A Compiler-Based Tool for Array Analysis in HPC Applications

Ahmad Qawasmeh, Barbara Chapman
Dept. of Computer Science
University of Houston
Houston, Texas
Email: {arqawasm,chapman}@cs.uh.edu

Amrita Banerjee *
Petroleum Geo-Services
Houston, Texas
Email: Amrita.Banerjee@pgs.com

***Abstract*— Array region analysis plays a significant role in various optimizations at compile time. Displaying array access information efficiently in HPC applications has been a vital challenge for scientists and developers for the past few years. Dragon array region analysis tool is a powerful and interactive tool that was built on top of the OpenUH compiler, an open source C/C++/Fortran compiler, that supports OpenMP and CAF programming models. We have extended the linear-based Region analysis method and the high level IR (WHIRL) of OpenUH to visualize the static and interprocedural array region accesses, the frequency of these accesses per access mode, the access mode in which the array is processed, the number of dimensions, the size of each dimension, the total size in bytes allocated to this array statically, and the memory location. We have also defined the access density term which illustrates the frequency of accesses per bytes allocated to these arrays. The information provided enables users to efficiently develop and optimize HPC applications by understanding procedure side effects and finding inefficiencies in defining arrays, which guides to a better memory allocation and cache usage. Moreover, we demonstrate the access density of the portions of arrays that have been accessed, which is crucial to reduce data transfers between host and device when using directive-based GPU programming models.**

***Index Terms*—Analysis tool, Linear-based techniques, Compiler-based tool, Array Region Analysis.**

I. Introduction

Arrays are the most indispensable and essential data structures in high performance computing and scientific applications. Data, which is stored in arrays, is heavily processed and accessed through any program. This fact indicates that further and deep analysis of the portions of arrays, which are being accessed, is crucial for developers, users and scientists in many aspects.

Arrays, in contrast with scalar variables, need thorough analysis from the user in order to decide whether they can be parallelized or not. Some multi-threaded programming APIs such as OpenMP, which is a de facto standard for shared memory programming, provide a productive directive-based programming approach for generating parallel versions of programs from the sequential ones. However, it is fundamental for users, with different levels of background, to have an interactive and powerful array analysis tool, which can provide them with actual information about the regions of interprocedural arrays that have been accessed, the number of references per access mode, and a comprehensive report about the attributes of these arrays.

*This work was performed while this author was at University of Houston

We propose an interprocedural analysis technique to summarize array accesses at both loop-level and statement level relying on our research compiler OpenUH [1]. We have extended the array region analysis module inside OpenUH to extract the bounds information for the array regions that have been accessed in a triplet notation format $[LB : UB : Stride]$. We have also coupled the aforementioned module with OpenUH's IR, which is called WHIRL (Winning Hierarchical Intermediate Representation Language) [2]. We use this tree-based IR to extract the features of the processed arrays such as access density, which is defined as the ratio of number of references per access mode to the memory bytes allocated to that array. Access mode can be one of USE, DEF, FORMAL or PASSED. A statement S is a definition of v (S defines v) iff S is an assignment statement with left-hand side v. S is a use of v iff during execution of S, right-hand side v is read. The term FORMAL parameter in our case refers to the array as found in the function definition (parameter), while PASSED refers to the actual value passed (argument). We integrate this combination to give the user the capabilty to explore and figure out how his or her application can efficiently be optimized. Finally, we present an extension to our Dragon tool, an analysis tool built on top of this compiler, to represent the interprocedural array region analysis information in addition to array features statically during compilation.

In this way, the extracted information can help the user to easily define candidates for auto-parallelization [3]. Moreover, array region analysis is essential in compile time optimizations for cache behavior in hierarchical memory machines. It is also needed for array privatization and generation of communications in distributed memory machines. Side effects of procedure calls can partially be handled by showing how the array parameters are being accessed. This necessity becomes critical when these procedures are invoked inside loops. Our tool gives an approach that can guide the user to effectively apply code transformations to maximize data locality by restructuring arrays, which have portions that are not being accessed through the whole program. Our tool can also serve as a primary step towards efficient data transfer between host and device. Different directive-based GPU programming models such as PGI accelerator directives [4] and HMPP [5] support subarray offloading from CPU host to attached kernels. Our tool enables users to find the portions of array that have been accessed and the access density for these portions. Knowing that would be significant to better optimize data transfers.

The main motivation behind our research lies in observing the viability of visualizing and displaying interprocedural array region accesses per access mode as well as per dimension, besides various array information that are relevant to memory and cache utilization. we also motivate the need for such an array analysis tool driven. We argue that this work is beneficial in many aspects. We categorize these aspects into three sections, based on the functionalities and features provided.

1) Arrays Defined Inefficiently: We provide this information to identify and fix inefficiencies in how arrays are defined and used in HPC applications, where analyzing arrays is principal in order to achieve good optimization. We deliver this task by

- leveraging array region analysis module in our OpenUH compiler
- Present array access patterns to user via Dragon tool
- Identify transformations based on Dragon feedback to improve locality and reduce cache misses

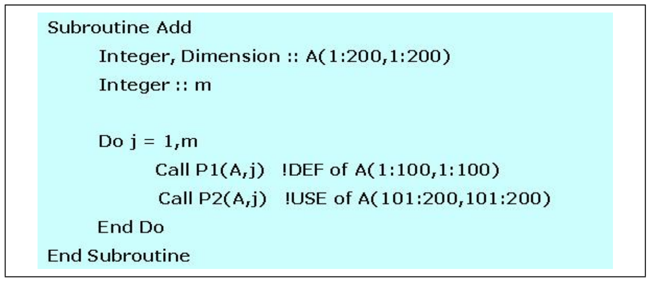
2) Reduce Data Movement: Various directive-based GPU programming models, such as PGI accelerator and HMPP, support subarrays offloading from CPU host to kernels. We assist the user to discover what should be ported in a user friendly manner through a visualized and robust tool. We support this functionality by

- Identifying costly data transfers based on array region analysis combined with parsing the HL-WHIRL
- Allowing the user to optimize communication for host-to-GPU or in PGAS context

3) Auto-parallelization: Identifying auto-parallelization opportunities adeptly is one of the main goals for modern compilers as well as for scientists and application developers. We provide an approach to detect and exploit parallelism in Fortran $77/90$, C, and C++ programs. We achieve this task through

- Compiler inter-procedural analysis of side effects
- Visual feedback on procedures that can be executed in parallel within Dragon

We demonstrate an example of interprocedural array access analysis in Figure 1 to illustrate one of the benefits that can be obtained by our tool. Once procedure $P1$ is invoked, the region of array A represented by the triplet notation format $(1 : 100 : 1, 1 : 100 : 1)$ will be defined. Similarly, on invocation of procedure $P2$, the region of array A represented by the triplet notation format $(101 : 200 : 1, 101 : 200 : 1)$ will be used. Our tool can show these regions with the associated access mode, in which the specified array region has been approached. This implies that both procedures can concurrently and safely be parallelized. Moreover, when a GPU programming model is applied, the information given by our tool guides the user to offload the portions of array A that has been accessed instead of porting the whole array in each procedure.



```
Subroutine Add
    Integer, Dimension :: A(1:200,1:200)
    Integer :: m

    Do j = 1,m
        Call P1(A,j)  !DEF of A(1:100,1:100)
        Call P2(A,j)  !USE of A(101:200,101:200)
    End Do
End Subroutine
```

Fig. 1: Example code for interprocedural access analysis

The remaining sections are organized as follows. Section 2 describes the related work that have been done in the past. In Section 3, we talk about array analysis techniques. Section 4 gives an overview of OpenUH and our extensions to its modules. We focus on our implementation, demonstrate our tool and its evaluation, and how we have accomplished our task in Section 5. Last but not least, we present our conclusion, the obstacles that we have faced and our future goals in the last two sections of this paper.

II. Related Work

Array region analysis is imperative in order to get a successful interprocedural data flow analysis. Moreover, performing interprodedural array access analysis efficiently has been a complex target to achieve for many years. However, visualizing these accesses, associated with the other aforementioned attributes of global, actual and formal parameters arrays in a good manner to the user, has been a harder task to fulfill. In the following, we cover the related work that has been done in this area and show the limitations that have been overcome.

Previous studies have focused on array analysis to exploit it in several manners. Array regrouping was targeted in [6]. They present an interprocedural analysis technique to summarize the access pattern at a loop level and then groups arrays with similar vectors. Their tool is effective for data layout transformations.

In the context of cache and memory utilization, [7] proposes an algorithm to detect code regions suitable for hardware and software optimizations to improve data cache locality. [8] proposes a method that combines access density with access pattern for a better data allocation in the on-chip memory to reduce external memory access time. Another paper [9] presents an interactive visualization tool that uses a three-dimensional plot to show miss rate changes across programs and evaluate compiler transformations. [10] proposes a blocked layout that improves locality in a way that exceeds the usual row, column layouts. All of the previous array analysis research suggested methods to reduce the memory access latency by increasing data locality, but they did not handle any kind of visualization to these arrays.

Modern microprocessors provide hardware performance monitors (HPMs) to help programmers understand the low-level behavior of their applications. Several library packages provide access to hardware performance counter information, including the HPM toolkit [11], PAPI [12], and OProfile [13]. These libraries provide facilities to instrument programs, record hardware counter data, and analyze the results. However, hardware counters do not reserve any information about arrays because they operate at the low-level where arrays lose their structures.

In terms of reducing data transfers, PGI accelerator compiler [14] applies array region analysis to reduce memory transfers. Par4All compiler [15] tackles data transfer management between host and accelerator. It uses convex array region analysis to characterize the data used and defined by each kernel. In comparison, our tool depicts what the user needs in order to guide him or her to perform code transformations and directives insertion for parallelization or GPU porting. Regarding Auto-parallelization, CAPO [16] depends on inter-procedural data dependence info to automate the insertion of compiler directives to facilitate parallelism on SMP machines.

Dragon [17] is a tool that was developed on top of Open64 compiler. It enables users to visualize diffent structures such as call graph, control flow graph and data dependency graph. It was later expanded to support interprocedural array region analysis [18], [19]. That version concentrated on array accesses without providing any information about the attributes of the accessed arrays or the frequency of references based on the access mode used. It did not also target memory and cache optimization, or subarray offloading for directive-based GPU programming models. Furthermore, some bounds information were missing due to linear equations projections, such as negative bounds and strides. Array accesses in loops were normalized, which prevents showing the exact stride values. MySQL database was a restriction in order to use Dragon. We have overcome these issues by implementing a from-scrtach algorithm, which combines both the array region module inside OpenUH and its intermediate representation. Our implementation is thoroughly discussed later in this paper. We tackle GPU offloading and cache optimization by experimenting our implementation with some HPC applications. Another main drawback was that the GUI was limited to a single platform in contrast with our GUI that has support for multiple platforms including Linux and Windows.

III. Array Analysis Techniques

Array analysis methods are categorized based on their access patterns, efficiency and accuracy. Classic methods were the first to tackle this issue. This method identifies arrays with references of types DEF and USE. It is the most efficient method of array analysis in terms of storage since it just uses two bits to represent array summaries. However it is less precise since it represents the array as a whole and not the portions of array elements.

Array region analysis is a technique to summarize the sub-region of an array that is affected by a statement or a sequence of statements or loops. The different methods for analyzing array access patterns are based mainly on three approaches: reference-list-based, triplet-notation-based, and linear constraint-based. As depicted in Figure 2, these methods differ in terms of efficiency and accuracy.

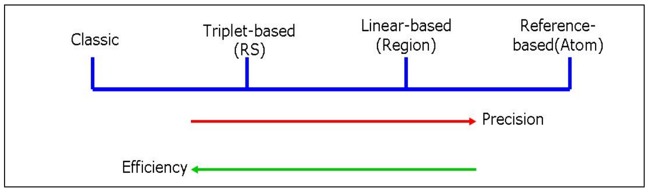


Fig. 2: Array Analysis Techniques

Reference-list-based methods maintain information about references of all the elements of the array and store them as a list. These methods do not provide any kind of summarization. Although this method provides a high degree of accuracy, it reserves a significant storage space. Linearization [20] and Atom Images [21] are two methods that follow this approach. Triplet-notation-based techniques represent array elements references using the bounds information: lower bound, upper bound and stride. This representation is quite simple in contrast with linear constraint-based methods since complex arithmetic is not involved in calculating regions. Summary representation is also faster. Regular Sections [22] is an example of these techniques.

On the other hand, linear-constraint-based methods group array elements into a region using linear constraints determined by the subscripts of arrays. These methods summarize array accesses in a shape of convex regions. We have developed the Regions method, which has been employed in OpenUH, to get the bounds information. This method was originally developed by Triolet [23] and later expanded by Creusillet [24] to support mapping formal to actual parameters. It expresses the set of array accesses as a convex region in a geometrical space. It is more accurate than triplet-notation-based approach for handling non-rectangular shapes. It has two main drawbacks. One is that Fourier-Motzkin linear system solver, which has worst case exponential time, is needed to compare Regions. The second is that the union of regions is approximated since in some cases, it does not form a convex hull. Using this method, We have grouped array accesses into a region based on one of DEF, USE, FORMAL, or PASSED access modes. Each region is determined by simplifying linear equations obtained from the bounds information of the array elements. This method associated with the WHIRL Node were extended together at the HL-WHIRL since array references must be explicit in order to get the interprocedural array bounds and attributes. Our implementation will be discussed in the next sections.

IV. OpenUH and its Extension

OpenUH is an open source compiler developed by the High Preformance Computing Tools (HPCTools) group at the Department of Computer Science, University of Houston.

It is a branch of Open64 compiler infrastructure that was originally developed by Silicon Graphics Inc (SGI) from SGI's MIPSPro compiler, and also derives from work done by Intel Corp, in conjunction with the Chinese Academy of Sciences [25]. Written mostly in C++ and C, OpenUH accepts Fortran, C/C++, cuda, as well as the shared memory API OpenMP and Coarray Fortran (CAF), which is a Partitioned Global Address Space (PGAS) programming model. OpenUH targets a variety of platforms including X86-64, IA-64, and IA-32. OpenUH is a well-written compiler performing a wide range of analysis at different phases and modules. It accommodates a fine environment to perform research on parallel programming modules and their implementation, static and dynamic analysis of parallel application, and compiler integration with external tools and APIs. These features allow OpenUH to be efficiently exploited by tools such as Dragon.

A. *OpenUH Modules*

OpenUH is a robust compiler that is based on a well-founded infrastructure. It can serve as a suitable environment for researchers since it consists of different modules that can be extended in a flexible fashion to implement a wide range of optimizations. Figure 3 demonstrates these modules besides the other components in OpenUH infrastructure.

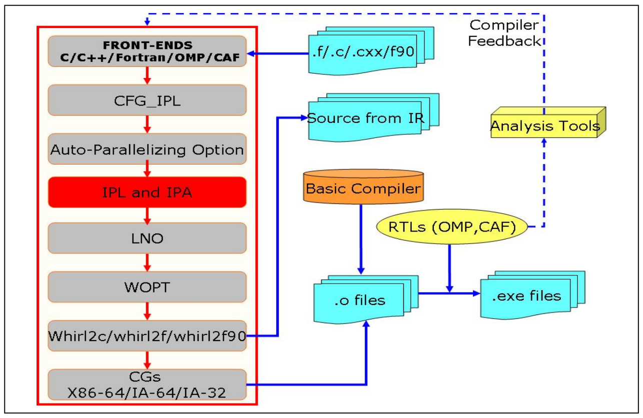


Fig. 3: OpenUH Infrastructure

OpenUH front ends (FE) are based on GNU technology and the OpenMP pragmas are parsed to GNU High Level IR; the Fortran directives are initially converted to Cray IR. These front ends parse C/C++/Fortran programs together with OpenMP and Coarray Fortran programming models and translate them into VHL WHIRL. The C/C++ Automatic parallelization module can be invoked via a special MIPSpro Auto-Parallelizing Option (APO) to discover and exploit parallelism in Fortran, C, and C++ programs. Some restrictions are involved in Auto-parallelization. As an example, function calls inside loops can not be handled by this module. Our tool can assist as a continuation and broadening to this module by giving the user a guidance to overcome this issue. CFG_IPL, which can be turned on or off via compiler flags, was previously added at the high levels of WHIRL in the first version of Dragon [17] to export control flow analysis results. Interprocedural analysis consists of two phases.

- An information gathering phase (IPL)
- The main optimization phase (IPA)

Once interprocedural analysis is optionally invoked via the (IPA) flag, IPL (the local interprocedural analysis part) first gathers data flow analysis and procedure summary information from each compilation unit, and the information is summarized for each procedure. Then, the main IPA module gathers all the IPL summary files to perform interprocedural analysis, transformations, and optimizations in order to be later passed to the following modules. The call graph is generated at this level, where each node in this graph represents a procedure and the caller-callee relationships are expressed by the edges. This call graph should be traversed to extract the necessary array analysis information needed by our tool. Various optimizations are done at this phase, but we mainly concentrate on deriving Array Analysis and extracting its information beside visualizing the call graph to show procedures mapping.

Array region analysis gathers and summarizes array accesses in a shape of convex regions. It mainly supports the transformations done in latter phases of optimizations, such as data dependencies analysis that happens in the Loop Nest Optimizer (LNO) phase. The remaining modules of the back end are invoked next; The compiler starts with the Loop Nest Optimizer (LNO) where several code transformations and optimizations are occured, depending on the analysis gathered at the IPA phase.

The global scalar optimizer, called the WHIRL optimizer WOPT, comes next in the back end module chain. WOPT operates at the mid WHIRL level and performs aggressive data flow analysis and optimizations based on the SSA form. The Code Generator (CG) creates assembly codes, which are finally transformed to binaries by AS (assembler).

OpenUH can be treated as a source to source compiler because of the functionalities that it supports. Very high and high level WHIRL can be translated back to C and Fortran source codes via WHIRL2c, WHIRL2f and WHIRL2f90 tools. However, this could incur minor loss of semantics.

B. *WHIRL Intermediate Representation*

WHIRL is the intermediate language (IR) for OpenUH, which consists of five levels. Compilation process can be defined as a transition from the high level languages constructs to the low level machine instructions. The structure of WHIRL enables OpenUH to support multiple languages and multiple processor targets. This structure provides plentiful advantages. In the compilation process, some optimization passes like constant propagation, dead code elimination, and various liveness problems, have to be re-applied at different times and in different components of the compiler. With WHIRL, a single implementation of an optimization pass is sufficient. Communication between the compilation phases is also easier, since they work under the same medium. These favored circumstances enabled us to employ and exploit the WHIRL tree, where each node in this tree is represented by the WHIRL Node (WN) data structure in OpenUH, and merge it with the array region analysis module inside IPA in order to explore the array information needed.

Our elaboration is essentially centralized at the high levels of WHIRL, where the IPA phase operates, and optimizations are done with respect to programming language constructs. These high levels have been affected and expanded by the implementation of our tool, since the form of array subscripting is preserved via ARRAY operator. We exploit these high levels to traverse the WHIRL tree by linking the WHIRL nodes to the source code procedures. These procedures form the nodes of the call graph, which is generated in the IPA phase. We take advantage of two main things. The first is that arrays keep their structures at the high level, and the second is that WHIRL is the common interface among the different phases of the compiler.

The front-ends generate a WHIRL file that consists of WHIRL instructions and WHIRL symbol tables [2]. We have used the fields ST_IDX and TY_IDX to refer to the symbol tables in order to extract the array information. ARRAY is an N-ary expression operator, since it is an internal node in an expression tree and has N operations. ARRAY operator exists in VH an H levels of WHIRL. Table I describes the key attributes of the WHIRL node that we have used in our tool with explanation about each of them.

TABLE I: The components of WHIRL Node used in our tool

Field	Description
prev	previous pointer
next	next pointer
linenum	source position information
offset	offset for loads, stores, LDA, IDNAME.
elem_size	element size for array
operator	WHIRL operator
res	result type
kid_count	number of kids for n-ary operators.
num_dim	Number of dimensions in array
array_dim	size of array dimension
array_index	index of array
array_base	name of array
const_val	64-bit integer constant.
st_idx	symbol table index.

C. OpenUH Extension and Information Extraction

This work was implemented in two non-trivial and complicated stages. In the first stage, OpenUH IPA optimization phase was extended in a way that merges the array region analysis module with the WHIRL-Tree in order to extract the array information interprocedurally and store them in a plain file. In the second stage, A GUI, which is implemented using the Qt application development framework [26] as part of Dragon tool, was developed. It demonstrates and displays the array information extracted from the compiler in a tabular format. It also has the ability to locate these arrays in the source code in order to perform any necessary code transformations or parallel directives insertion based on the informative visual feedback obtained.

The full picture of our integrated system can be described in Figure 4. ARA module indicates the Array Region Analysis module. This module consists of many data-structures that are constructed in a hierarchical format. Each of them can be considered as a group of the lower data-structure objects. We had to figure out all the references and relationships among these data-structures in order to extract the bounds information. A bound is mapped to one of these four types (CONST, IVAR, LINDEX, and SUBSCR) [18]. Bounds, that have expressions which cannot be linearized, are marked as MESSY or UNPROJECTED.

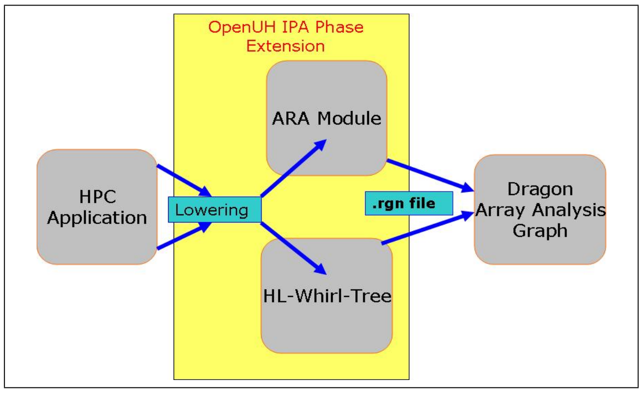


Fig. 4: IPA Extension

Once the application's source code gets lowered to VH WHIRL by the front ends, the compiler will next translate it to H WHIRL IR level where the IPA phase operates. Our extension to the IPA phase can be illustrated in Algorithm 1. We first traverse the call graph cg (pre-order) in which each node ($ipan$) consists of: procedure which the node represents, symbol Table index information and File header information from which the array regions information can be obtained per each source file based on the access mode. We then store the WHIRL tree for each $ipan$ in a WHIRL node wn. An array of regions, with a size $Region\_Size$, exists for each source file, can be obtained from the file header information. We iterate each region to extract the bounds information represented by $[LB, UB, Stride]$. Then, we iterate the WHIRL tree in which each vertex is represented by wn. We check whether the operator of the wn is an OPR_ARRAY. This operator uses (row-major, zero-based) to return an address. As previously described in Table I, the number of dimensions of the array, n, is inferred from kid-count shifted right by 1. An internal field, element size, gives the size of each array element in bytes. We can detect whether the array in Fortran90 is non-contiguous based on the element size value that we show in our tool. If it is negative, it specifies a non-contiguous array [2]. Kids 1 to n give the size of each dimension in contiguous arrays, and the multiplier for each index in non-contiguous arrays. Kids n+1 to 2n give the index expressions for dimensions 0 to n-1 respectively (adjusted so that the array index has a zero lower bound). We obtain the array address by the following formula. If we name Kids 1 to n as $h_1 ... h_n$, and if we name the values of the index expressions $y_1 ... y_n$, and if element size is z, then the obtained address should be:

$$base + z \sum_{i=1}^{n} \left(y_i \prod_{j=i+1}^{n} h_j \right)$$

After we obtain these values mentioned in Table I, we determine the access density for each region per access mode by dividing the number of accesses to the storage bytes allocated for each array. For variable length arrays, the size of entire array will be displayed as zero. We output these information to a comma separated plain file $.rgn$, where each row maintains information about each region per access mode.

This $.rgn$ file will later be processed by our array analysis graph which is part of the Dragon tool. This tool will thoroughly be explained in the next section.

```
Input: $ipan$,$wn$
Output: $.rgn$ file
while !$cg.empty()$ do
    $ipan \leftarrow cg.current()$ ;
    $wn \leftarrow ipan.Whirl\_Tree$ ;
    $Region\_Size \leftarrow FH.array\_region\_size$ ;
    for $j \leftarrow 1$ to $Region\_Size$ do
        Get bounds info for Array Regions ;
        while !$Whirl\_Tree.end()$ do
            Get Array info from $wn$ ;
        end
    end
end
```

Algorithm 1: Array Analysis Extraction

V. Dragon and Its Array Analysis Graph

Dragon is an updated OpenUH compiler-based software tool that has been rebuilt from scratch to help application developers or code owners to understand more about their applications. The current input languages for the tool are Fortran, C/C++ and OpenMP. It is also an interactive system with a powerful GUI providing a range of information about the structure of source program in a graphical browseable form, at the level of detail desired. The tool also helps the user to efficiently navigate through these structures.

Dragon is developed using the Qt framework and uses Graphviz library to represent code structure information in a scalable graphical form. The tool also provides a rich set of visualization features. As depicted in Figure 5, the current features of Dragon include - static/dynamic call graphs with feedback information, control flow graphs for each procedure and array analysis graph. The GUI features include: support for multiple platforms (Linux, Windows), syntax highlighting, source code analysis, scalable layout of graphical items and real-time search functionality.

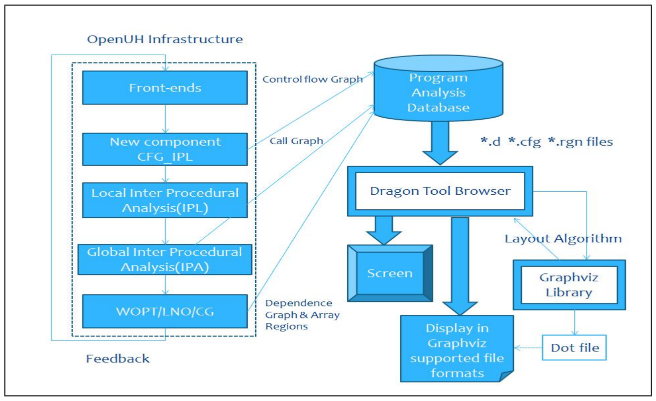
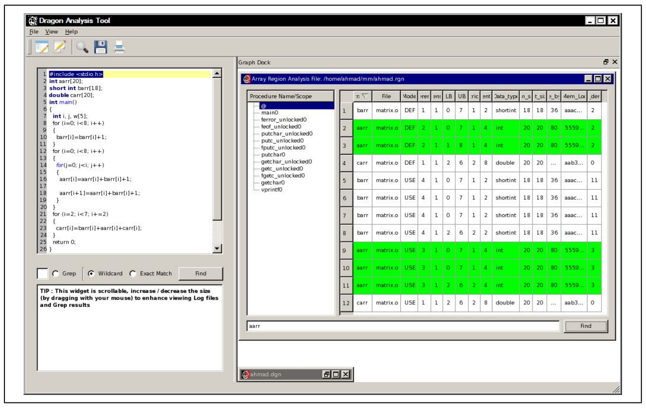


Fig. 5: Dragon Structure

A. Array Analysis Graph

The Array Analysis GUI has been implemented as an extension to the Dragon tool [17]. This GUI provides features such as: syntax highlighting as well as find /UNIX-like grep feature. Moreover, the developer has the ability to distinctly visualize the source code in order to refer to any particular global array or an array parameter of a procedure.



```
#include <stdio.h>
int aarr[20];
short int barr[18];
double carr[20];
int main()
{
  int i, j, w[5];
  for (i=0; i<8; i++)
  {
    barr[i]=barr[i]+1;
  }
  for (i=0; i<8; i++)
  {
    for(j=0; j<i; j++)
    {
      aarr[i]=aarr[i]+barr[i]+1;

      aarr[i+1]=aarr[i]+barr[i]+1;
    }
  }
  for (i=2; i<7; i+=2)
  {
    carr[i]=barr[i]+aarr[i]+carr[i];
  }
  return 0;
}
```

TIP : This widget is scrollable, increase / decrease the size (by dragging with your mouse) to enhance viewing Log files and Grep results

Procedure Name/Scope
- @
- main0
- ferror_unlocked0
- feof_unlocked0
- putchar_unlocked0
- putc_unlocked0
- fputc_unlocked0
- putchar0
- getchar_unlocked0
- getc_unlocked0
- fgetc_unlocked0
- getchar0
- vprintf0

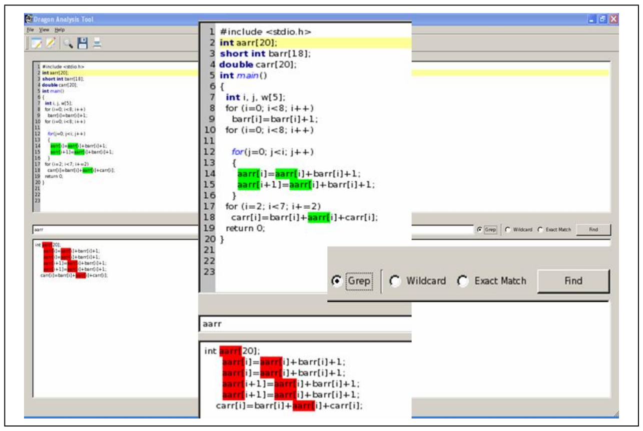
	n	File	Mode	erer	ens	LB	UB	ric	ent	Data_type	n_s	t_si	e_by	Mem_Loc	der
1	barr	matrix.o	DEF	1	1	0	7	1	2	shortint	18	18	36	aaac...	2
2	aarr	matrix.o	DEF	2	1	0	7	1	4	int	20	20	80	5559...	2
3	aarr	matrix.o	DEF	2	1	1	8	1	4	int	20	20	80	5559...	2
4	carr	matrix.o	DEF	1	1	2	6	2	8	double	20	20	...	aab3...	0
5	barr	matrix.o	USE	4	1	0	7	1	2	shortint	18	18	36	aaac...	11
6	barr	matrix.o	USE	4	1	0	7	1	2	shortint	18	18	36	aaac...	11
7	barr	matrix.o	USE	4	1	0	7	1	2	shortint	18	18	36	aaac...	11
8	barr	matrix.o	USE	4	1	2	6	2	2	shortint	18	18	36	aaac...	11
9	aarr	matrix.o	USE	3	1	0	7	1	4	int	20	20	80	5559...	3
10	aarr	matrix.o	USE	3	1	0	7	1	4	int	20	20	80	5559...	3
11	aarr	matrix.o	USE	3	1	2	6	2	4	int	20	20	80	5559...	3
12	carr	matrix.o	USE	1	1	2	6	2	8	double	20	20	...	aab3...	0

aarr

Fig. 6: Array Analysis Graph

Array Regions analysis information is displayed in a tabular structure demonstrated in Figure 6, which is divided into Figure 7 and Figure 9 to improve its readability. For each program, a procedure list is generated and displayed in the most-left column of the table. The @ symbol at the top of this column indicates global arrays. By clicking on any procedure name, the corresponding array analysis information will be displayed for all arrays that have been passed as formal or actual parameters. In the same manner, clicking on the @ symbol will display the analysis information for the global arrays accessed in the program. A find functionality feature has been added to the graph in order to search for any particular array.

The array analysis graph consists of many columns as shown in Table II and Table III. These columns are grouped in three categories based on the functionalties provided.



```
#include <stdio.h>
int aarr[20];
short int barr[18];
double carr[20];
int main()
{
  int i, j, w[5];
  for (i=0; i<8; i++)
    barr[i]=barr[i]+1;
  for (i=0; i<8; i++)

    for(j=0; j<i; j++)
    {
      aarr[i]=aarr[i]+barr[i]+1;
      aarr[i+1]=aarr[i]+barr[i]+1;
    }
  for (i=2; i<7; i+=2)
    carr[i]=barr[i]+aarr[i]+carr[i];
  return 0;
}
```

```
aarr
```

```
int aarr[20];
    aarr[i]=aarr[i]+barr[i]+1;
    aarr[i]=aarr[i]+barr[i]+1;
    aarr[i+1]=aarr[i]+barr[i]+1;
    aarr[i+1]=aarr[i]+barr[i]+1;
  carr[i]=barr[i]+aarr[i]+carr[i];
```

Fig. 7: A snapshot of source code browsing

The following columns help the user to identify the interprocedural arrays that have been accessed and locate them in the source code.

- Array: Array accessed in the selected proc/scope
- File: the source file where this array has been accessed.
- Mode: indicates one of the four modes of accesses: USE(array variable usage), DEF (assignment of values to array elements), FORMAL (using array variable as a formal parameter) and PASSED(an array variable passed as an actual parameter in a procedure call).
- References: The number of region accesses for the selected array based on the access mode
- Dimensions: The number of dimensions of the selected array

The next three columns define the portions of arrays that have been accessed.

- LB: The starting index of the array region that has been accessed
- UB: The ending index of the array region that has been accessed
- Stride: The step in which the array region elements have been accessed. The previous three columns indicate the portion of array that has been accessed

The remaining columns in our tabular graph deliver a comprehensive summary about the aforementioned arrays.

- Element_Size: The size of each array element.
- Data_Type: The Data type of the selected array
- Dim_Size: The size of each dimension of the array
- Tot_Size: The total size of the selected array
- Size_bytes: The allocated bytes for the array in memory
- Mem_Loc: The memory address of this array in hexadecimal. It helps the user to find arrays pointing to the same memory location.
- Acc_density: The access density indicates the ratio of number of references per access mode to the memory bytes allocated to any particular array. It helps the user to identify the hotspot arrays in the program in terms of memory allocation and frequency of accesses.

$$AD(Array, Mode) = \frac{References}{Size\_bytes}$$

For the sake of simplicity, we give a simple example for the source code shown in Figure 10. Once the array analysis graph is loaded, we click on the @ symbol to display all global arrays analysis information. We search for Array $aarr$ by typing its name and clicking on the find button at the bottom of the graph. All accesses to Array $aarr$ will be highlighted in green as demonstrated in Figure 6. As can be seen in Figure 7, the user can grep any array to display all the statements in which the array has been accessed. Let us go back to the source code displayed in Figure 10, array $aarr$ has been defined twice and used three times. Figure 9 displays the bounds of the regions of array $aarr$ and the stride. It also shows the source file in which it has been accessed. The element size is four with an integer data type. Since it is one dimensional array, the dimension size is equivalent to the total size (20). The bytes allocated for $aarr$ are 80. The memory location in hexadecimal is 55599870. The access density percentage is two for the DEF access mode and three for the USE mode. Figure 8 illustrates the concept of the access density term that we are using in our tool.

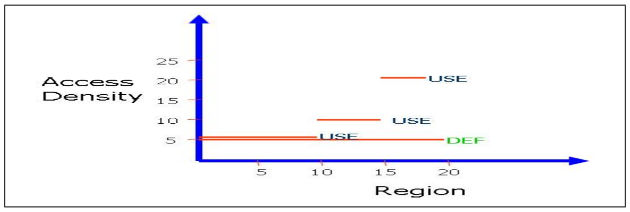
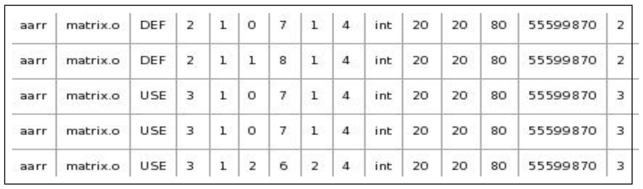


Fig. 8: Access Density Diagram

In this example, our tool shows that the user can redefine array $aarr$ to be (int aarr[8]) instead of (int aarr[20]) since the remaining elements have not been used anywhere in the program. At the same time, once the program is compiled in a compiler that supports directive-based GPU programming model such as PGI accelerator compiler, ($\#pragma\ acc\ region\ for\ copyin(aarr[2:7])$) can be inserted right before the last for loop in the source code. This can reduce the data trasferred from host to GPU and avoid parsing large source code applications to determine what array bounds to use in the different copy clauses of region directives.

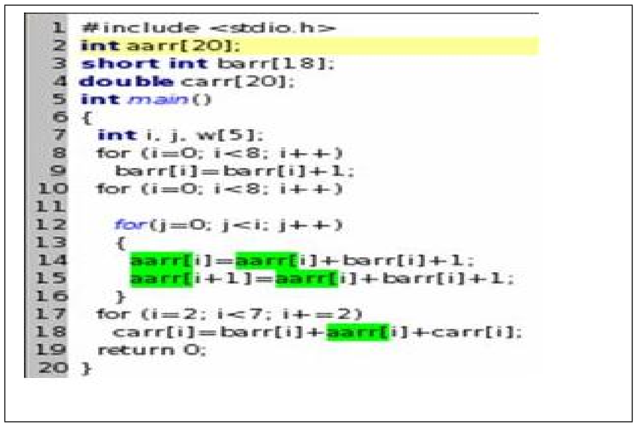


aarr	matrix.o	DEF	2	1	0	7	1	4	int	20	20	80	55599870	2
aarr	matrix.o	DEF	2	1	1	8	1	4	int	20	20	80	55599870	2
aarr	matrix.o	USE	3	1	0	7	1	4	int	20	20	80	55599870	3
aarr	matrix.o	USE	3	1	0	7	1	4	int	20	20	80	55599870	3
aarr	matrix.o	USE	3	1	2	6	2	4	int	20	20	80	55599870	3

Fig. 9: An example of a particular array accesses

B. Case Study

Our tool expedites the task of parallelization, cache utilization and data transfers using OpenMP or/and directive-



```
#include <stdio.h>
int aarr[20];
short int barr[18];
double carr[20];
int main()
{
  int i, j, w[5];
  for (i=0; i<8; i++)
    barr[i]=barr[i]+1;
  for (i=0; i<8; i++)

    for(j=0; j<i; j++)
    {
      aarr[i]=aarr[i]+barr[i]+1;
      aarr[i+1]=aarr[i]+barr[i]+1;
    }
  for (i=2; i<7; i+=2)
    carr[i]=barr[i]+aarr[i]+carr[i];
  return 0;
}
```

Fig. 10: A source code example

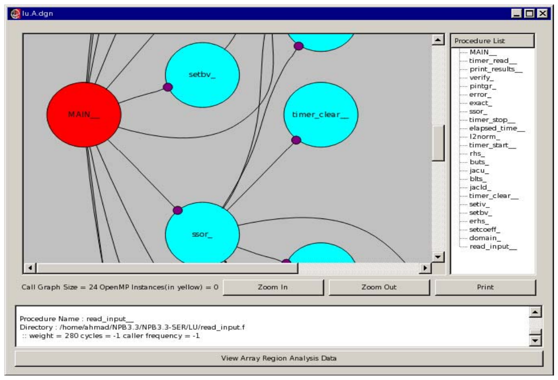
based GPU programming model by guiding the programmer to determine which portions of code should be parallelized in regards to interprocedural arrays and variables. However, the user interaction is still needed to decide whether it is possible to do any code transformations or directive insertion into the code. A basic strategy for optimizing a code using our analysis tool is as follows:

1) Modify the Makefile of the application to use the OpenUH compiler with interprocedural array analysis (-IPA:array_section:array_summary), optimization options selected as well as the (-dragon) flag.
2) Compile the application. A bunch of files will be generated that includes .dgn, .cfg and .rgn files.
3) Invoke our Dragon tool and load the .dgn project.
4) Invoke the array analysis graph by clicking on the button, "view array region analysis data" and locate the portions of interest that can be optimized based on our tool's feedback.

As clarified before, OpenUH uses (row major, zero indexing) for all languages because of the structure of its ARRAY operator. To surpass this obstacle, we modify the bounds, which are obtained from the compiler side, in Dragon. We perform this modification to make our tool aware of the application's source code language, and to fulfill our goal of showing the actual bounds of arrays that have been interprocedurally accessed.

We have developed the call graph in Dragon based on our implementation inside the IPA phase of OpenUH. The graph consists of nodes corresponding to the procedures and edges corresponding to the call sites. The call graph structure retrieves the total size of the graph which is useful while traversing. The call graph iterator is also a useful resource to access the various call graph nodes. Using the node's methods, the whole WHIRL tree can be loaded and also the corresponding symbol tables in the memory. After retrieving the node information, another iterator iterates within each node to traverse the callsite information.

The NAS Parallel Benchmarks (NPB 3.3) is a suite of eight codes designed to test the performance of parallel supercomputers. We use the serial version of LU (Lower-Upper Gauss-Seidel solver) to illustrate the previous steps. Figure 11 demonstrates the call graph generated for the LU benchmark once the user loads the .dgn project. Each node represents a procedure associated with its name. The user can display the source code of any procedure by clicking on the corresponding node. As can be seen at the bottom of Figure 11, the LU benchmark has 24 procedures. The user can zoom-in and zoom-out the garph as needed. We link this graph to the array analysis graph by clicking on the button "View Array Region Analysis Data".

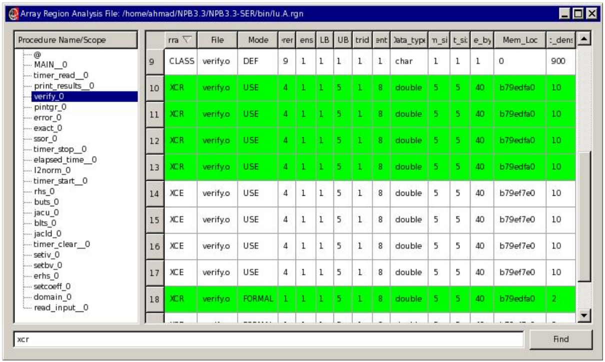


Call Graph Size = 24 OpenMP Instances(in yellow) = 0

Procedure Name : read_input__
Directory : /home/ahmad/NPB3.3/NPB3.3-SER/LU/read_input.f
:: weight = 280 cycles = -1 caller frequency = -1

Fig. 11: Dragon Call Graph

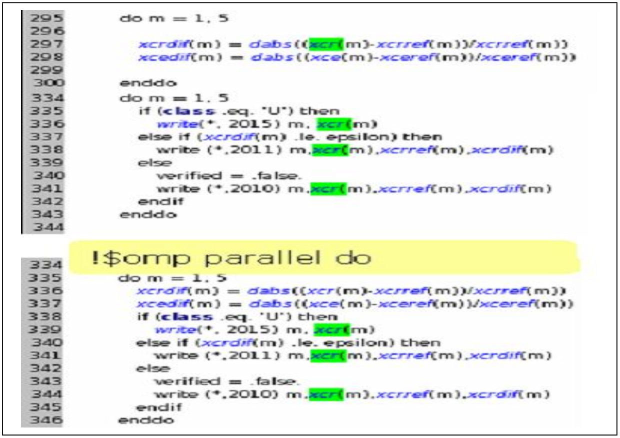
We have defind two interesting cases out of many in the LU benchmark that should illustrate our tool's features and functionalities.



	rra	File	Mode	rer	ens	LB	UB	trid	ent	Data_type	m_si	t_si	e_by	Mem_Loc	_dens
9	CLASS	verify.o	DEF	9	1	1	1	1	1	char	1	1	1	0	900
10	XCR	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79edfa0	10
11	XCR	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79edfa0	10
12	XCR	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79edfa0	10
13	XCR	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79edfa0	10
14	XCE	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79ef7e0	10
15	XCE	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79ef7e0	10
16	XCE	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79ef7e0	10
17	XCE	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79ef7e0	10
18	XCR	verify.o	FORMAL	1	1	1	5	1	8	double	5	5	40	b79edfa0	2

Fig. 12: One Dimensional Array Analysis Graph for NAS LU benchmark

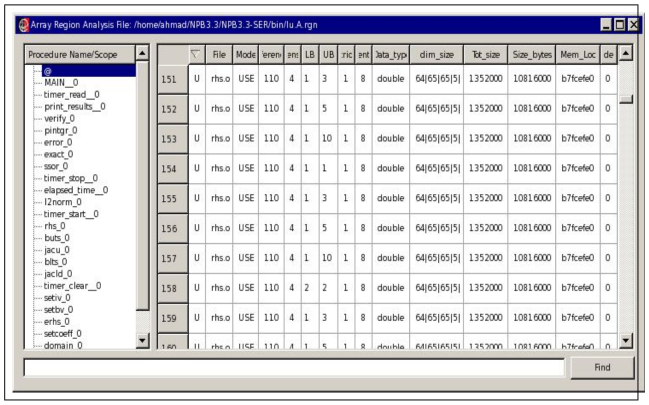
The first one is described in Figure 12. We create Table II to illustrate the accesses of array XCR, which are highlighted in green in Figure 12. It shows that XCR is a one dimensional double array that has been passed as a formal parameter in the verify procedure with bounds 1:5. It also shows the size in bytes that was allocated to it, which is 40. The access density for its use access mode is 10, which is significantly high and



```
        do m = 1, 5

          xcrdif(m) = dabs((xcr(m)-xcrref(m))/xcrref(m))
          xcedif(m) = dabs((xce(m)-xceref(m))/xceref(m))

        enddo
        do m = 1, 5
          if (class .eq. 'U') then
            write(*, 2015) m, xcr(m)
          else if (xcrdif(m) .le. epsilon) then
            write (*,2011) m,xcr(m),xcrref(m),xcrdif(m)
          else
            verified = .false.
            write (*,2010) m,xcr(m),xcrref(m),xcrdif(m)
          endif
        enddo

  !$omp parallel do
        do m = 1, 5
          xcrdif(m) = dabs((xcr(m)-xcrref(m))/xcrref(m))
          xcedif(m) = dabs((xce(m)-xceref(m))/xceref(m))
          if (class .eq. 'U') then
            write(*, 2015) m, xcr(m)
          else if (xcrdif(m) .le. epsilon) then
            write (*,2011) m,xcr(m),xcrref(m),xcrdif(m)
          else
            verified = .false.
            write (*,2010) m,xcr(m),xcrref(m),xcrdif(m)
          endif
        enddo
```

Fig. 13: One Dimensional Array Analysis code snapshot for NAS LU benchmark

makes it a good candidate for optimization. We can see that the region (1:5) of the XCR array has been used four times. By referring to the source code in the verify.f source file depicted in Figure 13 and exploiting our tool's browsing feature, we found that XCR has been used in two separate loops as shown in the upper part of Figure 13. Once in the first one, and three times in the second. Remembering that the same region is being used, and knowing that no dependencies exists, we can merge the two loops as depicted in the low part of the same Figure 13 and have one !$omp parallel do$ inserted right before the merged loop. We could optimize cache utilization and data locality by avoiding the delay resulting from fetching XCR from memory again. We were also able to avoid omp parallel region startup overheads by having one parallel do construct instead of two.



	▽	File	Mode	'eren	ens	LB	UB	ric	ent	Data_typ	dim_size	Tot_size	Size_bytes	Mem_Loc	de
151	U	rhs.o	USE	110	4	1	3	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
152	U	rhs.o	USE	110	4	1	5	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
153	U	rhs.o	USE	110	4	1	10	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
154	U	rhs.o	USE	110	4	1	1	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
155	U	rhs.o	USE	110	4	1	3	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
156	U	rhs.o	USE	110	4	1	5	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
157	U	rhs.o	USE	110	4	1	10	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
158	U	rhs.o	USE	110	4	2	2	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
159	U	rhs.o	USE	110	4	1	3	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0
160	U	rhs.o	USE	110	4	1	5	1	8	double	64|65|65|5|	1352000	10816000	b7fcefe0	0

Fig. 14: Multidimensional Array Analysis Graph for NAS LU benchmark

Another useful case is demonstrated in Figure 14. Table III was created to improve the readability of information shown in Figure 14 for array U. Our tool shows that array U is a global four dimensional double array with these dimension sizes $(64|65|65|5)$, and a total byte storage of 10816000, which is about 10MB. It has been used 110 times, which makes it a hotspot in our code. As illustrated in Figure 14, the regions of each dimension that have been accessed in one loop in rhs.f source file are (1:3,1:5,1:10,1:4). The elements in the last dimension were accessed separately. The array address in hexa is (b7fcefe0). Using the PGI accelerator compiler subarray porting features, the information provided by our tool can magnificently guide the user to insert a !$acc\ region\ copyin(U(1:3,1:5,1:10,1:4))$ instead of !$acc\ region\ copyin(U)$ right before the loop that accesses these portions. This directive means that only these portions of U will be offloaded to GPU. This should considerably reduce data transfers between host and GPU and guarantee a huge speedup. Table IV shows the speedup gained. Consequently, the user can exploit our tool's feedback to insert a data region directive properly with the right array portions without the need to traverse the whole source code.

TABLE IV: Experiments taken on a 24 core cluster for Case2

Portion	Time(seconds)
Whole Array	152
Specified portions	149

This benchmark comprises many other candidates for optimization that can be explored by our tool. However, we tried, through the previous examples, to exhibit the influence of our tool on helping users to achieve optimization.

VI. Future Work

Our tool is a distinct GUI that enables users to have a deep insight into the interprocedural nature of hpc applications in regards to arrays. We target exploring how our tool can be exploited by applications written with various parallel programming models. For example, support for the Partitioned Global Address Space (PGAS) model has been incorporated into the OpenUH compiler via *coarrays*, a language extension to arrays defined in the latest Fortran standard [27]. PGAS assumes a global memory address space that is logically partitioned and is locally available to each processor. Using the coarray abstraction, a programmer can easily express remote data accesses based on a one-sided communication model. We plan to extend our array analysis tool to support the analysis and visualization of remote array accesses. In the context of visualization, we try to enrich our tool's features by supporting high preformance 3D visualization via Qt OpenGL module [28]. This can be used to render 3D graphs in a way that refines the readability of information displayed via our tool. We also work on enhancing our tool and OpenUH to provide dynamic array region information, in order to better understand the actual array access patterns on an OpenMP thread basis. This feature may improve data privitization in OpenMP codes. We will record the information necessary to represent an accessed region including the thread which has accessed it. Our OpenUH-based tool provides a flexible environment that can be enlarged to help users in many aspects.

TABLE II: Tabular Format Information for Case1

Array	File	Mode	Ref	Dim	LB	UB	S	Element	type	dim_size	Tot_size	Size_bytes	Mem_Loc	Acc_density
XCR	verify.o	USE	4	1	1	5	1	8	double	5	5	40	b79edfa0	10
XCR	verify.o	FORMAL	1	1	1	5	1	8	double	5	5	40	b79edfa0	2

TABLE III: Tabular Format Information for Case2

Array	File	Mode	Ref	Dim	LB	UB	S	Element	type	dim_size	Tot_size	Size_bytes	Mem_Loc	Acc_density
U	rhs.o	USE	110	4	1	3	1	8	double	64|65|65|5	1352000	10816000	b7fcefe0	0
U	rhs.o	USE	110	4	1	5	1	8	double	64|65|65|5	1352000	10816000	b7fcefe0	0
U	rhs.o	USE	110	4	1	10	1	8	double	64|65|65|5	1352000	10816000	b7fcefe0	0
U	rhs.o	USE	110	4	1	1	1	8	double	64|65|65|5	1352000	10816000	b7fcefe0	0

VII. Conclusion

In this work, we unfold an interactive compiler-based visualization tool to find the hotspot portions of interprocedural arrays in HPC applications. We define these hotspots via frequency of accesses per access mode, and memory storage (in bytes) that has been allocated to these arrays. We show that this information can be critical and crucial for a better parallelization, cache and memory utilization. Some scientific applications have loops that do not access the whole arrays. Our tool points to the accessed portions in a triplet format (LB:UB:Stride), which allows users to reduce data transfers by exploiting the sub-array offloading functionality supported by directive-based GPU programming models. Our tool has been tested on many HPC applications and can efficiently be used by programmers, professionals and even beginners to have a better understanding of the overall structure of their programs.

Acknowledgment

The authors would like to thank their colleagues in the HPCTools group for their extensive collaboration to make this work a reality. This work is supported by the National Science Foundation under grants CCF-0917285 and CCF-0833201.

References

[1] (2012) Openuh: A portable and optimizing openmp compiler. [Online]. Available: http://www2.cs.uh.edu/ hpctools/OpenUH/

[2] J. Utke. (2007, Aug.) Open64 compiler whirl intermediate representation. open64A.pdf. [Online]. Available: http://www.mcs.anl.gov/OpenAD/

[3] (1998) Mipspro auto-parallelizing option programmer's guide. [Online]. Available: http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0650/bks/SGI_Developer/books/MPro_AP_PG/sgi_html/

[4] T. P. Group. (2010, Nov.) Pgi accelerator programming model for fortran & c. pgi_accel_prog_model_1.3.pdf. [Online]. Available: http://www.pgroup.com/lit/whitepapers/

[5] C. entreprise. (2010, Nov.) Hmpp directives. HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf. [Online]. Available: http://www.olcf.ornl.gov/wp-content/uploads/2012/02/

[6] X. Shen, Y. Gao, C. Ding, and R. Archambault, "Lightweight reference affinity analysis," in *In Proceedings of the 19th ACM International Conference on Supercomputing*, Boston, MA, USA, Jun. 2005, pp. 131–140.

[7] G. K. Memik and A. K. M. Choudhary, "An integrated approach for improving cache behavior," in *Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 796–801.

[8] H. Chang and W. Sung, "Access-pattern-aware on-chip memory allocation for simd processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst*, vol. 56, pp. 158–163, 2009.

[9] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 56, no. 3, 2007.

[10] E. Athanasaki and N. Koziris, "Improving cache locality with blocked array layouts," in *Proceedings. 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2004, pp. 308–317.

[11] (2008) Hardware performance monitor(hpm) toolkit users guide. [Online]. Available: https://wiki.alcf.anl.gov/images/5/59/HPM_ug.pdf

[12] P. J. Mucci, S. Browne, C. Deane, and G. Ho. (1999, Sep.) Papi: A portable interface to hardware performance counters. dodugc99-papi.pdf. [Online]. Available: http://web.eecs.utk.edu/ mucci/latest/pubs/

[13] W. E. Cohen. (2004) Tuning programs with oprofile. Oprofile.pdf. [Online]. Available: http://people.redhat.com/wcohen/

[14] P. Group. (2008) Pgi compilers, gpus and you! pgi_presentation_sc08.pdf. [Online]. Available: http://www.pgroup.com/lit/presentations/

[15] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static compilation analysis for host-accelerator communication optimization," in *The 24th International Workshop on Languages and Compilers for Parallel Computing*, Fort Collins, Colorado, Sep. 2011.

[16] (2001) Code parallelization with capo – a user manual. [Online]. Available: http://people.nas.nasa.gov/hjin/CAPO/nas-01-008-abstract.html

[17] O. Hernandez, C. Liao, and B. Chapman, "Dragon: A static and dynamic tool for openmp," in *In Workshop on OpenMP Applications and Tools (WOMPAT 2004)*, 2005, pp. 53–66.

[18] R. D. Venkatasubramanyam, "Array access analysis in open64," Master's thesis, University of Houston, Houston, TX, 2004.

[19] O. Hernandez, C. Liao, and B. Chapman, "A tool to display array access patterns in openmp programs," in *PARA'04 workshop on state-of-the-art in scientific computing*, 2005, pp. 490–498.

[20] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, Jul. 1986, pp. 162–175.

[21] Z. Li and P.Yew, "Efficient interprocedural analysis for program parallelization and restructuring," in *Proceedings of the SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, Jul. 1988.

[22] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, 1991.

[23] R. Triolet, F. Irigoin, and P. Feautrier, "Direct parallelization of call statements," in *In Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, ACM*, Jul. 1986, pp. 176–185.

[24] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," in *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, Springer, Verlag, USA, 1995, pp. 46–60.

[25] (2012) Open64 overview. [Online]. Available: http://www.open64.net/

[26] (2008-2012) Qt. [Online]. Available: http://qt.nokia.com/products

[27] J. Reid. (2008) The new features of fortran 2008. N1729.pdf. [Online]. Available: ftp://ftp.nag.co.uk/sc22wg5/N1701-N1750/

[28] (2008-2011) Qtopengl module. [Online]. Available: http://doc.qt.nokia.com/4.7-snapshot/qtopengl.html