ECS158 Term Project

Michael Romero, Allan Valencia, Anastasia Zimina, Bradley Singer

June 9, 2016

Contents

Introduction

OpenACC (Open Accelerators) simplifies parallel programming by providing pragma directives, much in the way of OpenMP. The code can seamlessly unify both a CPU and GPU architecture, while providing portability across various architectures. The directives used by OpenACC are implemented across a range of compilers, and provide information for the compiler to efficiently build and optimize code. Accelerators excel not at maximizing the efficiency of a small area of code, unlike low-level programming used to squeeze performance out of small subsections of a program, but instead serves at a high-level to quickly accelerate sections while maintaining portability with little effort required.

This portability is achieved across several architectures by well-defined abstraction. OpenACC strives to encompass a wide variety of machines, and future machines as well. A high-level view of OpenACC optimization would have the host machine offloading data into the host memory, which in turn communicates with the device memory, which feeds into the device itself to parallel processing. It's critical to note that although this high-level abstraction differentiates between separate memories and devices, when programming with OpenACC, a variable should be considered as a single object. Host memory and device memory do not need to be accessed independently. Interoperability is still possible, for instance, using CUDA shared memory.

1 The OpenACC Porting Cycle

Accelerating a program should take place in three steps:

1. Initial Assessment of Program Performance
2. Parallelization of Loops
3. Optimization of Data Transfer

Tools may be used to identify the most time-consuming aspects of a program. These sections will typically be loops working on large amounts of data. On initial addition of OpenACC directives, a program will typically slow down. Do not be frustrated. After identifying sections to improve, data movement needs to be optimized, such that the host and accelerator are not spending more time in the transfer of information that the computation of information.

OpenACC provides two methods for highlighting areas that can be parallelized, the **parallel** and **kernels** directives.

1.1 Kernels

The compiler can use the kernels construct to automatically parse the region, and analyze potential parallelization capabilities. The kernels construct serves as an excellent starting place for beginners to start parallelizing their code. Below lies a simple example of the kernel directive.

```
#pragma acc kernels
{
  for(i = 0; i < n; i++){
    y[i] = 0;
    x[i] = i+1;
  }
  for(i = 0; i < n; i++){
    y[i] = 2 * x[i];
  }
}
```

When compiling with pgcc, the compiler can give information on the actions taken in kernel segments as seen below:

```
 9, Generating copyout(x[:],y[:])
11, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
11, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
15, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
15,#pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

Line number 9 copies the matrices X and Y out to device memory. The compiler recognizes the code past line 11 can be parallelized using CUDA, and generates a parallel loop operation. A gang serves as a threadblock, and the openACC vector represents CUDA threads. However, using the kernel construct isn't always the best method. When the compiler cannot determine data independence, it will not parallelize the loop, and it will mention the accelerator region (the region locked within the curly braces for the pragma) will be ignored.

1.2 Parallel

The parallel directive requires additional information. The parallel region, combined with the loop directive, gives the programmer a finer control, notifying the compiler the loop is in fact able to be parallelized (compared to the kernel directive, which lets the compiler determine this itself, sometimes incorrectly). If the compiler is told the loop is parallelizable, when it is in fact not, incorrect results may be produced.

```
 9 #pragma acc parallel loop
10     for(i = 0; i < n; i++){
11       y[i] = 0;
12       x[i] = i+1;
13     }
14 #pragma acc parallel loop
15     for(i = 0; i < n; i++){
16       y[i] = 2 * x[i];
17     }
```

And the compiler output becomes:

```
 9, Accelerator kernel generated
    Generating Tesla code
10, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
 9, Generating copyout(x[:],y[:])
14, Accelerator kernel generated
    Generating Tesla code
15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
14, Generating copyout(y[:])
    Generating copyin(x[:])
```

Compare the above output to the previous output. In the **kernel** directive, **copyout** is performed initially. Copyout means the data is initially copied to the device, then copied back to the host at the end. Using the parallel loop directives on each segment of the for loop generates an initial copyout, where the x and y arrays are copied into the device from host, and at the end of the directive, copied back out to host. In the second loop, only the x array is copied in: the compiler recognizes x is not modified and doesn't need to be copied back out. The y array however, is copied in and out.

It's important to note OpenACC allows the programmer to "identify parallelism without dictating to the compiler how to exploit that parallelism" (OpenACC.org). Now that the compiler is allowed to parallelize the code, it can maintain portability across devices, whereas hard-coding parallelization would restrict portability.

As previously mentioned, the kernels directive will not parallelize code it cannot predict correctly if it is safe at compile time. An example would be **pointer aliasing**, when two arrays possibly share the same memory. If it cannot determine if two pointers share the same memory, it will not parallelize the loop containing the arrays in question. The solution would be to use the **restrict** keyword.

Example of pointer aliasing:

```
 4 void setArrays(int *x, int *y, int n){
 5   int i;
 6 #pragma acc kernels
 7   {
 8     for(i = 0; i < n; i++){
 9       y[i] = 0;
10       x[i] = i+1;
11     }
12     for(i = 0; i < n; i++){
13       y[i] = 2 * x[i];
14     }
15   }
16
17 }
```

Compiler output:

```
6,  Generating copyout(x[:n],y[:n])
8,  Complex loop carried dependence of y-> prevents parallelization
    Loop carried dependence of x-> prevents parallelization
    Loop carried backward dependence of x-> prevents vectorization
    Accelerator scalar kernel generated
    Accelerator kernel generated
12, Complex loop carried dependence of x-> prevents parallelization
    Loop carried dependence of y-> prevents parallelization
    Loop carried backward dependence of y-> prevents vectorization
    Accelerator scalar kernel generated
    Accelerator kernel generated
```

With fixed function declaration "int *restrict y":

```
 6, Generating copyout(x[:n],y[:n])
 8, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
12, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
12, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

2 Other Important Directives

2.1 Private

To prevent race conditions, the private directive can be used. It's declared as **private** (variable). A race condition occurs when a differing sequence of events, leads to a different outcome, i.e. if A modifies C and then B modifies C is different than if B modifies C followed by A. If the following two operations happen in parallel, multiple outcomes can occur.

$$C = A + C \qquad C = B + C$$

For instance, A might equal to 2, and B might be equal to 3, with C equal to 5. If both grab C (5) at the same time, C will end up equalling either 8 or 7, depending on which operation finishes last. Ideally, one operation will modify C, and then next operation will grab the modified version of C.

For private, this should be kept in mind when a variable might be spread across the machine, when one thread of execution might modify a variable and affect the outcomes of other threads of execution.

2.2 Reduction

Reduction acts much like the private directive, and is declared as **reduction** (operator:variable). The variable will be privatized across the parallelized iterations, and when the parallel operation is complete, the variables are reduced via the operator. For a complete list of OpenACC reduction operators, read the specification, but +, *, min, and max are common. The reduction directive is used in the Bright Spot example to sum the count of bright spots.

The kernel directive will sometimes auto-generate a reduction if it can detect such. Including a sum under line 24 in the code above, such as my_sum += $y[i]$; will have the kernels directive automatically detect and implement a sum reduction:

```
25, Sum reduction generated for my_sum
```

To manually implement the reduction:

```
#pragma acc loop reduction(sum:my_sum);
```

2.3 Routine

The routine directive shares information about a function to the device, so it can be used across the parallelized regions. Because it will be declared "sequentially" across iterations, it will be defined as a sequential (seq) routine.

Example without sequence routine defined:

```
 4 void arrays(int *x){
         ...
10 }
   ...
14 #pragma acc kernels
15   {
16     for(i = 0; i < n; i++){
17       y[i] = 0;
18       x[i] = i+1;
19       arrays(x);
20     }
```

```
$ pgcc -acc -g -Minfo=all -Mprof=ccff test.c -o test

PGC-S-0155-Procedures called in a compute region must have acc routine information:
                        arrays (test.c: 19)
PGC-S-0155-Accelerator region ignored; see -Minfo messages  (test.c: 14)
setArrays:
    14, Accelerator region ignored
    19, Accelerator restriction: call to 'arrays' with no acc routine information
PGC/x86-64 Linux 16.3-0: compilation completed with severe errors
```

Simply add `#pragma acc` routine `seq` and the code will compile normally:

```
 5, Generating acc routine seq
setArrays:
15, Generating copyout(y[:n],x[:n])
17, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
17, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
22, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
22, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

2.4 Atomic Operations

To avoid race conditions, we can use the atomic operations:

```
for(int i = 0; i < n; i++){
    #pragma acc atomic update
    x[i] += 1;
}
```

2.5 Data

Most of the methods to speed up a process involve moving the most compute intensive parts to the accelerator. However, copying data from the host to the accelerator and back can be more costly than the computation itself. [2] This is because the compiler will copy data regardless of use to ensure the data is available. The `data` construct allow the programmer to specify when data is copied, allowing more control and specific optimization.

In line 8, OpenACC makes space for the `a[]` at a higher level than both `for` loops, allowing both loops to use that space but restricting it from being used in other places:

```
1  void filter(float *a,int rows, float b){
2    int flag = 0;
3    int cols = rows;
4    float cn = 0;
5    int i;
6    int j;
7    int n = cols*rows;
8    #pragma acc data create(a[0:n])
9    {
10   #pragma acc loop gang reduction(+:cn)
```

```
  for( i = 0; i < rows; i++){
    cn = 0;
    #pragma acc loop vector(n)
    for(j = 0; j < cols; j++){
      if(a[cols*i+j] > b){
        cn++;
      }
      else{
        cn = a[cols*i + j];
     }

       a[cols*i + j] = cn;
    }
  }
}
}
```

3 Data Clauses

Sometimes programmer might want additional control. For this purpose, OpenAcc provides data clauses. Data clauses can be used with kernel, parallel or data constructs.Commonly used data clauses include:

copy – creates space on the device and initializes the variables by copying, and copies back to the host and releases the space at the end

create – creates space on the device, but doesn't initialize or use the space

copyin – creates space on the device and initializes variables by copying, does not copy back to the host

copyout – creates space on the device, but does not initialize the data, eventually copies back to the host [1] An example of using *copyin* and *copyout* for matrix multiplication:

```
//result = m1*m2
void matrixMult(float* m1 , int r1, int c1, float* m2, int r2 , int c2, float* result ){
        int i,j;
        float sum;
        #pragma acc data copyin(m1[0:r1*c1]) copyin(m2[0:r2*c2]) copyout(result[0:r1*c2])
        #pragma acc parallel loop collapse(2) reduction(+:sum)
        for(i = 0; i < r1; ++i){
                for( j = 0; j < c2; ++j){
                        sum = 0;
                        for(int k = 0; k < c1; ++k){
                                sum  += m1[i*c1 + k]*m2[k*c2 + j];
                        }
                        result[i*c2 + j] =  sum;
                }
        }
}
```

Multiplicand matrix and multiplier matrix are copied to a device. Since they are not going to be updated and do not need to be copied back, copyin is used. The resulting matrix also has to be on a device, but it has to be copied back to host at the end. Thus, it is necessary to use copyout. For additional information about the code, check Appendix B.

4 Array Operations

4.1 Declaring Arrays

We can further assist the compiler by manually declaring our arrays. In the parallel loop example, it copies to and from the device twice. If instead, we declared the data to be used prior to the parallel loop directive, we can increase efficiency.

Instead of

```
#pragma acc parallel loop
    for(i = 0; i < n; i++){
      y[i] = 0;
      x[i] = i+1;
    }
#pragma acc parallel loop
    for(i = 0; i < n; i++){
      y[i] = 2 * x[i];
```

We can have

```
#pragma acc data pcreate(x[0:N]) pcopyout(y[0:N]){
#pragma acc parallel loop
    for(i = 0; i < n; i++){
      y[i] = 0;
      x[i] = i+1;
    }
#pragma acc parallel loop
    for(i = 0; i < n; i++){
      y[i] = 2 * x[i];
}
```

Where `[0:N]` describes the size of the array.

4.2 Shaping Arrays

While Fortran users do not need to worry about array borders, as compiler is pretty good in identifying them, C and C++ users might find this section useful. Since compiler requires additional information about C/C++ arrays, OpenAcc provides *array shaping.*

For C/C++ users, the syntax is s a[start:size] where a is an array, start is the first element, and size is the number of elements. If start is the zeroth element of the array, start parameter may be omitted.

For Fortran users, the syntax is slightly different, a[start, end]. Start is the first element and end is the last element to copy, a is the array. If programmer wants to work with the entire array, from beginning to end, then specifying the first and last elements is not necessary.

Common uses for array shaping include dynamic memory allocation since compiler does not know the final size at build time and data copying when programmer want to copy only part of the array. [1] The following

function is part of algorithm to find bright spots of size k by k in a square matrix a (the rest of code can be found in Appendix A). In this function, array shaping is used in data construct to allocate space for the first n elements of the array a.

```
int findBright(float *a, int rows, int k){
  int cols = rows;
  int  brightspot = 0;
  int flag;
  int coljump;
  int n = cols*rows;
  int i,j;

  #pragma acc data pcopyin(a[0:n])
  {
    #pragma acc parallel loop collapse(2) reduction(+:brightspot)
    for( i = 0; i < (rows-(k-1)); i++){
      for(j = k-1; j < cols; j++){
         if(a[cols*i+j] >= k){
           flag = 1;
           for(coljump = 0; coljump < k-1; coljump++){
             if(a[(cols*(i+(coljump+1))+j)] < k){
               flag = 0;
             }
           }
           if(flag == 1){
              brightspot++;
           }
        }
      }
    }
}
  return brightspot;
}
```

5 PGProf

The PG Profiler can be used to analyze how much time was spent in the GPU and CPU. An analysis of the serial version of brightspots relays the following:

pgprof –cpu-profiling-mode flat ./serialBright 10000 5 brightspots: 378293

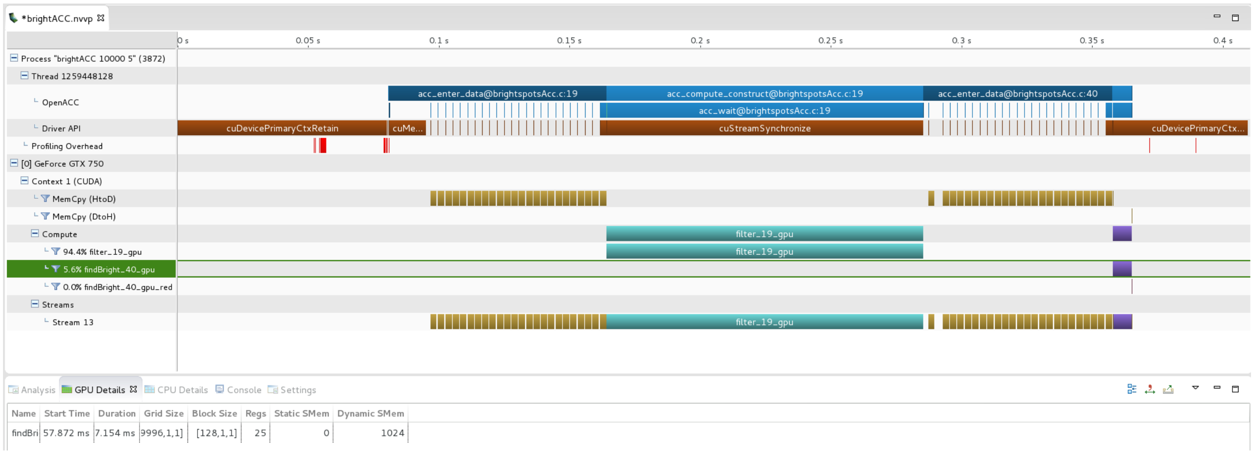
```
======== CPU profiling result (flat):
Time(%)      Time  Name
 50.00%     1.32s  random
 20.45%     540ms  findBright
 10.61%     280ms  filter
  8.33%     220ms  random_r
  6.82%     180ms  main
  2.27%      60ms  rand
  1.52%      40ms  malloc@@GLIBC_2.2.5

======== Data collected at 100Hz frequency
```

As we can see, the random and findBright function calls, along with filter, are taking significant amounts of time. While random cannot be included in the parallel region (outside function call via a library, only user functions can use the routine seq directive), findBright can be. Looking at the original serialized version under pgprof above, we noticed it takes 540ms.

With three simple additions:

```
19    #pragma acc parallel loop pcopyint(a[0:rows*cols])
40    #pragma acc parallel loop pcopyint(a[0:rows*cols]) reduction(+:brightspot)
43    #pragma acc loop
```



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem
findBri	57.872 ms	7.154 ms	9996,1,1]	[128,1,1]	25	0	1024

Figure 1: pgprof analysis

We can offset the labor to a gpu and record 7.154ms. (See figure 1 for pgprof analysis) PGProf will also display when memory is copied from the host to device and vice versa, profiler overhead, the time spent in the cpu and gpu, the grid size/blocksize, the registers used, and more.

Appendix A brightspotsACC.c

This program will find the number of "bright spots" (areas over a user given threshold of brightness) of user defined size kxk in an image represented by a matrix. This matrix maps each value to each of the image's pixels with brightness values in [0,1].

***** code highlight description ******

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// print the matrix for error checking
void print(float *a, int row, int col){
  for(int i = 0; i < row; i++){
    for(int j = 0; j < col; j++){
      printf("%4.2f ", a[col*i + j]);
    }
    printf("\n");
  }
}

// generate a random value between 0 and 1
float r2(){
  return (float)rand() / (float)RAND_MAX;
}

void filter(float *a,int rows, float b){
  int flag = 0;
  long cols = rows;
  float cn = 0;
  int i;
  int j;
 // Within the matrix at every row we record a consecutive count of successfully being within the thres
  #pragma acc data copyout(a[0:n])
  {
    #pragma acc parallel loop  reduction(+:cn)
    for( i = 0; i < rows; i++){
      cn = 0;
      #pragma acc parallel loop
      for(j = 0; j < cols; j++){
        if(a[cols*i+j] > b){
          cn++;
        }
        else{
          cn = a[cols*i + j];
       }

          a[cols*i + j] = cn;
      }
    }
  }

} // end filter()

```

```
// checks areas of size k for a minimum brightness and returns the number
// of spots that fit the criteria
int findBright(const float *a,const int rows,const int k){
  int cols = rows;
  int  brightspot = 0;
  int i,j;
  int flag;
  int coljump;
  int n = cols*rows;
  #pragma acc data copyin(a[0:n])
  {
    #pragma acc parallel loop collapse(2) reduction(+:brightspot)
    for( i = 0; i < (rows-(k-1)); i++){
      for(j = k-1; j < cols; j++){
         if(a[cols*i+j] >= k){
           flag = 1;
           for(coljump = 0; coljump < k-1; coljump++){
             if(a[(cols*(i+(coljump+1))+j)] < k){
               flag = 0;
             }
           }
           if(flag == 1){
              brightspot++;
           }
        }
      }
    }
  }

  return brightspot;
} // end findBright()

int brights(float *pix,int n, int k, float thresh){
  filter(pix,n,thresh);
  return findBright(pix,n,k);
}

int main(int argc, char* argv[]){
  int rows = atoi(argv[1]);
  int k = atoi(argv[2]);
  float t = atof(argv[3]);
  int cols = rows;

  float* a = (float*)malloc(cols*rows*sizeof(float));

  // create a matrix of size argv[1] by argv[1] with random values between 0 and 1
  for(int i = 0; i < rows*cols; i++){
    a[i] = (float)r2();
  }

  // start timer (tracks CPU time, NOT elapsed time)
  clock_t start = clock(), diff;

  printf("brightspots: %d\n",brights(a,rows,k,t));

  // end timer
  diff = clock() - start;
  int msec = diff * 1000 / CLOCKS_PER_SEC;
  printf("\nTime taken %d seconds %d milliseconds\n", msec/1000, msec%1000);
} // end main()
```

Appendix B nmfACC.c

This program finds a non-negative matrix factorization: matrix A ($r \times c$) is approximated using the product of two smaller matrices W ($r \times k$) and H($k \times c$) with rank k. It takes in the number of rows, number of columns, approximation rank, and number of iterations. First, it prints the original matrix, then it prints the matrix approximated using given parameters. Higher rank and more iterations will give more accurate result. With smaller number of iterations, the result is also affected by randomization. Since W and H are

initialized using rand() function, it might take longer for certain matrices to achieve accurate approximation. The approximation is calculated using the formulas:

$$W \leftarrow W * \frac{AH'}{WHH'}$$

$$H \leftarrow H * \frac{W'A}{W'WH}$$

Those formulas are computed in a loop given number of iterations. Final versions of W and H are multiplied together to get the approximation of the original matrix.

One highlight of this program is the use of `data copy` and the parallelization of the `for` loop in the `elementMult` function. Every `copy` variation directive will allocate space for listed variable on the device. For computing element-wise matrix multiplication we need the values of both matrices `m1` and `m2`. Specifically, we want to start off with the values of the `host` variable inside the device, to accomplish this we can use the directive `copy` to begin from the block with the device data being initialized with data from the host variable and then at the end of the block copying from device to back to the host. However, since we are using the variable `m1` to stored the resulting matrix we will need to copy back to the host `m1`, so using the directive `copy` is needed. Yet, for the variable `m2` we have no use to copy `m2` back to the host, by using the directive `copyin`, we accomplished the same as the directive `copy` with the exception of not copying at the end of the block back to the host.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>

void nmf(float *a, int r, int c , int k , int niters, float *w, float *h);
void matrixMult(float* m1 , int r1, int c1, float* m2, int r2 , int c2, float* result );
void elementMult(float* m1,float* m2, int r, int c);
void elementDiv(float* m1 , float* m2 , int r, int c);
void initMatrixH(float* mat, int r, int c, int k,float* h);
void initMatrixW(float* mat, int r , int c, int k,float* w);
void transpose( float* mat,float* t, int r , int c);
float max( float* mat,int r,int c);
void init(float* A , int r , int c);
void print(float* A, int r, int c);

//takes in number of rows, number of columns, approximation rank, and number of iterations.
int main(int argc,char* argv[]){
        int r = atoi(argv[1]);
        int c = atoi(argv[2]);
        int k = atoi(argv[3]);
        int niters = atoi(argv[4]);

        float* a = (float*)malloc(r*c*sizeof(float));
        // initial matrix a
        init(a,r,c);
        float *h = NULL;
        float *w = NULL;
        printf("current A\n");
        //print the initial matrix
        print(a,r,c);

        nmf(a,r,c,k,niters,w,h);
        //print the approximation
        printf("new A\n");
        print(a,r,c);
} // end main()

void nmf(float *a , int r , int c , int k, int niters, float* w, float *h){
        size_t sizeA = r*c*sizeof(float);
        size_t sizeW = r*k*sizeof(float);
        size_t sizeH = k*c*sizeof(float);
        size_t sizeRxK = r*k*sizeof(float);

        w = (float*)malloc(sizeW);// dim: rxk
        h = (float*)malloc(sizeH);// dim: kxc
        //memory allocation
        float* wT = (float*)malloc(sizeW);
        float* hT = (float*)malloc(sizeH);
        float* oldw = (float*)malloc(sizeW);

        float* updateWNum = (float*)malloc(sizeRxK);
        float* updateWDem1 = (float*)malloc(sizeA);
        float* updateWDem2 = (float*)malloc(sizeRxK);
        float* updateHNum = (float*)malloc(sizeH);
        float* updateHDem1 = (float*)malloc(k*k*sizeof(float));
        float* updateHDem2 = (float*)malloc(sizeH);
        //create matrices W and H
        initMatrixW(a,r,c,k,w);
        initMatrixH(a,r,c,k,h);

        for( int i = 0; i < niters; ++i){
                // since w will be update first
                memcpy(oldw,w,sizeW);
                // updating W
                transpose(h,hT,k,c); // h^t
                // a*h^t
                matrixMult(a,r,c,hT,c,k,updateWNum); // returns rxk
                // w*h
                matrixMult(w,r,k,h,k,c,updateWDem1); // returns rxc
                // (w*h)*h^t
                matrixMult(updateWDem1,r,c,hT,c,k,updateWDem2); // returns rxk
                // (a*h^t)/(w*h)*h^t
                elementDiv(updateWNum,updateWDem2,r,k); // return in updateWNum
                // computes new W
                elementMult(w,updateWNum,r,k);

                // updating H
                transpose(oldw,wT,r,k);  // w^t
                // w^t*a
                matrixMult(wT,k,r,a,r,c,updateHNum); // returns kxc
                // w^t*w
                matrixMult(wT,k,r,w,r,k,updateHDem1); // returns kxk
                // // (w^t*w)*h
                matrixMult(updateHDem1,k,k,h,k,c,updateHDem2); // returns kxc
                // (w^t*a)/(w^t*w*h)
                elementDiv(updateHNum,updateHDem2,k,c); // returns to updateHNum
                // computes new H
                elementMult(h,updateHNum,k,c);
        } // end for

        // computes new a
        matrixMult(w,r,k,h,k,c,a);

        free(wT);
        free(hT);
        free(oldw);

        free(updateWNum );
        free( updateWDem1 );
        free( updateWDem2 );
        free(updateHNum );
        free( updateHDem1);
        free(updateHDem2);
} // end nmf()

// print the matrix for testing
void print(float* A, int r,int c){
        for(int i = 0; i < r; i++){
                for(int j = 0; j < c; j++){
                        printf("%f ",A[c*i + j]);
                }
                printf("\n");
        }
        printf("\n");
}

// matrix multiplication
void matrixMult(float* m1 , int r1, int c1, float* m2, int r2 , int c2, float* result ){
        int i,j;
        float sum;
        // Using the directive copyout we specifically state that the device variable
        // at the start of the block will not copy the values of the host
        // variable, but at the end of the block copy the device to the host.
        #pragma acc data copyin(m1[0:r1*c1]) copyin(m2[0:r2*c2]) copyout(result[0:r1*c2])
        #pragma acc parallel loop collapse(2) reduction(+:sum)
        for(i = 0; i < r1; ++i){
                for( j = 0; j < c2; ++j){
                        sum = 0;
                        for(int k = 0; k < c1; ++k){
                                sum  += m1[i*c1 + k]*m2[k*c2 + j];
                        }
                        result[i*c2 + j] =  sum;
                } // end for inner
        } // end for outer
} // end matrixMult()

void elementMult(float* m1 , float* m2, int r , int c){
        int i;
        #pragma acc data copyin(m2[0:r*c]) copy(m1[0:r*c])
        #pragma acc parallel loop
        for(i = 0; i < r*c; ++i){
                m1[i] = m1[i]*m2[i];
        }
} // end elementMult()

// matrix element by element division. m1 is a dividend, m2 is a divisor, r
// is number of rows, c is number of columns. The result is stored in m1.
void elementDiv(float* m1 , float* m2, int r ,int c){
        int i;
        #pragma acc data copyin(m2[0:r*c]) copy(m1[0:r*c])
        #pragma acc parallel loop
        for(i = 0; i < r*c; ++i){
                m1[i] = m1[i]/m2[i];
        }
}

// matrix transpose. Takes in the matrix to transpose, mat, the matrix to
// store result, t, number of rows and number of columns, r and c
// respectively.
void transpose( float* mat , float* t, int r , int c){
        int i,j;
        #pragma acc data copyin(mat[0:r*c]) pcopyout(t[0:r*c])
        #pragma acc parallel loop collapse(2)
        for(i = 0; i < r; i++){
                for( j = 0; j < c; j++){
                        t[r*j + i] = mat[c*i + j];
                }
        }

}

// create initial matrix A
void init (float* A, int r, int c){
        for(int i = 0; i < r*c; i++){
                A[i] = i + 1;
        }
}

// Find the biggest element in a matrix
float max(float* mat,int r,int c){
    float mx = 0;
    int i;

    for(i = 0; i < r*c;i++){
        if (mx < mat[i]){
            mx = mat[i];
        }
    }
    return mx;
}

//create random matrix W.
void initMatrixW(float* mat, int r , int c, int k, float* w){
    float mx = sqrt(max(mat,r,c));
    int n = k*r;
    time_t t;
    srand((unsigned)time(&t));
    int i;

    for( i = 0; i < n; i++){
        w[i] = (rand() % (int)mx + 1);
    }
}

// create random matrix H
void initMatrixH(float* mat, int r , int c, int k, float * h){
    float mx = max(mat,r,c);
    int n = k*c;
    time_t t;
    srand((unsigned)time(&t));
    int i;

    for(i = 0; i < n; i++){
        h[i] = (float)( rand() % (int)(mx + 1) );
    }
} // end initMatrix()
```

Appendix C quadAcc.c Version 1

The following code calculates square form for a symmetric matrix and a vector, u'Mu where u' is a transpose of a vector. It take the length of the vector and outputs the result.

The code splits the vector in half: u1 being the first half and u2 - the second and splits the matrix into four smaller matrices of equal size:

m1	m2
m2'	m3

Then, it calculates the result using the formula:

$$u_1' m_1 u_1 + 2u_1' m_2 u_2 + u_2' m_3 u_2$$

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void getsubM(float* A, int n , int i, float* part);
//matrix multiplication. Also used in nmf
void matrixMult(float* m1 , float* m2, float* result, int c1 , int r1, int c2);

// print matrix for testing
void print(float *a, int row, int col){
  for(int i = 0; i < row; i++){
    for(int j = 0; j < col; j++){
      printf("%4.2f ", a[col*i + j]);
    }
    printf("\n\n");
  }
}

//splits matrix A into three smaller matrices a1, a2, a3.
void getsubMs(float* A, float* A1, float* A2, float* A3 ,   int n ){
    int m = n/2;
    #pragma acc data copyin(A[0:n*n])
    {
        int i;
        int j;
        #pragma acc data copyout(A1[0:m*m])
        #pragma acc parallel loop collapse(2)
        for( i = 0; i < m;i++ ){
            for(j = 0; j < m; j++){
                A1[m*i + j] = A[n*i + j];
            }
        }

        //Initializing A2(uv)
        int u = 0;
        int v;
        // if we were to use the directive parallel  on our own we would need to have
        // a critical section of atomic update
        // with the following parallel directives
        #pragma acc data copyout(A2[0:m*m])
        #pragma acc kernels
        {
            // #pragma acc parallel loop gang vector_length(m)
            for(i = 0; i < m; i++,u++ ){
                v = 0;
                // #pragma acc parallel loop vector
                for(  j = m; j < n; j++,v++){
                    A2[m*u + v] = A[n*i + j];
                    // #pragma acc atomic update
                    v++;
                }
                // #pragma acc atomic update
                u++;
            }
        }

        int x = 0;
        #pragma acc data copyout(A3[0:m*m])
        #pragma acc kernels
        {   // #pragma acc parallel loop gang vector_length(m)
            for(i = m; i <n; i++ ){
                int y = 0;
                 // #pragma acc parallel loop vector
                for( j = m; j < n; j++){
                    A3[m*x + y] = A[n*i + j];
                    // #pragma acc atomic update
                    y++;
                }
                // #pragma acc atomic update
                x++;
            } // end for outer
        } // end pragma acc data/kernel
    }// end pragma data copyin
}// end getsubMs()

void matrixMult(float* m1 , float* m2, float* result, int c1 , int r1, int c2){
    int i,j;
    float sum;
    // Using the directive copyout we specifically state that the device variable at the start of the bl
    // at the end of the block copy to the device to the host.
    #pragma acc data copyin(m1[0:r1*c1]) copyin(m2[0:c1*c2])  copyout(result[0:r1*c2])
    #pragma acc parallel loop collapse(2) reduction(+:sum)
    for(i = 0; i < r1; ++i){
        // #pragma acc parallel loop reduction(+:sum) vector
        for( j = 0; j < c2; ++j){
            sum = 0;
            for(int k = 0; k < c1; ++k){
                sum  += m1[i*c1 + k]*m2[k*c2 + j];
            }
            result[i*c2 + j] =  sum;
        } // end for inner
    } // end for outer
} // edn matrixMult
```

```
// vector multiplication. v1 and v2 are vectors to multiply, n is a size of
//a number, result is stored in scalar.
void vectorProduct(float* v1 , float* v2 , float* scalar , int n){
    int i;
    float sum = 0;
    #pragma acc data copyin(v1[0:n]) copyin(v2[0:n])
    #pragma acc parallel loop reduction(+:sum)
    for(i = 0; i < n; i++){
        sum +=  v1[i]*v2[i];
    }
    *scalar = sum;
} // end vectorProduct()

// finds quadratic form u'Au. a is a matrix, n is number of rows in a, u is a vector.
float quad(float* a , int n , float* u){
    size_t sizeAi = (n/2)*(n/2)*sizeof(float);
    float* a1 = (float*)malloc(sizeAi);
    float* a2 = (float*)malloc(sizeAi);
    float* a3 = (float*)malloc(sizeAi);

    float* u1 = (float*)malloc((n/2)*sizeof(float));
    float* u2 = (float*)malloc((n/2)*sizeof(float));

    float* r1 = (float*)malloc((n/2)*sizeof(float));
    float num1 = 0;
    float num2 = 0;
    float num3 = 0;

    getsubMs(a,a1,a2,a3,n);

    int i;
    for(i = 0; i < n/2; ++i){
        u1[i] = u[i];
        u2[i] = u[(n/2) + i];
    }

    float sum = 0;

    //u1^t*a1
    matrixMult(u1,a1,r1,n/2,1,n/2);
    vectorProduct(r1,u1,&num1,n/2);
    matrixMult(u1,a2,r1,n/2,1,n/2);
    vectorProduct(r1,u2,&num2,n/2);
    matrixMult(u2,a3,r1,n/2,1,n/2);
    vectorProduct(r1,u2,&num3,n/2);

    sum = num1 + 2*num2 + num3;

    free(u1);
    free(u2);
    free(a1);
    free(a2);
    free(a3);
    free(r1);
    return sum;
} // end quad()

//takes in one argument: number of rows for a symmetric matrix.
int main(int argc, char* argv[]){
    int n = atoi(argv[1]);
    float* a = (float*)malloc(n*n*sizeof(float));
    float* u = (float*)malloc(n*sizeof(float));
    for(int i = 0; i < n; i++){
        u[i] = i + 1;
    }

    for(int i = 0; i < n; i++){
        for(int j = 0; j < i; j++){
            a[i*n+j] = i;
            a[j*n+i] = i;
        }
    }

    for(int k = 0; k < n; k++){
        a[k*n+k] = k + 1;
    }

    float result = quad(a,n,u);
    printf("result:%f \n",result);
} // end main()
```

Appendix D quadAcc.c Version 2

The only difference between version 1 and version 2 is the way it creates the big matrix. Version 1 creates all three sub matrices at once; thus saving time by copying the big matrix once. Nevertheless, this is inconvenient for big matrices, since the parts and the big matrix occupy too much space. So, version 2 copies one part at a time, thus reusing space for both the smaller matrix and vector to store intermediate results. This version is, however, slower.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void getsubM(float* A, int n , int i, float* part);
void matrixMult(float* m1 , float* m2, float* result, int c1 , int r1, int c2);

// print the matrix for testing
void print(float *a, int row, int col){
  for(int i = 0; i < row; i++){
    for(int j = 0; j < col; j++){
      printf("%4.2f ", a[col*i + j]);
    }
    printf("\n\n");
  }
} // end print()

// instead of getting all sub-parts at once, this function returns only one
// subpart based on the user choice.
void getsubM(float* A, int n , int choice, float* part){
        int m = n/2;

        if( choice == 1){
                int i,j;
                #pragma acc data pcopyin(A[0:n*n])  pcopyout(part[0:m*m])
                #pragma acc parallel loop collapse(2)
                for( i = 0; i < m; i++ ){
                        for( j = 0; j < m; j++){
                                part[m*i + j] = A[n*i + j];
                        }
                }
        } else if(choice == 2) {
                int v = 0;
                int i,j;
                int u;
                #pragma acc data pcopyin(A[0:n*n])  pcopyout(part[0:m*m])
                #pragma acc kernels
                {
                        for( i = 0; i < m; i++ ){
                               u = 0;
                               for( j = m ; j < n; j++ ){
                                       part[m*v + u] = A[n*i + j];
                                       u++;
                               }
                               v++;
                        }
                } // end pragma acc data/kernel
        } else if(choice == 3){
                int v = 0;
                int i,j;
                int u;
                #pragma acc data pcopyin(A[0:n*n])  pcopyout(part[0:m*m])
                #pragma acc kernels
                {
                        for(i = m; i <n; i++ ){
                                 u = 0;
                                for( j = m; j < n; j++){
                                        part[m*v + u] = A[n*i + j];
                                        u++;
                                }
                                v++;
                        }
                } // end pragma acc data/kernel
        } // end else if chain
} // end getsubM()

void matrixMult(float* m1 , float* m2, float* result, int c1 , int r1, int c2){
        int i,j;
        float sum;
        // Using the directive copyout we specifically state that the device variable at the start of t
        // at the end of the block copy to the device to the host.
        #pragma acc data copyin(m1[0:r1*c1]) copyin(m2[0:c1*c2])  copyout(result[0:r1*c2])
        #pragma acc parallel loop collapse(2) reduction(+:sum)
        for(i = 0; i < r1; ++i){
                // #pragma acc parallel loop reduction(+:sum) vector
                for( j = 0; j < c2; ++j){
                        sum = 0;
                        for(int k = 0; k < c1; ++k){
                                sum  += m1[i*c1 + k]*m2[k*c2 + j];
                        }
                        result[i*c2 + j] =  sum;
                } // end for inner
        } // end for outer
} // end matrixMult()

// sum the product of two vectors
void vectorProduct(float* v1 , float* v2 , float* scalar , int n){
        int i;
        float sum = 0;
        #pragma acc data copyin(v1[0:n]) copyin(v2[0:n])
        #pragma acc parallel loop reduction(+:sum)
        for(i = 0; i < n; i++){
                sum +=  v1[i]*v2[i];
        }
        *scalar = sum;
} // end vectorProduct()

//same parameters as version 1
float quad(float* a , int n , float* u){
        size_t sizeAi = (n/2)*(n/2)*sizeof(float);
        float* ai = (float*)malloc(sizeAi);

        float* u1 = (float*)malloc((n/2)*sizeof(float));
        float* u2 = (float*)malloc((n/2)*sizeof(float));
        float* r = (float*)malloc((n/2)*sizeof(float));

        float num1 = 0;
        float num2 = 0;
        float num3 = 0;

        int i;
        for(i = 0; i < n/2; ++i){
                u1[i] = u[i];
                u2[i] = u[(n/2) + i];
        }

        float sum = 0;
        getsubM(a,n,1,ai);
        matrixMult(u1,ai,r,n/2,1,n/2);
        vectorProduct(r,u1,&num1,n/2);

        getsubM(a,n,2,ai);
        matrixMult(u1,ai,r,n/2,1,n/2);
        vectorProduct(r,u2,&num2,n/2);

        getsubM(a,n,3,ai);
        matrixMult(u2,ai,r,n/2,1,n/2);
        vectorProduct(r,u2,&num3,n/2);

        sum = num1 + 2*num2 + num3;

        free(u1);
        free(u2);
        free(ai);
        free(r);
        return sum;
} // end quad()

int main(int argc, char* argv[]){
        int n = atoi(argv[1]);
        float* a = (float*)malloc(n*n*sizeof(float));
        float* u = (float*)malloc(n*sizeof(float));
        for(int i = 0; i < n; i++){
                u[i] = i + 1;
        }

        for(int i = 0; i < n; i++){
                for(int j = 0; j < i; j++){
                        a[i*n+j] = i;
                        a[j*n+i] = i;
                }
        }

        for(int k = 0; k < n; k++){
                a[k*n+k] = k + 1;
        }

        float result = quad(a,n,u);
        printf("result:%f \n",result);
} // end main()
```

Appendix E Serial Brightspots

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>

using namespace std;

// print matrix for testing
void print(float *a, int row, int col){
  for(int i = 0; i < row; i++){
    for(int j = 0; j < col; j++){
      printf("%4.2f ", a[col*i + j]);
    }
    cout << endl;
  }
}

// generate a random value between 0 and 1
float r2(){
  return (float)rand() / (float)RAND_MAX;
}

void filter(float *a,int rows, float b){
  int cols = rows;
  float cn = 0;
  int i;
  int j;
  for( i = 0; i < rows; i++){
    cn = 0;
    for(j = 0; j < cols; j++){
      if(a[cols*i+j] > b){
        cn++;
      } else {
        cn = a[cols*i + j];
      }
       a[cols*i + j] = cn;
    } // end for inner
  } // end for outer
} // end filter()

// checks areas of size k for a minimum brightness and returns the number
// of spots that fit the criteria
int findBright(float *a,int rows,int k){
  int cols = rows;
  int  brightspot = 0;
  for(int i = 0; i < (rows-(k-1)); i++){
    for(int j = k-1; j < cols; j++){
       if(a[rows*i+j] >= k){
         int flag = 1;
         for(int coljump = 0; coljump < k-1; coljump++){
           if(a[(rows*(i+(coljump+1))+j)] < k){
             flag = 0;
           }
         }
         if(flag == 1){
            brightspot++;
         }
      } // end if
    } // end for inner
  } // end for outer
  return brightspot;
} // end findBrights()

int brights(float *pix,long n, long k, float thresh){
  filter(pix,n,thresh);
  return findBright(pix,n,k);
}

int main(int argc, char* argv[]){
  long rows = atoi(argv[1]);
  int k = atoi(argv[2]);
  long cols = rows;
  float t = atof(argv[2]);

  float* a = (float*)malloc(cols*rows*sizeof(float));

  for(long long i = 0; i < rows*cols; i++){
    a[i] = (float)r2();
  }

  cout << "brightspots:" << brights(a,rows,k,t) << endl;
} // end main()
```

Appendix F Contribution

Michael Romero - Secured PGI Accelerator Fortran/C/C++ Workstation for Linux University Developer License for the Nvidia OpenACC Toolkit and made it work on a 64-bit Debian Jessie system. Wrote the Introduction, OpenACC Porting Cycle, Kernels, Parallel, data clauses, Private, Reduction, Routine, Atomic Operations, and PGProf sections.

Allan Soria - main programmer in designing the programs and implementation. Mainly contributed in the write up of the comments within the programs and some of the appendices and editing.

Bradley Singer - Main latex formatting (organizing table of contents, verbatim setup, Appendix formatting, bibliography setup), created the first half of latex document, wrote Data directive section (2.5), wrote parts of the general code explanations for a few Appendices, added comments to code, formatted code, final edits of first half of the written content.

Anastasia Zimina - minor code contribution, program testing, wrote comments and code descriptions, latex formatting, created second half of latex document, wrote the data clauses and second array sections.

References

[1] OpenACC Organization. OpenACC API 2.5. `http://live-openacc.pantheonsite.io/sites/default/files/OpenACC_2.5_ref_guide.pdf`, 2015. [Online; accessed 9-June-2016].

[2] OpenACC Organization. OpenACC Programming and Best Practices Guide. `http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf`, 2015. [Online; accessed 7-June-2016].