Introduction to OpenACC

Francois Demoullin,
Qiwei Li,
Erin McGinnis,
Xiaotian Zhao

June 9, 2016

1 Overview of OpenACC

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator. [1] Much like Thrust and cuBLAS, OpenACC allows programmers to write GPU code without explicitly handling low-level tasks like copying data to the device or setting up threads.

The OpenACC interface is very close to OpenMP in its use of pragmas for compiler directives, and thus code utilizing OpenACC should feel familiar to a programmer with experience in OpenMP. The following tutorial assumes the reader has knowledge of both OpenMP and CUDA.

2 Compiling OpenACC

All examples are written in C and compiled with the Omni compiler[2]. Omni is one of the few open source OpenACC compiler available. The most well-known OpenACC compilers are PGI, Cray and CAPS. However, none of them are currently open source. The Omni compiler uses a source-to-source approach to translate OpenACC directives C code into CUDA C code and then run with CUDA compiler nvcc. C++ code is yet not supported.

To compile a file hello.c with OpenACC, one would type:

```
ompcc [options] -acc hello.c
```

Without the -acc option, OpenACC directives will be ignored and a CPU-only executable will be created.

3 Motivational Example: Matrix Multiplication

We will begin with a common example: matrix multiplication with $n \times n$ matrices.

$$\begin{bmatrix} * & * & * & * \\ 0 & 1 & 2 & 3 \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \times \begin{bmatrix} * & 0 & * & * \\ * & 1 & * & * \\ * & 2 & * & * \\ * & 3 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & * \\ * & 14 & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}$$

Figure 1: 4 by 4 matrix multiplication

3.1 OpenMP Matrix Multiplication

Matrix multiplication is "embarrassingly parallel"; we can assign different threads to different rows of the product matrix with no fear of thread p interfering with thread q.

We find the element in the i^{th} row and j^{th} column of the product matrix by taking the dot product

[1] http://www.openacc.org/About_OpenACC

[2] Distribution can be found from: http://omni-compiler.org/index.html

of the i^{th} row of the first matrix and the j^{th} column of the second matrix. The following code is an OpenMP implementation of matrix multiplication:

```
1  // OpenMP Matrix Multiplication
2  int main(int argc, char** argv){
3      int n = atoi(argv[1]);
4      float *a = (float*)malloc(sizeof(float) * n * n);
5      float *b = (float*)malloc(sizeof(float) * n * n);
6      float *c = (float*)malloc(sizeof(float) * n * n);
7
8      #pragma omp parallel
9      {
10         int i,j,k;
11
12         #pragma omp for
13         for(i = 0; i < n; i++){
14             for(j = 0; j < n; j++){
15                 a[i*n+j] = (float)i+j;
16                 b[i*n+j] = (float)i-j;
17                 c[i*n+j] = 0.0;
18             }
19         }
20
21         #pragma omp for
22         for(i = 0; i < n; i++){
23             for(j = 0; j < n; j++){
24                 for(k = 0; k < n; k++){
25                     c[i*n + j] += a[i*n + k] * b[k*n + j];
26                 }
27             }
28         }
29     }
30 }
```

3.2 CUDA Matrix Multiplication

OpenMP works well for smaller matrices, but for large matrices we want more threads running in parallel. The next code section is a CUDA implementation of matrix multiplication:

```
1  __global__
2  void matrixMultKernel(int *matOne, int *matTwo, int *res, int n) {
3    int bID = blockIdx.x;
4
5    for (int i = 0; i < n; i++) {
6      int sum = 0;
7      for (int j = 0; j < n; j++)
8        sum += matOne[bID * n + j] * matTwo[j * n + i];
9      res[bID * n + i] = sum;
10   }
11 }
12
13 int main(int argc, char **argv) {
14   int n = atoi(argv[1]);
```

```
15    int* matOne = new int[n * n];
16    int* matTwo = new int[n * n];
17    int* resMat = new int[n * n];
18
19    int *d_matOne, *d_matTwo, *d_resMat;
20
21    for(int i = 0; i < n; i++) {
22      for(int j = 0; j < n; j++){
23          matOne[i*n+j] = (float)i+j;
24          matTwo[i*n+j] = (float)i-j;
25      }
26    }
27
28    // set up the dimensions
29    dim3 dimGridN(n, 1);
30    dim3 dimBlock(1, 1, 1);
31
32    // allocate on device and copy in matrices
33    cudaMalloc((void**)&d_matOne, n * n * sizeof(int));
34    cudaMemcpy(d_matOne, matOne, n * n * sizeof(int), cudaMemcpyHostToDevice);
35    cudaMalloc((void**)&d_matTwo, n * n * sizeof(int));
36    cudaMemcpy(d_matTwo, matTwo, n * n * sizeof(int), cudaMemcpyHostToDevice);
37    cudaMalloc((void**)&d_resMat, n * n * sizeof(int));
38
39    // multiply and copy result to host
40    matrixMultKernel<<<dimGridN, dimBlock>>>(d_matOne, d_matTwo, d_resMat, n);
41    cudaMemcpy(resMat, d_resMat, n * n * sizeof(int), cudaMemcpyDeviceToHost);
42
43    return 0;
44  }
```

3.3 OpenACC Matrix Multiplication

The CUDA and OpenMP implementations both have benefits: OpenMP has straight-forward pragmas that inform the compiler of what to do, while CUDA runs our code on the GPU, drastically decreasing run-time for large matrices.

However, to take advantage of the GPU, about a third of our code was dedicated to copying to and from the device. We also had to choose grid dimensions, which are not necessarily optimal.

OpenACC allows us to mix the benefits of OpenMP and CUDA while hiding the low-level work. The following is an OpenACC implementation of matrix multiplication:

```
1  int main(int argc, char** argv){
2      int n = atoi(argv[1]);
3      float *a = (float*)malloc(sizeof(float) * n * n);
4      float *b = (float*)malloc(sizeof(float) * n * n);
5      float *c = (float*)malloc(sizeof(float) * n * n);
6
7      #pragma acc data copyin(a[0:n*n], b[0:n*n]), copy(c[0:n*n])
8      {
9          int i,j,k;
```

```
10
11          #pragma acc kernels
12          for(i = 0; i < n; i++){
13              for(j = 0; j < n; j++){
14                  a[i*n+j] = (float)i+j;
15                  b[i*n+j] = (float)i-j;
16                  c[i*n+j] = 0.0;
17              }
18          }
19
20          #pragma acc kernels
21          for(i = 0; i < n; i++){
22              for(j = 0; j < n; j++){
23                  for(k = 0; k < n; k++){
24                      c[i*n + j] += a[i*n + k] * b[k*n + j];
25                  }
26              }
27          }
28      }
29  }
```

Notice that the CUDA copying is taken care of in a single line (7), and we've retained the simple directive structure of OpenMP. The compiler performs an analysis of the code to determine what the grid and block dimensions should be.

3.4 OpenACC Directives

3.4.1 The kernels Pragma

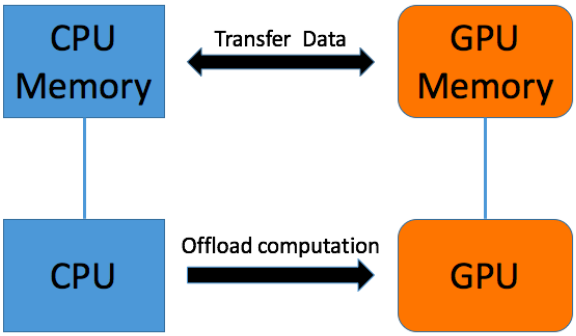
At first glance, OpenMP and OpenACC have nearly identical matrix multiply implementations. We can use higher-level pragmas to tell the compiler to identify potential parallelism for us and handle the parallelization with the **#pragma acc kernels** pragma. If the compiler decides there are dangerous dependencies within the code, it will ignore the pragma and the above will be executed on the CPU.

3.4.2 The data Pragma

The data directive tells the compiler how we want to move the data from the host to device and vice versa. We can translate line 7

```
1  #pragma acc data copyin(a[0:n*n], b[0:n*n]), copy(c[0:n*n])
```

as:
"Copy a of sizen*n starting at index 0 and b of size n*n starting at index 0 to the device. Once the created kernel has completed, copy c of size n*n starting at index 0 back out to the host device."

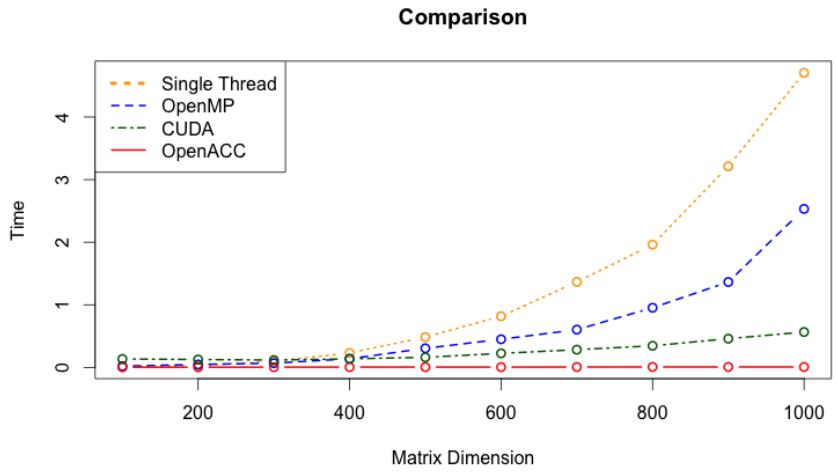


In C, arrays are pointers to the beginning of a sequence of elements. The compiler can find out the dimensions of locally declared arrays, but when arrays are dynamically allocated, the compiler only sees a pointer. We must specify the dimensions of the array to copy it properly, which is known as **array shaping**. The following is a table of other common data clauses[3]:

Clause	Definition
copy(list)	Allocates memory on GPU and copies data from host to GPU when entering region, and copies data to the host when exiting region
copyin(list)	Allocates memory on GPU and copies data from host to GPU when entering region
copyout (list)	Allocates memory on GPU and copies data to host when exiting region
create(list)	Allocates memory on GPU, but doesn't copy
present(list)	Data is already present on GPU from another containing data region
deviceptr(list)	The variable is a device pointer(e.g. CUDA) and can be used directly on the device

3.5 Speed Comparison

The following is a comparison of run-times on single-thread, OpenMP, CUDA, and OpenACC square matrix multiplication.



[3]Clauses and definitions from: https://www.youtube.com/watch?v=KgMJzmqenuc

When the computation is large enough, a parallel approach (if suitable) will always be faster than single thread because the overhead of setting up the threads will be relatively small. It's expected that the GPU out-performs CPU threads for large matrices.

However, the above graph shows that our OpenACC implementation out-performed our CUDA implementation. We used a straight-forward technique of assigning one thread per row in our CUDA implementation; the OpenACC compiler was able to optimize further by selecting dimensions according to its own analysis.

4 Example: Number of Bright Spots

Consider an $n \times n$ matrix of image pixels, with brightness values in [0,1]. Let's define a bright spot of size k and threshold b to be a $k \times k$ submatrix where each element is at least b. The total number of bright spots will include all such $k \times k$ submatrices, including overlapping matrices.

4.1 The Algorithm

The most intuitive approach is to have a nested for loop that iterates over each element A_{ij} in the $n \times n$ matrix. Within each nested for loop, there will be another nested for loop to compare each element to threshold b in the $k \times k$ matrix that begins on A_{ij}.

$$A = \begin{bmatrix} 0.0802 & 0.6990 & 0.2691 & 0.3094 \\ 0.7933 & 0.7007 & 0.1564 & 0.1516 \\ 0.3832 & 0.5982 & 0.1967 & 0.3397 \\ 0.4744 & 0.8362 & 0.9162 & 0.1480 \end{bmatrix}$$

Figure 2: n = 4, k = 2, threshold = 0.3

This approach contains a nested for loop of four levels, which is very slow for large n. Since each element belongs to k^2 sub-matrices, excluding edge elements, we're running compares on k^2 elements for each element of A! For example, we can see $A_{2,2}$ belongs to 4 sub-matrices of size 2 by 2.

$$\begin{bmatrix} * & * \\ * & 0.7007 \end{bmatrix} \begin{bmatrix} * & * \\ 0.7007 & * \end{bmatrix}$$

$$\begin{bmatrix} * & 0.7007 \\ * & * \end{bmatrix} \begin{bmatrix} 0.7007 & * \\ * & * \end{bmatrix}$$

Instead, we transform our original matrix into one that contains information about consecutive bright elements in a row. Then, for each sub-matrix, we check whether each element in the last column is a number greater than or equal to k. If so, this mean every row in the sub-matrix contains greater than or equal to k consecutive bright elements, implying that the sub-matrix is a bright spot. In the transformation below, we see that there are two bright spots.

$$\begin{bmatrix} 0.0802 & 0.6990 & 0.2691 & 0.3094 \\ 0.7933 & 0.7007 & 0.1564 & 0.1516 \\ 0.3832 & 0.5982 & 0.1967 & 0.3397 \\ 0.4744 & 0.8362 & 0.9162 & 0.1480 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 2 & 0 & 1 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

Figure 3: n = 4, k = 2, threshold = 0.3

Therefore, our program will contain two parts. First, we transform the matrix to a discrete matrix which contains information about row consecutive bright elements. Second, we iterate over each element in the matrix to count the number of bright sub-matrices.

4.2 Code

```
1  int brightSpots(float* pix, int n, int k, float thresh){
2      float* discrete = (float*)malloc(n*n*sizeof(float));
3      int nn = n-k+1;
4      int count = 0;
5
6      #pragma acc data copyin(pix[0:n*n]), copy(discrete[0:n*n], count)
7      {
8          #pragma acc parallel loop
9          for(int i = 0; i < n; i++){
10             float lastElement = 0;
11                 for(int j = 0; j < n; j++){
12                     if(pix[i*n+j] >= thresh){
13                     lastElement += 1;
14                     discrete[i*n+j] = lastElement;
15                 } else {
16                     lastElement = 0;
17                     discrete[i*n+j] = 0;
18                 }
19             }
20         }
21
22         #pragma acc parallel loop
23         for(int i = 0; i < nn; i++){
24             for(int j = k-1; j < n; j++){
25                 if((int)discrete[i*n+j] >= k){
26                     int anyDark = 0;
27                     for(int K = 1; K < k; K++){
28                         if((int)discrete[(i+K)*n+j] < k){
29                             anyDark = 1;
30                                 break;
31                         }
32                     }
33                     if(anyDark==0)
34                         count += 1;
35                 }
36             }
37         }
38     }
39     return count;
40 }
```

4.3 The parallel Pragma

A programmer could replace **#pragma acc parallel loop** with **#pragma acc kernels**, and get the same result. We can think of both **#pragma acc parallel** and **#pragma acc kernels** as an OpenMP-style shorthand for CUDA code that both creates a kernel to perform the multiplication and launches the kernel.

4.4 Differences between kernels and parallel

The main difference between the two clauses is that **#pragma acc parallel** tells the compiler that there is a parallel region, whereas **#pragma acc kernel** tells the compiler that there *might be* a parallel region.

Thus, a **parallel** region has an implicit **independent** clause attached to it. A loop equivalent **kernel** declaration would be:

```
1 #pragma acc kernel loop independent
```

which would override the compiler's data-independence analysis. [4]

Thus, it's safer for a programmer to use **kernels** when she is unsure if a region is parallelizable. It's possible that the compiler will make a mistake, which is why the programmer must fully consider his algorithm before beginning to parallelize it.

4.5 The loop Clause

The **loop** clause is near identcal to OpenMP's **for** clause, except that the loop is executed on the GPU in OpenACC.

5 Example: Non-Negative Matrix Factorization

Non-negative Matrix Factorization (NMF) is the factorization of an $r \times c$ matrix A into two smaller matrices W and H such that $WH \approx A$. Matrices W and H will have dimensions $r \times k$ and $k \times c$, respectively. Our goal is to find a good approximation WH such that $k << rank(A)$. NMF has many uses, from compression (if W and H are much smaller than A), to use in prediction models.[5]

There are many methods of computing W and H from A. The following example will use the **multiplicative update** algorithm from Lee and Seung[6]:

$$W \leftarrow W \circ \frac{AH'}{WHH'}$$
$$H \leftarrow H \circ \frac{W'A}{W'WH}$$

[4] https://www.pgroup.com/lit/articles/insider/v4n2a1.htm
[5] http://heather.cs.ucdavis.edu/NMFTutorial.pdf
[6] http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf

where $Q \circ R$ and $\frac{Q}{R}$ represent element-wise multiplication and division [7], and ' denotes transpose. We will run this algorithm for **niters**. The final W and H matrices should converge to more accurate factors of A as we increase the number of iterations.
The following code will decompose an $r \times c$ matrix into two submatrices of rank k.

5.1 The code

```
1  void nmf(float* a, int r, int c, int k, int niters, float* w, float* h) {
2    // dim of a: r x c
3    // dim of w: r x k
4    // dim of h: k x c
5
6    // initialize W and H
7    for(int i = 0; i < r * k; i++)
8      w[i] = (float)rand()/(float)RAND_MAX;
9
10   for(int i = 0; i < k * c; i++)
11     h[i] = (float)rand()/(float)RAND_MAX;
12
13   // set up tmp matrices for calculations
14   float* hT = (float*)malloc(c * k * sizeof(float));    // H' : c x k
15   float* wT = (float*)malloc(k * r * sizeof(float));    // W' : k x r
16   float* wTa = (float*)malloc(k * c * sizeof(float));   // W'A : k x c
17   float* ahT = (float*)malloc(r * k * sizeof(float));   // AH' : r x k
18   float* whhT = (float*)malloc(r * k * sizeof(float)); // WHH' : r x k
19   float* hhT = (float*)malloc(k * k * sizeof(float));   // HH' : k x k
20   float* wTw = (float*)malloc(k * k * sizeof(float));   // W'W : k x k
21   float* wTwh = (float*)malloc(r * k * sizeof(float)); // W'WH : r x k
22   float* wTmp = (float*)malloc(r * k * sizeof(float)); // updated w to be copied in
23   float* hTmp = (float*)malloc(k * c * sizeof(float)); // updated h to be copied in
24   int i, j, l, loop;
25
26   # pragma acc data copy(h[0:k * c], w[0:r * k]) copyin(a[0:r * c]) \
27     create(hT[0:c * k], wT[0:k * r], wTa[0:k * c], ahT[0:r * k], whhT[0:r * k]) \
28     create(hhT[0:k * k], wTw[0:k * k], wTwh[0:r * k], wTmp[0:r * k], hTmp[0:k * c]) \
29     create(i, j, l, loop) copyin(r, c, k)
30   {
31     for (loop = 0; loop < niters; loop++)
32     {
33       # pragma acc parallel
34       {
35         //
36         // update w
37         //
38
39         // update h transpose, result into hT
40         // matrixTranspose(h, k, c, hT);
41         # pragma acc loop independent
42         for (i = 0; i < k; i++)
43         {
44           for (j = 0; j < c; j++)
45           {
46             hT[j * k + i] = h[i * c + j];
```

[7] http://heather.cs.ucdavis.edu/NMFTutorial.pdf

```
47            }
48          }
49
50          // calculate A * H' and store in ahT
51          // matrixMult(a, hT, r, c, k, ahT);
52          #pragma acc loop independent
53          for (i = 0; i < r; i++)
54          {
55            for (j = 0; j < k; j++)
56            {
57              float dotProd = 0;
58              for (l = 0; l < c; l++)
59              {
60                dotProd += a[i * c + l] * hT[l * k + j];
61              }
62              ahT[i * k + j] = dotProd;
63            }
64          }
65          // H * H' , store result in temp
66          // matrixMult(h, hT, k, c, k, hhT);
67          #pragma acc loop independent
68          for (i = 0; i < k; i++)
69          {
70            for (j = 0; j < k; j++)
71            {
72              float dotProd = 0;
73              for (l = 0; l < c; l++)
74              {
75                dotProd += h[i * c + l] * hT[l * k + j];
76              }
77              hhT[i * k + j] = dotProd;
78            }
79          }
80
81          // calculate W * H * H' = W * temp, result into whhT
82          // matrixMult(w, hhT, r, k, k, whhT);
83          #pragma acc loop independent
84          for (i = 0; i < r; i++)
85          {
86            for (j = 0; j < k; j++)
87            {
88              float dotProd = 0;
89              for (l = 0; l < k; l++)
90              {
91                dotProd += w[i * k + l] * hhT[l * k + j];
92              }
93              whhT[i * k + j] = dotProd;
94            }
95          }
96
97          // calculate W * (AH' / WHH'), result into wTmp
98          #pragma acc loop independent
99          for (i = 0; i < r; i++)
100         {
101           for (j = 0; j < k; j++)
102           {
```

```
103             wTmp[i * k + j] = (w[i * k + j] * ahT[i * k + j]) / whhT[i * k + j];
104           }
105         }
106
107         // copy updated W into W
108         memcpy(w, wTmp, r * c * sizeof(float));
109
110
111         //
112         // update h
113         //
114
115         // update W transpose, result into wT
116         // matrixTranspose(w, r, k, wT);
117         # pragma acc loop independent
118         for (i = 0; i < r; i++)
119         {
120           for (j = 0; j < k; j++)
121           {
122             wT[j * r + i] = w[i * k + j];
123           }
124         }
125
126         // calculate W' * A, result into wTa
127         // matrixMult(wT, a, k, r, c, wTa);
128         #pragma acc loop independent
129         for (i = 0; i < k; i++)
130         {
131           for (j = 0; j < c; j++)
132           {
133             float dotProd = 0;
134             for (l = 0; l < r; l++)
135             {
136               dotProd += wT[i * r + l] * a[l * c + j];
137             }
138             wTa[i * c + j] = dotProd;
139           }
140         }
141
142         // calculate W' * W, result into wTw
143         // matrixMult(wT, w, k, r, k ,wTw);
144         #pragma acc loop independent
145         for (i = 0; i < k; i++)
146         {
147           for (j = 0; j < k; j++)
148           {
149             float dotProd = 0;
150             for (l = 0; l < r; l++)
151             {
152               dotProd += wT[i * r + l] * w[l * k + j];
153             }
154             wTw[i * k + j] = dotProd;
155           }
156         }
157
158         // calculate W' * W * H, result into wTwh
```

```
159           // matrixMult(wTw, h, k, k, c, wTwh);
160           #pragma acc loop independent
161           for (i = 0; i < k; i++)
162           {
163             for (j = 0; j < c; j++)
164             {
165               float dotProd = 0;
166               for (l = 0; l < k; l++)
167               {
168                 dotProd += wTw[i * k + l] * h[l * c + j];
169               }
170               wTwh[i * c + j] = dotProd;
171             }
172           }
173
174           // calculate H * (W'A / W'WH)
175           #pragma acc loop independent
176           for (i = 0; i < k; i++)
177           {
178             for (j = 0; j < c; j++)
179             {
180               hTmp[i * c + j] = (h[i * c + j] * wTa[i * c + j]) / wTwh[i * c + j];
181             }
182           }
183
184           // copy updated H into H
185           memcpy(h, hTmp, c * k * sizeof(float));
186
187         }
188       } // parallel loop
189     } // data pragma
190   } // NMF
```

5.1.1 How to parallelize NMF

While the NMF factorization algorithm is not trivially parallelizable it can be parallelized both on the CPU and the GPU with considerable performance improvements.

The reason parallelization of the NMF is not trivial is that NMF is an iterative approach. Both matrices W and H are updated little by little based on the W and H matrices from the previous iterations, which is why all iterations cannot be done in parallel. At the end of each iteration the values for W and H need to be updated, and thus, all threads must be synced before starting a new iteration.

However, parallelization within each interaction can be exploited. Each individual update consists of a matrix transpose computation, three matrix multiplication computations, and one combined element-wise operation.

All of the above components can be parallelized, our approach was to extract the above operations into separate functions which then have the parallelization logic implemented within them.

Matrix Transpose

Matrix transpose can be done element-wise. Each element can be independently be transposed from all other elements in the matrix. Ideally for an $n \times m$ matrix, $n * m$ threads would each transpose one element. However, the overhead of creating this many threads might be too high and thus, reduce performance. This is subject to performance tweaking

Matrix Multiplication
Each iteration of updating both H and W required 6 matrix multiplications. It is thus extremely important that matrix multiplication is done fast and in parallel. For a detailed discussion of this problem see section 3 of this tutorial.

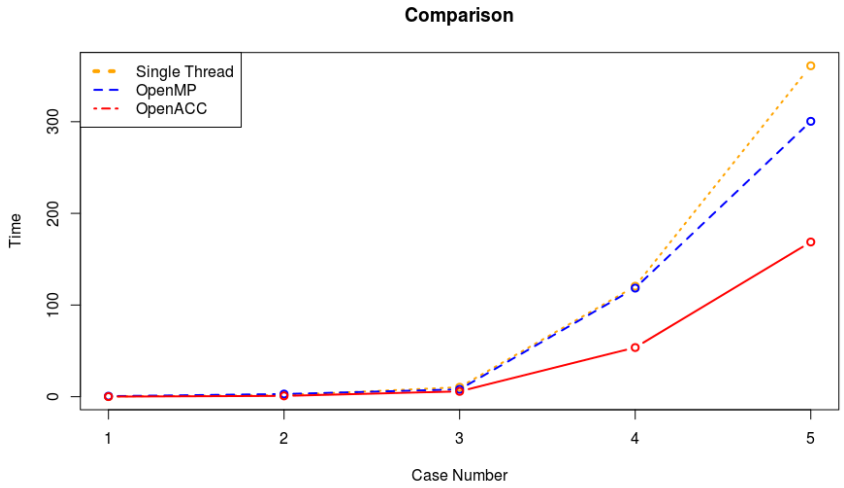
Element-wise operation
One NMF iteration requires operations that are done on each element of the matrix, in particular, element-wise multiplication and element-wise division. These two operations can be wrapped into one single kernel which is then run for each element of the matrix. Similarly to the matrix transpose approach, an $n \times m$ matrix can be operated on by $n * m$ threads which then each update one single element of the resulting matrix.

5.1.2 Performance

From the above code we can see that the NMF algorithm can be easily paralellized using OpenMP. The OpenMP code is not shown here; it is merely used as a measure of comparison between the serial version of the NMF code and the OpenACC code.
In order to evaluate how well OpenACC preforms, it was run in 5 test cases labeled 1 through 5. Each test case represents different matrix dimensions. The test cases are outlined below:

test case	Num. rows of A	Num. cols of A	Dim. k of W and H	Num. of Iterations
1	50	50	25	200
2	100	100	50	200
3	200	200	50	200
4	400	400	200	200
5	600	600	200	200

The results of the above test runs are as follows:



We can clearly see that OpenACC dominates both the serial and the OpenMP implementation. The discrepancy between OpenACC run times becomes obvious at higher matrix dimensions. At higher dimensions the granularity of each thread become bigger and the GPU can be used to it's full potential which gives a clear advantage over the CPU implementations.

5.1.3 Optional Section: Further Optimization of NMF

Each component (transpose, element-wise, multiplication) of the NMF algorithm is performed by a loop like the following:

```
1  #pragma acc loop
2  for (i = 0; i < k; i++)
3  {
4      for (j = 0; j < c; j++)
5      {
6          float dotProd = 0;
7          for (l = 0; l < k; l++)
8          {
9              dotProd += wTw[i * k + l] * h[l * c + j];
10         }
11         wTwh[i * c + j] = dotProd;
12     }
13 }
```

In the above, we've told the compiler that the loop can be parallelized, but we've left block and grid dimensions up to the compiler. We could instead specify what we want those to be using the **gang** and **vector** clauses.

```
1  #pragma acc loop gang(k * r / 256), vector(256)
2  for (i = 0; i < k; i++)
3  {
4      for (j = 0; j < c; j++)
```

```
5       {
6           float dotProd = 0;
7           for (l = 0; l < k; l++)
8           {
9               dotProd += wTw[i * k + l] * h[l * c + j];
10          }
11          wTwh[i * c + j] = dotProd;
12      }
13  }
```

In the above code, we've told the compiler to set up 256 threads per block and $k * r/256$ blocks. Trusting the compiler is generally a good option, but it's possible for the programmer to extract that final 10% of performance by performing her own analysis of the algorithm, which may surpass that of the OpenACC compiler.

6 Steps to Accelerating with OpenACC

The following is Nvidia's[8] recommended process for converting serial code to parallel code using OpenACC.

6.1 Identify Parallelism

In general, we immediately look to loops as potentially parallelizable sections. To avoid race conditions between threads, the loop must be "Data Independent" between each iteration. For example, we can parallelize individual multiplications in the NMF example, but the update iterations themselves must be sequential.

The loop must also be countable so that we can divide the work based on each thread. The OpenACC compiler may misinterpret "triangular" loops as non-independent. For example:

```
1  for i in 0:10
2    for j in i:10
3      x[i][j] = i * j
4  }
```

Though two threads will never access the same element, the OpenACC compiler may not discover that independence due to the index dependency.

6.2 Express Parallelism

In OpenMP, there are "Kernel Constructs" as in CUDA and "Parallel Constructs" as in OpenMP. By using **kernel**, the compiler creates parallelism by analysing the data dependency of loops. In the **parallel** construct, it is programmer's responsibility to make sure that it is safe to parallelize a loop.

6.3 Express Data Movement

Copying from host to device and vise-versa comes with huge cost, occasionally more than the cost of the computation itself. Thus, we need data clauses where the programmer can add information

[8] https://developer.nvidia.com/openacc

on how and when the data is created and copied. It is placed outside of **parallel** or **kernels** region.

```
1 #pragma acc data copy([data to be copied])
```

The programmer can generally expect a huge improvement in run time after this step.

6.4 Optimise Loop Performance

This step is to get the "last 10%" of performance. The programmer can optimize cache uses by arranging array accesses, and use the **gang, worker, vector** clauses to specify the number of blocks, warps, and threads respectively.[9]

7 Appendix

A Matrix Multiplication

The following code is a full test program for the Matrix Multiplication examples in Section **??**.

A.1 Normal C Matrix Multiply

```
1  int main(int argc, char** argv){
2      int n = atoi(argv[1]);
3      float *a = (float*)malloc(sizeof(float) * n * n);
4      float *b = (float*)malloc(sizeof(float) * n * n);
5      float *c = (float*)malloc(sizeof(float) * n * n);
6
7      int i,j,k;
8
9      // initialize values
10     for(i = 0; i < n; i++){
11         for(j = 0; j < n; j++){
12             a[i*n+j] = (float)i+j;
13             b[i*n+j] = (float)i-j;
14             c[i*n+j] = 0.0;
15         }
16     }
17
18     // matrix multiplication: a * b = c
19     for(i = 0; i < n; i++){
20         for(j = 0; j < n; j++){
21             for(k = 0; k < n; k++){
22                 c[i*n + j] += a[i*n + k] * b[k*n + j];
23             }
24         }
25     }
26 }
```

A.2 OpenMP Matrix Multiply

[9] http://on-demand.gputechconf.com/gtc/2012/presentations/S0517B-Monday-Programming-GPUs-OpenACC.pdf

```
int main(int argc, char** argv){
    int n = atoi(argv[1]);
    float *a = (float*)malloc(sizeof(float) * n * n);
    float *b = (float*)malloc(sizeof(float) * n * n);
    float *c = (float*)malloc(sizeof(float) * n * n);

    #pragma omp parallel
    {
        int i,j,k;

        #pragma omp for
        for(i = 0; i < n; i++){
            for(j = 0; j < n; j++){
                a[i*n+j] = (float)i+j;
                b[i*n+j] = (float)i-j;
                c[i*n+j] = 0.0;
            }
        }

        #pragma omp for
        for(i = 0; i < n; i++){
            for(j = 0; j < n; j++){
                for(k = 0; k < n; k++){
                    c[i*n + j] += a[i*n + k] * b[k*n + j];
                }
            }
        }
    }
}
```

A.3 CUDA Matrix Multiply

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

using namespace std;
// compile command: nvcc cuda.cu -o cuda.out

__global__
void matrixMultKernel(int *matOne, int *matTwo, int *res, int n) {
  int bID = blockIdx.x;

  for (int i = 0; i < n; i++) {
    int sum = 0;
    for (int j = 0; j < n; j++)
      sum += matOne[bID * n + j] * matTwo[j * n + i];
    res[bID * n + i] = sum;
  }
}

int main(int argc, char **argv) {
  int n = atoi(argv[1]);
  int* matOne = new int[n * n];
  int* matTwo = new int[n * n];
  int* resMat = new int[n * n];

```

```
25    int *d_matOne, *d_matTwo, *d_resMat;
26
27    for(int i = 0; i < n; i++) {
28      for(int j = 0; j < n; j++){
29          matOne[i*n+j] = (float)i+j;
30          matTwo[i*n+j] = (float)i-j;
31      }
32    }
33
34    // set up the dimensions
35    dim3 dimGridN(n, 1);
36    dim3 dimBlock(1, 1, 1);
37
38    // allocate on device and copy in matrices
39    cudaMalloc((void**)&d_matOne, n * n * sizeof(int));
40    cudaMemcpy(d_matOne, matOne, n * n * sizeof(int), cudaMemcpyHostToDevice);
41    cudaMalloc((void**)&d_matTwo, n * n * sizeof(int));
42    cudaMemcpy(d_matTwo, matTwo, n * n * sizeof(int), cudaMemcpyHostToDevice);
43    cudaMalloc((void**)&d_resMat, n * n * sizeof(int));
44
45    // multiply and copy result to host
46    matrixMultKernel<<<dimGridN, dimBlock>>>(d_matOne, d_matTwo, d_resMat, n);
47    cudaMemcpy(resMat, d_resMat, n * n * sizeof(int), cudaMemcpyDeviceToHost);
48
49    return 0;
50  }
```

A.4 OpenACC Matrix Multiply

```
1  int main(int argc, char** argv){
2      int n = atoi(argv[1]);
3      float *a = (float*)malloc(sizeof(float) * n * n);
4      float *b = (float*)malloc(sizeof(float) * n * n);
5      float *c = (float*)malloc(sizeof(float) * n * n);
6
7      #pragma acc data copyin(a[0:n*n], b[0:n*n]), copy(c[0:n*n])
8      {
9          int i,j,k;
10
11         #pragma acc kernels
12         for(i = 0; i < n; i++){
13             for(j = 0; j < n; j++){
14                 a[i*n+j] = (float)i+j;
15                 b[i*n+j] = (float)i-j;
16                 c[i*n+j] = 0.0;
17             }
18         }
19
20         #pragma acc kernels
21         for(i = 0; i < n; i++){
22             for(j = 0; j < n; j++){
23                 for(k = 0; k < n; k++){
24                     c[i*n + j] += a[i*n + k] * b[k*n + j];
25                 }
26             }
27         }
```

```
28      }
29  }
```

B OpenACC Finding Bright Spots

```
1   int brightSpots(float* pix, int n, int k, float thresh){
2       float* discrete = (float*)malloc(n*n*sizeof(float));
3       int nn = n-k+1;
4       int count = 0;
5
6       #pragma acc data copyin(pix[0:n*n]), copy(discrete[0:n*n], count)
7       {
8           #pragma acc parallel loop
9           for(int i = 0; i < n; i++){
10              float lastElement = 0;
11                  for(int j = 0; j < n; j++){
12                      if(pix[i*n+j] >= thresh){
13                      lastElement += 1;
14                      discrete[i*n+j] = lastElement;
15                  } else {
16                      lastElement = 0;
17                      discrete[i*n+j] = 0;
18                  }
19              }
20          }
21
22          #pragma acc parallel loop
23          for(int i = 0; i < nn; i++){
24              for(int j = k-1; j < n; j++){
25                  if((int)discrete[i*n+j] >= k){
26                      int anyDark = 0;
27                      for(int K = 1; K < k; K++){
28                          if((int)discrete[(i+K)*n+j] < k){
29                              anyDark = 1;
30                                  break;
31                          }
32                      }
33                      if(anyDark==0)
34                          count += 1;
35                  }
36              }
37          }
38      }
39      return count;
40  }
41
42  int main(int argc, char **argv){
43      int n = atoi(argv[1]);
44      int k = atoi(argv[2]);
45      float thresh;
46      sscanf(argv[3],"\%f",&thresh);
47      float* pix = (float*)malloc(n*n*sizeof(float));
48      srand(time(NULL));
49      for(int i = 0; i < n * n; i++)
50          pix[i] = (float)rand() / (float)RAND_MAX;
51      int count = brightSpots(pix, n, k, thresh);
```

```
52      printf("\%d \n", count);
53      return 0;
54  }
```

C OpenACC NMF

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4
 5  void printMatrix(float* mat, int row, int col)
 6  {
 7    for (int i = 0; i < row; i++)
 8    {
 9      for (int j = 0; j < col; j++)
10      {
11        printf("%f ", mat[i * col + j]);
12      }
13      printf("\n");
14    }
15  }
16
17
18  void matrixMult (float* a, float* b, int rowA, int colA, int colB, float* res)
19  {
20    for (int i = 0; i < rowA; i++)
21    {
22      for (int j = 0; j < colB; j++)
23      {
24        float dotProd = 0;
25        for (int l = 0; l < colA; l++)
26        {
27          dotProd += a[i * colA + l] * b[l * colB + j];
28        }
29        res[i * colB + j] = dotProd;
30      }
31    }
32  }
33
34  void matrixTranspose(float* a, int rowA, int colA, float* res)
35  {
36    for (int i = 0; i < rowA; i++)
37    {
38      for (int j = 0; j < colA; j++)
39      {
40        res[j * rowA + i] = a[i * colA + j];
41      }
42    }
43  }
44
45  void nmf(float* a, int r, int c, int k, int niters, float* w, float* h)
46  {
47    // dim of a: r x c
48    // dim of w: r x k
49    // dim of h: k x c
50
```

```
51    // initialize W and H
52    for(int i = 0; i < r * k; i++)
53    {
54      w[i] = (float)rand()/(float)RAND_MAX;
55    }
56
57    for(int i = 0; i < k * c; i++)
58    {
59      h[i] = (float)rand()/(float)RAND_MAX;
60    }
61
62    // dimensions work out!
63    float* hT = (float*)malloc(c * k * sizeof(float));    // H' : c x k
64    float* wT = (float*)malloc(k * r * sizeof(float));    // W' : k x r
65    float* wTa = (float*)malloc(k * c * sizeof(float));   // W'A : k x c
66    float* ahT = (float*)malloc(r * k * sizeof(float));   // AH' : r x k
67    float* whhT = (float*)malloc(r * k * sizeof(float)); // WHH' : r x k
68    float* hhT = (float*)malloc(k * k * sizeof(float));   // HH' : k x k
69    float* wTw = (float*)malloc(k * k * sizeof(float));   // W'W : k x k
70    float* wTwh = (float*)malloc(r * k * sizeof(float)); // W'WH : r x k
71    float* wTmp = (float*)malloc(r * k * sizeof(float)); // updated w to be copied in
72    float* hTmp = (float*)malloc(k * c * sizeof(float)); // updated h to be copied in
73    int i, j, l, loop;
74
75    # pragma acc data copy(h[0:k * c], w[0:r * k]) copyin(a[0:r * c]) \
76      create(hT[0:c * k], wT[0:k * r], wTa[0:k * c], ahT[0:r * k], whhT[0:r * k]) \
77      create(hhT[0:k * k], wTw[0:k * k], wTwh[0:r * k], wTmp[0:r * k], hTmp[0:k * c]) \
78      create(i, j, l, loop) copyin(r, c, k)
79    {
80      for (loop = 0; loop < niters; loop++)
81      {
82        # pragma acc parallel
83        {
84          //
85          // update w
86          //
87
88          // update h transpose, result into hT
89          // matrixTranspose(h, k, c, hT);
90          # pragma acc loop independent
91          for (i = 0; i < k; i++)
92          {
93            for (j = 0; j < c; j++)
94            {
95              hT[j * k + i] = h[i * c + j];
96            }
97          }
98
99          // calculate A * H' and store in ahT
100         // matrixMult(a, hT, r, c, k, ahT);
101         #pragma acc loop independent
102         for (i = 0; i < r; i++)
103         {
104           for (j = 0; j < k; j++)
105           {
106             float dotProd = 0;
```

```
107             for (l = 0; l < c; l++)
108             {
109               dotProd += a[i * c + l] * hT[l * k + j];
110             }
111             ahT[i * k + j] = dotProd;
112           }
113         }
114         // H * H' , store result in temp
115         // matrixMult(h, hT, k, c, k, hhT);
116         #pragma acc loop independent
117         for (i = 0; i < k; i++)
118         {
119           for (j = 0; j < k; j++)
120           {
121             float dotProd = 0;
122             for (l = 0; l < c; l++)
123             {
124               dotProd += h[i * c + l] * hT[l * k + j];
125             }
126             hhT[i * k + j] = dotProd;
127           }
128         }
129
130         // calculate W * H * H' = W * temp, result into whhT
131         // matrixMult(w, hhT, r, k, k, whhT);
132         #pragma acc loop independent
133         for (i = 0; i < r; i++)
134         {
135           for (j = 0; j < k; j++)
136           {
137             float dotProd = 0;
138             for (l = 0; l < k; l++)
139             {
140               dotProd += w[i * k + l] * hhT[l * k + j];
141             }
142             whhT[i * k + j] = dotProd;
143           }
144         }
145
146         // calculate W * (AH' / WHH'), result into wTmp
147         #pragma acc loop independent
148         for (i = 0; i < r; i++)
149         {
150           for (j = 0; j < k; j++)
151           {
152             wTmp[i * k + j] = (w[i * k + j] * ahT[i * k + j]) / whhT[i * k + j];
153           }
154         }
155
156         // copy updated W into W
157         memcpy(w, wTmp, r * c * sizeof(float));
158
159
160         //
161         // update h
162         //
```

```
163
164         // update W transpose, result into wT
165         // matrixTranspose(w, r, k, wT);
166         # pragma acc loop independent
167         for (i = 0; i < r; i++)
168         {
169           for (j = 0; j < k; j++)
170           {
171             wT[j * r + i] = w[i * k + j];
172           }
173         }
174
175         // calculate W' * A, result into wTa
176         // matrixMult(wT, a, k, r, c, wTa);
177         #pragma acc loop independent
178         for (i = 0; i < k; i++)
179         {
180           for (j = 0; j < c; j++)
181           {
182             float dotProd = 0;
183             for (l = 0; l < r; l++)
184             {
185               dotProd += wT[i * r + l] * a[l * c + j];
186             }
187             wTa[i * c + j] = dotProd;
188           }
189         }
190
191         // calculate W' * W, result into wTw
192         // matrixMult(wT, w, k, r, k ,wTw);
193         #pragma acc loop independent
194         for (i = 0; i < k; i++)
195         {
196           for (j = 0; j < k; j++)
197           {
198             float dotProd = 0;
199             for (l = 0; l < r; l++)
200             {
201               dotProd += wT[i * r + l] * w[l * k + j];
202             }
203             wTw[i * k + j] = dotProd;
204           }
205         }
206
207         // calculate W' * W * H, result into wTwh
208         // matrixMult(wTw, h, k, k, c, wTwh);
209         #pragma acc loop independent
210         for (i = 0; i < k; i++)
211         {
212           for (j = 0; j < c; j++)
213           {
214             float dotProd = 0;
215             for (l = 0; l < k; l++)
216             {
217               dotProd += wTw[i * k + l] * h[l * c + j];
218             }
```

```
219             wTwh[i * c + j] = dotProd;
220           }
221         }
222
223         // calculate H * (W'A / W'WH)
224         #pragma acc loop independent
225         for (i = 0; i < k; i++)
226         {
227           for (j = 0; j < c; j++)
228           {
229             hTmp[i * c + j] = (h[i * c + j] * wTa[i * c + j]) / wTwh[i * c + j];
230           }
231         }
232
233         // copy updated H into H
234         memcpy(h, hTmp, c * k * sizeof(float));
235
236       }
237     } // parallel loop
238   } // data pragma
239 }
240
241 int main()
242 {
243   // test program
244   int k = 4;
245   int r = 6;
246   int c = 5;
247   int niter = 500;
248   float* a = (float*)malloc(r * c * sizeof(float));
249   float* h = (float*)malloc(k * c * sizeof(float));
250   float* w = (float*)malloc(r * k * sizeof(float));
251
252   // initialize a with random values
253   for(int i = 0; i < r * c; i++)
254   {
255     a[i] = (float)rand()/RAND_MAX;
256   }
257   // run NMF. w and h will be returned with the approximation
258   nmf(a, r, c, k, niter, w, h);
259
260   // print to compare original to approx
261   printf("==========Original Matrix:==========\n");
262   printMatrix(a, r, c);
263
264
265   printf("==========Approximation:==========\n");
266
267   float* approx = (float*)malloc(r * c * sizeof(float));
268   matrixMult(w, h, r, k, c, approx);
269
270   printMatrix(approx, r, c);
271
272   // free variables
273   free(a);
274   free(h);
```

```
275     free(w);
276     free(approx);
277
278     return 0;
279 }
```

D Student Contribution

Francois contributed the NMF base code and write-up. Qiwei did the graphs, number of bright spots code, and write up
Erin - Introduction + Matrix Multiplication code + Write up
Xiaotian - Compiler and Summary