P4343—

**OpenUH: Open Source OpenACC Compiler**

Xiaonan (Daniel) Tian, Rengan Xu and Barbara Chapman

HPCTools Group

Computer Science Department

University of Houston

GTC2014, San Jose, CA; 03/26 /2014
I. Motivation
Motivation

**WHY** do we implement OpenACC support in OpenUH?

- Performance gap between OpenACC and CUDA → more research on OpenACC compiler optimization
- Open Source OpenACC compiler is required for research purposes.

**WHY** is this talk important?

- BETTER understand OpenACC implementation, BETTER knowledge on application optimization.
II. Introduction to OpenUH

Website: http://web.cs.uh.edu/~openuh/
Source: https://github.com/pumpkin83/OpenUH-OpenACC
Email: openuh@cs.uh.edu
Introduction to OpenUH

- Open Source Research Compiler
  - Open64 based
  - Support C/C++/Fortran/Coarray
- Parallel Programming model
  - OpenMP
  - OpenACC
  - COARRAY
Introduction to OpenUH OpenACC

- **OpenACC 1.0 implementation**
  - **Directives**: Parallel, kernels, Data, Loop, Wait
  - **Data Clause**: copy/copyin/copyout/create/update
  - **Loop** Scheduling Clauses: gang/worker/vector
  - **Async** clause: async/wait
  - Unsupported: host_data/declare/cache
Introduction to OpenUH OpenACC

OpenUH OpenACC Compiler Infrastructure

- Frontends (C, OpenACC)
- IPA (Inter Procedural Analyzer)
- PreLower (Preprocess OpenACC)
- WOPT (Global Scalar Optimizer)
- Lower (Transformation of OpenACC)
- LNO (Loop Nest Optimizer)
- WHIRL2CUDA
- CG (Code for IA-32, IA-64, X86_64)

Source Code with OpenACC Directives

- GPU Code
- CPU Binary
- Linker
- Executable

NVCC Compiler
PTX Assembler
Runtime Library
Loaded Dynamically
III. LOOP SCHEDULING
Loop Scheduling

1. What’s Loop Scheduling?

2. Parallel Loop Scheduling

3. Kernels Loop Scheduling
Loop Scheduling

• What is Loop Scheduling?
  – Solutions to distribute sequential loop iterations across a large number of threads

• Why we have two different Loop Scheduling strategies?
  – Explore multi-dimensional topology of NVIDIA GPGPU architecture
Loop Scheduling

- `#pragma acc loop gang(4)`
- `For(i=0; i<11; i++) {...}`

Iterations:
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

Gangs:
- 0
- 1
- 2
- 3
Loop Scheduling

• `#pragma acc loop gang(4)`
• `For(i=0; i<11; i++) {...}`
Loop Scheduling

• `#pragma acc loop vector(64)`
• `For(i=0; i<99; i++) {...}`
Loop Scheduling

- `#pragma acc loop gang(3) vector(32)`
- `For(i=0; i<130; i++){...}`
Parallel Loop Scheduling

- **Gang** → (CUDA) thread-block
- **Worker** → (CUDA) y dimensional threads in a thread block
- **Vector** → (CUDA) x dimensional threads in a thread block
  - 1D Grid, and 1D/2D thread-block.
  - # of Worker * # of Vector ≤ 1024
  - Requires minimal lower-level knowledge.
  - Follows OpenACC 2.0: gang contains worker and vector; worker can only include vector.
Parallel Loop Scheduling

1. Single Loop
   - #pragma acc loop gang worker vector
   - for(...){}

2. Two-level Nested Loop
   2.1. loop gang / loop worker vector
       - #pragma acc loop gang
       - for(...){
           - #pragma acc loop worker vector
               - for(...){
                   - }
           - }
   2.2. loop gang worker / loop vector
       - #pragma acc loop gang worker
       - for(...){
           - #pragma acc loop vector
               - for(...){
                   - }
           - }
   2.3. loop gang / loop vector
Parallel Loop Scheduling: example

- `#pragma acc loop gang(2) worker(4) vector(64)`
- `For(i=istart; i<iend; i++) {...}`
Parallel Loop Scheduling: example

- `#pragma acc loop gang(2) worker(4) vector(64)`
- `For(i=istart; i<iend; i++){...}`

<table>
<thead>
<tr>
<th>Iterations</th>
<th>CUDA Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 0</td>
<td></td>
</tr>
<tr>
<td>Block 1</td>
<td></td>
</tr>
</tbody>
</table>

:32 iterations/ threads
Parallel Loop Scheduling

- #pragma acc loop gang(2) worker(4) vector(64)
- For(i=istart; i<iend; i++){...}
Parallel Loop Scheduling

• `#pragma acc loop gang(2)`
  
  `for(i=istart; i<iend; i++)`
  
  `#pragma acc loop worker(4) vector(64)`
  `for(j=jstart; j<jend; j++)`...

  `}`
Parallel Loop Scheduling

- `#pragma acc loop gang(2)`

```c
for(i=istart; i<iend; i++){
    #pragma acc loop worker(4) vector(64)
    for(j=jstart; j<jend; j++){
        ...
    }
}
```

![Diagram of parallel loop iterations]

- Inner Loop Iterations
- Outer Loop Iterations
Parallel Loop Scheduling

- `#pragma acc loop gang(2)`

```c
for(i=istart; i<iend; i++){
    #pragma acc loop worker(4) vector(64)
    for(j=jstart; j<jend; j++){

    }
}
```

Diagram:
- Inner Loop Iterations
- Outer Loop Iterations
- ...
Parallel Loop Scheduling

- `#pragma acc loop gang(2) worker(4)`

```c
for(i=istart; i<iend; i++){
    #pragma acc loop vector(64)
    for(j=jstart; j<jend; j++){...}
}
```

Inner Loop Iterations

Outer Loop Iterations

...
Parallel Loop Scheduling

- #pragma acc loop gang(2) worker(4)

for (i=istart; i<iend; i++){
    #pragma acc loop vector(64)
    for (j=jstart; j<jend; j++){

    }

}
Parallel Loop Scheduling

3. Three level Nested Loop
   loop gang/loop worker/loop vector

#pragma acc loop gang
for(....)
  #pragma acc loop worker
  for(....)
    #pragma acc loop vector
    for(....)
    {
    }


Why do we need different strategies for implementing loop scheduling?

```cpp
#pragma acc loop gang(19)
for(i=0; i<19; i++)
    #pragma acc loop worker(32)
    for(j=0; j<1000000; j++)
        #pragma acc loop vector(32)
        For(k=0; k<1000000; k++)
        {
        }
What is the maximum threads we have?
19*32*32 = 19K
Try this loop scheduling

According the scheduling in the code, 2D grid and 2D thread-block in NVIDIA GPGPU are created.

```c
#pragma acc loop gang(19)
for(i=0; i<19; i++)
    #pragma acc loop gang(32) vector(32)
    for(j=0; j<1000000; j++)
        #pragma acc loop vector(32)
        for(k=0; k<100000; k++)
        {
            
        }
```

What is the maximum threads we have here?

- 19 * 32 * 32 * 32 = 32 * 19K
Kernels Loop Scheduling

- **Gang** → (CUDA) thread-block, can be in x, y, z dimension
- **Worker** → Ignored
- **Vector** → (CUDA) thread, can be in x, y, z dimension
  - Multi-dimensional grid/thread-block, both of them can be extended into 3 dimensional topology.
  - Fine tuning: provide more scheduling options for users.
  - Users need to have more knowledge about compiler and hardware information (currently, no autotuning)
  - Provided more choices to loop scheduling.
  - In some cases, it does help improve performance
Kernels Loop Scheduling

1. Single Loop
   #pragma acc loop gang vector
   for(...){}

2. Double Nested Loop
   2.1. loop gang / loop vector
   2.2. loop gang vector / loop vector
   2.3. loop gang / loop gang vector
   2.4. loop gang vector / loop gang vector
Kernels Loop Scheduling

3. Triple Nested Loop

3.1 loop gang / loop gang vector / loop vector
3.2 loop vector / loop gang vector / loop gang
3.3 loop gang vector / loop gang vector / loop vector
3.3 loop gang vector / loop gang vector / loop gang vector

...
Kernels Loop Scheduling: Example

- `#pragma acc loop gang(2) vector(4)`
  
  for(i=istart; i<iend; i++){

  #pragma acc loop gang(3) vector(64)
  
  for(j=jstart; j<jend; j++) {...}

}

Inner Loop Iterations

Outer Loop Iterations
Parallel Loop Scheduling

• #pragma acc loop gang(2) worker(4)

for(i=istart; i<iend; i++){
    #pragma acc loop vector(64)
    for(j=jstart; j<jend; j++){
        ...
    }
}

Inner Loop Iterations

Outer Loop Iterations
IV. DATA MOVEMENT
Data Movement

1. Data transfer between CPU and GPU

Multi-core CPU  GPU Thousands of Cores

Main Memory  Copyin  GPU Memory

Copyout

How to optimize?
Data Movement

2. Basic Implementation

- `copy → pcopy;
- `copyin → pcopyin
- `copyout → pcopyout
- `create → pcreate

Free buffer/variables when you exit the current region

Goal: Avoid duplicate data traffic (malloc, copyin, copyout)
Data Movement

2. Basic Implementation

#pragma acc data
data_clauses

{ }

#pragma acc data
data_clauses

{ }

#pragma acc kernels
data_clauses

{ ...

}
Data Movement

2. Basic Implementation

```c
#pragma acc data
data_clauses
{
  #pragma acc data
data_clauses
  {
    #pragma acc kernels
data_clauses
    {
      ...
    }
  }
}
```
Data Movement

- 3. Partial Array
  - `#pragma acc data create(xx[0:N])`
  - `{`
    - `Foo(&xx[start])`
  - `}`
  - `...`
Data Movement

- 3. Partial Array
- #pragma acc data create(xx[0:N])
- {
  - Foo(&xx[start])
- }
- ...

Memory Mapping Table:

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx</td>
<td>xx'</td>
</tr>
</tbody>
</table>

CPU Memory

GPU Memory
Data Movement

- 3. Partial Array
- #pragma acc data create(xx[0:N])
  
  ```
  { Foo(&xx[start]) }
  ```

- ... Foo(double* x)
  
  ```
  { #pragma acc parallel pcopy(x[n1:n2])
    { ... } }
  ```
Data Movement

3. Partial Array

```
#pragma acc data
create(xx[0:N])
{
  Foo(&xx[start])
}
```

... Foo(double* x)
```
{
  #pragma acc parallel
p-copy(x[n1:n2])
  {
    ...
  }
}
```
VI. Performance
Three-Level Nested Loop Scheduling

Wave13pt

Kernels
- OpenUH: g-gv-v scheduling
- PGI: default
- CAPS: default
- CRAY: default

Same experimental platform used for OpenUH, CAPS and PGI

CRAY platform used for Cray machine
NAS Benchmark

**NAS EP**

![Graph showing performance vs data size for NAS EP benchmark with different platforms and configurations.]

**Combined**: parallel + kernels

**Cray**: use default loop scheduling, #pragma acc loop

**NAS CG**

Same experimental platform used for OpenUH, CAPS and PGI

Cray platform used for Cray machine
NAS Benchmark

**NAS MG**

Combined: parallel + kernels

Same experimental platform used for OpenUH, CAPS and PGI

CRAY platform used for Cray machine
NAS Benchmark

**NAS BT**

**Combined**: parallel + kernels

**NAS LU**
V. Future and Conclusion
Future Work

- Support Fortran
- Support Xeon Phi/AMD GPGPUs and APU
- Perform more optimization: Irregular Memory access optimization
- Provide a more robust OpenACC implementation
Conclusion

- Open source OpenACC research compiler, based on Open64
- Competitive performance, compared to other commercial compilers
- Proposed regular loop scheduling for parallel region and non-standard loop scheduling for kernels region

Question?