OpenMP compiler optimizations in LLVM
Johannes Doerfert, Argonne National Laboratory
OpenMP Compiler Optimizations in LLVM

Johannes Doerfert  (Argonne National Laboratory)

Adapted from our LLVM-Developers Meeting 2020 talk

https://youtu.be/M0DrhQbjrro
OpenMP in LLVM
http://openmp.llvm.org/docs
OpenMP in LLVM

http://openmp.llvm.org/docs

Clang

OpenMP Parser
OpenMP Sema
OpenMP CodeGen
OpenMP in LLVM
http://openmp.llvm.org/docs

Clang
- OpenMP Parser
- OpenMP Sema
- OpenMP CodeGen

OpenMP runtimes
- libomp.so (classic, host)
- libomptarget + plugins (offloading, host)
- libomptarget-nvptx (offloading, device)
OpenMP in LLVM

http://openmp.llvm.org/docs

Flang

Clang
- OpenMP Parser
- OpenMP Sema
- OpenMP CodeGen

OpenMP runtimes
- libomp.so (classic, host)
- libomptarget + plugins (offloading, host)
- libomptarget-nvptx (offloading, device)
OpenMP in LLVM

http://openmp.llvm.org/docs

Flang

Clang

OpenMP Parser
OpenMP Sema
OpenMP CodeGen

OpenMPIRBuilder

frontend-independent
OpenMP LLVM-IR generation
favor simple and expressive
LLVM-IR
reusable for non-OpenMP parallelism

OpenMP runtimes

libomp.so (classic, host)
libomp+target + plugins
  (offloading, host)
libomp+target-nvptx
  (offloading, device)
OpenMP in LLVM

http://openmp.llvm.org/docs

Flang

Clang

OpenMPIRBuilder

frontend-independant
OpenMP LLVM-IR generation
favor simple and expressive
LLVM-IR
reusable for non-OpenMP
parallelism

OpenMPOpt

interprocedural
optimization pass
contains host & device
optimizations
run with -02 and -03
since LLVM 11

OpenMP runtimes

libomp.so (classic, host)
libomptarget + plugins
(offloading, host)
libomptarget-nvptx
(offloading, device)
OpenMP in LLVM

http://openmp.llvm.org/docs

Flang

Clang

OpenMP
Parser

OpenMP
Sema

OpenMP
CodeGen

OpenMP
IRBuilder

frontend-independant
OpenMP LLVM-IR generation
favor simple and expressive
LLVM-IR
reusable for non-OpenMP
parallelism

OpenMPOpt

interprocedural
optimization pass
contains host & device
optimizations
run with -O2 and -O3
since LLVM 11

OpenMP
runtimes

libomp.so (classic, host)
libomptarget + plugins
(offloading, host)
libomptarget-nvptx
(offloading, device)
Design Goal

Report every successful and failed optimization
Optimization Remarks
Example: OpenMP runtime call deduplication

double *A = malloc(size * omp_get_thread_limit());
double *B = malloc(size * omp_get_thread_limit());

#pragma omp parallel
do_work(&A[omp_get_thread_num() * size]);
#pragma omp parallel
do_work(&B[omp_get_thread_num() * size]);

OpenMP runtime calls with same return values can be merged to a single call
**Optimization Remarks**

Example: OpenMP runtime call deduplication

```c
double *A = malloc(size * omp_get_thread_limit());
double *B = malloc(size * omp_get_thread_limit());

#pragma omp parallel
do_work(&A[omp_get_thread_num()*size]);
#pragma omp parallel
do_work(&B[omp_get_thread_num()*size]);
```

OpenMP runtime calls with same return values can be merged to a single call

```
$ clang -g -O2 deduplicate.c -fopenmp -Rpass=openmp-opt
```

deduplicate.c:12:29: remark: OpenMP runtime call omp_get_thread_limit moved to deduplicate.c:11:29: [-Rpass=openmp-opt]
  double *B = malloc(size*omp_get_thread_limit());
deduplicate.c:11:29: remark: OpenMP runtime call omp_get_thread_limit deduplicated [-Rpass=openmp-opt]
  double *A = malloc(size*omp_get_thread_limit());
```
Design Goal

Communicate and explain OpenMP implementation details to users
void bar(void) {
    #pragma omp parallel
    {} 
}

void foo(void) {
    #pragma omp target teams
    {
        #pragma omp parallel
        {}
        bar();
        #pragma omp parallel
        {}
    }
}

remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.

remark: Parallel region is not known to be called from a unique single target region, maybe the surrounding function has external linkage?, will not attempt to rewrite the state machine use.

remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.

remark: Specialize parallel region that is only reached from a single target region to avoid spurious call edges and excessive register usage in other target regions. (parallel region ID: __ompOutlined__1_wrapper, kernel ID: __ompOffloading_35_a1e179_foo_i7)

remark: Target region containing the parallel region that is specialized. (parallel region ID: __ompOutlined__1_wrapper, kernel ID: __ompOffloading_35_a1e179_foo_i7)

remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.

remark: Specialize parallel region that is only reached from a single target region to avoid spurious call edges and excessive register usage in other target regions. (parallel region ID: __ompOutlined__3_wrapper, kernel ID: __ompOffloading_35_a1e179_foo_i7)

remark: Target region containing the parallel region that is specialized. (parallel region ID: __ompOutlined__3_wrapper, kernel ID: __ompOffloading_35_a1e179_foo_i7)

remark: OpenMP GPU kernel __ompOffloading_35_a1e179_foo_i7
OpenMP Advisor and Runtime Information

- Development of the llvm-openmp-advisor tool (LLVM 12, current git)
- Optimization remark explanations, examples, FAQs, ... all gradually added to http://openmp.llvm.org/docs
- Environment variable LIBOMPTARGET_INFO for runtime library interactions

```bash
$ clang -O2 generic.c -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -o generic
$ env LIBOMPTARGET_INFO=1 ./generic
```

CUDA device 0 info: Device supports up to 65536 CUDA blocks and 1024 threads with a warp size of 32
CUDA device 0 info: Launching kernel __omp_offloading_fd02_c2a59832_main_l106 with 48 blocks and 128 threads in Generic mode
Design Goal

Allow modular OpenMP code without performance penalty

no need for manual low-level optimizations
void foo() {
    #pragma omp parallel
    bar();
}

void bar() {
    if (omp_in_parallel()) {
        ...
    } else {
        ...
    }
}
void foo() {
    #pragma omp parallel
    bar();
}

void bar() {
    if (omp_in_parallel()) {
        ...
    } else {
        ...
    }
}

Can be deleted if `omp_in_parallel()` is known* to return true, e.g., after inlining into `foo`.
Design Goal

Allow modular OpenMP code without performance penalty

no need for manual high-level optimizations
Parallel Region Merging Optimization

... #pragma omp parallel
{ Activate threads
  do_computation_x()
}
Barrier

#pragma omp parallel
{ Activate threads
  do_computation_y()
}
Barrier

... #pragma omp parallel
{ Activate threads
  do_computation_x()
  #pragma omp barrier
  do_computation_y()
}
Barrier

...
Parallel Region Merging Optimization

```
#pragma omp parallel
{
  Activate threads
  do_computation_x()
}
Barrier

do_sequential_work()

#pragma omp parallel
{
  Activate threads
  do_computation_y()
}
Barrier

#pragma omp parallel
{
  Activate threads
  do_computation_x()

  #pragma omp barrier

  #pragma omp master {
    do_sequential_work()
  }

  #pragma omp barrier

  do_computation_y()
}
Barrier
```

Only if unsafe to run in parallel
Design Goal

Allow modular OpenMP code without performance penalty

no need for manual high-level optimizations
void process_array(double * restrict a, unsigned size) {

    some_computation();

    #pragma omp target data map(a[0:size], size)
    #pragma omp target teams
    for (int i = 0; i < size; i++)
        compute(a[i]);
}
void process_array(double * restrict a, unsigned size) {

#pragma omp target data map(a[0:size], size) depend(out:transfer) nowait
some_computation();   // We ensure this computation does not modify *a nor size.

#pragma omp taskwait depend(in:transfer)
#pragma omp target data map(a[0:size], size)
#pragma omp target teams
for (int i = 0; i < size; i++)
    compute(a[i]);
}
void process_array(double * restrict a, unsigned size) {
    #pragma omp target data map(tofrom: a[0:size], size)
    #pragma omp target teams
    for (int i = 0; i < size; i++)
        first_transformation(a[i]);

    some_computation();

    #pragma omp target data map(tofrom: a[0:size], size)
    #pragma omp target teams
    for (int i = 0; i < size; i++)
        second_transformation(a[i]);
}
void process_array(double * restrict a, unsigned size) {
    #pragma omp target data map(to: a[0:size], size) // no need to send `a` nor `size` back from the device.
    #pragma omp target teams
    for (int i = 0; i < size; i++)
        first_transformation(a[i]);

    some_computation(); // we make sure this does not modify `a` nor `size`.

    #pragma omp target data map(from: a[0:size], size) // no need to send `a` nor `size` to the device.
    #pragma omp target teams
    for (int i = 0; i < size; i++)
        second_transformation(a[i]);
}
Design Goal

Optimize offloading code

perform host + accelerator optimizations
void foo() {
    int N = 1024;

    #pragma omp target
    *mem = N;
}

Heterogeneous LLVM-IR Module

```c
extern void device_func7(int);

void foo() {
    int N = 1024;
    if (!offload(device_func7, N)) {
        // host fallback
        *mem = N;
    }
}

void device_func7(int N) {
    *mem = N;
}
```

```c
host.c

extern void device_func7(int);

void foo() {
    int N = 1024;
    if (!offload(device_func7, N)) {
        // host fallback
        *mem = N;
    }
}

device.c

void device_func7(int N) {
    *mem = N;
}
```

Heterogeneous LLVM-IR Module

user_code_1.c
void foo() {
    int N = 1024;
    #pragma omp target
    *mem = N;
}

host.c
extern void device_func7(int);

void foo() {
    int N = 1024;
    if (!offload(device_func7, 1024)) {
        // host fallback
        *mem = 1024;
    }
}

device.c
void device_func7(int N) {
    *mem = N;
}

extern void device_func7(int);

void foo() {
    int N = 1024;
    if (!offload(device_func7, 1024)) {
        // host fallback
        *mem = 1024;
    }
}

void device_func7(int N) {
    *mem = N;
}

Heterogeneous LLVM-IR Module

```c
// user_code_1.c

void foo() {
    int N = 1024;
    #pragma omp target
    *mem = N;
}

// heterogeneous.c

__attribute__((callback(Func, ...)))
int offload(void (*)(...) Func, ...);

target 0 void foo() {
    int N = 1024;
    if (!offload(device_func7, N)) {
        // host fallback
        *mem = N;
    }
}

target 1 void device_func7(int N) {
    *mem = N;
}
```


Heterogeneous LLVM-IR Module

```c
void foo() {
    int N = 1024;
#pragma omp target
*mem = N;
}

user_code_1.c

int offload(void (*)(...) Func, ...);

target 0 void foo() {
    int N = 1024;
    if (!offload(device_func7, N)) {
        // host fallback
        *mem = 1024;
    }
}

target 1 void device_func7(int N) {
    *mem = 1024;
}

* callback attribute: https://bit.ly/3z68ds
```
Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.
openmp.org  OpenMP API specs, forum, reference guides, and more

link.openmp.org/sc20  Videos and PDFs of OpenMP SC’20 presentations