Hybrid CPU-GPU Constraint Checking: Towards Efficient Context Consistency

Jun Sui\textsuperscript{a,b}, Chang Xu\textsuperscript{a,b,}\textsuperscript{*}, S.C. Cheung\textsuperscript{c}, Wang Xi\textsuperscript{a,b}, Yanyan Jiang\textsuperscript{a,b}, Chun Cao\textsuperscript{a,b}, Xiaoxing Ma\textsuperscript{a,b}, Jian Lu\textsuperscript{a,b}

\textsuperscript{a}State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
\textsuperscript{b}Department of Computer Science and Technology, Nanjing University, Nanjing, China
\textsuperscript{c}Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China

Abstract

\textbf{Context:} Modern software increasingly relies on contexts about computing environments to provide adaptive and smart services. Such contexts, captured and derived from environments of uncontrollable noises, can be inaccurate, incomplete or even conflict with each other. This is known as the context inconsistency problem, and should be addressed by checking contexts in time to prevent abnormal behavior to applications. One popular way is to check application contexts against consistency constraints before their uses, but this can bring heavy computation due to tremendous amount of contexts in changing environments. Existing efforts improve the checking performance by incremental or concurrent computation, but they rely on CPU computing only and can consume valuable CPU capabilities that should otherwise be used by applications themselves.

\textbf{Objective:} In this article, we propose GAIN, a GPU-supported technique to checking consistency constraints systematically and efficiently.

\textbf{Method:} GAIN can automatically recognize a constraint's parallel units and associate these units and their runtime instances with matched contexts under checking. GAIN coordinates CPU and GPU and utilizes their capabilities for task preparation and context checking, respectively.

\textbf{Result:} We evaluate GAIN experimentally with millions of real-life context data. The evaluation results show that GAIN can work at least 2–7× faster and requires much less CPU usage than CPU-based techniques. Besides, GAIN can also work stably for different and varying workloads.

\textbf{Conclusion:} Our experience with GAIN suggests its high efficiency in constraint checking for context consistency as well as its wide applicability to different application workloads.

\textsuperscript{*}Corresponding author

Email addresses: smilent sj0163.com (Jun Sui), changxu@nju.edu.cn (Chang Xu), scc@cse.ust.hk (S.C. Cheung), xuanex@gmail.com (Wang Xi), jiangyy@outlook.com (Yanyan Jiang), caochun@nju.edu.cn (Chun Cao), xxm@nju.edu.cn (Xiaoxing Ma), lj@nju.edu.cn (Jian Lu)
1. Introduction

We are living in a world surrounded by smart devices, which are equipped with various sensors for monitoring surrounding environments. Context-aware applications [2, 18, 27] facilitate adaptive and smart services based on sensed environmental data, which are also known as contexts. Due to uncontrollable unreliability of sensing devices and wireless transmission, contexts collected by such applications can be inaccurate, incomplete or even conflict with each other, leading to the context inconsistency problem [22].

To detect context inconsistency, one promising way is to specify consistency constraints to enforce necessary properties that must hold about contexts [15, 22], and check newly collected contexts against these constraints to detect potential inconsistencies. Detected inconsistencies are resolved before related contexts are used by applications. By doing so, context-aware applications are prevented from behaving abnormally due to inconsistent contexts [8].

For example, consider a location-based application that provides smart routing services for city taxis based on traffic conditions. Suppose that each taxi has a client sensing device installed and the device periodically sends its sensory data about this taxi (e.g., its location, speed, service status, etc.) to a remote central server. The application running at the server end has these collected sensory data as contexts for calculating traffic conditions as well as guiding smart routings. However, these contexts could be imperfect and indicate unreasonable information, e.g., suggesting that a taxi “jumps” from one location to another far away in very short time or it drives in an impossible area such as blocked tunnels or even in sea. To detect inconsistencies caused by such imperfect contexts, one can formulate consistency constraints for different concerns, e.g., “a taxi’s speed should be in a reasonable range” or “a taxi should drive in this city’s geographical scope”. However, such inconsistency detection requires considerable computation resources for continual consistency checking when contexts keep being collected during application running.

When the workload of consistency checking is high, its performance becomes an issue. To speed up constraint checking, existing efforts have proposed various techniques like incremental checking [16, 22] or concurrent checking [24]. However, these techniques cannot effectively handle high workload of constraint checking since they solely rely on CPU capabilities and directly compete with applications that also rely on CPU computation for context processing and behavior adaptation. In this article, we propose a novel GPU-supported constraint checking technique, named GAIN (stands for GPU-based constraint checking), to exploit massive GPU parallelism to accelerate context inconsistency detection. GAIN is capable of automatically recognizing parallel computing units in a constraint and evenly distributing them across different GPU cores, such that the whole constraint checking can be conducted in parallel.
GAIN is structure-oblivious and applies to first-order logic (FOL) based constraints [15, 16, 22, 23, 24]. It works in a way totally transparent to its upper-layer applications and users. Our later experimental evaluation shows that GAIN works faster than existing CPU-based techniques but consumes less CPU capabilities. Besides, it scales much better than existing techniques and behaves extremely stable for different workloads. Specifically, we have made the following contributions in this article:

- We have proposed a GPU-based constraint checking technique GAIN, which can automatically balance its checking workload across different GPU cores.
- We have proposed a two-level storage strategy that allows GPU threads to access data in parallel without internal conflict.
- We have optimized the data transmission between CPU and GPU by an asynchronous pipelined design to address the performance bottleneck issue in GPU-supported constraint checking.
- We have experimentally evaluated our GAIN technique against existing CPU-based efforts with millions of real-life taxi data, and the results confirm GAIN’s high and stable performance with respect to various types of consistency constraints.

The rest of this article is organized as follows. Section 2 presents background knowledge on constraint checking and GPU programming. Section 3 uses an illustrative example to overview our GAIN technique. Section 4 elaborates on our whole GAIN methodology. Section 5 evaluates and compares our GAIN to existing constraint checking efforts. Section 6 presents related work, and finally Section 7 concludes this article.

2. Background

In this section, we briefly introduce some background knowledge of constraint checking and GPU programming.

2.1. Constraint Checking

In this article, a context can refer to any piece of environmental information, e.g., current location of information of a taxi at a certain time point. In the process of checking contexts against consistency constraints, we are interested in two questions: (1) Is a certain constraint violated (i.e., its truth value is evaluated to false)? (2) If yes, what has caused this violation? Typically, truth value evaluation answers the first question and link generation answers the second (each generated link gives a cause) [23].
### 2.1 Constraint Checking

For ease of presentation, we in this article assume that consistency constraints are specified in the following first-order logic (FOL) based constraint language, which has been used in existing work [15, 16, 22, 23, 24]:

\[
f ::= \forall \gamma \in S(f) \mid \exists \gamma \in S(f) \mid (f) \text{ and } (f) \mid (f) \text{ or } (f) \mid (f) \text{ implies } (f) \mid \text{not}(f) \mid bfunc(\gamma_1, \gamma_2, ..., \gamma_n).
\]

In this language, \( S \) represents a set of contexts (or context set) and variable \( \gamma \) can take any context from \( S \) as its value. Terminal \( bfunc \) can be any user-defined or application-specific function that takes parameters \( \gamma_1, \gamma_2, ..., \gamma_n \) and returns a truth value.

Consider our aforementioned location-aware application. We know that any car cannot drive too much distance in a short period of time due to its physical speed limit, yielding the following consistency constraint:

\[
\forall\text{taxi}_1 \in \text{CITY} (\forall\text{taxi}_2 \in \text{CITY} (\text{Same} (\text{taxi}_1, \text{taxi}_2)) \text{ implies } (\text{Loc} (\text{taxi}_1, \text{taxi}_2))).
\]

In this constraint, \( \text{CITY} \) represents the set of location contexts collected in the last \( T \) seconds by the central server (each car can have multiple records in \( \text{CITY} \)). The \( \text{Same} \) function judges whether a pair of location contexts concern the same taxi. The \( \text{Loc} \) function calculates the distance between two contexts’ contained locations and judges whether this distance is reasonable for the duration \( T \) with respect to the speed limit.

Each consistency constraint can be represented by a syntax tree [23]. A syntax tree describes the hierarchical structure of a constraint. A tree node represents either a quantifier symbol (\( \forall/\exists \)), a logical connective (\( \text{and, or, implies, not} \)) or a user-specified function (\( bfunc \)). A tree edge denotes the nesting relationship between two nodes. For example, the syntax tree of the constraint specified by Equation (1) is illustrated in Figure 1.

To check contexts against a constraint, a syntax tree can be expanded with context information to form a runtime tree [24]. In a runtime tree, a formula like \( \forall \gamma \in S(f) \) or \( \exists \gamma \in S(f) \) is expanded to have multiple branches, each for a context in \( S \). Each branch corresponds to a concrete assignment for \( \gamma \in S \). Consider the syntax tree in Figure 1. Suppose that the context set \( \text{CITY} \) contains
two contexts \{ctxt_1, ctxt_2\}. Thus each \forall node in this syntax tree’s corresponding runtime tree would contain two branches, as illustrated in Figure 2.

The truth value of a constraint is evaluated in a bottom-up manner on its runtime tree. Each leaf node in the runtime tree (denoting a \textit{bfunc}) can be evaluated since its parameters are determined. Thus, its truth value is obtained simply by calling \textit{bfunc}. Then the evaluation proceeds for each internal node, until the root node is evaluated.

Links are generated for explaining what has caused a constraint violated. Each link takes the form of \{\textit{type}, \textit{bindings}\}. \textit{type} \in \{\textit{violated}, \textit{satisfied}\} indicates whether the constraint is violated or satisfied, while \textit{bindings} is a set of variable-context mappings, disclosing the witnesses of a specific truth value evaluation [23]. Consider the example constraint specified by Equation (1). When variable taxi_1 takes value ctxt_1 and variable taxi_2 takes value ctxt_2, \texttt{Same(taxi}_1,\texttt{taxi}_2) holds but \texttt{Loc(taxi}_1,\texttt{taxi}_2) does not, yielding the violation of the constraint with link \{\textit{violated}, \{(\texttt{taxi}_1, ctxt_1), (\texttt{taxi}_2, ctxt_2)\}\}. Links can be generated in a bottom-up manner on a constraint’s runtime tree [15, 23].

### 2.2 CUDA Programming

A GPU contains an array of streaming multiprocessors (SMs) and each SM contains a group of scalar cores that are used to execute GPU threads. Recently, GPUs have been used for general-purpose computing traditionally handled by CPUs. A GPGPU (General-Purpose computing on Graphics Processing Units) program can contain several parallel portions, each of which is called a \textit{kernel}. Launching a kernel is syntactically like calling a function:

\[
\text{kernel} \langle\langle\langle \text{block num, thread per block } \rangle\rangle\rangle(\text{parameters}).
\]

Before launching a kernel, a programmer needs to specify the number of blocks and threads per block used to execute the kernel. A kernel is executed by many GPU threads, and each GPU thread executes one instance of the kernel. Different kernels are usually executed sequentially (some up-to-date GPUs support concurrent kernel execution).

CUDA (\textit{Compute Unified Device Architecture}) [17] is a parallel computing platform and programming model for general-purpose computing on graphics processing units. It provides a C-like language called \textit{CUDA-C}, which extends C by allowing programmers to define device functions (\textit{kernels}), which, when called, are executed independently by multiple CUDA threads in parallel on hundreds of GPU cores. Threads in CUDA are organized hierarchically: a set of threads constitutes a block and an array of blocks constitutes a grid. The threads of a thread block are executed concurrently on one SM. Every 32 threads in a thread block are grouped as a \textit{warp} to execute synchronously. Threads in a warp must execute the same instruction at the same time, and this is known as the SIMD (Single Instruction Multiple Threads) model. Otherwise the performance would be degraded severely. For example, consider an \texttt{if-then-else} statement. If the threads in a warp execute different branches (some execute the \texttt{then} clause and others execute the \texttt{else} clause), these branches will actually be executed in
sequence rather than in parallel. When executing the *then* clause, all threads that execute the *else* clause will be deactivated, and when executing the *else* clause, the situation is reversed. Such situation is called *thread divergence* and should be avoided. Thus redesigning sequential algorithms into GPU-based parallel ones and making the new algorithms suitable for GPU execution are challenging.

CUDA exposes multiple kinds of memory to developers, namely, global memory, shared memory, registers, constant memory and texture memory. The global memory is shared by all GPU threads. It is slow and uncached, but it is large (usually 1–10 GB). The shared memory is shared by threads in one block. It is fast (roughly 100× faster than the global memory) and usually used for exchanging data among threads in the same block, but it can only store 16–112 KB data. The registers are the fastest, but they are simply thread-local. The constant memory is used to store constant data and kernel arguments. It is slow but cached. The texture memory is optimized for 2D spatial access patterns. These memory spaces vary in storage size, access latency, etc. Thus besides redesigning sequential algorithms into parallel ones, balancing memory usage and understanding trade-offs caused by memory latency is also challenging [12, 19].

3. Overview

In this section, we overview our GAIN technique.

Generally, constraint checking for context consistency (i.e., truth value evaluation and link generation) is realized by processing a constraint’s runtime tree sequentially and recursively (referred to as the traditional technique *Seq-C*). Specifically, to evaluate a runtime tree node’s truth value, *Seq-C* first evaluates its child nodes’ truth values, and based on them evaluates the truth value of this node according to its associated formula’s semantics. The truth value of a runtime tree’s root node indicates whether the tree’s corresponding constraint is satisfied or violated. All nodes in a runtime tree are thus processed. Links are generated in a similarly sequential way. *Seq-C* can be parallelized (named *Con-C* [24]). *Con-C* identifies *persistently balanced splitting nodes* (PB nodes) in a runtime tree, and distributes their checking tasks to parallel CPU threads. A PB node is usually a ∀/∃ node.

The *Con-C* algorithm contains on-the-fly thread creation. It creates threads to process branches of a PB node, and the current thread (the one creating new threads) would be blocked until all created threads complete their tasks. However, GPU applications do not work that way easily. GPUs with computing capability lower than 3.5 do not support this model at all. For GPUs with computing capability of 3.5 and higher, they support dynamic parallelism. This allows a CUDA kernel to create and synchronize new tasks directly on GPU, but its maximal nesting-depth is limited to 24 (for computing capability 3.5), and it is also limited by computing resources inside GPU (e.g., pending child grids and memory). Since a given constraint can have arbitrarily nested structures
and unlimited context data matching this constraint (as the constraint language allows), a GPU’s maximal nesting-depth may not be enough at runtime.

Besides the hardware limitation, another important reason for not directly porting Con-C to GPU is due to its efficiency issue. Checking a consistency constraint by dynamic parallelism yields frequent synchronization between upper and lower runtime tree nodes. This can easily cause many kernels to be blocked. To be concrete, there are two major challenges for GPU-based parallel constraint checking: (1) Achieving massive parallelism and balancing thread workload. (2) Efficient data access. To address the first challenge, GAIN splits a consistency constraint into multiple parts (called c-units), reorders c-units to avoid recursive checking, and each time delegates one c-unit to GPU threads for parallel processing. Figure 1 shows three c-units (separated by two dashed lines), and they are reordered from bottom up (not shown in the figure). By assigning contexts to c-units’ variables as values, we obtain c-copies. Figure 2 shows three groups of c-copies, which correspond to the aforementioned three c-units and related contexts. For this example, GAIN first processes the four c-copies at the bottom, then the two c-copies in the middle, and finally the top c-copy. Accordingly, GAIN will launch its checking kernel three times. Note that all GPU threads undertake the same tasks during each kernel execution. So thread workload is balanced and no thread divergence occurs. If contexts are many, each time GAIN will create many threads and exploit the full power of GPU’s massive parallelism. Data access also plays an important role. Truth values and links of a upper c-copy rely on truth values and links of its lower c-copies, respectively. Upper c-copies and lower c-copies are checked in different kernel executions. Therefore, GAIN has to store intermediate results. The efficiency of retrieving results of lower c-copies is important to the overall performance since this can be conducted frequently. We will discuss these technical details in the following sections.

4. Checking Constraints by GAIN

Figure 3 illustrates modules in our GAIN’s architecture. For each constraint, Splitter decomposes it into a group of c-units (Figure 1), and then all the de-
4.1 Constraint Preprocessing

A syntax tree can reference more than one variable. Each variable takes as its value from its associated context set to form a runtime tree. Our idea to parallelize constraint checking is to make each GPU thread to deal with one part of the runtime tree, and all parts are structurally identical but with different contexts (as values) assigned to concerned variables. In this way, GAIN follows the SIMT model. To maximize parallelism, each variable in a part is associated with exactly one context.

A straightforward way is to check runtime tree nodes in turn, which guarantees no thread divergency. However, it will bring overlap in both kernel launches and memory accesses, which we explain later. Here we explain our parallelization in more details.

Given a consistency constraint, **Splitter** decomposes it into a group of units called **c-units**. The root of a c-unit is either the root node of a syntax tree, or a child node of ∨/∃ node. C-units of a constraint are disjointed and together form the constraint.
4.1 Constraint Preprocessing

Algorithm 1 Split a syntax tree into multiple c-units

Input root (root of a syntax tree), rootOfCunit (a list containing roots of c-units), cunits[] (an array for storing the start of each c-unit), constraintNodes[] (an array for storing a reordered syntax tree)

Initialize rootOfCunit to be empty

1: procedure Split(root)
2:   rootOfCunit.push(root);
3:   currentNodeNum = 0, cunitNum = 0;
4:   while !rootOfCunit.empty() do
5:     rootOfNextCunit = rootOfCunit.top();
6:     rootOfCunit.pop();
7:     cunits[cunitNum++] = currentNodeNum;
8:     ParseCunit(rootOfNextCunit, currentNodeNum);
9:   end while
10: end procedure

11: procedure ParseCunit(node, currentNodeNum)
12:   if node == null then
13:     return ;
14:   end if
15:   constraintNodes[currentNodeNum++] = node;
16:   if node is a ∀/∃ node then
17:     rootOfCunit.push(node.child);
18:   else
19:     ParseCunit(node.leftChild);
20:     ParseCunit(node.rightChild);
21:   end if
22: end procedure

The key function of Splitter is Split, which decomposes a constraint and is illustrated in Algorithm 1. The queue rootOfCunit stores roots of c-units. At the beginning, rootOfCunit contains the root of a syntax tree only (Line 2). When rootOfCunit is not empty, we retrieve its first element, make this element the start of a c-unit (Line 7), and construct this c-unit by calling function ParseCunit (Line 8). Function ParseCunit adds nodes in a syntax tree into a c-unit recursively in a top-down manner (Lines 20-21), until it encounters ∀/∃ nodes, which are starts of new c-units (Line 17). After this preprocessing (Split), the syntax tree in Figure 1 is converted into into the following three c-units:

$$\{ (\forall taxi_1 \in CITY), (\forall taxi_2 \in CITY), (\text{implies, Same, Loc}) \}.$$  

We generate c-copies at runtime by assigning concrete contexts to variables as values in a c-unit, and each variable in a c-copy is assigned with exactly one
4.2 Truth Value Evaluation

Algorithm 2 Kernel execution

**Input** setNum (the number of set), ThreadPerBlock (thread per block), cnstr (constraint)

1. UpdateCtxtSetKer\((\langle\langle\langle(setNum+ThreadPerBlock-1)/ThreadPerBlock, ThreadPerBlock)\rangle\rangle)\);
2. UpdateCtxtSet();
3:
4. for each cnstr do
5.   branchSize[] = ComputeRTTBranchSize(cnstrcontexts);
6.   copy branchSize[] to GPU memory;
7. 
8.   for each c-unit unit ∈ cnstr do
9.     ccopyNum = ComputeCCopyNum(unit);
10.    TruthValueKer\((\langle\langle\langle(ccopyNum+ThreadPerBlock-1)/ThreadPerBlock, ThreadPerBlock)\rangle\rangle)\);
11. 
12.   end for
13. end for

context. For example, the four c-copies at the bottom in Figure 2 are generated by assigning contexts ctxt\(_1\) and ctxt\(_2\) to variables taxi\(_1\) and taxi\(_2\) of c-unit\(_0\) in a combinatorial way. Then each c-copy is processed by one GPU thread. C-copies derived from the same c-unit are structurally identical and they are subject to the SIMD model. Compared to checking one node each time, checking one c-unit each time can reduce the amount of kernel launches, and all c-copies from the same c-unit can be checked in parallel by GPU threads. It could be argued that sub-trees starting with and/or/implies nodes can also be processed in parallel. However, it can cause thread divergence, such that one has to arrange the left branch of a c-copy node and its right branch to different warps, since the two branches can undertake different computing tasks. This not only incurs extra complexity to our GAIN, but also brings unbalanced performance. The overall GAIN performance can instead be reduced.

Besides, nodes in a c-copy can be processed recursively (new versions of CUDA support device function recursion). However, it can harm the overall performance and is not recommended. In our GAIN, recursion inside a c-copy is avoided elegantly by processing nodes in a backward way. In the previous example, the nodes of c-copies derived from c-unit (implies, Same, Loc) are processed from Loc to implies.

4.2 Truth Value Evaluation

Algorithm 2 gives our constraint checking’s pseudo code. When any context change occurs (e.g., a new context is collected), our GAIN calls the kernel function UpdateCtxtSetKer (Line 1) and host function UpdateCtxtSet (Line
Algorithm 3 ComputeCCopyNum

1: procedure ComputeCCopyNum(cunit)
2:     node = cunit.root;
3:     copyNum = 1;
4:     while node != NULL do
5:         if node is ∀/∃ then
6:             copyNum *= correspondingCtxtNum;
7:         end if
8:         node = node.parent;
9:     end while
10: end procedure

(1) \(\forall \gamma \in S(f)\) \(\alpha \ ::= \{ t_i = \text{lookup}(\tau[f]\text{bind}(\alpha, (\gamma, x_i))) \mid x_i \in S; \text{return } \top \land t_1 \land \cdots \land t_m \}; \)
(2) \(\exists \gamma \in S(f)\) \(\alpha \ ::= \{ t_i = \text{lookup}(\tau[f]\text{bind}(\alpha, (\gamma, x_i))) \mid x_i \in S; \text{return } \bot \lor t_1 \lor \cdots \lor t_m \}; \)
(3) \(\tau[f_1 \text{ and } f_2]_\alpha \ ::= \text{lookup}(\tau[f_1]_\alpha) \land \text{lookup}(\tau[f_2]_\alpha); \)
(4) \(\tau[f_1 \text{ or } f_2]_\alpha \ ::= \text{lookup}(\tau[f_1]_\alpha) \lor \text{lookup}(\tau[f_2]_\alpha); \)
(5) \(\tau[f_1 \text{ implies } f_2]_\alpha \ ::= \text{lookup}(\tau[f_1]_\alpha) \rightarrow \text{lookup}(\tau[f_2]_\alpha); \)
(6) \(\tau[\text{not } f]_\alpha \ ::= \neg \text{lookup}(\tau[f]_\alpha); \)
(7) \(\tau[\text{bfunc}(\gamma_1, ..., \gamma_n)]_\alpha \ ::= \text{bfunc}(\text{get}(\alpha, \gamma_1), ..., \text{get}(\alpha, \gamma_n)). \)

Figure 4: Truth value evaluation semantics in GAIN

2) to update related context sets in GPU memory and CPU memory, respectively. The two functions add new contexts to, or remove expired contexts from, related context sets. Note that the two functions can execute in parallel due to their disjoint nature. Once UpdateCtxtSetKer is launched, the host calls UpdateCtxtSet without waiting for UpdateCtxtSetKer to finish, so that these two functions are executed concurrently. For each constraint, the algorithm calculates the number of nodes for each branch in the constraint’s corresponding runtime tree (Line 5). This information is to be used later for deciding locations in GPU memory for storing checking results (discussed later in Section 4.4).

For each constraint, the algorithm checks its c-units one by one from bottom up. To check a c-unit, one needs to know how many c-copies it generates (Line 9), which is calculated by Algorithm 3.

In the kernel launch configuration, we fix ThreadPerBlock to 64 in GAIN. In fact, we tested different values for ThreadPerBlock (from 32 to 512, with a growth of factor 2), but found almost no difference among these values (GAIN was about 1% faster when ThreadPerBlock = 64, than the worst case where ThreadPerBlock = 512). We thus used 64 as ThreadPerBlock’s value in experiments for simplicity (not for performance tuning).

Kernel function TruthValueKer evaluates the truth value for each node from bottom up in a c-unit (Line 10 in Algorithm 2). Note that we have already reordered nodes in a c-unit for this in the earlier preprocessing. A node’s truth value indicates whether it is satisfied or violated. Figure 4 gives our
4.3 Link Generation

GAIN’s truth value evaluation semantics. The auxiliary functions are explained as follows:

- \( \tau[f]_\alpha \) returns the truth value of \( f \) under variable assignment \( \alpha \) (“\( \top \)” stands for true or “\( \bot \)” for false).
- \( \alpha \) is a variable assignment containing variable-context mappings. For example, \( \alpha = \{(\text{taxi}_1, \text{ctxt}_1), (\text{taxi}_2, \text{ctxt}_2)\} \) indicates that variable \( \text{taxi}_1 \) takes context \( \text{ctxt}_1 \) as its value and variable \( \text{taxi}_2 \) takes context \( \text{ctxt}_2 \).
- \( \text{bind}(\alpha, (\gamma, x_i)) \) adds a new variable-context mapping \( (\gamma, x_i) \) to \( \alpha \) to form a new variable assignment.
- \( \text{get}(\alpha, \gamma_i) \) returns the context bound to variable \( \gamma_i \) in variable assignment \( \alpha \).
- \( \text{lookup}(\tau[f]_\alpha) \) retrieves calculated truth value \( \tau[f]_\alpha \).

All calculated truth values can be retrieved by a \( \text{lookup} \) function. For example, in Figure 2, when GAIN evaluates truth value for the \( \forall \text{taxi}_2 \in \text{CITY} \) node under variable assignment \( \{(\text{taxi}_1, \text{ctxt}_1), (\text{taxi}_2, \text{ctxt}_2)\} \), the function retrieves truth values of the two \text{implies} nodes under variable assignments \( \{(\text{taxi}_1, \text{ctxt}_1), (\text{taxi}_2, \text{ctxt}_1)\} \) and \( \{(\text{taxi}_1, \text{ctxt}_1), (\text{taxi}_2, \text{ctxt}_2)\} \), respectively. Based on these truth values, GAIN can then evaluate truth value for this \( \forall \text{taxi}_2 \in \text{CITY} \) node. The implementation of the \( \text{lookup} \) function relies on our storage strategies, which are explained later.

For each c-copy derived from the same c-unit, \text{TruthValueKer} schedules one GPU thread to process it. Each GPU thread processes nodes according to our truth value evaluation semantics in Figure 4.

4.3. Link Generation

Figure 5 shows our link generation semantics in GAIN, and the auxiliary functions are explained below.

- \( \mathcal{L}[f]_\alpha \) returns a set of links generated for formula \( f \) under variable assignment \( \alpha \), explaining why \( f \) is satisfied or violated.
- \( \otimes \) makes a Cartesian product between two sets of links. If \( l_1 \) links (explains) formula \( f_1 \)’s truth value and \( l_2 \) links formula \( f_2 \)’s truth value, then any link from \( l_1 \otimes l_2 \) explains the truth value of \( (f_1) \) and \( (f_2) \).
- \( \cup \) merges two sets of links by set union. If \( l_1 \) links formula \( f_1 \)’s truth value and \( l_2 \) links formula \( f_2 \)’s truth value, then any link from \( l_1 \cup l_2 \) explains the truth value of \( (f_1) \) or \( (f_2) \).
- \( \text{flipSet}(\text{links}) \) changes the link type from satisfied to violated or from violated to satisfied for all links in \text{links}.
- \( \text{lookup}(\mathcal{L}[f]_\alpha) \) retrieves calculated link.
Consider a universal formula $\forall \gamma \in S(f)$. If it is violated (i.e., evaluated to false), then there exists at least one context, which, if assigned to variable $\gamma$, would cause sub-formula $f$ to be evaluated to false. Therefore, this context should be included into links generated for explaining why this universal formula is violated. Operator $\otimes$ can connect this context with other links in $f$. Existential formula can be processed similarly. A slightly complicated case is ($f_1$) and ($f_2$), which contains four sub-cases. If $f_1$ and $f_2$ are both satisfied, the and formula is also satisfied. Thus links generated by $f_1$ and $f_2$ are both needed to explain this formula’s satisfaction. If $f_1$ and $f_2$ are both violated, either links generated by $f_1$ or $f_2$ can explain the formula’s violation. This is because either $f_1$ or $f_2$ is sufficient to induce the formula’s violation. The semantics for ($f_1$) or ($f_2$) and ($f_1$) implies ($f_2$) formulae are defined in a similar way. To generate links for formula not $f$, one only needs to reverse the type for links generated by formula $f$. This is because any link that explains $f$’s violation can also explain the not $f$’s satisfaction.

4.4. Storage Strategy and C-copy Location

We in the following explain how to realize a conflict-free storage strategy and efficient lookup of previous constraint checking results on GPU. These two requirements are essential for efficient GPU context consistency checking because: (1) Constraint checking results (both truth values and links) of c-copies are stored for later lookup purposes. If we simply allow all c-copies that belong to the the same c-unit to store their results at the same address, there would be racy shared memory writes to the same address. (2) The lookup function has to be implemented efficiently as it is called by GAIN frequently.
4.4 Storage Strategy and C-copy Location

Our idea is to map the structure of a runtime tree to a linear array by a post-order traversal. There are two issues to address: (1) Runtime tree exists logically, and one cannot traverse a runtime tree when it does not even exist physically in memory; (2) Links are irregular, and one cannot know the number of links produced by a runtime tree and their sizes in advance. To address the first issue, we use an auxiliary data structure `branchSize[]` (Equation 2) and Algorithm 2 to calculate the location of a c-copy’s root node in the array. Each GPU thread calculates similarly and stores its own c-copy’s results independently to the array, and the locations of these calculated results guarantee not to overlap. Equation 2 explains function `ComputeRTTBranchSize` in earlier Algorithm 2 (Line 5). There are four cases. The current node can be: (1) A `bfunc` node: The branch rooted at this node contains one node only (the `bfunc` node itself); (2) A `not` node: The branch contains the `not` node itself, and all nodes in this node’s branch. (3) A `∀/∃` node: In this case, the nodes’ associated variable can take `ctxtNum` concrete contexts, such that it contains `ctxtNum` branches. Therefore, the branch rooted at this `∀/∃` node contains the node itself, and all nodes in this node’s all branches; (4) An `and/or/implies` node: The branch contains the node itself, and all nodes in this node’s two branches. Function `ComputeRTTBranchSize` starts from the root of a constraint’s runtime tree, and recursively calculates `branchSize[]`.

\[
\text{branchSize}(\text{node}) = \begin{cases} 
1 & \text{node.type} = \text{bfunc} \\
\text{branchSize}(\text{node.child}) + 1 & \text{node.type} = \text{not} \\
\text{ctxtNum} \times \text{branchSize}(\text{node.child}) + 1 & \text{node.type} = \forall/\exists \\
\text{branchSize}(\text{node.left}) + \\
\text{branchSize}(\text{node.right}) + 1 & \text{node.type} = \text{and/or/implies}
\end{cases}
\]  

(2)

With `branchSize[]`, we can use Algorithm 4 to calculate the location of the root of a given c-copy in an array. It resembles the classic post-order traversal, but still some differences exist. Truth values of a given c-copy (for all its nodes) are stored continuously in an order as specified in the c-copy’s corresponding c-unit. For example, consider any c-copy at the bottom in Figure 2.
Algorithm 4 Calculate the location for storing the root of a c-copy

**Input** node (root of a c-copy), branchSize[] (an array to store the number of nodes in each branch)

1: procedure CalcOffset(node)
2:   offset = 0;
3:   currentNode = node;
4:   while currentNode.parent != -1 do // Not pass the root of the runtime tree
5:     if currentNode.parent is a ∀/∃ node then
6:       offset_copy = cunits[currentCunit] - currentNode.parent.index;
7:       offset += offset_copy + branchId * branchSize[currentNode];
8:     else if currentNode.parent is an and/or/implies node then
9:       if currentNode is parent’s right child then
10:          offset += 1 + branchSize[currentNode.leftSibling];
11:     else
12:        offset += 1;
13:     end if
14:   else
15:     offset += 1;
16:   end if
17:   currentNode = currentNode.parent;
18: end while
19: return offset;
20: end procedure

It contains three nodes and its corresponding c-unit is (implies, Same, Loc) from Algorithm 1. Therefore, the truth values of these three nodes are stored continuously in the same order. In this way, once GAIN decides the location for storing the truth value of a c-copy’s root node, the locations for storing truth values of the other nodes in that c-copy can also be determined by calculating their corresponding offsets from these nodes to the root node, while the locations of all root nodes of c-units are already computed in constraintNodes[] from Algorithm 1. Given a c-copy’s root node, Algorithm 4 scans the c-copy’s belonging runtime tree from this node to the tree’s root node. At the beginning, the offset is initialized to zero (Line 2). For each node, there are three cases: (1) Its parent is a quantifier node (∀/∃) (Line 5). In this case, the parent node belongs to another c-copy (above the current c-copy in the runtime tree) and GAIN first reserves space for it (Line 6). Recall that to construct a runtime tree, a node with formula ∀/∃γ S(f) is expanded to contain multiple branches, and each branch corresponds to a concrete context in S assigned to γ. Thus we increase the offset to reserve space for the branches left to the current node (Line 7), where branchId is the index of the context assigned to variable γ (e.g., if the branch’s variable γ takes the second context from S, branchId is 1). It is the offset_copy that makes the algorithm different from the classic post-order
4.5 Data Access Optimization

Data can be stored in different places in GPU memory. GAIN needs contexts and constraints as its input, and produces truth values and links as its output.
intermediate and final results. All these data should be stored in GPU memory. Besides, GAIN uses auxiliary data like $\text{branchSize}[]$, which are calculated in CPU and should be transferred to GPU memory. GAIN stores constraints and $\text{branchSize}[]$ in constant memory, since they keep static during one kernel execution. Contexts, truth values and links are stored in the global memory, since it is the only space large enough for storing these data. In addition, when executing a kernel, we can buffer results of previous nodes in registers. For example, consider evaluating truth values for a runtime tree branch $\text{not}(\text{bfunc(...))}$. We buffer the truth value of the $\text{bfunc}$ node in registers, so that we can read it from registers instead of global memory when we evaluate truth value for the $\text{not}$ node. Since a c-copy’s truth values are stored continuously, we can buffer a c-copy’s truth values during checking and write them to global memory in one time when we finish calculating its truth values. A problem is that registers are valuable and limited, and registers may be exhausted. Besides, using too many registers is not good for improving occupancy. We will evaluate the impact of buffering in the next section.

Data transmission can also be optimized. By default, data transmission between GPU and CPU is synchronous and will block the execution of CPU code. Nevertheless in our GAIN, all data that are generated at runtime (e.g., $\text{branchSize}[]$) are transferred asynchronously by assigning a different stream in the data transmission. To guarantee that all data are available before kernel execution, we assign the same stream to adjacent memory transmission and kernel. We will also evaluate this optimization in the next section.

5. Evaluation

In this section, we first evaluate the impact of four factors that may affect our GAIN’s performance. Then we compare GAIN with CPU-based constraint checking techniques under different application scenarios. Finally, we apply GAIN to a real-world application and evaluate GAIN’s performance in practice.

5.1. Experimental Design

Context-aware applications expect contexts efficiently ready after inconsistency detection. Our experiments thus focus mainly on GAIN’s performance and study the following four performance-relating factors:

1. Checking granularity. We consider two granularities: (1) C-unit: Each time we delegate one c-unit to GPU threads for parallel processing. (2) Node: Each time we delegate one runtime tree node to GPU threads.

2. Data transmission mode. CUDA provides two ways of transferring data: synchronous transmission and asynchronous transmission. In the synchronous mode, the host thread that launches the data transmission is blocked until the data transmission finishes. However, in the asynchronous mode, the program execution continues without waiting for the data transmission to finish.
3. Buffering child nodes’ results. When processing a c-copy, results (truth values or links) of child nodes can be buffered, so that the results may be read more efficiently by their parent nodes. The buffering is used only inside a c-copy.

4. Hardware configuration. We measure GAIN’s performance on two GPU cards: an Nvidia GT 640 DDR5 card\(^1\) and an Nvidia GTX 750 Ti card\(^2\). Nvidia GT 640 is based on Kepler, the third generation of Nvidia’s GPU microarchitecture. A GT 640 card contains 384 CUDA cores and 1GB global memory. Nvidia GTX 750 Ti is based on Maxwell, the latest generation of Nvidia’s GPU microarchitecture. A GTX 750 Ti card contains 640 CUDA cores and 2GB global memory.

We also compare GAIN’s performance with CPU-based constraint checking techniques. For comparison purposes, we implemented sequential checking (Seq-C) \cite{22} and concurrent checking (Con-C) \cite{24}, to compare with our GAIN implementation.

We consider that an application running in a pervasive computing environment consists of three layers, namely, application layer, middleware layer and sensor layer. The application naturally runs at the application layer. The middleware layer supports application runs by providing required services such as application and context management. Context data are collected at the sensor layer. Our GAIN works at the middleware layer for handling context problems if any, transparent to its upper applications. GAIN buffers contexts from the sensor layer, and feeds processed contexts (e.g., after detecting and handing context inconsistencies) to other services or its upper applications. In such a model, if the services at the middleware layer do not work efficiently enough, collected contexts may not be processed in time. As a result, the underlying network buffer may overflow (OS decides the buffer size), and the services may instead process contexts in an imprecise or even wrong way (time may not be aligned or contexts may be lost). One may argue to increase the buffer size, but this does not essentially solve the problem as long as the processing efficiency is less than required. We name such a checking model dynamic checking. On the other hand, for experimental or theoretical analysis purposes, one may conceptually assume an infinitely large buffer for storing all contexts and ignoring their timeliness requirement in processing, i.e., processing the next context only when the processing of its earlier contexts finishes. We name such a checking model static checking (actually assuming the availability of all contexts and able to access them whenever necessary). In this model, contexts will never be missed although they may in dynamic checking.

For static checking, all contexts are available before checking (all are buffered). We measure GAIN’s performance by its checking time (as dependent variable). We then identify three independent variables that can affect the checking time, as listed below:

\(^1\)http://www.geforce.com/hardware/desktop-gpus/geforce-gt640/specifications
\(^2\)http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750-ti/specifications
5.2 Experimental Setup

1. **Checking technique.** The evaluation is mainly conducted by comparing GAIN with other CPU-based constraint checking techniques (i.e., Seq-C and Con-C).

2. **Constraint structure.** When the contexts under checking are fixed, a consistency constraint’s structure decides the complexity of constraint checking. Specifically, the number of nodes in a constraint’s syntax tree decides this constraint’s complexity and its checking’s complexity. Quantifier nodes ($\forall$/$\exists$) can contain multiple runtime tree branches, and therefore the number of quantifier nodes ($\forall$/$\exists$) in a constraint and whether they are nested decides how “wide” the constraint’s runtime tree can be, and this is proportional to the checking workload. Besides, the height of a constraint’s syntax tree decides the logical depth of the constraint, which poses an upper bound on the number of c-units (for GAIN) and that of recursions (for Seq-C and Con-C).

3. **Workload.** A constraint checking technique’s performance also depends on the number of contexts received in unit time (i.e., density of contexts). We consider three levels of density of contexts, namely, light, medium and heavy workloads.

For dynamic checking, contexts are collected at runtime, and if they are not checked in time, they may overflow their containing buffer. Therefore, we need to measure the number of contexts that are actually checked under dynamic checking. Besides, under dynamic checking, applications themselves can directly compete with constraint checking services in CPU usage, and this may also affect the number of contexts that are actually checked. Therefore, we also consider resource budget as one independent variable:

4. **Resource budget.** We simulate scenarios where different levels of CPU computation resources are offered, and compare performance for different constraint checking techniques. In experiments, we control the assigned percentage of all CPU time.

The first three independent variables have similar impact on constraint checking techniques under both static and dynamic checking. Therefore, we compare different constraint checking techniques by controlling the first three independent variables for static checking, and controlling the last independent variable for dynamic checking.

5.2. **Experimental Setup**

We conducted experiments with data from a real-world SUTPC application [23], which aims for smart routing planning as introduced earlier. We used its collected contexts for continuous 24 hours, and this accounts for a total of 1.55 million real taxi data. Each taxi datum (context) contains multiple fields including the concerned taxi’s timestamp, id, current location, instant speed, service status, etc. The interval between two consecutive contexts varies from 20 ms to 3,000 ms (55.9 ms on average).
5.3 Experimental Results

The SUTPC application contains 12 built-in consistency constraints. However, realizing that these built-in constraints may not necessarily be adequate in covering different constraint structures, we also randomly generated 33 constraints, as explained in Table 1. Our generated constraints cover a range of 2–6 height levels. For each height level, we also generated different syntax tree layouts that have different number of nodes and quantifiers. A 6-level constraint with multiple \( \forall/\exists \) nodes contains multiple c-units, and these c-units may contain different numbers of nodes, making the constraint’s corresponding syntax tree complicated and imbalanced. We use \( \text{Diff}(\text{c-units}) \) to represent the maximal difference between c-units of a constraint in the number of contained nodes. For example, the constraint shown in Figure 1 contains three c-units with 1, 1, 3 nodes, respectively. Thus \( \text{Diff}(\text{c-units}) \) of this constraint is \(|3 - 1| = 2\). Among all these constraints, 33 are randomly generated, and they are labeled from “A” to “K” (11 groups). The remaining three constraints \( (A^*, D^*, H^*) \) are from the original 12 constraints (the other nine constraints are structurally equivalent to \( D^* \), and thus omitted from experiments).

Experiments were conducted on a desktop machine with an Intel Q9550 quad-core CPU and 4GB RAM running MS Windows 7. We used two GPU cards: Nvidia GT 640 GDDR5 and Nvidia GTX 750 Ti, as mentioned earlier. The compute capability of the two GPU cards was both set to 3.0. We set the number of GPU threads to 64 for GAIN as explained earlier. We set the maximal number of GPU threads to 4 for Con-C, since our CPU contains four cores.

5.3. Experimental Results

We discuss experimental results in the following.

5.3.1. Experiment 1: Impact of Checking Granularity

We first study how checking granularity affects GAIN’s performance. We chose two checking granularities: c-unit and node, as explained earlier. We evaluated GAIN’s performance on three segments of contexts from all SUTPC taxi data (representing morning, noon and afternoon of a day, respectively), and each segment lasts for about 5 minutes, containing 5,000 continual contexts.

The results are shown in Figure 7a. The x-axis indicates the start time for three segments of contexts (5am, 11am, 6pm, respectively). It shows that checking by c-unit is consistently faster than checking by node (about 18.8% faster on average). We think that it is because checking by node caused more kernel launching, which was time-consuming. Besides, checking by node also caused redundant workload. For example, consider evaluating truth values for a runtime tree branch \( \text{not}(\text{bfunc}(...)) \), which contains two nodes. When checking it by c-unit, GAIN launches the truth value evaluation kernel only once. GAIN calculates storage location for the \( \text{not} \) node, and then for the \( \text{bfunc} \) node based on that of \( \text{not} \). However, when checking this branch by node, GAIN needs to launch the truth value evaluation kernel twice, and each time it has to calculate storage locations for the nodes from scratch, causing redundancy.
5.3 Experimental Results

5.3.2. Experiment 2: Impact of Data Transmission Mode

Figure 7b shows the results of using different data transmission modes. The performance is improved by about 20.4% on average when adopting asynchronous data transmission. It indicates that data transmission can be a bottleneck for GPU applications. To get better performance, an application needs to improve data transmission efficiency or conduct more computation tasks and less data transmission.

5.3.3. Experiment 3: Impact of Buffering

When checking by c-unit, child nodes’ results can be buffered for efficiency. We evaluated GAIN with and without buffering, and the results are shown in Figure 7c. It seems that buffering only made marginal difference: buffering child nodes’ results improved the performance by 1.0% on average. This may be because buffering requires too many registers, and CUDA compiler has to allocate global memory for buffering instead, and this works as if no buffering was enabled (since unbuffered results are stored in global memory).

5.3.4. Experiment 4: Impact of Hardware Configuration

We evaluated GAIN on two GPU cards: an Nvidia GT 640 GDDR5 card and an Nvidia GTX 750 Ti card. The results are shown in Figure 7d. On GTX
### Table 1: Different constraint structures

<table>
<thead>
<tr>
<th>Height</th>
<th># $\forall/\exists$ quantifiers</th>
<th># generated constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-level</td>
<td>$\forall/\exists$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 (A*)</td>
</tr>
<tr>
<td>4-level</td>
<td>$\forall/\exists$</td>
<td></td>
</tr>
<tr>
<td></td>
<td># nodes $\leq$ 5</td>
<td>3 (B)</td>
</tr>
<tr>
<td></td>
<td># nodes $\geq$ 6</td>
<td>3 (C)</td>
</tr>
<tr>
<td></td>
<td>nested</td>
<td>3 (D)</td>
</tr>
<tr>
<td></td>
<td>not nested</td>
<td>3 (E)</td>
</tr>
<tr>
<td>6-level</td>
<td>$\forall/\exists$</td>
<td></td>
</tr>
<tr>
<td></td>
<td># nodes $\leq$ 10</td>
<td>3 (F)</td>
</tr>
<tr>
<td></td>
<td># nodes $\geq$ 20</td>
<td>3 (G)</td>
</tr>
<tr>
<td></td>
<td>nested</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{Diff}(c\text{-units}) \leq 3$</td>
<td>3 (H)</td>
</tr>
<tr>
<td></td>
<td>$\text{Diff}(c\text{-units}) = 5$</td>
<td>1 (H*)</td>
</tr>
<tr>
<td></td>
<td>$\text{Diff}(c\text{-units}) \geq 7$</td>
<td>3 (I)</td>
</tr>
<tr>
<td></td>
<td>not nested</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{Diff}(c\text{-units}) \leq 3$</td>
<td>3 (J)</td>
</tr>
<tr>
<td></td>
<td>$\text{Diff}(c\text{-units}) \geq 7$</td>
<td>3 (K)</td>
</tr>
</tbody>
</table>

For saving space, we list 3 built-in constraints (from A*, D* and H* in Table 1, representing all 12 built-in constraints) and 3 randomly generated constraints (from D, E and I in Table 1, representing all 33 randomly generated constraints) in the Appendix. For better readability, we have reorganized these constraints with internal structures indented for appropriate space. We list 3 built-in constraints. The remaining 9 ones (totally 12) are structurally identical to the D* constraint (only contained variables referencing different context sets), and therefore omitted. For randomly generated constraints, they can vary greatly in structure, as compared to built-in counterparts. We list 3 of them for illustrating how different they can be.

We study how a constraint’s structure affects the checking performance for
5.3 Experimental Results

GAIN, Seq-C and Con-C. For each constraint from 11 categories (A to K), we compared the performance for different constraint checking techniques. We still evaluated the performance using the three segments of contexts, as mentioned earlier, and compared the average checking time.

The evaluation results are shown in Figure 8. Figure 8a compares the performance for checking 1-quantifier constraints, while Figure 8b does for checking 2-quantifier constraints. We observe that GAIN outperformed both Seq-C and Con-C in all settings. As expected, Seq-C was less efficient, and its checking time increased dramatically when there were a large number of nodes in the syntax tree (G), or there were nested quantifiers (D, H, I). Con-C scaled better than Seq-C by taking advantage of CPU-level multi-threading (G), but was still less efficient than our GPU (D, H, I).

We then study the performance impact of syntax tree size, as shown in Figure 9. We observe that when two constraints’ syntax trees had a similar number of nodes, their checking time was also close for the three techniques (C vs. B). However, when the number of syntax tree nodes differs largely, Seq-C’s
5.3 Experimental Results

![Graph showing experimental results](image)

(a) 4-level constraints with two quantifiers
(b) 6-level constraints with two quantifiers

Figure 10: Impact of quantifier nesting

<table>
<thead>
<tr>
<th>Period of time (one hour)</th>
<th># contexts</th>
<th>Service status: on</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:00-6:00</td>
<td>69,397</td>
<td>3,895</td>
<td>Light workload</td>
</tr>
<tr>
<td>11:00-12:00</td>
<td>63,074</td>
<td>15,638</td>
<td>Medium workload</td>
</tr>
<tr>
<td>18:00-19:00</td>
<td>60,736</td>
<td>22,707</td>
<td>Heavy workload</td>
</tr>
</tbody>
</table>

Table 2: Three segments of context sequences

Performance was clearly lower, as compared to that of GAIN and Con-C, which scaled better.

Finally, we study the performance impact of quantifier nesting, as shown in Figure 10. We observe that quantifier nesting had a dominant impact on Seq-C’s and Con-C’s performance (D vs. E and H/I vs. J/K), but this impact became marginal for GAIN. Besides, Figure 10b also shows that in contrast to Seq-C and Con-C, increasing Diff(c-units) incurs negligible performance penalty to GAIN.

5.3.6. Experiment 6: Impact of Workload

We also study how the checking workload affects the performance of GAIN, Seq-C and Con-C. We evaluated each technique’s performance on three built-in constraints of SUTPC (A*, D*, H* in Table 1), and for each constraint we evaluated three different levels of workload, as listed in Table 2. Note that more contexts with a service status of “on” implies higher checking workload. The three segments of context sequences (each lasting for one hour) were from different periods of time in a day. They represent light (early morning before rush hour), medium (noon) and heavy (evening rush hour) workloads, respectively.

Figure 11 shows the evaluation results. The checking performance of GAIN is slightly less efficient than Con-C for light workload. This is because a light workload implies that most GPU cores are idle, while GAIN has to conduct
5.3 Experimental Results

![Graph showing impact of checking workload](image)

Figure 11: Impact of checking workload

Data initialization and transmission. However, when the checking workload increased (i.e., medium and heavy workloads), Seq-C and Con-C encountered their performance bottleneck. It is expected that GAIN outperformed Seq-C and Con-C for heavy workload, scaled much better (efficient and stable) as illustrated in Figure 11.

5.3.7. Experiment 7: Impact of Resource Budget

We study how resource budget affects the performance of constraint checking techniques under dynamic checking. For dynamic checking, timeliness is critical to avoiding context buffer overflow. A context will be missed if it fails to be buffered due to its earlier buffered contexts being processed not efficiently enough. We compare GAIN, Seq-C and Con-C using the same data set as in Section 5.3.6. To test an extremely heavy workload for comparison purposes, we also synthesized a new data set by randomly selecting a hour of taxi data and setting their service status to all “on”. These four segments of context sequences (each lasting for one hour) were all checked against the 12 built-in constraints under SUTPC under 100% and 50% CPU resource budget, respectively. We realized 50% CPU budget by enabling 2 CPU cores only.

Table 3a and b show the evaluation results for 100% and 50% CPU budget, respectively. In Table 3a, all three techniques were allowed to use 100% CPU budget. Both Seq-C and Con-C missed some contexts under the extremely heavy workload, in which Seq-C behaved worse, while our GAIN did not. Table 3b shows the case when only two CPU cores were available. This time, all three techniques missed some contexts. Nevertheless, CPU-based techniques missed much more contexts (Seq-C: 659 or 13.2% more, Con-C: 179 or 3.6% more), as compared to GAIN under the extremely heavy workload.

5.3.8. Case Study

Finally, we conducted a complete case study using all 1.55 million taxi data. All data were checked against the 12 built-in constraints under static checking.
5.3 Experimental Results

Table 3: Impact of Resource Budget (subscript “d” means dynamic checking)

(a) # contexts processed under 100% CPU budget

<table>
<thead>
<tr>
<th></th>
<th>Light</th>
<th>Medium</th>
<th>Heavy</th>
<th>Extremely heavy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq-C&lt;sub&gt;d&lt;/sub&gt;</td>
<td>5,000</td>
<td>5,000</td>
<td>5,000</td>
<td>4,337</td>
</tr>
<tr>
<td>Con-C&lt;sub&gt;d&lt;/sub&gt;</td>
<td>5,000</td>
<td>5,000</td>
<td>5,000</td>
<td>4,935</td>
</tr>
<tr>
<td>GAIN&lt;sub&gt;d&lt;/sub&gt;</td>
<td>5,000</td>
<td>5,000</td>
<td>5,000</td>
<td>5,000</td>
</tr>
</tbody>
</table>

(b) # contexts processed under 50% CPU budget

<table>
<thead>
<tr>
<th></th>
<th>Light</th>
<th>Medium</th>
<th>Heavy</th>
<th>Extremely heavy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq-C&lt;sub&gt;d&lt;/sub&gt;</td>
<td>5,000</td>
<td>4,999</td>
<td>4,989</td>
<td>4,335</td>
</tr>
<tr>
<td>Con-C&lt;sub&gt;d&lt;/sub&gt;</td>
<td>5,000</td>
<td>5,000</td>
<td>4,999</td>
<td>4,815</td>
</tr>
<tr>
<td>GAIN&lt;sub&gt;d&lt;/sub&gt;</td>
<td>5,000</td>
<td>5,000</td>
<td>4,999</td>
<td>4,994</td>
</tr>
</tbody>
</table>

Figure 12: Comparison of checking time (case study)
Figure 12 delivers the study results. Besides the original GAIN (denoted as GAIN-standard), there are three other variants, each slightly different from GAIN-standard: (1) GAIN-no-buffering did not buffer child nodes’ results; (2) GAIN-synchronous adopts the synchronous data transmission mode; (3) GAIN-node checks by node from bottom up for runtime trees. Besides, we also ran Seq-C and Con-C for comparison purposes. We observe that Seq-C was the least efficient and also very sensitive to varying workload (for different periods of time). Con-C behaved similarly but was better than Seq-C. In contrast to Seq-C and Con-C that were sensitive to varying workload, all GAIN variants did not have their performance noticeably affected by varying workload. We observe that in most periods, GAIN worked much better than CPU-based techniques. However, in few periods (from 16–19) when the workload was very light (most taxis are not on service from midnight to dawn), CPU-based techniques were slightly more efficient than GAIN since CPU-GPU data transmission took a high proportion of total checking time (43.8% on GT 640 and 66.9% on GTX 750 Ti). Among different GAIN variants, we observe that GAIN-standard was slightly more efficient than GAIN-no-buffering (about 0.9%), and it was 16.5% and 16.2% more efficient than GAIN-node and GAIN-synchronous, respectively. Finally, we also ran GAIN-standard on GTX 750 Ti (denoted as GAIN-GTX 750) for comparison, and we observe that its performance was increased by 29%. All these experimental results (in a case study setting) confirm our earlier conclusion about GAIN’s efficiency and scalability.

6. Related Work

Consistency is an important property of software systems. However, many software artifacts suffer the inconsistency problem. To detect inconsistencies, these artifacts (including contexts) need checking against consistency constraints. One pioneer piece of work is xLinkit [15, 16], which detects inconsistencies in XML documents. Pervasive computing also suffers the inconsistency problem [26, 27]. In work [22], to ensure the reliability of pervasive applications, contexts are checked with the notion of syntax tree (or Consistency Computing Tree). In work [21, 25], certain patterns are learned to identify inconsistency hazards so that unnecessary inconsistency detection and resolution can be avoided. GAIN focuses on detecting inconsistent application contexts. Since pervasive applications often require sensing devices to continuously collect environmental contexts, checking performance is an important concern for applications’ timeliness requirement. What our work concerns looks similar to the constraint satisfaction problem (CSP): they both use constraints to specify properties for enforcement. However, our work has to explore all causes explaining how a constraint is violated for inconsistency resolution. Besides, a constraint can contain nested quantifiers and reference a large number of contexts, and this leads to heavy computation in its checking. A popular way of improving checking performance is to conduct incremental checking, which exploits existing intermediate results to reduce redundant computation. Examples of incremental checking techniques include our previous work PCC [23], which
focuses on context consistency for pervasive computing applications, and UML/Analyzer [9], which checks consistency for evolving UML models. Con-C [24] explored ways of checking consistency constraints in parallel on CPU cores. It maintains balanced workload for all CPU threads, but cannot be easily applied to GPU-based parallel checking due to different programming models.

There is a large body of work on GPU computing. Recent work [6, 7] explored GPU-based techniques to parallelize solving CSP. The basic idea is to delegate subtasks of computation to different blocks. However, for our problem, a c-unit from a constraint’s syntax tree can generate tens of thousands of c-copies, which quickly exceed a block’s capability. Therefore, we need to explore new GPU-based techniques for parallel constraint checking. Some development aids have been proposed to ease GPU programming, and certain GPU algorithms have also been studied for specific problems. For the former, hiCUDA [10], Mars [11] and Medusa [28] provide high-level abstractions for CUDA programming. They provide APIs or directives to hide underlying details of GPU programming. Our GAIN technique is inspired by many pieces of existing work. Bakkum et al. parallelized SQL operation \textit{SELECT} by assigning each row to a GPU thread to execute queries [1]. Similarly, GAIN parallelizes a c-unit’s checking by assigning each c-copy of that c-unit to a GPU thread. Parallel algorithms based on tree structures with GPU have also been studied, e.g., parallelized R-tree queries and constructions [13] and GPU-based n-body algorithm [5]. In this article, we address the particular challenge of efficiently checking constraints for context consistency. Our work needs to process runtime tree nodes of different formula types, as well as links of irregular sizes in generation. Besides, links processed by constraint checking can be irregular. We used prefix sum [3] to realize a two-level storage strategy for conflict-free and efficient runtime tree construction and lookup. Prefix sum has been also used in GPU-based breadth-first search [14] for computing scatter offsets for run-length expansion. There is also previous research concerning principles for implementing and optimizing algorithms on GPU [12, 19]. Although these optimization techniques are application-specific and can vary greatly in nature, they all aim for reducing memory latency and maximizing parallelism. For our GAIN, we reduce memory latency by asynchronous data transmission and maximize parallelism by c-unit decomposition. There are also many pieces of work adopting multi-GPU to improve computation scalability. In such situations, application data can reside in different GPUs, and data transmissions between GPUs may be required for computing final results. For example, GPMR [20] proposed using multiple GPUs for map/reduce algorithms. Each GPU is a map/reduce node. GPMR divides data into chunks and each time feeds one chunk to one GPU. It uses partial reduction and accumulation to combine like-keyed pairs to reduce data transmission cost. In work [4], multi-spin Monte Carlo simulation is accelerated by dividing a quadratic lattice into smaller lattices, and letting each smaller lattice be handled by one of the installed GPUs. The spins at the borders of a lattice are affected by spins from its adjacent lattice. Therefore spins at the borders have to be transferred from one GPU to GPUs handling the adjacent lattices. For each smaller lattice, Block et al. [4] uses four arrays
to store these spins at the borders, and these arrays are exchanged between GPUs handling neighbor lattices. Communications between GPUs need to be carefully designed to reduce data transmission cost when multiple GPUs are adopted for more scalable computing. We are also investigating possible ways to utilize multiple GPUs for better parallel constraint checking.

7. Conclusion

In this article, we present GAIN, a CPU-GPU hybrid technique, to efficiently detect context inconsistencies by parallel constraint checking on GPU. GAIN works for general FOL-based consistency constraints. It achieves massive parallelism by automatically recognizing parallel units in constraints and delegating them to GPU threads, and it guarantees this to satisfy the SIMT model. We carefully designed GAIN’s storage strategy to make it retrieve checking results efficiently and avoid writing conflicts. We evaluated our GAIN with different configurations and compared it with CPU-based constraint checking techniques. We also evaluated GAIN with a real-world application. The evaluation results show that GAIN works much more efficiently than CPU-based techniques, and behaves consistently stable in performance. We note that GAIN is suitable for computation-intensive or data-intensive applications.

Our GAIN still has limitations or unexplored issues. First, new GPUs start to support new features like dynamic parallelism and unified memory, which our GAIN has not explored. These features can potentially further improve our GAIN’s performance, by addressing the trouble of dedicated design for splitting runtime tree branches with even workloads and of placing intermediate results in different places in GPU memory. Second, our current GAIN can only work with a single GPU. How to combine several (especially different) GPUs to make them cooperate for parallel constraint checking deserves further efforts. Third, GAIN requires no parameter tuning for parallel checking. One potential limitation is that GAIN’s performance might be affected by certain parameter values, although we did not observe this in experiments. We are investigating more GPU-supported parallel computing models and working along this line.

8. Acknowledgement

This work was supported in part by National Basic Research 973 Program (grant no. 2015CB352202), National High-Tech Research & Development 863 Program (grant no. 2015AA01A203) and National Natural Science Foundation (grant nos. 61472174, 91318301, 61321491) of China, and by Research Grants Council (611811) of HongKong.

References


Appendix. Example Constraints

- Constraint A*:
  \[ \forall \gamma_1 \in pat_{000} \]
  \[ sz\_loc\_range(\gamma_1) \]

- Constraint D*:
  \[ \forall \gamma_1 \in pat_{100} \]
  \[ \forall \gamma_2 \in pat_{101} \]
  \[ implies \]
  \[ same(\gamma_1, \gamma_2) \]
  \[ sz\_loc\_dist(\gamma_1, \gamma_2) \]

- Constraint H*:
  \[ \forall \gamma_1 \in pat_{001} \]
  \[ \forall \gamma_2 \in pat_{002} \]
  \[ implies \]
  \[ and \]
  \[ not \]
  \[ same(\gamma_1, \gamma_2) \]
  \[ sz\_loc\_close(\gamma_1, \gamma_2) \]
  \[ sz\_spd\_close(\gamma_1, \gamma_2) \]

- Constraint D:
  \[ \forall \gamma_1 \in pat_{001} \]
  \[ and \]
  \[ \exists \gamma_2 \in pat_{001} \]
  \[ same(\gamma_1, \gamma_2) \]
  \[ or \]
  \[ sz\_loc\_range(\gamma_1) \]
  \[ sz\_loc\_range(\gamma_1) \]
• Constraint E:

\[
\text{implies} \\
\text{not} \\
\forall \gamma_1 \in \text{pat}_{001} \\
\text{sz\_loc\_range}(\gamma_1) \\
\forall \gamma_1 \in \text{pat}_{001} \\
\text{sz\_loc\_range}(\gamma_1)
\]

• Constraint I:

\[
\exists \gamma_1 \in \text{pat}_{001} \\
\text{and} \\
\text{not} \\
\forall \gamma_2 \in \text{pat}_{001} \\
\text{not} \\
\text{same}(\gamma_1, \gamma_2) \\
\text{or} \\
\text{not} \\
\text{and} \\
\text{sz\_loc\_range}(\gamma_1) \\
\text{sz\_loc\_range}(\gamma_1) \\
\text{not} \\
\text{sz\_loc\_range}(\gamma_1)
\]