OpenACC Routine Directive Propagation using Interprocedural Analysis

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

Michael Wolfe
NVIDIA/PGI

WACCPD @ SC 2018 Dallas, TX

Porting Applications to OpenACC (1/2)

- Programmers need to specify:
  - what data to copy to or from the device memory
  - what code to compile for and run on the accelerator device
- Programmers insert directives around code regions (compute construct).
- If the compute construct calls a procedure:
  - Marking the procedure and any routine called inside the procedure, and so on
  - Marking the kind of parallelism exploited with the procedures and the routines

Porting Applications to OpenACC (2/2)

- Ways to determine device code:
  - Compile all procedures for the accelerator (not viable for all accelerators)
  - Compiler chooses the procedures for the accelerator (viable for procedures in the same file)
    - Ex: PGI C++ compiler
  - Programmer mark the procedures needed on the device (including level of parallelism)
- OpenACC and OpenMP target directives require programmers' involvement.
- OpenACC requires **`routine`** directive for each procedure called on device.

Marking OpenACC Routines

- For routines called inside compute construct in C/C++:
  - If only called in the same file as the definition, then at the definition only.
  - If called from a separate file(s), then at all declaration(s) and the definition.
- For Fortran, routine information propagated through module mechanism.
- In case of missing routine directive:
  - Compilation error
  - Link-time error
  - Runtime error, if mismatch in level of parallelism

Contributions

- Ease OpenACC programming using Interprocedural Analysis feature
  - Implemented as part of PGI IPA feature
- By adding or propagating OpenACC routine directives throughout an application
- Detecting error when existing directives don't match
- Detecting unannotated global variable usage

OpenACC Routines Directives

- Types of parallelism for routines:
  - **gang** clause
  - **worker** clause
  - **vector** clause
  - **seq** clause

`Order: gang > worker > vector > seq`

```
#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]);
}
```

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;
}
```

external.c

PGI Interprocedural Analysis (-Mipa)

- Three phases:
  - Summary phase (Compilation)
    - Saves information such as procedures called, loops in procedures, global variable modification, etc.
  - Interprocedural Analysis phase (Link-Time)
    - Collect IPA information from object files
    - Builds a complete call graph
    - Propagates information forward and backward through call graph
  - Recompilation phase
    - Decides what files need to recompiled

Routine Propagation using IPA

- Summary Phase
  - Add `routine` directive information to the summary for the definition
  - Add whether procedure call appears in an OpenACC compute construct
  - Suppress compiler error message for missing `routine` directives

Routine Propagation using IPA

- Interprocedural Analysis phase (Case 1)
  - Check if routine marked explicitly.
  - If not, mark device routine
  - Mark as **acc routine seq**
  - Recurse to any of its callees

```
#pragma acc routine seq
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]);
}
```

main1.c

```
#pragma acc routine seq
float func1b( float* x ){
        return x%2;
}
...
#pragma acc routine seq
float func1a( float* x ){
        return func1b(x) + 1;
}
```

func1.c

Routine Propagation using IPA

- Interprocedural Analysis phase (Case 2)
  - Propagate directive
  - Propagate level of parallelism
  - From Definition to Declarations

```
#pragma acc routine gang
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]);
}
```

main2.c

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

func21.c

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

func22.c

Routine Propagation using IPA

- Interprocedural Analysis phase (Case 3)
  - Propagate directive
  - Propagate level of parallelism
  - Declaration to Definition
  - Declaration to other Declarations

```
#pragma acc routine vector
extern 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]);
}
```

main3.c

```
#pragma acc routine seq
float func3b( float* x ){
        return x%2;
}
...
#pragma acc routine vector
float func3a( float* x ){
        return func3b(x) + 1;
}
```

func3.c

Routine Propagation using IPA

- Interprocedural Analysis phase (Case 4)
  - Detect mismatch in level of parallelism
  - Generate fatal error message

```
#pragma acc routine gang -> IPA error
extern float func4a( 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]);
}
```

main4.c

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

func4.c

Routine Propagation using IPA

- Interprocedural Analysis phase (Case 5)
  - Checks if global variable referenced in implicit routine
  - If directive present, propagate **`acc declare`**
  - If not, IPA generates error
  - Missing data movement

MISSING: acc declare … (glob_y) -> IPA error

```
int glob_y = 100;
...
#pragma acc routine seq
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]);
}
```

main5.c

Routine Propagation using IPA

- Recompile Phase
  - Create description of the implicit acc routine information
  - Reinvoke compiler to recompile object file only if:
    - New implicit acc routine directive
    - Calls procedure with no explicit routine information

Summary

- Easier porting of applications (C, C++, Fortran) to OpenACC
- Added feature in the PGI OpenACC compiler
- Advantages of the presented approach:
  - Add new implicit `acc routine` directives
  - Propagate redundant marking of `acc routine` across file
  - Prevents link-time and run-time errors in applications
  - Only recompile files with new information
- Limitation of the presented approach:
  - Indirect calls through procedure pointer