MOTIVATION
What does HPC programmer need today?

- Performance → GPUs, multi-cores, other accelerators.
- Continuity → Fortran! Current and future large codes.
- Quick Porting → OpenMP, OpenACC, MPI etc.
- Cross platform compiler → LLVM, PGI, GNU, etc.

Conclusion, Fortran programer needs
OpenMP GPU Offload in Flang
AGENDA

• Motivation
• Background
• Integration of OpenMP GPU Offload into Flang
• Experimental Evaluation
• Conclusion
Programming GPU-Accelerated Systems

Separate CPU System and GPU Memories

GPU Developer View

System Memory

GPU Memory

PCIe
OPENMP 4.5
Device Offload Model

```c
!$OMP TARGET ENTER DATA MAP(to:x, y)

!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
DO i=1, n
  y(i) = a * x(i) + y(i)
ENDDO

!$OMP TARGET EXIT DATA MAP(from: y)
```

copy array x, y to the default device

copy array y from the default device

target: Offload region
teams: Create teams of threads
distribute: Distribute the iteration across the master threads of all teams
parallel: Create threads
do: Distribute the iteration across the threads that already exist in the team
THE FLANG PROJECT
An open source Fortran front-end for LLVM

Multi-year project: NNSA Labs, NVIDIA/PGI

Based on the PGI Fortran front-end

Re-engineering for integration with LLVM

Develop CLANG-quality Fortran front end

Glossary
ILM: PGI-IR generated by frontend
ILI: PGI-IR generated by backend
OPENMP GPU OFFLOAD IN FLANG

Design Goals

Provide an implementation of OpenMP 4.5 GPU Offload targeting NVIDIA GPUs in Flang

Aim to keep the same design as Clang OpenMP 4.5

Leverage the Clang/LLVM driver

Take advantage of LLVM’s OpenMP runtimes

Compatibility with Flang and Clang OpenMP GPU Offload
Driver invokes the compiler for each OpenMP target

flang1 generates the same ILM

OpenMP device codegen is implemented here.

Flang2 generates device specific code

Two ways of adding libomptarget-nvptx
1. Inline *.bc (precompiled cuda-clang)
2. Link *.a by nvlink

LLVM Offload Runtime

Driver invokes the compiler for each OpenMP target

flang1 generates the same ILM

OpenMP device codegen is implemented here.

Flang2 generates device specific code

Two ways of adding libomptarget-nvptx
1. Inline *.bc (precompiled cuda-clang)
2. Link *.a by nvlink

LLVM Offload Runtime

Driver invokes the compiler for each OpenMP target

flang1 generates the same ILM

OpenMP device codegen is implemented here.

Flang2 generates device specific code

Two ways of adding libomptarget-nvptx
1. Inline *.bc (precompiled cuda-clang)
2. Link *.a by nvlink

LLVM Offload Runtime
CODE TRANSFORMATION
Example with OpenMP CPU

• Use libomp(kmpc) as CPU OpenMP runtime
• Outline for each OpenMP construct

```c
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
DO i=1, n
   y(i) = a * x(i) + y(i)
ENDDO
```

*Generated LLVM-IR for Host target*

```c
define void @MAIN_ () {
   call void @targetFUNC(i32* %0, ...)
}
define internal void @targetFUNC(){
   call __kmpc_for_static_init_4 (... , %teamsFUNC)
   ret void
}
define internal void @teamsFUNC(){
   call __kmpc_for_static_init_4 (...)
   call __kmpc_for_static_init_4 (... , %parallelFUNC)
   call __kmpc_for_static_init_4 (...)
   ret void
}
define internal void @parallelFUNC(){
   call __kmpc_for_static_init_4 (...)
   ; <The loop body >
   call __kmpc_for_static_init_4 (...)
   ret void
}
```

- Outlining for $target
- Outlining for $teams & fork teams
- Distribute loop across $teams
- Outline for $parallel & fork threads
- Distribute loop across threads
CODE TRANSFORMATION
Example with OpenMP GPU Offload

• Offload target region to the device
• New code transformation for device
• Use libomptarget for GPU offload
• Use libomptarget-nvptx in device OpenMP RT

```c
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO MAP(to:x) MAP(tofrom:y)
DO i=1, n
   y(i) = a * x(i) + y(i)
ENDDO
```

```c
define void @MAIN__ () {
   call void @__tgt_target_teams(i8* %kernelID, ...)
}
```

*Generated LLVM-IR for Host target*

```c
define ptx kernel void @kernel(){
   call @teamsFUNC (...)
   ret void
}
define internal void @teamsFUNC(){
   call void @__kmpc_for_static_init_4 (...) 
   call void @parallelFUNC(...)
   call void @__kmpc_for_static_fini (..)
   ret void
}
define internal void @parallelFUNC(){
   call void @__kmpc_for_static_init_4 (...)
   ; <The loop body >
   call void @__kmpc_for_static_fini (..)
   ret void
}
```

*Generated LLVM-IR for NVPTX target*

Outline for $parallel & call func
Outline for $teams & call func
Distribute loop across $parallel
Distribute loop across $teams
Outlining for $teams & call func

Offload $target to GPU
OUTLINER OPTIMIZATION
Reduce number of outlining for device code

Calling function in the device is not efficient

Omit outlining for combined constructs in device!

Generated LLVM-IR - Default

```llvm
define ptx_kernel void @kernel(){
  call @teamsFUNC (...)
  ret void
}
define internal void @teamsFUNC(){
  call void @_kmpc_for_static_init_4 (...) !$distribute
  call void @parallelFUNC(...)
  call void @_kmpc_for_static_fini (...)
  ret void
}
define internal void @parallelFUNC(){
  call void @_kmpc_for_static_init_4 (...) !$do
  ; <The loop body >
  call void @_kmpc_for_static_fini (...)
  call void @_kmpc_for_static_fini (...)
  ret void
}
```

Generated LLVM-IR - Reduced number of outlining in device

```llvm
define ptx_kernel void @kernel(){
  call void @__kmpc_for_static_init_4 (...) ; !$distribute
  call void @__kmpc_for_static_init_4 (...) ; !$do
  ; <The loop body >
  call void @__kmpc_for_static_fini (...)
  call void @__kmpc_for_static_fini (...)
  call void @__kmpc_for_static_fini (...)
  ret void
}
```
EXPERIMENTAL EVALUATION
Compilers studied in our experiments

We run our experiments on IBM POWER8NVL and NVIDIA Pascal GPUs with CUDA 8.0

<table>
<thead>
<tr>
<th>Compilers</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flang - OpenMP GPU (this work)</td>
<td>-O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -Mpreprocess</td>
</tr>
<tr>
<td>Clang 7.0</td>
<td>-O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda</td>
</tr>
<tr>
<td>PGI 18.7</td>
<td>-Mpreprocess -fast -ta=tesla,cc60,cuda8.0 -mp</td>
</tr>
<tr>
<td>IBM XLF 16.0.1</td>
<td>-qsmp -qoffload -qpreprocessor -O3</td>
</tr>
</tbody>
</table>
EXPERIMENTAL EVALUATION (II)

Compilers studied in our experiments

We experiment flang with different runtimes

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Target Offload Runtime</th>
<th>Device OpenMP Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flang</td>
<td>LLVM Offload RT - <em>libomptarget</em></td>
<td>Compile device RT with nvcc Link <em>libomptarget-nvxptx</em></td>
</tr>
<tr>
<td>Flang</td>
<td>IBM XL Offload RT - <em>libxlsmp</em></td>
<td>Compile device RT with nvcc Link <em>libomptarget-nvxptx</em></td>
</tr>
<tr>
<td>Flang</td>
<td>IBM XL Offload RT - <em>libxlsmp</em></td>
<td>Compile device RT clang-cuda Inline <em>libomptarget-nvxptx.bc</em></td>
</tr>
</tbody>
</table>
DAXPY

\[ y = y \times ax \]
Performs are considered ESTIMATES per SPEC run and reporting rules.
SPEC® is a registered trademark of the Standard Performance Evaluation Corporation (www.spec.org).
CONCLUSION

Flang OpenMP GPU Offload

• Introduced OpenMP GPU Offload to Flang
  • Support for combined-constructs

• Proposed compatible implementation with Clang

• Flang performance is in line with Clang
  • Obtained better performance with IBM XL’s target offload runtime