Profiling and Parallelizing with the OpenACC Toolkit

OpenACC Course: Lecture 2 - November 2, 2016

NVIDIA
Introduction to Accelerated Computing
Identifying Available Parallelism
Expressing Parallelism with OpenACC
Next Steps and Homework
Introduction to Accelerated Computing
Accelerated Computing
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Parallel Tasks
Accelerated Computing
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for Serial Tasks

CPU Strengths
- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses
- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt
Accelerated Computing
10x Performance & 5x Energy Efficiency for HPC

GPU Accelerator
Optimized for Parallel Tasks

GPU Strengths

- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance
Speed v. Throughput

Which is better depends on your needs...

*Images from Wikimedia Commons via Creative Commons
CPU and GPU have distinct memories
• CPU generally larger and slower
• GPU generally smaller and faster

CPU and GPU communicate via PCIe
• Data must be copied between these memories over PCIe
• PCIe Bandwidth is much lower than either memories
CUDA Unified Memory
Simplified Developer Effort

Without Unified Memory

With Unified Memory

System Memory

GPU Memory

Unified Memory

Sometimes referred to as “managed memory.”
Identify Available Parallelism

Optimize Loop Performance

Express Data Movement

Express Parallelism
Case Study

For lectures and labs 2 & 3 we will use an example code that solves a conjugate gradient problem.

- This code is available in C and Fortran, but only C will be shown in the lectures.
- We will be demonstrating OpenACC concepts using this code in the next 2 lectures.
- You will be profiling and accelerating this code in the next 2 labs.
- In addition to the labs, the code is available at https://github.com/NVIDIA-OpenACC-Course/nvidia-openacc-course-sources/tree/master/labs
Identifying Available Parallelism
NVIDIA NVPROF Profiler

NVPROF is a command-line profiler provided in the OpenACC and CUDA Toolkits

- Basic CPU Profiling (New in OpenACC Toolkit & CUDA 7.5)
- GPU Profiling
  - High-level usage statistics
  - Timeline Collection
  - Analysis Metrics
- Used behind-the-scenes by NVIDIA Visual Profiler (nvvp)
NVPROF CPU Profiling

$ nvprof --cpu-profiling on --cpu-profiling-mode top-down ./cg
Rows: 8120601, nnz: 218535025
Iteration: 0, Tolerance: 4.0067e+08
Iteration: 10, Tolerance: 1.8772e+07
Iteration: 20, Tolerance: 6.4359e+05
Iteration: 30, Tolerance: 2.3202e+04
Iteration: 40, Tolerance: 8.3565e+02
Iteration: 50, Tolerance: 3.0039e+01
Iteration: 60, Tolerance: 1.0764e+00
Iteration: 70, Tolerance: 3.8360e-02
Iteration: 80, Tolerance: 1.3515e-03
Iteration: 90, Tolerance: 4.6209e-05
Total Iterations: 100 Total Time: 33.926116s

======== CPU profiling result (top down):
99.89% main
| 83.22% matvec(matrix const &, vector const &, vector const &)
| 10.41% waxpby(double, vector const &, double, vector const &, vector const &)
| 3.81% dot(vector const &, vector const &)
| 2.42% allocate_3d_poission_matrix(matrix &, int)
| 0.03% free_matrix(matrix &)
| 0.03% munmap
0.11% __c_mset8

======== Data collected at 100Hz frequency
GPROF Profiler

Portable command-line profiler available from GCC.

When used with PGI or GCC, the following steps are required:

1. Add the -pg compiler flag to instrument your code
2. Run the executable (it will produce gmon.out)
3. Run gprof ./executable to analyze the collected data
### GPROF Output for Case Study

Add `-pg` to compiler flags, rebuild & rerun, use gprof

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>self</th>
<th>cumulative</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>77.32</td>
<td>24.82</td>
<td>24.82</td>
<td>101</td>
<td>245.74</td>
<td>245.74</td>
<td>matvec(matrix const&amp;, vector const&amp;, vector const&amp;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.36</td>
<td>29.75</td>
<td>4.93</td>
<td>302</td>
<td>16.32</td>
<td>16.32</td>
<td>waxpby(double, vector const&amp;, double, vector const&amp;, vector const&amp;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.24</td>
<td>31.11</td>
<td>1.36</td>
<td>200</td>
<td>6.80</td>
<td>6.80</td>
<td>dot(vector const&amp;, vector const&amp;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.08</td>
<td>32.10</td>
<td>0.99</td>
<td>1</td>
<td>990.00</td>
<td>990.00</td>
<td>allocate_3d_poission_matrix(matrix&amp;, int)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>32.10</td>
<td>0.00</td>
<td>5</td>
<td>0.00</td>
<td>0.00</td>
<td>allocate_vector(vector&amp;, unsigned int)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>32.10</td>
<td>0.00</td>
<td>4</td>
<td>0.00</td>
<td>0.00</td>
<td>free_vector(vector&amp;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>32.10</td>
<td>0.00</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>initialize_vector(vector&amp;, double)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>32.10</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>free_matrix(matrix&amp;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PGI Compiler Feedback

Understanding what the compiler does with your code is critical to understanding the profile.

PGI -Minfo flag, options we’ll use:

- accel - Print compiler operations related to the accelerator
- all - Print (nearly) all compiler output
- intensity - Print loop intensity information
- ccff - Add information to the object files for use by tools
Compiler Feedback for Case Study

Add `-Minfo=all, intensity` to compiler flags and rebuild

```
$ pgc++ -fast -Minfo=all, intensity main.cpp -o cg
waxpby(double, const vector &, double, const vector &, const vector &):
  5, include "vector_functions.h"
    24, Intensity = 1.00
    Generated 4 alternate versions of the loop
    Generated vector and scalar versions of the loop; pointer conflict tests determine which is executed
    Generated 2 prefetch instructions for the loop
matvec(const matrix &, const vector &, const vector &):
  7, include "matrix_functions.h"
    14, Intensity = (num_rows*((row_end-row_start)*2))/(num_rows+(num_rows+(num_rows+(row_end-row_start)+(row_end-row_start))))
    18, Intensity = 1.00
    Unrolled inner loop 4 times
    Generated 2 prefetch instructions for the loop
```
Computational Intensity

Computational Intensity of a loop is a measure of how much work is being done compared to memory operations.

\[ \text{Computation Intensity} = \frac{\text{Compute Operations}}{\text{Memory Operations}} \]

Computational Intensity of 1.0 or greater is often a clue that something might run well on a GPU.
Analyzing the Code: Matvec

for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    for(int j=row_start; j<row_end; j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
Analyzing the Code: Waxpy and Dot

```c
for(int i=0;i<n;i++) {
    wcoefs[i] =
        alpha*xcoefs[i] +
        beta*ycoefs[i];
}
```

```c
for(int i=0;i<n;i++) {
    sum+=xcoefs[i]*ycoefs[i];
}
```

Look for data dependencies:
- Does one loop iteration affect other loop iterations?
- Do loop iterations read from and write to different places in the same array?
Expressing Parallelism
OpenACC kernels Directive

Identifies a region of code where I think the compiler can turn *loops* into *kernels*

Syntax:

```c
#pragma acc kernels
{
  for(int i=0; i<N; i++)
  {
    x[i] = 1.0;
    y[i] = 2.0;
  }

  for(int i=0; i<N; i++)
  {
    y[i] = a*x[i] + y[i];
  }
}
```

The compiler identifies 2 parallel loops and generates 2 kernels.
OpenACC kernels Directive (Fortran)

Identifies a region of code where I think the compiler can turn loops into kernels

```fortran
!$acc kernels
  do i=1,N
    x(i) = 1.0
    y(i) = 2.0  
  end do

  y(:) = a*x(:) + y(:)

!$acc end kernels
```

The compiler identifies 2 parallel loops and generates 2 kernels.
Loops vs. Kernels

for (int i = 0; i < 16384; i++)
{
    C[i] = A[i] + B[i];
}

function loopBody(A, B, C, i)
{
    C[i] = A[i] + B[i];
}
Loops vs. Kernels

for (int i = 0; i < 16384; i++)
{
    C[i] = A[i] + B[i];
}

function loopBody(A, B, C, i)
{
    C[i] = A[i] + B[i];
}

Calculate 0 - 16383 in order.
Loops vs. Kernels

```c
for (int i = 0; i < 16384; i++)
{
    C[i] = A[i] + B[i];
}
```

```c
function loopBody(A, B, C, i)
{
    C[i] = A[i] + B[i];
}
```

Calculate 0 to 16383 in order.
Loops vs. Kernels

```c
for (int i = 0; i < 16384; i++)
{
    C[i] = A[i] + B[i];
}
```

```c
function loopBody(A, B, C, i)
{
    C[i] = A[i] + B[i];
}
```

Calculate 0 -16383 in order.
The Kernels Directive

Identifies a region of code where I think the compiler can turn loops into kernels

```c
#pragma acc kernels
{
for (int i = 0; i < 16384; i++)
{
    C[i] = A[i] + B[i];
}
}
```

The Compiler will...

1. Analyze the code to determine if it contains parallelism
2. Identify data that needs to be moved to/from the GPU memory
3. Generate kernels
4. Run on the GPU
Parallelizing the Code: Matvec

```c
#pragma acc kernels
{
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offets[i];
    int row_end=row_offsets[i+1];
    for(int j=row_start;
        j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
}
```

Let’s tell the compiler where we think it can turn loops into kernels.

Don’t worry about your data or how to parallelize these loops, let the compiler decide.
Building with OpenACC

Enable OpenACC with the `–ta` (target accelerator) flag.

Target Accelerator:

- **tesla** - NVIDIA Tesla GPU
- **managed** - Use CUDA Managed Memory (simplifies the process)
Building with OpenACC - Feedback

```bash
$ pgc++ -fast -Minfo=accel -ta=tesla:managed main.cpp -o challenge
```

```c
matvec(const matrix &, const vector &, const vector &):
  7, include "matrix_functions.h"
  15, Generating copyout(ycoefs[:num_rows])
       Generating
copyin(xcoefs[:],Acoefs[:],cols[:],row_offsets[:num_rows+1])
  16, Complex loop carried dependence of row_offsets-> prevents parallelization
       Loop carried dependence of ycoefs-> prevents parallelization
       Loop carried backward dependence of ycoefs-> prevents
vectorization
       Complex loop carried dependence of cols->,Acoefs->,xcoefs-> prevents parallelization
Accelerator kernel generated
Generating Tesla code
  20, #pragma acc loop vector(128) /* threadIdx.x */
  24, Sum reduction generated for sum
  20, Loop is parallelizable
```
False Loop Dependencies
Aliasing prevents parallelization.

The compiler thinks there’s a \textit{carried dependency} in our loop iterations, but we thought they were parallel. Only the innermost loop was parallelized.

In C/C++, the arrays are simply pointers, so they may be aliased (two pointers accessing the same memory differently). If the compiler doesn’t know pointers aren’t aliased, it must assume they are.

This is not a problem with Fortran arrays.
C99: restrict Keyword

- Promise given by the programmer to the compiler that pointer will not alias with another pointer
  - Applied to a pointer, e.g.
    ```c
    float *restrict ptr
    ```
  - Meaning: “for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”*
- Parallelizing compilers often require `restrict` to determine independence
  - Otherwise the compiler can’t parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

* Parallelizing compilers sometimes require the `restrict` keyword to determine independence of pointers. Without `restrict`, the compiler may not be able to optimize parallel execution of loops accessing the same memory location. If the programmer violates the `restrict` declaration, the behavior of the program is undefined.
Loop independent clause

Specifies that loop iterations are data independent. This overrides any compiler dependency analysis. This is implied for *parallel loop*.

```c
#pragma acc kernels
{
    #pragma acc loop independent
    for(int i=0; i<N; i++)
    {
        a[i] = 0.0;
        b[i] = 1.0;
        c[i] = 2.0;
    }
    #pragma acc loop independent
    for(int i=0; i<N; i++)
    {
        a[i] = b[i] + c[i]
    }
}
```

Informs the compiler that both loops are safe to parallelize so it will generate both kernels.
Fixing False Aliasing

By declaring our pointers with the restrict keyword, we’ve promised the compiler they will not alias.

We could also use loop independent on our loops, but restrict fixes the underlying issue.
Rebuilding with OpenACC - Feedback

$ pgc++ -fast -Minfo=accel -ta=tesla:managed main.cpp -o challenge
matvec(const matrix &, const vector &, const vector &):
  7, include "matrix_functions.h"
  15, Generating copyout(ycoefs[:num_rows])
     Generating
    copyin(xcoefs[:],Acoefs[:],cols[:],row_offsets[:num_rows+1])
  16, Loop is parallelizable
     Accelerator kernel generated
     Generating Tesla code
  16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
20, Loop is parallelizable
Performance Now

Remember, a slow-down is expected at this point due to excess data movement.
Re-profiling the code

Application Runtime

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>135 s</td>
</tr>
<tr>
<td>GPU Kernels (matvec)</td>
<td>8 s</td>
</tr>
<tr>
<td>Data Migration (Unified Memory)</td>
<td>71 s</td>
</tr>
</tbody>
</table>
Performance After Lab 2

Once you’ve moved all 3 functions to the GPU, data movement will go away.
OpenACC parallel loop Directive

**parallel** - Programmer identifies a block of code containing parallelism. Compiler generates a *kernel*.

**loop** - Programmer identifies a loop that can be parallelized within the kernel.

NOTE: parallel & loop are often placed together

```c
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
    y[i] = a*x[i]+y[i];
}
```

NOTE: The independent clause to loop is implied when used within a parallel region.
OpenACC loop directive: private & reduction

The **private** and **reduction** clauses are not optimization clauses, they may be required for correctness.

- **private** – A copy of the variable is made for each loop iteration
- **reduction** – A reduction is performed on the listed variables.
  - Supports +, *, max, min, and various logical operations

Note: The kernels directive will generally handle these for you.
Using Parallel Loop

```c
#pragma acc parallel loop
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum)
    for(int j=row_start; j<row_end; j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```

- Instead of letting the compiler analyze the loops, let’s tell the compiler they’re parallel.
- Adding a loop directive to inner loops will tell the compiler they’re also independent.
- We must specify the reduction on sum for correctness.
Rebuilding with Parallel Loop - Feedback

$ pgc++ -fast -Minfo=accel -ta=tesla:managed main.cpp -o challenge
matvec(const matrix &, const vector &, const vector &):
  8, include "matrix_functions.h"
  12, Accelerator kernel generated
    Generating Tesla code
    15, #pragma acc loop gang /* blockIdx.x */
    20, #pragma acc loop vector(128) /* threadIdx.x */
    Sum reduction generated for sum
  12, Generating copyout(ycoefs[:num_rows])
    Generating
    copyin(xcoefs[:],Acoefs[:],cols[:],row_offsets[:num_rows+1])
  20, Loop is parallelizable
OpenACC parallel loop vs. kernels

**PARALLEL LOOP**
- Programmer’s responsibility to ensure safe parallelism
- Will parallelize what a compiler may miss
- Straightforward path from OpenMP

**KERNELS**
- Compiler’s responsibility to analyze the code and parallelize what is safe.
- Can cover larger area of code with single directive
- Gives compiler additional leeway to optimize.

Both approaches are equally valid and can perform equally well.
Review

Today we discussed:

• Tools that can be used to profile a code and identify important routines and loops where there is available parallelism

• How to analyze the code for parallelism blockers

• How to use the kernels and parallel loop directives to express the available parallelism to the compiler

• How to build with PGI and OpenACC

• How to re-profile the accelerated code
Next Steps & Homework
The remaining steps will be covered on October 29.
Homework


Build and profile the example code using the PGI compiler and gprof.

Complete the acceleration of the example code by accelerating the matvec, waxpy, and dot functions using either kernels or parallel loop.

Periodically use nvprof and/or Visual Profiler to obtain accelerated profiles and observed the results of your changes.

Note: The GPUs provided via Qwiklabs will provide a much smaller speed-up than shown here (10-15%). This is expected. Lab 3 will improve upon this.