LLVM Support for OpenMP 4.0
Target Regions on GPUs

Supercomputing 2014
November 2014

Samuel Antao
Offloading in OpenMP

- OpenMP provides a **directive language** for controlling execution of a **parallel application**
  - **Control how threads cooperate**
    - Assign different loop iterations to different threads (parallel directive)
    - Assign task to threads controlled by a set of dependencies (task directive)
  - **Provide optimization hints** to take full advantage of the host machine
    - Mark loops for SIMDzation
    - Collapse loops, determine loop scheduling, identify reductions
  - **C/C++ and Fortran**
- **OpenMP 4.0 introduces offloading**: the ability to transfer execution to a target device present in the system
  - **Mark target regions** through dedicated directives
  - **Control data movement** between host and device
  - Most OpenMP **directives valid for the host** can also be **used inside target regions**

**Challenge**: A good OpenMP implementation for one target may perform poorly for others – Each target requires a tuned implementation
Offloading in OpenMP – Impl. components

- Input Program
  - C/C++
  - Fortran

- OpenMP enabled compiler

- Fat binary
  - Host code
  - Device code

- Device runtime library

- Host runtime library
  - Host component
  - Target agnostic component
  - Target API

- Host machine

- Device
  - Device driver
  - Device


© 2014 IBM Corporation
Offloading in OpenMP – Impl. components

Input Program
C/C++ Fortran

Clang

Fat binary
Host code
Device code

Device runtime library

Host runtime library

Host component
Target agnostic component

Target API

Host machine

Operating System

Device Driver

Device

NVIDIA CUDA

K40

Clang and LLVM in a nutshell

- **LLVM**
  - *Open-source* software infrastructure
  - Develop *compilers and other toolchain components*
  - Modern codebase with a large collection of *state-of-the-art algorithms and data structures* ready to be used for the most common compiler optimizations and code generation
  - Several *targets* supported including *PowerPC and CUDA-enabled GPUs*
  - Comprehensive but yet *simple intermediate language* (LLVM IR)
  - Rich and diverse ecosystem of projects and tools, including *Clang*

- **Clang**
  - LLVM IR client that operates as a *frontend to the C family programing languages*
  - Fast development pace, adopting the most recent language specs
  - Clang implements a driver to several LLVM components

- Both Clang and LLVM adopt a modular codebase that eases the integration of new features and enhances the maintainability
- **License** enables the development of commercial products on top of LLVM

- OpenMP support has been gradually added Clang as an external project that only recently started to be merged into the main project codebase.
Clang with OpenMP

• Compiler actions:
  – **Driver** preprocesses input source files *using host preprocessor*
  – For each preprocessed file, the driver spawns a **job using the host compiler** and an **additional job for each target** specified by the user
  – Flags informing the frontend that we are compiling code **for a target so only the relevant target regions are considered**
  – **Target linker creates a self-contained** (no undefined symbols) image file
  – **Target image file is embedded “as is”** by the host linker into the host fat binary
  – The **host linker** is provided with information to **generate the symbols required by the RTL**
Code generation for Nvidia GPUs

- **GPU constraints:**
  - GPUs have unique properties in terms of their execution model
    - Massive amount of threads executed in wavefronts (CUDA warps)
    - Threads divided by logical execution groups (CUDA blocks)
    - Threads within a block can cooperate more efficiently due to fast shared memory
    - Lightweight mechanism to schedule each thread
    - Considerable overhead when threads in the same warp diverge (serialization)
    - Threads spawning other threads causes significant overhead
    - Prone to deadlocks with misplaced synchronization barriers
  - GPU-specific constraints potentially require highly customized code generation in order to obtain interesting performance figures

- **Clang constraints**
  - Code generation should not disrupt the codebase
  - Centralize OpenMP related features into an independent module
  - Reuse the implementation of the code generation across different targets

- **Challenge: cope with both sets of constraints**
- **Our approach:** implement possible code generation approaches using CUDA and evaluate the implications.
Code generation for Nvidia GPUs – Example

```c
#pragma omp target map(to:a,b), map(c)
{
    #pragma omp teams num_teams(N), thread_limit(M)
    {
        // some sequential code

        #pragma omp distribute
        for (int i0 = 0 ; i0 < VECTOR_SIZE ; i0 += blockSize) {

            #pragma omp parallel for
            for (int i = i0 ; i < min (i0+blockSize, VECTOR_SIZE) ; i++)
                c[i] += a[i] + b[i];

        }

        // some sequential code
    }
}
```

![Diagram of code generation process](image-url)
Two possible approaches

- **CUDA dynamic parallelism**

  ```
  __global__ void parallelfor_kernel (..) {
    <codegen for parallel region>
  }

  __global__ void teams_kernel (..) {
    // sequential region
    parallelfor_kernel <<1, M>>> (..);
    cudaDeviceSynchronize ();
    // sequential region
  }

  __global__ void target_kernel (..) {
    teams_kernel <<N, 1>>> (..);
    cudaDeviceSynchronize ();
  }

  void hostFunction (..) {
    target_kernel <<1, 1>>> (..);
  }
  ```

- **If-master scheme**

  ```
  #define MASTER 0

  __global__ void kernel (..) {
    // only team master
    if (threadIdx.x == MASTER) {
      // sequential region
    }
    // all threads in block wait for master
    __syncthreads (); //required by #for

    // only team master
    if (threadIdx.x == MASTER) {
      // sequential region
    }
    // all threads in block wait for master
    __syncthreads ();
  }
  ```
Dynamic parallelism vs. if-master

- CUDA dynamic parallelism
- If-master scheme

- If-master perform more than 10x faster than dynamic parallelism
- If-master variance with #threads/blocks is lower

However...
- Control flow decisions may depend on multiple parallel or sequential regions
  - Require customization of code not directly related with OpenMP

- More complex control flows may require synchronizations in different paths
  - Barriers need to be reachable by all threads to avoid deadlocks
Control-loop scheme

- Single synchronization barrier hit by all threads
- Code generation for parallel and sequential regions remains untouched
- OpenMP-related code generation fully accomplished while processing a given OpenMP directive
- Less than 5% overhead regarding the if-master scheme

Full details:
Bertolli et al.
“Coordinating GPU Threads for OpenMP 4.0 in LLVM”
The LLVM Compiler Infrastructure in HPC Workshop, SC14
New Orleans, Louisiana, USA

```c
#define SEQ_REG1 0
#define PAR_REG1 1
__global__ void kernel () {
    __shared__ int labMaster; __shared__ int labOthers;
    if (threadIdx.x == MASTER)
        labelMaster = SEQ_REG1, labelOthers = IDLE;

    bool finished = false;
    while (!finished) {
        int nextLabel;
        __syncthreads ();
        if (threadIdx.x == MASTER)
            nextLabel = labMaster;
        else
            nextLabel = labOthers;
        switch (labelNext) {
            case IDLE: break;
            case SEQ_REG1:
                <code gen for seq region 1>
                <assign labMaster and labOthers>
                break;
            case PAR_REG1:
                <code gen for par region 1>
                if (threadIdx.x == MASTER)
                    <assign labMaster and labOthers>
                break;
            <..other cases.>
        }
    }
```
Where to get it

- **LLVM main repository**
  - [http://llvm.org](http://llvm.org)
  - Version 3.5
  - Version 3.6

- **Clang-OMP repository**
  - [http://clang-omp.github.io](http://clang-omp.github.io)
  - Initial version
  - Current version

- Clang/LLVM snapshot

- Changes gradually merged

- Added OpenMP features to Clang

How to use it:

- Grab the latest source files and **install LLVM as usual**
- Use the right options to **specify host and target** machines, e.g.:

  ```
  $ clang -fopenmp -target powerpc64le-ibm-linux-gnu -mcpu pwr8
  -omptargets=nvptx64sm_35-nvidia-cuda <source files>
  ```

Preliminary offloading support
Acknowledgments

• OpenMP 4.0 support in Clang has been a joint effort
  – Offloading model specification
  – Code drops
  – Code reviews

• Project contributors include
  – IBM
  – Intel
  – Texas Instruments
  – AMD
  – DoE Laboratories
  – Other distinguished members of the Clang/LLVM community
LLVM Support for OpenMP 4.0 Target Regions on GPUs