Exploratory Data Analysis of Software Repositories via GPU Processing

Jose Ricardo da Silva Junior,
Esteban Clua, Leonardo Murta
Universidade Federal Fluminense
Niterói - Brazil
{jricardo,esteban,leomurta}@ic.uff.br

Anita Sarma
Computer Science and Engineering
University of Nebraska, Lincoln
Lincoln – United States
asarma@cse.unl.edu

Abstract— **Analyzing software repositories with thousands of artifacts is data intensive, which makes interactive exploration analysis of such data infeasible. We introduce a novel approach, Dominoes, that can support automated exploration of relationships amongst project elements, where users have the flexibility to explore on the fly the numerous types of project relationships. Dominoes organizes data extracted from software repositories into multiple matrices that can be treated as domino pieces (e.g., [commit|method]). It allows connecting such pieces based on a set of matrix operations to derive additional domino pieces. These derived domino pieces represent semantics on project entity relationships (e.g., number of commits in which two methods co-occurred) and can be used for further explorations. This opens a vast possibility of data analysis, since these domino pieces can be iteratively combined. Our proposed matrix representation and operations allow for fast and efficient processing of a large volume of data by using a highly parallel architecture, such as GPUs.**

Keywords- ***Exploratory data analysis; software dependencies; GPU computing***

I. Introduction

When working on a software project, developers often need to answer numerous questions, such as: which other methods do I need to edit if I make this change, who was the person who last edited this method, who has expertise in this module, who do I need to coordinate my changes with, and so on [1]. Since software development leaves behind activity logs (i.e., commits recorded in the version control system and tasks recorded in the issue tracking system), it is possible to answer these questions by analyzing these software repositories. However, finding these answers from the repositories is not easy since there is extensive amount of data that is accrued over the project's lifecycle and this data is typically stored across different repositories [2]. This makes creating the right queries a nontrivial task [1].

Several research prototypes attempt to help in project explorations. For example, Tesseract [2] allows interactive investigation of relationships among files, developers, and issues through a network representation. Information Fragments [1] allows a user to compose information from tasks, change sets and teams to explore the relationships between these entities. CodeBook [3] builds a graph of all relationship, and then provides specific applications for answering specific questions (e.g., finding related developers or artifacts).

There are several critical deficiencies with these current approaches. First, these tools often focus only on a particular development aspect (e.g., EEL [4] helps in expert identification). Second, these tools typically allow explorations of specific relationships that are fixed a priori (e.g., Tesseract preprocesses the sets of dyadic relationships first). Third, these tools often need a complete history of the project (e.g., in order to traverse the relationship graph, Codebook requires the full history). Fourth, all these tools operate at coarse granularity (file level). Identifying change impact information at a fine-grained level can highlight nuanced relationships. For instance, although two developers have edited the same file, it is possible that these developers have complementary expertise in the functionality (method) of the file. Finally, these tools need to restrict the data that can be analyzed because performing interactive data analytics of software archives through visual explorations of relationships among project elements is infeasible at the scale of operation that is needed.

In this paper, we present a novel approach – Dominoes – designed to enable interactive exploratory analysis of relationships of different software entities at varying levels of granularity by utilizing matrix operations that can be computed via GPUs. Our approach organizes data from a software repository into multiple matrices that are treated as domino tiles, such as [developer|commit], [commit|method], [class|method], amongst many other combinations. Just as in the Dominoes game, where joining two congruent squares edge to edge can form a rectangle, our matrices can be combined to create additional (derived) matrices. This derivation process is guided by a set of matrix operations, such as addition, multiplication, and transposition. For example, a computation of logical coupling at the method level can be achieved as [method|method] = $[\text{commit}|\text{method}]^T \times$ [commit| method]. With this new (derived) domino tile, we can derive dependency among developers as [developer|developer] = [developer|commit] × [commit|method] × [method|method] × $[\text{commit}|\text{method}]^T \times [\text{developer}|\text{commit}]^T$. Many such different explorations are possible, with each derived domino tile representing a particular aspect in software engineering.

A primary goal of Dominoes is to enable users to explore the relationships in their project elements across different levels of granularity. Therefore, the granularity aspect is a central construct, with one of the domino tile types being that of "composition" (e.g. [package|class], [file|class], and [class|method]). Connecting any other domino tile with these composition tiles or their transpose allows navigation from coarse-grained to fine-grained analysis or vice versa.

Explorations of such relationships at a fine-grained level (methods across different versions of the software) while more accurate, can lead to extremely large data sets to be analyzed. Dominoes implements the exploratory analysis of software project entities as linear algebra operations over matrices, which can be parallelized in GPU (Graphics Processing Unit) [5]. This allows boosts in performance of about three orders of magnitude [7]. Therefore, Dominoes opens a new realm of exploratory software analysis, as endless combinations of domino pieces can be experimented with in an exploratory fashion.

II. Dominoes Approach

In this section we describe our overall approach and then focus on the matrices and how we operate over them. Dominoes extracts data from a software repository and converts them into multiple matrices, correlating the desired attributes in lines and columns. This strategy allows data manipulation and its operations using parallel architecture. In our case, this fact allows interactive manipulations even with large datasets, as we are using GPU to perform matrix operations.

We represent a matrix as M and its transpose by using a superscript (M^T). Individual elements in a matrix are denoted as M[*i,j*]. The operator "×" represents matrix multiplication. It is important to note that when multiplying two matrices the number of columns in the first operand must be equal to the number of rows in the second operand. In our case the column and rows of the operand over which we are multiplying also needs to be the congruent (same project element), similar to the Dominoes game. In other words, we can multiply [developer|commit] × [commit|method], but not [developer|commit] × [method|method].

A. Dominoes Tiles

Dominoes includes *basic building tiles*, which can be combined to create *derived building tiles*; which can be further combined with other basic or derived tiles. The *basic building tiles* are created by extracting data from existing software repositories (version control systems, issue tracking systems, etc.). For example, commits, issues, discussions about a commit, or pull request can be collected from GitHub. The *basic building tiles* around commits include:

- [class|method] (ClM): relationship between a class and its constituent methods, where cell ClM[*i,j*] has a value of 1 when class *i* contains method *j*.
- [commit|method] (CM): relationship between commits and methods, where cell CM[*i,j*] has a value of 1 when commit *i* adds or changes method *j*. Note that the index *i* does not necessary express the commit id.
- [developer|commit] (DC): relationship between developers and their commits, where cell DC[*i,j*] has a value of 1 when developer *i* is the author of commit *j*.
- [bug|commit] (BC): relationship between commits and bugs, where cell BC[*i,j*] has a value of 1 when commit *j* fixed bug *i*.

These *basic building tiles* can then be combined to form a series of *derived building tiles*. Here we show a small set of *derived building tiles* that can be computed using only multiplication and transposition operations:

- [method|method] ($MM = CM^T \times CM$): represents method dependencies, where MM[*i,j*] denotes the strength of the dependency of method *j* on method *i*. The rationale of this matrix is based on logical dependencies, as elements that are co-committed share some program logic. Note that we can also create an MM matrix through program analysis, in which case it would be termed as a *basic building tile*. Such MM matrices have been explored by Steward in creating Design Structure Matrices [8]. In Section III-B we explore more elaborate ways for computing MM.
- [class|class] ($ClCl = ClM \times MM \times ClM^T$): represents class dependencies, where ClCl[*i,j*] denotes the strength of the dependency of class *j* on class *i*. Note that using the composition tile, we can provide analysis results at a higher-level of abstraction easily.
- [bug|method] ($BM = BC \times CM$): represents the methods that were changed to fix each bug. This matrix could be used to identify which methods are "buggy".
- [developer|method] ($DM = DC \times CM$): represents the methods that a developer has changed. This matrix could be used to identify experts on a particular method as well as if there is breadth in expertise for a given method.
- [developer|class] ($DCl = DM \times ClM^T$): represents classes that a developer has changed. DCl uses the composition operation to provide expertise information at the class level, which is typically used during bug triaging [3].
- [developer|developer] ($DD = DM \times MM^T \times DM^T$): represents the expertise dependency among developers, where developer *j* depends on some knowledge of developer *i*, because of underlying technical dependencies in their work. It is worth no notice that this *derived building tile* used other *derived building tile* (MM and DM) in its definition.

By applying the composition operation, the above software engineering constructs can switch to class or file grains.

B. Specialized Operations

Our basic matrices are typically binary, that is, M[*i,j*] is either 1 or 0, whereas our derived matrices are not. This is largely because commits are atomic transactions and therefore most associated matrices with commits are binary. In the case of derived matrices cell values have associated semantics. Simple operations such as multiplication and transposition allow us to compose different types of domino tiles to derive more complex matrices and, thereby, different software engineering constructs. However, there are three "specialized" operations that can be applied on derived matrices where individual cells are not binary.

Let us take the example of the MM matrix. The diagonal shows how frequently a method has been changed and each cell (M[*i,j*]) shows how frequently a method (*i*) has changed with another method (*j*). This semantics is equivalent to *absolute support*, largely adopted in the data mining community. The support of an item set is defined as the proportion of transactions in the dataset that contains the item set. According to [9], the rule $X \longrightarrow Y$ has support **s** if **s**% of transactions contain $X \cup Y$. As this operation pattern of multiplying a matrix by its transpose is very popular and semantic rich, we treat it as a specialized operation and is computed according to Eq. 1.

$$\mathrm{M}^{\mathrm{sup}} = \mathrm{M} \times \mathrm{M}^{\mathrm{T}} \qquad (1)$$

The semantics of support allows us to answer software engineering related questions regarding the strength of the relationships. For example, if we are interested in predicting the other files a developer needs to edit because of a change, we can use the concept of logical dependencies to identify all those methods that are dependent of the edited method and may also need to be changed. We could use the $\mathrm{MM} = \mathrm{MC}^{\mathrm{sup}}$ matrix to answer this question.

Unfortunately, support is transitive, where $\mathrm{M}^{\mathrm{sup}}[i,j] = \mathrm{M}^{\mathrm{sup}}[j,i]$. Consequently, using support to represent dependencies is not precise, as program dependency is not transitive. For example, in our scenario, the area of a Cylinder depends on the circumference of the Circle, but not vice versa.

In order to obtain a more precise relationship that reflects the direction of the dependency, Zimmermann et al. [10] use *confidence* to represent logical coupling. This metric suggests which artifacts should be modified together, given that a specific artifact is being modified. According to [11], the rule $X \longrightarrow Y$ has confidence **c** if **c**% of transactions that contain X also contain Y [11]. In the context of our approach, when applied to compute MM matrix, confidence quantifies the occurrence of an entity (e.g., method) change given that the other entity (e.g., method) has also been changed. The confidence operator is computed according to Eq. 2.

$$\mathrm{M}^{\mathrm{conf}}[i,j] = \frac{\mathrm{M}^{\mathrm{sup}}[i,j]}{\mathrm{M}^{\mathrm{sup}}[i,i]} \qquad (2)$$

Confidence does not have a transitive property among elements, so it is possible to define different levels of dependency for each pair. However, confidence suffers form another type of problem. In the context of data mining, confidence is used to quantify relations such as *"those who buy product X also buy product Y"*. In this case, if product "*Y*" is presented in almost all orders, purchase of any product will lead to a high confidence in buying "*Y*". For this reason, analyzing confidence alone tends to be imprecise, and can exhibit false relationships.

To address this problem we can use a third metric – *lift* [9]. Lift measures the influence of the antecedent in the frequency of the consequent. Formally, the rule $X \longrightarrow Y$ has lift **l** if the frequency of Y increases in **l** times when X occurs. According to this definition, we are interested in dependencies with lift greater than 1, as any other value implies irrelevant (coincidental) relationships. The lift operator is defined by Eq. 3, where the scalar multiplication by the number of commits ($\mathrm{M}^{\mathrm{rows}}$) transforms the absolute support ($\mathrm{M}^{\mathrm{sup}}$) into relative support (values ranging from 0 to 1).

$$\mathrm{M}^{\mathrm{lift}}[i,j] = \frac{\mathrm{M}^{\mathrm{conf}}[i,j] \times \mathrm{M}^{\mathrm{rows}}}{\mathrm{M}^{\mathrm{sup}}[j,j]} \qquad (3)$$

III. Discussion

Here we describe our approach by drawing on a scenario detailed in Section III-A. We apply our approach in identifying artifact dependencies on Apache Derby[1], an open source relational database project, as explained in Section III-B. Finally, in Section III-C we discuss the Dominoes architecture.

A. Scenario Evaluation

Consider a scenario where three developers (*Alice, Bob*, and *Carlos*) work together on a "geometry project", consisting of four classes (Circle, Cylinder, Cone, and Shape). Circle has a method *circumf()* that calculates its circumference. Shape has a method *draw()* to render a shape. Finally both Cylinder and Cone have methods *area()* to calculate the area of the respective shapes. Table I describes five commits and their change descriptions. Table II shows which commits modified which method. Note that this is an intentionally simple example to explain the concepts in the paper.

Figure 1 represents the support, confidence, and lift values for the MM matrix, where Ci represents *Circle.circumference()*, Cy – *Cylinder.area()*, Co – *Cone.area()*, and S – *Shape.draw()*.

TABLE I. Commits made by developers

Commit #	Developer	Description
C_1	Alice	Change type of function parameter to compute the radius (Circle) and how to render it (in Shape)
C_2	Carlos	Change the side of Cone and how to render it
C_3	Alice	Change how a Shape is rendered
C_4	Alice	Calculation of how circumference and area are calculated using PI. Required modification on how to draw a Shape
C_5	Bob	Modify the height calculation of a cylinder and how it is rendered

TABLE II. Methods changed for commit

Commit #	Circle circumf()	Cylinder area()	Cone area()	Shape draw()
C_1	1	1	0	1
C_2	0	0	1	1
C_3	0	0	0	1
C_4	1	1	1	1
C_5	0	1	0	1

1 Derby Repository: https://github.com/apache/derby

$$\begin{array}{c|cccc} & Ci & Cy & Co & S \\ \hline Ci & 2 & 2 & 1 & 2 \\ Cy & 2 & 3 & 1 & 3 \\ Co & 1 & 1 & 2 & 2 \\ S & 2 & 3 & 2 & 5 \end{array} \quad \begin{array}{c|cccc} & Ci & Cy & Co & S \\ \hline Ci & 1 & 1 & 0.5 & 1 \\ Cy & 0.6 & 1 & 0.3 & 1 \\ Co & 0.5 & 0.5 & 1 & 1 \\ S & 0.4 & 0.6 & 0.4 & 1 \end{array} \quad \begin{array}{c|cccc} & Ci & Cy & Co & S \\ \hline Ci & 2.5 & 1.6 & 1.2 & 1 \\ Cy & 1.6 & 1.6 & 0.7 & 1 \\ Co & 1.2 & 0.8 & 2.5 & 1 \\ S & 1 & 1 & 1 & 1 \end{array}$$

Support Confidence Lift

Figure 1. Support, Confidence, and Lift calculated from previous scenario.

If we consider the confidence matrix, we notice that the dependency from *Cy* to *Ci* (100% conf.) is stronger than from *Ci* to *Cy* (60% conf.), because whenever *Ci* was changed it also required changes to *Cy* (commits C_1 and C_4) (see Table II). However, *Cy* was changed once without *Ci* (commit C_5). With such a confidence analysis we can state that *Cy* (always) depends on *Ci*, but *Ci* does not necessarily depend on *Cy*. Therefore, using confidence to derive the DD matrix would identify that Bob should communicate with Alice, but not necessary the other way around.

The confidence matrix also indicates high dependency from *S* to all other methods. However, this occurs not because *S* really depends on all other methods, but because *S* was changed in all commits, independently (see Table II). The lift matrix eliminates such coincidental dependencies, keeping only dependencies between *Cy* and *Ci*, and *Co* and *Ci*, since all other values are either equal to or below 1.

In summary, support alone is not sufficient to indicate dependencies among project entities, but helps in eliminating dependencies that appear by chance (e.g., *Co* and *Ci*). In a large project with thousands of commits thresholding on a predefined support level can help eliminate accidental dependencies. On the other hand, lift plays a complementary role of identifying dependencies to elements that are very frequent (e.g., *S*) and therefore may be a cause of coincidental changes. Finally, confidence is important to identify the direction of the dependency (e.g., from Cy to Ci). With such an analysis, we find that the only real dependency in our scenario is from Cy to Ci, which would lead to a communication requirement from Bob to Alice in the DD matrix.

Our approach, therefore, provides four distinct advantages. First, the confidence measure allows more nuanced investigations (e.g., direction of dependency). Second, the use of lift measure increases the accuracy of the finding by filtering out common, but unrelated changes. Third, the fine-grained analysis from the method level increases accuracy, since we can identify dependencies among individual methods. Therefore, if we find that *Cy* depends on *Ci*, we can find that Bob needs to coordinate with Alice who is working on *Ci* and not another developer who is working on the same file (Circle, but on a different method). Finally, GPU processing allows these investigations to be performed interactively.

B. *Derby Analysis*

In order to evaluate Dominoes in a real setting, we applied it over Apache Derby, an open source project. This evaluation aims to demonstrate how solely working with support is error prone for finding artifact dependencies. All analyses were made considering the repository data from 08/11/2004 to 01/23/2014, which comprises **7,578** commits, **36** distinct developers, **34,335** files, and **305,551** methods committed during approximately 10 years. This evaluation was performed at the file-level for easier interpretation of the results. However, as discussed before, Dominoes can easily navigate from coarse to fine grained analysis and vice-versa.

After processing the data, we found that due to the project characteristics of Derby, the lift analysis does not filter out any coincidental dependencies. This is because the Derby file dependencies are highly clustered, causing low support and a very high lift. When we filter the lift values by thresholding on 1, no data points were eliminated. Therefore, we continued with the support/confidence analysis.

Table III presents the top 5 logical dependencies in terms of support and with the biggest difference in confidence, considering a support threshold above 30. It is important to remember that confidence is not transitive.

Considering the first case as an example, it is possible to observe that using the common approach that is based on support, artifacts DataDictionary.java and DataDictionary-Impl.java would be considered as dependent to each other as they have a high support (in fact, 79 is the highest value of absolute support in the whole system). However, when observing the confidence, it is possible to see that only Data-DictionaryImpl.java has dependency with DataDictionary.java, which is reasonable as changing a method implementation normally does not result a change to its interface. The following two rows are also interface/implementation cases, presenting the same behavior. In the fourth row, we have a composition case, where DRDAConnThread.java possesses a DRDAStatement.java instance. In this case, modifying the latter does not necessary imply a modifications in the former. However, there is a high likelihood of a related change in the opposite direction, that is, modifications in DRDA-Statement.java can change method signatures used by DRDAConnThread.java, for instance.

Finally, the last case is a class specialization, thus normally requiring modification to both files, with a slightly higher dependence from the subclass to the superclass. These analyses show the importance of using confidence to identify the direction of the dependencies.

TABLE III. TOP 5 LOGICAL DEPENDENCIES IN TERMS OF HIGH SUPPORT AND BIGGEST CONFIDENCE DIFFERENCE

Artifact A	Artifact B	Support	Conf. (A-B)	Conf. (B-A)
DataDictionary.java	DataDictionaryImpl.java	79	0.88	0.37
DD_Version.java	DataDictionaryImpl.java	45	0.78	0.21
LanguageConnectionContext.java	GenericLanguageConnectionContext.java	44	0.86	0.48
DRDAConnThread.java	DRDAStatement.java	37	0.22	0.68
ResultSetNode.java	SelectNode.java	36	0.54	0.45

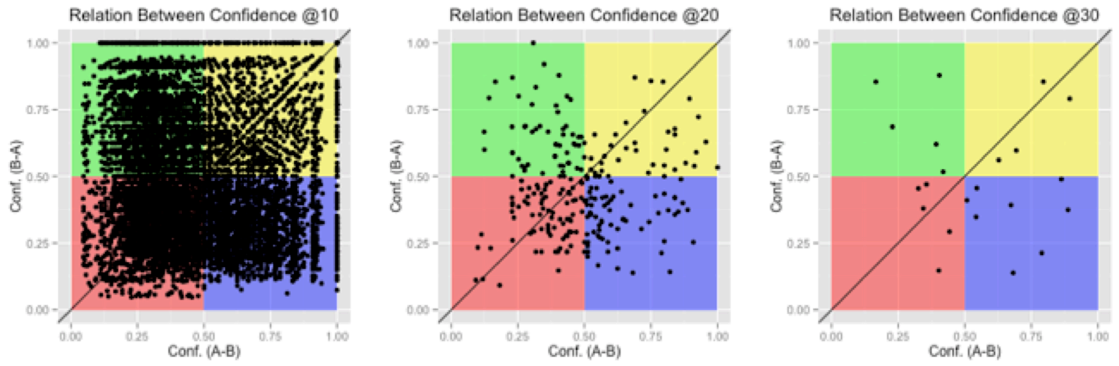

Figure 2. Relation among confidence for various support threshold. The leftmost chart considers a threshold of 10, while the middle uses 20, and finally the rightmost uses 30.

Besides these five top dependencies, Figure 2 presents a scatter plot chart with all dependencies at three specific support levels (10, 20, and 30). This chart plots each dependency according to its confidence in both directions (A-B and B-A). This way, dependencies with the same confidence value in both directions are plotted along the diagonal. As we can see, however, there are several cases where points are located far from the diagonal. When we consider the rightmost chart in Figure 2 (support threshold at 30) for discussion we can observe some distinct patterns. The red quadrant shows that both **conf(A-B)** and **conf(B-A)** are less than 0.5, thus containing weak bidirectional dependencies. The yellow quadrant, on the other hand, shows dependencies where both **conf(A-B)** and **conf(B-A)** are above 0.5, thus containing strong bidirectional dependencies. Finally, the green and blue quadrants show unidirectional dependencies with highest divergences among confidence. In this case, dependencies from these quadrants can be erroneously classified as bidirectional if we are to analyze dependencies solely by support.

Performing an analysis such as the one in Figure 2 can unveil how inaccurate dependencies extracted from support-based approaches tend to be. As demonstrated for the Derby project, only the yellow-quadrant dependencies should be classified as bidirectional. Both blue and green quadrants present unidirectional dependencies.

In this evaluation, the FC (i.e., [file|commit]) matrix was of size 34,335×7,578. The generation of FC^{sup} and FC^{conf} using GPU (NVidia GeForce GTX580) took about 0.7 minutes. However, doing the same computation using CPU (Intel Core 2 Quad Q6600) would take 696 minutes. This shows that we got a speedup of three orders of magnitude when GPU is in place – with just the simple calculation that requires 1 transposition and 3 multiplication operations. Similarly, when we process MC (i.e., [method|commit]), which is 305,551×7,578, it takes about 5 minutes in GPU, being impossible to be processed on CPU in a reasonable amount of time.

C. Dominoes Architecture

Dominoes Architecture is designed in a way that data from a software project repository is extracted and the associated change information is archived. Currently, we are mining projects that use Git by cloning and accessing the local repo. The local repo is then preprocessed to generate a tree of all modifications performed in all commits by analyzing which files, packages, classes, and methods were modified. It is important to note that information of each modification is decomposed to get a fine-grained view of the changes by using the Eclipse ASTParser (suitable for Java-based projects). For example, even if we represent changes at the package level (for a coarse-grained analysis), we know exactly which class was modified, as well the methods. This information is then stored in a relational database. Furthermore, after the initial data collection, information about subsequent changes can be updated incrementally to the database.

Basic Dominoes tiles are then constructed on the fly and become available to be used. Depending on the type of operation required by the user, these domino tiles are sent to the GPU, which performs the desired operations to generate the derived domino tiles. These derived domino tiles can also be saved as a template piece, should that piece be used extensively in calculations.

IV. Related Work

Our related work can be divided in two main groups: approaches that determine dependencies amongst artifacts or developers and approaches that provide support for exploratory analysis. There are numerous approaches that focus on identifying structural dependencies (through syntactic analysis) or logical dependencies (through change history) amongst artifacts. Cataldo et al. [12] stands out as they use matrices to process dependencies among developers based on dependencies among artifacts. In their approach, both structural dependencies and logical dependencies become Task Dependency (T_D) matrices, and change requests, associating developers to artifacts, becomes Task Assignment (T_A) matrix. These matrices are used in an equation that indicates coordination requirements $T_A \times T_D \times T_A^T$. Our approach generalizes this idea by allowing different kinds of exploration over matrices. Finally, our identification of relationships is innovative, as it allows combining support, confidence, and lift, to compose the dependency matrix depending on the research need.

Tools that enable exploratory analysis provide either predefined questions or are very limited to derive information that was not conceived beforehand. In the case of Tesseract [2], for example, the available relationships are preprocessed and the matrices are fixed at a coarse grain (file-file, file-developer, file-bug, bug-developer). CodeBook [3] is a similar approach that builds a graph of all relationship and then provides applications for answering specific questions (e.g., identifying related developers or artifacts). Gall et al. [13] built a tool for mining software archives at a fine grain in order to compare source code changes. From these analyses, recommendations such as change type patterns and consistency of changes can be made. Instead of a recommendation system, Dominoes provides a generic and flexible platform for exploratory analysis of project elements at a fine-grain level, which is compatible with multiple data types and relationships. Its interactive capabilities are mainly possible due to the adoption of GPU. It is important to state we have already used GPU for solving software engineering problems. In a previous work [6] we achieved boosts of two orders of magnitude when running image diff, patch, and merge operations in GPU.

V. CONCLUSION

Dominoes is an exploratory data analysis approach that allows users to select information about different project elements and their interrelationships from a repository. Relationships are represented by matrices, defined as *basic building tiles* and *derived building tiles*. Both kinds of building tiles can be combined iteratively to reveal deeper, complex relationships. Through such explorations, relationships that have not been computed or published before can be discovered through the operations over these building tiles. As all operations are performed in parallel over GPU, exploratory analysis can occur seamlessly at real time, even when computing relationships in fine-grained data. The current version of Dominoes tool extracts data from a Git repository and operates over the matrices by using GPU kernels in CUDA.

Our evaluation contrasted the use of support alone and the use of support and confidence to distinguish the dependence directions. In the Derby case, employing confidence leads to a more accurate analysis for finding dependencies among artifacts. Moreover, using confidence for thresholding a relationship is more natural for the user, as it represents a normalized value.

The Dominoes architecture was intentionally designed to easily accommodate the definition of new *basic building tiles*, such as relationships mined from communication channels (e.g., email, chat, dissuasion forums). The same extensibility feature also applies for operations. Besides the basic matrix operations, such as multiplication and transpose, specialized operations can also be plugged into Dominoes, as showed in section III-B for support, confidence, and lift. This leads to a relevant approach for the scientific community, as empirical studies can be reproduced over different corpora in order to validate an investigation. This has the potential of alleviating the pain of setting up an environment for each trial of an investigation.

Although we currently use matrices and GPU underneath Dominoes, other data representations and execution environments could be adopted in the future. For example, relational algebra is a compelling alternative to link sparse data. Moreover, SMP is the *de facto* architecture of modern personal computers. However, some kinds of analysis, such as reachability (used in impact analysis), can heavily benefit by operating over matrices in GPU. Besides that, due to the characteristic of our data, methods to deal with sparse matrix in order to reduce the memory usage can be adopted.

A concept not discussed in this paper, which is currently under development, is the use of three-dimensional (3D) building blocks. These 3D building blocks consider time as the third dimension over the matrices. In our experience, using this additional dimension collected through a specific time window allows observing the evolution of relationships over time. Another ongoing work is real time visualizations of basic and derived building tiles both in two and three dimensions to support explorations by end users on their own data.

ACKNOWLEDGMENT

This work is partially supported by CNPq, CAPES, and FAPERJ and NSF through awards: HCC- 1110916 and CCF-1253786.

REFERENCES

[1] T. Fritz and G. C. Murphy, "Using Information Fragments to Answer the Questions Developers Ask," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, New York, NY, USA, 2010, pp. 175–184.

[2] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 23–33.

[3] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and Exploiting Relationships in Software Repositories," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, New York, NY, USA, 2010, pp. 125–134.

[4] S. Minto and G. C. Murphy, "Recommending Emergent Teams," in *Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07*, 2007, pp. 5–5.

[5] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *ACM SIGGRAPH 2003 Papers*, New York, NY, USA, 2003, pp. 908–916.

[6] J. R. da Silva, T. Pacheco, E. Clua, and L. Murta, "A GPU-based Architecture for Parallel Image-aware Version Control," in *2011 15th European Conference on Software Maintenance and Reengineering*, Los Alamitos, CA, USA, 2012, vol. 0, pp. 191–200.

[7] S. Rajasekaran, L. Fiondella, M. Ahmed, and R. A. Ammar, *Multicore Computing: Algorithms, Architectures, and Applications*. New York, NY: CRC Press, 2013.

[8] D. V. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Trans. Eng. Manag.*, vol. EM-28, no. 3, pp. 71–74, 1981.

[9] S. Tuffery, *Data mining and statistics for decision making*. Chichester, West Sussex; Hoboken, NJ.: Wiley, 2011.

[10] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Softw. Eng. IEEE Trans. On*, vol. 31, no. 6, pp. 429–445, 2005.

[11] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1994, pp. 487–499.

[12] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, New York, NY, USA, 2008, pp. 2–11.

[13] H. C. Gall, B. Fluri, and M. Pinzger, "Change Analysis with Evolizer and ChangeDistiller," *IEEE Softw*, vol. 26, no. 1, pp. 26–33, Jan. 2009.