Compiling Dynamic Data Structures in Python to Enable the Use of Multi-core and Many-core Libraries

Bin Ren Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{ren, agrawal}@cse.ohio-state.edu

***Abstract*—Programmer productivity considerations are increasing the popularity of interpreted languages like Python. At the same time, for applications where performance is important, these languages clearly lack even on uniprocessors. In addition, the use of dynamic data structures in a language like Python makes it very hard to use emerging libraries for enabling the execution on multi-core and many-core architectures.**

This paper presents a framework for compiling Python to use multi-core and many-core libraries. The key component of our framework involves a suite of algorithms for replacing dynamic and/or nested data structures by arrays, while minimizing unnecessary data copying costs. This involves a novel use of an existing partial redundancy elimination algorithm, development of a new demand-driven interprocedural partial redundancy algorithm, a data flow formulation for determining that the contents of the data structure are of the same type, and a linearization algorithm.

We have evaluated our framework using data mining and two linear algebra applications written in pure Python. The key observations were: 1) the code generated by our framework is only 10% to 20% slower compared to the hand-written C code that invokes the same libraries, 2) our optimizations turn out to be significant for improving the performance in most cases, and 3) we outperform interpreted Python and the C++ code generated by an existing tool by one to two orders of magnitude.

***Keywords*-Python; Redundancy Elimination, Compilation for multi-core and many-core**

I. Introduction

The problem of scaling applications to large input sizes is becoming increasingly harder. This can be attributed to two important trends: first, starting from the last 6-7 years, it is no longer possible to improve computing speed by simply increasing clock frequencies. As a result, multi-core architectures and accelerators like GPUs have become cost-effective means for scaling performance. However, each of these imposes a *programmability* challenge, and existing sequential programs cannot directly benefit from these.

Second, as programmer productivity is becoming extremely important, there is a growing trend towards very high-level languages. Languages like MATLAB, Python, Perl, and Ruby are often simpler to learn (especially, for programmers in certain domains), and result in much more concise code. Thus, they can ease application development. But, because of their interpreted nature and the use of high-level constructs, they also often result in poor performance, besides being not able to exploit parallelism on multi-cores and GPUs.

Clearly, it will be very desirable if translators can be built to automatically or semi-automatically translate programs written in high-level languages for scalable execution on multi-cores and/or GPUs. This paper describes one such system. Our work is driven by the growing popularity of Python, and the need for scaling numerical computations on multi-cores and GPUs, using the existing libraries.

Though projects like *NumPy* [29] and *SciPy* [15] have tried improving the suitability of Python for HPC applications by providing support for arrays and array-based operations, compute-intensive and data-intensive applications in Python continue to be written in *pure* Python, using more dynamic structures. For example, consider the implementations of K-means clustering, one of the most popular data mining algorithms, from web-sources[1][2]. These implementations use lists in Python to flexibly manage dataset of any size.

With current interpreters and translation systems, such applications perform poorly, i.e. the programmers are clearly trading performance for programmability. Even though a translation system like Shedskin [8] can translate Python applications to C++, allow existing C++ compilers to be used, the resulting compiled code is still quite slow.

Besides the performance problems associated with high-productivity languages in general, and applications that use dynamic data structures in Python in particular, another important factor is the support for use of multi-core and many-core architectures. There is a growing trend towards libraries that can support a specific class of applications on multi-core or many-core architectures. Examples of classes of applications for which libraries have been developed include data-intensive applications [14] and linear algebra applications [23], [28]. These libraries expect parameters to be multi-dimensional arrays, and cannot be directly invoked when the application is based on dynamic data structures.

We have developed a Python based compilation system that can replace dynamic data structures with arrays, and invoke libraries for multi-core and many-core architectures for specific types of computations. To enable such support efficiently, we have developed several new algorithms. The key contributions include a *demand-driven* inter-procedural version of an existing Partial Redundancy Elimination (PRE) algorithm [30], and an algorithm for determining homogeneity of a list.

We have applied our framework to compile two data mining applications and two linear algebra applications. We demonstrate that by our translation and optimization framework, we are able to generate code which is only 10-20% slower than the hand-written C code that uses the same libraries. Thus, we have demonstrated that the productivity of a language like Python can be combined with performance, and furthermore,

[1] http://www.fantascienza.net/leonardo/so/kmeans/kmeans.html
[2] http://www.daniweb.com/software-development/python/threads/31449

multi-core and many-core architectures can be exploited starting from high productivity languages.

The rest of the paper is organized as follows. Section II describes the challenges in Python and gives an overview of our work. Interprocedural PRE and related algorithms are presented in Sections III and IV. In Section V, we outline our implementation and report results from a detailed experimental evaluation. We compare our work with related research efforts in Section VI and conclude in Section VII.

II. Challenges and Overview of Our Work

In this section, we will introduce the performance issues of Python, and give an overview to our translation framework.

A. Python and Performance Issues

While our work is applicable to all languages where dynamic data structures are used, the techniques we have developed and implemented have been motivated by features of Python. Python has been rapidly gaining popularity because of its support for high productivity and easy learning curve. This enables programmers to focus on developing and expressing algorithms, rather than programming itself. While it provides high productivity, performance efficiency of the applications developed using Python is not very good. Thus, for HPC applications, where performance is an important issue, the use of Python creates several challenges. At the same time, programmer productivity has become an important concern within HPC as well, promoting use of Python and similar languages, including specialized parallel languages like X10 [5]. One approach is to use language extensions and/or low level libraries to help improve performance. Successful projects in this area include *NumPy* [29], *SciPy* [15], *PyMPI* [26], *PyCUDA*, and *PyOpenCL* [16], among others.

The reasons for the low efficiency of Python arise because of multiple related reasons. The fact that Python is interpreted and not compiled is clearly a big factor. Moreover, one of the most attractive features of Python, the rich support for dynamic data structures, like *list*, *dictionaries*, and others, adds significant overheads. Dynamic typing, which further gives flexibility to programmers, also adds to the execution time overheads.

To look at the overheads in more details, let us take the *list* data structure supported in Python as an example. An attractive feature of a Python list is that the users can store different data types as different elements of the list. However, now dynamic type checking has to be applied to each element of the list. Moreover, since the list only stores pointers to the objects, rather than the actual objects themselves, the data is not stored continuously. As a result, data locality and cache usage is negatively impacted.

To quantify these overheads, we performed the following experiment. We implemented a linear algebra routine, *Double GEneral Matrix Multiplication* (DGEMM), in Python. We compared the execution time for Python program, executing using Python 2.4.3, with automatically translated C++ code (using Shedkin [8]), and a hand-written C code. In *Python* code, the *list* is used as the input data structure. *C++* code is generated from Python after data type inference is performed, and a user-defined vector-like container is used as the input data structure. The hand-written C code uses the primitive array as the input data structure.

It turned out that the calculation time of the pure Python ($484.46\ sec$) is around 8 times higher than the generated *C++* code ($59.56\ sec$). This is primarily because in the C++ code, all the type inference and type checking is performed at the compilation stage. However, the wrapper functions around this user-defined data structure still incur significant overheads. Thus, the hand-written C program ($11.96\ sec$) is 5 times faster than the generated C++ code, and overall, 40 times faster than the interpreted execution of Python.

In addition to the performance issues noted here, there is another challenge. For obtaining performance, it is increasingly becoming important to parallelize execution on multi-core and many-core architectures. Complex data structures pose significant challenges in parallelization. Moreover, the most common way of parallelizing computational steps is to use existing libraries. These libraries, however, are based on flatter data structures, like multi-dimensional arrays. Thus, the use of nested and dynamic data structures can prohibit the use of these libraries, and the application cannot benefit from parallelization on multi-core or many-core architectures.

B. Overview of Our Translation Framework

We now give an overview of the approach we have developed in this paper. As a motivating example, we use the Python code in Figure 1. The nested loop shown at the bottom of the Figure is similar to the computation performed in DGEMM example.

Before Linearization

```
#Data set structure definition
class A:
  def __init__ (self, a1):
    self.a1 = a1
    self.a2 = len (a1)
class B:
  def __init__ (self, b1):
    self.b1 = b1
    self.b2 = len (b1)
points = []
```

```
#Data set initialization
for i in range (t):
  b1 = []
  for j in range (n):
    a1 = []
    for k in range (m):
      a1.append (...)
    b1.append (A (a1))
  points.append (B (b1))
```

```
#Data access before linearization
for i in range (t):
  for j in range (n):
    for k in range (m):
      ... = points[i].b1[j].a1[k] ...
```

Fig. 1. Python Code to Illustrate Translation Challenges

As stated earlier, we can significantly improve performance over interpreted execution of Python code by using existing tools for translating the code to C++. However, dynamic data structures still impose a significant performance penalty, and disallow the use of existing libraries for multi-code and many-core systems. One approach for addressing this problem could be to copy the data to a *flatter* data structure, just before the execution of the main computational loop. This way, the main computation step may operate at an efficiency that is similar to that of the hand-written C code. Moreover, the arrays can be passed to the existing libraries that would allow parallel execution of the main loop.

While this idea seems simple, it still involves several challenges. First, flattening nested dynamic data structures may not be trivial, and we need a mechanism to perform the translation and for maintaining the correspondence between the two sets of data structures. Second, the copying step itself can be expensive, especially, if the procedure has to be repeated several times. Thus, we need mechanisms to avoid unnecessary copying of the data. Third, we can store data in arrays and operate on it only if the data in the dynamic data structure is *homogeneous*. We need an efficient mechanism to determine this.

We have developed techniques to address these three challenges, and have implemented them as part of our overall

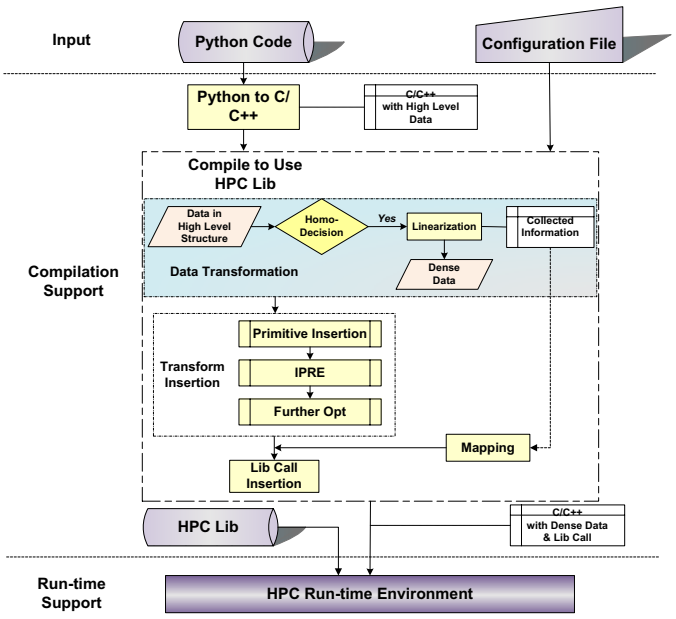

Fig. 2. Overview of the Translation Framework

framework. This framework is shown in Figure 2. There are three main stages in our translation process. In the first stage, the Python code is translated into C/C++ code, using the existing tool, Shedskin. Particularly, this tool transforms high-level containers in Python to pre-defined container classes in C++ (similar to those used in a template library like STL). Type checking and type inferencing is performed during this step.

In the second stage, the generated C/C++ code is translated further with an emphasis on the main computational steps. This is the key novel contribution of the paper, with algorithms for *Homogeneity Decision*, *Demand-Driven Inter-procedural Partial Redundancy Elimination*, and *Linearization* involved in this process. These methods are introduced in Sections III and IV, and have been implemented using the ROSE infrastructure [32].

In the last stage, the transformed C/C++ code with dense data structures is further analyzed to make appropriate library calls. This step is based on the existing work [24], [21], and the details are not described in this paper.

III. Insertion Algorithm

The objective of the insertion algorithm is to reduce the overhead caused by the *linearization* operation, which is done by reducing the frequency of execution of this statement. Our overall approach can be viewed as a two-level one:

Level 1: Insert a dense data structure (A, such as *array*) just before any usage of the high level structure (L, such as *list*). We copy the actual objects in L to A and replace L by A. This work can be followed by an optional step, in which we reorder the members in the objects according to our computational requirement, which can improve the data locality and the efficiency especially for data-intensive applications.

Level 2: In order to avoid multiple (unnecessary) copy operations, a powerful redundancy elimination algorithm, *inter-procedural partial redundancy elimination* (IPRE), is designed.

Level 1 optimization simply requires an ability to *linearize* the data in the dynamic data structure. The method for this is presented in the next Section. We focus on the second level optimization in the next 2 subsections.

A. Intra-procedural PRE Algorithm

Our Level 2 optimization involves a novel use of an existing partial redundancy elimination (PRE) algorithm, and its extension into a *demand-driven* inter-procedural algorithm. We initially show why our problem is related to PRE.

Along a certain *control flow path*, if a computation is performed more than once without any modification to its operands between them, it will be considered as partially (or fully) redundant. Over the last 30+ years, several PRE algorithms [27], [6], [17] can be applied to optimize the code. Similarly, in our work, if a copy operation is performed more than once along a certain path without any modification to the relative data elements, the copy operation can be treated as partially (or fully) redundant.

In order to explain the basic idea of the traditional PRE, Figure 3 shows an intra-procedural example. In the left-hand-side of this figure, a *Control Flow Graph* is given, while the transformed code by PRE is introduced in the right-hand-side. In our work, the IPRE algorithm is derived from an existing intra-procedural algorithm that is summarized in the Appendix [30]. This algorithm is chosen because of its conceptual simplicity.

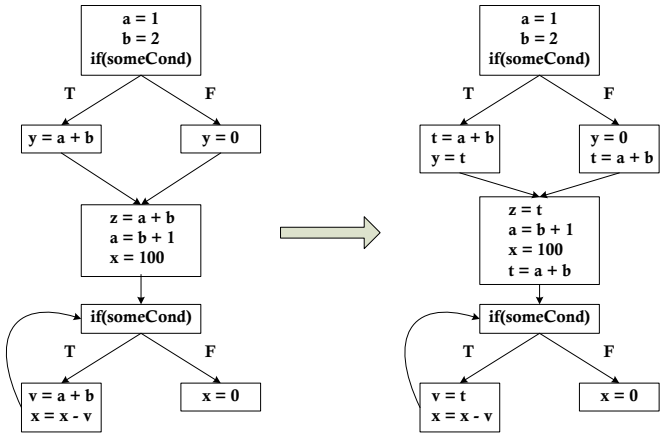

Fig. 3. An Example to Illustrate Basic PRE: Before (left) and After (right)

B. Inter-procedural PRE algorithm

For even a modest-sized application, the overheads of linearization cannot be reduced without applying PRE inter-procedurally. Though there have been a couple of efforts on developing an inter-procedural PRE algorithm [1], [18], we have developed a *demand-driven* inter-procedural algorithm, which analyzes procedures only if it is needed for placement of the linearization operations. In our applications, the key data structures are not modified frequently, so normally, there should be only a few linearization operations placement involved. Thus, our demand-driven algorithm results in analysis of only a small number of procedures from the application.

Our algorithm is based on the *inter-procedural control-flow graph* (ICFG), which has been widely used for inter-procedural analysis. This ICFG contains the control flow graphs (CFG) for the individual procedures. For each procedure p, an entry node $Entry_p$ and an exit node $Exit_p$ are defined. Each *call-site* to p is represented by two nodes:

```
void main (){
    List points;
    Initial_points (points);
    kmeans_reduction (points);
}
void kmeans_reduction (List points){
    List clusters;
    Initial_clusters (clusters, points);
    for (i = 0; i < iterations; i++){
        kmeans (points, clusters);
        update_clusters (clusters);
    }
}
void kmeans (List points, List clusters){
    for (point p in points){
        //min_cluster is the closest centroid
        min_cluster.min_distance = max (double);
        min_cluster.min_position = 1;
        for (cluster c in clusters){
          min_cluster = find_closest_centroid (p, c);
        }
        update_reduction_object (min_cluster);
    }
}
void update_clusters (List clusters){
    for (cluster c in clusters){
        //update the centroid by pre-defined reduction object
        c = ...reduction_object ...;
    }
}
```

Fig. 4. The C-like Pseudo-code for K-means Application

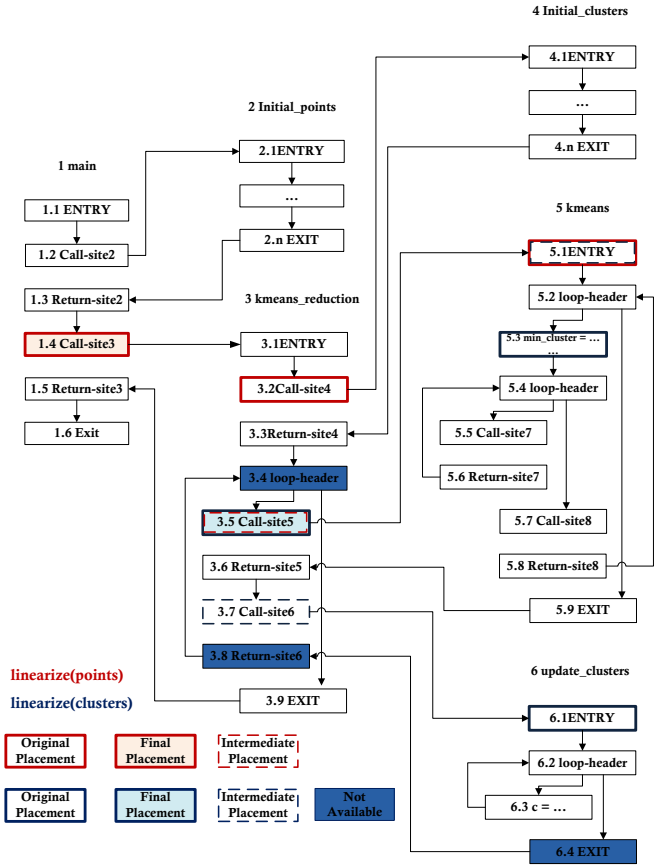

Fig. 5. The ICFG for K-means Application

$Call-site_p$ and $Return-site_p$. If a basic block contains a $Call-site_p$, it will be split into two basic nodes $B1$ and $B2$. There is an edge from $B1$ to the entry node of the procedure p, ($B1$, $Entry_p$), and similarly, there is an edge from the exit node of the procedure p to $B2$, ($Exit_p$, $B2$). In Figure 5, we show ICFG for the K-means application listed earlier in Figure 4.

Algorithm 1 analyze_all ($procedure\_set$, $linearize\_set$)

1: **for** each linearization expression $linearize(l_i) \in linearize\_set$ **do**
2: **for** each procedure $p_j \in procedure\_set$ **do**
3: intra-procedural analysis on $linearize(l_i)$ in p_j without considering the effect of call-sites
4: **end for**
5: pick-up p in which l_i is first define
6: p_{parent} = $p_{current}$ = p
7: **if** $l_i \in$ global_variables **then**
8: p_{parent} = $p_{current}$ =$main$
9: **end if**
10: analyze($p_{current}$, p_{parent})
11: **for** each procedure $p_j \in procedure\_set$ **do**
12: final insertion and deletion
13: **end for**
14: **end for**

Our IPRE method is shown through Algorithms 1 and 2. In our inter-procedural framework, we assume that inter-procedural pointer-analysis [13] and alias-analysis [12] have been performed in the preprocessing stage and all the variables that point to the same space are labeled with the same name.

In order to explain our algorithm, we use K-means example. First, an initial placement of the linearization operation is performed. After this stage, in Figure 5, the linearization operations for the list $points$ are placed at the beginning of the

Algorithm 2 analyze ($p_{current}$, p_{parent})

1: **for** each node $node_i$ in $p_{current}$ **do**
2: **if** $node_i$ includes procedure p **then**
3: p_{parent} = $p_{current}$
4: $p_{current}$ = p
5: analyze ($p_{current}$, p_{parent})
6: **else if** $node_i$ = $EXIT_{p_{current}}$ **then**
7: **if** $p_{current}$ = p_{parent} **then**
8: return {*arrive at the outer-most procedure*}
9: **else if** $p_{current}$ is completely transparent with l_i **then**
10: return {*nothing is affected*}
11: **else if** $p_{current}$ includes modification to l_i **then**
12: mark the availability of $AVIN_{RETURN-SITE_{p_{current}}}$ in p_{parent} according to the value of $AVOUT_{EXIT_{p_{current}}}$
13: **if** linearize(l_i) is safe at the $ENTRY_{p_{current}}$ **then**
14: mark $CALL-SITE_{p_{current}}$ as $ANTIN/ANTOUT$ and $COMP$
15: mark $ENTRY_{p_{current}}$ as $AVIN$
16: **end if**
17: propagate effect by work-list in p_{parent} and $p_{current}$
18: return
19: **end if**
20: **end if**
21: **end for**

node 3.2 and the end of the node 5.1. Similarly, linearization operations for the list $clusters$ are placed at the end of the node 5.3 and at the end of the node 6.1.

In next stage, we use the method presented through Algorithm 1 and 2. Initially, intra-procedural analysis is performed in the procedure(s) where the initial placement has been done, without considering the effect of the functions calls. During

this phase, we apply only Equations 1 through 10 in Figure 12, i.e. insertion or deletion logic is not computed.

Next, we move to the inter-procedural phase. If the list parameter is anticipable at the entry of current procedure, we will pull this linearization operation out of p, and try to propagate it further. For example, by this *pull out* strategy, the linearization operation *linearize(points)* can be pulled from the procedure *kmeans* to *kmeans_reduction*, and until the *main* function. Finally, we will mark the node 1.4 in Figure 5 as $COMP$ and $ANTOUT$ (stronger than $SPANTOUT$) by the line of 14 of the Algorithm 2. From the intra-procedural analysis, we have know that the node 1.4 has already been marked as $\neg SPAVIN$. Based on all of these, we can know that the final insertion for *linearize(points)* can happen at the beginning of the node 1.4. All others placements will be deleted since we have already marked them as $AVIN$ (according to the line 15 of the Algorithm 2 and the propagation operation), and there are no further modifications to $points$.

For a procedure call p from the current procedure $p_{current}$, we consider two possibilities. First, if p is *completely transparent* relative to the parameter list of the copy statement, no further analysis is done on p. Second, if p is not transparent, we just need to copy the availability from the Exit point of p. For example, in Figure 5, in the Exit node of the procedure *update_clusters*, *linearize(clusters)* is not available, so in the node 3.8, it is also not available, which will cause *linearize(clusters)* is $\neg AVIN$ in the node 3.5. Subsequently, we can infer that the final insertion for *linearize(clusters)* will happen at the beginning of the node 3.5 and other placements will be eliminated.

K-means application is an iteration process: before the reduction loop, the input data set *points* is initialized without any further modification during the whole process, and the output data set *clusters* is updated in each loop. From Figure 5, it is easy to know that after our elimination, the final placement of *linearize(points)* is out of the reduction loop, and the placement of *linearize(clusters)* is within the reduction loop, which is coherent to the basic logic as above.

C. Checking Homogeneity of a List

One of the requirements for converting a dynamic data structure to an array is that each element of the original dataset is of the same type. In this section, we describe an algorithm we have developed for this purpose.

Our description here assumes a list structure, though the algorithm can easily be applied to any other dynamic data structure. This decision algorithm is modeled as a *data flow analysis* problem, similar to the well-known *Constant Propagation* problem, for which many algorithms have been developed [3], [36]. Here, only the intra-procedural version is introduced, and the inter-procedural version can be developed easily similar to how we developed the IPRE algorithm above.

The entire algorithm can be expressed as a four-tuple $< G, D, L', F >$, where, $G = (N, E)$ is a control flow graph. D is the direction of the data flow, which is $FORWARDS$ here. L' is a three-tuple $< V', \wedge, m >$, in which there are three elements: V', $\wedge$ and m. V' is the domain of values, and each element in it is in this form: $(type_1, type_2, \ldots)$, i.e., a cross-product of the basic lattice L_i, in which, *UNDEF* is the top element $\top$, and *NON-HOM* is the bottom element $\bot$. In L', $\wedge$ is a meet operator, which follows the common definition of $\wedge$ for the product lattice, i.e, it is defined as:

$$(type_1, type_2, \ldots) \wedge (type'_1, type'_2, \ldots)$$
$$= (type_1 \wedge type'_1, type_2 \wedge type'_2, \ldots)$$

m is a map function used to map the list definition to the lattice. For example, the result of $m(list_i)$ will be $type_i$. Thus, each element in V' can also be expressed in this form $(m(list_1), m(list_2), \ldots)$. There is a special map function, m_0, which can initialize the type of the list variable into *UNDEF*.

Returning to the last element of the four-tuple, $F : V' \rightarrow V'$ is the domain for transfer functions. It has an identity function f_i, such that $f_i(x) = x$ for all x in V'. Like any standard intra-procedural data flow algorithm, we can consider two levels: within *basic block*, and inter *basic block*. Based upon this, we can classify the elements in F into two groups: working on statements within a *basic block* and working on *basic blocks*. The rules for the former case, denoted as f_s, can be defined as follows:

1) If the statement (s) is irrelevant to the given list, f_s is the identity function f_i;
2) If s is relevant to the given list, $list_a$, then for any $list_i \neq list_a$, $f_s(m(list_i)) = m(list_i)$, and for $list_a$, $m'(list_a) = f_s(m(list_a))$.

We further consider the following cases:

1) if s is a member function call without adding any new elements in the list, such as $list_a.remove()$, $m'(list_a) = m(list_a)$;
2) if s is a member function call adding an element, such as $list_a.append(x)$, or $list_a.insert(k, x)$, $m'(list_a) = m(list_a) \wedge typeof(x)$;
3) if s is a concatenation operation, and more than 1 element, i.e., $[x_1, x_2, \ldots]$ are added to the $list_a$, there are two cases: i) if the type of the new list is already calculated as $type'$, then $m'(list_a) = m(list_a) \wedge type'$, and ii) if this is not the case, examine the type of new elements by $type' = typeof(x_1) \wedge typeof(x_2) \wedge \ldots$, and then $m'(list_a) = m(list_a) \wedge type'$.

TABLE I
HOMOGENEITY DECISION EXPRESSION (GLOBAL LEVEL)

	Non-Hom decision Expression (Global Level)
Domain	Sets of Basic Blocks
Direction	Forward
Transfer Function	$f_B = x \wedge LOC[B]$
Boundary	$OUT[ENTRY] = \emptyset$
Meet($\wedge$)	Defined as above
Equations	$IN[B] = \wedge_{P,pred(B)} OUT[P]$; $OUT[B] = f_B(IN[B])$
Initialization	$OUT[B] = \emptyset$

After applying f_s to all statements in each *basic block*, we can get the result for each block at the point of $OUT[B]$. We refer to them as $LOC[B]$. The rule for the transfer functions working on each basic block, denoted as f_B, are summarized in the Table I.

IV. LINEARIZATION AND MAPPING ALGORITHM

The methods presented in the previous section have explained under what conditions contents of a dynamic data structure can be copied into a dense and flat data structure, and where the copy operation can be placed. In this section, we introduce *linearization* and *mapping* algorithms, with the goal of translating the high-level dynamic data structures in Python to low-level dense memory buffer in C++. This, in turn, will allow use of HPC libraries for multi-core and many-core architectures.

Specifically, we need to create a low-level continuous data storage (D_s) from the high-level data view (D_v). The entire process can be formally viewed as of computing the following

two functions: 1) $F_t \subseteq \{f \mid f : D_v \rightarrow D_s\}$, a *linearization* function, which can transform the high level data view to the low level data storage, and 2) $M \subseteq \{m \mid m : D_v \rightarrow D_s\}$, a *mapping* function created to enable mapping of the computations to the low-level data layout.

A. Linearization

Algorithm 3 computeLinearizeSize(Xs)

```
1: size = 0
2: if Xs.type = isPrimitive then
3:     size = sizeof(Xs)
4: else if Xs.type = isIterative then
5:     for x in Xs do
6:         size += computeLinearizeSize(x)
7:     end for
8: else if Xs.type = isStructureType then
9:     for each member m in Xs do
10:        size += computeLinearizeSize(m)
11:    end for
12:    ...
13: end if
14: ...
15: return size
```

Algorithm 4 linearizeIt(Xs, $size$)

```
1: ▷ allocate memory with the size of size
2: if Xs.type = isPrimitive then
3:     copy(Xs)
4: else if Xs.type = isIterative then
5:     for x in Xs do
6:         linearizeIt(x)
7:     end for
8: else if Xs.type = isStructureType then
9:     for each member m in Xs do
10:        linearizeIt(m)
11:    end for
12:    ...
13: end if
14: ...
15: return addressOfLinearizeData
```

The basic *linearization* is a two-step algorithm which includes two functions: *computeLinearizeSize*, shown as Algorithm 3 and *linearizeIt*, shown as Algorithm 4. The first function is used to compute the data size while the second one is used to copy the actual data to the continuous memory space.

Let us revisit the code from Figure 1, where a very common example of using the user defined input data structure in Python was shown. Figure 6 shows information that needs to be collected during the linearization process to enable code generation for the usage of the linearized data structure.

B. Mapping

The *mapping* algorithm can be divided into two stages: in the first stage, collecting the necessary information during the *linearization* process; in the second stage, computing the projected index of the low level data storage D_s from the collected information and the original index in D_v by the recursive strategy in algorithm 5. The parameters used by this algorithm are summarized in Table II.

Figure 6 shows the information that should be collected to apply the mapping algorithm. Most of the information should be collected during the *linearization* stage, while the index information is obtained from the usage loop. The entire mapping process is recursive. It starts from the outer-most level and terminates with the inner-most level. At each level, we calculate the offset caused by the index and the position offset.

Information Collected During Linearlization

```
levels = 3;
unitSize[levels] = {unitSize_B, unitSize_A, sizeof(data_type_a1)};
unitOffset[levels-1][2] = {{unitOffset_B[]}, {unitOffset_A[]}};
unitOffset_B[2] = {0, unitSize_A × n }
unitOffset_A[2] = {0, sizeof(data_type_a1) × m}
position[levels-1][2] = {{0, 1}, {0, 1}};
{*This should be collected in the accumulate function*}
myIndex[levels] = {i, j, k};
```

Data access after linearization

```
for(i = 0; i <t; i++){
    for(j = 0; j <n; j++){
        for(k = 0; k <m; k++){
            index = computeIndex(unitSize, unitOffset,
                    myIndex, position, 0, levels);
            ... = linea_points[index] ...
        }
    }
}
```

Fig. 6. The Example of Using Linearization and Mapping Functions

Algorithm 5 computeIndex($unitSize[]$, $unitOffset[][]$, $myIndex[]$, $position[][]$, i, $levels$)

```
1: ▷ During the linearization phase, collecting necessary information
2: if i < levels − 1 then
3:     index = unitSize[i] × myIndex[i] + unitOffset[i][position[i][]]
4:     index += computeIndex(unitSize[], unitOffset[][], myIndex[], position[][], i++, levels)
5: else
6:     index = unitSize[i] × myIndex[i]
7: end if
8: return index
```

V. Implementation and Experiments

In this section, we describe a prototype implementation of our framework and evaluate it by generating code for execution of *data-intensive* applications on a multi-core system, and *computation-intensive* applications on a GPU.

A. Implementation Overview

Python code was translated to C++ using an existing tool, Shedskin [8]. Our transformations were implemented on top of the ROSE compiler infrastructure [32]. ROSE is a powerful tool that supports program analysis and source-to-source transformations for C/C++, FORTRAN, and other languages. After the transformations are applied, low-level HPC libraries are invoked to support mapping on the multi-core and many-core libraries. Particularly, we used a data mining middleware for mapping data-intensive applications to multi-core architectures, and used existing libraries to execute linear algebra operations on GPUs [23]. All these libraries/middle-ware expect the data to be in multi-dimensional arrays, and cannot support processing of nested or dynamic data structures. The code generation was based on our earlier work, and the details are not presented here.

TABLE II
DESCRIPTIONS OF THE PARAMETERS IN MAPPING ALGORITHM

Collected During Linearization
$unitSize[]$ 1-Dimensional Array. It stores the unit size of the elements in each level with unitSize[$levels-1$] storing the inner-most elements.
$unitOffset[][]$ 2-Dimensional Array. It stores the offsets of the variables at each level. The first dimension is used to indicate the level and the second one indicates the start positions of the variables at current level.
$position[][]$ 2-Dimensional Array. It provides the position information for calculating the $unitOffset$.
$levels$ The total number of levels of the data.
Collected From D_v
$myIndex[]$ 1-Dimensional Array. It records the index for each level.
i An indicator to show the current level. Normally, it starts from 0 indicating that the current level is the outer-most.

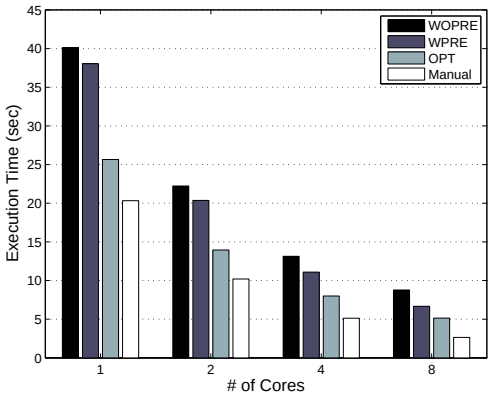

Fig. 7. K-means: Comparison of Performance of Different Versions (800 MB dataset, $k = 100$, $iter = 1$)

B. Evaluation Goals and Platforms

The objective of our evaluation is to compare the execution time of the original Python code (*Python*), Shedskin generated C/C++ code (*Gen C++*), transformed code with and without IPRE optimization (*WOPRE* and *WPRE*, respectively), and the hand-written C/C++ with library functions (*Manual*).

Our experiments are conducted on the following platforms. A multi-core machine with AMD Opteron(tm) Processor (2.6 GHZ frequency) and main memory size of 32 GB was used for data-intensive applications. The GPU used for compute-intensive applications was a Quadro FX 5800 GPU, with 240 cores and 4 GB memory.

C. Experiments with Data-Intensive Applications

We invoked a data-intensive computing library from transformed C++ code, and compare the performance of different versions we listed earlier. We used two popular data mining applications, which are K-means clustering and PCA.

An 800 MB representative dataset was used for K-means. In our experiments, we control the computation workload by modifying the iteration numbers. Very similar to the DGEMM example in Section II, the calculation time of the Python code, which uses a list as the main input data structure, is much longer than the generated code and the transformed code.

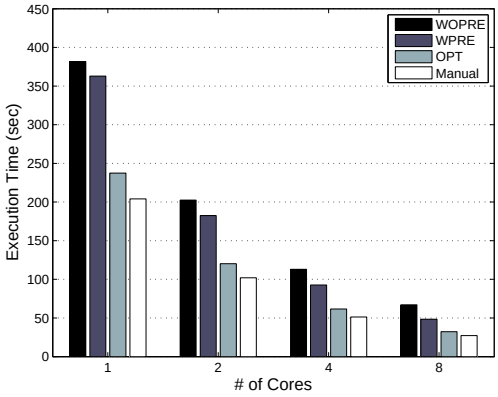

Fig. 8. K-means: Comparison of Performance of Different Versions (800 MB dataset, $k = 100$, $iter = 10$)

For example, even to a much smaller data set (8 MB), the calculation time of the *Python* code interpreted by *Python 2.4* is 109.60 seconds for 1 iteration and 1122.96 seconds for 10 iterations. For the data set of 800MB, the execution time of the *Gen C++* code is 59.28 seconds for 1 iteration and 593.06 seconds for 10 iterations.

In Figure 7 and Figure 8, we report the calculation time of the code transformed by our framework. From Figure 7, we can see that comparing to the *Gen C++* code, the efficiency of the sequential version of our transformed code for 1 iteration is improved by more than 30% even including the linearization overhead of the input data set (*WOPRE* version). Comparing with the *WPRE* version, we found that IPRE can help overcome nearly 50% of the linearization overhead, which is consistent to our analysis in Section III. In K-means, because the centroid set is a frequently accessed data structure, we can also linearize and apply the IPRE on it as described in Section III, resulting in a version we refer to as *OPT*. By comparing the versions *OPT* and *Manual*, we can see that their performance is very similar to each other, and the overhead caused by linearization and mapping is within 30% for 1 iteration. On the other hand, by comparing the sequential versions of *OPT* and *Gen C++*, we can see that by our optimization framework, the efficiency of the compiled code can be improved by a factor of more than 2 for the sequential version, and furthermore, we have enabled use of a parallel library.

From the comparison of Figure 7 and Figure 8, we can see that the linearization overhead can be reduced to a large extent by our IPRE method when there are multiple iterations. Finally, for the sequential version, the overhead of the *OPT* version is around 10% of the best version, which is mainly caused by the *mapping* operations and scalable to the number of processors. That is why we see good scalability of the *OPT* version. The relative impact of our optimizations is even more significant for parallel versions, since linearization is performed sequentially.

The datasets used for PCA experiments has 1000 rows and 100,000 columns. The calculation time of the *Python* code is very long, for example, even to a much smaller data set (1000×1000) it takes 634.45 seconds. The *Gen C++* code is also relatively slow, for example, to the data set of $1000 \times 100,000$ it takes 3280 seconds. By using our framework, the efficiency can be improved obviously, however, the IPRE optimization must be applied to the linearization of the input data set. Without the IPRE algorithm, the linearization is inserted in the inner-most loops, resulting in $\Theta(row^2)$ times copy operations to the input data set that is a

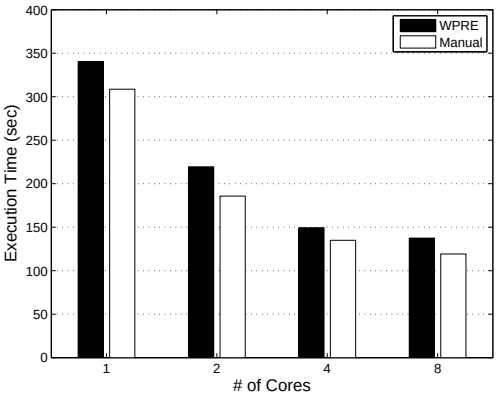

Fig. 9. PCA: Comparison of Performance of Different Versions ($row = 1000$, $column = 100,000$)

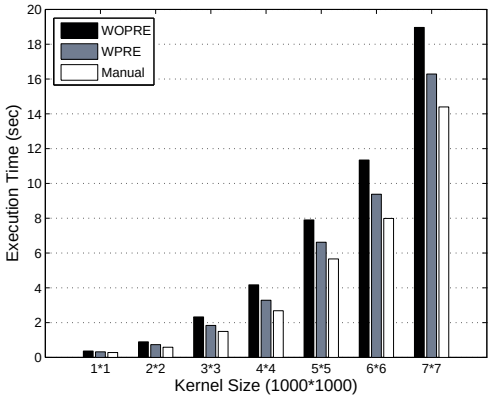

Fig. 10. Experiment Results for DGEMM

very large overhead. In Figure 9, we compare the calculation time of the *WPRE* version generated by our framework and *Manual* versions. As stated above, *WOPRE* version data is not shown, because it is extremely slow. From the comparison, we can see that the efficiency of the *WPRE* version is very similar to the *Manual* version, and the overhead caused by the linearization and mapping operation is around 10% to 20% for both sequential and parallel versions. Especially, the efficency of the sequential *WPRE* version is improved by a factor of 10 comparing to the *Gen C++* code.

D. Scaling Compute-Intensive Applications with a GPU

GPU has been gaining popularity in recent years because of their very favorable performance to cost ratio. Many GPU related libraries and automatic code generators have been developed in recent years. In our experiments, CUBLAS libraries [28] and tensor contractions [23] generated code are used for accelerating the execution of two linear algebra kernels written in Python.

The first linear algebra kernel is DGEMM. The implementation from the CUBLAS library can be invoked to replace the sequential computations in the Python implementation. Because the mapping function is not needed in this case, the mapping overhead is not considered in this and the next example. We experimented with seven datasets, which range from 1000×1000 to 7000×7000. The results of the experiment are shown in Figure 10. By comparing the results on the 1000×1000 dataset with the example in Section II, we can see that the performance of the CUBLAS version is much better than the *Python* code (more than 1000 times speedup) and *Gen C++* code (around 163 times speedup), even before the optimizations are applied.

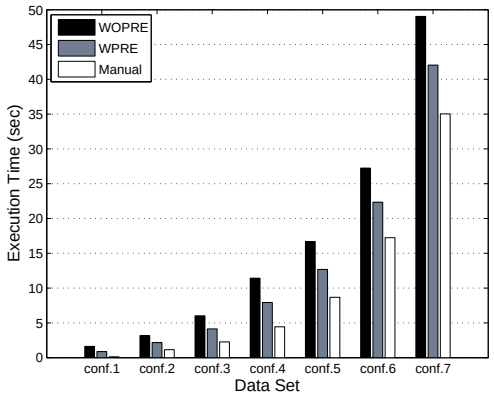

Fig. 11. Experiment Results for Tensor Multiplication

In this application, the IPRE optimization is mainly used to eliminate the linearization overhead during the matrix dimension validation. Since there is no modification between this stage and the main loop computation, there is no need to linearize the input matrices multiple times. This is also applied to the next experimental case. From the comparison between the versions of *WPRE* and *WOPRE* in Figure 10, we can see that the linearization overhead can be reduced by more than 50% by using IPRE. And also the linearization overhead becomes less significant with the increase in the data set sizes, and when the size of the kernel is 7000×7000, the linearization overhead is reduced to be less than 15% with our IPRE method comparing to the best *Manual* version.

The second linear algebra kernel is tensor contraction, which is a multi-dimensional matrix multiplication. In pure Python, if we want to perform such a computation, a highly nested list structure needs to be used, which decreases the performance severely. Thus, our transformations are even more crucial. The following expression was used in our work:

$$result[h_3, h_1, h_2, p_5, p_4, p_6] += x[h_7, p_4, p_5, h_1] \times y[h_3, h_2, p_6, h_7]$$

Figure 11 illustrates the execution time for different datasets of increasing sizes. Both *WOPRE* and *WPRE* are the versions generated by our framework and the *Manual* one is the version written manually to feed into the Code Generator. Again, the *Python* and *Gen C++* codes are very slow, i.e. even for *config1* they run for 261.39 seconds, 16.85 seconds, respectively. Details of both these versions are not shown here. The effect of the IPRE optimization is shown in this experiment by comparing the versions of *WOPRE* and *WPRE* that the linearization overhead is decreased by around 50%. From the comparison between the transformed code (*WOPRE* and *WPRE*) and the *Manual* version in Figure 11, we can see that the linearization overhead is very large when the data set is relatively small. However, the overhead reduces with increasing in dataset sizes. For instance, in *config7*, comparing the versions of *WPRE* and *Manual*, the linearization overhead is already smaller than 20% with IPRE.

VI. Related Work

We now compare our work with related research efforts.

Given the popularity of Python, there have been several efforts focusing on improving Python's efficiency. These efforts can be classified into two groups, which are adding extension libraries or constructs to *Pure* Python, and compiling Python to other languages, such as C/C++ or even CUDA. NumPy and SciPy [29], [15] are examples of the former, where the inefficiency caused by the dynamic data structure usage in pure Python is substantially reduced by adding an N-dimensional array object. For multi-processing, these efforts have been

integrated with PyMPI [26]. PyCUDA and PyOpenCL [16] are two library extensions where GPU code can be invoked from Python. More recently, Catanzaro *et al.* [4] developed a data parallel language named Copperhead which is based on Python. Compared to the above efforts, our goal is clearly different, in the sense that we start with pure Python, and automatically replace dynamic data structures with arrays.

In efforts that compile Python to other languages, prominent ones include Cython [2] and Pyrex [9], where type-annotated Python is compiled to C, and Shedskin [8], where a subset of Python is compiled to C++. To use multi-core or many-core system, Garg *et al.* [10] developed a framework to compile Python code to a hybrid CPU-GPU environment. The initial application is assumed to use array-based constructs in their work.

Applications with irregular pointer-based data structures have also received much attention. Pingali *et al.* [11], [31], [19] focus on exploring the nature of the irregular algorithms and improve their parallelism and efficiency. The application we target, in comparison, have regular data parallelism. Lattner *et al.* [20] proposed an *automatic pool allocation* method to manage the data structure layout in the heap to optimize the pointer intensive programs. Spek *et al.* [35], [34] developed a way to transform the recursive pointer-based data structure and related loops to the array-based data structure and counted loop structure that can be optimized by traditional methods.

There is significant research about linearization and reorganization of data and operations to reduce the cache misses or memory latency. Luk and Mowry [22] invented a *Linearization* method somewhat similar to our linearization method to support data prefetching. Ding and Kennedy [7] proposed a set of algorithms, including locality grouping and dynamic data packing, to improve the cache performance. Strout *et al.* [33] designed a compile-time framework to compose run-time data and iteration reordering transformation. Zhong *et al.* [37] proposed a structure splitting and array regrouping strategy based on the concept of *Whole Program Reference Affinity*. Mannarswamy *et al.* [25] presented a *Region Based Structure Layout* transformation method to reorganize the linked list-based data structures to increase the cache line utilization.

VII. Conclusions

In order to bridge the gap between the productivity and the performance in HPC applications, this paper has presented a framework to compile pure Python to invoke existing multi-core and many-core libraries. To enable such optimizations, a demand-driven inter-procedural algorithm has been developed. We have also developed a novel *Homogeneity Checking* algorithm, and a set of *Linearization-Mapping* schemes. By these algorithms, dynamic data constructs in Python can be transformed into dense memory buffer that can be accepted by the low level libraries.

Two data-intensive and two linear algebra applications were used to evaluate our framework. The evaluation results show that the code generated by our framework is only 10% to 20% slower than the hand-written C code that invokes the same libraries. IPRE optimization we perform turns out to be significant for improving performance in most cases. Moreover, the code generated by our framework outperforms interpreted Python and the C++ code generated by an existing tool by one to two orders of magnitude.

Acknowledgments

We would like to thank our reviewers, whose insightful comments and suggestions were very helpful. This work is supported by NSF grant CCF-0833101.

References

[1] Gagan Agrawal, Joel Saltz, and Raja Das. Interprocedural Partial Redundancy Elimination and Its Application to Distributed Memory Compilation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 258–269. ACM Press, June 1995. ACM SIGPLAN Notices, Vol. 30, No. 6.

[2] S. Behnel, R. Bradshaw, and D. Seljebotn. Cython: C-extensions for Python, 2008.

[3] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural Constant Propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.

[4] Bryan C. Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP*, pages 47–56. ACM, 2011.

[5] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[6] D. M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[7] Chen Ding and Ken Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *PLDI*, pages 229–241, 1999.

[8] M. Dufour. Shed Skin-An Experimental (Restricted) Python to C++ Compiler (2009-09-30).

[9] G. Ewing. Pyrex. A Language for Writing Python Extension Modules, 2006.

[10] Rahul Garg and José Nelson Amaral. Compiling Python to a Hybrid Execution Environment. In David R. Kaeli and Miriam Leeser, editors, *GPGPU*, volume 425 of *ACM International Conference Proceeding Series*, pages 19–30. ACM, 2010.

[11] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. Unordered: a Comparison of Parallelism and Work-Efficiency in Irregular Algorithms. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP*, pages 3–12. ACM, 2011.

[12] M. Hind, M. Burke, P. Carini, and J.D. Choi. Interprocedural Pointer Alias Analysis. *TOPLAS*, 21(4):848–894, 1999.

[13] M. Hind and A. Pioli. Which Pointer Analysis Should I Use? In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 113–123. ACM, 2000.

[14] Wei Jiang, Vignesh Ravi, and Gagan Agrawal. A Map-Reduce System with an Alternate API for Multi-Core Environments. In *Proceedings of Conference on Cluster Computing and Grid (CCGRID)*, 2010.

[15] E. Jones, T. Oliphant, and P. Peterson. SciPy: Open Source Scientific Tools for Python. *http://www. scipy. org/*, 2001.

[16] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. *Arxiv preprint arXiv:0911.3456*, 2009.

[17] J. Knoop, O. Ruething, and B. Steffen. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 224–234, San Francisco, CA, June 1992.

[18] J. Knoop and B. Steffen. Optimal Interprocedural Partial Redundancy Elimination. In *Proceedings of the Poster Session of the 4th International Conference on Compiler Construction (CC92)*, 1992.

[19] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *SPAA*, pages 217–228. ACM, 2008.

[20] Chris Lattner and Vikram S. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 129–142. ACM, 2005.

[21] Xiaogang Li and Gagan Agrawal. Supporting XML-Based High-level Interfaces Through Compiler Technology. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC)*, October 2003.

[22] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *ASPLOS*, pages 222–233, 1996.

[23] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski. Acceleration of Streamed Tensor Contraction Expressions on GPGPU-Based Clusters. In *Proceedings of the 2010 IEEE Cluster*. IEEE, 2010.

[24] Wenjing Ma and Gagan Agrawal. A Compiler and Runtime System for Enabling Data Mining Applications on GPUs. In *Proceedings of Principles and Practices of Parallel Programming (PPoPP)*, February 2009.

[25] Sandya S. Mannarswamy, Ramaswamy Govindarajan, and Rishi Surendran. Region Based Structure Layout Optimization by Selective Data Copying. In *PACT*, pages 338–347. IEEE Computer Society, 2009.
[26] P. Miller. PyMPI-An Introduction to Parallel Python Using MPI. *Livermore National Laboratories, Jan*, 2002.
[27] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.
[28] C. NVIDIA. CUBLAS Library. *NVIDIA Corporation, Santa Clara, California*, 2008.
[29] T.E. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing, 2006.
[30] Vineeth Kumar Paleri, Y. N. Srikant, and Priti Shankar. Partial Redundancy Elimination: a Simple, Pragmatic, and Provably Correct Algorithm. *Sci. Comput. Program.*, 48(1):1–20, 2003.
[31] Dimitrios Prountzos, Roman Manevich, Keshav Pingali, and Kathryn S. McKinley. A Shape Analysis for Optimizing Parallel Graph Programs. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 159–172. ACM, 2011.
[32] Daniel J. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.
[33] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-Time Composition of Run-Time Data and Iteration Reorderings. In *PLDI*, pages 91–102. ACM, 2003.
[34] Harmen L. A. van der Spek, Sven Groot, Erwin M. Bakker, and Harry A. G. Wijshoff. A Compile/Run-time Environment for the Automatic Transformation of Linked List Data Structures. *IJPP*, 36(6):592–623, 2008.
[35] Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. How to Unleash Array Optimizations on Code Using Recursive Data Structures. In Taisuke Boku, Hiroshi Nakashima, and Avi Mendelson, editors, *ICS*, pages 275–284. ACM, 2010.
[36] M.N. Wegman and F.K. Zadeck. Constant Propagation with Conditional Branches. *TOPLAS*, 13(2):181–210, 1991.
[37] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. In William Pugh and Craig Chambers, editors, *PLDI*, pages 255–266. ACM, 2004.

VIII. Appendix

This section summarizes the main steps in the Partial Redundancy Elimination method developed by Paleri *et al.* [30]. While this algorithm uses most of the same ideas as the original algorithm by Morel and Renvoise [27], as well as the subsequent algorithms by Dhamdhere [6] and Knoop *et al.* [17], it is conceptually simpler and has other properties, like the fact that it does not require any splitting of edges.

The data-flow equations of this algorithm are shown in Figure 12, and the terms are explained in Table III.

This algorithm can be divided into two phases: the *local phase* and the *global phase*. The local phase is applied to each *basic block* to reduce the redundancy within each *basic block*. After it, only the *first* and the *last* computation of the expression in this block will be considered.

Focusing now on the global phase, from Equation 1, we can know an expression is available at the entry of a *basic block*, if it is available at the exit of all the predecessor blocks. An expression is available at the exit of a *basic block*, if it is locally available or available at the entry of the current *basic block* without any operands modification in it (Equation 2). Similarly, from Equation 3 and Equation 4, we can know an expression is anticipable at the exit of a *basic block* if it is anticipable at the entry of all the successor blocks, while an expression is anticipable at the entry of a *basic block* if it is locally anticipable or anticipable at the exit of the current *basic block* without any operands modification in it.

The most interesting part of this algorithm is that it focuses on the *safe* points (SAFEIN and SAFEOUT), the points where we can insert the computation of some expression without introducing a new value along any path. The final insertion points and replace points are decided by Equations 11 to 14 based on the operators and terms in Table III.

$$AVIN_i = \begin{cases} FALSE & \text{if i = s,} \\ \prod_{j \in pred(i)} AVOUT_j & \text{otherwise,} \end{cases} \quad (1)$$

$$AVOUT_i = COMP_i + AVIN_i \cdot TRANSP_i. \quad (2)$$

$$ANTOUT_i = \begin{cases} FALSE & \text{if i = e,} \\ \prod_{j \in succ(i)} ANTIN_j & \text{otherwise,} \end{cases} \quad (3)$$

$$ANTIN_i = ANTLOC_i + ANTOUT_i \cdot TRANSP_i. \quad (4)$$

$$SAFEIN_i = AVIN_i + ANTIN_i, \quad (5)$$

$$SAFEOUT_i = AVOUT_i + ANTOUT_i. \quad (6)$$

$$SPAVIN_i = \begin{cases} FALSE & \text{if i = s or } \neg SAFEIN_i, \\ \sum_{j \in pred(i)} SPAVOUT_j & \text{otherwise,} \end{cases} \quad (7)$$

$$SPAVOUT_i = \begin{cases} FALSE & \text{if } \neg SAFEOUT_i, \\ COMP_i + SPAVIN_i \cdot TRANSP_i & \text{otherwise.} \end{cases} \quad (8)$$

$$SPANTOUT_i = \begin{cases} FALSE & \text{if i = e or } \neg SAFEOUT_i, \\ \sum_{j \in succ(i)} SPANTIN_j & \text{otherwise,} \end{cases} \quad (9)$$

$$SPANTIN_i = \begin{cases} FALSE & \text{if } \neg SAFEIN_i, \\ ANTLOC_i + SPANTOUT_i \cdot TRANSP_i & \text{otherwise.} \end{cases} \quad (10)$$

$$INSERT_i = COMP_i \cdot SPANTOUT_i \cdot (\neg TRANSP_i + \neg SPAVIN_i), \quad (11)$$

$$INSERT_{(i,j)} = \neg SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j, \quad (12)$$

$$REPLACE_{i_f} = ANTLOC_i \cdot (SPAVIN_i + TRANSP_i \cdot SPANTOUT_i), \quad (13)$$

$$REPLACE_{i_l} = COMP_i \cdot (SPANTOUT_i + TRANSP_i \cdot SPAVIN_i), \quad (14)$$

Fig. 12. Basic Intra-procedural PRE Data Flow Equations

TABLE III
Terms Used in the PRE Data Flow Equations

Symbols
$\cdot, \Pi$: Boolean conjunctions;
$+, \Sigma$: Boolean disjunctions;
$\neg$: Boolean negation.
Local properties
$TRANSP_i$: *transparent* In node i, if the operands of the expression are not modified;
$COMP_i$: *locally available* In node i, if there is at least one computation of the expression E, and *including* and after the *last computation*, there is no modification of the operands of E;
$ANTLOC_i$: *locally anticipable*. In node i, if there is at least one computation of the expression E, and before the *first computation*, there is no modification of the operands of E.
Global properties
$AVIN_i/AVIOUT_i$ The expression is available at the entry/exit of node i;
$ANTIN_i/ANTIOUT_i$ The expression is anticipable at the entry/exit of node i;
$SAFEIN_i/SAFEOUT_i$ The entry/exit of node i is safe. A point p is safe for some expression E, if we insert a computation of E at p without introducing any new value on any path through p;
$SPAVIN_i/SPAVOUT_i$ The expression is safe partial available at the entry/exit of i;
$SPANTIN_i/SPANTOUT_i$ The expression is safe partial anticipable at the entry/exit of i;
$INSERT_i/INSERT_{(i,j)}$ The computation of the expression should be placed before the *last computation* in node i; or on the edge between nodes i and j;
$REPLACE_{i_f}/REPLACE_{i_l}$ The replacement of the expression should be happened to the *first* / *last* computation in node i.