XSpeed: Accelerating Reachability Analysis on Multi-core Processors

Rajarshi Ray[1⋆], Amit Gurung[1], Binayak Das[1],
Ezio Bartocci[2], Sergiy Bogomolov[3], and Radu Grosu[2]

[1] National Institute of Technology Meghalaya, India
[2] Vienna University of Technology, Austria
[3] Institute of Science and Technology Austria, Austria

Abstract. We present XSpeed a parallel state-space exploration algorithm for continuous systems with linear dynamics and nondeterministic inputs. The motivation of having parallel algorithms is to exploit the computational power of multi-core processors to speed-up performance. The parallelization is achieved on two fronts. First, we propose a parallel implementation of the support function algorithm by sampling functions in parallel. Second, we propose a parallel state-space exploration by slicing the time horizon and computing the reachable states in the time slices in parallel. The second method can be however applied only to a class of linear systems with invertible dynamics and fixed input. A GP-GPU implementation is also presented following a lazy evaluation strategy on support functions. The parallel algorithms are implemented in the tool XSpeed. We evaluated the performance on two benchmarks including an 28 dimension Helicopter model. Comparison with the sequential counterpart shows a maximum speed-up of almost 7× on a 6 core, 12 thread Intel Xeon CPU E5-2420 processor. Our GP-GPU implementation shows a maximum speed-up of 12× over the sequential implementation and 53× over SpaceEx (LGG scenario), the state of the art tool for reachability analysis of linear hybrid systems. Experiments illustrate that our parallel algorithm with time slicing not only speeds-up performance but also improves precision.

1 Introduction

Reachability analysis is a standard technique for safety verification, analysis and synthesis of continuous and hybrid systems. Since exact computation of reachable states is in general intractable, set-based conservative computation methods has been proposed in the past with different choice of sets [1,2][7,8,9][12][22]. The reachable states are represented as a collection of continuous sets ($\subset \mathbb{R}^n$) with a

[0] **Acknowledgements.** This work was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23, S11405-N23 and S11412-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

⋆ Corresponding author e-mail address: raj.ray84@gmail.com

symbolic representation of each individual set. Precision and scalability has been the two challenges with such set-based methods. The symbolic set representation plays a key role in deciding the efficiency of the reachability algorithm. Recently, algorithms using convex sets represented by support functions [12] and zonotopes [9] have shown promising scalability. Systems having dimension as large as 100 have been shown to be computed efficiently with support-function-based algorithms.

The advent of multi-core architectures and many-core parallel co-processors like graphics processing units (GPUs) have provided tremendous computing power at our disposal. In this work, our goal is to leverage these powerful parallel architectures to speed-up the performance of reachability analysis and also possibly the precision of analysis. There has been prior work on devising parallel algorithms for discrete state concurrent systems in order to speed-up their performance on multi-core machines. Parallel state-space-search algorithms in the model-checker SPIN has been proposed in [6][13]. A GP-GPU implementation of the algorithms in SPIN has been proposed in [3]. However, no prior work is known to us on parallel state-space exploration of continuous and hybrid systems except for [17] which presents some preliminary results.

In particular, we consider the support-function-based reachability algorithm and propose two parallel versions of it. The first, samples the support functions along the template directions in parallel. This algorithm could be applied to any system with linear dynamics and nondeterministic inputs ($\dot{x} = Ax(t) + u(t), u(t) \in \mathcal{U}, x(0) \in \mathcal{X}_0$). The second computes the reachable sets in slices of the time horizon. This algorithm can be applied to the class of linear systems whose dynamics A is invertible and the input set $\mathcal{U}$ is a point set. We also propose a GP-GPU implementation of the algorithm by following a lazy evaluation strategy. Our current GP-GPU implementation restricts $\mathcal{X}_0$ and $\mathcal{U}$ to be specified as hyperbox.

The organization of the paper is as follows. In Section 2, we present preliminaries on support functions and a reachability analysis algorithm using support functions. In Section 3, a parallel implementation scheme of the algorithm and our parallel state-space exploration algorithm is presented. We present our GPU implementation scheme in Section 4 to sample support functions in parallel in GPU cores. The experimental results are presented in Section 5 illustrating the achieved speed-up and precision. We conclude in Section 6.

2 Preliminaries

Since our work is focused on the support-function representation of compact convex sets, we recap the definition of support functions and template polytopes in Section 2.1. The reachability algorithm using support functions is discussed in Section 2.2.

2.1 Support Functions

Definition 1. *[18] Given a nonempty compact convex set $\mathcal{X} \subset \mathbb{R}^n$ the* support function *of $\mathcal{X}$ is a function $sup_{\mathcal{X}} : \mathbb{R}^n \to \mathbb{R}$ defined as:*

$$sup_{\mathcal{X}}(\ell) = max\{\ell \cdot x \mid x \in \mathcal{X}\} \tag{1}$$

where $\ell \cdot x$ is the scalar product of direction ℓ and vector x, that is, the projection of x on direction ℓ. A compact convex set $\mathcal{X}$ is uniquely determined by the intersection of the halfspaces generated by support functions in all possible directions $\ell \in \mathbb{R}^n$.

$$\mathcal{X} = \bigcap_{\ell \in \mathbb{R}^n} \ell \cdot x \leq sup_{\mathcal{X}}(\ell) \tag{2}$$

Definition 2. *Given the support function $sup_{\mathcal{X}}$ of a compact convex set $\mathcal{X}$ and a finite set of template directions $\mathcal{D}$, the template polytope of the convex set $\mathcal{X}$ is defined as:*

$$Poly_{\mathcal{D}}(\mathcal{X}) = \bigcap_{\ell \in \mathcal{D}} \ell \cdot x \leq sup_{\mathcal{X}}(\ell) \tag{3}$$

Proposition 1. *Given a polytope $\mathcal{X} = \{x \in \mathbb{R}^n \mid P \cdot x \leq Q\}$, the support function of $\mathcal{X}$ in the direction ℓ is the solution to the Linear Program (LP):*

$$sup_{\mathcal{X}}(\ell) = \begin{cases} \textit{maximize } \ell \cdot x \\ \textit{subject to:} \\ P \cdot x \leq Q \end{cases}$$

Proposition 2. *Given a hyperbox $\mathcal{H} = \{x \in \mathbb{R}^n \mid x \in [a_1, b_1] \times \ldots \times [a_n, b_n]\}$, the support function of $\mathcal{H}$ in the direction $\ell = (\ell_1, \ell_2, \ldots, \ell_n)$ is given by:*

$$sup_{\mathcal{H}}(\ell) = \sum_{i=1}^{n} \ell_i \cdot h_i \text{ , where } h_i = \begin{cases} a_i \textit{ if } \ell_i < 0 \\ b_i \textit{ otherwise} \end{cases}$$

where a_i and b_i are the lower and upper bound respectively of $\mathcal{H}$ in the dimension i.

2.2 Reachability Analysis using Support Functions

In this work, we consider continuous linear systems with constrained inputs and initial states. The dynamics of such systems is of the form:

$$\dot{x} = Ax(t) + u(t), u(t) \in \mathcal{U}, x(0) \in \mathcal{X}_0 \tag{4}$$

where $\mathcal{X}_0, \mathcal{U}$ is the set of initial states and the set of inputs given as compact convex sets, respectively.

We now discuss the algorithm proposed in [12] for computing reachable states using support functions. The algorithm discretizes time by a time step δ and

computes an over approximation of the reachable set in time horizon T by a set of convex sets represented by their support functions, as shown in Equation 5.

$$Reach_{[0,T]}(\mathcal{X}_0) \subseteq \bigcup_{i=0}^{N-1} (\Omega_i) \tag{5}$$

The convex sets Ω_i are given by the following equations:

$$\begin{aligned} \Omega_{i+1} &= \Phi_\delta \Omega_i \oplus \mathcal{W} \\ \Omega_0 &= CH(X_0, \Phi_\delta X_0 \oplus \mathcal{V}) \end{aligned} \tag{6}$$

where $\oplus$, CH stands for minkowski sum and convex hull operation over sets respectively, $\Phi_\delta = e^{\delta A}$ and $\mathcal{W}$, $\mathcal{V}$ are convex sets given as follows:

$$\begin{aligned} \mathcal{V} &= \delta\mathcal{U} \oplus \alpha\mathcal{B} \\ \mathcal{W} &= \delta\mathcal{U} \oplus \beta\mathcal{B} \end{aligned} \tag{7}$$

α, β are constants depending on $\mathcal{X}_0, \mathcal{U}$, δ and the dynamics matrix A. $\mathcal{B}$ is a unit ball in the considered norm.

The support function representation of Ω_i can be seen as an abstraction of its template polyhedra representations. A concretization can be obtained by computing template polyhedra approximations of Ω_i along a set of directions $\mathcal{D}$. Such concretization provides an efficient computation of intersection, plotting and other efficient operations over polytopes but at the expense of an approximation error depending on the number of template directions and the choice of directions.

The algorithm considers a set of bounding directions, say $\mathcal{D}$, to sample the support functions of Ω_i and obtains a set of template polyhedra $Poly_{\mathcal{D}}(\Omega_i)$ whose union over-approximates the reachable set.The support function of Ω_i is computed with the following equation obtained using the properties of support functions:

$$\begin{aligned} sup_{\Omega_{i+1}}(\ell) &= sup_{\Omega_i}(\Phi_\delta^T \ell) + sup_{\mathcal{W}}(\ell) \\ sup_{\Omega_0}(\ell) &= max\big(sup_{\mathcal{X}_0}(\ell), sup_{X_0}(\Phi_\delta^T \ell) + sup_{\mathcal{V}}(\ell)\big) \end{aligned} \tag{8}$$

Simplification of Equation 6 yields the following relation:

$$sup_{\Omega_i}(\ell) = sup_{\Omega_0}\big((\Phi_\delta^T)^i \ell\big) + \sum_{j=0}^{i-1} sup_{\mathcal{W}}\big((\Phi_\delta^T)^j \ell\big) \tag{9}$$

3 Parallel State-Space Exploration

In this section, we present two approaches to parallelize the support-functions-based reachability algorithms in Section 3.1 and Section 3.2.

3.1 Parallel Samplings over Template Directions

The LGG (Le Guernic Girard) scenario of the state of the art tool, SpaceEx [8] computes reachable states using a support-functions algorithm with the provision of templates in *box*, *octagonal* and *p uniform* directions. A box polyhedron has $2n$ directions $(x_i = \pm 1, x_k = 0, k \neq i)$ whereas an octagonal polyhedron has $2n^2$ directions $(x_i = \pm 1, x_j = \pm 1, x_k = 0, k \neq i, k \neq j)$, giving a more precise approximation. The support function algorithm scales well when the number of template directions is linear in the dimension n of the system. When computing finer approximations with directions quadratic in n, we trade-off precision for scalability.

The support-functions-based algorithm is easy to parallelize by sampling the template directions in parallel [12]. However, there are implementation challenges. In this work, we propose a multi-threaded implementation with a master thread spawning worker threads for every direction in the template set $\mathcal{D}$. The pseudocode of master thread is shown in Algorithm 1. A global support function matrix M having R rows and N columns is allocated to store the computed support functions by different worker threads, where R is the number of directions in $\mathcal{D}$ and $N = T/\delta$ is the number of iterations. Each worker thread t_i computes the support function samples along a direction $d(i)$ in parallel. The results by thread t_i are written to the row $M[i]$ resulting in no write contention among the threads. An entry $M(i, j)$ stores the support function of Ω_j in the ith direction in $\mathcal{D}$ as shown in lines 2-3 in Algorithm 2.

The master thread waits for all the worker threads to complete. After all the worker threads have finished, we have a template polytope $Poly_{\mathcal{D}}(\Omega_i)$ for every convex set Ω_i which is obtained from the support function as shown in lines 7-9 in Algorithm 1.

Algorithm 1 Pseudocode of Master Thread

```
1: procedure REACH-PARALLEL-MASTER(D,N)
2:     for all ℓ_i ∈ D do                                   ▷ Master Thread
3:         Spawn a thread t_i to sample sup of Ω_0 ... Ω_{N-1} along ℓ_i
4:     end for
5:     Wait for all threads to finish.
6:     R_approx ← ∅
7:     for i ← 0, N − 1 do
8:         P_D(Ω_i) ← ⋀_{j=1}^{|D|} d(j).x ≤ M(j, i)
9:         R_approx ← R_approx ⋃ P_D(Ω_i)
10:    end for                                              ▷ Reach[0, T] ⊆ R_approx
11: end procedure
```

Sampling with Thread Safe GLPK It can be seen that when the initial set $\mathcal{X}_0$ in a location dynamics in 4 is given as a polytope, the convex sets Ω_i in Equation 6 are also polytopes. Sampling the support function of a polytope is a linear programming problem. SpaceEx LGG scenario expects initial set $\mathcal{X}_0$ and

Algorithm 2 Pseudocode of Worker Thread

```
1: procedure REACH-PARALLEL-WORKER(M,N,ℓ,i)   ▷ Worker Thread   ▷ Each thread gets an id i ∈ [1, R]
2:     for j ← 0, N − 1 do
3:         M[i][j] ← sup_{Ω_j}(ℓ)
4:     end for
5: end procedure
```

$\mathcal{U}$ to be polytopes and samples their support function using the GLPK (GNU Linear Programming Kit) library [15], an open source and highly optimized linear programming solver library. In our parallel implementation we also assume initial and input sets to be polytopes and use GLPK package to solve their support functions. However, note that the GLPK library is not thread safe This is due to the fact that GLPK implementation uses thread shared data that suffers from race condition in multithreaded executions. To overcome this, we identified the thread shared data and made them thread local to ensure thread safety. The thread safe GLPK objects are used per thread to compute the support functions at different directions in parallel.

3.2 Parallel Exploration of Reachable States

In addition to the parallelization introduced in the previous section, we also propose another parallelization where threads compute reachable states in disjoint intervals of the time horizon in parallel. To bring in parallelism, our key idea is to compute the reachable states at distinct times in the time horizon and treat them as initials states for independent reachability computations. The reachable states from each of the initials states is then computed by an independent thread in parallel. For load balancing, the time horizon T is sliced into equal intervals of size $T_p = T/N$, N being the degree of parallelism. The limitation of this approach is that the input set $\mathcal{U}$ is assumed to be a point set and the dynamics matrix A is assumed to be invertible.

Proposition 3. *Given a linear dynamics of the form $\dot{x} = Ax(t)+u(t), u(t) \in \mathcal{U}$, if the input set $\mathcal{U} = v$ is a point set and the matrix A is invertible, the set of states reachable at time $t_i = iT_p$ is defined as:*

$$\mathcal{S}(t_i) = e^{AiT_p}.\mathcal{X}_0 \oplus A^{-1}(e^{AiT_p} - I)(v) \tag{10}$$

where I is the identity matrix.

Proof. Solving the differential equation $\dot{x} = Ax(t) + u(t), u(t) \in \mathcal{U}$ gives:

$$\begin{aligned} x(t) &= e^{tA}x_0 + \int_0^t e^{(t-y)A}u(y)dy \\ &= e^{tA}x_0 + \int_0^t e^{(t-y)A}vdy \\ &= e^{tA}x_0 + A^{-1}(e^{At} - I)(v) \end{aligned}$$

When $x(0) \in \mathcal{X}_0$, we apply minkowski sum to get:

$$\mathcal{X}(t) = e^{tA}\mathcal{X}_0 \oplus A^{-1}(e^{At} - I)(v)$$

Substituting $t = iT_p$:

$$\mathcal{X}(i(T_p)) = \mathcal{S}(t_i) = e^{A(iT_p)}.\mathcal{X}_0 \oplus A^{-1}(e^{A(iT_p)} - I)(v)$$

□

Let $\Phi_1 = e^{A(iT_p)}$ and $\Phi_2 = A^{-1}(e^{A(iT_p)} - I)$, the support function of the $\mathcal{S}(t_i)$ is given by:

$$sup_{\mathcal{S}(t_i)}(\ell) = sup_{\mathcal{X}_0}(\Phi_1^T\ell) + sup_v(\Phi_2^T\ell) \tag{11}$$

The reachable states in each time interval $I_i = [iT_p, (i+1)T_p]$ starting from states $x \in S(t_i)$ is defined as $R(S_i)$ and can be computed sequentially using Equation 6. Computation of $R(S_i)$ can also be in parallel over the template directions as proposed in Section 3.1.

Proposition 4. *An approximation of the reachable states in time horizon T can be computed by the following relation:*

$$Reach_{[0,T]}(\mathcal{X}_0) \subseteq \bigcup_{i=0}^{N-1} R(\mathcal{S}_i) \tag{12}$$

Proof. $R(\mathcal{S}_i)$ is computed using Equation 6 with a discretization time step δ with $\mathcal{S}_i$ as the initial set. Since S_i gives the exact set of states reachable at time instant $t = iT_p$, the correctness argument shown in [12] guarantees that $Reach_{[I_i]}(\mathcal{X}_0) \subseteq R(\mathcal{S}_i)$. Therefore, we have:

$$Reach_{[0,T]}(\mathcal{X}_0) = \bigcup_{i=0}^{N-1} Reach_{[I_i]}(\mathcal{X}_0) \subseteq \bigcup_{i=0}^{N-1} R(S_i)$$

□

In Section 5 we show that computing the reachable states using Proposition 4 gives in some cases more precise results compared to the sequential algorithm in [11,12]. This is because the approximation error in the computation of Ω_0 in Equation 6 propagates in the sequential algorithm. In our algorithm, since we compute exact reachable states at partition time points in the time horizon and recompute $\Omega_0^{t_i}$ using them, the propagation of the error may diminish.

4 Sampling Support Functions in GPU

It can be observed from Equations 8 and 9 that the support function of Ω_i can be computed from the support function of $\mathcal{X}_0$, $\mathcal{V}$ and $\mathcal{W}$. The support function

of $\mathcal{V}, \mathcal{W}$ can be, in turn, computed from the support function of $\mathcal{U}$ and $\mathcal{B}$. Therefore, to compute the support functions of $\Omega_0, \ldots, \Omega_{N-1}$ along a direction ℓ, it suffices to compute the support function of $\mathcal{X}_0$, $\mathcal{U}$ and $\mathcal{B}$ along the directions $\ell, \Phi_\delta^T \ell, (\Phi_\delta^T)^2 \ell, \ldots, (\Phi_\delta^T)^N \ell$. Unlike the support function algorithm in [12] which computes the support functions iteratively using Equation 8, we propose to compute the support functions in a lazy fashion which involves delaying evaluation until all the directional arguments are computed. The support functions are then evaluated in parallel in lines 7-11, as shown in Algorithm 3.

Algorithm 3 Lazy evaluation of support functions

1: **procedure** EVAL-SUPPORT($\Omega[0 \ldots N-1]$, $\mathcal{X}_0$, $\mathcal{U}$, ℓ)
2: $D[0] \leftarrow \ell$; ▷ Computing directional arguments
3: **for** $i \leftarrow 1, N$ **do**
4: $D[i] \leftarrow \Phi_\delta^T \cdot \mathcal{D}[i-1]$
5: **end for**
▷ Computing support functions in parallel
6: Spawn thread T_i to evaluate i$^{\text{th}}$ loop iteration;
7: **for** $i \leftarrow 0, N$ **do**
8: $Sup\mathcal{X}_0[i] \leftarrow T_i.evalSup(\mathcal{X}_0, D[i])$
9: $Sup\mathcal{U}[i] \leftarrow T_i.evalSup(\mathcal{U}, D[i])$
10: $Sup\mathcal{B}[i] \leftarrow T_i.evalSup(\mathcal{B}, D[i])$
11: **end for**
12: Wait for all threads to finish.
▷ Computing support Functions of Ω_i
13: $sum \leftarrow 0$
14: $sup_{\Omega_0}(\ell) \leftarrow max\big(Sup\mathcal{X}_0[0], Sup\mathcal{X}_0[1] + \delta \cdot Sup\mathcal{U}[0] + \alpha \cdot Sup\mathcal{B}[0]\big)$
15: **for** $i \leftarrow 1, N-1$ **do**
16: $p \leftarrow max\big(Sup\mathcal{X}_0[i], Sup\mathcal{X}_0[i+1] + \delta \cdot Sup\mathcal{U}[i] + \alpha \cdot Sup\mathcal{B}[i]\big)$
17: $sum \leftarrow sum + \delta \cdot Sup\mathcal{U}[i-1] + \beta \cdot Sup\mathcal{B}[i-1]$
18: $sup_{\Omega_i}(\ell) \leftarrow p + sum$
19: **end for**
20: **end procedure**

Observe that we need to run the same support function evaluation routine for a convex set but on different directions in parallel. We build upon this observation and propose to compute the support functions in SIMT (Single Instruction Multiple Threads) parallel architecture wherein multiple instances of a procedure execute in parallel but on different data. The modern day GPU (Graphics Processing Unit) architectures are SIMT in nature and we therefore propose to offload the support function computations to the GPU. A brief introduction to GPU architecture and CUDA programming model is given in Section 4.1.

4.1 CUDA Programming Model

We now briefly present the GPU hardware architecture and the programming model used to implement our parallel algorithms. As illustrated in Fig. 1, the

GPU architecture consists of a scalable array of N multithreaded Streaming Multiprocessors (SMs), made up of M Stream Processor (SP) cores. Each core is equipped with a fully pipelined integer arithmetic logic unit (ALU) and a floating point unit (FPU) that executes one integer or floating point instruction per clock cycle. In our experiments we have used the NVIDIA GeForce GTX 670 having 7 SMs and 192 SPs for each SM, for a total of 1344 SPs.

The NVIDIA vendor provides also a special Application Programming Interface (API) called Compute Unified Device Architecture (CUDA) that facilitates the developing of efficient applications tuned for NVIDIA GPUs. CUDA extends the C and the FORTRAN languages with special keywords and language primitives that are suitable to achieve a high-performance hardware-based multithreading. We have implemented our parallel algorithms using the C extension.

The CUDA programming model consists in using thousands of lightweight threads arranged into one- to three-dimensional thread blocks. A thread executes a function called the *kernel* that contains the computations to be run in parallel using a GPU device. A CUDA program starts running in the Central Processing Unit (CPU) referred as the *host*. Whenever a kernel is launched from the host code, the execution continues then in the GPU. The max number of threads running a kernel is fixed at the launching time (this limitation has some exceptions in the modern GPU cards supporting dynamic parallelism).

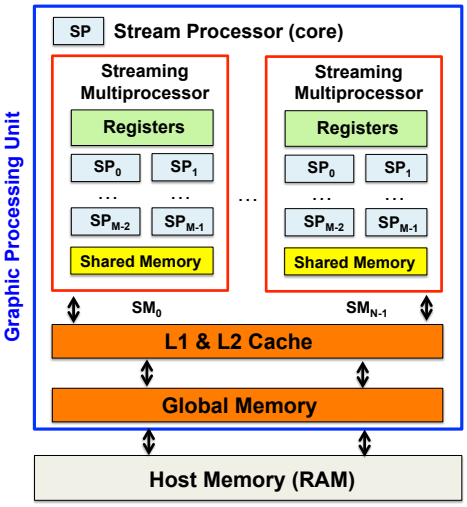


Fig. 1. GPU Architecture

Each thread is assigned to a SP and each thread block is processed by a SM. The thread execution model in CUDA is the Single Instruction Multiple Threads (SIMT). SIMT differs from the classical Single Instruction Multiple Data (SIMD) by the fact that the threads sharing the same instruction address and running synchronously are organised within a thread block into groups of 32 threads called *warps*. In a warp, the divergence of the threads execution in different branches due to *if-then-else* constructs, reduces considerably the level of parallelism and indeed degrades the performance of the kernel execution.

Threads can access different types of memory and their judicious use is key to performance. The most general is the off-chip global memory, to which all threads can read and write. Also the host can read and write the global memory and so this memory is usually used as a way of communication between the host and the GPU device. The global memory has slow performances and it is very important to access it in a coalesced way using a single memory transaction

of 32, 64, or 128 bytes. The two caches *L1* and *L2* shown in Fig. 1 mitigates this bottleneck by storing copies of the data most frequently accessed in the global memory. Significantly faster levels of memory are available within an SM, including 32-64KB of on-chip *registers* partitioned among all threads. As such, using a large number of registers within a CUDA kernel will limit the number of threads that can run concurrently. Finally, *local memory* is invoked when a thread runs out of available registers. In addition, each SM has a *shared memory* region (16-48KB). This level of memory, which can be accessed nearly as quickly as the registers, facilitates communication between threads and can be used as a programmer-controllable memory cache.

Threads located in the same thread block can cooperate in several ways. They can insert a synchronization point into the kernel, which requires all threads in the block to reach that point before execution can continue. They can also share data during execution. In contrast, threads located in different thread blocks cannot synchronize each other and they essentially operate independently. Although a small number of threads or blocks can be used to execute a kernel, this arrangement would not fully exploit the computing potential of the GPU.

4.2 Computing Support Functions of Polytopes in GPU

As discussed in Section 2, when $\mathcal{X}_0$ and $\mathcal{U}$ are polytopes, their support function evaluation is equivalent to solving a linear program (LP). The Simplex algorithm [4,5] is a well known and efficient procedure to solve LPs in practice. There is previous work on implementing the simplex algorithm on CUDA executing in a CPU-GPU heterogeneous system. An efficient implementation of the revised simplex method over a CPU-GPU system is shown in [21]. A multi-GPU implementation of the simplex procedure is reported in [14]. However, the reported results shows speed-up compared to sequential CPU implementation only when the size of LP is at least 500×500 (500 variables, 500 constraints). The reason why performance is poor for small size LPs is the CPU-GPU memory transfer latency to copy the simplex tableau and therefore the time gain due to parallelization is predominant over the CPU-GPU memory transfer latency only for large instances of LP. Since the benchmarks we know are of dimension much smaller than 500, we did not go for a simplex-algorithm implementation in GPU.

4.3 Computing Support Functions of Hyperbox in GPU

When the initial set $\mathcal{X}_0$ and input set $\mathcal{U}$ are given as hyperboxes, which are special cases of polytopes, their support function can be computed using Proposition 2 instead of solving an LP with simplex algorithm. This also avoids expensive memory transfer of simplex tableau, a data structure used in the simplex algorithm, from CPU to GPU. Building on this observation, we implemented a CUDA procedure to compute the support function of a hyperbox. There are challenges to have speed-up derived from GPU due to issues like warp divergence, memory transfer latency and GPU occupancy. We map a CUDA block to compute support function along a sampling direction. Since CUDA blocks

are scheduled to SMs, this ensures that all the GPU SMs are utilized when the number of sampling directions are more than the SMs in the GPU. Our block is one dimensional containing only 32 threads (warp size) since instructions in GPU are scheduled per warp which is a collection of 32 threads. The number of threads per block is kept to 1 warp-size since the task of computing the support function of a Hyperbox is lightweight. The maximum number of support function evaluation tasks that can be performed in parallel is limited by the number of directions that can be transferred to the GPU global memory. We attempt maximum parallelism by offloading tasks in batches of maximum possible size. The pseudocode of the GPU offloading routine is shown in Algorithm 4.

Algorithm 4 Offloading of Tasks in Batches to GPU

procedure GPU-OFFLOAD($\mathcal{H}$, $\mathcal{D}$, $numDirections$, Res)
  $gpuMemsize \leftarrow getGlobalMemsize()$;
  $sizePerDirection \leftarrow \mathcal{D}[0].memsize()$;
  $memsize \leftarrow numDirections * sizePerDirection$
  if memsize > gpuMemsize **then**
    $totalBatches \leftarrow ceil(memsize/gpuMemsize)$;
    $batchSize \leftarrow floor(gpuMemsize/sizePerDirection)$;
    $\mathcal{D}' \leftarrow malloc(sizePerDirection * batchSize * sizeof(double))$;
    $cur \leftarrow 0$;
    while cur $\leq$ totalBatches **do**
      $\mathcal{D}' \leftarrow$ copy $\mathcal{D}$ from $batchSize * cur$ to $batchSize * (cur + 1) - 1$;
      $gpuKernel <<< batchSize, 32 >>> (\mathcal{H}, \mathcal{D}', Res)$;
      $cur \leftarrow cur + 1$;
    end while
  else
    $batchSize \leftarrow numDirections$;
    $gpuKernel <<< batchSize, 32 >>> (\mathcal{H}, \mathcal{D}, Res)$;
  end if
end procedure

5 Experiments

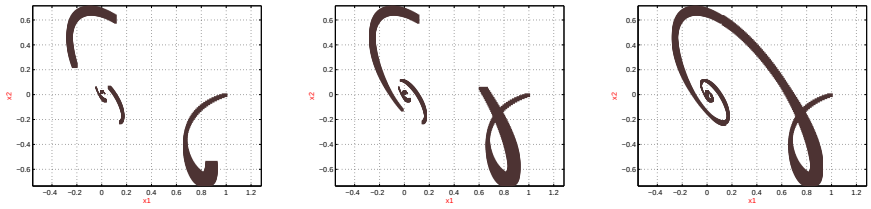
The parallel algorithms are implemented as part of the tool XSpeed including the CUDA implementation. To measure the performance of our parallel algorithms, we experiment on two benchmarks and compare our performance with the SpaceEx's (LGG) scenario [8] and with an optimized implementation of the support function algorithm in [12].

5.1 Five Dimensional System

We consider a five dimensional linear continuous system as a benchmark from [10]. Since we require the inputs set $\mathcal{U}$ to be a point set for our parallel state-space

exploration algorithm, we consider $\mathcal{U}$ = (0.01,0.01,0.01,0.01,0.01). We consider the initial set X_0 as a hyperbox with sides 0.02 centered at (1,0,0,0,0). For the matrix A, the reader may refer to [10].

Figure 2 illustrates the parallel exploration with slicing the time horizon. The figure shows that a time horizon of 5 units is sliced into five intervals each of size 1 unit. Five threads compute the reachable sets in parallel starting from initial sets $\mathcal{S}(t = 0)$,$\mathcal{S}(t = 1)$,$\mathcal{S}(t = 2)$,$\mathcal{S}(t = 3)$ and $\mathcal{S}(t = 4)$.



(a) Reachable states computed by individual threads in 0.5 time unit

(b) Reachable states computed by individual threads in 0.75 time unit

(c) Reachable states computed by individual threads in 1 time unit

Fig. 2. Illustrating Parallel State-Space Exploration in Sliced Time Horizon

5.2 Helicopter Controller

To measure the performance on a high dimensional system, we consider the benchmark of helicopter controller from [8][20]. This benchmark models the controller of a Westland Lynx military helicopter with 8 continuous variables. The controller is a 20 variables LTI system and the control system has 28 variables in total. We consider the initial set $\mathcal{X}_0$ to be a hyperbox and the input set $\mathcal{U}$ to be the origin $\{0\}$.

Table 1 shows the performance speed-up in computing reachable states with our parallel direction samplings compared to the sequential support-function algorithm in a 4 core and 6 core machine with hyper-threading, namely Intel Core i7-4770, 3.40GHz, 8GB RAM and Intel Xeon CPU E5-2420, 1.2Ghz, 46.847 GB RAM respectively. The results are an average of 10 runs for a time horizon of 5 units and a time step of $1.7e-3$ units. A speed-up of almost $7\times$ is observed for the Helicopter model. The gain in CPU utilization shows how our parallel implementation exploits the power of multicore processors effectively.

Figure 3 shows the speed-up obtained with the parallel state-space exploration with octagonal directions and time step of 0.0048 on Intel Core i7-4770, 4 core, 8 threads, 3.40GHz, 8GB RAM processor. The results from the SpaceEx tool are obtained by running the executable available at `http://spaceex.imag.fr/` on the same machine, with same parameters. We show that selecting the

right partition size is important to obtain optimal speed-up. Partitioning beyond a limit though give us high precision but degrades the performance as the threading overhead outruns the performance gain due to parallelism. The threshold depends on the number of cores in the underlying multi-core architecture. Figure 4 shows the gain in precision with box directions and time step of 0.01 and 0.0048 respectively for the five dimensional and the Helicopter benchmark. The gain in precision is because the time sliced algorithm computes exact reachable states at time points in the time horizon and diminishes the propagation of approximation error resulting from the computation of Ω_0 in the support-function algorithm.

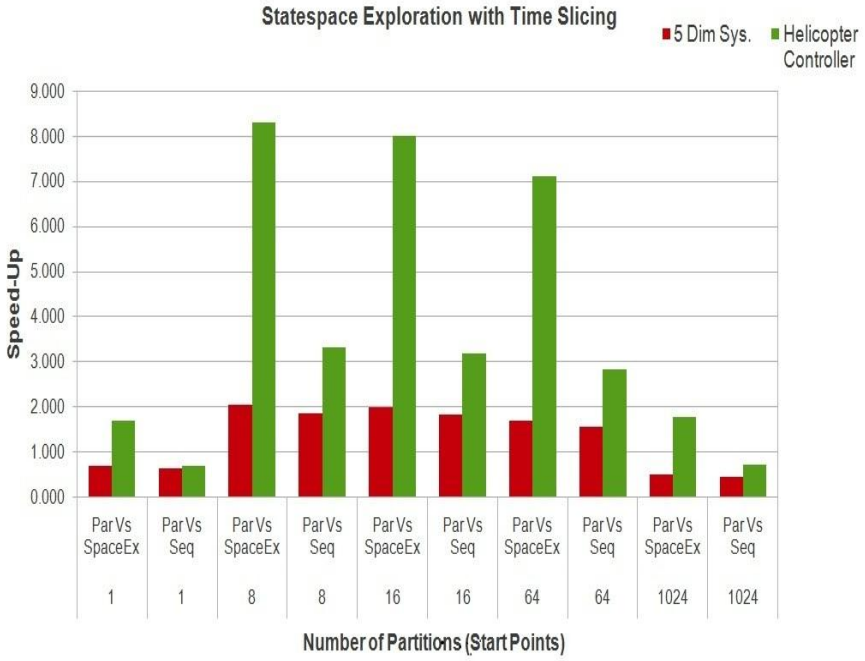
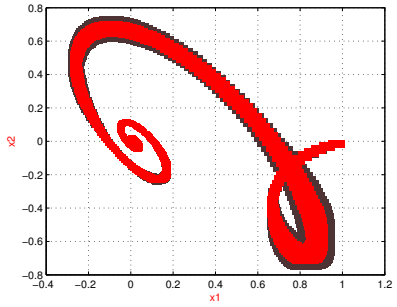
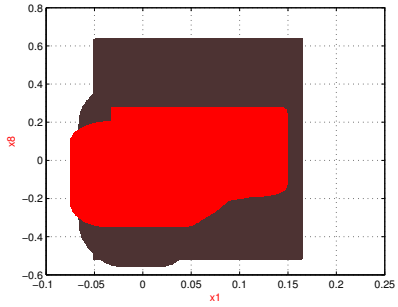


Fig. 3. Illustrating speed-up using parallel state-space exploration in time slices over sequential algorithm and the SpaceEx LGG scenario.

Table 2 shows the performance speed-up of reachability analysis when support functions are sampled in parallel in GPU (Algorithms 3, 4) compared to sequential support function algorithm using GLPK and SpaceEx LGG scenario. The iters in the table refers to the discretization factor of the time horizon. The experiments are performed in Intel Q9950, 2.84Ghz, 4 Core, no hyper-threading, 8GB RAM with GeForce GTX 670 GPU card. We observe a speed-up of 9× to 12× over sequential implementation. A maximum speed-up of 53× is observed for the five dimensional and 38× for the Helicopter model respectively compared



(a) Reachable states of the five dimensional model (Red-parallel exploration, Brown-support function algorithm)



(b) Reachable states of the Helicopter model (Red-parallel exploration, Brown-support function algorithm)

Fig. 4. Illustrating gain in precision with parallel state-space exploration over sequential algorithm.

Table 1. Speed-up and utilization gain with parallel support function samplings

Model	Dirs	4 Core (8 Threads)				6 Core (12 Threads)			
		Time (in secs)		CPU Util Gain (%)	Speed-up	Time (in secs)		CPU Util Gain (%)	Speed-up
		Seq	Par			Seq	Par		
5 Dim-Model	box	0.203	0.087	47.22	2.33	0.336	0.101	71.88	3.32
	Oct	0.937	0.337	68.75	2.78	1.532	0.401	74.46	3.82
	500	9.045	3.095	76.65	2.92	14.037	3.243	81.15	4.33
Helicopter Controller (28 dim)	box	2.418	0.571	71.05	4.23	3.36	0.608	79.96	5.52
	Oct	67.125	14.779	77.5	4.54	93.837	13.669	86.77	6.87
	3000	130.01	28.148	77.9	4.62	178.913	26.015	87.77	6.88

to SpaceEx on different parameters to the support-function algorithm. Observe that the performance of our algorithms improves with the increase in the number of cores in the machine and signifies that performance of XSpeed can scale automatically with future multicore machines and GPUs with higher degree of parallelism.

6 Conclusion

We presented a parallel implementation of the support-function algorithm and a time-sliced parallel state-space exploration algorithm. A lazy strategy of evaluating support functions to bring in parallelism is illustrated and implemented in CUDA to offload the computation task in GPU. We show that the performance of reachability algorithms for linear dynamical systems can be considerably im-

Table 2. Performance speed-up with samplings in GPU

Model	Dirs	Iters	Time (in secs)			Speed-up	
			Seq	SpaceEx	Par (GPU)	vs Seq	vs. SpaceEx
Five Dim. Model	Box	1000	0.133	0.345	0.018	7.27	18.82
	Box	2000	0.287	0.686	0.028	10.01	23.93
	Oct	1000	0.717	1.399	0.06	11.87	23.15
	Oct	2000	1.462	2.8	0.119	12.30	23.55
	500	1000	6.695	24.171	0.576	11.62	41.96
	500	2000	13.329	39.58	1.114	11.96	35.52
	1000	1000	13.128	59.996	1.121	11.71	53.52
	1000	2000	26.022	94.204	2.219	11.72	42.44
Helicopter Controller (28 dim.)	Box	1000	1.4	4.399	0.172	8.14	25.56
	Box	1500	2.077	7.263	0.249	8.33	29.11
	Box	2000	2.769	8.685	0.327	8.45	26.50
	Box	2500	3.444	11.014	0.405	8.50	27.18
	Oct	1000	39.089	123.794	4.246	9.21	29.15
	Oct	1500	57.632	248.769	6.321	9.12	39.35
	2000	1000	50.367	187.825	5.396	9.33	34.80
	3000	1000	75.086	311.652	8.054	9.32	38.69
	3000	2000	149.313	608.214	16.092	9.28	37.80

proved using the modern multi-core processors. The use of GP-GPU has shown a promising performance gain in many scientific applications and we show that they can also substantially improve the performance of reachability analysis. The parallel algorithms and the GP-GPU task offloading are implemented in the tool XSpeed.

References

1. Althoff, M., Krogh, B.H.: Zonotope bundles for the efficient computation of reachable sets. In: Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference, CDC-ECC 2011, Orlando, FL, USA, December 12-15, 2011. pp. 6814–6821. IEEE (2011), http://dx.doi.org/10.1109/CDC.2011.6160872
2. Asarin, E., Dang, T., Girard, A.: Reachability analysis of nonlinear systems using conservative approximation. In: HSCC'03. vol. 2623 in LNCS, pp. 20–35. Springer (2003)
3. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a gpgpu-parallel SPIN model checker. In: Rungta and Tkachuk [19], pp. 87–96, http://doi.acm.org/10.1145/2632362.2632379
4. Dantzig, G.B., Thapa, M.N.: Linear Programming 1: Introduction. Springer (1997)
5. Dantzig, G.B., Thapa, M.N.: Linear Programming 2: Theory and Extensions. Springer (2003)
6. Filippidis, I., Holzmann, G.J.: An improvement of the piggyback algorithm for parallel model checking. In: Rungta and Tkachuk [19], pp. 48–57, http://doi.acm.org/10.1145/2632362.2632375

7. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. STTT 10(3), 263–279 (2008)
8. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: Scalable verification of hybrid systems. In: Ganesh Gopalakrishnan, S.Q. (ed.) Proc. 23rd International Conference on Computer Aided Verification (CAV). LNCS, Springer (2011)
9. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari and Thiele [16], pp. 291–305, http://dx.doi.org/10.1007/978-3-540-31954-2_19
10. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari and Thiele [16], pp. 291–305
11. Girard, A., Le Guernic, C.: Efficient reachability analysis for linear systems using support functions. In: Proc. IFAC World Congress (2008)
12. Guernic, C.L., Girard, A.: Reachability analysis of hybrid systems using support functions. In: Bouajjani, A., Maler, O. (eds.) CAV. Lecture Notes in Computer Science, vol. 5643, pp. 540–554. Springer (2009)
13. Holzmann, G.J.: Parallelizing the spin model checker. In: Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings. pp. 155–171 (2012)
14. Lalami, M.E., Baz, D.E., Boyer, V.: Multi GPU implementation of the simplex algorithm. In: Thulasiraman, P., Yang, L.T., Pan, Q., Liu, X., Chen, Y., Huang, Y., Chang, L., Hung, C., Lee, C., Shi, J.Y., Zhang, Y. (eds.) 13th IEEE International Conference on High Performance Computing & Communication, HPCC 2011, Banff, Alberta, Canada, September 2-4, 2011. pp. 179–186. IEEE (2011), http://dx.doi.org/10.1109/HPCC.2011.32
15. Makhorin, A.: GNU Linear Programming Kit, v.4.37 (2009), http://www.gnu.org/software/glpk
16. Morari, M., Thiele, L. (eds.): Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings, Lecture Notes in Computer Science, vol. 3414. Springer (2005)
17. Ray, R., Gurung, A.: Poster: Parallel state space exploration of linear systems with inputs using xspeed. In: Girard, A., Sankaranarayanan, S. (eds.) Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015. pp. 285–286. ACM (2015), http://doi.acm.org/10.1145/2728606.2728644
18. Rockafellar, R.T., Wets, R.J.B.: Variational Analysis, vol. 317. Springer (1998)
19. Rungta, N., Tkachuk, O. (eds.): 2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014. ACM (2014), http://dl.acm.org/citation.cfm?id=2632362
20. Skogestad, S., Postlethwaite, I.: Multivariable Feedback Control: Analysis and Design. John Wiley & Sons (2005)
21. Spampinato, D.G., Elster, A.C.: Linear optimization on modern gpus. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009. pp. 1–8. IEEE (2009), http://dx.doi.org/10.1109/IPDPS.2009.5161106
22. Stursberg, O., Krogh, B.H.: Efficient representation and computation of reachable sets for hybrid systems. In: Proceedings of the 6th International Conference on Hybrid Systems: Computation and Control. pp. 482–497. HSCC'03, Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1768100.1768137