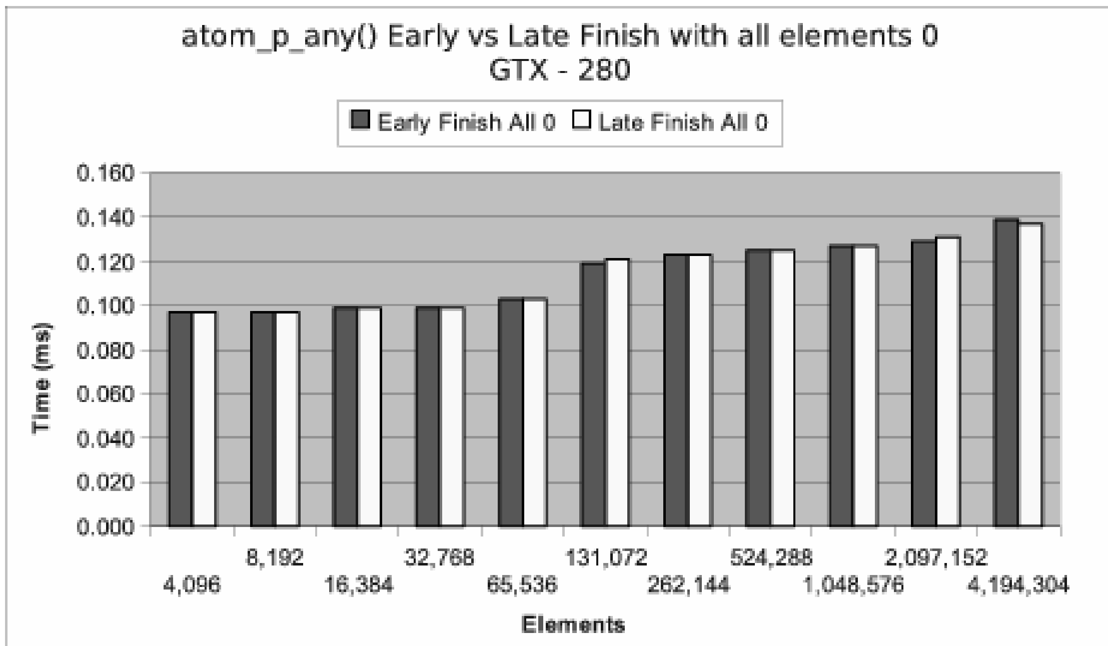

Figure 5: p_atom_any() Early vs Late finish with inputs 0

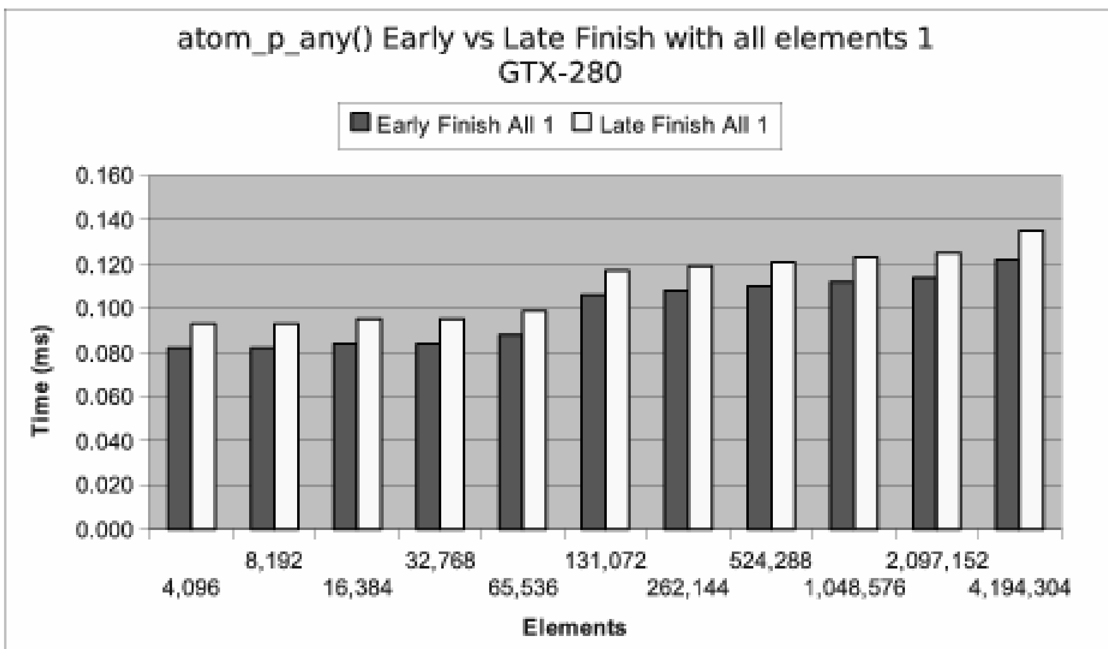

Figure 6: p_atom_any() Early vs Late finish with inputs 1

It is interesting to see that for less than 64K elements to reduce, the execution time is relatively stable and below 90 us; for 128K or more elements, the execution time slowly increases, going up to 120 us in the worst case. The reason behind this is because an atomic operation has to lock the hardware resources in order to enforce the read-modify-write cycle. This fact is less noticeable with a small input data stream because it can be hidden with the high computing performance of the single PEs, also, fewer elements imply less kernel recursion and less thread virtualization.

As expected, the early termination condition was very useful when the stream input was all 1 and saved some execution time compared with the results achieved for the late termination. Also, it was shown that when input data is all 0, there is no gain on using the early termination condition since the whole stream has to be read anyway.

4.2 CTRR native kernels

Figure 4.2 and 4.2 present the results of testing the basic CTRR native functions in both types of GPUs. It is clearly shown that the time used to find the solution by the kernel is mostly constant and very efficient compared with the performance reached with the NVIDIA built in `atomicOr()` function.

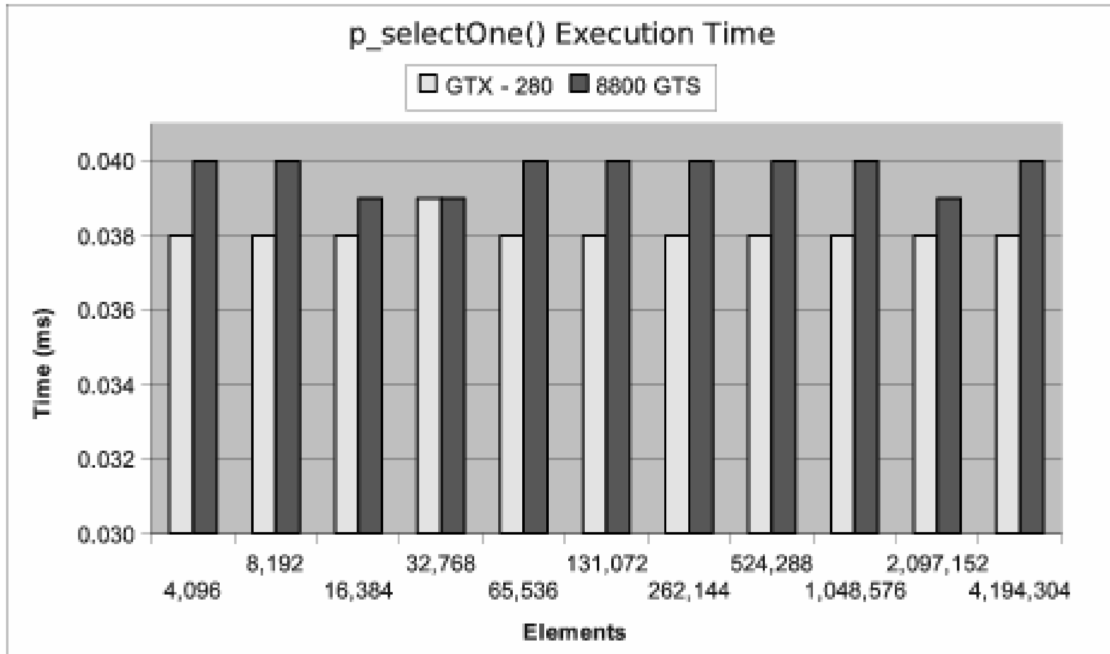

Figure 7: p_selectOne() Execution Time

Figure 4.2 shows the execution time of running the function `p_selectOne()`.This graph makes it very easy to understand and verify the functionality of the NVIDIA GPU system when multiple

threads try to reach the same spot in memory. The solution time is essentially constant, independent of the number of elements in the input data stream. It is also possible to see that the high end GTX-280 is slightly faster than the low end 8800 GTS. However, the difference in execution time between them is only about 2 us. This execution time difference could be due to the higher clock rate of the single processors included in the GTX-280, but the difference is surprisingly small no matter what the reason.

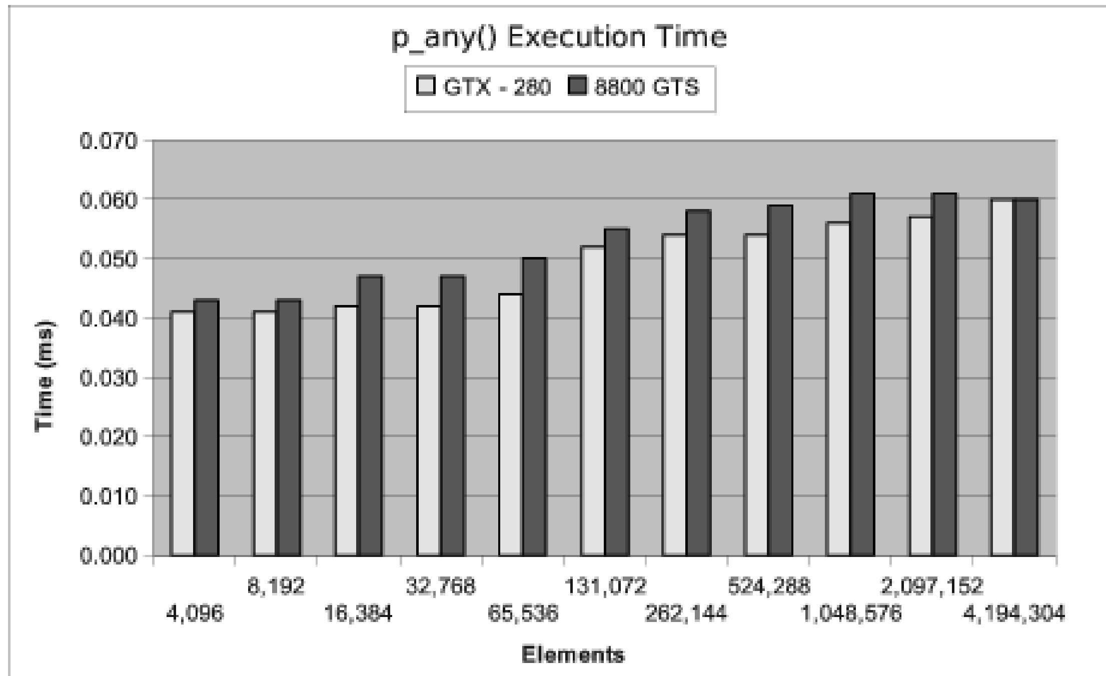

Figure 8: p_any() Execution Time

Figure 4.2 presents the execution found after running the `p_any()` native function in the test machines. The results follow the trend of constant time execution seen in the `p_selectOne()` algorithm; however, there are some differences. The data showed that for 64K elements or less the constant time resolution stands between 40 and 50 us and for over 128K elements the time is between 50 and 60 us. The difference between the values found here and those shown before in figure 4.2 are explained by the fact that threads in `p_any()` and `p_all()` performs a little computation, while threads in `p_selectOne()` do not make any.

Figures 9, 10, 11 and 12 present the results obtained when running the early and late termination conditions with input data stream either all 0 or all 1 for the `p_any()` native function. The same test could have been run for the `p_all()` kernel but since `p_any()` and `p_all()` only differ in the evaluation condition, we believe that the results found with `p_any()` also apply for `p_all()`.

The execution times obtained by running the benchmarks confirm the expected behavior. As in

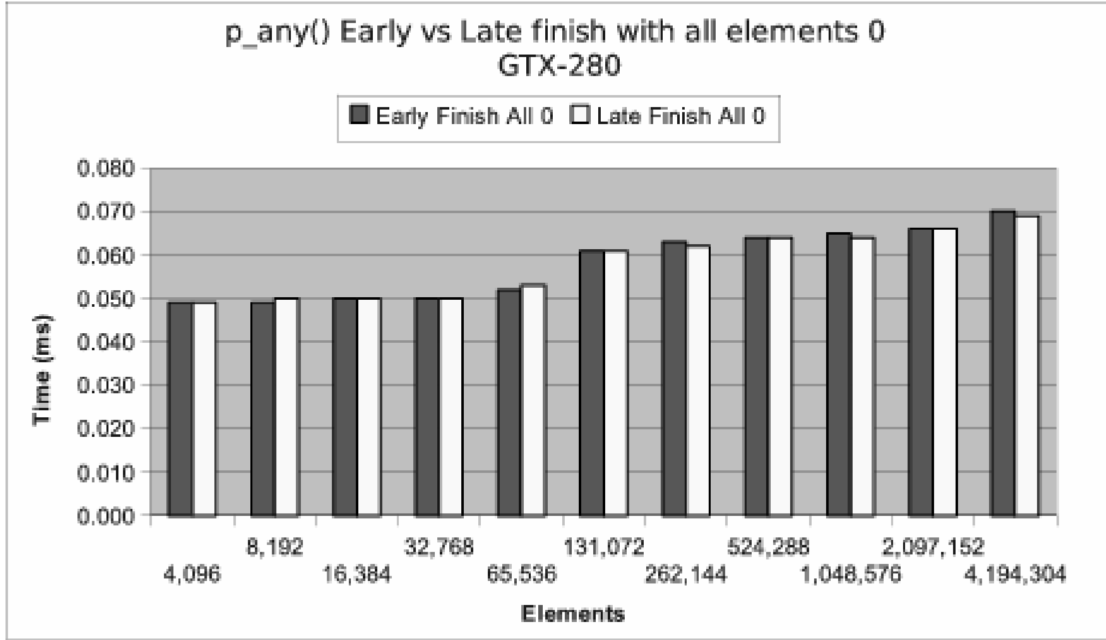

Figure 9: p_any() Early vs Late finish with inputs 0 - GTX 280

the `atom_p_any()` test, the early termination condition kernel with input data stream being all 1 was the fastest function in both GTX-280 and 8800 GTS systems. There was no effect on runtime for the kernel with early termination condition and input data stream being all 0 as compared to the performance obtained when running the kernel without the early termination verification and the same input data stream. Thus, the early termination test appears to be harmless even in the case in which it logically serves no purpose.

An interesting observation was found after running the `p_any()` benchmarks. In both test systems the difference between the execution times obtained with the high end machine and those found with the low end GPU are not significantly different. In fact for some of the tests, execution times found for the low end machine were slightly faster that those found on the GTX-280. This observation demonstrates that the CTRR mechanism appears to be a fundamental component of the GPU architecture and one would expect to obtain similar results with different machines even if they have different compute capabilities.

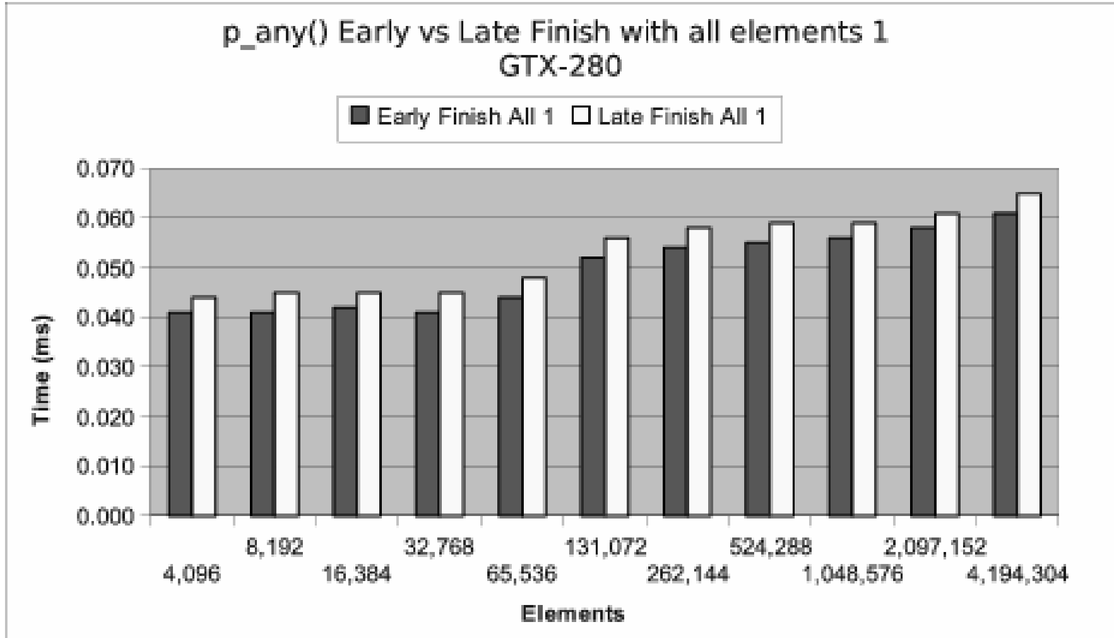

Figure 10: p_any() Early vs Late finish with inputs 0 - GTX 280

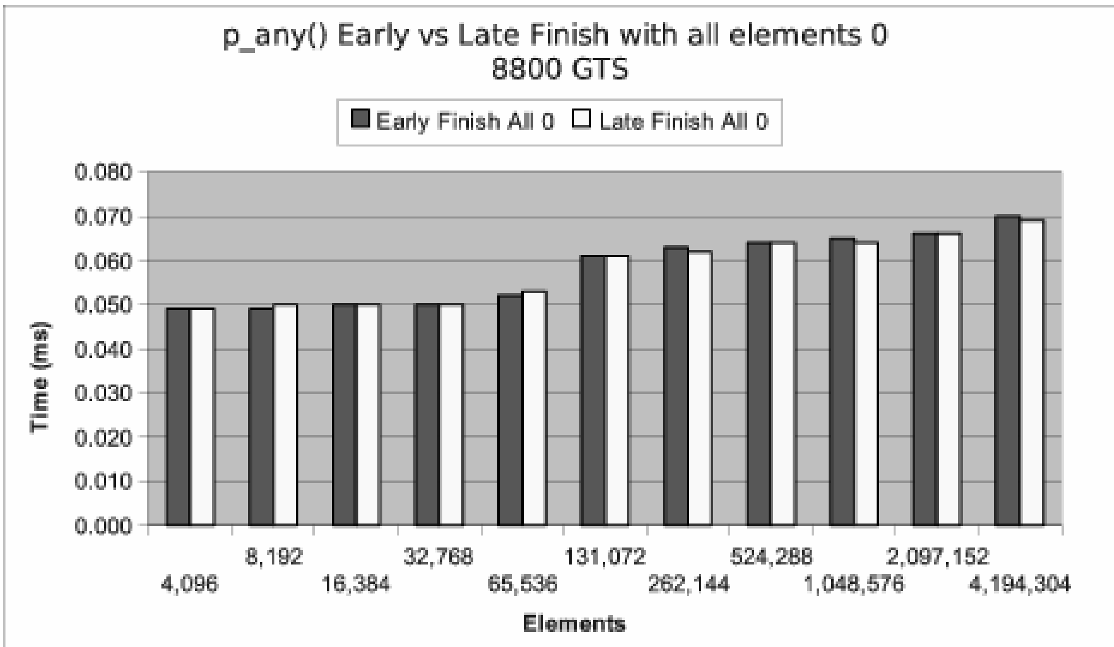

Figure 11: p_any() Early vs Late finish with inputs 0 - 8800GTS

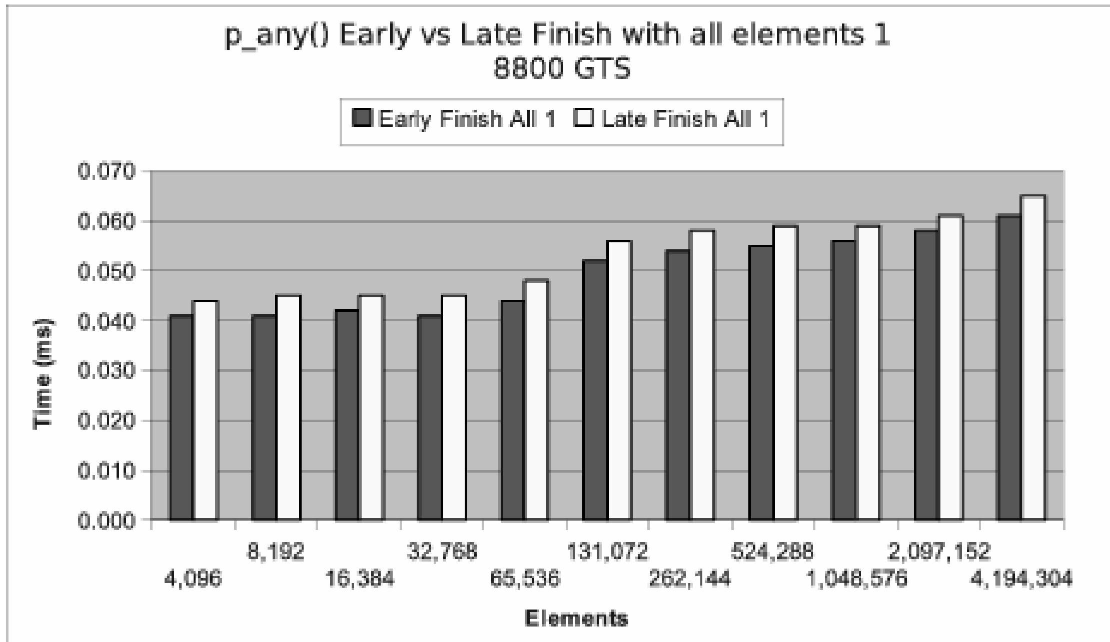

Figure 12: p_any() Early vs Late finish with inputs 1 - 8800GTS

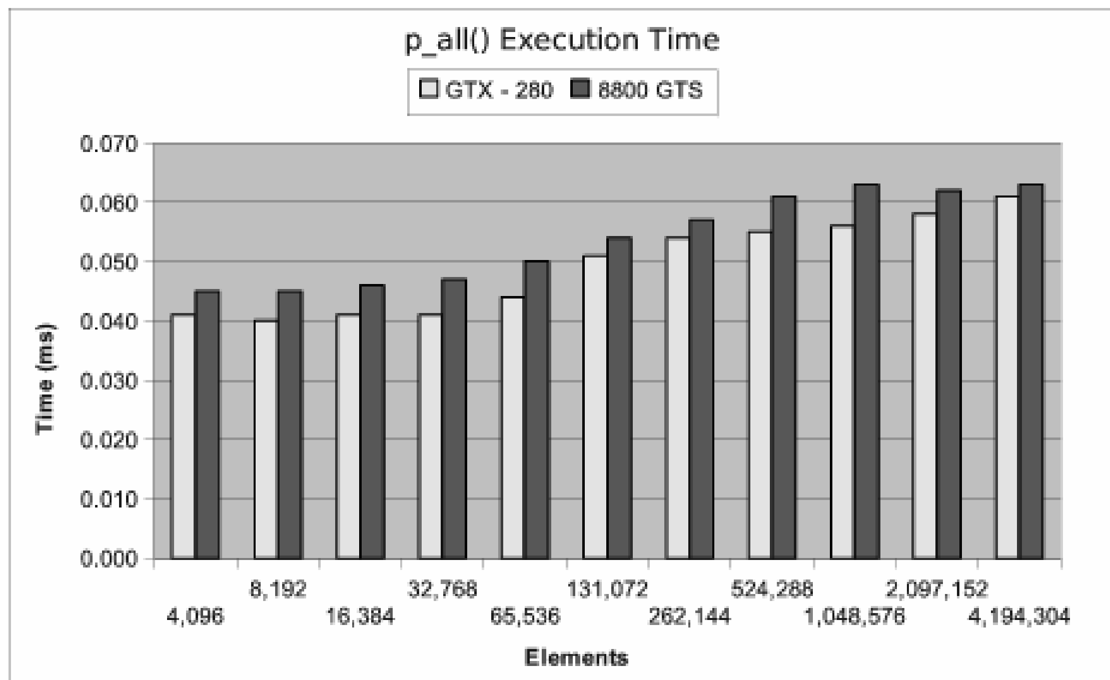

Figure 13: p_all() Execution Time

Figure 4.2 presents the results of running `p_all()` in the test machines. The results are almost identical to those findings in 4.2. It is interesting to see how for 4M elements the execution time for the GTX-280 and the 8800 GTS are very close in the case of `p_all()` and exact in the case of

p_any().

Comparing the graphics 4.1, p_atom_any() and 4.2 p_any(), one can see the difference of using the atomic method of synchronization with the CTRR. The major difference between the performance of these two methods is that the execution time of p_atom_any() increases considerably with the number of elements, while this is not the case for p_any(). The big gap existing between execution times suggests that the system locks from the atomic nature of p_atom_any() definitely hurts the performance of running p_atom_any() over big input data streams. Another large difference between the two schemes is that atomic operations cannot be executed in systems with compute capability 1.0, like the 8800 GTS, whereas p_any() does not present any hardware restriction.

4.3 GBS kernels

The basic unit of the global block synchronization mechanism is the __syncblocks() kernel. The execution time performance reached when running the function in the test machines is presented in figure 4.3.

Unlike the rest of the benchmarks run in this thesis, the test for the __syncblocks() kernel took into account the variable number of blocks to synchronize rather than the number of elements to reduce. This was done due to the nature of the kernel and the objective wanted to reach with it which is to synchronize multiple blocks using global memory.

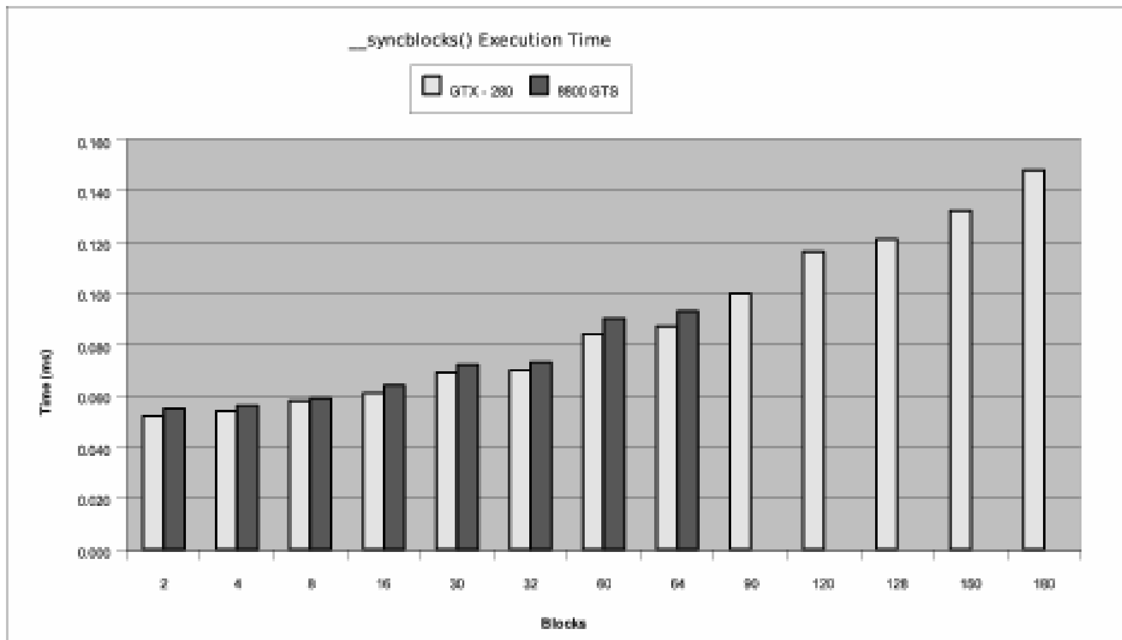

Figure 14: __syncblocks() Execution Time

By seeing the performance data, it was shown that the execution time increases almost linearly with the number of blocks to synchronize. This behavior was expected since the kernel runs a barrier synchronization counter within each block or SIMD engine, hence it is valid to assume that the more blocks to synchronize the more counters to evaluate, and more time computing the operation.

However, there was an interesting result found when running the test. For the 8800 GTS system the execution time raised up to almost 8 seconds when the number of blocks needed to synchronize was over 90. Although it is not completely clear why this happens, it is believed that the number of SIMD engines within the GPU affects the behavior of `__syncblocks()`, suggesting that for a tuned performance, a parametric optimum number of blocks must be set.

Figures 15, 16, 17, 18 and 19 present the performance results of the reduction operations Add, Or, And, Max and Min respectively. These algorithms used the GBS mechanism to find the reduced value. However, the reduction within each block was computed using the optimized reduction algorithm developed by NVIDIA [1]. A comparison between the pure NVIDIA `reduceAdd()` algorithm and the GBS `reduceAdd()` was also done and presented.

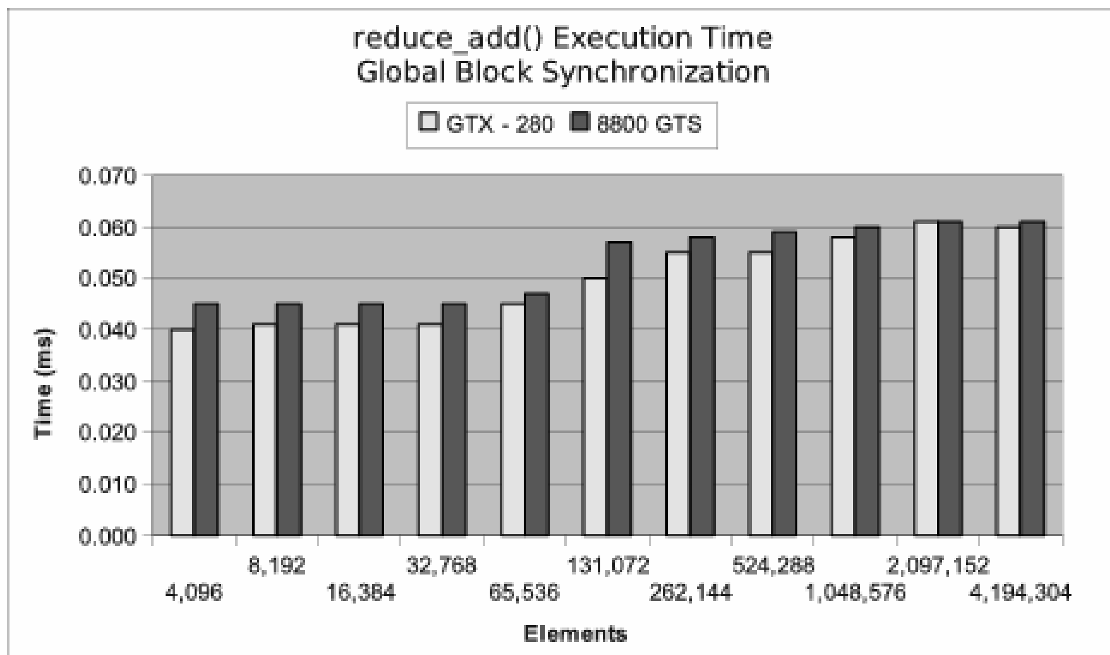

Figure 15: reduceAdd() using global block synchronization

Figure 15 corresponds to the execution time of running the `reduceAdd()` function on the test machines. Beside the optimizations explained by NVIDIA [1] in the process of finding the reduced value inside each block, the GBS algorithm presents an extraordinary performance almost matching the time values reached by the CTRR kernels.

The final solution is reached in two barriers. During the first barrier, each block computes its own result and writes it back to the designated cell memory in global memory. During the second barrier only the first block is in charge of gathering the partial results, computing the final result, and placing it in global memory.

It is interesting to see how for 2M elements and above, the timing results of running the benchmark in the low end 8800 GTS almost ties the execution time reached with the high end GTX-280.

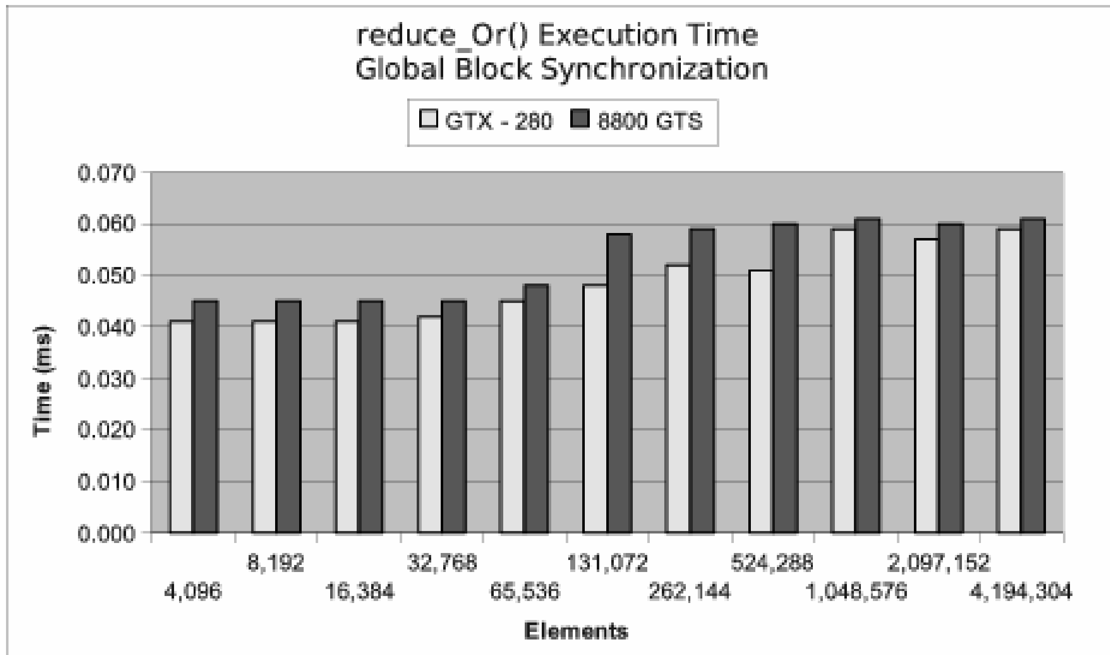

Figure 16: reduceOr() using global block synchronization

Figure 16 presents the performance results of running the `reduceOr()` benchmark on the test machines. The results are consistent with the findings in previous GBS algorithms presented here.

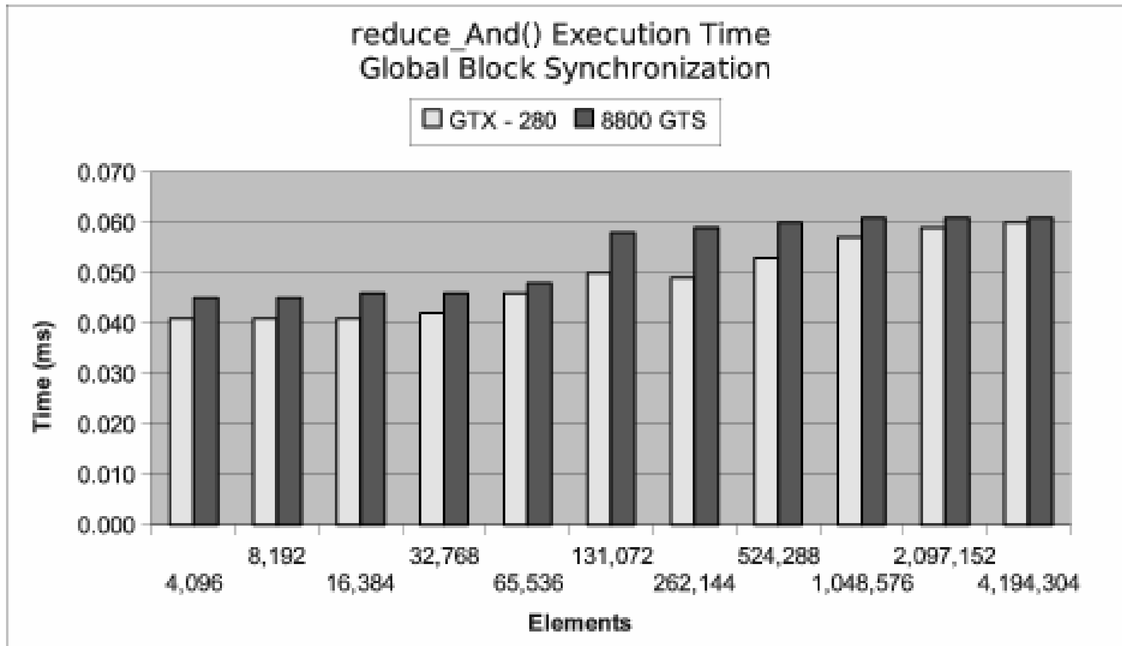

Figure 17: reduceAnd() using global block synchronization

Figure 17 shows the results of the execution time of running the `reduceAnd()` benchmark. Once again, since the techniques used by the algorithms are exactly the same, the results were expected to be similar.

Figure 18: reduceMax() using global block synchronization

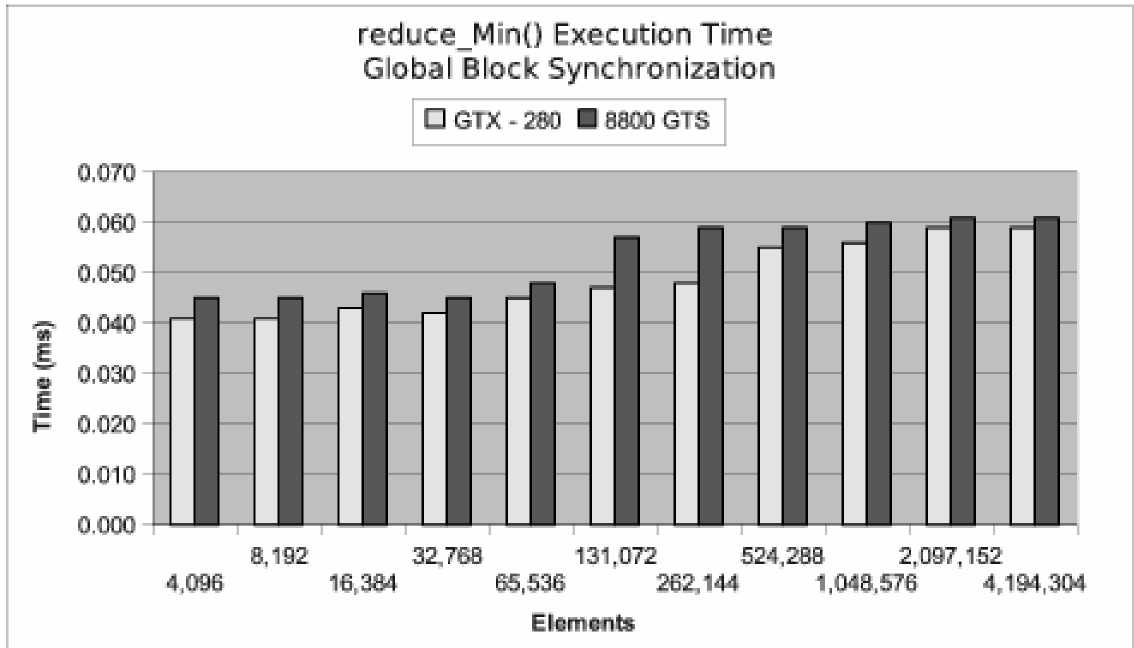

Figure 19: reduceMin() using global block synchronization

Figure 18 and figure 19 include the execution time of running the `reduceMax()` and `reduceMin()` algorithms respectively.

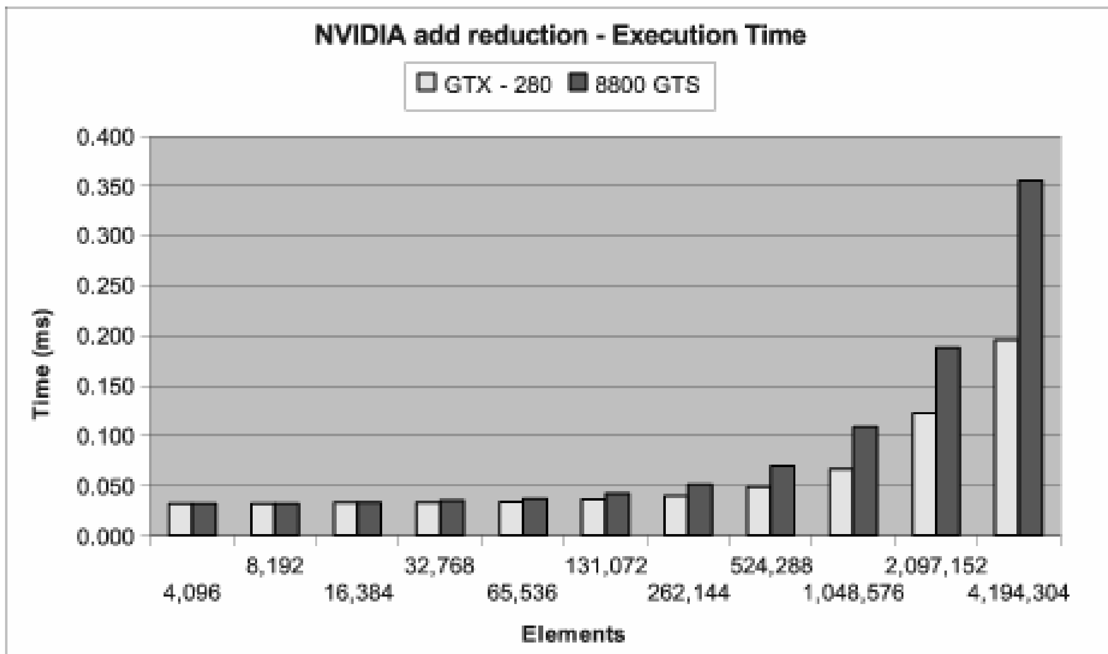

Figure 20: NVIDIA reductionAdd() Execution Time [1]

Figure 20 was included here as a reference value used to compare the effectiveness of the GBS

mechanism. The main difference between the two concepts is that while the NVIDIA algorithm reaches global synchronization by recursively calling and executing the kernel, GBS uses a barrier synchronization function `__syncblocks()` to coordinate blocks – thus avoiding kernel recursion calls.

Comparing the results of figure 15 of `reduceAdd()` using GBS with figure 20 of `reduceAdd()` using NVIDIA style, it is clearly seen that for larger input data streams GBS produces a speedup of 13%, 50% and 69% for input data streams of 1M, 2M and 4M respectively with the GTX-280. For smaller input data streams (4K to 512K), NVIDIA reduction style is about 28% faster. Note that these numbers are biased in favor of NVIDIA's methods, because the (usually very significant) cost of saving and restoring the state of GPU registers and shared memory is present with their algorithms, but not accounted here.

When testing the 8800 GTS system, GBS produces a speedup of 16%, 45%, 68% and 83% for input data streams of 512K, 1M, 2M and 4M respectively and NVIDIA style is approximately a 32% faster for 4K to 128K input data stream reductions.

This result confirms the fact that recursively calling the kernel hurts the overall performance of a computation and, because fewer input elements implies fewer kernel recursive calls, it is expected to reach better performance results for smaller input data streams.

Another observation made from this performance comparison is the confirmation of the trend followed by NVIDIA in terms of adjusting the architecture of their new systems to make kernel recursion less frequent in order to obtain better results. The old 8800 GTS system efficiently reduces up to 128K elements and the new GTX-280 efficiently reduces up to 512K elements.

Figures 21, 22, 4.3 and 24 present the performance results of testing the reduction operations Or, And, Max and Min combining the two methods, CTRR mechanism to find the partial reduction result within each block, and GBS to coordinate blocks in global memory to find the final result. The idea was to compare how different the execution time results between CTRR and NVIDIA styles are when reducing elements within a block of 128 threads.

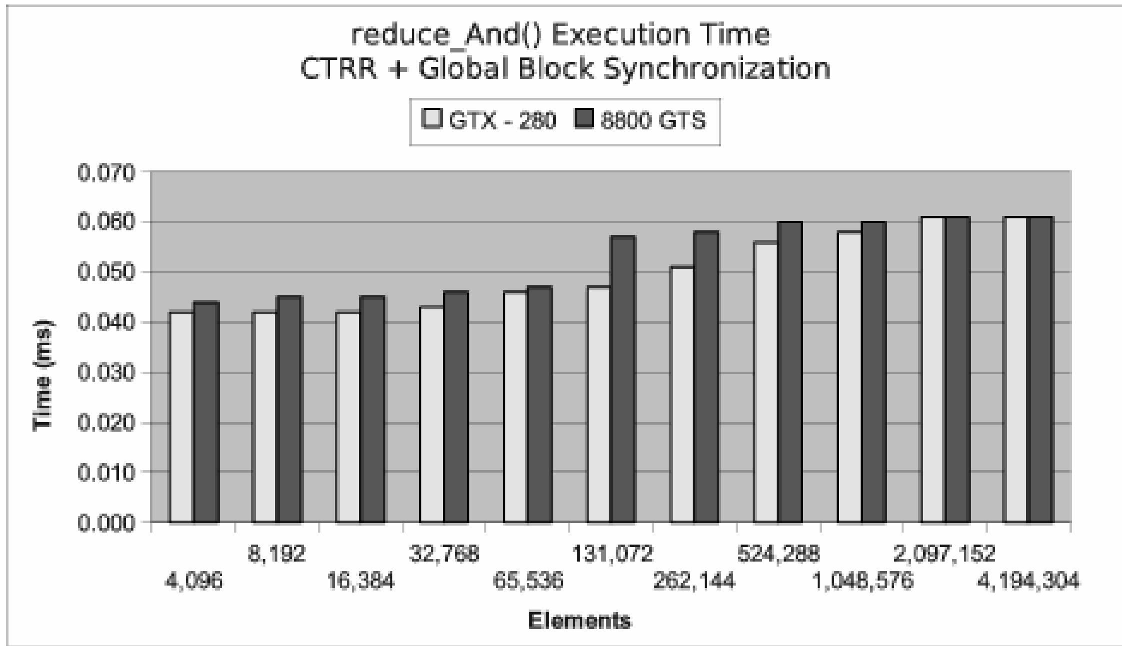

Figure 21: reduceAnd() using CTRR and global block synchronization

First, `reduceAnd()` is analyzed in figure 21. The performance information says that both methods of finding the partial reduction result within a block produce very similar results. However, using the NVIDIA style is an average of 2% faster than the CTRR style when using the GTX-280, but CTRR is an average of 1% faster when the test was executed in the 8800 GTS system.

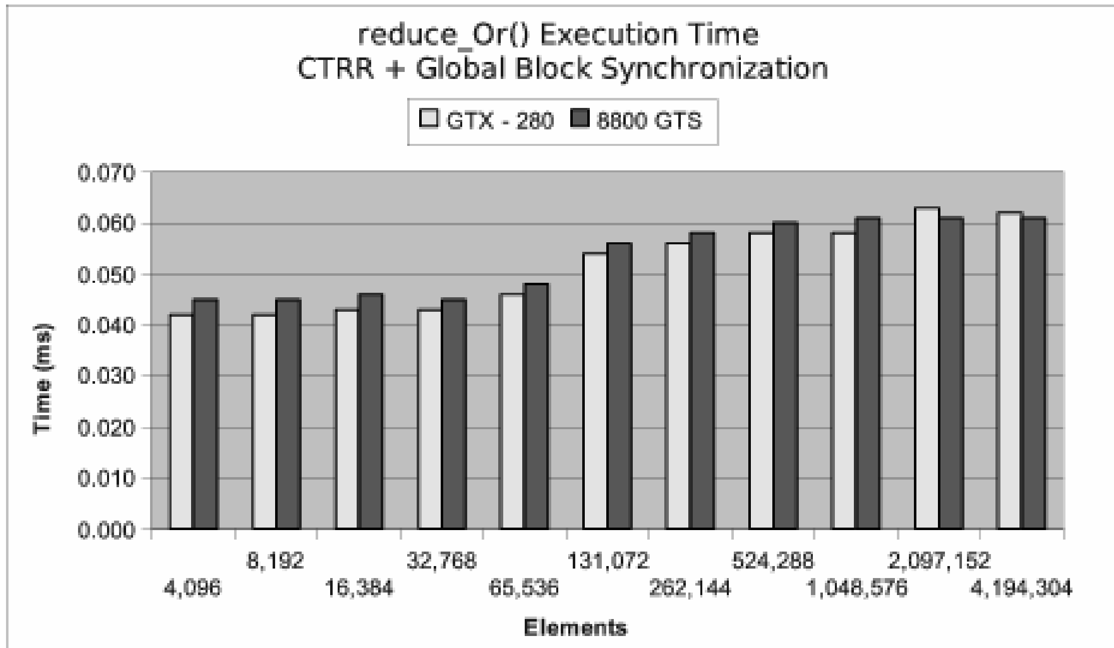

Figure 22: reduceOr() using CTRR and global block synchronization

Figure 22 shows the execution time for reduceOr(). In comparison with NVIDIA style, CTRR is an average 6% slower with GTX-280. For the 8800 GTS system, there is no average difference in execution time.

Figure 23: reduceMax() using CTRR and global block synchronization

The CTRR style is an average of 4% slower with GTX-280 when running the `reduceMax()` reduction and no average difference in execution time when running the benchmark in the 8800 GTS. Figure 4.3 presents the execution time results.

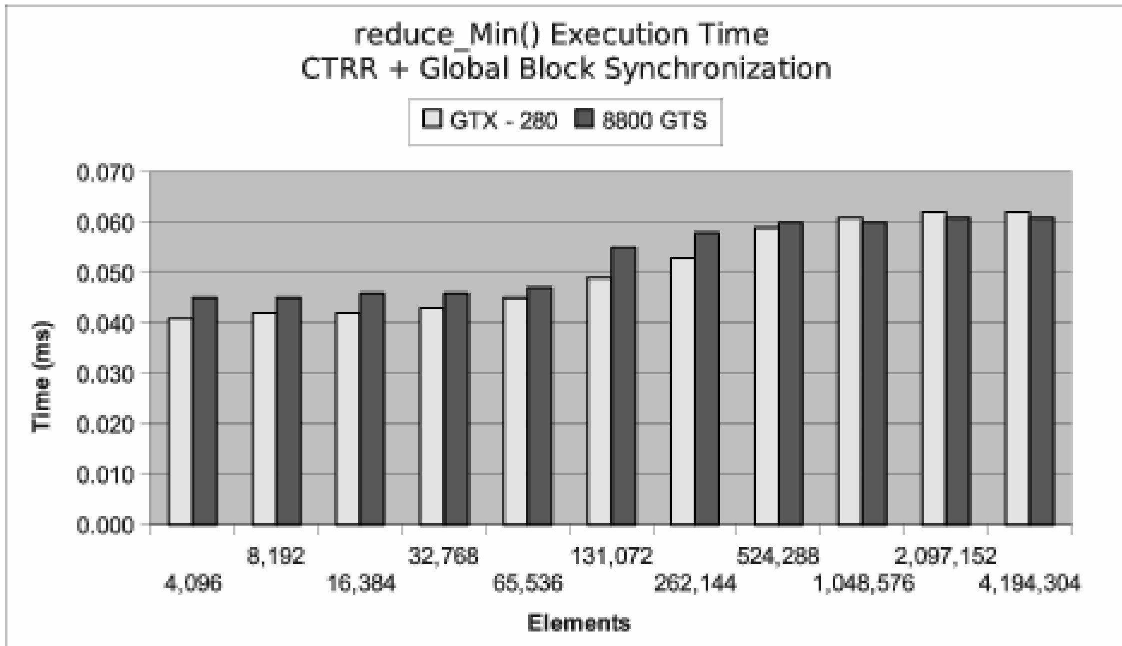

Figure 24: reduceMin() using CTRR and global block synchronization

Figure 24 shows the execution time results for the `reduceMin()` benchmark. Comparing the results with the NVIDIA style, CTRR style is also an average of 4% slower with GTX-280. Again, no differences were found when running the benchmark in the 8800 GTS.

4.4 Using GBS to implement some AFAPI functions

The next group of figures show the performance reached when testing some of the programs from the original AFAPI library using the `__syncblocks()` kernel instead of the original `p_wait()` function used back then. The new GBS mechanism seems to work efficiently with both GTX-280 and 8800 GTS. Figures 25, 4.4, 27, 28 and 29 presents the execution time of the functions `p_bcast()`, `p_count()`, `p_first()`, `p_quantify()` and `p_vote()`.

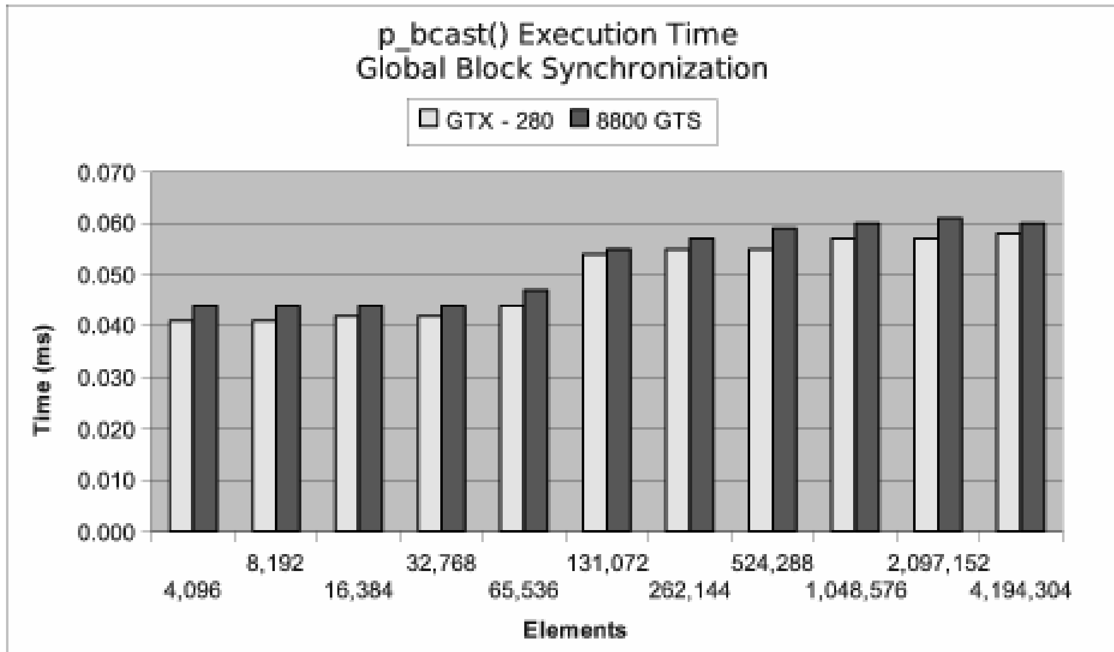

Figure 25: p_bcast() Execution Time

The results found when broadcasting data using the GBS concept to reach barrier synchronization between different blocks using global memory are shown in figure 25. As expected, the timing information of `p_bcast()` is highly affected by the behavior of __syncblocks() and it is consistent with the shape of previous GBS graphics.

Figure 4.4 to figure 29 demonstrate very similar behavior for the `p_bcast()` function described above.

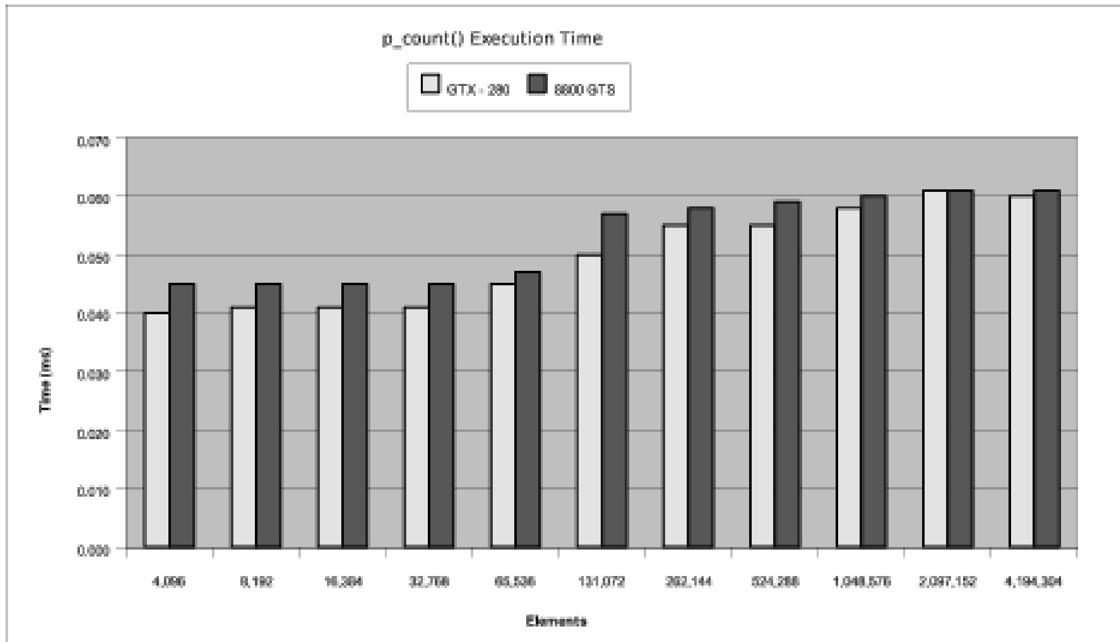

Figure 26: p_count() Execution Time

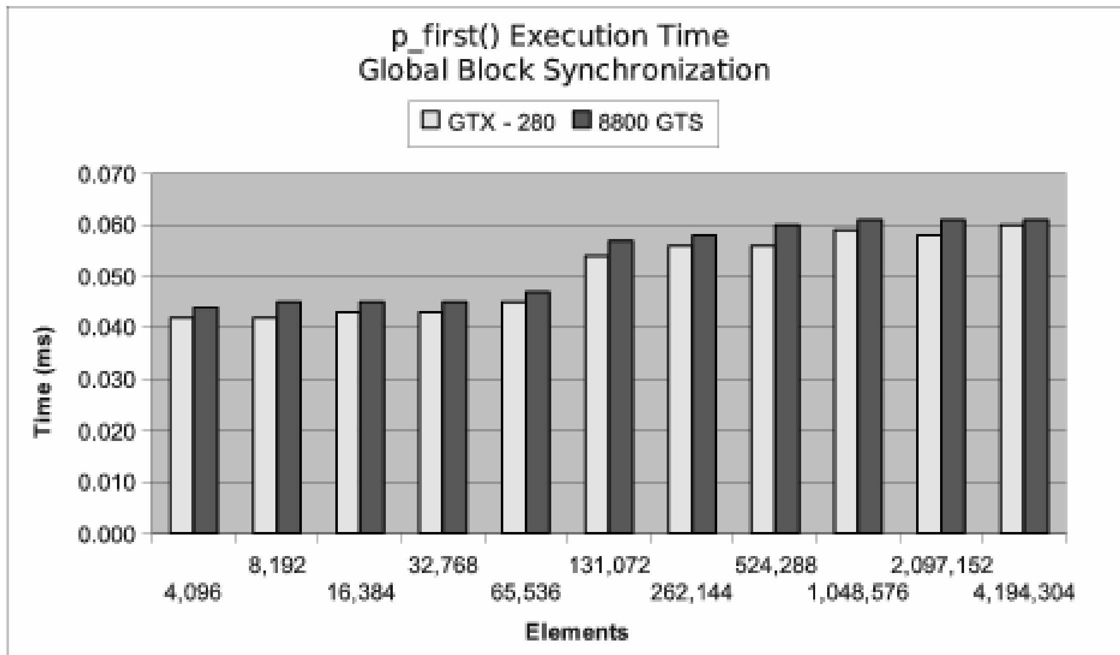

Figure 27: p_first() Execution Time

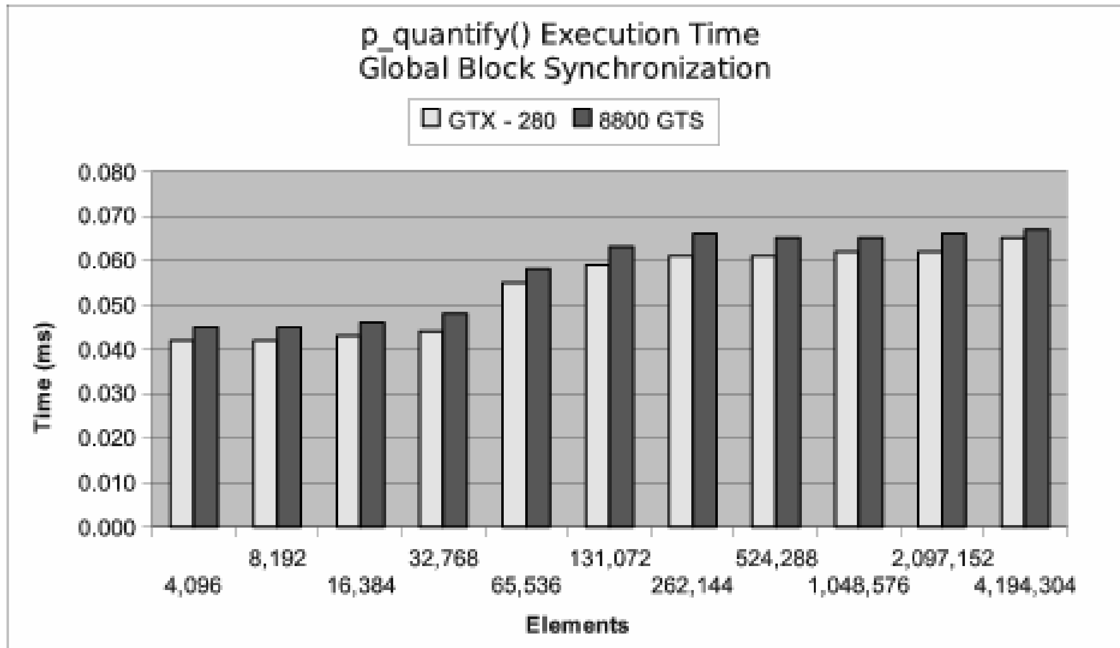

Figure 28: p_quantify() Execution Time

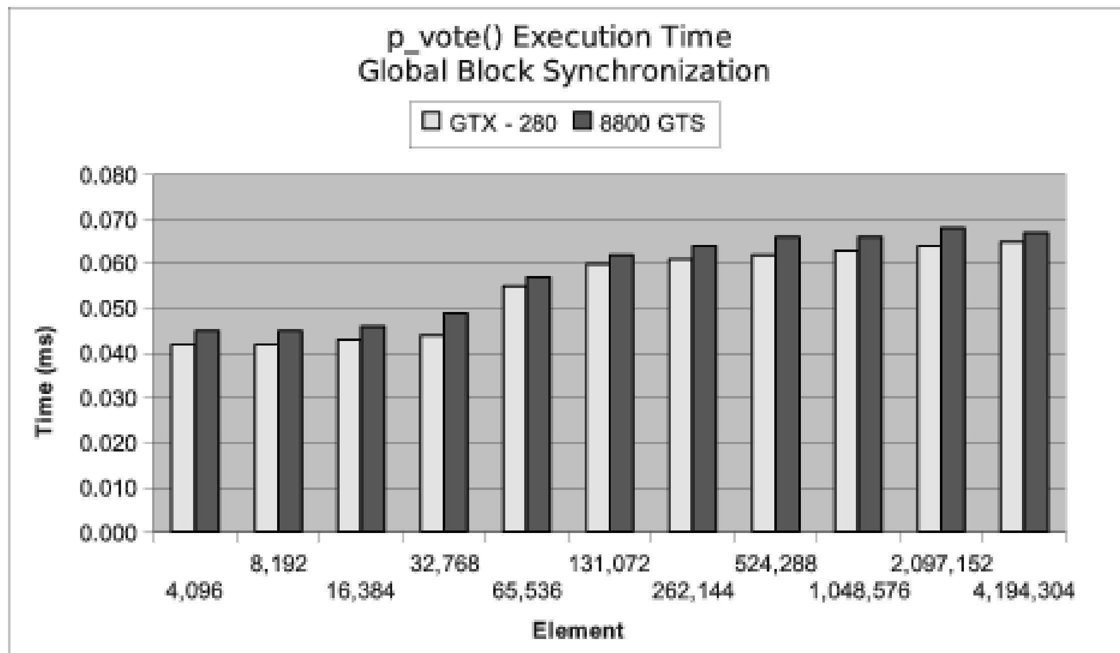

Figure 29: p_vote() Execution Time

5 Conclusion and future work

Based on the architecture model of the GPUs presented in Chapter 2 and the performance results in terms of execution time found in Chapter 4 we conclude that even though new GPU systems are no longer designed as a pure Single Instruction Multiple Data (SIMD) model, but as multiple SIMD multiprocessors, it is still possible to reach synchronization between threads in different SIMD cores in an efficient way using techniques like CTRR or barrier synchronization through the global block synchronization mechanism.

Using techniques such as CTRR and GBS offer a new approach for performing reduction operations and native functions within GPUs without the overhead produced by recursively calling and executing the kernel – the strategy suggested by NVIDIA in the CUDA programming guide [2]. Although the same functions could be implemented using the built in atomic functions of compute capability 1.3, CTRR and GBS offer better execution times. CTRR and GBS are mechanisms available to every NVIDIA GPU system independently of their compute capability or architecture.

Despite the compute capability difference between the test systems GTX-280 and 8800 GTS with compute capability 1.3 and 1.0 respectively, it was shown that the difference of execution time for the constant time race resolution algorithms between them is only 2us. The use of CTRR functions within low end GPUs makes them seen almost as powerful as a high end system, raising their price/performance ratio. Perhaps the extra circuit complexity invested in adding atomic instructions could be better used in other ways?

The constant time race resolution concept was determined to be useful when implementing algorithms that follow the race condition or those where randomly picking a winner thread provide an acceptable solution. Additional improvements to the CTRR algorithms were done for such functions that commonly present an early finish condition, like `p_any()` or `p_all()`. On the other hand, the extra branch instruction necessary to verify the early finish should generate a loss of performance for functions that imply arithmetic operations or those where the early finish condition is rarely met – although this performance loss was found to be generally negligible.

The functions where CTRR was tested and proved to work well were global OR, global AND, and Select One. Some other algorithms for which using CTRR is a possibility are Max, and Min. An extra thread control function must be included in those kernels to guarantee that every thread was analyzed; however, other types of algorithms like summations or scans are not easy to implement using CTRR.

Global block synchronizationwas was reached by using a kernel called `__syncblocks()`. It was

implemented and tested, the execution time analysis suggests that reaching global synchronization through that way avoids the use of kernel recursion, while reaching similar results in terms of coordination. Although it is still necessary to find an optimum parameter for the maximum number of blocks to synchronize, this thesis demonstrated that it is possible to use GPUs to effectively implement some of the algorithms included in the original AFAPI library [17] and according to analysis of the performance graphs by using the GBS kernel the difference of execution time between GTX-280 and 8800 GTS is unnoticeable.

Although this thesis introduced the concepts of global synchronization and reduction algorithms within NVIDIA GPUs and probed whether it is possible to include those concepts in CUDA kernels, there are still some improvements and future work needed.

It is necessary to implement the synchronization mechanisms in other types of GPUs, specifically in ATI models.

It is also necessary to implement the codes of the algorithms using the OpenCL language. OpenCL is an architecture independent high level language that can be used by various vendors. In April 2009, NVIDIA released the OpenCL driver and SDK [18].

References

[1] M. Harris, “Optimizing parallel reduction in cuda,” tech. rep., NVIDIA, http://www.nvidia.com/cuda, Nov 2008.

[2] N. Corporation, “Nvidia cuda programming guide,” www.nvidia.com/cuda, 2008.

[3] A. Corporation, “Ati stream sdk user guide,” http://developer.amd.com/gpu/ATIStreamSDK, 2008.

[4] Q. Hou, K. Zhou, and B. Guo, “Bsgp: bulk-synchronous gpu programming,” in *SIGGRAPH ’08: ACM SIGGRAPH 2008 papers*, (New York, NY, USA), pp. 1–12, ACM, 2008.

[5] D. Y. Henry Dietz, Diego Rivera, “A maze of twisty little passages,” tech. rep., University of Kentucky, http://aggregate.org/WHITE/sc08mog.pdf, Nov 2008.

[6] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃŒger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[7] P. H. Ha, P. Tsigas, and O. J. Anshus, “Wait-free programming for general purpose computations on graphics processors,” in *PODC ’08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, (New York, NY, USA), pp. 452–452, ACM, 2008.

[8] P. H. Ha, P. Tsigas, and O. J. Anshus, “The synchronization power of coalesced memory accesses,” in *DISC ’08: Proceedings of the 22nd international symposium on Distributed Computing*, (Berlin, Heidelberg), pp. 320–334, Springer-Verlag, 2008.

[9] N. Corporation, “Nvidia opencl jumpstart guide,” www.nvidia.com/cuda, April 2009.

[10] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, “Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors,” in *ICPP ’94: Proceedings of the 1994 International Conference on Parallel Processing*, (Washington, DC, USA), pp. 93–96, IEEE Computer Society, 1994.

[11] S. Mitta, “Kentucky’s adapter for parallel execution and rapid synchronization,” University of Kentucky, College of Engineering - http://hdl.handle.net/10225/621, 2007.

[12] A. Silberschatz and P. Galvin, *Operating System Concepts, 4th edition.* Addison-Wesley, 1994.

[13] G. J. Lipovski and P. Vaughan, "A fetch-and-op implementation for parallel computers," in *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, (Los Alamitos, CA, USA), pp. 384–392, IEEE Computer Society Press, 1988.

[14] H. Dietz, "Using the ppl maspar mp-1," tech. rep., Parallel Processing Laboratory, School of Electrical Engineering - Purdue University, West Lafayette, IN, Feb 1995.

[15] H. Dietz and W. Cohen, "A massively parallel mimd implemented by simd hardware?," tech. rep., Parallel Processing Laboratory, School of Electrical Engineering - Purdue University, West Lafayette, IN, Jan 1992.

[16] H. Dietz, "Afapi, aggregate function application program interface." `http://aggregate.org/AFAPI`.

[17] H. G. Dietz, T. Mattox, and G. Krishnamurthy, "The aggregate function api: It's not just for papers anymore," in *LCPC '97: Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, (London, UK), pp. 277–291, Springer-Verlag, 1998.

[18] NVIDIA, "Cuda zone." `http://www.nvidia.com/object/cuda_get.html`.

APPENDIX

The Appendix presents the code for the different functions implemented in the CUDA kernels used in this thesis. It first shows the atomic and constant time race resolution native kernels, then it shows the code for the block synchronization kernel and the reduction operations implemented with it. Finally, the CUDA kernel functions of a subset of the functions included in the AFAPI library are also shown.

Most of the functions follow the same code structure. A definition of the function using the `__global__` reserved word which is a special requirement of the CUDA language [2], the declaration of the type and name of the input parameters of the kernel. Then the declaration of the local variables and the declaration of the space in shared memory with the `extern __shared__ int sdata[]` line. The size of `sdata[]` is determined at launch time with the parameter specifying the amount of bytes that will be dynamically allocated per block [2].

Now, a brief description of each function is presented along with the input parameters.

p_atom_any()

This is the version of `p_any()` using the `atomicOr()` built in function of the NVIDIA GTX-280. It is intended to determine if any of the elements of the input stream data has a value of 1.

```
__global__ void p_atom_any(int *g_idata, int *g_odata, int n) {
extern __shared__ int sdata[];
int i = blockIdx.x*blockDim.x*2 + threadIdx.x;
int gridsize = blockDim.x*2*gridDim.x;
int tid = threadIdx.x;
if (g_odata[blockIdx.x]==0){
        while(i<n && sdata[0]==0){
                sdata[tid]|= g_idata[i] | g_idata[i + blockDim.x];
                atomicOr(&sdata[0],sdata[tid]);
                i+=gridsize;
        }
        g_odata[blockIdx.x]=sdata[0];
        }
__syncthreads();
}
```

p_warp_any()

This is the version of `p_any()` using the `__any()` built in vote function of the NVIDIA GTX-280. It is intended to determine warp by warp if any of the elements of the input stream data has a value of 1.

```
__global__ void p_warp_any(int *g_idata, int *g_odata, int n) {
extern __shared__ int sdata[];
int i = blockIdx.x*blockDim.x*2 + threadIdx.x;
int gridsize = blockDim.x*2*gridDim.x;
int tid = threadIdx.x;
if (g_odata[blockIdx.x]==0){
        while(i<n && sdata[0]==0){
                sdata[tid] |= g_idata[i] | g_idata[i + blockDim.x];
                sdata[0] = (__any(1) != 0);
                i+=gridsize;
                }
        g_odata[blockIdx.x]=sdata[0];
        }
__syncthreads();
}
```

p_selectOne()

This function returns the thread ID of the first thread to write in global memory. This function describes clearly the concept of constant time race resolution.

```
__global__ void p_selectOne(volatile int *g_odata) {
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        __syncthreads();
        g_odata[0] = i;
}
```

p_any()

This function determines if any of the elements of the input stream data has a value of 1, although this could be easily modified to evaluate any condition. The final version of the kernel includes the early finish validation.

```
__global__ void p_any(int *g_idata, int *g_odata, int n) {
        extern __shared__ int sdata[];
        int tid = threadIdx.x;
        int i = blockIdx.x*blockDim.x*2 + threadIdx.x;
        int gridsize = blockDim.x*2*gridDim.x;
        sdata[tid] = 0;
        __syncthreads();

        if (g_odata[blockIdx.x]==0){
                while(sdata[0]==0 && i < n) {
                        sdata[tid] |= g_idata[i] | g_idata[i + blockIdx.x];
                        if (sdata[tid]==1) sdata[0] = 1;
                        i += gridsize;
                        }
        g_odata[blockIdx.x]= (sdata[0]==1);
        }
__syncthreads();
}
```

p_all()

This function determines if all of the elements of the input stream data has a value of 1, although this could be easily modified to evaluate any condition. The final version of the kernel includes the early finish validation.

```
__global__ void p_all(int *g_idata, int *g_odata_in, unsigned int n) {
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n){
        sdata[tid] &= g_idata[i] & g_idata[i+blockSize];
        i += gridSize;
        if (sdata[tid] == 0) sdata[0] = 0;
        }
__syncthreads();
if (threadIdx.x == 0) g_odata_in[blockIdx.x] = sdata[0];
}
```

__syncblocks()

This function implements the barrier synchronization mechanism used to coordinate different blocks. Variables barno and hisbarno represent the barrier counter of current block and next block respectively.

```
__global__ void __syncblocks(register volatile unsigned int *barvec) {
__syncthreads();
if (threadIdx.x == 0) {
        register int i = ((blockIdx.x + 1) % gridDim.x);
        register int barno = barvec[blockIdx.x] + 1;
        register int hisbarno;
        barvec[blockIdx.x] = barno;
        do {
                do {
                        hisbarno = barvec[i];
                        } while (hisbarno < barno);
                if (++i >= gridDim.x) i = 0;
                } while ((hisbarno == barno) && (i != blockIdx.x));
        barvec[blockIdx.x] = barno + 1;
        }
}
```

reduceAdd_GBS()

Reduction summation using global block synchronization. The reduction within each block follows the optimized reduction algorithm presented by NVIDIA [1].

```
__global__ void reduceAdd_GBS(int *g_idata, int *g_odata_in,
                                unsigned int n)
{
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
register volatile int *g_odata = g_odata_in;
register volatile int *barvec = (g_odata + gridDim.x);
sdata[tid] = 0;

while (i < n){
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;
        }
__syncthreads();
```

```
if (blockSize >= 512){
        if (tid < 256) {
                sdata[tid] += sdata[tid + 256];
        }
__syncthreads();
}
if (blockSize >= 256){
        if (tid < 128) {
                sdata[tid] += sdata[tid + 128];
        }
__syncthreads();
}
if (blockSize >= 128) {
        if (tid <  64) {
                sdata[tid] += sdata[tid +  64];
        }
__syncthreads();
}
#ifndef __DEVICE_EMULATION__
if (tid < 32)
#endif
{
        if (blockSize >=  64) {
                sdata[tid] += sdata[tid + 32];
                EMUSYNC;
        }
        if (blockSize >=  32) {
                sdata[tid] += sdata[tid + 16];
                EMUSYNC;
        }
        if (blockSize >=  16) {
                sdata[tid] += sdata[tid +  8];
                EMUSYNC;
        }
        if (blockSize >=   8) {
                sdata[tid] += sdata[tid +  4];
                EMUSYNC;
        }
        if (blockSize >=   4) {
                sdata[tid] += sdata[tid +  2];
                EMUSYNC;
        }
        if (blockSize >=   2) {
                sdata[tid] += sdata[tid +  1];
                EMUSYNC;
        }
}
```

```
if (threadIdx.x == 0) g_odata[blockIdx.x] = sdata[0];
__syncblocks((volatile unsigned int *) barvec);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) {
                sdata[tid] = g_odata[tid];
        }
        __syncthreads();
        if (gridDim.x >= 128) {
                if (tid <  64) {
                        sdata[tid] += sdata[tid +  64];
                }
                __syncthreads();
        }
#ifndef __DEVICE_EMULATION__
        if (tid < 32)
#endif
        {
                if (gridDim.x >=  64) {
                        sdata[tid] += sdata[tid + 32];
                        EMUSYNC;
                }
                if (gridDim.x >=  32) {
                        sdata[tid] += sdata[tid + 16];
                        EMUSYNC;
                }
                if (gridDim.x >=  16) {
                        sdata[tid] += sdata[tid +  8];
                        EMUSYNC;
                }
                if (gridDim.x >=   8) {
                        sdata[tid] += sdata[tid +  4];
                        EMUSYNC;
                }
                if (gridDim.x >=   4) {
                        sdata[tid] += sdata[tid +  2];
                        EMUSYNC;
                }
                if (gridDim.x >=   2) {
                        sdata[tid] += sdata[tid +  1];
                        EMUSYNC;
                }
        }
        if (threadIdx.x == 0) g_odata[0] = sdata[0];
    }
}
#endif // #ifndef _REDUCE_KERNEL_H_
```

reduceOr_GBS()

Reduction OR operation over 32-bit words using global block synchronization. The reduction within each block follows the optimized reduction algorithm presented by NVIDIA [1].

```
__global__ void reduceOr_GBS
(int *g_idata, int *g_odata_in, unsigned int n) {
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
register volatile int *g_odata = g_odata_in;
register volatile int *barvec = (g_odata + gridDim.x);
sdata[tid] = 0;
while (i < n) {
        sdata[tid] |= g_idata[i] | g_idata[i+blockSize];
        i += gridSize;
}
__syncthreads();
if (blockSize >= 512) {
        if (tid < 256) {
                sdata[tid] |= sdata[tid + 256];
        }
        __syncthreads();
}
if (blockSize >= 256) {
        if (tid < 128) {
                sdata[tid] |= sdata[tid + 128];
        }
        __syncthreads();
}
if (blockSize >= 128) {
        if (tid <  64) {
                sdata[tid] |= sdata[tid +  64];
        }
        __syncthreads();
}
#ifndef __DEVICE_EMULATION__
        if (tid < 32)
#endif
{
        if (blockSize >=  64) {
                sdata[tid] |= sdata[tid + 32];
                EMUSYNC;
        }
        if (blockSize >=  32) {
                sdata[tid] |= sdata[tid + 16];
                EMUSYNC;
        }
```

```
        if (blockSize >=  16) {
                sdata[tid] |= sdata[tid +  8];
                EMUSYNC;
        }
        if (blockSize >=   8) {
                sdata[tid] |= sdata[tid +  4];
                EMUSYNC;
        }
        if (blockSize >=   4) {
                sdata[tid] |= sdata[tid +  2];
                EMUSYNC;
        }
        if (blockSize >=   2) {
                sdata[tid] |= sdata[tid +  1];
                EMUSYNC;
        }
}
if (threadIdx.x == 0) g_odata[blockIdx.x] = sdata[0];
__syncblocks((volatile unsigned int *) barvec);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) {
                sdata[tid] = g_odata[tid];
        }
        __syncthreads();
        if (gridDim.x >= 128) {
                if (tid <  64) {
                        sdata[tid] |= sdata[tid +  64];
                }
        __syncthreads();
        }
#ifndef __DEVICE_EMULATION__
        if (tid < 32)
#endif
        {
                if (gridDim.x >=  64) {
                        sdata[tid] |= sdata[tid + 32];
                        EMUSYNC;
                }
                if (gridDim.x >=  32) {
                        sdata[tid] |= sdata[tid + 16];
                        EMUSYNC;
                }
                if (gridDim.x >=  16) {
                        sdata[tid] |= sdata[tid +  8];
                        EMUSYNC;
                }
                if (gridDim.x >=   8) {
                        sdata[tid] |= sdata[tid +  4];
                        EMUSYNC;
                }
                if (gridDim.x >=   4) {
                        sdata[tid] |= sdata[tid +  2];
                        EMUSYNC;
                }
```

```
                if (gridDim.x >=   2) {
                        sdata[tid] |= sdata[tid +  1];
                        EMUSYNC;
                }
        }
        if (threadIdx.x == 0) g_odata[0] = sdata[0];
}
}
#endif // #ifndef _REDUCE_KERNEL_H_
```

reduceAnd_GBS()

Reduction AND operation over 32-bit words using global block synchronization. The reduction within each block follows the optimized reduction algorithm presented by NVIDIA [1].

```
reduceAnd_GBS(int *g_idata, int *g_odata_in, unsigned int n) {
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
register volatile int *g_odata = g_odata_in;
register volatile int *barvec = (g_odata + gridDim.x);
sdata[tid] = 0;
while (i < n) {
        sdata[tid] &= g_idata[i] & g_idata[i+blockSize];
        i += gridSize;
}
__syncthreads();
if (blockSize >= 512) {
        if (tid < 256) {
                sdata[tid] &= sdata[tid + 256];
        }
        __syncthreads();
}
if (blockSize >= 256) {
        if (tid < 128) {
                sdata[tid] &= sdata[tid + 128];
        }
        __syncthreads();
}
if (blockSize >= 128) {
        if (tid <  64) {
                sdata[tid] &= sdata[tid +  64];
        }
        __syncthreads();
}
```

```
#ifndef __DEVICE_EMULATION__
if (tid < 32)
#endif
{
        if (blockSize >=  64) {
                sdata[tid] &= sdata[tid + 32];
                EMUSYNC;
        }
        if (blockSize >=  32) {
                sdata[tid] &= sdata[tid + 16];
                EMUSYNC;
        }
        if (blockSize >=  16) {
                sdata[tid] &= sdata[tid +  8];
                EMUSYNC;
        }
        if (blockSize >=   8) {
                sdata[tid] &= sdata[tid +  4];
                EMUSYNC;
        }
        if (blockSize >=   4) {
                sdata[tid] &= sdata[tid +  2];
                EMUSYNC;
        }
        if (blockSize >=   2) {
                sdata[tid] &= sdata[tid +  1];
                EMUSYNC;
        }
}
if (threadIdx.x == 0) g_odata[blockIdx.x] = sdata[0];
__syncblocks((volatile unsigned int *) barvec);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) {
                sdata[tid] = g_odata[tid];
        }
        __syncthreads();
        if (gridDim.x >= 128) {
                if (tid <  64) {
                        sdata[tid] &= sdata[tid +  64];
                }
                __syncthreads();
        }
#ifndef __DEVICE_EMULATION__
if (tid < 32)
#endif
{
        if (gridDim.x >=  64) {
                sdata[tid] &= sdata[tid + 32];
                EMUSYNC;
        }
        if (gridDim.x >=  32) {
                sdata[tid] &= sdata[tid + 16];
                EMUSYNC;
        }
```

```
        if (gridDim.x >=  16) {
                sdata[tid] &= sdata[tid +  8];
                EMUSYNC;
        }
        if (gridDim.x >=   8) {
                sdata[tid] &= sdata[tid +  4];
                EMUSYNC;
        }
        if (gridDim.x >=   4) {
                sdata[tid] &= sdata[tid +  2];
                EMUSYNC;
        }
        if (gridDim.x >=   2) {
                sdata[tid] &= sdata[tid +  1];
                EMUSYNC;
        }
        }
        if (threadIdx.x == 0) g_odata[0] = sdata[0];
}
}
#endif // #ifndef _REDUCE_KERNEL_H_
```

reduceMax_GBS()

Reduction MAX operation over 32-bit words using global block synchronization. The reduction within each block follows the optimized reduction algorithm presented by NVIDIA [1].

```
__global__ void reduceMax_GBS
(int *g_idata, int *g_odata_in, unsigned int n)
{
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
register volatile int *g_odata = g_odata_in;
register volatile int *barvec = (g_odata + gridDim.x);
sdata[tid] = 0;
int temp = 0;

while (i < n) {
        temp = sdata[tid];
        sdata[tid] = (g_idata[i] > g_idata[i + blockSize]) ?
                              g_idata[i] : g_idata[i + blockSize];
        sdata[tid] = (sdata[tid] > temp) ? sdata[tid] : temp;
        temp = 0;
        i += gridSize;
}
```

```
        __syncthreads();
if(blockSize >= 512) {
        if(tid < 256) {
                sdata[tid] = (sdata[tid] > sdata[tid + 256])  ?
                                        sdata[tid] : sdata[tid+256];
        }
        __syncthreads();
}
if(blockSize >= 256) {
        if(tid < 128) {
                sdata[tid] = (sdata[tid] > sdata[tid + 128]) ?
                                        sdata[tid] : sdata[tid+128];
        }
        __syncthreads();
}
if(blockSize >= 128) {
        if(tid < 64) {
                sdata[tid] = (sdata[tid] > sdata[tid + 64 ]) ?
                                        sdata[tid] : sdata[tid+64];
        }
        __syncthreads();
}
#ifndef __DEVICE_EMULATION__
if (tid < 32)
#endif
{
        if (blockSize >= 64) {
                sdata[tid] = (sdata[tid] > sdata[tid + 32]) ?
                                sdata[tid] : sdata[tid+32];
                EMUSYNC;
        }
        if (blockSize >= 32) {
                sdata[tid] = (sdata[tid] > sdata[tid + 16]) ?
                                sdata[tid] : sdata[tid+16];
                EMUSYNC;
        }
        if (blockSize >=  16) {
                sdata[tid] = (sdata[tid] > sdata[tid + 8]) ?
                                sdata[tid] : sdata[tid+8];
                EMUSYNC;
        }
        if (blockSize >= 8) {
                sdata[tid] = (sdata[tid] > sdata[tid + 4]) ?
                                sdata[tid] : sdata[tid+4];
                EMUSYNC;
        }
        if (blockSize >= 4) {
                sdata[tid] = (sdata[tid] > sdata[tid + 2]) ?
                                sdata[tid] : sdata[tid+2];
                EMUSYNC;
        }
```

```
        if (blockSize >= 2) {
                sdata[tid] = (sdata[tid] > sdata[tid + 1]) ?
                                sdata[tid] : sdata[tid+1];
                EMUSYNC;
        }
}
if (threadIdx.x == 0) g_odata[blockIdx.x] = sdata[0];
__syncblocks((volatile unsigned int *) barvec);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) sdata[tid] = g_odata[tid];
        __syncthreads();
        if(gridDim.x >= 128){
                if(tid < 64){
                        sdata[tid] = (sdata[tid] > sdata[tid+64]) ?
                                sdata[tid] : sdata[tid+64];
                }
        __syncthreads();
}
#ifndef __DEVICE_EMULATION__
if (tid < 32)
#endif
{
        if (gridDim.x >=  64) {
                sdata[tid] = (sdata[tid] > sdata[tid + 32]) ?
                                sdata[tid] : sdata[tid+32];
                EMUSYNC;
        }
        if (gridDim.x >= 32) {
                sdata[tid] = (sdata[tid] > sdata[tid + 16]) ?
                                sdata[tid] : sdata[tid+16];
                EMUSYNC;
        }
        if (gridDim.x >= 16) {
                sdata[tid] = (sdata[tid] > sdata[tid + 8]) ?
                                sdata[tid] : sdata[tid+8];
                EMUSYNC;
        }
        if (gridDim.x >= 8) {
                sdata[tid] = (sdata[tid] > sdata[tid + 4]) ?
                                sdata[tid] : sdata[tid+4];
                EMUSYNC;
        }
        if (gridDim.x >= 4) {
                sdata[tid] = (sdata[tid] > sdata[tid + 2]) ?
                                sdata[tid] : sdata[tid+2];
                EMUSYNC;
        }
        if (gridDim.x >= 2) {
                sdata[tid] = (sdata[tid] > sdata[tid + 1]) ?
                                sdata[tid] : sdata[tid+1];
                EMUSYNC;
        }
}
```

```
if (threadIdx.x == 0) g_odata[0] = sdata[0];
}
}
#endif // #ifndef _REDUCE_KERNEL_H_
```

reduceAnd_CTRR_GBS()

Reduction AND operation over 32-bit words using global block synchronization. The reduction within each block follows the constant time race resolution algorithm.

```
__global__ void reduce_And
(int *g_idata, volatile int *g_odata,
        volatile int *g_barrier, int n, int blockSize)
{
unsigned int tid = threadIdx.x;
unsigned int bid = blockIdx.x;
unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
unsigned int t = 0xFFFFFFFF;
unsigned int temp = 0xFFFFFFFF;
extern __shared__ int sdata[];
sdata[0] = temp;
while (i<n) {
        temp &= g_idata[i] & g_idata[i + blockSize];
        i += gridSize;
}
__syncthreads();
while ((~temp & t) > 0) {
        sdata[0] &= temp;
        __syncthreads();
        t = sdata[0] ^ temp;
}
if (tid == 0)  g_odata[blockIdx.x] = sdata[0];
__syncblocks(g_barrier);
if (bid == 0) {
        if (tid < gridDim.x) temp = g_odata[tid];
        __syncthreads();
        while ((~temp & t) > 0) {
                sdata[0] &= temp;
                __syncthreads();
                t = sdata[0] ^ temp;
        }
        if (tid == 0)  g_odata[0] = sdata[0];
        }
}
```

reduceOr_CTRR_GBS()

Reduction OR operation over 32-bit words using global block synchronization. The reduction within each block follows the constant time race resolution algorithm.

```
__global__ void reduceOr
(int *g_idata, volatile int *g_odata,
volatile int *g_barrier, int n, int blockSize)
{
unsigned int tid = threadIdx.x;
unsigned int bid = blockIdx.x;
unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
unsigned int t = 0xFFFFFFFF;
unsigned int temp = 0;
extern __shared__ volatile unsigned int sdata[];
sdata[0] = 0;
while (i<n) {
        temp |= g_idata[i] | g_idata[i + blockSize];
        i += gridSize;
}
__syncthreads();
while ((temp & t) != 0) {
        sdata[0] |= temp;
        __syncthreads();
        t = sdata[0] ^ temp;
}
if (tid == 0)  g_odata[blockIdx.x] = sdata[0];
__syncblocks(g_barrier);
if (bid == 0) {
        if (tid < gridDim.x) temp = g_odata[tid];
        __syncthreads();
        while ((temp & t) != 0) {
                sdata[0] |= temp;
                __syncthreads();
                t = sdata[0] ^ temp;
        }
        if (tid == 0)  g_odata[0] = sdata[0];
        }
}
```

reduceMax_CTRR_GBS()

Reduction MAX operation over 32-bit words using global block synchronization. The reduction within each block follows the constant time race resolution algorithm.

```
__global__ void barrier_max
(volatile int *g_idata, volatile int *g_odata,
volatile int *g_barrier, int n, int blockSize)
{
extern __shared__ volatile int sdata[];
int tid = threadIdx.x;
int i = blockIdx.x*blockSize*2 + threadIdx.x;
int gridSize = blockSize*2*gridDim.x;
int flag = 1;
int temp = 0;
sdata[tid]=0;
while (i<n) {
        temp = sdata[tid];
        sdata[tid] = (g_idata[i] > g_idata[i + blockSize]) ?
                        g_idata[i] : g_idata[i + blockSize];
        sdata[tid] = (sdata[tid] > temp) ? sdata[tid] : temp;
        temp = 0;
        i += gridSize;
}
__syncthreads();
while(flag == 1) {
        sdata[0] = (sdata[0] < sdata[tid]) ?
                sdata[tid] : sdata[0];
        __syncthreads();
        flag = 0;
        flag = (sdata[tid] > sdata[0]);
}
g_odata[blockIdx.x] = sdata[0];
flag = 1;
__syncthreads();
__syncblocks(g_barrier);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) sdata[tid] = g_odata[tid];
        __syncthreads();
        while(flag == 1) {
                sdata[0] = (sdata[0] < sdata[tid]) ?
                        sdata[tid] : sdata[0];
                __syncthreads();
                flag = 0;
                flag = (sdata[tid] > sdata[0]);
        }
        g_odata[0] = sdata[0];
        }
}
```

reduceMin_CTRR_GBS()

Reduction MIN operation over 32-bit words using global block synchronization. The reduction within each block follows the constant time race resolution algorithm.

```
__global__ void reduce_Min
(int *g_idata, volatile int *g_odata,
volatile int *g_barrier, int n, int blockSize)
{
extern __shared__ volatile int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
volatile int flag = 1;
int temp = 0x7FFFFFFF;
sdata[tid]=0x7FFFFFFF;
while (i<n) {
        temp = sdata[tid];
        sdata[tid] = (g_idata[i] < g_idata[i + blockSize]) ?
                g_idata[i] : g_idata[i + blockSize];
        sdata[tid] = (sdata[tid] < temp) ? sdata[tid] : temp;
        temp = 0x7FFFFFFF;
        i += gridSize;
}
__syncthreads();
while(flag == 1) {
        sdata[0] = (sdata[0] <= sdata[tid]) ?
                        sdata[0]: sdata[tid];
        __syncthreads();
        flag = 0;
        flag = (sdata[tid] < sdata[0]);
}
g_odata[blockIdx.x] = sdata[0];
flag = 1;
__syncthreads();
__syncblocks(g_barrier);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) sdata[tid] = g_odata[tid];
        __syncthreads();
        while(flag == 1) {
                if (sdata[tid] < sdata[0]) sdata[0] = sdata[tid];
                __syncthreads();
                flag = 0;
                flag = (sdata[tid] < sdata[0]);
        }
        g_odata[0] = sdata[0];
        }
}
```

p_bcast()

Broadcast function. If thread ID corresponds to the parameter `b_cast`, then `thread[b_cast]` writes its value to global memory where is available for all other threads.

```
__global__ void p_bcast
(int *g_idata, volatile int *g_odata, volatile int *g_barrier,
        int b_cast)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (b_cast == i) g_odata[0] = g_idata[i];
   __syncblocks(g_barrier);
   g_idata[i] = g_odata[0];
   __syncblocks(g_barrier);
}
```

p_count()

Count function. Evaluates how many threads are ON and works similarly as the reduce summation function.

```
__global__ void p_count(int *g_idata, int *g_odata_in, unsigned int n)
{
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
register volatile int *g_odata = g_odata_in;
register volatile int *barvec = (g_odata + gridDim.x);
sdata[tid] = 0;

while (i < n){
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;}
__syncthreads();

if (blockSize >= 512){
        if (tid < 256) {
                sdata[tid] += sdata[tid + 256];}
__syncthreads();
}
if (blockSize >= 256){
        if (tid < 128) {
                sdata[tid] += sdata[tid + 128];}
__syncthreads();
}
```

```
if (blockSize >= 128) {
        if (tid <  64) {
                sdata[tid] += sdata[tid +  64];
        }
__syncthreads();
}
#ifndef __DEVICE_EMULATION__
if (tid < 32)
#endif
{
        if (blockSize >=  64) {
                sdata[tid] += sdata[tid + 32];
                EMUSYNC;        }
        if (blockSize >=  32) {
                sdata[tid] += sdata[tid + 16];
                EMUSYNC;        }
        if (blockSize >=  16) {
                sdata[tid] += sdata[tid +  8];
                EMUSYNC;        }
        if (blockSize >=   8) {
                sdata[tid] += sdata[tid +  4];
                EMUSYNC;
        }
        if (blockSize >=   4) {
                sdata[tid] += sdata[tid +  2];
                EMUSYNC;
        }
        if (blockSize >=   2) {
                sdata[tid] += sdata[tid +  1];
                EMUSYNC;
        }
}
if (threadIdx.x == 0) g_odata[blockIdx.x] = sdata[0];
__syncblocks((volatile unsigned int *) barvec);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) {
                sdata[tid] = g_odata[tid];
        }
        __syncthreads();
        if (gridDim.x >= 128) {
                if (tid <  64) {
                        sdata[tid] += sdata[tid +  64];
                }
                __syncthreads();
        }
#ifndef __DEVICE_EMULATION__
        if (tid < 32)
#endif
        {
                if (gridDim.x >=  64) {
                        sdata[tid] += sdata[tid + 32];
                        EMUSYNC;
                }
```

```
                if (gridDim.x >=  32) {
                        sdata[tid] += sdata[tid + 16];
                        EMUSYNC;
                }
                if (gridDim.x >=  16) {
                        sdata[tid] += sdata[tid +  8];
                        EMUSYNC;
                }
                if (gridDim.x >=   8) {
                        sdata[tid] += sdata[tid +  4];
                        EMUSYNC;
                }
                if (gridDim.x >=   4) {
                        sdata[tid] += sdata[tid +  2];
                        EMUSYNC;
                }
                if (gridDim.x >=   2) {
                        sdata[tid] += sdata[tid +  1];
                        EMUSYNC;
                }
        }
        if (threadIdx.x == 0) g_odata[0] = sdata[0];
    }
}
#endif // #ifndef _REDUCE_KERNEL_H_
```

p_first()

This function evaluates what is the lower thread ID corresponding to an ON thread.

```
__global__ void p_first
(int *g_idata, volatile int *g_odata,
        volatile int *g_barrier, int f, int blockSize)
{
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
int temp;
int flag = 1;
extern __shared__ volatile int sdata[];
sdata[0] = blockDim.x*gridDim.x;
temp = g_idata[i];
__syncthreads();

while (flag == 1) {
        if (temp == f) sdata[0] = tid;
        flag = 0;
        __syncthreads();
        if (sdata[0] > tid && temp == f) flag = 1;
}
g_odata[blockIdx.x] = sdata[0]+blockDim.x*blockIdx.x;
__syncblocks(g_barrier);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) {
                temp = g_odata[tid];
                flag = 1;
        }
        __syncthreads();
        if (tid == 0) {
                sdata[0] = temp;
                flag = 0;
        }
        while (flag == 1) {
                if (temp < sdata[0]) sdata[0] = temp;
                flag = 0;
                __syncthreads();
                if (temp < sdata[0]) flag = 1;
        }
        if (tid == 0)  g_odata[0] = sdata[0];
        }
}
```

p_quantify()

This function returns 0 if no threads are ON, returns 1 if only one thread is ON, or returns 2 if two or more threads are ON.

```
__global__ void p_quantify
(int *g_idata, volatile int *g_odata,
        volatile int *g_barrier, int n)
{
extern __shared__ volatile int sdata[];
int tid = threadIdx.x;
int i = blockIdx.x*blockDim.x*2 + threadIdx.x;
int gridsize = blockDim.x*2*gridDim.x;
sdata[0] = 0;
while (i<n) {
        sdata[tid] |= g_idata[i] | g_idata[i + blockDim.x];
        i += gridsize;
}
__syncthreads();

if (sdata[tid] != 0) sdata[0] = 1;
sdata[0] = (sdata[0] != 0);
__syncthreads();
if (sdata[0] != 0) {
        if (sdata[tid] == 1) { sdata[0] = tid; __syncthreads();
                if (sdata[0] == tid) { sdata[tid] = 0; __syncthreads();}
                if (sdata[tid] == 1) { sdata[0] = 2; __syncthreads();}
                else sdata[0] = 1;
        }
}
__syncthreads();
if (threadIdx.x == 0) g_odata[blockIdx.x] = sdata[0];
__syncblocks(g_barrier);
if (blockIdx.x == 0) {
        if (tid < gridDim.x) sdata[tid] = g_odata[tid];
        __syncthreads();
        if (sdata[tid]==2) { sdata[0]=2; goto L1; }
        if (sdata[tid]==1) { sdata[0]=1; goto L1; }
        if (sdata[tid]==0) { sdata[0]=0; goto L1; }
L1:
        if (threadIdx.x == 0) g_odata[0] = sdata[0];
        }
}
```

p_vote()

Returns a bit vector with element 2^K set to 1 *iff* thread K voted for thread with threadID = V.

```
__global__ void p_vote
(int *g_idata, volatile int *g_odata,
        volatile int *g_barrier, int v)
{
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
__syncblocks(g_barrier);
g_odata[i] = (g_idata[i] == v);
__syncblocks(g_barrier);
}
```