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 | | Anderson, Todd |  | | | |
| 1. Personnel number (Intel WWID, if Intel employee) | | | | | | | | |
| 11339958 | | 11380636 | | 10632205 |  | | | |
| 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 | | Research Scientist |  | | | |
| 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:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Automatic reordering of sparse matrices to accelerate sparse applications**  High-performance computing (HPC) on sparse data structures like graphs and sparse matrices are becoming increasingly important in diverse areas such as machine learning, computational biology/chemistry/physics, physical model simulation, web search, and knowledge discovery [1].  Compared to traditional HPC applications that deal with regular and dense data structures, sparse computation has several unique challenges:   1. Memory bandwidth-bound performance.   Sparse computation typically has considerably lower compute intensity and, therefore, its performance is often limited by memory bandwidth. However, optimizing for memory bandwidth presents unique challenges (compared to optimizing for compute-intensive code) because the former usually involves non-local restructuring of the code. Frequent non-contiguous data accesses exacerbate this difficulty.   1. Data-dependent performance.   Memory access patterns and the amount of parallelism vary wildly over specific sparsity pattern of the input data. Therefore, it is often hard for a compiler alone to do certain optimizations because much of the important optimization information is unknown at the compile time.   1. Low SIMD efficiency.   Vectorization is a key optimization for HPC applications, as they tend to be data-parallel, and modern architectures like Xeon Phi have wide vector units (e.g. KNC has 512-bit wide vector units). The input of sparse application, however, is sparse, in that there may be few adjacent non-zeros. This cannot fully utilize modern architectures’ wide SIMD instructions.  To overcome these challenges, it is important to obtain high data locality for each input data set. Some sparse matrices have non-zeros dispersed, thus data locality is poor. A well-known technique to remedy this is reordering, which permutes rows and columns of a matrix so that non-zeros are clustered together. Below is an example showing the effect of reordering. One does not have to understand the details, but it is clear that the non-zero data (colored dots) are dispersed originally (Left), and get clustered together after reordering (Right).      Clustering of the data makes better use of memory bandwidth and vectorization: one memory read gets more non-zeros to compute in parallel.  Different reordering algorithms have been invented over the past several decades, such as Reverse Cuthill–McKee (RCM), Self-Avoiding Walk (SAW), METIS Partitioner, King’s algorithm, and Modified Minimum Degree [3-5]. Their performance impact has been extensively studied and recognized. Each reordering algorithm has been invented to optimize different metrics. RCM is a popular reordering algorithm to optimize for cache locality in sparse matrix vector multiplication (SpMV), a fundamental kernel for sparse applications. Below is a study we performed on a 10-core Ivybridge processor, using RCM reordering to SpMV. For a set of real-life sparse matrices, their matrix bandwidth (how far two elements in the same row can be) is reduced by 403 times. Correspondingly, SpMV is sped up by 38%. This is significant, since SpMV is a well-tuned kernel.   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | **Matrix bandwidth** | | | **SpMV performance (GB/s)** | | | | **Matrices** | **Original** | **RCM** | **Reduction** | **Original** | **RCM** | **Speedup** | | 333SP | 3681176 | 2800 | 1314.706 | 20.5514 | 32.2761 | 1.570506 | | audikw\_1 | 925946 | 35924 | 25.77514 | 23.1149 | 42.0778 | 1.820376 | | delaunay\_n24 | 16769102 | 14470 | 1158.887 | 28.0673 | 34.5259 | 1.230111 | | dielFilterV3real | 1036475 | 25795 | 40.18124 | 27.3489 | 42.2179 | 1.543678 | | G3\_circuit | 947128 | 4073 | 232.5382 | 23.1617 | 34.3788 | 1.484295 | | inline\_1 | 502403 | 3686 | 136.3003 | 31.0359 | 41.2234 | 1.329638 | | NLR | 4163523 | 4830 | 862.013 | 20.0453 | 35.4984 | 1.770909 | | parabolic\_fem | 525820 | 769 | 683.7711 | 48.981 | 67.4745 | 1.377565 | | rajat31 | 4688751 | 7493 | 625.7508 | 37.5079 | 39.5975 | 1.055711 | | thermal2 | 1226000 | 733 | 1672.578 | 25.9935 | 34.609 | 1.331448 | | thermomech\_dK | 204277 | 525 | 389.099 | 45.3094 | 54.7614 | 1.20861 | | thermomech\_dM | 204276 | 264 | 773.7727 | 36.3561 | 49.0274 | 1.348533 | | tmt\_sym\_amd | 725454 | 907 | 799.839 | 33.998 | 35.9241 | 1.056653 | | geomean |  |  | 403.1434 |  |  | 1.379717 |   **Table 1. Speedup of SpMV with reordering**  Therefore, reordering is indeed an important optimization for sparse applications. These reordering algorithms have been applied manually by expert programmers to speed up sparse kernels, which are commonly-used library functions involving sparse matrices. However, *no algorithm has ever been proposed to* ***automatically*** *apply these algorithms to any* ***arbitrary*** *function other than the kernels, i.e. how to automatically determine if reordering is applicable to any arbitrary function, and if so, how to do it without changing its semantics.* This automatic reordering will help even expert programmers because applying the reordering optimization is a difficult, error-prone and time-consuming process. An incorrect reordering would change the semantics of the program. To manually generate a correct reordering, a programmer has to painstaking inspect multiple nested levels of loop, if not dozens of functions with complicated calling relations, to make sure that the reordering would not break the original semantics.  Solving this problem is part of the PSE (Problem-Solving Environment) mega-impact project. We have proposed a solution (described below) and prototyped it in our system.  Note: we are NOT proposing a new reordering algorithm; instead, given any reordering algorithm, we propose an algorithm to determine if it can be safely applied, and if so, how. |

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 a dataflow analysis to accurately determine the following:   1. Feasibility of reordering: is it legal to apply reordering to any given region of code (like a function, a loop, etc.)?   To do this, we propose a key concept of “distributivity”. We scan all statements in the region to see if they are distributive.   1. Arrays to reorder or reverse-reorder: If it is legal to perform reordering, what arrays (including multi-dimensional matrices and 1-dimensional vectors) should be reordered before the code region? What arrays should be reverse-reordered after the code region, such that the code outside the region is not affected?   To do this, we propose another key concept of “Inter-dependent Arrays (IAs)”. We scan each statement in the region to calculate IAs for it.  Based on IAs, we propose a bi-directional dataflow analysis. We scan all the statements in the region and compute the arrays to be reordered or reverse reordered. |

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

|  |
| --- |
| Currently, when reordering is performed, it is expert programmers who have been responsible for manually applying the reordering transformations to one computational kernel at a time. This invention improves upon this situation by allowing the reordering transformation to be done automatically within a compilation framework. We are not aware of any other algorithm that automatically answers the two questions in Section 2.a above. |

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).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Our invention applies a reordering transformation to a code region in order to improve the execution time. A code region can be an arbitrary part of a program. It may contain sequential statements, loop statements (such as **for**, **repeat … until**, and **while**), and control-flow statements (such as **if ... else**, **goto**, **break**, **exit**, etc.). A special, yet useful, case of a region is when it is a loop and it contains no control-flow statements in the loop body, which we call a *Linear Loop Region*. Note that a linear loop region can contain other loop statements inside, i.e., the loop can be a nested loop.  As shown in Fig.1, this reordering transformation affects a code region by reordering some arrays prior to their use within the region. Moreover, for correctness, we must reverse the reordering of an array after the code region if that array could be subsequently used. If the code region contains control-flow statements, in general, some arrays along some paths in the region might have to be reordered and reversely reordered, so that the reordering affects only parts of the region. The new code introduced by the transformation is highlighted in bold font. Fig.2 illustrates this with PCG (Pre-conditioned Conjugate Gradient) as an example. Note that besides the reordering and reverse-reordering outside the region, there is such code inside it as well. Such inside code may be eliminated as much as possible by making reordering of some arrays as early as possible, as we will describe later. Fig.3 shows this with PCG again, by making reordering of vector i as early as possible.  **Fig.1 A region before and after reordering**  **Fig.2 PCG as a reordering example for a general code region. Here the region is enclosed by a box. The code outside a box is outside the region.**  **Fig. 3 PCG as a reordering example for a general code region, when reordering is to be done as early as possible.**  For the special case, a linear loop region, reordering and reverse-reordering can only happen outside the region, as shown in Fig.4. Fig.5 illustrates this with PageRank as an example.  **Fig.4 A linear loop region before and after reordering**  **Fig.5 PageRank as a reordering example for a linear loop region.**  **Fig.6 Workflow**  Fig.6 shows the workflow of our invention. The user program can be in any language. First, the compiler identifies and forms a code region. This region can be an arbitrary part of the program. There is no restriction to its shape, what can be inside it, etc. For example, it can be a path, a tree, etc. So in principle, it is truly arbitrary and general. But usually, a region is a hot loop, or at least contains a hot loop, including its entire body, where the program spends much of its execution time.  Then, the compiler analyzes distributivity of the region. If it is not distributive, the compiler will not perform any reordering, and thus fail.  Next, for the code region, the compiler performs liveness analysis (a traditional dataflow analysis), inter-dependent array analysis, and reorderable array discovery. Finally, it transformed the code region.  Below we describe the new things proposed in the invention, i.e. distributivity analysis, inter-dependent array analysis, reorderable array discovery, and code transformation. We will also introduce a benefit-cost analysis of reordering for SpMV.   1. **Distributivity analysis**   A reordering R over an expression ε is **distributive** if its semantics remains the same whether its output is reordered, or its inputs are reordered. That is,  R(ε(i1, ..., in)) = ε(R(i1), ..., R(in)) (1)  where i1, ..., in are the inputs.  When a region has no control-flow statements, for example, when it is a linear loop region, the inputs of the region are unconditional. Then the region can be viewed collectively as a single expression. If a reordering R is distributive over all the expressions in a region, it is distributive over the collective single expression. Based on the definition (1), in order to reorder the result of the region, we can simply reorder all its inputs, without changing anything inside the region at all. An example has been shown in Fig.5.  When a region does have control-flow statements, some inputs may be conditional, and thus reordering of such inputs may also be conditional. An example has been shown in Fig.2.  Reordering is defined as the following:  R(x) = P'\*x\*P if x is a matrix (This is a similarity transformation)  = P'\*x if x is a vector  = x if x is a number  = error no other kind of input is accepted. (2)  where P is a permutation matrix, and P' the transpose of P, which is equal to the inverse of P [6].  Distributivity analysis scans all the expressions in the region, and tests if reordering R is distributive over all of them.  Commonly-seen array-related expressions are usually distributive. For examples, let  M represent a matrix, v a vector, and n a number, then the following expressions are distributive:  M\*M // For simplicity, we use M to represent both matrices here. But they do not have to be the same. Similar notation applies to the others below  M+M  M-M  M\*v  M\v // “\” is the inverse divide. M\v equals M-1\*v, where M-1 is the inverse of M  dot(v,v) // dot product of two vectors  v+v  v-v  n\*M  n\*v  By definition, reordering R is distributive over any expression with no inputs and outputs, and over any expression whose inputs and output are all numbers. For examples, R is distributive over a conditional or goto statement:  if (n)  goto  There are some cases that reordering is not distributive:   1. An expression requires its inputs or outputs to be a specific shape.   For example, a triangular solver assumes its input is a lower or upper triangular matrix. Reordering the matrix would make it non-triangular.   1. I/O   For example, a matrix used in printing cannot be reordered, because that breaks the expectation of the users.   1. User requires "bitwise reproducibility".   Reordering may change the precision of the result, since floating-point operations are not associative.   1. Unknown functions   To be conservative, we can only assume reordering is not distributive over such functions.  However, if the source code of the function is available, we can analyze that function in the same way as we do the current function.  It should be noted that although in Fig.6, we have separated region formation and distributivity analysis, in practice, the two steps may also be combined. For example, one might start from an empty region, and gradually grow the region by adding statements into it; for each statement, it can check if it is distributive or not; if not distributive, it simply does not add it to the region.   1. **Inter-Dependent Array Analysis**   Inter-dependent Arrays are a set of arrays that if anyone of them is reordered, all the others have to be reordered. The arrays may be variables, or constants. For example, for a statement  x = A \* y  where x and y are vectors, and A is a sparse matrix. If A is reordered (some columns and rows of it are exchanged), then y has to be reordered (the corresponding rows of y are exchanged), and consequently, x is also reordered (the corresponding rows of x are exchanged). Similarly, if any of x or y is reordered, A has to be reordered correspondingly.  In general, if there is a statement in the code region:  array1 = ε(array2, array3),  where ε is any expression, then arrays1~3 are inter-dependent arrays.  One statement can have more than 1 set of inter-dependent arrays. For the following example statement B:  B: v1 = v2 + v3\*dot(M\*v4, v5)  where M is a sparse matrix, while v1~5 are vectors. Note that dot() produces a number, therefore, the arrays inside dot() and those outside dot() do not affect each other, i.e., they are not inter-dependent. Therefore, there are two sets of inter-dependent arrays for this statement. Let us call each set a *cluster* for short:  cluster1: {v1 | v2, v3}  cluster 2: { | M, v4, v5}  Here we use “|” to separate variables that are defined (in the left hand side) from those that are used (in the right hand side). For a cluster C, we will use C.LHS and C.RHS to denote those variables that in its left and right hand side.  Then we define the following:  (*B*, *X*) = ∪*C* ∀cluster *C* of *B*, *X* ∩ *C*.RHS is not empty (3)  (*B*, *X*) = ∪*C* ∀cluster *C* of *B*, *X* ∩ *C*.LHS is not empty (4)  Here ∪*C* will unite each C’s LHS together, and each C’s RHS together. So and also have their LHS and RHS.  For example, with the above example statement B, we have  (B, {v1}) = { } // because v1 is not in any cluster’s RHS  (B, {v2}) = { v1 | v2, v3} // because v2 is in cluster 1’s RHS  (B, {v2, u}) = { v1 | v2, v3} // u is not in any cluster’s RHS and does not affect the result  (B, {v2, v4}) = { v1 | v2, v3, M, v4, v5} // because v1 is in cluster 1’s RHS, and v4 in cluster 2’s RHS  Similarly,  (*B*, {v1}) = { v1 | v2, v3 } // because v1 is in cluster 1’s LHS  (*B*, {v1, v4}) = { v1 | v2, v3 } // v4 is not in any cluster’s LHS and does not affect the result    **Fig.7 Inter-dependent array analysis**  **Fig.8 An example of inter-dependent array analysis**  Fig.7 shows how to compute IAs, and Fig.8 shows an example. For a statement, the analysis builds an expression tree for it, where all internal nodes are operations or functions, while all terminal nodes are variables or constants. Then break the tree into subtrees: if an internal node’s result type is a number, break the edge between its parent and it; if the internal node is a function, the subtree of which it is a root can be broken down by analyzing that function, or if the function source code is not available, by accepting external metadata on this function from the user or the system. Continue breaking down until the original tree cannot be broken into smaller subtrees.  Then for each subtree, the arrays within all the terminal nodes are inter-dependent.  It should be noted that the expression tree for a statement might not need explicitly built, depending on implementations. For example, if the statement is in the 3-adress format (result, operator, and two operands), it is already implicitly a tree.   1. **Reorderable array discovery**   This is a bi-directional dataflow analysis on the Control Flow Graph (CFG) of the function under consideration. Each basic block there is a single statement. Some or all the blocks belong to, i.e. are in, our selected region, the others do not belong to, i.e. are outside, the region.  The region has 1 or more *entries*, which are blocks that have no predecessor block inside the region, even though they may have outside the region.  Some sparse arrays’ data locality might be improved by a reordering transformation. Such arrays could simply be the first one or few sparse arrays related with some operations known to be important (such as SpMV) inside the region. Alternatively, the user could annotate the region to tell the compiler which array(s) to reorder right *after* each entry. We call these arrays *First Arrays to Reorder (FAR)*.  The compiler propagates the FAR in the region in forward and backward direction to identify all the arrays that must be reordered before and after each block, including the entries. We call them Reorderable Arrays, denoted In[*B*] and Out[*B*], before and after a block *B*.  Discovering reorderable arrays is a bi-directional analysis. For example, for the following region (a simplistic, not real, region. Only for illustration purpose):   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Block | State | Initial state | Precondit  -ioning | Growth | | | | | Backward  pass | Forward  pass | Backward  pass | Forward  pass | | B1: F=E | In  Out | {F} | {F} | {E, G }  {F, G} | {E, G}  {F, E, G} | No  change | No  change | | B2: H=F+G | In  Out | 𝕌 | {F}  {F, G, H} | {F, G}  {F, G, H} | {F, E, G}  {F, G, E, H} |   Assume the FAR after the entry of the region, B1, equals {F}. Out[B2] is initialized to be the universal set, 𝕌, which includes all the arrays. With this initial state, we repeat forward and backward propagation, until no change happens. The arrows show the order of the In and Out sets that are changed.  Let def[*B*] (or use[*B*]) is the set of arrays defined (or used) in statement *B*. When the compiler propagates a set of reorderable arrays *X* through a block (statement) *B* in forward direction, the forward transfer function is  *f*(*B, X*) *=* (*B*, *X*∩use[B]) ∪ (*X* – def[*B*] – use[*B*])  Explanation: Forward transfer means going from before the statement B to after it, through the statement’s RHS and LHS in order. For those arrays in *X*, there are two cases when propagated through *B*: (1) if they are also used by statement *B*, then all the clusters with them in their RHS are the new set of reorderable arrays. This explains the first term (*B*, *X*∩use[B]) . (2) Otherwise, they are not used by statement *B*. Then all of them, except those killed by the statement, are the new set of reorderable arrays. This explains the second term *X* – def[*B*] – use[*B*]. Note that some arrays, although killed and thus not appeared as the result of the second term, might appear as the result of the first term because of inter-dependent with some variable in *X* and used by the statement.  When the compiler propagates a set of reorderable arrays *X* through a block (statement) *B* in backward direction, the backward transfer function is  *b*(*B, X*) *=* (B, *X*∩def[B]).RHS ∪(B, (*X* - def[*B*])∩use[B]).RHS ∪ (*X* – def[*B*] – use[*B*])  Explanation: Backward transfer means going from after the statement B to before it, through the statement’s LHS and RHS in order For those arrays in *X*, there are three cases when propagated through *B*: (1) if they are also defined by statement *B*, then all the clusters with them in their LHS are the new set of reorderable arrays. Since we are propagating backward, i.e. from the end of the statement to the beginning of it, we take only the RHS of the result. This explains the first term (B, *X*∩def[B]).RHS. (2) If they are not defined by statement *B*, but used by it, then all the clusters with them in their RHS are the new set of reorderable arrays. Similarly, we take only the RHS of the result. This explains the second term (B, (*X* - def[*B*])∩use[B]).RHS. (3) Otherwise, they are not defined and not used by the statement. Then all of them are the new set of reorderable arrays. This explains the third term *X* – def[*B*] – use[*B*].  Note: the result of forward or backward transfer is before or after a statement, where there is no LHS or RHS. And thus the result is simply a set, no need to distinguish LHS and RHS variables any more.  The bi-directional dataflow analysis is iterative until a fixed point has been reached. The full algorithm is shown in Fig.9.  In this algorithm, *preds*(*B*)and *succs*(*B*)are the set of predecessors and successors of a region block *B* in the CFG. Note that a predecessor or successor may be in or outside the region. FAR(*B*) are the first arrays to reorder right after a region entry block *B*.  There are 3 steps: First, initialize the In and Out of any block outside the region as empty set ∅, because outside the region, there should not be any array reordered: reordering’s effect should be limited within the region. This setting for the outside region blocks will never change. For each region entry, its Out set is initialized to the FAR after it. For any other region node, the Out set is initialized to the universal set, 𝕌. The In sets of the region nodes need not to be initialized, as they will be instantiated automatically in the following steps.  Second, precondition the In and Out sets of the region nodes by propagating the initial state forward. For each block *B*, In[*B*] includes the arrays that are reoderable after every predecessor of it. Then Out[*B*] is the result of propagating In[*B*] through *B*, calculated by the forward transfer function *f*. Repeat this process until there is no change.  Third, repeatedly propagate in backward and forward direction, until there is no change happens.  In the backward pass, Out[*B*] is enlarged by adding the arrays that are reoderable before every successor of it. There are at least two optimizations we can do here:   1. If a variable is dead before a successor, i.e. not used at all in any execution path through the successor, then it can be artificially made reordered before the successor, as that won’t affect the program semantics: the array is not used anyway. To find such variables Dead[*S*] for a successor *S* of region block *B*, we can get all the reorderable arrays before all the successors of *B*, except those that are live into the successor. We call this OPTIMIZE\_DEAD. 2. If block *B* has more than 1 successor block, and their execution frequencies are very different, we can favor the most frequent successor *x* by always allow the reoderable arrays in In[*x*] to be propagated to Out[*B*]. For example, assume one successor *x* is in a loop, and all the others are outside a loop, then experientially, if we propagate that In[*x*] to Out[*B*], we can avoid inserting reordering of any array between *B* and *x*. However, we might have to insert reverse-reordering of some of these arrays between *B* and the successors other than *x*. We call this OPTIMIZE\_FREQ.   Finally, In[*B*] is enlarged by adding the arrays that are the result of propagating Out[*B*] through *B*, calculated by the backward transfer function *b*.  The forward propagation pass is similar to precontioning, except that In[*B*] and Out[*B*] keep their original values, and then grow with new arrays.  Input: a function’s CFG whose blocks are single statements. Part or all of the blocks belongs to the region *R* under consideration.  Steps:   1. Initialization   ∀*B* outside *R*: In[*B*] *=* Out[*B*] = ∅.  ∀*B* in *R*: Out[*B*] =   1. Preconditioning   repeat the following until there is no change:  ∀*B* in *R* such that *B* is not an entry of *R*:  In[B] =  Out[B] = *f*   1. Growth   repeat the following two passes until there is no change:  3.1. Backward pass  ∀*B* in *R*  where Dead[S] =, and  Frequent[B] = In[*x*] and executes  most frequently among all successors of *B*.  In[B] = *b*  3.2. Forward pass  ∀*B* in *R*  In[B] =  Out[B] =  **Fig.9 Reorderable array discovery**  Fig.10 shows the CFG for the PCG code in the left of Fig.2. All the blocks are in the region, except B13, are in the region. B13 is outside the region because the print statement is not distributive, as we explained before. There is only 1 entry to the region, which is B1.  For Fig.10, we apply the algorithm in Fig.9 to discover reorderable matrices. Assume that the FAR after the entry B1 is {A}, and we do not do any optimization. Then the state changes are shown in Table 2. In the growth step, we show only 1 backward pass: there are actually 1 more forward pass, and 1 more backward pass followed by 1 more forward pass. They will not change any state and are not shown.  Note: B7 has two successors: B8 and B13. In the backward pass in Table 2, array i does not appears in Out[B7], because array i is in In[B8] but not in In[B13]. Therefore, it is not propagated to Out[B7].  If, however, we enable OPTIMIZE\_DEAD, we can still propagate it to Out[B7]: reordering of array i after B7 does not affect B13 at all, since array i is dead before B13. The state change is shown in Table 3. We highlight the difference between it and Table 2 in red font.  Or if we enable OPTIMIZE\_FREQ, since B8 is in a loop while B13 is not, we always propagate In[B8], which includes array i, to Out[B7]. Similarly, we always propagate In[B3] to Out[B2]. The state change is shown in Table 4. We highlight the difference between it and Table 2 in red font.   1. **Code transformation**   For any block B2 in the region, if there is an edge B1->B2 in the CFG, where B1 is another block in the CFG that may or may not be in the region, then for every variable x∈LiveIn[B2],  if x∉Out[B1] but x∈In[B2], insert “x=reorder(x)” at the edge.  if x∈Out[B1] but x∉ In[B2], insert “x=reverse\_reorder(x)” at the edge.  As a special case, for any block B2 that is an entry of the region, for every variable x∈LiveIn[B2],  if x∈In[B2], insert “x=reorder(x)” before B2.  According to Table 2, where no optimization is done, we can get the final CFG in Fig.11, which corresponds to the code shown in the right of Fig.2.  According to Table 3, where OPTIMIZE\_DEAD was done, we can get the final CFG in Fig.12.  According to Table 4, where OPTIMIZE\_FREQ was done, we can get the final CFG in Fig.13, which corresponds to the code shown in the right of Fig.3.  **Fig. 10 The CFG for the PCG code in the left of Fig.2. All blocks except B13 are in the region.**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Node | State | Initialization | Preconditioning | 1st backward pass | | B1 | IN  OUT | {A} | {A} | {A}  {A} | | B2 | IN  OUT | 𝕌 | {A}  {A} | {A}  {A} | | B3 | IN  OUT | 𝕌 | {A}  {A} | {A, p, x, r}  {A, p, x, r} | | B4 | IN  OUT | 𝕌 | {A}  {A, p} | {A, p, x, r}  {A, p, x, r} | | B5 | IN  OUT | 𝕌 | {A, p}  {A, p, x} | {A, p, x, r}  {A, p, x, r} | | B6 | IN  OUT | 𝕌 | {A, p, x}  {A, p, x, r} | {A, p, x, r}  {A, p, x, r} | | B7 | IN  OUT | 𝕌 | {A, p, x, r}  {A, p, x, r} | {A, p, x, r}  {A, p, x, r} | | B8 | IN  OUT | 𝕌 | {A, p, x, r}  {A, p, x, r, i, z} | {A, p, x, r, i}  {A, p, x, r, i, z} | | B9 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B10 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B11 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B12 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B13 | IN  OUT | ∅  ∅ | ∅  ∅ | ∅  ∅ |   **Table 2. Illustrating the algorithm in Fig.9 with the CFG in Fig. 10, without any optimization**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Node | State | Initialization | Preconditioning | 1st backward pass | | B1 | IN  OUT | {A} | {A} | {A, p, r, i }  {A, p, r, i } | | B2 | IN  OUT | 𝕌 | {A}  {A} | {A, p, r, i }  {A, p, r, i } | | B3 | IN  OUT | 𝕌 | {A}  {A} | {A, p, x, r, i }  {A, p, x, r, i } | | B4 | IN  OUT | 𝕌 | {A}  {A, p} | {A, p, x, r, i }  {A, p, x, r, i } | | B5 | IN  OUT | 𝕌 | {A, p}  {A, p, x} | {A, p, x, r, i }  {A, p, x, r, i } | | B6 | IN  OUT | 𝕌 | {A, p, x}  {A, p, x, r} | {A, p, x, r, i }  {A, p, x, r, i } | | B7 | IN  OUT | 𝕌 | {A, p, x, r}  {A, p, x, r} | {A, p, x, r, i }  {A, p, x, r, i} | | B8 | IN  OUT | 𝕌 | {A, p, x, r}  {A, p, x, r, i, z} | {A, p, x, r, i}  {A, p, x, r, i, z} | | B9 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B10 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B11 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B12 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B13 | IN  OUT | ∅  ∅ | ∅  ∅ | ∅  ∅ |   **Table 3. Illustrating the algorithm in Fig.9 with the CFG in Fig. 10 when OPTIMIZE\_DEAD is enabled**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Node | State | Initialization | Preconditioning | 1st backward pass | | B1 | IN  OUT | {A} | {A} | {A, p, x, r, i }  {A, p, x, r, i } | | B2 | IN  OUT | 𝕌 | {A}  {A} | {A, p, x, r, i }  {A, p, x, r, i } | | B3 | IN  OUT | 𝕌 | {A}  {A} | {A, p, x, r, i }  {A, p, x, r, i } | | B4 | IN  OUT | 𝕌 | {A}  {A, p} | {A, p, x, r, i }  {A, p, x, r, i } | | B5 | IN  OUT | 𝕌 | {A, p}  {A, p, x} | {A, p, x, r, i }  {A, p, x, r, i } | | B6 | IN  OUT | 𝕌 | {A, p, x}  {A, p, x, r} | {A, p, x, r, i }  {A, p, x, r, i } | | B7 | IN  OUT | 𝕌 | {A, p, x, r}  {A, p, x, r} | {A, p, x, r, i }  {A, p, x, r, i} | | B8 | IN  OUT | 𝕌 | {A, p, x, r}  {A, p, x, r, i, z} | {A, p, x, r, i}  {A, p, x, r, i, z} | | B9 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B10 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B11 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B12 | IN  OUT | 𝕌 | {A, p, x, r, i, z}  {A, p, x, r, i, z} | {A, p, x, r, i, z}  {A, p, x, r, i, z} | | B13 | IN  OUT | ∅  ∅ | ∅  ∅ | ∅  ∅ |   **Table 4. Illustrating the algorithm in Fig.9 with the CFG in Fig. 10 when OPTIMIZE\_FREQ is enabled**  **Fig.11 The CFG in Fig.10 after code transformation, based on the result in Table 2, which has done no optimization.**  **Fig.12 The CFG in Fig.10 after code transformation, based on the result in Table 3, which has done OPTIMIZE\_DEAD.**  **Fig.13 The CFG in Fig.10 after code transformation, based on the result in Table 4, which has done OPTIMIZE\_FREQ.**   1. **Benefit-cost analysis for SpMV**   SpMV is an important kernel used by many applications. It reads one matrix as its input. The specific data of the matrix can be from many sources. Reordering does not improve the performance of SpMV for certain matrix data, and applying reordering algorithms like BFS to such matrix data will waste time. Therefore, we propose to use a cost/benefit analysis to predict if reordering will improve the overall performance. We build a linear regression model to predict the benefit of reordering. In one embodiment, we simply use two variables for linear regression: 1) SpMV performance before reordering measured in GB/s and 2) the machine peak bandwidth measured by STREAM benchmark. These two are good estimators for SpMP performance speedup from reordering because SpMV performance after reordering tends to be close to the machine peak bandwidth for many matrices. Note that the machine peak bandwidth measured by STREAM benchmark captures the difference between processors, making our linear regression robust across a wide range of processors. This two-variable model is simpler than a known state of art [7] but is still effective. In other embodiments, we can refine the linear regressing by adding other matrix properties such as number of rows, number of non-zeros, and the maximum distance between a non-zero and a diagonal element in any given matrix row. Since the performance of BFS does not vary much over matrices, we can easily predict the cost of BFS reordering, completing our cost/benefit analysis.  More specifically, this analysis is suitable for the case when SpMV is invoked inside a loop for many times. The region is the loop. We do the following:   1. Offline:    1. Build a linear regression model for SpMV.    2. For the SpMV function, add two additional parameters: *reorder\_enabled* and the address of *beneficial.* If *reorder\_ enabled* is true, then do SpMV as usual. Otherwise, measure the bandwidth with the usual SpMV, and consult the linear regression model to see if it is beneficial to turn on reordering. 2. At compile time, do what Fig.6 shows. The only difference is in the last step: code transformation. It now works in the following way:    1. Before the loop, do not generate any reordering, but insert two variables, *beneficial* and *reorder\_enabled*, initialized as false.    2. At the beginning of the loop body, test if *beneficial* is true but *reorder\_ enabled* is false, then do reordering of all the arrays shown in the In set of the first block of the loop body. Set *reorder\_ enabled* as true.    3. For reordering and reverse-reordering in any other place, guard them by checking *reorder\_ enabled*.    4. When invoking the SpMV function, pass in *reorder\_ enabled* and the address of *beneficial.*   For example, the PCG code with SpMV benefit-cost analysis turned on is shown in Fig.14. Refer to Fig.3 to see the difference from the code without such analysis.  **Fig.14. Equivalent code for the right part of Fig.3 with SpMV benefit-cost analysis enabled** |

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)?

|  |
| --- |
| If one suspected that another compiler was infringing, then one would take a source code example that lacked explicit reordering optimizations and run that source code through the other compiler. One could have that other compiler produce assembly code or machine code. Then, it is a simple matter to inspect that assembly or machine code to determine whether reordering and reverse reordering calls have been automatically inserted by that compiler. |

1. Is this idea related to work performed in a Standard 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. Xing Liu, Mikhail Smelyanskiy, Edmond Chow, Pradeep Dubey. Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors. ICS’13. 2. Reordering examples. <http://www.mathworks.com/help/matlab/examples/sparse-matrices.html?prodcode=ML#zmw57dd0e2186> 3. E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, in Proceedings of the 24th ACM National Conference, 1969, pp.157–172. 4. G. Heber, R. Biswas, and G. R. Gao, Self-avoiding walks over adaptive unstructured grids, Concurrency: Pract. Exper., 12 (2000), pp.85–109. 5. Leonid Oliker, Xiaoye Li, Parry Husbands, Rupak Biswas. Effects of Ordering strategies and Programming Paradigms on Sparse Matrix Computations. SIAM REVIEW. Society for Industrial and Applied Mathematics. Vol. 44, No. 3, pp. 373–393. 2002. 6. Permutation Matrix. <https://en.wikipedia.org/wiki/Permutation_matrix> 7. Samantha Wood. SMOReS: Sparse Matrix Omens of Reordering Success. <http://triceratops.brynmawr.edu/dspace/handle/10066/7572?show=full> |

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

|  |
| --- |
| Yes. We plan to open source our implementation as an external plug-in package to Julia, a HPC language, in June, 2015. |

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.).

|  |
| --- |
| Yes. We have implemented this idea as a package to Julia. It automatically speeds up SpMV by 20% on average on a 12-core Xeon machine.  We plan to give a demo of this work to Julia community at a conference (JuliaCon) in June 24~28, 2015. |

1. What aspect of your idea should be protected?

|  |
| --- |
| The workflow shown in Fig.3 to construct an automatic system. Particularly, we should protect the idea of the following:   1. The feasibility test for any reordering algorithm, that is, the distributivity analysis. 2. Computing inter-dependent arrays (Fig.4). 3. Computing reorder-in and out sets (Fig.5).   Reordering algorithms are not our contributions, and need not be protected. |

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 dataflow formulation that automates reordering. |

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. Poor data locality in sparse application is a serious bottleneck. Optimizing for data locality is particularly important for Intel because Intel processors invest a significant portion of on-chip silicon real estate to caches and heavily rely on them for high performance compared to competitors such as NVIDIA GPUs. With this invention, a compiler can automatically reorder sparse matrices for better locality, saving hardware caches, and guarantee correctness, without any manual efforts. By pre-processing sparse inputs, this invention also adds values to Intel high-performance libraries like MKL, making them run faster. |