Parallel Asynchronous Modelization and Execution of Cholesky Algorithm using Petri Nets

Gustavo Wolfmann  
Laboratorio de Computación  
Fac. Cs. Exactas Físicas y Naturales  
Universidad Nacional de Córdoba  
Córdoba - Argentina  
gwolfmann@efn.uncor.edu

Armando De Giusti  
III LIDI  
Fac. Informática  
Universidad Nacional de La Plata  
La Plata - Argentina  
degiusti@lidi.info.unlp.edu.ar

Abstract—Parallelization of algorithms with hard data dependencies has a lack of task synchronization. Synchronous parallel versions are simple to model and program, but inefficient in terms of scalability and processors use rate. The same problem for asynchronous versions with elemental static task scheduling. Efficient asynchronous algorithms implement out of order execution and are complex to model and execute. In this paper we introduce Petri Nets as a tool that simplifies the modeling and execution of parallel asynchronous versions of these kind of algorithms, while using an efficient dynamic task scheduling implementation. The Cholesky factorization algorithm was used as testbed. Simulations was done as a proof of concept, based on real execution times on GPGPU’s, and shows excellent performances.

Keywords—Petri Net Modelization - Asynchronous Parallel Execution - Dynamic Task Scheduling - Cholesky Factorization Algorithm.

I. INTRODUCTION

The fork-join parallelization model is a natural step from a sequential to a parallel versions of an algorithm. An important drawback is the insertion of synchronization points in the algorithm that compel all the processors involved in the execution, to wait in idle state, the slower. This fact cause a poor performance and scalability in algorithms where the tasks load of each parallel thread differ, which is typical in algorithms with data dependency [1]. QR, LU and Cholesky factorizations are algorithms with this kind of problems.

Tiled algorithms emerge as a solution to the problem of load balance for dense linear algebra algorithms on multicore processors [2]. These kind of algorithms are an evolution from rectangular block based algorithms, where data reusability was the concept to optimize. Tiled algorithms hold, as many LAPACK algorithms do, two fundamentals steps in the algorithm: panel factorization and trailing submatrix update. But now, the key concepts are fine granularity and asynchronicity to achieve better thread level parallelism.

Tiled algorithms divide data in square blocks that allow to compute “out of order”, increasing the number of tasks that can run in parallel. As with block based algorithms, factorizations and updates consists in to apply the proper routines (“kernels”) between the operations defined in the BLAS [3] and LAPACK libraries [4]. Blocks size are tuned to reach a good performance in the execution of the kernels involved in the algorithm.

The major difference between block an tiled algorithms is that while the former are synchronous, the later are asynchronous. That difference is well exposed graphically in [5]. Asynchronicity and fine granularity make possible that many tasks run in parallel. “Out of order” means that while one processor compute a factorization, the others can compute updates, simultaneously.

As the number of tasks available to run in parallel exceed the number of processors, different selection of tasks can be done, defining the scheduling of the parallel algorithm. Static scheduling are those defined prior the algorithm execution. Common examples are the left looking (LL) or right looking (RL), which differ according if priority updates are on the left or on the right of the actual factorization panel [1], [6]. Both algorithms are exposed inf Fig. 1 and 2, and has in common that are fork-join synchronized.

Other known technique of static scheduling is look ahead. As LL and RL, is based on performing panel factorization in one thread while the remaining update submatrix from previous stages. It has been observed that LL and RL are extreme points of a wide spectrum of possibilities of tasks selection, acting parametrized look ahead as a path for to go from one point to other [1]. All alternatives generate bubbles of idleness in the algorithm due its static nature.

It has been used Directed Acyclic Graphs (DAG) to model the algorithms, where the vertex represents tasks and the edges the dependency among them. The graph is also know as Dependency Graph. Asynchronous execution is helped with the use of DAG’s to control the dependency of tasks. The
DAG is principally used by the scheduler to select the next task [2].

Dynamic scheduling is introduced to improve statics, by selecting the task on run time according the availability of free processors and enabled tasks. These kind of scheduling are aimed to prevent the existence of stalled point of static scheduler. However, are complex and cause overhead in the algorithm execution [6].

Hogg’s research shows no significant advantage in using dynamic or static schedulers [7]. He also use a DAG to model the algorithm and scheduling control. Concurrency control and DAG implementation generate an overhead that seems to consume the improvements of the dynamic scheduler.

Also in the line of dynamic scheduling, LAWN 243 [6] introduce the use of the “locality” parameter to help the scheduler to select dynamically the next task to a processor, according the data previously used. Improvements in parallel execution depends on the variant of algorithm (LL or RL), the size of the DAG’s window resident in memory and the number of tiles in what is divided the matrix.

At the best of our knowledge, all the dynamics intents are based on DAG, which is good to represent the structure of the algorithm, but not for the execution and the scheduler. Both are implemented in an ad-hoc, sophisticated style, without parallel execution modelization.

A key factor to achieve a performing parallel algorithm, is to minimize the idle time of the processors due synchronization. Asynchronous execution is a big step in this path. Scheduling is other. Tiled algorithms improves the parallelism of an algorithm by increasing the number of tasks. The drawback is the complexity of managing a big number of parallel asynchronous tasks. The lack of a model for this produces complex implementations.

Our research has two main objectives:

- To model the structure and parallel execution of dense linear algebra algorithms with a simple tool.
- To improve the performance by minimizing idle times of processors using a dynamic scheduler

The second follow the first: with a simple model, dynamics overhead decrease and the scheduler can perform an adequate selection without loss of performance.

Petri Net is the formalism chosen to represent the algorithm. Is know its capability to represent parallel processes. Few additions to this well-know formalism are enough to achieve our objectives. As a “proof of concept” we develop a simulation tool to represent and execute the Petri Nets, to simulate different running parameters.

Cholesky factorization was elected as a testbed algorithm. We follow the kernels and DAG representation used in tiled algorithm defined in [8]. These kernels are xPOTRF, xGEMM, xTRSM and xSYRK, where x can be ’s’ or ’d’ depending on using single or double precision data.

II. PETRI NET MODEL OF PARALLEL ALGORITHMS

A Petri Net (PN) is a bipartite directed graph composed by Places and Transition nodes. Usually, Places represent “states” and Transitions “actions”. Arcs links always one Place to one Transition (acting as input) or vice versa (as output). There exists tokens, which only live in places, and represents “facts”. The overall state evolves when a transition is ”fired”, moving tokens from input places to output places. One transition can be fired when all input places has enough tokens [9]. This net is also known as Token Petri Net (TPN).

Petri Nets are used to model the algorithm, representing operations (kernels to execute) with Transitions and data with Places. Input parameters are represented by arcs from Places to Transitions and operations results, with arcs from Transitions to Places.

Petri Net can model the algorithm dependencies, and also can describe the execution by the firing of Transitions. Next subsections explain how the net is used to both purposes.

A. Coloured Petri Net

Coloured Petri Nets (CPN) is a kind of the many defined “High Level Petri Nets”. The major difference with TPN is that tokens have different values (“colours”) from a domain. This fact allow to model with a high level of abstraction. Now, transitions are enabled by having not only enough tokens in his input places and also from the ”color” defined. CPN definition is pursued from [9], [10].

Coloured Petri Nets allow to model complex nets at high level in a simple manner. DAG of task dependencies with many blocks divisions are difficult to understand due its high number of nodes (see Fig. 10 of LAWN 243 [6]). To model tiled algorithms with CPN, the main domain used to define tokens is tile position, represented by pair row-column.

The strategy to modelize the algorithm is:
1) Each operation is represented by one transition
2) For each transition, there are so many input Places as data blocks parameters are involved in the operation.
3) No more places or transitions are used.
4) Output arcs represents data dependency.

To specify conditions in places, we extend or restrict the tile-block domain. Also, multisets are used to represent repetitions of blocks and arcs expressions functions to limit token flowing [10].

Fig. 3 show the CPN that represents the Cholesky algorithm. It has only four transitions and eight places, according to the strategy suggested. Name of the places follow the number of the block used in each operation. Color token is represented by < x,y >, multiset repetitions by braces { x }, and functions arcs are only booleans of the form if(cond).

In each place, the domains used are:

For potr1 and trsm2, the domain is < i , i >, i = 1 . . . n.

For trsm1, syrk1, and gemm1, the domain is < j, i > = j = 2 . . . n, i = 1 . . . j − 1, j > i

For gemm2 the domain is < j, i >, j = 3 . . . n, i = 1 . . . j − 2, j > i

For syrk2 the domain is < j, j , i >, j = 2 . . n ∧ i = 1 . . j − 1 ∧ j > i
Fig. 3. Coloured Petri Net that represents Cholesky factorization algorithm.

For gemm3 the domain is $< j, i, q >, j = 3 \ldots n, i = 2 \ldots n - 1, q = 1 \ldots i - 1 \land j > i \land j > q$

The initial places mark is:

- In potr1: $< 1, 1 >$
- In trsm1: $< i, 1 >, i = 2 \ldots n$
- In syrk2: $< i, 1 >, i = 2 \ldots n$
- In gemm3: $< j, i, 1 >, j = 3 \ldots n, i = 2 \ldots j - 1$

In this way, a tiled algorithm is generically defined by a CPN, dispense with any consideration in the number of tiles, which is only a parameter to domain definition. Is highlighted its simplicity and facilities to analyze the parallel algorithm.

Nevertheless, CPN’s are not used to execute the algorithm. The overhead to represent abstractly domains and function arcs, is expensive in terms of high performance computing. By other side, the CPN developed in this way fulfill the definition of well-formed CPN [9]. This kind of nets are easily transformed to a TPN, which computational implementation is simple and light to execute.

B. Token Petri Net

Previous section shows the facilities to model a parallel algorithm with a CPN and the procedure used to define places and transitions. The resulting net is easily unfolded to a TPN.

To unfold a CPN we follow the steps defined in Diaz [9]. Each Place $P_j$ in a CPN has a Domain $D(P_j)$ associated with it, and is unfolded to generate so many Places in TPN as is the cardinality of $D(P_j)$ in the colored Place. The repetitions from the bag that represent the Place, must to be respected. Thus, each Place in TPN is associated with an unique value from the pairs (color, place) in CPN and repeated according the bag pair cardinal.

Table in Fig. 4 show an unfolding example for Places from CPN to TPN, for the case of $3 \times 3$ tiles divisions. The names of Places in TPN follow the respective name in CPN, concatenate with the color of the token that represents. By example, syrk132, is the Place in TPN, that came from Place syrk1 with color $< 3, 2 >$ in CPN, and represents the first argument in syrk operation of the tile in third row, second column. Graphically, the unfolded TPN of the example is exposed in Fig. 5.

Is not difficult to see how fast grow the number of Places and Transitions in TPN with an incresing number of tiles divisions. Is practically impossible to show and understand its graphical representation. However, the matricial representation is elemental and easy to use.

<table>
<thead>
<tr>
<th>Place in CPN</th>
<th>Domain in CPN</th>
<th>Places in TPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>potr1</td>
<td>$&lt; i, i &gt;$</td>
<td>potr111, potr112, potr113</td>
</tr>
<tr>
<td></td>
<td>$i = 1 \ldots n$</td>
<td></td>
</tr>
<tr>
<td>trsm1</td>
<td>$&lt; j, i &gt;$</td>
<td>trsm121, trsm122, trsm123</td>
</tr>
<tr>
<td></td>
<td>$j = 2 \ldots n$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$i = 1 \ldots j - 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$j &gt; i$</td>
<td></td>
</tr>
<tr>
<td>trsm2</td>
<td>$&lt; i, i &gt;$</td>
<td>trsm211, trsm222</td>
</tr>
<tr>
<td></td>
<td>$i = 1 \ldots j - 1$</td>
<td>${2}$</td>
</tr>
<tr>
<td></td>
<td>$j &gt; i$</td>
<td></td>
</tr>
<tr>
<td>syrk1</td>
<td>$&lt; j, i &gt;$</td>
<td>syrk121, syrk131, syrk132</td>
</tr>
<tr>
<td></td>
<td>$j = 2 \ldots n$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$i = 1 \ldots j - 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$j &gt; i$</td>
<td></td>
</tr>
<tr>
<td>syrk2</td>
<td>$&lt; j, i &gt;$</td>
<td>syrk2221, syrk2331, syrk2332</td>
</tr>
<tr>
<td></td>
<td>$j = 2 \ldots n$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$i = 1 \ldots j - 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$j &gt; i$</td>
<td></td>
</tr>
<tr>
<td>gemm1</td>
<td>$&lt; j, i &gt;$</td>
<td>gemm121, gemm122</td>
</tr>
<tr>
<td></td>
<td>$j = 2 \ldots n$</td>
<td>${1}$</td>
</tr>
<tr>
<td></td>
<td>$i = 1 \ldots j - 1, j &gt; i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>${n - j}$</td>
<td></td>
</tr>
<tr>
<td>gemm2</td>
<td>$&lt; j, i &gt;$</td>
<td>gemm231, gemm232</td>
</tr>
<tr>
<td></td>
<td>$j = 3 \ldots n$</td>
<td>${1}$</td>
</tr>
<tr>
<td></td>
<td>$i = 1 \ldots j - 2, j &gt; i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>${j - i - 1}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>${n - j}$</td>
<td></td>
</tr>
<tr>
<td>gemm3</td>
<td>$&lt; j, i, q &gt;$</td>
<td>gemm3321</td>
</tr>
<tr>
<td></td>
<td>$j = 3 \ldots n$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$i = 2 \ldots n - 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$q = 1 \ldots i - 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$j &gt; i &gt; q$</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4. Unfold example for Places for the Coloured Petri Net in Fig.3 supposing only $3 \times 3$ tiles divisions.
The importance of unfolding is that a TPN can be represented with two matrix and one vector of natural numbers, and elemental matrix - vector operations models the execution of the net.

The structure of TPN net can be represented by Negative and Positive Incidence Matrix (NIM / PIM). Both have dimension $p \times t$, where $p$ is the number of Places and $t$ the number of Transitions. Each position in the matrix represents the relation between a pair place/transition, the arc between them in terms of graph theory. A position with zero represents absence of arc. A positive value represents the number of tokens that will be absorbed / injected by the transition depending on Negative or Positive case, if the transition is fired.

Token existence in Places is represented by a Mark Vector (MV). It has dimension $1 \times p$, and values are also naturals numbers. Values represents the number of tokens that exists in the respective Place.

In the matricial representation, is highlighted the facility to compute enabled transitions and fire them. We call $NI_j^-$ and $PI_j^+$ the j-th column (transition) in NIM and PIM respectively. By computing $MV - NI_j^-$, if the result has no negative values, then, MV has enough tokens in the input Places of j-transition, and can be fired.

Computing the difference for all the columns, determine all the transitions that are enabled to fire. By construction, Places of the unfolded TPN has only one transition to act as input. That guarantees no competition of enabled transitions for input tokens, and all enabled transitions can be fired simultaneously.

To modelize the net execution, an additional function is defined to compute the set of transitions enabled to be fired. The function is $h : \mathbb{N}^{1 \times p} \times \mathbb{N}^{p \times t} \rightarrow \mathbb{N}^{1 \times t}$, that has parameters $M$ y $NI^-$ and results is a vector, which values are:

$$h(j) = \begin{cases} 0 & \text{if } (M - NI_j^-) \text{ has negatives values} \\ 1 & \text{if } (M - NI_j^-) \text{ else} \end{cases} \quad j = 1 \ldots t$$

then, $h$ positions with value 1 reference to transitions enabled to be fired.

Firing all enabled transitions defines a new mark for Mark Vector ($MV'$):

$$MV' = MV - h \times NIM^t + h \times PIM^t \quad (1a)$$

III. Execution Model

DAG’s can model dependencies of tasks in a parallel algorithm, but it doesn’t modelize the execution. Petri Nets has implicit modelization of execution: by representing task as Transitions, all enabled Transitions all tasks that can be executed.

Nevertheless, TPN is not sufficient to model the execution of a parallel algorithm. It has not information about running time of a task, and has no limit about the number of processors that execute the task.

To solve the problem of time execution, we extend TPN to Timed Petri Nets (TiPN). They have an important feature, the representation of the time in Transitions. By adding a delay between the time in that tokens are absorbed from input places and the time in that tokens are injected in output places, transitions can represents the notion of execution time.

In TiPN, firing a transition $k$ implies MV update in two times:

$$MV' = MV - NI_k^- \quad \text{in } t_{ini} \quad (2a)$$

$$MV'' = MV' + PI_k^+ \quad \text{in } t_{ini} + \Delta(T_k) \quad (2b)$$

where $t_{ini}$ is the initial firing time, and $\Delta(T_k)$ is the execution time of task $T_k$. Due a time and delay considerations, is highly improbable that two transitions has the same initial firing time, then MV update is done be each transition fired.

To solve the problem of many processors, we define an execution model. The model is composed by a set of processors and one TiPN that represent the algorithm and its dependencies as we have use along this paper. Each processor know how to do the task that each transition represent. The TiPN is shared by all the processors. Each processor check the TiPN state to determine a task to do, between all enabled transitions. To prevent multiple selection of the same task, a mutual exclusion mechanism is added to TiPN.

Each processor execute the following pseudo-code parallel execution algorithm:
**Details of processor execution pseudo-code:**

- Main algorithm is the represented by the Petri Net.
- The exclusion is hold until the tokens are absorbed from input places, before the processor begins the task execution. No colision is produced by tokens injection.
- When more than one transition is enabled, a selection policy must to placed.
- A delay is introduced if the processor can’t hold the exclusion to avoid starvation.

The overhead introduced by the parallel execution is defined by three factors. First, the mutual exclusion mechanism, which execution use few clock cycles. Second, the integer matrix operations and vector updates, that are highly optimized to run in mili-seconds in today processors. Third, the selection policy must to be elected balancing between selection algorithm and overall algorithm performance. In fact, the sum of the time execution of three factors are several orders of magnitude smaller than task execution time, what means a minimum overhead.

### IV. Dynamic Scheduling

Task selection between all the enabled tasks is a key factor in the execution model. In the model implementation, Patterns from Object Oriented Design was chosen as a design tool. The design has three basic objects: one Petri Net, many Processors that interact with the PN, and Selectors that collaborate with Processors, to select the next executable task.

Each Processor has a link to the Petri Net Object and a link to one Selector. The Selector object is the responsible to define the task that the Processor will do. It have a method that, taking as input parameters the state of the PN and the processor, returns the next task to the Processor. The design principle is decouple processing from selecting tasks.

Different selecting policies are put in place with only implement the selecting method of Selector accordingly. This is a way to modelize homogeneous or even heterogeneous processors with different scheduling policy. Also, static or dynamic scheduling can easily implemented using the appropriate Selector collaborator.

Simulation tests was developed to run static and dynamic tasks schedulers. In both cases, task assignation to a processor is dynamic, i.e. static or dynamic refers to the execution sequence, not the execution processor.

Two static schedulers are tested, following LL and RL algorithms. They where implemented easily by defining the order of tasks that Selector must to follow. The sequence was defined from the algorithms showed in Fig. 1 and Fig. 2.

Two dynamic schedulers are tested. Both are based on DAGs, but differs in the selecting metric applied. The first, called *height tree* (*HT*), select the enabled task that is a higher in the dependency tree. The second, called *inverse tree* (*IT*), select the enabled task that has longer path to finish in the graph. By longer path we means that has a bigger number of steps in the longest path from the actual to the end task. In both cases, non deterministic selection is done for equal height.

Figures 6 and 7 shows examples of DAG’s of dynamics schedulers used in tests. In *height tree* the level of a task is assigned according the step in which is enabled the task. In *inverse tree*, the level is assigned according to number of steps.
in the longest path to the end. By example, task \textit{trsm41} has level 8 in the first graph and level 6 in the second.

Differences between both schedulers are exposed in the next example. Suppose you have three processors. Following the dynamic schedulers previously showed, the first step is compute \textit{potr11}, and then compute \textit{trsm21}, \textit{trsm31} and \textit{trsm41}. In the third step, scheduler \textit{height tree}, must select any task from all that have assigned level 7, but scheduler \textit{inverse tree} will select exactly \textit{syrk21}, \textit{gemm2131} and \textit{gemm2141}. Is easy to see in Fig. 7 that \textit{syrk21} is a priority tasks in the path to the end because it enable \textit{potr22}. Scheduler \textit{height tree}, due its non determination, may delay it selection.

V. SIMULATION RESULTS

Simulations of parallel algorithm was tested with different values to four parameters: matrix size, number of parallel processors, block division number and scheduler used. To test performance of each combination of parameters, a simulation tool was developed using a high level language (Smalltalk). The tool take the number of blocks and define all the tasks to execute, then take matrix size and decide the block size, and finally take the processors number and create the same number of Processors objects and one thread for each one to execute in parallel. According the scheduler, the respective Selector object is linked to each Processor. The execution of each kernel is simulated by throwing a delay of time according the task.

To execute the simulations, was obtained the time of running of each kernel, for different block sizes in a NVIDIA GTX 470 GPU. CUDA was used for BLAS kernels xTRSM, xGEMM, xSYRK and MAGMA for LAPACK kernel xPOTRF. Results are showed in Table I. In all cases, is considered the time of communication of all data from main memory to GPU and vice versa. Was taken the assumption that the main processor use one thread to control each GPGPU.

Table II shows results of simulations with only four processors and block range of 6000 and 8000, single precision. Due space limitation, only these results are presented, nevertheless they are representative of other combination of execution parameters. The metric used is the idleness of processors, which is calculated from simulations as a difference between total execution time and total of processing time. Is clear that it includes the selection time.

For Cholesky factorization algorithm, RL algorithm brings the best results for static scheduling, consistent with previous work [8]. For dynamic scheduling, \textit{inverse tree} brings results near the optimum. A timeline for both schedulers are show in Figures 8 and 9. Two things are noted: the idle time of processors in synchronization points in RL and the almost absence of idle time in IT.

The nature of Cholesky algorithm imposes no parallelism in the beginning and in the end of execution, which represents one and three task respectively. Beyond that points, also at the end of execution, there are a limited number of parallel tasks which produce idle state for some processors. The rest of execution, all processors are always working.

VI. CONCLUSION AND FUTURE RESEARCH

We have developed a model of parallel programming starting from coloured petri nets, unfolding them to token petri nets and executed by a set of distributed processors that share in a memory area, the representation of the state of the algorithm, and have decoupled the execution from the selection of the next task to do.

The model was used as simulation tool, but it is easy to adapt to run real algorithms. We hope to get performance improvements, due a minimal overhead of the scheduling policy and its almost optimal "idleness" rate of processors.

The model is adaptable to several numbers of processors and data block partitions: the unfolding of the CPN capture the number of partitions by generating the respective incidence matrix. Data dependencies are automatically generated. Besides, the execution model only need as parameter, the matricial information, so, to execute different algorithms, no programming is necessary, only to change the matrix.

Dynamic scheduling can be executed without need of previous time execution of each kernel. That information is necessary to achieve an optimal scheduling, at cost of more complex schedulers. Our simulations shows a result that is near the optimal, with a very light overload. Others dynamic scheduling policies may achieve optimal or sub-optimal results, but they are complex to understand and implement.

Execution on a set of asymmetric processors can be implemented with only change the Selector of task in each processor. By restricting to slower processors the execution of tasks that have forwarding dependencies in a non critical path, that kind of processors can aid to the overall parallel execution.

Future work will implement the effective execution with this model, and not only for linear algebra factorizations, but also for others algorithms. An implementation in a distributed memory parallel architecture will be researched.

<table>
<thead>
<tr>
<th>Block Size</th>
<th># Blocks</th>
<th># Procs.</th>
<th>Algor.</th>
<th>Time (sec)</th>
<th>% idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000</td>
<td>6</td>
<td>4</td>
<td>LL</td>
<td>17.50</td>
<td>53.66</td>
</tr>
<tr>
<td>6000</td>
<td>6</td>
<td>4</td>
<td>RL</td>
<td>13.26</td>
<td>38.77</td>
</tr>
<tr>
<td>6000</td>
<td>6</td>
<td>4</td>
<td>HT</td>
<td>10.16</td>
<td>20.56</td>
</tr>
<tr>
<td>6000</td>
<td>6</td>
<td>4</td>
<td>IT</td>
<td>9.51</td>
<td>14.97</td>
</tr>
<tr>
<td>6000</td>
<td>8</td>
<td>4</td>
<td>LL</td>
<td>40.59</td>
<td>53.35</td>
</tr>
<tr>
<td>6000</td>
<td>8</td>
<td>4</td>
<td>RL</td>
<td>26.33</td>
<td>28.47</td>
</tr>
<tr>
<td>6000</td>
<td>8</td>
<td>4</td>
<td>HT</td>
<td>20.98</td>
<td>10.95</td>
</tr>
<tr>
<td>6000</td>
<td>8</td>
<td>4</td>
<td>IT</td>
<td>20.69</td>
<td>9.65</td>
</tr>
<tr>
<td>8000</td>
<td>6</td>
<td>4</td>
<td>LL</td>
<td>36.89</td>
<td>53.35</td>
</tr>
<tr>
<td>8000</td>
<td>6</td>
<td>4</td>
<td>RL</td>
<td>27.79</td>
<td>38.08</td>
</tr>
<tr>
<td>8000</td>
<td>6</td>
<td>4</td>
<td>HT</td>
<td>21.37</td>
<td>19.64</td>
</tr>
<tr>
<td>8000</td>
<td>6</td>
<td>4</td>
<td>IT</td>
<td>19.95</td>
<td>13.96</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>4</td>
<td>LL</td>
<td>85.34</td>
<td>53.16</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>4</td>
<td>RL</td>
<td>55.05</td>
<td>25.37</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>4</td>
<td>HT</td>
<td>44.30</td>
<td>10.21</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>4</td>
<td>IT</td>
<td>43.71</td>
<td>9.87</td>
</tr>
</tbody>
</table>

TABLE I. OBSERVED TIME FOR THE KERNELS EXECUTED OVER AN NVIDIA GTX 470 GPU, IN SECONDS.

TABLE II. OBSERVED VALUES OF SIMULATIONS.
**ACKNOWLEDGMENT**

The authors would like to thank to professor Orlando Micolini for his continuous suggestions and contributions.

**REFERENCES**


