Exploiting GPU On-Chip Shared Memory for Accelerating Schedulability Analysis

Swaroop Nunna[1], Unmesh D. Bordoloi[2], Samarjit Chakraborty[1], Petru Eles[2], Zebo Peng[2]
[1]TU Munich, Germany
[2]Linköpings Universitet, Sweden
[1]E-mail: {swaroop.nunna, samarjit.chakraborty}@rcs.ei.tum.de
[2]E-mail:{unmbo, petel, zebpe}@ida.liu.se

***Abstract*—Embedded electronic devices like mobile phones and automotive control units must perform under strict timing constraints. As such, schedulability analysis constitutes an important phase of the design cycle of these devices. Unfortunately, schedulability analysis for most realistic task models turn out to be computationally intractable (NP-hard). Naturally, in the recent past, different techniques have been proposed to accelerate schedulability analysis algorithms, including parallel computing on Graphics Processing Units (GPUs). However, applying traditional GPU programming methods in this context restricts the effective usage of on-chip memory and in turn imposes limitations on fully exploiting the inherent parallel processing capabilities of GPUs. In this paper, we explore the possibility of accelerating schedulability analysis algorithms on GPUs while exploiting the usage of on-chip memory. Experimental results demonstrate upto $9\times$ speedup of our GPU-based algorithms over the implementations on sequential CPUs.**

I. Introduction

Many embedded devices like mobile phones and automotive control units have to satisfy strict timing constraints. Hence, the design cycles of these devices involve a schedulability or timing analysis phase. Typically, a system designer would choose the values of certain parameters (e.g., processor frequency and task deadlines) and invoke a schedulability analysis tool to determine whether the timing constraints are met. If such an analysis returns a negative answer, then some of the parameters are modified and the analysis is invoked once again. This iterative adjustment of the system parameters is repeated till the performance constraints are satisfied. This process is illustrated in Figure I. However, schedulability analysis for most realistic task models are computationally intractable (NP-hard) [7]. Hence, each iteration takes a long time to run. This critically impacts the usability of the tool in the interactive design sessions. In order to reduce the long running times, in this paper, we propose a technique for accelerating a schedulability analysis algorithm by implementing it on Graphics Processing Units (GPUs). In particular, we show that effectively exploiting the GPU on-chip shared memory can lead to more speedups compared to a straightforward parallel implementation of the algorithm on the GPU.

Motivation for using GPU: Our work has been motivated by the recent trend of applying GPUs to accelerate non-graphics applications. There are many compelling reasons behind exploiting GPUs for such non-graphics related applications (in contrast to using, say, ASIC/FPGA-based accelerators).

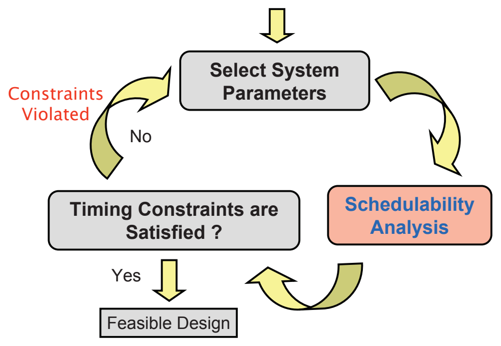

Fig. 1. The iterative design cycle.

Firstly, modern GPUs are extremely powerful; high-end GPUs, such as the nVIDIA GeForce 8800 GTX, have a FLOPS rates of around 330 GigaFLOPS, whereas high-end general-purpose processors are only capable of around 25 GigaFLOPS. Secondly, GPUs are now commodity items as their costs have dramatically reduced over the last few years. Hence, the attractive price-performance ratios of GPUs give us an enormous opportunity to change the way computer-aided tools for embedded system design perform, with almost no additional cost. In fact, recent years have seen the increasing use of graphics processing units (GPUs) for different general-purpose computing tasks. These span across numerical algorithms [11], computational geometry [1], database processing [2], image processing [14]. Of late, there has also been a lot interest in accelerating computationally expensive algorithms in the computer-aided design of electronic systems [10], [9], [12] using GPUs. Our paper follows this line of work and proposes a novel technique to implement schedulability analysis algorithms on GPUs.

Related Work and Our Contributions: To accelerate the tedious design cycles associated with schedulability analysis, many different approaches have been proposed in the literature. In [3], [8], approximation algorithms have been introduced, while an *interactive* analysis scheme has been presented in [5]. However, unlike exact schedulability analysis, approximate schedulability analysis might return false positives or false negatives. For example, if the algorithm is allowed to return false positives, then, in some cases, although a task set is not schedulable, the algorithm incorrectly returns schedulable. Nevertheless, it can be guaranteed that even in such cases no task will miss its deadline by more than a prespecified time interval.

On the other hand, the *interactive* analysis scheme [5] exploits the repetitive nature of the iterative design cycle in order to achieve speedup. When the algorithm is invoked for the first time, the full algorithm is allowed to run, but certain data structures are created and stored. When the algorithm is invoked in successive iterations after modifying a small set of system parameters, these data structures are exploited to partially run the algorithm and still guarantee the correct result. However, this scheme allows only a *small* number of the system parameters to be changed in each iteration, and hence doesn't scale well for large number of changes or when a completely new design needs to be analyzed.

Recently, Graphics Processor Units (GPUs) were also utilized to improve the running times of the schedulability analysis engine [6]. Unlike the techniques mentioned above, the approach proposed in [6] always gives optimal results and it is not restricted to *small* number of changes to the parameters to achieve the speed up. However, this technique used a traditional GPU programming model with Cg [13] and OpenGL [16]. This model does not expose the on-chip memory to the programmers and thus, imposes limitations on fully exploiting the inherent parallel processing capabilities of GPUs. In this context, Compute Unified Device Architecture (CUDA) [15] — which is a new parallel computing architecture based on GPUs — seems to be a promising alternative. Unlike the traditional GPU programming model, CUDA GPUs can be programmed using an extension of C and requires no previous expertise in graphics programming. In this paper, we explore the possibility of accelerating system-level design analysis algorithms with General Purpose GPU (GPGPU) programming with CUDA. In particular, we present the execution speed-ups achieved by implementing the schedulability analysis on CUDA as well as the performance enhancements corresponding to the usage of on-chip memory.

We demonstrate our technique using the schedulability analysis algorithm of the recurring real-time task model proposed by Baruah in [4]. We chose this task model because it is especially suited for accurately modeling conditional real-time code with recurring behavior, i.e., where code blocks have conditional branches and run in an infinite loop, as is the case in many embedded applications. We describe the recurring real-time task model and its schedulability analysis in the next section. Section III describes the CUDA architecture and programming model. Thereafter, our proposed technique of accelerating the schedulability analysis using CUDA is described in Section IV and the results are reported in Section V.

II. Recurring Real-time Task Model

As mentioned above, in this paper we consider the recurring real-time task model. A recurring real-time task T is represented by a task graph which is a directed acyclic graph with a unique source (a vertex with no incoming edges) and a unique sink (a vertex with no outgoing edges). Associated with each vertex v of this graph is its execution requirement $e(v)$, and deadline $d(v)$. Whenever a vertex v is triggered, it generates a job which has to be executed for $e(v)$ amount of time within $d(v)$ time units from the triggering-time. Each directed edge (u, v) in the graph is associated with a minimum intertriggering separation $p(u, v)$, denoting the minimum amount of time that must elapse before vertex v can be triggered after the triggering of vertex u.

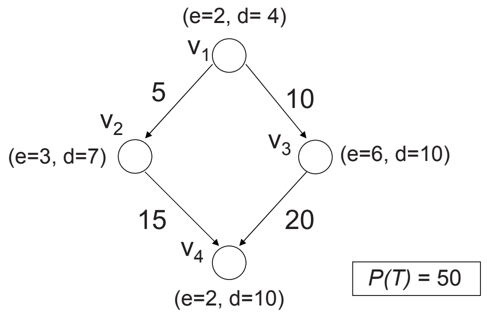

Fig. 2. An example recurring real time task.

The semantics of the execution of such a task graph state that the source vertex can be triggered at any time, and if some vertex u is triggered then the next vertex v can be triggered only if there exists a directed edge (u, v) and at least $p(u, v)$ amount of time has passed since the triggering of the vertex u. If there are directed edges (u, v_1) and (u, v_2) from the vertex u (representing a conditional branch) then only one among v_1 and v_2 can be triggered, after the triggering of u. The triggering of the sink vertex can be followed by the source vertex getting triggered again but any two consecutive triggerings of the source vertex should be separated by $P(T)$ units of time, called the *period* of the task graph.

Therefore, a sequence of vertices $v_1, v_2, \ldots, v_k$ getting triggered at time instants $t_1, t_2, \ldots, t_k$, is legal if and only if there are directed edges (v_i, v_{i+1}), and $t_{i+1} - t_i \geq p(v_i, v_{i+1})$ for $i = 1, 2, \ldots, k-1$. The only exception is that v_{i+1} can also be the source and v_i the sink vertex. In this case if there exists some vertex v_j with $j < i$ in the sequence such that v_j is also the source vertex, then $t_{i+1} - t_j >= P(T)$ must be additionally satisfied. The real-time constraints require that the job generated by triggering vertex v_i, where $i = 1, 2, \ldots, k$, be assigned the processor for $e(v_i)$ amount of time within the time interval $(t_i, t_i + d(v_i)]$.

Figure II illustrates an example of a recurring real-time task. In this task, vertex v_3, for instance, has an execution requirement $e(v_3) = 6$, which must be met within 10 time units (its deadline) from its triggering time. The edge (v_1, v_3) has been labeled 10, which implies that the vertex v_3 can be triggered only after a minimum of 10 time units from the triggering of v_1 (i.e., the minimum intertriggering separation time). Edges (v_1, v_2) and (v_1, v_3) from vertex v_1 imply that either v_2 or v_3 can be triggered after v_1. The period of the task (the minimum time interval between two consecutive triggerings of the source vertex) is 50.

A. Task Sets and Schedulability Analysis

A task set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ consists of a collection of task graphs, the vertices of which can get triggered independently of each other. A triggering sequence for such a task set $\mathcal{T}$ is legal if and only if for every task graph T_i, the subset of vertices of the sequence belonging to T_i constitute a legal triggering sequence for T_i. In other words, a legal triggering sequence for $\mathcal{T}$ is obtained by merging together (ordered by triggering times, with ties broken arbitrarily) legal triggering sequences of the constituting tasks.

The schedulability analysis of a task set $\mathcal{T}$ is concerned with determining whether the jobs generated by all possible legal triggering sequences of $\mathcal{T}$ can be scheduled such that their associated deadlines are met. In this paper, we assume earliest deadline first (EDF) based preemptive uniprocessor schedules. However, all results presented here can be extended to other scheduling policies (e.g., fixed-priority) as well.

A *demand bound criteria*-based schedulability analysis states that a task set $\mathcal{T}$ is schedulable if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $0 < t \leq t_{\max}$. For any given task T, the function $T.dbf(t)$ is referred to as the *demand-bound function*. It takes as an argument a positive real number t and returns the maximum possible cumulative execution requirement of jobs that can be legally generated by T and which have their ready-times and deadlines both within a time interval of length t. It can be proved that

$$t_{\max} = \frac{\sum_{T \in \mathcal{T}} 2E(T)}{1 - \sum_{T \in \mathcal{T}} \frac{E(T)}{P(T)}}$$

where $E(T)$ is the maximum cumulative execution requirement arising from a sequence of vertices on any path from the source to the sink vertex of the task graph T (see [4] for details). The schedulability analysis algorithm therefore involves two steps.

(i) Computing $T.dbf(t)$ for all $t \leq t_{\max}$ and $T \in \mathcal{T}$, and
(ii) Checking that $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$, $\forall\, 0 < t \leq t_{\max}$.

For the recurring real-time task model, it turns out that computing $T.dbf(t)$ for any t is NP-hard (see [7]) and therefore forms the computationally intensive kernel of the schedulability analysis algorithm. In what follows, we outline a dynamic programming (DP) based algorithm for computing $T.dbf(t)$ for any task graph T and time interval length t. For details on this algorithm, we refer the reader to [8]. In Section IV, we describe our approach to reformulate this algorithm in order to implement it effectively using CUDA.

Algorithm 1 Computing $T.dbf(t)$ using dynamic programming

```
Require: Task graph $T'$, and a real number $t \geq 0$
 1: for $e \leftarrow 1$ to $nE$ do
 2:     $t_{1,e} \leftarrow \begin{cases} d(v_1) & \text{if } e(v_1) = e \\ \infty & \text{otherwise} \end{cases}$
 3:     $t^1_{1,e} \leftarrow t_{1,e}$
 4: end for
 5: for $i \leftarrow 1$ to $n-1$ do
 6:     for $e \leftarrow 1$ to $nE$ do
 7:         Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to $v_{i+1}$
 8:         $t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) \\ \quad + d(v_{i+1}) \mid j = 1, 2, \ldots, k\} \text{ if } e(v_{i+1}) < e, \\ d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \text{ and } \infty \text{ otherwise} \end{cases}$
 9:         $t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
10:     end for
11: end for
12: $T.dbf(t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$
```

B. *Computing the* demand-bound function

In this section we present a dynamic programming algorithm for computing the demand-bound function $T.dbf(t)$ for any task graph T. For any task graph T, computing the value of $T.dbf(t)$ for some (large) value of $t \leq t_{\max}$ might involve multiple traversals (loops) through the task graph. It was shown in [4] that if for a task graph T, $T.dbf(t)$ is known for all "small values" of t then it is possible to calculate, from these, the value of $T.dbf(t)$ for *any* t. "Small values" of t for a task graph T are those for which the sequence of vertices that contribute towards computing $T.dbf(t)$ contain the source vertex at most once. The value of $T.dbf(t)$ for larger values of t is made up of some multiple of $E(T)$ plus $T.dbf(t')$ where t' is "small" in the sense described above. It follows that $T.dbf(t)$ for any t can be computed as follows (for a more detailed description, refer to [4])

$$T.dbf(t) = \max\{\lfloor t/P(T) \rfloor E(T) + T.dbf(t \bmod P(T)), \\ (\lfloor t/P(T) \rfloor - 1)E(T) + T.dbf(P(T) + t \bmod P(T))\} \qquad (1)$$

To compute $T.dbf(t)$ for "small" values of t, [4] constructs a new task graph by taking two copies of the task graph of T and adding an edge from the sink vertex of the first graph to the source vertex of the second and finally replacing the source vertex of the first with a "dummy" vertex with execution requirement and deadline equal to zero. The intertriggering separations on all edges outgoing from this source vertex is also made equal to zero. $T.dbf(t)$ for all values of t are then calculated by enumerating all possible paths in this new graph. For arbitrary task graphs, this incurs a computation time which is exponential in the number of vertices in the task graph.

We first outline an algorithm for computing the demand-bound function of a task graph for "small values" of t. Using this, we then compute the demand-bound function for any value of t as explained above.

Given a task graph T, let T' denote the graph formed by joining two copies of T by adding an edge from the sink vertex of the first graph to the source vertex of the second and replacing the source vertex of the first copy by a "dummy" vertex. The newly added edge is labeled with an intertriggering separation of $p = d(v_{sink})$. Now we give a pseudo-polynomial algorithm based on dynamic programming, for computing $T'.dbf(t)$ for values of t that do not involve any looping through T', i.e., we consider only "one-shot" executions of T'. Let there be n vertices in T' denoted by $v_1, v_2, \ldots, v_n$, and without any loss of generality we assume that there can be a directed edge from v_i to v_j only if $i < j$. Following our notation above, associated with each vertex v_i, is its execution requirement $e(v_i)$ which here is assumed to be integral, and its deadline $d(v_i)$. Associated with each edge (v_i, v_j) is the minimum intertriggering separation $p(v_i, v_j)$.

Let $t_{i,e}$ be the minimum time interval within which the task T' can have an execution requirement of exactly e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_1, v_2, \ldots, v_i\}$, if all the triggered vertices are to meet their respective deadlines. Let $t^i_{i,e}$ be the minimum time interval within which a sequence of vertices from the set $\{v_1, v_2 \ldots, v_i\}$, and ending with the vertex v_i, can have an execution requirement of exactly e time units, if all the vertices have to meet their respective deadlines. Lastly, let $E = \max_{i=1,i=2,\ldots,n} e(v_i)$. Clearly, nE is an upper bound on $T'.dbf(t)$ for any $t \geq 0$ for one-shot executions of T'. It can be shown by induction that Algorithm 1 correctly computes $T'.dbf(t)$, and has a running time of $O(n^3 E)$.

We would like to emphasize that $T'.dbf(t)$ values obtained from Algorithm 1 are essentially the values of $T.dbf(t)$ for "small" values of t. The notation T' was used in the above explanation in order to explicitly denote the graph formed by joining two copies of T. Thus, T' is an input to the Algorithm 1 and the output is $T.dbf(t)$ for "small" values of t. Following this, one may now use Equation 1 to calculate $T.dbf(t)$ for any value of t.

III. CUDA

In this section, we provide a brief description of CUDA [15]. CUDA abstracts the GPU as a powerful multi-threaded coprocessor capable of accelerating data-parallel, computationally intense operations. The data parallel operations, which are similar computations performed on *streams* of data, are referred to as *kernels*. Essentially, with its programming model and hardware model, CUDA makes the GPU an efficient streaming platform. Below, we discuss CUDA's programming and hardware model, followed by a short insight into the memory access latencies of the GPU.

A. Programming Model

In CUDA, *threads* execute data parallel computations of the kernel and are clustered into blocks of threads referred to as thread blocks. These thread blocks are further clustered into grids. During implementation, the designer can configure the number of threads that constitute a block as well as the number of blocks that constitute a grid. Each thread inside a block has its own registers and local memory. The threads in the same block can communicate with each other through a memory space shared among all the threads in the block and referred to as *Shared Memory*. However, an explicit communication and synchronization between threads belonging to different blocks is only possible through GPU-DRAM. GPU-DRAM is the dedicated DRAM for the GPU in addition to DRAM of the CPU. It is divided into *Global Memory*, *Constant Memory* and *Texture Memory*. We note that the *Constant* and *Texture Memory* spaces are read-only regions whereas *Global Memory* is a read-write region. Figure 3 illustrates the above described CUDA programming model. Note that in contrast to the GPU-DRAM the *Shared Memory* region is a on-chip memory space.

B. Hardware Model

CUDA hardware architecture is implemented as a set of SIMD (Single-Instruction-Multiple-Data) multiprocessors with on-chip memory. Each of these multiprocessors also consist a set of registers. The thread blocks (described in the previous subsection) are executed on these multiprocessors such that each multiprocessor executes one or more thread blocks through time slicing. However, each thread block is processed by a single multiprocessor in order to facilitate communication

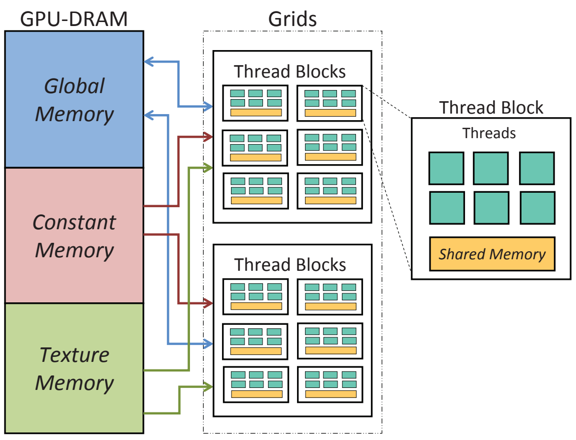

Fig. 3. CUDA programming model.

between different threads in a block through on-chip memory. Thus, the on-chip memory of a multiprocessor forms the *Shared Memory* space of the thread block and is typically in the order of KB.

C. Memory Access Latencies

A typical memory instruction in CUDA issued by a multiprocessor consumes 4 clock cycles. However depending upon the memory space where the memory location that is being accessed resides, there will be additional latencies. In case the memory location being accessed resides in GPU-DRAM, i.e., either in *Global*, *Texture* or *Constant Memory* spaces, the memory instruction consumes an additional 400 to 600 cycles. On the other hand, if the memory location resides on-chip in the registers or *Shared Memory*, then there will be almost no additional latencies in the absence of memory access conflicts. These additional latencies might obscure the speedups that can be achieved due to parallelization and hence the on-chip shared memory must be judiciously exploited.

IV. Schedulability Analysis using CUDA

In order to accelerate the schedulability analysis algorithm (Algorithm 1) described in Section II using CUDA, there are two broad challenges. Firstly, we need to identify and isolate the data parallel computation of the algorithm so that they may be compiled as the *kernels*. These kernels must be then mapped to CUDA thread blocks. Secondly, one has to efficiently exploit the on-chip *Shared Memory* to enhance the achievable speedups. In light of these two challenges, we now provide a systematic implementation of Algorithm 1.

As mentioned above, our first goal is to identify the data parallel portions (kernels) that can be computed in a SIMD fashion using CUDA threads. The kernels must not have any data dependencies (on each other) because they will be executed by threads running in parallel. Towards this, we first identify the data dependcies in Algorithm 1. Algorithm 1 (lines $6-10$) essentially builds a dynamic programming (DP) matrix. The $i+1$-th row in the matrix corresponds to vertex v_{i+1} of the task graph described in Section II. Each of the cells in row $i+1$ consists of $t_{i+1,e}$ and $t^{i+1}_{i+1,e}$ values where $e = 1, 2, \ldots, nE$. According to Algorithm 1 (line 8), the computation of these values in the cells of the $i+1$-th row depends *only* on the values present in the *previously* computed rows. This implies the values of the cells of the *same* row in the DP-based matrix can be computed independently of each other by using different CUDA threads in a SIMD fashion. Therefore, we segregate this task (lines 8 and 9 of Algorithm 1) as the *kernel* of our CUDA implementation.

We store the DP-matrix in *Global Memory* space, i.e., in GPU-DRAM. Note that we use *Global Memory* space instead of *Constant* or *Texture Memory* because *Constant* and *Texture Memory* are read-only regions. During the computation of our DP-based matrix we need to perform both read (to fetch values from previously computed rows) and write (to update the DP-matrix with the values of the row computed in the current iteration) operations which can only be done explicitly with *Global Memory*. Also, note that, we have not used the on-chip *Shared Memory* because the size of the *Shared Memory* is typically quite small (see Section III) and the entire DP-matrix cannot fit into it.

However, the on-chip *Shared Memory* can be exploited to store other frequently accessed data structures. To identify such data structures, we once again focus on the kernel operations of our algorithm (lines 8 and 9 in Algorithm 1). We note that the computation of $t_{i+1,e}^{i+1}$ and $t_{i+1,e}$ values of vertex v_{i+1} (i.e., the $i+1$-th row of the DP-matrix) needs certain values from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$, where v_{i+1} has an incoming edge from each of these vertices. Let us denote the tuple $\{p(v_{i_j}, v_{i+1}), d(v_{i_j})\}$ as $\Omega_{i,j}$. Thus, from the lines 8 and 9 of Algorithm 1, the computation $i+1$-th row of the DP-matrix requires the values $\Omega_{i,1}, \Omega_{i,2}, \ldots, \Omega_{i,k}$.

The set, $\{\Omega_{i,2}, \ldots, \Omega_{i,k}\}$, is essentially a subset of the overall specification of the task graph. Also, in iteration $i+1$ of computing the DP-matrix this set of required data structure remains constant, i.e., information about the other parts of the task graph is not required. This set changes only at the next iteration because it corresponds to a different vertex which might have a different set of incoming edges. This observation provides an opportunity to significantly reduce the GPU based execution times by loading these values $\{\Omega_{i,2}, \ldots, \Omega_{i,k}\}$ to the on-chip *Shared Memory* at the beginning of each iteration. Compared to the DP-matrix, this set of values is much smaller and can fit into the on-chip shared memory. Figure 4 illustrates our scheme of prefetching the required data structure from *Global Memory* to *Shared Memory* at the start of each iteration. The figure shows a thread block (which consists of 32 threads) fetching the required data from the *Global Memory* at the $i+1$-th iteration.

We recall from Section III that the on-chip memory is shared only between the threads within a single block. Hence, configuring the thread blocks to an appropriate size is also important to effectively exploit the GPU on-chip memory. For example, on one hand, if we choose a very small thread block size, then the computation of each row in our DP-based matrix will involve lot of thread blocks. However, only the threads within a thread block share the same chunk of on-chip memory. This implies that data from the *Global Memory* to *Shared Memory* will have to be transfered for a large

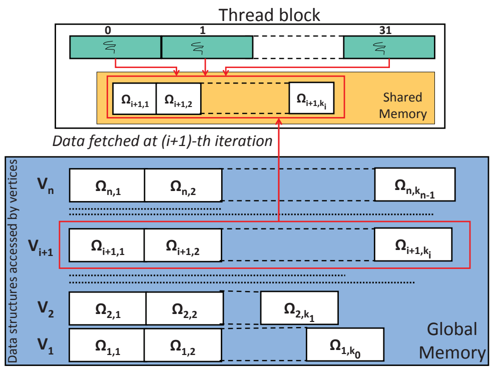

Fig. 4. Accessing data from *Shared Memory*. Since on-chip *Shared Memory* space is limited, data structures are selectively prefetched at each iteration.

number of thread blocks, inspite of the fact that, all the threads in a single iteration need the same data structures - $\{\Omega_{i,2}, \ldots, \Omega_{i,k}\}$, as described above. This in turn obscures the speedups that can be achieved. On the other hand, one can choose a thread block of very large size which has a lot of threads in each block. In this case each thread block will be executed on a single multiprocessor via time slicing because the number of processors on multiprocessor are typically around 8, while there are thousands of threads running in parallel. This time slicing will also inhibit the acceleration that can be achieved. To strike the right balance with the size of the thread block, we perform experiments with different thread block sizes. Note that CUDA allows thread block sizes only as powers of 2. Hence the design space for exploration is not huge and it is possible to find the right size in reasonable time. Moreover, after a threshold size the performance deteriorates or remains constant and hence it is not necessary to experiment with larger sized thread blocks beyond this threshold.

V. Experimental Results

For our experiments, we compared three different implementations of the the schedulability analysis algorithm — on the CPU (as described in Algorithm 1), on GPU without using shared memory and on GPU while exploiting shared memory (as described in Section IV). We randomly generated synthetic task graphs consisting of 10, 20, 30, 40 and 50 vertices respectively. The value of E which represents the maximum possible execution time for a vertex was set to $10,000$. It may be noted that the execution requirement associated with any vertex of a graph is expressed in terms of *time units*. Such time units depend on the application at hand and might denote milliseconds, microseconds, or even the number of clock cycles of the processor on which the task graphs are required to execute. Hence, experiments with values of E like $10,000$ are realistic.

All the experiments were conducted on a machine with 3.0 GHz Intel Pentium 4 CPU and 1 GB RAM running Windows XP. The machine was equipped with a nVIDIA GeForce 8800 GTX GPU, which was used for our GPU based implementations. The code has been implemented and compiled with release mode in Microsoft Visual Studio 2005 which had CUDA Toolkit 1.1 integrated into it.

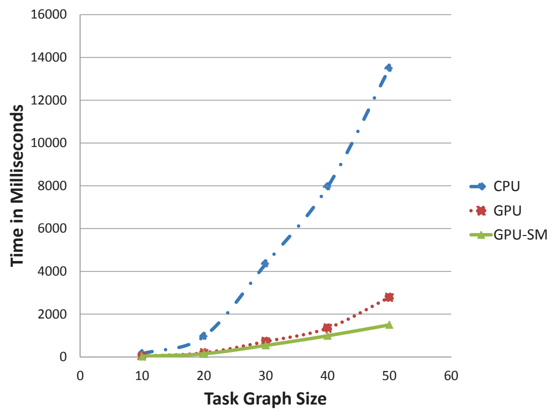

Fig. 5. Comparison of CPU, GPU and GPU-SM execution times.

In our experiments, we measured the execution time of the three different implementations for each of the tasks graphs. Table I presents the exact values of these measurements for task graphs of different sizes. The first column corresponds to the execution time on CPU referred to as 'CPU Time'. The second column corresponds to the execution time on GPU without any optimizations involving exploitation of *Shared Memory* space and is referred to as 'GPU Time'. The third column referred to as 'GPU-SM Time' corresponds to the execution time on GPU as well but with the exploitation of *Shared Memory* as described in Section IV. Figure 5 provides a visual representation of these speedups. As mentioned in Section IV, we experimented with different sizes of thread blocks. We observed a performance enhancement (i.e., speed ups) as we increased the thread block size upto 512. However, with larger thread blocks (1024 and 2048) there were no further speed ups. Thus, 512 was the best size for the thread blocks and the 'GPU-SM Time' reported here correspond to experiments with thread block size 512. From our results, we observe that as we progressively increase the task graph size from 10 to 50 the GPU implementation without *Shared Memory* provides a speedup of $5.2\times$ on average and a maximum of $6\times$ compared to CPU. The GPU implementation with *Shared Memory*, i.e., GPU-SM attains a maximum speedup of $9\times$ (for a task graph with 50 vertices) and an average of $7.4\times$ speedup compared to the execution times on CPU. It is noteworthy that the GPU-SM implementation provides upto $1.86\times$ acceleration compared to a GPU implementation without shared memory utilization.

From these results it is evident that as the size of the task graph increases, memory access latencies can degrade the speedups that can be attained by GPU implementation. This can be noticed from the measurements corresponding to task graphs of sizes 10 and 50. In case of graph with 10 vertices, the GPU and GPU-SM implementations provide $4.2\times$ and $5.3\times$ speedups respectively. However, in the case graph with 50 vertices these speedup values are $4.8\times$ and $9\times$, respectively for GPU and GPU-SM. These numbers, therefore, indicate that for GPU-based implementations, exploiting on-chip *Shared Memory* is an effective method to overcome performance degradations caused by memory access latencies.

It may be noted that all our implementations were stand-alone C codes and they did not make use of any graphical interfaces for specifying the task graphs. Instead, the code was specifically optimized for running the schedulability analysis. In practice, a design tool supporting schedulability analysis would be more involved. More specifically, the task graphs might be integrated with other application-specific data structures that are not optimized for the schedulability analysis algorithm. In such cases, the speedups obtained by our interactive schedulability analysis might be considerably higher compared to the results reported here.

Size	CPU Time (milliseconds)	GPU Time (milliseconds)	GPU-SM Time (milliseconds)
10	147.683716	34.910538	28.038548
20	955.779358	192.749374	146.997284
30	4346.823242	720.375671	537.149719
40	7945.673828	1344.817139	987.577637
50	13490.95606	2783.330078	1496.557251

TABLE I
EXECUTION TIMES OF CPU, GPU AND GPU-SM IMPLEMENTATIONS.

VI. Conclusion

In this paper, we presented a technique to implement a computationally expensive schedulability analysis algorithm on GPUs. Our proposed method can effectively utilize the on-chip shared memory, which was not possible with previously proposed techniques.

References

[1] P. K. Agarwal, S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian. (Book Chapter). Streaming geometric optimization using graphics hardware. In *European Symposium on Algorithms*, pages 544–555. Springer Berlin / Heidelberg, 2003.
[2] A. Ailamaki, N. K. Govindaraju, S. Harizopoulos, and D. Manocha. Query co-processing on commodity processors. In *VLDB*, 2006.
[3] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *ECRTS*, 2004.
[4] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.
[5] U. D. Bordoloi and S. Chakraborty. Interactive schedulability analysis. *ACM Trans. Embedded Comput. Syst.*, 7(1), 2007.
[6] U. D. Bordoloi and S. Chakraborty. GPU-based acceleration of system-level design tasks. *International Journal of Parallel Programming*, 38(3-4), 2010.
[7] S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. In *Proc. 7th International Workshop on Algorithms and Data Structures (WADS)*, Lecture Notes in Computer Science 2125, pages 38–49, 2001.
[8] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *RTSS*, 2002.
[9] D. Chatterjee, A. De Orio, and V. Bertacco. GCS: High-performance gate-level simulation with GP-GPUs. In *DATE*, 2009.
[10] K. Gulati and S. P. Khatri. Towards acceleration of fault simulation using graphics processing units. In *DAC*, 2008.
[11] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.
[12] B. A. Jose S. K. Shukla M. Nanjundappa, H. D Patel. Scgpsim: A fast SystemC simulator on gpus. In *ASP-DAC*, 2010.
[13] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.
[14] K. Mueller and F. Xu. Ultra-fast 3d filtered backprojection on commodity graphics hardware. In *International Symposium on Biomedical Imaging*, pages 571–574, 2004.
[15] NVIDIA. CUDA Programming Guide version 1.0, 2007.
[16] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley, 2006.