OPENACC ONLINE COURSE

Lecture 2: GPU Programming with OpenACC

John Urbanic, Pittsburgh Supercomputing Center
October 26, 2017
COURSE SYLLABUS:

October 19: Introduction to OpenACC
October 26: GPU Programming with OpenACC
November 2: Optimizing and Best Practices for OpenACC
HOMEWORK SOLUTION
EXERCISE 1: C SOLUTION

while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
    }
}

dt = 0.0; // reset largest temperature change

#pragma acc kernels
{ 
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }
}

if((iteration % 100) == 0) {
    track_progress(iteration);
}

iteration++;
do while (dt > max_temp_error .and. iteration <= max_iterations)

!$acc kernels
do j=1,columns
  do i=1,rows
    temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                  temperature_last(i,j+1)+temperature_last(i,j-1))
  enddo
enddo
!$acc end kernels

dt=0.0

!$acc kernels
do j=1,columns
  do i=1,rows
    dt = max(abs(temperature(i,j) - temperature_last(i,j)), dt)
    temperature_last(i,j) = temperature(i,j)
  enddo
enddo
!$acc end kernels

if( mod(iteration,100).eq.0 ) then
  call track_progress(temperature, iteration)
endif

iteration = iteration+1
enddo
## CPU PERFORMANCE

3372 steps to convergence

<table>
<thead>
<tr>
<th>Execution</th>
<th>Problem Size</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000x1000 (Lab1)</td>
<td>Time (s)</td>
<td>Speedup</td>
<td>Time (s)</td>
<td>Speedup</td>
</tr>
<tr>
<td>CPU Serial</td>
<td></td>
<td>6.2</td>
<td>--</td>
<td>40.3</td>
<td>--</td>
</tr>
<tr>
<td>OpenACC 2 CPU Cores</td>
<td></td>
<td>3.0</td>
<td>2.07</td>
<td>26.2</td>
<td>1.54</td>
</tr>
<tr>
<td>OpenACC 4 CPU Cores</td>
<td></td>
<td>1.5</td>
<td>4.13</td>
<td>21.3</td>
<td>1.89</td>
</tr>
<tr>
<td>2000x2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MOVING TO THE GPU

Change the target accelerator flag from “-ta=multicore” to “-ta=tesla”

% pgcc -acc -ta=tesla -Minfo=accel -DMAX_ITER=4000 laplace_kernels.c -o laplace.out

main:

65, Generating implicit copyin(Temperature_last[:][:])
   Generating implicit copyout(Temperature[1:1000][1:1000])
66, Loop is parallelizable
67, Loop is parallelizable
   Accelerator kernel generated
   Generating Tesla code
   66, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
   67, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
76, Generating implicit copyin(Temperature[1:1000][1:1000])
   Generating implicit copy(Temperature_last[1:1000][1:1000])
77, Loop is parallelizable
78, Loop is parallelizable
   Accelerator kernel generated
   Generating Tesla code
   77, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
   78, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
79, Generating implicit reduction(max:dt)
RUNNING ON THE GPU

% ./laplace.out
...

---------- Iteration number: 3100 ----------
[995,995]: 97.58  [996,996]: 98.18  [997,997]: 98.71  [998,998]: 99.17  [999,999]: 99.55
[1000,1000]: 99.86

---------- Iteration number: 3200 ----------
[995,995]: 97.62  [996,996]: 98.21  [997,997]: 98.73  [998,998]: 99.18  [999,999]: 99.56
[1000,1000]: 99.86

---------- Iteration number: 3300 ----------
[995,995]: 97.66  [996,996]: 98.24  [997,997]: 98.75  [998,998]: 99.19  [999,999]: 99.56
[1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 40.462415 seconds.

Our time slows down by almost 7x from the original serial version of the code!
WHAT WENT WRONG?

% pgprof –cpu-profiling-mode top-down ./laplace.out
....

==455== Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>57.41%</td>
<td>13.2519s</td>
<td>13488</td>
<td>982.49us</td>
<td>959ns</td>
<td>1.3721ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>36.08%</td>
<td>8.32932s</td>
<td>10116</td>
<td>823.38us</td>
<td>2.6230us</td>
<td>1.5129ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>3.42%</td>
<td>788.42ms</td>
<td>3372</td>
<td>233.81us</td>
<td>231.79us</td>
<td>235.79us</td>
<td>main_80_gpu</td>
</tr>
<tr>
<td>2.80%</td>
<td>646.58ms</td>
<td>3372</td>
<td>191.75us</td>
<td>189.69us</td>
<td>194.04us</td>
<td>main_69_gpu</td>
</tr>
<tr>
<td>0.29%</td>
<td>66.916ms</td>
<td>3372</td>
<td>19.844us</td>
<td>19.519us</td>
<td>20.223us</td>
<td>main_81_gpu_red</td>
</tr>
</tbody>
</table>

======== CPU profiling result (top down):

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.15%</td>
<td>16.3609s</td>
<td>__pgi_uacc_dataexitdone</td>
</tr>
<tr>
<td>9.11%</td>
<td>16.2809s</td>
<td>__pgi_uacc_dataonb</td>
</tr>
</tbody>
</table>

Including the copy, there is an additional CPU overhead needed to create and delete the device data.

13.2 seconds copying data from host to device
8.3 seconds copying data from device to host
ARCHITECTURE
BASIC CONCEPT

SIMPLIFIED, BUT SADLY TRUE
Excessive Data Transfers

MULTIPLE TIMES EACH ITERATION
EXCESSIVE DATA TRANSFERS

while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + ...
    }
}

dt = 0.0;

#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + ...
    }
}
DATA MANAGEMENT

THE FIRST, MOST IMPORTANT, AND POSSIBLY ONLY OPENACC OPTIMIZATION
DATA CONSTRUCT SYNTAX AND SCOPE

C

```c
#pragma acc data [clause ...]
{
    structured block
}
```

Fortran

```fortran
!$acc data [clause ...]
    structured block
!$acc end data
```
DATA CLAUSES

**copy( list )**
Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

**Principal use:** For many important data structures in your code, this is a logical default to input, modify and return the data.

**copyin( list )**
Allocates memory on GPU and copies data from host to GPU when entering region.

**Principal use:** Think of this like an array that you would use as just an input to a subroutine.

**copyout( list )**
Allocates memory on GPU and copies data to the host when exiting region.

**Principal use:** A result that isn’t overwriting the input data structure.

**create( list )**
Allocates memory on GPU but does not copy.

**OpenACC**
**Principal use:** Temporary arrays.
ARRAY SHAPING

• Compilers sometimes cannot determine the size of arrays, so we must specify explicitly using data clauses with an array “shape”. The compiler will let you know if you need to do this. Sometimes, you will want to for your own efficiency reasons.

C

#pragma acc data copyin(a[0:count]), copyout(b[s/4:3*s/4])

Fortran

!$acc data copyin(a(1:end)), copyout(b(s/4:3*s/4))

• Fortran uses `start:end` and C uses `start:count`

• Data clauses can be used on data, kernels or parallel
COMPILER WILL (INCREASINGLY) OFTEN MAKE A GOOD GUESS…

C Code

```c
int main(int argc, char *argv[]) {
    int i;
    double A[2000], B[1000], C[1000];
    #pragma acc kernels
    for (i=0; i<1000; i++){
        A[i] = 4 * i;
        B[i] = B[i] + 2;
        C[i] = A[i] + 2 * B[i];
    }
}
```

Compiler Output

```
pgcc -acc -Minfo=accel loops.c
main:
  6, Generating implicit copy(B[:])
  Generating implicit copyout(C[:])
  Generating implicit copyout(A[:1000])
  7, Loop is parallelizable
  Accelerator kernel generated
  Generating Tesla code
  7, #pragma acc loop gang, vector(128)
```
DATA REGIONS HAVE REAL CONSEQUENCES

### Simplest Kernel

```c
int main(int argc, char** argv){
    float A[1000];

    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++ ){
        A[iter] = 1.0;
    }
    A[10] = 2.0;
}
```

**Output:**

```
A[10] = 2.0
```

### With Global Data Region

```c
int main(int argc, char** argv){
    float A[1000];

    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++ ){
        A[iter] = 1.0;
    }
    #pragma acc data copy(A)
    {
        A[10] = 2.0;
    }
}
```

**Output:**

```
A[10] = 1.0
```
int main(int argc, char** argv) {
    float A[1000];
    #pragma acc data copy(A)
    {
        #pragma acc kernels
        for (int iter = 1; iter < 1000; iter++) {
            A[iter] = 1.0;
        }
        A[10] = 2.0;
    }
}

Output:
A[10] = 1.0
DATA MOVEMENT DECISIONS

• Much like loop data dependencies, sometime the compiler needs your human intelligence to make high-level decisions about data movement. Otherwise, it must remain conservative – sometimes at great cost.

• You must think about when data truly needs to migrate, and see if that is better than the default.

• Besides the scope-based data clauses, there are OpenACC options to let us manage data movement more intensely or asynchronously. We could manage the above behavior with the `update` construct:

```c
#pragma acc update [self(), device(), …]
```

```fortran
!$acc update [self(), device(), …]
```

Ex: `#pragma acc update self(Temp_array)  //Get host a copy from device`
ANALYZING / PROFILING
OPENACC DEVELOPMENT CYCLE

- **Analyze** your code, and predict where potential parallelism can be uncovered. Use profiler to help understand what is happening in the code and where parallelism may exist.

- **Parallelize** your code, starting with the most time consuming parts. Focus on maintaining correct results from your program.

- **Optimize** your code, focusing on maximizing performance. Performance may not increase all-at-once during early parallelization.
PROFILING GPU CODE

Using PGPROF to profile GPU code

- This is the view of PGPROF after profiling a GPU application
- The first obvious difference when compared to our command line profile: a visual timeline of events
PROFILING GPU CODE
Using PGPROF to profile GPU code

- **MemCpy(HtoD):** This includes data transfers from the Host to the Device (CPU to GPU)
- **MemCpy(DtoH):** These are data transfers from the Device to the Host (GPU to CPU)
- **Compute:** These are our computational functions.
INSPECTING THE PGPROF TIMELINE

- Zooming in gives us a better view of the timeline. It looks like our program is spending a significant amount of time transferring data between the host and device. We also see that the compute regions are very small, with a lot of distance between them.

* Note we use a 1 kernel solution (available from the Resources console)
PROFILE AFTER ADDING A DATA REGION

After adding a data region, we see lots of compute and data movement only when we update.
EXERCISE 2
Use acc data to minimize transfers

Q: What speedup can you get with data + kernels directives?

Start with your Exercise 1 solution (or ours).

Add data directives where it helps.

• Think: when should I move data between host and GPU? Think how you would do it by hand, then determine which data clauses will implement that plan.

• Hint: you may find it helpful to ignore the output at first and just concentrate on getting the solution to converge quickly (at 3372 steps). Then worry about updating the printout (hint, hint).
EXERCISES: GENERAL INSTRUCTIONS

• The exercise is found at https://nvlabs.qwiklab.com

✅ Create an account or log in (use the same email you registered for the course with)

✅ Open class: OpenACC Course Oct 2017
TODAY WE DISCUSSED

• What OpenACC data directives are
• Why they are important for GPU use
• How to spot these, and many other types of performance hotspots, with a Profiler

Questions? Email openacc@nvidia.com
OPENACC RESOURCES

Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

Resources
https://www.openacc.org/resources

Success Stories
https://www.openacc.org/success-stories

Compilers and Tools
https://www.openacc.org/tools

Events
https://www.openacc.org/events

FREE Compilers
PGI
Community Edition

https://www.openacc.org/community#slack
OPENACC FOR EVERYONE
New PGI Community Edition Now Available

<table>
<thead>
<tr>
<th>PROGRAMMING MODELS</th>
<th>PLATFORMS</th>
<th>UPDATES</th>
<th>SUPPORT</th>
<th>LICENSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC, CUDA Fortran, OpenMP, C/C++/Fortran Compilers and Tools</td>
<td>X86, OpenPOWER, NVIDIA GPU</td>
<td>1-2 times a year</td>
<td>User Forums</td>
<td>Annual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6-9 times a year</td>
<td>PGI Support</td>
<td>Perpetual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6-9 times a year</td>
<td>PGI Professional Services</td>
<td>Volume/Site</td>
</tr>
</tbody>
</table>

FREE

DOWNLOAD
https://www.pgroup.com/products/community.htm
NEW BOOK: OPENACC FOR PROGRAMMERS

Edited by: By Sunita Chandrasekaran, Guido Juckeland

- Discover how OpenACC makes scalable parallel programming easier and more practical
- Get productive with OpenACC code editors, compilers, debuggers, and performance analysis tools
- Build your first real-world OpenACC programs
- Overcome common performance, portability, and interoperability challenges
- Efficiently distribute tasks across multiple processors

Use code OPENACC during checkout for 35% off

*Discount taken off list price. Offer only good at informit.com
XSEDE Monthly Workshop Series

- NSF Funded for the national scientific community
- Wide Area Classroom (WAC) Format
- Next OpenACC event is November 7th
- Classroom based with 25 remote sites per event
- 50 events, 8000 students and counting

Contact Tom Maiden, tmaiden@psc.edu
COURSE SYLLABUS:

October 19: Introduction to OpenACC
October 26: GPU Programming with OpenACC
November 2: Optimizing and Best Practices for OpenACC

Questions? Email openacc@nvidia.com