Targeting GPUs with OpenMP 4.5 Device Directives

James Beyer, NVIDIA
Jeff Larkin, NVIDIA
AGENDA

OpenMP Background
Step by Step Case Study
  Parallelize on CPU
  Offload to GPU
  Team Up
  Increase Parallelism
  Improve Scheduling
Additional Experiments
Conclusions
Motivation

Multiple compilers are in development to support OpenMP offloading to NVIDIA GPUs.

Articles and blog posts are being written by early adopters trying OpenMP on NVIDIA GPUs, most of them have gotten it wrong.

If you want to try OpenMP offloading to NVIDIA GPUs, we want you to know what to expect and how to get reasonable performance.
A Brief History of OpenMP

1996 - Architecture Review Board (ARB) formed by several vendors implementing their own directives for Shared Memory Parallelism (SMP).

1997 - 1.0 was released for C/C++ and Fortran with support for parallelizing loops across threads.


2005 - Version 2.5 released, combining both specs into one.

2008 - Version 3.0 released, added support for tasking

2011 - Version 3.1 release, improved support for tasking

2013 - Version 4.0 released, added support for offloading (and more)

2015 - Version 4.5 released, improved support for offloading targets (and more)
OpenMP In Clang

Multi-vendor effort to implement OpenMP in Clang (including offloading)

Current status - interesting

How to get it - https://www.ibm.com/developerworks/community/blogs/8e0d7b52-b996-424b-bb33-345205594e0d?lang=en
OpenMP In Clang
How to get it, our way

Step one - make sure you have: gcc, cmake, python and cuda installed and updated

Step two - Look at
  http://llvm.org/docs/GettingStarted.html
  https://www.ibm.com/developerworks/community/blogs/8e0d7b52-b996-424b-bb33-345205594e0d?lang=en

Step three -
git clone https://github.com/clang-ykt/llvm_trunk.git
cd llvm_trunk/tools
git clone https://github.com/clang-ykt/clang_trunk.git clang
cd ../projects
git clone https://github.com/clang-ykt/openmp.git
OpenMP In Clang
How to build it

cd ..
mkdir build

cd build

cmake -DCMAKE_BUILD_TYPE=DEBUG|RELEASE|MinSizeRel \ 
-DCMAKE_INSTALL_PREFIX="<where you want it>" \ 
-DLLVM_ENABLE_ASSERTIONS=ON \ 
-DLLVM_ENABLE_BACKTRACES=ON \ 
-DLLVM_ENABLE_WERROR=OFF \ 
-DBUILD_SHARED_LIBS=OFF \ 
-DLLVM_ENABLE_RTTI=ON \ 
-DCMAKE_C_COMPILER="GCC you want used" \ 
-DCMAKE_CXX_COMPILER="G++ you want used" \ 
-G "Unix Makefiles" \ !there are other options, I like this one

..

make [-j#]
make install
OpenMP In Clang

How to use it

export LIBOMP_LIB=<llvm-install-lib>
export OMPTARGET_LIBS=$LIBOMP_LIB
export LIBRARY_PATH=$OMPTARGET_LIBS
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OMPTARGET_LIBS
export PATH=$PATH:<llvm_install-bin>
clang -O3 -fopenmp=libomp -omptargets=nvptx64sm_35-nvidia-linux ...
Case Study: Jacobi Iteration
Example: Jacobi Iteration

Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

Common, useful algorithm

Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$

$$A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}$$
while ( err > tol && iter < iter_max ){
    err=0.0;
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
Parallelize on the CPU
OpenMP Worksharing

PARALLEL Directive

Spawns a team of threads

Execution continues redundantly on all threads of the team.

All threads join at the end and the master thread continues execution.
OpenMP Worksharing

FOR/DO (Loop) Directive

Divides ("workshares") the iterations of the next loop across the threads in the team

How the iterations are divided is determined by a schedule.
while ( error > tol && iter < iter_max )
{
    error = 0.0;

    #pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++ )
    {
        for( int i = 1; i < m-1; i++ )
        {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                               + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma omp parallel for
    for( int j = 1; j < n-1; j++ )
    {
        for( int i = 1; i < m-1; i++ )
        {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
while (error > tol && iter < iter_max)
{
    error = 0.0;

    #pragma omp parallel
    {
        #pragma omp for reduction(max:error)
        for(int j = 1; j < n-1; j++)
        {
            for(int i = 1; i < m-1; i++)
            {
                error = fmax(error, fabs(Anew[j][i] - A[j][i]));
            }
        }
    }
    #pragma omp barrier
    #pragma omp for
    for(int j = 1; j < n-1; j++)
    {
        for(int i = 1; i < m-1; i++)
        {
            A[j][i] = Anew[j][i];
        }
    }
    }
    
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
CPU-Parallelism

```c
while ( error > tol && iter < iter_max )
{
    error = 0.0;

    #pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++ ) {
        #pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] 
                                  + A[j-1][i] + A[j+1][i] );
            error = fmax( error, fabs(Anew[j][i] - A[j][i]) );
        }
    }

    #pragma omp parallel for
    for( int j = 1; j < n-1; j++ ) {
        #pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Some compilers want a SIMD directive to *simdize* on CPUS.
CPU Scaling (Smaller is Better)

Execution Time (seconds)

Intel Xeon E5-2690 v2 @ 3.00GHz
Targeting the GPU
OpenMP Offloading

TARGET Directive

Offloads execution and associated data from the CPU to the GPU

- The target device owns the data, accesses by the CPU during the execution of the target region are forbidden.
- Data used within the region may be implicitly or explicitly mapped to the device.
- All of OpenMP is allowed within target regions, but only a subset will run well on GPUs.
Target the GPU

while ( error > tol && iter < iter_max )
{
    error = 0.0;
    #pragma omp target
    {
        #pragma omp parallel for reduction(max:error)
        for( int j = 1; j < n-1; j++ )
        {
            for( int i = 1; i < m-1; i++ )
            {
                error = fmax( error, fabs(Anew[j][i] - A[j][i]) );
            }
        }
        #pragma omp parallel for
        for( int j = 1; j < n-1; j++ )
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
    }
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
Target the GPU

```c
while ( error > tol && iter < iter_max )
{
    error = 0.0;

    #pragma omp target map(alloc:Anew[:n+2][:m+2]) map(tofrom:A[:n+2][:m+2])
    {
        #pragma omp parallel for reduction(max:error)
        for( int j = 1; j < n-1; j++ ) {
            for( int i = 1; i < m-1; i++ ) {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] 
                                 + A[j-1][i] + A[j+1][i] );
                error = fmax( error, fabs(Anew[j][i] - A[j][i]) );
            }
        }
    }

    #pragma omp parallel for
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Moves this region of code to the GPU and explicitly maps data.
Execution Time (Smaller is Better)

Original: 1.00X
CPU Threaded: 5.12X
GPU Threaded: 0.12X

NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz
GPU Architecture Basics

GPUs are composed of 1 or more independent parts, known as *Streaming Multiprocessors* ("SMs")

*Threads* are organized into *threadblocks*.

Threads within the same threadblock run on an SM and can synchronize.

Threads in different threadblocks (even if they’re on the same SM) cannot synchronize.
Teaming Up
OpenMP Teams

TEAMS Directive

To better utilize the GPU resources, use many thread teams via the TEAMS directive.

- Spawns 1 or more thread teams with the same number of threads
- Execution continues on the master threads of each team (redundantly)
- No synchronization between teams
OpenMP Teams

DISTRIBUTE Directive

Distributes the iterations of the next loop to the master threads of the teams.

- Iterations are distributed statically.
- There’s no guarantees about the order teams will execute.
- No guarantee that all teams will execute simultaneously
- Does not generate parallelism/worksharing within the thread teams.
OpenMP Data Offloading

TARGET DATA Directive

Offloads data from the CPU to the GPU, but not execution

- The target device owns the data, accesses by the CPU during the execution of contained target regions are forbidden.

- Useful for sharing data between TARGET regions

- NOTE: A TARGET region is a TARGET DATA region.
Teaming Up

```c
#pragma omp target data map(alloc:Anew) map(A)
while ( error > tol && iter < iter_max )
{
    error = 0.0;

#pragma omp target teams distribute parallel for reduction(max:error)
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
                              + A[j-1][i] + A[j+1][i] ) ;
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
    }
}

#pragma omp target teams distribute parallel for
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}

if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
iter++;
}
```

Explicitly maps arrays for the entire while loop.

- Spawns thread teams
- Distributes iterations to those teams
- Workshares within those teams.
Execution Time (Smaller is Better)

NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz
Increasing Parallelism
Increasing Parallelism

Currently both our distributed and workshared parallelism comes from the same loop.

- We could move the PARALLEL to the inner loop
- We could collapse them together

The COLLAPSE(N) clause

- Turns the next N loops into one, linearized loop.
- This will give us more parallelism to distribute, if we so choose.
Splitting Teams & Parallel

```c
#pragma omp target teams distribute
for( int j = 1; j < n-1; j++)
{
    #pragma omp parallel for reduction(max:error)
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                             + A[j-1][i] + A[j+1][i] );
        error = fmax( error, fabs(Anew[j][i] - A[j][i]) );
    }
}
```

Distribute the “j” loop over teams.

```c
#pragma omp target teams distribute
for( int j = 1; j < n-1; j++)
{
    #pragma omp parallel for
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}
```

Workshare the “i” loop over threads.
#pragma omp target teams distribute parallel for reduction(max:error) collapse(2)
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
    }
}

#pragma omp target teams distribute parallel for collapse(2)
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}
Execution Time (Smaller is Better)

- Original: 1.00X
- CPU Threaded: 5.12X
- GPU Threaded: 0.12X
- GPU Teams: 1.01X
- GPU Split: 2.47X
- GPU Collapse: 0.96X

NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz
Improve Loop Scheduling
Improve Loop Scheduling

Most OpenMP compilers will apply a static schedule to workshared loops, assigning iterations in $N / \text{num\_threads}$ chunks.

- Each thread will execute contiguous loop iterations, which is very cache & SIMD friendly
- This is great on CPUs, but bad on GPUs

The SCHEDULE() clause can be used to adjust how loop iterations are scheduled.
Effects of Scheduling

```
!$OMP PARALLEL FOR SCHEDULE(STATIC)
Thread 0   0 - (n/2-1)  
Thread 1   (n/2) - n-1
```

- Cache and vector friendly

```
!$OMP PARALLEL FOR SCHEDULE(STATIC,1)*
Thread 0   0, 2, 4, ..., n-2
Thread 1   1, 3, 5, ..., n-1
```

- Memory coalescing friendly

*There’s no reason a compiler couldn’t do this for you.
Improved Schedule (Split)

```c
#pragma omp target teams distribute
for( int j = 1; j < n-1; j++)
{
#pragma omp parallel for reduction(max:error) schedule(static,1)
  for( int i = 1; i < m-1; i++ )
  {
    Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] 
      + A[j-1][i] + A[j+1][i] );
    error = fmax( error, fabs(Anew[j][i] - A[j][i]) );
  }
}

#pragma omp target teams distribute
for( int j = 1; j < n-1; j++)
{
#pragma omp parallel for schedule(static,1)
  for( int i = 1; i < m-1; i++ )
  {
    A[j][i] = Anew[j][i];
  }
}
```

Assign adjacent threads adjacent loop iterations.
Improved Schedule (Collapse)

```c
#pragma omp target teams distribute parallel for \
   reduction(max:error) collapse(2) schedule(static,1)
   for( int j = 1; j < n-1; j++)
   {
      for( int i = 1; i < m-1; i++ )
      {
         
         Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] 
            + A[j-1][i] + A[j+1][i]); 
         
         error = fmax(error, fabs(Anew[j][i] - A[j][i]));
      }
   }
```

Assign adjacent threads adjacent loop iterations.
Execution Time (Smaller is Better)

Original: 1.00X
CPU Threaded: 5.12X
GPU Threaded: 893 (0.12X)
GPU Teams: 1.01X
GPU Split: 2.47X
GPU Collapse: 0.96X
GPU Split Sched: 4.00X
GPU Collapse Sched: 17.42X

NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz
#pragma omp \
#ifdef GPU
target teams distribute \
#endif
parallel for reduction(max:error) \
#ifdef GPU
collapse(2) schedule(static,1)
#endif
for( int j = 1; j < n-1; j++)
{
    for( int i = 1; i < m-1; i++ )
    {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] 
                             + A[j-1][i] + A[j+1][i]);
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
    }
}
usegpu = 1;
#pragma omp target teams distribute parallel for reduction(max:error) \
#ifdef GPU
collapse(2) schedule(static,1) \
#endif
if(target:usegpu)
    for( int j = 1; j < n-1; j++)
    {
        for( int i = 1; i < m-1; i++ )
        {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] \
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }
Additional Experiments
Increase the Number of Teams

By default, CLANG will poll the number of SMs on your GPU and run that many teams of 1024 threads.

This is not always ideal, so we tried increasing the number of teams using the `num_teams` clause.

<table>
<thead>
<tr>
<th>Test</th>
<th>SMs</th>
<th>2*SMs</th>
<th>4*SMs</th>
<th>8*SMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.00X</td>
<td>1.00X</td>
<td>1.00X</td>
<td>1.00X</td>
</tr>
<tr>
<td>B</td>
<td>1.00X</td>
<td>1.02X</td>
<td>1.16X</td>
<td>1.09X</td>
</tr>
<tr>
<td>C</td>
<td>1.00X</td>
<td>0.87X</td>
<td>0.94X</td>
<td>0.96X</td>
</tr>
<tr>
<td>D</td>
<td>1.00X</td>
<td>1.00X</td>
<td>1.00X</td>
<td>0.99X</td>
</tr>
</tbody>
</table>
Decreased Threads per Team

CLANG always generate CUDA threadblocks of 1024 threads, even when the num_threads clause is used.

This number is frequently not ideal, but setting num_threads does not reduce the threadblock size.

Ideally we’d like to use num_threads and num_teams to generate more, smaller threadblocks.

We suspect the best performance would be collapsing, reducing the threads per team, and then using the remaining iterations to generate many teams, but are unable to do this experiment.
Scalar Copy Overhead

In OpenMP 4.0 scalars are implicitly mapped "tofrom", resulting in very high overhead. Application impact: ~10%.

OpenMP 4.5 remedied this by making the default behavior of scalars "firstprivate".

Note: In the meantime, some of this overhead can be mitigated by explicitly mapping your scalars "to".
Conclusions

It is now possible to use OpenMP to program for GPUs, but the software is still very immature.

OpenMP for a GPU will not look like OpenMP for a CPU.

Performance will vary significantly depending on the exact directives you use. (149X in our example code)