An OpenACC Example Code for a C-based heat conduction code

Introduction

Accelerators have been considered as the leading candidate by many scientific and technical programmers to program and accelerate huge complex scientific applications. Accelerators such as GPUs have immense potential in terms of high compute capacity but programming these devices is a challenge. CUDA, OpenCL and other vendor-specific models are definitely a way to go, but these are low-level models that demand excellent programming skills; moreover, they are time consuming to write and debug. There is a need to approach GPU computing from a higher-level. This is the motivation behind OpenACC, a portable and directive-based programming model that helps programmers identify compute-intensive areas of the code and instruct compilers to offload such computations to accelerators.

In this post, we will discuss how the programmers can use OpenACC to simplify parallel programming of heterogeneous CPU/GPU systems and yet achieve 'good' performance. We have considered a 2D heat conduction application that is a stencil-based code as the application to run on GPUs. We chose this code primarily because the computation/communication ratio is quite high. The computation portion is intensive and it can be distributed to several threads. The performance is already very good by using GPU global memory, and it can be further improved by using GPU shared memory.

2D Heat Conduction

The formula to represent 2D heat conduction is given as follows:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

where T is temperature, t is time, αis the thermal diffusivity, and x and y are points in a grid.

To solve this problem, one possible finite difference approximation is:

$$\frac{\Delta T}{\Delta t} = \alpha\left[\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}\right]$$

where ΔT is the temperature change over time Δt and i, j are indices in a grid. In the beginning, the boundary points of the grid are assigned with initial temperature and the inner points update their temperature iteratively. Each inner point updates its temperature

by using the previous temperature of its neighboring points and itself. The temperature updating operation for all inner points in a grid needs to last long enough which means many iterations is required to get the final stable temperatures. We fix the number of iterations to 20000. The grid has 1024*1024 points.

Following is the original code snippet of 2D heat where we have inserted an OpenMP pragma. *ni* and *nj* are X and Y dimensions of the grid, respectively. At the beginning, the initial temperatures of all points in the grid are stored in *temp1_h*. *temp2_h* duplicates the temperatures in *temp1_h*. After each iteration step, *temp2_h* stores the new temperatures and then we swap the pointer so that in the next iteration the input of the kernel points to the current new temperatures. Because after all the iterations are completed, the pointers still need to be swapped. The final temperatures are stored in *temp1_h*.

```
void step_kernel(int ni, int nj, float fact,
            float* temp_in, float* temp_out)
{
#pragma omp parallel for shared(tfac,temp_in,temp_out) \
                         private(i,j,i00,im10,ip10,i0m1,i0p1,d2tdx2,d2tdy2)
   // loop over all points in domain (except boundary)
   for (j=1; j < nj-1; j++) {
     for (i=1; i < ni-1; i++) {
       // find indices into linear memory
       // for central point and neighbours
       i00 = I2D(ni, i, j);
       im10 = I2D(ni, i-1, j);
       ip10 = I2D(ni, i+1, j);
       i0m1 = I2D(ni, i, j-1);
       i0p1 = I2D(ni, i, j+1);
       // evaluate derivatives
       d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
       d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];
       // update temperatures
       temp_out[i00] = temp_in[i00]+tfac*(d2tdx2 + d2tdy2);
     }
   }
}

int main(int argc, char* argv[])
{
 ……
 gettimeofday(&tim, NULL);
 start = tim.tv_sec + (tim.tv_usec/1000000.0);

 for (istep=0; istep < nstep; istep++) {
   step_kernel(ni, nj, tfac, temp1_h, temp2_h);
```

```
    // swap the temperature pointers
    temp_tmp = temp1_h;
    temp1_h = temp2_h;
    temp2 _h= temp_tmp;
  }

  gettimeofday(&tim, NULL);
  end = tim.tv_sec + (tim.tv_usec/1000000.0);
  printf("Time for computing: %.2f s\n", end-start);
  ……
}
```

Parallelizing using OpenMP and OpenACC:

We will parallelize this program using both OpenMP and OpenACC and compare their performances to the serial code.

This program is running on a system consisting of two quad-core Intel Xeon x86_64 CPU (2.27GHz) with one Nvidia Tesla K20 GPU.

STEP 1:

In the first step we try to optimize this code for a multicore system. Here we insert OpenMP `parallel for` directive before the nested loop in *step_kernel*. This directive tells the compiler to distribute the work among all threads set in OMP_NUM_THREADS environment variable. We use the commonly used host compiler GCC to compile this program.

```
$ module load gcc/4.4.7
$ gcc –fopenmp –O3 –o heat heat.c
```

The following table shows the performance and speedup of OpenMP code compared to the serial code. It does not show significant speedup because the kernel where we have inserted the OpenMP `parallel for` directive is inside the main loop, therefore the overhead incurred is quite high; creating and destroying threads when entering and exiting the parallel region respectively.

Program version	Execution time (s)	Speedup compared to serial
Serial	134.04	1
OpenMP 2 threads	112.74	1.19
OpenMP 4 threads	66.44	2.02
OpenMP 8 threads	44.33	3.02

STEP 2:

In this step we will see if we can improve the performance by using OpenACC directives. The compilers that support OpenACC include HMPP, PGI and Cray. Here we will show how to parallelize the code with PGI's compiler. First we simply insert the following one line directive:

```
#pragma acc kernels copyin(temp_in[0:ni*nj])
copy(temp_out[0:ni*nj])
```

The directive tells the compiler to convert the following loop nest into a kernel that will be executed on the accelerator (in this case GPU is the accelerator). We also specify that data *temp_in* should be copied to the device before the kernel execution but not copied back to the host after the kernel execution. The data *temp_out* needs to be both copied to the device before the kernel execution and copied back to the host after the kernel execution. The reason that *temp_out* also needs to be copied to the device is that the kernel only updates the inner points value, while *temp_out* also includes boundary points values. If we just use copyout(temp_out), then the boundary points values that transferred to the host would be garbage values. The following command is used to compile this program:

Code Compilation:

```
$module load cuda/5.0
$module load pgi/13.1
$ pgcc –acc –ta=nvidia,cuda5.0,nofma,cc35  -Minfo=accel –
Msafeptr –O0 –o heat heat.c
step_kernel:
     26, Generating present_or_copy(temp_out[0:nj*ni])
         Generating present_or_copyin(temp_in[0:nj*ni])
         Generating NVIDIA code
         Generating compute capability 3.5 binary
     29, Complex loop carried dependence of '*(temp_in)'
prevents parallelization
         Complex loop carried dependence of '*(temp_out)'
prevents parallelization
         Parallelization would require privatization of
array 'temp_out[0:i1+nj*ni]'
         Accelerator scalar kernel generated
     30, Complex loop carried dependence of '*(temp_in)'
prevents parallelization
         Complex loop carried dependence of '*(temp_out)'
prevents parallelization
         Parallelization would require privatization of
array 'temp_out[0:i1+nj*ni]'
```

Here "-acc" flag tells the compiler to enable OpenACC and "-ta=nvidia,cuda5.0,cc35" tells the compiler that the target accelerator is NVIDIA GPU. The cuda version is 5.0 and

the compiler will generate the executable for the GPU and that its computation capability is 3.5. Note that this number needs to be supported by both the PGI compiler (13.1) and the GPU hardware (K20).

The flag "-Minfo=accel" helps to view the compilation messages about the parallelization that is taking place using OpenACC directives. The flat "-Msafeptr" is used in order to remove the pointer alias issue. Because this program is very sensitive to the order of floating point operations and PGI applies some aggressive optimization in "-O3" level, the result would be slightly different from the serial version compiled by GCC. NVIDIA GPU uses Fused Multiply-Add (FMA) by default to fuse the operations of multiply and add into one operation. Using FMA for this application also delivers result different from the serial version. For easy comparison of the output produced by different versions of the program, we disable optimizations and simply use "-O0" with PGI compiler and disable FMA by passing the "nofma" option. Note that this does not mean that the GPU result is wrong. We notice that the result using FMA appeared to be more accurate than the result that does not use FMA.

For details about FMA, you can refer to the paper "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs".

After setting all of the above-mentioned flags, the compilation messages show that there are loop carried dependences in the nested loop; interestingly actually there are no dependencies in the loop. This is because we are using a computed index in a loop that the compiler is trying to automatically schedule. So we add the following directive after "acc kernels" line to give the compiler more hints:

```
#pragma acc loop collapse(2) independent
```

Now the compilation message shows the following: (we notice that the compiler no longer complains that there are loop-carried dependencies in the loop)

```
step_kernel:
     26, Generating present_or_copy(temp_out[0:nj*ni])
         Generating present_or_copyin(temp_in[0:nj*ni])
         Generating NVIDIA code
         Generating compute capability 3.5 binary
     29, Loop is parallelizable
     30, Loop is parallelizable
         Accelerator kernel generated
         29, #pragma acc loop gang /* blockIdx.y */
         30, #pragma acc loop gang, vector(128) /*
blockIdx.x threadIdx.x */
```

This means that both the loops can be parallelized by and the PGI compiler automatically uses its own scheduling policy for the number of gangs and the vector length. By executing this program, we get the following:

$ ./heat
Time for computing: 94.92 s

However this is only slightly faster than the OpenMP version with 2 threads.

Let us try to find out why this is happening!

STEP 3:

We need performance profiling tools. We can either use NVIDIA's command line profiler nvprof or by setting "export PGI_ACC_TIME=1" when running the program.

The output of using nvprof is as follows:

```
$ nvprof ./heat
======== NVPROF is profiling heat...
======== Command: heat
Time for computing: 109.14 s
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max
Name
   58.70    40.98s   40000    1.02ms    1.01ms    1.19ms
[CUDA memcpy HtoD]
   38.36    26.78s   20000    1.34ms    1.30ms    1.43ms
[CUDA memcpy DtoH]
    2.94     2.05s   20000  102.69us  101.95us  104.26us
step_kernel_30_gpu
```

Or by using

```
$ export PGI_ACC_TIME=1
```
Time for computing: 98.83 s

```
Accelerator Kernel Timing data
/home/rengan/GTC13/heat.c
  step_kernel  NVIDIA  devicenum=0
        time(us): 85,290,940
        26: data copyin reached 40000 times
             device time(us): total=51,427,629 max=9,224
min=1,246 avg=1,285
        30: kernel launched 20000 times
            grid: [8x1022]  block: [128]
             device time(us): total=2,378,772 max=1,250
min=115 avg=118
            elapsed time(us): total=2,608,648 max=1,264
min=124 avg=130
        47: data copyout reached 20000 times
```

```
            device time(us): total=31,484,539 max=2,519
min=1,528 avg=1,574
```

Both the profiling tools show that the main performance bottleneck is due to the expensive data transfer cost.

Nvprof found that the data transfer time takes up to 97.06% of the total execution time. To reduce the data transfer time, we should try to keep the data on the GPU as long as possible, and here it is possible to do that. The solution is to add a data region outside the main loop. Note that this time *temp1_h* needs to copyin and copyout and *temp2_h* only needs to copyin. This is because after all the iterations, the pointers of *temp1_h* and *temp2_h* are swapped, and the final result is stored in *temp1_h*. Because the pointer *temp_tmp* is used just for swapping other two pointers, we make it as a device pointer so that the pointer swapping also happens on the device side.

```
#pragma acc data copy(temp1_h[0:ni*nj]) \
                                 copyin(temp2_h[0:ni*nj]) \
                                 deviceptr(temp_tmp)
{
    // main iteration loop
    for (istep=0; istep < nstep; istep++) {
            // CPU kernel
            step_kernel(ni, nj, tfac, temp1_h, temp2_h);
        // swap the temp pointers
            temp_tmp = temp1_h;
            temp1_h = temp2_h;
            temp2_h = temp_tmp;

    }
}
```

In the kernel, the directive:

```
#pragma acc kernels copyin(temp_in[0:ni*nj])
copy(temp_out[0:ni*nj])
```

is changed to

```
#pragma acc kernels
present(temp_in[0:ni*nj],temp_out[0:ni*nj])
```

since the data in *temp_in* and *temp_out* are already present in GPU.

After compiling, the new compilation message shows:

```
step_kernel:
     26, Generating present(temp_out[0:nj*ni])
```

```
          Generating present(temp_in[0:nj*ni])
          Generating NVIDIA code
          Generating compute capability 3.5 binary
      29, Loop is parallelizable
      30, Loop is parallelizable
          Accelerator kernel generated
          29, #pragma acc loop gang /* blockIdx.y */
          30, #pragma acc loop gang, vector(128) /*
blockIdx.x threadIdx.x */
main:
     116, Generating copyin(temp2_h[0:nj*ni])
          Generating copy(temp1_h[0:nj*ni])
```

By profiling again, we get:

```
$ nvprof ./heat
======== NVPROF is profiling heat...
======== Command: heat
Time for computing: 4.32 s
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max
Name
   99.84      2.04s   20000  102.10us  101.54us  103.68us
step_kernel_30_gpu
    0.11    2.21ms       2    1.11ms    1.09ms    1.12ms
[CUDA memcpy HtoD]
    0.05    1.05ms       1    1.05ms    1.05ms    1.05ms
[CUDA memcpy DtoH]
```

And after setting "PGI_ACC_TIME=1", we get the following:

Time for computing: 2.59 s

```
Accelerator Kernel Timing data
/home/rengan/GTC13/heat.c
  step_kernel  NVIDIA  devicenum=0
        time(us): 2,077,249
        30: kernel launched 20000 times
            grid: [8x1022]  block: [128]
             device time(us): total=2,077,249 max=862
min=100 avg=103
            elapsed time(us): total=2,310,052 max=874
min=111 avg=115
/home/rengan/GTC13/heat.c
  main  NVIDIA  devicenum=0
        time(us): 4,305
        116: data copyin reached 2 times
```

```
            device time(us): total=2,809 max=1,449
min=1,360 avg=1,404
        131: data copyout reached 1 times
            device time(us): total=1,496 max=1,496
min=1,496 avg=1,496
```

The profiling results show that the data transfer bottleneck does not exist any more, almost all the time is spent only in the computation part. And the data copy time is reduced from 60000 times to 3 times.

To reduce the profiling overhead, we run the program without any profiling tool and get the following:

$ ./heat
Time for computing: 2.42 s

Finally we once again compare the performances obtained by OpenACC against the OpenMP and the serial version. The following table highlights the performance numbers.

Program version	**Execution time (s)**	**Speedup compared to serial**
Serial	134.04	1
OpenMP 2 threads	112.74	1.19
OpenMP 4 threads	66.44	2.02
OpenMP 8 threads	44.33	3.02
OpenACC	2.42	55.39

We see that porting a code using OpenACC is not very hard. Of course, the complexity of the code matters. With the help of profiling tools, we can check the bottleneck of the current performance, and then we could perform optimizations of the code incrementally. In this example, we get more than 55x compared to the serial version and more than 18x (55.39/3.02) compared to the OpenMP version using 8 threads. We also learnt a very important lesson that while programming GPUs, the performance bottleneck is usually due to the data transfer part. This can prove to be quite challenging when the complexity of the application grows.