Instructions

1. Do not remove this page. The Brevetto Web application will reject this document if this page is removed.
2. You must enable macros so the embedded checkboxes and radio buttons are enabled.
3. Fill in your invention description on the following pages. It is important to provide accurate and detailed information and that you answer ALL of the questions. This information will be appended to the questions answered online and will be used to evaluate your invention for potential filing as a patent application.
4. Review this document with your co-inventors.
5. Once complete, edit your Disclosure in the Brevetto Web application. Use the “Upload IDF Document” button on the “Upload IDF Document” tab to upload this document to your disclosure. You may upload revisions of this document prior to submitting it for approval to enable your co-inventors to view the document online while it is still being edited. Uploading revisions will overwrite what is currently on file in Brevetto.
6. To ensure complete functionality of the template in Microsoft Word, make sure to use Word 2010 and save the file locally in the Word Document (\*.docx) file format.

PLEASE NOTE: Contact the TAC for any issues relating to the Brevetto application by phone (e.g. your local prefix +-1234, choose language then option 1, 3, 4, 2) OR online at [**http://it.intel.com/**](http://it.intel.com/).  Click on **Software** then select **Brevetto** under the **Legal Apps** menu.  
  
    Click [Submit/View a Request]  
    Category="Application Software"   
    Sub-Category="L",  
    Item="Legal - Brevetto"

**ATTENTION**

**All inventors MUST complete questions A – D below.** If any inventor is employed by one of Intel’s European legal entities in Germany, Austria, Denmark, or France, each inventor MUST also answer questions E – H.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1st. inventor | | 2nd. inventor | | 3rd. inventor | 4th. inventor | | | |
| 1. Name (Last, First): | | | | | | | | |
| Rong, Hongbo | | Park, Jongsoo | | Smelyanskiy, Mikhail |  | | | |
| 1. Personnel number (Intel WWID, if Intel employee) | | | | | | | | |
| 11339958 | | 11380636 | | 10673623 |  | | | |
| 1. What percentage share of the invention at the time of submission do you hold? Please specify whole numbers only and the total for all inventors must add up to 100%. If not specified below, the default is an equal percentage share for all inventors. | | | | | | |
| % | | % | | % | % | | | |
| 1. In what country did you reside when the invention was created? | | | | | | | | |
| USA | | USA | | USA |  | | | |
| 1. At the time of the invention: Were you on a long- or short-term assignment to another legal entity? | | | | | | | | |
|  | |  | |  |  | | | |
| 1. Occupation / Position / Title within the company: (e.g. design engineer) | | | | | | | | |
| Research Scientist | | Research Scientist | | Principle Engineer |  | | | |
| 1. Was the invention developed in   (select one only) | *a. your direct work field?*  *b. different field of work at your employer?*  *c. other area which does not relate to your employer?* | | | | | |
|  | |  |  | | |  | |
| 1. Was the invention developed in response to a request from your employer to solve a particular problem? | | | | | | | | |
|  | |  | |  |  | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5th. inventor | | 6th. inventor | | 7th. inventor | 8th. inventor | | | |
| 1. Name (Last, First): | | | | | | | | |
|  | |  | |  |  | | | |
| 1. Personnel number (Intel WWID, if Intel employee) | | | | | | | | |
|  | |  | |  |  | | | |
| 1. What percentage share of the invention at the time of submission do you hold? Please specify whole numbers only and the total for all inventors must add up to 100%. If not specified below, the default is an equal percentage share for all inventors. | | | | | | |
| % | | % | | % | % | | | |
| 1. In what country did you reside when the invention was created? | | | | | | | | |
|  | |  | |  |  | | | |
| 1. At the time of the invention: Were you on a long- or short-term assignment to another legal entity? | | | | | | | | |
|  | |  | |  |  | | | |
| 1. Occupation / Position / Title within the company: (e.g. design engineer) | | | | | | | | |
|  | |  | |  |  | | | |
| 1. Was the invention developed in   (select one only) | *a. your direct work field?*  *b. different field of work at your employer?*  *c. other area which does not relate to your employer?* | | | | | |
|  | |  |  | | |  | |
| 1. Was the invention developed in response to a request from your employer to solve a particular problem? | | | | | | | | |
|  | |  | |  |  | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9th. inventor | | 10th. inventor | | 11th. inventor | 12th. inventor | | | |
| 1. Name (Last, First): | | | | | | | | |
|  | |  | |  |  | | | |
| 1. Personnel number (Intel WWID, if Intel employee) | | | | | | | | |
|  | |  | |  |  | | | |
| 1. What percentage share of the invention at the time of submission do you hold? Please specify whole numbers only and the total for all inventors must add up to 100%. If not specified below, the default is an equal percentage share for all inventors. | | | | | | |
| % | | % | | % | % | | | |
| 1. In what country did you reside when the invention was created? | | | | | | | | |
|  | |  | |  |  | | | |
| 1. At the time of the invention: Were you on a long- or short-term assignment to another legal entity? | | | | | | | | |
|  | |  | |  |  | | | |
| 1. Occupation / Position / Title within the company: (e.g. design engineer) | | | | | | | | |
|  | |  | |  |  | | | |
| 1. Was the invention developed in   (select one only) | *a. your direct work field?*  *b. different field of work at your employer?*  *c. other area which does not relate to your employer?* | | | | | |
|  | |  |  | | |  | |
| 1. Was the invention developed in response to a request from your employer to solve a particular problem? | | | | | | | | |
|  | |  | |  |  | | | |

Please provide a description of the invention and include the following information:

1. Describe your invention:

What problem(s) does your invention solve? Briefly describe the problem you are addressing and previous solution(s), if any:

|  |
| --- |
| **To Reviewers:** this re-submitted version addresses the request of the last review: *“*Please provide more information on how the compiler is going to figure out what the RUNTIME distribution of sparse elements will be.” We address this question in Section 2.c, highlighted in bold fonts. There is no other change to this disclosure.  **Automatic context-sensitive acceleration of sparse applications**  High-performance computing (HPC) on huge sparse data structures like graphs with trillions of nodes has emerged as a new area, and become increasingly important in big-data analytics, computational biology, web search, and knowledge discovery, etc. [1].  These sparse applications usually contain function calls to a HPC library, e.g. Intel Math Kernel Library (MKL), which contains a set of basic sparse kernels, including sparse matrix vector multiply (SpMV), sparse triangular solver, sparse matrix matrix multiply (SpMM), etc.  The problem to address in this invention is: *how to automatically optimize sparse applications for any architecture, and achieve close-to-ninja performance with high productivity*? This is an important goal of the Problem-Solving Environment (PSE) mega-impact project.  Unlike traditional HPC applications that deal with regular data structures, sparse computation has different challenges [2]:   1. Performance is bound by memory bandwidth.   Sparse computation typically has considerably lower arithmetic intensity and, therefore, its performance is limited by memory bandwidth. However, optimizing for the memory bandwidth is often harder than optimizing for compute-intensive code: the former usually involves non-local restructuring of the code, and frequent non-contiguous data access exacerbates this difficulty.   1. Parallelism is dependent on specific data.   Sparse inputs tend to vary in intensity across different parts. Thus these parts may have imbalanced workloads. They may also have different memory access patterns, and thus different amount of cache misses. Loops in some key sparse computation routines are constrained by data-specific dependencies, making parallelization difficult.  There are several known approaches to speed up sparse applications:   1. Manually optimize sparse applications by ninja experts.   This is by far the most widely followed approach, due to the difficulty of automatically optimizing sparse applications [1, 2, 3].  This manual optimization process is extremely time-consuming. It usually takes an expert several months or years to achieve close-to-limit performance for a new architecture. For example, it has taken two experts 1 year to optimize for KNC the High Performance Conjugate Gradient benchmark, which is considered a small sparse application. It will take substantially more time for other sparse solvers that have huge code bases, such as HYPRE, PETSc and Trillinos.   1. Sparse kernel libraries as dataflow graphs.   As we said, there are a set of basic kernels shared by various sparse applications. They can be manually optimized by experts, and built into a library. The approach of “libraries as dataflow graphs” [5] expresses each library function as a dataflow graph. A compiler can inline this graph into a sparse application calling it, and thus optimize the library function with the application together, within the specific context.  This approach, however, is disruptive: it requires an HPC library to be rewritten in dataflow components. It might be too risky to have such a surgery for mature products like MKL.   1. Sparse domain-specific languages.   Domain-specific languages [6, 7] provide some basic language constructs specific to sparse inputs and operations. A compiler can take advantage of the semantics of these language constructs, and performs optimizations.  HPC sparse applications, however, have universally been written in general-purpose languages based on high-performance libraries. We have not seen any real application (except a few tiny examples) written in sparse DSLs. DSLs have limited applicability, may have difficulty integrating with other languages, and they do not solve the problem how to optimize successive sparse library function calls. |

1. Your Idea – Provide a high-level summary of your idea that includes a figure or flowchart. Be sure to note differences between your idea and the previous solutions and to note what advantages your idea provides. Please break down the high-level summary of your idea into 3 portions as indicated by the 3 questions 2a, 2b, and 2c below:
2. What is the basic principle? Please explain in just a few sentences (do not provide results or use cases here).

|  |
| --- |
| This invention proposes an approach to automatically analyze and optimize sparse applications to achieve close-to-ninja performance.  The key characteristic is a compiler-based approach able to automatically analyze the access patterns to sparse data structures, and insert data-specific, sparse-input-aware, optimization directives into the user program.  We envision this approach will be particularly helpful for sparse applications where sparse data structures undergo no or very slow rate of change, a common scenario seen in many sparse applications, so that the overhead of the sparse-input-aware optimizations is amortized over large number of uses of the optimized code.  The approach includes the following:   1. A user writes an application in a high-level productivity language, calling into sparse library kernels. 2. A compiler automatically analyzes the application for the specific context around the sparse kernel calls. It then transparently inserts data-specific, sparse-input-aware, optimization directives into the user program. 3. A library expert writes the sparse kernels to be sensitive to the directives. Guided by the directives, the kernels can adjust their behavior for the specific context to achieve the maximum efficiency.   The directives express the ninja expertise of PCL (Parallel Computing Lab) that enables Intel systems stay at the top of TOP 500. They are summarized out of many representative sparse applications, including sparse solvers, physical simulation, molecular dynamics, etc. |

1. How is your invention better than the known solutions (answer can include results, applications, or use cases)?

|  |
| --- |
| 1. A compiler enables context-sensitive optimizations automatically. This contrasts sharply with the time-consuming, manual, ninja optimizations approach. 2. A context can cover many calls to the same or different library functions. The compiler makes these calls collaborate with each other by sharing the context information.   Even though each call still happens individually, its behavior is adjusted by the compiler to the specific context and cooperates with previous calls. For example, redundant computations between calls are eliminated by being done only once by one call, and reused by the other calls, under the control of the compiler.  For instance, when a sparse triangular solver is called repeatedly with the same input matrix in a context, it builds a concrete dependence graph only the first time it is called in this context, and reuses it other times.  For another instance, a function call can internally convert its input matrix into another sparse format for the best performance, and the next call to the same or another function can use the converted matrix directly without converting the original matrix again.   1. The compiler identifies frequent patterns between two successive calls with producer-consumer relationship, and controls the library to fuse them together. This removes (usually big) temporary data, saving memory bandwidth, a key optimization for sparse applications as their performance is bound by memory bandwidth. 2. The compiler identifies the usage of the results of a library function call, and controls the library to output the results with streaming stores if necessary, which saves half of the memory bandwidth. 3. The compiler identifies dead results of a library function call, and controls the library not to compute them at all, saving both computation and bandwidth. 4. High-performance libraries are leveraged, and made to be sensitive to contexts with only small changes. This is in sharp contrast with the disruptive library-as-task-graph approach. 5. A library function adjusts its behavior to its calling context, as passed from the compiler. Thus its behavior may adapt over time. For example, a sparse solver may rebuild a concrete dependence graph every 20 calls, directed by the compiler. 6. The compiler and the library are co-designed such that the compiler-generated context information can enable key ninja optimizations shared by many sparse applications, including sparse format conversion, concrete dependence graph building, scheduling and synchronization strategy, library function fusion, etc.   However, the compiler does not need to know the details how the context information is to be used by the library. It focuses on analyzing the context of the application, and gives only optimization directives, leaving the actual optimizations to be done inside the library. This makes the compiler’s job easy, and at the same time, leaves enough freedom to the library writer for optimizing the library with ninja expertise. |

1. Provide a more detailed description of your invention, highlighting what is new (please include block diagrams, process flow diagrams, etc., and limit the text to no more than 3 pages).

|  |
| --- |
| Below we first describe the invention. Then to show that it covers key optimizations in a wide range of sparse applications, we illustrate the invention with 5 important sparse applications, which are also representative of a class of other applications. They are Preconditioned Conjugate Gradient, Interior-Point Method, Newton-Krylove CFD Solver, Physical simulation, and Molecular Dynamics.  **DESCRIPTION**  Fig. 1. The overall workflow. Solid arrows: compile flow. Dotted arrows: execution flow.  Fig.1 shows the overall workflow of the invention. A user program contains calls to sparse functions in a high performance library. The program is preferably written in high-level productivity languages such as Julia, Python, R, Matlab, etc. so that it is easy to program and easy to do compiler analysis. The sparse functions may be called in a loop for many times. For example, a sparse linear solver may repeatedly call SpMV() and triangular solvers until a solution with expected accuracy is found. Also it is often to see that the result of a sparse function call might be immediately consumed by another sparse function call.  The compiler translates the user program into binary, during which it optimizes the sparse function calls. First, the compiler defines a “calling context” for the sparse function calls. The context is a scope of the program that surrounds the calls. It can be arbitrarily defined. For example, the whole program can be the context, the whole function enclosing the calls can be the context, a loop enclosing the calls can be the context, or each iteration of such a loop can be the context.  Once the context is fixed, the compiler can perform analyses specific to it. The compiler may identify what inputs to a sparse function call are constant within this context, i.e. never change in it. It might identify which input matrices have constant structures or constant values in the context. It may find out the sparse formats of the input matrices.  **Note: sparse format is how a sparse matrix is represented. There are many known representations, and user chooses one representation in writing a program. The representation is usually explicit, and thus it is easy for the compiler to know it. For example, in Julia language, a sparse matrix in CSC format is explicitly declared as SparseMatrixCSC. For another example, in MKL, one can call function sparseCreateCSRMatrix() to explicitly create a CSR format sparse matrix.**  **However, sparse format does NOT tell the RUNTIME distribution of the elements of the matrix. The compiler does not figure out the runtime distribution statically: instead, the sparse library function, when invoked at runtime the FIRST time, will figure out the distribution and may transform the matrix into another format for better performance.**  **For example, the original format of a sparse matrix A is CSR. When calling a library function SpMV() with A as input the first time, the library function may figure out that the CSR format does not have good spatial locality with the specific distribution of the elements of the matrix A. Therefore, the library function SpMV() might convert the format of A into another format, say ESB, for better spatial locality.**  **The compiler helps a library function to retain its reusable results. For the above example, it provides SpMV() with a data structure (a knob), to keep the converted format, so that when the function is invoked next time with the same matrix, no conversion is needed again.**  The compiler may also identify what outputs of a sparse function call are dead, i.e. never used in and outside the context.  Meanwhile, the compiler may gather some other statistics of the inputs and outputs, e.g. the average number of non-zeroes per row in a sparse matrix input, how far are the results of a function call got used, etc.  In addition, the compiler can match patterns in the program, e.g. *p*T*Ap*, where *A* is a sparse matrix and *p* a vector. A pattern may not have a direct implementation in the library, but it is equivalent to a series of function calls composed together. For example, *p*T*Ap*=InnerProduct(*p*, SpMV(*A, p*)). Here InnerProduct() consumes the result of SpMV(), and thus is the consumer of the latter. Some other patterns often seen in sparse applications are:  *|a – Ab|* // *a* and *b* are vectors. *A* is a sparse matrix.  *a* ± α*Ab* // *a* and *b* are vectors. *A* is a sparse matrix. αis a scalar constant.  *PTAP* // *P* and *A* are square sparse matrices  The compiler generates code for initializing a data structure, denoted as KNOB in this invention, with the analyses results for each sparse function call. By “KNOB”, we imagine a sparse library function is a black box but with a knob that the compiler can use to adjust its behavior.  Then the compiler generates code for each function call. The call is the same as usual, except that the initialized KNOB is added as an additional argument.  Finally, the compiler has generated a binary for the user program. When the binary is executed, the KNOB for each function call is initialized (Step 1 as shown in Fig.1). The KNOB has two parts: one part with public fields, and the other with private fields.  The public fields are visible to the compiler and the library function. The initialization code works on the public fields. They contain the following information found by the compiler: (1) the inputs of the function call that are constant in structure or value within the context, (2) the specific sparse formats of the input matrices (Note: there can be many possible sparse formats to represent the same data. Not every one of them is good for performance in the specific context), (3) other statistics of the inputs, e.g. the average number of non-zeroes per row in an input sparse matrix, (4) dead outputs of the function call, (5) other statistics of the outputs, e.g. use distance, which is a measure of how far the outputs will be used after the function returns, and (6) the consumer function and its parameters, if there is a consumer for this function call. The results of this function call are (part or all of) the parameters of the consumer function.  The private fields are visible to the library function only. When the function is invoked, it may compute some data and store them into these fields. These data are usually determined by the constant inputs, and thus they are constant across all the invocations of the function in the context. They need to be computed only once, when the function is invoked the first time it is called, and in the subsequent invocations of the function, they can simply be reused without re-computation.  Then the sparse function is invoked (Step 2 in Fig.1). The function reads the public fields from its KNOB (Step 3 in Fig.1), and starts computation. It can save some persistent data into the private fields of KNOB for reuse in future (Step 4 in Fig.1).  Conceptually, it looks like below:  function f(the original parameters, KNOB\* knob) {  **if** (knob != NULL) **then**  **for** any data that are completely determined by constant inputs **do**  **if** the data are not computed **then**  compute the data, and store into the private fields of knob  fetch the data from the private fields when the data are used  **for** any input matrix **do**  **if** its format has not been converted, and the current format  is not optimal for performance **then**  convert the current format to an optimal format  store the converted matrix into the private fields of knob  **else**  store the matrix into the private fields of knob  when the input matrix is used, use the matrix from the  private fields instead  **for** any dead output **do**  do not compute it  **if** (consumer function exists) **then**  invoke the consumer function with this function’s outputs as its inputs, and then take the consumer function’s outputs as this function’s outputs  **if** (use distance of the outputs exceeds a threshold) **then**  write the outputs in streaming stores to save memory bandwdith  behave as usual in all the other parts  **else**  behave as usual  }  The persistent, reusable data stored in the private fields of the KNOB enables the function to save time on computing the same data every time it is called. This is important, considering that sparse functions are often called repeatedly inside a loop until certain accuracy is reached. Such reusable data are specific to sparse applications, including the following:   1. Concrete dependence graph.   This dependence graph is different from the traditional dependence graph built in a compiler, where each graph node represents a program statement or instruction. Instead, a node in the concrete dependence graph represents a concrete iteration of a loop. This graph is built based on the concrete structures of the input matrices. For example, in a forward triangular solver, if the input matrix A has a non-zero element *A*[6, 2], then there is a dependence from node 2 (iteration 2) to node 6 (iteration 6).     1. Scheduling strategy.   Level scheduling is a commonly used strategy, which schedules the nodes in the concrete dependence graph level by level. It, however, may have limited parallelism. Multi-color reordering, in contrast, can schedule nodes as long as they are not directly connected. But it is an approximate heuristic and may degrade the convergence rate. Block multi-color reordering is a tradeoff between level scheduling and multi-color reordering.   1. Synchronization strategy.   Barrier can be inserted between each level. In contrast, in point-to-point (P2P), each node in the concrete dependence graph checks for a flag set by its parent.   1. Format-converted sparse input matrices.   The formats of the input matrices may have been converted to a format optimal for performance. A not-converted matrix can be treated as having been converted to itself. So all the input matrices can be recorded here.  All the above data, concrete dependence graph, scheduling and synchronization strategy, and converted input matrices, are determined by the inputs. If the inputs are constant in structure or value within the context, they may be computed only once, and reused in future iterations.  For dead outputs, the library function can simply do not compute them.  If the consumer function exists, this function and the consumer function can be fused together. Remember the consumer function is found by the compiler in identifying patterns. For example, the compiler found the pattern *p*T*Ap* in the program. Traditionally, we have to call SpMV(*A, p*) first, which saves its results into a temporary vector *v*, and then we have to call InnerProduct(*p, v*) to get the final result. With our invention, however, when SpMV() finishes computing a vector element, it reads the corresponding element in *p*, and multiply them. Finally the returned value would be *p*T*Ap*, instead of *Ap*, although the compiler generated only a call for computing *Ap* only, i.e. a call to SpMV() only.  In general, we can use a function pointer for the consumer function. However, since there are usually a limited number of frequent patterns related with the current function, there are also limited possibilities of consumer functions. So it is also possible to represent the consumer function as an integer like an enum in C. Then for each integer value, the corresponding consumer function like subtraction or multiplication can be directly implemented using machine instructions for the maximal performance, instead of calling the function through a pointer.  Sparse functions may share many features, including building concrete dependence graph, sparse matrices’ format conversion, scheduling, synchronization, and data transfer. These features can be grouped together as a runtime library, and they will be invoked when a sparse function runs (Step 5 in Fig.1).  **ILLUSTRATIONS**  Below we illustrate the invention with 5 representative sparse applications. We show a concrete example, Preconditioned Conjugate Gradient, in detail. We then show the others briefly, as they are similarly processed with the invention. These illustrations show that the invention covers a wide range of sparse applications.  Fig.2(a) shows the basic PCG algorithm. It spends most time in 2 sparse functions, as highlighted in colors: SpMV(), and a forward and backward solver. The forward and backward solver is especially time-consuming, as they have limited and fine-grain parallelism [3]. There are also 2 patterns: one is SpMV() followed by vector subtraction, the other is SpMV() followed by inner product.  To facilitate understanding without losing generality, we extract a skeleton of the algorithm, shown in Fig.2(b). We keep only the sparse function calls and the related patterns. Fig.2(c) shows the corresponding skeleton in a pseudo programming language.  As we said, the solvers are the most time-consuming due to dependence-limited parallelism. Take a forward solver shown in Fig.2(d) for example. There is a loop-carried dependence. Therefore, inside the forward solver, it builds a concrete dependence graph based on the sparse matrix *M*, and schedules the nodes according to the graph. The backward solver has to do similar things. Since the forward and backward solver is called within a loop, this can be costly.  However, as we can see from Fig.2(a), matrix *M* is not changed at all in the algorithm. Therefore, the dependence graph needs to be constructed only once, stored into a KNOB, and then it can be reused.  Fig.2(e) shows the optimized skeleton with the invention. For each function call, a KNOB is created, and context-specific contents are filled into the KNOB. The KNOBs are defined in a header file shown in Fig.2(f) . The meaning of the fields in a KNOB has been explained before.  Fig.2(g) shows the implementation of the functions. They have minimal changes to the original ninja implementation. The changes they have are the optimizations with the KNOBs, e.g. reusing the concrete dependence graph, etc.     1. The Preconditioned Conjugate Gradient algorithm [4]  1. A simplified skeleton of the above algorithm with the main operations  1. The simplified skeleton in a programming language        1. Forward solver. There is a loop-carried dependence as shown in an arrow.  1. The Preconditioned Conjugate Gradient skeleton with this invention. The new things are generated automatically by the compiler, and are highlighted in bold font.  1. The header file defining KNOBs and declaring functions.  1. The implementation of the functions   Fig.2 An illustrating example with Preconditioned Conjugate Gradient.  Below we briefly illustrate the invention with the other 4 important sparse applications that will also benefit from our approach:   1. Interior-Point Method (IPM).   This is used to solve optimization problems. In the past decade, the IPM has become a method of choice for solving large linear optimization problems of the form:  min *cTx*, subject to *Ax=b, x* ≥ 0.  Here the vector *x = (x1,…,xn)* is the optimization decision variable of the problem, the function *cTx* is the objective function, *A* is an *m* by *n* matrix of linear constraints, and vectors *x, c*, and *b* have appropriate dimensions. A vector *x\** is called an optimal solution of the optimization problem if it has the smallest objective value among all vectors that satisfy the constraints.  The figure below outlines the *k*’th iteration of the main optimization loop of IPM [8]. Note that typical IPM solver runs for up to 100 iterations.  The method starts with an initial approximation to the solution of the optimization problem, *x*. The core of the method is the main optimization loop, which updates the vector *x* at each iteration until the convergence to the optimal solution vector *x\** is achieved.    As the figure shows, IPM spends most of its computation time in a small number of linear algebra kernels. Optimizing and parallelizing these kernels is the key to efficient implementation of IPM. These are the most important kernels:   * Formulation of linear systems of equations, *Mx=b*, where *M* is the symmetric matrix of the form *M = A Z-1A*, where *A* is the original matrix of constraints. This requires a matrix-matrix multiplication operation. * Cholesky factorization of matrix *M = L LT* in order to solve the system of linear equations, *Mx=b*. Here *L* is the lower triangular, and *LT* is its transpose. This step is normally the most time-consuming step of the IPM. * Triangular solver uses the result of factorization to solve a system of linear equations *(L LT)x=b*, using the following two steps   a.       Forward solver solves *Ly=b*.  b.      Backward solver solves *LTx=z*.   * Matrix-vector multiplication (MVM) computes *Ax* and *ATx* for different vectors *x*. * Basic Linear Algebra Subroutines (BLAS1) performs inner products, vector additions, vector norm and ratio test computation.   Note that both the structure and the values of *A* matrix do not change across iterations of IPM. The structures of *M* and *L* don’t change either, but the values do change at each iteration. Hence optimizations of *Ax*, for example, can take advantage of both constant structure and constant non-zero values, while optimization of triangular solver ( *(L LT)x=b*) can only take advantage of non-changing structure of *L*.   1. Newton-Krylove CFD Solver   Computational aerodynamics is both an important and a representative workload for high performance computing, frequently employed in benchmarking and tuning hardware and software-programming environments. Implicit solution methods are typically used to solve CFD problem, modeled by partial differential equations (PDEs) with different temporal and spatial scales.  The method of choice today is Newton-Krylov method. Its example high level flow-chart is shown below. In each iteration, its time is flux calculation, solving sparse linear system of equations and gradient calculation. GMRES solver is used for linear solver, together with some ILU preconditioner.  The structure of system matrix that participates in the linear solver does not change across both inner and outer iterations, while the values change across outer iterations, but do not change across inner iterations. The same holds true for the pre-conditioner: the pre-conditioner structure does not change across outer iterations but values do change, since they are specific for each linear system being solved in the inner iterations of the loop.  cid:image004.png@01CFEBA1.CDFCEA00   1. Physical Simulation.   Rigid body dynamics simulates motion and interaction of non-deformable objects when forces and torques are present in the system. Rigid body dynamics is the most commonly used physical simulation in video games today. Examples of rigid bodies in games are vehicles, rag dolls, cranes, barrels, crates, and even whole buildings.  The traditional approach solves a system of ordinary differential equations, which represents Newton’s second law of motion, *F=ma*, where *m* is the mass of an object, *a* is its acceleration, and *F* is the applied force. The applied force determines the acceleration of the object, so velocity and position are obtained by integration of the above equation. The main computational challenge comes from the fact that rigid bodies’ motion is constrained due to their interaction with the environment. For example, consider a destructive environment in a video game where 1000s of rigid objects explode, collapse, and collide, resulting in 100,000s of interactive contacts. To realistically simulate such a scene requires determination of collisions, calculation of collision contact points, and physically correct computation of the contact forces that result from these contacts. To accelerate collision detection relies on spatial partitioning data structures, such as grids or bounding volume hierarchies. To determine contact forces that result from collision contact, we model the contact as a linear complementary problem (LCP). To solve linear complimentary problem one often uses projected Guass-Seidel linear solver. This solver is the crux of inner loop that is part of time-step outer loop. As opposed to two previous examples, the structure of the matrix used in LCP solver changes gradually across outer iterations, as simulation progresses in time. Change is gradual, because scene changes very slowly.   1. Molecular Dynamics (MD).   Molecular Dynamics is widely used to simulate many particle systems such as solids, liquids, and gases. During MD simulation, Newton’s equations of motion for the system are integrated numerically at every time step. For the system in equilibrium, one can study various static and dynamic properties such as temperature and pressure as well as heat transport.  cid:image007.jpg@01CFEBA9.95FA1A30  Simulation proceeds with each time step, as shown in figure above. At every time step for each particular one build neighbor list (“update neighbor list” in above figure), which is effectively sparse data structure (similar to column index in sparse MVM) that for each molecule identifies its neighbor within simulation grid. This neighbor list data structure changes very slowly with simulations, especially at low temperatures. Once build, NL is used to do force calculation. Force calculation, **(“get forces”** in the figure above) is effectively a sparse matrix vector multiplication algorithm, which uses neighbor list sparse data structure as indices. Often times, neighbor list is only rebuilt every 20 iterations or so, during which index structure does remains constant and is subject to the optimizations described in our invention. |

1. Which of our competitors are likely to use your idea or something similar?

|  |
| --- |
| NVIDIA is heavily investing on their HPC software package including sparse applications based on CUDA run-time. IBM, Oracle, and AMD are also potential competitors. |

1. How would we be able to determine if someone outside of Intel was using your idea (e.g. from visual inspection, from the product literature, from reverse engineering)?

|  |
| --- |
| From product literature, the library and compiler interface will reveal if our idea is used.  From reverse engineering, the interface, the persistent data used across the function calls, the related compiler analyses for contexts, the redundancy removal and fusion in executing the library functions will all reveal if the idea is used. |

1. Is this idea related to work performed in a Stand Development Organization (SDO) or Special Interest Group (SIG)? 

If YES, please answer the following (a, b, c):

1. Identify the SDO or SIG involved:

|  |
| --- |
|  |

1. Is this IDF prepared in anticipation of being part of an Intel proposal or submission for this SDO/SIG? 

If YES:

1. Identify the date of first disclosure to a party outside of Intel and outside of a confidentiality agreement (actual or anticipated):

|  |
| --- |
|  |

1. Provide the name of the person(s) driving Intel’s efforts for this SDO/SIG that have been made aware of this idea, if any:

|  |
| --- |
|  |

1. Is there an implementation of this idea which might not be covered by a standard requirement?

|  |
| --- |
|  |

1. To the best of your knowledge, identify any other pertinent information related to your idea.

|  |
| --- |
| 1. John R.Gilbert, Steve Reinhardt, Viral B.Shah. High-performance graph algorithms from parallel sparse matrices. Applied Parallel Computing. Lecture Notes in Computer Science, Vol.4699 2007, pp 260-269 2. Xing Liu, Mikhail Smelyanskiy, Edmond Chow, Pradeep Dubey. Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors. ICS’13. 3. Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D. Kalamkar, Xing Liu, Md. Mosotofa Ali Patwary, Yutong Lu, Pradeep Dubey. Efficient Shared-Memory Implementation of High-Performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices. SC’14. 4. <http://en.wikipedia.org/wiki/Conjugate_gradient_method>. 5. Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. SOSP’13. 6. Sungpack Hong, Hassan Chafi, Eric Sedlar, Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. ASPLOS’12. 7. Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, Mooly Sagiv. Specifying and Verifying Sparse Matrix Codes. ICFP’10. 8. I. J. Lustig and E. Rothberg. Gigaflops in Linear Programming. Operations Research Letters, 18(4):157–165, 1996. |

1. Is this idea related to work that is planned to be released as open source software? If so, please explain.

|  |
| --- |
|  |

1. Has your idea been reduced to practice? If so, describe the nature of the reduction to practice (e.g. demo, emulation, simulation, prototype, experimental verification, code written, etc.).

|  |
| --- |
| The idea abstracts the field experiences and optimizations of the PCL team in accelerating sparse applications, and automates a large and important part of their human efforts. These experiences and optimizations have been consistently shown effective in achieving ninja performance, by making Intel systems stay at the top of TOP 500. So from this respect, the idea is partially used and is effective in practice.  The PSE mega-project is planning the implementation of the idea. |

1. What aspect of your idea should be protected?

|  |
| --- |
| The co-design between the compiler and the library. More specifically:   1. The KNOB data structures. 2. The library function interface with KNOB as a parameter. 3. The compiler’s focus on analyzing contexts from user applications. 4. The library’s focus on optimizing performance guided by contexts. 5. The compiler’s analyses including context-specific constant inputs, dead outputs, formats of sparse input matrices, statistics, and fusion patterns. 6. The library’s internal optimizations including reusing persistent data (concrete dependence graph, converted sparse matrices, scheduling and synchronization strategy) across function calls in a context, avoiding computing dead outputs, and fusing with a consumer. |

1. Describe any aspects of your idea relating to unusual results or unusual function of the components/techniques in the idea, or check the box below:

 The unique combination of components/techniques in this idea provides an improvement over previously known structures and techniques:

|  |
| --- |
| A compiler automatically enables context-sensitive, data-specific, sparse optimizations with minimal changes to the libraries, without the help of any ninja experts or user annotation in source code. |

1. What is the value of your idea to Intel (how will it be used by Intel or a competitor)?

|  |
| --- |
| Sparse applications have emerged as a new trend in the big-data era. To win the market, it is important that Intel systems can quickly deliver customers high performance for these applications. With this invention, the customers get close-to-ninja performance without any manual efforts. This invention also adds values to Intel high-performance libraries like MKL by enabling context-sensitive and sparse-input-aware optimizations. |