Efficient Calculation and Simulation of Product Cost Leveraging In-Memory Technology and Coprocessors

Christian Schwarz, Christopher Schmidt, Michael Hopstock, Werner Sinzig, Hasso Plattner
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
Email: {christian.schwarz, werner.sinzig, hasso.plattner}@hpi.de, {christopher.schmidt, michael.hopstock}@student.hpi.de

***Abstract*—Determination of product cost and resource consumption during manufacturing, is a crucial scenario for manufacturing companies. While enterprises have the required data for these calculations already, the computational complexity makes it still hard to build interactive applications that can be used for end-to-end simulation, from procurement to sales. Nevertheless, these applications are required for collaborative product cost simulations that support business experts to make fact-based and data-driven decisions. Within this paper, we present the generic calculation engine that can calculate product cost and other resource based features of thousands of products within the time-frame requirements for interactive applications efficiently. The engine leverages the potential of today's high-end in-memory databases in combination with the computational power of coprocessors. To solve the problem, we transfer the input drivers and their interdependencies into a system of equations that can be solved by either the coprocessor or a large scale multiprocessor system. For our evaluation, we use actual enterprise data of a Fortune 200 company also providing the scenario. We evaluate the approach based on this data, comparing our enterprise application specific problem solution with standard solution techniques of the same domain.**

Keywords–in-memory database; enterprise coprocessing; business data processing; product cost calculation.

I. Introduction

The knowledge about the costs of products and services offered by a company is mandatory information to guarantee the long-term success of a company. Therefore, the ability to calculate product cost based on product and manufacturing data is an integral part of enterprise systems. Cost drivers, like material prices, labor cost, cost of machinery and others, together with structural data, such as bill of material, and process data, such as routing, influence the cost of a product. Being able to determine the effects of changes of these cost drivers presents an advantage for decision makers, as they can make fact-based and data-driven decisions for the business how to best react to these changes. Enterprise systems already store the majority of the information required to calculate these costs.

State-of-the-art enterprise systems use in-memory databases to store and analyse huge amounts of data [1]. Those database systems are able to handle high volume workloads, while still guaranteeing maximum performance for database queries [2]. With the emerging trend of predictive analytics and complex simulation models requiring more business data, application logic is being transferred to the database execution level to avoid expensive data transfer. In addition, one can benefit from the available computational resources of the database server, reducing requirements on the client's side. SAP's HANA, one of the leading in-memory database systems in the enterprise application market, for example, offers a variety of tools to support developers with algorithms for predictive analytics [3]. While these algorithms benefit from the computational resources of the in-memory database system, they differ in computational complexity from transactional and analytical enterprise database operations.

The calculation of product cost fits into this category of enterprise application logic. To calculate the cost of a product, all relations can be expressed by a set of interdependant equations. The system of equations is then populated with data stored inside the database system and needs to be solved numerically in order to determine the cost of every raw material, semi-finished and finished good.

Within our research, we focused on calculating product features, such as the costs of companies' products, enabling companies to run collaborative planning sessions, including real-time simulation of changes to influential feature drivers. While today's product cost calculation systems focus on the influence of changes on single products, our system aims to provide end-to-end simulation capabilities. Therefore, we use transactional enterprise data to build a matrix representation of the relations between business entities. To solve the calculation problem as fast as possible, we make extensive use of data parallelization techniques and use coprocessors, such as Intel's Xeon Phi [4] and Nvidia's 2nd Generation Maxwell graphic processing units (GPUs) [5]. The use of these coprocessors is beneficial in our scenario for two major reasons: performance-price ratio and additional level of scalability.

These coprocessors offer computational resources for a subset of enterprise problems for lower prices than CPUs. A modern CPU (Intel Xeon E7-8890-v3 [6]) has a theoretical performance peak at 558 GFLOPs for a price of $7,488, while a current coporcessor (Nvida GeForce GTX Titan X[5]) offers 6,600 GFLOPs for $999. Even under the consideration that coprocessors cannot be used for all operations a traditional CPU can execute, the approximately 90 times lower price for performance ratio makes the coprocessor a valid option to add computational resources to the database node.

While the coprocessor calculates the product features, the CPUs of the database server are available for other computational tasks, such as database transactions. This becomes even more important, as enterprise systems have to handle the workload of many hundreds and thousands of users. All results in this paper are based on a dataset we got from a Fortune 200 company, manufacturing goods for the consumer packaged goods market.

The remainder of this paper is organized as follows: Section II gives an overview of related work in the area of linear equation system solving, followed by Section III presenting the calculation logic of product cost in detail. Section IV describes the insides of the implemented inversion algorithms and other important parts of the prototype. These implementations are evaluated in Section V. The paper concludes and gives an outlook on future work in Section VI.

II. Related Work

As the costs of a product can be expressed as a system of linear equations, we will present related work in the area of equation solving. The optimization of linear equation solving is a well-established field of research. Linear equations are known to be executable efficiently on GPUs, in either format as sparse or dense matrices [7], [8], [9]. To solve both sides of a linear equation system, matrices need to be inverted. Ezzatti et al. [10] compared CPU only, GPU only and hybrid implementations of matrix inversion, using an LU factorization from LAPACK [11] and the Gauss-Jordan elimination algorithm. Their investigation indicates that the Gauss-Jordan elimination is a well-suited algorithm for parallel computing. Additionally they show that hybrid implementations, exploiting the underlying platform features can still outperform pure GPU versions.

One issue arising with transferring computations to a coprocessor are the memory constraints. In comparison to today's available large main memory systems, the memory of coprocessors is still rather small. To reduce the memory required during the inversion of matrices, DasGupta [12] proposes a modified version of the Gauss-Jordan elimination calculating the inverse within the original matrix space. Another possible approach to solve this issue is to split the calculation into sub matrices [13].

A different approach to solve linear equations in a general form is to use linear solvers instead of an inversion and matrix-vector multiplication. Krüger and Westermann [14] have proposed a framework for implementing linear algebra operators on GPU and used it to implement an efficient sparse conjugate gradient solver. More work on linear solvers on GPU includes Tomov et al. [15], who present solver implementations on hybrid systems, which are GPU accelerated.

For our work, we focused on using the inverse calculation approach, as it is beneficial for our use case in two aspects. Firstly, we only have to calculate the inverse of the matrix once, while the calculation is executed multiple times, either for different input parameters or different features. This reduces the overall application runtime overhead of the inverse calculation in comparison to the interactive simulation process. Lastly, the inverse matrix can be used for fast search of materials and cost center activities, as it condenses all information for specific materials and cost center activities either in its corresponding rows or columns. Thereby the inverse matrix enables fast filtering on materials that is not required by the product cost calculation directly, but it enables an interactive navigation through the material database. The inverted matrix can even be used to reduce the load on the database management system, by providing an index-like structure for the material hierarchy, at the cost of additional memory requirements. The additional memory is also required for the algorithms in any case, which is why it is not considered to be a drawback.

III. Product Cost Calculation

While we aim to calculate multiple product features, including carbon dioxide, water requirements and energy usage, which indicate the environmental impact of a product, we started with the calculation of monetary features of a product within the prototype based on the data obtained. Nevertheless, the calculation of product cost is a common scenario in enterprises and therefore poses a relevant use case to build a prototype on. Based on the data and complexity of today's products, this calculation is computational intensive. We present a solution for the problem, using parallelizable algorithms for modern hardware. All manufacturing and non-manufacturing related expenses of a company determine the costs of a product. The prototype enables a group of experts to change cost drivers according to predictions and to evaluate the influence on product cost instantly. Due to the immediate feedback, these planning sessions become interactive and collaborative.

Within this section, we present the calculation process of the prototype, the equations used for the calculation, and the real dataset provided by our industry partner.

A. Multiphase Product Cost Calculation

To demonstrate the feasibility of using coprocessors to solve enterprise equation systems, we implemented a prototype focusing on manufacturing related product cost of a company. The prototype is split into a lightweight user interface for result visualization and a server application executing all calculations for the requested production periods. We aim to enable the user to calculate and modify product cost and single cost components to support an interactive simulation scenario. The logic executed in the server could be integrated at a later stage into procedures for the database system. At the time of writing, we left this open for future work.

We use the transactional data entered and stored in an enterprise resource planning (ERP) system as a base for the calculation of product cost. The data contains information about raw material prices, operational expenses, product design, production process, foreign currency exchange rates, and many more. These cost drivers are parts of the equation system.

The product cost calculation process is split into three phases: virtual data model creation, data retrieval and homogenization, and cost calculation.

1) Virtual Data Model Creation: The virtual model is created during the design phase of the application. It is required to create a simplified view on the ERP data model to be accessible for the product cost calculation. Within this phase, we create non-materialized views that fit the needs of data granularity, based on tables of an ERP system. These views are a logical description that is populated during query execution time, rather than having a materialized representation inside the database.

A simplified version of these views used within the prototype is depicted in Figure 1. For ease of readability, text fields

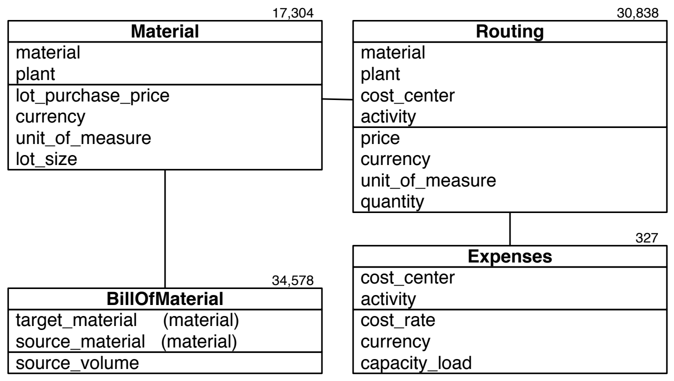

Figure 1. Simplified Virtual Data Schema

and additional information are removed from the illustrated model.

In the simulation prototype, the data is stored within a database and retrieved from it.

a) Material: The *Material* view contains the data of all materials. Information about lot purchase price and corresponding currency, lot size and unit of measure is available in the view.

b) Bill of Material: The *Bill of Material* (BOM) contains the list of all raw materials, parts, intermediates, subassemblies, commodities, semi-finished goods and finished goods required to construct or repair a product. Therefore, it stores the relational network between materials. Specifically, the number of units of a source material that is required to produce a lot of the target material is represented in the table. Furthermore, the BOM can be seen as a directed graph, having nodes representing materials and edges representing amounts required to produce the material at the end of that edge.

c) Routing: Routing describes how many units of a specific cost center activity are required during the manufacturing process of a specific material. In addition, the costs per unit and additional meta-data are stored.

d) Expenses: The *Expenses* view contains all expected expenses by general ledger account for the combination of cost center, activity, and work center. These expenses are not necessarily bound to a specific manufacturing step or material.

2) Data Retrieval and Homogenization Phase: The data retrieval and homogenization phase relies on features available in the in-memory database engine. In this phase, the views are requested and executed to transfer the data to the application. During the query execution, all volumes are translated into their base measure and are unified into a common target size, either lots or units. We decided to convert everything into lots, due to the more meaningful values for business users. Currencies are converted using SAP HANAs currency conversion feature [1]. The majority of the data is retrieved at application start time and is prepared for the feature calculation. At the end of this phase, a block matrix containing multiple weighted adjacency matrices is available inside the server application.

3) Cost Calculation: The Cost Calculation (CC) represents the simulation part of the application. The user changes input drivers for the calculated features. For the prototype, we decided to use monetary values, such as purchase price and cost center activity rates. These can also be replaced by other factors, while a change of the matrix part of the equation is not required. The influence on the costs for all materials is calculated and the result is returned to the client. The fast response time enables an interactive and collaborative use of the application. The calculations are either done on a multi core CPU or preferably on a coprocessor, like Intel's Xeon Phi or a GPGPU. Using the coprocessor for calculation decreases the load in the CPU, making resources available for database query processing. In addition, the simulation benefits from the parallel execution of the calculation on the coprocessor to deliver results within a smaller time-period than the CPU.

B. Problem Formalization

Multiple cost drivers, e.g., material prices, labor costs, machinery hours, transportation, and more influence product cost. Based on the calculated cost per unit, the costs of goods sold can be determined, applying this rate to the sales volume.

1) Bill of Material: An entry b_{st} in the bill of material describes how many units of material s are required to produce one unit of material t. Instead of units, lots can be used as well. S_t represents the set of materials required for a target material t, while T represents the set of all materials that are produced. T_s represents all materials t that require s for their production.

2) Routing: Routing A describes how many hours of a specific cost center activity are required during the manufacturing process for one lot of a target material. In particular, a_{it} represents the amount of a specific cost center activity a_i that is required to manufacture material t. The cost center activity rate is determined by r_i.

3) Manufacturing Costs: As an example feature, we calculate the manufacturing costs mc of a product t. These costs describe, how many working hours and how many source materials are required for the production of t. The purchase price pp is part of the equation to cover the manufacturing costs of raw materials.

$$mc_t = pp_t + \sum_{a_i \in A} a_{it} r_i + \sum_{s \in S_t} b_{st} mc_s \tag{1}$$

While 1 uses the purchase price as variable input factor for the cost feature, any other linear factors for other features could also be part of the equation.

4) Capacity Load: The capacity load of a cost center activity is defined by the sum of hours spent for the specific cost center activity during the production process. T_i represents the set of all materials that require cost center activity i.

$$l_i = \sum_{t \in T_i} \big(a_{it}(pd_t + md_t) \big) \tag{2}$$

The demand d_t of a material t is defined by two parts: primary demand and manufacturing demand. The primary demand pd_t determines the amount of material t that is sold to the markets. The manufacturing demand md_s determines the amount of material s that is required to manufacture all demands of materials t that require s during production cycle.

$$md_s = \sum_{t \in T_s} \big(b_{st}(pd_t + md_t) \big) \tag{3}$$

C. Solving the System of Equations

To solve the system of equations, multiple methods exist: using a linear system solver or by executing matrix operations. General feedback is that using a linear solver, like the funtions provided by Intel's Math Kernel library[16] (`sgetrf` to compute the LU decomposition, followed by `sgetrs` , which solves the linear equations), is preferable for performance reasons, especially if a matrix has to be inverted. This is due to the optimized implementations and reduced number of calculations required to solve an equation system. For the presented use case, we decided to go for the matrix operation path, because we saw benefits for operation parallelization. Based on the previously defined equations, all summations and multiplications can be done using a matrix-vector multiplication.

$$\begin{pmatrix} pp_t \\ pr_i \end{pmatrix} = \begin{pmatrix} b_{st} & a_{it} \\ 0 & l_i \end{pmatrix} \times \begin{pmatrix} mc_s \\ r_i \end{pmatrix} \tag{4}$$

The matrix stores all relevant relations between materials and cost center activities. It stores the adjacencies between these to be used for the calculation and is logically partitioned into four quadrants, as shown in4. For our implementation, we use row major ordering based to early experiences with the Xeon Phi and Intel's Math Kernel Library. In early experiments, we have seen that the usage of a transposed matrix vector multiplication was faster than matrix vector multiplication.

The first quadrant (upper left) represents the BOM. The quadrant is aligned by source materials on the vertical and target materials on the horizontal axis. The entries b_{st} represent the inventory change that will be applied if the specified source material is used for production.

The second quadrant (lower left) contains only 0 and is therefore called Zero.

The third quadrant (upper right) stores the routing information a_{it}. This represents the amount of hours spent during each manufacturing step.

The last quadrant (lower right) contains the capacity load l_i of all cost center activities. It represents the total amount of hours spent by a specific cost center activity in the time period stored inside the matrix. The numbers in this quadrant depend on the material demands and routing data factors of the other quadrants.

IV. Prototype Architecture and Algorithms

To calculate multiple versions of the manufacturing costs, the matrix presented in4 has to be inverted. Matrix inversion is hereby preferred to linear equation solvers, due to the number of different configurations of the equation system. To speed-up the cost intensive inversion process, the implemented inversion algorithm reuses knowledge about the matrix's quadrants and data stored inside the database. Within this section, we present some of the implemented inversion algorithms and other relevant parts of the architecture.

A. Prototype Architecture

To evaluate the approach of using a coprocessor for product feature calculation, we build the prototype depicted in Figure 2. A frontend application is provided to the user, enabling the user to configure and run simulation scenarios. The frontend application communicates via HTTPS with the backend that sends JSON responses. The frontend communicator is written in C++ and uses `mongoose` and `rapidjson` for data serialization. When a user triggers a simulation run, the simulator executes the given simulation scenario. The simulator fetches all required data from the database and sends it to the engine, which is the central computation unit. The engine itself consists of three subparts: the matrix inverter, the demand calculator, and the matrix-vector operator. The matrix inverter is responsible to execute an inversion algorithm and returns the inverse matrix that is used later on for the matrix-vector multiplication executed by the matrix-vector operator. The demand calculator is required to calculate the load for all cost center activities for the given production period, and it is part of the matrix inversion. All elements of the engine can be replaced by different implementations and are either executed on the CPU, the coprocessor, or both. The engine builder is responsible to set the implementations of the engines algorithms, to provide a user defined engine that can be used by the simulator. Within the prototype, the user is able to set different engines to enable a comparison of the implemented execution strategies. The database connector is responsible to fetch the data from the database. For our prototype, we used SAP's HANA in-memory columnar database. The database connector uses `nanodbc` to communicate with the database via ODBC.

B. Inversion Algorithms

To invert the matrix containing the enterprise equation system, we implemented several algorithms: Naiv inversion, AVX inversion, upper triangular transformation, and CUDA inversion. These algorithms are presented in detail within this section.

The current implementations require twice the amount of memory than what is required to store the data in dense matrix format, due to the implemented Gauss-Jordan inversion. The memory is required as space for a temporary working copy and the identity matrix of the same size as the original matrix. Each operation described later on is applied to the temporary copy and the identity matrix. Optimizations for the Gauss-Jordan elimination have been proposed by DasGupta [12] and Amestoy et al. [13] and can be applied in future versions of the inversion.

1) Naiv Inversion: At first, the quadrant containing the cost center load is inverted. Because only the diagonal value is set, the number of cost center activities determines the number of divisions required to finish this step. Based on initial performance measurements, an OpenMP `parallel for` pragma is used to parallelize the operation on a per cost center activity base (row level).

The next quadrant to be modified, contains the routing data. All columns of a row that are not equal to 0 get a multiple of the corresponding load line subtracted. Because the zero quadrant only contains zeros, we do not modify the bill of material quadrant in neither the temporary nor the identity matrix. Thus, additional computations can be eliminated, as the subtraction has to be done on the routing columns only. To speedup the execution, this step gets parallelized with an OpenMP `parallel for` pragma using the rows for parallelization.

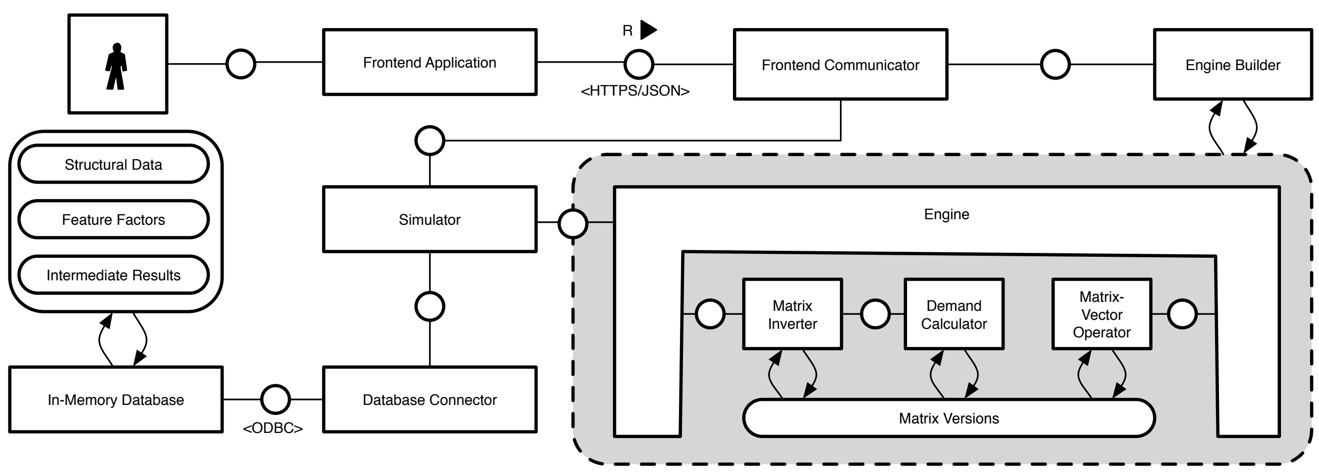

Figure 2. Architecture of the Product Feature Calculation Prototype

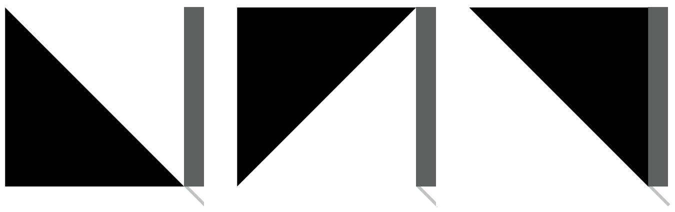

Figure 3. Matrix reordering enables the use of upper triangular matrix inversion

Finally, the BOM quadrant is converted to the identity form. This quadrant has two properties that are beneficial for the inversion process: the diagonal is set to 1.0, due to restrictions we applied during the virtual data model creation phase. This quadrant is also a lower triangular matrix. Thus the inversion can be done iteratively, starting at the first row and subtracting it from all other rows that have a value not equal to 0 at the corresponding position. This is done in parallel on row level, picking one row that can potentially be subtracted from all other rows. After this last step, an inverted matrix is calculated and the temporary copy of the matrix is removed.

2) AVX Inversion: During the second and third step of the naiv inversion, rows are subtracted from each other. This operation had a major performance impact on the inversion and therefore is a subject for performance optimization. To speedup the process, three changes to the initial implementation were made: the algorithm is NUMA aware, all memory is aligned to its AVX requirements, and we make use of Intels AVX instruction set, applying the same operation to multiple values at once. The implementation uses the functions `_mm256_set1_ps`, `_mm256_load_ps`, `_mm256_mul_ps`, `_mm256_sub_ps` and `_mm256_store_ps`, defined in `immintrin.h`.

3) CUDA Inversion: A CUDA-enabled implementation of the matrix inversion allows the execution of the algorithm on the GPU and outperforms a GPU baseline inversion algorithm from the Cula dense library [17]. The implementation focuses on reducing the access to global memory to gain performance. Utilizing the GPU's shared memory during the subtraction in the third step of the naiv inversion achieves the performance improvement.

4) Upper Triangular Transformation: In order to evaluate and compare the performance of our inversion algorithm, we required a baseline we could measure against. Based on the initial matrix format, we evaluated the LAPACK methods for general inversion (`sgetrf` and `sgetri`), which solved the problem. In order to use an optimized inverter implementation, we decided to implement row and column reordering for the inversion process, thus enabling the use of LAPACKs triangular matrix inversion.

The process of reordering is visualized in Figure 3. In the original format (left), the linear equation part of the bill of material quadrant (black) is a lower triangular matrix. During the first phase, the rows of the bill of material and routing (dark grey) are reordered (middle). In a second phase, the columns of the bill of material are reordered to convert the matrix into an upper triangular format (right). The quadrant containing the cost center load (light grey) is not modified during the conversion. White spaces contain only 0 values.

Afterwards we were able to use `strtri` for matrix inversion, which delivered faster results than the general inversion. In all measurements done for this paper, the time of row and column reordering is excluded and thus not part of the inversion time.

C. Scenario Data

Having described the construction of the matrix in detail, we will examine now the data sources for each part and give some characteristics of the dataset used by the current prototype. This dataset will later also provide the basis for the performance evaluation in Section V.

The data is a limited dataset provided by a manufacturing company. It contains data for a single location the company operates in. It consists of 17,304 materials and 327 cost center activities, resulting in a 17,631 $\times$ 17,631 sized matrix. Hence, the dominant part of the matrix is the bill of material, being over 50 times larger than the cost center activity related quadrants. Besides the general size of the matrix, it is also helpful to look at the number of raw materials, which is 6,379 and the number of products 2,978. This leaves 7,947 materials to be semi-finished goods and packaging materials. To get an understanding of the structure of the bill of material and the routing the following measures are helpful. Starting from products moving down the material hierarchy, there is an average depth between two and three levels of intermediate products included, with a maximum of five. This means when not using the matrix structure, but a tree-like structure, it is necessary to calculate the product cost for each of the intermediate products on the different levels, before the product cost for one product could be determined.

Another measure to consider is the ratio of non-zero values compared to zero values. On average, seven different raw materials or semi-finished goods are needed during each manufacturing step, resulting in a rather small number of values other than 0 in the bill of material quadrant. The maximum count is 27. Taking the average results in a ratio of non-zero data fields compared to data fields containing a zero of 1:2,472. Similar for the cost centers an average of six are required during each manufacturing step, with a maximum number of 39. The average ratio of non-zero data fields compared to data fields containing a zero is 1:55 within the routing quadrant. Therefore, the matrix in general is a sparse matrix.

We calculate the inverse of the matrix and it is not guaranteed that an inverse of a sparse matrix is also going to be a sparse matrix. While this may lead to additional overhead for runtime memory allocations for these new non-zero data fields in the inverted matrix, we consider a dense matrix for out implementation. When considering bringing the algorithm on to a GPU, this results in performance loss due to branching within kernels.

V. EVALUATION

Within this section, we will evaluate the central components of our simulation prototype: matrix inversion, and matrix-vector multiplication. To compare our solutions with standard methods, we compare our implementation with LAPACK implementations that are part of Intel's MKL [16]. All experiments were executed on a two-socket server, equiped with Intel Xeon E5-2640 CPUs. For the CUDA inversion, we used an Nvidia K20Xm Tesla card. The matrix-vector multiplication was executed on an Intel Xeon Phi 5110P.

A. Matrix Inversion

To calculate the product features based on different feature drivers, the equation system needs to be transformed. Therefore, the matrix representing the equation system is inverted. An additional inversion might also be necessary, if structural changes, e.g. product design changes or new manufacturing machinery, are part of the simulation process. Therefore, the inversion represents an important step during the feature calculation. The results for the presented algorithms are shown in Figure 4, depicting an average of 20 executions per algorithm.

To make sure to get as close to the theoretical performance as possible, we created a fine-tuned version of the AVX inversion algorithm. As a next step, the coprocessor based implementation needs to be tuned. It has to be noted that the system executed one inversion at a time exclusively, executing no other operations at the same time. Additional background load would decrease the performance of the CPU based inversion. To transfer the data to and from the coprocessor, additional 827 ms are required. In the case that the matrix-vector multiplication is run on the same coprocessor, the data

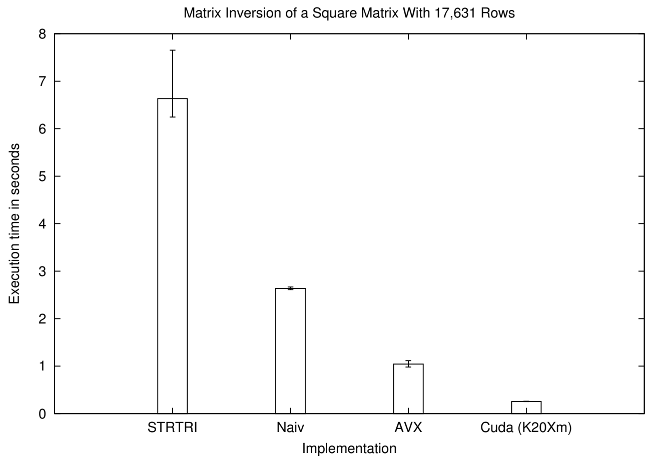

Figure 4. Performance evaluation for the matrix inversion algorithms

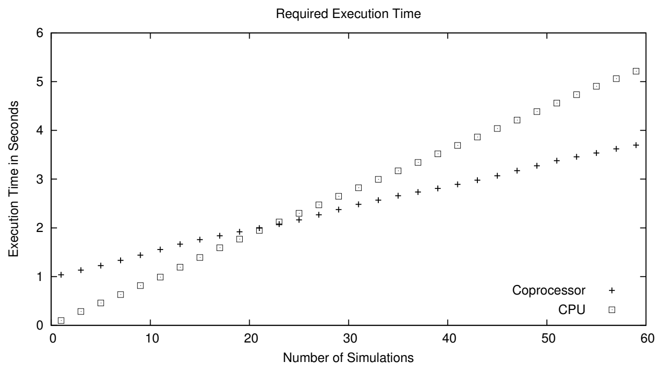

Figure 5. Performance evaluation for the execution of simulations.

can be reused and the transfer time gets amortized. Overall using available business knowledge leverages the performance of the matrix inversion compared to a standard library version and can be further improved by exploiting the underlying hardware.

B. Matrix-Vector Multiplication

The central component for simulating a new set of product feature input parameters is the matrix-vector multiplication. The results can be seen in Figure 5. The initial data placement to the coprocessor is dominating its execution time. The CPU has no off-load attached but the computational performance is lower than on the coprocessor, which is why the initial cost for offloading the inverted matrix gets amortized after 22 simulations.

Using the Intel library `pragma offload_transfer` preprocessor macros, we were able to offload the necessary simulation data in $0.98s$, using the Intel Xeon Phis PCIe 2.0 connection.

Offloading all data for every simulation is inefficient and reduces the performance advantages of the coprocessor. Since base and modified costs are the only changing features in our scenario, they need to be transferred for every single matrix-vector operation, while the inverted matrix has to be offloaded only once. This allows us to reduce the overall transfer overhead.

It has to be noted that there was no additional activity on the server during our tests, which would influence the execution performance of the CPU negatively.

VI. Conclusion and Future Work

With our prototype, we have shown that enterprise systems can benefit from the use of coprocessors. The usage of these specialized hardware components had a benefit from a runtime performance point of view. We showed with this prototype the influence of the increased performance from an application perspective based on actual enterprise data, enabling an interactive product feature simulation. While these results clearly demonstrate how a hybrid architecture, consisting of CPUs and coprocessors, can leverage business scenarios, such as the presented product feature calculation and simulation.

While the nature of database operations as parallelizable operations on sets and the parallel computing resources of coprocessors are a good fit, advanced application logic is different. To use the theoretical performance in an enterprise environment, data and algorithms have to be aligned to make efficient use of these parallel computation resources.

To solve this initial issue, supporting structures have to be defined, which enable the user to determine applications that will benefit from hybrid execution. Based on these findings, data structures have to be modeled that are suitable to deliver a data representation a coprocessor will be able to work on as directly as possible, reducing the upfront cost of execution time and memory during the virtual data model creation and data retrieval and homogenization phases. To avoid further continuous transfer of matrix versions between main memory and the coprocessors memory, the use of different sparse matrix representations might be necessary. Thereby the focus should not only be on memory consumption, but on the cost to construct them from the data fetched from an in-memory columnar database as well. Providing the enterprises with the ability to model different versions of the matrix, compare them with each other and even base their simulations upon them, enriches the foundation for their decision making. This extension requires to calculate the product features for multiple matrices at the same time, requiring additional memory, which will exceed the available memory on a coprocessor. While using sparse matrices relaxes the issue, an investigation on how to create an efficient versioned matrix data structure, which only requires to store a base matrix and the changes to it, is needed.

As enterprise systems tend to be used by hundreds to thousands of users in parallel, access to limited resources like coprocessors needs to be optimized. Multiple users might be able to share the same basic data structure to modify their application needs, such as the structural data used for the presented simulation scenario. This assumption has to be reconsidered once we allow users to alter these shared structures, such as the matrix encapsulating the equation system. To reduce the potential transfer overhead to the coprocessor, an intelligent data placement strategy reducing the effects of data transfers while considering the data of multiple users is required. Summarizing the results, we believe that coprocessors will play an essential role in the development of future enterprise applications.

Acknowledgement

The authors like to thank Alexander Franke and Cornelia Rehbein for helping with the implementation of the product feature simulation prototype and the corresponding calculation engine.

References

[1] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA Database - Data Management for Modern Business Applications," ACM Sigmod Record, vol. 40, no. 4, Dec. 2011, pp. 45–51.

[2] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in SAP HANA database: the end of a column store myth." in Proceedings of the 2012 ACM SIGMOD International Conference on Management of data. New York, New York, USA: ACM Press, May 2012, pp. 731–742.

[3] SAP SE, SAP HANA Predictive Analysis Library (PAL) - SPSS 11, 1st ed., Nov. 2015.

[4] Intel Corporation, "Xeon Phi Coprocessor x100 Product Family," Apr. 2015.

[5] NVIDIA Corporation. GeForce GTX TITAN X Specifications. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications [retrieved: Feb., 2016]

[6] Intel Corporation. Intel® Xeon® Processor E7-8890 v3 (45M Cache, 2.50 GHz) Data Sheet. [Online]. Available: http://ark.intel.com/de/products/84685/Intel-Xeon-Processor-E7-8890-v3-45M-Cache-2_50-GHz [retrieved: Feb., 2016]

[7] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Aug. 2007, pp. 1–12.

[8] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA Corporation., Tech. Rep. VR-2008-004,, Dec. 2008.

[9] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing Symmetric Dense Matrix-Vector Multiplication on GPUs," in 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Nov. 2011, pp. 1–10.

[10] P. Ezzatti, E. S. Quintana-Orti, and A. Remon, "Using graphics processors to accelerate the computation of the matrix inverse," The Journal of Supercomputing, vol. 58, no. 3, Apr. 2011, pp. 429–437.

[11] E. Anderson, Z. Bai, C. H. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. C. Sorensen, LAPACK Users' Guide, ser. Third Edition, Society for Industrial and Applied Mathematics Philadelphia, PA, USA, Aug. 1999.

[12] D. DasGupta, "In-Place Matrix Inversion by Modified Gauss-Jordan Algorithm," Applied Mathematics, vol. 04, no. 10, 2013, pp. 1392–1396.

[13] P. R. Amestoy, I. S. Duff, Y. Robert, F.-H. Rouet, and B. Ucar, "On computing inverse entries of a sparse matrix in an out-of-core environment," SIAM Journal on Scientific Computing, vol. 34, no. 4, Jul. 2012, pp. 1975–1999.

[14] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2003, Jul. 2003, pp. 908–916.

[15] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense Linear Algebra Solvers for Multicore with GPU Accelerators," IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, Apr. 2010, pp. 1–8.

[16] Intel Corporation, Intel Math Kernel Library Reference Manual - C, 11th ed., Intel Corporation, Aug. 2015.

[17] NVIDIA Corporation. CULA Tools: GPU-Accelerated Libraries. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications [retrieved: Feb., 2016]