An Introduction to OpenAcc

ECS 158 Final Project

Robert Gonzales

Matthew Martin

Nile Mittow

Ryan Rasmuss

Spring 2016

1 Introduction: What is OpenAcc?

OpenAcc stands for Open Accelerators. Developed by CAPS, Cray, Nvidia, and PGI. OpenAcc is used to simplify parallel programming and can translate into a wide range of accelerators such as APUs and GPUs. OpenAcc allows users to write high-level parallel programming with C++ or Fortran code.

2 Terminology

Similar to OpenMP, the OpenACC directive general syntax is as follows:

```
#pragma acc <directive> [clause [[,] clause]... ]
```

Also like OpenMP, the directive will apply to the block of code that follows the given directive. This includes loops or structured blocks of code, where applicable. Directives are case sensitive. OpenACC supports C, C++, and Fortran, however, our tutorial focuses on C.

OpenACC is designed to move code that is parallelizable from the host CPU to an accelerator device (GPU) for execution. Due to the fact that OpenACC works with many different types of accelerator devices, there are Internal Control Variables (ICVs) used to determine the type of accelerator device and which accelerator device you wish to use. The environment variables used for the device type and device number are ACC_DEVICE_TYPE and ACC_DEVICE_NUM respectively. These environment variables can be altered via a call to acc_set_device_type and acc_set_device_num from within your code and retrieved via a call to acc_get_device_type and acc_get_device_num. If you do not set the ICVs, the default values are implementation dependent. The types of supported accelerator devices and the number of devices you can use during a single compilation are also implementation dependent.

When declaring an OpenACC directive, the directive can be applied to certain accelerators or all accelerators via the device_type clause. The arguments to the device_type clause are either a comma separated list of accelerator architecture identifiers or an asterisk for all accelerators that were not previously mentioned in another device_type clause. When compiling, the compiler will use the most specific clauses that apply to the architecture you are using, with the asterisk being the least specific. The general syntax for the device_type clause is:

```
device_type(*)
device_type(<comma-separated device type list>)
```

One of the most widely used OpenACC directives is the parallel loop construct. Its syntax is:

```
#pragma acc parallel loop [clause [[,] clause]..]
    {for-loop}
```

This directive works in a similar way to OpenMPs #pragma omp parallel for construct. When using the #pragma acc parallel loop directive, the compiler will attempt to run the code in the for loop in parallel even if it is not safe to do so. For instance, in the situation when a later iterations calculation depends on a previous iterations calculation. Due to this fact, this directive should be used with caution and only when you completely understand the underlying algorithm. If you do not use the async clause, there is an implicit barrier at the end of the parallel loop region. Some of the clauses that can be used in the parallel loop construct are as follows:

```
async [( int-expr )]
reduction( operator:var-list )
copyin( var-list )
copyout( var-list )
collapse( n )
seq
```

The async (asynchronous) clause is used to remove the implicit barrier at the end of the code region. The reduction clause works in a similar way to OpenMPs reduction clause. Reduction takes an operator and a list of variables as arguments. Reduction will apply the operator provided individually to the list of variables given. At the end of the code region in which the reduction applies, it will combine the result of applying the operator and store said result in the variable provided in the variable list. The operators that can be used with reduction in C are +, $*$, max, min, &, |, %, &&, and ||. Copyin and copyout directives are used to manage device memory. Upon entering a code region with the copyin directive, if the variables listed are not already on the device, memory will be allocated and the data that pertains to the variable will be copied in. Copyout works in a similar manner, except its work is done upon exiting the code region. Upon exiting the code region in which the copyout clause was used, the data will be copied back to the local host and the memory on the device will be deallocated. The collapse clause also works like OpenMPs collapse. It basically tells the compiler how many loops should be associated with the preceding loop directive. Finally, the seq clause tells the compiler that the loops should be executed in a sequential

manner.

Another commonly used OpenACC directive used is the kernel construct. Its syntax is as follows:

```
#pragma acc kernels [clause-list]
    {structured-block}
```

The OpenACC kernel construct will attempt to make the code within the block parallel by creating accelerator kernels where possible and safe to do so. This is a safe way to make your code parallel because the compiler will only make the code that is safe to make parallel, parallel. However, the generated code may not always be, and usually isnt, the most efficient code possible. The kernels generated will run on the GPU. If you do not use the async clause, there is an implicit barrier at the end of the kernels region. Some of the clauses allowed in the kernels construct are as follows and are explained above in the parallel loop section:

```
async [( int-expr )]
copyin( var-list )
copyout( var-list )
```

A third frequently used OpenACC directive is the data construct. It has the following syntax:

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

This construct is used to give the compiler hints as to how to handle device memory. By using this directive, you tell the compiler that the device memory should remain on the device while in the region/structured-block following the directive. You can also tell the compiler if/what data should be copied into the device upon entering the region and if/what data should be copied from the device to the host upon exiting the region. Some of the clauses that can be used with this directive follow and their explanations are above in the parallel loop section:

```
copyin( var-list )
copyout( var-list )
```

Another regularly used OpenACC directive is the routine directive. It's syntax is:

```
#pragma acc routine [clause-list]
```

This directive is similar to using __device__ when declaring a device function in Cuda. It basically tells the compiler to create a version of the function for both the host and the device.

3 OpenAcc and CUDA

The first example we will introduce is a simple matrix computation. We will find the quadratic form u'Au. We have an n x n symmetric matrix, A, and a n x 1 column vector, u. We will compute u'Au, by multiplying matrices of the following sizes: (1 x n)(n x n)(n x 1) = (1 x 1). This results in a 1 x 1 matrix, which we will treat as a scalar. Thus, we can write a C function to compute u'Au. The signature of this function will be as follows: float gpuquad(float *a, int n, float *u), where 'a' and 'u' are float arrays, and the return value of the function is u'Au treated as a scalar. In this introductory example, we will take a naive approach in order to illustrate the functionality of OpenAcc as compared to CUDA. First we will multiply A and u, then multiply u' by this result in order to produce the final result.

The OpenAcc code for the function is included below:

```
#include <stdlib.h>

// Finds u'Au
float gpuquad(float *a, int n, float *u)
{
        // Finds Au
        float *au = (float*)malloc(n * sizeof(float));
        #pragma acc data copyin(a, u)
        #pragma acc kernels
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        au[i] += a[(i * n) + j] * u[j];
                }
        }

        // Finds (u')(Au) = u'Au
        float result = 0;
        #pragma acc kernels
        for (int i = 0; i < n; i++)
        {
                result += u[i] * au[i];
        }

        #pragma acc end data copyout(result)

        return result;
}
```

This simple example uses the 'data' and 'kernels' pragmas in order to interface with the accelerator. Notice that this example uses only three OpenAcc pragmas in order to parallelize the code, but would work correctly if the pragmas were omitted. The line

```
#pragma acc data copyin(a, u)
```

tells the compiler to copy 'a' and 'u' to the GPU and keep the data there until the line

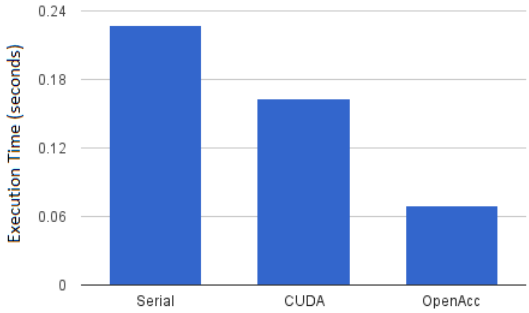
```
#pragma acc end data copyout(result)
```

where the result is copied back to the host. Inside this data section, we use the line

```
#pragma acc kernels
```

in order to ask to compiler to run kernels on the device where possible.

This is in contrast to a CUDA version of the same function, which is included in the appendix. In the CUDA version we used declarations prepended with "__global__" in order to specify kernel functions. In the OpenAcc version, we used the "kernels" pragma in order to ask the compiler to do this work for us. In the CUDA version, we use cudaMalloc and cudaMemcpy in order to copy to/from the device and host. With OpenAcc "copyin" and "copyout" are used in order to achieve similar functionality.



The graph above shows the performance of the quadratic form computation using the serial, CUDA, and OpenAcc versions of the code with n of 10,000. The OpenAcc version is about twice as fast as the CUDA version.

4 OpenAcc and OpenMP

The next example computes the number of brights spots in a matrix that represents pixel brightness in an image.

The function brights takes in an array of floats and finds the maximal number of bright spots; where a bright spot is a square subset of that array in which

all the values in that subset are greater than some threshold value.

The algorithm is simple and naive without any optimizations: it scans through the array pixel by pixel from right to left accross rows and whenever it finds a single value greater than the threshold it spawns a function which checks to see if that signle value is the upper-right corner of a bright spot.

The parallel implementation of this algorithm is quite naive in that it simply distributes the workload by rows: one thread wis responsible for some number of rows.

There are two implementations in the appendix, one written with OpenAcc and one written with OpenMP; only the interesting portions are shown here.

1) The serial implementation′s main loop:

```
74 int brights_serial(float* pix, int n, int k, float thresh) {
75 int masterCount = 0;
76
77    for(int i=1;i<=n;i++) {
78     for(int j=1;j<=n;j++) {
79      masterCount += isBrightSpot_serial(pix,n,k,thresh,i,j);
80     }
81    }
82    return masterCount;
83 }
```

Notice the function isBrightSpot() (shown in the appendix in detail). It simply spawns a second set of double for loops that checks the square of size k*k for any non-bright spots. When it finds one, it returns 0. if it does not, it returns 1. This loop will check every single pixel, multiple times (depending on how many bright pixels there are).

2) The OpenMP implementation improves on this algrotihm by allowing different CPU threads (usually 2 per core on modern intel machines that support hyperthreading) to partition the array by rows:

```
78 int brights_omp(float* pix, int n, int k, float thresh) {
79   int masterCount = 0;
80
81   /**** DIVERGE: begin omp parallel *********************/
82   #pragma omp parallel
83   {
84     #pragma omp for collapse(2) reduction(+:masterCount)
85     for(int i=1;i<=n;i++) {
86       for(int j=1;j<=n;j++) {
87         masterCount += isBrightSpot_omp(pix,n,k,thresh,i,j);
88       }
89     }
90   }
91   return masterCount;
92 }
```

Here, there are two major additions. Firstly, line 82 calls the pragma omp

parallel indicating that the following block of code is to be executed in parallel by some number of cores. Line 84 then goes on to create a for loop of depth 2, which has a reduction clause. the reduction clause creates a copy of the variables listed as it's arguments (masterCount in this case) for each partitioned instance of the for loop. Upon the completion of the loop, the individual instances are merged by the operand listed just before each variable (summation, +, in this case). By structuring the for loop in this manner we avoid having to deal with critical sections required to ensure that the different threads don't overlap when adding their respective bright spot counts back to the main total.

3) The OpenACC version of this same code provides a very OpenMP like interface for accomplishing the same goal, while allowing the flexibility to be run on a wide variety of hardware:

```
 91 int brights_acc(float* pix, int n, int k, float thresh) {
 92   int masterCount = 0;
 93
 94   /**** DIVERGE: begin parallel ********************/
 95   int sf = sizeof(float);
 96   #pragma acc parallel loop collapse(2) reduction(+:masterCount) copyin(pix[0:n*n])
 97   for(int i=1;i<=n;i++) {
 98     for(int j=1;j<=n;j++) {
 99       masterCount += isBrightSpot_acc(pix,n,k,thresh,i,j);
100     }
101   }
102   return masterCount;
103 }
```

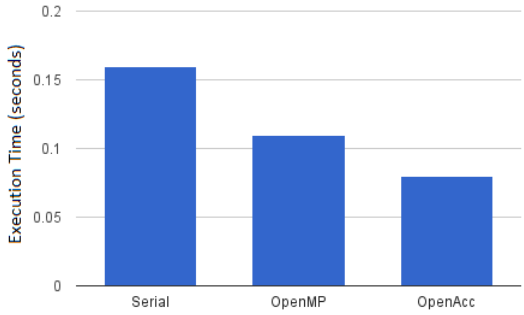
When line 84 from the brights_omp example is compared to line 91 from the brights_acc example, there are a couple differences to be noted. Firstly, instead of the for directive in OpenMP, we have the loop directive, however the two function nearly identically. Second, notice the last directive in the list, copyin(). Copyin() can be thought of in a similar manner to the CUDA subroutine cudaMemcpy(). Copyin() functionally exports the data conatined within whatever data structure is passed to it (along with the size of the structure) over to the device memory of the device that is being used for the actual parallel computation. Because OpenACC can be used for either GPUs or Multicore CPUs, the compiler of choice will ultimately make the decision as to where this memory will be allocated and how it will be addressed; as a programmer, we only need to deal with the abstractions of that memory, not the memory itself. The last two unmentioned clauses, collapse and reduction, function identically to the OpenMP counterparts.

Because OpenACC is funcdamentally an abstraction of parallel code rather than device or machine specific parallel code itself, there is one other impor-

tant directive that needs to be invoked for the above code to run: the routine directive:

```
70 #pragma acc routine seq
71 int isBrightSpot_acc(float* pix, int n, int k, float thresh,
72                       int i,      int j);
```

Routine can be thought of as the __device__ marking for functions in CUDA. Routine tells the compiler that the function immediately following it (or the function name passed to it as an argument: routine(isBrightSpot_acc) seq) needs to be available on the device as well as the host. The seq clause after the directive tells the compiler that this function must be implemented sequenctially. Other possible clauses include vector, gang, and worker which indicate the type of parallelism which the given function contains, i.e. whether parallelism can be implemented with respect to a thread's gang, a thread's worker, or a thread's vector. The routine directive can also be seen on line 78 in the brights_acc code.



The graph above shows the performance of the bright spots computation using the serial, OpenMP, and OpenAcc versions of the code with 1,000 by 1,000 matrices. The OpenAcc version is slightly faster than the OpenMP version.

5 Appendix

OpenAcc and CUDA

1.1 Quadratic Form in OpenAcc

```
#include <stdlib.h>

// Finds u'Au
float gpuquad(float *a, int n, float *u)
{
        // Finds Au
        float *au = (float*)malloc(n * sizeof(float));
        #pragma acc data copyin(a, u)
        #pragma acc kernels
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        au[i] += a[(i * n) + j] * u[j];
                }
        }

        // Finds (u')(Au) = u'Au
        float result = 0;
        #pragma acc kernels
        for (int i = 0; i < n; i++)
        {
                result += u[i] * au[i];
        }

        #pragma acc end data copyout(result)

        return result;
}
```

1.2 Quadratic Form in CUDA

```
#include <cuda.h>

// Finds au.
__global__ void findAu(float *a, int n, float *u, float *au)
{
        int me = blockIdx.x * blockDim.x + threadIdx.x;
        int numTotalThreads = blockDim.x * gridDim.x;

        float result;
        for (int rowNum = me; rowNum < n; rowNum += numTotalThreads)
        {
                result = 0;
                for (int i = 0; i < n; i++)
                {
                        result += (a[(rowNum * n) + i] * u[i]);
```

```
                }
                au[rowNum] = result;
        }
}

// Replaces au[i] with au[i] * u[i].
__global__ void findAu2(int n, float *u, float *au)
{
        int me = blockIdx.x * blockDim.x + threadIdx.x;
        int numTotalThreads = blockDim.x * gridDim.x;

        for (int rowNum = me; rowNum < n; rowNum += numTotalThreads)
        {
                au[rowNum] = u[rowNum] * au[rowNum];
        }
}

float gpuquad(float *a, int n, float *u)
{
        const int threadsPerBlock = 128;
        const int numBlocks = 192;

        // Device a.
        float *da;
        cudaMalloc((void**)&da, n * n * sizeof(float));
        cudaMemcpy(da, a, n * n * sizeof(float), cudaMemcpyHostToDevice);

        // Device u.
        float *du;
        cudaMalloc((void**)&du, n * sizeof(float));
        cudaMemcpy(du, u, n * sizeof(float), cudaMemcpyHostToDevice);

        // Device intermediate au.
        float *dau;
        cudaMalloc((void**)&dau, n * sizeof(float));

        dim3 dimGrid(threadsPerBlock, 1);
        dim3 dimBlock(numBlocks, 1, 1);

        findAu<<<dimGrid,dimBlock>>>(da, n, du, dau);
        findAu2<<<dimGrid,dimBlock>>>(n, du, dau);

        // Copy the device au back to the host.
        float* au = (float*)malloc(n * sizeof(float));
        cudaMemcpy(au, dau, n * sizeof(float), cudaMemcpyDeviceToHost);

        // Calcualte the final result on the host.
        float result = 0;
        for (int i = 0; i < n; i++)
        {
                result = result + au[i];
        }

        cudaFree(da);
        cudaFree(du);
        cudaFree(dau);
        free(au);
```

```
        return result;
}
```

OpenAcc and OpenMP

2.1 Bright Spots in OpenAcc

```
#include <openacc.h>

/*****************************************************************************
 * Function Prototypes
 * ###########################################################################*/

/** brights ******************************************************************/
//  given some nxn pixel array, count how many 'bright' kxk submatricies there
//  are where 'bright' means every pixel within the submatrix is greater than
//  some value thresh.  return count
int brights_acc(float* pix, int n, int k, float thresh);

/** isBrightSpot *************************************************************/
//  given a single point (i,j) within the matrix pix, check to see if that point
//  is the upper-left corner of a kxk submatrix where all values are greater than
//  thresh.
//  stops when first value less than thresh is found, and saves the coordinates
//  of the offending value (r,c) as badPixelR & badPixelC (for optimization)
#pragma acc routine seq
int isBrightSpot_acc(float* pix, int n, int k, float thresh,
                     int i,      int j);

/** index ********************************************************************/
// given some matrix mat as float array and indexes I & j, index inteprets mat
// as a column-major matrix with _r rows and _c columns.  it returns a pointer
// to the value stored in the matrix position i,j
#pragma acc routine seq
float* index_acc(float* mat, unsigned _r, unsigned _c, unsigned i, unsigned j);


/*****************************************************************************
 * Function Definitions
 * ###########################################################################*/

/*************************************************/
int brights_acc(float* pix, int n, int k, float thresh) {
  int masterCount = 0;

  /**** DIVERGE: begin parallel ********************/
  int sf = sizeof(float);
  #pragma acc parallel loop collapse(2) reduction(+:masterCount) copyin(pix[0:n*n])
  for(int i=1;i<=n;i++) {
    for(int j=1;j<=n;j++) {
      masterCount += isBrightSpot_acc(pix,n,k,thresh,i,j);
    }
  }
  return masterCount;
}
```

```
/*********************************************************/
int isBrightSpot_acc(float* pix, int n, int k, float thresh,
                 int i,      int j) {

  /* Case: bright spot would extend beyond array bounds */
  if (((i+k-1)>n)||((j+k-1)>n)) return 0;

  /* Assuming Bright spot is possible, check every point in potential bright spot
   * for values under threshold. if found, stop, save bad pixels, & return 0
*/
  for(unsigned r=i;r<(i+k);r++) {
    for(unsigned c=j;c<(j+k);c++) {
      if ((*index_acc(pix,n,n,r,c)) < thresh) {
        return 0;
      }
    }
  }
  return 1;
}

/*********************************************************/
float* index_acc(float* mat, unsigned _r, unsigned _c, unsigned i, unsigned j) {
  unsigned pos = (j-1) + ((i-1)*_c);
  if (pos >= _r*_c) return '\0';
  else return &mat[pos];
}
```

2.2 Bright Spots in OpenMP

```
#include <omp.h>

/*****************************************************************************
 * Function Prototypes
 * ###########################################################################*/

/** brights ******************************************************************/
//  given some nxn pixel array, count how many 'bright' kxk submatricies there
//  are where 'bright' means every pixel within the submatrix is greater than
//  some value thresh.  return count
int brights_omp(float* pix, int n, int k, float thresh);

/** isBrightSpot *************************************************************/
//  given a single point (i,j) within the matrix pix, check to see if that point
//  is the upper-left corner of a kxk submatrix where all values are greater than
//  thresh.
//  stops when first value less than thresh is found, and saves the coordinates
//  of the offending value (r,c) as badPixelR & badPixelC (for optimization)
int isBrightSpot_omp(float* pix, int n, int k, float thresh,
                     int i,      int j);

/** index ********************************************************************/
// given some matrix mat as float array and indexes I & j, index inteprets mat
// as a column-major matrix with _r rows and _c columns.  it returns a pointer
// to the value stored in the matrix position i,j
float* index_omp(float* mat, unsigned _r, unsigned _c, unsigned i, unsigned j);
```

```
/*****************************************************************************
 * Function Definitions
 * ###########################################################################*/

/*********************************************************/
int brights_omp(float* pix, int n, int k, float thresh) {
  int masterCount = 0;

  /**** DIVERGE: begin omp parallel ********************/
  #pragma omp parallel
  {
    #pragma omp for collapse(2) reduction(+:masterCount)
    for(int i=1;i<=n;i++) {
      for(int j=1;j<=n;j++) {
        masterCount += isBrightSpot_omp(pix,n,k,thresh,i,j);
      }
    }
  }
  return masterCount;
}

/*********************************************************/
int isBrightSpot_omp(float* pix, int n, int k, float thresh,
                     int i,      int j) {

  /* Case: bright spot would extend beyond array bounds */
  if (((i+k-1)>n)||((j+k-1)>n)) return 0;

  /* Assuming Bright spot is possible, check every point in potential bright spot
   * for values under threshold. if found, stop, save bad pixels, & return 0
*/
  for(unsigned r=i;r<(i+k);r++) {
    for(unsigned c=j;c<(j+k);c++) {
      if ((*index_omp(pix,n,n,r,c)) < thresh) {
        return 0;
      }
    }
  }
  return 1;
}

/*********************************************************/
float* index_omp(float* mat, unsigned _r, unsigned _c, unsigned i, unsigned j) {
  unsigned pos = (j-1) + ((i-1)*_c);
  if (pos >= _r*_c) return '\0';
  else return &mat[pos];
}
```

Group Member Contributions

R. Gonzales

Wrote terminology section, edited document for grammar.

M. Martin

Wrote OpenAcc and CUDA section (including code) and appendix, compiled Latex/PDF document, edited document for grammar.

N. Mittow

Wrote OpenAcc and OpenMP section (including code), edited document for grammar.

R. Rasmuss

Wrote introduction, compiled Latex/PDF document, produced images, edited document for grammar.