

OpenACC Routine Directive Propagation using Interprocedural Analysis

Aniket Shivam¹ and Michael Wolfe²

¹ University of California, Irvine
(Work done at NVIDIA/PGI)

`aniketsh@uci.edu`

² NVIDIA/PGI
`mwolfe@nvidia.com`

Abstract. Accelerator programming today requires the programmer to specify what data to place in device memory, and what code to run on the accelerator device. When programming with OpenACC, directives and clauses are used to tell the compiler what data to copy to and from the device, and what code to compile for and run on the device. In particular, the programmer inserts directives around code regions, typically loops, to identify compute constructs to be compiled for and run on the device. If the compute construct calls a procedure, that procedure also needs to be marked for device compilation, as does any routine called in that procedure, and so on transitively. In addition, the marking needs to include the kind of parallelism that is exploited within the procedure, or within routines called by the procedure. When using separate compilation, the marking where the procedure is defined must be replicated in any file where it is called. This causes much frustration when first porting existing programs to GPU programming using OpenACC. This paper presents an approach to partially automate this process. The approach relies on interprocedural analysis (IPA) to analyze OpenACC regions and procedure definitions, and to propagate the necessary information forward and backward across procedure calls spanning all the linked files, generating the required accelerator code through recompilation at link time. This approach can also perform correctness checks to prevent compilation or runtime errors. This method is implemented in the PGI OpenACC compiler.

Keywords: Automatic Directive Propagation, Interprocedural Analysis (IPA), OpenACC, GPU Computing

1 Introduction

Accelerator programming has been used for at least forty years. Accelerators are typically optimized for certain common program structures, often without the ability to run a whole program or to support a full operating system. They are used when the program can be sped up by offloading the bulk of the computation to the accelerator, leaving the main program on the CPU to handle

input/output and interaction with a user. Programming a typical accelerator-enabled machine requires the programmer to identify when to allocate space in device memory, when to copy data to and from system memory, and when to launch a computation on the device.

While early accelerators came with a set of preprogrammed libraries and procedures, current accelerators allow users to write their own programs. To do so, the user must identify what parts of the program need to be compiled for and run on the accelerator. One approach is to use a language or language extensions, such as CUDA C++ [8, 11] or CUDA Fortran [15] for NVIDIA GPUs. With CUDA C++ or Fortran, the programmer explicitly allocates device memory, explicitly copies data between host and device, and explicitly launches *kernels*, which are specially written procedures that run in parallel on the GPU. OpenCL [13] is a similar programming model supported on a variety of devices, including GPUs.

The OpenACC Application Programming Interface (API) [2, 9, 12] is another parallel programming model that supports offloading parallel computations to accelerators, including NVIDIA and AMD GPUs. OpenACC uses directives in C++, C, and Fortran programs, similar to OpenMP [14], to tell the compiler what data to copy between system and device memory, and to mark the regions of code, typically loops, to run in parallel on the device. OpenACC has proved popular because it doesn't require a wholesale rewrite of the program, though many programs need restructuring to expose enough parallelism to fully exploit a GPU. OpenACC programs have three *levels* of parallelism, *gang*, *worker*, and *vector*. At the most coarse level, *gang* parallelism is used for fully parallel loops that need no synchronization or barriers beyond atomic operations and reductions; for GPU execution, iterations of a *gang* parallel loop can be mapped to CUDA thread blocks or OpenCL work groups. Within a *gang*, *worker* parallelism is used for parallel loops where the iterations may need to coordinate or synchronize; for GPU execution, iterations of a *worker* parallel loop can be mapped to groups of threads with a CUDA thread block (such as a warp), or groups of work items in an OpenCL work group. Within a *worker*, *vector* parallelism is used for SIMD operations or parallelism which is synchronous or mostly synchronous, and which is sensitive to memory strides; for GPU execution, iterations of a *vector* parallel loop are typically mapped to the *threadIdx.x* dimension of a CUDA thread block or dimension 0 of an OpenCL work group, to benefit from contiguous memory references. OpenACC programs can also be recompiled for host or multicore parallel execution, with *gang* and/or *worker* loops compiled for execution across the cores, and *vector* loops compiled for SIMD execution.

As more nontrivial programs were ported to GPUs using OpenACC, it became obvious that there needed to be a way to compile and run whole procedures on the device. Moreover, because of the way OpenACC parallelism on GPUs is commonly implemented, when there are parallel loops in the called procedure, both the caller and callee must know what level of parallelism is used in the called procedure. There were three options available for implementing this. One way would be to compile all procedures for GPU execution, just as all procedures are

compiled for the host CPU. While this may become viable in the future, there are many operations that current GPUs just do not support, such as direct I/O, host memory management, and so on. Also, the compiler wouldn't necessarily know the level of parallelism used in a procedure that appears in another file.

Another way would be to have the compiler determine which procedures are called within a compute construct and compile those for the device as well. Transitively, the compiler would need to find all procedures called from any of those procedures and generate device code, and so on. This is straightforward for C or C++ within a single file, and in fact the PGI C++ compiler already implements this, though it doesn't propagate the level of parallelism in the procedure to the call site. This is a bit more problematic for Fortran, since Fortran doesn't have the concept of *file scope*, but that could be worked around.

A third way, and the way selected in OpenACC (and in the OpenMP 4.0 and 4.5 target directives) because it depends least on compiler technology, is to have the programmer mark the procedures needed on the device, along with the level of parallelism used within those procedures. This solution requires the most work from programmers, and has been the source of much frustration at various GPU Hackathons³ when beginning users start to add OpenACC directives to an existing program. In OpenACC, the programmer must insert a **routine** directive for each procedure that must be compiled for the GPU where that procedure is defined. A procedure so marked is called a *device routine*, since it can be called on the device. If it is called from a different file, the programmer must insert a matching **routine** directive in that file (or in an appropriate header file). Fortran *modules* simplify this somewhat, since the necessary **routine** information is propagated through the normal Fortran module mechanism.

In current implementations, if a procedure call appears in an OpenACC compute construct or routine compiled for the device with no appropriate **routine** directive for that procedure, the compiler can either issue an error message, or assume that device code for that procedure will be provided. The latter case can result in a link-time error, if the procedure was not compiled for the device, and can result in a run-time error, if the level of parallelism assumed at the call site does not match the level of parallelism expected in the called procedure.

The PGI compilers currently provide the most complete and mature implementation of OpenACC. They also support a comprehensive and extensible version of interprocedural analysis (IPA), or whole program optimization. In the PGI implementation, the compiler saves both a brief *program summary* and complete *recompile information* in the generated object file. At link time, a fast interprocedural analyzer is invoked, which reads all the available program summary information. For files and libraries not built with the summary information, such as system libraries, IPA reads the symbol table from the object files or libraries. The summary information and symbol tables allow IPA to build a complete call graph and propagate information both forward and backward through the graph. Files containing procedures that would benefit from IPA optimizations are then recompiled before the linker is invoked.

³ <https://www.olcf.ornl.gov/training-event/2017-gpu-hackathons/>

For this work, we explored using the PGI IPA facility to ease OpenACC programming, by adding or propagating OpenACC `routine` directives throughout an application at link time. This can detect and insert missing `routine` directives at call sites or at routine definitions, or detect errors when existing `routine` directives don't match. The IPA step then recompiles the files which contain either call sites or called procedures where new `routine` directives are inserted.

The remainder of the paper is organized as follows. Section 2 describes the OpenACC `routine` directive and the clauses. Section 3 describes our implementation. Section 4 describes cases where our implementation benefits OpenACC programmers. Section 5 summarizes our work and the benefits for the OpenACC programmers.

2 OpenACC Routine Directive

When a procedure call is encountered in an OpenACC compute region, the *callee* might be defined in the same C or C++ file or the same Fortran module, or might be (explicitly or implicitly) declared with a prototype or Fortran interface block. For each such *callee* procedure, to avoid compile-time and link-time errors, the user needs to tell the compiler to generate both host and accelerator code. This is done with the OpenACC `routine` directive. For C and C++, a `routine` directive should appear immediately before the definition of the function or its prototype. For Fortran, the `routine` directive should appear in the specification part of the subprogram or its interface block. Alternatively, a `routine` directive can specify a procedure name, and can then appear anywhere in a C or C++ file that a function prototype is allowed or in a Fortran specification part of a subprogram.

A `routine` directive has a list of clauses, including one required clause to specify the level of parallelism used in the routine. One of the following clauses must appear on any `routine` directive:

- `gang` clause, if the routine has a *gang*-parallel loop, or calls a routine with a *gang*-parallel loop.
- `worker` clause, if the routine has a *worker*-parallel loop, or calls a routine with a *worker*-parallel loop.
- `vector` clause, if the routine has a *vector*-parallel loop, or calls a routine vector a *worker*-parallel loop.
- `seq` clause, if the routine has no parallel loops and calls no routines with parallel loops.

There are other clauses (`bind`, `device_type` and `nohost`), but they are not related to this paper's contribution.

For procedures that are defined in one source file and called in a compute construct in other files, a `routine` directive must appear in the file where the procedure is defined, and another `routine` directive with the same level-of-parallelism clause must be included in each source file where the function is called. This requires training and additional programming effort for beginning

```

#pragma acc routine seq
extern float externfunc1( float );
...
void test( float* x, int n) {
    #pragma acc parallel loop copy(x[0:n])
    for( int i = 0; i < n; ++i )
        x[i] = externfunc1(x[i]);
}

```

(a) main.c

```

#pragma acc routine seq
float externfunc2( float* x ){
    return x%2;
}
...
#pragma acc routine seq
float externfunc1( float* x ){
    return externfunc2(x) + 1;
}

```

(b) external.c

Fig. 1: Function call for an external function inside a parallel loop of a OpenACC C program

OpenACC programmers, and can require significant effort and attention from users when porting, maintaining, and updating large applications to OpenACC. We intend to ease these restrictions in this work.

An example of how a OpenACC programmer must mark all the declarations and the definition for the function being called inside a parallel region is shown in Figure 1. In the example, the parallel construct defined in the *test* function in *main.c* calls an external function called *externfunc1* defined in *external.c*. The declaration for *externfunc1* in *main.c* is marked with the OpenACC **routine** directive with the **seq** clause. The same OpenACC directive is also marked for the function in *external.c* where it is defined. Not doing so will lead to link time error, since the accelerator code for *externfunc1* would not be generated during compilation. In addition to this, *externfunc1* calls another function called *externfunc2* which needs to be marked with the **routine** directive as well and must be marked before its usage in *externfunc1*. If this function is not marked with **routine** directive then compiler will generate a compile time error at the call site, since it appears in the same file as the call site.

3 Implementation of Automatic Routine Directive Propagation

This section describes our implementation of OpenACC routine propagation using the PGI compiler Interprocedural Analysis (IPA) feature. The PGI IPA feature is described briefly in Section 3.1. Section 3.2 describes how the modifications to the initial compile step, and in particular what information is gathered for IPA. The following subsections describe how IPA navigates the call graphs to determine what `routine` directives need to be added, and then what functions or files need to be recompiled. The final subsection briefly describes the recompile step.

3.1 PGI Interprocedural Analysis

The PGI IPA feature is based on the whole program analysis work done by researchers at Rice University [1, 3–7, 10]. It is divided into three distinct parts: the initial compile step, the link-time interprocedural analysis, and the recompile step, when necessary.

When compiling a file with the PGI compilers and IPA enabled (with the `-Mipa` flag), the compiler adds one additional analysis phase early in the compile sequence. This is the IPA *summary* phase, which saves a concise summary of each procedure and all the global variables in the program. The summary contains information such as:

- all procedures called, including indirect procedure calls, along with the arguments to those procedures
- what variables or procedures have had their addresses taken
- assignments to global variables or reference arguments
- loops in the procedure
- an estimate of the size and complexity of the procedure

This phase also saves *recompile* information, which includes the program intermediate representation as well as information such as the compiler flags used to build the object file, so the routines in this file can be recompiled without the source files or can be inlined into procedures in other files.

Unlike almost all other compilers with interprocedural analysis or whole program optimization, the PGI compiler then continues to produce a valid object file. That object file would obviously not be optimized with IPA, but otherwise is a valid object file that can be saved in an object library or linked without IPA. The IPA summary and recompile information is appended in special sections of the object file. This means that an object file compiled with IPA enabled can be linked with or without IPA. When linked without IPA, the generated program will not benefit from IPA, since it will be using the non-IPA-optimized object file, but there are no drawbacks.

The interprocedural analysis occurs at link time. When linking with IPA enabled, the PGI interprocedural analyzer is invoked before the linker itself.

The analyzer opens all the object files on the link line, looking for the IPA summary information saved there. It also opens object libraries, just like the linker, satisfying unresolved external references and looking for IPA summary information in the library objects.

For object files that don't have IPA summary information, the analyzer reads the object file symbol table and assumes *worst case* information. Any symbol defined in a worst-case object file that is called from another object file is assumed to be a procedure. Any symbol defined in a worst-case object file that is not called from another object is assumed to be a global variable. Any variable symbol referenced in a worst-case object file is assumed to be read and written, and to have its address taken, in any procedure defined in that object. Any procedure symbol referenced in a worst-case object file is assumed to be called from any procedure defined in that file.

The analyzer then builds a complete call graph, using the mechanism described by Cooper and Kennedy [6], along with the argument binding graph. The call graph nodes are the procedures, and edges represent the procedure caller to callee relationship. The argument binding graph nodes are variables, constants, and procedure dummy arguments, and the edges represent when a variable, constant, or a dummy argument is passed as a value or reference actual argument at a call site to a dummy argument. These are the main data structures used in the analyzer. Most IPA analyses requires one or a few traversals of these graphs, either forward or backward, to propagate information across the edges and make decisions about what optimizations may be performed.

When the analysis is complete, the IPA step decides what functions (and hence what files) to recompile. Because of the structures used, the PGI compiler will recompile whole object files. The analyzer reinvokes the compiler on those files, using the generated IPA information, to generate *optimized object files*. When all these recompilations are complete, the analyzer then invokes the system linker, substituting optimized object files on the link line, or adding optimized object files for any that were extracted from libraries.

There are a few engineering details to make the recompile step more efficient. When recompiling files at link time, IPA can do several recompiles at a time, much like `make -j`. Also, the optimized object files are not generally deleted, but saved with a `.oo` suffix. The interprocedural information that was used to optimize that file is appended as a special section to the optimized object. If the same program is rebuilt in that directory, and IPA sees an existing optimized object for a file, it will compare the dates for the original and optimized object file, and the interprocedural information that was used to generate the existing optimized object to the interprocedural information that it just generated. If the optimized object is stale (the original object file is newer), or the interprocedural information has changed, then the optimized object must be rebuilt.

3.2 Initial Compile Summary Information And Error Suppression

This subsection describes how we augmented the IPA summary information collection step for routine propagation. First, if a procedure was compiled for

the device with an **acc routine** directive, the information for that directive is added to the summary information. Second, the summary information for a procedure calls includes whether it appears in an OpenACC compute construct or a procedure marked by a **routine** directive. Third, any information from **acc routine** directives for called procedures is also added to the summary information. Fourth, information about global variables is extended to include whether that variable appears in an **acc declare** directive.

Normally, a procedure call to a routine in an OpenACC compute construct or device routine that does not have appropriate **acc routine** information results in a compiler error message. Since our new method will implicitly generate or propagate the **routine** information at link time, we suppress these errors for the initial compile step. As mentioned in the previous subsection, the PGI IPA initial compile phase produces a complete object file. In this case, however, the object file cannot be complete, since the compiler can't determine what level of parallelism should be used for procedures without **routine** information. This is the most significant change in the compile workflow. If a user builds a program with this optimization enabled but then links without IPA, the program will fail to link.

3.3 Propagating acc routine information with IPA

There are five small modifications to the interprocedural analysis to support **acc routine** propagation. These are all done with a single depth-first pass through the call graph. Indirect procedure calls (through a function pointer) are not handled. See the discussion on limitations in Section 5.

First, a procedure call in an OpenACC compute construct or in a device routine was so annotated in the summary information. This allows IPA to check whether the callee was already marked as a device routine. If not, IPA will mark it as an implicit device routine and recurse to any of its callees. Second, if the callee was marked as a device routine but the file containing the caller did not have that routine information, IPA will propagate that information to the callee. Third, if the callee is marked as a device routine with one parallelism level, but the caller had an **acc routine** directive for that routine with a different parallelism level, IPA will generate a fatal error message. Fourth, IPA will assign a parallelism level for any implicit device routine. Normally, IPA will treat any implicit device routine as **acc routine seq**. However, if this routine calls another routine with a higher parallelism level (gang > worker > vector > seq), the implicit device routine is assigned the highest parallelism level of all routines called.

During the call graph traversal, IPA checks for several types of errors, and issues fatal error messages if any are found. IPA checks that the call site is at the proper level of parallelism for the callee. For instance, a call within a gang-parallel loop cannot be made to a **routine gang** procedure, because gang-level parallelism was already exploited at the call site. IPA also checks that explicit **acc routine** information at each call site and procedure definition match. Finally, IPA checks that any global variable referenced in an implicit device routine has appeared in an **acc declare** directive. IPA can't implicitly insert an **acc**

`declare` directive without also inserting data movement to keep the host and device copies of the global variable coherent, and that’s a problem that we don’t know how to solve at this point. However, it can and does propagate the `acc declare` from the definition site to any file where it is referenced in an OpenACC construct or a device routine.

3.4 Recompile Step

Assuming no errors were detected, IPA finishes by determining whether any new interprocedural information can be used to optimize any function in each object file. Implicit OpenACC routine information is one such type of information. As mentioned earlier, interprocedural optimizations are usually optional, as the existing initial compile generates a complete object file. In this case, the recompile is required, since the initial compile was made with inadequate information. IPA will create a description of the implicit `acc routine` information along with any other interprocedural optimization information, and reinvoke the compiler to reprocess any object file that needs to be recompiled. This might be because the object file contains a procedure that needs to be compiled with a new implicit `acc routine` directive for the device, or because the object file contains an OpenACC compute construct or device routine that calls some other procedure for which no routine information was available in the initial compile. The recompile then generates an optimized object file, which is then used in the final link.

4 Examples

In this section, we use C code snippets to show how OpenACC routine propagation benefits programmers.

4.1 Propagating `acc routine seq` as default

This example shows implicit device routine generation. The original code is shown in Figure 2. The parallel region in function *test1* in file *main1.c* calls function *func1a* in file *func1.c*. Further, *func1a* calls *func1b* in that same source file. In this case the programmer omits the explicit `acc routine` directive from the *func1a* prototype in *main1.c* as well as from the definitions of *func1a* and *func1b* in *func1.c*. IPA recognizes the call to *func1a* in the compute construct in *test1* and marks it by default as a device routine. It then marks *func1b* as a device routine because it is called in *func1a*. Since *func1b* has no further calls, IPA will by default mark it as `acc routine seq`. Since *func1a* only calls one `seq` routine, it is marked the same way. Here, both source files are recompiled with the implicit `routine` directive information in both files.

4.2 Propagating routine type across files

Here we present two examples of IPA propagation of `acc routine` level of parallelism. The first example propagates the level of parallelism from the procedure

```

extern float func1a( float );
...
void test1( float* x, int n){
    #pragma acc parallel loop copy(x[0:n])
    for( int i = 0; i < n; ++i )
        x[i] = func1a(x[i]);
}

```

(a) main1.c

```

float func1b( float* x ){
    return x%2;
}
...
float func1a( float* x ){
    return func1b(x) + 1;
}

```

(b) func1.c

Fig. 2: Implicit **acc routine** generation.

definition to the call sites; the sample code is shown in Figure 3. The programmer has inserted an **acc routine** directive with the **gang** clause for the function definition of *func2a* in *func21.c*. IPA propagates this annotation to the function prototype in *main2.c*. Similarly, the programmer has inserted an **acc routine** directive with the **worker** clause for the function definition of *func2b* in *func22.c*. IPA propagates this to its prototype in *func21.c*. In this case, IPA only needs to recompile the two files *main2.c* and *func21.c*, because no new information was propagated to *func22.c*.

The second example propagates the level of parallelism from a prototype declaration to the function definition; the sample code is shown in Figure 4. Here the prototype of *func3a* in *main3.c* is annotated with **acc routine vector**. IPA propagates this annotation to the definition of *func3a* in *func3.c*. Since *func3b* has no annotation, IPA adds the default **acc routine seq** directive. Only *func3.c* is recompiled, since *main3.c* has no new information.

4.3 Detecting routine level of parallelism mismatch across files

This example shows error detection of mismatched level of parallelism between between a prototype at the call site and the definition in another file, or between two prototypes in different files; the sample code is in Figure 5. Here the programmer has marked the declaration of *func4a* in *main4.c* as **acc routine gang**, but marked its definition in *func4.c* as **acc routine vector**. This could have led to incorrect execution and wrong output on execution. IPA detects this mismatch and generates an error identifying the inconsistency.

```

extern float func2a( float );
...
void test2( float* x, int n){
    #pragma acc parallel loop copy(x[0:n])
    for( int i = 0; i < n; ++i )
        x[i] = func2a(x[i]);
}

```

(a) main2.c

```

extern float func2b( float );
...
#pragma acc routine gang
float func2a( float* x ){
    return func2b(x) + 1;
}

```

(b) func21.c

```

#pragma acc routine worker
float func2b( float* x ){
    return x%2;
}

```

(c) func22.c

Fig. 3: Propagating level of parallelism from function definition to call sites.

4.4 Detecting unannotated global variable usage

One additional feature of the implementation is described here in Figure. 6. In this case the function *func5a* is called from a parallel region in function *test5*. While the programmer has omitted the **acc routine** annotation for function *func5a*, but IPA detects this and marks it by default as **acc routine seq**. But function *func5a* accesses the global variable *glob_y*. Since this global variable does not appear in a **acc declare** directive, it cannot be accessed from the device. IPA detects this and generates an appropriate error message. Without this feature, the error would have been produced in later stage where this file was being recompiled.

5 Summary

OpenACC requires that routines to be called on an accelerator device be annotated, so the compiler will know to generate device code for calls to them. The directive annotation must also declare the level of parallelism used in that procedure, so the correct code can be made at the call site. The current OpenMP

```

#pragma acc routine vector
float func3a( float );
...
void test3( float* x, int n){
    #pragma acc parallel loop copy(x[0:n])
    for( int i = 0; i < n; ++i )
        x[i] = func3a(x[i]);
}

```

(a) main3.c

```

float func3b( float* x ){
    return x%2;
}
...
float func3a( float* x ){
    return func3b(x) + 1;
}

```

(b) func3.c

Fig. 4: Propagating routine type from a declaration to the definition.

```

#pragma acc routine gang
extern float func2a( float );
...
void test4( float* x, int n){
    #pragma acc parallel loop copy(x[0:n])
    for( int i = 0; i < n; ++i )
        x[i] = func4a(x[i]);
}

```

(a) main4.c

```

#pragma acc routine vector
float func4a( float* x ){
    return x++;
}

```

(b) func4.c

Fig. 5: routine type mismatch between the definition and the declarations.

target directive set has an equivalent requirement. This is a cause of much frustration when porting programs, particularly for C++ programs where many calls are made to templated functions that appear in system header files that users

```

int glob_y = 100;
...
float func5a( float* x ){
    return x*glob_y + x;
}
...
void test5( float* x, int n){
    #pragma acc parallel loop copy(x[0:n])
    for( int i = 0; i < n; ++i )
        x[i] = func5a(x[i]);
}

```

(a) main5.c

Fig. 6: Global variable inside accelerator code without declare directive.

can't modify. The specific problem for C++ programs is handled by the PGI compiler in the C++ front end, by transitively marking any unannotated procedure that can be called from an OpenACC compute construct with **acc routine seq**. The same mechanism could be applied to C or Fortran programs as well. However, while extremely useful, it does not address the problem of inserting these directives when the caller and callee appear in separate files.

This paper describes a semiautomatic approach to detecting and generating implicit device routine annotations across files, using interprocedural analysis. For simple procedures, IPA will mark these for single thread execution with the **seq** level of parallelism. For procedures that contain parallel loops, the user has to mark only the procedure definition with the appropriate level of parallelism, and IPA will propagate that to all call sites. IPA can also detect common errors, such as an invalid call site, mismatched directives for the same routine, or global variable use in an implicit device routine. While our implementation is designed for OpenACC, it could be applied equally for an OpenMP compiler that implements target offload to an accelerator device.

Our implementation is a straightforward addition to the flexible interprocedural analysis and optimization feature in the PGI compilers. Similar enhancements could be added to any compiler that implements whole program or interprocedural analysis. The major change to the PGI IPA framework is that with this feature enabled, interprocedural analysis and optimization at link time becomes required, since without the link-time propagation of the **routine** information, the device code will not link successfully.

There are limitations to the general procedure and limitations in our specific implementation. The implementation is based on traversal of the call graph. Indirect calls through procedure pointers generally will generate an imprecise call graph, and would require that either all possible target procedures be made available on the device, or that the user explicitly limit the set of target pro-

cedures. Our IPA implementation does not propagate information for indirect calls at this time.

Even though a programmer may not include the `routine` directive for a procedure, he or she may add `loop` directive for loops inside the routine. The `loop` directive annotations could be used to guide the level of parallelism to use for implicit device routines. Even without explicit `loop` directives, the compiler may be able to determine that a loop in a procedure could be safely executed in parallel. If such a procedure were called at sites where such parallelism could be exploited, an implementation could implicitly add a level of parallelism other than `seq`. Neither of these features is included in our implementation.

We are also exploring other situations where IPA can be used to shift work from the programmer to the compiler. One instance, specific to the PGI compiler for NVIDIA GPUs, has to do with using CUDA Managed Memory for allocatable data. The PGI compiler supports a `-ta=tesla:managed` option, which compiles the OpenACC compute constructs for execution on NVIDIA Tesla GPUs, and also replaces all dynamic memory allocation to use CUDA managed memory. Managed memory will be automatically migrated or paged between host and device memories, and thus reduces the need for OpenACC data directives and simplifies the job for the programmer. However, managed memory allocation and deallocation is relatively expensive. Currently, the `managed` suboption replaces all dynamic memory allocation. We are exploring whether we could use IPA to only replace the dynamic allocation for data that gets used in OpenACC compute constructs, using the cheaper system memory allocation for data that is only used on the host CPU.

Our IPA currently allows the compiler to automatically choose which procedures to inline, even across files. Inlining is especially beneficial for NVIDIA GPUs; the overhead of maintaining the call stack and the information lost at procedure boundaries is relatively more costly for a GPU. We are exploring optimizing the inlining decisions to prioritize inlining of procedures called in a device compute construct.

We are also exploring whether we can use IPA to find more errors at compile time. For instance, if a compute construct includes a `present` clause for some data structure, IPA might be used to determine whether there are calls to the procedure contains the compute construct where the data structure is not present, and to issue a compile-time warning or error, avoiding the need to trace this error at runtime.

Finally, we are working with the OpenACC committee to explore whether it is useful to compile multiple versions of a procedure, each with a different level of parallelism. For instance, a SAXPY procedure might be called in several different contexts. If called within a `vector` loop, the procedure must be compiled to run on a single thread in `routine seq` mode. However, if called in a `gang` loop, it could be compiled to run in parallel in `routine worker` or `routine vector` mode. We have seen at least one production application where such a feature would be useful. If that feature is added to OpenACC, then the `routine`

propagation procedure would be modified to detect this situation and propagate the information appropriately.

References

1. D. Callahan, K. Cooper, K. Kennedy, and L. M. Torczon. Interprocedural constant propagation. In *Proc. SIGPLAN '86 Symp. on Compiler Construction*, pages 152–161, Palo Alto, CA, June 1986.
2. S. Chandrasekaran and G. Juckeland, editors. *OpenACC for Programmers*. Addison-Wesley, Boston, Mass., 2018.
3. K. Cooper. Analyzing aliases of reference formal parameters. In *Conf. Record 12th Annual ACM Symp. Principles of Programming Languages*, pages 281–290, Jan. 1985.
4. K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proc. SIGPLAN '84 Symp. on Compiler Construction*, pages 247–258, Montreal, Canada, June 1984.
5. K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Proc. ACM SIGPLAN '89 Conf. on Principles of Programming Languages*, pages 29–41, February 1986.
6. K. Cooper and K. Kennedy. Efficient computation of flow-insensitive interprocedural summary information (a correction). Technical Report TR87-60, Rice University, 1987.
7. K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proc. ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 57–66, Atlanta, GA, June 1988.
8. CUDA toolkit documentation. <http://docs.nvidia.com/cuda/>.
9. R. Farber, editor. *Parallel Programming with OpenACC*. Morgan Kaufmann, Boston, Mass., 2017.
10. M. Hall and K. Kennedy. Efficient call graph analysis. *Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.
11. J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, Mar. 2008.
12. The OpenACC application programming interface, version 2.6. <https://www.openacc.org/>, Nov. 2017.
13. OpenCL. <https://www.khronos.org/opencl/>.
14. The OpenMP application programming interface, version 4.5. <https://www.openmp.org/>, Nov. 2015.
15. G. Ruetsch and M. Fatica. *CUDA Fortran for Scientists and Engineers*. Morgan Kaufmann, San Francisco, Cal., 2013.