OpenACC Worked Example

Nick Johnson

EPCC
The University of Edinburgh
Scotland

Agenda

Introduction

In this worked example, I'll take you through the steps to OpenACC'ing a larger code.

As with the lectures, stop me if you want to ask questions.

The practical after lunch is essentially following what I do here for yourselves.

Himeno

The code we are using is the scalar Himeno CPU benchmark.

Credit to:

Ryutaro Himeno, Dr. of Eng.

Head of Computer Information Center,

The Institute of Pysical and Chemical Research (RIKEN)

Email : `himeno@postman.riken.go.jp`

Himeno

The code has a fairly simplistic structure as makes sense for a good benchmark.
Here's a cut down version of main:

```
initmt();
cpu0 = second();
gosa = jacobi(NN);
cpu1 = second();
cpu1 = cpu1 - cpu0;
nflop = (kmax-2)*(jmax-2)*(imax-2)*34;
if(cpu1 != 0.0)
  xmflops2 = nflop/cpu1*1.0e-6*(real)NN;
score = xmflops2/32.27;
int nn2 = 20.0/cpu1*NN;
cpu0 = second();
gosa = jacobi(nn2);
cpu1 = second();
cpu1 = cpu1 - cpu0;
```

Himeno

There's only two real functions and some maths to worry about:

```
initmt();
gosa = jacobi(NN);
nflop = (kmax-2)*(jmax-2)*(imax-2)*34;
if(cpu1 != 0.0)
  xmflops2 = nflop/cpu1*1.0e-6*(real)NN;
score = xmflops2/32.27;
int nn2 = 20.0/cpu1*NN;
gosa = jacobi(nn2);
```

The idea is to do a fixed number of iterations of jacobi (3), see how long it takes and then do 20 seconds worth of work for the benchmark.
For testing purposes, we will fix the number of iterations each time to 3.

Functions of interest

Here are the two functions, initmt & jacobi.

- `initmt` initializes the arrays.
- `jacobi` does the work and computes the residual (gosa).

Himeno

The first step is to confirm our visual inspection, that jacobi will take most of the runtime.

You can use any CPU-based profiler for this, I picked pgprof as it was to hand on my machine.

```
                      Routine              Source   Line
   Calls   Time(%)      Name                 File    No.

       2        74    jacobi  himeno_C_v00.c    207
       1        26    initmt  himeno_C_v00.c    166
       1         0      main  himeno_C_v00.c     96
       4         0    second  himeno_C_v00.c    249
```

This seems about right. Jacobi is our most expensive function call and is called twice.

Initmt is called once and second most expensive.

Second is the timing routine used in the original.

First Pass

Let's examine the jacobi function and try to accelerate it.

```
for(n=0;n<nn;++n){
  gosa = 0.0;
  for(i=1 ; i<imax-1 ; ++i)
    for(j=1 ; j<jmax-1 ; ++j)
      for(k=1 ; k<kmax-1 ; ++k){
        s0 = a[i][j][k][0] * p[i+1][j  ][k  ]
           + a[i][j][k][1] * p[i  ][j+1][k  ]
           + a[i][j][k][2] * p[i  ][j  ][k+1]
           + b[i][j][k][0] * ( p[i+1][j+1][k  ] - p[i+1][j-1][k  ]
                             - p[i-1][j+1][k  ] + p[i-1][j-1][k  ] )
           + b[i][j][k][1] * ( p[i  ][j+1][k+1] - p[i  ][j-1][k+1]
                             - p[i  ][j+1][k-1] + p[i  ][j-1][k-1] )
           + b[i][j][k][2] * ( p[i+1][j  ][k+1] - p[i-1][j  ][k+1]
                             - p[i+1][j  ][k-1] + p[i-1][j  ][k-1] )
           + c[i][j][k][0] * p[i-1][j  ][k  ]
           + c[i][j][k][1] * p[i  ][j-1][k  ]
           + c[i][j][k][2] * p[i  ][j  ][k-1]
           + wrk1[i][j][k];
        ss = ( s0 * a[i][j][k][3] - p[i][j][k] ) * bnd[i][j][k];
        gosa = gosa + ss*ss;
        wrk2[i][j][k] = p[i][j][k] + omega * ss;
      }
  for(i=1 ; i<imax-1 ; ++i)
    for(j=1 ; j<jmax-1 ; ++j)
      for(k=1 ; k<kmax-1 ; ++k)
        p[i][j][k] = wrk2[i][j][k];
} /* end n loop */
```

First Pass

Let's examine the jacobi function and try to accelerate it.

```
  for(n=0;n<nn;++n){
    gosa = 0.0;
#pragma acc parallel loop private(i,j,k,s0,ss) reduction(+:gosa)
    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j)
        for(k=1 ; k<kmax-1 ; ++k){
          s0 = a[i][j][k][0] * p[i+1][j  ][k  ]
             + a[i][j][k][1] * p[i  ][j+1][k  ]
            <snipped>
             + wrk1[i][j][k];
          ss = ( s0 * a[i][j][k][3] - p[i][j][k] ) * bnd[i][j][k];
          gosa = gosa + ss*ss;
          wrk2[i][j][k] = p[i][j][k] + omega * ss;
        }


    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j)
        for(k=1 ; k<kmax-1 ; ++k)
          p[i][j][k] = wrk2[i][j][k];
  } /* end n loop */
  return(gosa);
```

First Pass

Results...

```
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   80.28     1.36s    9189  148.08us    1.79us  207.23us  [CUDA memcpy HtoD]
   19.72  334.22ms    1533  218.02us    2.11us  220.67us  [CUDA memcpy DtoH]
    0.00       0ns       3       0ns       0ns       0ns  jacobi_215_gpu_red
    0.00       0ns       3       0ns       0ns       0ns  jacobi_215_gpu
```

The profiler couldn't capture much data about the calls to jacobi (a bug?) but we can see that the memory copies take most of the time.
Why? The data is copied between device and host every iteration of the n loop.

Second Pass

Try to reduce the data movement in the jacobi function.

```
#pragma acc data copyin(a,b,c,bnd,wrk1,p) create(wrk2)
  for(n=0;n<nn;++n){
    gosa = 0.0;
#pragma acc parallel loop private(i,j,k,s0,ss) reduction(+:gosa)
    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j)
        for(k=1 ; k<kmax-1 ; ++k){
          s0 = a[i][j][k][0] * p[i+1][j  ][k  ]
             + a[i][j][k][1] * p[i  ][j+1][k  ]
             <snipped>
             + wrk1[i][j][k];
          ss = ( s0 * a[i][j][k][3] - p[i][j][k] ) * bnd[i][j][k];
          gosa = gosa + ss*ss;
          wrk2[i][j][k] = p[i][j][k] + omega * ss;
        }
#pragma acc parallel loop
    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j)
        for(k=1 ; k<kmax-1 ; ++k)
          p[i][j][k] = wrk2[i][j][k];
  } /* end n loop */
  return(gosa);
```

Second Pass

Results...

```
======== Profiling result:
 Time(%)       Time   Calls       Avg       Min       Max  Name
   67.94   337.35ms       9   37.48ms    1.67us  103.95ms  [CUDA memcpy HtoD]
   28.71   142.58ms       3   47.53ms   47.25ms   47.92ms  jacobi_217_gpu
    3.34    16.57ms       3    5.52ms    5.51ms    5.53ms  jacobi_242_gpu
    0.00    11.75us       3    3.92us    3.89us    3.93us  jacobi_217_gpu_red
    0.00     6.85us       3    2.28us    2.08us    2.69us  [CUDA memcpy DtoH]
```

This is roughly a factor of four improvement (2s –¿ 0.5s)!

More speed

So, we have sped up our jacobi function, but what other functions could we attack?

```
initmt();
gosa = jacobi(NN);
nflop = (kmax-2)*(jmax-2)*(imax-2)*34;
if(cpu1 != 0.0)
  xmflops2 = nflop/cpu1*1.0e-6*(real)NN;
score = xmflops2/32.27;
int nn2 = 20.0/cpu1*NN;
gosa = jacobi(nn2);
```

Two approaches:

- Do something with `initmt`.
- Do more with data movement.

Third Pass

Let's look at initmt:

```
for(i=0 ; i<imax ; ++i)
  for(j=0 ; j<jmax ; ++j)
    for(k=0 ; k<kmax ; ++k){
      a[i][j][k][0]=0.0;
      <snip>
      p[i][j][k]=0.0;
      wrk1[i][j][k]=0.0;
      bnd[i][j][k]=0.0;
    }

for(i=0 ; i<imax ; ++i)
  for(j=0 ; j<jmax ; ++j)
    for(k=0 ; k<kmax ; ++k){
      a[i][j][k][0]=1.0;
      <snip>
      p[i][j][k]=(real)(k*k)/(real)((kmax-1)*(kmax-1));
      wrk1[i][j][k]=0.0;
      bnd[i][j][k]=1.0;
    }
```

Those loops look like they could be run on the device...

Third Pass

Let's look at initmt:

```
#pragma acc parallel loop
  for(i=0 ; i<imax ; ++i)
    for(j=0 ; j<jmax ; ++j)
      for(k=0 ; k<kmax ; ++k){
        a[i][j][k][0]=0.0;
        <snip>
        p[i][j][k]=0.0;
        wrk1[i][j][k]=0.0;
        bnd[i][j][k]=0.0;
      }
#pragma acc parallel loop
  for(i=0 ; i<imax ; ++i)
    for(j=0 ; j<jmax ; ++j)
      for(k=0 ; k<kmax ; ++k){
        a[i][j][k][0]=1.0;
        <snip>
        p[i][j][k]=(real)(k*k)/(real)((kmax-1)*(kmax-1));
        wrk1[i][j][k]=0.0;
        bnd[i][j][k]=1.0;
      }
```

Those loops look like they could be run on the device.
However, this would mean copying the data to the device, initialising and copying back many times.
Not the best strategy.

Third Pass

Time to combine our two approaches: get data on the device once and keep it there and use accelerated `initmt`

```
main:
#pragma acc data create(a,b,c,p,wrk1,bnd,wrk2)
  {
  initmt(); //call to initmt inside data region
  jacobi(); //call to jacobi inside same region

initmt:
#pragma acc data present(a,b,c,p,wrk1,bnd)
{
// Use data already on the device
}

jacobi:
#pragma acc data present(a,b,c,bnd,wrk1,p,wrk2)
{
// Use data already on the device
}
```

Third Pass

Time to combine our two approaches: get data on the device once and keep it there and use accelerated `initmt`

```
======== Profiling result:
 Time(%)      Time    Calls       Avg       Min       Max  Name
   47.61  143.98ms        3   47.99ms   47.70ms   48.27ms  jacobi_226_gpu
   23.43   70.85ms        1   70.85ms   70.85ms   70.85ms  initmt_195_gpu
   23.42   70.84ms        1   70.84ms   70.84ms   70.84ms  initmt_176_gpu
    5.54   16.75ms        3    5.58ms    5.54ms    5.62ms  jacobi_251_gpu
    0.00   11.69us        3    3.90us    3.88us    3.91us  jacobi_226_gpu_red
    0.00    6.11us        3    2.04us    1.76us    2.56us  [CUDA memcpy DtoH]
    0.00    4.54us        3    1.51us    1.38us    1.79us  [CUDA memcpy HtoD]
```

Even better speed up!

Now we are down to 0.16s

Summary

To summarise:

- Profiled CPU code to find most expensive function(s).
- Set a fixed number of iterations for comparison.
- Accelerated single loop nest.
- Stopped excessive data movement inside jacobi & added second loop nest.
- Moved data region as high up the call-tree as possible.
- Initialized data on device.